An Upward Compatible Ethernet Device Driver for Bare PC Applications

Faris Almansour
College of Business Administration
American University in the Emirates
Dubai, UAE
faris.almansour@aue.ae

Ramesh Karne
Department of Computer and Information Sciences
Towson University
Towson, MD, US
rkarne@towson.edu

Alexander Wijesinha
Department of Computer and Information Sciences
Towson University
Towson, MD, US
awijesinha@towson.edu

Rasha Almajed
College of Computer Information Technology
American University in the Emirates
Dubai, UAE
rasha.almajed@aue.ae

Hamdan Alabsi
Department of Business, Mathematics and Science
Bemidji State University
Bemidji, MN, US
hamdan.alabsi@bemidjistate.edu

***Abstract*—An upward compatible device driver is a driver that can be upgraded to work with future versions of the hardware with minimal changes. We consider upward compatibility of network interface controller (NIC) drivers for bare machine computing (BMC) applications that run without the support of any operating system (OS) or kernel. Specifically, we describe an approach for designing an upward compatible gigabit Ethernet Intel NIC device driver for BMC applications. Compared to similar Windows NIC drivers, the BMC driver has fewer external calls and a smaller binary executable. The same approach can be used to design and develop upward compatible BMC device drivers for NIC hardware with similar chip sets.**

Keywords—bare machine computing, bare PC, NIC, Ethernet device driver, upward compatibility, platform independence

I. Introduction

Bare machine computing (BMC) systems enable computer applications to run on a bare machine without an operating system (OS) or kernel. Currently, BMC applications run on an Intel CPU-based ordinary desktop or laptop with no other software support. In principle, these applications can be modified to also run on other CPU architectures and devices. The BMC concept differs from minimal kernels such as the minimal Rust kernel [1], microkernels such as RIOT [2], and library OSs such as Graphene [3] in that there is no additional OS/kernel software needed to run the completely self-supporting BMC applications. They are also not the same as embedded systems since BMC systems are used to run Web servers [4], email servers [5], high performance servers [6], SQLite databases [7], VoIP clients [8], and network middleboxes [9].

BMC systems are application-centric and have several characteristics that are useful from a security viewpoint. For example, there are no OS-related vulnerabilities that can be used to compromise BMC applications; there are no dynamic-link library (DLL) vulnerabilities-BMC applications are statically compiled; it is easier to analyze the code and find (and fix) security vulnerabilities-BMC applications are simpler and have a smaller code size; there is no permanent storage such as a hard drive storing valuable resources that can be corrupted-BMC applications are booted from and store their data (if needed) on a secured removable medium such as an external drive; it is not possible to execute BMC applications with elevated privileges-all the BMC code runs as a single monolithic executable in user mode; and there is no support for additional functionality that attackers can exploit-only the hardware/software interfaces and network protocols (API) that are necessary for a given BMC application to run are included in a BMC application. Conventional (non-BMC) applications can also be designed to have many of the above security features. However, they usually require the support of some form of an OS or kernel.

Upward compatible NIC drivers enable an existing driver to be used with a new version of the hardware by making a few changes to the code [10]. In this paper, we present the architecture, software design, and implementation of a BMC gigabit Ethernet Intel network controller (NIC) device driver that is upward compatible. We then show how this upward compatible BMC NIC driver can be modified to work with several gigabit Ethernet Intel NICs. We also provide internal details of the driver that would be useful for developing upward compatible OS-independent NIC drivers in the future.

NIC drivers are frequently updated or rewritten to work with new NIC hardware because of advances in technology. By reusing existing NIC driver code, which is a goal of upward compatible drivers, the development effort and the potential for introducing errors and security vulnerabilities in the driver are reduced. In general, upward compatibility is not easily achieved since features of new NIC hardware are not known until the hardware becomes available. In special cases, such as when the new NIC has a similar design, it is possible to modify the existing NIC driver without rewriting it.

