86Box

86Box Project

Jun 03, 2026

CONTENTS

1 Using 86Box **2**

1.1 Getting started . . . 2

1.2 Frequently asked questions . . . 4

1.2.1 What is the difference between 86Box and applications like VirtualBox or Virtual PC? . . . 4

1.2.2 What is the difference between 86Box and QEMU? . . . 4

1.2.3 What is the difference between 86Box and PCem? . . . 4

1.2.4 Why is 86Box unable to run Xbox-level hardware when emulators for PS3 and Switch exist? 5

1.2.5 My virtual machine does not run at 100% speed, what can I do to fix this? . . . 5

1.2.6 What is the top VM configuration my system will handle? . . . 5

1.2.7 What are some era-appropriate configurations for 86Box? . . . 5

1.2.8 Why is my emulated PS/2 mouse slow/laggy under Windows 95/98/98 SE/ME even at 100% speed? . . . 6

1.2.9 Can I use 86Box to run a Windows XP system? . . . 6

1.2.10 Why doesn't 86Box have the XYZ board/card/peripheral? . . . 6

1.2.11 Are you going to add emulation of the Pentium III and/or newer CPUs? . . . 7

1.3 ROM set and assets . . . 8

1.3.1 Search path . . . 8

1.3.2 Asset pack . . . 9

1.4 VM manager . . . 10

1.4.1 Machine list . . . 10

1.4.2 Machine details . . . 11

1.4.3 Menu bar . . . 11

1.4.4 Toolbar . . . 12

1.4.5 Status bar . . . 12

1.5 Menu bar . . . 13

1.5.1 Action . . . 13

1.5.2 View . . . 13

1.5.3 Media . . . 15

1.5.4 Tools . . . 15

1.5.5 Help . . . 16

1.6 Toolbar . . . 17

1.6.1 Pause/resume execution . . . 17

1.6.2 Fast forward . . . 17

1.6.3 Hard reset . . . 17

1.6.4 ACPI shutdown . . . 17

1.6.5 Press Ctrl+Alt+Del/Ctrl+Alt+Esc . . . 17

1.6.6 Settings . . . 17

1.6.7 Take/copy screenshot . . . 17

1.6.8 Status area . . . 18

1.7 Status bar . . . 19

1.7.1 Cassette deck . . . 19
1.7.2 PCjr cartridges . . . 19
1.7.3 Floppy drives . . . 19
1.7.4 CD-ROM drives . . . 20
1.7.5 Removable disk, MO and tape drives . . . 20
1.7.6 Hard disks . . . 20
1.7.7 Network . . . 20
1.7.8 Sound . . . 21
1.7.9 Additional information area . . . 21
1.7.10 Refresh rate indicator . . . 21
1.7.11 Keyboard indicators . . . 21
1.8 Settings . . . 22
1.8.1 Machine . . . 22
1.8.2 Display . . . 24
1.8.3 Input devices . . . 26
1.8.4 Sound . . . 27
1.8.5 Network . . . 29
1.8.6 Ports (COM & LPT) . . . 30
1.8.7 Storage controllers . . . 33
1.8.8 Hard disks . . . 35
1.8.9 Floppy & CD-ROM drives . . . 37
1.8.10 Other removable devices . . . 38
1.8.11 Other peripherals . . . 38
1.9 Preferences . . . 41
1.9.1 Emulator . . . 41
1.9.2 Input . . . 41
1.9.3 Key bindings . . . 41

2 Emulated hardware **42**
2.1 Machine-specific notes . . . 42
2.1.1 8088 . . . 42
2.1.2 80286 . . . 42
2.1.3 i386SX . . . 43
2.1.4 i486 . . . 43
2.1.5 Socket 7 . . . 43
2.1.6 Socket 8 . . . 44
2.1.7 Slot 1 . . . 44
2.1.8 Slot 1/2 . . . 44
2.1.9 Slot 2 . . . 45
2.1.10 Socket 370 . . . 45
2.1.11 Miscellaneous . . . 45
2.1.12 Footnotes . . . 45
2.2 Device-specific notes . . . 47
2.2.1 Display . . . 47
2.3 Keyboard . . . 48
2.3.1 Host systems . . . 48
2.3.2 Special keys . . . 48
2.4 Disk images . . . 50
2.4.1 Hard disk images . . . 50
2.4.2 Floppy disk images . . . 50
2.4.3 MO / removable disk images . . . 51
2.4.4 CD-ROM / DVD-ROM optical disc images . . . 51
2.4.5 Cassette tape images . . . 52
2.4.6 PCjr cartridge images . . . 52

2.4.7 Creating and using disk images . . . 52
2.5 Tertiary and quaternary IDE . . . 54
2.5.1 System resources . . . 54
2.5.2 BIOS support . . . 54
2.5.3 Operating system support . . . 54
2.6 Networking . . . 57
2.6.1 Null Driver . . . 57
2.6.2 SLiRP . . . 57
2.6.3 PCap . . . 57
2.6.4 Local Switch . . . 58
2.6.5 VDE . . . 59
2.6.6 TAP . . . 61
2.6.7 Modem . . . 61
2.6.8 Advanced networking features . . . 63
2.7 ISABugger . . . 65
2.7.1 Background . . . 65
2.7.2 Registers . . . 65
2.8 External OPL audio . . . 67
2.8.1 Usage . . . 67

3 Development **68**
3.1 Build guide . . . 68
3.1.1 Toolchain files . . . 68
3.1.2 Presets . . . 68
3.1.3 Obtaining the source code . . . 69
3.1.4 Prerequisites . . . 69
3.1.5 Building . . . 71
3.2 Advanced builds . . . 73
3.2.1 Variants . . . 73
3.3 Manager interface . . . 74
3.3.1 JSON protocol . . . 74
3.3.2 Plain text protocol . . . 75
3.3.3 Window message protocol (Windows-only) . . . 75
3.4 API . . . 77
3.4.1 Devices . . . 77
3.4.2 Timers . . . 85
3.4.3 Threads . . . 87
3.4.4 Port I/O . . . 90
3.4.5 DMA . . . 94
3.4.6 PCI . . . 95
3.5 File formats . . . 112
3.5.1 86F . . . 112
3.5.2 Extended HDI (HDX) . . . 114

86Box is an IBM PC system emulator that specializes in running old operating systems and software designed for IBM PC systems and compatibles from 1981 through fairly recent system designs based on the PCI bus.

86Box is released under the GNU General Public License, version 2 or later. For more information, see the COPYING file.

The project maintainer is OBattler.

If you need a configuration manager for 86Box, use the Avalonia 86, our officially endorsed 86Box configuration manager, developed by notBald (notBald). Please note that Avalonia 86 currently only works on Windows and Linux.

Community

We operate an IRC channel and a Discord server for discussing anything related to retro computing and, of course, 86Box. We look forward to hearing from you!

Additionally, subscribe to our YouTube channel for content such as installation videos, and follow us on Twitter for occasional updates and fun facts.
For more links, visit our official website.

CHAPTER

ONE

USING 86BOX

1.1 Getting started

Here are the basic steps to help you get started with 86Box. The user interface has been designed to resemble Virtual PC, VirtualBox and other virtualizers, so if you used those programs before, this should all look familiar to you.

Important

The steps on this guide are outdated and will be updated soon to reflect the *VM manager*.

Step 1: Get the ROM set

86Box relies on a set of ROM dumps gathered from physical hardware to emulate it. This includes the system BIOS, as well as any option ROMs used by expansion cards. If you try to start 86Box without one, you'll receive an error and 86Box will close. You need to download the ROM set from here, and extract it into one of the *supported locations*.

Step 2: Meet the main window

Once you got the romset in the right place, you can start `86Box.exe`. The main window has three important areas:

- **The menu bar at the top**, where most controls and options are located. See *Menu bar* for more information.
- **The display area in the middle**, which is where the display output from the emulated machine will be rendered.
- **The status bar at the bottom**, containing icons for quickly accessing the configured peripheral devices. See *Status bar* for more information.

Step 3: Configure the hardware

When you start an emulated machine, you probably want to configure it with the hardware options you want. This is much like putting together the hardware components to build a PC. To do this, go to the *Tools* menu and select *Settings*. This will bring up the *Settings* window, which has many options to choose from, split into *a handful of categories*.

Step 4: Configure the BIOS

Once you've selected the hardware components you wish to emulate, you need to make sure they're properly configured. This is done through the system BIOS, the same way it's done on a real computer. The specifics of this will of course differ from one machine to another, but generally speaking, you need to know how to enter the BIOS, which options to change, and which options to leave alone.

Step 5: Mount some images

Now that you've configured everything, you're ready to run some software in your emulated machine. Maybe you want to install an operating system or play a booter game. In any case, you'll have to mount some virtual media to get going. You can do this with the icons in the *status bar*. Icons representing removable media appear semi-transparent when their associated drive is empty, and fully opaque when media is inserted.

When you want to eject virtual media, click on the particular icon again and select *Eject* (for floppy and removable disks) or *Empty* (for CD-ROMs). The icon becomes semi-transparent again.

Step 6: Mouse and keyboard interaction

Now you're ready to do some stuff inside the emulated machine. Keyboard input is redirected there automatically whenever the emulator window has focus. All key presses and combinations will be redirected to the emulated machine.

Mouse input has to be manually "captured" and "released". To capture the mouse in the emulated mahine, simply click inside the renderer area. Your host mouse cursor will disappear and your mouse movement and clicks will be redirected into the emulated machine. Now you can use the mouse inside the emulated machine - if the software and hardware configuration supports it, of course.

To release the mouse, press `Ctrl+End` simultaneously. You can also use the middle mouse button for this if the emulated mouse only has two buttons.

Step 7: What now?

If you made it this far, you got the basics of using 86Box, but there's more features and options to explore. For example, you can try out Avalonia 86 for easier management of multiple emulated machines. You can see what's under the *View* menu, or look at some of the more obscure options in the *Settings* window.

You may eventually encounter the need to get files *into* your machine. Please see *this section for information on disk image formats* and *this section on creating and using disk images*.

Keep in mind that because 86Box is constantly in development, various problems will come and go. If you think something's not working the way it should, consider submitting an issue on GitHub or joining official support channels on Discord or IRC.

Have fun!

1.2 Frequently asked questions

1.2.1 What is the difference between 86Box and applications like VirtualBox or Virtual PC?

VirtualBox, Virtual PC and similar applications are *hypervisors*. For the most part, they execute code running in the virtual machine as is, and only step in whenever it is required to enforce the separation of a virtual machine from the rest of the system. This also means that the virtual machine sees the same CPU as the host system.

They also mostly implement peripherals that are custom designed to let the guest take full potential of the virtualizer as long as appropriate drivers, which are distributed with the provided additions, are installed. This is great for modern operating systems and software that is not designed for a specific hardware target, but rather an abstraction interface such as DirectX; however, running older applications and games will often lead to a suboptimal experience, as hypervisors don't tend to be designed with this usecase in mind.

On the other hand, 86Box is a *system emulator*. It implements a whole system in software, which includes the CPU, chipset and additional cards, if any. Furthermore, it interprets every single instruction running in the virtual machine, and while that comes with the obvious tradeoff of emulation being more CPU intensive than virtualization, it also makes it possible to simulate authentic behavior of the original hardware, including its speed. This in turn allows running countless games and demos that wouldn't have run in a hypervisor before, as they simply run too fast or fail to make use of various hardware quirks that don't exist in modern processors.

In addition, the large variety of peripherals emulated by 86Box also makes it possible to use existing software, games and drivers that had been specifically designed for such peripherals. However, this obviously means that the emulator is also stuck with the limitations of the original hardware, and therefore it is not possible to offer advanced features such as mouse pointer integration.

1.2.2 What is the difference between 86Box and QEMU?

86Box and QEMU are both emulators, as they can both implement a whole computer system in software. However, QEMU is primarily designed to performantly translate between different instruction sets by implementing a generic CPU, and therefore doesn't try to emulate all the various quirks of the real hardware like 86Box.

Similarly to hypervisors, QEMU also implements certain fictional peripherals that are designed to reduce the emulation overhead. Again, this all is great for modern software that is not designed for a specific hardware target, but not so great for older software.

A common point of confusion is also QEMU's option to pick a specific CPU to be emulated, similarly to 86Box. This however only has the effect of changing the reported CPU identification (`CPUID` instruction in x86) and does not impact the behavior of the emulation in any way.

1.2.3 What is the difference between 86Box and PCem?

86Box and PCem are both PC system emulators. In fact, 86Box originally started out as a fork of PCem in 2016. However, the codebases of both emulators have since then diverted by a lot. Because of this, features and bugs found in one emulator do not necessarily have to be present in the other.

In general, 86Box focuses more on the accuracy of emulation, especially for older 8088/8086 based systems. This makes it more compatible with older applications, games and demos that make use of clever hardware tricks to make do with the limited computing power of the time.

Meanwhile, PCem often takes various shortcuts to improve performance at the cost of accuracy, which does end up limiting the selection of software it can run.

1.2.4 Why is 86Box unable to run Xbox-level hardware when emulators for PS3 and Switch exist?

The difference is in how high level emulation and low level emulation works. High level lets you run much faster speeds at the cost of accuracy and compatibility. See the Emulation General Wiki for a detailed explanation.

Back in the late 90s, Ultra HLE came out that emulated an N64 at 100% speed on a system with a Pentium II 300. It would take another decade before computers got fast enough to do it in low level. The tradeoff is that HLE only ran about 20 games because the emulator had to be built specifically to run each game.

Simply put, think of it this way:

- **high level** = good enough to run just "these things"
- **low level** = emulate the *entire* system to run literally anything that original hardware could have run, even including the hardware incompatibilities

In 86Box it's entirely possible to put a video card with a motherboard that would not boot and that replicates real life. Any console emulator doing PS2 or higher is doing high level emulation. This is why they have lists of compatible games - if they actually full emulated all the hardware, every game would just work because it would not know the difference but they cannot because no computer is fast enough.

1.2.5 My virtual machine does not run at 100% speed, what can I do to fix this?

If the emulation speed is consistently way under 100%, then your configuration is too demanding for your host system. Try to pick a slower emulated CPU speed.

However, if you only experience casual drops in emulation speeds, you should not instantly worry, as the guest might simply be doing some heavy I/O operations.

1.2.6 What is the top VM configuration my system will handle?

There is no formula that would tell you this with 100% certainty. In general, the higher the host's IPC (instructions per clock) rating, the higher emulated CPU speeds it can handle. However, the emulation speeds also depend on the kind of software that runs in the virtual machine.

A good way to estimate the limit of your host setup is by looking at single-thread benchmarks. The higher a CPU is on this list, the faster it will run 86Box.

For example:

- **~4000** = Pentium II 300 MHz
- **~3400** = Pentium II 233 MHz
- **~2600** = Pentium 200 MHz
- **~1600** = Pentium 75 MHz
- **~700** = 486DX2 66 MHz (assuming the GPU on such a system can keep up)

Keep in mind that these are only rough estimates. The best way to optimize your virtual machine configuration is simply trial and error.

1.2.7 What are some era-appropriate configurations for 86Box?

- **1988** - 386DX 25 MHz, 1 MB RAM, ATI VGA Wonder, AdLib, DOS 4.01 + Windows 2.1x/386
- **1990** - 486DX 33 MHz, 4 MB RAM, Video 7 VRAM VGA + XGA, Sound Blaster 1.5 (CT1320C), DOS 4.01 + Windows 3.0
- **1992** - 486DX2 66 MHz, 8 MB RAM, S3 924 / ATI Mach32, Sound Blaster 16 or Gravis Ultrasound, DOS 5.0 + Windows 3.1

- **1994** - Pentium 100 MHz, 32 MB RAM, S3 Vision964 / ATI Mach64, Sound Blaster AWE32, DOS 6.22 + Windows 3.11
- **1996** - Pentium 200 MHz non-MMX (or Pentium Pro 200 MHz), 64 MB RAM, Matrox Mystique + Voodoo 1, Sound Blaster AWE64 Gold, Windows 95 OSR2 (or Windows NT 4.0 for a Pentium Pro)
- **1998** - Pentium II 450 MHz, 128 MB RAM, Matrox Mystique 220 + Voodoo 2 SLI, Ensoniq AudioPCI, Windows 98 FE

1.2.8 Why is my emulated PS/2 mouse slow/laggy under Windows 95/98/98 SE/ME even at 100% speed?

These operating systems set the polling rate of the mouse to 40 Hz by default, which is generally considered a really low polling rate.

Either install Microsoft IntelliPoint (3.0 for Windows 95, 4.0/4.1 for Windows 98/98 SE/ME) or use PS2Rate and select 200 Hz polling rate. The former is only recommended if the Wheel options are selected; it does not require any other programs to be enabled as it will select high polling rates by default and will work better with MS-DOS programs and games running inside the MS-DOS Prompt. Note that 4.0 requires Internet Explorer 5.5 to be installed.

Alternatively, you can check the "Update mouse every CPU frame" option in the Actions menu, but this is not accurate to the emulation and may cause noticeable performance drops.

1.2.9 Can I use 86Box to run a Windows XP system?

We strongly discourage the use of 86Box to run Windows XP or newer Windows operating systems.

While the operating system does run, the Windows XP system requirements are steep compared to what 86Box can offer, and what most host systems are capable of running at 100%. Microsoft publishes the *minimum* system requirements of Windows XP to include a 233 MHz Pentium. The recommended requirements of Windows XP bump the requirements up to include a 300 MHz Pentium or better.

Even though these requirements seem modest in the present day, only some of the newest AMD (Ryzen 5000 series or better), Intel (Core 12th generation or better), and Apple (M2 or better) processors are capable of meeting the bare minimum Windows XP requirements under 86Box's full system emulation. On top of the operating system requirements, most software and games that require Windows XP will demand even more out of the hardware than Windows XP alone, reducing the utility of Windows XP in 86Box to a very narrow set of software.

It is almost always better to run Windows XP in a different virtualizer or emulator. On AMD and Intel based PCs, software such as VirtualBox or Hyper-V can run Windows XP close to bare-metal performance, as most CPU instructions can run unmodified. On Apple macOS computers, UTM is capable of either running Windows XP in full hardware virtualization on Intel Macs, or with highly optimized x86 emulation on M-series Macs. UTM does not chase the fine-detail historic accuracy of 86Box and as such, is able to take "shortcuts" that can allow Windows XP and software on it to run with acceptable performance even under x86 emulation.

1.2.10 Why doesn't 86Box have the XYZ board/card/peripheral?

86Box is entirely comprised of volunteers. Any hardware added is done if someone is willing to contribute the code and work required to make it feasible. Some cards/hardware may never be in 86Box. This might be because documentation is non-existant, the hardware might be out of scope of the project, or other reasons. Asking for cards doesn't make it happen. There is a LOT of work that has to be accomplished to add anything. Anyone is welcome to contribute to the code base and make additions though, provided you follow our guidelines.

1.2.11 Are you going to add emulation of the Pentium III and/or newer CPUs?

In short, no. Newer CPUs are way too powerful and even the top-end systems that are currently on the market are not nearly performant enough to be able to emulate them at usable speeds. In fact, we already had to add some low-clocked variants of the Pentium II that never actually existed, just so more people could use it!

For further reading, team member RichardG867 wrote a blog post that goes into the details of what makes the emulation of newer CPUs so controversial.

1.3 ROM set and assets

86Box relies on a set of ROM dumps gathered from physical hardware to emulate it. This includes the system BIOS, as well as any option ROMs used by expansion cards. Additionally, some features require an *asset pack* installed separately from the ROM set.

The ROM set and asset pack are organized into several directories for each device type, each of which contains futher subdirectories for each machine or device model or category.

Note

The expected file names of the ROM dumps and their locations within the set are hardcoded in the emulator. If you rename them or add your own dumps with different file names, the emulator will not be able to make use of them.

1.3.1 Search path

The emulator utilizes a search path mechanism to locate ROMs. By default, the following locations are considered:

1. `roms` subdirectory in the VM path
2. `roms` subdirectory in the same directory as the emulator executable
3. Platform-specific locations

A custom location can be specified by using the `-R` or `--rompath` command line argument, which then precedes any other considered locations.

Tip

The list of all paths searched when loading ROMs is printed to the log and standard output when 86Box starts.

Windows

The following locations are searched on Windows:

1. `%LOCALAPPDATA%\86Box\roms`
2. `%PROGRAMDATA%\86Box\roms`

Unix

86Box honors the XDG base directory specification on Linux and other Unix-compatible platforms. The following locations are searched:

1. `$XDG_DATA_HOME/86Box/roms`
2. `86Box/roms` subdirectory in each path listed in `$XDG_DATA_DIRS`

This usually resolves to `~/.local/share/86Box/roms`, `/usr/local/share/86Box/roms` and `/usr/share/86Box/roms` (in order).

macOS

The following locations are searched on macOS:

1. `~/Library/Application Support/86Box/roms`
2. `/Library/Application Support/86Box/roms`

Note

If 86Box cannot find any ROM dumps even after you extracted the set into one of the directories listed above, please make sure that the `roms` folder name does not contain a hidden extension. This can be fixed by selecting the folder in Finder, choosing *Get Info* in the context menu or the *File* menu, and renaming the folder in the *Name & Extension* section of the information window.

1.3.2 Asset pack

Some features also require an **asset pack**, which is included with release builds of 86Box downloaded from GitHub, as the `assets` folder inside the downloaded archive on Windows, or embedded within the application on Linux and macOS. It must be installed manually in other situations, such as when using *advanced builds*.

Currently, the *floppy audio* and *hard disk audio* features require the asset pack to be installed.

The **same search path mechanism** *described above* applies to the asset pack, but with a directory named `assets` instead of `roms`. The `-A` or `--assetpath` command line argument can be used to specify a custom location.

1.4 VM manager

Opening 86Box will start the **virtual machine manager**, which allows for creating, managing, starting and controlling multiple emulated machine configurations.

Note

- This manager is currently a preview, with a limited feature set expanding upon the previous standalone 86Box Manager app. Other managers with more features can still be used.
- Running 86Box directly no longer creates or starts an emulated machine in the current folder like on previous versions. The `-P`/`--vmpath` command line option can be used to start a machine directly instead.
- The manager can be fully disabled (restoring the behavior of previous 86Box versions when launched without specifying a virtual machine path on the command line) by adding `vmm_disabled = 1` to the global configuration file, which is stored at one of the following locations based on the platform:
  - **Windows**: `C:\Users\[username]\AppData\Local\86Box\86box_global.cfg`
  - **Linux**: `~/.config/86Box/86box_global.cfg`
  - **macOS**: `~/Library/Preferences/86Box/86box_global.cfg`

1.4.1 Machine list

The **left-hand side** of the manager window displays a list of all machines found in the *system directory*, along with their current state and icon. Click on a machine to select it.

The following options are available by **right-clicking** a machine:

- **Start:** start the machine.
- **Hard reset:** force a reset of the machine.
- **Force shutdown:** force a shutdown of the machine. If this fails due to a frozen 86Box process, try *Kill*.
- **Ctrl+Alt+Del:** send a *Ctrl+Alt+Del* key combination to the machine.
- **Settings:** open the *Settings* window to configure the machine.
- **Change display name:** change the name by which the machine is identified on the manager and 86Box window. Changing this will not rename the machine's folder.
- **Set icon:** change the icon displayed next to the machine on the list.
  - Select an icon from the preset list, or click **Reset** to restore the default icon.
- **Clone:** make a copy of the machine.
- **Kill:** forcibly terminate the machine's 86Box process if one is running.
- **Wipe NVRAM:** clear the machine's CMOS non-volatile memory. On models with Flash ROM, the original BIOS is also reflashed.
- **Delete:** delete the machine, along with **everything** stored within its directory.
- **Open folder:** open the directory where the machine's configuration file is stored.
- **Open printer tray**: open the directory where documents printed by the machine's *emulated printers* are saved.
- **Open screenshots folder**: open the directory where screenshots of the machine are saved.
- **Show config file:** display the contents of the machine's `86box.cfg` file for sharing, support requests and bug reports.

Search

The **search box** at the bottom of the machine list allows for filtering the list by any of the following criteria:

- **Display name** and **folder name**.
- Names of **hardware components** present in the machines, as displayed in the *details pane*.
- **Image file names** for any media inserted into the machines, including hard disks, floppies and CDs.

Advanced users can *enable regular expressions* to perform more complex searches.

1.4.2 Machine details

The **right-hand side** of the manager window displays information and controls for the selected machine:

- A **summary** of the machine's *configuration*.
- A gallery of **screenshots** saved through *Take screenshot* or the respective keyboard shortcut.
- A small text area for writing any **notes** about the machine.
- Controls for the machine: **Settings**, **Hard reset**, **Force shutdown**, **Start/Pause**, **Ctrl+Alt+Del**.
- The machine's current **status**, with the 86Box process ID if one is running.

1.4.3 Menu bar

The **menu bar** located at the top of the manager window provides controls for the manager as a whole.

File

- **New machine:** create a new machine from scratch or from an existing configuration file.
- **Exit:** quit the manager. Requires confirmation if any machines are currently running.

View

- **Hide toolbar:** hides the *toolbar* below the menu bar.

Tools

- **Preferences:** open the *Preferences* window, which provides the following options:
  - **System Directory:** view or change the folder where emulated machines are stored. The *Default* button resets to the default folder.
  - **Language:** select a language for the 86Box user interface. The *Default* button resets back to the system language.
  - **Remember size & position:** automatically save the manager window's size and position and the machine list's width.
  - **Check for updates on startup:** automatically check for 86Box updates when starting the manager.
  - **Use regular expressions in search box:** enable the use of Perl-syntax regexes to perform more complex searches with the search box.
  - **Color scheme:** select a visual style for the 86Box user interface. *System* uses the operating system's global preference if possible.

Note

- The manager **must be restarted** for any changes to the system directory to take effect.
- The system directory is **scanned recursively** for machines through their `86box.cfg` files.

- **Check for updates:** check for and download any available 86Box version update.

Help

- **Documentation:** open the very documentation you're reading.
- **About 86Box:** show credits, license and build information about 86Box.

1.4.4 Toolbar

The **toolbar** located at the top of the manager window (right below the menu bar) provides controls for the machine list.

- **New machine:** create a new machine from scratch or from an existing configuration file.
- **Start:** start the selected machine.
- **Hard reset:** force a reset of the selected machine.
- **Force shutdown:** force a shutdown of the selected machine.
- **Ctrl+Alt+Del:** send a *Ctrl+Alt+Del* key combination to the selected machine.
- **Settings:** open the *Settings* window to configure the selected machine.

1.4.5 Status bar

The **status bar** located at the bottom of the manager window displays a **count** of running, paused and total available machines.

