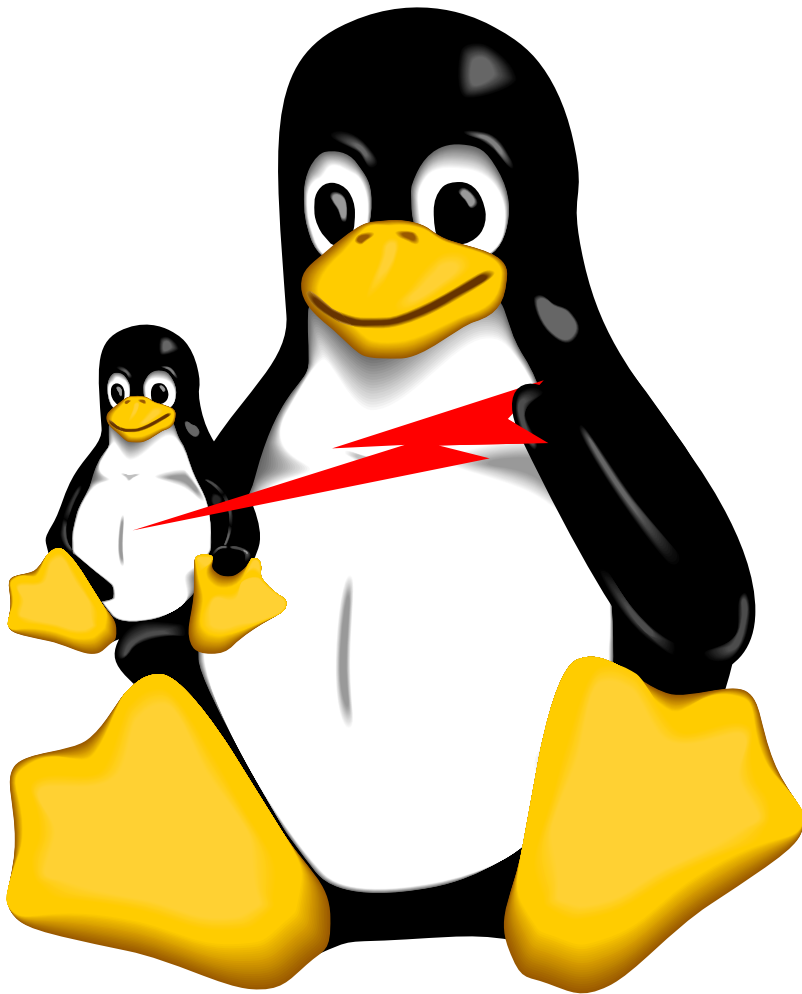


FAUmachine Usage Guide

Version 20050913

FAUmachine Team



<http://www.faumachine.org>
email: info@faumachine.org

Contents

1	How to Use this Manual	1
2	Installation of a FAUmachine Binary Distribution	3
2.1	Requirements	3
2.2	Installation	4
3	Configuring a Virtual Linux Machine Using the ConfigWizard	7
3.1	Starting the ConfigWizard	7
3.2	First Configuration Based on Default Settings	7
3.3	Editing Existing Configurations	7
3.4	Files Used by the Launcher	8
4	Configuring a Virtual Linux Machine Manually	9
4.1	Processor	9
4.2	Main Memory	10
4.3	VGA Card	10
4.4	IDE Block Devices	10
4.5	Floppy	12
4.6	Network Card	13
4.7	Serial Interface	14
4.8	Parallel Interface	14
4.9	CIM - Component Interconnection Method	15
4.10	Virtual Machine Name Displayed by Frontend	16
5	Running a Virtual Linux Machine	17
5.1	Using the Launcher	17
5.2	Starting a Virtual Machine	17
6	Connecting a Virtual Machine to the Real Network	21
6.1	Overview	21
6.2	Configuration of the network bridge	21
6.3	Network Configuration of the Virtual Machines	23
7	Creating an Automatic Load Generator for a Virtual System	25
7.1	Setting up the Test Environment — <code>simulation.setup</code>	26
7.2	The Scripting Language — VHDL	30
7.3	Describing the Target System Hardware — <code>system.vhdl</code>	30
7.4	Automating Text-Based User Interfaces Via a Serial Terminal	37
7.5	Automating User Interfaces Via Monitor, Mouse and Keyboard	41
7.6	Connecting Shortcuts	45
8	Injecting Faults into the Hardware of a Virtual Machine	47
8.1	Available Failures	47
8.2	Injecting Faults Using the GUI	48
8.3	Automated Fault Injection	49
9	Automation Examples	53
9.1	<code>CDROM.images</code>	53
9.2	<code>install-SuSE-*-vesa, install-Redhat-*-vesa</code>	53
9.3	<code>install-Debian-*-serial</code>	54
9.4	<code>single-node, dual-node, serial</code>	55

10 Building FAUmachine from the Source Distribution	57
10.1 Installing the FAUmachine Source Distribution	57
10.2 Building FAUmachine	57
10.3 Building bootloaders and kernels	58
11 When Things Don't Work	59
11.1 Creating Bugreports	59
11.2 Caveats	59

List of Figures

4.1	Example for CPU Configuration File	10
4.2	Example for NE2000 Configuration File	14
4.3	Example for Serial Port Configuration File	15
4.4	Example for Parallel Port Configuration File	15
7.1	Minimal simulation.setup	26
7.2	Example of simulation.setup for Simple Client-Server System	28
7.3	Predefined Components	32
7.4	Internal Structure of a Virtual Machine	35
7.5	VHDL-Code for Figure ??	36
7.6	Connecting a serial_terminal Component	38
7.7	VHDL-Code for Defining Signals for Use with serial_terminal Component	38
7.8	VHDL-Code for Defining the serial_terminal Component	38
7.9	VHDL-Code for Defining a Process (Using wait_string)	38
7.10	Connecting a serial_pattern Component	39
7.11	VHDL-Code for Defining Signals for Use with serial_pattern Component	39
7.12	VHDL-Code for Defining the serial_pattern Component	40
7.13	VHDL-Code for Defining a Process (Using Shortcuts)	40
7.14	Automated Mouse Movement	44
7.15	Searching for a Pattern	46

1 How to Use this Manual

This userguide is intended both for casual users of FAUmachine as well as for those who want to use it in a professional and productive environment.

To get your first virtual machine up and running without a lot of fuss and extras start `faum` after installation of the package.

For a little more information read through chapters 2 "Installation of a FAUmachine Binary Distribution", 3 "Configuring a Virtual Linux Machine Using the ConfigWizard " and 5 "Running a Virtual Linux Machine". These chapters will tell you how to install FAUmachine, configure your first virtual machine and boot it.

To connect your virtual machine to the real network, read through chapter 6 "Connecting a Virtual Machine to the Real Network". Interconnecting several virtual machines is the default and you have read about it in chapter 3.

If you want to find out what happens to a machine, when hardware starts failing, read through section 8.2 in chapter 8 "Injecting Faults into the Hardware of a Virtual Machine" for an easy introduction on how to make your virtual hardware fail.

If you prefer to edit configuration files directly using your favourite text editor, check out chapter 4 "Configuring a Virtual Linux Machine Manually".

For those of you really into scripting complex automated tests, read through chapter 7 "Creating an Automatic Load Generator for a Virtual System" and if you want to integrate hardware failures into your automated tests, check out section 8.3 in chapter 8 "Injecting Faults into the Hardware of a Virtual Machine".

If you want to check out the code or even build FAUmachine from scratch yourself, chapter 10 "Building FAUmachine from the Source Distribution" is for you.

Finally, if things don't work as you expect, please have a look at chapter 11 "When Things Don't Work" before you send us an email.

2 Installation of a FAUmachine Binary Distribution

This chapter briefly lists the hardware and software requirements which need to be fulfilled to run FAUmachine. You then learn how to install the precompiled FAUmachine-binaries from a tarball, rpm or debian package.

2.1 Requirements

This section tells you, what kind of hardware and operating system you will need to run a FAUmachine virtual machine. The requirements for building FAUmachine from scratch are much more extensive and are listed in chapter 10 "Building FAUmachine from the Source Distribution".

2.1.1 Hardware

Except for the fact, that the processor must be an Intel i386 or compatible, there are no known special hardware requirements.

You should keep in mind, though, that apart from the space needed to install a minimal FAUmachine binary distribution, you will need enough room on your harddisk for the harddisks of the virtual machines you create as well as their main memory. The files containing the disk-images and memory-images of the virtual machines cannot be split across partitions.

Of course, the faster your hardware is, the faster the virtual machines will be, too. You should have enough main memory for both the guest and the host operating system. Otherwise, performance will drop dramatically as the host operating system is forced to use swap space.

2.1.2 Software

FAUmachine currently only runs on Linux. We have completed a raw prototype running on Windows with Cygwin installed as a feasibility study, but due to the lack of manpower, we currently do not pursue the Windows/Cygwin port.

The precompiled FAUmachine binaries and kernels should run on all default 2.4 kernel-versions, i.e. FAUmachine should work fine on any current Linux distribution (except Debian, which still uses a 2.2 kernel per default; however, Debian provides an upgrade package to a 2.4 kernel).

As we currently do not provide FAUmachine binaries for 2.2 kernels, if you need FAUmachine to run on a 2.2 kernel you will need to compile it yourself. You might also have to compile your own FAUmachine binaries if you built your own kernel and changed parameters concerning the placement and size of your kernel's kernel memory (for details see chapter 10).

Because of the simulation method used by FAUmachine, you need kernels and boot loaders which are modified to run in the simulator. Several modified versions are already available for major distributions. Upon boot, the bootloader and kernel from your simulated system will be replaced by the corresponding modified version provided by FAUmachine. I.e., if you want to run SuSE 8.2 on FAUmachine, you will need to install the faumachine-suse-8.2 package in addition to faumachine-base.

If you think FAUmachine runs a little slow, we do have kernel patch for some kernel versions which will speed up FAUmachine. To install it on your real machine would entail patching the kernel sources and building/installing a new kernel. If you are interested in the patch, send an email to info@faumachine.org.

2.2 Installation

Version 20050913 of FAUmachine includes rpms for SuSE 8.1, SuSE 8.2 and SuSE 9.0 as well as for RedHat 8.0 and RedHat 9. There are debs for Debian 3.0r0 and a tarball for all the rest.

2.2.1 Installation of FAUmachine Binary RPM

Download the current rpms from our download area at www.faumachine.org. To get the files necessary for the distribution on your real machine, follow the appropriate link. For each real system, the following files are available:

- faumachine-base-20050913-1.i386.rpm:
the basic binaries (configuration wizard, simulator, network bridge, etc.)
- faumachine-debian-*-20050913-1.i386.rpm,
faumachine-redhat-*-20050913-1.i386.rpm,
faumachine-suse-*-20050913-1.i386.rpm:
bootloaders and kernels of the respective distribution precompiled to run on FAUmachine. I.e. if you want to run Redhat 8.0 on your FAUmachine, you will need to install the faumachine-redhat-8.0-20050913-1.i386.rpm.

You will need at least the base-package and one of the support packages.

To install the rpms you will need to be root. Type

```
rpm -hiv packagename
```

at the command prompt to install the package.

2.2.2 Installation of FAUmachine Binary Debian Package

You can tell your apt about FAUmachine by adding the line

```
deb http://www3.informatik.uni-erlangen.de/FAUmachine/debian ./
```

to `/etc/apt/sources.list`, then execute `apt-get update` and `apt-get install faumachine` as root. You can update these packages through the usual `apt-get upgrade`.

You can also download the current debs from our download area at www.faumachine.org (follow the appropriate link).

- faumachine-base_0-20050913.1_i386.deb:
the basic binaries (configuration wizard, simulator, network bridge, etc.)
- faumachine-debian-*_0-20050913.1_i386.deb,
faumachine-redhat-*_0-20050913-1_i386.deb,
faumachine-suse-*_0-20050913-1_i386.deb:
bootloaders and kernels of the respective distribution precompiled to run on FAUmachine. I.e. if you want to run Redhat 8.0 on your FAUmachine, you will need to install the faumachine-redhat-8.0.0-20050913-1_i386.deb.

You will need at least the base-package and one of the support packages.

To install the debs you will need to be root. Type

```
apt-get install package
```

at the command prompt to install the package.

2.2.3 Installation of FAUmachine Binary Tarball

Download the current binary tarball `faumachine-20050913.tar.gz` from our download area at **www.faumachine.org**. This contains the virtual machine, graphical user interface and ready-to-run FAUmachine-kernels for the supported distributions. This is all you need to get started.

If you want other kernels, it becomes more difficult, as you will have to build these yourself. Please refer to chapter 10.

Extract the tarball into directory of your choice, e.g. using the command `tar xzf faumachine-20050913.tar.gz`. This directory will be referred to as FAUmachine install directory from now on.

After extracting the binary tarball you will have several subdirectories called `/usr/bin`, `/usr/lib`, `/usr/share/man`, and `/usr/share/doc/faumachine`. The following list gives an overview.

`/usr/bin/`

FAUmachine program binaries for graphical frontend, virtual machines and automatic experiment controller.

`/usr/share/doc/faumachine`

This FAUmachine documentation as HTML and PDF.

`/usr/lib/faumachine/kernels/`

Precompiled FAUmachine-kernels for Debian3.0r0, SuSE 8.1, SusE 8.2, RedHat8.0, RedHat9.

`/usr/lib/faumachine/loaders/`

Precompiled FAUmachine-bootloaders for Debian3.0r0, SuSE 8.1, SusE 8.2, RedHat8.0, RedHat9.

`/usr/lib/faumachine/tools/`

Additional binaries, useful in combination with the automatic experiment controller.

`/usr/share/faumachine/vhdl/`

VHDL libraries needed for scripting the automatic experiment controller.

`/usr/share/faumachine/xpm/`

Pixmaps for the graphical frontend.

`/usr/share/man`

man pages.

Add `/usr/bin` of your FAUmachine install directory to the `PATH` environment variable so that your shell can find the binaries automatically or call `faum` with an absolute path. All programs part of FAUmachine automatically load the necessary libraries (they figure out the path to the library from the path used to call `faum`), so no further configuration is necessary.

3 Configuring a Virtual Linux Machine Using the ConfigWizard

FAUmachine comes with a ConfigWizard, which can help you create and configure new virtual machines quickly without having to learn about the weird syntax of configuration files.

This chapter shows how to use the ConfigWizard to set up and start virtual machines.

3.1 Starting the ConfigWizard

You can start the ConfigWizard with the binary called `faum`. This starts the **Launcher**. If no virtual machines have been created yet, the ConfigWizard will automatically be started, too. You can start it manually by choosing the **New (Wizard)** entry from the **VM** menu.

3.2 First Configuration Based on Default Settings

If you simply click **Next** (and **Finish** on the final page) in the ConfigWizard, a new virtual machine with the following properties will be created in your home directory.

Virtual Machine's Hardware:

Processor: A single processor.

Harddisk: A single 2GB harddisk.

CDROM: A single CDROM-drive.

Network: A single network card with a cable plugged in. A new virtual network called `net0` (for the very first network) will be created on the fly and interconnected with the real network (where your real machine may be the only computer on the real network).

Main Memory: 128MB of main memory.

Graphics Adapter: VGA Card with 8MB of onboard memory.

Other Configuration Parameters:

Machine Name: `vm000` for your very first virtual machine, `vm001` for your second one, and so on.

Machine Directory: `vm000` for your very first virtual machine, `vm001` for your second one, and so on. The default is to create new directories which will be created in your home directory.