The rest of the paper is organized as follows. Section II discusses related work. Section III describe an Ethernet device driver design that is simple and upward compatible. Section IV demonstrates the upward compatibility of the driver, using different NICs and PCs/laptops. Section V describes Windows based NIC drivers and their complexity related to a bare PC NIC driver. Section VI provides insights into NIC driver design and implementation. Section VII contains the conclusion.

II. Related Work

The upward compatibility of Ethernet NIC drivers is not typically supported in OS-based systems. Instead, a legacy

mode option may be available to allow an older driver to be used with a newer NIC [11][12] at the expense of losing some of its design features. The network driver interface specification (NDIS) enables NIC drivers to be portable across platforms that support the Windows OS [13]. NDIS can also serve as a wrapper allowing other OSs such as Linux and FreeBSD to use the NIC. However, it still requires vendor-specific NDIS API drivers. Several samples that illustrate how to write NIC drivers in a Windows environment are given in [14]. There are many sites that provide information on writing NIC drivers in Linux. An overview of network interfaces in Linux is given in [15]. Data Plane Development Kit (DPDK) drivers operate in user space and directly access the NIC thus avoiding Linux kernel overhead [16]. The implementation and performance of a small simple Linux user space driver for the Intel ixgbe family of NICs is presented in [17]. Previous work on BMC NIC drivers investigated Ethernet bonding with dual NICs [18].

III. NIC DRIVER

Conventional NIC drivers operate with an underlying OS or kernel in a general-purpose computing environment. Such drivers vary depending on platform, vendor and CPU architecture. Writing a conventional NIC driver thus requires knowledge not only of the underlying CPU architecture, the hardware interfaces and the software specification for the driver, but also of the OS environment. As the platform, hardware, software and technology change rapidly, NIC drivers are not usually designed to be upwardly compatible. For example, a variety of Ethernet drivers evolved over the years in a heterogeneous manner without being upward compatible. However, OS-independent Ethernet drivers are universal in nature for the most part enabling their architecture to be unified and standardized to eliminate heterogeneity. For example, BMC NIC device drivers are designed for an underlying CPU instruction set architecture rather than for a given OS or platform. It is then possible to provide a simple API for applications enabling the NIC driver to directly communicate to the underlying hardware. In this case, the entire NIC driver code is integrated with the rest of the TCP/IP protocols as part of the BMC application. Unlike in an OS-based system, there is no other code or kernel running in the BMC system to enable the driver to work.

A. Architecture

The general architecture for a BMC NIC driver is illustrated in Fig. 1. The application uses the NIC driver API that directly communicates with the host controller (HC) in a BMC architecture. The HC in turn controls the NIC hardware. PC BIOS interrupts are used to obtain a device address for the NIC. The software application and the NIC hardware communicate through user memory to exchange data, commands and status information. The HC acts as middleware enabling communication between the software application and the hardware. Fig. 2 shows a typical kernel mode OS-based environment. A user mode OS-based environment is similar except that the NIC driver is in user space. In both cases, the NIC driver and application require the OS/kernel to communicate with the NIC hardware. As shown in the figure, the user application does I/O by using system calls provided by the underlying OS/kernel. The NIC and the NIC driver use shared memory to communicate, which is under the control of the OS/kernel and not visible to application programmers. BIOS interrupts are used to obtain the NIC device address. BMC uses an API (Fig. 1), whereas the OS communicates with the BIOS (Fig. 2). Design and development of the BMC driver is based on the NIC specification [11]. The data structures required for communicating to the NIC hardware reside in shared memory.

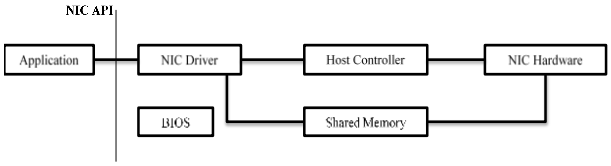

Fig. 1. BMC based architecture.

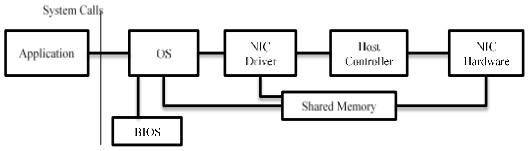

Fig. 2. OS based architecture (kernel mode).

