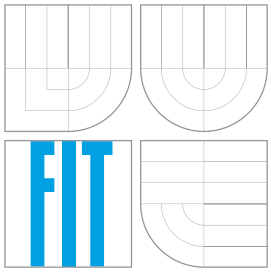
BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

# IMPROVED TOOLS FOR HANDLING DELTARPM FILES

VYLEPŠENÍ NÁSTROJŮ PRO PRÁCI SE SOUBORY DELTARPM

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE
MATĚJ CHALK

SUPERVISOR
VEDOUCÍ PRÁCE
Prof. Ing. TOMÁŠ VOJNAR, Ph.D.

BRNO 2016

*Bachelor Project Specification/18789/2015/xchalk00*

**Brno University of Technology - Faculty of Information Technology**

Department of Intelligent Systems Academic year 2015/2016

# Bachelor Project Specification

For: **Chalk Matěj**
Branch of study: Information Technology
Title: **Improved Tools for Handling deltarpm Files**
Category: Operating Systems

Instructions for project work:

1. Get acquainted with the format of deltarpm files, with methods of creating them and with their usage.
2. Analyse the currently existing reference implementation of programs for dealing with deltarpm and its weaknesses. Concentrate on impossibility of using this implementation as a library and on that it does not support some types of rpm files.
3. Propose a new implementation solving weaknesses of the current one.
4. Implement your proposed solution and test its functionality, including in particular the fact that it covers the functionality of the original implementation.
5. Discuss the obtained results and possibilities of their further development.

Basic references:


- Percival, C.: Matching with Mismatches and Assorted Applications, Ph.D. thesis, Oxford University, 2006.
- Bailey, E.C.: Maximum RPM (Taking the Red Hat Package Manager to the Limit), Red Hat Inc., 2000. Available online: http://www.rpm.org/max-rpm/
- Larssona, N.J., Sadakane, K.: Faster suffix sorting, In: Theoretical Computer Science, 387(3), Elsevier, 2003.
- Free Standards Group: Linux Standard Base Core Specification 3.1, 2005. https://refspecs.linuxbase.org/LSB_3.1.0/LSB-Core-generic/LSB-Core-generic/book1.html

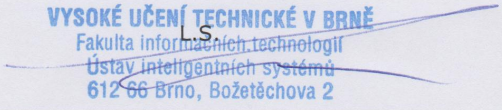

Requirements for the first semester:
The first two points and at least a beginning of works on the third one.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Bachelor Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Vojnar Tomáš, prof. Ing., Ph.D.**, DITS FIT BUT
Beginning of work: November 1, 2015
Date of delivery: May 18, 2016




Petr Hanáček
*Associate Professor and Head of Department*

# Abstract

RPM packages are used for software installation in Fedora. Every version of software packaged in this way corresponds to a separate RPM file. Updating software therefore entails downloading a large RPM file that is actually quite similar to the RPM already installed. An alternative for software updates is provided by DeltaRPM packages, which are special patch files that store the difference between two RPM files. An update then consists of downloading a much smaller file and applying this patch to the older version of the RPM. The deltarpm project defines the format of DeltaRPM files and supplies command-line tools for creating and applying them. However, this implementation is unsuitable for use as a library. The aim of this thesis is to create a new implementation of these tools, which is backwards compatible and provides a library for C developers that solves some of the weaknesses of the current implementation.



# Abstrakt

Na platformě Fedora se používají balíčky RPM pro instalaci softwaru. Každá verze takto distribuovaného softwaru odpovídá samostatnému souboru RPM. Aktualizace softwaru pak odpovídá stáhnutí velkého souboru RPM, který je ve skutečnosti velmi podobný již nainstalovnému balíčku. Balíčky DeltaRPM poskytují alternativu pro aktualizaci softwaru. Jedná se o speciální patch soubory, které uchovávají rozdíl mezi dvěma soubory RPM. Aktualizace pak spočívá ve stáhnutí daleko menšího souboru a aplikaci tohoto patche na starší verzi příslušného RPM. Projekt deltarpm definuje formát souborů DeltaRPM a nabízí nástroje pro příkazovou řádku, které realizují jejich vytváření a aplikaci. Tato implementace je však nevhodná pro použití jako knihovna. Cíl této práce je vytvořit novou implementaci nástrojů pro vytváření a aplikaci souborů DeltaRPM, která je zpětně kompatibilní a poskytuje knihovnu pro vývojáře v jazyce C, která vyřeší některé slabiny současné implementace.



# Keywords

DeltaRPM, RPM, Fedora, Red Hat, C, library, reimplementation, binary patch.

# Klíčová slova

DeltaRPM, RPM, Fedora, Red Hat, C, knihovna, reimplementace, binární patch.


# Reference


CHALK, Matěj. *Improved Tools for Handling deltarpm Files*. Brno, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vojnar Tomáš.

# Rozšířený abstrakt

RPM Package Manager je systém pro řízení balíčků v řadě Linuxových distribucí (tato práce se zaměřuje na platformu Fedora). Software takto distribuovaný má typicky mnoho různých verzí, každá z nich si pak vyžaduje zvláštní soubor RPM. Při aktualizaci softwaru je tedy nutné stáhnout celý nový soubor RPM, který může mít značnou velikost, ale přitom nemusí být obsahem příliš odlišný od předešlé verze. Alternativou je použití balíčků DeltaRPM, které umožňují aktualizaci softwaru realizovat stáhnutím mnohem menšího souboru, který popisuje rozdíly mezi starou a novou verzí, a následnou aplikací tohoto rozdílového souboru dosáhnout vytvoření nové verze balíčku.

Současná implementace nástrojů pro použití souborů DeltaRPM, zprostředkována projektem *deltarpm*, poskytuje nástroje pro příkazovou řádku, které mimo jiné umožňují vytváření a aplikaci souborů DeltaRPM. Tato implementace je však nevhodná pro použití jako knihovna, což vyžaduje projekt *createrepo_c*. Vznikla tedy potřeba pro novou implementaci, která poskytuje stejnou funkcionalitu vytváření a aplikace souborů DeltaRPM ve formě knihovny pro jazyk C. Nová implementace musí být také zpětně kompatibilní s původní implementací.

Cílem této práce, řešenou se společností Red Hat, je navázat na analýzu současné implementace návrhem a implementací nového aplikačního rozhraní pro jazyk C, které poskytuje nástroje pro vytváření a aplikaci souborů DeltaRPM. Tohoto cíle bylo dosaženo a nová implementace prošla testama, které ověřily, že nejenom vede kombinace poskytovaných nástrojů k přesné rekonstrukci původního souboru RPM, ale zároveň i použité soubory DeltaRPM se zcela shodují s těmi, které vytváří ekvivalentní nástroje *deltarpm* při spuštění se stejnými argumenty, a tím prokázaly zachování zpětné kompatibility. Nová implementace je nyní dostupná jako knihovna pod názvem *drpm* na platformě Fedora.

V rámci této práce je nejdříve popsán formát souborů DeltaRPM a jak nástroje realizující jejich vytváření a aplikaci fungují. Dále je provedena analýza současné implementace a jejích slabin. Následuje návrh nové implementace. Další část popisuje, jak byla tato implementace testována. Následuje nastínění, jakými dalšími směry se bude práce dál vyvíjet. Závěr pak vyhodnocuje výsledky, jich bylo dosaženo.

# Improved Tools for Handling deltarpm Files

## Declaration

I hereby declare that this bachelor's thesis was prepared as an original author's work under the supervision of Prof. Ing. Tomáš Vojnar, Ph.D. Supplementary information was provided by Red Hat, Inc. All the relevant sources of information used in the preparation of this thesis are properly cited and included in the list of references.

.....
Matěj Chalk
May 16, 2016


## Acknowledgements

I would like to thank Prof. Ing. Tomáš Vojnar, Ph.D., for his guidance and helpfulness in preparing this bachelor's thesis, and Red Hat employees for their support.

# Contents

# Chapter 1

# Introduction

RPM is a package management system used in many Linux distributions. Software packaged in such files comes in many different versions, each of which require a different RPM file. When updating a package, it is therefore necessary to download a whole new file, which may be of considerable size yet similar in content to the previous version. However, DeltaRPM packages allow software to be updated by downloading a much smaller file, containing the differences between the old and new versions, and applying those differences to create the updated version.

The current implementation of DeltaRPM functionality, provided by the `deltarpm` project, provides command-line tools for creating and applying DeltaRPM files. This implementation is not suitable for use as a library, which is needed by the `createrepo_c` project[1]. Thus, there is a need for a new implementation, which provides the same DeltaRPM creation and application functionality, but as a library for the C language. This implementation must also retain backwards compatibility with the current implementation.

The aim of this thesis, developed within the Red Hat community, is to follow an analysis of the current implementation with designing and implementing a new C API for Fedora that provides the tools for creating and applying DeltaRPMs. This has been accomplished and the new implementation has been subjected to tests that verify not only that these tools combine to reconstruct an RPM file exactly as it was originally, but that the created and applied DeltaRPM files are identical to those created by `deltarpm` tools when invoked with the same arguments, and so backwards compatibility has been ensured. The new implementation is now available as the `drpm` library in Fedora.

The rest of this thesis is structured as follows. Chapter 2 describes the format of DeltaRPM files and how they are created and applied, revealing to some degree how the `deltarpm` project implements these tools for handling DeltaRPMs. Chapter 3 looks at this implementation analytically and delves into its weaknesses. The new implementation is proposed in Chapter 4. Chapter 5 then describes how the implementation was tested. It is followed by Chapter 6, which discusses further development of the implementation. Finally, Chapter 7 offers a conclusion on the obtained results.

[1] This is a faster implementation in C of the `createrepo` project, which creates a format for critical metadata from RPM packages for dependency resolving and installation [18].

# Chapter 2

# Inner Workings of DeltaRPM Packages

This chapter describes the format of DeltaRPM package files and how the tools for creating and applying them work. First, Section 2.1 outlines the motivation for using DeltaRPM packages. Section 2.2 then briefly describes the two different types of DeltaRPM files. Section 2.3 moves on to the file format. The tools for creating and applying DeltaRPM files are described in Sections 2.4 and 2.5, respectively.

## 2.1 DeltaRPM Usage

The main advantage of DeltaRPMs is that they reduce the amount of data that has to be downloaded in order to update software. They accomplish this by creating a patch file, known as a DeltaRPM package, that stores the difference between an older and newer version of an RPM package, which can then be used to reconstruct the newer version on a system that has the older version installed. Using DeltaRPMs therefore trades bandwidth or cost for processing power, making it particularly advantageous for people with a limited network connection.

DeltaRPM tools are currently implemented in the `deltarpm` project, which defines the file format and contains command-line programs for various DeltaRPM-related tasks. The `makedeltarpm` and `applydeltarpm` programs are the focus of this thesis.

## 2.2 Types of DeltaRPM Packages

There are currently three different versions of DeltaRPM files, with each later version adding some more information. Version 3 introduces a type of DeltaRPM different from the standard type, called *"rpm-only"*. The main difference between the two types is that, while **standard** DeltaRPMs can be applied to installed RPM packages, as well as an RPM file passed explicitly as a parameter, **rpm-only** DeltaRPMs only work with the latter

option. The advantage of **rpm-only** DeltaRPMs is they are smaller and faster to combine [24].

## 2.3 File Format

This section describes the format of a DeltaRPM file, mentioning all the information contained therein. The significance of many of the data structures will become clearer in Sections 2.4 and 2.5, which describe their creation and application, respectively.

A DeltaRPM package is a binary file, whose format, in the case of a standard DeltaRPM, is similar to that of RPM files. Integer values are stored as 4 bytes in network byte order (i.e. big-endian), while variable length data is invariably preceded by a 4-byte integer representing its length, followed by the data itself. The format is illustrated in Figure 2.1.

The file format differs for different versions of DeltaRPM, with later versions adding several fields, as well as for the two different types of DeltaRPM files described in Section 2.2, which differ in the format of the uncompressed part found at the beginning of the file.

