A Specific Domain Language for Network Interface Cards

Apostolos Manolitzas

Abstract— **Writing a device driver has never been pleasant. Maintaining or updating the driver is almost hard. We address this problem by suggesting the use of specific domain language (SDL) for the programming of the device. To support our idea, we present the procedure of defining an SDL language for network interface cards. Through the definition procedure will shine the advantages of these languages.**

I. Introduction

FEELING the heat of their competitors, hardware companies create new products with a frenetic pace. Those devices need the support of drivers, which should be developed, debugged and tested in the most limited time in order to follow this pace. Although device drivers play the most critical part in the terms of performance, time pressure doesn't allow extensive testing. In combination with the assembly language, the low-level programming and the bits operations increase the error possibility and decrease the productivity. Not to mention the error-prone nature of that kind of programming.

We support that the most efficient solution to those problems would be the complete abstraction from the lower programming levels and the use of a language, in which by providing the contents of the specification data sheet, a language generator should create the major part of the driver. This approach is not utopia, but requires intelligent, specialized compilers and rich libraries that would encapsulate more of the hardware knowledge.

Furthermore, the different OS platforms provide another Sisyphean labor in writing device drivers, because the developer has to create multiple different instances of the same driver to support those differentiations. From our point of view, we support that the solution will be given by using SDL(specific domain languages) with high level of abstraction, that could generate a specialized output adjusted even to the OS platform.

This paper describes an approach to developing a language for automatically generation of device drivers for network interface cards. Our approach allows device drivers to be written with high abstraction and strong typing rules so as to address most of the problems that the developer faces.

II. An SDL for network devices

We introduce a Specific Domain Language (SDL) dedicated to the specific domain of the device drivers. Particularly our concern focuses on the network interface cards. In this approach, we attempt to develop a programming framework that encourages the programmer to concentrate only on the device specification and ignore implementation details.

For the SDL language development, we follow the methodology as defined by Consel [9]. The first step of the process encourages the language analysis for defining the commonalities and the variations in the corresponding program family. In the second step an informal interface of the language is defined as well as the semantics and the notation of the language. Due to lack of time we restrict our research attempt only in the first two steps, however we mention the next steps briefly. In the next step the semantics of the language is split between the compile-time and run-time actions. A successful formal definition is then possible. An important step is the grouping of the dynamic semantics algebra to form a dedicated abstract machine that models the dynamic semantics of the SDL. The last step is consisted of the implementation of the required libraries as well as the application of the partial evaluation technique [10],[11], to automatically transform a SDL program into a compiled program, given only an interpreter. In practice, the whole process needs to be iterated more than once.

The rest of the paper is organized as follows. Section III provides an overview of how to write a device driver for Linux but focused on components interesting for identifying family members. Section IV describes in abstraction the interface of the language. Section V presents the language analysis along with some informal interface. Section VI describes related work. Section VII concludes and suggests future work.

III. Basic components of a device driver

The purpose of this section is to present the basic components of a device driver for network cards. Those components will consist the basic guide in generating device drivers. We will focus on a specific operating system, Linux 2.4.8. This section is not a tutorial on how to write device drivers. For further information someone can check Rubini's bible [8] on device drivers.

The network subsystem of the Linux kernel is designed to be completely independent. This applies to both networking protocols and hardware protocols. Interaction between a network driver and the kernel proper deals with the one network packet at a time; this allows protocol issues to be hidden neatly from the driver and the physical transmission to be hidden from the protocol. The network cards usually implement the physical and link layer.

A.Manolitzas is a graduate student in the Department of Electrical Engineering in Columbia University of New York.

