ESPRIT / HPCN
PROJECT 29737 – HPGIN

High Performance Gigabit I_2O Networking Software

Specification of the Software Package D
HPGIN-Linux / Task D1

University of Helsinki
Department of Computer Science

Date: 15.01.2001
Document-Id: UHEL.15.01.04-DR-D1

ESPRIT / HPCN
PROJECT 29737 – HPGIN
High Performance Gigabit I_2O Networking Software

Specification of the Software Package D
HPGIN-Linux / Task D1

Written by:	Auvo Häkkinen, Juha Sievänen
Organization:	University of Helsinki, Department of Computer Science
Date:	15.01.2001
Delivered by:	**Frank Hohmann**
Organization:	SysKonnect GmbH
Date:	15.01.2001
Document-Id:	UHEL.15.01.04-DR-D1

TABLE OF CONTENTS

LIST OF FIGURES AND TABLES

1 INTRODUCTION

1.1 Description of the deliverable

This document is the updated version (version 4) of the *Specification of the Software Package D.* The previous ones were dated 09.03.1999, 01.10.1999 and 30.3.2000. The document contains the description of those parts of the Linux I_2O environment that was implemented by the University of Helsinki in the HPGIN-project. The Block Class OSM, SCSI Class OSM, event handling and the dynamic LCT update were out of the scope of the HPGIN-project, but have been implemented by other developers. They are not described in this document.

This document has been submitted in draft to all contractors and has been approved. The contents of this document are applicable to the partners of the consortium of the ESPRIT/HPCN project 29737, HPGIN. The report is addressed to the EC Project Officer.

1.2 Intelligent I/O

Intelligent I/O (I_2O) is an industrial standard for high-performance I/O subsystems. It is defined and maintained by the I_2O Special Interest Group.

The *I_2O Specification* [2] defines architecture for I/O that is independent of both the specific device being controlled, and the host operating system. The specification makes it easier to implement cross-platform I/O, thus broadening availability and applicability of reliable intelligent I/O devices.

I_2O defines an approach to I/O where low-level interrupts are offloaded from the CPU to I/O processors specifically designed to handle I/O. With support for message-passing between multiple independent processors, the I_2O architecture relieves the host of interrupt-intensive I/O tasks. This improves greatly I/O performance in high-bandwidth applications such as networked video, groupware, and client/server processing.

The *I_2O Specification* [2] defines a split driver model (see Figure 1.1) for creating drivers that are portable across multiple operating systems and host platforms.

The split I_2O drivers are composed of two parts: the Operating System Service Module (OSM), which resides on and interfaces to the host OS, and the Device Driver Module (DDM), which resides on and interfaces with the adapter to be managed by the driver.

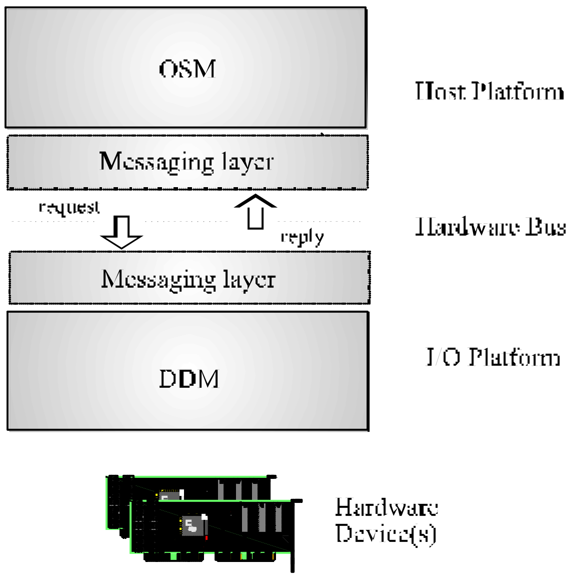

Figure 1.1 Split driver model.

These modules interface with each other through a message-passing system based on shared memory areas. The OS I/O requests are converted in OSMs into specific I_2O messages and are passed through the Messaging layers to the DDM. Requests are dispatched to DDMs that process them. DDMs generate replies to be delivered back to the originators of the requests.

Split driver model decreases significantly the number of drivers required. OS vendors write a single I_2O-ready driver for each class of device – such as LAN adapter - and device manufacturers write a single I_2O-ready driver for each device, which will work for any OS that supports I_2O.

The I_2O model can be applied in single-processor, multiprocessor and clustered-processor systems, as well as desktop, communications, and real-time system environments.

I_2O is the basis for driver standardization, system performance enhancement, resource sharing, clustering, and distributed heterogeneous systems. It will be supported by the Standard Network Operating Systems as well as by specific Real Time Operating

Systems. It is expected that I_2O will change the design of I/O subsystems dramatically within the coming years.

1.3 The Objectives of the HPGIN-Linux

The HPGIN project is aimed to develop I_2O communication layer software for high performance network devices, supporting version 1.5 of the *I_2O Specification* [2]. Results to be achieved within the scope of the HPGIN-project include

- implementation of I_2O communication layers for gigabit LAN I/O subsystems,
- implementation of appropriate OS specific I_2O communication layers for a Standard Network Operating System (Linux) and
- implementation of appropriate OS specific I_2O communication layers for a high performance Real Time Operating System (Virtuoso).

The objective of the HPGIN-Linux inside this project is to implement the common parts of the I_2O execution environment into Linux operating system. This includes the implementation of the I_2O message passing layer and I_2O resource management, which establish the base for the adapter class dependent OSMs.

The implementation of an OS Service Module for the LAN adapter class is also within the scope of the project. The functionality of the implemented I_2O support can be verified by combining an I_2O supporting Linux system with an I_2O aware network interface (e.g. a hardware platform running the embedded software package developed within the project).

Both the HPGIN-project and its objectives are described in detail in the *HPGIN Project Programme* [1].

1.4 Overview of the document

Section 2 describes briefly the I_2O communication model and illustrates flow of the I/O operations in the I_2O execution environment. The basic requirements for Intelligent I/O are listed in section 3. Section 4 deals with resource management and its basic data structures. The design of the Linux I_2O subsystem is introduced in section 5, and the I_2O modules (Pci, Core, OSMs) are explained in sections 6, 7 and 7.4.2. Configuration interface, which implements a controlled mechanism for a human operator is described in section 9. Its counterpart, configuration information via proc file system, is described in section 10. Error handling is discussed in section 11. The previous version of this document contained also a section explaining testing. It has now been moved into a separate document titled *Test Specification Plan*.

Since the shell interface (the host OS / IOP API) is described precisely in the *I_2O Specification* [2], a lot of references back to it is used. You should have the specification on hands when you are reading this document.

1.5 References

This specification refers to the following documents

[1] *EP 29737 – ESPRIT/HPGIN Cost Reimbursement Contract, Annex I – Project Programme*, September 1998.

[2] I_2O Special Interest Group: *Intelligent I/O (I_2O) Architecture Specification Version 1.5*, March 1997.

[3] Beck, Böhme, Dziadzka, Kunitz, Magnus, Verworner: *Linux Kernel Internals.* 2nd ed., Addison-Wesley, 1998.

[4] I_2O Special Interest Group*: Intelligent I/O (I_2O) Shell Up (OSM) Compliance Test Specification Version 1.5,* August 1998.

[5] PCI Special Interest Group. *PCI Local Bus Specification*, Rev. 2.1, June 1995.

[6] University of Helsinki, EP 29737 – ESPRIT/HPGIN, Task E1. *HPGIN-TEST HPGIN-Linux Test Specification*. September 1999.

1.6 Definition of Terms

This specification uses the following terms

Glossary entry	**Entry definition**
DDM – device driver module	DDMs are the lowest level in the I_2O architecture, and are directly responsible for control and data transfer of the hardware device, such as a network connection and storage device
Hrt – hardware resource table	A list of adapters and their configuration information, including the identity of the controlling DDM. HRT tells the Host of any adapters the IOP controls, and thus that the Host should not touch, as well as adapters the IOP can control.
Host	Host is composed of one or more application processors and their associated resources. Host execute a single homogenous OS and is dedicated to process applications. The Host is responsible for configuring and initializing the IOP into the system.

I_2O – intelligent input/output device driver model	An open, standards-based split driver I/O model which provides device driver portability across multiple OSs, processor and bus technology independence, and support for intelligent, message passing I/O subsystems.
Inbound queue	A message queue of a particular I/O platform that receives messages from Host or from another IOP.
IOP - I/O platform	A platform consisting of a processor, memory, I/O, adapters and I/O devices. They are managed independently from other processors within the system.
IRTOS – I_2O Real-Time OS	A special purpose real-time OS for the IOP to support high-speed, low-overhead I/O operations.
Lct – logical configuration table	A list of logical devices whose service is abstracted by the IOP (through a DDM). The Host and other IOPs query this table about available resources.
MFA – message frame address	The address of an I_2O message buffer residing on Inbound queue or in Outbound queue.
Message layer	The message layer provides the communication and queuing model between service modules. The messages passed are in OS-neutral format.
Messenger	The messaging layer running on a particular platform, initializing, configuring, and operating its client modules. Each processor or SMP group has a single Messenger. Each IOP has a Messenger.
OSM - operating system service module	A driver module that provides the interface between the Host OS and the I_2O Message Layer. It represents the portion of the driver that forwards Host OS requests to a DDM for processing.
Outbound queue	A message queue for a specific IOP for posting messages to the local host, in lieu of the Host's Inbound queue.
PCI - peripheral component interconnect	An industry standard for a high-performance expansion bus. PCI supports bus concurrency, auto-configuration and multiple bus masters.
PDB - packet description block	LAN DDM describes each bucket it consumes, the bucket's order, and the location and length of each packet in the bucket, by building a PDB. Is part of a ***LanReceivePost*** reply.
SGL - scatter-gather list	A structured list of memory addresses that specifies data buffers and their respective lengths.
SMP – symmetric multi processing	A multiprocessor environment where all processors share the same main memory and the same I/O subsystem.
SysTab - system configuration table	A table build by the Host that informs the I/O platform of the existence and addresses of other IOPs.
TID – target id	Logical address of a service registered with the message layer. The target ID is the address the message layer uses to deliver requests to a service mo dule.
TRL – transaction reply list	A structured list of replies to I_2O messages containing transaction contexts and transaction details.

1.7 Conventions

The conventions used in this document are presented below.

Text	Description	Example
italics	Reference to document	*I_2O Specification*
Bold Italics	I_2O message names	***ExecStatusGet***
`Courier`	Function names, field of a message	`i2o_lan_receive_post()`, `InitiatorContext`

The basic principle of naming the functions is presented below. In some cases one or more parts may be omitted.

`result i2o_{class}_{verb}_{noun}(Parameters)`	
`class`	is the I_2O base class of the object
`verb`	is the function to perform on that object
`noun`	is the abbreviation for the name of the object

Example:

`int i2o_lan_register_device(struct i2o_device *i2o_dev)`	
`lan`	identifies the base class of the message
`register`	describes the operation
`device`	identifies the target of the operation

2 I_2O EXECUTION ENVIRONMENT

This section gives an introduction to the basics of the I_2O execution environment and describes the flow of the I/O operations. This description focuses only to the host OS's view of the system. For a more detailed description refer to the *I_2O Specification* [2].

2.1 Communication model

The system execution environment is outlined in Figure 2.1.

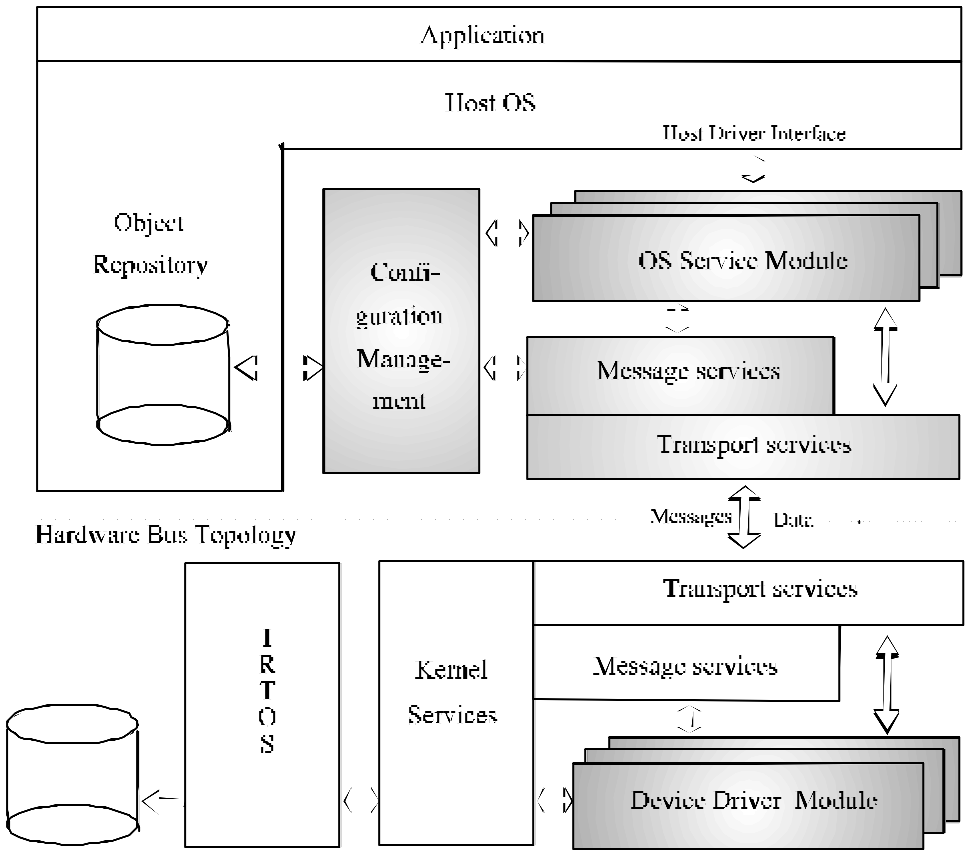

Figure 2.1. I_2O execution environment.

The host OS provides a number of OSMs and the communication service (so called Messenger). In addition to the message transport function, the OS provides

- Executive functions that initialize and maintain the I_2O system.
- System resource management that configures and maintains I_2O system.
- Configuration functions that provide the user interface, file system access, and configuration dialogue with an IOP and its DDMs. These functions enable installing and configuring IOPs and their DDMs.

The communication model used by the I_2O architecture is a message-passing protocol. The communication service provides message transport service for OSMs and DDMs. Each message contains a header and a payload. Message header format is constant for all messages and provides the return address to the originator. The format for the payload varies between messages and is established by the function type value in the header.

Each device is a virtual interface for a particular class of I/O messages. A TID (target id) identifies a device and, thus, an instance of a device class specific interface. The IOP administers TIDs when a device is first created, and the TID acts as the local address of the device. Responses are addressed to the initiator of the request.

In addition, the OS provides ability to install OSMs produced by third-party vendors. The OSM interface provides the ability to query the Messenger for the list of IOPs and their registered devices (i.e., logical configuration table information) and the ability to send request and receive replies.

2.2 Flow of I/O operations

Figure 2.2 illustrates the flow of the I/O operations in I_2O execution environment. The following text describes the events (the numbers corresponds to the steps in
Figure 2.2).

1. The host OS issues an I/O request.
2. The OSM accepts the request, reads a message frame address (MFA) from the IOP inbound queue and translates the request into an I_2O message addressed to a DDM. The `InitiatorContext` field is set to indicate the message handler for the reply. The OSM has the option to place a pointer to the OS I/O request in the message's `TransactionContext` field.
3. The host's Messenger queues the message by writing the MFA to IOP’s inbound queue.

