USBImager

User's Manual

version 1.0.9

Copyright

MIT licence

Table of Contents

Page left blank intentionally

Preface

“Make each program do one thing well.”

/ Ken Thompson /

I felt a niche in a simple to use multi-platform application that can write a compressed disk dump image to an USB device. There are existing solutions, but they are either single platform (mostly Windows only), or incredibly bloated, and some have been found to spying on its users. Others work perfectly, but invoked from command line, which makes them unsuitable for average users.

So I’ve decided to create the simplest GUI application possible that does write images to devices. Because many OS images are distributed in compressed format, it makes sense for such an application to be able to decompress on-the-fly to save storage space and user’s time. Although it wasn’t originally planned, but due to pressure from users I’ve added backup capability as well.

What this application wasn’t designed to do, and never will do, is downloading images from the internet. First, USBImager is capable of writing *any* image to disks, and it would be impossible to list all available options. Second, those options are changing all the time, new versions appear, and some become discontinued. There’s no way to keep such a list always up-to-date. And finally, I wanted the application to work without any internet connection, to eliminate even the possibility of telemetry.

For the user interface, I’ve decided to use the native interface on all platforms. This made the development a bit harder, but has many benefits from the user’s point of view. This guarantees that the application can be distributed as a single portable executable, as it has no library dependencies. It also guarantees that the application is small in size (currently less than 256 kilobytes on each platform).

The final result of this development (including the source and precompiled binaries for several platforms) can be obtained at:

`https://gitlab.com/bztsrc/usbimager`

This application is Open Source and Free Software, and comes without any warranty in the hope that it will be useful.

Baldaszti Zoltán Tamás

Page left blank intentionally

Installation

USBImager is distributed in three flavours:

1. zip archive
2. deb package
3. source

The zip archive is the most universal, available for all platforms (Windows, MacOS, Linux). The deb package can be used on apt based Linux distributions (primarily RaspiOS and Ubuntu LTS, but should work on other distributions). Source is recommended for advanced users and on POSIX systems without binary USBImager distribution (BSDs, Minix etc.), see Appendix.

Zip Archive

Download one of the zip archives for your desktop from the repository. Extract it to:

C:\Program Files (Windows)

/Applications (MacOS)

/usr (Linux)

On **Windows**, right-click on usbimager.exe and create a shortcut. On the shortcut's "Security" tab, you can set to run as Administrator. Then you can move the .lnk shortcut where ever you like (to your Desktop, into the Menu folder etc.) You can also set the command line options here, see below.

On **MacOS**: go to "System Preferences", "Security & Privacy" and "Privacy". Add USBImager to the list of "Full Disk Access". Or use "sudo /Applications/USBImager.app/Contents/MacOS/usbimager".

That's all. You can use USBImager without any further ado. Under MacOS and Linux it should appear in the Applications menu right away.

Deb Package

If you've downloaded the .deb version, then you can install it with the following command:

```
sudo dpkg -i usbimager_*.deb
```

The deb versions depend on `GTK-3+` and `udisks2`.

Command Line Options

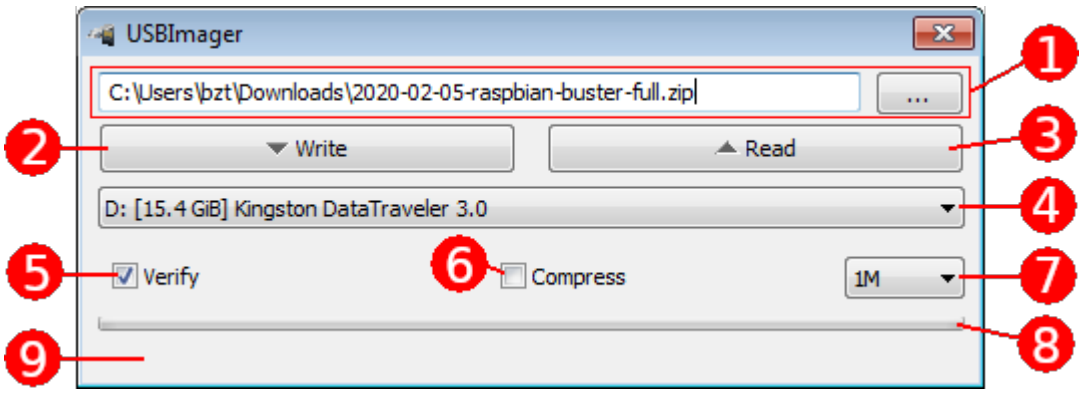
USBImager is a GUI application, but more advanced users can specify options on the command line to alter its behaviour. Average users are free to skip this section.