The main association between the kernel and the network device is the *struct net_device*. So the first thing that a driver must do, is to establish this connection by initializing the *struct net_device*. In Figure 1 we can see the members of the struct as well as the initialization of that struct. The functions that are presented in the right part of the assignments, represent the functions that the device driver need to implement. The second critical structure is the *struct netdrv*. This structure contains private members adjusted to the device needs. So every useful variable for the drive functionality can be declared there. Such variable is the *ioaddr*, which is the address where the device communicates with the kernel.

```
ether_setup(dev); /* assign some of the fields */

dev->open               = mydev_open;
dev->stop               = mydev_release;
dev->set_config         = mydev_config;
dev->hard_start_xmit    = mydev_tx;
dev->do_ioctl           = mydev_ioctl;
dev->get_stats          = mydev_stats;
dev->rebuild_header     = mydev_rebuild_header;
dev->hard_header        = mydev_header;
#ifdef HAVE_TX_TIMEOUT
dev->tx_timeout         = mydev_tx_timeout;
dev->watchdog_timeo     = timeout;
#endif
dev->flags              |= IFF_NOARP;
SET_MODULE_OWNER(dev);
```

Fig. 1. The core of *mydev_init*.

Based on the figure we will examine every function in detail and we can conclude about the usability of each function. The initialization function plays catalytic role to the driver's functionality. It has the responsibility of assigning the related functions to the right member of the struct net_device and identifying the device. From the hardware part of the device, it is important to reset the device and its state. Depending on the driver implementation, some approaches choose to allocate all necessary resources in the init function. But common practice dictates the resource allocation should be done during the opening of the device.

Open and close operations don't differ from all the device drivers. In general open() requests any system resources it needs and tells the interface to come up; stop() shuts down the interface and releases system resources. The kernel will open and close the interface in response to the ifconfig command. However there are a couple of additional steps to be performed. Firstly, the hardware address need to be copied from the hardware device before the interface can communicate with the outside world. Secondly, the open() function should start the interface's transmit queue once it is ready to start sending data. So a typical operations sequence for the open function would be first requesting system resources, then firing the device and last starting the transmit queue. From the other hand the close() function has to do the reverse job, which is first to stop the queue and secondly to release any system resources allocated by open().

The most important tasks performed by network interfaces are data transmission and reception. The kernel uses a structure called socket buffer to store incoming or outgoing packets. The role of the transmission is to copy the content of the buffer to the device's buffer. After that the hardware is responsible for the data transmission. The psychical transmission mechanism has been isolated in another appropriate function so as to be independent from different vendors implementations. In some hardware architectures the transmission is interrupt driven so role of the function is restricted into copying the content of the socket buffer to the device buffer.

Receiving packet is not as easy as sending, because the allocation of the socket buffer and passing the packet to upper layers must be handled by the interrupt handler. Packet receiving is mostly an interrupt driven event. The buffer allocation for the new incoming packet isn't a simple malloc() operation but it strongly depends on the way that the device communicates with the memory. Many devices in order to increase the I/O throughput they use DMA access. In this case intelligent memory management is required and memory allocation a priori the packet receiving. Concurrency and locking issues may appear on those situations. The last step consists of passing the packet to the upper layer using netif_rx() operation.

Another issue that the driver must deal with is hardware fail. In case of a hardware error, the driver must be able to recover from that loss in order not to lose a new interrupt. The solution hears to the name of timers. Many drivers set timers, and if an operation is not completed in the given time, something is wrong and a function must handle this error. The correct error handling is updating the statistical information of the interface and restoring the device in a state that can continue the packet transmission or reception.

The interrupt handler always plays the most critical part in terms of the driver performance. The basic functionality of the handler function must be first to acknowledge the interrupts and then to discriminate three types of interrupts. One when a new packet has arrived, one when a packet was transmitted (successfully of not) and one when some error occurred. The discrimination is based on a status interrupt register that every device has. Locking issues arise and the programmer must ensure the atomicity of every action in the handler.

Furthermore, functions that change the device configuration such as ioctl, can be supported by the device driver. Any ioctl command that is not recognized by the network layer is passed to the device layer. Each interface can define its own ioctl commands. In addition to ioctl, the device should support functions that keep statistical information such as packet sent, lost, discarded and every other useful information about the device operation. The implementa-

tion of such methods is pretty easy.

