HelenOS networking

Table of Contents

1. Introduction

This manual describes the support for writing network interface controllers (aka NIC's) in HelenOS operating system. It is intended mainly for developers who want to write NIC drivers or software tightly cooperating with lower layers of network stack (such as firewalls or the networking stack itself). This manual also contains the description of DMA support developed by part of NIC development support.

1.1. Goals and achievements of the project

HelenOS [1] is microkernel operating system which means the kernel itself tends to be as small as possible and the tasks including device drivers should run in userspace. The networking support is the important part of the modern operating system. By the time when this project started HelenOS already featured partly functional networking stack with basic TCP, UDP, ARP and IP protocols support as well as a simple Novell NE2000 NIC driver ported from the Minix operating system. However this driver operated only through basic port I/O and lacked any higher functionality.

The completion of this project provided a unified way how to easily write NIC drivers with DMA support. We have proved the concept by developing a driver for Realtek RTL8139 network card with full exploitation of its abilities, including hardware packet filtering, autonegotiation support and another features. We have also developed a fully operational driver for Intel E1000 and ported existing Novell NE2000 driver to the NIC framework.

The new DMA interface allows privileged processes to allocate memory from various physical ranges or query information about already allocated parts of memory.

All significant operating systems like Linux, BSD family, MINIX or GNU Hurd contains the DMA memory support, functions for making development of the network controller developement and several network interface controllers drivers, the HelenOS obtains this abilities by this project.

We have completely meet the goals of project's submission including the optional parts.

[1] http://helenos.org

2. HelenOS Architecture Overview

This section provides an overview about NIC and DMA related architecture existing in HelenOS in the time of the project start

2.1. IPC

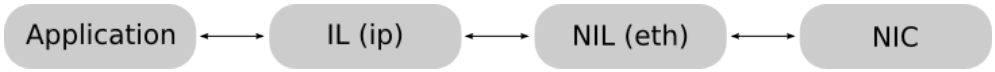
In HelenOS microkernel operating system drivers are userspace processes, which are divided from kernel. It means, when a driver of a device fails, it can be replaced by a new running instance without system restart. But because this division of drivers from kernel, there must be a method, how to pass data between two processes.

In HelenOS the way, how to pass data between processes, is passing messages. For this passing is used IPC framework. It is asynchronous, it means, process which passes a message does not wait until it is delivered and can continue it’s work. Of course there is a way, how to wait for answer to sent request.

2.2. HelenOS networking architecture

Unlike most monolithic operating systems the network stack in HelenOS is not a part of the kernel. As a matter of fact it is distributed between several processes, one for each layer and protocol - these are called **modules** in the network stack architecture. Originally there was an option to bundle some of the modules into processes together for the sake of higher performance but this option was already abandoned, although some parts of code can suggest its previous existence.

Figure 1. Modules in sent/received packet processing in HelenOS



It is worth mentioning that both the overall design of the network stack and the actual implementation are not meeting HelenOS quality standards and are subject to change in near future but the interconnection between networking stack and NIC framework should prevail.

Aside from apparent modules for network protocols (as TCP or UDP) there is a central service `/srv/net` called **NET** further in this documentation. This service starts protocol’s modules and reads both common and NIC device’s configuration from the `/cfg/net/` directory (in sources this is found under `/uspace/srv/net/cfg/`). This service manages the configuration and distributes it between modules which query for it. The assignment of the configuration file to the interface is based on the devices hardware path.

The packet is represented by `packet_t` structure. Its instances are allocated by the **NET** server and distributed by memory sharing through all the system. The packets are identified by whole system unique identificator system and there is native support for storing packets in packet queues linked by its idetificators. This attitude allows to pass more than one packet in time but it also leads to the system inconsistency danger.

More detailed description of the HelenOS networking can be found in the master thesis of Lukas Mejdrech [2].

[2] Lukas Mejdrech: Networking and TCP/IP stack for HelenOS system; 2009; http://www.helenos.org/doc/theses/lm-thesis.pdf

2.3. Device Driver Framework (DDF)

In HelenOS neither the drivers are running in kernel mode but these take the form of common userspace processes, communicating with other tasks through IPC. Kernel intervention is required only for certain privileged operations such as interrupt handling, port I/O enabling or physical memory allocation.

As a matter of fact, there are still some drivers in the kernel because moving them into userspace would be impractical but these are only few and these will be not discussed here. NIC drivers, which are of our interest, are completely userspace processes, although trusted ones (these possess some higher privileges than common tasks).

The DDF architecture is deeply described in the master thesis written by Lenka Trochtova [3], the overview of parts related to project follows.

Services and driver starting

Device Driver Framework consists of the central Device Manager service (aka **DevMan**) and a lot of drivers located in the `/drv/` directory. When DevMan boots it probes which drivers are installed in the system and which devices these drivers support. This is done through `.ma` files associated with each driver identifying the supported devices. Each line in the `.ma` file contains identificator priority (how good is the driver for such device) and hardware identificator (match id). The match id for the PCI device contains its vendor id and device id, the example of configuration line for PCI device with the vendor id 10ec and device id 8139 is in Example 1, “Configuration line for PCI device”.

Example 1. Configuration line for PCI device

```
10 pci/ven=10ec&dev=8139
```

After getting the informations from the `.ma` files the **DevMan** starts the root hardware driver and virtual device driver. Bus drivers report child devices to the **DevMan** which chooses the best suitable driver based on given match id. If the driver is not running the **DevMan** starts it and invokes `add_device()` callback on the driver side by IPC message. Then the driver initializes the device and informs **DevMan** about success/unsuccess by the callback return value.

Communication with the device

Starting the drivers is only a first part of DDF functionality. The second one is to provide a uniform way how to communicate with the devices.

Each device is identified by a URI-like path, called hardware (HW) path or DDF path later in this manual. Task which wants to communicate with the device’s driver sends this path to **DevMan** and this responds with a phone to the driver already associated with the particular device.

Example 2. Hardware path of the PCI device

```
/hw/pci0/00:03.0/port0
```

[3] Lenka Trochtova: Device drivers interface in HelenOS system; 2010; http://www.helenos.org/doc/theses/lt-thesis.pdf

Communication on the phone should follow the DDF interface supported by the device. This is a simple set of RPC methods that can be called on the device. Unfortunately the RPC stubs and skeletons (code packing and unpacking method request must be written manually. However it is not a responsibility of driver's author to write this code, it is already provided in libraries (**`libc`** on client side and **`libdrv`** on driver side).

Interrupt handling

The interrupts are mostly processed in the userspace. In the case of level triggered devices there is a necessity of handling the interrupt before leaving the kernel - the device must clear the interrupt before kernel enables the interrupts again. The interrupt handling is divided to two parts - kernel part defined by pseudocode in `irq_code_t` structure instance and userspace handler, both passed to DDF and kernel by `register_interrupt_handler()` function. The driver must include `<ddf/interrupts.h>` from **`libdrv`** to include interrupt handling part of DDF.

2.4. Memory management

Kernel memory management consists of two basic layers - frame allocator and virtual memory manager.

The HelenOS uses the buddy allocator for frame allocation, the allocation is possible by the power of two chunks of frames only.

The virtual address space is divided into continuous address space areas represented by `as_area_t` structure. The different areas needs different management thus memory backend is assigned to each area. The backend is responsible for the page faults handling, proper sharing, resizing or destroying whole area. The example of such backend is anonymous space area backend (`anon_backend`) implemented in the `kernel/generic/src/backend_anon.c` handling common memory without specific requirements for physical addresses.

The memory sharing is invoked by the IPC communication between the participated processes, the kernel waits until the sharing is confirmed and crates new memory area managed by the same backend as the area in the original process. The backend is notified about area sharing during the first sharing request and it should initialize structure containing the memory sharing information.

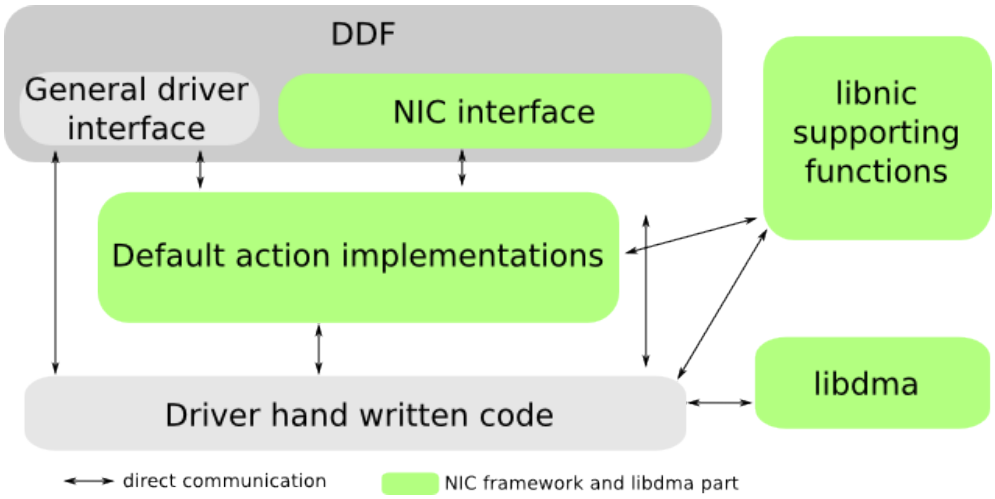
3. NIC Framework architecture

NIC Framework evolved from the former network stack architecture exploiting the merits of the Device Driver Framework. Its first objective was to design and implement the DDF interface for network interface controllers and setup the existing drivers for DDF. Another objective was to identify the code common for most drivers and extract it to the library - for example the driver does not need to implement all setters/getters for various settings, the driver implements only the change event handler. The general conception was to keep the driver minimalistic, only controlling the hardware and to concentrate all added-value features into the library.

3.1. Framework overview

The framework consists of the NIC interface added to the Device Driver Framewor and **`libnic`** library which contains default handlers for the most of NIC and DDF interface parts and supporting functions for the NIC driver development. The structure of driver implemented by using NIC framework is shown in Figure 2, “Driver structure overview”.

Figure 2. Driver structure overview



One goal of the framework was the reduction of hand written driver code thus the default implementations were added to all possible interface functions. The NIC interface will be described in Section 3.2, “NIC interface in DDF” and Section 3.3, “NIC Interface methods” , the libnic will be described in Section 3.4, “NIC driver structure and libnic”.

The DMA support library and other tools will be described in Section 4, “DMA interface”.

3.2. NIC interface in DDF

Here is a brief example how is application request - procedure call - translated into IPC and back into procedure call on the driver side. Requests which send and receive multiple arguments and blocks of data are more complicated, of course. The overall functionality is a part of the Device Driver Framework but the actual implementation of the RPC stubs was required during the NIC Framework. Detailed description of concrete methods in the interface will follow in Section 3.3, “NIC Interface methods”, the table of default handlers and its requirements is in the Appendix B, *NICF Default handlers summary*.

Application side

All IPC communication encapsulation was added to **libc** to the `<device/nic.h>`. The function marshalls arguments into an IPC call together with the interface (`DEV_IFACE_ID(NIC_DEV_IFACE)`) and method identifiers (`NIC_SEND_MESSAGE` in the example below). Then it verifies that the return code is correct (otherwise it returns `EPARTY`) and unmarshalls the output arguments.

The example method below has no output arguments, therefore nothing needs to be unmarshalled back. Many methods also send or receive blocks of data - in these cases the stub is rather more complicated.

Example 3. Application-side function implementation

```
int nic_send_message(int dev_phone, packet_id_t packet_id)
{
    int rc = async_req_2_0(dev_phone, DEV_IFACE_ID(NIC_DEV_IFACE),
        NIC_SEND_MESSAGE, packet_id);
    if (rc != EOK && rc != EINVAL && rc != EBUSY) {
        return EPARTY;
    }
    return rc;
}
```

Driver Side

The driver side of interface contains two tightly related structures: **remote interface** and **nic interface**. The **remote interface** is hardcoded in the **libdrv** while the **nic interface** is provided by driver, although usually partially filled with default methods provided from the NICF.

When a IPC request comes, the service routine is picked from the **remote interface**. This routine is responsible for IPC communication, decoding parameters and obtaining all request-related data and calling the proper callback from nic interface. The **remote interface** functions will be referred as **remote functions** in further text.

Both interfaces were added to **libdrv** to the `uspace/lib/drv` directory. **Remote interface** is defined in `uspace/lib/drv/generic/remote_nic.c` and it consists of `remote_nic_iface` instance of `remote_instance_t` structure containing array of callbacks to remote wrappers and the implementation of these wrappers.

Example 4. Remote function implementation

```
static void remote_nic_send_message(ddf_fun_t *dev, void *iface,
                ipc_callid_t callid, ipc_call_t *call)
{
    /* Get the interface of callbacks */
    nic_iface_t *nic_iface = (nic_iface_t *) iface;
    /* Decode callback argument from the IPC message */
    packet_id_t packet_id = (packet_id_t) IPC_GET_ARG2(*call);
    /* Call the proper high-level callback */
    int rc = nic_iface->send_message(dev, packet_id);
    /* Return the result */
    async_answer_0(callid, rc);
}
```

The **nic interface** is defined in structure `nic_iface_t` declared in `<ops/nic.h>` in **`libdrv`**. The interface consists from some mandatory and optional functions - the mandatory callbacks (as `send_message()`) must be implemented by driver. In lots of cases the default implementation of the callback can be used to decrease amount of hand written code. The functionality descriptions as the default implementation will be described in following sections.

3.3. NIC Interface methods

The NIC interface is designed to allow to use the most of features the network controller offers.

Initialization

The device itself does not know which NIL (network interface layer) module will be used for the device. The `connect_to_nil()` callback is responsible for connecting driver to the NIL service given as parameter.

Device states

The device can be in one of following state: **active** (`NIC_STATE_ACTIVE`), **down** (`NIC_STATE_DOWN`) and **stopped** (`NIC_STATE_STOPPED`). The states diagram with possible transitions is in the Figure 3, “NIC states diagram”.

Figure 3. NIC states diagram

In the **stopped** and **down** states the device does not transmit nor receives packets. The difference between these states is that the device sets all properties to its defaults when entering to **stopped** state and keeps the settings when enters **down** state.

