86Box

86Box Project

Apr 16, 2023

USING 86BOX

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 86Box Manager, our officially endorsed 86Box configuration manager, developed by Overdoze (daviunic).

CHAPTER

ONE

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!

CHAPTER

TWO

CONTENTS

2.1 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.

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 extension 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 ZIP 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 `F8 + F12` simultaneously (on Windows) or `Ctrl + End` (on Linux). 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 86Box Manager 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.

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 here on GitHub or joining official support channels on Discord or IRC.

Have fun!

2.2 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 extension cards.

The ROM set is 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.

2.2.1 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.

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/net.86box.86Box/roms`
2. `/Library/Application Support/net.86box.86Box/roms`

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

2.3 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.

2.3.1 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 86Box 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.
- **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.
- **Ctrl+Alt+Del:** send a *Ctrl+Alt+Del* key combination to the emulated machine. You can alternatively press *Ctrl+F12* to send that combination.
- **Ctrl+Alt+Esc:** send a *Ctrl+Alt+Esc* key combination to the emulated machine.
- **Pause:** pause emulation of the machine. Uncheck this option to resume emulation.
- **Exit:** quit 86Box. Requires confirmation, which can be disabled by checking the *Don't show this message again* box.

2.3.2 View

- **Hide status bar:** hides the *status bar* at the bottom of the window.
- **Hide toolbar:** hides the *toolbar* below the menu bar.
- **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 size and position of the 86Box window to the emulated machine's configuration file.
- **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.
  - **Target framerate:** select the framerate at which the emulated display is updated. *Synchronize with video* automatically uses the emulated display's current refresh rate.
  - **VSync:** enable vertical sync. Recommended if tearing artifacts are observed.
  - **Browse:** load a `.glsl` shader file to apply to the emulated display.
  - **Remove:** disable the currently-loaded shader.

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.

- Shaders that take advantage of multipass and previous frames are not supported.
- `.cg` and `.cgp` shaders are not supported either, as these formats are long deprecated.

- **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.
- **Window scale factor:** scale the emulated display to half (*0.5x*), normal (*1x*), 50% larger (*1.5x*) or double (*2x*) sizes.
- **Filter method:** select the filtering method (*Nearest* or *Linear*) to be used when scaling the emulated display.
- **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.

- **Fullscreen:** enter full screen mode. Press *Ctrl+Alt+Page Down* to go back to windowed mode. You can also enter full screen mode by pressing *Ctrl+Alt+Page Up*.
- **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.
- **EGA/(S)VGA settings:** contains display settings specific to EGA, VGA and Super VGA video hardware.
  - **Inverted VGA monitor:** emulate a VGA monitor with inverted colors.
  - **VGA screen type:** select the VGA monitor type to emulate. *Color*, *Grayscale*, *Amber*, *Green* and *White* phosphor monitors can be selected.
  - **Grayscale conversion type:** select the color-to-grayscale conversion profile (*BT.601*, *BT.709* or *Average*) to use when a grayscale monitor is selected.
- **CGA/PCjr/Tandy/EGA/(S)VGA overscan:** add an overscan border around the display. This border is only added when emulating the specified video hardware types.
- **Change contrast for monochrome display:** optimize the contrast of monochrome CGA monitors for 4-color operation.

2.3.3 Media

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

2.3.4 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.
- **Preferences:** open the *Preferences* window, which provides the following options:
  - **Language:** select a language for the 86Box user interface.
  - **Icon set:** select an icon theme for the *status bar* and *Settings window*.
- **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 x86 or x64 Windows, `x86_64` Linux or Intel 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. Screenshots are saved as .png images in the `screenshots` subdirectory found in the emulated machine's directory.
- **Sound gain:** open the *sound gain control*, which is also accessible through the status bar.

2.3.5 Help

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

2.4 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.

2.4.1  Pause/resume execution

Pause emulation of the machine. Press again to resume emulation.

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

2.4.2  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.

2.4.3  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.

2.4.4  Settings

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

2.4.5 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.

2.5 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. Additionally, a green indicator light will appear on an icon when its device is in use, unless *Update status bar icons* is disabled.

2.5.1  Cassette deck

A cassette tape icon will appear if *IBM cassette emulation* is enabled.

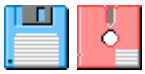
- **New image:** create a new cassette tape image file.
- **Existing image:** insert a *cassette tape image file* into the deck.
- **Existing image (Write-protected):** insert a cassette tape image file into the deck as a read-only tape.
- **Record:** start recording data to the cassette tape. Not available 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.

2.5.2  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.
- **Eject:** remove the currently-inserted cartridge from this slot.

2.5.3  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.
- **Existing image (Write-protected):** insert a disk image file into this drive as a read-only disk.
- **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.

2.5.4  CD-ROM drives

A CD icon will appear for each configured *CD-ROM drive*.

- **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.
- **Empty:** remove any disc inserted into this drive.
- **Reload previous image:** reinsert the last disc image file selected through the *Image* option.
- **Image:** insert a *CD-ROM or DVD-ROM disc image file* into this drive.