TABLE I. INTERFACES

AOAgetShared Mem()	setBaseAddress()
getMACEPROM(SrcMAC)	ColdeReset()
Inittest(Data Structure Parameters)	TEnable()
REnable()	ReadData()
getMAC()	IPInsertPkt()
ARPInsertPkt()	sndCall()
getTDTail()	getRDTail()
setTDTail()	setRDTail()
IncSendInPtr()	initRDL()
incSendOutPtr()	initTDL()
TDLfull()	RDLfull
setRegister()	getRegister()
IsRdescDone()	isTdescDone()

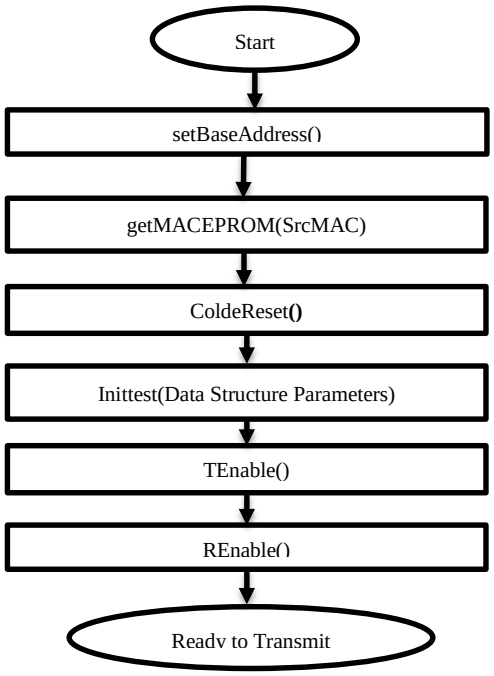

Fig. 3. Sequence of operations.

```
int EtherObj::setTDTail(int  tailindex)
{
int retcode=0;
int temp=0;
// TDT  0x3818
setRegister(0x3818, IOADDR);
//mask higher 16 bits
temp = tailindex;
temp = temp & 0x0000ffff;
setRegister(temp, IODATA);
return retcode;
}
```

Fig. 4. Interface SetTDTail.

```
int EtherObj::IPInsertPkt(char* PACK, int PACK_Size, int
PROTOCOL, char* Target_HAdd, int CurrentTask)
{
 // variables not included …..
x = TDLPointer + SendInPtr * 16 - ADDR_OFFSET;
    //TDL address is absolute
//……
PACK = PACK - 14;
   p1 = (long*)x; // p1 is address of tdlPointer
//….error code not included
SendInPtr++;
if(SendInPtr == NO_OF_TDL)
   {
    SendInPtr = 0; //circular list
   }
  p1++;
  p1++;
   sizeOfPacket = PACK_Size + 14;
 for(i=0; i< 6; i++)
  {
    PACK[i] = Target_HAdd[i];
   PACK[i+6] =  mac[i];
 }
    PACK[12] = ((PROTOCOL>>8) & 0x00FF);
    PACK[13] = ((PROTOCOL) & 0x00FF);
  temp = *p1;
  temp = temp & 0xffff0000;
  temp = temp + sizeOfPacket;
 *p1 = temp;  //TDL entry for length
  retcode = sndCall();
  return 0;
}
```

Fig. 5. Interface IPInsertPkt.

B. Interfaces

The driver requires about 150 interfaces in the API. About half of these interfaces are used to debug the driver and some are internal function calls. They include some C++ interfaces used by the BMC application and two assembly calls as shown in Table I. A total of 26 interfaces (API) are sufficient to build BMC applications. The two assembly call interfaces are used to access control registers of the NIC [11]. Fig. 3 shows a sequence of operations that are needed before the NIC is ready to receive and send data. An example use of an assembly interface is shown in Fig. 4. Two assembly calls setRegister() and getRegister() are sufficient to write this device driver in C++. In this example, we are writing to a transmit descriptor tail register, which has a control register address of 0x3818 [11]. Writing values to a register requires two steps. In the first step, we select the register using IOADDR and in the second step we write to the register using IODATA. Reading from a register is also similar, except we do setRegister() in the first step and readRegister() in the second step.