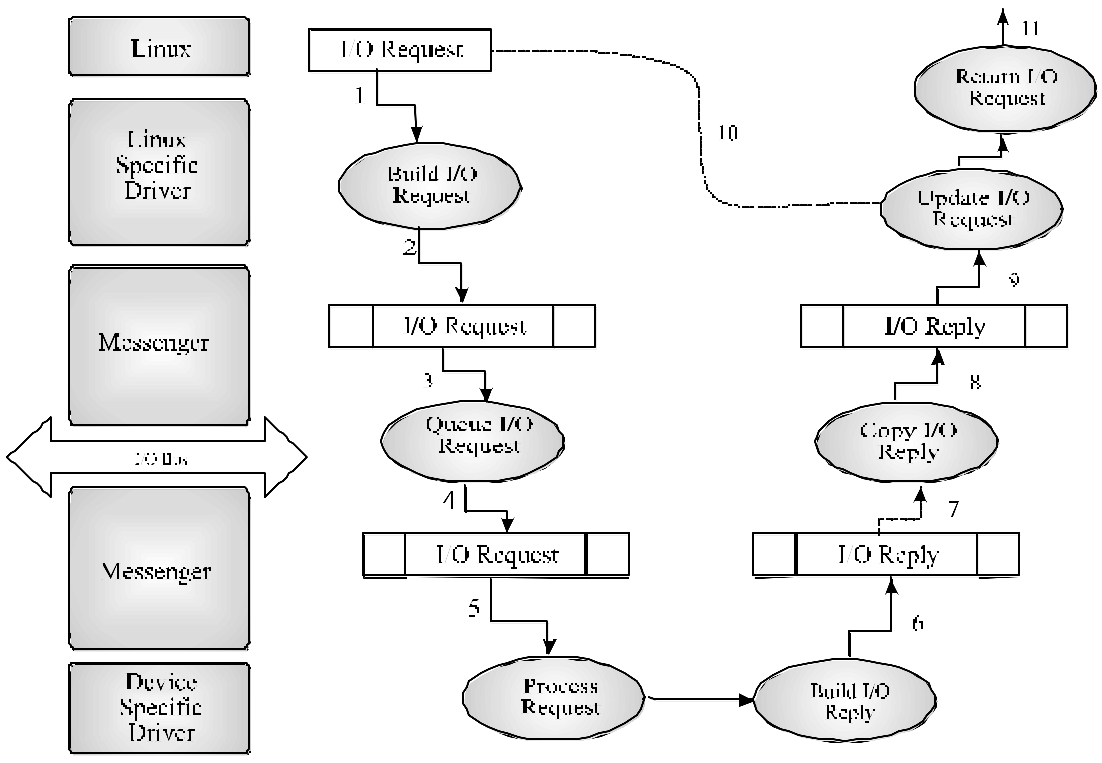

Figure 2.2 Flow of I/O operations.

4. The IOP's Messenger reads the MFA from the inbound queue, and posts the message to the DDM's event queue.
5. The DDM processes the request.
6. The DDM builds a reply, copies the `InitiatorContext` and `TransactionContext` fields from the request to the reply, addresses the reply to the Initiator, and finally invokes the IOP's Messenger.
7. The IOP's Messenger queues the reply by copying it into an outbound queue message frame residing at the host's Messenger.
8. The IOP alerts the host's Messenger via an interrupt. The control is moved to interrupt handler. The host's Messenger reads the reply from the IOP's outbound queue, copies it into a local buffer, and frees the message frame by writing the MFA to the IOP's outbound queue.
9. The host's Messenger inspects message's `InitiatorContext` field and invokes the OSM's me ssage handler with the reply.
10. The OSM retrieves the pointer to the OS‘s I/O request from the message's `TransactionContext` field to establish the original request context and completes the I/O request.
11. The Driver returns the request to the OS.

3 DESIGN ARCHITECTURE DESCRIPTION

This section describes the design principles of the Linux I_2O environment, and the basic requirements for the host/IOP message passing.

3.1 Design Principles

The main target architecture in this project is Intel x86. It is the most common environment for Linux, and the Linux environment of the Department of Computer Science at University of Helsinki is composed solely of Intel x86 based Linux machines. The portability issues will be considered, although we are not able to test on all Linux platforms (like Alpha, MIPS, Sparc, M68000, and PowerPC). A lot of the portability issues are already handled in Linux kernel at the source level. Internal data structures can be tailored at the source level to match the hardware requirements as closely as necessary. The compiler is also free to optimize away operations not needed on a specific architecture.

The implementation is started on Linux version 2.2. The later modifications to the Linux kernel will be tracked during the project.

The SMP (Symmetric Multi Processing) needs also to be considered. This means that the critical data structures must be locked when they are processed.

Linux is based on monolithic kernel, and the base of the implementation may be compiled straight into the kernel. The I_2O adapter class dependant modules (OSMs) will be programmed as separate dynamically loadable modules. Therefore only those OSMs needed in a specific environment need to be loaded to memory.

3.2 Basic Requirements

The following subsections list the basic requirements for the host platform and for the I/O platform. Most of the cases listed (e.g. booting, shared memory and cache) are implemented completely on the hardware level (or firmware level) and don’t need any attention from the OS level. Some of the cases have to be considered on the software level, e.g. address size and byte order, locations and usage of the message queues.

3.2.1 Boot

In a system with I_2O compliant OS, the BIOS (or its extensions) does not need to be I_2O aware unless it boots the OS from an I_2O device. However, BIOS that is I_2O aware allows an OS that is not I_2O aware to access an I_2O device. In this instance the BIOS (or its extensions) must abstract the I_2O subsystem to the OS and provide I_2O functionality via its normal BIOS function calls.

3.2.2 I_2O Subsystem

A hardware, or a system vendor supplying an I_2O adapter (e.g. an intelligent adapter card providing both the IOP and embedded controllers), without supporting third party DDMs, must adhere to the requirements for the shell interface (chapter 4) and the message requirements in chapters 3 and 6 of the *I_2O Specification* [2]. Although the device need not implement the core interface (chapter 5) it must function externally as if it does. When responding to an installation or load request for a DDM, the IOP can reject the request, reporting function not supported.

A vendor supplying an I_2O subsystem (e.g. an IOP on the motherboard) that can support third party DDMs must also adhere to the core interface (chapter 5 of the *I_2O Specification* [2]). Features that differentiate between designs include the amount of non-volatile memory for storing third party DDMs, as well as the physical expansion bus capability.

3.2.3 Shared Memory

I_2O message passing is based on shared memory. IOPs must have access to shared system memory for the hardware level queuing model. Each IOP must provide its own units for receiving messages from the host and other IOPs and for queuing messages to the host. At a minimum, a region of memory accessible via system bus contains the inbound message frames where the host and the IOPs deposit their messages.

Memory allocated as shared system memory must be cache coherent. That is, it should not be cached unless the processor and memory controller support cache-coherent protocols. Efficient memory coherency support is required if shared memory writes involve a write-to-cache, versus a write-through or copy-back. If so, an efficient mechanism to flush modified cache lines must be provided.

3.2.4 System Bus

Because PCI is the predominant bus in new server designs, the version 1.5 of the *I_2O Specification* [2] focuses on current PCI bus specification, and describes functional interfaces based only on the current PCI bus specification (refer to *PCI Specification* [5] for details). This does not preclude other bus types, but defining extensions for other bus types is left out until support comes necessary. The possible new coming bus types will be prepared in the design of the Linux I_2O implementation.

The host identifies and locates an IOP by its PCI class code. The class code has three fields: base class, subclass, and programming interface. Locations of the inbound and the outbound queue are specified by the programming interface. The data passed through the queues are either free message frames or posted messages. Queues are accessed through two port locations in the PCI address space. The inbound queue port is at memory offset 0x40, and the outbound queue is at memory offset 0x44 in the PCI address space.

3.2.5 Address size

Three domains affect address size: the OS, the I/O subsystem, and the IOP. The version 1.5 of the *I_2O Specification* [2] specifies operation for 32-bit IOP physical addressing, 32-bit I/O subsystem operation, and both 32-bit and 64-bit OS operation. The OS address size relates to the size of the `MessageContext` fields. The `MessageVersion` field in the message header supports future capabilities, such as 64-bit physical addressing. Critical messages for initializing the IOP are address-size generic, allowing the OS to appropriately instate the IOP into the system.

3.2.6 Byte order

The version 1.5 of the *I_2O Specification* [2] discusses operation for little endian addressing only. The `MessageVersion` field supports future capabilities, such as big endian messages.

4 RESOURCE MANAGEMENT AND DATA STRUCTURES

The I_2O resource management is distributed among distinct IOPs. Each IOP has its own *Configuration Status Block*, *Hardware Resource Table* (Hrt) and *Logical Configuration Table* (Lct). During the initialization the host reads the configuration status block and the Hrt from each IOP, and builds a global *System Table* (SysTab). The SysTab is posted to each IOP, so they get information about other IOPs and their devices in the system. Based on the information given in the SysTab, each IOP creates its local Lct.

Any Messenger can query from any IOP its Lct to find out which services are available and how to use them. To each service is assigned a Target Id (TID), which are unique inside one IOP. All communication is performed using a TID, which is carried in each message as an initiator or a target.

This section lists the basic ideas of these tables, refer to *I_2O Specification* [2] for more detailed information, e.g. for the detailed structure of the table entries.

4.1 Configuration Status Block

The host gets IOP's configuration status by sending the ***ExecStatusGet*** message. There is no reply to this message, but the IOP writes its status block directly to the buffer specified by the host. Thus, it is possible to send this message before the IOP's outbound queue is initialized or the IOP's state is known. The status block describes the capabilities and the parameters of the IOP. These include among others the identity of the IOP, locations of the private memory, the size and the number of the inbound message frames, as well as the number of outbound message frames.

The Configuration Status Block is defined in figure 4-38 in the *I_2O Specification* [2].

4.2 Hardware Resource Table (Hrt)

The hardware resource table (Hrt) is a list of devices and their configuration information, including the identity of the controlling DDM. Each IOP builds its own Hrt during the boot (based on the permanent configuration). The host or another IOP obtains a copy of the IOP's Hrt by sending the ***ExecHrtGet*** message. Hrt tells the host and other IOPs of any devices controlled by the IOP. In general, the Hrt lists all devices and locations that the IOP controls or can control.

Hrt and its entries are defined in figures 4-19 and 4-20 in the *I_2O Specification* [2].

4.3 Logical Configuration Table (Lct)

The logical configuration table (Lct) is a list of logical devices whose service is abstracted through a DDM by the IOP. The host and other IOPs query this table about available resources by sending the ***ExecLCTNotify*** request. DDMs may send this message to the local IOP to determine when configuration is complete.

When the DDM registers a device, it provides the configuration information for the Lct entry. Each entry in the IOP's Lct contains `ClassId` and `SubClassInfo`. `ClassId` is the I_2O message class of the registered device. The structure of `SubClassInfo` is defined by each class and identifies the major capabilities of the device. The OSM uses this information when it determines which devices to query.

Lct and its entries are defined in figures 4-27 and 3-36 in the *I_2O Specification* [2].

4.4 System Table (SysTab)

The System Table (SysTab) describes the system as a set of IOPs and their message attributes. SysTab informs an IOP of the existence and addresses of other IOPs. Once the host finishes initializing IOPs (i.e. has read the configuration status and Hrt from each IOP), it builds the SysTab and sends it to each IOP in an ***ExecSysTabSet*** request. This message gives each IOP the identity (location) of the other IOPs in the system, as well as declarations of memory and I/O for private space. The private memory and I/O space declarations lets the IOP hide devices from the system and bring devices on-line after the system is configured.

SysTab and its entries are defined in figures 4-46 and 4-47 in the *I_2O Specification* [2].

4.5 Array of I_2O Controllers

In Linux IOPs' configuration information is saved into the `i2o_controllers[]`table (see Figure 4.1). Each entry in this table is a pointer to `struct i2o_controller`, which describes one IOP and collects the previous tables (status block, Hrt, Lct). Devices controlled by one IOP are linked in a separate list pointed by the `devices` field in the IOP's entry.

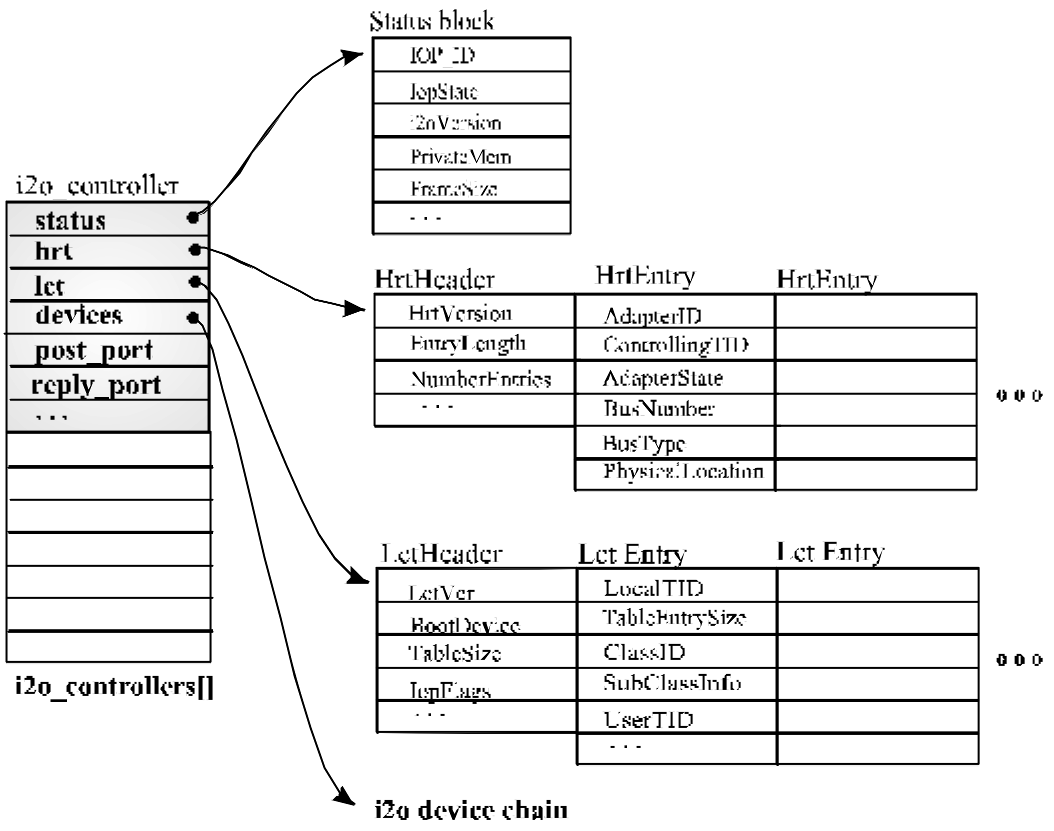

Figure 4.1 `i2o_controllers[]` table.

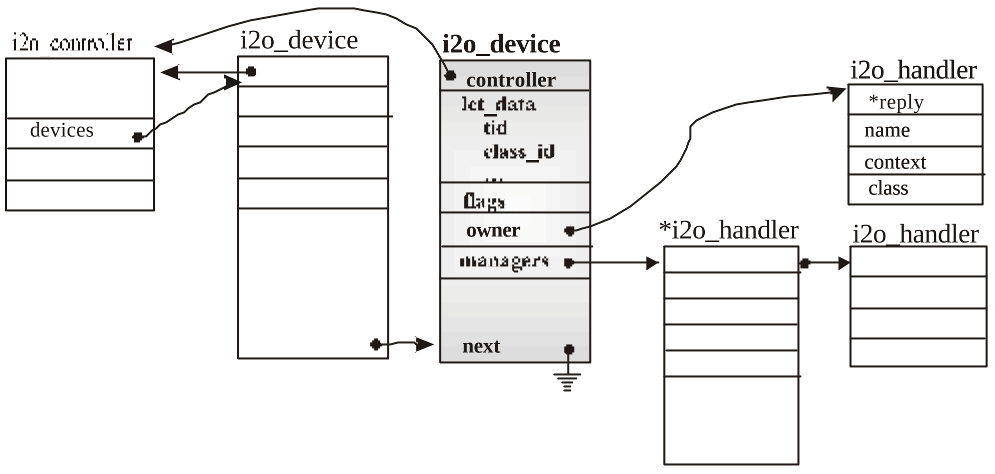

Figure 4.2 I$_2$O Device Chain.

4.6 I_2O Device Chain

The devices controlled by each IOP are linked in a list pointed by the `devices` field in the IOP's `struct i2o_controller` (see Figure 4.2). Each `i2o_device` has also a link back to the controlling IOP and to Logical Table Entry associated to this device.

Users claiming the device are registered into `struct i2o_device`. The device has only one primary user (`owner`), but it may have multiple management users (`managers`). Generally management agents do not claim devices unless they desire to change parameters (for details refer to section 6.1.3.2 in the *I_2O Specification* [2]).

The `struct i2o_device` is also linked to the conventional Linux Driver API. For network devices this is done via a pointer on the private area in the network driver API (dev->`priv->i2o_dev`). For block devices this is done through an auxiliary table by indexing with the respective unit number (`i2ob_dev[unit]->i2o_dev`). For character devices this can be done respectively.

5 THE LINUX I_2O SUBSYSTEM

The Linux I_2O subsystem consists of several modules, which are described in detail in next sections. By module we mean here loadable Linux kernel module, although the I_2O modules can be also completely compiled into the kernel. Loadable modules that belong to the Linux I_2O host environment are illustrated in Figure 5.1.