2.5.5  ZIP and MO drives

A ZIP or MO 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.
- **Existing image (Write-protected):** insert a disk image file into this drive as a read-only disk.
- **Eject:** remove the currently-inserted disk from this drive.
- **Reload previous image:** reinsert the last disk image file selected through the *Existing image* options.

2.5.6  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.

2.5.7  Network

This icon will appear if *networking* is enabled. No options are available.

2.5.8  Sound

This icon is always present. Double-clicking it opens a sound gain control, which allows you to increase the loudness of all audio produced by the emulated machine's PC speaker, *sound cards* and other sound hardware.

Note: The gain control does not apply to MIDI music sent to a software synthesizer through the *System MIDI* device, as these synthesizers are external to 86Box.

2.5.9 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*.

Monitor sleep mode

The *Monitor in sleep mode* message will be 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.

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. If both the ISABugger and the POST card are enabled simultaneously, the POST card takes over whenever a POST code is reported, and the ISABugger takes over whenever one of its registers is written to. The *Monitor in sleep mode* message is high-priority and will override all other components.

2.6 Settings

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

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

2.6.1  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 type / Machine

Machine/motherboard model to emulate. The *Machine* box lists all available models for the machine class selected on the *Machine type* box.

The *Configure* button opens a new window with settings specific to the machine's onboard devices, such as the amount of installed video memory for an onboard video chip.

CPU type / Speed

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.

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.

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.

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.

2.6.2  Display

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

Video

Video card to emulate. This box only lists cards supported by the machine's expansion buses. On machines equipped with an onboard video chip, the *Internal* option enables the onboard video.

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

Voodoo Graphics

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

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.

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. For these cards, the *Configure* button next to the *Video* box provides similar settings to the ones listed here.

- **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.
- **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.
- **Recompiler:** enable the Voodoo recompiler for faster emulation.

8514/A

Emulate an **IBM 8514/A** add-on graphics accelerator. Both the original IBM card for the MCA bus and a generic clone card for the ISA bus are available; the correct card is automatically selected based on the machine's supported expansion buses.

Note: Pairing the 8514/A with a video card from **S3** may result in compatibility issues, as those cards implement a subset of the 8514/A's features.

2.6.3  Input devices

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

Mouse

Emulate a pointing device. The following mouse 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. The selected serial port must be enabled on the *Ports tab*.
- **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 mouse type, such as the number of buttons, or the serial port for a serial mouse.

Joystick

Emulate one or more game port controller(s). The following controller types are supported:

- **None:** no controller connected.
- **2-axis, 2-button joystick(s):** up to two controllers, each with two buttons and an analog stick.
- **2-axis, 4-button joystick:** single controller with four buttons and an analog stick.
- **3-axis, 2-button joystick:** single controller with two buttons and an analog stick and a throttle.
- **3-axis, 4-button joystick:** single controller with four buttons and an analog stick and a throttle.
- **2-axis, 6-button joystick:** single controller with four regular buttons, two additional buttons mapped to the third and fourth axes, and an analog stick.
- **2-axis, 8-button joystick:** single controller with four regular buttons, four additional buttons mapped to the third and fourth axes, and an analog stick.
- **4-axis 4-button joystick:** single controller with four buttons and two analog sticks (or four axes).
- **CH Flightstick Pro:** flight controller with four buttons, three axes and a POV hat.
- **Microsoft SideWinder Pad:** up to four controllers, each with 10 buttons and a directional pad. Not compatible with standard game port joysticks; requires a driver in the emulated machine.
- **Thrustmaster Flight Control System:** flight controller with four buttons, two 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 Plug and Play or PCI sound card) to accomodate the selected controller. This generic card uses the default I/O ports 200-207h, which can be changed through ISA Plug and Play. On IBM PS/1 machines, the generic card uses port 201h only and has no Plug and Play capability.

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 real controller.

If you're not sure as to what axis or button numbers map to which sticks and buttons on the real controller, use the *Test* feature of Windows' *Game Controllers* control panel (`joy.cpl`). Keep in mind 86Box's button numbers start with 0, whereas the control panel's numbers start with 1.

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

2.6.4  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*.

Sound card

Sound card to emulate. Only cards supported by the machine's expansion buses will be listed. On machines equipped with an onboard sound chip, the *Internal* option enables the onboard 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 OPL3 library.

MIDI Out Device

Device to output MIDI music to, for sound cards equipped with an external MIDI output.

- **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, or (on Windows hosts) if the included `libfluidsynth.dll` or `libfluidsynth64.dll` file is missing from the 86Box directory.
- **Roland MT-32/CM-32L Emulation:** emulate a Roland synthesizer module. Emulation is provided by the Munt library.
- **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*.

MIDI In Device

Device to receive MIDI music from, for sound cards equipped with an external MIDI input.

- **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*.

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.

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

Innovation SSI-2001

Emulate the **Innovation SSI-2001** ISA sound card, based on the MOS Technology 6581 chip (commonly known as the Commodore SID) and supported by a limited number of games.

SID emulation is provided by the reSID library.

CMS / Game Blaster

Emulate the **Creative Music System** or **Game Blaster** ISA sound card, based on dual Philips SAA1099 chips and supported by some games.

Gravis Ultrasound

Emulate the **Gravis UltraSound** ISA sound card.