- All versions support **standard** DeltaRPMs. These begin in the same format as an RPM, containing an RPM lead, signature and header [5]. The lead is the same as in the new RPM. The signature is used as it is in normal RPMs, i.e. it contains the size and checksum of all subsequent data. The header is an exact copy of the new RPM's header, except that the payload format tag now specifies `"drpm"` instead of `"cpio"`.
- Version 3 deltas also support **"rpm-only"** DeltaRPMs, which do not work with installed RPMs. Instead of an RPM lead, signature and header, they start with a magic 4-byte string `"drpm"`, followed by another 4-byte string `"DLT3"`, where the last character identifies the version of the DeltaRPM (a lower version than 3 would not be permissible for this delta type). The next 4 bytes specify how many subsequent bytes are taken up by the *NEVR* (*name-epoch:version-release*) string of the target RPM (standard DeltaRPMs can look it up in the header). The *NEVR* is followed by the length and content of the *"add block"* (which will be elaborated on later in this chapter).

The rest of the DeltaRPM may be compressed, like a normal RPM, and is the same for both types. The first 4 bytes contain a string like `"DLT3"`, where the last character may vary as it denotes the version of the DeltaRPM (this string is therefore present twice in rpm-only deltas). The source RPM's *NEVR* string follows, identifying the old RPM.

The length of the *sequence* is next, which will be equal to 16 for rpm-only deltas (since no file system data is used), and may be longer for standard deltas. The *sequence* consists of a 16-byte MD5 sum of the contents of the old RPM, followed by a specially compressed sequence, defining which files were included, and in what order, from the list of files found in the header of the target RPM. An MD5 sum of the new RPM takes up the next 16 bytes.

If the delta version is 2 or higher, then it contains the following 4-byte integers: the full size of the complete RPM, the compression type of the target RPM (the algorithm and

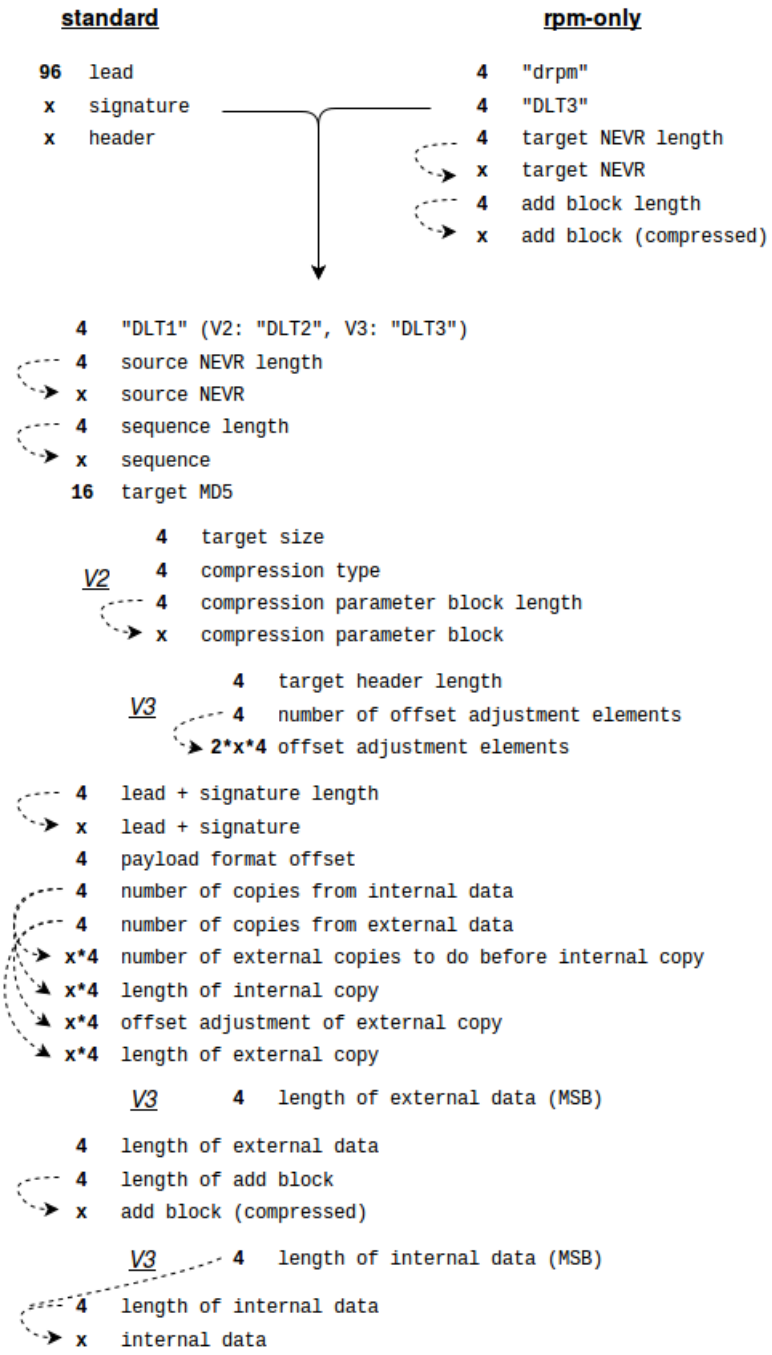



Figure 2.1: The format of a DeltaRPM file. The first part differs for standard and rpm-only DeltaRPMs, the rest may be compressed. Some entries are only present in later versions (*V2*, *V3*). The number of bytes taken up by an entry is denoted to its left in bold ($x$ means the length varies). Variable length data is preceded by a 4-byte integer that is used to determine its length, the dashed arrows illustrate these relationships.

compression level), and the size and content of a target compression parameter block (which

is not actually used or calculated and is therefore expected to be empty).

For version 3 deltas, two more things are included. First is the length of the target header (zero if it is not included in the *bsdiff* algorithm, which is the case for standard deltas), second is the number and list of offset adjustment elements, which store differences in offsets between the original and remade versions the old RPM's archive, and are used in combining DeltaRPMs.

All delta versions then include the length and content of the target RPM's lead and signature (which will be simply copied when reconstructing the RPM). Since standard DeltaRPMs patch the payload format in the header from `"drpm"` to `"cpio"`, the offset of this string within the header is then stored, to simplify changing it back (this is redundant for the rpm-only type).

What follows are two critical data structures for reconstructing data included in the *bsdiff* algorithm. They are referred to as *internal* and *external copies*. The numbers of both types of *copies* are denoted as 4-byte integers, and are followed by the contents of both. *Internal copies* are an array of 4-byte integer pairs, whose elements define the number of external copies to be performed before the next internal copy, and the length of the internal copy, respectively. *External copies* have the same structure, except the paired elements differ in meaning, the first being an offset adjustment value of the external copy, while the second specifies the length of the external copy.

The next 4 bytes (or 8 bytes since version 3) store the length of *external data*. This is the number of bytes included in the *bsdiff* algorithm from the old RPM. In the case of an rpm-only DeltaRPM, the header of the old RPM is included. The other part (or only part for standard deltas) of the old RPM included the *bsdiff* algorithm is its archive.

After that come the size and content of the *"add block"*. This is a compressed sequence of bytes created by the *bsdiff* algorithm. It stores the results of subtractions of compared bytes, where the values will tend to repeat and most often will be zero, making it highly compressible, especially with *bzip2* [10]. If the type of delta is rpm-only, the *add block* stored here will be empty, as it will have already been included at the start of the file.

Finally, the length of *internal data* is stored as a 4-byte (or 8-byte since version 3) integer, followed by the *internal data* itself. The *internal data* stores data segments from the new RPM included in the *bsdiff* algorithm that are to be copied. As with the old RPM, the header is included in the *bsdiff* algorithm for rpm-only deltas (this is because of the header not being present in the DeltaRPM file as it would be for standard deltas), and the CPIO archive of the new RPM is included for both types.

## 2.4 Creating DeltaRPM Packages

The `deltarpm` project supplies the `makedeltarpm` command-line program, which performs the creation of a DeltaRPM package [24].

### 2.4.1 Parameters

Typically, the file names of the old and new RPM must be passed to this program, but it is also possible to specify a variety of other parameters to customize its behaviour.

- `-v`: Makes `makedeltarpm` more verbose about its work.
- `-r`: An rpm-only DeltaRPM will be created, rather than the default standard DeltaRPM.
- `-V <version>`: This allows the creation of a different version of DeltaRPM than the default, which is version 3.
- `-z <compression>`: This can be used to specify a compression method to be used for compressing the DeltaRPM file (or to disable compression). The default behaviour is to use the same compression method as used in the target RPM. The compression method for the *add block* may also be specified this way. It is also possible to forbid the creation of an *add block*.
- `-s <seqfile>`: If this is set then `makedeltarpm` will write the *sequence ID* of the created DeltaRPM to the specified file. The *sequence ID* is a string concatenation of the *source NEVR* and *sequence*, and can be used to check if reconstruction is possible (see Section 2.5).
- `-p <oldrpmprint> <oldpatchrpm>`: If patch RPMs are used, this option specifies the rpm-print of the old RPM and the created patch RPM. This option tells `makedeltarpm` to exclude the files that were not included in the patch RPM but are not byteswise identical to the ones in the old RPM.
- `-u`: This switch results in the creation of an *"identity"* DeltaRPM. In this case only one RPM need be specified. An *identity* DeltaRPM can be useful to just replace the signature of an RPM or reconstruct an RPM's archive from the file system.
- `-m <mbytes>`: Memory considerations are the reason for this option, as `makedeltarpm` needs about three to four times the size of the RPMs' uncompressed archive. This option trades memory usage with the size of the created DeltaRPM, specifying the number of megabytes of memory to be used by a sliding block algorithm.

### 2.4.2 Implementation of DeltaRPM Creation

This section describes how the creation of a DeltaRPM package is implemented. The `makedeltarpm` program starts by parsing the parameters[1], before reading the old and new RPMs. The first few magic bytes of the archive are used to determine which decompression method to use (if any), and, unless the user requests otherwise, this same method is later used for compressing the payload of the DeltaRPM. The content of each RPM is used to

[1] If an rpm-only *identity* DeltaRPM is requested (i.e. both `-r` and `-u` flags are set), then a *"no-diff"* DeltaRPM is created. The RPM is read (but the archive is not decompressed) in order to create the MD5 sums and store some basic information about the RPM (*NEVR*, header size, full size) and that is all.

calculate corresponding MD5 checksums. The archive format is looked up in the header and checked to be CPIO[2].

Other information is also looked up in the RPM headers, including the compression level and the source RPM's *NEVR*. The *NEVR* string identifies an RPM package. It is made up of the package's name, version and release number, which are mandatory, and the epoch, which is optional. This string takes the format *name-epoch:version-release* (or *name-version-release* without the epoch). If an rpm-only DeltaRPM is requested, the target RPM's *NEVR* is also looked up in its header, as rpm-only deltas do not store the target RPM's header and must therefore store its *NEVR* separately.

The old and new byte sequence that will form the input for the *bsdiff* algorithm are then prepared. DeltaRPMs of type rpm-only simply concatenate the header and archive for the old and new RPMs. Standard deltas do not include the RPM headers in the *bsdiff* algorithm, but only the CPIO archives, though the archive of the old RPM is altered based on file information from the RPM header (the details of this are described in Section 2.4.3).

The *bsdiff* algorithm then takes the two (old and new) byte sequences as its input and sets up the creation of the DeltaRPM's *internal* and *external copies*, as well as its *internal data*. More information on this is to be found in Section 2.4.4.

### 2.4.3 Altering the CPIO Archive of the Old RPM

A description of how the archive of the old RPM is altered before being fed into the *bsdiff* algorithm follows.

All entries in the CPIO archive are iterated over and, unless they are skipped, are reproduced to some degree in an altered version of the archive. Such an entry is made up of a CPIO header[3], which contains information about the file, followed by the file name and its contents.

The RPM header contains information for a list of files containing the following data [6].

- The file name.
- File flags.
- An MD5 string verifying its contents.
- A device ID, if the file is character or block special.
- The file size.
- The file mode.
- Verify flags, indicating which types of verification are supported.

[2] XAR archives introduced in RPM 5.0 are not supported [15].
[3] In new ASCII format [14].

- The file name of a linked file in the case of a symbolic link.
- The file colour [17].

The file name from the CPIO entry is used to look it up in the header. The file will be skipped if any of the following conditions are true.