Additionally, any information about **available updates** will be displayed in the status bar if *checking for updates on startup* is enabled.

1.5 Menu bar

The menu bar located at the top of the 86Box window provides controls for the emulated machine as a whole, its display, and the 86Box user interface.

> **Important**
>
> On macOS, the **Exit** (Quit), **Preferences** and **About 86Box** options are found in the **86Box** application menu instead of the locations outlined here.

1.5.1 Action

- **Keyboard requires capture:** require the mouse to be captured for keypresses to be forwarded to the emulated machine. Enabling this option allows the use of keyboard combinations (such as Alt+Tab) on the host system while the 86Box window is focused.
- **Right CTRL is left ALT:** let the right Ctrl key act as a left Alt key, to simulate some special keyboards where the Alt key is located on the right side of the space bar.
- **Update mouse every CPU frame:** force the emulated mouse to send movements as fast as possible, ignoring any polling rate set by the emulated operating system.
- **Auto-pause on focus loss:** automatically pause emulation while the 86Box window is not focused.
- **Pause / Resume:** pause emulation of the machine. Select this option again to resume emulation. You can alternatively press `Ctrl+Alt+F1` (*customizable*) to pause or resume emulation.
- **Fullscreen:** enter full screen mode. You can press `Ctrl+Alt+Page Up` (*customizable*) to enter full screen mode or go back to windowed mode. The menu bar, *toolbar* and *status bar* are hidden by default in full screen mode; press `Ctrl+Alt+Page Down` (also *customizable*) to show or hide them.
- **Fast forward:** run the emulated machine at the highest speed your host system can handle. Uncheck this option to return to normal speed. You can alternatively press `Ctrl+Alt+F` (*customizable*) to control this option.
- **Force interpretation / Allow recompilation:** temporarily disable the *dynamic recompiler*. Uncheck this option to reenable the recompiler. You can alternatively press `Ctrl+Alt+I` (*customizable*) to control this option.
- **Hard Reset:** force a reset of the emulated machine. Requires confirmation, which can be disabled by checking the *Don't show this message again* box. You can alternatively press `Ctrl+Alt+F12` (*customizable*) to hard reset.
- **Ctrl+Alt+Del:** send a *Ctrl+Alt+Del* key combination to the emulated machine. You can alternatively press `Ctrl+F12` (*customizable*) to send that combination.
- **Ctrl+Alt+Esc:** send a *Ctrl+Alt+Esc* key combination to the emulated machine. You can alternatively press `Ctrl+F10` (*customizable*) to send that combination.
- **ACPI shutdown:** send a power button press to the emulated machine. Only available on machines with ACPI soft power off support.
- **Exit:** quit 86Box. Requires confirmation, which can be disabled by checking the *Don't show this message again* box.

1.5.2 View

- **Hide toolbar:** hides the *toolbar* below the menu bar.
- **Hide status bar:** hides the *status bar* at the bottom of the window.
- **Show non-primary monitors:** shows or hides the secondary display window if a *secondary video card* is configured.

- **Resizeable window:** allow the 86Box window to be freely resized. Unchecking this option will also return the window to its normal size.
- **Remember size & position:** automatically save the 86Box window's size and position for this emulated machine.
- **Specify dimensions:** open a window where an exact size (in pixels) for the emulated display can be set. If checked, the *Lock to this size* box prevents changes in the emulated display's resolution from overriding the specified size.
- **Force 4:3 display ratio:** stretch the emulated display to a 4:3 aspect ratio, independently of the emulated machine's screen resolution.
- **Fullscreen stretch mode:** select the picture mode to use when in full screen mode.
  - **Full screen stretch:** stretch the emulated display to completely fill the host display.
  - **4:3:** stretch the emulated display to a 4:3 aspect ratio, then scale it to fit the host display.
  - **Square pixels (keep ratio):** scale the emulated display to fit the host display, without changing the aspect ratio.
  - **Integer scale:** scale the emulated display to the largest integer scale factor to fit the host display. This provides the highest possible picture quality, at the cost of black bars if the host display's resolution is not divisible by the emulated display's resolution.
  - **4:3 integer scale:** stretch the emulated display to a 4:3 aspect ratio, then scale it to the largest integer scale factor to fit the host display.
- **Apply fullscreen stretch mode when maximized:** apply the picture mode selected above in windowed mode if *Resizeable window* is enabled and the window is maximized.
- **CGA composite settings:** adjust the picture's hue, saturation, brightness, contrast and sharpness. Only available when emulating a composite CGA monitor.
- **Window scale factor:** scale the emulated display to half (*0.5x*), normal (*1x*), 50% larger (*1.5x*), double (*2x*) or larger (up to *8x*) sizes.
- **HiDPI scaling:** automatically scale the emulated display to real size if your host system has a HiDPI display. This option can be used alongside *Window scale factor* above.

> **Note**
>
> If HiDPI scaling is disabled on a host with a HiDPI display, the emulated display's size may be off by one pixel due to an integer scaling limitation.

- **Filter method:** select the filtering method (*Nearest* or *Linear*) to be used when scaling the emulated display.
- **Renderer:** select a graphical renderer for the emulated display.
  - **Qt (Software)** is recommended in most cases.
  - **Qt (OpenGL)** and **Vulkan** are known to perform better on some host systems. Try these if your system is struggling to maintain 100% emulation speed. *Vulkan* may not be available if the host GPU is not Vulkan-capable.
  - **OpenGL (3.0 Core)** allows for shader effects to be applied to the emulated display, however, it is not compatible with older integrated GPUs.
- **Renderer options:** open a window to configure the *OpenGL (3.0 Core)* renderer. This option will be available if that renderer is selected.
  - **Synchronize with video:** update the emulated display at its current refresh rate.

- **Use target framerate:** update the emulated display at the selected refresh rate.
- **VSync:** enable vertical sync. Recommended if tearing artifacts are observed.
- **Add:** add a `.glsl` or `.glslp` file to the list of shaders to apply on the emulated display.
- **Remove:** remove the selected shader from the list.
- **Configure:** open a window to configure parameters on the selected shader.
- **Move up/down:** move the selected shader up or down in the processing order.

Note

- Many shaders are available for simulating CRT displays, VHS tapes and other aesthetics; the RetroArch glsl-shaders repository is a good place to start.
- `.cg` and `.cgp` shaders are not supported, as these formats are long deprecated.

- **OpenGL input stretch mode:** select the picture mode to use for the raw image data fed into the applied OpenGL shader(s). Refer to *Fullscreen stretch mode* below for the available options.
- **OpenGL input scale:** scale the raw image data fed into the applied OpenGL shader(s) to half (*0.5x*), normal (*1x*), 50% larger (*1.5x*), double (*2x*) or larger (up to *8x*) sizes.

1.5.3 Media

This menu lists all storage drives and network cards attached to the emulated machine, and provides the same options that are accessible by clicking the respective device's icon on the *status bar*.

The **Clear image history** option empties the list of recently-loaded images on all storage drives.

1.5.4 Tools

- **Settings:** open the *Settings* window to configure the emulated machine.
- **Update status bar icons:** enable the activity lights on *status bar* icons. Unchecking this option may improve emulation performance on low-end host systems.
- **Enable Discord integration:** enable Discord Rich Presence. 86Box shares the emulated machine's name, model and CPU with other Discord users.

Note

Integration requires the Discord desktop app, running on Windows (x64 only), Linux (`x86_64` only) or macOS. Discord does not provide integration support for other operating systems / architectures or the browser app. Additionally, integration will not be available on Windows if the included `discord_game_sdk.dll` file is missing from the 86Box directory.

- **Take screenshot:** take a screenshot of the emulated display and save it as a .png image in the `screenshots` subdirectory found in the emulated machine's directory, which can be opened with the **Open screenshots folder** option below. You can alternatively press `Ctrl+F11` (*customizable*) to take a screenshot.
- **Take raw screenshot:** same as *Take screenshot* above, but making a pixel-perfect capture of the machine's video output instead, without any processing such as scaling and shaders.
- **Copy screenshot:** take a screenshot of the emulated display and copy it to the host system's clipboard.

- **Copy raw screenshot:** same as *Copy screenshot* above, but making a pixel-perfect capture of the machine's video output instead, without any processing such as scaling and shaders.
- **Sound:** provides the same options that are accessible by clicking the *sound icon on the status bar*.
- **Preferences:** open the *Preferences window* window to configure options related to 86Box as a whole.
- **MCA devices**: open the *MCA devices* window, which lists the IDs and required Adapter Definition Files of all Micro Channel devices installed on the emulated machine. This option will only be available when emulating a Micro Channel Architecture-based machine.
- **Open printer tray**: open the host system's file browser on the directory where documents printed by *emulated printers* are saved.
- **Open screenshots folder**: open the host system's file browser on the directory where screenshots of this emulated machine are saved. Screenshots can also be viewed through the *VM manager*.

1.5.5 Help

- **Documentation:** open the very documentation you're reading.
- **About 86Box:** show credits, license and build information about 86Box.

1.6 Toolbar

The toolbar located at the top of the 86Box window (right below the *menu bar*) has two purposes: it provides quick actions for the emulated machine on its left hand side, and displays status information on its right hand side.

1.6.1 Pause/resume execution

Pause emulation of the machine. Press again to resume emulation. You can alternatively press `Ctrl+Alt+F1` (*customizable*) to pause or resume emulation.

Note

Emulation is automatically paused when the emulated machine enters **ACPI sleep mode**.

1.6.2 Fast forward

Run the emulated machine at the highest speed your host system can handle. Press again to return to normal speed. You can alternatively press `Ctrl+Alt+F` (*customizable*) to control this option.

Note

When fast forwarding, all emulated audio is automatically **muted** to prevent distortion.

1.6.3 Hard reset

Force a reset of the emulated machine. Requires confirmation, which can be disabled by checking the *Don't show this message again* box. You can alternatively press `Ctrl+Alt+F12` (*customizable*) to hard reset.

1.6.4 ACPI shutdown

Send a power button press to the emulated machine. Only available on machines with ACPI soft power off support.

1.6.5 Press Ctrl+Alt+Del/Ctrl+Alt+Esc

Send a *Ctrl+Alt+Del* (left-most icon) or *Ctrl+Alt+Esc* (right-most icon) key combination to the emulated machine. You can alternatively press `Ctrl+F12` to send a *Ctrl+Alt+Del* combination, or `Ctrl+F10` to send *Ctrl+Alt+Esc*; both key combinations are *customizable*.

1.6.6 Settings

Open the *Settings* window to configure the emulated machine.

1.6.7 Take/copy screenshot

Take a screenshot of the emulated display. The left-most icon saves the screenshot as a .png image in the `screenshots` subdirectory found in the emulated machine's directory, which can be opened with the **Open screenshots folder** option in the *Tools menu*, while the right-most icon copies the image to the host system's clipboard instead. You can alternatively press `Ctrl+F11` to take a screenshot to file; the key combinations for all screenshot commands are *customizable*.

1.6.8 Status area

The right hand side of the toolbar displays status information, such as:

- **Emulation speed** in percentage. If this number stays consistently below 100%, your host system is not keeping up with emulating the configured hardware.
- **Mouse state** (captured or released) if a *mouse* is enabled.
- **Pause indicator** if emulation is paused.

1.7 Status bar

The status bar located at the bottom of the 86Box window provides icons related to devices attached to the emulated machine. Move your mouse cursor over an icon to see what device it represents. **Most icons can be clicked on** to access options related to their respective devices, which are listed below, and image files can be dropped on the icons for removable media devices such as floppy and CD-ROM drives.

Additionally, green or red indicator lights will appear on the lower corners of an icon when its device is in use, denoting reads and writes respectively, unless *Update status bar icons* is disabled. Some devices also provide an additional indicator on the lower left corner.

1.7.1 Cassette deck

A cassette tape icon will appear if *IBM cassette emulation* is enabled. An indicator will appear on the lower left corner of the icon while the cassette is played or recorded.

- **New image:** create a new cassette tape image file.
- **Existing image:** insert a *cassette tape image file* into the deck. Dragging and dropping an image file on the icon will also load it.
- **Existing image (Write-protected):** insert a cassette tape image file into the deck as a read-only tape.
- A history of the last few images that were loaded into the deck. Click on an entry to load it back.
- **Record:** start recording data to the cassette tape. Not available if the tape is read-only.
- **Play:** start playing the cassette tape.
- **Rewind to the beginning:** rewind the cassette tape to its beginning.
- **Fast forward to the end:** fast forward the cassette tape to its end.
- **Eject:** remove the currently-inserted cassette tape from the deck.

1.7.2 PCjr cartridges

Two cartridge icons will appear if the **IBM PCjr** is being emulated. Each icon corresponds to a cartridge slot on the PCjr's front panel.

- **Image:** insert a *cartridge image file* into this slot. Inserting a cartridge will reset the PCjr. Dragging and dropping an image file on the icon will also load it.
- A history of the last few images that were loaded into this slot. Click on an entry to load it back.
- **Eject:** remove the currently-inserted cartridge from this slot.

1.7.3 Floppy drives

A 3.5" or 5.25" floppy icon will appear for each configured *floppy drive*.

- **New image:** create a new disk image file. Opens the *New Image* window, which lets you select the image size and where to save the file.
- **Existing image:** insert a *disk image file* into this drive. Dragging and dropping an image file on the icon will also load it.
- **Existing image (Write-protected):** insert a disk image file into this drive as a read-only disk.
- A history of the last few images that were loaded into this drive. Click on an entry to load it back.
- **Use host floppy drive:** attach a host floppy drive to this emulated drive. Only available on Linux and macOS hosts.

- **Export to 86F:** convert the currently-inserted disk image file to 86Box's *86F* surface image format. You will be asked where to save the converted file.
- **Eject:** remove the currently-inserted disk from this drive, or detach a host drive.

1.7.4 CD-ROM drives

A CD or DVD icon will appear for each configured *CD-ROM drive*. An indicator will appear on the lower left corner of the icon while *CD audio* is played or paused.

- **Mute:** mute any CD audio played through this drive's analog output. CD audio is unmuted by default on the first configured CD-ROM drive.
- **Image:** insert a *CD-ROM or DVD-ROM disc image file* into this drive. Dragging and dropping an image file on the icon will also load it.
- **Folder:** insert a virtual CD-ROM or DVD-ROM with the contents of a host folder into this drive. Dragging and dropping a folder on the icon will also load it.
- A history of the last few images, folders or host drives that were loaded into this drive. Click on an entry to load it back.
- A list of host CD-ROM or DVD-ROM drives available for passthrough. Click on an entry to attach it to the emulated drive.
- **Eject:** remove any disc inserted into this drive, or detach a host drive.

1.7.5 Removable disk, MO and tape drives

A removable disk, ZIP, MO or tape icon will appear for each configured *additional removable storage drive*.

- **New image:** create a new disk image file. Opens the *New Image* window, which lets you select the image size and where to save the file.
- **Existing image:** insert a *disk image file* into this drive. Dragging and dropping an image file on the icon will also load it.
- **Existing image (Write-protected):** insert a disk image file into this drive as a read-only disk.
- A history of the last few images that were loaded into this drive. Click on an entry to load it back.
- **Eject:** remove the currently-inserted disk from this drive.

1.7.6 Hard disks

A hard disk icon will appear for each configured *hard disk bus*. For example, if you have both IDE and SCSI hard disks configured, two hard disk icons will appear: one representing all IDE disks, and another one representing all SCSI disks. No options are available.

1.7.7 Network

A network icon will appear for each configured *network card*.

- **Connected:** connect this card to its network. Network cards with link state detection support will report a disconnected cable if this option is unchecked.

1.7.8 Sound

This icon is always present, providing options to control all audio produced by the emulated machine's PC speaker, *sound cards* and other sound hardware.

- **Mute:** mute all audio. You can alternatively press `Ctrl+Alt+M` (*customizable*) to mute or unmute audio.
- **Sound gain:** open a gain control, which allows for increasing the loudness of all audio.

> **Note**
>
> Sound options do not apply to MIDI music sent to a software synthesizer through the *System MIDI* device, as these synthesizers are external to 86Box.

1.7.9 Additional information area

This area, located to the right of the icons described above, contains additional information which may be provided by components such as the *ISABugger* and *POST card*.

MT-32 display

Any text messages sent to the LCD screen of an *emulated Roland MT-32/CM-32L synthesizer* are displayed here.

ISABugger

The ISABugger's hexadecimal displays and LED banks are displayed here. See *ISABugger* for more information.

POST card

The leftmost hexadecimal value is the most recent POST code reported, while the rightmost value is the second most recent code, like on a real dual-display POST card. A value of `--` indicates that no POST code has been reported yet.

> **Note**
>
> The additional information area can only be used by one component at a time. The MT-32 display has the highest priority, followed by both the ISABugger and POST card with the same priority (taking over whenever they're written to).

1.7.10 Refresh rate indicator

The emulated monitor's current refresh rate, as defined by the *display hardware*.

A *Monitor in sleep mode* message is displayed if the emulated monitor has been put into DPMS sleep mode by the operating system. Pressing a key or moving the mouse is often enough to wake the monitor up.

1.7.11 Keyboard indicators

Indicator lights for Num Lock, Caps Lock and Scroll Lock on the emulated keyboard are displayed on the right side of the status bar.

A Kana Lock indicator is also displayed when emulating a *Japanese AX keyboard*.

1.8 Settings

The *Settings* window allows you to configure the emulated machine.

Changing most settings requires a hard reset of the emulated machine. If any changes were made to those settings, you will be asked whether or not you want to save the changes and hard reset the machine upon pressing *OK* or closing the window. Press *Cancel* to immediately discard all changes.

1.8.1 Machine

The **Machine** page contains settings related to the emulated machine as a whole, such as the machine type, CPU type and amount of memory.

Machine

Machine type / Machine

Machine/motherboard model to emulate. The *Machine* box lists all available models for the machine class selected on the *Machine type* box. Click ⊗ to search for a machine by name or chipset across all classes.

The *Configure* button opens a new window with settings specific to the machine, such as BIOS type selection.

Note

Settings for the machine's on-board devices have been moved to the *Configure* buttons at the devices' respective locations; for instance, configuring the amount of installed video memory for an on-board video chip is now done through the *Configure* button next to the *Display page's Video box* when the *Internal device* option is selected there.

Memory

Amount of RAM to give the emulated machine. The minimum and maximum allowed amounts of RAM will vary depending on the machine selected above.

Time synchronization

Automatically copy your host system's date and time over to the emulated machine's hardware real-time clock. Synchronization is performed every time the emulated operating system reads the hardware clock to calibrate its own internal clock, which usually happens once on every boot.

- **Disabled:** do not perform time synchronization. The emulated machine's date and time can be set through its operating system or BIOS setup utility. Time only ticks while the emulated machine is running.
- **Enabled (local time):** synchronize the time in your host system's configured timezone. Use this option when emulating an operating system which stores *local time* in the hardware clock, such as DOS or Windows.
- **Enabled (UTC):** synchronize the time in Coordinated Universal Time (UTC). Use this option when emulating an operating system which stores *UTC time* in the hardware clock, such as Linux.

Processor

CPU type / Frequency

Main processor to emulate. The *Speed* box lists all available speed grades for the processor family selected on the *CPU type* box. These boxes only list processor types and speed grades supported by the machine selected above.

FPU

Math co-processor to emulate. This box is not available if the processor selected above has an integrated co-processor or lacks support for an external one.

Wait states

Number of memory wait states to use on a 286- or 386-class processor. This box is not available if any other processor family is selected above.

Softfloat FPU

Enable a slower but more accurate math co-processor emulation, for running a limited set of operating systems and applications which demand full 80-bit precision from the floating point unit.

Performance

PIT mode

Programmable Interval Timer emulation mode. **Auto** should cover most use cases, automatically selecting **Fast** mode on 486-class and newer processors or **Slow** mode on older ones. A limited set of timing-sensitive applications require **Slow** mode, which is slower but more accurate.

Use the 486 interpreter for 286 and 386 processors

Enable a faster but less accurate CPU emulation engine on 286- or 386-class processors. Recommended for low-end host systems which are struggling to emulate faster 386s. This option is not available if any other processor family is selected above.

Dynamic Recompiler

Enable the dynamic recompiler, which provides faster but less accurate CPU emulation. The recompiler is available as an option for 486-class processors, and is mandatory starting with the Pentium.

Note

The recompiler can be disabled temporarily (even on processors where it is mandatory) through an option on the *Action menu*, in the unlikely event that an application performs worse with the recompiler enabled. Selecting this temporary option again or restarting 86Box will reenable the recompiler.

CPU frame size

Change the emulator's frame timing behavior. This mostly affects the smoothness of the emulated mouse and other input peripherals.

- **Larger frames (less smooth):** same behavior as 86Box 4.2.1 and older. Switching to it may improve performance on low-end host systems, at the expense of reduced input smoothness.
- **Smaller frames (smoother):** the new default behavior introduced in 86Box 5.0, which improves input smoothness.

1.8.2 Display

The **Display** page contains settings related to the emulated machine's 2D and 3D video cards.

General

Video

Video card to emulate. This box only lists cards supported by the machine's expansion buses. Click ⊗ to search for cards by name, bus or model variant. On machines equipped with an on-board video chip, the *Internal device* option enables the on-board video.

The *Configure* button opens a new window with settings specific to the selected video card, such as the amount of video memory.

Video #2

Optional secondary video card to emulate. Click ⊗ to search for cards by name or bus. Only the **MDA**, **Hercules**, **Hercules Plus** and a limited set of **PCI/AGP VGA** cards are currently supported as secondary options. The secondary card's video output is displayed on a separate window.

As with the primary card above, the *Configure* button can be used to configure the selected card.

Voodoo 1 or 2 Graphics

Emulate a **3dfx Voodoo** add-on 3D accelerator, connected to both the PCI bus and the video card selected in the *General* tab.

> **Note**
>
> The **Voodoo Banshee** and **Voodoo 3** are independent video cards, which are not found here; they must be selected on the *Video* box above, and this Voodoo Graphics option **cannot be selected** alongside them. For these cards, the *Configure* button next to the *Video* box provides similar settings to the ones listed below.

The *Configure* button provides the following settings:

- **Voodoo type:** type of Voodoo card to emulate.
  - **Voodoo Graphics:** the original Voodoo model, with a single Texture Mapping Unit operating at 50 MHz.
  - **Obsidian SB50 + Amethyst:** a variant of the Voodoo Graphics, with two Texture Mapping Units operating at 50 MHz.
  - **Voodoo 2:** the second Voodoo model, with two Texture Mapping Units operating at 90 MHz, as well as SLI support.
- **Framebuffer memory size / Texture memory size**: amount of video memory for the Frame Buffer Interface and Texture Mapping Unit(s), respectively.
- **Bilinear filtering:** apply bilinear filtering to smooth out textures displayed on screen.
- **Dither subtraction:** apply a different texture dithering algorithm.
- **Screen Filter:** apply a filter to make the screen picture resemble the DAC (digital-to-analog converter) output of a real Voodoo card.
- **Render threads:** split the workloads of each Voodoo card into different CPU threads for faster emulation. The recommended amount of render threads depends on your host system's CPU core count, and whether or not Voodoo 2 SLI is enabled:

Host cores	Recommended render threads	
	Single card	Voodoo 2 SLI
2	1	1
4	2	1
6 or 8	4	2
10 or more	4	4

- **SLI:** add a second Voodoo 2 card to the system, connected to the first one through a Scan Line Interleave (SLI) interface.
- **Dynamic Recompiler:** enable the Voodoo recompiler for faster emulation.

IBM 8514/A / XGA / PS/55 Display Adapter Graphics

Emulate an **IBM 8514/A**, **XGA** or **PS/55 Display Adapter** add-on graphics accelerator. The 8514/A is available for both MCA and ISA buses (emulating a generic clone card on the latter), while the other two are available for the MCA bus only.

The *Configure* buttons next to each card open a new window with settings specific to that card, such as the amount of video memory for the 8514/A and model type for the XGA.

> **Note**
>
> Pairing the 8514/A and XGA with each other or with video cards from **ATI** or **S3** may result in compatibility issues, as each card implements a set of 8514/A features.

Monitor

All settings in this tab, except for *Monitor EDID*, can be changed without a hard reset of the emulated machine.

VGA screen type

Select the VGA monitor type to emulate. Color, grayscale, amber, green and white phosphor monitors are available.

Grayscale conversion type

Select the color-to-grayscale conversion profile to use when a grayscale monitor is selected. BT.601, BT.709 and Average profiles are available.

Monitor EDID

Customize the emulated monitor's Extended Display Identification Data, which reports a model name, supported resolutions and other information to DDC2-compatible video cards.

The *Custom…* box allows for loading a custom EDID, which must be a binary file up to 256 bytes in size, or a plain text file containing an `edid-decode` report including the `edid-decode (hex):` section. The *Export…* button saves the default 86Box EDID to a binary file for customization.

Overscan

Add an overscan border around the display. Most video hardware types support this option.

Inverted VGA monitor

Emulate a VGA monitor with inverted colors. Only applicable when emulating VGA-compatible video hardware.

Change contrast for monochrome display

Optimize the contrast of monochrome CGA monitors for 4-color operation. Only applicable when emulating CGA-compatible video hardware with a monochrome monitor selected through the *Configure* button next to the *Video* box.

1.8.3 Input devices

The **Input devices** page contains settings related to the emulated machine's mouse, joysticks and other input devices.

Note

The **key bindings** previously available here are now in the *Preferences window*.

Keyboard

Select the keyboard type to emulate. This box only lists keyboards supported by the machine. Click to search for keyboards by name.

The *Configure* button opens a new window with settings specific to the selected keyboard type, such as the number of keys.

Mouse

Emulate a pointing device. Click to search for devices by name. The following types are supported:

- **Bus mouse:** ISA expansion card with a mouse interface. The I/O port and IRQ used by the card are configurable.
- **Serial mouse:** connected to the serial port of your choosing.
- **Serial tablet:** also connected to a serial port, but providing absolute (seamless) input on supported operating systems and/or applications.
- **PS/2 mouse:** connected to the PS/2 port. Only available on machines with a PS/2 mouse port.

The *Configure* button opens a new window with settings specific to the selected device type, such as the number of buttons, or the serial port for a serial mouse or tablet.

Note

- **Serial pointing devices** require the configured serial port to be enabled and set to *None* on the *Ports page's Serial ports tab*.
- The **middle mouse button** cannot be used to release mouse capture when emulating a pointing device with 3 or more buttons.