The type of UltraSound to emulate (Classic or MAX), I/O port and amount of onboard RAM can be configured through the *Configure* button.

Note: MAX support is only implemented on Dev builds at the moment.

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.

2.6.5  Network

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

Network type

Network emulation mode to use. See *Networking* for more information on these.

- **None:** disable networking.
- **PCap:** connects directly to a host network adapter. Similar to the **Bridge** mode on other emulators and virtualizers.
- **SLiRP:** creates a private network with a virtual router. Similar to the **NAT** mode on other emulators and virtualizers.

PCap device

Host network adapter to use for PCap mode. If no adapters appear on this list, make sure that:

- A WinPcap-compatible driver is installed;
- The installed driver is compatible with your version of Windows;
- At least one compatible (wired) network adapter is present.

Network adapter

Network card to emulate. Only cards supported by the machine's expansion buses will be listed.

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

The **[LPT] Parallel Port Internet Protocol** network adapter requires a **PLIP Network** device to be attached to a *parallel port*.

2.6.6  Ports (COM & LPT)

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

LPT1-4 Device

Emulated device to connect to the given parallel (LPT) port.

- **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 (through compatibility with the *Disney Sound Source* above), demos and trackers.
- **Stereo LPT DAC:** stereo version of the LPT DAC, using the *Strobe* pin to select the active output channel.
- **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 Dot-Matrix:** EPSON ESC/P-compatible printer.
  - Printed pages are saved as .png files in the `printer` subdirectory found in the emulated machine's directory.
  - Use the **EPSON LQ-2500** printer driver for best results.
  - This printer will not work on Windows hosts if the included `freetype.dll` file is missing from the 86Box directory.
- **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.
  - The original .ps files may remain in the directory if PDF conversion fails, or (on Windows hosts) if the included `gsdll32.dll` or `gsdll64.dll` file is missing from the 86Box directory.

  - Use the generic PostScript printer driver provided by your operating system.
  - 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.
- **PLIP Network:** A Parallel Line Internet Protocol cable connected to the *emulated network*.
  - The *emulated network adapter* must also be set to **[LPT] PLIP**.
  - PLIP 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.
  - PLIP works best with the **SLiRP** *network type* due to its point-to-point nature.

Serial port 1-4

Enable emulation of serial ports ranging from COM1 to COM4. Any ports not provided by the machine's motherboard will be emulated as generic ISA or VLB serial cards.

Parallel port 1-4

Enable emulation of parallel ports ranging from LPT1 to LPT4. Any ports not provided by the machine's motherboard will be emulated as generic ISA or VLB parallel cards.

Note: The 4th parallel port is not widely supported. It is located at I/O port *0x268*.

2.6.7  Storage controllers

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

HD Controller

Hard disk drive controller card to emulate. This box only lists cards supported by the machine's expansion buses. MFM, RLL, ESDI and IDE controllers are available. Selecting an IDE controller is not required for machines with onboard IDE.

The *Configure* button opens a new window with settings specific to the selected controller card, such as the BIOS option ROM address.

FD Controller

Floppy disk drive controller card to emulate. 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 *Settings* button.

Tertiary / Quaternary IDE Controller

Add a third or fourth (respectively) IDE channel to the emulated machine, through a generic ISA or VLB IDE controller card.

The IRQ used by each controller can be configured through its respective *Settings* button.

Note: The tertiary and quaternary controllers are not Plug and Play compliant by default; they may require manual configuration of emulated operating systems, and may not be bootable. See *Tertiary and quaternary IDE* for more information.

SCSI Controllers

SCSI host bus adapter cards to emulate. Up to 4 SCSI cards are supported. The selection boxes only list cards supported by the machine's expansion buses.

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.

Cassette

Enable IBM cassette tape emulation. The cassette deck can be controlled through the *status bar* or *Media menu*.

Note: While cassette emulation can be enabled on any machine, it is only usable on the IBM PC, PCjr and other machines with an IBM cassette port.

2.6.8  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 system 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.
- **C/H/S:** disk size in cylinders, heads and sectors, respectively.
- **MB:** disk size in megabytes.

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.
  - 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 *SCSI controller* (starting at 0 instead of 1), and 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.

Press the *OK* button to create the disk image file, or *Cancel* to close the window.

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.

Removing a disk

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

2.6.9  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 system, 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. Some types have special properties and should only be used in very specific applications:
  - **5.25" 1.2M PS/2** and **3.5" 1.44M PS/2:** IBM PS/2 drives, which invert the polarity of the Density Select pin.
  - **5.25" 1.2M 300/360 RPM** and **3.5" 1.44M 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" 1.44M PC-98:** NEC PC-98 drive, which is 3-mode and inverts the polarity of the Density Select pin.
- **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:** if unchecked, 86Box will ignore 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.

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*.

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

CD-ROM drives

Up to four CD-ROM / DVD-ROM optical disc drives can be attached to the emulated system. 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.
- **Speed:** maximum transfer speed for the drive. Up to 72x is supported.

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

2.6.10  Other removable devices

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

MO / ZIP drives

Up to four Magneto-Optical and four Iomega ZIP disk drives can be attached to the emulated system. 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** (MO only): drive model to identify as. A list of drive models to choose from is provided. Each model supports different types of MO media, while the *86BOX* model supports all types.
- **ZIP 250** (ZIP only): enable the drive to read and write 250 MB ZIP disks.