TABLE II. TRANSMIT CONTROL REGISTER

Bit No.	Field
0	Reserved
1	Transmit Enable
2	Reserved
3	Pad Short Packets
4 : 11	Collision Threshold
12 : 21	Collision Distance
22	Software XOFF Transmission
23	Reserved
24	Re-transmit on Late Collision
25	No Re-transmit on underrun
26 : 31	Reserved

TABLE III. RECEIVE CONTROL REGISTER

Bit No.	Field	Bit No.	Field
0	Reserved	16 : 17	Receive Buffer Size
1	Receiver Enable	18	VLAN Filter Enable
2	Store Bad Packets	19	Canonical Form Indicator Enable
3	Unicast Promiscuous Enabled	20	Canonical Form Indicator bit value
4	Multicast Promiscuous Enabled	21	Reserved
5	Long Packet Reception Enable	22	Discard Pause Frames
6 : 7	Loopback mode.	23	Pass MAC Control Frames
8 : 9	Receive Descriptor Minimum Threshold Size	24	Reserved
10	Reserved	25	Buffer Size Extension
12 : 13	Multicast Offset	26	Strip Ethernet CRC from incoming packet
14	Reserved	27 : 31	Reserved
15	Broadcast Accept Mode.		

Fig. 5 illustrates IPinsertPkt() interface as used in a BMC application. In order to insert a packet, we get the transmit ring slot to be inserted, form the packet, insert the packet in the descriptor, and call sndCall(). The sndCall() function will read the tail register, increment it and set up the pointer for transmission. In BMC, the entire process of sending an IP packet is controlled by the application program. The details of driver operation are hidden in the IPinsertPkt() function. Similarly, all other interfaces can be controlled by the application program and the details are encapsulated in the interface object.

The device driver designed and implemented as described above is for the NIC in a Dell Optiplex 260. With a few changes, it can be used in other Dell PCs and in a HP laptop. Specifically, these changes involve two control registers. The TEnable and REnable functions are used to set control registers 0x400 and 0x100 respectively. Table II and Table III show the transmit and receive control register fields that will be discussed in the next section.

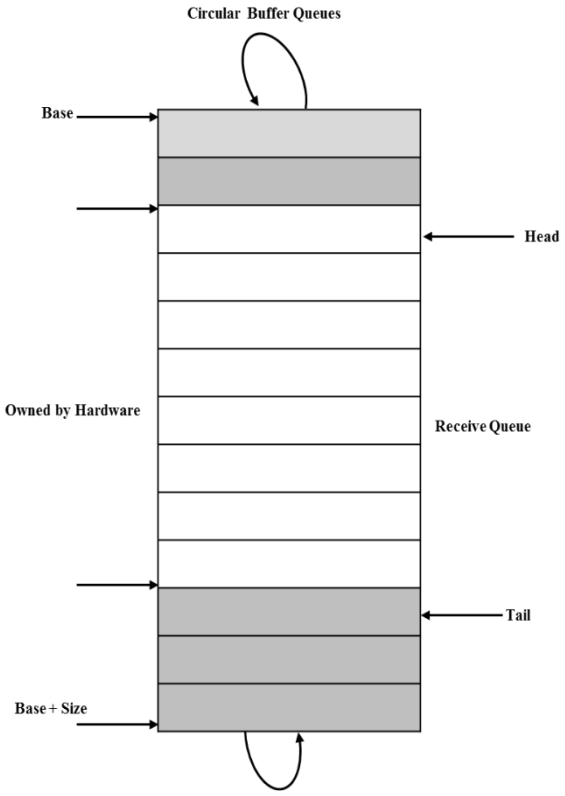

Fig. 6. Descriptor ring data structure.