The interface part for the state changing is the `get_state()` and `set_state()` callbacks, both mandatory.

Packet sending and receiving

The packet sending and reception is allowed only in **active** state.

The mandatory `send_message()` callback must be implemented for sending packets to the network.

When a packet comes from network it is reported to NIL layer through an IPC message - this goes out of scope of the NIC interface provided by DDF. Packet reception is expected to be done either upon an interrupt or through device polling. Several device polling modes can be set by optional interface part - `poll_set_mode()`, `poll_get_mode()` and `poll_now()`. The available modes with description are described in Table 1, “Available polling modes”. The device polling should do all the stuff as the interrupt would do, no interrupts are expected in **on demand** and **software periodic** modes.

Table 1. Available polling modes

Mode	Enumeration value	Description
immediate	`NIC_POLL_IMMEDIATE`	The device polling is invoked by interrupt immediately after packet reception
on demand	`NIC_POLL_ON_DEMAND`	The device polling is invoked explicitly by `poll_now()` callback
periodic	`NIC_POLL_PERIODIC`	The device polling is invoked in specific periodic intervals
software periodic	`NIC_POLL_SOFTWARE_PERIODIC`	The same as **periodic** but the period is invoked by software

MAC address

The NIC address can be changed by optional `set_address()` callback, the `get_address()` callback for obtaining the MAC address is mandatory.

Filtering

The driver can set what kind of packets are accepted. This functionality is optional, the promiscuity mode (accept whatever comes) is supposed in the case the driver does not support this interface parts. The filtering of unicast packets should be set by `unicast_set_mode()` and obtained by `unicast_get_mode()`. The possible modes are in Table 2, “Unicast filter modes”

Table 2. Unicast filter modes

Mode	Enumeration value	Received unicast packets target
blocked	`NIC_UNICAST_BLOCKED`	None accepted
default	`NIC_UNICAST_DEFAULT`	sent to devices physical address
list	`NIC_UNICAST_LIST`	devices physical address or the address from the list
promisc	`NIC_UNICAST_PROMISC`	all unicast packets

The multicast and broadcast filters can be set by the same way as the unicast - by `multicast_set_mode()` and `multicast_get_mode()`, resp. `broadcast_set_mode()` and `broadcast_get_mode()`. The multicast modes are in Table 3, “Multicast filter mode”, the broadcast mode can be either accepting (`NIC_BROADCAST_ACCEPTED`) or blocking (`NIC_BROADCAST_BLOCKED`).

Table 3. Multicast filter mode

Mode	Enumeration value	Received multicast packets target
blocked	`NIC_MULTICAST_BLOCKED`	None accepted
list	`NIC_MULTICAST_LIST`	one of multicast address in the list
promisc	`NIC_MULTICAST_PROMISC`	all multicast packets

The possibility to block packets from some sources can be enabled by implementing `blocked_sources_set()` and `blocked_sources_set()` - no packets with the source address on the

list should be accepted. Some defective packet reception (bad CRC or runt packets (shorter than 60B)) can be set and detected in `defective_set_mode()` and `defective_get_mode()` callbacks. If not implemented, no such packets should be received. The possible defective packet types are set as bits defined by `NIC_DEFECTIVE_` macros in `<net/device.h>` header in **`libc`**.

Operation mode, flow control and autonegotiation

The NIC controllers can set its speed, duplexity (full/half duplex) and role (master or slave, gigabit ethernet only), which can be either set by hand or autonegotiated. The autonegotiation is the preferred way and the driver should enable it by default if supports, but this is only recommendation. Implementation of this functionality is only optional.

The callback for manual setting is `set_operation_mode()`, the mode can be obtained by `get_operation_mode()`. The autonegotiation can be enabled by `autoneg_enable()` callback, disabled by `autoneg_disable()`, new autonegotiation is forced by `autoneg_restart()` and the current autonegotiation setting can be obtained from `autoneg_probe()`. If the autonegotiation is enabled, the manual setting is discarded and replaced by the autonegotiated, if the manual setting is forced, autonegotiation should be disabled automatically.

The autonegotiation advertisement is passed by bitmask of `ETH_AUTONEG_` macros defined in `<net/eth_phys.h>` in **`libc`**. For example `ETH_AUTONEG_10_BASE_T_HALF | ETH_AUTONEG_10_BASE_T_FULL` is used for forcing autonegotiation of 10MBit half and full duplex modes. The zero advertisement means "all supported by driver". The driver should return error value if some unsupported mode is requested rather than silently enable unsupported mode.

The flow control can be also set manually or be autonegotiated. In some autonegotiated modes the controller can allow setting flow control modes. The flow control setting should not disable autonegotiation, error code should be returned instead. The flow control setting can be taken by `get_pause()` callback, it can be set by `set_pause()` callback. When forced time of pause packet is not supported by the controller (but the pause packet transmition can be enabled), the nearest possible time supported should be set. The time value 0 lets driver choose the best suitable time.

Device statistics and information

The device should keep and updated statistics in the instance of `nic_device_stats_t` structure. The statistics can be requested by `get_stats()` callback.

Another useful informations, like supported ethernet physical layers or supported autonegotiation modes, can be requested by `get_device_info()` callback. The actual status of cable connection should be obtained by `get_cable_state()` callback.

Implementation of all these callbacks is optional.

Offload computing

Computation of IP, TCP and UDP checksums requires massive CPU sources. Some NIC’s are trying to ease the CPU and verify the checksum automatically, displaying the result in a single bit and filling the checksums into transmitted packets.

Ability to do this can be probed through `offload_probe()` callback and requested mode set through `offload_set()`. Then the driver can fill in the verification result into the offload info through `packet_set_offload()` and higher layers may consider the bits.

VLAN support

Some cards have the possibility to automatically tag and untag frames by a 12-bit identifier, called VLAN tag. This behaviour can be controlled through `vlan_set_tag()` callback. The presence of desired tag (maybe stripped) should be indicated in the offload field of the packet.

Similarly, the packets can be just filtered according to the tag. Because the 12-bit identifier space is pretty narrow, the filtering can be perfectly specified through a 512-byte mask. See callbacks `vlan_set_mask()` and `vlan_get_mask()`.

Wake-on-LAN

NIC interface considers also NICs which offer the possibility to wake the computer up. There are multiple types of frames that can cause the wake up - these are called WOL virtues. Combinations of virtues are sometimes complicated. The currently allowed combination of virtues can be determined up by calling the `wol_virtue_get_caps()`. New virtues are added or removed through `wol_virtue_add()` and `wol_virtue_remove()` callbacks, currently active virtues are listed through `wol_virtue_list()` and their type and parameters through `wol_virtue_probe()`. When the computer boots after a wakeup event, some information about the frame that has woken the computer can be loaded through `wol_load_info()`.

3.4. NIC driver structure and libnic

The **`libnic`** library was developed for the NIC controller drivers development. It contains some helper stuff and default implementation of the NIC interface functionalities. Its source root directory is `uspace/lib/nic`. The header file the driver should include is `<nic.h>` where all functions supposed to be used by driver are defined. The rest of headers is considered library-internal and compilation fails upon including them in the driver.

The main structure in the **`libnic`** is `nic_t` defined in `<nic_driver.h>`. This structure contains data needed by general NIC driver like MAC address currently assigned, connections to the DDF structures, phones to other parts of networking (NET server, NIL layer). The structure is not directly accessible by the driver for safety reasons, framework functions must be used for the structure manipulation, more information about the `nic_t` access can be found in the section called “Accessing the nic_t structure”.

One important part of the **`libnic`** library are the default handlers for NIC and DDF interface requests. The handlers provides the preprocessing and postprocessing of the request data and the hand written code is needed only for the parts where hardware cooperation is expected, some of the default handlers are able to provide whole request.

The description of default handlers is placed in the feature support description. The framework convention is that the default handler of "`callback()`" is implemented by "`nic_callback_impl()`" in file `uspace/lib/nic/generic/nic_impl.c`. The brief summary of the default handlers requirements can be found in Appendix B, *NICF Default handlers summary*.

Driver Initialization Support

The **`libnic`** contains function `nic_driver_init()` which should be called when the driver starts - it initializes internal **`libnic`** structures like internal logging and initializes packet manager. Next

the driver should initialize all DDF interfaces and call `nic_driver_implement()` function which replaces unimplemented functions by default implementations if possible.

Device Initialization Support

Device initialization should be done in `add_device()` callback. The `nic_t` instance should be created by `nic_create_and_bind()` function, which also initializes it and connects to the DDF device structures, From this moment the conversion between `nic_t`, `ddf_fun_t` and `ddf_dev_t` can be done by `nic_get_from_ddf_dev()`, `nic_get_from_ddf_fun()`, `nic_get_ddf_fun()` and `nic_get_ddf_dev()`.

The device can set pointer to its private data structure by `nic_set_specific()`, the pointer can be obtained by `nic_get_specific()` method. The `nic_t` instance is the owner of the assigned private data memory, the memory will be deallocated together with `nic_t` instance. The `nic_unbind_and_destroy()` removes `nic_t` structure from the DDF infrastructure and deallocates the memory.

The device should report its default MAC address by `nic_report_address()` to the framework to allow proper checking of request validity, this should also be done together with all MAC address changes.

As the final step the device should connect itself to the NET server and APIC controller by calling `nic_connect_to_services()` and register itself into DDF structures by `nic_register_as_ddf_fun()`.

In the following `device_added()` callback the device should call the `nic_ready()` to notify NET service that the driver is prepared to work. The default implementation for `device_added()` callback provide this notification.

Device state handling

The state handling is provided by default implementations of `set_state()` and `get_state()` handlers.

The `get_state()` default handler just returns the current device state stored in `nic_t` structure. The `set_state()` handler notifies the driver about the change by calling `on_activated()`, `on_stopped()` or `on_going_down()` callback

When moving to the **stopped** state the framework resets all the settings to its defaults and calls the proper callbacks to propagate the settings to the driver. In the `on_stopped()` state the driver should reset the controller. When moving to the **down** state the driver can turn the device off but the controller settings must be restored when activating again.

Packet transmition support

The packets to send are given to the framework by `send_message()` callback by ID of the first packet in the queue to send. The default implementation of this callback checks the device state, then goes through the queue, checks each packet validity, obtains its data from the NET server in `packet_t` structure, and goes through all packets in the queue to send and calls `write_packet()` callback

assigned to the device. The implementation of `write_packet()` callback is responsible for sending the checked packet to the network through hardware.

The `write_packet()` callback must be assigned to the device by `nic_set_write_packet_handler()` function during the device initialization.

If the driver is not able to send more packets because of full transmition buffers, it can set the "Transmitter busy" sign by `nic_set_tx_busy()` function, the default implementation will discards packets until the busy sign is cleared (set to 0).

After the packet transmition is finished, the `nic_release_packet()` should be call to release the packet from the system. It is not done by default implementation because sometimes the packet cannot be released until the interrupt comes to confirm its transmission. Here should be the packet released in the interrupt handler long after the `send_message()` callback has been served.

Packet reception

The driver must process the received data, store them to the instance of `packet_t` structure and send it to the NIL layer.

The packet representation structure is obtained by the NET server, the `nic_alloc_packet()` encapsulates the communication with the server. The most of controllers receive more than one packet during one poll event. The received packets are stored in `nic_frame_t` structure (further referenced as packet frame) containing the `packet_t` field and the connection to linked list of received packets. The whole linked list of packets is represented by `nic_frame_list_t` type, the `nic_alloc_frame_list()` and `nic_frame_list_append()` encapsulates the work with the list.

The packet frame can be allocated by `nic_alloc_frame()` - this function also obtains the empty packet from the NET server. In the case of error the packet frame is expected to be released by `nic_release_frame()` which also releases the allocated packet. The `nic_release_packet()` should be used to release standalone `packet_t` structure (e.g. allocated by `nic_alloc_packet()`)

After copying received packets data to the packet representation the `nic_received_()` function family should be used to pass the packets to the framework filtering layer described in the following section. This layer also sends the packets to higher networking layers. Single packet can be passed by `nic_received_packet()`, single packet frame by `nic_received_frame()`, the `nic_received_frame_list()` should be used to pass whole list of packet frames.

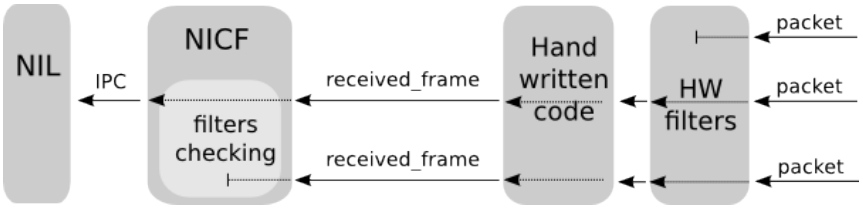
Packet filtering and statistics

First layer of filtering is directly in the hardware. Driver is notified when the filtering mode changes through callback handlers in the `nic_t` structure. Then it reports how perfectly is the hardware able to implement the filtering mode back to the NIC framework.

When a frame is received, it is passed to framework's filtering layer. This layer checks if the packets should be really received according to the current filters settings. No software filtering is required in driver itself.

Note: The filtering layer drops all packets if the device is not in the **active** state.

Figure 4. Packet filtering in driver



The filters are controlled through `unicast_set_mode()`, `multicast_set_mode()`, `broadcast_set_mode()`, `blocked_sources_set()` and `vlan_set_mask()`. The framework checks the requested setting and if valid, the driver is notified to do the hardware setting by `on_unicast_mode_change()`, `on_multicast_mode_change()`, `on_broadcast_mode_change()`, `on_blocked_sources_change()` and `on_vlan_mask_change()` settings. These handlers are set by `nic_set_filtering_change_handlers()` during the initialization phase.

The simplest way how the driver can handle these callbacks is to set the device to the promiscuous mode, the framework filtering will do all filtering in software. If the driver sets the hardware filtering in the callbacks it should report the filtering precision to the framework by `nic_report_hw_filtering()` method - if the driver provides exact filtering as requested the software filtering will be skipped by the framework.