IV. Details about the SDL

The following section describes in details, the results of commonality analysis. The analysis and the rest of the elaboration is based on the drivers that are mentioned in the Appendix. The commonality analysis forms the basis for designing reusable assets that can be used to produce rapidly family members. We divide the family members to operations and properties. Operations are called all possible actions that could happen in a network card such as read or write of a register and properties are called the set of registers, control modules and interfaces.

A. Operations

Following Thibault et al [3] example, we identified three patterns which appear in the drivers that could be used as a guide for defining different operation families.

A.1 Operation Pattern

The first pattern fits to a model of atomic operations that can always be identified in the device driver code. Those operations are repeated frequently and they are the primitive modules for the rest of the operations. The code differs by the data arguments that it uses. From our experience we can classify the I/O operations as the most useful and frequently used. The major interaction with the hardware consists of code fragments that read or write some register. But those operations vary from vendor to vendor causing incompatibility issues. For example several device cards have 16-bit operations and others have 32-bit operations. The programmer should be protected from having to make such discriminations. For the programmer there would be only one read or write command, appropriate to each width.

Furthermore, cases where the developer has to add an offset so as to access some register must be prevented. For example something like:

```
outb(0x09434, ioaddr+0x12345);
```

must be replaced with a different notation and some abstraction. Several programmers use macros to avoid such notations. We want to formalize those operations and hide the details from the programmer. An informal example could be:

```
register TxConfing := TxCRC; /* (write) */
variable status[1:5] := MediaStatus; /* (read) */
```

The language must provides facilities to handle cases where two or more continuous registers must be read or written, so as the serialization procedure to be transparent to the developer.

A.2 Combination of operations pattern

Actions that target critical operations of the device belong here and are consisted from the operation patterns that mentioned before. Those patterns don't belong to a certain family, which means that we couldn't find any commonalities between them. In this category we can place operations such as bit shifting and bit masking. Those operations are dedicated to every device and it's hard to parameterize. Moreover, attempts are made to use a more common interface over different vendors. The example of the MII (media independent interface) registers, which provide a common register interface to the Physical Coding Sublayer independent from the media reveals those attempts.

A.3 Control pattern

Code fragments that belong to control pattern category, represent actions that are consisted from operations patterns but the sequence and the data passed as arguments have highly dependence with the hardware architecture of the device. A very common operation that belong to this set, is the activation and deactivation of the hardware transmission and reception mechanisms. Different series of I/O operations must take place in every hardware implementation. The range of control patterns increase if you think the number of the different media types that exists and a device can support. So in the same device could exist different processes for supporting multiple medias. Not to forget the auto negotiation mechanism that encourages such policy.

Moreover, actions like hardware initialization, hardware reset are dedicated to the hardware family that the device belongs and are strongly connected with the hardware architecture. Those operations are defined in the specification of the data sheet of the device and must be declared by the developer.

B. Properties

As properties we consider the complete set of the characteristics that can be found in the device data sheet and are used to operate the device. Every network interface card has those properties and the variation on those could specify important details about defining a language.

B.1 Identification

The device driver should first probe for the identification of the chip used device. Every chip has different characteristics that must be specified by the programmer. An informal example could be:

```
begin identification
variable chip_id := Chip_ID;
case chip_id:
 0x0A=> RTL8139(0x10ec,0x8139);
 0x0c=> DELTA8139(0x1500,0x1300);
end case
end
```

Those assignment can fill the pci table for the device. Not to mention the fact that the driver has to declare the MODULE_DEVICE_TABLE, which is strongly related to the identification and the properties of the board.

B.2 Registers

Probably, the most important piece of the hardware puzzle in terms of communication with the outside world is the operational registers. According to the complexity and the functionality of the device, the registers can be split in families. Status, control and configuration registers are the most common. The developer has to define each register's address in the beginning of the driver so as to access without remembering their exact address. Some registers must have a common name, for example the interrupt register can be found to every device so keeping the same name is justified. So far, macro definitions and programmer's style resolved the register declaration, but in our proposal this becomes a de-facto feature.