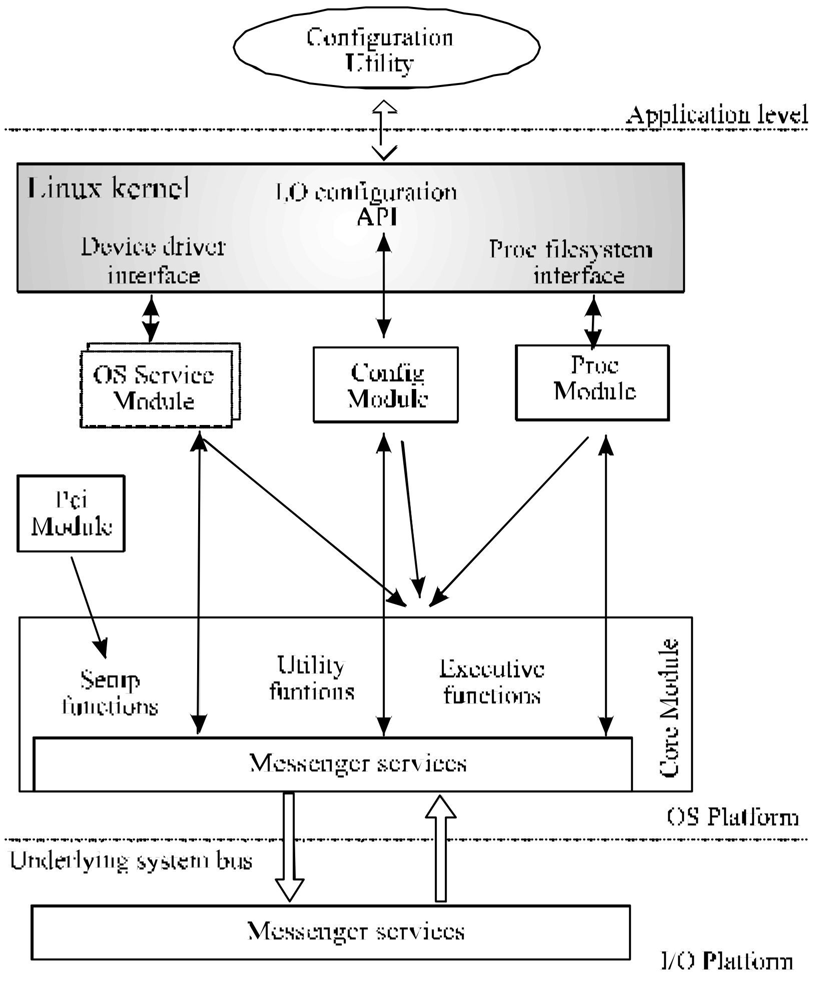

Figure 5.1. I_2O subsystem.

Pci module (`i2o_pci`) contains the PCI bus specific part of the initialization.

Core module (`i2o_core`) implements the common parts of the host I_2O environment. It includes functions to initialize and set up the system, Messenger services to send and receive I_2O messages, and Executive and Utility class functions to maintain configuration and resource management information.

OS Service Modules implement device class specific parts of the I_2O split driver model. In this project, only LAN class OSM (`i2o_lan`) is implemented. Block Storage OSM, Tape Storage OSM, SCSI OSM, and Bus Adapter OSM are out of the scope of this project.

Configuration module (`i2o_config`) implements the configuration API to install and configure IOPs and their DDMs. The Configuration Utility is the user level application to use this API.

Proc module (`i2o_proc`) implements the Linux proc file system interface to list and to set I_2O configuration information.

All these modules use Messenger services to send I_2O messages. Each module has to implement at least one handler for the replies and register that for the Messenger. The registering returns a unique context number, which is used in the `InitiatorContext` field in messages. When the reply arrives, the Messenger dispatches the reply to the right handler according to the contents of the `InitiatorContext` field. Currently only the lowest 7 bits are used for the context, the highest bits can be used for module specific purposes e.g. LAN OSM puts also device unit number into `InitiatorContext`.

The current device driver interface to Linux OS is preserved. The names and arguments to device methods are kept unchanged, so that all existing non-I_2O drivers will work as well.

Following sections describe the Linux I_2O subsystem in more details.

6 PCI MODULE

Pci module (`i2o_pci`) contains bus specific functions to find I_2O capable controllers (IOPs) on the PCI bus. Linux creates at boot time a global `pci_devices` list from the adapters on the bus. I_2O class devices (IOPs) are picked from this list and a new `struct controller` entry is added to the `i2o_controllers[]` table (see section 4.5) for each I_2O capable controller. Rests of the functions needed during the setup, i.e. functions that are not dependent of the underlying bus, are located in the Core module.

PCI bus is currently the only bus supported by the *I_2O Specification* [2]. Later, if the specification supports also other buses, this module should be updated or a new module should be implemented.

6.1 Setup functions

`int init_module(void) / void cleanup_module(void)`

These dummy functions are called by module initialization and by module cleanup.

`void __init i2o_pci_init(void)`

This function calls `i2o_pci_scan()` during the initialization if the code is compiled into the kernel (instead of using as a module). Otherwise not used.

`int __init i2o_pci_scan(void)`

This function scans kernels global `struct pci_dev` list to find I_2O class controllers from the PCI bus and to install them into Linux environment.

`int i2o_pci_core_attach(struct i2o_core_func_table *table)`

If i2o_pci module is used as a loadable module, this function calls `i2o_pci_init()`, and attaches i2o_pci module to i2o_core module, so that i2o_core module is dependent of i2o_pci module, not the other way round.

`void i2o_pci_core_detach(void)`

This function detaches the i2o_pci module from i2o_core module.

```
static int i2o_pci_bind(struct i2o_controller *iop,
                        struct i2o_device i2o_dev)
```

This function is currently just a placeholder (dummy) for the bus specific handling on I_2O initialization.

```
static int i2o_pci_unbind(struct i2o_controller *iop,
                          struct i2o_device *i2o_dev)
```

This function is currently just a placeholder (dummy) for bus specific handling on I_2O shutdown.

```
static void i2o_pci_enable(struct i2o_controller *iop)
```

This function enables PCI bus by clearing the PCI IRQ mask register and enables PCI bus.

```
static void i2o_pci_disable(struct i2o_controller *iop)
```

This function disables PCI bus by setting PCI IRQ mask register to 0xFFFFFFFF.

```
static i2o_pci_dispose(struct i2o_controller *iop)
```

This function frees the IRQ and unmaps the shared memory from the system memory.

```
int __init i2o_pci_install(struct pci_dev *dev)
```

This function creates and fills an `i2o_controller` entry for the IOP, maps shared memory area into system memory and request an IRQ for the IOP. This function calls `i2o_install_controller()` (in i2o_core module) to install i2o class devices.

```
static void i2o_pci_interrupt(int irq, void *dev_id, struct pt_regs *r)
```

This is the interrupt handler routine called by the Linux kernel. The identity of the interrupting device is passed in `(struct i2o_controller *)dev_id`. Other parameters are unused. This function calls Messengers dispatcher function `i2o_run_queue()`.

7 CORE MODULE

Core module (`i2o_core`) contains functions needed to setup, initialize and shutdown the I_2O environment, functions for message passing (Messenger), Executive and Utility class functions and functions for debugging.

The host has to adhere to the requirements specified in chapters 4 (I_2O Shell Interface Specification) and chapter 3 (Basic Requirements) of the *I_2O Specification* [2]. This means support for the complete set of Executive class messages (chapter 4.4) and Utility class messages (chapter 6.1).

7.1 Setup functions

Setup functions are used to bring the I_2O system into operational state, and to shut down the system. These includes functions to

- add / remove I_2O controller to the `i2o_controllers[]` table,
- add / remove I_2O device to I_2O device chain pointed from `struct i2o_controller` and
- initialize all IOPs found.

The I_2O initialization sequence is illustrated in chapter 4.5.1 in the *I_2O Specification* [2]. The initialization consists of two phases: first, the Status Block and the Hardware Resource table (Hrt) of each IOP are read, and the outbound queues of each IOP are initialized. In the second phase, the host creates a global System Table (SysTab) from all the status blocks and Hrts, and sends it to each IOP. Then the host reads the Logical Configuration Table (Lct) from each IOP and finishes the initialization. After that all IOPs are in OPERATIONAL state.

The I_2O messages used by the following functions are described in chapter 7.3 Executive functions.

```
int init_module(void)
```

This function is called if i2o_core is compiled as a loadable module. This function registers a handler for the replies processed in i2o_core module, attaches i2o_pci module to i2o_core module and starts the system initialization by calling i2o_sys_init(). Returns 0.

```
int __init i2o_init(void)
```

This function is called if i2o_core is compiled to kernel. This function registers a handler for the replies to be processed in i2o_core module, starts the system initialization by calling i2o_sys_init() and calls initialization routines in other I_2O modules. Returns 0.

```
static void __init i2o_sys_init(void)
```

This function runs the initialization sequence described in figure 4-56 in the *I_2O Specification* [2]. All IOPs outbound queues are initialized, their Hardware resource tables are read, System Table is created and posted to all IOPs, Logical configuration tables are read and all IOPs are enabled. When this function finishes, IOPs are in OPERATIONAL state.

```
void cleanup_module(void)
```

This function is called if i2o_core is compiled as a loadable module. This function calls i2o_sys_shutdown() to shut the I_2O subsystem, detaches i2o_pci module from i2o_core module and removes the reply handler.

```
static void i2o_sys_shutdown(void)
```

This function deletes IOPs from the i2o_controller chain. That will reset all IOP's into RESET state.

```
int i2o_install_device(struct i2o_controller *iop,
                       struct i2o_device *i2o_dev)
```

This function adds a device `*i2o_dev` into the `iop->devices` chain. Returns 0.

```
int i2o_delete_device(struct i2o_device *i2o_dev)
```

This function removes device `*i2o_dev` from the `iop->devices` list. Returns 0 (succeed), `-EBUSY` (the device is on use) or –`EINVAL` (device not found in the list).

```
int i2o_install_controller(struct i2o_controller *iop)
```

This function adds a new i2o_controller structure `*iop` to `i2o_controllers[]` table. The global counter `i2o_num_controllers` is increased. Returns 0 (succeed) or –`EBUSY` (no space left in the table).

```
int i2o_delete_controller(struct i2o_controller *iop)
```

This function deletes all devices from the `iop->devices` list, resets the `*iop` and removes the entry from the `i2o_controllers[]` table. The global counter `i2o_num_controllers` is decreased. Returns 0 (succeed), `-EBUSY` (the `*iop` is on use, the device is on use) or –`ENOENT` (`*iop` not found).

```
struct i2o_controller *i2o_find_controller(int n)
```

This function gets an iop number `n` and returns a pointer to the corresponding i2o_controller structure. The `iop->users` counter in the i2o_controller structure is increased. Returns NULL, if there is no corresponding entry.

```
void i2o_unlock_controller(struct i2o_controller *iop)
```

This function decrements the `iop->users` counter in the i2o_controller structure.

```
int i2o_activate_controller(struct i2o_controller *iop)
```

This function brings IOP into HOLD state, i.e. reads the status, initializes the outbound queue and reads the Hardware resource table. Returns 0 (succeed) or –1.

```
int i2o_online_controller(struct i2o_controller *iop)
```

This function brings IOP from HOLD state to OPERATIONAL state, i.e. sends System table to all IOPs, enables them and reads their Logical configuration tables. Returns 0 (succeed) or –1.

7.2 Messenger services

Messenger implements the functionality needed for message passing. Messenger deals with message queues, IOPs' interrupts and address translations. The basic task is to dispatch messages from the IOPs' outbound queue to the registered message handlers.

7.2.1 Message Queues

The I_2O messaging layer delivers I/O transaction messages (request and replies) from one software module to another, anywhere in the I_2O domain. The physical portion of the interface specifies a single queuing model for shared memory architectures. This queuing technique for transferring messages uses

- One inbound queue for each IOP. The inbound queue of a platform receives messages from all other platforms, including the host.
- One outbound queue for each IOP. The outbound queue of all IOPs collectively functions as the inbound queue for the host. This allows each IOP to provide hardware support for efficiently passing messages without requiring additional host hardware.

Each Messenger is running on a single platform: there is one instance per processor or Symmetric Multi Processor group. Each Messenger communicates by placing I_2O messages in the target's inbound queue. The data passed through the queues are either free messages or posted messages.

Queues are accessed through two port locations in the PCI address space (the current I_2O Specification focuses only on PCI bus). The inbound queue port is at memory offset `0x40`, and the outbound queue port is at memory offset `0x44` in the PCI address space of each IOP. Both queues consist of two FIFOs: *Free_List FIFO* and *Post_List FIFO.* When the host reads from the IOP's inbound queue port, it gets a free message frame to fill, and when it writes to the IOP's inbound queue port, it gets the frame delivered to the IOP. Similarly, when the host reads from the IOP's outbound port, it gets a message (reply) from the IOP. The message frame is released when the host writes the message frame address (MFA) to the IOP's outbound queue. If the queue is empty, address `0xFFFFFFFF` is returned.

IOP initializes its inbound queue during the boot. The host initializes IOP's outbound queue by allocating free message frames and writing their addresses into the IOP's outbound queue.

7.2.2 Address Translation

The implementation of the message queues is based on physically shared memory. IOP and its DDMs use system addresses to refer to a shared memory location when communicating with the host or other IOPs. A translation mechanism is needed to convert host's local memory address references (virtual addresses) to a system address references (physical addresses), and vice versa. The difference between the system and the local address is a constant for all shared memory, so the translation is easy.

In Linux address translations are done by functions

```
inline unsigned long virt_to_bus(volatile void * address)
inline void * bus_to_virt(unsigned long address)
```

7.2.3 Sending I_2O Messages

To send a message (I_2O request), the host reads a free message frame from the target IOP's inbound queue, fills the I_2O message header and the message payload with I_2O specific data, and finally writes the address of the frame (MFA) to the target IOP's inbound queue. Messenger functions for these purposes are `i2o_post_message()`, `i2o_post_this()` and `i2o_post_wait()`.

```
inline void i2o_post_message(struct i2o_controller *iop, u32 addr)
```

> This function writes the message frame address `addr` into the `iop`'s inbound queue.

```
int *i2o_post_this(struct i2o_controller *iop, u32 *data, int len)
```

> This function reserves an inbound queue message frame from the `*iop`, copies the message pointed by `*data` into it and posts the message. Returns 0 (succeed) or `-ETIMEDOUT` (inbound free queue empty)

```
int *i2o_post_wait(struct i2o_controller *iop, u32 *msg, int len,
                   int timeout)
```

> This function calls `i2o_post_this()` to send an I_2O request `*msg` to `*iop`. The sending process is put into wait queue maximum for `timeout` seconds to wait for a reply. Returns `I2O_POST_WAIT_OK` (success), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=`SUCCESS`).

```
static void i2o_post_wait_complete(u32 context, int status)
```

When the reply for the request posted in i2o_post_wait() arrives, this function copies the reply `status` into the wait queue structure and wakes up the waiting process.

7.2.4 Receiving I_2O Messages

Before modules start to send I_2O requests, they have to register at least one call back function as a handler for the incoming replies. The data structure for a handler contains a pointer to the function to be called when an I_2O interrupt is generated (see Figure 7.1). Registered handlers are collected into `i2o_handlers[]` table. The dispatching via this table is based on the unique `context` number (i.e. the array index), which is delivered to IOP in the I_2O requests' `InitiatorContext` field (last 16 bits). The IOP copies the `InitiatorContext` field unchanged into the reply, where from the interrupt handler is able to find it.

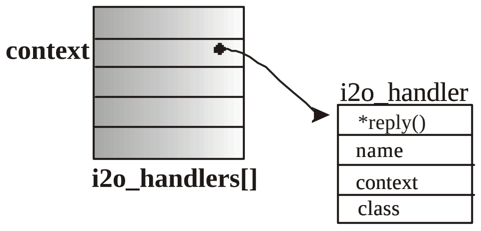

Figure 7.1. `i2o_handlers[]` table.