Joystick

Emulate one or more game port controller(s). Click to search for controllers by name. The following types are supported:

- Generic **joysticks**, **gamepads**, **flight yokes** and a **steering wheel**, all with a variable number of buttons and analog axes (two axes make one analog stick).
- **CH Flightstick Pro:** flight controller with four buttons, three or four axes and a POV hat.

- **Microsoft SideWinder Pad:** up to four gamepads, each with 10 buttons and a directional pad. Not compatible with standard game port joysticks; requires a driver in the emulated operating system.
- **Thrustmaster Flight Control System:** flight controller with four buttons, two or three axes and a POV hat.

Note

A **generic game port card** is emulated if the machine has no game ports (either built-in or as part of a sound card) to accomodate the selected controller. This generic card uses the default I/O ports 200-207h, configurable through ISA Plug and Play on supported machines. On IBM PCjr and PS/1 machines, the card uses port 201h only.

Joystick 1-4...

Configure the mappings for each emulated game port controller. The *Device* box lists all game controllers connected to the host, while the other boxes allow you to map each axis or button of the emulated controller to the host controller.

If you're not sure as to what axis or button numbers map to which sticks and buttons on the host controller, use the *Test* feature of Windows' *Game Controllers* control panel (`joy.cpl`) or another controller testing utility for your platform.

Note

XInput controllers are accessed through their DInput emulation mode at the moment.

1.8.4 Sound

The **Sound** page contains settings related to the emulated machine's audio hardware.

Parallel port sound devices such as the **Disney Sound Source** and **Covox Speech Thing** are not present on this page; they can be configured through the *Ports page's Parallel ports tab*.

General

Sound card #1-#4

Sound cards to emulate. Up to 4 different sound cards are supported. Only cards supported by the machine's expansion buses will be listed. Click ⊗ to search for cards by name or bus. On machines equipped with an on-board sound chip, the *Internal device* option for sound card #1 enables the on-board sound.

The *Configure* button opens a new window with settings specific to the selected sound card, such as the I/O ports, IRQ and DMA channels for ISA cards.

Emulation for the Yamaha OPL series of synthesizers (used by many of the emulated cards) is provided by a modified Nuked OPL2 Lite, Nuked OPL3 or ymfm library, per the *selection below*. MOS Technology 6581 SID emulation for the Innovation SSI-2001 and The Entertainer is provided by the reSIDfp component of the libsidplayfp library. General Instrument AY-3-8913 emulation for the Mindscape Music Board is provided by the Ayumi library.

The **OPL2Board** requires an external hardware device containing an OPL2 chip. See *External OPL audio* for more information.

Use FLOAT32 sound

Use the 32-bit floating point (instead of 16-bit integer) data type for audio output, which is less prone to clipping but may not work at all on some host systems. Try disabling this if you're getting no audio output from 86Box at all.

Audio output device

Select a host audio device to use for all audio produced by the emulated machine's PC speaker, *sound cards* and other sound hardware.

Note

This option does not apply to MIDI music sent to a software synthesizer through the *System MIDI* device, as these synthesizers are external to 86Box.

FM synth driver

Yamaha OPL2/3 emulation back-end to use. **Nuked** is the default, while **YMFM** may improve emulation performance at the cost of accuracy.

Note

YMFM is always used for OPL4 and OPM emulation on sound cards equipped with either of those synthesizers.

MIDI

MIDI Out Device

Device to output MIDI music to, for sound cards equipped with an external MIDI output. Click ⊗ to search for devices by name. The following devices are supported:

- **None:** don't output MIDI music.
- **FluidSynth:** a software soundfont synthesizer. Selecting a soundfont file is required; there will be no synthesizer output if no soundfont is configured.
- **Roland MT-32/CM-32L Emulation:** emulate one of these two Roland synthesizer modules. Emulation is provided by the Munt library.
- **Roland Sound Canvas:** emulate a Roland SC-55 synthesizer module. Setup instructions for the DOSBox Staging CLAP plugin to be concluded.
- **System MIDI:** output to a MIDI device on the host system, such as the Windows software synthesizer or a USB MIDI adapter.

The *Configure* button opens a new window with settings specific to the selected output device, such as the soundfont to use for *FluidSynth* and the host MIDI device to use for *System MIDI*. All output devices can be attached, removed or configured without a hard reset of the emulated machine.

MIDI In Device

Device to receive MIDI music from, for sound cards equipped with an external MIDI input. Click ⊗ to search for devices by name. The following devices are supported:

- **None:** don't receive MIDI music.
- **System MIDI:** receive from a MIDI device on the host system, such as a USB MIDI adapter.

The *Configure* button opens a new window with settings specific to the selected input device, such as the host MIDI device to use for *System MIDI*. All input devices can be attached, removed or configured without a hard reset of the emulated machine.

Standalone MPU-401

Emulate a standalone **Roland MIDI Processing Unit** ISA card, which allows for MIDI input and output without a MPU-401-equipped sound card, and for running the few applications which require *intelligent mode* capability.

The I/O port and IRQ can be configured through the *Configure* button.

1.8.5 Network

The **Network** page contains settings related to the emulated machine's network connectivity.

Adapter 1-4

Network interface cards to emulate. Up to 4 independent network cards are supported.

Mode

Network emulation mode to use on this card. See *Networking* for more information on these. The following modes are supported:

- **Null Driver:** emulate an empty network. All packets are dropped.
- **SLiRP:** creates a private network with a virtual router. Similar to the **NAT** mode on other emulators and virtualizers.
- **PCap:** connects directly to a host network adapter. Similar to the **Bridge** mode on other emulators and virtualizers.
- **VDE:** attaches to a virtual switch created by *VDE*. Only available on Linux and macOS hosts.
- **TAP:** creates or attaches to a virtual bridge through *TAP*. Only available on Linux hosts.
- **Local Switch:** creates or attaches to a *virtual switch* which automatically connects 86Box machines running on the same host and other hosts on the same network.

Note

If PCap mode is not listed, make sure PCap is *enabled on your host and a supported network connection is present*.

Adapter

Network card to emulate. Only cards supported by the machine's expansion buses will be listed. Click to search for cards by name or bus.

The *Configure* button opens a new window with settings specific to the selected network card, such as the MAC address for Ethernet cards, and the I/O port and IRQ for ISA cards.

Note

- On most Ethernet cards, the MAC address is partially configurable and always prefixed by an Organizationally Unique Identifier belonging to the card's manufacturer, such as `00:E0:4C` for Realtek; while on some generic cards, the full MAC is configurable.
- The **[LPT] Parallel Port Internet Protocol** network adapter requires a parallel port (LPT1 by default) not in use by *other parallel devices*; it no longer requires the separate PLIP parallel device from previous 86Box versions.

- The PLIP adapter is compatible with the DOS `plip.com` packet driver and the Linux `plip` driver (only with interrupts enabled); it is not compatible with the Windows Direct Cable Connection feature or any other parallel port networking implementations.
- Settings for the **[COM] Standard Hayes-compliant Modem** are described on *Networking*.

Options

Settings for the network emulation mode selected above. Only settings relevant to the selected mode will be shown.

- **Interface:** host network adapter to use in **PCap** mode.
- **VDE Socket:** virtual switch socket path to use in **VDE** mode. See the *VDE setup guide* for more information.
- **TAP Bridge Device:** virtual bridge name to use in **TAP** mode. See *TAP* for more information.
- **Shared secret:** set a password to isolate networks in **Local Switch** mode. See *Shared secret* for more information.
- **Hub Mode:** listen to every packet going through the network in **Local Switch** mode. See *Hub Mode* for more information.

1.8.6 Ports (COM & LPT)

The **Ports (COM & LPT)** page contains settings related to the emulated machine's I/O ports.

> **Note**
>
> The **serial port passthrough** options previously available here are now part of the **Serial Passthrough**, **Named Pipe** and **Virtual Console** *serial devices*.

Parallel ports

Internal LPT ECP DMA

ISA DMA channel number to use for the on-board parallel port's Extended Capabilities Port mode. Only available on machines with physical DMA configuration jumpers for an on-board ECP-capable parallel port.

LPT1-4

The check box (left) enables emulation of the corresponding parallel port. Any ports not provided by the machine's motherboard will be emulated as generic ISA or VLB parallel cards.

> **Note**
>
> The LPT4 port is not widely supported. It is located at I/O port 268h.

The dropdown (middle) selects an emulated device to connect to the parallel port. Click ⊗ to search for devices by name. The following devices are supported:

- **None:** no device connected.
- **Disney Sound Source:** sound device with a resistor ladder DAC (digital-to-analog converter) and FIFO, supported by many games.

- **LPT DAC / Covox Speech Thing:** sound device with a simple resistor ladder DAC, supported by many games, demos and trackers.
- **Stereo LPT DAC:** stereo version of the **LPT DAC**, using the *Strobe* pin to select the active output channel.
- **FTL Sound Adapter:** enhanced version of the **Stereo LPT DAC**, with additional circuitry for Dungeon Master.
- **SiliconSoft SoundJr:** enhanced version of the **Stereo LPT DAC**, with volume control through the printer control pins.
- **AdLib-on-LPT / Creative Music System-on-LPT / Tandy-on-LPT:** sound devices providing standard ISA sound chips through the parallel interface.
  - Special drivers are required for games and applications to access these sound chips through the parallel port.
- **Generic Text Printer:** simple printer capable of outputting text only.
  - Printed documents are saved as .txt files in the `printer` subdirectory found in the emulated machine's directory.
- **Generic ESC/P 2 Dot-Matrix:** EPSON ESC/P 2-compatible printer.
  - Printed pages are saved as .png files in the `printer` subdirectory found in the emulated machine's directory.
  - The printer type, paper size and print quality (draft quality uses a dot-matrix font and letter quality uses TrueType fonts) can be configured through the *Configure* button.
  - Use these printer drivers according to the selected printer type for best results:
    * *EX-1000* (in order): EPSON EX-1000, EX-800, FX-286, FX-185, FX-85, JX-80, FX-100+, FX-80+, FX-100, FX-80, HS-80, MX-100 Type III, MX-82 F/T Type III, MX-80 F/T Type III, MX-80 Type III, MX-100, MX-82, MX-80 F/T Type II, MX-80 Type II, MX-80
    * *ESC/P 2*: EPSON LQ-2500
  - If the emulation speed decreases drastically during printing, disable ECP/EPP mode in the emulated machine's BIOS setup.
- **Generic PostScript Printer:** PostScript-compatible printer with PDF output.
  - Printed documents are saved as .ps files in the `printer` subdirectory found in the emulated machine's directory. These files are automatically converted to .pdf once printing is completed; this conversion can be disabled by setting *Language* to *Raw* through the *Configure* button.
  - The original .ps files may remain in the directory if PDF conversion fails, or (on Windows x64 hosts) if the included `gsdll64.dll` file is missing from the 86Box directory. PDF conversion is not available on Windows ARM hosts.
  - Use the generic PostScript printer driver provided by your operating system; note that generic drivers may support grayscale only.
  - Windows 95 and newer do not have a generic PostScript driver; use the **Apple LaserWriter IIf** driver for grayscale, or the **Apple Color LW 12/660 PS** driver for color.
- **Generic PCL Printer:** HP Printer Command Language-compatible printer.
  - Printed documents are saved as .pcl or .pxl files in the `printer` subdirectory found in the emulated machine's directory.
  - The GhostPCL library required to convert output files to .pdf is not included with 86Box due to a license incompatibility. Set *Language* to *Raw* through the *Configure* button to remove the warning displayed on startup.
  - The following PCL standards can be selected through the *Configure* button:

    * *PCL 5e* (enhanced): introduced in 1992 with HP LaserJet 4;
    * *PCL 5c* (color): introduced in 1992 with HP PaintJet 300XL and HP Color LaserJet;
    * *HP-RTL* (Raster Transfer Language): a subset of PCL;
    * *PCL 6* (PXL): introduced in 1995.

- **Named Pipe:** create or connect to a named pipe on the host system.
  - The same options as the *Named Pipe serial device* apply here, with an added option for the parallel cable type.
  - The *Unidirectional / LapLink* cable transmits 8-bit and receives 5-bit data, following the simple cross-over wiring supported by many PC-to-PC connection software, including MS-DOS Interlnk, Windows Direct Cable Connection and PLIP.
  - The *Bidirectional* cable transmits and receives 8-bit data with no control/status lines.
  - The *DirectParallel FAST* cable is backwards compatible with the LapLink cable and adds support for faster bidirectional and ECP modes on Windows Direct Cable Connection.
- **File:** write all outgoing data to a file on the host system.
  - Raw data is written to the file, without the document separation and page formatting performed by the **Generic Text Printer**.
- **Loopback Plug:** a parallel plug with pins wired together in a specific manner, for use with diagnostic software.
  - Different wirings can be selected through the *Configure* button.

The *Configure* button (right) opens a new window with settings specific to the selected device, such as the output file format for printers. All devices can be attached and some can be removed or configured without a hard reset of the emulated machine, as long as the port is not claimed by *PLIP*.

Serial ports

COM1-4

The check box (left) enables emulation of the corresponding serial port. Any ports not provided by the machine's motherboard will be emulated as generic ISA or VLB serial cards.

The dropdown (middle) selects an emulated device to connect to the serial port. Click ⊗ to search for devices by name. The following devices are supported:

- **None:** no device connected.
- **Serial Passthrough:** connect to a serial port on the host system.
  - The host port's parameters (baud rate, parity, data bits and stop bits) are automatically configured to match the emulated port's parameters, unlike in previous 86Box versions which required manual configuration in the passthrough settings.
- **Named Pipe:** create or connect to a named pipe on the host system.
  - On Windows hosts, *Auto* mode creates or connects to the pipe depending on whether or not it already exists, *Server* mode always creates the pipe and *Client* mode always connects to an existing pipe. The `\\.\pipe\` prefix is optional.
  - On Linux and macOS hosts, two pipes are created (adding `.in` and `.out` suffixes to the *Pipe path*) for bidirectional communication:

Mode	`.in` pipe function	`.out` pipe function
Auto	Same as Client mode if an application is already reading from this pipe; otherwise, same as Server mode	Opposite direction of `.in` pipe
Server	Write data to emulated machine	Read data from emulated machine
Client	Read data from emulated machine	Write data to emulated machine

  - On Linux and macOS hosts, the *Pipe path* can also point to a character device (such as a pseudoterminal created by the **Virtual Console** device on another emulated machine), in which case the *Auto* and *Client* modes will connect to that device instead.

- **File:** write all outgoing data to a file and/or read incoming data from a file on the host system.
  - If *Append to file if it exists* is unchecked, the outgoing data file is cleared every time the device is connected, including when the emulated machine is started or hard reset.
- **Virtual Console:** connect to a terminal on the host system, in one of multiple modes.
  - On Windows hosts, this device always connects to a Command Prompt window (limited to one per emulated machine). The modes below are only available on Linux and macOS hosts.
  - *Use standard input/output* connects to stdin and stdout, available when starting the machine directly from a terminal through the `-P/--vmpath` command line option.
  - *Create pseudoterminal* creates a PTY device, connects to it and displays its path when the machine is started.
    * The **Named Pipe** device can be used to connect another machine to this pseudoterminal by specifying its path as the pipe path.
  - *Start terminal emulator* connects to the system's default terminal emulator.
    * On Linux hosts, `xdg-terminal-exec` or `x-terminal-emulator` is used; if neither of those is available, a suitable terminal is guessed.
    * On macOS hosts, the Apple Terminal app is always used; note that its default settings keep the terminal window open after the port is disconnected.
  - *Run custom command* creates a PTY device, connects to it and executes the configured *Custom command*.
    * The default command (leave blank to restore it) connects to a new GNU Screen session, which runs in the background and can be attached to by running `screen -r` on a terminal.
    * Variables `$PTY` (path to the PTY device), `$VMNAME` (machine *display name*), `$PORT` (emulated port name such as `COM1`) and `$PIPECMD` (command used in *Start terminal emulator* mode) are passed to the command.
- **Loopback Plug:** a serial plug with pins wired together in a null-modem configuration, for use with diagnostic software.

The *Configure* button (right) opens a new window with settings specific to the selected device, such as the host serial port to use for passthrough. All devices can be attached, removed or configured without a hard reset of the emulated machine, as long as the port is not claimed by a *serial mouse* or *modem*.

1.8.7 Storage controllers

The **Storage controllers** page contains settings related to the emulated machine's disk drive controllers.

Note

The **Vision Systems LBA Enhancer** previously available here is now an ISA ROM card, which can be enabled through the *Other peripherals page*.

General

Floppy drive controller

Floppy disk drive controller card to emulate. Only cards supported by the machine's expansion buses will be listed. Click ⊗ to search for cards by name or bus. Selecting a controller is not required, unless you wish to use one of the add-on controllers for adding high-density 1.44M floppy support to XT machines.

The BIOS option ROM address used by the selected controller can be configured through the *Configure* button.

CD-ROM controller

Standalone CD-ROM controller card to emulate. Only cards supported by the machine's expansion buses will be listed. Click ⊗ to search for cards by name or bus. These cards provide vendor-specific CD-ROM interfaces beyond *ATAPI (IDE)* or *SCSI*.

The I/O port used by the selected controller can be configured through the *Configure* button.

Hard disk controllers

MFM, RLL, ESDI or IDE hard disk drive controller cards to emulate. Up to 4 controller cards are supported. Only cards supported by the machine's expansion buses will be listed. Click ⊗ to search for cards by name or bus. On machines equipped with an on-board disk controller, the *Internal device* option for controller #1 enables the on-board controller; this is not required for machines with on-board IDE.

The *Configure* buttons open a new window with settings specific to the corresponding controller card, such as the I/O port and IRQ for ISA cards.

Note

The **tertiary and quaternary IDE controllers** are now selectable here, replacing the previous separate options for each. These controllers are not Plug and Play compliant by default, potentially requiring manual configuration of the emulated operating system, and may not be bootable; see *Tertiary and quaternary IDE* for more information.

Cassette

Enable IBM cassette tape emulation. Only available on machines with a cassette port. The cassette deck can be controlled through the *status bar* or *Media menu*.

SCSI controllers

SCSI host bus adapter cards to emulate. Up to 4 SCSI cards are supported. Only cards supported by the machine's expansion buses will be listed. Click ⊗ to search for cards by name or bus.

The *Configure* buttons open a new window with settings specific to the corresponding SCSI card, such as the I/O port and IRQ for ISA cards.

1.8.8 Hard disks

The **Hard disks** page contains settings related to the emulated machine's fixed disks.

Hard disk list

All hard disks attached to the emulated machine are listed, with the following information:

- **Bus:** storage bus the disk is attached to, as well as the disk's bus channel or ID. These can be changed through the *Bus* and *Channel/ID* boxes below the list.
- **File:** path to the disk image file.
- **Geometry:** disk size in cylinders, heads, sectors and MiB, respectively.
- **Model:** *emulated model profile* for the disk.

Model / Audio

The *Model* box below the hard disk list determines the **disk model** to emulate. Click ⊗ to search for models by name or bus. Model profiles adjust the disk's identification data, as well as its performance according to rotation speed, physical layout and cache size. Generic profiles adjust performance to match an average period-correct disk, while the **RAM Disk** profile runs the disk as fast as the host can manage.

Additionally, the *Audio* box allows for emulating the mechanical sounds of a real hard disk drive. A list of drive models to choose from is provided, according to the selected model's spindle speed; the *None* option disables these sounds.

Note

The *Audio* option is only available if the *asset pack* is installed.

Adding a new disk

The *New...* button opens a new window allowing you to create an existing hard disk image file.

- **File name:** where to save the disk image file. See *Hard disk images* for a list of supported image formats.
- **Cylinders/Heads/Sectors:** CHS parameters for the disk image. These boxes control the *Size (MB)* box below.
- **Size (MB):** the disk image's size in MB. This box controls the *Cylinders*, *Heads* and *Sectors* boxes above. There are limits to how big a hard disk image can be; see *Hard disk size limits* for more information.
- **Bus:** storage bus to attach the disk to.
- **Channel/ID:** where to attach the disk on the selected storage bus. Channels/IDs that are already in use cannot be selected.
  - On IDE disks, the first number corresponds to the IDE channel, and the second number corresponds to the Master/Slave position:

Value	Channel	Device
0:0	Primary	Master
0:1	Primary	Slave
1:0	Secondary	Master
1:1	Secondary	Slave
2:0	Tertiary	Master
2:1	Tertiary	Slave
3:0	Quaternary	Master
3:1	Quaternary	Slave

- On SCSI disks, the first number corresponds to the controller's index, starting from 0 and following the order of: on-board SCSI controllers if present, then *sound cards* with SCSI if present, then *configured SCSI controllers*; the second number is the SCSI ID within that controller:

Value	Controller	SCSI ID
0:00	Controller 1	0
…		…
0:15		15
1:00	Controller 2	0
…		…
1:15		15
2:00	Controller 3	0
…		…
2:15		15
3:00	Controller 4	0
…		…
3:15		15

- On MFM/RLL, XTA and ESDI disks, the second number is 0 for the first drive on the controller, and 1 for the second drive.

Note

If the disk is attached to a channel or controller that doesn't exist, such as the tertiary IDE channel with no tertiary IDE controller present, it will be effectively disabled.

- **Model:** *model profile* to use for the disk. A list of drive models to choose from is provided; click ⊗ to search for models by name or bus.
- **Image Format:** file format to use for the disk image.
- **Block Size:** size of each dynamic data block in a dynamic or differencing VHD image. The default 2 MB is ideal in most cases.

Press the *OK* button to create the disk image file, or *Cancel* to close the window without adding the disk.

Adding an existing disk

The *Existing…* button opens a similar window to the *New…* button, except that it lets you select an existing disk image file. The CHS parameters are guessed from the image's file size, or the file header if the image is of a format which contains a header.

After selecting the image file and checking if the parameters are correct, select the *Bus* and *Channel/ID* for the hard disk and press *OK* to add it. Press *Cancel* to close the window without adding the disk.

Using physical disks

Real disks connected to the host system can be attached to the emulated machine, as long as they meet the *131071 MB size limit and any other limits set by the emulated hardware*. Use the *Existing…* button and set *File name* to:

- **Windows:** `\\.\PhysicalDriveX` where `X` is the disk number displayed in the Disk Management (`diskmgmt.msc`) tool.
- **Linux:** full path to the disk's block device, `/dev/sdb` for example.

The *Cylinders*, *Heads*, *Sectors* and *Size (MB)* parameters are set automatically when *OK* is pressed.

Note

- You must have the correct permissions to access the raw physical disk. This entails running 86Box as administrator on Windows and adjusting block device permissions on Linux.
- Any partitions on the disk must be unmounted to prevent data corruption. On Windows, use Disk Management to unmount the disk by right-clicking it and selecting *Offline* (not available on removable drives) or by removing the drive letters from all of its partitions.

Removing a disk

Select a disk on the list and press *Remove* to remove it.

1.8.9 Floppy & CD-ROM drives

The **Floppy & CD-ROM drives** page contains settings related to the emulated machine's base removable storage drives.

Floppy drives

Up to four floppy disk drives can be attached to the emulated machine, although not all machines provide BIOS support for more than two drives. The following settings apply to the selected drive:

- **Type:** floppy drive to emulate. Click ⊗ to search for types by name. Some types have special properties and should only be used in very specific applications:
  - **5.25" 1.2M 300/360 RPM**, **3.5" 1.44M 300/360 RPM** and **3.5" 2.88M 300/360 RPM**: "3-mode" drives, which are capable of reading 360K 5.25" or NEC PC-98 3.5" disks if the emulated machine's BIOS supports 3-mode operation.
  - **3.5" 1.25M PC-98:** NEC PC-98 drive, which is 3-mode and inverts the polarity of the Density Select pin.
  - The special **PS/2** drive types have been removed, as the required inversion of the Density Select pin is now performed automatically on IBM PS/2 machines.
- **Turbo timings:** run the drive mechanism as fast as possible. This decreases access times and makes some incorrectly-dumped floppies readable, but may cause issues with some operating systems and applications.
- **Check BPB:** use the DOS BIOS Parameter Block when determining the physical media format for a floppy image on this drive. See *Floppy disk detection* for more details.
- **Audio:** emulate the mechanical sounds of a real floppy drive. A list of drive models to choose from is provided; the *None* option disables these sounds.

Note

- Disabling *Check BPB* may be required in order to access UNIX/Linux installation floppies or other non-DOS disks, as outlined on *Floppy disk detection*.
- The *Audio* option is only available if the *asset pack* is installed.

Floppy disk images can be inserted and removed through the *status bar* or *Media menu*.

CD-ROM drives

Up to eight CD-ROM or DVD-ROM optical disc drives can be attached to the emulated machine. The following settings apply to the selected drive:

- **Bus:** storage bus to attach the drive to. *ATAPI (IDE)*, *SCSI* and *Panasonic/MKE* interfaces are supported.
- **Channel/ID:** where to attach the drive on the selected storage bus. See *Adding a new disk* for more information.
- **Speed:** maximum transfer speed for the drive. Up to 72x is supported.
- **Type:** CD-ROM drive model to identify as. A list of drive models (suitable for the selected bus) to choose from is provided; click ⊗ to search for models by name or speed.
- **Use EDC/ECC emulation:** enable checking of CD-ROM error correction data. Can be changed without a hard reset of the emulated machine.

Note

- Only models with DVD support (indicated by the DVD icon on the *status bar*) are able to read discs or host folders larger than 1 GB, unlike in previous 86Box versions.
- Some emulated machines have manufacturer restore discs locked to a specific drive model.
- The **86Box 86B_CD 1.00** model emulates early versions of the ATAPI and SCSI standards as required by older drivers.
- Disabling *Use EDC/ECC emulation* may be required for some copy-protected discs.

CD-ROM / DVD-ROM disc images or host drives can be inserted and removed through the *status bar* or *Media menu*.

1.8.10 Other removable devices

The **Other removable devices** page contains settings related to the emulated machine's additional removable storage drives.

MO / Removable disk / Tape drives

Up to four Magneto-Optical, four removable disk drives and four tape drives can be attached to the emulated machine. The following settings apply to the selected drive:

- **Bus:** storage bus to attach the drive to. ATAPI (IDE) and SCSI are supported.
- **Channel/ID:** where to attach the drive on the selected storage bus. See *Adding a new disk* for more information.
- **Type**: drive model to identify as. A list of drive models to choose from is provided; click ⊗ to search for models by name. Each model supports different types of media, while the *86BOX* model supports all types.

MO, removable disk and tape images can be inserted and removed through the *status bar* or *Media menu*.

1.8.11 Other peripherals

The **Other peripherals** page contains settings related to disk drive controllers, memory expansions and other expansion cards.