B.3 Bit Fields

Most of the case the registers provide indication for more than one operation. For example the interrupt status register can have different values according to the interrupt source. So we want to extract those individuals' informations for every field. The method used so far is bit masking and shifting, which is error-prone. In a better case, declarations of macros or enums types help to avoid dealing with plain bits. We suggest a restriction mechanism that should allow the developer to declare at the beginning of the drivers those bit fields in more formal way. Declaration types that are use in hardware description languages such as Verilog would be more applicable to this idea.

B.4 Bit Values

Specific values must be used to fill all the options that the registers provide. Of course the plain bit use, creates the problems that we are trying to solve. An alternative, more sophisticated representation would be more beneficial for the developer. For example interrupt masks or configuration bit vectors require a name so as to represent an entity and not plain bit vectors. Not to mention flags, that should have at least a name to make them useful. Optimal solution would be to define those values in combination with the bit fields so as a type check can be applied during the driver code generation. For example, in the interrupt register should assign only values describing the interrupt status. Gathering the previous remarks and notations we suggest the following declaration form:

```
register Config[0:16] => 0x00a6;
register fields Config
begin
Config[0] =>PM_Enable {on, off};
Config[1] =>PIO {on, off};
Config[2:3] =>LWAKE {on(0x11), off(0x00)}
end
```

Following that declaration style a similar call for write operation could be:

```
register Config[LWAKE] := on;
```

where in the first declaration is determined the address of the register plus the ioaddr. In the next steps the fields of the register along with their domain are declared.

B.5 Synchronization Issues

The software architecture allows high concurrency features in order to increase I/O performance and support high throughputs. This environment has multiple threads accessing the same registers or variables. The issue that arises here is how much freedom must be left to the developer to secure it's masterpiece with locking mechanism. One solution would be to add some flag to the register declaration, which indicates that should apply special locking default by default without the programmer interfere. The other solution would be to allow the programmer to use the already adequate locking mechanism that Linux provides. Atomic reads and writes are part of the Linux library and it's programmer's responsibility to take advantage of them. We believe that this issue is an active challenge for our proposal.

V. Language Analysis

In the following section, we will present our language analysis by explaining in details, patterns discovered in the component functions and identifying operations essential for building language blocks. The result will be a detailed explanation of hardware and software steps made toward the driver implementation, and how those steps can be automatically generated after the guidance of the developer. To be more methodical, in every function we will split analysis to operations associated with the hardware and to operations associated with the software part and the operating system. The hardware operations could be a single read or write operation or a sequence of them. Furthermore, they belong to family set already identified in the paper.

To support our proposal we will present in every function the necessary operations that the developer *must* define. The language requires those definitions in order to generate the hardware operations of the device driver. The definition will have a format:

```
begin operation_name
register A := A_value;
variable b := B_register;
...;
end operation_name
```

Besides, the software part shouldn't require any involvement by the developer except when the user has to define major architecture characteristics such as the hardware bus where the device is plugged in. Utilizing that property we can hide any implementation from the user.

A. init function

The initialization function is split into the part where all the hardware initialization is done and into the software part where system resource allocation and memory mapping are the primarily targets. The hardware initialization

is a procedure strictly defined in the device data sheet. Particularly, a programmer must define the following hardware operations based on that data sheet: *soft_chip_reset*, *chip_probe*, *find_media_type*,*read_eeprom*. The eeprom reading is a more complicated operation with sometimes some delay on it. So in the language we could define an operation: *#delay_time*, which would be equivalent with the C statement *delay(delay_time)*.

Regarding the software operations we can divide them into two sets, operations standardized in every network driver, where you have to register the device (*register_netdev*) and device's functions to the kernel as in Figure 1 and the operations where you have to allocate system resources like memory and interrupt. The first type of operations won't involve the programmer. However for the second type, the configuration is based on the type of memory and data transfer that is preferred. The paradigm of PCI DMA bus for data transfer is the most common, nevertheless the language needs more flexibility.An assignment operation with a strict domain (PCI,ISA,USB,PCMCIA,PnPISA) can be used. The programmer shouldn't be aware of the implementation.