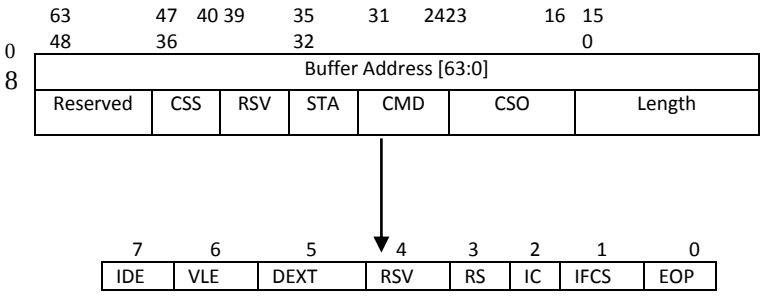

Fig. 7. Transmit descriptor (TDL).

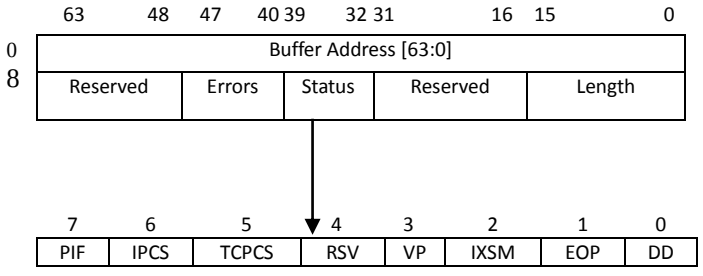

Fig. 8. Receive descriptor (RDL).

C. Software Design

The software design of the Ethernet driver depends on the internal data structures of the Ethernet hardware [11] and the type of NIC. The Intel NIC hardware defines transmitter and receiver descriptor rings. A descriptor ring as shown in Fig. 6 consists of a set of descriptors organized as a circular list [11]. It has a head and tail to indicate the beginning and end of a ring, and IN and OUT pointers for each ring. Each descriptor in this design consists of a 16-byte data structure. There can be up to 4096 descriptors and a minimum of 8 is required for this architecture. A transmit descriptor (16 bytes) is shown in Fig. 7 [11]. It has 8 bytes of packet address (64 bit address) and 8 bytes of control. Control fields include Length (2 bytes of packet length), CSO (1 byte), CMD byte, STA (status 4 bits TU, LC, EC, DD) [11], RSV (4 bits reserved), CSS (1 byte) and Special (2 bytes). Similarly, a receive descriptor (16 bytes) is shown in Fig. 8 [11]. It consists of 8 bytes of packet address and 8 bytes of control information. The status field consists of 8 bits (PIF, IPCS, TCPCS, RSV, VP, IXSM, EOP, DD) and Error byte (RXE, IDE, TCPE, CXE, RSV, SEQ, SE, CE). Some of the fields shown here are reserved and not used in a given driver.

The NIC driver designed here uses polling instead of interrupts. Polling is convenient in BMC applications as the control flow always returns to the main task, which is idle most of the time. When a packet arrives from the network, NIC hardware places it in the receive descriptor and sets the status bit DD in the descriptor. During the polling process, a packet is received by an application by checking the DD bit in the receive descriptor status field. The "isDescDone()" function will perform this polling for receiving packets after the initialization process as shown in Fig. 3 is complete, Similarly, the "incSendOutPtr()" function will perform polling of sent packets to acknowledge the transmitted packet. When a packet is ready to be sent, it is simply placed in the transmit ring and control returns to its caller. If the transmit or receive butter is full, the resulting error will stop the machine. Many other error conditions are also detected and resolved as needed.

D. Implementation

The Ethernet device driver is implemented as a single class with EtherObj.cpp and EtherObj.h files. In addition, it has two assembly functions consisting of 26 lines of code. The C++ code in this driver is about 3600 lines including comments and its object file size is 37 KB. The driver is compiled using the Microsoft Visual Studio 12.0 C++ compiler. When this compiler is used, no system libraries or include files are used from this OS based compiler (/NODEFAULTLIB option).