General

ISA RTC

Emulate an ISA real-time clock card, for machines without an integrated real-time clock. Click ⊗ to search for cards by name.

The I/O port and/or IRQ used by the selected controller can be configured through the *Configure* button.

ISABugger

Emulate an **ISABugger** debugging interface card, equipped with two hexadecimal displays and two LED banks, all controlled by the emulated machine. See *ISABugger* for documentation on the card's features.

POST card

Emulate a diagnostic POST card, which displays POST code values issued by the emulated machine's BIOS on the status bar. See *Status bar: POST card* for more information.

The POST card will automatically use the correct diagnostic I/O ports for the emulated machine:

Port	Machine types

`0x10`
`0x11`
`0x12`
	IBM PCjr

`0x60`	IBM XT
`0x80`	IBM AT, clones and the XT-based Xi 8088
`0x84`	Early Compaq
`0xE0` `0xE4`	Dell (4-character text display after the port `0x80` hex display)
`0x190`	IBM PS/1 and PS/2 not based on the Micro Channel Architecture
`0x378`	Olivetti
`0x680`	Micro Channel Architecture
`0x5080`	ASUS ISA-486C

Note

Some operating systems and applications use port `0x80` (which is shared with the POST card on most machines) for other purposes. If you notice the POST code display is flickering and the emulation speed has decreased drastically, try disabling the POST card.

86Box Unit Tester

Emulate a special device for software in the emulated machine to control 86Box for testing purposes. The *Configure* button provides settings specific to this device.

Note

Documentation to be concluded.

Novell NetWare 2.x Key Card

Emulate the hardware license key card required by Novell NetWare 2.0a among other versions.

The serial number displayed in software can be configured through the *Configure* button.

ISA Memory Expansion

Add up to four ISA-based memory expansion cards, for machines which support memory expansion through the ISA bus. Click ⊗ to search for cards by name.

The memory start address and size for each card can be configured through its respective *Configure* button.

ISA ROM Cards

Add up to four ROM expansion cards, which can load a predefined or custom BIOS option ROM directly into the emulated machine. Click ⊗ to search for cards by name.

The **Vision Systems LBA Enhancer** ROM can help work around the *hard disk size limits* on many older machines. Hard disks up to 8055 MB (16367 cylinders, 16 heads, 63 sectors) in size are supported; they must be manually configured to **8 cylinders, 8 heads and 8 sectors** on the machine's BIOS setup utility for the Enhancer to handle them.

The base address for each card, as well as the ROM file and size for a generic card, can be configured through its respective *Configure* button.

1.9 Preferences

The **Preferences** window allows you to configure options related to 86Box as a whole, which are saved system-wide and apply to all emulated machines.

Note

Any changes made to the preferences in 86Box versions older than 5.0 (saved per-machine) have been reset.

1.9.1 Emulator

Language

Select a language for the 86Box user interface. Click ⊗ to search for languages by their English name. The *Default* button resets back to the system language.

Select media images from program working directory

Start the first file open/save prompt on the emulated machine's directory. This option is particularly useful for macOS users.

Ask for confirmation before saving settings / quitting / hard resetting

Enable confirmation messages for the specified actions. These options can reenable confirmations after they have been disabled through the *Don't show this message again* box.

Color scheme

Select a visual style for the 86Box user interface. *System* uses the operating system's global preference if possible.

1.9.2 Input

Mouse sensitivity

Adjust the emulated mouse's tracking sensitivity. The *Default* button resets back to the default sensitivity.

Inhibit multimedia keys

Disable the passthrough of multimedia keys such as `Volume Up` and `Play/Pause` to the emulated machine.

1.9.3 Key bindings

View and change keyboard shortcuts for common emulator actions. Most listed actions are also available on the *menu bar* or the *toolbar*; see the respective pages for more information.

The *Clear binding* button removes the shortcut associated with the selected action, and the *Bind* button allows for entering a new shortcut.

Note

The `F8+F12` key combination used for releasing mouse capture in previous 86Box versions can no longer be configured as a shortcut.

CHAPTER

TWO

EMULATED HARDWARE

2.1 Machine-specific notes

This section contains important notes related to specific machine models emulated by 86Box.

2.1.1 8088

IBM PC

- The 1981 and 1982 variants correspond to the earlier 16KB-64KB and later 64KB-256KB revisions of the motherboard, with different BIOS versions and memory size limits.
  - Those limits apply to on-board RAM; more can be added through *ISA memory expansion* cards.

IBM PCjr

- Some applications may shift the display slightly to one side due to unconventional use of the PCjr video hardware. Unchecking the **Apply overscan deltas** option accessible through the internal video's *Configure button* can help bring the display back into position.
- Hard disks are not supported, as a PCjr-compatible hard disk controller is not emulated by 86Box.

IBM XT

- The 1982 and 1986 variants correspond to the earlier 64-256KB and later 256-640KB revisions of the motherboard, with different BIOS versions and memory size limits.
  - Those limits apply to on-board RAM; more can be added through *ISA memory expansion* cards.

2.1.2 80286

IBM AT

- On-board RAM is limited to 512 KB; more can be added through *ISA memory expansion* cards.
- The IBM Personal Computer Diagnostics disks are not Y2K-compliant and will produce a *0152 ERROR - SYSTEM BOARD* code if *time synchronization* is enabled. This code can be cleared by disabling time synchronization, then wiping NVRAM *through the VM manager* or by deleting `ibmat.nvr` from the machine's `nvr` directory.

IBM XT Model 286

- On-board RAM is limited to 640 KB; more can be added through *ISA memory expansion* cards.

GRiD GRiDcase 1520

- The BIOS is locked to specific Conner IDE hard disk models. Any hard disks must be set to the **Conner CP3024**, **CP3044** or **CP3104** *model profiles*.
- The Yamaha V6366 video chip is not emulated by 86Box. An **IBM CGA** set to amber monochrome mode (through the *Configure button*) is recommended as an approximation.

2.1.3 i386SX

Amstrad MegaPC

- The BIOS does not configure itself on first boot or after clearing CMOS; the machine will not work properly until an automatic configuration is performed by pressing `F9` on the BIOS setup's main *Setup* menu, then saving with `F10` and exiting with `Esc`.

2.1.4 i486

Intel Classic R/R Plus (Monsoon)

- The messages for entering the BIOS setup and skipping the memory test are not displayed by default. To enter the setup utility, press `F1` when the number *135* or an error is displayed. To skip the memory test, press `Space`. Both messages can be enabled through the *POST Setup Prompt* and *POST Memory Test Prompt* options on page 1 of the BIOS setup respectively.
- The internal IDE hard disk controller is disabled by default. It can be enabled through the *Onboard IDE* option on page 1 of the BIOS setup.

Zida Tomato 4DPS

- Floppy drive support is completely disabled by default. It can be enabled through the *Onboard FDD Controller* option of the *Chipset Features Setup* menu on the BIOS setup; the floppy drives themselves must also be configured in the *Standard CMOS Setup* menu.

2.1.5 Socket 7

MSI MS-5119

- 86Box versions prior to 4.0.1 used BIOS version *A37E*, which has PS/2 mouse issues. The fixed *A37EB* BIOS is not applied automatically to existing setups; it can be applied by wiping NVRAM *through the VM manager* or deleting `ms5119.bin` from the machine's `nvr` directory.

ASUS P/I-P65UP5 (C-P55T2D)

- Modular motherboard, consisting of a **P/I-P65UP5** baseboard and one of the following CPU cards:
  - **C-P55T2D:** Socket 7 with Intel 430HX northbridge;
  - **C-P6ND:** Socket 8 with Intel 440FX northbridge;
  - **C-PKND:** Slot 1 with Intel 440FX northbridge.
- While the northbridge depends on the selected CPU card, the southbridge always remains the Intel PIIX3, as it is located on the baseboard.

- The real CPU cards support dual CPUs. As 86Box does not emulate multiprocessing, only a single CPU will be present.
- Due to a lack of I/O APIC emulation at the moment, 86Box will patch the MultiProcessor Specification tables out of RAM during boot, so that operating systems will not hang or exhibit other erratic behavior due to the missing I/O APIC.

NEC Mate NX MA23C

- Accessing the BIOS setup utility takes an additional step. Press `F2` during the NEC logo screen and a Japanese message will appear; once a different message appears, press (right arrow) to enter the setup utility.
- The first setup option below the date and time can be used to change the BIOS language to English.

Siemens Simatic OP47

- Equipped with SMC FDC37C932FR and Intel 82091AA Super I/O chips driving four serial (COM1-COM4) and one parallel (LPT1) ports.

2.1.6 Socket 8

ASUS P/I-P65UP5 (C-P6ND)

See: *ASUS P/I-P65UP5 (C-P55T2D)*

2.1.7 Slot 1

ASUS P/I-P65UP5 (C-PKND)

See: *ASUS P/I-P65UP5 (C-P55T2D)*

A-Trend ATC6310BXII

- Equipped with the obscure SMSC Victory66 southbridge instead of the regular Intel PIIX4E.
  - The Victory66 has faster IDE - up to Ultra ATA/66 as opposed to the PIIX4E's Ultra ATA/33 - and a different USB controller.
  - Drivers for Windows 95, 98, Me and 2000 are available here. Windows XP, Vista and 7 include drivers out of the box.

NEC Mate NX MA30D/23D

See: *NEC Mate NX MA23C*

2.1.8 Slot 1/2

Freeway FW-6400GX

- The maximum amount of RAM is limited to 2032 MB due to a BIOS bug with 2048 MB.
- ACPI is disabled by default. It can be enabled through the *ACPI Aware O/S* option of the *Power Management Setup* menu on the BIOS setup.
- Once enabled, ACPI *does not work correctly* if a non-Intel CPU is selected.

2.1.9 Slot 2

Gigabyte GA-6GXU

- The BIOS display will corrupt itself during the memory test if the maximum of 2048 MB RAM is selected. This is a visual glitch which does not otherwise negatively impact the machine.

2.1.10 Socket 370

A-Trend ATC7020BXII

See: *A-Trend ATC6310BXII*

AEWIN AW-O671R

- Equipped with dual Winbond W83977EF Super I/O chips driving four serial (COM1-COM4) and two parallel (LPT1-LPT2) ports.
  - The I/O ports and IRQs used by all these ports can be configured in the BIOS setup.
- ACPI is disabled by default, unlike other machines with AwardBIOS v6.00PG. It can be enabled through the *ACPI function* option of the *Power Management Setup* menu on the BIOS setup.

ASUS CUBX

- Equipped with an on-board CMD PCI-0648 IDE controller on the *tertiary and quaternary channels*, on top of the PIIX4E southbridge controller on the primary and secondary channels.

Samsung CAIRO-5 (MSI MS-6309)

- The BIOS is in Korean by default. Press `F2` during the Samsung logo screen to enter the setup utility, where the first option can be used to change the BIOS language to English.
- *Broken ACPI* causes some operating systems such as Windows 2000 and Linux to reboot during startup. *Disabling ACPI at operating system level* is required, as the BIOS lacks an option for it.

2.1.11 Miscellaneous

Microsoft Virtual PC 2007

- This machine loads the AMIBIOS 8 ROM from Virtual PC 2007 on 86Box. It does not use the virtualization engine or any other components from Virtual PC.
- Virtual PC's special 8 MB video card, WDM sound card and Guest Additions are not emulated by 86Box.

2.1.12 Footnotes

Broken ACPI

Some machines may have faulty or otherwise incomplete Advanced Configuration and Power Interface implementations in their BIOSes, symptoms of which include:

- Windows 2000 and higher will install the "Standard PC" HAL, which does not enable ACPI features such as soft power off and sleep mode;
- Booting an existing Windows installation with the ACPI HAL will result in a STOP 0x000000A5 blue screen;
- Booting Windows Vista or 7 (which require ACPI) will also result in a STOP 0x000000A5 blue screen, or a Windows Boot Manager 0xc0000225 error.

There is no solution to this issue outside of disabling ACPI, as none of the affected machines ever received a BIOS update to fix it. ACPI can be disabled through the BIOS setup on many machines, or at operating system level if that is not an option:

- **Windows 2000 or XP:** during installation, press `F7` when the *Press F6 if you need to install a third party SCSI or RAID driver. . .* message appears; this disables ACPI even though no indication is displayed on screen.
- **Linux:** add `acpi=off` to the kernel command line.

2.2 Device-specific notes

This section contains important notes related to various devices emulated by 86Box.

2.2.1 Display

ATI Mach64GX

- After creating an emulated machine or *wiping NVRAM*, the card must be configured using the `INSTALL.EXE` DOS utility included with Windows driver packages, otherwise the drivers for many operating systems will exhibit issues.

Cirrus Logic GD5420

- The driver included with Windows NT 3.1 does not properly support the extra video modes that are unlocked when the card is configured with 1 MB of video memory.

2.3 Keyboard

This section outlines keyboard behavior specific to different host operating systems, real keyboards and emulated machines.

2.3.1 Host systems

Windows

- Not all multimedia function keys can be passed through to the emulated machine due to a Windows limitation.
- System-wide key remapping through the **Scancode Map** registry key (as performed by applications such as SharpKeys) is fully supported.

Linux

- The xkbcommon library is used to accurately map physical keys to the emulated keyboard on both X11 and Wayland. When compiling 86Box from source, make sure the development files for `libxkbcommon` and `libxkbcommon-x11` are installed, as this is an optional build component.

macOS

- Apple keyboards with **European or other ISO layouts** may have *the key below Esc* and *the key to the right of Left Shift* switch places in the emulated machine, due to a hardware quirk in many of those keyboards (both internal and external) and a limitation in the way macOS corrects it.
- The `Num =` key is only usable in operating systems which recognize that key on Microsoft PS/2 multimedia keyboards.
- Mac special keys are mapped to their PC equivalents where possible:

Emulated key	Host key
Windows	Command
Alt	Option
Print Screen	F13
Scroll Lock	F14
Pause	F15
Insert	Command + Fn + Delete (MacBook and tenkeyless keyboards) Command + Forward Delete (full size keyboards)
Num Lock	Clear

2.3.2 Special keys

Some keyboards provide additional function or otherwise special keys on top of the standard PC layout. Those keys are mapped to ones present on modern keyboards within reason.

Multimedia keys

Passthrough of Microsoft multimedia function keys such as `Volume Up` and `Play/Pause` can be disabled through the *Inhibit multimedia keys option in the Preferences window's Input page*.

Olivetti

The Olivetti M series special keys are mapped as such:

Emulated key	Host key
CLEAR	Page Up
BREAK	Page Down
SCR PRT	Print Screen
HELP	Menu
00	Left Windows
F13	Insert
F14	Home
F15	Del
F16	End
F17	Right Alt
F18	Right Win

Toshiba

The Toshiba T1000 series function keys can be accessed by holding Right Alt or Right Ctrl:

Function	Right Alt/Ctrl +
Show/hide numeric keypad overlay	Num Lock
Change internal display font	Right
Use internal display	Home
Use external display	End
Turbo mode on (T1200)	Page Up
Turbo mode off (T1200)	Page Down
Show/hide pop-up window (T1200)	Print Screen

2.4 Disk images

86Box supports a large variety of disk image formats for the emulated storage drives.

2.4.1 Hard disk images

Supported formats:

Format	File extension	Notes
Raw image	Many	Extensions include: .hdd .ima .img
Japanese FDI	.hdi	
Extended HDI (HDX)	.hdx	
Virtual Hard Disk	.vhd	Fixed, Dynamic and Differencing VHDs are supported through the MiniVHD library.

Hard disk size limits

There are limits to how big of a hard disk an emulated machine can accept. Such limits will vary depending on the machine's BIOS. The table below lists all important limits applicable to the IDE bus:

Limit	Disk size	Cylinders	Heads	Sectors
20-bit CHS	504 MB	1024	16	63
12-bit cylinder	2015 MB	4095	16	63
ECHS translation	4032 MB	1024	128	63
Revised ECHS	7560 MB	1024	240	63
LBA translation	8064 MB	1024	256	63
16-bit cylinder	32255 MB	65535	16	63
28-bit LBA	131071 MB	65536	16	256

Disk overlay software such as the *Vision Systems LBA Enhancer* or *Ontrack Disk Manager* can work around BIOS limits and allow booting of larger IDE hard drives, with the same caveats as using such software on a real machine. The maximum supported disk image size for IDE or SCSI is 131071 MB.

2.4.2 Floppy disk images

Supported formats:

Format	File extension	Notes
Raw image	Many	Extensions include: .bin .dsk .flp .hdm .ima .img .vfd .xdf
86F	.86f	Once loaded, any image can be converted to 86F through the *status bar* or *Media menu*.
CopyQM	.cq / .cqm	
DiskDupe	.ddi	
EZ-DisKlone plus	.fdf	
Formatted Disk Image	.fdi	Read only.
HxC MFM	.mfm	Read only.
ImageDisk	.imd	
Japanese FDI	.fdi	
PCjs JSON	.json	Read only. v2 format only, as the previous v1 is no longer in circulation.
Teledisk	.td0	Read only.

Floppy disk detection

86Box determines the physical media format (sides, tracks per side, sectors per track, bytes per sector) of a floppy disk image through the following methods:

1. Image file header data - not applicable for **Raw** and **DiskDupe** formats;
2. DOS BIOS Parameter Block data within the image;
3. If all else fails, a guess is made based on the image file's size.

The BIOS Parameter Block detection method may behave incorrectly with non-DOS floppy disks. Installation floppies for UNIX and Linux are common examples of non-DOS disks. Disabling *Check BPB* is strongly recommended when accessing these, as an inaccurate BPB detection may result in read errors, data corruption and other issues.

Note

When using a **Raw** image of a non-DOS floppy with Check BPB disabled, make sure the image file is not truncated (smaller than its media size), otherwise incorrect behavior may still occur.

2.4.3 MO / removable disk images

Supported formats:

Format	File extension	Notes
Raw image	Many	Extensions include: .ima .img
Japanese FDI	.mdi / .zdi	

2.4.4 CD-ROM / DVD-ROM optical disc images

Supported formats:

Format	File extension	Notes
Cue sheet	.cue + .bin (+ optional audio)	*Audio tracks are supported.*
ISO	.iso	
Alcohol 120%	.mds + .mdf	Support for Daemon Tools MDS v2 images will not be available on Windows hosts if the included `mdsx.dll` file is missing from the 86Box directory.
Daemon Tools	.mdx	Support will not be available on Windows hosts if the included `mdsx.dll` file is missing from the 86Box directory.

CD audio

Compact Disc Digital Audio (CDDA) playback through the emulated CD-ROM drives is supported on **Cue sheet** and **Daemon Tools** images. Audio output is enabled on the first CD-ROM drive and muted on subsequent drives by default; individual drives can be muted or unmuted through the *status bar* or *Media menu*.

For **Cue sheet** images, audio tracks in raw (.bin), encapsulated (.wav) and compressed (.mp3 .ogg .flac) formats are supported.

2.4.5 Cassette tape images

Supported formats:

Format	File extension	Notes
Raw PCM audio	Many	Extensions include: .pcm .raw Audio format must be unsigned 8-bit mono.
PCE cassette	.cas	
Wave audio	.wav	Audio format must be unsigned 8-bit mono.

2.4.6 PCjr cartridge images

Supported formats:

Format	File extension	Notes
Raw image	Many	Extensions include: .a .b .bin
JRipCart	.jrc	

2.4.7 Creating and using disk images

Disk images are a convenient way to transfer files in and out of emulated machines, without the complexity of setting up networking. There are many different command line and GUI tools available for manipulating disk images on each host operating system.

Warning

Before editing or mounting any disk images, make sure they are **not in use** by any emulated machine that is currently running.

Editing and mounting on Windows

WinImage or **PowerISO** can be used to create and manipulate disk images on Windows.

VHD images can be natively mounted by double-clicking them on File Explorer, or through the **Disk Management** tool (`diskmgmt.msc`): select *Action* > *Attach VHD* to mount an image. Eject the drive through File Explorer to unmount. The `diskpart` command line utility also provides VHD mounting/unmounting functionality.

Mounting on macOS

macOS can natively mount raw hard disk and floppy images formatted as **FAT** and its variants. Open the image in Finder to mount it, and eject the disk to unmount.

Editing on Linux and macOS

The mtools suite is "a collection of utilities to access MS-DOS disks from GNU and Unix without mounting them". It can be used to create floppy disk images and directly copy files to them. The `mtools` package is available on many Linux distributions, as well as macOS Homebrew.

Creating floppy images

The following command will create a 1.44M (1440 KB, double-sided, 18 sectors per track, 80 cylinders) floppy image named `floppy.img` with a label of `LABEL`:

```
mformat -f 1440 -v LABEL -C -i floppy.img ::
```

The `-f` option specifies the format of the floppy being created. The command can be adjusted for format, label, and image name as needed. Refer to the mtools documentation for a full list of supported formats.

Copying files to floppy images

The following command will copy `file1` and `file2` to the floppy image `floppy.img`:

```
mcopy -i floppy.img file1 file2 ::
```

Wildcards are also supported with `mcopy`.

Note

The `::` is required to let `mtools` know there are no more files to copy or arguments to process.

Mounting on Linux

Linux can natively mount raw disk images (floppy or hard disk) of most types (`FAT` and `NTFS` included). The easiest path is to use losetup so that partitions can be properly recognized. Floppies are not normally partitioned, and you can use mount directly.

All following commands must be run as root:

```
losetup -fP /path/to/86box/hdd
losetup                         # to verify which loopback device was set up.
                                # Assuming /dev/loop0 was selected:
mount /dev/loop0p1 /mnt         # Mount the first partition at /mnt
```

Disk images should at least be unmounted before running 86Box again, and preferably detached too:

```
umount /mnt
losetup -d /dev/loop0
```

Partitionless media can be mounted directly:

```
mount /path/to/86box/fdd /mnt
```

VHD images may be mounted via qemu-nbd:

```
modprobe nbd max_part=16
qemu-nbd -f vpc -c /dev/nbd0 /path/to/86box/hdd
mount /dev/nbd0p1 /mnt
  # After doing some work...
umount /mnt
qemu-nbd -d /dev/nbd0
```

2.5 Tertiary and quaternary IDE

The generic additional tertiary and quaternary IDE controllers, enabled through the *Storage controllers settings page's General tab*, are not supported by all emulated BIOSes and may require manual configuration of the emulated operating system. The specific details are outlined on this section.

2.5.1 System resources

The following resources are used by these additional controllers:

Channel	Main I/O port	Status I/O port	IRQ
Tertiary	1E8h	3EEh	11
Quaternary	168h	36Eh	10

Note

The tertiary and quaternary I/O ports and IRQs were incorrectly switched in 86Box versions prior to 4.0.1.

Each controller's IRQ can be configured through its respective *Configure* button on the *hard disk controller selector*. The *Plug and Play* option on the *IRQ* box enables Plug and Play functionality, allowing a PnP compliant operating system to automatically set the controller's IRQ, while all other options set a static IRQ with no Plug and Play.

Note

- When using a non-Plug and Play IDE controller on an emulated machine which supports Plug and Play, remember to mark the IRQ as being used by a legacy ISA device in the BIOS setup utility.
- Many operating systems do not allow non-Plug and Play IDE controllers to use IRQs outside of the default ones listed on the table above.

2.5.2 BIOS support

The tertiary and quaternary controllers are not visible and not bootable by the BIOS on most machines currently emulated by 86Box, no matter whether or not they are Plug and Play.

Machines with **MR BIOS version 3** are the rare exception to this rule, since that BIOS provides full support for non-Plug and Play controllers (as long as the *default IRQs for each controller* are used), including bootability and INT 13h services.

2.5.3 Operating system support

DOS and real mode

DOS and other real mode operating systems rely on INT 13h services provided by the BIOS to access hard disks. These are only provided for the tertiary and quaternary channels by **MR BIOS version 3**, as mentioned above.

Windows 95, 98 and Me

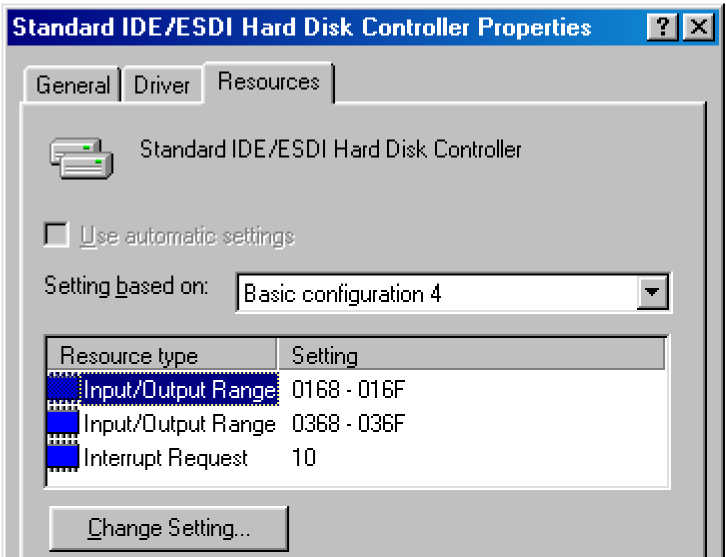
The **Windows 9x family** will automatically detect Plug and Play IDE controllers on boot. Non-Plug and Play controllers will be detected during installation *only if the BIOS supports them*. Follow these steps to enable a non-Plug and Play controller on an already-installed system:

1. Go to the *Add New Hardware* control panel.

2. Add a *Standard IDE/ESDI Hard Disk Controller* from the *Hard disk controllers* category.
3. Don't restart the system when asked to.
4. Go to the *Device Manager* tab of the *System* control panel.
5. Select the newly-added *Standard IDE/ESDI Hard Disk Controller* device from the *Hard disk controllers* category and click *Properties*.
6. Go to the *Resources* tab.
7. Select *Basic configuration 4* in the *Settings based on* box.
8. Change the resource settings to match the I/O ports on the *table above* and the configured IRQ. The first *Input/Output Range* range corresponds to the **main** I/O port, the second one corresponds to the **status** I/O port, and *Interrupt Request* corresponds to the IRQ.

- The status I/O port range is off by 6. Select 03E8 for the tertiary channel or 0368 for the quaternary channel.
- The screenshot below shows an example configuration for the tertiary channel.

9. If both the tertiary and quaternary channels are enabled, repeat the steps above to enable the other controller.



Windows NT, 2000 and XP

Windows 2000 and XP will automatically detect Plug and Play IDE controllers on boot. Additionally, **Windows NT 3.5, 4.0, 2000 and XP** will automatically detect non-Plug and Play controllers during installation, regardless of BIOS support; however, this auto-detection of non-PnP controllers does not work on most machines with **Award BIOS**.