- The file cannot be found in the RPM header.
- The file is not included in the patch RPM, yet is not bytewise identical to the one in the old RPM (this check is only performed if patch RPMs are included).
- The file mode indicates a regular file and
  - the file sizes in the CPIO and RPM headers do not match,
  - the file flags denote a configuration file, a file that need not exist on the installed machine, or a file that is not to be included in the package[4],
  - verification by MD5 checksum or file size is unchecked[5], or
  - the file is "coloured" yet not in a *multilib* directory [17, 11].

If the file is not skipped, a new entry for it is created and is appended to the new altered archive. If the file mode indicates a symbolic link, the file contents are replaced with the name of the link's target. If the mode denotes a character or block special file, device ID information is included. The pattern of file names is also unified, so that all file names begin with the prefix `"./"`.

The end of a CPIO archive is defined by a CPIO trailer, which has the same format as any other entry, except the file name matches the string `"TRAILER!!!"`. As offsets of entries within the CPIO archive can change during this alteration, DeltaRPMs (since version 3) will also store a sequence of offset adjustment elements.

It is during this process that the DeltaRPM *sequence* is constructed[6]. Its first part is an MD5 checksum, which is calculated from the newly created CPIO entries, as well as file modes, sizes, associated device IDs, and, in the case of regular non-empty files, the MD5 or SHA256 digest for the file contents. The second part is a compressed sequence of indexes into the list of files in the RPM header, which serves to store the order in which they are included in the archive, which in turn makes looking them up faster when reconstructing this information (see Section 2.5).

### 2.4.4 Creating the Binary Patch

The `deltarpm` project uses the *bsdiff* algorithm for storing the difference between the two byte sequences. This algorithm is very efficient at creating binary patches [20].

[4] These flags correspond with the RPM macros `%config`, `%config(missingok)` and `%ghost` [7].

[5] These *verify* flags correspond with the RPM macros `%verify(md5)` and `%verify(size)` [7].

[6] This is only the case for standard deltas, rpm-only deltas do not alter the old RPM's archive, so their *sequence* is simply an MD5 checksum of the unaltered archive.

One small modification in the source code results in changes throughout the resulting binaries, as locations of pointers change, and so delta algorithms that are tailored for text files perform poorly in the case of binary files [21]. However, there are some observations to be made about the way in which executable files change. One is that in regions not directly affected by a modification, differences will be quite sparse. Another is that the locality of references will result in a large number of addresses adjusted by the same offset within one region. This means that, if these regions are matched against each other, the bytewise difference will be mostly zero, and the non-zero values will often repeat, which means this sequence of differences will be highly compressible [20]. The *bsdiff* algorithm takes advantage of these observations.

Using this algorithm, `deltarpm` goes about creating a patch in the following way. It creates an index for the "old" byte sequence, using hashing [26] by default, though `makedeltarpm` may be configured to use suffix sort instead [16]. This index is used to move through the "new" byte sequence and find the next region that matches exactly, but also contains at least 32 bytes[7] that do not match the forward extension of the previous match. The previous match is then extended forwards and the new match is extended backwards, with both extensions requiring at least half of the bytes to match (if these extensions result in an overlap, `makedeltarpm` finds a good place to split in order to maximize the percentage of matched bytes). This approximate match will roughly correspond to a block of executable code derived from an unmodified region of source code [20].

Figure 2.2 shows how the algorithm creates an array of two *offset-length* pairs, as well as the *add block* [10]. The array is used to construct the data structures that will be stored on disk.

- The **internal copies** are pairs of 4-byte integers, made up of
  - the number of external copies that precede this internal copy, and
  - the length of this internal copy.
- The **external copies** are also pairs of 4-byte integers, made up of
  - the offset adjustment of this external copy (i.e. a number that should be added to the offset in the external data), and
  - the length of this external copy.
- The **internal data** is a copy of all the regions in the "new" byte sequence that did not match up, and is therefore stored (consecutively) in the DeltaRPM, the boundary of each region being determined by the lengths of the individual *internal copies*.

## 2.5 Applying DeltaRPM Packages

DeltaRPM packages can be applied, i.e. used to reconstruct the newer version of the RPM package, via the `applydeltarpm` program [23]. As an alternative to using the old RPM

[7] The `bsdiff` program differs in that it requires mismatches in only 8 bytes.

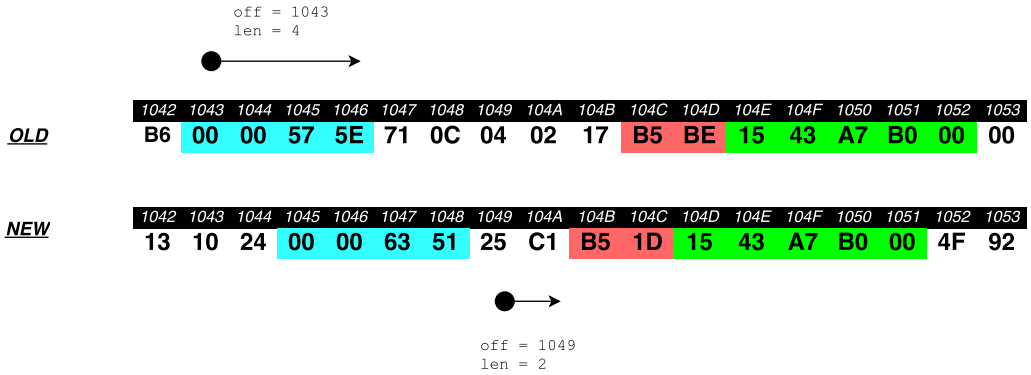



Figure 2.2: A new match has been found and has been extended backwards, while the previous match has been extended forwards. The offset and length of the forward extension of the previous match in the old sequence, and the offset and length of the bytes between the extensions in the new sequence, are both stored and later used to construct the *internal* and *external copies* and *internal data*, that end up being written to the DeltaRPM. The bytewise subtractions of the previous match's forward extensions are added to the *add block* (in this case `00 00 0C F3` would be appended).

to perform the reconstruction, on-disk data may be used if the old RPM is installed. The `applydeltarpm` program may also be used to check that the reconstruction is possible, or to print information contained within the DeltaRPM.

### 2.5.1 Parameters

The basic use case of `applydeltarpm` is to reconstruct the new RPM. The name of the DeltaRPM file, as well as the file name the new RPM will take, are mandatory in this case. The `-r` parameter enables passing the name of the old RPM file explicitly. If this parameter is not present, then the old RPM that was used to construct the DeltaRPM must have been installed, and data from the file system is used in the reconstruction instead. The command-line program may also be verbose about its work or print the percentage of completion of the reconstruction while performing it, if the `-v` or `-p` options are set.

Another mode of usage is to check that the reconstruction is possible, without actually performing it. This is done by checking file information contained in the header of the old RPM. The user specifies whether it is sufficient to check that the file sizes have not changed (option `-C`), or a (slower) full check that the contents of the files have not changed should be performed (option `-c`). Either way, this checking may be done by supplying the DeltaRPM (in this case the `-r` option cannot be present, as the files must be present on disk in order to be checked), or by merely supplying the *sequence ID* of the DeltaRPM instead, which contains all the information that is needed to perform reconstruction checking (this is done by passing an `-s` option followed by the *sequence ID*).

Finally, setting the `-i` flag makes `applydeltarpm` simply read the DeltaRPM file and print information that it contains.

### 2.5.2 Implementation of DeltaRPM Application

First, `applydeltarpm` parses the command line arguments. Unless only a *sequence ID* check is requested, the DeltaRPM is read and its data structures and additional information extracted (if the information flag is set, `applydeltarpm` will then simply print out the extracted information and exit).

If the old RPM is passed explicitly, then its entire contents are read. The MD5 checksum contained in its signature is used to verify that its contents match the old RPM used to create the delta, by comparing it with the checksum found in the DeltaRPM *sequence*.

If the old RPM is not passed explicitly, it is assumed that it has been installed and so data from the file system should be used in the reconstruction. The RPM database is queried, and if an installed RPM with a matching name, epoch (if specified), version and release is found, its header is extracted from the database. If the old RPM is not installed, an error is returned. An error is also returned if the DeltaRPM is of type rpm-only, or the target header stored in the standard delta indicates a source RPM, as neither of these variants work with file system data.

Both options end up yielding the header of the old RPM, and the *NEVR* tag contained therein is checked, to see that it matches the *source NEVR* stored in the DeltaRPM.

In the case of a standard delta, the header's file list and file order (stored in the second part of the DeltaRPM *sequence*) are used to construct an index of the files contained in the CPIO archive, marking the order, length of the CPIO header (including the file name), length of the file content and the offset within the archive for each file.

If `applydeltarpm` is run in check mode, then during this expansion of the file data, each file is checked to match the file originally used to create the DeltaRPM. Depending on the input parameter, either only the file size is checked, or the file contents are used to construct an MD5 or SHA256 hash (depending on what the file digest algorithm specifies in the RPM header), that is then compared against the one stored in the RPM header, to see that the contents are identical[8]. Any mismatches are reported, and the program run in check mode will now exit, as what follows is the reconstruction itself, which is described in Section 2.5.3.

### 2.5.3 Reconstructing the RPM Package

The target RPM is to be reconstructed exactly, including all its parts, i.e. the lead, signature, header and archive.

The lead and signature of the target RPM are stored in the DeltaRPM, so their reconstruction is trivial.

The reconstruction of the header differs for different types of DeltaRPMs. The file format

[8] If the file size has increased, `applydeltarpm` will check if it is an ELF library modified by `prelink` [12]. If so, it has `prelink` write the original binary to a temporary file in order to be able to perform the check.

of the standard delta actually includes a copy of the target RPM's header, with the only difference being that the contents of the payload format tag are rewritten from `"cpio"` to `"drpm"`. All that is needed, therefore, is to patch the payload format back and an exact copy of the header is done. On the other hand, rpm-only DeltaRPMs do not contain an RPM header as part of the file format, and it is for this reason that the old and new headers are included in the *bsdiff* algorithm by `makedeltarpm`. In this case, therefore, the reconstruction of the header is part of the reconstruction algorithm (see Listing 2.1). The only difference is that the header is not compressed before being written to the file, which can be accomplished by only starting to compress outputted data after the amount exceeds the size of the target header (which is stored in rpm-only deltas).

If present, the *add block* is decompressed and is applied (i.e. bytewise added) to all *external data*. This *external data* has to be fetched from somewhere (only its size is stored in the delta), and the methods for doing this vary in the following ways.

- If file system data is used, then (constructed) CPIO entries for each file in the order specified in the file index are gradually concatenated to form the *external data*[9].
- If the old RPM is used, then the method of fetching data differs for the two delta types.
  - Standard DeltaRPMs read the CPIO archive of the old RPM one entry at a time and alter the entries in the same way `makedeltarpm` does (see 2.4.3), in order to produce data identical to that which formed the input for the *bsdiff* algorithm.
  - Since no such alteration is done for rpm-only deltas, the *external data* is simply made up of an exact copy of the CPIO archive in this case.

In order to abstract the fetching of *external data* from the reconstruction algorithm, `applydeltarpm` uses a list of equally large blocks which are gradually filled by these various methods, with a new block being created or an older block being reused when the current block has been filled up.

The reconstruction is done by using the information from the DeltaRPMs *internal* and *external copies* to write appropriate amounts of *external* and *internal data* at a time. The algorithm follows directly from the information that the *copies* contain (described in Section 2.3).