IV. Upward Compatibility of the Driver

In OS based systems, vendors write device drivers as the platforms change, regardless of whether external NICs or internal onboard chips are used. However, each vendor chip is different in its hardware and software specifications causing heterogeneity. Also, newer models added new facilities and features without a concern for upward compatibility. This resulted in different device drivers for different NICs.

Initially, the BMC gigabit Ethernet NIC device driver was designed for an onboard Intel chip 82540EM, which was in a Dell Optiplex 260 desktop. This driver was used in a BMC Web server application and many other applications that ran on this desktop. Eventually, the Dell Optiplex 260 model became obsolete and was replaced by newer models. To deal with this issue, we originally replaced the onboard Intel NIC with an equivalent external NIC. Later, based on our experience with writing NIC drivers and other drivers for BMC applications [19], we began to investigate upward compatibility as an alternate solution to deal with NIC evolution. The rest of this section discusses upward compatibility of the above BMC NIC device driver. This driver is referred to as D1.

A. Dell Optiplex 960

The Dell Optiplex 960 model has an internal gigabit Ethernet NIC Intel 82567. The D1 driver was upgraded so that it works with this model by making the changes **(1)-(3)** below. The modified driver is referred to as D2. **(1)** To obtain the device address, the device id (0x10de) was changed for the BIOS interrupt call. **(2)** The TEnable function that uses the transmit control register (0x400) [20] was changed as shown below. Fig. 7 gives details of the register fields. In D1, the transmit control register was initialized to 0x002000f2. The reserved bit (28) is 0, collision distance bits (21-12) are 0x200, and bits 29, 28 are 0s. In D2, the reserved bit 28 must be 1 [20], collision distance bits (21-12) are 0x3f, and read request threshold bits (30, 29) are 01. In D2, the reserved bit setting is strictly enforced by the device hardware and it is 1 instead of 0. We kept more threshold although the read request threshold does not have much impact on the driver. Whereas the collision distance in D1 is 0 and it is not used, the collision distance in D2 is very sensitive as the processor is faster and it appears that there are more collisions. **(3)** The getMACEPROM() function does not work in D2. This is because the Optiplex 260 uses ICH4 and the EERD [20] register to read the MAC from the EPROM, while the Optiplex 960 uses ICH9 and the EERD register is not available as per our knowledge. We addressed this problem by reading the MAC address using the resource address obtained from a Windows machine for this model. For the machine under test, the resource address was 0xfe6e0000 (each machine may have a different resource address. This memory mapped address was used to read the MAC address from memory.

B. Dell Optiplex 9010 and HP Elite Book 8460P

The Dell Optiplex 9010 models have the internal gigabit Ethernet NIC Intel 82579LM. The D1 driver was upgraded to work with this model by making the changes **(1)-(3)** below. This driver is referred to as D3. **(1)** To obtain the device address, the device id (0x1502) was changed for the BIOS interrupt call. **(2)** The TEnable function that uses the transmit control register (0x400) [21] was changed as shown below. In D1, the transmit control register was initialized to 0x002000f2. The reserved bit (28) is 0, collision distance bits (21-12) are 0x200, and bits 29, 28 are 0s. In D3, the reserved bit 28 must be 1 [21], collision distance bits (21-12) are 0x3f, and the read request threshold bits (30, 29) are 01. These changes are the same as those in D2. The REnable function that uses the receive control register (0x100) was changed. The code for D1 was 0x04408002 and bit 22 (discard pause frames) is 1. In the Optiplex 9010, this bit is reserved and it must be 0. The new code is 0x04008002. **(3)** The getMACEPROM() function does not work in D3. As noted earlier, the Optiplex 260 uses ICH4 and thr EERD register to read the MAC from the EPROM. The Optiplex 9010 does not use the ICH controller, but uses instead the Intel 6 series chipset [20], which different from the ICH architecture. Similarly, for the HP Laptop, the chip is different (Intel QM67) [21]. We addressed this problem by reading the MAC address using the resource address obtained from a Windows machine for this model. For the machine under test, the resource address was 0xf7do0000 (for the Optiplex 9010) and 0xd480000 (for the HP Laptop). This address can be used to read the MAC address from memory. Note that the NICs in the 9010 and the laptop are essentially the same except for their resource addresses.