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

2.6.11  Other peripherals

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

ISA RTC

Emulate an ISA real-time clock card, for machines without an integrated real-time clock.

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

ISA Memory Expansion

Add up to four ISA-based memory expansion cards, for machines which support memory expansion through the ISA bus.

The memory start address and size for each expansion card can be configured through its respective *Settings* 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 port for the emulated machine:

Port	Machine types
`0x10`	IBM PCjr
`0x60`	IBM XT
`0x80`	IBM AT, clones and the XT-based Xi 8088
`0x84`	Early Compaq
`0x190`	IBM PS/1 and PS/2 not based on the Micro Channel Architecture
`0x680`	Micro Channel Architecture

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.

2.7 Machine-specific notes

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

2.7.1 80286

IBM AT

- 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 clearing the CMOS by deleting `ibmat.nvr` from the `nvr` directory.

2.7.2 Socket 7

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.

2.7.3 Socket 8

ASUS P/I-P65UP5 (C-P6ND)

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

2.7.4 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.

2.7.5 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.

Freeway FW-6400GX

- Hybrid motherboard supporting both Slot 1 and Slot 2 CPUs.
- 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.

2.7.6 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 Award v6.00PG BIOS. It can be enabled through the *ACPI function* option of the *Power Management Setup* menu on the BIOS setup.

2.7.7 Miscellaneous

Microsoft Virtual PC 2007

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

2.7.8 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, as none of the currently emulated machines with broken ACPI ever received a BIOS update to fix it.

2.8 Disk images

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

2.8.1 Hard disk images

Supported formats:

Format	File extension
Raw image	Many *
Japanese FDI	.hdi
Extended HDI (HDX)	.hdx
Virtual Hard Disk	.vhd

* Raw image extensions recognized by 86Box include: .hdd .ima .img

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	8032 MB	1024	255	63
16-bit cylinder	32255 MB	65535	16	63
28-bit CHS	130558 MB	65535	255	63
86Box	131071 MB	Not applicable		

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

2.8.2 Floppy disk images

Supported formats:

Format	File extension
Raw image	Many *
86F	.86f
CopyQM	.cq / .cqm
DiskDupe	.ddi
EZ-DisKlone Plus	.fdf
Formatted Disk Image	.fdi
HxC MFM	.mfm
ImageDisk	.imd
Japanese FDI	.fdi
PCjs JSON	.json
Teledisk	.td0

* Raw image extensions recognized by 86Box include: .bin .dsk .flp .hdm .ima .img .vfd .xdf

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.

2.8.3 MO / ZIP removable disk images

Supported formats:

Format	File extension
Raw image	Many *
Japanese FDI	.mdi (MO)
	.zdi (ZIP)

* Raw image extensions recognized by 86Box include: .ima .img

2.8.4 CD-ROM / DVD-ROM optical disc images

Supported formats:

Format	File extension
Cue sheet	.cue + .bin
ISO	.iso

CD audio

Compact Disc Digital Audio (CDDA) playback through the emulated CD-ROM drives is supported on **Cue sheet** 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*.

Note: Only **raw format** (.bin) tracks are supported. Compressed or otherwise encapsulated audio tracks (.wav, .mp3, .ogg, .flac and other formats) are not supported.

2.8.5 Cassette tape images

Supported formats:

Format	File extension
Raw PCM audio	Many *
PCE cassette	.cas
Wave audio	.wav

* Raw audio extensions recognized by 86Box include: .pcm .raw

2.8.6 PCjr cartridge images

Supported formats:

Format	File extension
Raw image	Many *
JRipCart	.jrc

* Raw image extensions recognized by 86Box include: .a .b .bin

2.9 Tertiary and quaternary IDE

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

2.9.1 System resources

The following resources are used by these additional controllers:

Channel	Main I/O port	Status I/O port	IRQ
Tertiary	0168h	036Eh	10
Quaternary	01E8h	03EEh	11

Each controller's IRQ can be configured through its respective *Settings* button on *Tertiary / Quaternary IDE Controller*. 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.

2.9.2 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.

2.9.3 Operating system support

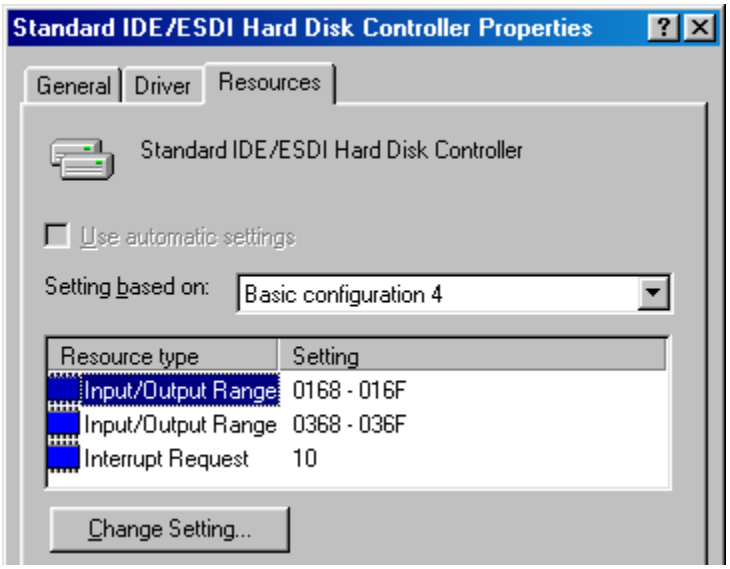
DOS and real mode

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 0368 for the tertiary channel or 03E8 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 0003* corresponds to the **tertiary** channel, while *Basic configuration 0002* 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.