For **Windows** users: right-click on usbimager.exe, and select "Create Shortcut". Then right-click on the newly created ".lnk" file, and select "Properties". In the "Target" field, you can add the flags.

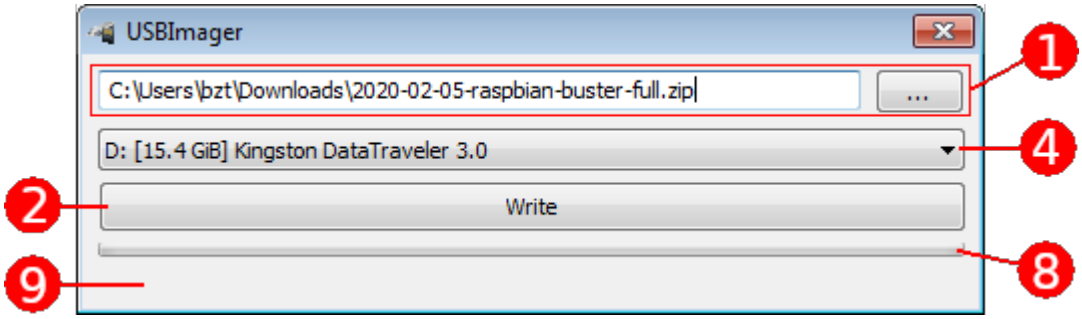
-v / -vv	Print verbose diagnostics and statistics to *stdout* (under Windows, a new console window will be opened).
-L*xx*	Using this flag forces a specific language dictionary and avoids language autodetection.
-1 .. **-9**	Set the buffer size to power of two Megabytes (1 = 2M, 2 = 4M, 3 = 8M, 4 = 16M, … 9 = 512M). When not specified, defaults to 1M. Same as using 7 on the interface.
-a	List all devices, even system disks and large disks. Be careful. Under Windows this will also list the physical disks instead of device labels.
-f	Force write, do not compare disk with buffer first.
-s*[baud]*	This option will allow you to send images to serial ports as well. With this, the client must wait indefinitely for the first byte to arrive, then read further bytes with a timeout.
-S*[baud]*	Uppercase ‘S’ too allows serial ports, but will use a raspbootin hand-shake. See section *Sending Image to MicroController* for more details.
--version	Prints version information.
(backupdir)	First argument which is not a flag must be an existing directory (on any drive). Backups will be saved in this directory instead of the Desktop. Useful if under Windows C: does not have enough space for the image, but D: for example does. For more details, see section *Creating Backup Image File from Device*.

User Interface

The interface is very simple, has one window only.



Binaries with the "**wo**" in their names, and when compiled with USE_WRONLY, the interface is even simpler and write-only:



Binaries with "**uf**" in their name (X11 versions only) look the same, but they have GNU Unifont embedded in them, so they don't rely on X11 fontconfig.

1. Image file

The image file selection allows you to specify an image on your file system.

2. Write out Button

Pressing this button will start the write operation, reading the image file and writing to device.

3. Read in Button

Pressing this button will start a backup operation, reading the device and writing to an image file.