Host: The default is to start the virtual machine on the local host.

3.3 Editing Existing Configurations

Of course you can edit existing configurations, too. Select the **Edit** menu entry from the **VM** menu and choose one of the virtual machines from the list, to open its edit window.

Toggle the check buttons to unfold the list of available devices of each kind, then click on the device to show its editing window.

You can currently configure IDE drives, network adapters, serial and parallel ports and main memory size as well as the miscellaneous parameters virtual machine name and host.

Use the online help to guide you through the process.

3.4 Files Used by the Launcher

The ConfigWizard keeps information in the hidden directory `.FAUmachine` inside your home directory. If that directory does not exist in your home yet, it will be created as needed.

The ConfigWizard keeps information about the virtual machines which have been created in `.FAUmachine/config`.

The ConfigWizard keeps information about virtual networks it knows about in directories called `.FAUmachine/net#`. Each network configuration directory contains the files `host`, `name`, and the Bridge configuration files `upstream` and `upstream-method`. These files are read by the Bridge when you choose a virtual network from the **Connect** menu entry in the **Network** menu. `host` contains the host the Bridge should be started on. `name` is the name of the virtual network as displayed by the Launcher in the menus and lists. The actual Bridge configuration is described in chapter 6 "Connecting a Virtual Machine to the Real Network".

The configuration files generated by the wizard are described in chapter 4 "Configuring a Virtual Linux Machine Manually".

4 Configuring a Virtual Linux Machine Manually

This chapter describes the syntax and semantics of the hardware configuration files, in case you need it. The files used to configure the virtual hardware are not freeform. Unless noted otherwise, the syntax must be exactly as described, including the number of spaces and newlines (be careful of trailing newlines at the end of files).

All files belonging to a single virtual machine are kept in a common directory. There is a unique different directory for each virtual machine.

To be able to boot your virtual machine, you will need at least one processor (section 4.1), some main memory (section 4.2) and a VGA card (section 4.3). Your virtual machine won't be much good without a place to store some data, so you should configure at least one block-device (section 4.4). The rest of the available hardware is purely optional and need not be configured.

4.1 Processor

Your virtual machine will not work without a processor, so you must configure it!

To provide your virtual machine with a single (default) processor create a file called `cpu-0`. The first line holds the processor type. The list of known processors is currently

- AMD Athlon(tm) XP 1700+
- AMD Athlon(tm) XP 2100+
- AMD Duron(tm)
- AMD Duron(tm) processor
- AMD-K6tm w/ multimedia extensions
- Celeron (Mendocino)
- Intel(R) Celeron(TM) CPU 1200MHz
- Intel(R) Pentium(R) 4 CPU 1.60GHz
- Pentium II (Klamath)
- Pentium III (Coppermine)
- Pentium III (Katmai)

If you use an unknown processor type, the virtual machine will stop and print the list of processor types it currently knows about on the console, where you started it (no GUI error message).

The second line holds a serial number for the processor which is used to configure apic IDs.

The third line holds the speed in MHz of your processor.

Figure 4.1 is an example of a CPU configuration using a Pentium II at 200 MHz:

```
Pentium II (Klamath)
0
200
```

Figure 4.1: Example for CPU Configuration File

4.2 Main Memory

Your virtual machine will not work without main memory, so you must configure it!

To configure the size of the main memory of the virtual machine, create a file called `memory` in its configuration directory. This file contains a single line with the number of bytes of memory in hexadecimal notation (but without leading "0x". Table 4.1 gives a selection of possible entries.

Larger main memory sizes are not supported by precompiled kernels. If you do need them, you will have to compile your own FAUmachine binaries with matching kernels (see chapter 10 "Building FAUmachine from the Source Distribution").

4.3 VGA Card

Currently your virtual machine will not work without a vga card, so you must configure it!

To configure the size of the video memory of the virtual machine's vga-card, create a file called `vga` in its configuration directory. This file contains a single line with the number of bytes of video memory in hexadecimal notation. Table 4.1 gives a selection of possible entries.

Which resolution and how many colors are available on your virtual machine depends on the amount of video memory of the VGA Card. Table 4.2 may help you find the right configuration. The column headers give the resolution, the line headers the color depth. The video mode and a suggestion for the size of video memory for this combination of resolution and color depth is given.

To set the video mode, pass the correct video mode as a kernel parameter at boot up (e.g. use `vga=791` to switch to 1024x768 in 16bit colors). Most distributions allow you to configure video modes during the first installation steps, too.

4.4 IDE Block Devices

Currently FAUmachine virtual machines can only have IDE block devices (harddisks and cdrom-drives). To make use of your virtual machine, you will probably need at least one harddisk.

MB of memory	entry in file
1	00100000
2	00200000
4	00400000
8	00800000
16	01000000
32	02000000
64	04000000
128	08000000
256	10000000
512	20000000
768	30000000
896	38000000

Table 4.1: Memory Sizes in Bytes (Hexadecimal)

	640x480	800x600	1024x768	1280x1024
8 bit	769 / 1 MB	771 / 1 MB	773 / 1 MB	775 / 2 MB
15 bit	784 / 1 MB	787 / 1 MB	790 / 2 MB	793 / 4 MB
16 bit	785 / 1 MB	788 / 1 MB	791 / 2 MB	794 / 4 MB
32 bit	786 / 2 MB	789 / 2 MB	792 / 4 MB	795 / 8 MB

Table 4.2: Video Modes and Required Video Memory

IDE block devices each have their own configuration subdirectory called `ide-#`, where `#` is a single digit between "0" and "3". These names identify the IDE-controllers. Your standard machine will have two IDE-controllers, but you can have up to four, if you want to. The interrupt and IO-address of each IDE-controller are configured in a file called `config` in the `ide-#`-directories. Which controller is considered to be first, second etc. by the kernel depends on its interrupt and IO-address, not on the name given to the directory! (Little stickers on your real IDE-controllers with names on them won't make the kernel give them those names either). Which names (such as `/dev/hda` in Linux) are given to the devices depends on the number of the controller and the unit. (Which means, change the IO-addresses and your boot device will change, too, which usually is not what you want!) Table 4.3 gives you a list of possible values for interrupts and IO-addresses.

Each IDE-controller can handle up to two devices. The information describing the devices are located in subdirectories called `unit-#`, where `#` is 0 or 1.

Each IDE block device configuration subdirectory must contain a file called `config`. `config` is used, so the virtual IDE-controller can return information on the type of the virtual block device.

The following sections detail how to configure harddisks (section 4.4.1) and CDROM-drives (section 4.4.2) and how to enable the copy-on-write feature (section 4.4.3).

4.4.1 Virtual Harddisks

If the device is a virtual harddisk, the contents of `config` are the single line

```
disk
```

The directory must then also contain a file called `media` (a harddisk without any media to save your stuff would be quite useless). The size of this file is fixed, once it is created (sadly, there is no way to enlarge your real harddisk either). `media` will contain the actual contents of the harddisk. Use

```
dd if=/dev/zero of=media ibs=1k count=numberM
```

to create an empty virtual harddisk of *number* Gigabytes in size (*number* must be an integer) or

```
dd if=/dev/zero of=media ibs=1k count=numberk
```

to create an empty virtual harddisk of *number* Megabytes size.

It is possible to simply copy virtual harddisks from other virtual machines. To connect the new virtual machine to the same network as the original, you will then have to reconfigure the virtual machine's hostname and IP-address (just as you would for real Linux machines using the same cloned harddisk).

Detected Controller ID	Interrupt	First IO-address	Second IO-address
0	14	0x01f0	0x03f6
1	15	0x0170	0x0376
2	11	0x01e8	0x03ee
3	10	0x0168	0x036e
4	8	0x01e0	0x03e6
6	12	0x0160	0x0366

Table 4.3: Interrupts and IO-Addresses of IDE-Controllers

4.4.2 Virtual CDROM-drives

If the device is a virtual CDROM-drive, the contents of `config` are the single line

```
cdrom
```

The directory may not also contain a file called `media`, since initially, no CD is inserted into the CDROM (which is usually the case for a freshly bought CDROM-drive). When a virtual CDROM is inserted into the drive (see chapter 5 "Running a Virtual Linux Machine"), a softlink is created, which points to the CD image (which may be a real CD in your real CDROM-drive).

To create an CD image in a file use `mkisofs` or `cp`. Assuming the CDROM of the host machine is accessible as device `/dev/hdc` and contains a CDROM, you can create an image using the command

```
cp /dev/hdc image-file
```

4.4.3 Copy-on-Write Feature

It is possible to enable a copy-on-write on those read-write IDE block devices with a total capacity up to 8GB. When copy-on-write is enabled, nothing is written to the file `media` containing the virtual harddisks data. Instead, the changes are protooled in an extra file called `cow`. Delete the files `map` and `cow` to undo all the changes.

To enable copy-on-write create a file called `map` in the directory of the appropriate harddrive. You can create `map` using the command

```
touch map
```

If you want to revert all changes made to the copy-on-write disk, just delete `map` and create it again. Using copy-on-write, you can even use the same (readonly) `media` for several virtual machines, storing only machine-local changes. This is useful to simulate many similar machines without having to store their harddisks multiple times.

There is also a little commandline tool called `faum-mergecow`, to merge the changes saved in `cow` with the data on the original disk:

```
faum-mergecow dir file
```

`dir` must be the directory containing the `media`, `map` and `cow` files, `file` is the name of the file containing the merged data.

4.5 Floppy

Your virtual machine will work fine without a floppy drive, so if you do not need one you don't have to configure it at all.

To configure a floppy disk controller create a directory named `fdc-0` (first floppy disk controller). To configure a floppy drive connected to the first floppy disk controller create a directory named `fdc-0/drive-0` (first floppy drive, usually A) or `fdc-0/drive-1` (second floppy drive, usually B). The floppy drive configuration directories must contain two files called `media` and `config`.

`media` is an image of a diskette or a softlink to the real floppy. To create an image you can use `dd` or simply (assuming your real floppy drive is accessed as `/dev/fd0`) `cp /dev/fd0 media`.

`config` contains a single digit, defining the type of the floppy disk drive. For possible values please refer to table 4.4.

digit	type of floppy
0	unknown
1	360kB
2	1.2MB
3	720kB
4	1.4MB
5	2.8MB

Table 4.4: Floppy Configuration File

4.6 Network Card

Your virtual machine will work fine without a network card, so if you do not need one you don't have to configure it at all.

The only virtual network card currently available is an NE2000 compatible network card. The configuration files for NE2000 compatible network cards (there may be more than one in a single virtual machine) are called `ne2000-n`, where *n* is a number. To connect several virtual machines to a virtual network, resources on the host machine are used and must therefore be specified in this file. So the rest of this section may be a bit technical.

The NE2000 configuration files contain information about the setup of the virtual network adapter card as well as information about the virtual network the adapter is connected to. The information about the virtual network consists of the virtual network name and the configuration for the CIM (component interconnection method, see section 4.9) which connects the machines on the virtual network to each other.

The first line of the configuration file contains the virtual network name. The following three lines of the configuration file describe the configuration of the virtual network controller hardware.

```
virtual-network-name
interrupt
io-address
virtual-mac
```

The rest of the lines describes the configuration of the CIM (4.9).

Except for the interrupt, which is given as decimal number, all numbers are given as hexadecimal numbers. The IO-address configures the card's memory mapped IO-space. Table 4.5 gives you a list of possible combinations of interrupt and IO-address. Be careful to avoid conflicts if you also have a lot of other devices.

virtual-network-name is a name for the virtual network this card is connected to. Allowable characters for this name are English letters and capital letters (no umlauts, accents etc.), digits and the underscore. Figure 4.2 shows an example. The netbridge and the faum setup tools use this name to configure the right network.

In the example configuration the network card has the irq 9 and its IO-Address is at 0x300. It's MAC is 00:3e:01:f8:06:c4. The virtual network the adapter is connected to is net0. The 4.9 is configured to work in UDP mode and to connect the network card to 3 peers, whose IP addresses and ports are given in the last three lines. The local CIM instance listens on port 16385.

Interrupt	IO-Address
9	0x0300
10	0x0280
11	0x0320
12	0x0380
14	0x0360
7	0x0340

Table 4.5: Interrupts and IO-Addresses for NE2000 Network Controllers

```
net0
9
300
00:3e:01:f8:06:c4
1
3
16385
127.0.0.1:16384
127.0.0.1:16383
131.188.3.15:8738
```

Figure 4.2: Example for NE2000 Configuration File

4.7 Serial Interface

Your virtual machine will work fine without a serial interface, so if you do not need one you don't have to configure it at all.

The configuration file is called `serial-n`, where *n* is a number between 0 and 3 designating the number of the serial interface. The common case is that only `serial-0` and `serial-1` are available. The configuration file contains information about the interrupt used by the corresponding serial interface and some information pertaining to mapping the virtual serial interface to resources on the real host machine. This approach is similar to the network interfaces, as serial interfaces are exported as sockets, too.

The configuration file should contain these lines:

1. Version of config-File (for compatibility with older versions): current Version number is 10.
2. interrupt number (must be 3 or 4, decimal)
3. IO-address space (hexadecimal)
4. Definition of protocol for data-transfer to the exported sockets:
 - 0 Protocol 0 defines a kind of short-circuit protocol to use when other programs connect to the serial interface socket: Only raw data is sent and received, with no throughput limitation, and no further checks (baud-rate, stop-bit, word-length etc.) Typically protocol 0 is used for experiments that utilize the serial terminal (i.e. virtual console on serial-port).
 - 1 Protocol 1 is currently not supported (yet).
5. The rest of the lines describes the configuration of the CIM (4.9).

Refer to table 4.6 for a list of valid interrupt and IO-address combinations. Figure 4.3 is an example of a configuration for `serial-0` as generated by the ConfigWizard:

4.8 Parallel Interface

Your virtual machine will work fine without a parallel interface, so if you do not need one you don't have to configure it at all.

Interrupt	IO-Address
4	0x03f8
3	0x02f8
4	0x03e8
3	0x02e8

Table 4.6: Interrupts and IO-Adresses for Serial Interfaces

```

10
4
3f8
0
1
1
16384
127.0.0.1:16384

```

Figure 4.3: Example for Serial Port Configuration File

Please note that support for external devices on the parallel interface (e.g. printers) is still somewhat experimental.

The configuration file is called `parallel-n`, where *n* is a number between 0 and 3 designating the number of the parallel interface. The common case is that only `parallel-0` is available.

A parallel port configuration file contains information about the setup of the parallel port controller hardware as well as a part containing the configuration of the 4.9 connecting the parallel controller to the external lp-bridge.

1. *IO base address*

The IO base address is given in hexadecimal notation

2. *Interrupt number*

The interrupt number is given in decimal notation; if -1, the interface will have no interrupt assigned.

3. *DMA channel*

The DMA channel number is given in decimal, but is silently ignored as there's no parallel port DMA implemented yet.

Following these three lines the configuration file includes several lines of CIM (see section 4.9) configuration.

Common IO-Adresses for parallel ports are 0x278, 0x378 (default) and 0x3bc.

Common IRQs for parallel ports are 5 and 7.

Figure 4.4 is an example of a configuration file:

4.9 CIM - Component Interconnection Method

The CIM is a module implementing the "glue" between the FAUmachine simulator core and external hardware component simulators. Technically it's an extensible library providing different methods of local/remote communication to the FAUmachine components. It's used to virtually wire and connect FAUmachine components, peripherals and networks.