- **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.
- **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=0x168,0x36e,10` (assuming IRQ 10)
    - **Quaternary:** `ide3=0x1e8,0x3ee,11` (assuming IRQ 11)

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

2.10 Networking

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

2.10.1 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:

- **IP address:** 10.0.2.15
- **Subnet mask:** 255.255.255.0
- **Default gateway:** 10.0.2.2
- **DNS server:** 10.0.2.3

The host can be reached through IP address 10.0.2.2, while other devices on the host's network can be reached through their normal IP addresses.

Note: SLiRP is only capable of routing TCP and UDP traffic. Other protocols such as IPX and NetBEUI can only be used with *PCap* networking.

2.10.2 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.

This mode requires Npcap (or another WinPcap-compatible driver) to be installed on the host. 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 machine's MAC address.

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*.

2.10.3 Advanced features

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

MAC address

With the exception of **[LPT] Parallel Port Internet Protocol**, every emulated network card stores its MAC address in the `mac` directive of its respective configuration file section. Only the suffix (last three octets) of the MAC address can be edited; the prefix (first three octets) will always be an Organizationally Unique Identifier belonging to the manufacturer, such as `00:E0:4C` for Realtek.

Example: MAC address `00:E0:4C:35:F4:C2` for the Realtek RTL8029AS

```
[Realtek RTL8029AS]
mac = 35:f4:c2
```

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]` section of the configuration file.

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 be set to 10.0.2.15 (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]
0_external = 8080
0_internal = 80
1_protocol = udp
1_external = 5555
```

2.11 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 on the *status bar* as described in the diagram below:

2.11.1 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.
```

2.11.2 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.

2.12 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

2.12.1 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 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.

2.12.2 Presets

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

Build name	Debug	New dynarec	Dev. branch	Optimized
`regular`				
`debug`				
`experimental`				
`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.

2.12.3 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)

2.12.4 Prerequisites

The build process requires the following tools:

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

Development files for the following libraries are also needed:

- FreeType
- libpng
- RtMidi
- SDL2
- FAudio (optional on Windows)
- 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-pkg-config $MINGW_PACKAGE_PREFIX-openal $MINGW_
↪PACKAGE_PREFIX-freetype $MINGW_PACKAGE_PREFIX-SDL2 $MINGW_PACKAGE_PREFIX-zlib $MINGW_
↪PACKAGE_PREFIX-libpng $MINGW_PACKAGE_PREFIX-rtmidi $MINGW_PACKAGE_PREFIX-qt5-static
↪$MINGW_PACKAGE_PREFIX-qt5-translations
```

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 libfaudio-dev␣
↪qtbase5-dev qtbase5-private-dev qttools5-dev libevdev-dev
```

macOS (Homebrew)

```
$ brew install cmake ninja pkg-config freetype sdl2 libpng openal-soft rtmidi faudio qt@5
```

2.12.5 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.

2.13 Advanced builds

The 86Box Jenkins provides all kinds of pre-release testing builds for advanced users. These are linked to the 86Box git repository on GitHub; a new build is produced with the latest source code every time the repository is updated.

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

2.13.1 Variants

86Box builds are available in a number of variants. The Jenkins 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 you have disabled JavaScript), 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 such as the Raspberry Pi.
- On **Windows**, **x86 (32-bit)** is recommended even if you have a 64-bit system.
  - x64 (64-bit) allows for emulating more than 2 GB of RAM on some later machines and using larger soundfonts with FluidSynth, at a slight performance loss.
- On **Linux**, select the correct architecture for your system, as most distributions lack x64-to-x86 and ARM64-to-ARM32 backwards compatibility by default.
- 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.

Note: Debug variants have been moved to a special page on the 86Box website as of July 18th 2022.

Discontinued variants

- Dev variants (**86Box-Dev** and **86Box-DevODR**) as of November 18th 2021.
  - These variants 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 variants' aggressive microarchitecture-specific optimizations provided very little performance improvement (within margin of error on modern CPUs) while introducing bugs and other incorrect behavior.
  - Optimized binaries can still be produced by *compiling 86Box from source* with the `--preset=optimized` CMake flag, which enables optimizations for the build host's CPU microarchitecture. No support will be provided for those.

2.14 API

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

2.14.1 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 page are provided by `86box/device.h`.