4. Device selection

This selection allows you to specify the device. Only removable devices are listed to prevent accidental overwriting of the system disk.

5. Verify Checkbox

If checked, then the write operation will read back each block and compare to the original image in memory. This verifies if the write was indeed successful.

6. Compress Checkbox

If checked, then the backup operation will also compress the image file before it is saved on your Desktop.

7. Buffer Size Selection

The image and the device will be processed in this big chunks. Keep in mind that the actual memory requirement is threefold, because there's one buffer for the compressed data, one for the uncompressed data, and one for the data read back for verification.

8. Progress Bar

This bar shows the actual progress of the operation.

9. Status Bar

The status bar displays the progress in a textual form. It shows how many bytes has been processed as well as the estimated remaining time.

Writing Image File to Device

Select the image file to be written by clicking on 1. The file can be a raw disk dump image (**.img**, **.bin**, **.raw**, **.iso**, **.dd**, etc.), compressed image (**.gz**, **.bz2**, **.xz, .zst**) or archive (**.zip**, both PKZip and Zip64 supported, **.zzz** (ZZZip), **.tar** or **.cpio** (plus their compressed variants)). For archives with multiple files, the first file in the archive is used as input.

Select the target device to be written to by clicking on 4.

If you don't trust the device, you can enable write verification by checking 5.

If you know the optimal block size for the device, you can select it by clicking on 7. If you're unsure, leave it at "1M".

Click on 2 to start.

As soon as the progress bar finishes, the image is physically written, you can detach the device.

Creating Backup Image File from Device

Select the source device to be backed up by clicking on 4.

If you want to save storage space, you can enable compression by checking 6. This will compress the image using zstandard algorithm.

If you know the optimal block size for the device, you can select it by clicking on 7. If you're unsure, leave it at "1M".

Click on 3 to start.

The image will be saved on your Desktop, and you'll see the image file's name in 1. The name is in the form "usbimager-(date)T(time).dd". If "Compress" option is checked, then a ".zst" suffix will be added.

Note: on Linux, if ~/Desktop is not found, then ~/Downloads will be used. If even that doesn't exists, then the image file will be saved in your home directory. On other platforms the Desktop always exists, but if by any chance not, then the current directory is used. On all platforms, if an existing directory is given on the command line, that is used to save backups. See section *Command Line Options*.

Sending Image to MicroController

For this, you'll have to start USBImager with the "-S" command line flag. Under Linux your user has to be the member of the "uucp" or "dialout" groups (differs in distributions). The serial line is set to 115200 baud, 8 data bits, no parity, 1 stop bit. For a simple boot loader that's compatible with USBImager, take a look at Image Receiver (available for RPi1, 2, 3, 4 and IBM PC BIOS machines).

This mode is also used to send emergency initrd images to BOOTBOOT compatible boot loaders.

If the 115200 does not suit your needs, just add a different baud after the flag, like "-S230400".

Connect your PC with the microcontroller / other machine using a serial cable.

Select the kernel image file to be sent by clicking on 1. The file should be in raw executable format, but depending on the receiver software, could be ELF as well. Archives are not supported, and compressed images will be sent as-is to minimize transfer time. To support compressed images, decompression has to be implemented on the receiver side.

Select the serial port where the cable is connected to by clicking on 4.

Click on 2 to start. The status bar (9) will show "Waiting for client".

Turn on the microcontroller / other machine running a simple image receiver boot loader. As soon as the handshake is done, USBImager will start sending the image automatically.

Appendix

Compilation from Source

Windows

Dependencies: just standard Win32 DLLs, and MinGW for compilation.

1. install MinGW, this will give you "gcc" and "make" under Windows

2. open MSYS terminal, and in the src directory, run `make`

3. to create the archive, run `make package`

MacOS

Dependencies: just standard frameworks (CoreFoundation, IOKit, DiskArbitration and Cocoa), and command line tools (no need for XCode, just the CLI tools).