TABLE IV. DRIVER CHARACTERISTICS

	OPTIPX 260	**OPTIPX 960**	**OPTIPX 9010**	**BMC**
DRIVER NAME	E1g60i32	E1k6232	E1c62x64	EtherO
BINARY SIZE	116 KB	220 KB	484 KB	36 KB
EXTERNAL CALLS	71 calls	89 calls	------	26 calls
MODULE DPNDCIES	3	3	3	0
DPNDNT MODULE CALLS	ntoskrnl-14 hal -7 ndis-5	ntoskrnl-32 hal-6 ndis-51	ntoskrnl hal ndis	0

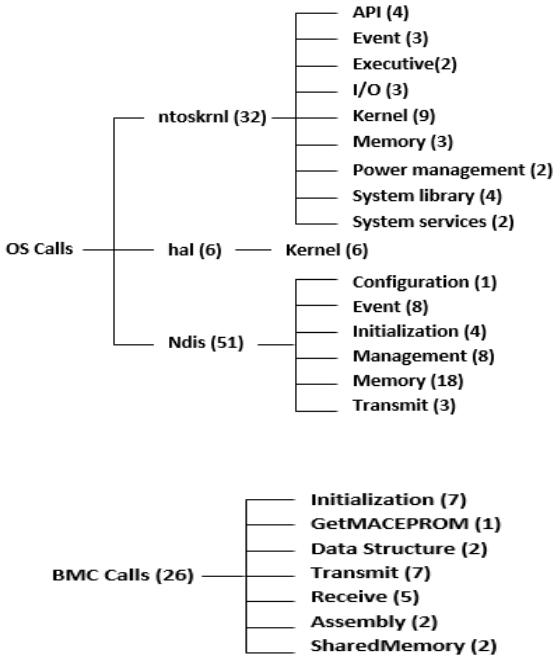

Fig. 9. Classification of interfaces.

V. WINDOWS BASED NIC DRIVERS

This section describes some characteristics of Windows based Ethernet NIC drivers. We use the same machines and Ethernet NICs as before (Optiplex 260 (O1), Optiplex 960 (O2), and Optiplex 9010/HP Laptop (O3)), and the related driver binary executables for the Windows OS. Each driver interfaces with the NDIS, Kernel and HAL (hardware abstraction layer) modules, and none of these drivers can operate in standalone mode. Table IV compares some

characteristics of the Windows drivers obtained by using the PE Explorer tool [22] with those of the BMC driver.

Fig. 9 classifies interfaces for the Windows drivers and the BMC driver. Compared to the Windows drivers, the BMC driver has simplicity, smaller binary size, fewer essential interfaces, and no dependencies on external modules. In addition, the BMC driver directly interfaces with the application program and the NIC. The Windows drivers are not upward compatible, whereas the BMC driver is upward compatible with a few changes made in the control registers. These control register changes were needed as the NIC design does not directly support upward compatibility of drivers.

VI. Discussion

Ethernet drivers have common architectural features and characteristics such as transmit and receive data structures, initialization steps, control of transmit and receive facilities, and status and command controls. This commonality can be exploited by designing upward compatible drivers that can be modified with a few changes to support new NIC hardware. Yet, driver design and driver interfaces are different for different NICs, NIC vendors, and OS platforms. For example, as noted earlier, the Windows NDIS library provides a standard interface for NICs but does not address driver heterogeneity. Similarly, OS-based drivers for gigabit Ethernet Intel NIC families are not designed to be upward compatible. However, we observed that it is possible to enhance NIC driver functionality while retaining their upward compatibility characteristics using reserved bits in some cases. The architecture and design of upward compatible BMC Ethernet Intel NIC drivers provide insight into designing standardized Ethernet NICs and OS-independent drivers that eliminate heterogeneity.