Table 1: 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_ISA: 8-bit ISA; • DEVICE_AT: 16-bit ISA; • DEVICE_EISA: EISA (reserved for future use); • DEVICE_VLB: VESA Local Bus or proprietary equivalents; • DEVICE_PCI: 32-bit PCI; • DEVICE_AGP: AGP 3.3V; • DEVICE_AC97: AMR, CNR or ACR; • DEVICE_PCJR: IBM PCjr; • DEVICE_PS2: IBM PS/1 or PS/2; • DEVICE_MCA: IBM Micro Channel Architecture; • DEVICE_CBUS: PC-98 C-BUS (reserved for future use); • DEVICE_COM: serial port (reserved for future use); • DEVICE_LPT: parallel port (reserved for future use).
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*.
close	Function called whenever this device is de-initialized, either from

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 */

    /* 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_AT, /* 16-bit ISA */
    .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,
```

(continues on next page)

(continued from previous page)

```
    .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 port devices:** `lpt_devices` in `src/lpt.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 */
```

(continues on next page)

(continued from previous page)

```
    /* ... */
};
```

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:

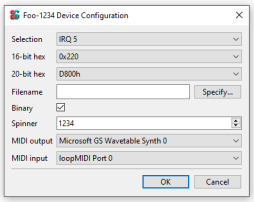


Fig. 1: All option types currently configurable through the user interface.

These options are stored in the emulated machine's configuration file, in a section identified by the device's `name`:

```
[Foo-1234]
selection = 0
hex16 = 0220
hex20 = D8000
fname = D:/VMs/86Box/86Box.exe
binary = 1
spinner = 1234
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[] = {
    { "selection", "Selection",   CONFIG_SELECTION, "", 5,       "", { 0 },
        {
            { "IRQ 5", 5 },
            { "IRQ 7", 7 },
            { ""         }
        }
    },
    { "hex16",     "16-bit hex",  CONFIG_HEX16,     "", 0x220,   "", { 0 },
        {
            { "0x220", 0x220 },
            { "0x330", 0x330 },
            { ""             }
        }
    },
```

(continues on next page)

(continued from previous page)

```
    { "hex20",     "20-bit hex",  CONFIG_HEX20,     "", 0xd8000, "", { 0 },
        {
            /* While the memory *segment* is displayed to the user, we store the
               *linear* (segment << 4) base address in the configuration file. */
            { "D800h", 0xd8000 },
            { "DC00h", 0xdc000 },
            { ""               }
        }
    },
    { "string",    "String",      CONFIG_STRING,    "Default" },
    { "fname",     "Filename",    CONFIG_FNAME,     "", 0, "File type (*.foo)|*.
↪foo|Another file type (*.bar)|*.bar" },
    { "binary",    "Binary",      CONFIG_BINARY,    "", 1 /* checked by default */ },
    { "int",       "Integer",     CONFIG_INT,       "", 1234 },
    { "spinner",   "Spinner",     CONFIG_SPINNER,   "", 1234, "", { 1204, 1294, 10 } },
    { "mac",       "MAC address", CONFIG_MAC,       "", 0 },
    { "midi_out",  "MIDI output", CONFIG_MIDI_OUT,  "", 0 },
    { "midi_in",   "MIDI input",  CONFIG_MIDI_IN,   "", 0 },
    { "",          "",            CONFIG_END }
};

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

Table 2: 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, currently **not visible nor configurable** in the user interface; • CONFIG_FNAME: arbitrary file path entered by the user directly or through a file selector button; • CONFIG_BINARY: checkbox; • CONFIG_INT: arbitrary integer number, currently **not visible nor configurable** in 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, currently **not visible nor configurable** in the user interface; • 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_END: **mandatory** terminator to indicate the end of the option list.
default_string	Default string value for a CONFIG_STRING option. Can be "" if not applicable.
default_int	Default integer value for a CONFIG_HEX16, CONFIG_HEX20, CONFIG_BINARY, CONFIG_INT or CONFIG_SPINNER option. Can be 0 if not applicable.
file_filter	File type filter for a CONFIG_FNAME option. Can be "" if not applicable. Must be specified in Windows description|mask|description|mask... format, for example: "Raw image (*.img)|*.img|Virtual Hard Disk (*.vhd)|*.vhd"
spinner	device_config_spinner_t sub-structure containing the minimum/maximum/step values for a CONFIG_SPINNER option. Can be { 0 } if not applicable. Member | Description min | Minimum selectable value. max | Maximum selectable value. step | Units to be incremented/decremented with the

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 3: `device_get_config_string`

Parameter	Description
`name`	The option's `name`. Accepted option types are `CONFIG_STRING` and `CONFIG_FNAME`.
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 4: `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 5: `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.

2.14.2 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 page 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 like 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)
{
    /* 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 6: `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);
}

/* Our device handles I/O port register 0x__80 like this:
   - 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 addr, uint8_t val, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;

    /* Handle writes to register 0x80. */
    if ((addr & 0xff) == 0x80) {
```

(continues on next page)

(continued from previous page)

```
        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 7: `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.

2.14.3 Threads

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

Warning: 86Box API functions (excluding those in this page) are generally **not thread-safe** and must be called from the **main emulation thread**. Thread-unsafe actions (like 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 8: `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 9: `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 10: `thread_create_event`

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

Table 11: `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 12: `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 13: `thread_create_mutex`

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

Table 14: `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 15: `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*.