The CIM has its own configuration file format. CIM configuration files are included in some FAUmachine component configuration files (e.g. network configuration, see section ??, serial/parallel port configuration, see sections 4.7 and 4.8).

```

378
-1
-1
1
1
9009
131.188.33.39:8008

```

Figure 4.4: Example for Parallel Port Configuration File

A CIM configuration file consists of at least the first three lines of the following:

1. *CIM protocol*
2. *number of peers to communicate with*
3. *local port / peer UID*
4. *peer 1 ip:port*
5. ...
6. *peer n ip:port*

The CIM protocol number currently can only be 1 for UDP or 2 for TCP.

The number of peers describes the number of communication partners the CIM will communicate with. This number can be zero. This number logically also describes the number of lines to continue reading after the mandatory three lines of CIM configuration, which the CIM expects to contain information about the communication partners.

The local port describes the local port at which the CIM will listen for peer connection requests. In case the CIM operates in UDP multicast mode (which will be the case if the first peer given has a UDP multicast address) the number is interpreted as virtual-network-wide unique identifier of the local CIM instance. The lines describing the peers contain IPv4 addresses in numbers-and-dots notation, followed by a colon and the TCP/UDP port at which the peer listens for connections, given in decimal.

4.10 Virtual Machine Name Displayed by Frontend

The frontend displays the list of virtual machines it finds in the current directories. The name displayed for a virtual machine is read from a file called `name` in the `virtual-machine-directory`. This file contains a single line with the desired name. If the name is longer than 128 characters, it will be truncated to 128 characters. The frontend may not display special characters (like letters with accents) correctly.

5 Running a Virtual Linux Machine

This chapter describes how to start and interact with your virtual machines.

5.1 Using the Launcher

The easiest way to start virtual machines is by using the **Launcher**. Execute `faum` to start it.

Once you have configured your virtual machines (for example by using the **ConfigWizard**, see chapter 3 "Configuring a Virtual Linux Machine Using the **ConfigWizard**"), you sit down in front of your machine (virtually of course) by selecting **VM/Start**. A new window will pop up, representing the virtual machine. It will be described in detail in the next section.

If your virtual machines need network connectivity to your real network, you have to start a network bridge, too. To do so, select **Network/connect**.

The **Launcher** is able to configure both virtual machines and network bridges. Use the corresponding **edit** menu item to get a dialog box with configuration options.

5.2 Starting a Virtual Machine

You can start a virtual machine by using the **Launcher** (see previous section), or by executing `faum-node-pc` in the directory of the virtual machine you wish to start.

If you call `faum-node-pc` yourself, you can pass some options on the command line:

- B *dir*** Change to this directory before starting the virtual machine. This directory must contain all the configuration files needed for this virtual machine.
- D *#*** Set debuglevel to *#*. Only really useful for FAUmachine developers.
- d** Same as `-D 1`.
- e** This option is only useful for the Automatic Experiment Controller.
- h** Display short usage information.
- p** Automatic power on at startup. This is equivalent to pressing **On** immediately. Especially useful with `-X`, as there is no other way to power on the machine without the GUI.
- o** Automatic termination of the `faum-node-pc`, if the power led changes from on to off.
- t** Map an incoming sigterm to the frontend to a CTRL-ALT-DEL into the virtual system.
- r** Reuse TCP-ports with `bind` in case you messed up everything during the last run, to avoid waiting for timeout before the ports become reusable. Don't forget to kill all pending `faum-node-pc` and `faum` processes!
- g** Choose one or more user interfaces (UI). If you attach more than one UI to a simulator, input to the FAUmachine is mixed between all attached UIs. Of course, output is the same on each UI. You are allowed to choose up to four UIs in parallel by supplying the option `-g` several times. UI specific options are supplied after the `-g` option up to the next `-g`.

Without this option, the X11-gtk User Interface is chosen.

At the moment FAUmachine supports several types of UIs:

X11-gtk This is a graphical user interface using the GTK graphics library. If you want to enable a specific virtual keyboard choose the `-K` option followed by `DE` or `US`, the only supported keyboard options. In addition, all the options like `-display`, `-geometry` etc. may be used for this UI.

X11-motif This is a graphical user interface using the Motif graphics library. If you want to enable a specific virtual keyboard choose the `-K` option followed by `DE` or `US`, the only supported keyboard options. In addition, all the options like `-display`, `-geometry` etc. may be used for this UI.

ascii This is a textual user interface for usage within a terminal (xterm or text console). It starts in a command mode, from which you can control the virtual machine. To see the actual output, you can connect to the graphics display or to serial consoles. By pressing “#” (hash sign) and “.” (dot), you can switch back to command mode. Type `help` to get a list of commands.

You can use `--con dev` to automatically connect to your virtual machine. This is especially useful in combination with `-p`. `dev` is either `G` for the graphical console or `S0..S3` for a serial console.

All output can be logged to stderr by specifying `--log`.

none Don't start a UI at all. Use this UI for automatically controlled FAUmachines if you don't want to see their output. This UI has no specific options.

prolong This UI can be used in combination with the `prolong` binary to reduce the UI interactions to a unix or tcp socket. Start FAUmachine started with this UI, and attach to the socket with the `faum-gui-prolong` command choosing the selected socket. This is useful if your virtual machine runs somewhere in the net and you want to access the FAUmachine console somewhere else. Example:

Start your FAUmachine on a host with IP address e.g. 131.188.33.37:

```
$ faum-node-pc-g prolong -tcp 1234
```

Then you can connect to your virtual console from somewhere via tcp by issuing the following command:

```
$ faum-gui-prolong tcp 131.188.33.37:1234
```

You may use unix sockets as well (e.g. if you have security considerations):

```
$ faum-node-pc-g prolong -unix /tmp/FAUmachine.UI
```

```
$ faum-gui-prolong unix /tmp/FAUmachine.UI
```

The following option is only available to you, if you are running a specially patched FAUmachine kernel on your real machine (see section 2.1.2 in chapter 2 "Installation of a FAUmachine Binary Distribution"):

-E n If you run FAUmachine with an accelerated kernel, the default is to use all acceleration methods which are supported by the real kernel. If you don't want that, use 0 to run without any acceleration, 1 to run only with signal-handler acceleration and 2 to run with signal-handler and mmap acceleration.

When you start a virtual machine, a new window is displayed showing the console of your virtual machine (unless you use `-X`, of course). This window consists of a small toolbar and the big console display. The toolbar contains the power-on, power-off and reset buttons as well as a button to exit the simulator. The console display shows the contents of the virtual monitor.

In addition to powering on and off and resetting the machine, the menu **Machine** allows you to change the virtual CD.

The menu **Screenshot** lets you take a screenshot, which will be saved in the virtual machine's directory. You can use these screenshots to automate graphical user interfaces with Expect as described in chapter 7 "Creating an Automatic Load Generator for a Virtual System".

The menu **Inject** lets you inject errors (see section 8.2 in chapter 8 "Injecting Faults into the Hardware of a Virtual Machine" for details).

Click on **On** to power-on your virtual machine. You should see the bootscreen of the FAUmachine BIOS. It tries to boot from the CD-ROM and hard disk.

To install an operating system in your virtual machine, you have to insert your Install-CD. Use **Machine/Change CD/Browse** to select a CD-image (often named *.iso) or just select **Machine/Use host**

CDROM and insert a real CD-ROM into your real drive (`/dev/cdrom` will be used). The virtual machine should now boot from that CD, allowing you to install the operating system as you would do on a real machine.

Please note: Currently only *RedHat 8.0/9*, *SuSE 8.1/8.2* and *Debian 3.0r1/r2* are supported out of the box. I.e. if you try to install another operating system/Linux distribution, the **Launcher** will output an error message and the virtual machine will not boot. However, you can make them work by compiling a modified version of their bootloader and kernel. See chapter 10 "Building FAUmachine from the Source Distribution" for details.

If you want to interact with the virtual machine, move your mouse pointer over the console display and click or press any key. Now your mouse and keyboard is redirected to the virtual machine. To stop that redirection, press **Ctrl+Alt+ESC**. Now all mouse and keyboard presses are back to normal again.

6 Connecting a Virtual Machine to the Real Network

This chapter describes, how to connect your virtual machines to the real network.

6.1 Overview

To connect a virtual network to real machines, there is a special FAUmachine Networking Process which transfers network packets from and to a virtual machine.

This binary is called `faum-bridge-net`; only a single invocation is allowed.

The configuration file is very similar to that of the virtual ethernet interfaces.

There are several modes of operation for the network bridge: *slirp* or *tuntap*.

6.1.1 slirp Mode

When using this mode, network packets sent by the virtual system are interpreted by the network bridge, which issues corresponding system calls on the hosting machine. This way, the virtual machine can connect to any machine or service that is reachable for a normal user on the hosting machine, too. However, it is not possible to listen for incoming connections and to use special services that depend on privileged ports (for example nfs).

This mode is recommended for most users as it is easy to set up and does not require special privileges. We have included a slightly modified *slirp* (for the specific needs of FAUmachine) with FAUmachine. The original *slirp* can be found at <http://slirp.sourceforge.net/>.

6.1.2 tuntap Mode

When using the tuntap mode, a new network interface is created on the hosting system. All network packets transferred or received at this interface are transferred to/from the virtual network.

By using standard network tools, this new interface can be configured to forward or bridge packets to a real network interface. The network bridge needs access to `/dev/tuntap` which can be achieved in usermode by setting access permissions for this device. But setting up the new interface requires root privileges for the `ifconfig` calls. The transfer rate is much higher than with *slirp* (about 500kB/s here).

6.2 Configuration of the network bridge

The easiest way to set up a network bridge for your virtual network is to use our Launcher which can create the configuration for you. If you create a virtual machine, you can connect it to a virtual network. Each virtual network can be connected to the real network via a network bridge. If you select 'Bridged' or 'Routed' network in the Network/Edit dialog, then a corresponding network bridge configuration will be created in the `/FAUmachine/net#` directory (where # is a number).

The rest of this sections describes the configuration files in detail.

6.2.1 Connecting to virtual machines

Configuration of the network bridge is stored in a directory, just like the configuration of your virtual machines. This directory contains the files `host`, `name`, `upstream` and `upstream-method`.

The files `host` and `name` are used by the **Launcher** only, the network bridge itself ignores them. `host` tells the **Launcher** on which real machine the bridge should be started. `name` contains the network name displayed in the **Launcher**'s menus.

The files `upstream` and `upstream-method` are read by the network bridge. `upstream` lists all those virtual machines belonging to this virtual network. It looks similar to the NE2000 config for virtual machines (see section 4.6), but does not contain an interrupt number or IO address.

`upstream-method` contains configuration information for the method used to connect the virtual machines to the real network (either `slirp` or `tuntap`). The following sections give the details.

6.2.2 slirp configuration

There are two submodes that select how the network bridge responds to ARP requests.

If you use `"slirp router ip-address"` in your `upstream-method`, the network bridge responds to the given IP-address, which should be used as the default router in the virtual system.

If you use `"slirp bridge eth-interface"`, the bridge forwards requests from the virtual network to the ethernet interface on the real machine given in the file. This way, a virtual machine will think that all real machines present in the real network are present in the virtual network as well. However, all `slirp` constraints are still valid and real machines cannot see the virtual machines.

To avoid some traffic in the real network (necessary for ARP-lookups), you can explicitly provide a list of IP-address/ARP-address pairs for those real machines, which should be visible in the virtual network. Create a file `slirp-arp`, consisting of lines with mac address and IP address, as in `/etc/ethers(5)`. To use only this file, without any automatic configuration, put `"slirp manual"` into `upstream-method`.

6.2.3 tuntap configuration

Tuntap mode is enabled by putting `"tuntap eth-interface"` into `upstream-method`. The specified interface will be created on the real machine. This interface can be used to communicate with the virtual machines.

In order to use the interface, it has to be configured, requiring root privileges. Everything that is possible with a normal network interface can be accomplished with the new interface.

There are two common configurations which are described below. If you want to automatically set up the interface, you should have a look at the network setup of your linux distribution. They often provide a flexible framework which can do that.

- Routing:

Set up the interface: `ifconfig FAUmachine0 10.10.1.1 up`, then enable forwarding:
`echo 1 > /proc/sys/net/ipv4/ip_forward.`

- Bridging:

You need a kernel with bridging support and the corresponding userspace tools (`bridge-utils`). Then create a bridge (`brctl addbr br0`) and add all interfaces that should be connected to that bridge (`brctl addif eth0; brctl addif FAUmachine0`). All interfaces that are added to the bridge should be up, but without assigning an IP address (`ifconfig FAUmachine0 0.0.0.0 up`). When the bridge is ready, you can assign an IP address and routes to that bridge as you would do with a normal interface. All interfaces will be connected so that it looks as if your virtual machines are part of the normal network.

6.3 Network Configuration of the Virtual Machines

Both slirp and tuntap mode have two submodes: router and bridge. Network configuration of the virtual machines has to be different for each of these submodes.

In bridge mode, the virtual machines have to be configured similar to the host machine. That means, you need the same network/netmask and the same default route, but a different, unique IP address.

In router mode, the virtual machines are in a different network. You can use any IP address which belongs to that net. If you do not know which net you should use, just take network 10.0.0.0 / netmask 255.0.0.0. The default route you configure in your virtual machines must match the IP address specified in the `slirp router` config or the one given to the tuntap interface.

7 Creating an Automatic Load Generator for a Virtual System

This chapter treats testautomation using VHDL and the automatic experiment controller Expect. Expect can be seen as driver system for your virtual machines. Expect is able to observe monitor and/or serial ports and trigger actions if patterns appear at the observed interfaces. Expect is able to send characters to virtual serial interfaces and/or to move the mouse and/or to press keys on your virtual keyboard. The language to describe these interaction is VHDL. Basically Expect can simulate the user sitting in front of your virtual machine and interacting with it.

The second task of Expect is to describe the (hardware) setup of your virtual system which may consist of several virtual machines connected by network (ethernet or serial ports) to each other. Expect can also be used for describing the hardware only without applying a load to that system, see option `-n`. If you want to generate the virtual hardware setup only without starting up the machines and simulating the virtual users sitting in front of the machines, call `faum-expect-n` in the directory describing your virtual system setup.

The third task of Expect is to activate faults in your virtual system according to your descriptions in the VHDL-model. All faults available with the menus of a FAUmachine can be triggered from VHDL script, too.

To use Expect, you will need to create a file `simulation.setup` containing some general information about the environment and a file `system.vhdl` containing a VHDL-model of your system (including virtual hardware and the actions of the simulated user). If you do it right, calling `faum-expect` in the directory containing these files handles everything else. Virtual systems described in `system.vhdl` are created in the current directory (if none have been created yet).

Section 7.1 Sections 7.2 and 7.3 explain how to describe the virtual hardware in `system.vhdl`. Read sections 7.4 and 7.5 to find out how to model virtual users, so you can have a coffee while the virtual users do some testing or experimenting for you. For automated fault injection please refer to section 8.3.

The driver system is started by `faum-expect`. If you have prepared all the files described in the next sections and have used the default names, all you have to do to start your experiment is run `faum-expect` in the directory containing the files. `faum-expect` will create the virtual hardware (if it doesn't exist yet) and will then activate the virtual users so they can go about their business.

Please note, that in VHDL the term *generic* is used to refer to variables. The term *base entity* usually refers to the VHDL-component representing your system.