Functions to register and remove reply handlers are `i2o_install_handler()` and `i2o_remove_handler()`. Functions that process replies written to IOP's outbound queue are `i2o_run_queue()` and `i2o_flush_reply()`.

```
int i2o_install_handler(struct i2o_handler *h)
```

This function adds a handler structure `*h` into the `i2o_handlers[]` table. The `struct handler` contains the address of the callback function to be run when the reply arrives, and a context number `h->context` (same as the index of the new entry) to be used as an identification part of the `InitiatorContext` field in the I_2O requests (last 16 bits). Returns `h->context` (succeed) or `–ENOSPC` (no space left in the table).

```
int i2o_remove_handler(struct i2o_handler *h)
```

This function sets `h->context` to –1, and removes handler structure `*h` from the `i2o_handlers[]` table. Returns 0.

```
static void i2o_run_queue(struct i2o_controller *iop)
```

This interrupt service routine reads replies from the IOP's outbound queue and runs the correct OSM's handler, until the outbound queue is empty. The identity of the interrupting IOP is passed in i2o_controller structure pointed by `*iop`. The function uses reply's `InitiatorContext` field to identify the handler.

```
while ((m = *iop->read_port) != 0xFFFFFFFF) {
     msg = (struct i2o_message *)bus_to_virt(m);
     context = msg.InitiatorContext & 0xFFFF;
     handler = i2o_handlers[context];
     handler->reply(handler, iop, msg);
     i2o_flush_reply(iop, m);
}
```

The handler checks the message status and does module specific tasks, e.g. copies bytes to device specific buffers. When the control later returns back from the handler, the message frame is freed by calling `i2o_flush_reply()`.

```
inline void i2o_flush_reply(struct i2o_controller *iop, u32 m)
```

This function frees the message frame used for the reply by writing its physical address `m` into outbound queue of the pointed `*iop`. This is implemnetd as an inline code to be done efficiently.

7.3 Executive functions

Executive class messages are defined in Chapter 4 of *I_2O Specification* [2]. The messages are targeted to the IOP and its Executive DDM. The functions in this class manage IOPs system initialization, configuration and peer-to-peer connections.

7.3.1 Executive class functions implemented in HPGIN-project

Currently only a subset of the Executive class messages are used and implemented. They include functions needed during the initialization and functions needed by Software management and by Configuration management. The function names and short descriptions are given below. Section numbers after the I_2O message name refer to corresponding chapter in the *I_2O Specification* [2], where the details of the requests and the replies are described.

```
int i2o_hrt_get(struct i2o_controller *iop)
```

This function posts ***ExecHrtGet*** request (4.4.3.15) to get the IOP's hardware resource table. The function reserves memory for the table and sets `*iop->hrt` to point to it. Returns 0 (success), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=`SUCCESS`).

```
int i2o_clear_controller(struct i2o_controller *iop)
```

This function posts ***ExecIopClear*** request (4.4.3.16) to the `*iop` to abort pending requests. The IOP rebuilds its inbound message queues and deletes all entries in external connection table. Since in the beginning of the initialization normal I_2O replies can't be delivered (the IOP's outbound queue is not initialized), there is no normal reply to this request and memory polling is used instead. Returns 0 (success), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=`SUCCESS`).

```
int i2o_reset_controller(struct i2o_controller *iop)
```

This function posts ***ExecIopReset*** request (4.4.3.18) to the `*iop` to abort pending requests The IOP rebuilds its environment – reloads IRTOS and resident DDMs. Since in the beginning of the initialization normal I_2O replies can't be delivered (the IOP's outbound queue is not initialized), there is no normal reply and memory polling is used instead. Returns 0 (state=RESET), –`ETIMEDOUT` (timeout) or -`ENOMEM` (kernel memory allocation error).

```
int i2o_lct_get(struct i2o_controller *iop)
```

This function posts ***ExecLctNotify*** request (4.4.3.19) to the `*iop` to get IOP's logical configuration table after next configuration change. The function reserves memory for the table and sets `*iop->lct` to point to it. When the target IOP modifies its Lct, it replies to this message, sending Lct (i.e. broadcasting) to everyone who made this request. Returns 0 (success), `-ENOMEM` (kernel memory allocation error), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=`SUCCESS`).

```
int i2o_parse_lct(struct i2o_controller *iop)
```

This function parses the Lct, prints debugging information to log and installs i2o_device structures for I_2O devices by calling `i2o_install_device()`. Returns 0 (succeed) or `-ENOMEM` (kernel memory allocation error).

```
int i2o_init_outbound_q(struct i2o_controller *iop)
```

This function posts ***ExecOutboundInit*** request (4.4.3.20) to the `*iop` to clear IOPs outbound message queue to its initial (empty) state. Returns 0 (success), `-ENOMEM` (kernel memory allocation error), `-ETIMEDOUT` (timeout) or `-EINVAL` (initialize rejected).

```
int i2o_status_get(struct i2o_controller *iop)
```

This function posts ***ExecStatusGet*** request (4.4.3.26) to the `*iop` to get the IOP status: state, size of message frames, and size of inbound and outbound queues, etc. Returns 0 (succeed), `-ENOMEM` (kernel memory allocation error) or `-ETIMEDOUT` (timeout).

```
int i2o_enable_controller(struct i2o_controller *iop)
```

This function posts ***ExecSysEnable*** request (4.4.3.30) to the `*iop` to release *ExecSysQuiesce* state and resume normal operation. Returns 0 (success), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=`SUCCESS`).

```
int i2o_quiesce_controller(struct i2o_controller *iop)
```

This function posts ***ExecSysQuiesce*** request (4.4.3.32) to the `*iop` to stop IOP sending messages and ignore all except system messages. Returns 0 (success), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=`SUCCESS`).

```
int i2o_systab_send(struct i2o_controller *iop)
```

This function posts ***ExecSysTabSet*** request (4.4.3.33) to the `*iop` to provide system configuration table (SysTab) and to enable peer operation. Returns 0 (success), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=`SUCCESS`).

7.3.2 Executive class messages not implemented in HPGIN-project

The following table (Table 7.1) lists Executive class messages that are out of the scope of the HPGIN-project and are not implemented.

ExecAdapterAssign	4.4.3.1	Assign an adapter to the specified HDM.
ExecAdapterRead	4.4.3.2	Request that the IOP read the registers of a hidden adapter.
ExecAdapterRelease	4.4.3.3	Revoke the adapter assignment.
ExecBiosInfoSet	4.4.3.4	Indicate a device accessible via BIOS function call – sets field in logical configuration table.
ExecBootDeviceSet	4.4.3.5	Indicate device used to boot the OS – set field in logical configuration table.
ExecConnSetup	4.4.3.7	Establish aliases for sending messages between I_2O devices on different IOPs.
ExecDdmDestroy	4.4.3.8	Terminate local DDM operation – release all signed adapters and I_2O devices; destroy all devices created (registered) by the specified module.
ExecDdmEnable	4.4.3.9	Release ***ExecDdmQuiesce*** state and resume normal operation with specified DDM.
ExecDdmQuiesce	4.4.3.10	Stop sending messages to specified remote DDM (on another IOP) and ignore messages from that DDM. Used when shutting down the other DDM.
ExecDdmReset	4.4.3.11	Clear all connections with the specified DDM. Sent when reloading the DDM.
ExecDdmSuspend	4.4.3.12	Suspend local DDM operation – quiesce devices created (i.e., registered) by the specified module
ExecDeviceAssign	4.4.3.13	Assign a device to the specified ISM (i.e., invite a connection between the ISM and the device)
ExecDeviceRelease	4.4.3.14	Release a device – break connection.
ExecIopConnect	4.4.3.17	Establish aliases for sending messages between IOP executives.
ExecPathEnable	4.4.3.21	Release PathQuiesce state and resume normal operation with specified IOP.
ExecPathQuiesce	4.4.3.22	Stop sending messages to specified IOP and ignore messages from that IOP. Used when shutting down the other IOP. Sent before resetting the other IOP.
ExecPathReset	4.4.3.23	Clear all connections with specified IOP. Sent when resetting the other IOP.
ExecStaticMfCreate	4.4.3.24	Create and stuff a static message frame.
ExecStaticMfRelease	4.4.3.25	Release a static message frame.
ExecSysModify	4.4.3.31	Stop sending messages and ignore all but system messages. Also, suspend all activity to adapters on the system bus, in preparation for a physical system configuration change. Especially useful when the host is about to change PCI configuration (e.g. physical address of one or more IOPs).

Table 7.1. Executive class messages not implemented in HPGIN-project.

7.4 Utility functions

Utility class messages are defined in section 6.1.3 in the *I_2O Specification* [2]. Utility class messages are common to all driver classes.

7.4.1 Utility class functions implemented in HPGIN

Currently only a subset of the Utility class messages are used and implemented. They include functions to claim and release devices, to handle events and to get and set parameters. The names of the functions and short descriptions are given below. Section numbers after the I_2O message names refer to corresponding chapters in the *I_2O Specification* [2], where the details of the requests and the replies are described.

```
int i2o_claim_device(struct i2o_device *i2o_dev, struct i2o_handler *h)
```

This function posts ***UtilClaim*** (6.1.3.2) message to request use of the i2o device `*i2o_dev`. `i2o_dev->owner` is set to `*h`. Returns 0 (succeed), `-EBUSY` (has already a primary user or too many managers) or `–ETIMEDOUT` (timeout).

```
int i2o_release_device(struct i2o_device *i2o_dev,
                       struct i2o_handler *h)
```

This function posts ***UtilClaimRelease*** (6.1.3.3) request to release the claimed device `*i2o_dev` owned by `*h`. Returns 0 (success), `-ENOENT` (not owner) or `-ETIMEDOUT` (timeout).

```
int i2o_event_ack(struct i2o_controller *iop, u32 *msg)
```

This function posts ***UtilEventAck*** (6.1.3.7) request to `*iop` acknowledge an event. `*msg` is the original ***UtilEventRegister*** reply. Returns 0 (success), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=SUCCESS).

```
int i2o_event_register(struct i2o_controller *iop, int tid,
                  int init_context, u32 tr_context, u32 evt_mask)
```

This function posts ***UtilEventRegister*** (6.1.3.8) to turn on/off event notification. The `init_context` is the value for the InitiatorContext field (i.e. unit number and handler_context), `tr_context` is the value for the TransactionContext field and `evt_mask` contains new value for the event mask. To turn off the event notification, use zero value for the `evt_mask`. The target device is identified by `*iop` and `tid`. Returns 0 (success), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=SUCCESS).

```
int i2o_query_scalar(struct i2o_controller *iop, int tid,
                     int group, int field, void *buf, int buflen)
```

This function posts ***UtilParamsGet*** (6.1.3.12) request to read selected `field` from a device scalar parameter `group` or a whole parameter `group` if `field==-1`. The result value or the list of result values is returned in memory area *`buf`. The target device is identified by `*iop` and `tid`. See chapter 3.4.7 in the I_2O Specification for the operation result details. Returns number of bytes written into *buf, -`ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=SUCCESS).

```
int i2o_query_table(int oper,
                    struct i2o_controller *iop, int tid,
                    int group,
                    int fieldcount, void *ibuf, int ibuflen,
                    void *resblk, int reslen)
```

This function posts ***UtilParamsGet*** (6.1.3.12) request to read fields from a device table parameter `group`. The device is identified by `*iop` and `tid.` The result block is given in a memory buffer `*resblk.` The length of the buffer (in bytes) is given in `reslen`. Returns number of bytes written into *resblk, -`ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=SUCCESS).

1) If `oper==I2O_PARAMS_TABLE_GET`, returns <u>from all rows</u>

- <u>all fields</u> when `fieldcount==-1`. In this case `*ibuf` and `ibuflen` are unused.
- <u>specified fields</u> when `fieldcount>0`. Field indexes are given in the memory buffer `*ibuf`, and `fieldcount` is the number of indexes. The length of the buffer (in bytes) is given in `ibuflen`.

2) If `oper==I2O_PARAMS_LIST_GET`, returns <u>from specified rows</u>

- <u>all fields</u> when `fieldcount==-1`. The memory buffer pointed by `*ibuf` contains the row count and key values for queried rows.
- <u>specified fields</u> when `fieldcount>0`. Field indexes, number of following row keys (key count) and key values are given in the memory buffer `*ibuf`. `fieldcount` is the number of field indexes in the buffer. The length of the buffer (in bytes) is given in `ibuflen`.

See chapter 3.4.7 in the I_2O Specification for the operation list details and for operation results details.

```
int i2o_set_scalar(struct i2o_controller *iop, int tid,
                   int group, int field, void *buf, int buflen)
```

This function posts ***UtilParamsSet*** (6.1.3.13) request to set a selected `field` in a device scalar parameter `group` or to set all fields in the `group` if `field==-1`. Memory area `*buf` contains the operation list. The target device is identified by `*iop` and `tid` See chapter 3.4.7 in the I$_2$O Specification for the operation list details. Returns number of bytes used in *buf, `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=SUCCESS).

```
int i2o_clear_table(struct i2o_controller *iop, int tid, int group)
```

This function posts ***UtilParamsSet*** (6.1.3.13) request to clear a table parameter group i.e. to delete all rows. The target device is identified by `*iop` and `tid`. Returns 0 (success), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=SUCCESS).

```
int i2o_row_add_table(struct i2o_controller *iop, int tid,
                    int group, int fieldcount, void *buf, int buflen)
```

This function posts ***UtilParamsSet*** (6.1.3.13) request to add rows to a table parameter group `group`. The target device is identified by `*iop` and `tid`. Field indexes, number of following row keys (key count) and key values are given in the memory buffer `*buf`. `fieldcount` is the number of field indexes in the buffer. The length of the buffer (in bytes) is given in `buflen`. Returns 0 (success), `-ETIMEDOUT` (timeout) or -DetailedStatus (ReqStatus!=SUCCESS).

7.4.2 Utility class messages not implemented in HPGIN-project

The following Utility class messages are out of the scope of the HPGIN project and are not yet implemented.

UtilAbort	6.1.3.1	Abort previous transaction(s).
UtilDeviceRelease	6.1.3.5	Release ownership of device.
UtilDeviceReserve	6.1.3.6	Acquire ownership of device.
UtilLock	6.1.3.9	Request temporary exclusive control of device.
UtilLockRelease	6.1.3.10	Release lock.
UtilReplyFaultNotify	6.1.3.14	Reply message can’t be delivered by the transport layer.

Table 7.2. Utility class messages not implemented in HPGIN-project.

7.5 Debugging and Error Reporting functions

Functions whose name start by i2o_report_ and function i2o_dump_message are for debugging and error reporting.

```
void i2o_dump_message(u32 *msg)
```

This function prints to log the contents of the message frame `*msg`.

```
void i2o_report_status(const char *severity, const char *str, u32 *msg)
```

This function prints the string `*str`, the command name, the request status and the detailed status of the reply `*msg`.

```
void i2o_report_failure(const char *severity,
         const struct i2o_controller *iop, const char *str, u32 *msg)
```

This function prints the string `*str`, the request status and the detailed status of the reply *msg, and dumps out the contents of the message.

```
void i2o_report_transaction_error(const char *severity,
         const char *str, u32 *msg)
```

This function prints the string `*str`, the request status and the detailed status of the reply *msg, when the request is rejected for a general cause.

```
static void i2o_report_exec_cmd(u8 cmd)
```

This function prints the Executive class command name corresponding to number `cmd`.

```
static void i2o_report_util_cmd(u8 cmd)
```

This function prints the Utility class command name corresponding to number `cmd`.

```
static void i2o_report_lan_cmd(u8 cmd)
```

This function prints the LAN class command name corresponding to number `cmd`.

```
static void i2o_report_common_status(u8 req_status)
```

This function prints the request status string corresponding to `req_status` code. The common request status codes are used by all I_2O class replies.

```
static void i2o_report_common_dsc(u16 detailed_status)
```

This function prints the detailed status string corresponding to `detailed_status` code. The common detailed status codes are used by all I_2O class replies.

```
static void i2o_report_lan_dsc(u16 detailed_status)
```

This function prints the LAN detailed status string corresponding to `detailed_status` code.

```
static void i2o_report_fail_status(u8 req_status)
```

This function prints request status string corresponding to `req_status` code, when a message failure has occurred.

```
static void i2o_report_controller_unit(struct i2o_controller *iop,
         int unit)
```

This function queries and prints vendor and device information of the `*iop` whose unit number `unit`.

8 OS SERVICE MODULES

The OSM must adhere to the message requirements specified in chapter 3 (Basic Requirements) and chapter 6 (Class Specifications) of the *I_2O Specification* [2]. This means support for all drivers' Base class messages (chapters from 6.4 to 6.12), and optional support for private messages.

The OSM must send only messages specified for the class for which the target is registered. The OSM must be capable of processing replies from the message layer as well as replies from its intended target. The OSM must be able to correlate replies with the appropriate request, based on the context of the `TransactionContext` field. OSMs only send requests and receive replies. They neither send replies nor receive requests. OSMs do not need to establish connections, but they do need to claim devices they intend to consume.

There are three basic types of device in Linux: block-oriented devices, character-oriented devices and network devices. Block devices are those to which there is random access, which means that any block can be read or written to at will. Character devices are devices, which can usually be processed sequentially. Network devices are used to connect to other computers.

8.1 LAN OSM

The LAN OSM module (`i2o_lan`) implements the interface to the local area network devices. This section describes the layer structure for the network, the Linux network interface and the functionality of the I_2O LAN class messages.