The Python code shown in Listing 2.1 illustrates how the reconstruction is done. The *external* and *internal data* are represented by the variables `ext_data` and `int_data`, which for the sake of simplicity are here assumed to be lists of bytes already filled with the necessary data. The *copies* represented by `int_copies` and `ext_copies` are presented here as lists of objects with two named attributes each (while in fact the DeltaRPM stores them as 4-byte integer arrays), in order to better reflect the semantics of the values they hold. It may also be useful to clarify that the `[off:][:len]` notation repeatedly used represents Python list slicing, meaning that `len` elements are read from the list starting at index `off`. Finally, `new_rpm` is assumed to be an object that defines the `compr_write` method,

[9]The possibility of an ELF library modified by `prelink` [12] is taken into account, and in that case the original is restored into a temporary file and its contents are read from there.

which compresses and then writes the supplied bytes to the new RPM file (it should also be assumed that this method only starts compressing after the size of the target RPM header has been reached in the case of an rpm-only delta).

```
aoff = 0 # add block offset
ioff = 0 # internal copies offset
eoff = 0 # external copies offset
edone = 0 # number of completed external copies

for icopy in int_copies:

    # perform specified number of external copies
    for ecopy in ext_copies[edone:][:icopy.todo]:
        eoff += ecopy.offadj # adjust external data offset
        buf = ext_data[eoff:][:ecopy.length]

        # patch external data with add block
        if add_block:
            for i in range(ecopy.length):
                buf[i] += add_block[aoff+i]
            aoff += ecopy.length

        new_rpm.compr_write(buf)
    edone += icopy.todo

    # perform internal copy
    new_rpm.compr_write(int_data[ioff:][:icopy.length])
    ioff += icopy.length
```

Listing 2.1: The Reconstruction Algorithm

After this algorithm is complete and all the data has been written to the new RPM file, a final check is performed. An MD5 checksum of all the written data is constructed and compared with the *target MD5* contained in the DeltaRPM[10]. A match confirms that `applydeltarpm` has succeeded in reconstructing an exact copy of the new RPM package.

[10] If the *target MD5* is empty, then the MD5 checksum from the new RPM's signature is compared instead.

# Chapter 3

# Analysis of the Current Implementation

This chapter analyses the current implementation of DeltaRPMs, i.e. the `deltarpm` project (implemented in C), and focuses on its weaknesses. Specifically, Section 3.1 covers the documentation for the project, Section 3.2 then analyses the code structure, Section 3.3 describes where the implementation makes use of external libraries, before Section 3.4 focuses on its most important weakness, namely that it is unsuited to be reused as a library for developers.

## 3.1 The Documentation

The documentation for this project is generally lacking, especially regarding how `deltarpm` actually works.

The format of DeltaRPM files is described in good detail in the `README` provided. The exact structure of the binary file is well laid out. It also from here that one can get a small insight into the significance of most the data structures contained inside a DeltaRPM.

Other documentation can be found in the manual pages for the programs distributed by the project (e.g. `makedeltarpm` and `applydeltarpm`). The various options all get a concise user-oriented explanation, and obsolete options are handled by being left quietly undocumented[1].

The source code is mostly uncommented, though those comments that are occasionally included are very useful (in particular some unusually detailed comments in the *bsdiff* section, even including one illustration of the significance of critical variables' values in relation to the compared byte sequences). On the whole, however, the source code is

[1] From the man page of `makedeltarpm` in an older `deltarpm` version: "Specifying `-l` file, one can give a file with a list of files needed to be excluded from the delta process. Nowadays it is relayed on all of those files be either marked as `%config` or as `%verify(nomd5)` in spec files. Thus, this option is obsolete." The usage message printed by `makedeltarpm` when run with invalid arguments still contains this option, however, while leaving most other options out.

badly suited to trying to understand how the whole process works. In addition to the lack comments, variable names rarely give a good indication of their significance and are mostly very short, with one- to three-character long variable names being the norm. Function names are mostly an improvement in this regard.

The principles of the BSD licensed *bsdiff* code that `deltarpm` uses (and slightly modifies) are documented elsewhere by its original author [20, 21]. However, the general lack of documentation for the rest means that, in order to acquire a decent understanding of how `deltarpm` works, one is left slowly piecing it together from the code itself (which, only taking source files relevant to `makedeltarpm` and `applydeltarpm` into account, contains over eight thousand lines of code).

## 3.2 Code Structure

The `main` functions that implement `makedeltarpm` and `applydeltarpm` are both rather lengthy, averaging around eight hundred lines of code. This is partly due to the fact that `deltarpm` for the most part only uses functions to avoid repetitions of larger parts of the code, not to deconstruct the work flow into logical segments.

Both programs can run in a variety of modes, which overlap with each other to varying degrees. An effort has been made for the code of all these modes to intertwine in the `main` function, instead of delegating particularly disparate modes to different function calls and putting common code segments in functions. One unfortunate result of this is that there are several examples of code being executed in a mode that has no use for it[2].

The scope of variables is mostly well defined, and several structures are implemented that group related variables. This discipline seems to have dwindled with later additions to the code, however, as the source code of both programs now contains a large amount of global variables. These are often not grouped together in a data structure even when the logic of the program indicates that they should be. Sometimes these variables are only used by one function, and so the reasons for them being global disappear entirely. Global variables are mainly used for `makedeltarpm`'s sliding block algorithm in memory usage limiting mode[3] and `applydeltarpm`'s *external data* blocks[4].

On the other hand, the implementation does separate some of the code into several useful and fairly well encapsulated modules.

As `deltarpm` has to do a lot of work with compressed data, a structure called `cfile`

[2]For example, `applydeltarpm -i`, which only needs to read the DeltaRPM and print out information about it, will also needlessly set up blocks that only modes that actually perform a reconstruction make use of. This happens when run in one of the check modes too, and they will also create an index of entry offsets and lengths within the CPIO archive, which, again, is only used when reconstructing.

Another example is that `makedeltarpm` creates an array of offset adjustment elements for the CPIO archive whatever the version, despite the fact that only version 3 deltas include this information in the DeltaRPM file.

[3]Admittedly, this latest addition to the code would have otherwise warranted a more significant code restructure, and is accompanied with the comment: “hack: global for now”.

[4]Twenty three global variables, only three which are data structures.

is defined. This structure functions much like a class (its components include function pointers as simulations of methods), and is used for compressing or decompressing data (usually to or from a file, though buffers in memory can also be used). It can be instructed to update an MD5 context with outputted data, and keeps track of its size. The supported compression algorithms are *gzip*, *bzip2*, *lzma* and *xz*. These do not include *lzip*, which may also be used in RPM packages [2].

A structure called `rpmhead` is used for reading an RPM header structure (both the signature and header are in this format [5]). This module then contains several functions for extracting information of various data types from the header structure, using a tag number used by RPM as an argument. The data is stored in a raw format, as only the number of index entries, the size of the store and a pointer to where the store starts are separated from the binary representation of the header.

There also modules for initializing, updating and finalizing checksums, both for MD5 and SHA256. These modules are made up of code from public domain implementations of these algorithms.

## 3.3 Use of External Libraries

The modules for RPM header structures and digest algorithms do not use existing libraries that specialize in these things. This poses a potential problem for RPM headers particularly, as a change in format would necessitate a reimplementation in order to avoid errors[5].

The `cfile` module does, however, use the appropriate libraries for compressing and decompressing. These are `zlib` (*gzip*) [4], `libbzip2` (*bzip2*) [25] and `liblzma` (*lzma* and *xz*) [8].

Although `rpmlib`'s RPM Header API [3] is not used by `deltarpm`, this library is used for fetching the header of an installed RPM from the database[6].

## 3.4 Re-Usability

The `deltarpm` project offers DeltaRPM creation and application functionality only as command line programs. However, a C API is more suited to projects like `createrepo_c` [18].

[5] "1. The file format is subject to change. 2. If a package file is to be manipulated somehow, you are *strongly* urged to use the appropriate rpmlib routines to access the package file. Why? See point number 1!" [5]

[6] The implementation of this is particularly curious, however. A tool named `rpmdumpheader` is also distributed with the `deltarpm` project. It accepts an RPM's *NEVR* string as an argument and, if the RPM database succeeds in retrieving it, prints the header in its binary format to the standard output. The way `applydeltarpm` makes use of this tool is to create a child process, pipe its output to the input of the parent process and have it execute `rpmdumpheader` (supplying the *NEVR* string) before exiting, while the parent process uses the `rpmhead` module to read it.

This technique is perplexing, as `rpmdumpheader` and `applydeltarpm` code are both part of the same repository, so it makes no sense not to retrieve this header with a simple "internal" function call instead.

The problem is that the current implementation is not well suited to be used as such.

One of the chief reasons for this is error management. Every time an error is discovered, it is followed by printing an error message to the standard error output and an `exit(1)` call (or occasionally `abort()`). The result of this is that most functions do not return error indicators. A library, on the other hand, needs a mechanism for errors to surface from nested function calls to the called API function. Another result of the `exit` calls is that `deltarpm` relies on it to free all resources (allocated memory, open file descriptors). Again, this is not possible with a library, which cannot terminate the program and must therefore ensure all resources are freed "manually".

# Chapter 4

# Design of the New Implementation

This chapter describes the design and implementation of the new library for handling DeltaRPM packages, called `drpm`. While many of the algorithms are essentially the same is those used in `deltarpm`'s implementation, since there are no reasons to change them and major alterations would present unnecessary complications in trying to reproduce the same data structures stored in the DeltaRPM, an effort has been made to make the code more readable and more adaptable to changes.

Section 4.1 presents the new interface provided by `drpm`. Section 4.2 then describes changes in code structure in comparison to the original implementation. Finally, Section 4.3 focuses on helpful modules created as part of the implementation.

## 4.1 The Developer Interface

Redistributing command-line tool functionality as a C library means a change of interface by definition. Instead of executable programs modifiable by command-line options, the new implementation provides a header file (`drpm.h`) declaring a set of functions, and a shared library (`libdrpm.so`) for the linker.

The program `makedeltarpm`, whose CLI is described in Section 2.4.1, is therefore replaced with a function called `drpm_make`. This function accepts four arguments. The first three are strings representing the file names of the old and new RPM packages, as well as the DeltaRPM package that is the main result of this call[1]. The last argument is a pointer to a structure defined as `drpm_make_options` in the API, and is used to specify various options in order to customize the DeltaRPM creation process. The reason a structure is necessary for this, instead of e.g. an integer with options passed by a bitwise OR of macros, is that several of the additional options necessarily take the form of strings (file names for writing out the *sequence ID* or adding RPM patches) or integers (e.g. version, compression level). The API contains a set of functions for adding options to this structure, as well as a

[1]The `drpm_make` function may also be used to create an identity DeltaRPM, which only uses one RPM file (see `makedeltarpm`'s `-u` flag in Section 2.4.1). This may be done by passing a `NULL` pointer instead of one of the RPM file names.

function to initialize it and another to destroy it. In order to simplify creating a DeltaRPM with the default options, one may invoke `drpm_make` while passing a `NULL` pointer as the fourth argument, and ignore the `drpm_make_options` suite. This shortens the basic use case to one function call.

Unlike `makedeltarpm`, where, despite the multitude of options, the type of output is the same (i.e. whatever the options, the result will be the creation of a DeltaRPM), `applydeltarpm` can be used for a variety of different tasks. As described in Section 2.5.1, while the basic usage is to (re)construct an RPM file using a DeltaRPM file (with the older version of the RPM provided either implicitly or explicitly), `applydeltarpm` may also be used to merely check that the reconstruction is possible (based on the DeltaRPM or its *sequence ID*) or print out information about the DeltaRPM[2]. These behaviours, despite having some code in common, are quite disparate, and while that is not unusual for a CLI, capturing this interface in a single C function does not lend itself to intuitive or elegant usage. The `drpm` project therefore splits these modes of usage into separate functions.

- `drpm_apply`: This function performs the reconstruction of an RPM. It takes three arguments, all of them being file names. The first is the name of the old RPM (if installed data is to be used, this should be `NULL`), the second is the name of the DeltaRPM file, and the third is the file name that the resulting RPM will take.
- `drpm_check`: This is used to check that the reconstruction is possible based on the provided DeltaRPM (its file name is the first argument of this function). This is only possible if the old RPM is installed. There are two modes in which to check that the files installed by the RPM have not changed since the DeltaRPM was created. One only verifies that file sizes have not changed, while the other also checks the contents of the files. The second parameter (an integer expecting the value of one of the `DRPM_CHECK_FILESIZES` or `DRPM_CHECK_FULL` macros) serves to specify which mode is required (corresponding to `applydeltarpm`'s `-c` and `-C` flags, as described in Section 2.5.1).
- `drpm_check_sequence`: This function is used for the same purpose as `drpm_check`, except that a *sequence ID* is provided instead of a DeltaRPM, which is reflected in the corresponding parameter. Since the implementation of `deltarpm` allows for it, it is also possible to specify the name of the old RPM (a parameter is added for this) or not request a file check (a `DRPM_CHECK_NONE` macro is used for this). In these cases, the check performed by this function will be a lot less thorough.