1. in a Terminal, run `xcode-select --install` and in the pop-up window click "Install". This will give you "gcc" and "make" under MacOS.

2. in the src directory, run `make`

3. to create the archive, run `make package`

By default USBImager is compiled for native Cocoa with libui (included). You can also compile for X11 (if you have XQuartz installed) by using `USE_X11=yes make`. Running make will complain about setgid, don't mind.

Linux

Dependencies: libc, libX11 and standard GNU toolchain.

1. in the src directory, run `make`

2. to create the archive, run `make package`

3. to create a Debian package, run `make deb`

4. to install, run `sudo make install`

You can also compile for GTK+ by using `USE_GTK=yes make` or `USE_LIBUI=yes make`. That'll use libui (included), which in turn relies on hell a lot of libraries (pthread, X11, wayland, gdk, harfbuzz, pango, cairo, freetype2 etc.) Also note that the GTK version cannot be installed with setgid bit, so that write access to disk devices cannot be guaranteed. The X11 version gains "disk" group membership on

execution automatically. For GTK you'll have to add your user to that group manually or run USBImager via sudo, or compile with udisks2 support, otherwise you'll get "permission denied" errors.

Ubuntu

As for Ubuntu, you must compile in udisks2 support. You'll need the following packages:

```
sudo apt-get install build-essential libgtk-3-dev libudisks2-dev \
  libglib2.0-dev
```

and compile with

```
USE_LIBUI=yes USE_UDISKS2=yes make all deb
```

Compilation Options

Options can be set in environment variables when running `make`.

DEBUG

Adds debugging information such as extra symbols and source file references to the executable. These can be read by both valgrind and gdb.

USE_WRONLY

Use a simplified interface which has only a "Write" button.

USE_X11

Under platforms where libui is the default, selects X11 frontend. Currently MacOS.

USE_UNIFONT

Because missing fonts is a constant problem with X11, the USE_X11 has this option to embed Unifont.

USE_LIBUI

Under platforms where libui is not the default, selects libui. Currently Linux. Windows should support this too, but I was unable to statically link with libui under Windows. Libui under Linux in turn depends on GTK-3+.

USE_GTK

Compile with native GTK-3+ support (without the libui wrapper).

USE_UDISKS2

Make USBImager to fallback to libudisks2 if it cannot umount or open the device. Be warned, using udisks2 is a dependency hell, and you'll need many libraries and many daemons running in order to work. Don't even try if you're not using a Gnome-based desktop.

DESTDIR and PREFIX

The `make install` recipe accepts install directory in DESTDIR / PREFIX environment variables. If not defined, DESTDIR is empty and PREFIX defaults to "/usr". For example:

```
sudo DESTDIR=/usr/src/package PREFIX=/usr/local make install
```

Hacking the Source

To compile with debugging, use `DEBUG=yes make`. This will add extra debugging symbols and source file references to the executable, parsed by both valgrind and gdb.

Editing Makefile and changing `DISKS_TEST` to 1 will add a special `test.bin` "device" to the list on all platforms. You can test the decompressors with this.

X11 uses only low-level X11 (no Xft, Xmu nor any other extensions), so it should be trivial to port to other POSIX systems (like BSD or Minix). It does not handle locales, but it does use UTF-8 encoding in file names (this only matters for displaying, the file operations can handle any encoding). If you don't want this, set the `USEUTF8` define to 0 in the beginning of the main_x11.c file.

The source is clearly separated into 4 layers:

- stream.c / stream.h is responsible for reading in and uncompressing the data from file as well as compressing and writing out
- disks_*.c / disks.h is the layer that reads and writes out data to disks, separated for each platform
- main_*.c / main.h is where you can find main() (or WinMain), the user interface stuff
- lang.c / lang.h provides the internationalization and language dictionaries for all platform

Reporting Bugs

Please use the issue tracker on GitLab https://gitlab.com/bztsrc/usbimager/issues