The current filtering setting is returned by default implementations of `unicast_get_mode()`, `multicast_get_mode()`, `broadcast_get_mode()`, `blocked_sources_get()` and `vlan_get_mask()` functions, no code in driver is required.

Device polling

The framework supports device polling by default handlers of `poll_set_mode()`, `poll_get_mode()` and `poll_now()` actions.

The `poll_now()` default handler just checks if the manual polling is allowed (the **on demand** polling mode) and calls the `on_poll_request()` callback.

The `poll_set_mode()` default implementation checks the validity of the requested mode and notifies driver about new mode by `on_poll_mode_change()` callback. The **immediate** and **on demand** modes should be supported by driver. If the **periodic** or **software periodic** is not supported by driver, the framework tries to switch the driver to **on demand** mode by `on_poll_mode_change()` request and starts fibril which periodically calls `on_poll_request()` on the device.

The default `poll_get_mode()` handler just returns last mode set successfully.

The `on_poll_mode_change()` and `on_poll_request()` callbacks must be set during the initialization.

Driver could report current polling mode by calling `nic_report_poll_mode()`. This is useful for setting default polling mode during the initialization.

Accessing the nic_t structure

The `nic_t` structure is considered internal part of **`libnic`** library and therefore it cannot be directly accessed from the driver. However, there may come a necessity to write another implementation of some callback handler which needs to access it.

Such functionality should be encapsulated into separate file, considered as an extension for the **`libnic`** itself. You can define the macro `LIBNIC_INTERNAL` just for this single source and then you can include also the private libnic's headers (mainly `<nic_driver.h>`). Please keep this extensions separate from your driver's code to ensure clean design of the code.

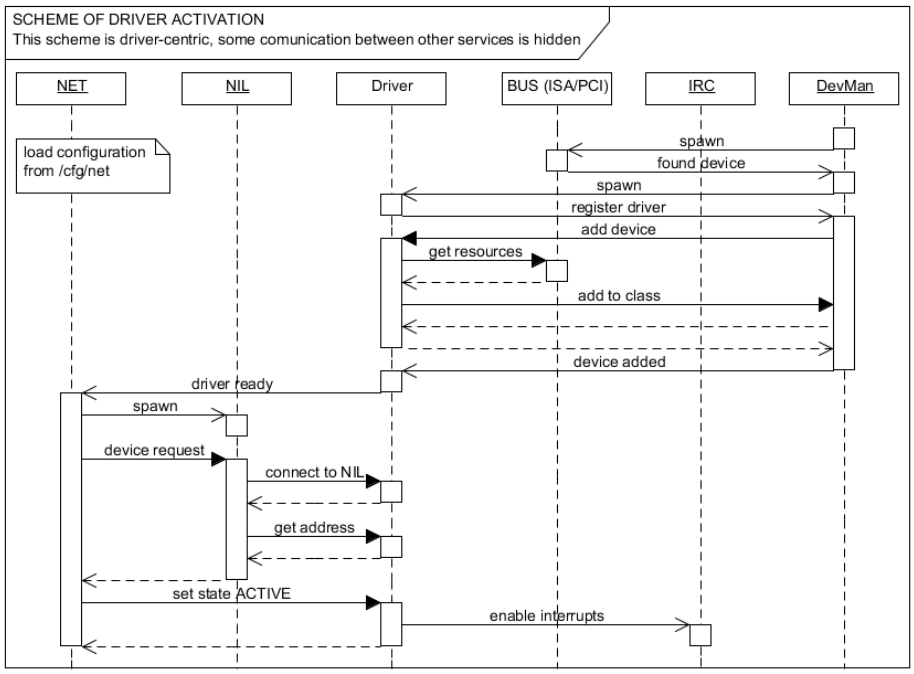
3.5. Driver activation

Originally the NIC drivers (loopback and Novell NE2000) used the network stack architectural model, these had taken the form of modules sharing the structure with protocol (TCP, UDP, IP…) modules. Drivers were started directly by the **NET** service and each required its own service identifier. This was a working but insufficient temporal solution.

Currently is the activation process rather complicated. The **DevMan** and **NET** services start in parallel, **DevMan** spawning the drivers and **NET** the protocol modules. **NET** service probes directory `/cfg/net` (`/uspace/srv/net/cfg/` in sources) and loads the configuration for network interfaces, used in higher layers of network stack. Meanwhile the NIC driver obtains resources for the device from parent bus, registers device in DevMan and informs the **NET** service that the device is ready for operation. **NET** spawns appropriate link-layer (**NIL**) protocol module (**ETH** as for Ethernet) and this requests backward connection and device's address. After that the device is set to the active state (see below), notifying higher layers of network stack about a new possible routing path through this device. When the device is activated, interrupts from the device (as a hardware) are enabled.

The process is summarized in the Figure 5, "Activation sequence diagram".

Figure 5. Activation sequence diagram



4. DMA interface

4.1. Introduction

DMA interface is newly introduced set of syscalls, procedures closely bound to them, DMA server with communication framework and DMA driver interface. For using basic DMA framework it is necessary to include `<dma.h>` and for writing drivers using DMA controller interface `<device/dma_controller.h>`.

4.2. Userspace methods and syscalls

Userspace memory allocation

Userspace interface of DMA framework (except DMA controller driver) is defined in `<dma.h>` and contained in **`libc`**.

First operation before using DMA framework should be it's initialization. There is a procedure `dma_allocator_init()`, which initializes all internal structures and waits for connection to DMA server. In current implementation it is not necessary to call when driver uses only syscall wrappers.

For allocating continuous memory areas usable for basic DMA transfer we have introduced basic syscall `dma_allocate()`. It allocates continuous physical memory area. From this memory is created a memory area at passed virtual address.

Because Intel architecture is specific by it's backward compatibility, even modern systems may contain devices using ISA bus. ISA bus has 24-bit wide address space, so it can reaches only lowest 16 MiB of physical memory. Most devices are stationed on the PCI bus, which is 32 or 64 bit wide, depending on it's version. That's why the memory is zoned and why the DMA memory may be allocated from different ranges: 64-bit range (this range is by default common for all platforms, not only Intel), 32-bit range and 24-bit range.

Another important fact is that memory allocated by `dma_allocate()` syscall is already mapped and is safely accessible for device. Pages mapped this way to virtual memory area must not be remapped to another memory range by the swapping process (currently not implemented in HelenOS).

Function `dma_allocate()` is quite lowlevel, so there is a wrapper `dma_allocate_anonymous()`. It has almost the same functionality as the syscall described above. However, there are two differences: it searches also suitable virtual address (here comes the suffix anonymous) and it's arguments are wrapped into the structure `dma_mem_t`, which contains virtual address, variables describing the type of memory allocation and of course physical address of beginning of the allocated area.

Deallocation of a memory area is possible by simple destroying the memory area, but for consistency with other DMA functions there is a wrapper `dma_unmap()` which destroys memory area passed in argument.

Memory locking

Allocation part of framework is useful only when all memory area sizes and properties are known before it's creating. (So this is useful in drivers or applications like sending network packets with known maximal size.) But very often is necessary to pass to device a (sometimes very large) area of

memory combined from small discontinuous areas of physical memory. This problem is solved by two syscalls.

First of them is `dma_lock()` which returns largest suitable area of continuous physical memory mapped from passed virtual memory area. The address is also registered in kernel and the swapping mechanism cannot do anything with such locked area. This syscall does not work over memory area bounds and after any destroy of memory area the pages are automatically unlocked.

The second syscall `dma_unlock()` pairs with the previous one - it again enables already locked memory area for swapping.

Below are few examples how the above mentioned functions should be used. As first the framework should be properly initialized:

```
#include <dma.h>

int main(int argc, char * argv[])
{
    /* Initialize DMA framework */
    dma_allocator_init();
        ...
}
```

Here is an example how to lock larger area combined from multiple smaller physical areas. Remember, `dma_lock()` can lock only single memory area.

```
#include <dma.h>

/* Lock whole area */
int dev_lock_area(void * vaddr, void * paddr[], size_t size)
{
    size_t count = size;
    size_t currently_locked;
    size_t index;

    for (index = 0; count > 0; ++index){
        if(dma_lock(vaddr, &paddr[index], count, &currently_locked) != EOK)
                /* Handle error */

        if (currently_locked == 0)
                /* Handle error - no page has been locked*/

        count -= currently_locked;
        vaddr = (void*)(((uintptr_t)vaddr) + currently_locked * PAGE_SIZE);
    }
    return EOK;
}
```

Sometimes it is necessary to work with directly allocated and mapped memory:

```
#include <dma.h>

int dev_do_direct_stuff(...)
{
    unsigned long num_of_pages;
    dma_mem_t memory;

    /* Do some stuff and count sizes etc. */
```

```
    memory.size = num_of_pages;
    memory.mapping_flags = AS_AREA_READ | AS_AREA_WRITE;

    /* Allocate area num_of_pages large, with flags
     * AS_AREA_READ | AS_AREA_WRITE and from any address
     * range (the parameter 0) */
    if ((dma_allocate_anonymous(&memory, 0)) != EOK)
        /* Handle error */

    /* Use the allocated memory */

    /* Unmap the area and free the memory */
    if (dma_free(&memory) != EOK)
        /* Handler error */

    /* Do other stuff */
}
```

Very often it is necessary to work with virtual memory area obtained randomly from other parts of system. This method uses `dev_lock_area()` from example above.

```
#include <dma.h>

int dev_do_indirect_stuff(...)

    /* Get a buffer with unknown origin */

    void * area = dev_get_area(...)
    void ** paddr = dev_get_paddrs(...);
    size_t in_size = deg_get_area_size(...);
    size_t out_size;
    if (dev_lock_area(area, &paddr, size_in, &size_out) != EOK)
        /* Handle error */

    /* Do some stuff with locked area */

    /* Unlock the memory */
    if (dma_unlock(area, size_in, &size_out) != EOK)
        /* Handler error */

    /* All memory is again released and kernel structures are clean */
}
```

4.3. Kernel changes

Because of missing suitable kernel memory allocation strategy the kernel required several important changes. New optional allocator interlayer between buddy allocator and memory management has been introduced.

This allocator asks the buddy kernel allocator for buddy block. When the request is smaller than the buddy block, the rest is saved and prepared to satisfy another request. Here is used the "best fit" strategy for satisfying request from remainder of other buddy.

In normal cases this strategy leads to many little unusable remainders, but we supposed that most required requests would be one page large requests for network packets, so this would not be a problem. It leads to reduced time for searching suitable block as well.

Kernel DMA allocator

Handling buddies

Allocated buddies are stored in AVL tree (special generic type of AVL tree was introduced to HelenOS). They are split to used parts and unused rests. Unused parts are stored in two AVL trees: one sorted by size for searching block to satisfy allocation request and second sorted by physical address for merging with freed block.

When a block of DMA memory is freed the tree of possible neighbours is searched at first. If this contains freed block's neighbour which forms a buddy with the freed block these are merged. This procedure is repeated until some neighbour exists. If it does not, two cases are possible:

- 1) There is some allocated and used block, so merged block is inserted back to remainders, ready to be allocated.

- 2) Just merged blocks together are one allocated buddy. The tree containing already allocated buddies is searched and if it contains the buddy it is returned back to kernel buddy frame allocator.

Zoning memory

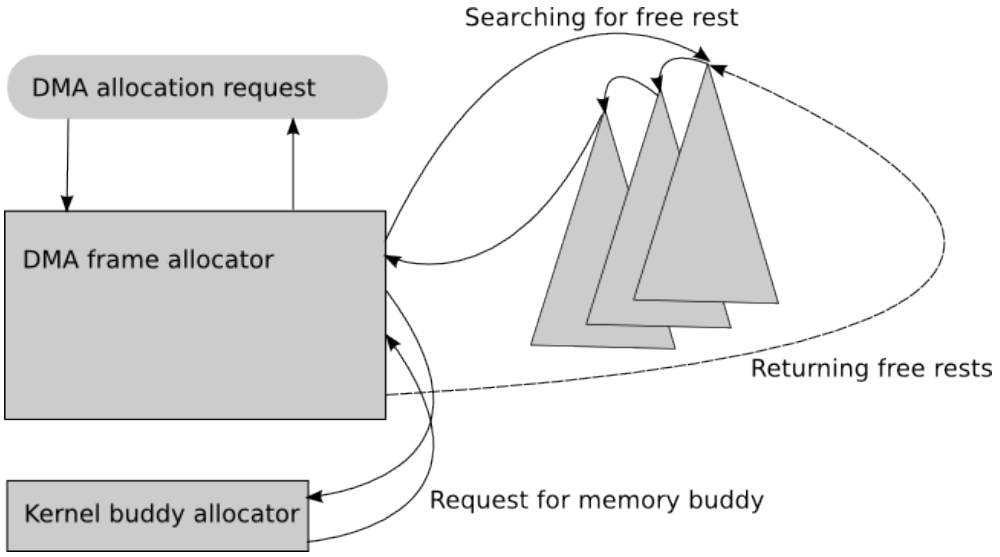
As some devices on Intel architecture use ISA bus, these can use only 24-bit address space. However, in Pentium Pro and newer processors the physical address space can be even larger than 32 bits. This results in some parts of memory being unreachable from some devices. This problem can be solved by charging restrictions on the allocated memory.

When kernel physical memory zones are created, the creating procedure assigns flags to them according to their placement in physical memory. Allocating process can restrict which memory wants to allocate. Upon the request for DMA memory the areas are scanned from the highest suitable one towards lower ones. Memory problem is reported only if even the lowest area has insufficient memory.

This works for Intel architecture, but other architectures could have another restrictions. It is up to developer of other platforms to adapt zones creating on them. At this time memory at all other platforms is marked as zone with 64-bit range.

Some structures for holding free areas in DMA interlayer allocator have to be tripled to hold proper ranges, which also complicates the design.

Figure 6. Scheme of kernel DMA memory allocator



Allocator speed

Because of used data structures, searching (and allocating) of a suitable block internally in DMA allocator is always logarithmic with number of already done allocations. Freeing the DMA memory is again logarithmic with the same number.