All of the described functions return an integer representing an error value[3]. The types of errors that are defined by `drpm` are

- `DRPM_ERR_OK` (no error occurred),

[2]The first distributed version of `drpm` provided a small library for fetching information from DeltaRPM files. The library has now been extended significantly (to enable the creation and application of DeltaRPMs, as described in this thesis), but the DeltaRPM reading interface remains the same, though the implementation of `drpm_read` has been modified for the purpose of having the code for reading DeltaRPM files be reused by `drpm_apply` and `drpm_check`, which also need to read DeltaRPMs.

[3]A function called `drpm_strerror` may be used to get the string representation of an error value.

- `DRPM_ERR_MEMORY` (memory allocation error),
- `DRPM_ERR_ARGS` (bad user arguments),
- `DRPM_ERR_IO` (I/O error – probably caused by trying to read a non existent file),
- `DRPM_ERR_FORMAT` (wrong file format – probably of an RPM or DeltaRPM),
- `DRPM_ERR_CONFIG` (misconfigured external library),
- `DRPM_ERR_OTHER` (an unspecified error),
- `DRPM_ERR_OVERFLOW` (file too large – specifically the uncompressed RPM archive),
- `DRPM_ERR_PROG` (internal programming error),
- `DRPM_ERR_MISMATCH` (file changed – detected by `drpm_check`, `drpm_check_sequence` or `drpm_apply`),
- `DRPM_ERR_NOINSTALL` (old RPM not installed).

## 4.2 Code Structure

The new implementation's code structure seeks to amend many of the weaknesses of `deltarpm`'s code structure, which is described in Section 3.2. At the level of implementing the functions provided in the API, parts of the code that form logical segments have been divided into separate functions. This not only makes for better readability, as the top level functions work on a higher level of abstraction and are much shorter, but also solves a potential problem caused by splitting `applydeltarpm` into several functions. Many of their tasks overlap, but as these tasks are isolated and delegated to functions, reoccurrences of larger stretches of code are prevented. Instead, the overlapping of a task presents itself merely as a call to the same function, and slight differences within that task are resolved by passing different arguments.

Global variables are avoided and a greater effort has been made to group related variables into data structures. When these data structures also lend themselves to an object-like usage, i.e. they need an initializing function and a function that frees resources, and operations performed on them may be encapsulated, they are also separated out into modules, which are described in Section 4.3. When available, external libraries are used for tasks that they specialize in (e.g. `openssl`'s cryptographic functions are used to calculate MD5 and SHA256 checksums [19], and RPM related tasks are performed to a much greater extent with the aid of `rpmlib` APIs, as described in Section 4.3.1).

Memory management, which `deltarpm` leaves to `exit`, is done in a consistent style, with most functions having a `cleanup` label, that may be prematurely jumped to on error, where all resources used by the function are freed. Error detection also works very differently. Since a library function cannot terminate the program and is not expected to print messages, a mechanism for enabling errors to surface from a stack of function calls is necessary. It is realized by most functions returning an error value. If it indicates an error, the calling function will perform a clean-up and return the same error, and this behaviour will be

replicated all the way back down to the library function, ensuring the user may detect the error and no memory leaks are caused. It may be noted that, while this change in error and memory management was necessary in order to reuse code for `deltarpm`'s command-line programs for a library, the library's implementation does allow for its extension into a command-line tool, as one need only parse some command-line arguments and then call one of the API functions.

Finally, an effort has been made for the code to be more readable, including descriptive variable and function names and accompanying comments. Parametrized macros are utilized to more clearly express what a calculation does, especially if it often reoccurs[4].

## 4.3 Modules

This section describes some of the modules contained in the `drpm` implementation. Most of them are designed like classes, coming with "constructor" and "destructor" functions, which must be called explicitly, however, as they are confined by the limits imposed by C not being an object oriented language. These modules encapsulate their data, which is realized in C by only allowing a pointer to the data structure to be used outside the module (the initializing function must therefore also allocate memory for the structure itself).

It should be noted that there are also data structures whose contents are not encapsulated. Most notably this is true of `struct deltarpm`, which contains all the DeltaRPM data and has functions dedicated to writing and reading it. Its data encapsulation would be impractical as it is so central to the whole implementation.

### 4.3.1 RPM

DeltaRPMs by definition deal very closely with the inner structure of RPM files. One of the tasks often required is to extract information from an RPM header structure, which is the format of both the signature[5] and header. As noted in Section 3.3, `deltarpm` traverses this format directly, despite the fact that `rpmlib` provides a Header API for these kinds of tasks, which ensures they will be performed compatibly with the current RPM format [3].

The `rpm` module serves to read an RPM and store it in a data structure that separates the lead, signature, header and archive. The signature and header are stored using the `Header` data type provided by `rpmlib`'s Header API, and all operations on these structures are performed using the functions provided with it[6]. The archive may be read along with the

[4] E.g. RPM signatures and CPIO header entries are followed by padding bytes, their amount is calculated with the macro `PADDING(offset, align)`.

[5] A more tricky case of manipulating an RPM header structure comes when constructing a standard DeltaRPM, where it is also necessary to remake the RPM signature, so that it validates the content of the DeltaRPM. The implementation of `deltarpm` does this by writing out the individual bytes directly, while `drpm` uses the appropriate `rpmlib` routines.

[6] The one exception is the storing of the payload format offset stored in the DeltaRPM (see Section 2.3), since this is not possible using `rpmlib`, as it breaks encapsulation. Since `drpm` uses `rpmlib` routines for patching the payload format, this offset is not actually used, but it is calculated nonetheless in order

rest of the RPM, and the `rpm` module provides functions for iteratively fetching its data. The module also contains a function that fetches the header of an installed RPM from the database using `rpmlib` routines (improving on `deltarpm`'s `rpmdumpheader` usage described in a footnote in Section 3.3).

### 4.3.2 Compression and Decompression

As mentioned in Section 3.2, `deltarpm` contains a module for dealing with compression and decompression. The new implementation is similar in this regard, though it has separate modules for compression (`compstrm`) and decompression (`decompstrm`), as there is no reason for them to overlap. Furthermore, `drpm` tries to implement the data structures in a more semantic way, e.g. preferring the use of C unions instead of having different types of data stored in the same variable (of type `void *`) depending on the context.

The data is also encapsulated, not allowing for direct manipulation outside of the module, unlike `deltarpm`, which takes care of making sure the header is not compressed in `applydeltarpm` by temporarily changing function pointers. This is realized with a small wrapper module in `drpm`.

Both implementations use the same libraries for performing the (de)compressions (these are mentioned in Section 3.3). In addition, `drpm` implements *lzip* compression using the `lzlib` library [9]. The reason for this is that newer versions of RPM also support *lzip* compression[7] [2].

### 4.3.3 Blocks

This module is used for buffering *external data* when reconstructing the RPM (see Section 2.5.3). Blocks of a fixed size are gradually filled with the *external data* using different methods.

- If an old RPM is passed explicitly and the DeltaRPM is a standard delta, then the RPM's CPIO archive is read and its entries are altered in the same way as in the creation of the DeltaRPM (see Section 2.4.3).
- If an old RPM is passed explicitly and the DeltaRPM is an rpm-only delta, then the archive is read without any alterations, but only after the header of the old RPM has been prepended (once again, this corresponds to what forms the input to the *bsdiff* algorithm when creating the delta).
- If no old RPM is passed explicitly, and therefore its files have been installed and are present in the file system, these files are read in the order in which they appear in the RPM header (this is why the file order is stored in the delta's *sequence*).

to create identical DeltaRPM files to those created by `makedeltarpm`, which simplifies verification of their equivalence.

[7] This is only the case for OpenSUSE, however, and so *lzip* compression is only available to be used on that platform.

This technique is used in the same way that it is used by `deltarpm`, the difference is that `drpm` encapsulates all this in a module instead of having a large amount global variables scattered throughout the code base. Since the different methods of filling the blocks require storing different variables, the `blocks` data structure uses unions to store them effectively and semantically.

# Chapter 5

# Testing the New Implementation

This chapter describes the tests performed on the new implementation of DeltaRPM tools to verify its success. Since the new implementation seeks to offer the same functionality as the current implementation, but with a different interface, the aim of the tests is to verify that both implementations are equivalent in functionality, i.e. the same inputs to `drpm`'s tools (the functions `drpm_make`, `drpm_apply` and so on) result in the same outputs as they do with `deltarpm`'s tools (`makedeltarpm` and `applydeltarpm`).

As far as creating DeltaRPM packages is concerned, the method used to test the functional equivalence is to create DeltaRPM files using both tools, specifying the same arguments (adjusted to their different interfaces, of course), and check that the files are identical[1].

The method of testing the application of DeltaRPM packages follows directly from what this process is meant to achieve, that is to reconstruct the RPM exactly. The RPM packages constructed by `drpm_apply` are checked to be identical to the original target RPM packages that were used to create the used DeltaRPM. In this case, there is no need to run the corresponding `deltarpm` tool to compare the functionality.

These methods were used to test the new implementation, and were instrumental in detecting many bugs. The `drpm` project comes with a test suite that uses these methods to check whether the code works correctly. Its implementation is described in Section 5.1. One functionality that is not tested in this suite is checking and reconstructing using file system data as opposed to an RPM file. The reason for this is that these tests are designed to run automatically during the process of installing the `drpm` library. A prepared RPM package would have to be installed on the user's system before running tests, and it is dubious that the user would be willing to grant privileges for this. This functionality, i.e. that of `drpm_apply`, `drpm_check` and `drpm_check_sequence` using an installed RPM package, has therefore been tested separately on Fedora 20 running on *x86_64* architecture.

[1] It is not strictly speaking necessary that the DeltaRPMs be entirely identical in order to both be used equivalently in reconstructing the RPM. However, the permissible differences are few and unimportant (e.g. an rpm-only delta's payload format offset is not actually used, and so could contain any value), so in the interest of simplifying the equivalence test to checking that the files are identical, `drpm` makes sure these few cases also match `deltarpm`'s behaviour.

## 5.1 Test Suite

The new implementation comes with a set of automated tests that are run during the installation process. These tests come with two pairs of RPM packages (an old and new version for each) and consist of three parts.

- A program implemented in C uses the framework provided by the `cmocka` library to run groups of tests on `drpm`'s API functions [22]. The first group tests `drpm_make`, running it several times with different parameters to produce DeltaRPM packages that will be used by later tests. The second group uses `drpm` API functions to extract information from the created DeltaRPMs, and checks these various pieces of information to see that they reflect the parameters used to create the deltas (e.g. the specified compression was used, the stored size of the target RPM matches the actual file size) and also meet certain logical requirements for a DeltaRPM in general (e.g. MD5 checksums take up the defined number of bytes, the length of the header will be zero or non-zero depending on the type of delta). The third group only runs `drpm_check_sequence`, providing it with the old RPM file, to check that it verifies it successfully. The fourth group then uses the DeltaRPMs created by the first to reconstruct the new RPMs by running `drpm_apply`.
- A script written in Bash is executed after the C program finishes. Its role is to use the `sha256sum` program to check whether files are identical (files are compared indirectly by generating a SHA256 hash for each and comparing the hashes) [13]. It runs `makedeltarpm` with equivalent parameter combinations to those used in the C program, and the DeltaRPM files that it generates are checked to be identical to those created by the C program[2]. The parameters of one the tests of DeltaRPM creation specifies writing out the DeltaRPM's *sequence ID* to a file, and these files are also compared to see that they are identical for both invocations. The RPM files reconstructed by `drpm_apply` are then compared with corresponding original target RPMs, again using `sha256sum`.
- The last test reruns the C program through the `valgrind` tool, if it is available, in order to detect any memory leaks that may be caused by `drpm` functions.

These tests were run in scratch builds of the `drpm` package on Fedora[3], using the Koji build system [1]. They succeed on all architectures tested with Koji, which include *x86_64*, *i686* (Intel), *armv7hl* (ARM), *ppc64*, *ppc64le* (PowerPC) and *s390x* (IBM).

[2] If `makedeltarpm` is not installed, `drpm` has the SHA256 sums of files created by the same `makedeltarpm` executions prepared in a text file, and uses these to perform the comparison instead.