VII. Conclusion

This paper presents the design and implementation of a novel OS-independent Ethernet Intel NIC device driver that is simple, small and upward compatible. It shows how the same driver code can be used with similar Ethernet Intel NICs by making a few changes in the control registers. The BMC driver design is compared with existing OS based drivers by comparing driver characteristics and classifying their interfaces. While upward compatible BMC NIC drivers can address NIC heterogeneity to some extent, a better solution is for vendors to specify a standard architecture for NICs in the future.

References

[1] P. Oppermann, A Minimal Rust Kernel, https://os.phil-opp.com/minimal-rust-kernel/, Accessed: Jan 2020.

[2] RIOT: The Friendly Operating System for the Internet of Things, https://www.riot-os.org/, Accessed: Jan 2020.

[3] Graphene-a Library OS for Unmodified Applications, https://grapheneproject.io/, Accessed: Jan 2020.

[4] L. He, R. K. Karne, and A. L. Wijesinha, The design and performance of a bare PC Web server, International Journal of Computers and Their Applications, IJCA, Vol. 15, No. 2, June 2008, pp. 100-112.

[5] G. H. Ford, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, The design and implementation of a bare PC email server, 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC), 2009, pp. 480-485.

[6] B. Rawal, R. K. Karne, and A. L. Wijesinha. Mini Web server clusters for HTTP request splitting, IEEE Conference on High Performance, Computing and Communications (HPCC), 2011, pp. 94-100.

[7] W. Thompson, R. K. Karne, and A. L. Wijesinha, Interoperable SQLite for a Bare PC, 13th International Conference Beyond Database Architectures and Structures (BDAS'17), 2017, p177-188.

[8] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, A peer-to-peer bare PC VoIP application, 4th IEEE Consumer Communications and Networking Conference (CCNC), 2007, pp. 803-807.

[9] A. K. Tsetse, A. L. Wijesinha, R. K. Karne, and A. Loukili, A 6to4 Gateway with Co-located NAT, IEEE International Conference on Electro Information Technology (EIT), 2012.

[10] J. Spacey, Backward Compatibility vs Forward Compatibility, https://simplicable.com/new/backward-compatibility-vs-forward-compatibility, Accessed: Jan 2020.

[11] Intel; PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual, https://pdos.csail.mit.edu/6.828/2006/readings/hardware/8254x_GBe_SDM.pdf, Accessed: Jan 2020.

[12] Intel®ICH8/9/10and82566/82567/82562V:Developers Manual, http://www.intelcore.mx/content/www/uk/en/embedded/products/networking/i-o-controller-hub-8-9-10-82566-82567-82562v-software-dev-manual.html, Accessed: Jan 2020.

[13] Portability in Network Drivers, https://docs.microsoft.com/en-us/windows-hardware/drivers/network/portability-in-network-drivers, Accessed: Jan 2020.

[14] Networking Driver Samples, https://docs.microsoft.com/en-us/windows-hardware/drivers/samples/networking-driver-samples, Accessed: Jan 2020.

[15] Greg Kroah-Hartman, Linux Device Drivers 3rd Edition, O'Reilly, https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch17.html, Accessed: Jan 2020.

[16] R. Rosen, User Space Networking with DPDK, Linux Journal April 23, 2018.

[17] P. Emmerich et. al, User Space Network Drivers, ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2019.

[18] F. Almansour, R. K. Karne, A.L. Wijesinha, and B. Rawal, Ethernet Bonding on a Bare PC Webserver with Dual NICs, 33rd ACM/SIGAPP Symposium on Applied Computing (SAC), 2018.

[19] R. K. Karne, A. L. Wijesinha, and S. Liang, A Bare PC Mass Storage USB Driver, International Journal of Comptuers and Their Applications, March 2013.

[20] Intel® I/O Controller Hub 8/9/10 and 82566/82567/82562V Software Developer's Manual.

[21] Intel® 82579 Gigabit Ethernet PH. https://www.mouser.com/pdfdocs/82579datasheetvol21.pdf, Accessed: Jan 2020.

[22] PE. Explorer, http://www.heaventools.com/overview.htm, Accessed: Jan 2020.