Handling memory areas

Direct memory backend

There are several requirements that must hold for the DMA memory:

- 1. When page fault occurs, the mapped frame must correspond to the offset from beginning of the area
- 2. The area must not be swapped out
- 3. The frames must be freed by the DMA memory allocator, not directly to the buddy allocator

These demands are applied through the new memory backend. Each block has counter meaning how many areas are holding this block in their parts of paging tables. Here are few rules describing the behaviour:

- When new area is created, at least one frame is inserted by simulating page faults.
- When a page is removed from area, `frame_free()` callback decrements the counter.
- When the counter reaches 0, the area is deallocated.
- When sharing structure is created, one next reference is added to that area.

The last rule is important in the situation, when sharing has been started but the source area is destroyed and the destination area has not incremented the counter. In this case the physical memory would be freed, what would lead to system inconsistency.

It was also necessary to introduce a new callback in memory backend `share_finish()` notifying that the share structure has been removed. In this situation the counter should be decremented by one.

Memory locking and unlocking

The above described functionality is well suitable if the buffer size is known before the transfer is requested. When a driver should fill memory area provided by a client application, there are three problems:

- 1. Anonymous memory area in userspace does not contain information about physical frames mapped to it.
- 2. Memory may be incontinuous and in worse case even not present.
- 3. Anonymous memory area can be swapped - in fact this is not actual in HelenOS today (swapping is not implemented) - but we have to think ahead.

So the HelenOS kernel has been enhanced with the possibility to pin anonymous memory to physical one - to lock it. Userspace process can ask kernel to search physical address of begin of the area and look forward, how long is the continuous physical memory area. If there are some missing frames the kernel simulates page faults and populates this space by physical memory.

Implementation

At this moment there is no swapping mechanism implemented in HelenOS. Therefore in fact the locking mechanism is not necessary and simple lookup for physical memory addresses would be sufficient for DMA purposes. However, swapping is core feature of operating system and we already implemented a way how to mark some pages as unswappable.

Each locked page is registered in a tree (implemented as generic AVL tree) in its memory area and its frame is registered in another kernel-wide tree (B-tree in this case), respectively a counter for this frame in this tree is increased.

The unlocking operation traverses through tree containing virtual addresses and removes areas, which should be unlocked. It also decrements counters to their physical frames. When the counter reaches zero, it means that the frame is not used by any locked area and is removed from the tree.

The further swapping mechanism will have to traverse the maps mentioned above before swapping the page out.

4.4. DMA memory server

DMA memory server function

On memory area destroy (e.g. termination of process by a failure) pages allocated for DMA transfer and locked for swapping are unlocked and returned to kernel memory allocator. Then they can be reused for another process.

If the device with terminated driver still holds physical memory reclaimed in the process above them, it can rewrite them. This behaviour should be evaded using a memory server which can share memory areas with other processes and this way protect them from deallocating.

The server is launched on bootstrap of HelenOS and is bound as a service. Clients connect to it and identify themselves by unique strings (e.g. hardware paths of devices). After connecting server searches it's memory structure for residual areas owned by preceding instance of the same identificator. If there are no such areas, it generates new unique integer id and returns it to calling process. In other case the old instance of the same identificator was not correctly terminated. The old id is returned and now the new instance has chance to reinitialize the device and then destroy all mapping.

The fact of returning old or new id is not reported to client - it should always initialize the device and call cleanup for the old memory (release all memory owned by it's identifier). After connecting to the DMA server the identification of client is done only by it's unique id.

When the client is connected it can share memory to server (let the server guard the memory) and stop the sharing (when the memory is no longer used by the device).

For sharing is necessary address of shared memory area, offset from begin of first page which should be also locked at server to protect possible swapping out and number of really locked pages. Unsharing memory is easier, client only sends request for unlocking the memory and physical address of first really locked page.

DMA server API:

The API of DMA server is declared in `<dma.h>` in **`libc`**

Table 4. DMA server functions

Function	Description
`dma_cli_register`	Establishes client's connection to server.
`dma_cli_unregister`	Unregisters client from server. Unregistration is successfully done only when all memory already shared by calling process has been successfully released.
`dma_cli_set_ownership`	Shares memory area to server and queries for locking described subarea.
`dma_cli_clear_ownership`	Stops sharing the memory and queries the server to unlock the area.
`dma_cli_cleanup`	Queries the server to release all memory already allocated with the client.

Here is an example of driver code using the DMA server:

```
#include <dma.h>

int dev_init_procedure(....)
{
    /* Obtain handle from server */
    int rc =  dma_cli_register("my unique device identifier", &id);
    if (rc != EOK)
        /* Handle error */

    /* Do device initialization, so it does not need any memory buffers */
```

```
    dev_initialize_first_stage();

    /* Tell the server, it can free all shared memory by this id */
    rc = dma_cli_cleanup(&id);
    if (rc != EOK)
                /* Handle error */

    /* Do other stuff */
    dev_initialize_next_stage();
}
```

When the driver runs, it should share buffers of device with the DMA server:

```
#include <dma.h>

int dev_work_stuff_procedure(...)
{
    /* Obtain memory area */
    size_t number_of_pages;
    size_t offset_in_pages;
    void * area = dev_get_memory_area(...);

    int rc = dma_cli_set_ownership(area, offset_in_pages, number_of_pages, &id);
    if (rc != OK)
        /* Handle error */

    dev_func(area, ....);

    /* If the memory is no more needed, release it */
    int rc = dma_cli_clear_ownership(paddr, &id);
    if (rc != OK)
        /* Handle error */
}
```

On exit the driver should stop the device and release all memory, which is still held by server and unsubscribe from the server:

```
#include <dma.h>

void dev_stop(...)
{
    /* Stop the device */

    int rc = dma_cli_cleanup(&id);
    if (rc != OK)
        /* Handle error */

    int rc = dma_cli_unregister(&id);
    if (rc != OK)
        /* Handle error */
}
```

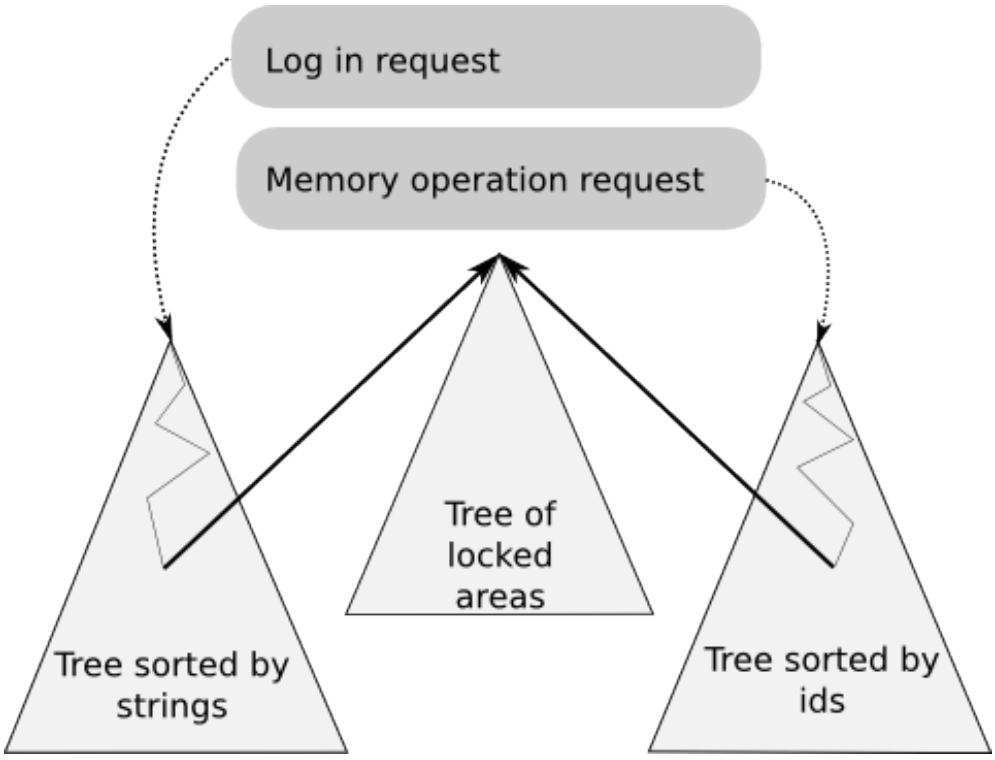
DMA server implementation

DMA server contains two main memory structures: both of them are AVL trees. They contain structures `dma_client_tree_t`. This structure is used to describe areas shared from one identificator to the server. First AVL tree is sorted by string identifiers. This tree is used only when client without knowledge of `id` logs into the server. Then the server assigns to it integer id and from this moment

is used second tree sorted by the `id`. This solution has been chosen because of speed of comparing strings and integers and speed of passing integers and string via IPC. Chosen structures are again generic AVL trees because they can simply implement comparison of string keys.

Each logged in identifier has a set of locked areas. The set is again implemented as a generic AVL tree. Here the tree is sorted by physical address of first locked page from described area. This is because it is most simple way of passing data between client and server. Both of them knows this address and it is not necessary to pass some other data from server to client.

Figure 7. Internal request servicing in DMA server



4.5. DMA controller framework

The device using DMA transfers needs to provide data transfer to the memory without CPU support. The device with bus mastering mode support can provide the transfer on its own, other devices must use DMA controller device - this device locks the memory bus and does the transfer. The DMA controller framework provides the simple way for cooperation between device driver and DMA controller.

The framework contains the set of IPC stubs and skeletons used by DDF framework. It is used for driving access to DMA controller driver. The framework is divided into four basic parts - driver capabilities oriented call, channel oriented calls, device transfer oriented calls and memory oriented calls. The interface is declared in the `<device/dma_controller.h>` **libc** header file.

Driver capabilities oriented call

Calls oriented to driver capabilities allow to detect which features are supported by the DMA controller (memory-memory operations, transfer requests queue,…). The procedure `dma_query_driver_capabilities()` can be used to obtain this information.

Channel oriented calls

DMA channel is an entity representing slot for transfers. They can be physical oriented (e.g. Intel 8237 [4]), where each channel has wires in the bus. HelenOS DMA framework expects even logical channels, which are created by software (the DMA controller driver). It is possible to imagine a channel like a queue (even with maximal length 1), where the transfers are inserted. DMA controller driver exports these channels to client application and the application can use allocated channels. They are represented by structure `dma_channel_t` and it's internal structure is not important for user.

Procedure `dma_query_channels()` queries server for a list of all channels available for transfers. When client has list of all channels, it can call `dma_occupy_channel()` to ask server to allocate specified channel to it. Every driver should use only channels allocated this way. When a channel is not needed anymore `dma_release_channel()` should be called. It marks at server the channel as free and usable for other processes.

Device transfer oriented calls

The device can request the DMA controller to perform the transfer. Each such transfer is identified by the `dma_transfer_id_t` identificator and it must be requested on the channel previously reserved by channel oriented calls. The request must contain the physical address of the memory used in the transfer.

The transfer from the memory to the device can be requested by `dma_set_read_transfer()` or `dma_add_read_transfer()` calls, the `dma_set_write_transfer()` and `dma_add_write_transfer()` requests the device to memory transfers. The transfers requested by `set_()` family are started immediately, the currently processed transfer should be terminated. The transfers requested by `add_()` family are added to the end of transfer request queue on the channel.

The transfer status can be obtained by `dma_check_transfer_status()` method. When the transfer is finished, server should deallocate the transfer and it is not possible to check it's status again. The transfer can be cancelled by `dma_cancel_transfer()` - the running transfer is terminated, queued transfer is removed from the waiting queue and deallocated.

Memory oriented calls

Because some devices like Intel 8237 supports copy operation from memory to memory. This functionality can be useful for many purposes, so here are two additional calls. Their parameters are ranges of physical memory where the transfer should be done. It is necessary to manage these transfers, so the call also returns handle to transfer (type `dma_transfer_t`).

Here are two procedures: First, the `dma_set_mem_cpy_transfer()` sets transfer from physical memory to physical memory on any suitable channel, overwrites any transfer possibly running on this channel, so it is very dangerous to use this call. On the other hand `dma_add_mem_cpy_transfer()` adds transfer from physical memory to physical memory to any suitable channel. When someone tries to rewrite this transfer, the rewriting is delayed until the transfer is completed or the rewritten transfer is sent to another channel, or if it is permitted by flag the rewritten transfer is delayed until rewriting transfer finishes. Of course this transfer can be explicitly cancelled by user. It is strongly recommended to use this call instead.

[4] http://zet.aluzina.org/images/8/8c/Intel-8237-dma.pdf

DMA controller driver interface

DMA controller driver interface is server part of communication protocol between device driver and DMA controller driver. It is placed in `libdrv.a` and definition of the interface is in `<ops/dma_controller.h>` and `<remote_dma_controller.h>`.

Server at initialization of new device instance should fill new instance of `dma_iface_t` with proper callbacks and behave as a driver in other ways. Prototypes of that callbacks are in `<ops/dma_controller.h>`.

4.6. Writing basic driver using DMA bus mastering

Basic driver does not use scatter/gather technology, when you are writing so driver, follow these points:

1. Initialize the DMA framework (by calling `dma_allocator_init()`)
2. Initialize the device and then call `dma_cli_cleanup()`.
3. Allocate continuous buffers by `dma_allocate()`.
4. The buffers should be shared to the DMA server.
5. If the driver uses DMA with external bus transfer driver, it should connect to proper driver.
6. Obtain list of available channels.
7. Select proper channel and tell to server that this channel is occupied.
8. Use the device, it is possible to allocate new DMA memory areas and deallocate old and send requests do DMA transfer driver. Memory, which is not released by application and server is unusable for further usage.
9. Release DMA channel.
10. Disconnect from servers. Before it, all shared memory should be freed so when unsure, whether there is not any area shared to server, call `dma_cli_cleanup()` and then `dma_unregister()`.

4.7. Writing driver using scatter/gather

scatter gather memory access is the most common way of using DMA at modern systems containing PCI bus. The way of using DMA framework is similar to preceding case, but little differs.

1. Initialize the DMA framework (calling `dma_allocator_init()`)
2. Initialize the device and then call `dma_cli_cleanup()`.
3. Allocate continuous areas of physical memory with `dma_allocate()` or lock parts of it’s virtual address space by `dma_lock()`. When the locked memory is not needed anymore, call `dma_unlock()`.
4. Share device buffers to the memory DMA server.