[3] Koji builds on the release currently in development, which is the future Fedora 25 at the time writing.

# Chapter 6

# Future Development

This chapter concerns itself with the current state of the new implementation and how it may be built upon in the future. There is still one aspect of `makedeltarpm` not captured by `drpm_make`, which is the memory usage limiting mode (see Section 2.4.1). The reason for this is that this mode was not fully understood when designing the implementation, which had much to do with how loosely it is integrated in `deltarpm`'s implementation (partly due to the fact that it was a late addition in `deltarpm`), and so the design failed to take it into account. The main problem that needs to be addressed in making room for this mode, is that `drpm`'s RPM and (de)compression modules, are implemented in a way that they store large amounts of data in memory, which goes against the whole principle of this mode. It is therefore necessary to change this approach for both modules. The `rpm` module can do this by keeping the RPM file open for interrupted reads until it is no longer needed instead of reading its whole contents at the first opportunity. The `compstrm` and `decompstrm` modules then need to reuse parts of their internal buffers that have already been outputted. This change is one for the near future.

As for longer term developments, Red Hat employees from the RPM team are considering having DeltaRPMs use parallel compression, which has been introduced to RPMs. An argument has also been made that it would be more practical if DeltaRPMs were used to, instead of reconstructing the RPM file exactly as it was and compressing its payload in the process, rather reconstruct the same RPM but leave its payload uncompressed. The reasoning is that in the typical use case the RPM's reconstruction would be immediately followed by its installation, and it would be a waste of processing power to compress the payload in the reconstruction before decompressing it again in the installation, while missing the main advantage of a reduced file size because the file would not need to be transferred in between. The problem that would need to be solved in making this change is that this would disrupt the way DeltaRPMs are used to verify that the resulting RPM is correct, which is to compare its MD5 checksum to the one stored in the DeltaRPM. These checksums are calculated using compressed data. In order to verify the successful reconstruction of the RPM without compressing, it would be necessary to calculate checksums of uncompressed data. These checksums could be easily calculated during the creation of DeltaRPMs, since this already entails decompressing the RPM's payload, but storing them in the DeltaRPM while preserving backwards compatibility would necessitate extending the file format and labelling

it as a version 4 DeltaRPM. Another option would be to add an option to DeltaRPM creation that would result in writing a checksum of uncompressed RPM data to a separate file (similarly to how the *sequence ID* may be generated), which could then supplied as an argument to the reconstruction, signalling that this RPM should not be compressed and the provided checksum should be used to verify its legitimacy instead.

An issue that would surely arise with these extensions is whether it would not make sense to extend the `deltarpm` tools[1] in the same way as `drpm`. And ultimately, it may well make the most sense to merge these two implementations into one that provides both the command-line and C API interfaces. The structure of the new implementation allows for a simple extension into a command line interface, as one would merely have to add a command line argument parser before calling a `drpm` function.

[1] The code base for `deltarpm` is now administered by the RPM team, though it has not made any changes to it so far.

# Chapter 7

# Conclusion

This thesis describes the aim and the realization of a new implementation for handling DeltaRPM packages, which solves the main weakness of the original implementation, which is that it cannot be used as a library. The new implementation comes in the form of a C API, and it makes an effort to be more readable and better documented. It is also well suited to be extended into a command line program, with a view to the plausible future need of merging the two DeltaRPM handling suites.

At the same time, backwards compatibility with the original implementation is maintained. This is proved by tests which verify that the created DeltaRPM packages are identical for both tools when invoked with the same arguments. The tests also prove that RPM packages reconstructed with the new tools match their corresponding original RPM packages exactly, which is a prerequisite for a successful implementation of tools that apply DeltaRPM packages.

The new implementation has been accepted by other Red Hat employees, and is ready to go through the Fedora package update process, after which the `createrepo_c` project can starts making use of its new functions.

# Bibliography


[1] *Koji* [online], 2008 [cit. 2016-05-14]. https://fedoraproject.org/wiki/Koji.

[2] *Add lzip support* [online], 2011 [cit. 2016-05-03]. http://rpm.org/ticket/839.

[3] *Header API* [online], 2014 [cit. 2016-05-03]. http://rpm.org/api/4.12.0.1/group__header.html.

[4] Mark Adler and Jean-loup Gailly. *zlib 1.2.8 Manual* [online], 2013. http://www.zlib.net/manual.html.

[5] Edward C. Bailey. *Maximum RPM*, chapter RPM File Format. Red Hat Inc., 2000. ISBN 1-888172-78-9.

[6] Edward C. Bailey. *Maximum RPM*. Red Hat Inc., 2000. ISBN 1-888172-78-9.

[7] Edward C. Bailey. *Maximum RPM*, chapter Directives for the %files list. Red Hat Inc., 2000. ISBN 1-888172-78-9.

[8] Lasse Collin. *XZ Utils* [online], 2016 [cit. 2016-05-03]. http://tukaani.org/xz/.

[9] Antonio Diaz Diaz. *Lzlib Manual* [online], 2009 [cit. 2016-05-14]. http://www.nongnu.org/lzip/manual/lzlib_manual.html.

[10] Jonathan Dieter. On binary delta algorithms. In: *The Cedar and the Thistle* [online], 2009-11-06 [cit. 2016-04-24]. https://cedarandthistle.wordpress.com/2009/11/06/on-binary-delta-algorithms/.

[11] Jonathan Dieter. Deltarpm problems (Part I). In: *The Cedar and the Thistle* [online], 2009-11-16 [cit. 2016-05-15]. https://cedarandthistle.wordpress.com/2009/11/16/deltarpm-problems-part-i/.

[12] Ulrich Drepper. *GNU C Library Version 2.3*. Red Hat, Inc., 2002.

[13] Ulrich Drepper, Scott Miller, and David Madore. *sha256sum(1) – Linux man page* [online], 2010 [cit. 2016-05-14]. http://linux.die.net/man/1/sha256sum.

[14] FreeBSD File Formats Manual. *cpio – format of cpio archive files* [online]. https://people.freebsd.org/~kientzle/libarchive/man/cpio.5.txt.

[15] Jeff Johnson. *RPM Project Roadmap* [online], 2007. http://rpm5.org/roadmap.php.

[16] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*. 2007, **387**(3), 258–272.

[17] Dusty Mabe. RPM File Colors. In: *A Random Walk Down Tech Street* [online], 2013-08-25 [cit. 2016-04-24]. http://dustymabe.com/2013/08/25/rpm-file-colors/.

[18] Tomáš Mlčoch. *createrepo_c* [online], 2013 [cit. 2016-04-24]. https://fedorahosted.org/createrepo_c/.

[19] OpenSSL Software Foundation. crypto library. In: *OpenSSL: Cryptography and SSL/TLS Toolkit* [online], 2015 [cit. 2016-05-14]. https://www.openssl.org/docs/manmaster/crypto/.

[20] Colin Percival. *Naïve Differences of Executable Code*. 2003.

[21] Colin Percival. *Matching with Mismatches and Assorted Applications*. PhD thesis, Oxford University, 2006.

[22] Andreas Schneider. *cmocka* [online], 2013 [cit. 2016-05-14]. https://cmocka.org/.

[23] Michael Schroeder. *applydeltarpm(8) - Linux man page* [online], 2005 [cit. 2016-05-14]. http://linux.die.net/man/8/applydeltarpm.

[24] Michael Schroeder. *makedeltarpm(8) - Linux man page* [online], 2010 [cit. 2016-05-14]. http://linux.die.net/man/8/makedeltarpm.

[25] Julian Seward. *bzip2 and libbzip2* [online], 2007. http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html.

[26] Robert C. Uzgalis. *General Hash Functions*. Technical report, The University of Hong Kong, 1993.

# Appendices

# List of Appendices

# Appendix A

# API Documentation (Generated by Doxygen)

# drpm

Generated by Doxygen 1.8.6

Thu May 12 2016 15:31:25

# Contents



# 1 Module Index

## 1.1 Modules

Here is a list of all modules:

# 2 File Index

## 2.1 File List

Here is a list of all documented files with brief descriptions:



# 3 Module Documentation

## 3.1 DRPM Make

Tools for creating a DeltaRPM file from two RPM files, providing the same functionality as `makedeltarpm(8)`.

### Modules

- DRPM Make Options

  *Tools for customizing DeltaRPM creation.*

### Functions

- int drpm_make (const char *oldrpm, const char *newrpm, const char *deltarpm, const drpm_make_options *opts)

  *Creates a DeltaRPM from two RPMs.*

### 3.1.1 Detailed Description

Tools for creating a DeltaRPM file from two RPM files, providing the same functionality as `makedeltarpm(8)`.

### 3.1.2 Function Documentation

#### 3.1.2.1 int drpm_make ( const char * *oldrpm*, const char * *newrpm*, const char * *deltarpm*, const drpm_make_options * *opts* )

Creates a DeltaRPM from two RPMs.

The DeltaRPM can later be used to recreate the new RPM from either filesystem data or the old RPM.

Does the same thing as the `makedeltarpm(8)` command-line utility.

Examples of function calls (without error handling):

```
// makedeltarpm foo.rpm goo.rpm fg.drpm
drpm_make("foo.rpm", "goo.rpm", "fg.drpm", NULL);


// makedeltarpm -r -z xz.6 -s seqfile.txt foo.rpm goo.rpm fg.drpm

drpm_make_options *opts;

drpm_make_options_init(&opts);
drpm_make_options_set_type(opts, DRPM_TYPE_RPMONLY);
drpm_make_options_set_seqfile(opts, "seqfile.txt");
drpm_make_options_set_delta_comp(opts,
      DRPM_COMP_XZ, 6);
```

```
drpm_make("foo.rpm", "goo.rpm", "fg.drpm", &opts);

drpm_make_options_destroy(&opts);


// makedeltarpm -V 2 -z gzip,off -p foo-print.rpml foo-patch.rpml foo.rpm goo.rpm fg.drpm

drpm_make_options *opts;

drpm_make_options_init(&opts);
drpm_make_options_set_version(opts, 2);
drpm_make_options_set_delta_comp(opts,
      DRPM_COMP_GZIP, DRPM_COMP_LEVEL_DEFAULT);
drpm_make_options_forbid_addblk(opts);
drpm_make_options_add_patches(opts, "foo-print.rpml", "foo-patch.rpml");

drpm_make("foo.rpm", "goo.rpm", "fg.drpm", &opts);

drpm_make_options_destroy(&opts);


// makedeltarpm -z uncompressed,bzip2.9 foo.rpm goo.rpm fg.drpm

drpm_make_options *opts;

drpm_make_options_init(&opts);
drpm_make_options_set_delta_comp(opts,
      DRPM_COMP_NONE, 0);
drpm_make_options_set_addblk_comp(opts,
      DRPM_COMP_BZIP2, 9);

drpm_make("foo.rpm", "goo.rpm", "fg.drpm", &opts);

drpm_make_options_destroy(&opts);


// makedeltarpm -u foo.rpm foo.drpm
drpm_make("foo.rpm", NULL, "foo.drpm", NULL);
```

#### Parameters

| | | |
|---|---|---|
| in | *oldrpm* | Name of old RPM file. |
| in | *newrpm* | Name of new RPM file. |
| in | *deltarpm* | Name of DeltaRPM file to be created. |
| in | *opts* | Options (if `NULL`, defaults used). |

#### Returns

Error code.

#### Note

If either `old_rpm` or `new_rpm` is `NULL`, an "identity" deltarpm is created (may be useful to just replace the signature of an RPM or to reconstruct an RPM from the filesystem).

#### Warning

If not `NULL`, `opts` should have been initialized with drpm_make_options_init(), otherwise behaviour is undefined.

## 3.2 DRPM Make Options

Tools for customizing DeltaRPM creation.