8.1.1 Layer structure for the network

When a user process communicates via the network, it uses functions provided by the BSD socket layer (Figure 8.1). This administers a general data structure for sockets. Below BSD socket layer is the INET socket layer, which manages the communication end points for IP-based protocols TCP and UDP. The layer that underlies the INET socket layer is determined by the type of socket, and may be the UDP or TCP layer or the IP layer directly. The UDP layer implements the User Datagram Protocol on the basis of IP, and the TCP layer implements the Transmission Control Protocol for reliable communication links. The IP layer contains the code for the Internet Protocol.

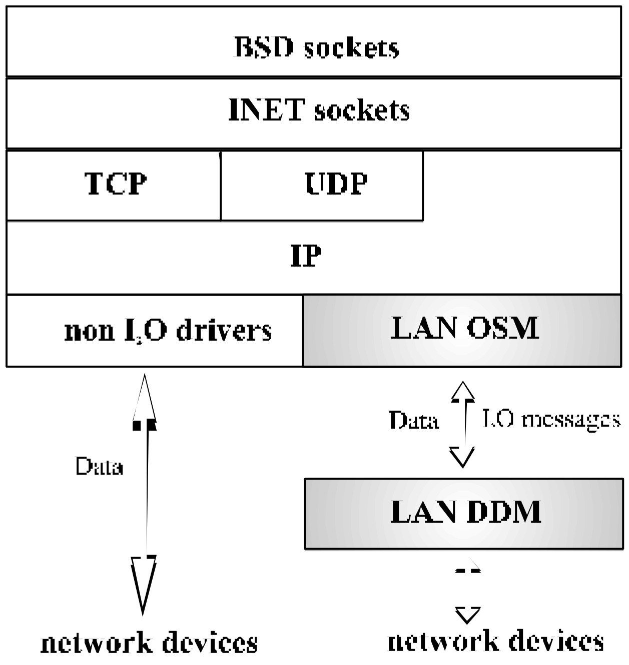

Figure 8.1 The layer structure of the network.

Below the IP layer are the network device drivers, to which the IP passes the final packets. These take care of physical transport of the information. For I_2O aware devices, only the LAN class OSM is needed. For conventional devices, there is one driver for each type of network device.

The data sent by a user process is passed downwards through the protocol stack. Each layer takes care of administrative functions and adds its own header. A feature of Linux is that all headers are written to memory in a linear sequence. If the length of the data exceeds the maximum segment size, it is divided into number of packets. It is also possible for short data blocks to be collected together in one segment.

Each packet handled by the kernel is contained in a socket buffer structure (`struct sk_buff *skb`, see include/linux/skbuff.h). Each network packet belongs to a socket in the higher network layers, and the input/output buffers of any socket are lists of `struct sk_buff` structures. The same `sk_buff` structure is used to host network data throughout all the Linux network subsystems, but a socket buffer is just a packet as far as the interface is concerned.

Finally the IP layer calls the network driver method `hard_start_xmit(struct sk_buff *, struct net_device *)`, which passes the packet to the driver and to the network.

The various layers are also connected together in the opposite direction. When packets are received from the network, the hardware triggers an interrupt. The interrupt is handled on the interrupt handler registered by the driver. In the I_2O implementation this is done by the host Messenger's `i2o_run_queue()` function. The incoming I_2O message is then dispatched to the LAN OSM, that forwards `sk_buff` structures to higher layers of the network implementation by calling Linux Driver API method `netif_rx()`.

8.1.2 Linux Network Device Interface

The Linux interface to network device is as follows (Refer e.g. to *Linux Kernel Internals* [3] for details, see also include/linux/netdevice.h):

```
struct net_device
{    ...
     /* Pointer to the devices private memory area. */
     void *priv;
     ...
     /* Pointers to the fundamental device methods. */
     int (*open)(struct net_device *dev);
     int (*stop)(struct net_device *dev);
     int (*hard_start_xmit) (struct sk_buff *skb,
                             struct net_device *dev);
     struct net_device_stats* (*get_stats)(struct net_device *dev);
     void (*set_multicats_list)(struct net_device *dev);
     ...
     /* Pointers to the optional device methods */
     ...
}
```

The network device interface can be conceptually divided into two parts: "visible" and "invisible". The visible part of the structure is composed of the fields that are explicitly assigned in the `struct net_device`. The remaining fields are used internally. Some of them are accessed by drivers, e.g. during the initialization, while some shouldn't be touched in drivers. Some of the fields convey information about the interface, while some exists only for the benefit of the driver.

There are also other fields in `struct net_device`, most notably the device methods that are part of the kernel-driver interface. Device methods can be divided into fundamental and optional methods. Fundamental methods include those that are needed to be able to use the interface; optional methods implement more advanced functions that are not strictly required. The device methods in LAN OSM convert network issues into I_2O messages and post them to the IOP, which dispatches them to the destination LAN DDM.

The device is linked into the global network device chain by calling function `register_netdev(struct net_device *)`.

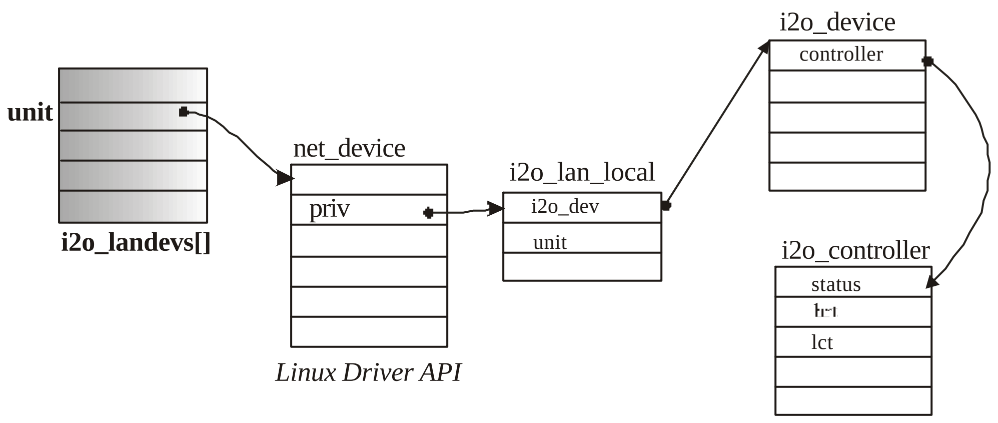

Figure 8.2. Data structures used by the LAN OSM.

The new I_2O device dependent data is registered separately in the `struct i2o_device`. It is linked to the `struct net_device` through the private data area (`priv->i2o_dev`, see Figure 8.2). The private data area and the appropriate device methods are set during the initialization.

The I_2O reply handler gains access to the `struct net_device` by indexing the table `i2o_landevs[]` with the `unit` number. The unit number is passed to the DDM in the request's `InitiatorContext` field together with the OSM specific `handler context` number. The `InitiatorContext` field is copied unchanged into the reply.

8.1.3 Setup functions

Section numbers after the following I_2O message names refer to corresponding chapters in the *I_2O Specification* [2]. Refer to these chapters for the details of the requests and replies.

```
int __init i2o_lan_init(void) / init_module(void)
```

This is the module initialization function. The function installs reply handlers for ***LanPacketSend*** (6.10.8.1), ***LanSduSend*** (6.10.8.2) and ***LanReceivePost*** (6.10.8.3) requests and registers to kernel LAN class devices found in global `i2o_controllers[]` table.

```
void cleanup_module(void)
```

This is the module cleanup function. The function unregisters all LAN class devices and removes reply handlers.

```
static struct net_device *i2o_lan_register_device(
         struct i2o_device *i2o_dev)
```

This function registers the network device into the kernel. The function reserves memory for the net_device structure and for the private area (`dev->priv`) and initializes it with module parameters, with values queried from the DDM and with callback function addresses. Returns a pointer to the created net_device structure.

```
static int i2o_lan_open(struct net_device dev)
```

This function opens the network device for the transfering. The function claims the device, registers an event mask, resets the device, sets it into batch mode, posts free buckets to the controlling DDM and starts the Linux network queue. Return 0 (succeed), `-EAGAIN` (unable to claim) or –`ENOMEM` (kernel memory allocation error).

```
static int i2o_lan_close(struct net_device *dev)
```

This function ends the transfering. The function stops the Linux network queue, suspends the device and releases the device. Return 0 (success) or –`EBUSY` (unable to release).

```
static void i2o_lan_set_ddm_parameters(struct net_device *dev)
```

This function sets default values for LAN Class parameters in DDM. The into batch mode.

8.1.4 Functions to send LAN class requests

The DDM registers a LAN class device for each port it provides, and identifies devices by a unique Target ID. The OSM claims the device and performs LAN operations by sending requests to target LAN devices, and by listening for replies from all LAN class devices. Both sending and receiving can be in batch mode, i.e. requests and replies may contain multiple buckets of packets.

LAN class messages are defined in chapter 6.10 of the *I_2O Specification* [2]. Functions to create and send LAN class requests to the target IOP's inbound queue are `i2o_lan_packet_send()`, `i2o_lan_sdu_send()`, `i2o_lan_batch_send()`, `i2o_lan_receive_post()`, `i2o_lan_reset()` and `i2o_lan_suspend()`. Each function gets a pointer `*dev` to device to which the request will be send. Functions `i2o_lan_packet_send()` and `i2o_lan_sdu_send()` get also a pointer `*skb` to a socket buffer containing the outgoing packet.

```
static int i2o_lan_packet_send(struct sk_buff *skb,
         struct net_device *dev)
```

This function creates a message containing a batch of packets to be sent to the DDM. This function is registered to the Linux network API as callback function `hard_start_xmit()`. The batch is filled on subsequent calls. Each time a packet to be added is passed in a socket buffer pointed by `*skb`. The address `*skb` is copied to message `TransactionContext` field, and is copied to the reply. The function increments `dev->priv->tx_out` counter. Returns 0 (success) or 1 (out of free message frames).

```
static int i2o_lan_sdu_send(struct sk_buff *skb,
         struct net_device *dev)
```

This function is similar to `i2o_lan_packet_send()`except, that the MAC header is excluded and is generated by the DDM.

```
static void i2o_lan_batch_send(struct net_device *dev)
```

This function posts the batch ***LanPacketSend*** request or ***LanPacketSend*** request pointed by `dev->priv->m`. The function sets `dev->priv->tx_count` to 0 and `dev->priv->send _active` to 0.

```
static void i2o_lan_tx_timeout(struct net_device *dev)
```

This is the timeout function to be called by the Linux network interface when a timeout occurs during the above packet send. The function restarts the network queue if it is stopped.

```
static int i2o_lan_receive_post(struct net_device *dev)
```

This function reserves socket buffers for buckets to receive incoming packets and posts them in a ***LanReceivePost*** request. The function increments `dev->priv->buckets_out` counter. Returns 0 (success), `-ENOMEM` (kernel memory allocation error) or –`ETIMEDOUT` (out of free message frames)

```
static int i2o_lan_reset(struct net_device *dev)
```

This function posts ***LanReset*** request (6.10.8.5) to the target DDM and causes a hardware reset to be issued. Returns 0 (success) or –`ETIMEDOUT` (timeout or request failed).

```
static int i2o_lan_suspend(struct net_device *_dev)
```

This function posts ***LanSuspend*** request (6.10.8.6) to put the adapter in an idle (suspended) state. Returns 0 (success) or –`ETIMEDOUT` (timeout or request failed).

8.1.5 Functions to handle replies

The above mentioned functions fill request's `InitiatorContext` field with a `unit` number and a `handler context` number. The `InitiatorContext` is copied unchanged into reply by the DDM. The `unit` number is an index to `i2o_landevs[]` entry, which points directly to `struct net_device` structure identifying the requestor.

```
unit = msg.InitiatorContext >> 16;
dev = (struct net_device *)i2o_landevs[unit];
```

The `handler context` is an index to the `i2o_handlers[]`entry, which points to an i2o_handler structure. The structure contains the address of the handler routine, which is called from the interrupt handler in `i2o_core` module when the reply arrives.

The handlers for the incoming replies in LAN OSM are `i2o_lan_send_post_reply()`, `i2o_lan_receive_post_reply()` and `i2o_lan_reply()`. They are registered when the LAN OSM is loaded. The corresponding `handler context` numbers are stored into `lan_post_context`, `lan_receive_context` and `lan_context`. Also the functions called by the handlers are presented below.

```
static void i2o_lan_send_post_reply(struct i2o_handler *h,
        struct i2o_controller *iop, struct i2o_message *m)
```

This is the handler for ***LanPacketSend*** and ***LanSduSend*** replies. The function inspects the reply status, calls error handling functions if necessary, and frees socket buffers listed in the reply'sTransaction List. The function decrements `dev->priv->tx_out` counter.

```
static void i2o_lan_receive_post_reply(struct i2o_handler *h,
        struct i2o_controller *iop, struct i2o_message *m)
```

This is the handler for ***LanReceivePost*** replies. The function inspects the reply status, calls error handling functions if necessary, and calls `netif_rx()` function for all packets (i.e. socket buffers) listed in the Packet Description Block. If the DDM is just returning unused buckets (i.e. socket buffers), they are freed. The function decrements `dev->priv->buckets_out` counter. If the DDM has already used `dev->priv->treshhold` buckets, the function posts new buckets to the DDM.

```
static void i2o_lan_release_buckets(struct net_device *dev, u32 *msg)
```

This function is used to release unused buckets returned by the DDM.

```
static void i2o_lan_reply(struct i2o_handler *h,
         struct i2o_controller *iop, struct i2o_message *m)
```

This is the handler for other incoming replies. The function inspects the reply status, calls error handling functions if necessary, and calls `i2o_lan_handle_event()` for event notifications.

```
static void i2o_lan_handle_event(struct net_device *dev, u32 *msg)
```

This function handles the incoming ***UtilEventRegister*** or ***UtilEventAck*** replies.

```
static void i2o_lan_handle_failure(struct net_device *dev, u32 *msg)
```

This function is called if the reply’s MSG_FAIL bit is set. The function prints error information into log, frees returned socket buffers and releases the preserved message.

```
static void i2o_lan_handle_status(struct net_device *dev, u32 *msg)
```

This function inspects reply’s request status and detailed status fields and calls i2o_lan_handle_failure() or i2o_lan_handle_transaction_error() functions.

```
static void i2o_lan_handle_transaction_error(struct net_device *dev,
         u32 *msg)
```

This function is called if the reply’s detailed status indicates that a transaction error has occured. The function prints error information into log and frees returned socket buffer.

8.1.6 Other LAN OSM functions

```
static struct net_device_stats *i2o_lan_get_stats(
         struct net_device *dev)
```

This function queries device statistics from the DDM and fills `dev->priv->stats` with the replied values. Returns a pointer to the `dev->priv->stats`.

```
static void i2o_lan_set_mc_filter(struct net_device *dev)
```

This function inspect `dev->flags` and sets the corresponding value to the FilterMask in `LAN_MAC_ADDRESS` parameter group. By setting the mask the network device is enabled to receive packets not send to the protocol address.

```
static void i2o_lan_set_mc_table(struct net_device *dev)
```

This function inspect `dev->flags` and sets the corresponding value to the FilterMask in `LAN_MAC_ADDRESS` parameter group. By setting the mask the network device is enabled to receive packets not send to the protocol address.

```
static void i2o_lan_set_multicast_list(struct net_devive *dev)
```

This function simply queues a task to call later `i2o_lan_set_mc_list()`.

```
static int i2o_lan_change_mtu(struct net_device *dev, int new_mtu)
```

This function changes the `dev->mtu` value to `new_mtu`. Returns 0 (succeed) or `-EFAULT` (`new_mtu` out of range).

The following subsections describe in details the flow of operations when the host sends packets to the network, prepares to receive packets and receives packets from network.

8.1.7 Sending packets to the network

The LAN OSM sends packets using ***LanPacketSend*** or ***LanSduSend*** request. For the ***LanPacketSend***, the user supplies the complete packet. For ***LanSduSend*** the LAN device supplies the MAC header and the user supplies the rest of the packet.

1. The OS issues an I/O request by calling the network driver API function registered by the OSM:

   ```
    i2o_lan_packet_send(struct sk_buff *skb, struct net_device *dev)
   ```

   Linux's socket/protocol layers write all headers and data to memory in linear sequence, so `*skb` points to a single packet.