`faum-expect` currently takes the following options:

- `-d loglevel` Set the logging level for debug-output to something between zero and 9. Useful for developers, but not useful for finding errors in VHDL-models.
- `-e integer` This option runs expect several times. You must declare a generic named `run_index` in your base entity in `system.vhdl` which holds the number of the run during simulation. All logfiles will be indexed automatically. This option can be used to model faultloads.
- `-f filename` Request to write a list of possible faults into file *filename*.
- `-gint generic=value` Set a generic name *generic* (appearing in the base entity in `system.vhdl`) to the integer value *value*.
- `-n` Generate virtual hardware only and don't start the load generator.
- `-po` Causes the VHDL-parser to verbosely tell you what it's doing. May be useful to find errors in your VHDL-script.
- `-setup file` Load a file *file* instead of the default `simulation.setup`.

7.1 Setting up the Test Environment — `simulation.setup`

`simulation.setup` contains some information concerning the environment of the experiment.

If you rename `simulation.setup`, you will need to pass its name to `faum-expect` using the option `-setup`. `simulation.setup` is structured into the global sections `[options]`, `[lib:libname]` and section for each component, which you wish to configure. To begin a new section place `"[section-name]"` at the first character of a line.

7.1.1 The `options`-Section of `simulation.setup`

The **options**-section of `simulation.setup` covers variables being relevant for running the driver system.

The two most important pieces of information are the name of the file containing the VHDL-model (keyword `vhdl_model`) and the name of the system described in this model (keyword `base_entity`). The name of the system is identical to the name of the component topmost in the hierarchy of VHDL-components. The line naming the system must be present in `simulation.setup`.

If you rename `system.vhdl`, you'll have to change a line in `simulation.setup`. Figure ?? shows an example for a minimal `simulation.setup`.

The following items explain all the available options:

vhdl_model Sets the name of the file containing the VHDL description of your virtual system.

base_entity Use the entity (VHDL-component) with this name as base-entity. The base-entity is at the top of the component hierarchy and usually contains the description of the complete system. (Default is to use `system` as base-entity.)

index Here a short explanation, without going into too much detail. FAUmachine network, serial or other connections are simulated with sockets (see section 4.6). Each socket is uniquely identified by an IP-address/port-number pair. Each virtual machine needs one socket for each virtual network interface it has. If several virtual setups are generated (i.e. several independent groups of virtual machines with each group being interconnected) and run on the same host, there will be a conflict, since automatic setup always generates the same port numbers. To avoid this, use `index`. The value given for `index` must be an integer greater than zero. It is used to modify the default autogenerated port-number and default autogenerated hardware address of a virtual machine. `Index` also changes the network generated for the dhcp. Given an `index` of 8, for example, the virtual machines would be configured to use network 10.8.0.0/24. (Default is to use an **index** of zero.)

log_path Sets the log-path. The `stderr` of all the started FAUmachine systems and other helpers will be logged into files in the directory specified here. If `log-path` is not set, logs are written to the current directory. The files are named `errors.node-name`.

node_path Sets the path, where the virtual machine directories will be created. If not set this will be the current directory.

local_host If you start a distributed simulation and some nodes are started locally (e.g. serial terminals) remote nodes are addressing the local nodes with 127.0.0.1 which does not connect. This option replaces the loopback address by the IP provided in the value field.

```
[options]
vhdl_model      single-machine.vhdl
base_entity     machine
```

Figure 7.1: Minimal `simulation.setup`

7.1.2 The `lib`-Section of `simulation.setup`

VHDL is able to handle libraries. Libraries are used to reuse code and can contain VHDL-functions, -procedures and -component-definitions. Consult a VHDL programming manual for details. A VHDL-library may be split across several source files. The `lib`-section associates a VHDL-library name with a list of source files belonging to this library. If you now use a **library**-clause in VHDL, Expect will load all the source files of this library as defined in the appropriate `lib` section.

Lets say you defined a set of VHDL-procedures for starting up your virtual machines and logging in a virtual user. These procedures are defined in the files `load_gen.system_startup.vhdl` and `load_gen.user_login.vhdl`. If you want to access the procedures in these files in your VHDL-script by using

```
library load_gen;
use load_gen.types.all;
use load_gen.procedures.all;
```

then your `simulation.setup` has to include the following lines:

```
[lib:load_gen]
load_gen.system_startup.vhdl
load_gen.user_login.vhdl
```

Keep in mind, that there must be no spaces in these lines.

7.1.3 The Components-Sections of `simulation.setup`

For some components of the virtual machine and the helper nodes

an additional section may be created describing parameters strongly coupled to that component. The names used in the section header `[component-name]` must be the names defined in the VHDL-description of your system (usually in `system.vhdl` (see section 7.3).

The following sections cover the possible parameters for each component type.

The most common use of these sections is to define the simulation mode of a virtual machine (where it should be executed and maybe if it should not be started automatically), a `serial_terminal` (if it should write log-files and/or open a window) and the `harddisks` (if they should be created from a image, if they should use copy on write).

You begin a component-section with name of the component according to the VHDL model in brackets at the start of a line, e.g.:

```
[/server]    # start of component-section for /server
              # place options for /server here

[/client]    # start of another component section for /client
              # place options for /client here
```

The options for a component are pairs of keywords and values.

All those components which are standalone, support the options **host** and **dont.start**.

host Set the real machine on which to run the virtual machine or node. You must use an IP-address to identify the real host. Addressing by hostnames is not supported yet. `faum-expect` creates an ssh connection to the given real host before starting a virtual machine or node.

dont.start Supplying a value not equal to **no** suppresses starting the node. Default value is **no**, i.e. this node will automatically be started.

Standalone-components are, for example virtual machines (represented by their `ctrl`-component), terminal emulation components (`serial_terminal` or `serial_pattern`), network connectors (`network_router` or `network_bridge` for network bridging from real to virtual network). Components such as `harddisks`, `networkcards` or `parallel interfaces` are not considered standalone.

Figure 7.2 shows an example configuring two virtual machines, a client and a server.

```
[/client]    # start of another node section for /client
host        131.188.33.33    # IP address where /client should be executed
dont_start  yes              # don't start this node

[/server]    # start of node section for /server
host        131.188.33.32    # IP address where /server should be executed
# omitting dont_start means start this node

[options]
vhdl_model  system.vhdl
base_entity system
```

Figure 7.2: Example of **simulation.setup** for Simple Client-Server System

7.1.3.1 Options of Virtual Machines

A virtual machine always contains a single instantiation of the entity **ctrl** as one of its subcomponents. A virtual machine is uniquely identified by the name of its **ctrl**-component. In `simulation.setup` the following options are possible for a virtual machine component:

keyboard This option can be set to `us` or `de`. Don't use X-keycode translation for keyboard simulation but translate keysyms instead. This is only useful if `faum-node-pc` is started remotely with X11-redirection. If unsure, don't set this option. Corresponds to option `-K` of `faum-node-pc`.

no_window If this option is set to something not equal `no`, the GUI window of a virtual machine is not created. Default is to show the GUI. Corresponds to option `-X` of `faum-node-pc`.

autostart If this option is set to something not equal `no`, the virtual machine is automatically powered on at startup of the simulator. Corresponds to option `-p` of `faum-node-pc`.

dont_use_kernel_acceleration If set to **both** or **yes** kernel acceleration will not be used. If set to **mmap** only signal-handler acceleration is used and `mmap` acceleration is not used. All other values enable both acceleration mechanisms, if present. Corresponds to option `-E` of `faum-node-pc`.

debug If this option is set to something not equal to `no` the virtual machine will be started with the debug option set. This creates lots of debug output. Corresponds to option `-d` of `faum-node-pc`.

7.1.3.2 Options of Harddisks

A harddisk is an instantiation of the entity **idedisk**. In `simulation.setup` the following options are possible for a virtual harddisk component:

image The value for this options is the path to a valid disk image. If omitted a clean (all data is zero) diskimage is created.

create If set to something not equal `no`, Expect will create the diskimage of the corresponding disk each time Expect is started. If omitted `no` is assumed.

cow If set to something not equal `no` Expect will create the images in copy on write mode. That means the image will never be written to. All the changes will be recorded in a separate files (`cow`, `map`). If omitted `no` is assumed. You can merge the recorded changes into the original image using `faum-mergecow` (see section 4.4.3).

path In case of distributed simulation you may specify a path, where your virtual disk will be created. This is useful because the configuration directory must be shared between all the real hosts, which are used to run virtual machines and therefore must be an NFS or similar mount. Using disk images over NFS may be too slow. Thus, you can enter the path where the diskimage should be created relative to the remote system where your virtual machine runs. If omitted no redirection is done.

7.1.3.3 Options of DVD/CDROM Drives

A DVD/CDROM drive is an instantiation of the entity `idecdrom`. In `simulation.setup` the following options are possible for a virtual DVD/CDROM component:

image The value for this options is the path to a valid image, which will be loaded into the drive at startup.

7.1.3.4 Options of Serial Connected Components

There are four components which can be connected to a serial interface of a virtual machine:

`serial_terminal`

`serial_pattern`

`serial_ip`

`serial_unixpty`

If these components are used in `system.vhdl`, Expect will start them automatically. In `simulation.setup` the following options are possible for the virtual components listed above:

no_window If this option is set to something not equal `no`, no window will appear for this terminal. The default value is `no`.

font If this option is set and if a window is opened, the window will be opened using the suggested font, adding `-fn value` to the call of the started xterm.

logging If this option is set to not equal `no` all outputs on this serial interface will be logged into a file while the virtual system is running. The file is placed in the current directory or if `log_path` if the `option`-section is set to that directory. Default is not to log.

debug If this option is set to not equal `no` serial.interface is started with the debug option set. This enables debugging output of the serial component.

log_stamp If this option is set to not equal `no` every new line will be preceded by a time stamp of the first character which was received after a newline. Default is to leave timestamps out.

input If this option is set to not equal `no`, window inputs (your interaction with the window) is mixed with the input created by Expect. Default is to ignore window inputs during Expect runs. This option is useful for some manual tuning when developing a load generator script.

7.1.3.5 Options of Network Components

There are two components that connect virtual networks with the real network:

`network_bridge`

`network_router`

Both of these VHDL-components will start the network bridge described in chapter 6 "Connecting a Virtual Machine to the Real Network".

In `simulation.setup` the following options are possible for the virtual network components listed above:

method This option must be set! There are two possible values for this option: `slirp` and `tuntap`. They define how the network component should operate, see 6.1.

iface This option must be set! It describes, for the `slirp`-method to which interface of the real system the bridging/routing process should bind (see 6.2.2). For the `tuntap`-method this name is used for the name of the tuntap device (see 6.2.3).

7.2 The Scripting Language — VHDL

To make maximum use of the examples, some knowledge of VHDL syntax and semantics is required. Check out an online-tutorial if you are new to VHDL. Here is a list of VHDL tutorials online at the time of writing.

- <http://www.cs.ucr.edu/content/esd/labs/tutorial/>
- <http://www.gmvhdl.com/VHDL.html>
- <http://www.eng.auburn.edu/department/ee/mgc/vhdl.html>
- http://www.seas.upenn.edu/ee201/vhdl/vhdl_primer.html
- <http://www.vhdl-online.de/tutorial/>

When browsing through the tutorials, do keep in mind, though, that you do not need to describe any hardware at gate level or design any chips. FAUmachine is distributed with a library of ready-made VHDL-components such as network cards, harddisks, floppy-drives etc. What is important, though, is that you understand the syntax and semantics of VHDL so that you can read, understand and adapt to your needs the examples distributed with FAUmachine.

Keep in mind that Expect currently only supports a subset of VHDL and not the complete language. At the moment it does very unsufficient type checking ...

Caution: shortcuts currently not supported in procedures or functions!

Caution: unusual string handling!

7.3 Describing the Target System Hardware — `system.vhdl`

The automatic experiment controller Expect looks for the VHDL-model of the system used in the experiment in a file called `system.vhdl`. If you wish to rename this file, see section 7.1.

VHDL combines two ways of describing hardware: structural (what it is) and behavioural (what it does). You will use structural VHDL to describe the virtual hardware of a system.

This section tells you about the virtual hardware components provided by the VHDL-library distributed with FAUmachine. For most up-to-date information, have a look at the library itself in `/usr/share/faumachine/vhdl/expect.vhdl`.

A virtual machine consists of different predefined components which are already known to the VHDL-compiler and can be used as is, e.g. processor, memory, IDE-controller, harddisks.

You can link components by connecting their interfaces in the right way. E.g. if the interface contains a port for an ethernet cable, you can plug in such a cable and plug the other end into the appropriate port of another component's interface. VHDL doesn't allow plugging cables (the VHDL-term is *buses*) into ports of the wrong type. Having said this, you also shouldn't plug your ISDN-cable into the ethernet port of your machine!

On the one hand each component may be viewed as a black box presenting a certain interface (consisting of one or more *ports*) to the outside world. The sections

```
entity name is
    port (
        port-name : port-type,
        ...
        port-name : port-type
    );
end name;
```

contain this black box representation. On the other hand, you can also have a look at the internal structure of the component. This is done in the sections with the format

```
architecture structural of name is
    ... (1) ...
begin
    ... (2) ...
end structural;
```

(1) lists all the buses/cables inside the component

```
signal busname : bustype;
```

giving their name and type. (2) describes how the interfaces of the components are connected to these cables.

```
name : component port map(portname => busname);
```

The line above instantiates a new component *name* of the component-type *component*. The port *portname* of this component is connected to the bus *busname*.

The internal structure of the predefined components is not of interest. It is used to define the possible failures for the components and prepare the shortcut connectors which Expect can use to trigger faulty behaviour in a component (see chapter 8 "Injecting Faults into the Hardware of a Virtual Machine").

To avoid having to describe entities for similar components, entities can take parameters, which are passed when a component is instantiated from an entity description. The VHDL-term for these parameters is *generic*. The parameters (a decent programming language would probably call them variables) have a name, a type and a value.