5. If the driver uses DMA with external bus transfer driver, connect to proper driver.

   a. Obtain list of available channels.

   b. Select proper channel and tell to server that this channel is occupied.

6. Use the device, it is possible to allocate new DMA memory areas and deallocate old and send requests do DMA transfer driver. When shared memory area is destroyed, server remembers it until it obtains explicit request to free it. Memory, which is not released by application and server is unusable for further usage. The DMA transfer driver can be polled by `remote_dma_check_transfer_status()`.

7. Release DMA channel if it has been allocated.

8. Disconnect from servers. Before it all shared memory should be freed so when unsure, whether there is not any area shared to server, call `dma_cli_cleanup()` and then `dma_unregister()`.

5. Implemented and integrated drivers

The framework alone would be useless without any working drivers. Implementing example drivers for it provides both functional and mainly architectural testing, proving framework's correctness and good design. This section describes several drivers we have implemented or ported to NICF.

5.1. Loopback

In fact the loopback driver is not necessary. All packets that should be send through the loopback virtual device are immediately received by the same device and sent back up to the network stack. It does not use Ethernet module but its own special Nildummy module, which is just a simplified Ethernet module, so the looping could be done already in the Nildummy module.

However, the architecture can be more symmetric with loopback as separate driver, there are no exceptions needed for having loopback as a network interface. For example statistics are counted here using the NIC framework and any task can get them in the same way as from usual physical NIC. The design is cleaner at the expense of minor performance hit and that's the way HelenOS is going.

Note: Currently there is a bug in network stack causing all packets going to loopback to be sent twice. This is not a bug of the driver itself but probably an error in **ip** module. That's why if you try to `ping 127.0.0.1` and then `nicconf lo --stats` you will see 8 packets to be sent and received instead of 4, what you would expect.

5.2. Realtek RTL8139

RTL8139 is the network interface controller from the late 90's operating on 10/100M Ethernet. The advantage of this controller is its big availability and its emulation in QEMU (although there some differences between real hardware and QEMU emulator behaviour). The driver implements all features the NIC interface allows, the default framework implementations are used when possible, the hardware settings are used if possible to avoid unnecessary software emulation.

Although the controller supports DMA, too strong restrictions exists. The transmition buffer must be aligned to 4B boundary, unfortunately the packets obtained from the higher layers are not aligned properly thus copying to internal properly aligned buffer is used. For receiving only one huge buffer is used and the driver must copy the packet contents by itself. The both buffers are allocated by DMA library and the controller reads/write the packet memory directly.

The development of the RTL8139 driver was tightly related to the development of NIC framework and some parts of code were moved to the **`libnic`**.

Controller Documentation Sources

The driver implementation is based primarily on the RTL8139 datasheets [5] and RTL8139 programming guide [6] published by Realtek company. Unfortunately the official documentation is targeted to the register description rather than functionality description and lacks some important important information (e.g. header of received data is mentioned only in the source examples in the

[5] The datasheets for RTL8139B(L), RTL8139C(L) and RTL8139C(L)+
[6] RTL8139(A/B) Programming guide: (V0.1)

Programming Guide, packet header value during the DMA processing is not documented,…), the OsDev wiki [7], QEMU source files [8] and the source files of Linux [9] and FreeBSD [10] driver were used for studying unclear parts of controller functionalities. The final code of packet reception is based on FreeBSD driver code.

Implementation

The driver implementation is in `uspace/drv/rtl8139` and consist of general sources in `rtl8139_general.[ch]`, register constants definition in `rtl8139_defs.[ch]` and the driver implementation in `rtl8139_driver.[ch]`. The general sources contains the functions used in rtl8139 implementation are not related to the hardware and can be simply moved to the more general library in the future.

The driver uses NIC framework structures, rtl8139 specific data are stored in instance of `rtl8139_t` structure defined in `rtl8139_driver.h`. The implementation boldly uses the default implementations of interface functions of the framework and implements only the callbacks which needs to touch hardware registers. The default callback implementation is used when it is possible, only those parts which needs to work with registers itself are implemented.

The callbacks for NIC interface are assigned in `rtl8139_nic_iface` instance of `nic_iface_t` structure, default implementation is used for others and it's callbacks are assigned through framework interface in `rtl8139_create_dev_data()` called during device initialization in `add_device()` callback of general driver interface in DDF.

The driver uses internal locks for transmitter `tx_lock` and receiver `rx_lock` part of code, the policy locking `tx_lock` first and `rx_lock` second was used for deadlock prevention.

The raw packet transmittion is provided in `rtl8139_write_packet()` function, this function is used as callback for the default `send_message()` handler. In the controller. For the packet transmittion the four buffers are used in the cyclic order. The driver assignes the 2kB to each descriptor, the data are coppied from the packet to the buffer and the controller provides data transfer by PCI bus mastering. If there is no buffer free, the transmitter busy mark is set in NIC framework. In the interrupt handler all previously used descriptors are checked for transfer completion and marked as free, the transmitter busy is unset if some descriptor was released.

The reception is provided in `rtl8139_receive_packets()`. The controller copies the packets to the cyclic buffer allocated by driver, if the buffer end is reached the writing continues from the buffer start. When processing new packets the driver obtains the last processed position in the buffer and the last position in the buffer written by the controller and processes all data between this positions. All frames read in one processing are sent to NIC framework where the software filtering is provided. The hardware filtering is used to prefilter packets before its software processing to decrease CPU work.

Other callbacks are implemented mostly by reading/writing controller registers, the periodic polling mode is implemented by using internal timer increasing by external PCI clock ticks.

The driver was successfully tested on RTL8139B, RTL8139C and RTL8139D controller versions.

[7] http://wiki.osdev.org/RTL8139
[8] http://wiki.qemu.org/Download
[9] `drivers/net/8139too.c` in linux kernel source tree
[10] http://fxr.watson.org/fxr/source/pci/if_rl.c

5.3. Intel E1000

E1000 driver was written for 8254x family of gigabit ethernet controllers from Intel. It was tested on 82541PI controller and virtual e1000 device in Qemu 0.14. These controllers operates on 10/100/1000M Ethernet.

Controller Documentation Sources

The driver implementation is based on 8254x Family of Gigabit Ethernet Controllers Software Developer's Manual [11]

Implementation

The driver implementation is in `uspace/drv/e1000` and consist of general sources in `e1000.c` and hardware representation structures and constants definitions in `e1000_defs.h`.

Device registers are memory mapped. PIO functions are used for accessing them same way as IO ports. EERD registers is used for reading device EEPROM. Because EERD description differs on some devices PCI device_id is used for determining which variant to use.

For using interrupts manual mapping needs to be created. See PCI improvements section. Interface could work without this mapping only in software periodic or on demand polling mode.

The driver uses four internal locks. `tx_lock` for transmitter and `rx_lock` receiver part of code. `ctrl_lock` guards access to CTRL register and `eeprom_lock` guards access to eeprom. The policy locking in this sequence: `rx_lock`, `tx_lock`, `ctrl_lock` and `eeprom_lock` was used for deadlock prevention.

No interrupts are used for transmitting. Packet physical address locked by `dma_lock()` is written into transmit ring. Device uses 64 bit long address. No packet copying is used. The packet transmition is provided in `e1000_write_packet()` function

Packet is preallocated before receiving and it's physical address is locked by `dma_lock()` and filled into receive descriptor ring. Once the packet is received, new empty packet is allocated. No packet copying is necessary here either. Possible improvement could be to postpone allocation of some packets to avoid memory consumption. Question is whether it is worth the risk of losing some packet. For example while receiving a lot of small packets. The packet reception is provided in `e1000_receive_packets()` function.

It is possible to limit interrupt frequency (periodic polling mode) or disable interrupts by using software polling. Default is at least 250 micro seconds between interrupts.

There is 16 register array for receive address filtering. The first one must be filled with interface address. In this driver top registers are used for unicast addresses and the bottom registers are used for multicast addresses. Border between them is variable. When there is not enough space in receive address array the device is switched into unicast or multicast promiscuous mode and NIC framework does the filtering. For multicast there is possibility for using multicast array to prefilter traffic for framework. This has not been implemented yet.

[11] http://download.intel.com/design/network/manuals/8254x_GBe_SDM.pdf

Device supports adding and stripping VLAN tags. It is also possible to filter specified VLAN tags while receiving.

5.4. Novell NE2000

HelenOS already featured a driver for Novell NE2000 ported from Minix as a part of the Lukas Mejdrech's thesis (HelenOS network stack). This driver was not working very well under stress conditions and the code was not fitting to HelenOS, therefore Martin Decky had rewritten the driver from scratch in HelenOS mainline.

Both the Minix version (in the beginning of the project) and MD's version were inserted into NIC Framework, removing the duplicated parts. This driver was also extended with filtering support, support for MAC address change and tweaked to better work under stress conditions.

Although this driver is fully integrated to the NIC framework, it does not exploit the capabilities of DMA. This NIC's documentation (datasheets) describes DMA access only on chip's local bus (this is rather confusing indeed), all communication with this device can be done only through port I/O. This hardly limits the speed of this NIC's operation and causes high CPU utilization.

6. Tools developed

There were several tools developed both for user (system administrator) and drivers' authors.

6.1. NIC configuration utility - nicconf

Although the main purpose of NIC's is to send and receive messages, the interface is much richer. The NIC can range several states, its address can be changed and there are various settings that can be queried and modified. This tool is communicates with the lowest level of network stack, with the network interface card drivers. It can also report current settings for other layers (**IP** configuration…) but this is provided just because there is currently no other tool in HelenOS that could display it. This feature is about to be removed in the future.

Its functionality is similar (but not identical) to those of `ifconfig` and `ethtool` in Linux.

Nicconf commands to follow one of these formats:

```
nicconf global_action
nicconf interface interface_action
```

The interface is name of the network interface as stated in its configuration file (in `/cfg/net/` directory) under the `NAME` property. If `all` is used as the interface name the actions specified are applied on all currently present interfaces. By default, the `nicconf` command is equal to `nicconf all --state --config --stats`.

Global actions

Currently the only supported global action is `-h` or `--help` - the request for displaying help for the `nicconf` command. For information for concrete action type `netconf --help` action where action is the long name of the action (without the two dashes, e.g. `netconf --help config` for the `--config` action).

Interface actions

Specifying action without arguments only displays information about current settings, those actions which allow changing the settings require an argument.

`-a --autoneg`	Display/set current autonegotiation state. The argument (if applied) must be one of these: `enable`/`disable`/`auto`.
`-c --config`	Display current configuration obtained from the **NET** service - binding to another services in network stack, current MAC address and DDF path as well as IP configuration. This option is deprecated.
`-i --info`	Display device's description and list its capabilities.
`-m --mac`	Display/set MAC address for the device. The address must be specified as six two-digit hex numbers separated by hyphens (-) or colons (:), e.g. `12:34:56:AB:CD:EF`.
`-o --opmode`	Display/set operation mode for the device. The argument (if applied) must be in form `speed`,`duplex`,`role`, where `speed` is the desired speed in Mbps, `duplex` is either `full` or `half` and `role` is `master`, `slave` or `auto`.

`-p --pollmode`	Display/set device's polling mode. The argument (if applied) must be either `immediate` (default mode triggering interrupts when a new event is detected), `on_demand` (some application manually polls the NIC for new events), `periodic,seconds,microseconds` (either the card or NICF automatically polls with period specified as the arguments) or `software_periodic,seconds,microseconds` (same as periodic, no hardware interrupts).
`-s --state`	Display/set NIC's state - one of these: `active`/`down`/`stopped`. `active` is the usual operation state, `down` state keeps settings but all communication (both outcomming and incomming) is disabled and going to the `stopped` state results in a complete restart of the NIC.
`-t --stats`	Display statistics for the device.
`-u --pause`	Display/set pause mode. The argument (if applied) must have format `rec_mode,trans_mode[,time]` where the modes must be `on` or `off` and the `time` is a 16-bit number.
`-v --vlan`	Set VLAN. The argument is number between 0 - 0xFFF.

6.2. NIC testing tool - nictest

Application `nictest` should provide a way to both manually and automatically test NIC driver's functionality. You can specify filters configuration - this is not available in the `nicconf` command because the filters should be set up by another application (firewall) through NICF external API rather than by user manually. Nictest can send messages with fake source and arbitrary destination, receive messages from the NIC and perform several types of automatic tests, described below.

Because of poor architecture of current network stack, nictest cannot work with usual network settings - it's not possible to eavesdrop or fork the communication between layers. Therefore the internet layer (**IL**) must be instead of by the **IP** module implemented by our module, called **ILdummy**. This is done by specifying

```
IL=ildummy
```

in the NIC's configuration file located in `/uspace/srv/net/cfg/` directory. With this settings are the usual applications (`ping` etc.) not working.

ILdummy does not modify packets that should be sent to the network, the source and destination are left as they are and the packet is handed over to the **ETH** module. This module just copies the addresses and frame type to the packet header and then is the packet handed to our NIC driver.

When a packet is received (from the NIC through the **ETH** module) it is placed into a queue in the **ILdummy** module. If this queue is full (it's limited to 16 packets) the oldest packet is released (and the most recent is placed on the end of the queue). The number of such discarded packets can be queried from the module. When nictest is ready to receive a packet, it polls the **ILdummy** module and simply fetches the oldest packet from the queue.

During the development you would probably develop the driver on an emulator rather than on physical hardware. There is a nice step-by-step tutorial how to bridge qemu emulator to real network on HelenOS wiki. When the bridge is set up you can see the communication using tcpdump or wireshark on the host computer or bridge two instances of HelenOS and send messages between them.