2. The OSM creates an I_2O message addressed to target device

   ```
   i2o_dev = dev->priv->i2o_dev;
   ```

```
iop = i2o_dev->controller;
msg = i2o_wait_message(iop);
msg->TargetAddress = i2o_dev->lct_data->tid;
msg->Function = I2O_LAN_PACKET_SEND;
msg->InitiatorContext = priv->unit << 16 | lan_send_context;
msg->SGL[0]->PhysicalAddress = virt_to_bus(skb->data);
msg->SGL[0]->Flags = 0xD5000000 | skb->len;
msg->SGL[0]->TransactionContext = skb;
```

`TransactionContext` is used to identify this packet in the reply so that it can be freed later. `InitiatorContext` is set to indicate this device (`unit`) and the message handler for the reply (`lan_send_context`).

3. The OSM calls `i2o_post_this(iop, i2o_dev->id, msg, sizeof(msg))` and the host's Messenger queues the message into the IOP's inbound queue port.

4. The IOP and target DDM process the message, and send ***LanPacketSend*** reply.

5. The IOP alerts the host's Messenger via an interrupt. Control is moved to the interrupt handler `i2o_interrupt()`, which reads the reply from the IOP's outbound queue and calls the handler for the reply

```
context = msg.InitiatorContext & 0xFFFF;       // use last 16 bytes
i2o_handlers[context]->reply(context, iop, msg);
```

6. In this case we get reply to *LanSend* and the registered handler is

```
i2o_lan_send_post_reply(struct i2o_handler *h,
                     struct i2o_controller *iop, struct message *m)
```

Reply's `InitiatorContext` contains also the `unit` number. It is an index to the `i2o_landevs[]` entry, which points directly to the `struct net_device` structure, which is the basic Linux interface to network device.

```
unit = msg->InitiatorContext >> 16;
dev = i2o_landevs[unit];
```

7. The OSM handler inspects the transmission status and message failures and transaction errors are handled if necessary.

A single reply may acknowledge multiple packet transmissions of multiple requests. The Transaction Reply List (TRL) is processed, and socket buffers used by the sent packets are freed. The TRL contains pointers to the original packets (which were passed to DDM in requests' `TransactionContext` fields). The detailed TRL format is explained in *I_2O Specification* [2] on section 3.4.3.

```
trl_count = msg->trl_count;
while (trl_count) {
       skb = msg->TRL[trl_count];
       dev_kfree_skb_irq(skb);
       trl_count—-;
}
```

8. Finally the control returns back to the interrupt handler `i2o_interrupt()`. The interrupt handler frees the message frame back to outbound free queue by calling `i2o_flush_reply(iop, m)`.

8.1.8 Preparing to receive packets from the network

All received packets are transferred from the DDM by using buckets reserved in forehand by the OSM. The Initiator (LAN OSM) allocates memory buffers, and describes them in ***LanReceivePost*** messages using SGLs. Each buffer marked by the `end_of_buffer` entry in the SGL corresponds to one bucket. Buckets do not have to be physically contiguous, and they can be of varying sizes.

The DDM writes incoming packets into these buckets. The DDM describes in reply in a Packet Descriptor Block (PDB) each bucket it consumes, the bucket's order, and the location and length of each packet in the bucket.

The SGL element of each bucket contains a `BufferContext` field, analogous to the `TransactionContext` in messages. The host tracks buckets by `BufferContext`, which is passed to the DDM in the SGL and reported back in the PDB.

When buckets are posted to the DDM, the DDM owns them. When a packet is received, the DDM (or its hardware) copies the packet into one or more buckets, depending on its size and the space remaining in the particular bucket. The DDM can use buckets in arbitrary order. When the DDM reports a packet buffer back in the PDB, the ownership of the bucket returns to the host. The DDM does not touch that bucket again unless it is reposted by the Initiator. The detailed bucket format is explained in *I_2O Specification* [2] on page 6-103.

1. The OSM calls function

   `i2o_lan_receive_post(struct net_device *dev)`.

2. Host OSM reads a free MFA from the IOP inbound queue.

   ```
   i2o_dev = dev->priv->i2o_dev;
   iop = i2o_dev->controller;
   m = i2o_wait_message(iop);
   msg = bus_to_virt(iop->mem_ofset + m);
   ```

3. OSM allocates memory for buffers of size = MTU (maximum transfer unit), and describes this memory by using a SGL. Each buffer marked in the SGL corresponds to a bucket. `BufferContext` is set to identify the allocated buffer in the reply so that it can be later freed. In addition, the total number of buckets is passed to the DDM. `InitiatorContext` is set to indicate the reply handler.

   ```
   msg->Function = I2O_LAN_RECEIVE_POST;
   msg->TargetAddress = i2o_dev.id;
   msg->InitiatorContext = priv->unit << 16 | lan_receive_context;
   do
          skb = dev_alloc_skb(dev->mtu + dev->hard_header_len);
          msg->SGL[i]->PhysicalAddress = virt_to_bus(skb->data);
          msg->SGL[i]->Flags = 0x51000000 | skb->len;
          msg->SGL[i]->BufferContext = skb;
   while (i++ <= N_BUFS);
   msg->BucketCount = N_BUFS;
   ```

4. The OSM calls Host Messenger's function `i2o_post_message()`to write the request into the IOP's inbound queue port.

5. IOP and DDM process the message. There is no immediate reply to ***LanReceivePost*** request. ***LanReceivePost*** replies are send later, when there are incoming packets to be delivered. The reply is handled as described in the following section.

8.1.9 Receiving packets from the network

In immediate mode and under a low load in batch mode, the DDM indicates a receiving packet immediately. Under a heavy load, the DDM collects receiving packets until a threshold is exceeded or a timer expires. In both cases, the DDM indicates the received packets in PDB in the ***LanReceivePost*** reply. The `BucketsRemaining` field is the running count of buckets that DDM has left to consume. The host judges how badly the

DDM needs more buckets by this field. If DDM runs out of buckets, it posts an *Overrun* code in `DetailedStatusCode`.

1. DDM writes incoming packets into buckets allocated earlier by the OSM. The DDM describes each bucket it consumes, the bucket's order, and the location and length of each packet in the bucket, by building a PDB into the ***LanReceivePost*** reply.

2. DDM copies the `InitiatorContext` and `BucketContext` fields from the earlier request to the reply, addresses the reply to the Initiator, and finally invokes the IOP's Messenger. Messenger queues the message into the IOP’s outbound queue.

3. The Host is interrupted and the host’s Messenger dispatches the reply to the handler (see chapter 8.1.7 numbered item 5). In this case the handler is

```
i2o_lan_receive_post_reply(struct i2o_handler *h,
                      struct i2o_controller *iop, struct message *m)
```

4. The OSM handler inspects the transmission status, and message failures and transaction errors are handled if necessary.

5. PDB is a list of packet buffers that contain the received packets in the order DDM received them. The `BucketContext` field identifies the previously posted buckets. The handler goes through the list of buckets and passes the packets to upper protocol layers by calling `netif_rx(struct sk_buff *)`.

```
i2o_dev = dev->priv->i2o_dev;
do
       skb = msg->PDB[i].BucketContext
       netif_rx(skb);
       i++;
while (i < msg->trl_count);
```

   Note, that in the current Linux solution one bucket may not contain several packets, or one packet may not be split into two or more buckets (`PacketOrphanLimit` is set to maximum packet size, see LAN_OPERATION parameter group in *I_2O Specification [2]*).

6. The OSM keeps account of outstanding buckets. If the DDM has already used as many buckets as a chosen threshold, the host allocates and sends new buckets to the DDM calling again `i2o_lan_receive_post()`.

```
if (priv->buckets_out <= priv->max_buckets_out – priv->buckettresh)
```
   i2o_lan_receive_post(dev);

8.1.10 Setting LAN Control Parameters

The user can query and adjust various control parameters of the LAN device both for the OSM and for the DDM.

OSM parameters are read and set via the *`/proc` file system* (see chapter 10). The user configurable parameters for each port in the OSM are

- `MaxBucketsOut` - maximum number of buckets send to DDM
- `BucketThresh` - send more buckets to DDM when this many used
- `TxBatchMode`

  `0:` use immediate mode for transmissions, always send one packet per bucket and post immediately

  `1:` use batch mode for transmissions

  `2:` switch automatically between immediate and batch mode
- `rx_copybreak` - copy receiving packet into a new socket buffer and reuse the old socket buffer if the packet lenght < rx_copybreak
- `event_mask` – set the UtilEventRegister mask to get replies when the specified events occur. Use `0xFFC00002` to get all generic and LAN events, `0x00000000` for none. See *I_2O Specification [2]* for event codes.

LAN parameter groups includes various set of parameters for LAN devices (i.e. for DDMs), e.g. batch control, error control, timeouts and timeout policy, number of retries, recovery etc. Refer to for a complete list of LAN DDM parameters and parameter groups. These DDM specific parameters are read via *Configuration API* (see chapter 9) or via the *`/proc` file system* (see chapter 10). Currently DDM parameter values can be set only via *Configuration API*. The /proc interface will be implemented later.

Batch control specifies how to batch up packets into buffers, and when to notify the user of their arrival. Under a light load the only few packets are put into each bucket and are returned quickly, to minimize latency. Under a heavy load, multiple packets are filled into buckets and multiple buckets are reported with a single reply. Batch control specifies the load conditions when the DDM switches between batch and light modes, and how much to batch in batch mode. See *I_2O Specification* [2] section 6.10.7 LAN Configuration and Operating Parameters.

Error control specifies which transaction errors to report in the transaction status. Since the protocol stack above the LAN OSM (and the user itself) uses various timeouts on packets, it may be pointless to report most errors. Therefore, the DDM supports turning off reporting of individual transmission errors. If a packet encounters a transmission error when errors are disabled, the transaction is reported successful. Other errors, such as in the format of the packets or their batch list, are always reported.

8.2 Other OSMs

Although the implementation of other I_2O class OSMs than LAN OSM are out of the scope of HPGIN project, we present here a short description of block device and character device interface. Currently there exist prototypes also for Block OSM (`i2o_block`) and for SCSI OSM (`i2o_scsi`) made by third parties.

The common parts of the Linux I_2O implementation are designed to be general enough to fulfil the needs for other OSMs too. Adding new OSM should be straightforward.

8.2.1 Block Device Interface

The host OS interface for block storage devices is as follows (see fs/devices.c, include/linux/fs.h). Each block device has an `i`node associated through the directory entry `/dev/name`. The name of the device and its methods are registered in device tables indexed by the major device numbers.

```
static struct device_struct blkdevs[MAX_BLKDEV];

struct device_struct {
     const char * name;
     struct file_operations * fops;
};

struct file_operations {
     loff_t (*llseek)(struct file *, loff_t, int);
     ssize_t (*read)(struct file *, char *, size_t, loff_t *);
     ssize_t (*write)(struct file *, const char *, size_t,loff_t *);
     ...
     int (*ioctl)(struct inode *, struct file *, unsigned int,
                  unsigned long);
     ...
     int (*lock) (struct file *, int, struct file_lock *);
 };
```

The entries in the `blkdevs[]` and the `chrdevs[]` tables are initialized by functions

```
     register_blkdev(unsigned int major, const char *name,
                     struct file_operations *fops)
```

and

```
     register_chrdev(unsigned int major, const char *name,
                     struct file_operations *fops)
```

Pointers to the I_2O device methods corresponding to file operations are set during the initialization. The device methods convert requests to I_2O messages and deliver them to IOP. IOP's Messenger dispatches them to the destination DDM.

To reach the I_2O dependent data, a new table is created during the initialization:

```
struct i2ob_device * i2ob_dev[];

struct i2ob_device {
   struct i2o_controller *iop;
   struct i2o_device *i2odev;
   int tid;
   int flags;
   int refcnt;
   struct request *head, *tail;
   int done_flag;
};
```

The entry contains pointer to `struct i2o_controller` and to `struct i2o_device` identifying the target IOP and target device. Also this table is indexed by the major device number.

8.2.2 Character Device Interface

The character device interface is similar to block device interface and the I2O implementation can be done respectively.

9 CONFIGURATION INTERFACE

The following sections describe the basic idea of the configuration dialogue and how the user level programs can use it.

9.1 Configuration Dialogue

The purpose of the configuration dialogue mechanism is to have a DDM-defined and controlled communication mechanism with a human operator. The facility is self-contained in a downloaded DDM and is available in any I_2O-enabled system.

Static operating parameters, which can be modified only before a session starts, are read-only and must be changed by the configuration dialogue. Several messages support installing and loading DDMs. Installation primarily stores the module's executable code in the IOP's permanent store so that it can be loaded next time the IOP initializes.

DDM's configuration mechanism is invoked when the host sends a ***UtilConfigDialog*** request. The reply to a configuration dialogue request is a set of instructions for displaying configuration information on the console, prompting the user for input, accessing disk drive, and terminating the session. This dialogue modifies the IOP's profile, establishing user-configurable parameters, such as the number of inbound message frames.

The host can initiate the configuration dialogue at any time. The IOP indicates its need for a configuration dialogue by setting a flag bit in its logical configuration table (Lct). The configuration dialogue also applies to each module loaded on the IOP, but the dialogue is invoked independently for each device, using a ***UtilConfigDialog*** request addressed to it. Again, a flag bit for the device in Lct entry indicates that a configuration dialogue is requested. Setting the flag causes a response to the ***ExecLctNotify*** request, if one was posted. Resetting the flag does not.

The host-to-IOP dialogue protocol is based on HTML (Figure 9.1). Every device supplies a page number 0, the device's home page. Other pages are typically accessed by HTML links. The I_2O request message to IOP's specific DDM contains a number of the dialogue page, any form data being returned and a buffer where the device places the reply. The form data is typically generated from an HTML form submitted with the HTTP POST method. The text is in the form `field1=value1&field2=value2`, and usually represents new values for selected fields in selected parameter groups. The reply contains HTML text that the host presents to the human operator via an HTML viewer, such as a Web browser.

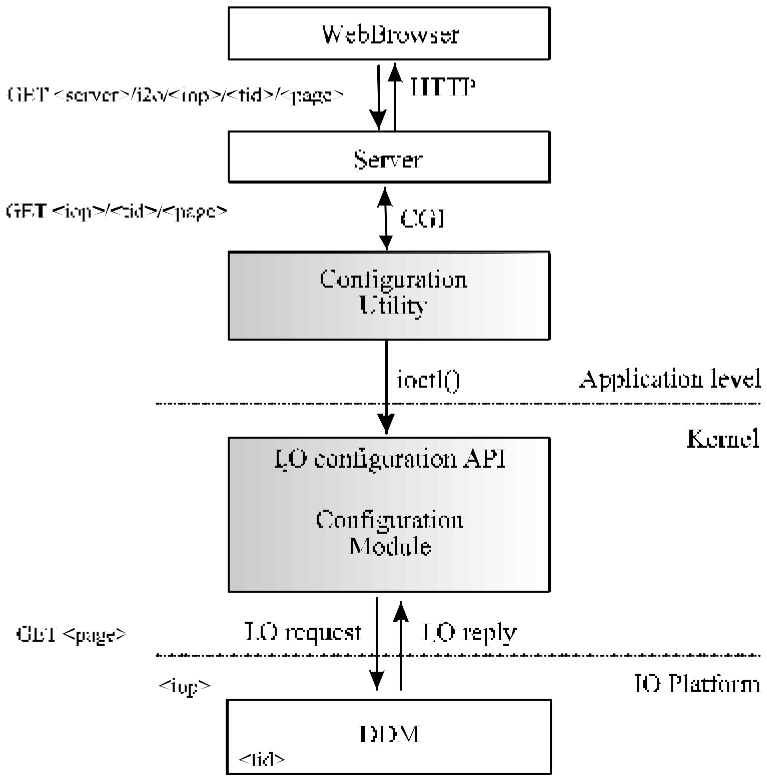

Figure 9.1. Configuration dialogue.

To make possible centralized configuration of I_2O systems safely, the HTTP request/response chain is secured by using a SSL (Secure Sockets Layer) wrapper, which encrypts connections from outside to the server.

9.2 I_2O Configuration API

Access to the I_2O subsystem is provided through the device file named `/dev/i2octl`. This file is a character file with major number 10 and minor number 166. The device interface provides a set of *ioctl()* commands that can be utilized by user space applications to communicate with IOPs and devices on individual IOPs. These ioctl() commands post respective I_2O messages to the specified IOP (`<iop>`) and its target device (`<tid>`), and copies data from I_2O replies to the user space buffer.

The following interface was originally specified by Depax Saxena. It includes basic functions to determine active IOPs, to read Hardware resource table (Hrt) and Logical configuration table (Lct), to get and set parameters in parameter groups, to use HTHLbased configuration interface and to manage IOP's software. The event handling is not yet specified.