The following list describes the library of predefined components, their interface (ports) and how to customize them via parameters (or **generics**). Figure 7.3 shows a graphical representation of the predefined components. Ignore the part about the shortcut connectors for now. You will need those for automizing tests and triggering faults with the experiment controller Expect.

cpu:

description: The processor.

port: **mem** (type **membus**, direction **inout**): Your processor is connected to the memory via the memory bus.

generics: **speed** (type **integer**): the speed of your processor

model (type **string(1 to 5)**): the make of your processor

serial (type **integer**): the serial number of your processor

shortcut connectors: **bitflip** (type **boolean**): Shortcuts to this signal additionally need the register name and the bit number, which will be flipped when this signal is activated.

ctrl:

description: In a way, the chassis of your machine. Most importantly, the power and reset buttons.

ports: **keyboard** (type **ps2bus** direction **inout**): This is where you plug in your keyboard.

mouse (type **ps2bus** direction **inout**): This is where you plug in your mouse.

isa (type **isabus** direction **inout**): This is the connection to your motherboard with the rest of the components like idctrl, mem2isa bridge, ethernet,...

poweron (type **boolean** direction **in**)

Set it to **true** if you want to power on the node.

reset (type **boolean** direction **in**)

If set to **true** and then again to **false** the system is resetted.

idecdrom:

description: A CDROM-drive which can be connected to the IDE bus.

port: **ide** (type **idebus** direction **inout**)

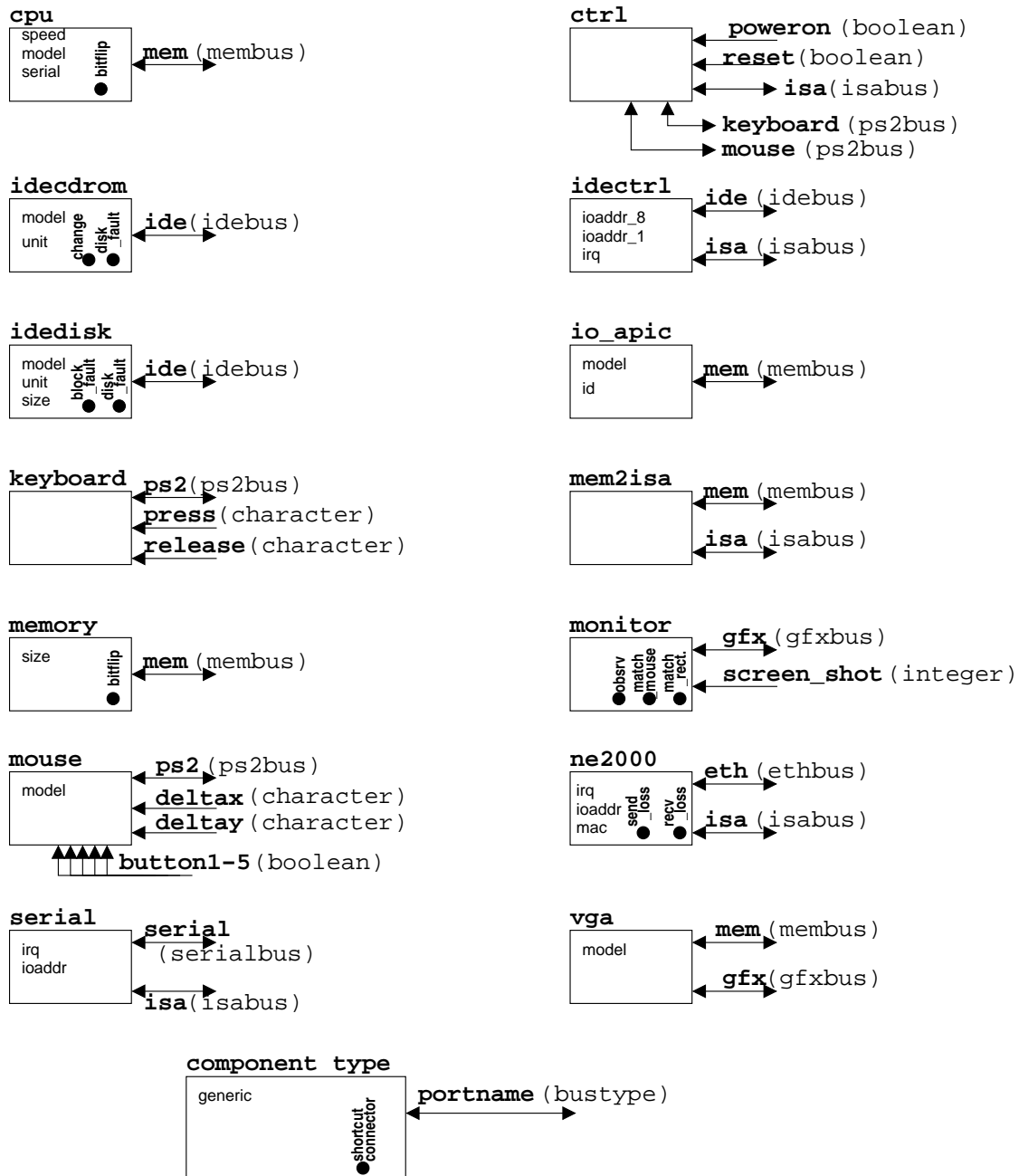


Figure 7.3: Predefined Components

generics: `model` (type `string(1 to 16)`)

`unit` (type `integer`): can be `0` for master and `1` for slave.

shortcut connectors: `change` (type `boolean`): This shortcut connector is needed, when you want to change the CD in this drive from an Expect script.

`byte_fault` (type `boolean`): Shortcuts to this signal additionally need the location of the affected byte and the range as long long arguments. Set range parameter to `0` if whole disk should be affected (for details see 7.6).

idectrl:

description: IDE-controller

ports: `isa` (type `isabus`, direction `inout`):

`ide` (type `idebus`, direction `inout`):

generics: `irq` (type `integer`): The interrupt this controller uses. If set to -1 Expect will generate irqs and ioaddresses for each instance of an IDE controller automatically (avoiding resource conflicts!).

`ioaddr_8` (type `integer`): The 8 IO addresses this controller uses.

`ioaddr_1` (type `integer`): The single IO address this controller uses.

idedisk:

description: an IDE harddisk

port: `ide` (type `idebus`, direction `inout`):

generics: `model` (type `string(1 to 15)`):

`unit` (type `integer`): Use 0 for master and 1 for slave. If you attach two units to the same IDE-controller, they must have different unit-numbers!

`size` (type `integer`): Default is 2 Gigabyte.

shortcut connectors: `disk_fault` (type `boolean`): This shortcut is used for triggering a fault of this disk.

`block_fault` (type `boolean`): This shortcut is used for triggering a fault of a block within this disk. Additional information is needed (starting block and range).

io_apic:

description: An IO-APIC for a processor (necessary for multiprocessor machines).

port: `mem` (type `membus`, direction `inout`):

generics: `model` (type `string(1 to 7)`): The make of your APIC (e.g. "82093AA")

`id` (type `integer`): Use -1 to let Expect number the APIC IDs automatically. This is the ID you must use when defining a processor with an APIC (generic `apic_id` in component `cpu`).

keyboard:

description: keyboard

ports: `ps2` (type `ps2bus`, direction `inout`): to be connected with the ctrl-component.

`press` (type `character`, direction `in`): keycode (use `showkey` to get it) of the key which should be simulated pressed

`release` (type `character`, direction `in`): keycode of the key which should be simulated released.

mem2isa:

description: (north-/south-) bridges usually integrated in chipsets. Connects memory bus to ISA bus.

ports: `mem` (type `membus`, direction `inout`)

`isa` (type `isabus`, direction `inout`)

memory:

description: The machine's main memory.

port: `mem` (type `membus`, direction `inout`):

generics: `size` (type `integer`): Size in Megabyte.

shortcut connectors: `bitflip` (type `boolean`): Shortcuts to this signal additionally need the location of the number of the byte and the bit as additional arguments.

monitor:

description: The monitor of the machine.

ports: `gfx` (type `gfxbus`, direction `inout`)

`screen_shot` (type `integer`, direction `in`): causing an event by assigning an integer number has the effect of writing a screen shot to a file named `screenshot-integer.pnm` in the node directory. *integer* is the number you have assigned.

shortcut connectors: **observe** (type **boolean**): Shortcuts to this signal additionally need arguments describing the pattern to be observed and other details. The monitor will send results of rectangles or mouse pointers matched (position etc.) to Expect along the respective **match_rectangle** and **match_mouse** shortcut connectors. For details refer to section 7.6.
match_rectangle (type **integer_array(1 TO 3)**): If the observer matches a rectangular pattern it is signaled here.
match_mouse (type **integer_array(1 TO 4)**): If the observer matches a mouse pattern it is signaled here.

mouse:

description: Pointer device (usually mouse).

ports: **ps2** (type **ps2bus**, direction **inout**)
 deltax (type **character**, direction **in**)
 deltay (type **character**, direction **in**)
 button1 (type **boolean**, direction **in**)
 button2 (type **boolean**, direction **in**)
 button3 (type **boolean**, direction **in**)
 button4 (type **boolean**, direction **in**)
 button5 (type **boolean**, direction **in**)

ne2000:

description: NE2000 compatible ethernet card

ports: **isa** (type **isabus** direction **inout**)
 eth (type **ethbus** direction **inout**)

generics: **irq** (type **integer**): The interrupt this card uses. If set to **-1** Expect does autonumbering.

ioaddr (type **integer**): The IO address this card uses. If **ioaddr** is set to **-1** Expect tries this automatically.

mac (type **string(1 to 17)**): the MAC address this component uses. Default **mac** is set to **00:00:00:00:00:00**, which means Expect generates it automatically.

shortcut connectors: **recv_loss** (type **integer**): Use this shortcut to enable receive loss fault of that adapter.

send_loss (type **integer**): Use this shortcut to enable send loss fault of that adapter.

serial:

description: Serial interface.

ports: **isa** (type **isabus**, direction **inout**)
 serial (type **serialbus**, direction **inout**)

generics: **irq** (type **integer**): the interrupt this component uses. If set to **-1** Expect does autonumbering.

ioaddr (type **integer**): the IO address this component uses. If **ioaddr** is set to **-1** Expect generates it automatically.

vga:

description: graphics card

ports: **mem** (type **membus**, direction **inout**)
 gfx (type **gfxbus**, direction **inout**)

generic: **model** (type **string(1 to 3)**): default "VGA"

In real life, components inside your machine are connected to each other with buses (e.g. IDE-, ISA-, PCI-bus). Buses are complex things with protocols, arbitor, handshake, data ports and so on. We do not need this complexity in our VHDL-description, but we do need something to describe how the components are connected to each other. We therefore map our real-life buses to simple VHDL-signals in our system description. (For VHDL-insiders: We use the signals only to describe the connection structurally, but not to handle events).

There are a number of predefined buses in our VHDL-library, namely the

- frontside bus (**membus**)
- ISA bus (**isabus**)
- IDE bus (**idebus**)
- ethernet bus (**ethbus**)
- monitor cable (**gfxbus**)
- mouse/keyboard cables (**ps2bus**)
- serial interface (**serialbus**)

Your NE2000 card, for example, has two ports: One connected to the ISA-bus (of your motherboard) and the other connected to your hub/switch or another virtual machine. The two ports together make up the card's interface. Don't make the mistake of trying to model the internal architecture of your machine in all detail! **membus** and **isabus** really stand for all machine-internal bus-systems, be they AGP, EISA, VLB, PCI, NuBus or whatever. Analogously the **ethbus** includes all kinds of network connections and the **idebus** all kinds of buses connecting the processor to storage devices.

Of course you can define your own components, building on the predefined ones. One of the first components you will probably construct in this way may be quite similar to the single virtual machine shown in figures 7.4 (graphical) and 7.5 (VHDL code). Once you have (preferably parametrized) virtual machine definitions, you can go ahead and build a network of virtual machines. For more examples please have a look at chapter 9.

To define a new component, you need to define two things: the interface (ports) it presents to the outside world and its internal structure. The interface of a component is defined in its **entity**-description. Each interface can consist of an arbitrary number (may be zero) of **ports**. A port has a certain type and can only be connected to a bus of the same type. No ports means no connection to other components!

In VHDL the term *entity* is used to mean “component-template”. Several components can be instantiated from a single template. If you have a template for a machine with some memory and a single harddisk, you

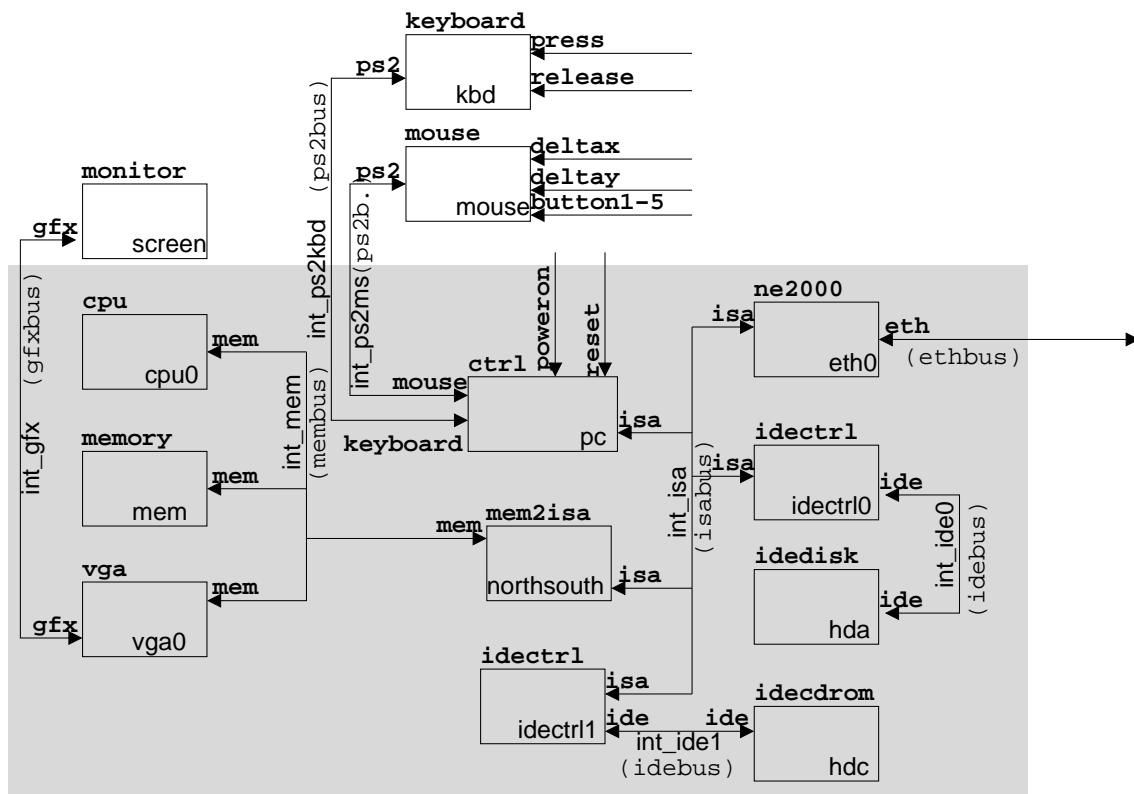


Figure 7.4: Internal Structure of a Virtual Machine

```
library expect;
use expect.types.all;
use expect.procedures.all;

5  entity node is
    generic( memsize   : integer := 256;
             disksize  : integer := 3000;
             faults    : integer := -1 );
    port( ext_eth0 : inout ethbus );
10 end node;

    architecture structural of node is
        signal int_mem : membus;
        signal int_isa : isabus;
15        signal int_gfx : gfxbus;
        signal int_ide0, int_ide1 : idebus;
        signal int_ps2kbd, int_ps2ms : ps2bus;

    begin
20        node : ctrl port map( keyboard => int_ps2kbd,
                               mouse    => int_ps2ms,
                               isa      => int_isa );

        cpu0 : cpu port map( mem => int_mem );
25        mem0 : memory generic map( size => memsize )
            port map( mem => int_mem );

        northsouth : mem2isa port map( mem => int_mem,
30                                     isa => int_isa );

        video : vga port map( mem => int_mem,
                              gfx => int_gfx );

35        mon : monitor port map( gfx => int_gfx );

        kbd   : keyboard port map( ps2 => int_ps2kbd );

        mouse : mouse port map( ps2 => int_ps2ms );
40        eth0 : ne2000 port map( isa => int_isa );

        idectl0 : idectl port map( isa => int_isa,
                                   ide => int_ide0 );
45        idectl1 : idectl port map( isa => int_isa,
                                   ide => int_ide1 );

        hda : idedisk generic map ( size => disksize,
50                                     unit => 0 )
            port map( ide => int_ide0 );

        hdc : idecdrom generic map( unit => 0 )
            port map( ide => int_ide1 );
55    end structural;
```

Figure 7.5: VHDL-Code for Figure 7.4

can instantiate a number of nodes from it, which may differ in the size of memory and harddisk (if you used generics to allow customization).

The internal structure of a component, named *architecture* in VHDL is enclosed in an **architecture**-block.