Note

If you install the system to a hard disk on one of the additional controllers, it will not be bootable unless *the BIOS supports booting from these controllers*.

On **Windows 2000 only**, non-Plug and Play controllers can be enabled on an already-installed system through *Add New Hardware* similarly to *Windows 9x as shown above*. The resource parameters cannot be changed, and therefore, only the *default IRQs for each controller* are supported. *Basic configuration 0002* corresponds to the **tertiary** channel, while *Basic configuration 0003* corresponds to the **quaternary** channel.

Windows Vista and 7

The **Windows NT 6 family** does not support legacy (ISA or VLB) IDE controllers, and therefore cannot use the additional channels as currently emulated by 86Box.

Linux

There are different steps for enabling additional IDE controllers on Linux, depending on which IDE driver stack is used by your distribution's kernel.

Modules can be loaded at any time with the `modprobe` command, or loaded on boot by adding the module's name (and parameters if required) to a file in `/etc/modules-load.d` on newer systemd-based distributions, or the `/etc/modules` file on older distributions.

- **Legacy IDE** (typically kernels **older than 2.6.19**):
  - Load the `ide-pnp` module to enable Plug and Play controllers.
  - Non-Plug and Play controllers require editing the kernel command line on your bootloader to add each controller's I/O ports and IRQ:
    - **Tertiary:** `ide2=0x1e8,0x3ee,11` (assuming IRQ 11)
    - **Quaternary:** `ide3=0x168,0x36e,10` (assuming IRQ 10)
- **libATA** (typically kernels **2.6.19 and above**):
  - Load the `pata_isapnp` module to enable Plug and Play controllers.
  - Load the `pata_legacy` module with the `probe_all=1` parameter to automatically detect and enable non-Plug and Play controllers. Only the *default IRQs for each controller* are supported.

> **Note**
>
> Some distributions may automatically detect additional IDE controllers; however, that is very rarely the case.

2.6 Networking

86Box supports different connection modes for the *emulated network cards*. The specific details on these connection modes and network emulation as a whole are outlined on this section.

2.6.1 Null Driver

Default mode. The selected network card will be available to the emulated machine, but any packets sent through it will be dropped.

In this mode, the network card will default to a disconnected cable on startup; if the cable is connected through the *status bar* or *Media menu*, cards with link state detection support will report a connection, but packets will still be dropped.

2.6.2 SLiRP

SLiRP creates a private network with a virtual router, allowing the emulated machine to reach the host, its network and the Internet; on the other hand, the host and other devices on its network cannot reach the emulated machine, unless *port forwarding* is configured. This is similar to the **NAT** mode on other emulators and virtualizers.

The virtual router provides automatic IP configuration to the emulated machine through DHCP. If that is not an option, use the following static IP settings, replacing *x* with 2, 3, 4 or 5 for the first, second, third or fourth network card to use SLiRP respectively:

- **IP address:** 10.0.*x*.15
- **Subnet mask:** 255.255.255.0
- **Default gateway:** 10.0.*x*.2
- **DNS server:** 10.0.*x*.3

The host can be reached through IP address 10.0.*x*.2, while other devices on the host's network can be reached through their normal IP addresses. Advanced users can *override the private network's address*.

Note

SLiRP is only capable of routing TCP and UDP traffic, with limited ICMP ping support. Other protocols such as IPX and NetBEUI can only be used with the other network modes.

2.6.3 PCap

PCap connects directly to one of the host's network adapters. The emulated machine must be configured as if it were a real machine on your network. This is similar to the **Bridge** mode on other emulators and virtualizers.

On Windows hosts, this mode requires Npcap to be installed with **WinPcap API-compatible Mode enabled** (or use another WinPcap-compatible driver), or the correct permissions to be set for accessing `pcap` on Linux or `bpf` on macOS hosts. **Only wired Ethernet network connections** are compatible; Wi-Fi and other connections will not work at all, as they do not allow PCap to listen for packets bound to the emulated card's MAC address.

Note

PCap is not supported on the **Linux AppImage**, as the `cap_net_raw` and `cap_net_admin` permissions cannot be passed through to it. Extracting the AppImage or running it as root is not recommended.

Private PCap network

If you have an incompatible network connection on your host system (such as Wi-Fi), or if you wish to connect the emulated machine to the host without also connecting it to your network, a private network can be created with PCap in one of two ways:

- Install and configure the *Microsoft KM-TEST Loopback Adapter* included with Windows.
  - Guides on how to install this adapter are available online.
  - The adapter alone only provides a direct connection to the host, with no DHCP server, therefore requiring manual IP configuration on both the host and the emulated machine.
  - Windows' *Internet Connection Sharing* feature can be used to connect the emulated machine to the host's network and the Internet, with DHCP for automatic IP configuration, similarly to SLiRP but with the added benefit that the host can reach the emulated machine without port forwarding.
    - Port forwarding can be performed through Internet Connection Sharing's *Settings*.
- If VMware is installed, use one of the VMnet adapters included with it.
  - *VMnet1* (Host-only) connects to the host only.
  - *VMnet8* (NAT) connects to the host, its network and the Internet.
    - Port forwarding can be performed through the Virtual Network Editor's *NAT Settings*.

2.6.4 Local Switch

The local switch is a virtual Ethernet switch which uses multicast networking to **automatically connect** multiple 86Box machines running on the same host and on other hosts connected to the same real network; these machines can then communicate with each other as if they were connected through an actual switch.

Unlike *VDE* and *TAP*, the switch works on **all host operating systems** supported by 86Box, and is **fully plug-and-play** with no configuration required on the host side *in most cases*. However, it only provides connectivity between machines, so connecting to an external network and the Internet requires a machine running router software with two network cards: one connected to the switch and the other connected to the external network through *SLiRP*, *PCap* or other means.

Shared secret

The shared secret *option* allows for isolating virtual networks by setting a password. An emulated network card configured with a shared secret can **only** communicate with other emulated network cards set to the same shared secret, even if the machines are running on hosts connected to the same real network. This allows for different users on a shared real network to have their own separate virtual networks, for instance.

Important

Shared secrets only provide basic isolation, without additional security nor privacy. Packets are sent in plain text, and therefore they can be captured and spoofed by network analysis tools or by modifying the 86Box source code.

Hub Mode

The hub mode *option* turns the local switch into an Ethernet hub. In this mode, also referred to as *promiscuous mode*, the emulated network card will listen in to **all packets** sent through the switch, including those not bound to the card's MAC address. Due to the performance impact, enabling hub mode is only recommended for specific applications such as packet sniffing.

Troubleshooting

If you're having trouble getting machines to communicate with each other through the local switch only (while other modes work), follow this checklist:

- All machines should be running on the **same 86Box version** to avoid incompatibilities.
- All machines must have the **same** *shared secret* if one is set.
- All hosts must have a **single connection** to the **same network**.
  - A host with multiple interfaces and/or IPv4 addresses on the same network may cause issues related to packet duplication.
  - Connecting two hosts on separate networks through a third middle-man host is not supported.
- Any **firewalls** must allow traffic to UDP port 8086, with IP fragmentation, on multicast groups 239.255.86.86 (used if no shared secret is set) and 239.255.80.86 (used if a secret is set).
- Try connecting all hosts through a **wired connection**, as some routers have trouble handling multicast between Wi-Fi and wired devices, or even between Wi-Fi devices.
- If you use **enterprise switches**, try changing the *IGMP Snooping* option in their settings.

Interoperability

If no *shared secret* is set, the local switch uses a simple protocol which allows **other emulators and virtualizers** to connect to it. Currently only QEMU supports connecting to the switch.

QEMU

QEMU 7.2 and newer can connect to the local switch through the `dgram` network backend with options `remote.type=inet,remote.host=239.255.86.86,remote.port=8086` as shown in the example below. This requires starting QEMU from the command line, as `dgram` is **not available through front-ends** such as libvirt/virt-manager, Proxmox and UTM.

Example: starting an x86 machine with a PCnet network card connected to the local switch

```
qemu-system-x86_64 \
    -device pcnet,netdev=net0 \
    -netdev dgram,id=net0,remote.type=inet,remote.host=239.255.86.86,remote.port=8086 \
    # add other options, and on Windows replace \ with ^
```

Technical details

Applications can connect to the switch by sending and receiving raw Ethernet frames as UDP datagrams on multicast group 239.255.86.86 port 8086. If a shared secret is set, multicast group 239.255.80.86 is used instead, and each frame is prefixed by a 32-byte SHA3-256 hash of the secret string. Unicast and broadcast are currently not supported in normal operation and only used as a fallback on network interfaces flagged as non-multicast-capable by the operating system.

2.6.5 VDE

Virtual Distributed Ethernet is a virtual Ethernet switch system for connecting different applications such as 86Box to each other. See VDE Basic Networking for a brief overview.

Note

VDE is only available on **Linux** and **macOS** hosts.

One of VDE's core concepts is the *plug*. 86Box allows for *plug*ging an emulated machine into a virtual switch created by VDE; this virtual layer 2 switch is capable of carrying any Ethernet-based protocols such as IP and IPX.

Installing VDE tools

The VDE tools are required to create the virtual switch that 86Box attaches to with a virtual cable.

Linux

On Debian, Ubuntu and derivatives, VDE and some of its associated commands are split into different packages. Install the libraries and their associated tools:

```
apt install libvdeplug2 vde-switch vde2
```

Note

Other distributions should have similar package names.

macOS

VDE is available through Homebrew or MacPorts.

```
brew install vde
```

```
port install vde2
```

Creating the virtual switch

Before connecting 86Box, a virtual switch must be created with the `vde_switch` tool.

```
vde_switch --numports 8 --mgmt /tmp/vde.mgmt -s /tmp/vde.ctl
```

This command:

- Creates the *management* socket at `/tmp/vde.mgmt`
- Creates the *control* socket at `/tmp/vde.ctl`
- Sets the number of switch ports to 8 (default is 32)

Adding `--daemon` to the command will run `vde_switch` in the background.

Note the `/tmp/vde.ctl` path for the control socket, which is what should be provided in the *network settings*.

Note

You can adjust the file paths or permissions as necessary. Refer to `vde_switch -h` for more information on available options.

Configuring 86Box for VDE

Go to the emulated machine's *network settings* and select *VDE* as the mode for the emulated network card. Enter the *control* socket path, which is `/tmp/vde.ctl` for the example above, in the *VDE Socket* box.

Once these settings are saved, the machine should automatically connect to the VDE switch. Check the *status bar* or *Media menu* to make sure the emulated network cable is actually connected.

VDE switch status

The `vdeterm` command can be used to view the status of the virtual switch. It requires the path to the *management* socket (instead of the *control* socket) created alongside the switch; the command would be `vdeterm /tmp/vde.mgmt` for the example above.

Once in the command line, enter `help` to view a list of available commands. One helpful command is `port/allprint` which displays a list of all virtual switch ports and the processes attached to them:

```
vde[/tmp/vde.mgmt]: port/allprint

Port 0001 untagged_vlan=0000 ACTIVE - Unnamed Allocatable
 Current User: myusername Access Control: (User: NONE - Group: NONE)
  -- endpoint ID 0003 module unix prog   : 86Box virtual card user=myusername PID=12345
Success
```

In addition to `vdeterm`, the command line interface can be accessed through `vde_switch` if it was started without the `--daemon` option, by pressing Enter on its terminal.

Other VDE features

This guide only covers the basics of VDE. It provides many more useful features such as:

- Connecting virtual switches **across host machines** with `vde_cryptcab`
- Bridging virtual switches with **network interfaces** to provide access to the Internet and other networks
- Connecting to **other emulators and virtualizers** with VDE support such as QEMU and VirtualBox
- Creating **VLANs and access control policies** which can be assigned to switch ports

2.6.6 TAP

TAP is a simpler method for connecting 86Box machines and other applications together without the complexity of *VDE*.

Note

TAP is only available on **Linux** hosts.

The bridge name specified in the *network settings* represents a virtual network: all applications set to the same bridge name will be connected to the same network. 86Box will automatically create the bridge if it doesn't already exist.

2.6.7 Modem

The emulated modem can **dial-out** to Telnet servers as a client, receive **dial-in** calls as a server, or connect to a network through SLIP **dial-up**.

Important

The *Telnet client* and *SLIP* modes cannot understand **country and area codes**, which are enabled by default on Windows. Check your dialing settings if you always get *no answer* or *no carrier*.

The *Configure* button next to the *network card selector* provides these settings for a modem:

- **Serial Port:** port to attach the modem. When using a *serial mouse* or *other serial devices*, make sure to select a port that is not used.
- **Baud Rate:** bit rate for communicating with the modem. This has an impact on the reported and actual transfer speeds.
- **TCP/IP listening port:** TCP port number used for the *Telnet server* to receive dial-in connections, or `0` to disable dial-in.
- **Phonebook File:** text file containing a list of phone numbers and the addresses they should connect to. Format is one entry per line (up to 200 entries), with the phone number, followed by a space or tab, followed by the address to connect to.
- **Telnet emulation:** handle Telnet protocol sequences instead of passing them through the modem. If this option is off, then a raw connection is established.

Note

Telnet emulation must be **on** for connecting to some Telnet servers which require option negotiation, and **off** for connecting two 86Box machines or other binary applications.

Telnet client

Dial the `address` (defaults to port 23) or `address:port` of a Telnet server as a phone number to connect to it through the modem. Both IP addresses and DNS names are accepted.

For dialer software which does not support entering `.` or `:` as part of a phone number, an alternative option is specifying a zero-padded IP optionally followed by a port number, such as `010176001086` for `10.176.1.86` on the default port 23, or `0101760010864321` for `10.176.1.86:4321`.

Telnet server

By setting the modem's **TCP/IP listening port** option, a **dial-in server** is started on that port, allowing a Telnet client or another application to dial into the emulated machine by connecting to the host on the specified port number. The modem rings as soon as a TCP connection is received by the listening server; the server will refuse connections if an incoming or outgoing call is already active, as call waiting is not emulated.

For games and other applications that support dial-in, a connection **between 86Box machines** can be established with this mode, by configuring the first machine to listen as a server, and having the second machine dial into the first one's IP and port.

SLIP

Dial number `0.0.0.0` or `000000000000` (twelve zeros) to establish a **Serial Line Internet Protocol (SLIP)** dial-up network connection. SLiRP is the only supported network mode when using a modem; other modes are ignored.

Important

This is not the standard PPP connection mode typically used by dial-up providers around the world; therefore, dialer software that is compatible with and configured for SLIP must be used.

Example configuration for Windows 98

1. Open the *Connect to the Internet* desktop shortcut. Depending on your Windows region, this opens either the MSN setup wizard or the Internet Connection Wizard, which will in turn open the modem setup wizard to add a modem.
2. Check *Don't detect my modem; I will select it from a list* and click *Next*.
3. Select any of the standard modem types and click *Next*. The speed does not matter, as the configured baud rate is automatically used.
4. Select the serial port to which the modem is connected (COM1 by default) and click *Next*.
5. Wait for the modem to install and click *Finish*.
6. If you're on the MSN setup wizard, click *Lan/Manual* to switch to the proper Internet Connection Wizard.
7. Select *Connect using my phone line* and click *Next*.
8. Enter `0.0.0.0` as the phone number and **uncheck** *Dial using the area code and country code*.
9. Click *Advanced...* and perform these changes:

- On the *Connection* tab, set the **connection type** to *SLIP (Serial Line Internet Protocol)*.
- On the *Addresses* tab, set the IP and DNS addresses **manually** according to *the SLiRP rules above*. For the first SLiRP instance, these are 10.0.2.15 for IP and 10.0.2.3 for DNS.
- Click *OK* then *Next*.

10. Leave the username and password **blank** and click *Next* then *Yes* on both warnings.
11. Set the connection name (or leave the default alone) and click *Next*.
12. Skip setting up an e-mail account and click *Next* then *Finish*.

To connect through dial-up, open Internet Explorer and click *Connect* when prompted. When the *Terminal Screen* shows up, click *Continue* and the connection will be established.

2.6.8 Advanced networking features

The following advanced features can be accessed by directly editing the emulated machine's configuration file, which is `86box.cfg` by default.

SLiRP network address

The private network provided by SLiRP can be moved to any /24 IPv4 block through the `net_XX_addr` directive in the `[Network]` section of the configuration file, where `XX` is the number of the emulated network card, in the range of 01 to 04.

The host addresses within the custom network cannot be changed; for instance, if the network is configured to 192.168.86.0, then the emulated machine's IP will be 192.168.86.15.

Note

Any IP passed to `net_XX_addr` that is not a /24 network address (ending in .0) will be ignored.

Example: change first network card's SLiRP network to 192.168.86.0/24

```
[Network]
net_01_net_type = slirp
net_01_addr = 192.168.86.0
```

SLiRP port forwarding

Port forwarding allows the host system and other devices on its network to access TCP and UDP servers running on the emulated machine. This feature is configured through the `[SLiRP Port Forwarding #x]` section of the configuration file, where `x` is the number of the emulated network card, in the range of 1 to 4.

Each port forward must be assigned a number, starting at 0 and counting up (skipping a number will result in all subsequent port forwards being ignored), which replaces `X` on the following directives:

- `X_protocol`: Port type: `tcp` or `udp` (default: `tcp`)
- `X_external`: Port number on the host (default: same port number as `X_internal`)
- `X_internal`: Port number on the emulated machine (default: same port number as `X_external`)

The host system can access forwarded ports through 127.0.0.1 or its own IP address, while other devices on the network can access them through the host's IP address.

Note

The emulated machine's IP address must end in .15 (the default IP provided through DHCP) for port forwarding to work.

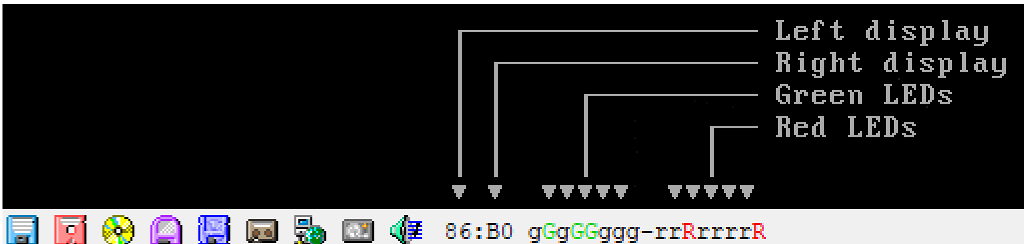
Example: forward host TCP port 8080 to emulated machine port 80, and host UDP port 5555 to emulated machine port 5555

```
[SLiRP Port Forwarding #1]
0_external = 8080
0_internal = 80
1_protocol = udp
1_external = 5555
```

2.7 ISABugger

The ISABugger card provides a debugging interface for software developers, consisting of two 8-bit hexadecimal displays and two banks of 8 LEDs, all controlled by the emulated machine. It can be enabled through the *Peripherals settings page*.

These displays and LEDs are displayed in the *status bar* as described in the diagram below:



2.7.1 Background

From `src/device/bugger.c`:

```
Implementation of the ISA Bus (de)Bugger expansion card
sold as a DIY kit in the late 1980's in The Netherlands.
This card was a assemble-yourself 8bit ISA addon card for
PC and AT systems that had several tools to aid in low-
level debugging (mostly for faulty BIOSes, bootloaders
and system kernels...)

The standard version had a total of 16 LEDs (8 RED, plus
8 GREEN), two 7-segment displays and one 8-position DIP
switch block on board for use as debugging tools.

The "Plus" version, added an extra 2 7-segment displays,
as well as a very simple RS-232 serial interface that
could be used as a mini-console terminal.
```

2.7.2 Registers

The ISABugger's control registers can be accessed through the following operations on I/O ports `0x7a` and `0x7b`:

- **Writing:** write the register's index to port `0x7a`, then write the value to port `0x7b`.
- **Reading:** write the register's index to port `0x7a`, then read the value from port `0x7b`.
- **Index reading:** the last register index written to port `0x7a` can be read back from the same port. The most significant bit is always set, as an indicator that the ISABugger is enabled.

> **Note**
>
> The ISABugger I/O ports only support byte (`inb`/`outb`) operations. Word (`inw`/`outw`) and dword (`inl`/`outl`) operations will result in undefined behavior; so will selecting or attempting to read back an unknown register index, or performing an illegal operation such as reading from a write-only register.

Register reference

Index `0x00` - Red LEDs (write-only)

Index `0x01` - Green LEDs (write-only)

Each LED bank shows a binary representation of the 8-bit value written to its register, from the most significant bit on the left to the least significant bit on the right. Setting a bit will light up its corresponding LED (displayed as **G** or **R**), and clearing a bit will dim its LED (displayed as **g** or **r**).

Index `0x02` - Right display (write-only)

Index `0x04` - Left display (write-only)

Each display shows a hexadecimal representation of the 8-bit value written to its register.

Index `0x20` - Serial port data (not implemented) (read/write)

Index `0x40` - Serial port configuration (not implemented) (read/write)

While the aforementioned real ISABugger card is equipped with an independent RS-232 serial interface, that feature is currently not implemented on 86Box in an user-facing manner.

Index `0x80` - Initialize (not implemented) (write-only)

This register has **no effect** on 86Box, as the emulated ISABugger is always enabled and ready.

Index `0xff` - Reset (special)

Writing register index `0xff` to port `0x7a` will immediately reset all registers to their startup value, clearing all displays and LED banks.

This is a **special register** which cannot be read or written; writing to port `0x7b` immediately after a reset will result in the value being sent to the default register index of `0x00`, which corresponds to the red LEDs.

2.8 External OPL audio

The **OPL2Board** *sound card* allows for a real Yamaha OPL2 (YM3812) chip to be connected to the emulated machine for authentic FM synthesis output.

2.8.1 Usage

1. Connect the OPL2 Audio Board from Cheerful Electronic to a supported Arduino board.
2. Connect the Arduino board to the host system.
3. Select the **OPL2Board (External Device)** sound card on the *emulated machine's configuration*.
4. Use the *Configure* button to select the Arduino's serial port.

Note

- The **OPL3 Duo!** board is currently not supported.
- Regular **PCM/wave audio** still requires an emulated sound card to be configured. If the emulated sound card provides its own OPL, it can often be **muted** through a mixer utility within the machine.

CHAPTER

THREE

DEVELOPMENT

3.1 Build guide

86Box is built using CMake in combination with other build systems. The build actions are described in CMakeLists.txt files in most directories, which are the translated to the build system of choice by a CMake generator.

The following files are of particular interest:

- ./CMakeLists.txt is the top level file, which defines the 86Box project and available configuration options;
- ./src/CMakeLists.txt defines the main 86Box executable target

3.1.1 Toolchain files

Toolchain files are contained in the cmake directory. They define compiler flags and the 86Box-specific Release, Debug and Optimized build types.

It is not required to use the included toolchain files, but it is highly recommended to make sure your build is compiled with the same configuration as used by the rest of the team and our userbase.

The currently included files are:

- flags-gcc.cmake contains the generic flags used by GCC-like compilers
  - flags-gcc-<arch>.cmake includes flags specific to builds for a given architecture
- llvm-win32-<arch>.cmake defines the build environment for use with LLVM/clang and vcpkg on Windows

Toolchain files are consumed during the initial project generation stage by passing their path in the CMAKE_TOOLCHAIN_FILE variable, e.g.:

```
$ cmake ... -D CMAKE_TOOLCHAIN_FILE=./cmake/flags-gcc-x86_64.cmake
```

Note

When using vcpkg, which uses its own toolchain file, the 86Box toolchain files must be chainloaded using the VCPKG_CHAINLOAD_TOOLCHAIN_FILE variable.

3.1.2 Presets

The CMakePresets.json file contains several common compilation options for 86Box:

Build name	Debug	Dev. branch	Optimized
`regular`	×	×	×
`development`	×	✓	×
`debug`	✓	×	×
`dev_debug`	✓	✓	×
`optimized`	×	×	✓

The presets are consumed during the initial project generation stage by using the `--preset` CMake command line option, e.g.:

```
$ cmake ... --preset regular
```

Note

Presets require CMake 3.21 or newer.

3.1.3 Obtaining the source code

There are multiple ways to obtain the 86Box source code in order to build it:

- Use the `git` command line. The utility needs to be installed and present in the search path.

  ```
  $ git clone https://github.com/86Box/86Box.git
  ```

- Use GitHub Desktop, SourceTree, Git for Windows or other Git frontend on your host.
- Download a ZIP file from GitHub and extract it. (not recommended)

3.1.4 Prerequisites

The build process requires the following tools:

- CMake (>= 3.15)
- `pkg-config`
- libatomic

Development files for the following libraries are also needed:

- FluidSynth
- FreeType
- libpng
- libslirp
- libsndfile
- RtMidi
- SDL2
- OpenAL (by default) or FAudio (installing FAudio is optional on Windows)
- libserialport (optional, needed for external OPL2Board support)
- Qt5 or Qt6 (optional, can be disabled)

Obtaining the dependencies

MSYS2

```
$ pacman -Syu $MINGW_PACKAGE_PREFIX-ninja $MINGW_PACKAGE_PREFIX-cmake $MINGW_PACKAGE_
↪PREFIX-gcc $MINGW_PACKAGE_PREFIX-pkgconf $MINGW_PACKAGE_PREFIX-openal $MINGW_PACKAGE_
↪PREFIX-freetype $MINGW_PACKAGE_PREFIX-SDL2 $MINGW_PACKAGE_PREFIX-zlib $MINGW_PACKAGE_
↪PREFIX-libpng $MINGW_PACKAGE_PREFIX-rtmidi $MINGW_PACKAGE_PREFIX-fluidsynth $MINGW_
↪PACKAGE_PREFIX-libslirp $MINGW_PACKAGE_PREFIX-libsndfile $MINGW_PACKAGE_PREFIX-qt5-
↪static $MINGW_PACKAGE_PREFIX-qt5-translations $MINGW_PACKAGE_PREFIX-vulkan-headers
```

Note

The command installs the packages only for the currently used MinGW environment, therefore you will need to repeat the procedure for every target you plan to build for.