2.14.4 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 16: 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`	I/O read operation callback functions. Can be `NULL`. Each callback takes the form of: `TYPE callback(uint16_t addr, void *priv)` • `TYPE`: operation width: `uint8_t` for `inb`, `uint16_t` for `inw`, `uint32_t` for `inl`; • `addr`: 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.
`inw`	
`inl`	
`outb`	I/O write operation callback functions. Can be `NULL`. Each callback takes the form of: `void callback(uint16_t addr, TYPE val, void *priv)` • `addr`: 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).
`outw`	
`outl`	
`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:

```
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:

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

The same rule applies to write callbacks:

- `outl` callback present:

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

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

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

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

```
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

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

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

/* 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 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 addr, uint8_t write, uint8_t val, void *priv)
{
    /* Get the device state structure. */
    foo_t *dev = (foo_t *) priv;
```

(continues on next page)

(continued from previous page)

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

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

Table 17: 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 addr, uint8_t write, uint8_t val, void *priv)` • `size`: I/O operation width: `1`, `2` or `4`; • `addr`: 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 18: 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.
`addr`	First I/O port (0x0000-0xffff) covered by this trap.
`size`	Amount of I/O ports (1-65536) covered by this trap.

2.14.5 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 19: `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 20: `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 21: `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 22: `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.

2.14.6 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>

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

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)
```

(continues on next page)

(continued from previous page)

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

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

    return dev;
}

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

Table 23: `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*.
Return value	`int` value (subject to change in the future) representing the newly-added device.

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_SCSI`: on-board SCSI controller;
- `PCI_ADD_SOUND`: on-board sound controller;
- `PCI_ADD_IDE`: on-board IDE controller;
- `PCI_ADD_NETWORK`: on-board network 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". */
    dev->slot = pci_add_card((info->local & FOO_ONBOARD) ? PCI_ADD_SCSI : PCI_ADD_NORMAL,
                             foo_pci_read, foo_pci_write, dev);

    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,
    /* ... */
};
```

(continues on next page)

(continued from previous page)

```
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;

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 */
```

(continues on next page)

(continued from previous page)

```
            dev->pci_regs[1][addr] = val;
            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[0], 0, sizeof(dev->pci_regs[0]));

    /* 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. */
    dev->slot = pci_add_card(PCI_ADD_NORMAL, foo_pci_read, foo_pci_write, dev);

    /* 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 24: Memory BAR (example: 4 KB large, starting at `0x10`)

Byte	`0x13`				`0x12`				`0x11`				`0x10`			
Bit	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
Value	Base memory address (4096-byte aligned)																Always `0`															`0`

Table 25: I/O BAR (example: 64 ports large, starting at `0x14`)

Byte	`0x17`				`0x16`				`0x15`				`0x14`			
Bit	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
Value	Ignored (`0` recommended)								Base *I/O port* (64-byte aligned)								Always `0`															1

Code example: memory and I/O BARs described above

```
#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.
           We skip reading dev->pci_regs[0x10] as it contains nothing of interest. */
```

(continues on next page)

(continued from previous page)

```
        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
foo_reset(void *priv)
```

(continues on next page)

(continued from previous page)

```
{
    /* 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 26: Option ROM (example: 32 KB large)

Byte	`0x33`								`0x32`								`0x31`								`0x30`							
Bit	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
Value	Base memory address (32768-byte aligned)																Always `0`															

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. */
            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;
```

(continues on next page)

(continued from previous page)

```
        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");
}

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... */
```

(continues on next page)

(continued from previous page)

```
    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>

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;
```

(continues on next page)