Except the typical functionality of the network cards, advanced features have been added to some cards to promote them. So the driver should get advantage of them. A good example of such feature is the auto negotiation that detects the various modes that exist in the device on the other end of the wire and advertises its own abilities to automatically configure the highest performance mode of inter-operation. We conclude from this example, that the language should be flexible to include advanced features of the devices. It shouldn't trap itself to a specific model.

B. open function

From the hardware side, the driver should start the network card. The procedure of starting the network card is described mostly by initializing the configuration registers and most important by starting the interrupt utility. We identify the following batch operations that must be predefined by the developer: *enable_tx, enable_rx, enable_interrupts, reset_counters, verify_enable_rxtx*.

From the software side, the driver obtains, if it is possible, DMA resources for data transmission. After Linux kernel 2.4.x, that allocation task has been simplified by the use *pci_alloc_consistent* function. Another critical task, important for the driver functionality, is the irq establishment and the registration of the interrupt handler. Any failure on those tasks causes the termination of the driver. In addition the timer must be initialized and registered to the kernel.

C. transmit function

Depending on the network card implementation the transmit functions plays a different role. In the simplest case the functions copies the contents of the socket buffer, into the hardware output buffer and then fires packet transmission. So from the hardware part the definition of an operation: *hw_tx* would be enough. But in more sophisticated hardware implementations the role of transmission function is to copy the content of the socket buffer to the Tx buffer, after that the card is instructed to move the data from the buffer to the internal transmit FIFO in PCI master mode and when the FIFO is filled to the programmed threshold level the card begins transmission. So no hardware operations are needed.

For the software part the driver copies the socket buffer to the Tx buffer. Also increases possible variables used to indicated the size of the FIFO and possible data structures for statistical purposes.

D. receive function

The reception of a packet is strongly connected with the interrupt handler. It's the service routine for the receive interrupt. The operation of the receive function involves reading of the Rx FIFO, allocating memory for the packets and push them to the upper layers. The only hardware operation needed here is a check if the Rx FIFO has packets in it: *Rx_Buf_Empty*

Hardware implementations sometimes, support error reports through the interrupt status register,so the routine should be able to identify the error that corresponds to its side and deal with it.

E. interrupt handler

The interrupt handler has to know the status of the interrupt and the domain of interrupt types. Some useful operation would be: *clr_int_status*, which would clear the status of the interrupt.

From software side the execution is straightforward. Initially the overwrite of the status register acknowledges all the interrupts. In the core part of the code, the status of the interrupt is checked and according to its value the proper service function is invoked.

F. stop function

Follows the reverse direction from open function. The first action that must be made is shutting down the transmission and the reception of packets, also no interrupts are accepted any more. Those are hardware operations that can be summed up by the following: *disable_rx, disable_tx, disable_int, low_power_mode*

On the software side, the driver should release any data paths obtained during opening the driver. Furthermore it must return the interrupt number to the operating system, unregister the timer and first of all close the packet queue to the upper layers.

G. miscellaneous function

To this family belong the rest of the functions that have a supplementary role. The function responsible for the statistics accumulation usually is implemented without any hardware interference. Of course special cases where the device is keeping its own statics can be consider as another set of operations.

VI. Related Work

We based our approach on the first successful attempt on writing a domain specific language for video device drivers by Thibeault et al [3]. They produced a language that could generate the code for the driver by describing only the characteristics of the device in terms of registers, ports, clocks and most basic actions' sequence. To support their idea, they implemented a driver for graphics devices of the S3 chips series using their language. The generated driver was compared with the current hand-crafted implementation of the S3 drivers. In the results of the comparison they pointed out that the performance of their driver surpasses in performance the original, implemented in C, driver in some cases. In addition they claimed that their code is very easy to write, to maintain and to reuse it. Though, their idea requires much effort to support a broader domain of applications and it is still very restricted.

The same research group identified that problem and Consel et al [1] proposed a different approach for solving the same problem. They introduced Devil, a more generic language based on the Interface Definition Language (IDL) for hardware functionalities. They used IDL to describe the hardware and its functionality. It provides the programmer with abstractions and syntactic constructs that are specific for describing devices. Particularly, Devil is a compiler that automatically generates stubs, which provide an interface to the device. The interface is mostly consisted of macros and nicely defined registers. Their approach emphasized on protecting the developer from logical and mistyping errors. They didn't touch issues such as multi OS portability.

