Implementing multiple network cards in Bochs

Timo Teifel, ttri@teifel-net.de, University of Tübingen, Chair for Computer Networks and Internet, February 2004 (Bochs 2.1). April 2005: Updated for Bochs Version 2.2pre3.

1 Introduction

Bochs is an open source PC emulator (http://bochs.sourceforge.net). The original version comes with only one network interface card and it's not possible to use more network connections in the emulated computer.

As part of a project in the workgroup I work at the University of Tübingen, we required more than one network interface card. I modified Bochs so that it supports up to 4 distinct network interface cards. The patch is available from www.teifel.net/projects.

2 Configuration of Bochs and the System inside

To enable the network interface cards, you have to run the `./configure` script with `--enable-ne2k` to switch on network-interface card (NIC) support. After that, you can build Bochs with the make command.

At the moment, no GUI (if I write "GUI" I also mean the Bochs textconfig interface) can set the configuration of the network cards 1-3. Only the card number 0 can be set. So, open your .bochsrc config file in a text editor and add lines like this:

```
ne2k_0: ioaddr=0x280, irq=9, mac=fe:fd:00:00:00:03, ethmod=linux, ethdev=eth0
ne2k_1: ioaddr=0x380, irq=10, mac=fe:fd:00:00:00:01, ethmod=tuntap,
 ethdev=/dev/net/tun, script=tuntapnetwork.sh
```

The ne2k is equivalent to ne2k_0 for backward compatibility. You should prefer the _0 notation though.

If you're running Linux in the Bochs (I'm using Knoppix 3.7 at the moment), you may have to specify aliases for your network cards in the /etc/modules.conf file like this:

```
alias eth0 ne
alias eth1 ne
options ne io=0x280,0x380 irq=9,10
```

You should then load the module with:

```
modprobe eth0 eth1
```

Or just use a command like this:

```
modprobe ne io=0x280,0x380 irq=9,10
```

After that you can use both ethernet devices as usual.

3 Implementation

The changes made to the sourcecode are explained by source files.

3.1 Automatic configuration of the sourcecode

With multiple NICs, it's not possible to use static member functions. These have to switched off in `config.h.in`, which is used to create `config.h` by the `./configure` script. The "1" has to be changed to a "0" so that the `BX_USE_NE2K_SMF` line looks like this:

```
#define BX_USE_NE2K_SMF 0
```

After this, `./configure --enable-ne2000 && make` works fine.

3.2 bx_options

All options for the virtual hardware are listed in a `struct` called `bx_options`. It's defined in `bochs.h`:

```
typedef struct BOCHSAPI {...} bx_options_t;
```

That typedef contains

```
  bx_ne2k_options   ne2k;
```

which has to become an array of 4 devices, 0..3:

```
  bx_ne2k_options   ne2k[4];
```

3.3 Hardware config class

In `config.cc`, the hardware configuration options are read from the config file and initialized.
`bx_init_options` has to init all 4 NICs.
`bx_reset_options` resets all 4 NICs.
`parse_line_formatted` reads one line of the configure file (.bochsrc) and extracts the configuration for the devices. I modified this function, so that it can read the configuration for the 4 different NICs. The number is extracted from the input file and is used to access the correct card in the array.
`bx_write_ne2k_options` writes the configuration which the user changed in the GUI back to the bochsrc file. Accepts a new parameter: `cardNumber`.
`bx_write_configuration` writes information of all 4 NICs.

3.4 Simulation interface options

Every value which the user can change has to have a unique ID, set in an enumeration `bx_id` in `gui/siminterface.h`. The 8 values for the NIC card have to be copied for each card, and, since an enumeration cannot contain arrays, be renamed. The IRQ for the NIC with number 0 becomes:

```
BXP_NE2K_0_IRQ,
```

and so on. Every value has the number of the card in its name.

3.5 User interfaces

The user interfaces like the textconfig GUI or wxGUI allow to enter the configuration values for the NIC card. I didn't implement the possibility to specify the values for every card. Only card number 0 can be changed.

For configuration of the other cards, the user has to edit the .bochsrc file manually.

3.6 The four devices

In `iodev/devices.cc`, the four devices are created.

The constructor of `bx_devices_c` assigns a stub class of a NIC to the `pluginNE2kDevice[x]` for all 4 network cards.

In `bx_devices_c::init` the plugin for every device that is present is loaded. This has to happen for every network card. The `PLUG_load_plugin` macro resolves to `libne2k*_LTX_plugin_init`. There are functions with number 0 to 3 at the place of the `*`, for each NIC number. Unfortunately, the compiler couldn't find these functions, even though I placed them at the same position where the original Bochs had the corresponding function for only one card. I therefore added a forward

declaration of these functions to solve the problem.

These four functions are defined in `iodev/ne2k.cc`.

3.7 bx_devices

The `class bx_devices_c` contains pointers to all devices and is defined in `iodev/iodev.h`. The pointer to the NIC device is now an array of pointers:

```
bx_ne2k_stub_c   *pluginNE2kDevice[4];
```

3.8 Ne2K

The class `bx_ne2k_c` in `iodev/ne2k.[hc]*` represents the NICs, and to access its options in the `bx_options.ne2k[]` array, it has to know its NIC number. Therefore I added a public int-value `interfaceNo` in the `bx_ne2k_c` class, in ne2k.h.

The devices get created and constructed in the `libne2k_LTX_plugin_init(...)` function, which sets the `interfaceNo` value after constructing the object. The function also registers the newly created object with `BX_REGISTER_DEVICE_DEVMODEL`with a pluginname that is unique for every NIC.

The original Bochs version of this file created one ne2k object and named it `theNE2kDevice`, which was in global scope of ne2k.cc.

In my case, with several such objects, this obviously didn't work any longer. In the non-static case, where several NICs are possible, theNE2kDevice is a Preprocessor-Macro that just gets "`this`". So it should work with the static-case with only one NIC, too.

The function `bx_ne2k_c::rx_frame`, which is called when an ethernet frame has been received, uses a `static unsigned char bcast_addr[6]`... it's probably not necessary to remove the 'static' here.

`bx_ne2k_c::init()` reads in the values from the configuration file. Therefore, it reads `bx_options.ne2k[interfaceNo]` with the interface-Number given to it by the `libne2k_LTX_plugin_init` function.

The `init` function registers the objects read- and write-handlers with `DEV_register_*_handler`. It passes along the corresponding handler and a this-pointer which is necessary for the static handler to know which object to send the read/write request to. It also sends a `char* pluginName` which is unique for every card.

After that, the `init` function calls `eth_locator_c::create` with the user-selected module name (`linux`, `tuntap`, `null`, etc.). This creates an `eth_pkt_mover` object, that is saved in the `ethdev` pointer of the `bx_ne2k_c` Object. `ethdev` is used to send packets to the network in the `bx_ne2k_c::write_cr` function.

3.9 LOG_THIS

Many of the `iodev/eth_*.cc` files use a preprocessor macro called `LOG_THIS`. It's only used for Log entries, warning or error messages in the macros `BX_INFO`, `BX_ERROR` and so on.

```
#define LOG_THIS bx_devices.pluginNE2kDevice->
```

This does work no longer because there are several NE2K devices. At the definition of the macro, it is unknown to which device the code belongs. Since it's not crucial for the simulation that it points to the correct device and it's only used for messages to the user, the macro is now defined to always point to the device 0.