Commands

Nictest has similar syntax of commands to the nicconf:

```
nictest interface command [arguments]
```

Here is a list of possible commands with their arguments:

Command and arguments		Description
`send`		Send frame
	`to MAC`	Destination MAC address (default: broadcast)
	`from MAC`	Source MAC address (all zeros = default: NIC's address)
`dump`		Dump all received frames until keypress
`unicast`		Setup NIC's unicast settings
	`block`	Block all unicast frames
	`default`	Pass only frames with NIC's MAC as destination
	`list MACs…`	Pass frames with NIC's MAC or any MAC from the list as destination
	`promisc`	Promiscuous mode (all unicast is received)
	`info`	Display info about current unicast settings
`mcast`		Setup NIC's multicast settings
	`block`	Block all multicast frames
	`list MACs…`	Pass frames with any MAC from the list as destination
	`promisc`	Promiscuous mode (all multicast is received)
	`info`	Display info about current multicast settings
`bcast`		Setup NIC's broadcast settings
	`block`	Block broadcast frames
	`accept`	Accept broadcast frames
	`info`	Display info about current broadcast settings
`blocksource`		Setup blocked source addresses
	`set MACs…`	Set blocked source address from the list
	`clear`	Remove all blocked sources
	`info`	Display info about current blocked sources settings
`poll`		Poll the device
`test_master`		Start automatic testing as master
	`states`	Do automatic testing of states transitions and behaviour
	`filters`	Do automatic testing of filtering modes
	`stress`	Do stress test
	`throughput`	Benchmark NIC's network throughput
	`address`	Do automatic testing of address changes
`test_slave`		Start automatic testing as slave (no arguments)

Automatic testing

This variant of testing requires two bridged (either virtually or connected with a cable) instances of HelenOS compiled for the same architecture. On one machine you start the master part of the test, on the second one the slave part. Slave does not require any argument, everything is sent through the network by the master.

States test

NIC's behaviour in each state is strictly defined. It should transmit and receive messages only in the **active** state, going to the **stopped** state means that all settings must be reset etc. This test controls that this behaviour is correct and that the NIC is operable after every possible transition.

Filter test

Nictest defines several configurations of filters with test packets and information which of them should be accepted or refused. For each configuration the master sends requested filters settings to the slave and after the slave confirms that the filters have been set up, it sends the test packets. After a timeout slave sends a response with a list of received packets to the master. Master then compares the list with correct result and prints out the report.

Stress test

There are two parts of this test. In the first part the master just starts sending a lot of (32 000) packets to the slave. After the slave does not receive any new packet for a few seconds, it sends a report how many packets were received by nictest and how many were discarded in **ILdummy**. The results are printed out on the master. In the second part the slave confirms each received packet by another acknowledgement packet to the master. Master sends the test packets and polls ildummy for acknowledgements. After all (again 32 000) test packets are sent master waits 15 seconds for acknowledgements that are yet pending and then reports how many packets were acknowledged.

Throughput test

This test is similar to the first phase stress test - it is only using the longest possible frames (by default, you can also set lower lengths), and measuring how much time it takes to send them. The slave also measures how long it took to receive all test messages and reports the number of received bytes and the time to the master.

Address test

The address can be changed either directly by calling `nic_set_address()` or implicitly by restarting the NIC (going to the **stopped** state and back to **active**). This test tries both ways and verifies that packets are received only for the set up address.

6.3. Logging support

Logging is a common useful debugging technique. However, there was no logging service in HelenOS - the only possibility was `stdout` and `stderr` redirected to kernel log displayed on kernel console and backed up in a file. Moreover, there was a bug causing the log file to be strongly corrupted. Because

currently HelenOS does not have utilities like `grep` or `sed`, even if the file was not corrupted the filtering could not be done directly in HelenOS.

The necessity of some logging system was found in the USB team as well approximately in the same time as we have. They have developed much simpler system just as a wrapper to printing into separated file. This may be sufficient for logging actions of single process but because the network stack is scattered across many processes, we have decided to create a standalone service called **Logger** (`/srv/logger`), similar to e.g. `syslog` in UNIX-like systems. This service gathers log records through IPC from multiple processes and stores them in single file together. This file uses binary format in order to allow easy filtering and faster seeking through that file. The logs can be probed through application `logview` rather than by raw access to the file.

Because of name-clash with USB team's logging support functions from our login system use prefix `nlog_` instead of simple `log_`.

With each message is recorded its severity (the levels were adopted from `syslog`), time, source task ID and labels associated with the record. These labels are added to the record automatically - there is one default and possibly more optional profiles in each task, with the labels recorded in the profile. Labels should form a hierarchy, for example for each driver's logs are tagged with labels `networking/nic` and `drv/nic/driver_name`. Then you can filter for example only records with label `networking`, showing both records from the driver, **NET** service etc.

Although the logging is working well for debugging, in stress tests it revealed to be radically slowing down. That's why the driver can easily suppress unimportant records (with low level of severity) both at runtime and also at compile time by defining appropriate macros, completely removing the logging overhead. This can be also useful for example if you wan't to suppress debugging records from the **libnic** library but use your own messages at debug level in the driver itself.

Above mentioned application logview allows three modes of viewing the records:

- dump: simple print out logged records
- continuous: print out logged records and continue periodically polling the logger for new messages
- interactive: browsing the records forward or backward and jumping through the records (controlled by the user)

Naturally it allows filtering messages according to source task, severity and labels and modifying the print-out format.

7. Changes not related to NICF

7.1. PIO improvements

`pio_<size>_read()` and `pio_<size>_write()` functions could now be used for accessing device memory and IO ranges. This is useful for devices with memory mapped registers.

This functions are used in interrupt pseudocode. Unfortunately before using them there it is also necessary to have address mapped in kernel space. Since current HelenOS version has some memory limitations (see issue #3 and #343 from the HelenOS mainline bug tracker [12] [13]) mapping is added manually in `kernel/arch/(amd64|ia32)/src/mm/page.c` files. In IA32 architecture, identical mapping is used. `PA2KA()` mapping is used for AMD64 architecture.

7.2. DDF callback device_added()

Originally the DDF specified only single callback for driver, called when a new device is discovered and should be registered in the driver - this is called `add_device()`. However, the process is finished only after the user-defined callback returns.

The **NET** service needs to be notified at the moment when the NIC is operable, which is not until the `add_device()` returned. That's why the optional `device_added()` callback was added to the DDF interface. This allows the driver to do some post-initialization operations, in case of NIC to notify the **NET** service.

7.3. Hardware resources parsing

The driver can ask its parent device (the bus where it is connected) about the hardware resources assigned to the device (interrupt number, I/O ports area of the device). The original **DDF** interface passes the resources in as the (in general case) unsorted array of the C unions (interrupt number, memory range or I/O ports range). The layer which splits the resources according to the array was added to the **`libc`**. It is implemented in `uspace/lib/c/src/device/hw_res_parsed.c` and can be included by `<device/hw_res_parsed.h>` header file.

The new interface allows to parse resources list obtained by the parent by `hw_res_list_parse()` function or obtain the parsed list of resources from the parent device directly by `hw_res_get_list_parsed()` function.

7.4. PCI interface

During development of drivers the need of PCI configuration space access occurred, the interface for reading and writing PCI configuration space registers was added. Unfortunately the similar interface was developed and merged to the mainline during the USB device driver development. Because of the interface similarities the change to using mainline version will be trivial.

[12] http://trac.helenos.org/ticket/3
[13] http://trac.helenos.org/ticket/343

7.5. ILdummy network module

Testing of NICs required simpler behaviour of network stack. The IP and higher layers can be bypassed to the ILdummy network module, lying on the IL layer but communicating directly with the testing application, without any abstraction of sockets. More about ILdummy's functionality is described in Section 6.2, "NIC testing tool - nictest" section.

8. How to write a NIC driver

This tutorial should teach you how to integrate an ethernet card driver into HelenOS and use the existing libraries and services. It should help you both with writing a new driver from scratch or porting an existing one. The details regarding the actual implementation of driver's business logic are up to you, of course.

8.1. Compilation

HelenOS drivers are just usual userspace executables. However, in order to categorize the executables these are separated into `/app/`, `/srv/` and `/drv/` directories. Your driver should be located in the last one. Here are the steps to do that:

1. Create a new directory with your driver's name under `/uspace/drv/`, e.g. `/uspace/drv/mydriver/`.

2. Write simple `main.c` file and `Makefile` and put them into the directory. The `Makefile` can look like this:

```
USPACE_PREFIX = ../..
BINARY = mydriver
SOURCES = \
	main.c

include $(USPACE_PREFIX)/Makefile.common
```

1. Add another file, called `mydriver.ma` to the directory. You can leave it empty for now.

2. Open `/uspace/Makefile` and add `drv/mydriver` to the `DIRS` variable (for appropriate platforms).

3. Open `/boot/arch/your-driver-platform/Makefile.inc` and add your driver name to the `RD_DRVS` variable.

Now build HelenOS and run it. The driver will be not started yet (it is not associated with the device), however, you can run it manually from the `/drv/mydriver/` directory.

8.2. Configuration files

The file `mydriver.ma` is responsible for associating the driver with the devices for whose is your driver suitable. Each line describes one device in a form of `match-score match-id` pair, separated by whitespace.

Match ID specifies the device itself - the exact manner how this string is generated is bus-specific, e.g. on the PCI bus this follows the form `pci/ven=VENDOR_ID&dev=DEVICE_ID`. Match score specifies how suitable the driver is for the device - usual value is 10. For example the RTL8139 driver uses `.ma` file with this content:

```
10 pci/ven=10ec&dev=8139
```

Another important file you have to write is located in the `/uspace/srv/net/cfg/` directory - you should call it `mydevice.nic` . Actually only the `.nic` extension is really important, the filename

just must be unique in the directory. The contents of this file is in standard HelenOS CFG format (`key=value` pairs). There are two keys (properties) important from the driver perspective:

- `NAME`: this is the identificator under which will be the device known through the system
- `HWPATH`: (DDF path) binds the configuration file to the particular device.

Other properties configure network stack settings and their exact meanings are out of scope of this document. Here is an example of such configuration file:

```
# My driver configuration
NAME=mydriver
HWPATH=/hw/pci0/00:03.0/port0

NIL=eth
IL=ip
ETH_MODE=DIX
ETH_DUMMY=no
IP_CONFIG=static
IP_ADDR=10.0.2.15
IP_ROUTING=yes
IP_NETMASK=255.255.255.240
IP_BROADCAST=10.0.2.255
IP_GATEWAY=10.0.2.2
ARP=arp
MTU=1492
```

In order to insert this file into HelenOS image (not only into the sources), you have to add path to this file to the `NET_CFG` variable in `/boot/arch/your-driver-platform/Makefile.inc`.

8.3. DDF and NICF integration

Aside from common **`libc`** which is included by default each NIC driver uses three libraries:

- **`libdrv`**: the DDF library
- **`libnet`**: library for cooperation with the network stack
- **`libnic`**: NIC framework library

You have to set these to be linked in the `Makefile` - add these lines before including the `Makefile.common`:

```
LIBS += $(LIBDRV_PREFIX)/libdrv.a
LIBS += $(LIBNET_PREFIX)/libnet.a
LIBS += $(LIBNIC_PREFIX)/libnic.a
EXTRA_CFLAGS += -I$(LIBDRV_PREFIX)/include
EXTRA_CFLAGS += -I$(LIBNET_PREFIX)/include
EXTRA_CFLAGS += -I$(LIBNIC_PREFIX)/include
```

In your `main.c` file you have to include several header files from these libraries:

```
#include <nic.h>            // NICF library (includes most of DDF library internally)
#include <ddf/interrupt.h>  // Part of DDF library handling the interrupts
```

You may also find handy to include the `<nlog.h>` for comfortable logging. See Section 6.3, “Logging support” for details.

Each driver has to setup several structures `driver_t`, `driver_ops_t` and after initializing global data, run the main message loop which is implemented in the `ddf_driver_main()` function. The `driver_t` structure just specifies the name and sets pointer to the `driver_ops_t` structure. Regarding the `driver_ops_t` structure, you have to fill only the `add_device` field. It specifies a handler to your function, where you initialize the driver structures for a newly discovered device.

Other processes communicate with drivers via DDF interface. For ethernet card drivers, this is provided through `nic_iface_t` structure - you can see the specification in `/uspace/lib/drv/include/ops/nic.h` file. Some of the functions (methods) specified in the interface are mandatory, some of them are just optional. NICF provides default implementations for some of these methods - you just have to call the `nic_driver_implement()` function and these are automatically filled in. Of course, if you need your own specific implementation, you can specify them manually in the `nic_iface_t` structure and the `nic_driver_implement()` function will keep them as you set.

Here is a excerpt of the `main.c` file with these structures:

```
static ddf_dev_ops_t mydriver_dev_ops;
static nic_iface_t mydriver_nic_iface;

static driver_ops_t mydriver_driver_ops = {
    .add_device = mydriver_add_device
};

static driver_t mydriver_driver = {
    .name = "mydriver"
    .driver_ops = &mydriver_driver_ops
};

int main()
{
    nic_driver_init("mydriver");
    nic_driver_implement(&mydriver_driver_ops, &mydriver_dev_ops,
        &mydriver_nic_iface);

    nlog_info("Starting my driver");
    return ddf_driver_main(&mydriver_driver);
}
```

Now some common procedure that should be used in the `add_device()` function:

1. Create a new structure `nic_t` using the `nic_create_and_bind()` function. This is the main NICF structure associated with each device.

2. If you need your own structure for the driver, you should allocate it and fill into the `nic_t` structure using the `nic_set_specific()` function.

3. Fill your handlers to the nic_t structure using `nic_set_ something _handler()` functions. Actually, the only mandatory handler is `write_packet()`.