Ubuntu, Debian

```
$ sudo apt install build-essential cmake extra-cmake-modules pkg-config ninja-build␣
↪libfreetype-dev libsdl2-dev libpng-dev libopenal-dev librtmidi-dev libfluidsynth-dev␣
↪libsndfile1-dev libserialport-dev qtbase5-dev qtbase5-private-dev qttools5-dev␣
↪libevdev-dev libxkbcommon-dev libxkbcommon-x11-dev libslirp-dev
```

Arch

```
$ sudo pacman -Sy base-devel cmake extra-cmake-modules pkgconf ninja libfreetype sdl2␣
↪libpng openal rtmidi libslirp fluidsynth libsndfile libserialport qt5-base qt5-xcb-
↪private-headers qt5-tools libevdev libxkbcommon libxkbcommon-x11 vulkan-devel
```

Fedora

- Fedora 41 and newer (DNF5)

```
$ sudo dnf group install c-development
$ sudo dnf install cmake extra-cmake-modules pkg-config ninja-build freetype-devel␣
↪SDL2-devel libatomic libpng-devel libslirp-devel libXi-devel openal-soft-devel␣
↪rtmidi-devel fluidsynth-devel libsndfile-devel libserialport-devel qt5-linguist␣
↪qt5-qtconfiguration-devel qt5-qtbase-private-devel qt5-qtbase-static wayland-
↪devel libevdev-devel libxkbcommon-x11-devel zlib-ng-compat-static libpng-static
```

- Fedora 40 and older (DNF4)

```
$ sudo dnf groupinstall "C Development Tools and Libraries"
$ sudo dnf install cmake extra-cmake-modules pkg-config ninja-build freetype-devel␣
↪SDL2-devel libatomic libpng-devel libslirp-devel libXi-devel openal-soft-devel␣
↪rtmidi-devel fluidsynth-devel libsndfile-devel libserialport-devel qt5-linguist␣
↪qt5-qtconfiguration-devel qt5-qtbase-private-devel qt5-qtbase-static wayland-
↪devel libevdev-devel libxkbcommon-x11-devel zlib-ng-compat-static libpng-static
```

macOS (Homebrew)

```
$ brew install cmake ninja pkg-config freetype sdl2 libpng openal-soft rtmidi libslirp␣
↪fluid-synth libsndfile libserialport qt@5
```

FreeBSD

```
$ pkg install cmake pkgconf freetype-gl sdl2 libspng openal-soft rtmidi qt5 libslirp␣
↪fluidsynth libsndfile
```

> **Note**
>
> If you get an error about `linux/input.h` while building, this is due to `libevdev.h` mistakenly importing a Linux header file on FreeBSD. Copy the full path of `libevdev.h` from the error and open it with an editor. Within the file, replace the `linux/input.h` reference with `dev/evdev/input.h` and try building again.

3.1.5 Building

Building 86Box can generally be condensed to the following steps:

1. Generate the project. This generally involves invoking the following base command line with additional options according to the development environment:

   ```
   $ cmake -B <build directory> -S <source directory>
   ```

   Build directory is where the resulting binaries and other build artifacts will be stored. Source directory is the location of the 86Box source code.

   Toolchain files and presets are specified at this point by using the respective options.

   Other options can be specified using the `-D` option, e.g. `-D NEW_DYNAREC=ON` enables the new dynamic recompiler. See `CMakeLists.txt` in the root of the repository for the full list of available options.

2. Build the project itself. This can be done by changing to the chosen build directory and invoking the chosen build system, or you can use the following universal CMake command:

   ```
   $ cmake --build <build directory>
   ```

   Appending the `-jN` option (where `N` is a number of threads you want to use for the compilation process) will run the build on multiple threads, speeding up the process some.

   > **Note**
   >
   > If you make changes to the CMake build files, running the command will automatically regenerate the project. There is no need to repeat step 1 or to delete the build directory.

3. If everything succeeds, you should find the resulting executable in the build directory. Depending on the build system, it might be located in some of its subdirectories.

> **Tip**
>
> The executable can be copied to a consistent location by running the following command:

```
$ cmake --install <build directory> --prefix <destination>
```

The emulator file should then be copied into a `bin` directory in the specified location.

Appending the `--strip` parameter will also strip debug symbols from the executable in the process.

3.2 Advanced builds

The experimental builds page, backed by the 86Box Jenkins, provides pre-release testing builds for advanced users. These are linked to the 86Box git repository on GitHub; a new build is produced from the latest source code every time the repository is updated.

> **Important**
>
> Testing builds are development snapshots which may contain bugs, unfinished features, reduced performance or other issues. These should only be used if you know what you're doing.

3.2.1 Variants

86Box builds are available in a number of variants. The experimental builds page will automatically detect the recommended variant for the system you're viewing it on, but if you're downloading builds for a different system or directly through Jenkins, use the guide below to select a variant:

- The **Old Recompiler** is recommended. The **New Recompiler** is in beta; you may experience bugs and performance loss with it.
  - The Old Recompiler is not available for the ARM architecture. You must select the New Recompiler for running 86Box on ARM Linux systems.
- On **Windows**, select **x64** or **ARM** according to your system's architecture.
- On **Linux**, select **x64** or **ARM** according to your system's architecture.
- On **macOS**, **Universal** supports both Intel and Apple Silicon Macs.
  - The New Recompiler is always used on Apple Silicon due to its ARM architecture, even if the Old Recompiler is selected.

Discontinued variants

The following variants are no longer built by Jenkins and can only be *compiled from source* where applicable:

- 32-bit architectures (**x86** and **ARM32**) as of September 3rd 2024.
  - These were eliminated to better focus development on relevant 64-bit architectures, since systems old enough to be 32-bit-only lack the performance for a satisfactory emulation experience.
  - The dynamic recompiler and other components no longer support 32-bit architectures as of September 13th 2025.
- Debug variants (**86Box-Debug**) as of April 2nd 2023.
  - These were compiled with debug symbols and reduced optimizations to help with running the emulator under `gdb` or other debuggers. They were eliminated as the setup process for debugging grew closer to just compiling from source instead.
- Dev variants (**86Box-Dev** and **86Box-DevODR**) as of November 18th 2021.
  - These contained incomplete and experimental features subject to change at any time, with the -Dev variant also containing the New Recompiler beta.
- Optimized variants (**86Box-Optimized**) as of March 18th 2021.
  - These had aggressive microarchitecture-specific optimizations which provided very little performance improvement (within margin of error on modern CPUs) while introducing bugs and other incorrect behavior.

3.3 Manager interface

The manager interface allows third-party applications to determine the status of 86Box instances and issue commands to them.

3.3.1 JSON protocol

This is the latest interface protocol, introduced with 86Box 5.0 and used by the built-in *VM manager*.

The manager attaches to the 86Box instance by launching it with the `VMM_86BOX_SOCKET` environment variable set to the full path to a named pipe on Windows or a `SOCK_STREAM` domain socket on Unix-compatible operating systems. The pipe/socket must already exist.

Messages are sent as `QString` objects via `QDataStream` version `Qt_5_7` (17).

JSON messages are formatted as such:

Name	Type	Description
`type`	String	`"Client"` for incoming messages from the 86Box instance, or `"VMManager"` for outgoing messages from the manager.
`version`	String	86Box version.
`message`	String	Message name.
`params`	Object	Message parameters, if applicable. Contents are specific to each message.

The following messages are sent by the 86Box instance to the manager:

Name	Description
`WindowBlocked`	Sent when the main window is blocked due to a modal dialog box being shown.
`WindowUnblocked`	Sent when the main window is unblocked.
`RunningStateChanged`	Sent when the emulated machine transitions to a new state. The `status` (number) parameter can be one of: • `0`: Running • `1`: Paused • `2`: Waiting for user input • `3`: Paused and waiting for user input
`ConfigurationChanged`	Sent when the emulated machine's *settings* are changed.
`WinIdMessage`	Sent upon startup to pass the platform-specific window handle through the `params` (number) parameter.
`GlobalConfigurationChanged`	Sent when *preferences* are changed. The manager should respond by sending a `GlobalConfigurationChanged` message to all running instances.

The following messages can be sent to the instance by the manager:

Name	Description
`Pause`	Pause or unpause the emulated machine.
`CtrlAltDel`	Send a *Ctrl+Alt+Delete* keyboard sequence to the emulated machine.
`ShowSettings`	Open the emulated machine's *Settings window*.
`ResetVM`	Force a reset of the emulated machine.
`RequestShutdown`	Send a shutdown request, which displays a confirmation prompt if enabled.

continues on next page

Table 2 – continued from previous page

Name	Description
ForceShutdown	Force a shutdown.
GlobalConfigurationChanged	Update *preferences*. Should be sent to all running instances after receiving a GlobalConfigurationChanged message from one instance.

3.3.2 Plain text protocol

This earlier protocol, introduced in 86Box 3.3, uses plain text messages instead of structured JSON.

The manager attaches to the 86Box instance by launching it with the 86BOX_MANAGER_SOCKET environment variable set to the full path to a named pipe on Windows or a SOCK_STREAM domain socket on Unix-compatible operating systems. The pipe/socket must already exist.

Commands sent by the manager must be followed by a newline character (\n). The following commands are recognized:

Name	Description
showsettings	Open the emulated machine's *Settings window*.
pause	Pause or unpause the emulated machine.
cad	Send a *Ctrl+Alt+Delete* keyboard sequence to the emulated machine.
reset	Force a reset of the emulated machine.
shutdown	Send a shutdown request, which displays a confirmation prompt if enabled.
shutdownnoprompt	Force a shutdown.

Furthermore, the emulator writes an ASCII 1 to the pipe/socket when the main window is blocked by a modal dialog box, and an ASCII 0 when the window is unblocked.

3.3.3 Window message protocol (Windows-only)

Warning

This protocol is **deprecated** as of 86Box 5.0 and will be removed in a future release. It is documented here for completeness only.

This earlier protocol, used by the legacy 86Box Manager application., uses Windows window messages sent to the emulator window and received on a specified window handle.

The manager attaches to the 86Box instance by launching it with the -H/--hwnd vm_id,hwnd command line option, where vm_id is an arbitrary 64-bit identifier number and hwnd is the window handle to receive messages on, both **in hexadecimal** without the 0x prefix.

All window messages sent by the emulator include the main window's handle in LPARAM, including WM_SENDHWND which is sent on startup and can be used to match the window handle to the identifier provided in the command line.

Name	Value	Sent by	Description
WM_SHOWSETTINGS	0x8889	Manager	Open the emulated machine's *Settings window*.
WM_PAUSE	0x8890	Manager	Pause or unpause the emulated machine.
WM_SENDHWND	0x8891	86Box	Sent when the emulator window is first displayed. WPARAM contains the manager-provided machine ID.
WM_HARDRESET	0x8892	Manager	Force a reset of the emulated machine.

continues on next page

Table 4 – continued from previous page

Name	Value	Sent by	Description
`WM_SHUTDOWN`	`0x8893`	Manager	Trigger a shutdown. `WPARAM 0` sends a shutdown request, which displays a confirmation prompt if enabled, while `1` forces a shutdown.
`WM_CTRLALTDEL`	`0x8894`	Manager	Send a *Ctrl+Alt+Delete* keyboard sequence to the emulated machine.
`WM_SENDSTATUS`	`0x8895`	86Box	Sent when the emulated machine is paused (`WPARAM 0`) or unpaused (1).
`WM_SENDDLGSTATUS`	`0x8896`	86Box	Sent when the emulator is waiting for user input in a modal dialog box (`WPARAM 0`) or when said dialog box has closed (1).
`WM_HAS_SHUTDOWN`	`0x8897`	86Box	Sent when the emulated machine shuts down.

3.4 API

This section documents the internal **Application Programming Interface** for extending 86Box.

3.4.1 Devices

The **device** is the main unit of emulated components in 86Box. Each device is represented by one or more constant `device_t` objects, which contain metadata about the device itself, several callbacks and an array of user-facing configuration options. Unless otherwise stated, all structures, functions and constants in this section are provided by `86box/device.h`.

Table 5: `device_t`

Member	Description
`name`	The device's name, displayed in the user interface. `"Foo-1234"` for example. Suffixes like `"(PCI)"` are removed at run-time.
`internal_name`	The device's internal name, used to identify it in the emulated machine's configuration file. `"foo1234"` for example.
`flags`	One or more bit flags to indicate the expansion bus(es) supported by the device, for determining *device availability* on the selected machine: • `DEVICE_SIDECAR`: IBM PCjr sidecar; • `DEVICE_ISA`: 8-bit ISA; • `DEVICE_ISA16`: 16-bit ISA; • `DEVICE_EISA`: EISA (reserved for future use); • `DEVICE_VLB`: VESA Local Bus; • `DEVICE_OLB`: OPTi Local Bus; • `DEVICE_AT32`: Mylex AT32 local bus; • `DEVICE_PCI`: 32-bit PCI; • `DEVICE_AGP`: AGP 3.3V; • `DEVICE_PCMCIA`: PCMCIA (reserved for future use); • `DEVICE_CARDBUS`: CardBus (reserved for future use); • `DEVICE_AC97`: AMR, CNR or ACR; • `DEVICE_PCJR`: IBM PCjr; • `DEVICE_PS2`: IBM PS/1 or PS/2; • `DEVICE_MCA`: 16-bit IBM Micro Channel Architecture; • `DEVICE_MCA32`: 32-bit IBM Micro Channel Architecture; • `DEVICE_CBUS`: PC-98 C-BUS (reserved for future use); • `DEVICE_HIL`: HP HIL (reserved for future use); • `DEVICE_COM`: serial port; • `DEVICE_LPT`: parallel port.
`local`	32-bit value which can be read from this structure by the `init` callback. Use this value to tell different subtypes of the same device, for example.
`init`	Function called whenever this device is initialized, either from starting 86Box or from a hard reset. Can be `NULL`, in which case the opaque pointer passed to other callbacks will be invalid. Takes the form of: `void *init(const struct device_t *info)` • `info`: pointer to this `device_t` structure; • Return value: opaque pointer passed to the other callbacks below, usually a pointer to the device's *state structure*.

continues on next page

Table 5 – continued from previous page

Member	Description
close	Function called whenever this device is de-initialized, either from closing 86Box or from a hard reset. Can be NULL. Takes the form of: `void close(void *priv)` • priv: opaque pointer previously returned by init.
reset	Function called whenever this device undergoes a soft reset. Can be NULL. Takes the form of: `void reset(void *priv)` • priv: opaque pointer previously returned by init.
available	Function called whenever this device's availability is being checked. Can be NULL, in which case the device will always be available. Takes the form of: `int available()` • Return value: 1 if the device is available for selection, or 0 if it is unavailable (due to missing ROMs, for example).
speed_changed	Function called whenever the emulated CPU clock speed is changed. Can be NULL. Timer intervals (when using the undocumented legacy timer API) and anything else sensitive to the CPU clock speed should be updated in this callback. Takes the form of: `void speed_changed(void *priv)` • priv: opaque pointer previously returned by init.
force_redraw	Function called whenever the emulated screen has to be fully redrawn. Can be NULL. Only useful for video cards. Takes the form of: `void force_redraw(void *priv)` • priv: opaque pointer previously returned by init.
config	Array of *device configuration options*, or NULL if no options are available.

State structure

Most devices need a place to store their internal state. We discourage the use of global structures, and instead recommend allocating a **state structure** dynamically in the init callback and freeing it in the close callback.

Code example: allocating and deallocating a state structure

```
#include <86box/device.h>

typedef struct {
    uint32_t type; /* example: copied from device_t.local */
    uint8_t  regs[256]; /* example: 256*8-bit registers */
} foo_t;

static void *
foo_init(const device_t *info)
{
    /* Allocate the device state structure. */
    foo_t *dev = (foo_t *) malloc(sizeof(foo_t));
    memset(dev, 0, sizeof(foo_t)); /* blank structure */

    /* Do whatever you want. */
    dev->type = info->local; /* copy device_t.local value */
```

(continues on next page)

(continued from previous page)

```
    /* Return a pointer to the state structure. */
    return dev;
}

static void
foo_close(void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Do whatever you want, then deallocate the state structure. */
    free(dev);
}

const device_t foo1234_device = {
    .name = "Foo-1234",
    .internal_name = "foo1234",
    .flags = DEVICE_ISA16,
    .local = 1234,
    .init = foo_init,
    .close = foo_close,
    /* ... */
};

const device_t foo4321_device = {
    .name = "Foo-4321",
    .internal_name = "foo4321",
    .flags = DEVICE_PCI, /* 32-bit PCI */
    .local = 4321, /* different device subtype */
    .init = foo_init,
    .close = foo_close,
    /* ... */
};
```

Registration

New devices must be **registered** before they can be selected by the user. This is usually accomplished by adding one or more `device_t` pointers to the **device table** for the device's class:

- **Video cards:** `video_cards` in `src/video/vid_table.c`
- **Sound cards:** `sound_cards` in `src/sound/sound.c`
- **Network cards:** `net_cards` in `src/network/network.c`
- **Parallel and serial port devices:** `char_devices` in `src/char/char.c`
- **Hard disk controllers:** `controllers` in `src/disk/hdc.c`
- **Floppy disk controllers:** `fdc_cards` in `src/floppy/fdc.c`
- **SCSI controllers:** `scsi_cards` in `src/scsi/scsi.c`
- **ISA RTC cards:** `boards` in `src/device/isartc.c`
- **ISA memory expansion cards:** `boards` in `src/device/isamem.c`

Devices not covered by any of the above classes may require further integration through modifications to the user interface and configuration loading/saving systems.

Availability

A device will be **available** for selection by the user if these criteria are met:

1. The device is *registered*, so that the user interface knows about it;
2. The selected machine has any of the expansion buses specified in the device's `flags`;
3. The device's `available` callback returns `1` to indicate the device is available (this will always be true if the `available` callback function is `NULL`).

The `available` callback can be used to verify the presence of ROM files if any ROMs are required by the device.

Code example: `available` checking for the presence of a ROM

```
#include <86box/device.h>
#include <86box/rom.h>

static int
foo1234_available()
{
    return rom_present("roms/scsi/foo/foo1234.bin");
}

const device_t foo1234_device = {
    /* ... */
    { .available = foo1234_available }, /* must have brackets due to the union */
    /* ... */
};
```

Configuration

Devices can have any number of user-facing configuration options, usually accessed through the **Configure** button next to the selection box for the device's class:

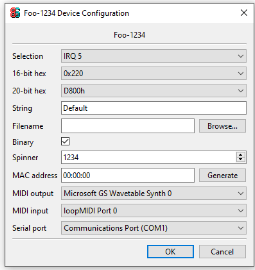


Fig. 1: All option types currently configurable through the user interface.

These options are stored in the emulated machine's configuration file, in a section identified by the device's `name`, optionally followed by the device's instance number (preceded by `#`) if it's different from the default `0`:

```
[Foo-1234 #1]
selection = 0
hex16 = 0220
```

(continues on next page)

(continued from previous page)

```
hex20 = D8000
string = Default
fname = D:/VMs/86Box/86Box.exe
binary = 1
spinner = 1234
mac = 00:00:00
midi_out = 0
midi_in = 0
```

Configuration options can be specified in the `config` member of `device_t`, as a pointer to a `const` array of `device_config_t` objects terminated by an object of `type CONFIG_END`.

Code example: device configuration options

```
#include <86box/device.h>

static const device_config_t foo_config[] = {
    {
        .name        = "selection",
        .description = "Selection",
        .type        = CONFIG_SELECTION,
        .default_int = 5,
        .selection   = {
            { .description = "IRQ 5", .value = 5 },
            { .description = "IRQ 7", .value = 7 },
            { NULL                               }
        }
    },
    {
        .name        = "hex16",
        .description = "16-bit hex",
        .type        = CONFIG_HEX16,
        .default_int = 0x220,
        .selection   = {
            { .description = "0x220", .value = 0x220 },
            { .description = "0x330", .value = 0x330 },
            { NULL                                   }
        }
    },
    {
        .name        = "hex20",
        .description = "20-bit hex",
        .type        = CONFIG_HEX20,
        .default_int = 0xd8000,
        .selection   = {
            /* While the memory *segment* is displayed to the user, we store the
               *linear* (segment << 4) base address in the configuration file. */
            { .description = "D800h", .value = 0xd8000 },
            { .description = "DC00h", .value = 0xdc000 },
            { NULL                                     }
        }
    },
    {
```

(continues on next page)

(continued from previous page)

```
        .name           = "string",
        .description    = "String",
        .type           = CONFIG_STRING,
        .default_string = "Default"
    },
    {
        .name           = "fname",
        .description    = "Filename",
        .type           = CONFIG_FNAME,
        .default_string = "",
        .file_filter    = "File type (*.foo)|*.foo|Another file type (*.bar)|*.bar"
    },
    {
        .name        = "binary",
        .description = "Binary",
        .type        = CONFIG_BINARY,
        .default_int = 1 /* checked by default */
    },
    {
        .name        = "int",
        .description = "Integer",
        .type        = CONFIG_INT,
        .default_int = 1234
    },
    {
        .name        = "spinner",
        .description = "Spinner",
        .type        = CONFIG_SPINNER,
        .default_int = 1234,
        .spinner     = {
            .min  = 1204,
            .max  = 1294,
            .step = 10
        }
    },
    {
        .name        = "mac",
        .description = "MAC address",
        .type        = CONFIG_MAC
    },
    {
        .name        = "midi_out",
        .description = "MIDI output",
        .type        = CONFIG_MIDI_OUT
    },
    {
        .name        = "midi_in",
        .description = "MIDI input",
        .type        = CONFIG_MIDI_IN
    },
    {
        .name        = "serport",
```

(continues on next page)

(continued from previous page)

```
        .description = "Serial port",
        .type        = CONFIG_SERPORT
    },
    { .name = "", .description = "", .type = CONFIG_END }
};

const device_t foo_device = {
    /* ... */
    .config = foo_config
};
```

Table 6: device_config_t

Member	Description
name	Internal name for this option, used to identify it in the emulated machine's configuration file.
description	Description for this option, displayed in the user interface.
type	One of the following option types: • CONFIG_SELECTION: combobox containing a list of values specified by the selection member; • CONFIG_HEX16: combobox containing a list of 16-bit hexadecimal values (useful for ISA I/O ports) specified by the selection member; • CONFIG_HEX20: combobox containing a list of 20-bit hexadecimal values (useful for ISA memory addresses) specified by the selection member; • CONFIG_STRING: arbitrary text string entered by the user; • CONFIG_FNAME: arbitrary file path entered by the user directly or through a file selector button, with a file type filter specified by the file_filter member; • CONFIG_BINARY: checkbox; • CONFIG_INT: arbitrary integer number, currently treated as CONFIG_SELECTION by the user interface; • CONFIG_SPINNER: arbitrary integer number entered by the user directly or through up/down arrows, within a range specified by the spinner member; • CONFIG_MAC: last 3 octets of a MAC address, with a *Generate* button to randomize the octets; • CONFIG_MIDI_OUT: combobox containing a list of system MIDI output devices; • CONFIG_MIDI_IN: combobox containing a list of system MIDI input devices; • CONFIG_SERPORT: combobox containing a list of host serial ports; • CONFIG_END: **mandatory** terminator to indicate the end of the option list.
default_string	Default string value for a CONFIG_STRING option. Can be NULL if not applicable.
default_int	Default integer value for a CONFIG_SELECTION, CONFIG_HEX16, CONFIG_HEX20, CONFIG_BINARY, CONFIG_INT or CONFIG_SPINNER option. On CONFIG_FNAME options, setting this to 1 makes the file selector button bring up a save dialog instead of an open dialog. Can be 0 if not applicable.
file_filter	File type filter for a CONFIG_FNAME option. Can be NULL if not applicable. Must be specified in Windows description|mask|description|mask... format, for example: "Raw image (*.img *.ima)|*.img,*.ima|Virtual Hard Disk (*.vhd)|*.vhd"

continues on next page

Table 6 – continued from previous page

Member	Description
spinner	device_config_spinner_t sub-structure containing the minimum/maximum/step values for a CONFIG_SPINNER option. Can be { NULL } if not applicable. Members: • min: minimum selectable value. • max: maximum selectable value. • step: units to be incremented/decremented with the arrow buttons. Note that the user can still type in arbitrary numbers that are within min and max but not aligned to step.
selection	Array of device_config_selection_t sub-structures containing the choices for a CONFIG_SELECTION, CONFIG_HEX16 or CONFIG_HEX20 option. Can be { 0 } if not applicable. Must be terminated with an object with a description of "". Members: • description: description for this choice, displayed in the user interface. • value: integer value corresponding to this choice, used to identify it in the emulated machine's configuration file.

Configured option values can be read from within the device's init callback with the device_get_config_* functions. These functions automatically operate in the context of the device currently being initialized.

Note

device_get_config_* functions should **never** be called outside of a device's init callback. You are responsible for reading the options' configured values in the init callback and storing them in the device's *state structure* if necessary.

Table 7: device_get_config_string

Parameter	Description
name	The option's name. Accepted option types are CONFIG_STRING, CONFIG_FNAME and CONFIG_SERPORT.
Return value	The option's configured string value, or its default_string if no value is present. Note that a const char * is returned.

Table 8: device_get_config_int / device_get_config_hex16 / device_get_config_hex20

Parameter	Description
name	The option's name. Accepted option types are: • device_get_config_int: CONFIG_SELECTION, CONFIG_BINARY, CONFIG_INT, CONFIG_SPINNER, CONFIG_MIDI_OUT, CONFIG_MIDI_IN • device_get_config_hex16: CONFIG_HEX16 • device_get_config_hex20: CONFIG_HEX20
Return value	The option's configured integer value (CONFIG_BINARY returns 1 if checked or 0 otherwise), or its default_int if no value is present.

Table 9: device_get_config_int_ex / device_get_config_mac

Parameter	Description
name	The option's name. Accepted option types are: • device_get_config_int_ex: CONFIG_SELECTION, CONFIG_BINARY, CONFIG_INT, CONFIG_SPINNER, CONFIG_MIDI_OUT, CONFIG_MIDI_IN • device_get_config_mac: CONFIG_MAC
dflt_int	The default value to return if no configured value is present.
Return value	The option's configured integer value (CONFIG_BINARY returns 1 if checked or 0 otherwise), or dflt_int if no value is present.

3.4.2 Timers

Timers allow devices to perform tasks after a set period. This period is **automatically scaled** to match the emulation speed, which helps 86Box stay relatively accurate, unlike other emulators and virtualizers which may operate timers in real time independently of speed. Unless otherwise stated, all structures, functions and constants in this section are provided by `86box/timer.h`.

Note

Timers are processed after each CPU instruction in interpreter mode, or each recompiled code block in dynamic recompiler mode (unless an instruction requests a Time Stamp Counter (TSC) update). In both cases, timer accuracy **should** be in the single-digit microsecond range at a minimum, which is good enough for most time-sensitive applications such as 48 KHz audio.

Adding

Timers can be added with the `timer_add` function. The best place for adding a timer is in a *device*'s `init` callback, storing the `pc_timer_t` object in the *state structure*.