7.4 Automating Text-Based User Interfaces Via a Serial Terminal

At this stage you may want to read up on VHDL-processes, since this is what you need to describe the actions of the virtual users. Basically, what the virtual users simulated by Expect are capable of, is looking at the screen and recognizing text or pictures, typing away at the keyboard and moving the mouse. And of course, pressing the on/off/reset buttons on the computer.

For handling text based user interfaces via a serial terminal, all that is needed are procedures for recognizing text received via the serial interface and sending text out via serial interface (no mouse or graphics).

You basically have two options for setting this up. First, configure your virtual machine to send console output (remember, text-based means no X-Window, no KDE etc.!) to the serial interface and to expect keyboard input via the serial interface.

Then, for the first option, connect a **serial_terminal** to the serial interface of your virtual machine. You can now use **send_string** and **wait_string** to simulate typing at the keyboard and looking at the console output. Section 7.4.1 describes how to do this.

For the second option, connect a **serial_pattern** to the serial interface of your virtual machine. You can now connect virtual clamps to the virtual shortcut connectors of the **serial_pattern** component to tell it to let you know when certain patterns are seen on the serial bus. Use **send_string** as in the first option to simulate typing at the keyboard. Section 7.4.2 describes how to do this.

Now what's the difference between these two options? They seem to accomplish the same thing, right? Right! The difference is, **serial_terminal** sends every character that passes by on the serial bus to **wait_string** for pattern matching. **wait_string** is a VHDL-procedure and, because VHDL code is only interpreted at run time, it is slow. If there are many characters which have to be searched for a pattern, time in Expect will be slower at the virtual time in your virtual machine. Using the clamp-and-shortcut-connector method, the pattern matching is done inside the **serial_pattern** component, i.e. it happens in C, which is fast and sends only a single event, when the pattern is matched. But this method needs connecting and disconnecting the virtual clamps.

7.4.1 Using **serial_terminal** and **wait_string**

Here's how you would do this using **serial_terminal**, **wait_string** and **send_string**. For an example see section 9.4 in chapter 9 "Automation Examples" and have a look at the *single-node example*.

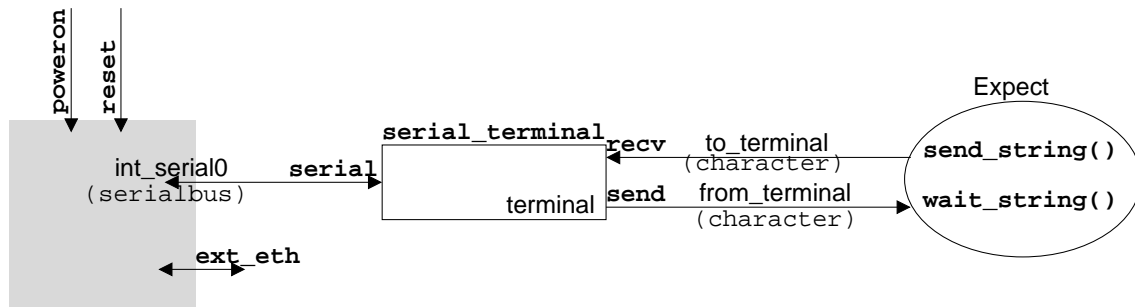
Figure 7.6 shows a graphical representation of how things are connected in this case. The grey box is the machine from figure 7.4. Figure 7.7 defines the necessary signals. Insert these lines after after line 15 and before line 20 into the VHDL-model shown in figure 7.5.

Figure 7.8 defines the **serial_terminal**. Insert this before the last line in figure 7.5.

Figure 7.9 defines the process (equivalent to the one in figure 7.13). Add this code before the last line in figure 7.5 (i.e. after the code for the **serial_terminal** you just inserted above). The process handles a login as root (up to the point of the password prompt appearing) once the computer has booted. Add this code before the last line in figure 7.5 (i.e. after the code for the **serial_pattern** you just inserted above).

Here are the details about the two procedures, necessary to communicate with the **serial_terminal**:

send_string

Figure 7.6: Connecting a **serial_terminal** Component

```
signal to_terminal, from_terminal : character;
```

Figure 7.7: VHDL-Code for Defining Signals for Use with **serial_terminal** Component

```
terminal : serial_terminal
port map(
    recv => to_terminal,
    send => from_terminal,
    serial => int_serial0
);
```

Figure 7.8: VHDL-Code for Defining the **serial_terminal** Component

```
login: process
variable found: boolean;
begin
    poweron <= 1=1;
    wait_string(found, from_terminal, "login: ", 1800000);
    send_string(to_terminal, "root", 1000);
    wait_string(found, from_terminal, "password: ", 1800000);
end process login;
```

Figure 7.9: VHDL-Code for Defining a Process (Using **wait_string**)

description: This simulates a user typing away at the keyboard.

parameters: output (type **character signal**, direction **out**): The keys typed are sent along this bus. It should be connected to your virtual machine somehow, e.g. connect it to a serial terminal on your virtual machine.

send (type **string variable**, direction **in**): The string the user types at the keyboard. Pass this as an input parameter when you call this procedure.

timeout (type **integer variable**, direction **in**): The time between single keystrokes. Pass this as an input parameter when you call this procedure.

wait_string

description: This simulates a user looking at the monitor and waiting for a certain string to appear on the monitor.

parameters: result (type **boolean variable**, direction **out**): Output variable, true when the string waited for was actually found on the bus given within the timeout given, false otherwise.

input (type **character signal**, direction **in**): Input bus, this is where to look for the

string. It should be connected to your virtual machine, in such a way, that console output, for example, comes along this bus (e.g. a serial line).

recv (type **string variable**, direction **in**): This is the string we are looking for. Pass this as an input parameter when you call this procedure.

timeout (type **integer variable**, direction **in**): This is how long we wait before we give up. Pass this as an input parameter when you call this procedure.

And here is the **serial_terminal**:

serial_terminal

description:

ports: **serial** (type **serialbus**, direction **inout**)

recv (type **character**, direction **in**)

send (type **character**, direction **out**)

generics: **x** (type **integer**): Number of characters which fit on this terminal horizontally (default 80)

y (type **integer**): Number of lines which fit on this terminal vertically (default 25)

7.4.2 Using **serial_pattern** and Shortcut Connectors

To use this option, all you need to know is how to connect a virtual clamp to one of those virtual shortcut connectors from somewhere within your VHDL-code. Of course your virtual clamp must be attached to some cable on your side. To create such a cable inside your VHDL-model, simply define a signal of the same type as the bus attached to the shortcut connector you want to connect to. Don't try to use **wait_string** in combination with the **serial_pattern**, it won't work.

Figure 7.10 shows a graphical representation of how things are connected. The grey box is the machine from figure 7.4. The small black circles represent the shortcut connectors to which the virtual clamps are attached.

Figure 7.11 defines the signals necessary to use the virtual-clamp-on-virtual-shortcut-connector method. Insert these lines after after line 15 and before line 20 into the VHDL-model shown in figure 7.5.

Figure 7.12 defines the **serial_pattern**. Insert this before the last line in figure 7.5.

Finally, figure 7.13 defines the process (equivalent to the virtual user's actions) for automatic login as root

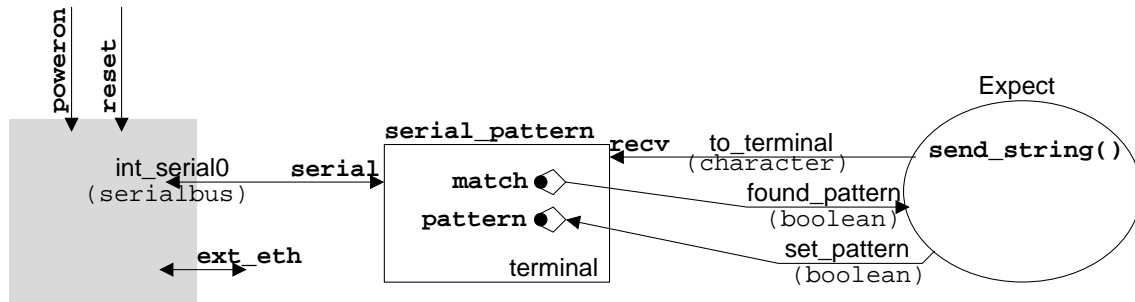


Figure 7.10: Connecting a **serial_pattern** Component

```
signal set_pattern, found_pattern : boolean;
signal to_terminal : character;
```

Figure 7.11: VHDL-Code for Defining Signals for Use with **serial_pattern** Component

```
terminal : serial_pattern
  port map(
    recv => to_terminal,
    serial => int_serial0
  );
```

Figure 7.12: VHDL-Code for Defining the **serial_pattern** Component

```
login: process
begin
  poweron <= true;
  found_pattern' shortcut_in("/terminal%match/0");
5  found_pattern' activate("login:");
  wait on found_pattern until found_pattern for 60000;
  assert found_pattern report "login prompt not detected" severity failure;
  found_pattern' activate("password:");
  wait on found_pattern until NOT found_pattern;
10 send_string(to_terminal, "root\0d", 1000);
  wait on found_pattern until found_pattern for 60000;
  assert found_pattern report "password prompt not detected" severity failure;
  found_pattern' deactivate;
  wait on found_pattern until NOT found_pattern;
15 end process login;
```

Figure 7.13: VHDL-Code for Defining a Process (Using Shortcuts)

(up to the point of the password prompt appearing) once the computer has booted. Add this code before the last line in figure 7.5 (i.e. after the code for the **serial_pattern** you just inserted above).

Here's an explanation of what happens in figure 7.13:

line 3 Set signal `poweron` to true. This turns on the virtual machine. `poweron` is one of the signals defined in the virtual machine (see figure 7.5 line 19).

line 4 Connect the signal (virtual cable) `found_pattern` to the signal `match` of the component `terminal` using one of our virtual clamps. `shortcut_in` is the function to use for the actual clamping. Refer to 7.6 for details concerning the format of the parameter passed to the `shortcut_*` functions. We expect the component `terminal` to set this signal to true when it has found the pattern. The parameter 0 after `match` tells Expect to connect `found_pattern` to the pattern matcher with number 0. There are up to 31 pattern matcher accessible.

line 5 Send information about the pattern we want the component `terminal` to look for. Again after

line 6 Wait until the pattern is found or 60 s have passed.

line 7 If the pattern did not appear within 60 seconds, write out a log message and terminate simulation (level failure).

line 8 Program the pattern matcher to look for the password prompt. Because we now match the login prompt, this signal must become false immediately.

line 9 Wait until the pattern matcher got recognizes the new pattern. This is a kind of handshake between expect and the virtual machine.

line 10 Enter the login name. Wait 1 second between each character.

line 11 Wait until the virtual machine prompts for the password or 60 seconds have passed.

line 12 Report an error and terminate simulation if there was no password prompt within 60 seconds.

line 13 Deactivate the pattern matcher mechanism.

line 14 Wait until the pattern matcher acknowledges this deactivation.

The procedure **send_string** is the same one as explained in section 7.4.1. The **serial_pattern** component is defined as follows:

serial_pattern

description:

ports: **serial** (type **serialbus**, direction **inout**)
 recv (type **character**, direction **in**)

shortcut connectors: **match** (type **boolean**): attach a virtual clamp here, so the component can let you know, when it has found a pattern.

7.5 Automating User Interfaces Via Monitor, Mouse and Keyboard

If you want to automate user-machine interaction where the user is actually sitting in front of the machine, looking at the monitor and using mouse and keyboard to give input to the machine, you will need to use graphical pattern matching (as in 'looking at the monitor), whether what you are looking at is text or pictures (it's all just pixels to your monitor).

Test automation of graphical user interfaces is possible using virtual clamps and is very similar to the method described in section 7.4.2. Except that you will use graphical patterns instead of strings and might want to move the mouse in addition to typing away at the keyboard.

Section 7.5.1 tells you how to prepare the graphical patterns. Section 7.5.2 treats automatized typing on the keyboard, section 7.5.3 automatized mouse movements. Watching and recognizing patterns on the monitor is treated in section 7.5.4.

The examples in the directories `install-SuSE-*-vesa` (see section 9.2 in chapter "Automation Examples") install a complete SuSE distribution automatically.

7.5.1 Preparations

To automatize interaction with graphical user interfaces (GUI), you will need more than the simple strings used in section 7.4 with text-based user interfaces. Instead of strings, you will need to recognize graphical patterns such as buttons or mouse cursors. We make a distinction here between mouse cursors (which may take different forms like the classical arrow or hourglass) and all other patterns (such as buttons). We will henceforth refer to patterns of mouse cursors as cursor-patterns and other patterns as non-cursor patterns.

The patterns must currently be saved as ASCII-PPM files. Since you pass the full filename of the file containing a non-cursor pattern to the pattern matching routine, you can put non-cursor patterns anywhere. All cursor patterns must be in the same directory, as only the directory is passed to the pattern matching routine, which then watches for all the different cursors found in this directory.

To generate a screenshot of a GUI running in a virtual machine as an ASCII-PPM file use the **Create Screenshot** menu entry in the **Screenshot** menu. This will create a file called `screenshot###.ppm` in the virtual machine directory. `###` are digits numbering the files. When saving the file the next available number will be chosen automatically. `screenshot###.ppm` will contain a screenshot of the complete screen.

You will now have to use your favorite program (such as `gimp` or `xv`) to edit the screenshots and cut out the part (e.g. button) you want to detect automatically. Save the file under a name of your choice in the ASCII-PPM format.

Parsing of the pattern files is currently still very crude and you will therefore now have to use your favorite text editor (such as `vim` or `xemacs`) to remove the additional headers added by the graphics program from the file.

The first four lines of each ASCII-PPM file *must* look as follows (no leading whitespace):

```
P3
# ignore decimal
width height
256
```

Here is an explanation of the lines given above:

line 1 A magic header.

line 2 A crosshead character followed by the keyword “ignore” and a decimal number (each separated by spaces), telling the pattern matcher which colors to ignore when matching. You will usually use this when matching mouse cursors (which are not usually rectangular) to ignore the background in the rectangle you cut out around the mouse cursor.

Use “-1” to not ignore any colors by writing the following:

```
# ignore -1
```

Otherwise, to get the number to put behind the “# ignore”, write down the color you wish to ignore as RGB value (usually three-byte hexadecimal number) and transform this value into a decimal number.

line 3 Contains the height and width of the pattern. This is generated by the graphics program.

line 4 Must be 256.

You can use the `faum-fixppm` tool to automatically convert standard PPM files to this special format.

7.5.2 Using the Keyboard

The keyboard is slightly different to send characters out to a serial connection. Instead of sending ASCII codes you have to provide keystrokes, which are interpreted by the virtual machine according to the keyboard model you choose.

send_keyboard

description: Use this procedure to type away at the keyboard, i.e. to press and release keys in a certain order. Remember, that the letter “A” is created by pressing the Shift-Key, then pressing the “a” key, then releasing both keys! Cannot be used to press mouse buttons.

parameters: `press` (type **character signal**, direction **out**): Output variable. Expect will send the keycodes of the keys you want to press along this signal.

`release` (type **character signal**, direction **out**): Output variable. Expect will send the keycodes of the keys you want to release along this signal.

`kbd_type` (type **integer variable**, direction **in**): Currently only 0 (us keyboard layout) is supported.

`send` (type **string variable**, direction **in**): This is the string you want to type. Don’t worry about pressing and releasing keys in the right order to generate capital letters etc. Expect will do this for you automatically (mostly, anyway ;-)

`timeout` (type **integer variable**, direction **in**): The time to wait between keypresses and releases. 100 works fine.