### Typedefs

- typedef struct drpm_make_options drpm_make_options

  *Options for drpm_make()*

### Functions

- int drpm_make_options_add_patches (drpm_make_options *opts, const char *oldrpmprint, const char *oldpatchrpm)

  *Requests incorporation of RPM patch files for the old RPM.*
- int drpm_make_options_copy (drpm_make_options *dst, const drpm_make_options *src)

  *Copies drpm_make_options.*
- int drpm_make_options_defaults (drpm_make_options *opts)

  *Resets options to default values.*
- int drpm_make_options_destroy (drpm_make_options **opts)

  *Frees drpm_make_options.*
- int drpm_make_options_forbid_addblk (drpm_make_options *opts)

  *Forbids add block creation.*
- int drpm_make_options_get_delta_comp_from_rpm (drpm_make_options *opts)

  *DeltaRPM compression method is the same as used in the new RPM.*
- int drpm_make_options_init (drpm_make_options **opts)

  *Initializes drpm_make_options with default options.*
- int drpm_make_options_set_addblk_comp (drpm_make_options *opts, unsigned short comp, unsigned short level)

  *Sets add block compression type and level.*
- int drpm_make_options_set_delta_comp (drpm_make_options *opts, unsigned short comp, unsigned short level)

  *Sets DeltaRPM compression type and level.*
- int drpm_make_options_set_seqfile (drpm_make_options *opts, const char *seqfile)

  *Specifies file to which to write DeltaRPM sequence ID.*
- int drpm_make_options_set_type (drpm_make_options *opts, unsigned short type)

  *Sets DeltaRPM type.*
- int drpm_make_options_set_version (drpm_make_options *opts, unsigned short version)

  *Sets DeltaRPM version.*

### 3.2.1 Detailed Description

Tools for customizing DeltaRPM creation.

### 3.2.2 Function Documentation

#### 3.2.2.1 int drpm_make_options_init ( drpm_make_options ** *opts* )

Initializes drpm_make_options with default options.

Passing `*opts` to drpm_make() immediately after would have the same effect as passing `NULL` instead.

**Parameters**

| | | |
|---|---|---|
| out | *opts* | Address of options structure pointer. |

**Returns**

Error code.

**See Also**

drpm_make()

#### 3.2.2.2 int drpm_make_options_destroy ( drpm_make_options ** *opts* )

Frees drpm_make_options.

**Parameters**

| | | |
|---|---|---|
| out | *opts* | Address of options structure pointer. |

**Returns**

Error code.

**See Also**

drpm_make()

#### 3.2.2.3 int drpm_make_options_defaults ( drpm_make_options * *opts* )

Resets options to default values.

Passing `opts` to drpm_make() immediately after would have the same effect as passing `NULL` instead.

**Parameters**

| | | |
|---|---|---|
| out | *opts* | Structure specifying options for drpm_make(). |

**Returns**

Error code.

**See Also**

drpm_make()

#### 3.2.2.4 int drpm_make_options_copy ( drpm_make_options * *dst*, const drpm_make_options * *src* )

Copies drpm_make_options.

Copies data from `src` to `dst`.

**Parameters**

| | | |
|---|---|---|
| out | *dst* | Destination options. |
| in | *src* | Source options. |

**Returns**

Error code.

**Warning**

`dst` should have also been initialized with drpm_make_options_init() previously, otherwise behaviour is undefined.

**See Also**

drpm_make()

#### 3.2.2.5 int drpm_make_options_set_type ( drpm_make_options * *opts*, unsigned short *type* )

Sets DeltaRPM type.

There are two types of DeltaRPMs: standard and "rpm-only". The latter was introduced in version 3. It does not work with filesystem data but is smaller and faster to combine.

**Parameters**

| | | |
|---|---|---|
| `out` | *opts* | Structure specifying options for drpm_make(). |
| `in` | *type* | Type of deltarpm. |

**Returns**

Error code.

**See Also**

drpm_make()
DRPM_TYPE_STANDARD, DRPM_TYPE_RPMONLY

#### 3.2.2.6 int drpm_make_options_set_version ( drpm_make_options * *opts*, unsigned short *version* )

Sets DeltaRPM version.

The default DeltaRPM format is V3, but an older version may also be specified.

**Parameters**

| | | |
|---|---|---|
| `out` | *opts* | Structure specifying options for drpm_make(). |
| `in` | *version* | Version (1-3). |

**Returns**

Error code.

**See Also**

drpm_make()

#### 3.2.2.7 int drpm_make_options_set_delta_comp ( drpm_make_options * *opts*, unsigned short *comp*, unsigned short *level* )

Sets DeltaRPM compression type and level.

By default, the compression method is the same as used in the new RPM.

**Parameters**

| | | |
|---|---|---|
| out | *opts* | Structure specifying options for drpm_make(). |
| in | *comp* | Compression type. |
| in | *level* | Compression level (1-9 or default). |

**Returns**

Error code.

**See Also**

drpm_make()
DRPM_COMP_NONE, DRPM_COMP_GZIP, DRPM_COMP_BZIP2, DRPM_COMP_LZMA, DRPM_COMP_XZ
DRPM_COMP_LEVEL_DEFAULT

#### 3.2.2.8 int drpm_make_options_get_delta_comp_from_rpm ( drpm_make_options * *opts* )

DeltaRPM compression method is the same as used in the new RPM.

May be used to reset DeltaRPM compression option after previously calling drpm_make_options_delta_comp().

**Parameters**

| | | |
|---|---|---|
| out | *opts* | Structure specifying options for drpm_make(). |

**Returns**

Error code.

**See Also**

drpm_make()

#### 3.2.2.9 int drpm_make_options_forbid_addblk ( drpm_make_options * *opts* )

Forbids add block creation.

An "add block" is a highly compressible block used to store bytewise subtractions of segments where less than half the bytes have changed. It is used in re-creating the new RPM with drpm_apply(), unless this functions is called to tell drpm_make() not to create an add block.

**Parameters**

| | | |
|---|---|---|
| out | *opts* | Structure specifying options for drpm_make(). |

**Returns**

Error code.

**See Also**

drpm_make()

#### 3.2.2.10 int drpm_make_options_set_addblk_comp ( drpm_make_options * *opts*, unsigned short *comp*, unsigned short *level* )

Sets add block compression type and level.

The default add block compression type is bzip2, which gives the best results.

**Parameters**

| | | |
|---|---|---|
| out | *opts* | Structure specifying options for drpm_make(). |
| in | *comp* | Compression type. |
| in | *level* | Compression level (1-9 or default). |

**Returns**

Error code.

**See Also**

drpm_make()
DRPM_COMP_NONE, DRPM_COMP_GZIP, DRPM_COMP_BZIP2, DRPM_COMP_LZMA, DRPM_COMP_XZ
DRPM_COMP_LEVEL_DEFAULT

#### 3.2.2.11 int drpm_make_options_set_seqfile ( drpm_make_options * *opts*, const char * *seqfile* )

Specifies file to which to write DeltaRPM sequence ID.

If a valid file name is given, drpm_make() will write out the sequence ID to the file `seqfile`.

**Parameters**

| | | |
|---|---|---|
| out | *opts* | Structure specifying options for drpm_make(). |
| in | *seqfile* | Name of file to which to write out sequence. |

**Returns**

Error code.

**Note**

If `seqfile` is `NULL`, sequence ID shall not be written.

**See Also**

drpm_make()

#### 3.2.2.12 int drpm_make_options_add_patches ( drpm_make_options * *opts*, const char * *oldrpmprint*, const char * *oldpatchrpm* )

Requests incorporation of RPM patch files for the old RPM.

This option enables the usage of patch RPMs, telling drpm_make() to exclude all files that were not included in the patch RPM but are not bytewise identical to the ones in the old RPM.

**Parameters**

| | | |
|---|---|---|
| out | *opts* | Structure specifying options for drpm_make(). |
| in | *oldrpmprint* | The rpm-print of the old RPM. |
| in | *oldpatchrpm* | The created patch RPM. |

**Returns**

Error code.

**See Also**

drpm_make()

## 3.3 DRPM Apply

Tools for applying a DeltaRPM file to re-create a new RPM file (from an old RPM file or from filesystem data), providing the same functionality as `applydeltarpm(8)`.

### Modules

- DRPM Check

  *Tools for checking if the reconstruction is possible (like `applydeltarpm { -c | -C }`).*

### Functions

- int drpm_apply (const char *oldrpm, const char *deltarpm, const char *newrpm)

  *Applies a DeltaRPM to an old RPM or on-disk data to re-create a new RPM.*

### 3.3.1 Detailed Description

Tools for applying a DeltaRPM file to re-create a new RPM file (from an old RPM file or from filesystem data), providing the same functionality as `applydeltarpm(8)`.

### 3.3.2 Function Documentation

#### 3.3.2.1 int drpm_apply ( const char * *oldrpm*, const char * *deltarpm*, const char * *newrpm* )

Applies a DeltaRPM to an old RPM or on-disk data to re-create a new RPM.

**Parameters**

| | | |
|---|---|---|
| in | *oldrpm* | Name of old RPM file (if `NULL`, filesystem data is used). |
| in | *deltarpm* | Name of DeltaRPM file. |
| in | *newrpm* | Name of new RPM file to be (re-)created. |

**Returns**

Error code.

## 3.4 DRPM Check

Tools for checking if the reconstruction is possible (like `applydeltarpm { -c | -C }`).

### Functions

- int drpm_check (const char *deltarpm, int checkmode)

  *Checks if the reconstruction is possible based on DeltaRPM file.*
- int drpm_check_sequence (const char *oldrpm, const char *sequence, int checkmode)

  *Checks if the reconstruction is possible based on sequence ID.*

### 3.4.1 Detailed Description

Tools for checking if the reconstruction is possible (like `applydeltarpm { -c | -C }`).

### 3.4.2 Function Documentation

#### 3.4.2.1 int drpm_check ( const char * *deltarpm*, int *checkmode* )

Checks if the reconstruction is possible based on DeltaRPM file.

**Parameters**

| | | |
|---|---|---|
| in | *deltarpm* | Name of DeltaRPM file. |
| in | *checkmode* | Full check or filesize changes only. |

**Returns**

Error code.

**See Also**

DRPM_CHECK_FULL, DRPM_CHECK_FILESIZES

#### 3.4.2.2 int drpm_check_sequence ( const char * *oldrpm*, const char * *sequence*, int *checkmode* )

Checks if the reconstruction is possible based on sequence ID.

**Parameters**

| | | |
|---|---|---|
| in | *oldrpm* | Name of old RPM file (if `NULL`, filesystem data is used). |
| in | *sequence* | Sequence ID of the DeltaRPM. |
| in | *checkmode* | Full check or filesize changes only. |

**Returns**

Error code.

**See Also**

DRPM_CHECK_FULL, DRPM_CHECK_FILESIZES

## 3.5 DRPM Read

Tools for extracting information from DeltaRPM files.

### Typedefs

- typedef struct drpm drpm

  *DeltaRPM package info.*

### Functions

- int drpm_destroy (drpm **delta)

  *Frees memory allocated by drpm_read().*
- int drpm_get_string (drpm *delta, int tag, char **target)

  *Fetches information representable as a string.*
- int drpm_get_uint (drpm *delta, int tag, unsigned *target)

  *Fetches information representable as an unsigned integer.*
- int drpm_get_ullong (drpm *delta, int tag, unsigned long long *target)

  *Fetches information representable as an unsigned long long integer.*
- int drpm_get_ulong (drpm *delta, int tag, unsigned long *target)

  *Fetches information representable as an unsigned long integer.*
- int drpm_get_ulong_array (drpm *delta, int tag, unsigned long **target, unsigned long *size)

  *Fetches information representable as an array of unsigned long integers.*
- int drpm_read (drpm **delta, const char *filename)

  *Reads information from a DeltaRPM.*

### 3.5.1 Detailed Description

Tools for extracting information from DeltaRPM files. Limits memory usage.

As drpm_make() normally needs about three to four times the size of the rpm's uncompressed payload, this option may be used to enable a sliding block algorithm that needs `mbytes` megabytes of memory. This trades memory usage with the size of the created DeltaRPM.

**Parameters**

| | | |
|---|---|---|
| `out` | *opts* | Structure specifying options for drpm_make(). |
| `in` | *mbytes* | Permitted memory usage in megabytes. |

**Returns**

Error code.

**See Also**

drpm_make()

### 3.5.2 Function Documentation

#### 3.5.2.1 int drpm_read ( drpm ** *delta*, const char * *filename* )

Reads information from a DeltaRPM.