9.2.1 Determining active IOPs

Synopsis

```
    ioctl(fd, I2OGETIOPS,  int *count);
         u8 count[MAX_I2O_CONTROLLERS];
```

This function fills the system's active IOP table. `*count` should point to a buffer containing `MAX_I2O_CONTROLLERS` entries. Upon returning, each entry will contain a non-zero value if the given IOP unit is active, and 0 if it is inactive or non-existent. Returns 0 (succeed) or –1. If an error occurs, `errno` is set appropriately:

EFAULT	Invalid user space pointer was passed

9.2.2 Getting Hardware Resource Table

Synopsis

```
    ioctl(fd, I2OHRTGET, struct i2o_cmd_hrt *hrt);
         struct i2o_cmd_hrtlct {
              u32  iop;      /* IOP unit number        */
              void *resbuf;  /* Buffer for result      */
              u32  *reslen;  /* Buffer length in bytes */
         };
```

This function fetches the Hardware Resource Table of the IOP specified by `hrt->iop` into the buffer pointed to by `hrt->resbuf`. The actual size of the data is written into `*(hrt->reslen)`. Returns 0 (succeed) or –1. If an error occurs, `errno` is set appropriately:

EFAULT	Invalid user space pointer was passed
ENXIO	Invalid IOP number
ENOBUFS	Buffer not large enough. If this occurs, the required buffer length is written into `*(hrt->reslen)`

9.2.3 Getting Logical Configuration Table

Synopsis

```
    ioctl(fd, I2OLCTGET, struct i2o_cmd_lct *lct);
         struct i2o_cmd_hrtlct  {
              u32  iop;      /* IOP unit number        */
```

```
          void *resbuf;  /* Buffer for result      */
          u32  *reslen;  /* Buffer length in bytes */
     };
```

This function returns the Logical Configuration Table of the IOP specified by `lct->iop` in the buffer pointed to by `lct->resbuf`. The actual size of the data is written into `*(lct->reslen)`. Returns 0 (succeed) or –1. If an error occurs, `errno` is set appropriately:

EFAULT	Invalid user space pointer was passed
ENXIO	Invalid IOP number
ENOBUFS	Buffer not large enough. If this occurs, the required buffer length is written into `*(lct->reslen)`

9.2.4 Getting Parameters

Synopsis

```
ioctl(fd, I2OPARMGET, struct i2o_parm_setget *ops);
     struct i2o_parm_setget {
          u32  iop;     /* IOP unit number                        */
          u32  tid;     /* Target device TID                      */
          void *opbuf;  /* Operation List buffer                  */
          u32  oplen;   /* Operation List buffer length in bytes */
          void *resbuf; /* Result List buffer                     */
          u32  *reslen; /* Result List buffer length in bytes    */
     };
```

This function posts a ***UtilParamsGet*** message to the device identified by `ops->iop` and `ops->tid`. The operation list for the message is sent through the `ops->opbuf` buffer, and the result list is written into the buffer pointed to by `ops->resbuf`. The actual size of data written is placed into `*(ops->reslen)`. Returns 0 (succeed) or –1. If an error occurs, `errno` is set appropriately:

EFAULT	Invalid user space pointer was passed
ENXIO	Invalid IOP number
ENOBUFS	Buffer not large enough. If this occurs, the required buffer length is written into `*(ops->reslen)`
ETIMEDOUT	Timeout waiting for reply message
ENOMEM	Kernel memory allocation error

A return value of 0 does not mean that the value was actually properly retrieved. The user should check the result list to determine the specific status of the transaction.

9.2.5 Setting Parameters

Synopsis

```
    ioctl(fd, I2OPARMSET, struct i2o_parm_setget *ops);
        struct i2o_cmd_psetget {
            u32  iop;     /* IOP unit number                         */
            u32  tid;     /* Target device TID                       */
            void *opbuf;  /* Operation List buffer                   */
            u32  oplen;   /* Operation List buffer length in bytes */
            void *resbuf; /* Result List buffer                      */
            u32  *reslen; /* Result List buffer length in bytes    */
        };
```

This function posts a ***UtilParamsSet*** message to the device identified by `ops->iop` and `ops->tid`. The operation list for the message is sent through the `ops->opbuf` buffer, and the result list is written into the buffer pointed to by `ops->resbuf`. The number of bytes written is placed into `*(ops->reslen)`. Returns the size in bytes of the data written into ops->resbuf. If an error occurs, -1 is returned and errno is set appropriatly:

EFAULT	Invalid user space pointer was passed
ENXIO	Invalid IOP number
ENOBUFS	Buffer not large enough. If this occurs, the required buffer length is written into `*(ops->reslen)`
ETIMEDOUT	Timeout waiting for reply message
ENOMEM	Kernel memory allocation error

A return value of 0 does not mean that the value was actually changed properly on the IOP. The user should check the result list to determine the specific status of the transaction.

9.2.6 Configuration Dialog HTML-pages

Synopsis

```
     ioctl(fd, I2OHTML, struct i2o_html *htquery);
         struct i2o_html {
             u32   iop;      /* IOP unit number             */
             u32   tid;      /* Target device ID            */
             u32   page;     /* HTML page                   */
```

```
          void  *resbuf;  /* Buffer for reply HTML page   */
          u32   *reslen;  /* and its length in bytes      */
          void  *qbuf;    /* Pointer to HTTP query string */
          u32   qlen;     /* and ist length in bytes      */
        };
```

This function posts an ***UtilConfigDialog*** message to the device identified by `htquery->iop` and `htquery->tid` (see Figure 9.1). The requested HTML page number is provided by the `htquery->page` field, and the resultant HTML text is stored in the buffer pointed by `htquery->resbuf`. If there is an HTTP query string that is to be sent to the device, it should be sent in the buffer pointed to by `htquery->qbuf`. If there is no query string, this field should be set to NULL. The actual size of the reply received is written into `*(htquery->reslen)`. Returns 0 (succeed) or –1. If an error occurs, `errno` is set appropriately:

EFAULT	Invalid user space pointer was passed
ENXIO	Invalid IOP number
ENOBUFS	Buffer not large enough. If this occurs, the required buffer length is written into `*(htquery->reslen)`
ETIMEDOUT	Timeout waiting for reply message
ENOMEM	Kernel memory allocation error

9.2.7 Software Management

Any time a new or replacement driver is installed on an IOP, it is tagged experimental. The old driver is tagged obsolote and retained until the new driver is validated by the user. The next time the IOP is booted, it loads the experimental version of the driver, changes its status to suspect, and waits for the host to send a configuration validation message (see section 9.2.7.4). If the IOP does not receive confirmation within a reasonable period, it may invoke a configuration dialogue asking the user to accept, reject, or defer the suspect driver. If the user accepts the new (suspect) version, the old (obsolete) version is removed from the IOP's store and the suspect status of the new driver is changed to validated. If the user rejects the suspect version, it is removed from the IOP's store, and the obsolote tag on the original version is cleared. If the IOP boots a second time and the user neither accepts nor rejects the suspect module, the inaction constitutes an implicit rejection. The suspect version is removed and the old version reinstalled.

9.2.7.1 Downloading Software

Synopsis

```
      ioctl(fd, I2OSWDL, struct i2o_sw_xfer *sw);
          struct i2o_sw_xfer {
```

```
        u32   iop;       /* IOP unit number           */
        u8    flags;     /* DownLoadFlags field       */
        u8    sw_type;   /* Software type             */
        u32   sw_id;     /* Software ID               */
        void  *buf;      /* Pointer to software buffer */
        u32   *swlen;    /* Length of software data   */
        u32   *maxfrag;  /* Number of fragments       */
        u32   *curfrag;  /* Current fragment number   */
    };
```

This function downloads new software pointed by `sw->buf` into the permanent store or into the memory of the iop identified by `sw->iop`. The `DownloadFlags`, `SwID`, `SwType` and `SwSize` fields of the ***ExecSwDownload*** message are filled in with the values of `sw->flags`, `sw->sw_id`, `sw->sw_type` and `sw->swlen`. Once the ioctl() is called and software transfer begins, the user can read the value `*(sw->maxfrag)` and `*(sw->curfrag)` to determine the status of the software transfer. As the IOP is very slow when it comes to SW transfers, this can be used by a separate thread to report status to the user. The user should not write to this memory location until the ioctl() has returned.

Returns 0 (succeed) or –1. If an error occurs, `errno` is set appropriately:

EFAULT	Invalid user space pointer was passed
ENXIO	Invalid IOP number
ETIMEDOUT	Timeout waiting for reply message
ENOMEM	Kernel memory allocation error

9.2.7.2 Uploading Software

Synopsis

```
ioctl(fd, I2OSWUL, struct i2o_sw_xfer *sw);
    struct i2o_sw_xfer {
        u32   iop;      /* IOP unit number             */
        u8    flags;    /* Unused                      */
        u8    sw_type;  /* Software type               */
        u32   sw_id;    /* Software ID                 */
        void  *buf;     /* Pointer to software buffer  */
        u32   *swlen;   /* Length in bytes of software */
        u32   *maxfrag; /* Number of fragments         */
        u32   *curfrag; /* Current fragment number     */
    };
```

This function uploads software from the IOP identified by `sw->iop` and places it in the buffer pointed to by `sw->buf`. The `UploadFlags`, `SwID`, `SwType` and `SwSize` fields of the ***ExecSwUpload*** message are filled in with the values of `sw->flags`, `sw->sw_id`, `sw->sw_type` and sw->sw_size If the the software size is unknown, use 0 instead. IOP uses this value to verify the correct identification of the module to upload.

Once the ioctl() is called and software transfer begins, the user can read the value `*(sw->maxfrag)` and `*(sw->curfrag)` to determine the status of the software transfer. As the IOP is very slow when it comes to SW transfers, this can be used by a separate thread to report status to the user. The user should not write to this memory location until the ioctl() has returned.

Returns 0 (succeed) or –1. If an error occurs, `errno` is set appropriately:

EFAULT	Invalid user space pointer was passed
ENXIO	Invalid IOP number
ETIMEDOUT	Timeout waiting for reply message
ENOMEM	Kernel memory allocation error

9.2.7.3 Removing Software

Synopsis

```
      ioctl(fd, I2OSWDEL, struct i2o_sw_xfer *sw);
          struct i2o_sw_xfer {
              u32   iop;      /* IOP unit number                  */
              u8    flags;    /* Unused                           */
              u8    sw_type;  /* Software type                    */
              u32   sw_id;    /* Software ID                      */
              void  *buf;     /* Unused                           */
              u32   *swlen;   /* Length in bytes of software data */
              u32   *maxfrag; /* Unused                           */
              u32   *curfrag; /* Unused                           */
          };
```

This function deletes software from the permanent store of the IOP identified by `sw->iop`. The software continues to operate if it is loaded, but does not load the next time IOP is reset.The `RemoveFlags`, `SwID`, `SwType` and `SwSize` fields of the ***ExecSwRemove*** message are filled in with the values of `sw->flags`, `sw->sw_id`, `sw->sw_type` and `sw->swlen`. If the the software size is unknown, use 0 instead. IOP uses uses this value to verify the correct identification of the module to remove. Returns 0 (succeed) or –1. If an error occurs, `errno` is set appropriately:

EFAULT	Invalid user space pointer was passed
ENXIO	Invalid IOP number
ETIMEDOUT	Timeout waiting for reply message
ENOMEM	Kernel memory allocation error

9.2.7.4 Validating Configuration

Synopsis

```
    ioctl(fd, I2OVALIDATE, int *iop);
         u32 iop;
```

This function posts an ***ExecConfigValidate*** message to the IOP specified by *(`iop`). This message indicates that the host accepts the current configuration as valid. The IOP changes the status of suspect drivers to current and may delete old drivers from its store. Returns 0 (succeed) or –1. If an error occurs, `errno` is set appropriately:

ENXIO	Invalid IOP number
ETIMEDOUT	Timeout waiting for reply message

9.2.8 Events

User interface to event reporting is not yet implemented (Event handling is out of the scope of the HPGIN-project). Current idea is to use the `select()` interface to allow user applications to periodically poll the `/dev/i2octl` device for events. When `select()` notifies the user that an event is available, the user would call `read()` to retrieve a list of all the events that are pending for the specific device.

9.3 Configuration Utility

The Configuration Utility is a set of programs using the configuration API. The functionality is in the CGI programs I2O, IOPDetails, Configure, Download, Remove, Upload and Validate. The common CGI and HTML handling functions are in separate C source files, which are linked to the programs.

The following structure defines the format for IOP's software module header. It is used by the IOP software management programs Download, Remove and Upload.

```
struct I2O_MODULE_DESC_HDR {
     unsigned int   headerSize;  /* size of this header and tables   */
     unsigned short orgId;       /* I2O organization ID              */
     unsigned short moduleId;    /* assigned to vendor of module     */
     unsigned short day;         /* ascii 4 digit day DDM produced   */
     unsigned short month;       /* ascii 4 digit month DDM produced */
     unsigned int   year;        /* ascii 4 digit year DDM produced  */
     unsigned char  i2oVersion;         /* I2O version info    */
     unsigned char  majorCapabilities;  /* capbilities bits    */
     unsigned short reserved;           /* reserved */
     unsigned int   codeSize;           /* text/data/bss       */
     unsigned int   tableOffset;        /* offset to numTables */
     unsigned int   memoryReq;          /* memory requiremets  */
     unsigned int   memoryPreferred;    /* additional desired  */
     char           moduleVersion[4];   /* 4 ascii characters  */
     unsigned char  processorType;      /* IOP processor type  */
     unsigned char  processVersion;     /* IOP processor type  */
     unsigned char  objCodeFormat;      /* DDM object module format */
     unsigned char  reserved1;          /* reserved */
     unsigned int   numTables;          /* # of descriptor tables */
     char           moduleInfo[24];     /* ascii string name */
}
```

Module type numbers are mapped to user readable names via the following `moduleinfo[]` table

```
struct mod_info {
     char *moduletype;
     unsigned char value;
     char *moduledesc;
}
struct mod_info moduleinfo[NUM_MODULE_TYPES];
```

9.3.1 I2O

The I2O program generates the first page, and the page simply displays a list of IOPs. The user may choose the IOP he wishes to configure by selecting it from the list and by activating the "Configure" button. This starts the IOPDetails program and gives the IOP identification as CGI query parameter. If there is only one IOP in the system the IOPDetails program is started immediately.

9.3.2 IOPDetails

The IOPDetails program displays a page that has five buttons (Configure, Download, Upload, Remove and Validate). When the button is pressed, the corresponding program is run and user parameters are passed to it.

9.3.3 Configure

The Configure program lets the user to browse the pages the IOP provides. Only one parameter is passed to this program: the path to the first html-page. It is sent in the URL part of the query and must be composed of three numerical parts separated by '/' (slash) signs. The parts are in order the IOP, the TID and the page number. The program constructs a button that will load the Executive DDMs (TID 0) page number 0. The pages sent by the IOP will conform to this scheme.

9.3.4 Download

The Download program downloads the specified software module to the IOPs memory. It understands the following parameters:

`TargetIOP`	integer	The software is downloaded to this IOPs memory
`ModuleType`	string	The module type. Legal values are in i2o_util.c
`Source`	string	Path to the downloaded file
`LoadType`	string	Specifies whether the software should be saved into permanent memory
`OverrideMode`	string	If set to Override, the old version of the software (if any) is overwritten

9.3.5 Upload

The Upload program uploads the specified software module from the IOPs memory. The following parameters must be specified:

`ModuleType`	string	The type of the module as specified in i2o_util.c
`IOP`	integer	The software is uploaded from this IOPs memory
`SWID`	integer	This is the software module identifier
`SWVersion`	string	Currently a no-op (only in v2.0 of I2O spec)

9.3.6 Remove

The Remove program removes the specified software module from the IOPs memory. The following parameters must be specified (same as in Upload):

`ModuleType`	string	The type of the module as specified in i2o_util.c
`IOP`	integer	The software is uploaded from this IOPs memory
`SWID`	integer	This is the software module identifier

`SWVersion`	string	Currently a no-op (only in v2.0 of I2O spec)

9.3.7 Validate

The Validate program validates all suspect software modules on an IOP. There is only one parameter, which must be specified:

`iopdestvalidate`	integer	Specified the IOP whose modules are validated

For the ease of use, the IOPDetails program sets meaningful default values for all the parameters. The user is allowed to change the values within acceptable limits.

9.3.8 Common parts

The parts that are common to all programs are put in separate files. The HTML page creation and sending code is in html.c. The CGI FORM handling routines are in de-cgi.c, and I2O helper functions and tables are in i2o_util.c (not to be confused with I2O Utility Class!)