4. Query the hardware resources provided by the parent driver (see function `nic_get_resources()`).

5. Enable port IO (function `pio_enable()`) and prepare your device for operation.

6. Inform the NICF about your device's MAC address - use `nic_report_address()`.

7. Setup DDF interfaces and create a new DDF function for the device - function `nic_register_as_ddf_fun()`.

8. Connect to the NET and IRC service using `nic_connect_to_services()`. The NIL service will be bound later.

9. If everything went OK, return `EOK` as the return value.

Here is an example implementation of this function:

```
static int mydriver_add_device(ddf_dev_t *dev)
{
    int rc;
    /* Allocate NIC structure for the device. */
    nic_t *nic_data = nic_create_and_bind(dev);
    if (nic_data == NULL) {
        return ENOMEM;
    }
    nic_set_write_packet_handler(nic_data, mydriver_write_packet);
    nic_set_state_change_handlers(nic_data, mydriver_on_activating, NULL,
        mydriver_on_stopping);

    /* Allocate your own data */
    mydriver_data_t *my_data = malloc(sizeof(mydriver_data_t));
    if (my_data != NULL) {
        memset(my_data, 0, sizeof(mydriver_data_t));
        nic_set_specific(nic_data, my_data);
    } else {
        /* Do cleanup: see nic_unbind_and_destroy() */
        return ENOMEM;
    }

    /* Get HW resources */
    hw_res_list_parsed_t hw_res_parsed;
    hw_res_list_parsed_init(&hw_res_parsed);
    rc = nic_get_resources(nic_data, &hw_res_parsed);
    if (rc != EOK) {
        /* Do cleanup */
    }
    /* Check if the resources are correct (IRQ number, I/O port range... */
    /* Fill in the resources into your data structure */
    hw_res_list_parsed_clean(&hw_res_parsed);

    /* Enable port I/O */
    if (pio_enable(my_data->port, my_data->range_size, &my_data->port) != EOK) {
        /* Do cleanup */
        return EADDRNOTAVAIL;
    }

    /* Initialize your device here */

    /* Report MAC address to the NIC framework */
    rc = nic_report_address(nic_data, &my_data->mac);
    if (rc != EOK) {
        /* Do cleanup */
        return rc;
    }

    /* Setup DDF interface and create a new DDF function for the device */
```

```
    rc = nic_register_as_ddf_fun(nic_data, &mydriver_dev_ops);
    if (rc != EOK) {
        /* Do cleanup */
        return rc;
    }

    /* Connect to NET and IRC services */
    rc = nic_connect_to_services(nic_data);
    if (rc != EOK) {
        /* Do cleanup */
        return rc;
    }

    return EOK;
}
```

Leave the `mydriver_write_packet()` handler implementation empty for now. At this moment, you can try to compile the driver, run HelenOS and try typing `nicconf` on the console. The device should be reported on the output. However, in order to be able to send and receive messages, there are still a couple of things you have to do.

8.4. Sending and receiving

Sending the message is rather a matter of the driver itself. You should include `<packet_client.h>` and use functions `packet_get_data_length()` and `packet_get_data()` to get the data. Filling them to the send it to the network is up to you. After the packet is processed (both successfully or with an error), you have to call `nic_release_packet()` to free the packet.

If DMA is used the NIC requires physical address of the memory with desired data. If you just need a buffer with known physical memory, you should allocate it using `dma_allocate_anonymous()`. Such buffer can be destroyed using `dma_unmap()`.

If you want to pass the packet data as they are, the memory holding them must be prepared - locked. The function `dma_lock()` does this for you - it forces physical location of the memory and prohibits any swapping out until you call `dma_unlock()`. For easy use on packets there are functions `nic_dma_lock_packet()` and `nic_dma_unlock_packet()`.

Receiving messages is a bit more complicated. You have to register interrupts using `register_interrupt_handler()` function. You should do this in the `add_device()` function. The exact way how interrupts are handled in HelenOS and the pseudo-code instructions are out of scope of this document - see Lenka Trochtova's thesis [14] for details. So, add this piece of code into the `add_device()` function after initializing the device.

```
    /* Register interrupts */
    rc = register_interrupt_handler(nic_get_ddf_dev(nic_data), my_data->irq,
        mydriver_interrupt_handler, &my_data->irq_code);
    if (rc != EOK) {
        /* Do cleanup */
        return EINVAL;
    }
```

There are three states in which can be the driver:

[14] Lenka Trochtova: Device drivers interface in HelenOS system; 2010; http://www.helenos.org/doc/theses/lt-thesis.pdf

- `ACTIVE`: usual mode = operating, sending and receiving messages
- `DOWN`: neither sending nor receiving messages, but still keeping its settings
- `STOPPED`: down and with erased settings

NICF provides three handlers for transition between theses states: `on_activating()`, `on_going_down()` and `on_stopping()`. These handlers are set using `nic_set_state_change_handlers()` in the `add_device()` function. The driver is initially in the `STOPPED` state.

In the `on_activated()` handler, you should prepare the device to be able to accept frames and then enable interrupts from this device (`nic_enable_interrupt()`). In the `on_going_down handler()` you may disable the interrupts from your NIC in order to discard frames directly in the hardware. Nevertheless, these frames would be discarded anyway in the NICF after being reported to come. In the `on_stopped()` handler you should disable interrupts as well and bring the device to its initial state.

After you have successfully received a frame (through the interrupt handler set up above), you should call the `nic_received_frame()` or `nic_received_packet()` function (see also `nic_alloc_frame()` and `nic_alloc_packet()`). All the filtering (see below) and propagation into higher layers in the network stack is already implemented in the NICF framework. If your device supports receiving multiple frames upon single interrupt, you can use function `nic_received_frame_list()` as well (see also `nic_alloc_frame()`, `nic_alloc_frame_list()` and `nic_frame_list_append()`).

```
void mydriver_interrupt_handler(ddf_dev_t *dev, ipc_callid_t iid,
    ipc_call_t *call)
{
    nic_t *nic_data = (nic_t *) dev->driver_data;

    if (/* There are frames to be received */) {
        nic_frame_list_t *frames = nic_alloc_frame_list();
        while (/* There are frames to be received */) {
            nic_frame_t *frame = nic_alloc_frame(nic_data, length, 0);
            if (frame != NULL) {
                /* Get packet length and allocate enough space in the frame */
                void *data_buffer = packet_suffix(frame->packet, length);
                /* Move the data from the device to the data_buffer */
                nic_frame_list_append(frames, frame);
            } else {
                /* Discard packets from device */
            }
        }
        nic_received_frame_list(nic_data, frames);
    }
    /* Handle if there were multiple reasons for interrupt */

    async_answer_0(iid, EOK);
}
```

However, this example does not exploit the DMA - data are copied from the device to the data buffer. For high-performance drivers you need to preallocate the frames in advance in the `add_device()` function and then each time some frames are removed from NIC's RX ring.

For the version using DMA you need functions mentioned several paragraphs above - `nic_dma_lock_packet()` and `nic_dma_unlock_packet()`.

Correctly received packets (both accepted or filtered in NICF) are counted automatically in the statistics. If there were problems with receiving frames, you should report them using the `nic_report_receive_error()` function.

Most NICs report successfully sent packets or errors in transmission in the interrupt routine as well - you should call `nic_report_send_ok()` and `nic_report_send_error()` to update the statistics.

So, now you should be aware of what to do to set up basic operation of your driver.

8.5. Advanced operations

Filters

Most network interface cards are able to limit the set of accepted frames, these do not receive each frame that is detected on the link medium. The default behaviour is defined as receiving only frames with destination address equal to NIC's MAC address, and broadcast frames (all without any restrictions on VLAN tags). NICF defines handlers `on_unicast_mode_change()`, `on_multicast_mode_change()`, `on_broadcast_mode_change()`, `on_blocked_sources_change()` and `on_vlan_mask_change()`. You should set them using `nic_set_filtering_change_handlers()` in your `add_device()` function.

These handlers should change the mode on the device. If the mode cannot be used on the device at all (for example it does not support promiscuous mode), the handler should just return `ENOTSUP` and the mode change fails.

However, the device could only support more coarse modes than NICF. Therefore the driver should also inform the NICF how exact the filtering is using the function `nic_report_hw_filtering()` which enables software filtering inside the NICF.

You can query the current mode (prior to the change) by calling functions `nic_query_unicast()`, `nic_query_multicast()`, `nic_query_broadcast()`, `nic_query_blocked_sources()` and `nic_query_vlan_mask()` in the handlers.

Wake-on-LAN

Some NICs support the wake-on-LAN feature. There are many ways how the computer can be waken up, that is why the NICF defines so-called WOL virtues. WOL's cannot be software emulated and setting them is a matter of each device, therefore the NICF just keeps a records with current WOL settings and does some checks on the arguments. There are two handlers for adding and removal WOL virtues, called `on_wol_virtue_add()` and `on_wol_virtue_remove()` - their meaning is quite obvious. You should set the handlers in the `add_device()` function by calling `nic_set_wol_virtue_change_handlers()`.

Other operations

There are several another concepts defined in the DDF interface for NICs as the autonegotiation control, querying current cable state and link operation mode, VLAN tagging or offload computing. However, their functionality is rather hardware-oriented and therefore NICF does not encapsulate them. If you can implement them (which is recommended, of course), you should set DDF callbacks in the `nic_iface_t`.

Debugging compilation

During the development it is recommended to build both **`libnic`** and your driver with debug-level logging enabled during development. This is done using line

```
EXTRA_CFLAGS += -DNLOG_COMPILE_MIN_SEVERITY=NLOG_COMPILE_SEVERITY_DEBUG
```

in both libnic's and your driver's Makefiles and setting runtime severity level to `DEBUG` via `nlog_set_min_severity(DEBUG);` (do it in the main function after `nic_driver_init()`). If anything goes wrong, you can see the logs by typing `logview -d` on the console (type `logview --help` for another options to view log records).

9. Driver testing

The testing of the driver functionalities can be provided in few different ways using HelenOS networking infrastructure present in the HelenOS before the project has stared or new application implemented during the project.

As all of the developed controllers are emulated by QEMU, the QEMU usage will be described. The default configuration in the source tree is prepared for usage with QEMU. The examples will be provided on RTL8139 controller, it's interface name in QEMU is `eth0`.

9.1. Nicconf

The first step in the testing can be running nicconf to see if the controller was detected properly and to see some statistics and configuration. The detailed description if the `nicconf` command is in the Section 6.1, "NIC configuration utility - nicconf".

9.2. Ping command

The simple ping command already present in the HelenOS before the NIC project has started can be used to verify wheter the controller is working correctly. This way assumes the other part prepared for replying echo request. The advantage is that default configuration for the controller can be used - the `ip` as the IL and `eth` as NIL layer.

In QEMU the user networking should be used. To provide RTL8139 testing in QEMU, use the IA32 project image and run

```
qemu -net nic,model=rtl8139 -net user -cdrom image.iso
```

In the bshd run

```
ping 10.0.2.2
```

to ping default gateway created by QEMU. The disadvantage of this way is dependency on nontrivial ip layer functionality and ARP protocol implementation, only basic sending and receiving can be tested. Testing on real hardware is also possible, only need is that other side answers ECHO requests correctly.

9.3. Nictest

Aside from simple ping to the gateway, the application nictest was developed to provide advanced tests of the controller functionality by both interactive and automatic testing. The disadvantage of this kind of testing is necessity of the second nictest instance on the other side of network.

Configuration

The simple ildummy layer, which just simply sends packets to application/NIL layer, is used instead of ip. In the controller configuration (`eth0.nic` file in `uspace/srv/net/cfg` directory) set

```
IL=ildummy
```

System start

The two running instances of nictest running on different computers/QEMU instances are needed.

> **Note**
>
> The combination of real HW and QEMU is also possible - in that case the network traffic must be bridged to the QEMU on the computer where QEMU instance is running. One possible way to do so is on HelenOS official website [15]

For qemu-qemu testing, run two instances of qemu connected by QEMU socket networking. For the proper functionality the each instance must have different MAC address assigned

```
qemu -net nic,model=rtl8139 -net socket,listen=:1234 -cdrom image.iso
```

```
qemu -net nic,model=rtl8139,macaddr=52:54:00:12:12:12
    -net socket,connect=:1234 -cdrom image.iso
```

> **Note**
>
> there is no necessity for running the same controller on both instances, better attitude for testing is using some controller with already tested driver on one side and the new controller on the other side.

Running tests

Manual testing

The first simple testing can be provided by `dump` and `send`. Run

```
nictest eth0 dump
```

in the one instance and

```
nictest eth0 send
```

in second instance.

The packets received by the first instance will be dumped by nictest. There is possibility to use `nictest eth0 send to ADDR` for specify address (e.g. to test unicast/multicast packets reception, reception of packets sent to another address), the broadcast address is used by default.

The manual testing is also good for testing different polling modes - you run `dump` at the tested instance, send few packets and wait for reception (in the case of periodic polling modes) or force polling device by

```
nictest eth0 poll
```

in **on demand** polling mode

[15] http://trac.helenos.org/wiki/NetworkBridging

Automatic tests

The main profit from the nictest application is in its ability to provide automatic test. There is `master` and `slave` instance of nictest, the `slave` is the side with the controller to be tested. To invoke the master run

```
nictest eth0 test_master $test
```

and to invoke

```
nictest eth0 test_slave
```

The test can be e.g. `filters` test to test reception rules changing, `states` or `stress` test for testing driver stability (lots of packets are sent), one test invoking all other can be invoked by

```
nictest eth0 test_master all
```

The complete `nictest` documentation can be found in Section 6.2, “NIC testing tool - nictest”

Note

While testing on real hardware we had expirienced frozen reception after extreme amount of packets. This is probably issue somewhere else in HelenOS and we expect it resolve after merge with mainline version.

10. User documentation

The HelNet project improved the networking capabilities of the HelenOS operating system by implementation of the framework for NIC controller development, DMA support for the devices and NIC related tools.

10.1. Supported platforms

The HelenOS subsystems developed by the HelNet project are supported on IA-32 and AMD-64 platform, running in QEMU emulator [16] or on the real hardware - every current computer should be usable. The supported network interface controllers are **RTL8139** in all versions and 8254x family of **Intel E1000** network interface controller.

10.2. CD-ROM content

The CD-ROM contains the electronic version of the documentation, system source files and images compiled with QEMU configuration. The directory structure is

`README`	The README file
`helnet.pdf`	The electronic version of this documentation
`html/`	Doxygen cross reference root directory
`html/index.html`	Doxygen cross reference main file
`img/image_ia32_ip.iso`	IA32 cdrom image for QEMU, **ip** as the IL module
`img/image_ia32_ildummy.iso`	IA32 cdrom image for QEMU, **ildummy** as the IL module
`img/image_amd64_ip.iso`	AMD64 cdrom image for QEMU, **ip** as the IL module
`img/image_amd64_ildummy.iso`	AMD64 cdrom image for QEMU, **ildummy** as the IL module
`src/`	The root of project source files

10.3. System compilation

The precompiled system images for QEMU settings are placed on the CD-ROM, if the specific system setting must be done (e.g. for the real hardware) the compilation from the source files is the only way.