7.5.3 Controlling the Mouse

Moving the Mouse To move the mouse by an offset, you can simply connect signals to the mouse’s two ports `deltax` and `deltay` (**character signals**, compare with definition of mouse in section 7.3) and then send offsets along these signals to move the mouse. Because we are dealing with character signals here, you can move the mouse at 128 screen units right or down and 127 screen units left or up. Positive values move the mouse right or down, negative values to move it left or up (origin is in the upper left corner of the screen).

Pressing Mouse Buttons To press and release mouse buttons, you can simply connect signals to the mouse's button-ports `button[1..5]`. Assign boolean `true` value to a button to press it, assign `false` to release it.

control_mouse

description: Use this procedure to move the mouse near the absolute coordinates given, where near in this case is within 4 pixels of the absolute coordinates given. It's just as hard for Expect to position the mouse accurately on a given pixel as it is for you! Cannot be used to press mouse buttons.

parameters: `target_x` (type **integer variable**, direction **in**): Input variable, the new horizontal position of the mouse in screen units (pixels).
`target_y` (type **integer variable**, direction **in**): Input variable, the new vertical position of the mouse in screen units (pixels).
`mouse_coords` (type **integer_array(1 to 3) signal**, direction **in**): This parameter must be a signal/virtual cable, which has been virtually clamped onto the **match_mouse** connector of the **monitor** component (section 7.3). It is passed on to **detect_mouse**. The coordinates where the mouse is found are stored there.
`dx`, `dy` (type **character signals**, direction **out**): These signals must be connected to the mouse's **deltax** and **deltay** ports. Expect will use them to send the appropriate offsets to the mouse to move it to the target position.

detect_mouse

description: This procedure is used internally by **control_mouse**. It detects the current absolute coordinates of the mouse pointer on screen. If the mouse is hidden at the border of your screen it tries to move it a little bit around to discover it.

parameters: `found_x` (type **integer variable**, direction **out**): Output variable, the current horizontal position of the mouse in screen units (pixels).
`found_y` (type **integer variable**, direction **out**): Output variable, the current vertical position of the mouse in screen units (pixels).
`mouse_coords` (type **integer_array(1 to 3) signal**, direction **in**): Needs to be connected to the **match_mouse** shortcut connector of the **monitor** component. Values of the output variables `found_x` and `found_y` are set from information received on this signal.
`dx`, `dy` (type **character signal**, direction **out**): This parameter must be connected to the **deltax** and **deltay** signals of your mouse. Delta moves are sent along this signal/cable to the mouse.

Figure 7.14 shows a short example. Lines 2 through 20 of figure 7.14 are part of the hardware description. You can only use a mouse, if there is one attached to your computer!

Lines 21 through 39 are part of a process called `remote_control`, e.g. to automatically install a newly created virtual machine.

In line 27, we connect a shortcut to the signal `mouse_coords` to the shortcut connector `match_mouse` of our monitor component `mon`. We then activate the mouse-matching in line 30 (passing the directory containing the pictures of the mouse pointers as parameter).

In line 31, we move the mouse to the coordinates saved in the variables `x` and `y`. Usually, these would be the coordinates of the middle of a button or some such.

Lines 32 through 34 click mouse button 1 once.

We deactivate the mouse-matching in line 35 (and wait for the monitor to acknowledge the deactivation in line 36).

At the end of the process, we disconnect the shortcut in line 38.

7.5.4 Watching Screen Output

To be able to detect the current mouse position or an OK button, you will have to watch screen output. Detecting the mouse has been discussed in the previous section. This section deals with finding specific

```
...
architecture structural of node is
    ...
    signal m_dx, m_dy : character; -- mouse movement
5    signal m_b1, m_b2, m_b3, m_b4, m_b5 : boolean; -- mouse buttons
    signal mouse_coords : integer_array(1 to 3);
    ...
begin
    ...
10    mouse : mouse port map(
        ps2 => int_ps2ms,
        deltax => m_dx,
        deltax => m_dy,
        button1 => m_b1,
15        button2 => m_b2,
        button3 => m_b3,
        button4 => m_b4,
        button5 => m_b5
    );
20    ...
    remote_control : process
        variable x, y : integer;
        ...
    begin
25        ...
        -- connect shortcuts for mouse control
        mouse_coords' shortcut_in("/mon%match_mouse");
        ...
        -- point and click
30        mouse_coords' activate("2/../../suse82-pointers");
        control_mouse(x, y, mouse_coords, m_dx, m_dy);
        m_b1 <= true;
        wait for 300;
        m_b1 <= false;
35        mouse_coords' deactivate;
        wait on mouse_coords;
        ...
        mouse_coords' shortcut_in_remove("/mon%match_mouse");
    end process remote_control;
40    ...
end structural;
```

Figure 7.14: Automated Mouse Movement

patterns (like buttons, scrollbars etc.) on the screen. Finding a specific pattern on the screen involves the following steps:

1. Install a rectangle observer. Basically what this does is get you a line (VHDL signal) from a rectangle observer inside the monitor (there are several) to your VHDL script. Whenever the rectangle observer detects the rectangle it is looking for, it will send the coordinates to you along this signal.

The line

```
signal pattern : integer_array(1 to 4);
```

defines a signal which you can connect to a rectangle observer.

The line

```
pattern' shortcut_in("/mon%match_rectangle/1");
```

connects the signal defined above to the rectangle observer with the ID 1 of the monitor component called mon.

Connecting the shortcut in this way need only be done once, before the first use of the shortcut.

2. Activate the rectangle observer, telling it the pattern it should look for. The line `pattern' activate("0/-1/-1/-1/-1/ ../patterns/000.ppm");` sends an activation command to the rectangle observer connected to the signal `pattern`. The parameters of the activation command are separated with slashes. The number of parameters is fixed and all parameters must always be provided. A filename is always passed as the last parameter and any slashes occurring in the filename lose their special meaning as parameter-separator.
The first parameter is currently ignored.
The next 4 parameters can safely be set to -1 (the pattern matcher will then search the whole virtual screen for the pattern). If you set all 4 parameters to values larger or equal to zero, they will specify a rectangle (in the order top-left corner x-coordinate, top-left corner y-coordinates, width and height). The search will be restricted to the specified rectangle.
The final parameter is the filename of the pattern-file.
3. Wait for the acknowledge from the rectangle observer. The line `wait on pattern;` waits for an acknowledge from the rectangle observer connected to the signal `pattern`.
4. Wait until the rectangle observer has found the pattern. The line `wait on pattern(1) until -1 < pattern(1);` waits until the first element of `pattern` becomes larger than -1. The coordinates of the pattern are passed in `pattern` (in the same order as in the activation command).
5. Optional: Save the coordinates at which the pattern was found in case you need them again later (e.g. to move the mouse to these coordinates). Assuming the pattern matched is a button, the lines `x := pattern(1) + pattern(3) / 2;`
`y := pattern(2) + pattern(4) / 2;` save the coordinates of the (approximate) centre of the button in `x` and `y`.
6. Deactivate the rectangle observer using e.g. `pattern' deactivate;` if the observer is connected to `pattern`.
7. Wait for the rectangle observer to acknowledge deactivation using `wait on pattern until (-2 = pattern(1));`
The -2 in `pattern(1)` signals that the observer is now deactivated.
8. Remove the shortcut, when it is not longer needed.
`pattern' shortcut_in_remove("/mon%match_rectangle/1");`
This can be done once at the very end of the process.

Figure 7.15 puts it all together. Be sure to save any coordinates in variables before you deactivate the pattern, as deactivation sets the coordinates saved in the associated signal to -2!

You can use `assert` statements to output progress information, as shown in lines 18 and 26 of figure 7.15.

7.6 Connecting Shortcuts

Virtual clamps are connected to component signals by addressing a component and then connecting to the internal signal of that component.

The address of a component is the complete path through all instances starting at the root component of your system. The names of the instantiations are concatenated with the slash character (/). In case of `generate` statements the name of that statement is followed by a hash (#) character and the number corresponding to the `generate` statement. Thus, only integer iterators are allowed in `generate` statements.

The last component name is followed by a percent character (%) indicating that the component address is finished at that point. After the percent the signal name to which you want to shortcut has to be concatenated. If that signal needs additional parameters, they are concatenated by using the slash (/) character again.

You can only pass a single filename as parameter, which must always be passed as last parameter. Slashes in filenames may not be quoted.

```
...
architecture structural of node is
    ...
    signal pattern : integer_array(1 to 4);
5    ...
    begin
        ...
        ...
        remote_control : process
10        variable x, y : integer;
        ...
        begin
            ...
            pattern'shortcut_in("/mon%match_rectangle/1");
15        ...
            pattern'activate("0/-1/-1/-1/-1/../../patterns/003.ppm");
            wait on pattern;
            assert false report "now searching for 003.ppm" severity note;

20        wait on pattern(1) until -1 < pattern(1);
            x := pattern(1) + pattern(3) / 2;
            y := pattern(2) + pattern(4) / 2;

            pattern'deactivate;
25        wait on pattern until (-2 = pattern(1));
            assert false report "found 003.ppm" severity note;
            ...
            pattern'shortcut_in_remove("/mon%match_rectangle/1");
        end process remote_control;
30        ...
    end structural;
```

Figure 7.15: Searching for a Pattern

Valid component addresses for the system described in figure 7.5 are, for example, /mon, /eth0, and /hda.

You can connect and disconnect shortcuts using the appropriate shortcut function. The generic format is *signal-name' shortcut-function("component-address%parameters")*; There is an apostrophe between the *signal-name* and the *shortcut-function*.

Some examples are in figures 7.13 and 7.14.

The following shortcut functions are available:

shortcut_in.add This function attaches a shortcut, along which data can flow from the component to faum-expect. Everytime the component sends data, the value of the attached signal will change.

shortcut_in

shortcut_in.remove

shortcut_in.activate

shortcut_in.deactivate

shortcut_out.add

shortcut_out.remove

shortcut_out.print

shortcut_out.random.select

8 Injecting Faults into the Hardware of a Virtual Machine

This chapter tells you how to make the hardware of your virtual machine fail. Section 8.1 gives you an overview over available failures. You can use the graphical user interface to inject faults interactively (section 8.2 or you can include fault injection into your scripts for automated testing (sections 8.3 and 7).

8.1 Available Failures

The following sections show in which ways you can make your virtual hardware fail.

We distinguish between transient and permanent faults. An example for a transient fault is a bitflip, an example for a permanent fault is a stuck-at-one fault.

In a bitflip, a single bit (e.g. in a processor register or in main memory) flips to zero (if it was one, before) or to one (if it was zero, before). That is all, nothing more happens. So if the next thing you do is write a new value to the register or memory location containing the flipped bit, this bit will happily change state to the new value and you won't even notice anything bad has happened. Of course, if the next thing you do is read from this register or memory location, you will get the faulty value and bad things may or may not happen.

Things are different for a permanent fault. Permanent in this case does not mean "forever". It just means, that this fault is not transient and may persist for a certain duration of time. A stuck-at-one fault is such a permanent fault. A bit becomes stuck at the value one and does not change its value any more. Even if the next thing you do is write a new value to the register or memory location containing the flipped bit, and the bit should really be zero after the write, it won't be, because it is stuck at the value one.

8.1.1 Processor Failures

The only failures you can currently induce in the processor are random transient bitflips in processor registers.

You will have to specify the processor (on single processor machines the processor usually has the ID 0), the register and the bit to flip. Since the processor of the virtual machines are all i386 architecture processors, the possible registers are "eax", "ebx", "ecx", "edx", "edi", "esi", "ebp", "esp", "eip", "eflags", "ds", "es", "fs", and "gs".

Legal bit-IDs to flip are 0 to 31 for the long registers and 0 to 15 for the short ones.

8.1.2 Main Memory Failures

The only failures you can currently induce in the main memory are random transient bitflips.

You will have to specify the address of the byte in which you want to flip the bit as well as the bit to flip.

8.1.3 Block Device Failures

The only failures you can currently induce in a block device are permanent failures of random bytes on this device.

You will have to specify the device, the first affected byte and the number of affected bytes (range). Any time the operating systems tries to read or write from the affected parts of this block device, the controller will return an error code.

Of course you can specify several ranges for the same device.

8.1.4 Network Failures

You can make the network interfaces lose packets when sending and receiving. These are permanent faults, i.e. the affected network interface will lose packets as long as this fault is active (turned on).

You will have to specify the network interface, whether it is a send or receive problem and the percentage of packages lost (0% means no packets are lost which is equivalent to no failure).

8.2 Injecting Faults Using the GUI

Fault injection via the graphical user interface lets you inject faults interactively at your leisure using the Inject-menu.

8.2.1 Creating a New Fault Description

To create a new fault description, you will need to specify the necessary information concerning your fault (see section 8.1) in the Fault Description-window. You can open this window using the entry Show List from the Inject-menu and then choosing Add. You can also choose Edit New from the Inject-menu directly.

This window will give you a list of the hardware devices available for fault injection to the left. If you choose a device, a new list or text entry field will appear to let you enter more details.

If you choose a memory device, you will have to enter the address of the affected byte as a hexadecimal number. After you have entered this number and hit the Return-Key, you can choose the number of the bit to flip from a list. After you have chosen a number, an OK-button will appear in the lower left hand corner of the window. Choose OK to add the new fault to the list of currently loaded fault descriptions. Choose Cancel anytime to abort creating a new fault description.

8.2.2 Editing an Existing Fault Description

To edit an existing fault description choose entry Show List from the Inject-menu to display the list of loaded fault descriptions. Then choose a fault description and click Edit. The Fault Description-window filled with the details of the chosen fault description will appear. Choose OK to commit the changes to this fault description. Choose Cancel anytime to abort the changes.

8.2.3 Deleting an Existing Fault Description

To delete an existing fault description choose entry Show List from the Inject-menu to display the list of loaded fault descriptions. Then choose a fault description and click Delete. You will not be asked to confirm the deletion! If the fault is currently active, this will *not* deactivate the fault! Use with care!

8.2.4 Saving Fault Descriptions to a File

You can save the currently loaded list of fault descriptions to a file. Choose entry Show List from the Inject-menu to display the list of loaded fault descriptions. Click Save and a file selection box will pop up to let you select a filename and location for saving. Click OK in the file selection box to save the list of fault descriptions.

8.2.5 Loading Fault Descriptions from a File

If you have previously saved a fault description list (see section 8.2.4, you can load it again. Choose entry **Show List** from the **Inject**-menu to display the list of currently loaded fault descriptions. Click **Load** and a file selection box will pop up to let you select the filename and location of the file to load. Click **OK** in the file selection box to actually load the file. The fault descriptions from the file will be appended to the list of currently loaded fault descriptions. If you do not want this, use **Clear** first, to clear the current list of fault descriptions. If there are currently active faults, this will *not* deactivate these faults! Use with care!

Don't try to load a file which does not contain fault descriptions in the right format! This will currently crash the GUI.

8.2.6 Miscellaneous and Summary

Use the **Show List** to look at the list of faults currently loaded. This list tells you whether the fault is currently active and shows you the fault description.

There are buttons for the following actions along the bottom of the **Show Fault List**-window:

Load Load a previously saved fault description list from a file, appending it to the list of currently loaded fault descriptions.

Save Save a fault description list to a file.

Add Add a fault description to the list.

Edit Edit the selected fault description.

Delete Delete the selected fault description. If the fault is currently active, this will *not* deactivate the fault! Use with care!

Clear Clear the fault description list. This unloads the fault description list. If there are currently active faults, this will *not* deactivate these faults! Use with care!