The file **html.c** contains functions to create html-page. The page structure is defined as follows

```
struct page {
     FILE *output;
     char *str;
     char type[40];
     int binary;
     int offset;
     int outputfd;
}
```

The flag `binary` defines whether end_page() function uses the `*output` pointer or `outputfd` file descriptor for writing the data. The page data is pointed to by `*str`. The type string is written in the HTTP headers. The `offset` parameter is used by hprintf() function.

```
void start_page(struct page *p)
```

This function starts a new page. If `p` is NULL, a page structure is allocated. If allocation fails, an error page is constructed and sent to the browser. If p points to a non-empty page, the function bails out. When the page is set up correctly, `p->outputfd` is set to 1, otherwise to -1.

```
void error_page(const char *format, ...)
```

This function prints out the format string and exits.

```
void change_type(struct page *p, char *type)
```

This function changes the MIME type of the page `*p`. Only first 40 characters of type argument are considered. No checking is made based on the type.

```
int hprintf(struct page *p, const char *format, ...)
```

This function writes to the page `*p`. The `format` parameter is as it would be for printf. Returns the number of characters written.

```
void write_page(struct page *p, char *data, unsigned int len)
```

This function puts the contents of `*data` buffer on the page `*p`. The length of the buffer is given in `len`. Any text written earlier is lost.

```
void empty_page(struct page *p)
```

This function makes the page *`p` empty. Any text written earlier is lost.

```
void end_page(struct page *p)
```

This function constructs the HTTP headers and adds them to the page `*p` and the page to the browser.

The **de-cgi.c** file defines the following functions.

```
void html_error(const char *error)
```

Prints out an error message `*error` and exits gracefully. This is used if a system call fails in early set up.

```
char *extract(const char *qstr, const char *var)
```

This function extracts a "variable=value" type assignment from a CGI string `*gstr`. The string may contain several such assignments and they are separated with '&' (ampersand) signs. The name of the queried variable given in `*var`. Returns the "value" part of the string, or NULL if the "variable=" part is not found or length of "value" string is 0.

```
char *decode(const char *cgistr)
```

This function replaces all the occurrences of the string '%XX', where XX is a hexadecimal number written in ASCII, with the corresponding byte value. For example, '%41' would be replaced with 'A', which is the character number 0x41.

```
char *getquery(void)
```

This function returns the query string, or NULL, if no string can be read. It assumes a CGI-style set up. The query type is given in environment variable REQUEST_METHOD and depending on type, the string is either in environment variable QUERY_STRING or can be read from stdin.

In file <u>**i2o_util.c**</u> are the helper functions.

```
unsigned char modulevalue(const char *moduletype)
```

This function searches the string `*modulevalue` from the `moduleinfo[]` table. Returns the corresponding numerical value or 0xff, if the string can not be found.

```
struct i2o_driver *getdst(int fd, int iop)
```

This function gets the Driver Store Table. The IOP number is given in `iop` and `fd` is a file descriptor of the opened /dev/i2octl character special file. Returns a pointer to i2o_driver structure (succeed) or NULL.

```
char *modinfo(struct I2O_MODULE_DESC_HDR *hdr)
```

This function adds the software module header's information to a string. Returns the modified string, or NULL (kernel memory allocation error).

10 INTERFACE TO THE PROC FILE SYSTEM

Linux uses proc file system f.g. to give information about the state of processes, kernel and hardware. It can also be used to set system parameters during the runtime. The proc file system is pure virtual file system - the directory and file entries are generated on the fly from the kernel data sctructures and process information. Detailed information about proc file system of can be found in chapter 6.3 of *Linux Kernel Internals* [3].

10.1 The /proc/i2o file hierarchy

For the I_2O subsystem, `/proc/i2o` directory tree (Figure 10.1) is added to the proc file system. It can be used to read and set IOP and DDM parameters instead of using the Configuration Utility. This allows setting I_2O device parameters, for example during boot up, simply by writing desired value to the specific proc file.

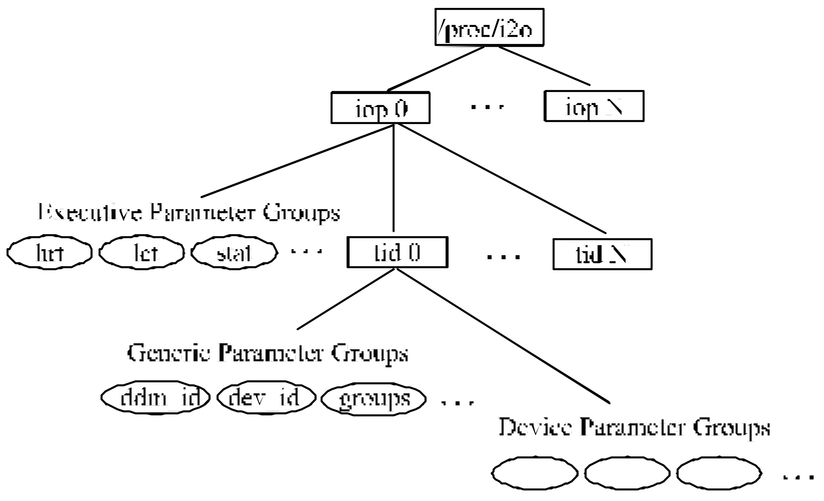

Figure 10.1. I_2O subtree in the proc file system.

Each IOP has its own directory entry (`iop0...iopN`) that contains file entries for Executive parameter groups and directory entries (`0x000...0x00N`) for devices controlled by that IOP. Each device directory has file entries for Generic Parameter Groups and Device Parameter Groups.

Executive parameter groups are defined in section 4.4.4, Generic parameter groups in section 3.4.7.6 and the device class dependent Device parameter groups in section 6 of the *I_2O Specification* [2].

10.2 Interface to the Linux kernel

The i2o_proc module contains functions to create the /proc/i2o directory tree on the fly, and functions to read from and write into these files. The struct i2o_proc_entry contains name of the file entry, its security permission mode, and pointers to functions to read from and write to that file.

```
struct i2o_proc_entry {
     char *name;
     mode_t mode;
     read_proc_t *read_proc;
     read_proc_t *read_proc;
};
```

10.2.1 Functions to read from a proc file

There are many functions to read file entries on the fly. The read functions are called whenever a user reads a file in proc file system. The functions gather IOP and DDM information by using functions i2o_query_scalar() and i2o_query_table() from i2o_core module. There is also a function to read LAN OSM parameters from its private structure (struct i2o_lan_local).

Functions to read from a file in proc file system have the form

```
int (read_proc_t)(char *page, char **start, off_t off, int count,
                  int *eof, void *data);
```

Read functions' parameters are

- A pointer to the memory page containing the virtual file (page),
- a pointer to the pointer of start of the file (start),
- an offset from the start (off),
- the number of data read from the file (count),
- an indicator if we are already at the end of file (eof), and
- a pointer to the i2o_device or i2o_controller structure (data).

10.2.2 Functions to write into a proc file

There are also a few functions to set device parameters according to the data written into a file in proc file system. The write functions uses function i2o_set_scalar()(from i2o_core module) to set the IOP or DDM parameter. For example, setting MAC address of an I_2O LAN device with TID 8 on the first IOP is done by writing new MAC address value to file /proc/i2o/iop0/0x008/lan_mac_addr. The given value is then sent to IOP.

Functions to write to a file in proc file system have the form

```
int (write_proc_t)(struct file *file, const char *buf,
                   unsigned long count, void *data);
```

Write functions' parameters are

- a pointer to the file we are writing to (file),
- a pointer to the buffer where our data to be written is (buf),
- the number of data written to the file (count), and
- a pointer to the i2o_device or i2o_controller structure (data).

10.3 Generating /proc/i2o directory tree

The proc file system is initialized by calling function i2o_proc_init(). It calls function create_i2o_procfs(), which calls proc file system interface function proc_mkdir() to create directory /proc/i2o. Then all IOPs are added to directory /proc/i2o using function i2o_proc_add_controller(). That function adds generic IOP information files (Table 10.1) using function i2o_proc_create_entries(), and a directory for all devices controlled by that IOP. The directories for devices are named after the Target ID of the device, 0x000 being the Executive DDM. For all devices generic_dev_entries (Table 10.2) are added. There are also generic entries for LAN class devices (Table 10.3), and entries for FDDI, Token Ring, and Ethernet statistics (Table 10.4).

The function i2o_proc_create_entries() goes through a list of struct i2o_proc_entry and adds new entries to that directory using proc file system interface function create_proc_entry().

The /proc/i2o directory is removed from proc file system calling exit function of the module, cleanup_module(). It calls function destroy_i2o_procfs(), which in turn calls for each IOP i2o_proc_remove_controller() and finally proc file system interface function remove_proc_entry() to remove directory /proc/i2o. For each IOP all generic device entries and class dependant entries are removed (in function i2o_proc_remove_entries()), and then the directory itself is removed using proc interface function remove_proc_entry(). After that generic IOP entries and finally the

IOP directory is removed and the proc entry is marked as `NULL` in `struct i2o_controller`.

File	Parameter group
`hrt`	0100h – Hardware Resource Table
`lct`	0102h – Local Configuration Table
`status`	N/A – Status Block
`hw`	0000h – IOP hardware
`ddm_table`	0003h – Executing DDM List
`driver_store`	0004h – Driver Store
`drivers_stored`	0005h – Driver Store Table

Table 10.1 Generic IOP entries.

File	Parameter group
`groups`	F000h – Params descriptor
`phys_dev`	F001h – Physical Device
`claimed`	F002h – Claimed Table
`users`	F003h – User Table
`priv_msgs`	F005h – Private Message Extensions
`authorized_users`	F006h – Authorized User Table
`dev_identity`	F100h – Device Identity
`ddm_identity`	F101h – DDM Identity
`user_info`	F102h – User Information
`sgl_limits`	F103h – SGL Operating Limits
`sensors`	F200h – Sensors

Table 10.2 Generic device entries.

File	Parameter group
`lan_dev_info`	0000h – Device Info
`lan_mac_addr`	0001h – MAC Address Table
`lan_mcast_addr`	0002h – Multicast MAC Address Table
`lan_batch_control`	0003h – Batch Control
`lan_operation`	0004h – LAN Operation
`lan_media_operation`	0005h – Media Operation
`lan_alt_addr`	0006h – Alternate Address

`lan_tx_info`	0007h – Transmit Info
`lan_rx_info`	0008h – Receive Info
`lan_hist_stats`	0100h – LAN Historical Statistics
	0180h – Supported Optional Historical Statistics
	0182h – Optional Non Media Specific Transmit Historical Statistics
	0183h – Optional Non Media Specific Receive Historical Statistics
`settings`	N/A – Settings for the LAN OSM and DDM (see Table 10.5)

Table 10.3 Generic LAN entries.

File	Parameter group
`lan_eth_stats`	0200h - Required Ethernet Statistics
	0280h - Supported Ethernet Historical Statistics
	0281h - Optional Ethernet Historical Statistics
`lan_tr_stats`	0300h - Required Token Ring Statistics
`lan_fddi_stats`	0400h - Required FDDI Statistics

Table 10.4 LAN subtype specific statistics.

10.4 Reading parameter group information

Functions to get information about IOP and DDMs are named as `i2o_proc_read_<entry>`. For example function `i2o_proc_read_lan_batch_ctrl()` (see below) is used `to query` LAN parameter group 0x0003 (Lan Batch Control) and generate file `lan_batch_ctrl`. The function calls `i2o_query_scalar()` to retrieve information about parameter group 0x0003 and prints information found from result buffer to the buffer buf. Querying the parameter group is locked using a spin lock to prevent other processes to modify the information at the same time.

```
i2o_proc_read_lan_batch_ctrl(char *buf, char **start, off_t offset, int len,
                             int *eof, void *data)
{
     struct i2o_device *d = (struct i2o_device*)data;
     struct i2o_lan_batch_control_scalar result;
     int token;

     spin_lock(&i2o_proc_lock);
     len = 0;
     token = i2o_query_scalar(d->controller,    // IOP
                              d->lct_data->tid, // TID
                              0x0003,           // Parameter group #
                              -1,               // Query all values
                              &result,          // Results are here
                              9*4);             // Size of result buffer
     if (token < 0) {
         len += i2o_report_query_status(buf+len, token,
                                        "0x0003 LAN Batch Control");
         spin_unlock(&i2o_proc_lock);
         return len;
     }

     len += sprintf(buf, "Batch mode ");
     if (result.batch_flags&0x00000001)
         len += sprintf(buf+len, "disabled");
     else if (result.batch_flags&0x00000004)
         len += sprintf(buf+len, "enabled");
     else {
         len += sprintf(buf+len, "automatic");
         if (result.batch_flags&0x00000002)
             len += sprintf(buf+len, " (on)");
         else
             len += sprintf(buf+len, " (off)");
     }
     len += sprintf(buf+len, "\n");

     len += sprintf(buf+len, "Max Rx batch count : %d\n",
                    result.max_rx_batch_count);
     len += sprintf(buf+len, "Max Rx batch delay : %d\n",
                    result.max_rx_batch_delay);
     len += sprintf(buf+len, "Max Tx batch delay : %d\n",
                    result.max_tx_batch_delay);
     len += sprintf(buf+len, "Max Tx batch count : %d\n",
                    result.max_tx_batch_count);

     spin_unlock(&i2o_proc_lock);
     return len;
}
```

10.5 Setting LAN OSM and DDM parameters

Some LAN OSM and DDM parameters can be read and set using the proc file system (see also chapter 8.1.10.). The parameters that can be set using the proc file are located in file entry /proc/i2o/<iop>/<tid>/`settings` (see Table 10.5).

Setting	Definition	Values
`max_buckets_out`	Maximum number of buckets sent to DDM	1 –
`bucket_thresh`	Send more buckets to DDM when this many used	1 –
`rx_copybreak`	Maximum size of received packet that is copied to new socket buffer	1 – MTU
`err_reporting`	Whether errors are sent to OSM or handled by DDM	0 – DDM 1 – OSM
`tx_batch_mode`	OSM batching	0 – off 1 – on 2 – automatic
`rx_batch_mode`	HDM batching	0 – off 1 – on 2 – automatic
`event_mask`	Event mask for receiving events from DDM	0x00000000 – 0xFFC00002
`tx_timeout`	Timeout for softnet watchdog timer	

Table 10.5 Settings for LAN OSM and DDM.

Reading the entry `settings` lists all parameter names, their current value, minimum and maximum values, and the read/write mode. The OSM parameter values are located in LAN device's private structure `struct i2o_lan_local`. DDM parameters are located on the I/O platform, and they are read by querying the corresponding entry in the parameter group.

New values to the above mentioned parameters are set by echoing the name and the value to the file. For example setting the batch mode for sending packets on for the LAN device with TID 0x008 is done as follows.

```
gin$ echo "tx_batch_mode:1" > /proc/i2o/iop0/0x008/settings
```

11 ERROR HANDLING

The IOP and its DDMs report from errors via the message replies. Each reply has a status code (`ReqStatus`) and a detailed status code (`DetailedStatusCode`). The status code may imply that the request was completed normally, the request was aborted (e.g. because of timeout), there was an error in execution, or that the reply is just a progress report.

Message replies and possible errors are handled by module reply handlers, and they may decide how to handle the error. Handlers may

- simply discard the reply e.g. in case the requestor is not interested of the reply or because there is already sufficient error control in upper levels (as e.g. in networking),
- process the situation, e.g. by using Executive or Utility class messages, or
- pass the problem to the requesting function, which will handle the situation or report upwards if needed.

If status code implies an error, the detailed status code may be inspected to get a more exact description. There are detailed status codes among others for malformed messages, invalid values, missing parameters, overflows etc. Values for `DetailedStatusCode` are defined by the particular message class and message function. Reply status codes and detailed status codes for Executive class, DDM class, Utility class, and TransactionError replies are specified in Table 3-2 in the *I_2O Specification* [2]. Detailed status codes for the other OSMs are specified in the respective chapters in the *I_2O Specification* [2].

If the request message was a multiple transaction request, the error reply is repeated for each transaction that the target rejects.

When a request cannot be at all delivered to the target, a ***FaultNotification*** reply (see Figure 3-7 in the *I_2O Specification* [2]) is returned to the initiator of the failed request. The reply details why the message could not be delivered, and contains also the original request. When the Messenger can't deliver reply to the initiator, there is no mechanism to reply to it, so the failing module creates and sends an ***UtilReplyFaultNotify*** request message (see Figure 6-18 in the *I_2O Specification* [2]).

The *Configuration Utility* or the *`/proc` file system* can be used to query and adjust various error and control parameters in device parameter groups, e.g. timeouts and timeout policy, number of retries, recovery etc. Groups of generic parameters are defined for all device class and each message class defines additional parameter groups.

12 TESTING

The test specification is now available as a separate document titled *HPGIN-TEST HPGIN-Linux Test Specification.*