Reads information from DeltaRPM package `filename` into `*delta`. Example of usage:

```
drpm *delta = NULL;

int error = drpm_read(&delta, "foo.drpm");

if (error != DRPM_ERR_OK) {
   fprintf(stderr, "drpm error: %s\n", drpm_strerror(error));
   return;
}
```

**Parameters**

| | | |
|---|---|---|
| out | *delta* | DeltaRPM to be filled with info. |
| in | *filename* | Name of DeltaRPM file whose data is to be read. |

**Returns**

Error code.

**Note**

Memory allocated by calling drpm_read() should later be freed by calling drpm_destroy().

#### 3.5.2.2 int drpm_get_uint ( drpm * *delta*, int *tag*, unsigned * *target* )

Fetches information representable as an unsigned integer.

Fetches information identified by `tag` from `delta` and copies it to address pointed to by `target`.

Example of usage:

```
unsigned type;

int error = drpm_get_uint(delta, DRPM_TAG_TYPE, &type);

if (error != DRPM_ERR_OK) {
   fprintf(stderr, "drpm error: %s\n", drpm_strerror(error));
   return;
}

printf("This is a %s deltarpm\n", (type == DRPM_TYPE_STANDARD) ? "standard" : "rpm-only")
      ;
```

**Parameters**

| | | |
|---|---|---|
| in | *delta* | DeltaRPM containing required info. |
| in | *tag* | Identifies which info is required. |
| out | *target* | Tagged info will be copied here. |

**Returns**

error number

**Warning**

`delta` should have been previously initialized with drpm_read(), otherwise behaviour is undefined.

**See Also**

DRPM_TAG_VERSION
DRPM_TAG_TYPE
DRPM_TAG_COMP
DRPM_TAG_TGTCOMP

#### 3.5.2.3 int drpm_get_ulong ( drpm * *delta*, int *tag*, unsigned long * *target* )

Fetches information representable as an unsigned long integer.

Fetches information identified by `tag` from `delta` and copies it to address pointed to by `target`.

Example of usage:

```
unsigned long tgt_size;

int error = drpm_get_ulong(delta, DRPM_TAG_TGTSIZE, &tgt_size);

if (error != DRPM_ERR_OK) {
   fprintf(stderr, "drpm error: %s\n", drpm_strerror(error));
   return;
}

printf("Size of new RPM: %lu\n", tgt_size);
```

**Parameters**

| | | |
|---|---|---|
| in | *delta* | Deltarpm containing required info. |
| in | *tag* | Identifies which info is required. |
| out | *target* | Tagged info will be copied here. |

**Returns**

Error code.

**Warning**

`delta` should have been previously initialized with drpm_read(), otherwise behaviour is undefined.

**See Also**

DRPM_TAG_TGTSIZE
DRPM_TAG_TGTHEADERLEN
DRPM_TAG_PAYLOADFMTOFF

#### 3.5.2.4 int drpm_get_ullong ( drpm * *delta*, int *tag*, unsigned long long * *target* )

Fetches information representable as an unsigned long long integer.

Fetches information identified by `tag` from `delta` and copies it to address pointed to by `target`.

Example of usage:

```
unsigned long long int_data_len;

int error = drpm_get_ullong(delta, DRPM_TAG_INTDATALEN, &int_data_len);

if (error != DRPM_ERR_OK) {
   fprintf(stderr, "drpm error: %s\n", drpm_strerror(error));
   return;
}

printf("Length of internal data: %llu\n", int_data_len);
```

**Parameters**

| | | |
|---|---|---|
| in | *delta* | Deltarpm containing required info. |

| | | |
|---|---|---|
| in | *tag* | Identifies which info is required. |
| out | *target* | Tagged info will be copied here. |

**Returns**

Error code.

**Warning**

`delta` should have been previously initialized with drpm_read(), otherwise behaviour is undefined.

**See Also**

DRPM_TAG_EXTDATALEN
DRPM_TAG_INTDATALEN

#### 3.5.2.5 int drpm_get_string ( drpm * *delta*, int *tag*, char ** *target* )

Fetches information representable as a string.

Fetches string-type information identified by `tag` from `delta`, copies it to space previously allocated by the function itself and saves the address to `*target`.

Example of usage:

```
char *tgt_nevr;

int error = drpm_get_string(delta, DRPM_TAG_TGTNEVR, &tgt_nevr);

if (error != DRPM_ERR_OK) {
   fprintf(stderr, "drpm error: %s\n", drpm_strerror(error));
   return;
}

printf("Target NEVR: %s\n", tgt_nevr);

free(tgt_nevr);
```

**Parameters**

| | | |
|---|---|---|
| in | *delta* | Deltarpm containing required info. |
| in | *tag* | Identifies which info is required. |
| out | *target* | Tagged info will be copied here. |

**Returns**

Error code.

**Note**

`*target` should be freed manually by the user when no longer needed.

**Warning**

`delta` should have been previously initialized with drpm_read(), otherwise behaviour is undefined.

**See Also**

DRPM_TAG_FILENAME
DRPM_TAG_SEQUENCE
DRPM_TAG_SRCNEVR
DRPM_TAG_TGTNEVR
DRPM_TAG_TGTMD5
DRPM_TAG_TGTCOMPPARAM
DRPM_TAG_TGTLEAD

#### 3.5.2.6 int drpm_get_ulong_array ( drpm * *delta*, int *tag*, unsigned long ** *target*, unsigned long * *size* )

Fetches information representable as an array of unsigned long integers.

Fetches information identified by `tag` from `delta`, copies it to space previously allocated by the function itself, saves the address to `*target` and stores size in `*size`.

Example of usage:

```
unsigned long *ext_copies;
unsigned long ext_copies_size;

int error = drpm_get_ulong_array(delta, DRPM_TAG_EXTCOPIES, &
      ext_copies, &ext_copies_size);

if (error != DRPM_ERR_OK) {
   fprintf(stderr, "drpm error: %s\n", drpm_strerror(error));
   return;
}

for (unsigned long i = 1; i < ext_copies_size; i += 2)
   printf("External copy: offset adjustment = %lu, length = %lu\n", ext_copies[i-1], ext_copies[i]);

free(ext_copies);
```

**Parameters**

| | | |
|---|---|---|
| in | *delta* | Deltarpm containing required info. |
| in | *tag* | Identifies which info is required. |
| out | *target* | Tagged info will be copied here. |
| out | *size* | Size of array will be copied here. |

**Returns**

Error code.

**Note**

`*target` should be freed manually by the user when no longer needed.

**Warning**

`delta` should have been previously initialized with drpm_read(), otherwise behaviour is undefined.

**See Also**

DRPM_TAG_ADJELEMS
DRPM_TAG_INTCOPIES
DRPM_TAG_EXTCOPIES

#### 3.5.2.7 int drpm_destroy ( drpm ** *delta* )

Frees memory allocated by drpm_read().

Frees memory pointed to by `*delta` and sets `*delta` to `NULL`.

Example of usage:

```
int error = drpm_destroy(&delta);

if (error != DRPM_ERR_OK) {
   fprintf(stderr, "drpm error: %s\n", drpm_strerror(error));
   return;
}
```

**Parameters**

| | | |
|---|---|---|
| `out` | *delta* | Deltarpm that is to be freed. |

**Returns**

Error code.

**Warning**

`delta` should have been previously initialized with drpm_read(), otherwise behaviour is undefined.

# 4 File Documentation

## 4.1 drpm.h File Reference

### Macros

#### Errors / Return values

- #define DRPM_ERR_OK 0
  *no error*
- #define DRPM_ERR_MEMORY 1
  *memory allocation error*
- #define DRPM_ERR_ARGS 2
  *bad arguments*
- #define DRPM_ERR_IO 3
  *I/O error.*
- #define DRPM_ERR_FORMAT 4
  *wrong file format*
- #define DRPM_ERR_CONFIG 5
  *misconfigured external library*
- #define DRPM_ERR_OTHER 6
  *unspecified/unknown error*
- #define DRPM_ERR_OVERFLOW 7
  *file too large*
- #define DRPM_ERR_PROG 8
  *internal programming error*
- #define DRPM_ERR_MISMATCH 9
  *file changed*
- #define DRPM_ERR_NOINSTALL 10
  *old RPM not installed*

#### Delta Types

- #define DRPM_TYPE_STANDARD 0
  *standard deltarpm*
- #define DRPM_TYPE_RPMONLY 1
  *rpm-only deltarpm*

#### Compression Types

- #define DRPM_COMP_NONE 0
  *no compression*
- #define DRPM_COMP_GZIP 1
  *gzip*
- #define DRPM_COMP_BZIP2 2
  *bzip2*
- #define DRPM_COMP_LZMA 3
  *lzma*
- #define DRPM_COMP_XZ 4
  *xz*
- #define DRPM_COMP_LZIP 5
  *lzip*

#### Info Tags

- #define DRPM_TAG_FILENAME 0

### Functions

- int drpm_apply (const char *oldrpm, const char *deltarpm, const char *newrpm)
  *Applies a DeltaRPM to an old RPM or on-disk data to re-create a new RPM.*
- int drpm_check (const char *deltarpm, int checkmode)
  *Checks if the reconstruction is possible based on DeltaRPM file.*
- int drpm_check_sequence (const char *oldrpm, const char *sequence, int checkmode)
  *Checks if the reconstruction is possible based on sequence ID.*
- int drpm_destroy (drpm **delta)
  *Frees memory allocated by drpm_read().*
- int drpm_get_string (drpm *delta, int tag, char **target)
  *Fetches information representable as a string.*
- int drpm_get_uint (drpm *delta, int tag, unsigned *target)
  *Fetches information representable as an unsigned integer.*
- int drpm_get_ullong (drpm *delta, int tag, unsigned long long *target)
  *Fetches information representable as an unsigned long long integer.*
- int drpm_get_ulong (drpm *delta, int tag, unsigned long *target)
  *Fetches information representable as an unsigned long integer.*
- int drpm_get_ulong_array (drpm *delta, int tag, unsigned long **target, unsigned long *size)
  *Fetches information representable as an array of unsigned long integers.*
- int drpm_make (const char *oldrpm, const char *newrpm, const char *deltarpm, const drpm_make_options *opts)
  *Creates a DeltaRPM from two RPMs.*
- int drpm_make_options_add_patches (drpm_make_options *opts, const char *oldrpmprint, const char *oldpatchrpm)
  *Requests incorporation of RPM patch files for the old RPM.*
- int drpm_make_options_copy (drpm_make_options *dst, const drpm_make_options *src)
  *Copies drpm_make_options.*
- int drpm_make_options_defaults (drpm_make_options *opts)
  *Resets options to default values.*
- int drpm_make_options_destroy (drpm_make_options **opts)
  *Frees drpm_make_options.*
- int drpm_make_options_forbid_addblk (drpm_make_options *opts)
  *Forbids add block creation.*
- int drpm_make_options_get_delta_comp_from_rpm (drpm_make_options *opts)
  *DeltaRPM compression method is the same as used in the new RPM.*
- int drpm_make_options_init (drpm_make_options **opts)
  *Initializes drpm_make_options with default options.*
- int drpm_make_options_set_addblk_comp (drpm_make_options *opts, unsigned short comp, unsigned short level)
  *Sets add block compression type and level.*
- int drpm_make_options_set_delta_comp (drpm_make_options *opts, unsigned short comp, unsigned short level)
  *Sets DeltaRPM compression type and level.*
- int drpm_make_options_set_seqfile (drpm_make_options *opts, const char *seqfile)
  *Specifies file to which to write DeltaRPM sequence ID.*
- int drpm_make_options_set_type (drpm_make_options *opts, unsigned short type)
  *Sets DeltaRPM type.*
- int drpm_make_options_set_version (drpm_make_options *opts, unsigned short version)
  *Sets DeltaRPM version.*
- int drpm_read (drpm **delta, const char *filename)
  *Reads information from a DeltaRPM.*
- const char * drpm_strerror (int error)
  *Returns description of error code as a string.*

#### 4.1.1 Detailed Description

**Author**

Pavel Tobias ptobias@redhat.com
Matej Chalk mchalk@redhat.com

**Date**

2014-2016

**Copyright**



#### 4.1.2 Function Documentation

##### 4.1.2.1 const char* drpm_strerror ( int *error* )

Returns description of error code as a string.

Works very similarly to strerror(3).

**Parameters**

| | | |
|---|---|---|
| in | *error* | error code |

**Returns**

error description

## Index