Code example: adding a timer

```
#include <86box/device.h>
#include <86box/timer.h>

typedef struct {
    pc_timer_t countdown_timer;
} foo_t;

/* Called once the timer period is reached. */
static void
foo_countdown_timer(void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Do whatever you want. */
}

static void *
foo_init(const device_t *info)
```

(continues on next page)

(continued from previous page)

```
{
    /* Allocate the device state structure. */
    foo_t *dev = /* ... */

    /* Add timer. */
    timer_add(&dev->countdown_timer, foo_countdown_timer, foo, 0);
}

const foo1234_device = {
  /* ... */
  .init = foo_init,
  /* ... */
};
```

Table 10: `timer_add`

Parameter	Description
`timer`	Pointer to a `pc_timer_t` object stored somewhere, usually in a device's *state structure*.
`callback`	Function called every time the timer's period is reached. Takes the form of: `void callback(void *priv)` • `priv`: opaque pointer (see `priv` below).
`priv`	Opaque pointer passed to the `callback` above, usually a pointer to a device's *state structure*.
`start_timer`	Part of the *legacy API*, should always be `0`.

Triggering

The `timer_on_auto` function can be used to start (with the provided microsecond period) or stop a timer. It can also be called from a timer callback to restart the timer:

Code example: starting, restarting and stopping a timer

```
#include <86box/timer.h>

typedef struct {
    uint8_t     regs[256];
    pc_timer_t countdown_timer; /* don't forget to timer_add on init, per the example␣
↪above */
} foo_t;

static void
foo_countdown_timer(void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Restart timer automatically if the relevant
       bit (see register description below) is set. */
    if (dev->regs[0x80] & 0x02)
        timer_on_auto(&dev->countdown_timer, 100.0);
}
```

(continues on next page)

(continued from previous page)

```
/* Our device handles I/O port register 0x__80 as such:
   - Bit 0 (0x01) set: start 100-microsecond countdown timer;
   - Bit 0 (0x01) clear: stop countdown timer;
   - Bit 1 (0x02) set: automatically restart timer. */
static void
foo_outb(uint16_t port, uint8_t val, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Handle writes to register 0x80. */
    if ((port & 0xff) == 0x80) {
        dev->regs[0x80] = val;
        if (val & 0x01) /* bit 0 set */
            timer_on_auto(&dev->countdown_timer, 100.0);
        else /* bit 0 clear */
            timer_on_auto(&dev->countdown_timer, 0.0);
    }
}
```

Table 11: `timer_on_auto`

Parameter	Description
`timer`	Pointer to the timer's `pc_timer_t` object.
`period`	Period after which the timer callback is called, in microseconds (1/1,000,000th of a second or 1/1,000th of a millisecond) as a `double`. A period of `0.0` stops the timer if it's active.

Legacy API

Existing devices may use the `timer_set_delay_u64` and `timer_advance_u64` functions, which are considered legacy and will not be documented here for simplicity. These functions used an internal 64-bit period unit, which had to be obtained by multiplying the microsecond value by the `TIMER_USEC` constant, and updated by the device's `speed_changed` callback. The new `timer_on_auto` function is much simpler, requiring no constant multiplication or updates.

3.4.3 Threads

Compute-intensive tasks can be offloaded from the main emulation flow with **threads**. Unless otherwise stated, all structures, functions and constants in this section are provided by `86box/plat.h`.

Warning

86Box API functions (excluding those in this section) are generally **not thread-safe** and must be called from the **main emulation thread**. Thread-unsafe actions (such as raising an interrupt) can be performed by the callback of a free-running *timer* which looks for data written to the device's *state structure* by a thread, as timers run on the main emulation thread.

Note

The contents of `thread_t` and other structures used by `thread_*` functions are platform-specific; therefore, pointers to those structures should be treated as opaque pointers.

Starting

Threads can be started with the `thread_create` function. Additionally, the `thread_wait` function can be used to wait for a thread's function to return.

Table 12: `thread_create`

Parameter	Description
`thread_func`	Function to run in the thread. Takes the form of: `void thread_func(void *priv)` • `priv`: opaque pointer (see `priv` below).
`priv`	Opaque pointer passed to the `thread_func` above, usually a pointer to a device's *state structure*.
Return value	`thread_t` pointer representing the newly-created thread. That pointer will become **invalid** once the thread's function returns.

Table 13: `thread_wait`

Parameter	Description
`arg`	`thread_t` pointer representing the thread to wait for.
Return value	• `0` on success; • Any other value on failure.

Events

Events allow for synchronization between threads. An event, represented by an `event_t` pointer returned by the `thread_create_event` function, can be *set* (`thread_set_event` function) or *reset* (`thread_reset_event` function), and a thread can wait for an event to be *set* with the `thread_wait_event` function. Events that are no longer to be used should be deallocated with the `thread_destroy_event` function.

Table 14: `thread_create_event`

Parameter	Description
Return value	`event_t` pointer representing the newly-created event.

Table 15: `thread_set_event` / `thread_reset_event` / `thread_destroy_event`

Parameter	Description
`arg`	`event_t` pointer representing the event to *set* (`thread_set_event`), *reset* (`thread_reset_event`) or deallocate (`thread_destroy_event`).

Table 16: `thread_wait_event`

Parameter	Description
`arg`	`event_t` pointer representing the event to wait for.
`timeout`	Maximum amount of time in **milliseconds** (not microseconds, unlike *timers*) to spend waiting for this event to be *set*. If set to `-1`, this function will not return until the event is *set*.
Return value	• `0` on success; • Any other value if `timeout` was reached or the wait otherwise failed.

Note

A `thread_wait_event` call does not *reset* the event once it is *set*; the event must be *reset* manually with `thread_reset_event`. `thread_wait_event` returns immediately if the event is already *set*.

Mutexes

Mutexes, also known as **locks**, can control access to a shared resource, ensuring no concurrent modifications or other issues arise from multiple threads attempting to use the same resource at the same time. A mutex, represented by a `mutex_t` pointer returned by the `thread_create_mutex` function, can be *locked* with the `thread_wait_mutex` function (which waits until the mutex is *released*) and *released* with the `thread_release_mutex` function. Additionally, the status of a mutex can be independently checked with the `thread_test_mutex` function. Mutexes that are no longer to be used should be deallocated with the `thread_close_mutex` function.

Table 17: `thread_create_mutex`

Parameter	Description
Return value	`mutex_t` pointer representing the newly-created mutex.

Table 18: `thread_wait_mutex` / `thread_release_mutex` / `thread_close_mutex`

Parameter	Description
`arg`	`mutex_t` pointer representing the mutex to *lock* (`thread_wait_mutex`), *release* (`thread_release_mutex`) or deallocate (`thread_close_mutex`). If this mutex is locked, `thread_wait_mutex` will not return until the mutex is *released* by another thread.

Table 19: `thread_test_mutex`

Parameter	Description
`arg`	`mutex_t` pointer representing the mutex to check.
Return value	• `0` if this mutex is *locked*; • Any other value if the mutex is *released*.

3.4.4 Port I/O

86Box handles the x86 port I/O space through **I/O handlers**. These handlers can be added with the `io_sethandler` function and removed with the `io_removehandler` function, both provided by `86box/io.h`.

Table 20: `io_sethandler` / `io_removehandler`

Parameter	Description
`base`	First I/O port (0x0000-0xffff) covered by this handler.
`size`	Amount of I/O ports (1-65536) covered by this handler, starting at `base`.

`inb`
`inw`
`inl`
	I/O read operation callback functions. Can be `NULL`. Each callback takes the form of:
	`TYPE callback(uint16_t port, void *priv)`
	• `TYPE`: operation width: `uint8_t` for `inb`, `uint16_t` for `inw`, `uint32_t` for `inl`;
	• `port`: exact I/O port being read;
	• `priv`: opaque pointer (see `priv` below);
	• Return value: 8- (`inb`), 16- (`inw`) or 32-bit (`inl`) value read from this port.

`outb`
`outw`
`outl`
	I/O write operation callback functions. Can be `NULL`. Each callback takes the form of:
	`void callback(uint16_t port, TYPE val, void *priv)`
	• `port`: exact I/O port being written;
	• `TYPE`: operation width: `uint8_t` for `outb`, `uint16_t` for `outw`, `uint32_t` for `outl`;
	• `val`: 8- (`outb`), 16- (`outw`) or 32-bit (`outl`) value being written to this port;
	• `priv`: opaque pointer (see `priv` below).

`priv`	Opaque pointer passed to this handler's read/write operation callbacks, usually a pointer to a device's *state structure*.

I/O handlers can be added or removed at any time, although `io_removehandler` must be called with the **exact same** parameters that `io_sethandler` was originally called with. For non-Plug and Play devices, you might want to add handlers in the `init` callback; for ISA Plug and Play devices, you'd add and/or remove handlers on the `config_changed` callback; for PCI devices, you'd do the same whenever the Command register or Base Address (BAR) registers are written to; and so on.

Note

There is no need to call `io_removehandler` on the device's `close` callback, since a hard reset already removes all I/O handlers.

Callback fallbacks

When an I/O handler receives an operation with a width for which it has no callback, the operation will automatically **fall back** to a lower width for which there is a callback. For example, if an `inl` operation falls on a handler which has no `inl` callback, 86Box will break the operation down to `inw` or `inb` callbacks on successive port numbers, then combine their return values:

- `inl` callback present:

```
uint32_t val = inl(port);
```

- `inl` callback not present, but `inw` callback present:

```
uint32_t val = inw(port);
val |= (inw(port + 2) << 16);
```

- `inl` and `inw` callbacks not present, but `inb` callback present:

```c
uint32_t val = inb(port);
val |= (inb(port + 1) << 8);
val |= (inb(port + 2) << 16);
val |= (inb(port + 3) << 24);
```

- `inl`, `inw` and `inb` callbacks not present:

```c
uint32_t val = 0xffffffff; /* don't care */
```

The same rule applies to write callbacks:

- `outl` callback present:

```c
uint32_t val = /* ... */;
outl(port, val);
```

- `outl` callback not present, but `outw` callback present:

```c
uint32_t val = /* ... */;
outw(port,     val & 0xffff);
outw(port + 2, (val >> 16) & 0xffff);
```

- `outl` and `outw` callbacks not present, but `outb` callback present:

```c
uint32_t val = /* ... */;
outb(port,     val & 0xff);
outb(port + 1, (val >> 8) & 0xff);
outb(port + 2, (val >> 16) & 0xff);
outb(port + 3, (val >> 24) & 0xff);
```

- `outl`, `outw` and `outb` callbacks not present:

  Don't care, no operation performed.

Note

Each broken-down operation triggers the I/O handlers for its respective port number, no matter which handlers are responsible for the starting port number. A handler will **never** receive callbacks for ports outside its `base` and `size` boundaries.

This feature's main use cases are devices which store registers that are 8-bit wide but may be accessed with 16- or 32-bit operations:

Code example: `inb` handler for reading 8-bit registers

```c
typedef struct {
    uint8_t regs[256];
} foo_t;

static uint8_t
foo_io_inb(uint16_t port, void *priv)
{
    foo_t *dev = (foo_t *) priv;
    return dev->regs[port & 0xff]; /* register index = I/O port's least significant byte␣
```

(continues on next page)

(continued from previous page)

```
↪*/
}

/* No foo_io_inw, so a 16-bit read will read two 8-bit registers in succession.
   No foo_io_inl, so a 32-bit read will read four 8-bit registers in succession. */
```

Multiple I/O handlers

Any given I/O port can have an **unlimited** amount of I/O handlers, such that:

- when a **read** operation occurs, all read callbacks will be called, and their return values will be logically **AND**ed together;
- when a **write** operation occurs, all write callbacks will be called with the same written value.

Read callbacks can effectively return "don't care" (without interfering with other handlers) by returning a value with all bits set: `0xff` for `inb`, `0xffff` for `inw` or `0xffffffff` for `inl`.

Note

The same callback fallback rules specified above also apply with multiple handlers. Handlers without callbacks for the operation's type and (same or lower) width are automatically skipped.

I/O tracing

The I/O module provides a compile-time option to unconditionally log every port operation performed by the emulated machine. Add `#define ENABLE_IO_LOG 1` to the top of `src/io.c` to enable global I/O tracing.

Trace entries are logged (to stdout by default) as such:

```
[E000:0000C0A6] (0, 1, 0001) in b(0061) = 2C
[E000:0000C0AC] (0, 0, 0000) outb(00EB, 3C)
[E000:0000C16A] (0, 2, 0002) outl(C808, 000CF000)
[E000:0000C142] (0, 1, 0002) in w(C400) = 0000
[E000:0000C15E] (0, 2, 0001) outw(C400, 0001)
```

- In brackets: current `CS:(E)IP`, pointing past the I/O instruction
- In parentheses:
  - SMM flag: **1** if the CPU is in System Management Mode, **0** otherwise
  - Handler types found: as byte widths **OR**ed together; for example, 3 indicates both 8-bit (1) and 16-bit (2) handlers were found for this port
  - Call count: total amount of callbacks called in this operation; fallbacks count as multiple calls
- Operation type: `in` or `out` followed by `b`, `w` or `l`
- Port being accessed (in hexadecimal)
- Value being read or written (in hexadecimal)

Note

I/O tracing produces a high logging volume, which can bottleneck emulation to a near halt in I/O-heavy workloads (such as POST) when outputting to a terminal. Using the `-L/--logfile` command line option to redirect logging to a file on SSD storage is highly recommended.

I/O traps

A second type of I/O handler, **I/O traps** allow a device (usually System Management Mode on chipsets or legacy compatibility mechanisms on PCI sound cards) to act upon a read/write operation to an I/O port operation without affecting its result.

Code example: I/O trap on ports `0x220-0x22f`

```
typedef struct {
    void *trap_220;
} foo_t;

static void
foo_trap_220(int size, uint16_t port, uint8_t write, uint8_t val, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Do whatever you want. */
    pclog("Foo: Trapped I/O %s to port %04X, size %d\n",
          write ? "write" : "read", port, size);
    if (write)
      pclog("Foo: Written value: %02X\n", val);
}

static void *
foo_init(const device_t *info)
{
    /* Allocate the device state structure. */
    foo_t *dev = /* ... */

    /* Add I/O trap. */
    dev->trap_220 = io_trap_add(foo_trap_220, dev);

    /* Map I/O trap to 16 ports starting at 0x220. */
    io_trap_remap(dev->trap_220, 1, 0x220, 16);

    return dev;
}

static void
foo_close(void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Remove I/O trap before deallocating the device state structure. */
    io_trap_remove(dev->trap_220);
    free(dev);
```

(continues on next page)

(continued from previous page)

```
}

const device_t foo4321_device = {
    /* ... */
    .init = foo_init,
    .close = foo_close,
    /* ... */
};
```

Table 21: `io_trap_add`

Parameter	Description
`func`	Function called whenever an I/O operation of any type or size is performed to the trap's I/O address range. Takes the form of: `void func(int size, uint16_t port, uint8_t write, uint8_t val, void *priv)` • `size`: I/O operation width: `1`, `2` or `4`; • `port`: I/O address the operation is being performed on; • `write`: `0` if this operation is a *read*, or `1` if it's a *write*; • `val`: value being written if this operation is a write; • `priv`: opaque pointer (see `priv` below).
`priv`	Opaque pointer passed to the `func` callback above, usually a pointer to a device's *state structure*.
Return value	Opaque (`void`) pointer representing the newly-created I/O trap.

Table 22: `io_trap_remap`

Parameter	Description
`trap`	Opaque pointer representing the I/O trap to remap.
`enable`	• `1` to enable this trap; • `0` to disable it.
`port`	First I/O port (0x0000-0xffff) covered by this trap.
`size`	Amount of I/O ports (1-65536) covered by this trap.

3.4.5 DMA

86Box offers two mechanisms for **Direct Memory Access**: 8237 DMA for ISA devices and direct memory read/write for PCI devices.

8237 DMA

`86box/dma.h` provides the `dma_channel_read` and `dma_channel_write` functions to read or write (respectively) a value from or to an **8237 DMA channel**.

Table 23: `dma_channel_read`

Parameter	Description
`channel`	DMA channel number: `0-3` for 8-bit channels or `5-7` for 16-bit channels.
Return value	8- (channels `0-3`) or 16-bit (channels `5-7`) value read from the given DMA channel, or `DMA_NODATA` if no data was read. May include a `DMA_OVER` bit flag (located above the most significant data bit so as to not interfere with the data) indicating that this was the last byte or word transferred, after which the channel is auto-initialized or masked depending on its configuration.

Table 24: `dma_channel_write`

Parameter	Description
`channel`	DMA channel number: `0-3` for 8-bit channels or `5-7` for 16-bit channels.
`val`	8- (channels `0-3`) or 16-bit (channels `5-7`) value to write to the given DMA channel.
Return value	• `0` on success; • `DMA_NODATA` if no data was actually written; • `DMA_OVER` if this was the last byte or word transferred, after which the channel is auto-initialized or masked depending on its configuration.

Direct memory read/write

`86box/mem.h` provides the `mem_read*_phys` and `mem_write*_phys` functions, which read or write physical memory directly. These are useful for **PCI devices**, which perform DMA on their own.

Table 25: `mem_readb_phys` / `mem_readw_phys` / `mem_readl_phys`

Parameter	Description
`addr`	32-bit memory address to read.
Return value	8- (`mem_readb_phys`), 16- (`mem_readw_phys`) or 32-bit (`mem_readl_phys`) value read from the given memory address.

Table 26: `mem_writeb_phys` / `mem_writew_phys` / `mem_writel_phys`

Parameter	Description
`addr`	32-bit memory address to write.
`val`	8- (`mem_readb_phys`), 16- (`mem_readw_phys`) or 32-bit (`mem_readl_phys`) value to write to the given memory address.

3.4.6 PCI

PCI devices are more complex than ISA devices; they are individually addressable through a **device number**, and contain a *configuration space* for configuring several aspects of the device.

Adding a device

PCI devices can be added with the `pci_add_card` function in the device's `init` callback. A PCI slot is *automatically selected* for the device according to the `add_type`; if the emulated machine runs out of slots, a **DEC 21150** PCI-PCI bridge is automatically deployed to add 9 more slots, and new devices are placed in the secondary PCI bus under it.

Code example: adding a PCI device

```
#include <86box/device.h>
#include <86box/pci.h>

typedef struct {
    int     slot;
    uint8_t pci_regs[256]; /* 256*8-bit configuration register array */
} foo_t;

static uint8_t
foo_pci_read(int func, int addr, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Ignore unknown functions. */
    if (func > 0)
        return 0xff;

    /* Read configuration space register. */
    return dev->pci_regs[addr];
}

static void
foo_pci_write(int func, int addr, uint8_t val, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Ignore unknown functions. */
    if (func > 0)
        return;

    /* Write configuration space register. */
    dev->pci_regs[addr] = val;
}

static void *
foo_init(const device_t *info)
{
    /* Allocate the device state structure. */
    foo_t *dev = /* ... */

    /* Add PCI device. */
    pci_add_card(PCI_ADD_NORMAL, foo_pci_read, foo_pci_write, dev, &dev->slot);

    return dev;
```

(continues on next page)

(continued from previous page)

```
}

const device_t foo4321_device = {
    .name = "Foo-4321",
    .internal_name = "foo4321",
    .flags = DEVICE_PCI,
    .local = 4321,
    .init = foo_init,
    /* ... */
};
```

Table 27: `pci_add_card`

Parameter	Description
`add_type`	*PCI slot type* to add this card to.
`read`	*Configuration space* register read callback. Takes the form of: `uint8_t read(int func, int addr, void *priv)` • `func`: *PCI function* number; • `addr`: configuration space register index being read; • `priv`: opaque pointer (see `priv` below); • Return value: 8-bit value read from this register index.
`write`	*Configuration space* register write callback. Takes the form of: `void write(int func, int addr, uint8_t val, void *priv)` • `func`: *PCI function* number; • `addr`: configuration space register index being written; • `val`: 8-bit value being written from this register index. • `priv`: opaque pointer (see `priv` below);
`priv`	Opaque pointer passed to this device's configuration space register read/write callbacks, usually a pointer to a device's *state structure*.
`slot`	Pointer to an `int` in the state structure, which will store a value representing the newly-added device. That value is subject to change if the PCI slot manager sees fit to move the device after it has been added.

Slot types

A machine may declare **special PCI slots** for specific purposes, such as on-board PCI devices which don't correspond to a physical slot. The `add_type` parameter to `pci_add_card` determines which kind of slot the device should be placed in:

- `PCI_ADD_NORMAL`: normal 32-bit PCI slot;
- `PCI_ADD_AGP`: AGP slot (AGP is a superset of PCI);
- `PCI_ADD_VIDEO`: on-board video controller;
- `PCI_ADD_HANGUL`: on-board supplementary language-specific video controller;
- `PCI_ADD_IDE`: on-board IDE controller;
- `PCI_ADD_SCSI`: on-board SCSI controller;
- `PCI_ADD_SOUND`: on-board sound controller;
- `PCI_ADD_MODEM`: on-board modem controller;

- `PCI_ADD_NETWORK`: on-board network controller;
- `PCI_ADD_UART`: on-board serial port controller;
- `PCI_ADD_USB`: on-board USB controller;
- `PCI_ADD_NORTHBRIDGE`, `PCI_ADD_AGPBRIDGE`, `PCI_ADD_SOUTHBRIDGE`: reserved for the chipset.

A device available both as a discrete card and as an on-board device should have different `device_t` objects with unique `local` values to set both variants apart.

Code example: device available as both discrete and on-board

```
#include <86box/device.h>
#include <86box/pci.h>

#define FOO_ONBOARD 0x80000000  /* most significant bit set = on-board */

typedef struct {
    int slot;
} foo_t;

static void *
foo_init(const device_t *info)
{
    /* Allocate the device state structure. */
    foo_t *dev = /* ... */

    /* Add PCI device. The normal variant goes in any normal slot,
       and the on-board variant goes in the on-board SCSI "slot". */
    pci_add_card((info->local & FOO_ONBOARD) ? PCI_ADD_SCSI : PCI_ADD_NORMAL,
                 foo_pci_read, foo_pci_write, dev, &dev->slot);

    return dev;
}

const device_t foo4321_device = {
    .name = "Foo-4321",
    .internal_name = "foo4321",
    .flags = DEVICE_PCI,
    .local = 4321, /* on-board bit not set */
    .init = foo_init,
    /* ... */
};

const device_t foo4321_onboard_device = {
    .name = "Foo-4321 (On-Board)",
    .internal_name = "foo4321_onboard",
    .flags = DEVICE_PCI,
    .local = 4321 | FOO_ONBOARD, /* on-board bit set */
    .init = foo_init,
    /* ... */
};
```

Configuration space

The PCI configuration space is split into a standard register set from `0x00` through `0x3f`, and device-specific registers from `0x40` through `0xff`. Not all standard registers are present or writable (partially or fully) on all devices; consult the documentation for the device you're trying to implement to determine which registers and bits are present or writable.

> **Note**
>
> The documentation for some devices may treat configuration space registers as 16- or 32-bit-wide. Since 86Box works with 8-bit-wide registers, make sure to translate all wider register offsets and bit numbers into individual bytes (in little endian / least significant byte first).

> **Important**
>
> Aside from the configuration space, devices will very often have a different set of registers in *I/O or memory space*; from now on, "registers" will refer to configuration space registers.

The most important registers in the standard set are:

Offsets	Register	Description
`0x00-0x01`	Vendor ID	Unique IDs assigned to the device's vendor (2 bytes) and the device itself (2 more bytes). The PCI ID Repository is a comprehensive repository of many (but not all) known PCI IDs.
`0x02-0x03`	Device ID	
`0x04-0x05`	Command	Control several core aspects of the PCI device: • **I/O Space** (bit 0 or `0x0001`) should enable all I/O base address registers if set, or disable them if cleared; • **Memory Space** (bit 1 or `0x0002`) should enable all memory base address registers if set, or disable them if cleared; • **Interrupt Disable** (bit 10 or `0x0400`) should prevent the device from triggering interrupts if set.
`0x0e`	Header type	Usually `0` to indicate a normal PCI header. Bit 7 (`0x80`) must be set if this is the first function (function `0`) of a *multi-function device*.
`0x10-0x27`	*Base Address Registers*	Sets the base address for each memory or *I/O* range provided by this device.
`0x2c-0x2d`	Subvendor ID	Unique vendor (2 bytes) and device (2 bytes) IDs sometimes assigned to different implementations of the same PCI device without having to change the main Vendor and Device IDs. Usually all `0` if the device doesn't call for such IDs.
`0x2e-0x2f`	Subsystem ID	
`0x30-0x33`	Expansion ROM	Base address and enable bit for the device's *option ROM*. Must be read-only if the device does not provide an option ROM.
`0x3c`	Interrupt Line	The PIC IRQ number assigned to this device's *interrupt pin* (see `Interrupt Pin` below). While this register's contents should not be used by the device, the register itself **must be writable** if the device uses interrupts.
`0x3d`	Interrupt Pin	Read-only value indicating the PCI *interrupt pin* (`INTx#`) used by this device: • `0` if the device does not use interrupts; • `PCI_INTA` to indicate the `INTA#` pin is used (most devices use this); • `PCI_INTB` to indicate the `INTB#` pin is used; • `PCI_INTC` to indicate the `INTC#` pin is used; • `PCI_INTD` to indicate the `INTD#` pin is used.

Multi-function devices

PCI defines the concept of **functions**, which allow a physical device to contain up to 8 sub-devices (numbered from `0` to `7`), each with their **own configuration space**, and their **own resources** controlled by *Base Address Registers*. Most (but not all) multi-function PCI devices are chipset southbridges, which may implement a function for the PCI-ISA bridge (and general configuration), another one for the IDE controller, one or more for USB and so on.

The `func` parameter passed to a device's configuration space read/write callbacks provides the **function number** for which the configuration space is being accessed. There are two main requirements for implementing multi-function devices:

1. The first function (function `0`) must have bit 7 (`0x80`) of the Header Type (`0x0e`) register set;
2. Unused functions must return `0xff` on all configuration register reads and should ignore writes.