Close Close the window. This does not unload or clear the fault list and does not deactivate any faults. It simply gets the window out of your way.

You can create or edit a faultload file even when the virtual machine is not turned on. Choose **Inject**→**Show List** to show the list of currently loaded fault descriptions. Choose **Add** to add a new fault description.

8.3 Automated Fault Injection

Please refer to section 7 for general information on how to use Expect for test automation and automatic experiment control. This section only treats fault injection via Expect.

`expect` is an automatic experiment controller, which can wait for certain events to occur on a virtual machine and in return send some input to this virtual machine. The input `expect` sends will usually simulate a user sitting in front of the machine. Events occurring on a virtual machine are such things as screen output (e.g. the login-prompt appearing). `expect` is generating a workload.

Additionally, `expect` may activate/deactivate faults in a virtual machine. `expect` is generating a faultload. `expect` is able to activate all faults defined in `example/faultloads/README`.

Currently, we use it as experiment controller creating a workload in a client/server system, see `example/oracle`. For a short description: `serial_interface` is used to translate the input/output of a serial port of a node into `expect`-events. Using this Babel-fish it is possible to respond to characters on a serial line in `expect`-protocol. Then, `expect` logs in at four virtual machines each over serial port `ttys0` starting application servers on the serial ports (`ttys1`, `ttys2` and `ttys3`). `expect` simulates the user by creating partly random input at the masks of the application servers at all serial interface simultaneously. In detail: there are 4 virtual machines each running 3 application servers at its serial ports. Thus, there are 12 serial interfaces controlled with `expect`.

Additionally `expect` creates a faultload at the virtual machine running an Oracle database server. The fault is selectable via setting a generic from the command line starting `expect`. For this setup `expect` seems to work well.

If you want `expect` to wait for the login-prompt, for example, your virtual machine will have to fulfill the following requirements:

- Configure the virtual machine to have a serial interface (4.7).
- Configure your virtual machine to run a `getty` listening on the serial interface. (Don't use `mgetty` for the moment!) Debian-`getty` has been tested and works, whereas Caldera-`getty` does not work. It should probably work to insert a line such as

```
T0:23:respawn:/sbin/getty -L ttyS0 9600 vt100
```

into `/etc/inittab` on your virtual machine. On Debian-machines, for example, you will just need to uncomment this line.

- Connect a serial-terminal or serial-pattern to the corresponding serial interface of that virtual machine (`system.vhdl`).
- Write a login script in a VHDL-process, sending the login name, after recognition of the login-prompt.

Ready to go. Good Luck!

To inject faults clamp shortcuts to the fault injector signals of the components. E.g. if you want to make the harddisk inoperatable a piece of VHDL source could be:

```
SIGNAL disk_fault : boolean;
.
.
.
hda : idedisk PORT MAP (ide => ide_bus);
.
.
.
inject : PROCESS BEGIN
    disk_fault' shortcut_out_add("hda%byte_fault/0/0");
    disk_fault <= 1=1;
    WAIT;
END PROCESS inject;
```

Here is some rough information on the syntax of the fault descriptions which you can use in the shortcut (printf-like notation): You will have to substitute the VHDL-name of the component instead of "cpu", "eth", "hd", "memory".

```
/cpu/%d/%s/%d
/eth/%d/%s/%d
/hd/%c/%d/%d
/memory/%d/%d
```

Semantics:

```
/cpu/<cpu-id>/<register>/<bit>
<cpu-id>: The ID of the CPU. For monoproductors this is 0, possible
          values for multiproductors depend on the setup.
<register>: Which register. Possible registers are most
            intel-architecture registers, eg. eax, ebx, ecx, esp, eip etc.
<bit>: Which bit to flip. Must be between 1 and 32 (inclusive).
```


/eth/<interface>/<type>/<percent>
 <interface>: Interface number, possible values are usually between 0 and 15. Which interfaces are available depends on the configuration.
 <type>: "receive" or "send"
 <percent>: Percentage of packages to drop.

/fd/<drive>/<start>/<range>
 <drive>: Which drive (usually 0 or 1 are possible). Availability depends on the node configuration.
 <start>: First defect byte.
 <range>: How many defect bytes. All bytes falling within <start> .. <start>+<range> are defect.
 if range is -1, the whole disk is defect

/hd/<drive>/<start>/<range>
 <drive>: Which drive (usually a, b, c etc. are possible). Availability depends on the node configuration.
 <start>: First defect byte.
 <range>: How many defect bytes. All bytes falling within <start> .. <start>+<range> are defect.
 if range is -1, the whole disk is defect

/memory/<address>/<bit>
 <address>: Address of the defect word (must be word aligned).
 <bit>: Which bit to flip. Must be between 1 and 32 (inclusive).

9 Automation Examples

The FAUmachine package includes several automated (almost) ready to run setups. All you need is some (a lot of) disk-space and some installation CDs.

This chapter gives an overview of some of the examples and the general layout of the example directory.

You should be able to start most examples by changing your current working directory into the example-subdirectory (e.g. `install-SuSE-8.2-vesa` and executing
`make experiment`

Most examples will start by creating a virtual machine. This includes creating a virtual harddisk of up to 3Gigabytes. So make sure you have enough disk space!

For some examples you may be able to specify the size of the harddisk. These examples will tell you how to do this at the command shell where you started them.

Most example-directory will contain the following files:

Makefile should include the make-target "experiment", which will run the experiment. Other make-targets are mainly useful for developers.

pointers graphical installations should contain this directory, which contains the PPM-images of the various mouse-pointer shapes.

screens graphical installations should contain this directory, which contains the PPM-images of the various buttons and other parts of the screen necessary to operate the GUI.

simulation.setup contains some basic initialization information for `faum-expect` (see 7.1).

system.vhdl contains the automation script information for `faum-expect` (see various sections in 7).

9.1 CDROM.images

Most examples expect their (virtual) installation CDs to be in the directory `CDROM.images` with a specific name (such as `SuSE-8.1-CD-1` or `RedHat-8.0-CD-2`). Check the `Makefile` in this directory for a list of names. Most examples You can

- create a CD-image of the CD and give it the right name
- create a link with the right name to an existing CD image
- create a link with the right name to your real CDROM-drive (often `/dev/cdrom`). Note: With some CDs this will not work, as FAUmachine needs to know how large the image file/real CD is and some CDs do not provide this information. In this case, create an image file.

9.2 `install-SuSE-*-vesa`, `install-Redhat-*-vesa`

This directories contain information for automated graphical installations for some SusE and RedHat versions.

The default size of the single harddisk of the newly created FAUmachine is 3GB. If you want to change this, execute

faum-expect-gint disksize=#
where # is the size of the disk in Megabyte.

There may be three sets of screenshots for the same installation procedure (all three sets available for the SuSE installations):

- ###.ppm: screenshots for installation with KDE
- ###a.ppm: screenshots for installation without KDE but with X
- ###b.ppm: screenshots for installation without X

This is necessary, because some installations adapt the packages installed to the disk space available, and this in turn influences which screens are displayed during installation.

Otherwise the screenshots are numbered more or less in order of appearance (exceptions may be buttons, which appear in several screens).

Also, is a screen (e.g. Network Configuration) appears in all variations of the installation procedure, the corresponding files will have the same number (e.g. 014.ppm, 014a.ppm, 014b.ppm).

The file `system.vhdl` includes comments, which will help you figure out which part of the installation the screenshots belong to.

9.3 install-Debian*-serial

A minimal Debian installation via the serial interface. Used as the basis for several other examples. If you want an example of how to do pattern-matching on the serial interface, this is it.

There is one big difference between the logic to matching patterns on a serial interface or on a monitor. The stream of data from the serial interface is not random access, it is sequential access only. If you've missed a pattern (because you started looking for it too late), you won't find it again. Whereas the graphical data from a monitor is more like an ever changing file, which you can randomly access to search for you patterns.

Assuming you have the following sequence of actions and patterns during the harddisk-partitioning part of your installation via the serial interface:

1. you typed whatever was the answer to the previous prompt/question and hit the return key
2. you see the prompt "Create a new primary partition"
3. you hit the return key
4. you see the prompt "Size (in MB)"
5. you type 2000 and hit the return key

For patternmatching on the serial interface, your sequence of actions should be the following:

1. Activate the pattern matcher to search for the string "Create a new primary partition"
2. Virtually type whatever was the answer to the previous prompt/question and hit the return key
3. If all is well you should next find the string "Create a new primary partition". Activate the pattern matcher to search for the string "Size (in MB)"
4. Virtually hit the return key
5. If all is well you should next find the string "Size (in MB)". Activate the pattern matcher to search for the next prompt.
6. Virtually type 2000 and hit the return key.

The important thing is to activate the pattern matcher to search for the string which will appear *after* the next input action *before* you execute this action.

9.4 single-node, dual-node, serial

These examples are derivatives of the `install-Debian-3.0r0-serial` example.

The example in `single-node` boots a virtual machine installed in the `install-Debian-3.0r0-serial` example, waits for a while and then turns off the virtual machine. This example uses the COW-mechanism (see 4.4.3), so the harddisk image of `install-Debian-3.0r0-serial` remains untouched when executing this example.

`dual-node` uses `install-Debian-3.0r0-serial` to create two (the default) identical virtual machines connected to the same network. You can use the option `-gint` of `faum-expect` (see optionlist at beginning of chapter 7) to set the variable `number_of_nodes` to a value between 2 and 9 to autogenerate this many virtual machines.

`dual-node` uses the serial terminal to configure a different IP for each virtual machine. One virtual machine then starts a ping to the other virtual machines. Shutdown after pinging for a while.

10 Building FAUmachine from the Source Distribution

10.1 Installing the FAUmachine Source Distribution

The following information concerns the FAUmachine source distribution. No installation procedures are necessary for the FAUmachine binary snapshot. It is usable out-of-the-box. For usage information, please refer to the appropriate chapters in this user guide.

10.1.1 Extracting the Sources

The following explanation assumes that the current working directory is `/proj`. To extract the source type `tar -xzf FAUmachine-version.src.tar.gz` at the command prompt. This will create a directory `FAUmachine-version-src` in the current working directory with a number of subdirectories.

10.1.2 Contents of FAUmachine/ directory

The important subdirectories in the FAUmachine/ directory are

docs some of the papers on FAUmachine we have published so far as well as this user's guide `guide.pdf`.

launcher the graphical frontend, including the wizard and configuration dialogs

node-pc the actual simulator

node-serial_terminal the serial terminal which is able to connect to a virtual machine.

expect the automatic experiment controller Expect

experiments contains a few example, including the necessary setup-files for generating appropriately configured nodes automatically. Things change quickly here.

bridge-net interconnection between real and virtual net

scripts collection of Perl and shell scripts we are using

faum-* modified versions of bootloaders and kernels. These directories include all changed source files together with some ready-to-use precompiled binaries for the most important distributions.

10.2 Building FAUmachine

There are two things you can build from source: FAUmachine itself or bootloaders/kernels modified for FAUmachine.

To build FAUmachine, simply call the standard sequence `./configure && make && make install`. Everything will be installed into the directory `/usr/local`. If you want to install to another directory, pass `--prefix=dir` to `./configure` (remember to get root access prior to make `install` if necessary).

Note: in order to build, you need the `xfree86` and `openmotif` development files.

Building modified versions of bootloaders or kernels is a bit more complicated and is discussed in the next sections.

10.3 Building bootloaders and kernels

Because of space limitations, we don't ship the full sources of the modified boot loaders and kernels. If you want to build them yourself, you need to obtain their original sources first.

10.3.1 Downloading and Installing Original Sources

Downloading the original kernel and bootloader sources is done with scripts called `unpack-*` in the `scripts` subdirectory. These download and unpack all required sources (edit the script if you want to change the mirror used). All sources are unpacked into the current working directory.

There is another script called `los` (link-original-sources), which creates symbolic links to the location of your unpacked sources. By default, it will look for your sources in `/usr/src`. If you unpacked them to another directory, set the environment variable `SRC` before running the script, e.g. by executing `SRC= /src scripts/los` (when using `bash` as shell).

Be careful, this scripts downloads *all* sources, requiring a *lot* of free disk space. If you only want to build only one specific kernel, just download, unpack and link it yourself.

10.3.2 Building

Several make targets in the top-level Makefile can be used to build the bootloaders and kernels. A normal `make` run will build FAUmachine together with some default kernels and bootloaders (well, it won't actually build them as they are already shipped with the source package).

To build all modified linux versions, execute `make faum-linux-allall`. Building bootloaders is similar: just replace 'linux' with the bootloader that should be build: `lilo`, `syslinux`, `gfxboot` or `grub`. To build only one special version, set the environment variable `KERNEL_VERSION`, `LILO_VERSION`, `SYSINUX_VERSION`, `GFXBOOT_VERSION` or `GRUB_VERSION` and execute `make faum-**-all`.

Note: It may not be possible to build all kernels with the same version of your compiler. Older kernels often require older compilers, newer kernels require newer compilers. For that reason we build kernels for some distributions using the correct compiler for that distribution.

10.3.3 Installation

FAUmachine uses the file `/usr/lib/faumachine/loaders/Config` to find the correct bootloaders and kernels for the simulated system.

If the BIOS prints a message that it cannot find a replacement bootloader or kernel, you have to add a line to this config-file to point FAUmachine to the correct image for replacement. Every line in that file contains a search string (the one printed by the BIOS) and a filename (the image to load).

11 When Things Don't Work

If things don't work, you can send us a bugreport. Some known caveats and problems are listed in section 11.2.

11.1 Creating Bugreports

If things do not work, please let us know about it, so that we have a chance to fix it! If you feel confident that you can solve it, please go ahead, and don't forget to include your fix in the bugreport!

In any case, please create a bugreport (simple text file) and mail it to **bugs@faumachine.org**. Please try to use the bugreport-script `faum-bugreport` provided. It will ask you about the things we need to know to have a chance to reproduce the bug, find the cause and fix it.

A word of caution is appropriate here: `faum-bugreport` is designed for internal use with `cvs` and the full FAUmachine-sources, so it might splutter some warnings, when used externally. These should not worry you!

To create a bugreport use `faum-bugreport`, which will be installed along with FAUmachine.

To create a new bug do `faum-bugreport--new --offline`. It will ask you a few questions and then create a directory `BUGREPORTS` in which it will put the bug-file. The name of the bug-file is constructed out of the current time and your username. The file is ASCII and you can edit it if you like (try not to destroy the formatting, though). Send this file to **bugs@faumachine.org**.

If `faum-bugreport` isn't for you feel free to inform us about the bug anyway at **bugs@faumachine.org**!

If you've downloaded the source-distribution, please use the appropriate make targets to create bugreports.

11.2 Caveats

Finally a few caveats and known bugs.

- FAUmachine has not been tested/developed with internationalization in mind. It has worked fine so far with "US English" locales, keyboard layouts etc.
- Syntax errors in configuration files are not tolerated. There may not even be error messages, processes may simply terminate. If something doesn't work, try using the `ConfigWizard` (see section 3) to generate the config-files. That's what the `ConfigWizard` is there for!
- If you find things, which you think should work on a virtual Linux machine (because they work on a real Linux machine), send in a bug report and we'll see what we can do.
- If you don't know much about Linux, there are a few good books around. Please do not ask us any general Linux questions.

To give an example, we consider questions like "How do I install an X-Server?", "What are boot parameters?" or "How should I partition my harddisk?" to be general Linux questions. A starting point for finding answers to general Linux questions is **<http://www.linux.org>**.