Source files

The source files are located in the `src` directory on the CD-ROM, optionally the most actual version can be obtained from the Bazaar [17] repository by

```
bzr co http://bazaar.launchpad.net/~helenos-nicf/helenos/nicf src
```

System configuration

The specific configuration is needed only for compilation for the real hardware or during the NIC controller testing compilation.

[16] http://qemu.org
[17] http://bazaar.canonical.com

The NIC configuration is provided by configuration files placed in `uspace/srv/net/cfg` directory. Each file contains the interface setting related to the specific hardware path depending on the hardware placement (like PCI slot). The new configuration file with the name related with hardware path should be added, the simplest way is to modify existing file and update its setting. The only setting to change is are the lines:

```
HWPATH=/hardware/path/of/the/device
IL=ip
```

The hardware paths of all recognized devices in the system can be obtained by

```
ls /dev/devices
```

invoked inside the running HelenOS instance.

The IL can be **ip** for the IPv4 related infrastructure running or **ildummy** for the network card testing infrastructure support. If the **ip** module is used, the setting of the IPv4 configuration variables - address by `IP_ADDR`, network mask by `IP_MASK` and network gateway by `IP_GATEWAY`.

Compilation

To compile the system run

```
make
```

from the root source directory. Select `amd64` or `ia32` from the `Load preconfigured defaults` menu, disable `Support for SMP`, change the settings specific to target computer and press `Done` to confirm the settings.

> **Note**
>
> The SMP support can remain enabled but the controller will be functional only with **software periodic** polling mode in multiprocessor environment due the APIC driver limitations.

> **Note**
>
> You need specific version of cross compiler. You can install it by toolchain build script `tools/toolchain.sh`.

10.4. Running the system in QEMU

To run the system in the QEMU run

```
qemu -net nic,model=MODEL -net user -cdrom IMAGE
```

The MODE can be either `rtl8139` or `e1000`, the IMAGE is the .iso image file obtained from the CD-ROM or as the compilation result. The system is expected to use **ip** layer. To test network controller by ping the default QEMU gateway by running

```
ping 10.0.2.2
```

inside the HelenOS.

For the NIC tests provided by `nictest` utility, used the image compiled with the **ildummy** configuration. The two instances of the QEMU should be used, the more detailed information about can be found in Section 9.3, “Nictest”.

10.5. Nicconf utility

After the system start you can run

```
nicconf
```

to obtain list of available network interface. The nicconf help can be invoked by `nicconf -h` command, detailed description including all arguments is in Section 6.1, “NIC configuration utility - nicconf”.

10.6. Nictest

When running two instances of HelenOS with **ildummy** configuration, the driver testing by **nicconf** utility can be provided. The testing using `nictest` is described in Section 9.3, “Nictest”, the detailed description of the `nictest` utility is in the Section 6.2, “NIC testing tool - nictest”.

11. Future development

11.1. More drivers

Having full scale of NIC drivers as in other operating systems would be very nice. The two well-developed drivers provided within this project enable further development of HelenOS's networking abilities and also may serve as reference implementations of drivers. Drivers for other NICs should come upon the necessity to control hardware where these are installed, the development should be straightforward now.

11.2. DMA framework future development

DMA framework is functional on IA32 and AMD64. But in can be extended to other platforms supported by HelenOS. Next DMA improvement is related to AGP/PCI-X support adding. These technologies can enable pseudoIOMMU operations, which can translate bus address to physical memory address. Today is bus address same as physical memory address. But support of this feature is above requests to this project.

Another possibility of development can be implementation of drivers of busmaster devices. These devices are not required on modern Intel architecture systems, but some other systems could require them.

11.3. Support for multiport NICs

Although we have considered the existence of multi-port NICs as no device of this kind is supported in Qemu and we had no access to a real HW, we have mostly ignored them. The NICF would require some changes in order to support them.

11.4. Power management

HelenOS currently does not use any power management. NICs possibilities to conserve power were not exposed, because the ability to do so should be implemented in different interface than the NIC interface. Solving the design of power management was out of scope of this project.

11.5. Removable NICs

The NICs are usually situated on firm bus such as PCI or ISA. However, there are NICs that can be inserted on the fly to USB ports or PCMCIA ports. Neither DDF nor NICF and network stack anticipate that the NIC could be removed from the system.

A. NIC Interface

The interface consists of few mandatory methods, which must be implemented in the driver (otherwise the driver cannot operate in any way). Other methods are optional.

Table A.1. Mandatory methods

Interface method	Description
`connect_to_nil`	Assigns the device its `device_id` - identifier which will be used in higher layers of network stack - and requests connection to specified **NIL** module.
`get_address`	Queries device's MAC address.
`get_state`	Queries device's state - this can be either **active**, **down** or **stopped**.
`set_state`	Sets device's state.
`send_message`	Requests the device to send a frame with specified ID to the network.

Following methods provide useful information about device:

Table A.2. Informational methods

Interface method	Description
`get_stats`	Queries device's statistics: number of sent and received frames, errors encountered etc.
`get_device_info`	Queries static information about the NIC - the result should be the same for all calls on the same device.
`get_cable_state`	Queries whether the cable is currently plugged in.

The next set contains methods controlling operation mode:

Table A.3. Operation mode control methods

Interface method	Description
`set_address`	Sets NIC's MAC address. Although the ability to query the address is mandatory, setting MAC address does not need to be implemented (on some cards it is even not possible).
`get_operation_mode`	Sets device's operation speed (usually 10/100/1000 Mbps), mode (full/half duplex) and in case of gigabit Ethernet the role.
`set_operation_mode`	Sets the operation speed, mode and role (disables autonegotiation).
`autoneg_enable`	Enables the autonegotiation, possibly limiting some options in the advertisement.
`autoneg_disable`	Disables the autonegotiation (the current operation mode will be preserved).
`autoneg_probe`	Probes which modes are we and the second party really advertising.
`autoneg_restart`	Forces the autonegotiation to be performed again from scratch.

Interface method	Description
`get_pause`	Queries the currently set (or autonegotiated) state of flow-control mechanism.
`set_pause`	Limits the flow-control mechanism of PAUSE frame.

Methods in following set control which frames will be accepted and which should not be received.

Table A.4. Filtering methods

Interface method	Description
`unicast_get_mode`	Queries which unicast (physical) MAC addresses in frame's destination field are to perceived as "our" (therefore accepting those frames).
`unicast_set_mode`	Requests accepting frames with particular unicast (physical) MAC addresses in the destination field.
`multicast_get_mode`	Queries which multicast addresses are accepted in frame's destination field.
`multicast_set_mode`	Requests accepting frames with particular multicast addresses in the destination field.
`broadcast_get_mode`	Queries whether broadcast frames are received or discarded.
`broadcast_set_mode`	Sets whether broadcast frames are received or discarded.
`defective_get_mode`	Queries which types of defective packets (too long, to short, invalid CRC…) are accepted.
`defective_set_mode`	Sets which types of defective packets should be accepted.
`blocked_sources_get`	Queries which MAC addresses in frame's source field cause the packet to be refused (blacklisting).
`blocked_sources_set`	Sets which MAC addresses in frame's source field cause the packet to be refused (blacklisting).

Some cards allow filtering according to VLAN tags in the frames and even automatic tagging and untagging.

Table A.5. VLAN methods

Interface method	Description
`vlan_get_mask`	Queries for VLAN tags filtering mask.
`vlan_set_mask`	Sets the VLAN tags filtering mask. This mask has one bit for each of 4096 possible VLAN tags determining whether the frame should be accepted or refused.
`vlan_set_tag`	Sets a single VLAN tag that should be automatically added and/or stripped from the packet.

Many PCI NICs have the possibility to boot the computer even if it is turned off upon receiving a special frame. The conditions can be rather complicated - each way how to wake up the computer is so called a virtue.

Table A.6. WoL methods

Interface method	Description
`wol_virtue_add`	Add a new virtue, specified by arguments subject to the type.
`wol_virtue_remove`	Remove a virtue with specified ID.
`wol_virtue_probe`	Query the type and arguments of existing virtue with specified ID.
`wol_virtue_list`	Query a list of IDs from virtues with specified type or a list of all registered virtues.
`wol_virtue_get_caps`	Query how many virtues of this type can be yet added.
`wol_load_info`	After the computer was activated by WoL, query the type of virtue and exact frame that caused the wakeup.

High speed NICs help the network stack with various automatic checksums (IP checksums, TCP checksums…). The result can then be stored in a single bit in packet's meta information and queried in appropriate module.

Table A.7. Offload computing methods

Interface method	Description
`offload_probe`	Query which offload options are supported and which offload computations are currently performed.
`offload_set`	Set which offload computations should be performed.

Interrupts can cause significant overhead - therefore there is a possibility to poll the device manually or periodically rather than fire an interrupt upon receiving each single frame.

Table A.8. Polling methods

Interface method	Description
`poll_get_mode`	Query the current polling mode and eventually the period of polling.
`poll_set_mode`	Sets the polling mode and eventually the period of polling
`poll_now`	Forces NIC to check the status, receive frames etc.

B. NICF Default handlers summary

Table B.1. DDF interface handlers

Interface callback	Requirements
`device_added`	none
`open`	none
`close`	none

Table B.2. General NICF handlers

Interface callback	Requirements
`set_state`	setting `on_stopping()`, `on_activating()` and `on_going_down()` callbacks by `nic_set_state_change_handlers()` or keeping the device in **active** state all the time
`get_state`	state handling by `set_state()` default implementation
`send_message`	setting `write_packet()` callback by `nic_set_write_packet_handler()`
`connect_to_nil`	none
`get_address`	reporting device address by `nic_report_address()` during the device initialization and `set_address()` handling
`get_stats`	updating statistics by **`libnic`** interface
`poll_get_mode`	using default `poll_set_mode()` handler
`poll_now`	setting `on_poll_request()` callback by `nic_set_poll_handlers()`, using default `poll_set_mode()` handler
`poll_set_mode`	setting `on_poll_mode_change()` callback by `nic_set_poll_handlers()`, using default `poll_now()` handler, implementing **on demand** and **immediate** polling modes in `on_poll_mode_change()` callback

Table B.3. Filtering interface handlers

Interface callback	Requirements
`unicast_get_mode`	using `unicast_set_mode()` default handler
`unicast_set_mode`	setting `on_unicast_mode_change()` by `nic_set_filtering_change_handlers()` callback reporting filtering precision by `nic_report_hw_filtering()` in the handler reporting MAC address by `nic_report_address()` passing packets to higher modules by **`libnic`** interface
`multicast_get_mode`	using `multicast_set_mode()` default handler
`multicast_set_mode`	setting `on_multicast_mode_change()` by `nic_set_filtering_change_handlers()` callback reporting filtering precision by `nic_report_hw_filtering()` in the handler

Interface callback	Requirements
	passing packets to higher modules by **libnic** interface
broadcast_get_mode	using broadcast_set_mode() default handler
broadcast_set_mode	setting on_broadcast_mode_change() callback by nic_set_filtering_change_handlers() passing packets to higher modules by **libnic** interface
blocked_sources_get	using blocked_sources_set() default handler
blocked_sources_set	setting on_blocked_sources_change() callback by nic_set_filtering_change_handlers() (optional) passing packets to higher modules by **libnic** interface
vlan_get_mask	using vlan_set_mask() default handler
vlan_set_mask	setting on_vlan_mask_change() callback by nic_set_filtering_change_handlers() (optional) passing packets to higher modules by **libnic** interface

Table B.4. WoL interface handlers

Interface callback	Requirements
wol_virtue_add	setting on_wol_virtue_add() callback by nic_set_wol_virtue_change_handlers() proper setting and updating of maximal WoL capabilities by nic_set_wol_max_caps()
wol_virtue_remove	setting on_wol_virtue_remove() callback by nic_set_wol_virtue_change_handlers()
wol_virtue_probe	using default wol_virtue_add() and wol_virtue_remove() handlers
wol_virtue_list	using default wol_virtue_add() and wol_virtue_remove() handlers
wol_virtue_get_caps	managing capabilities by default wol_virtue_add/remove() callbacks

C. Project timeline

November 2010	Beginning of work, studying of HelenOS and hardware started
December 2010	First team meeting, work divided to members, set rules of work. Conversion of NE2000 and loopback drivers to use DDF. Start writing RTL8139 driver, MAC address read from the device.
January 2011	First drafts of physical memory allocator. First packet transmitted by RTL8139 and E1000. Logging service added.
February 2011	Advanced NIC framework functionalities. Manual nictest operation. Limited packet reception on RTL8139, MAC address setting, ping functional. DMA allocator allocates memory with requested size, added direct_backend.
10Th February 2011	Official project start
March 2011	Automatic address space searching in DMA framework, NICF driver lifecycle defined, E1000 demonstrator is able to receive packet. Full packet reception on RTL8139, autonegotiation, filtering work started
April 2011	scatter/gather memory page locking designed. Near-to-final version of NIC interface. RTL8139 filtering finished. E1000 is transmitting an receiving multiple packets.
May 2011	State polling in RTL8139 driver, work at page locking. E1000 works on AMD64.
June 2011	Work on polling mode, locking pages finished. E1000 is filtering.
July 2011	Automatic testing in nictest. Rapid testing (QEMU + real hardware), bug hunting and reparations E1000 works on real hardware, autonegotiation. Documentation started.
August 2011	Packet reception rewritten in RTL8139. Improved packet accepting, DMA server finished tested and debugged. Real hardware tests, designed DMA controller interface. VLAN support in E1000 driver. Documentation finished.
1St September 2011	Project delivered

D. Team members and work distribution

Zdenek Bouska *zdenek.bouska@gmail.com*	e1000 driver web services administration (team wiki and bug tracker)
Jiri Michalec *michy@tiscali.cz*	rtl8139 driver code quality checks
Radim Vansa *radim.vansa@matfyz.cz*	NIC framework + supporting tools team management
Jan Zaloha *jzaloha@centrum.cz*	DMA subsystem team meeting summaries