Code example: device with two functions

```
typedef struct {
    int     slot;
    uint8_t pci_regs[2][256]; /* two 256*8-bit configuration register arrays,
                                 one for each function */
} foo_t;

static uint8_t
foo_pci_read(int func, int addr, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Read configuration space register on the given function. */
    switch (func) {
        case 0: /* function 0 */
            return dev->pci_regs[0][addr];

        case 1: /* function 1 */
            return dev->pci_regs[1][addr];

        default: /* out of range */
            return 0xff;
    }
}

static void
foo_pci_write(int func, int addr, uint8_t val, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Write configuration space register on the given function. */
    switch (func) {
        case 0: /* function 0 */
            dev->pci_regs[0][addr] = val;
            break;

        case 1: /* function 1 */
            dev->pci_regs[1][addr] = val;
```

(continues on next page)

(continued from previous page)

```
            break;

        default: /* out of range */
            break;
    }
}

static void
foo_reset(void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Reset PCI configuration registers. */
    memset(dev->pci_regs[0], 0, sizeof(dev->pci_regs[0]));
    memset(dev->pci_regs[1], 0, sizeof(dev->pci_regs[1]));

    /* Write default vendor IDs, device IDs, etc. */

    /* Flag this device as multi-function. */
    dev->pci_regs[0][0x0e] = 0x80;
}

static void *
foo_init(const device_t *info)
{
    /* Allocate the device state structure. */
    foo_t *dev = /* ... */

    /* Add PCI device. No changes are required here for multi-function devices. */
    pci_add_card(PCI_ADD_NORMAL, foo_pci_read, foo_pci_write, dev, &dev->slot);

    /* Initialize PCI configuration registers. */
    foo_reset(dev);

    return dev;
}

const device_t foo4321_device = {
    /* ... */
    .init = foo_init,
    .reset = foo_reset,
    /* ... */
};
```

Base Address Registers

Each function may contain up to six **Base Address Registers** (BARs), which determine the base and size of a **memory** or **I/O** resource provided by the device. The base address may be set by the BIOS and/or operating system during boot. Each 4-byte BAR has two parts:

- The most significant bits store the resource's base address, **aligned** to its size;
- The least significant bits are **read-only** flags related to the BAR:

- Bit 0 is the **resource type**: `0` for memory or 1 for *I/O*;
- Bits 1-3 on memory BARs are **positioning flags** not really relevant to the context of 86Box;
- Bit 1 on I/O BARs is **reserved** and must be `0`.

The aforementioned base address alignment allows software (BIOSes and operating systems) to tell how big a BAR resource is, by checking how many base address bits are writable. All bits ranging from the end of the flags to the start of the base address must be read-only and always read `0`; for example, on a memory BAR that is 4 KB (4096 bytes) large, bits 31-12 must be writable (creating a 4096-byte alignment), bits 11-4 must read `0`, and bits 3-0 must read the BAR flags.

Note

The minimum BAR sizes are 4 KB for memory and 4 ports for I/O. While memory BARs can technically be as small as 16 bytes, 86Box can only handle device memory in aligned 4 KB increments.

Table 28: Memory BAR (example: 4 KB large, starting at `0x10`)

`0x13`								`0x12`								`0x11`								`0x10`							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Base memory address (4096-byte aligned)																				Always `0`								FLAGS (Read-only)			`0`

Table 29: I/O BAR (example: 64 ports large, starting at `0x14`)

`0x17`								`0x16`								`0x15`								`0x14`							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Ignored (`0` recommended)																Base *I/O port* (64-byte aligned)										Always `0`				R (R ser (re on	1

Code example: memory and I/O BARs described above

```c
#include <86box/io.h>
#include <86box/mem.h>

typedef struct {
    uint8_t       pci_regs[256];
    uint16_t      io_base;
    mem_mapping_t mem_mapping;
} foo_t;

static void
foo_remap_mem(foo_t *dev)
{
    if (dev->pci_regs[0x04] & 0x02) {
        /* Memory Space bit set, apply the base address.
           Least significant bits are masked off to maintain 4096-byte alignment.
```

(continues on next page)

(continued from previous page)

```
           We skip reading dev->pci_regs[0x10] as it contains nothing of interest. */
        mem_mapping_set_addr(&dev->mem_mapping,
                             ((dev->pci_regs[0x11] << 8) | (dev->pci_regs[0x12] << 16) |␣
↪(dev->pci_regs[0x13] << 24)) & 0xfffff000,
                             4096);
    } else {
        /* Memory Space bit not set, disable the mapping. */
        mem_mapping_set_addr(&dev->mem_mapping, 0, 0);
    }
}

static void
foo_remap_io(foo_t *dev)
{
    /* Remove existing I/O handler if present. */
    if (dev->io_base)
        io_removehandler(dev->io_base, 64,
                         foo_io_inb, foo_io_inw, foo_io_inl,
                         foo_io_outb, foo_io_outw, foo_io_outl, dev);

    if (dev->pci_regs[0x04] & 0x01) {
        /* I/O Space bit set, read the base address.
           Least significant bits are masked off to maintain 64-byte alignment. */
        dev->io_base = (dev->pci_regs[0x14] | (dev->pci_regs[0x15] << 8)) & 0xffc0;
    } else {
        /* I/O Space bit not set, don't do anything. */
        dev->io_base = 0;
    }

    /* Add new I/O handler if required. */
    if (dev->io_base)
        io_sethandler(dev->io_base, 64,
                      foo_io_inb, foo_io_inw, foo_io_inl,
                      foo_io_outb, foo_io_outw, foo_io_outl, dev);
}

static void
foo_pci_write(int func, int addr, uint8_t val, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Ignore unknown functions. */
    if (func > 0)
        return;

    /* Write configuration space register. */
    switch (addr) {
        case 0x04:
            /* Our device only supports the I/O and Memory Space bits of the Command␣
↪register. */
            dev->pci_regs[addr] = val & 0x03;
```

(continues on next page)

(continued from previous page)

```
            /* Update memory and I/O spaces. */
            foo_remap_mem(dev);
            foo_remap_io(dev);
            break;

        case 0x10:
            /* Least significant byte of the memory BAR is read-only. */
            break;

        case 0x11:
            /* 2nd byte of the memory BAR is masked to maintain 4096-byte alignment. */
            dev->pci_regs[addr] = val & 0xf0;

            /* Update memory space. */
            foo_remap_mem(dev);
            break;

        case 0x12: case 0x13:
            /* 3rd and most significant bytes of the memory BAR are fully writable. */
            dev->pci_regs[addr] = val;

            /* Update memory space. */
            foo_remap_mem(dev);
            break;

        case 0x14:
            /* Least significant byte of the I/O BAR is masked to maintain 64-byte␣
↪alignment, and
               ORed with the default value's least significant bits so that the flags␣
↪stay in place. */
            dev->pci_regs[addr] = (val & 0xc0) | (dev->pci_regs[addr] & 0x03);

            /* Update I/O space. */
            foo_remap_io(dev);
            break;

        case 0x15:
            /* Most significant byte of the I/O BAR is fully writable. */
            dev->pci_regs[addr] = val;

            /* Update I/O space. */
            foo_remap_io(dev);
            break;

        case 0x16: case 0x17:
            /* I/O BARs are only 2 bytes long, ignore the rest. */
            break;
    }
}

static void
```

(continues on next page)

(continued from previous page)

```
foo_reset(void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) dev;

    /* Reset PCI configuration registers. */
    memset(dev->pci_regs, 0, sizeof(dev->pci_regs));

    /* Write default vendor ID, device ID, etc. */

    /* The BAR at 0x10-0x13 is a memory BAR. */
    //dev->pci_regs[0x10] = 0x00; /* least significant bit already not set = memory */

    /* The BAR at 0x14-0x17 is an I/O BAR. */
    dev->pci_regs[0x14] = 0x01; /* least significant bit set = I/O */

    /* Clear all BAR memory mappings and I/O handlers. */
    //dev->pci_regs[0x04] = 0x00; /* Memory and I/O Space bits already cleared */
    foo_remap_mem(dev);
    foo_remap_io(dev);
}

/* Don't forget to add the PCI device on init first. */

const device_t foo4321_device = {
    /* ... */
    .reset = foo_reset,
    /* ... */
};
```

Option ROM

A PCI function may have an **option ROM**, which behaves similarly to a *memory BAR* in that the ROM can be mapped to any address in 32-bit memory space, aligned to its size. As with BARs, the BIOS and/or operating system takes care of mapping; for example, a BIOS will map the primary PCI video card's ROM to the legacy `0xc0000` address.

The main difference between this register and BARs is that the ROM can be enabled or disabled through bit 0 (`0x01`) of this register. Both that bit and the Command (`0x04`) register's Memory Space bit (bit 1 or `0x02`) must be set for the ROM to be accessible.

Note

The minimum size for an option ROM is 4 KB (see the note about 86Box memory limitations in the *BAR* section), and the maximum size is 16 MB.

Table 30: Option ROM (example: 32 KB large)

0x33								0x32								0x31								0x30							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Base memory address (32768-byte aligned)																	Always `0`														E (ROM Enable)

Code example: 32 KB option ROM

```
#include <86box/mem.h>
#include <86box/rom.h>

typedef struct {
    uint8_t pci_regs[256];
    rom_t   rom;
} foo_t;

static void
foo_remap_rom(foo_t *dev)
{
    if ((dev->pci_regs[0x30] & 0x01) && (dev->pci_regs[0x04] & 0x02)) {
        /* Expansion ROM Enable and Memory Space bits set, apply the base address.
           Least significant bits are masked off to maintain 32768-byte alignment.
           We skip reading dev->pci_regs[0x30] as it contains nothing of interest. */
        mem_mapping_set_addr(&dev->rom.mapping,
                             ((dev->pci_regs[0x31] << 8) | (dev->pci_regs[0x32] << 16) | (dev->pci_regs[0x33] << 24)) & 0xffff8000,
                             4096);
    } else {
        /* Expansion ROM Enable and/or Memory Space bits not set, disable the mapping. */
        mem_mapping_set_addr(&dev->rom.mapping, 0, 0);
    }
}

static void
foo_pci_write(int func, int addr, uint8_t val, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Ignore unknown functions. */
    if (func > 0)
        return;

    /* Write configuration space register. */
    switch (addr) {
        case 0x04:
            /* Our device only supports the Memory Space bit of the Command register. */
            dev->pci_regs[addr] = val & 0x02;

            /* Update ROM space. */
```

(continues on next page)

(continued from previous page)

```
            foo_remap_rom(dev);
            break;

        case 0x30:
            /* Least significant byte of the ROM address is read-only, except for the␣
↪enable bit. */
            dev->pci_regs[addr] = val & 0x01;

            /* Update ROM space. */
            foo_remap_rom(dev);
            break;

        case 0x31:
            /* 2nd byte of the ROM address is masked to maintain 32768-byte alignment. */
            dev->pci_regs[addr] = val & 0x80;

            /* Update ROM space. */
            foo_remap_rom(dev);
            break;

        case 0x32: case 0x33:
            /* 3rd and most significant bytes of the ROM address are fully writable. */
            dev->pci_regs[addr] = val;

            /* Update ROM space. */
            foo_remap_rom(dev);
            break;
    }
}

static void
foo_reset(void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) dev;

    /* Reset PCI configuration registers. */
    memset(dev->pci_regs, 0, sizeof(dev->pci_regs));

    /* Write default vendor ID, device ID, etc. */

    /* Clear ROM memory mapping. */
    //dev->pci_regs[0x04] = 0x00; /* Memory Space bit already cleared */
    //dev->pci_regs[0x30] = 0x00; /* Expansion ROM Enable bit already cleared */
    foo_remap_rom(dev);
}

static int
foo4321_available()
{
    /* This device can only be used if its ROM is present. */
    return rom_present("roms/scsi/foo/foo4321.bin");
```

(continues on next page)

(continued from previous page)

```
}

static void *
foo_init(const device_t *info)
{
    /* Allocate the device state structure. */
    foo_t *dev = /* ... */

    /* Don't forget to add the PCI device first. */

    /* Load 32 KB ROM... */
    rom_init(&dev->rom, "roms/scsi/foo/foo4321.bin", 0, 0x8000, 0x7fff, 0, MEM_MAPPING_
↪EXTERNAL);

    /* ...but don't map it right now. */
    mem_mapping_disable(&dev->rom.mapping);

    /* Initialize PCI configuration registers. */
    foo_reset(dev);

    return dev;
}

const device_t foo4321_device = {
    /* ... */
    .init = foo_init,
    .reset = foo_reset,
    { .available = foo4321_available },
    /* ... */
};
```

Interrupts

PCI devices can assert an interrupt on one of four **interrupt pins** called `INTA#`, `INTB#`, `INTC#` and `INTD#`. Each function can only use one of these pins, specified by read-only register `0x3d`. Each pin is connected to a system-wide **interrupt lane** (most chipsets provide 4 lanes), which is then routed to a PIC or APIC IRQ at boot time by the BIOS and/or operating system, through a process called **steering**. Different interrupt pins on different devices may share the same lane, and more than one lane may share the same PIC IRQ (or APIC IRQ if the APIC has no dedicated PCI interrupt inputs).

The diagram below exemplifies a system with **interrupt steering performed by the chipset**. Early PCI chipsets are not capable of steering by themselves, instead requiring interrupt lanes to be manually routed to PIC IRQs using **jumpers** and the BIOS to be configured accordingly. On machines with non-steering-capable chipsets, 86Box skips the jumpers and uses the IRQs configured in the BIOS; this is done by snooping on the values the BIOS writes to register `0x3c`.

An emulated PCI device can assert or de-assert an interrupt on any pin with the `pci_set_irq` and `pci_clear_irq` functions respectively. The PCI subsystem transparently handles interrupt lane routing (using the per-machine PCI slot table), sharing and steering. Once an interrupt is asserted, a device usually de-asserts it when an **interrupt flag** is cleared or an **interrupt mask flag** is set in its configuration, I/O or memory register space.

Code example: PCI interrupts

```
#include <86box/pci.h>
```

(continues on next page)

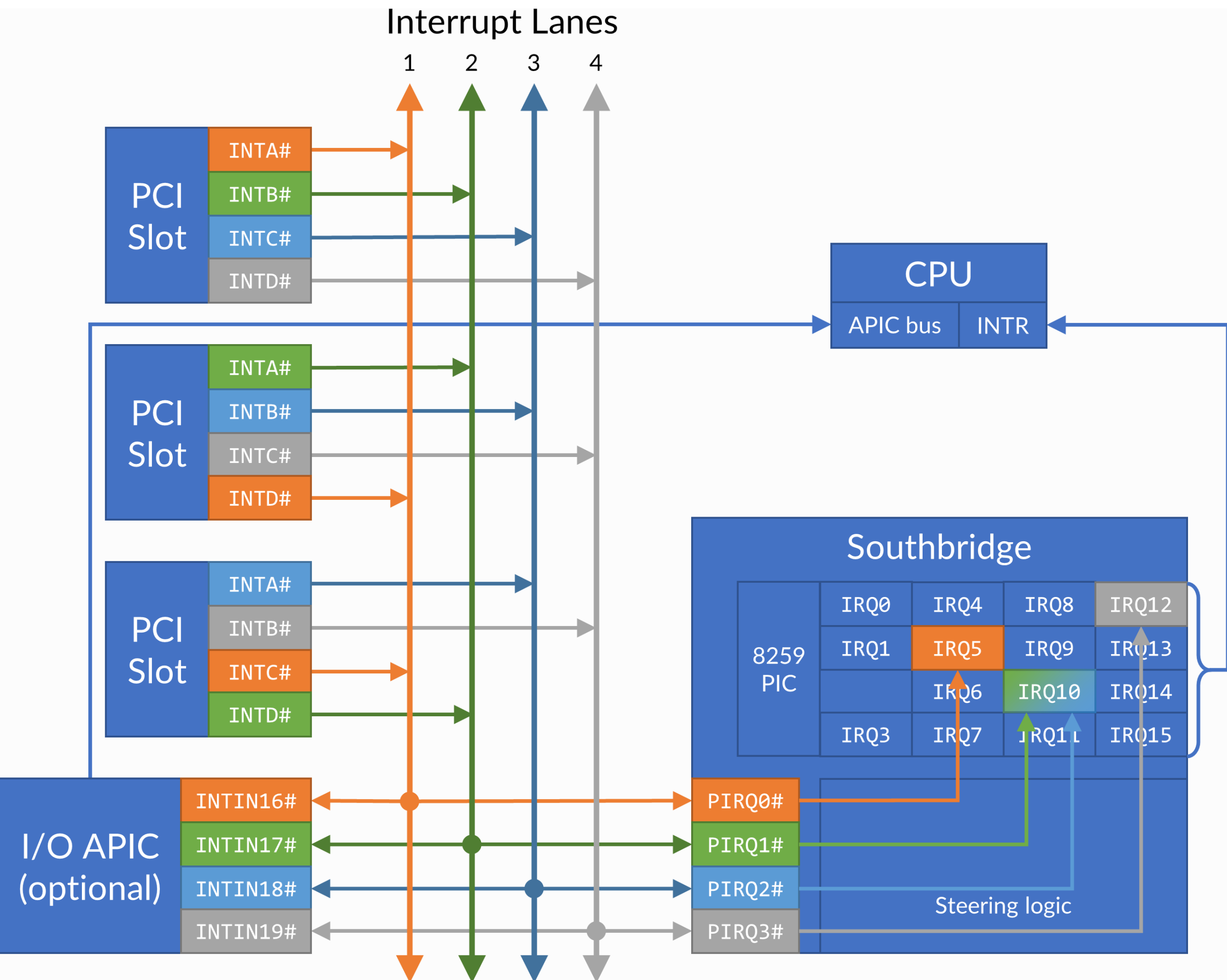


Fig. 2: PCI interrupt topology example. IRQ 10 is shared by two interrupt lanes. The machine is free to route any `INTx#` pin to any lane (sequentially or not), and free to route any lane to any available IRQ.

(continued from previous page)

```
typedef struct {
    int     slot;
    uint8_t irq_state;
    uint8_t pci_regs[256];
} foo_t;

static void
foo_pci_write(int func, int addr, uint8_t val, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Ignore unknown functions. */
    if (func > 0)
        return;

    /* The Interrupt Line register must be writable for 86Box to
       know the IRQ to use on machines with non-steering chipsets. */
    if (addr == 0x3c) {
        dev->pci_regs[0x3c] = val;
        return;
    }

    /* Example: PCI configuration register 0x40:
       - Bit 0 (0x01) set: manually assert interrupt;
       - Bit 0 (0x01) clear: de-assert interrupt. */
    if (addr == 0x40) {
        dev->pci_regs[0x40] = val;
        if (val & 0x01)
            pci_set_irq(dev->slot, PCI_INTA, &dev->irq_state);
        else
            pci_clear_irq(dev->slot, PCI_INTA, &dev->irq_state);
    }
}

static void
foo_reset(void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) dev;

    /* Reset PCI configuration registers. Clearing active
       interrupts is left as an exercise to the reader. */
    memset(dev->pci_regs, 0, sizeof(dev->pci_regs));

    /* Write default vendor ID, device ID, etc. */

    /* Our device uses the INTA# interrupt line. */
    dev->pci_regs[0x3d] = PCI_INTA;
}
```

(continues on next page)

(continued from previous page)

```
/* Don't forget to add the PCI device on init first. This example uses
   the slot value, which is provided to pci_add_card as a pointer. */

const device_t foo4321_device = {
    /* ... */
    .reset = foo_reset,
    /* ... */
};
```

Table 31: `pci_set_irq` / `pci_clear_irq`

Parameter	Description
`slot`	Value representing this PCI device, stored in the `int` passed to `pci_add_card`.
`pci_int`	Interrupt pin to assert (`pci_set_irq`) or de-assert (`pci_clear_irq`): `PCI_INTA`, `PCI_INTB`, `PCI_INTC` or `PCI_INTD`.
`irq_state`	Pointer to an `uint8_t` in the state structure, which is used internally by the interrupt manager. The stored value is opaque and should not be used or changed.

Motherboard interrupts

Some chipsets may provide steerable **motherboard IRQ** (MIRQ) lines for on-board devices to use. The amount of available lines depends on the chipset, and the purposes for those lines depend on the machine. 86Box supports up to 8 MIRQ lines, which can be asserted or de-asserted with the `pci_set_mirq` and `pci_clear_mirq` functions respectively.

Table 32: `pci_set_mirq` / `pci_clear_mirq`

Parameter	Description
`mirq`	MIRQ line to assert (`pci_set_mirq`) or de-assert (`pci_clear_mirq`): `PCI_MIRQ0` through `PCI_MIRQ7`.
`level`	`1` if this MIRQ should be level-triggered, `0` if it should be edge-triggered.
`irq_state`	Pointer to an `uint8_t` in the state structure, which is used internally by the interrupt manager. The stored value is opaque and should not be used or changed.

3.5 File formats

86Box introduces new file formats for disk images and other purposes. These formats are documented on this section.

3.5.1 86F

A floppy disk surface image format which stores data in FM- or MFM-encoded transitions.

Specification for v2.12

All offsets are in hexadecimal.

```
00000000: Magic 4 bytes ("86BF")
00000004: Minor version (0C)
00000005: Major version (02)
00000006: Disk flags (16-bit)
    Bit 0           Has surface description data (1 = yes, 0 = no)
                    This data indicates if the corresponding bit on the FM/MFM
                    encoded surface is a normal bit or a special bit (weak bit
                    or hole, depending on the other bit):
                        0 = The corresponding FM/MFM encoded surface bit is normal
                        1 = The corresponding FM/MFM encoded surface bit is either
                            a weak bit or a hole:
                                Corresponding FM/MFM encoded bit is 0:
                                    Hole (noise on read, not overwritable)
                                Corresponding FM/MFM encoded bit is 1:
                                    Weak bit (noise on read, overwritable)
    Bits 2, 1       Hole (3 = ED + 2000 kbps, 2 = ED, 1 = HD, 0 = DD)
    Bit 3           Sides (1 = 2 sides, 0 = 1 side)
    Bit 4           Write protect (1 = yes, 0 = no)
    Bits 6, 5       RPM slowdown (3 = 2%, 2 = 1.5%, 1 = 1%, 0 = 0%)
    Bit 7           Bitcell mode (1 = Extra bitcells count specified after
                                  disk flags, 0 = No extra bitcells)
                    The maximum number of extra bitcells is 1024 (which
                    after decoding translates to 64 bytes)
    Bit 8           Disk type (1 = Zoned, 0 = Fixed RPM)
    Bits 10, 9      Zone type (3 = Commodore 64 zoned, 2 = Apple zoned,
                               1 = Pre-Apple zoned #2, 0 = Pre-Apple zoned #1)
                    Ignore if disk type is 0 (fixed RPM)
    Bit 11          Data and surface bits are stored in reverse byte endianness
    Bit 12          If set:
                        If bits 6, 5 are not 0, they specify % of speedup instead
                        of slowdown;
                        If bits 6, 5 are 0, and bit 7 is 1, the extra bitcell count
                        specifies the entire bitcell count
                    For converting other stuff to 86F, I recommend to set this bit
                    and bit 7 and clear bits 6 and 5,
                    and just specify the entire bitcell count.
00000008: Offsets of tracks
    Note that thick-track (eg. 360k) disks will have (tracks * 2) tracks, with each
    pair of tracks being identical to each other.
    Each side of each track is stored as its own track, in order (so, track 0 side 0,
    track 0 side 1, track 1 side 0, track 1 side 0, etc.).
```

(continues on next page)

(continued from previous page)

```
Track offset + 00000000: Track flags (16-bit)
    Bits 7, 6, 5    RPM:
                        000 = 300 rpm
                        001 = 360 rpm
    Bits 4, 3       Encoding:
                        00 = FM
                        01 = MFM
                        10 = M2FM
                        11 = GCR
    Bits 2, 1, 0    Bit rate, if encoding is MFM:
                        000 = 500 kbps
                        001 = 300 kbps
                        010 = 250 kbps
                        011 = 1000 kbps
                        101 = 2000 kbps
                    If encoding is FM, the bit rate is half that.
If the bitcell count is present:
    Track offset + 00000002: Extra (or total, depending on disk flags) bit cells count␣
↪(32-bit)
                             If this specifies extra bit cells rather than total, it is␣
↪a signed
                             integer, and when negative, makes the track smaller.
    Track offset + 00000006: Bit cell where index hole is (32-bit)
    Track offset + 0000000A: FM/MFM/M2FM/GCR-encoded data (track length bytes)
    Track offset + 0000000A + track length: Surface description data if present (track␣
↪length bytes)
Else:
    Track offset + 00000002: Bit cell where index hole is (32-bit)
    Track offset + 00000006: FM/MFM/M2FM/GCR-encoded data (track length bytes)
    Track offset + 00000006 + track length: Surface description data if present (track␣
↪length bytes)

Track lengths if the bitcell count is not present or it does not represent total bit␣
↪cells:
    Hole 0 (DD) or 1 (HD):
        2.0% RPM slowdown:         12750 words
        1.5% RPM slowdown:         12687 words
        1.0% RPM slowdown:         12625 words
        0.0% RPM slowdown/speedup: 12500 words
        1.0% RPM speedup :         12376 words
        1.5% RPM speedup :         12315 words
        2.0% RPM speedup :         12254 words
    Hole 2 (ED):
        2.0% RPM slowdown:         25250 words
        1.5% RPM slowdown:         25375 words
        1.0% RPM slowdown:         25250 words
        0.0% RPM slowdown/speedup: 25000 words
        1.0% RPM speedup :         24752 words
        1.5% RPM speedup :         24630 words
        2.0% RPM speedup :         24509 words
    Hole 3 (ED + 2000 kbps):
        2.0% RPM slowdown:         51000 words
```

(continues on next page)

(continued from previous page)

```
        1.5% RPM slowdown:          50750 words
        1.0% RPM slowdown:          50500 words
        0.0% RPM slowdown/speedup:  50000 words
        1.0% RPM speedup :          49504 words
        1.5% RPM speedup :          49261 words
        2.0% RPM speedup :          49019 words
    1 word = 2 bytes (so 16 bits)
    If extra bit cells count is present and it indicates extra bit cells count:
        Track length = (Track length << 4) + Extra bitcells count
        If (Track length & 15)
            Track length + (Track length >> 4) + 1
        Else
            Track length + (Track length >> 4)
    If extra bit cells count is present and it indicates total bit cells count,
    then the total bit cells count become the track length, padded upwards to the
    nearest word in the file.
```

3.5.2 Extended HDI (HDX)

A derivative of the *Japanese FDI* disk image format, with a more compact header as well as support for images larger than 4 GB.

Specification

All offsets are in hexadecimal. The `[Translation]` values are for future use.

```
00000000: Signature (59 54 44 44 20 A8 78 D7 / "YTDD " A8 78 D7)
00000008: Full size of the data in bytes (64-bit)
00000010: Sector size in bytes (32-bit)
00000014: Sectors per cylinder (32-bit)
00000018: Heads per cylinder (32-bit)
0000001C: Cylinders (32-bit)
00000020: [Translation] Sectors per cylinder (32-bit)
00000024: [Translation] Heads per cylinder (32-bit)
00000028: Raw data (size set in offset 00000008)
```