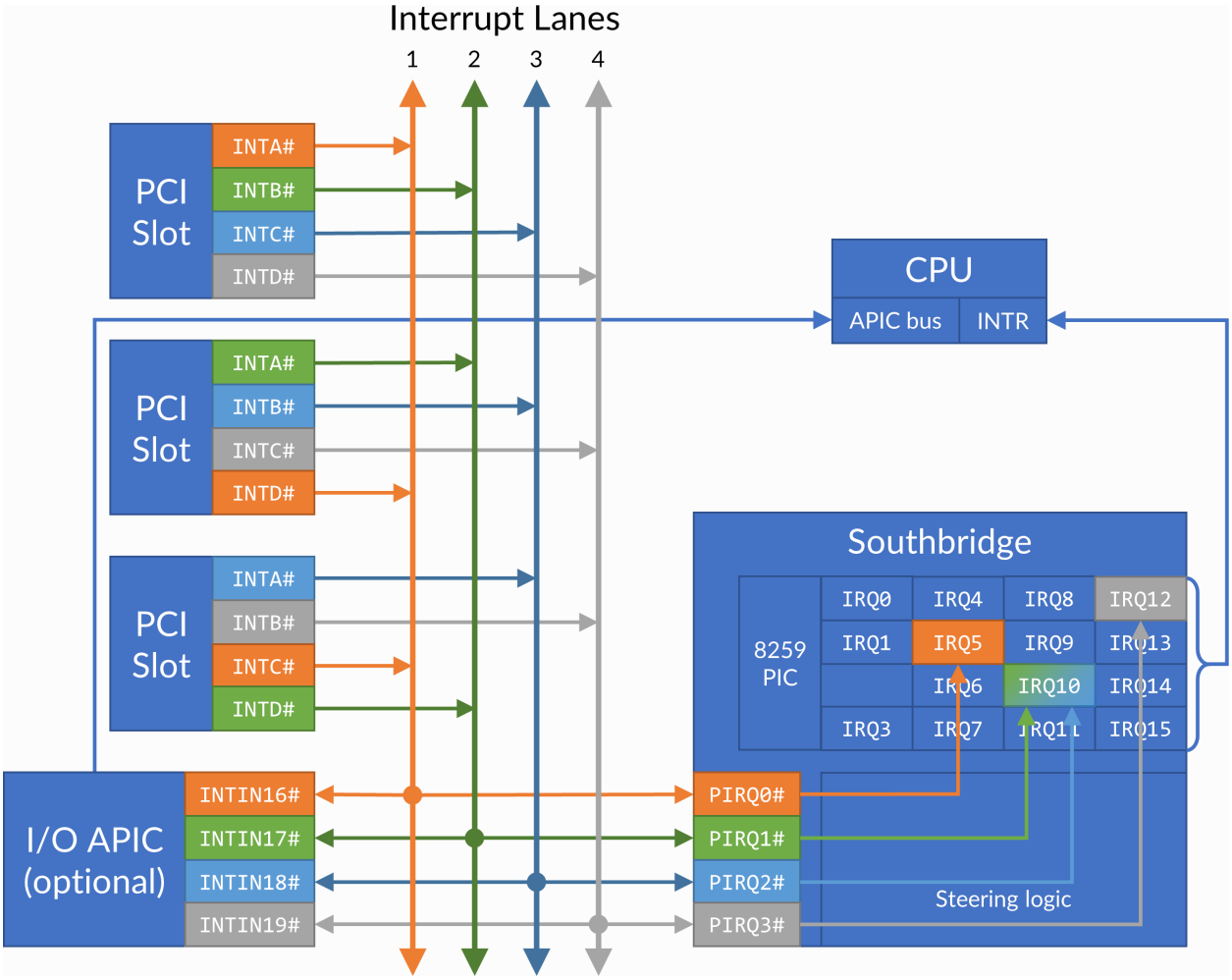


Fig. 2: PCI interrupt topology example. IRQ 10 is shared by two interrupt lanes. The machine is free to route any `INTx#` pin to any lane (sequentially or not), and free to route any lane to any available IRQ.

(continued from previous page)

```
    /* 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);
        else
            pci_clear_irq(dev->slot, PCI_INTA);
    }
}

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. */

    /* Our device uses the INTA# interrupt line. */
    dev->pci_regs[0x3d] = PCI_INTA;
}

/* Don't forget to add the PCI device on init first, and to save
   the return value of pci_add_card (to dev->slot in this case). */

const device_t foo4321_device = {
    /* ... */
    .reset = foo_reset,
    /* ... */
};
```

Table 27: `pci_set_irq` / `pci_clear_irq`

Parameter	Description
`card`	Value representing this PCI device, returned by `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`.

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 28: `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.

2.15 File formats

86Box introduces new file formats for disk images and other purposes. These formats are documented on this section.

2.15.1 86F

A floppy disk surface image format which stores data in FM- or MFM-encoded transitions.

Preliminary specification for v2.20

All offsets are in hexadecimal. This specification is subject to change before its final release.

```
00000000: Magic 4 bytes ("86BF")
00000004: Minor version (0x14)
00000005: Major version (0x02)
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)
        Bit 5           Bitcell mode (1 = Extra bitcells count specified after
                        disk flags, 0 = No extra bitcells)
                        The maximum number of extra bitcells is 1024 (which
```

(continues on next page)

(continued from previous page)

```
                        after decoding translates to 64 bytes)
        Bit 6           Revolutions (0 = one revolution, 1 track has 16-bit number of␣
↪revolutions)
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.).
        The table of the offsets of tracks is 2048 bytes long, each track offset is an␣
↪unigned 32-bit
        integer. An offset of 00000000 indicates the track is not present in the file.
        As an example, an 86F representing a disk with 80 thin tracks and 2 sides per␣
↪track, where all
        the tracks are present in the file, would have the first 160 offsets filled in,␣
↪same for a disk
        with 40 thick tracks and 2 sides. Same with only 1 side but only the offsets at␣
↪0000000, 0000008,
        etc. (so every second offset) would be filled in.

Track offset + 00000000: Track flags (16-bit)
        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.
        The RPM is determined from track length and data rate.
Track offset + 00000002: Total bit cells count (32-bit)
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)

If this is a multi-revolution 86F, then track offset + 00000000 has a 16-bit number of␣
↪track revolutions,
and the track header + data appears for each revolution, while surface description data,␣
↪if present,
can appear any number of times, but only once per encoding + bit rate combination.
This needs work to properly make surface data work with flexible multi-revolution␣
↪support.

Track lengths:
        The total bit cells count is always present.
        The track is stored as (bit cells >> 8) bytes, with one extra bit cells if the␣
↪number of bit cells
```

(continues on next page)

(continued from previous page)

```
        is not divisible by 8.
```

2.15.2 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)
```