To solve the portability issue several vendors have made an attempt to define a common driver interface called UDI (Uniform Driver Interface) [2]. This interface surrounds conforming device drivers modules, providing them with consistent interface to and from the host operating system and among cooperating drivers. However UDI focuses mainly, only on the high level part of the drivers and their interaction with the operating system.

Several others attempts have been made to make the life of the device driver programmer easier. Libraries [6] have been built providing tools for partial code generation dedicated to a single environment.Also the WinDK [5] Toolkit helps programmers write device drivers guided by some initial menus in a Visual C++ environment. Those products solve partial some problems nevertheless they don't deal with the whole range of the problem. A prominence attempt is a commercial product called jungo [7]. By plugging the device hardware in a computer slot, their diagnostics scan it and they create a framework based on the characteristic probed. Following, the developer provides the specification of the device through an interactive dialog and finally the program generates the driver for the device. Although this environment hides most of implementation details from the developer, it restrains him from expanding the driver's functionality with extra device features or handling any peculiar hardware bug.

Concluding, every approach has a different motivation and a particular range of applications. Some solutions were planed to provide an absolute solution to the complete set of device drivers and others are attempting to provide tools only for a small subset of drivers. Thus, there is no standard tool for writing device drivers, it means that the absolute solution haven't been found yet.

VII. Conclusion and Future Work

The problem of writing device drivers that can be easily reproduced, maintained, reused and debugged exists. Some weak steps have been made to address the problem either by providing more tools to the developers such as libraries and interfaces or by defining new languages specific for device drivers. Following those steps, we propose the definition of an SDL for network interface cards. The paper describes only the language analysis but the implementation and the evaluation of the language will prove how successful is.

The primary future step is the implementation of the language compiler. As we have already mentioned defining an SDL language that could describe the total number of cases is not easy due to the broad range of network devices and particularities in those devices. However an initial prototype would help for further expansion of the idea.

Furthermore issues such as locking mechanisms, debugging tools for the programmer and hardware bugs are challenges that need solution.

We currently believe that SDLs would change the way the drivers are written.

References

[1] Fabrice Merillon,Laurent Reveillere,Charles Consel,Renaud Marlet, Gilles Muller. Devil: An IDL for Hardware Programming. *OSDI 2000*, pages 17-30, San Diego, October 2000.

[2] Project UDI. *UDI Specifications, Version 1.0*, September, 1999. URL: www.project-udi.org.

[3] S. Thibault, R. Marlet, and C. Consel. Domain-specific languages: from design to implementation - application to video device drivers generation. *IEEE Transactions on Software Engineering*, 25(3):363-377, May-June 1999.

[4] E. Eide,K. Frei,B. Ford,J. Lepreau,G. Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation* (PLDI), Las Vegas,NV, June 15-18,1997.

[5] BlueWater Systems, Inc. *WinDK Users Manual.* URL: www.bsquare.com

[6] Compuware NuMega. *DriverWorks User's Guide.* URL: www.compuware.com/products/numega/drivercentral

[7] Jungo Ltd. *WinDriver V5 User s Guide.* URL: www.jungo.com

[8] A. Rubini, J. Corbet. *Linux Device Drivers, 2nd Edition.* O'Reilly, second edition, June 2001.

[9] Charles Consel and Renaud Marlet. Architecturing Software Using A Methodology for Language Development. *Principles of Declarative Programming. 10th International Symposium, PLIP'98.* pp. 170-194, Pisa, Italy, Sep 16–18, 1998

[10] N.D. Jones An introduction to partial evaluation. *ACM Computing Surveys*, 28(3):480-503, Sep 1996

[11] N.D. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice-Hall, EngleWood Cliffs, NJ, June 1993.

VIII. APPENDIX

The following files, under the directory *linux/drivers/net/** , have been studied to support the language analysis:

1. *ne2k-pci.c*
2. *pci-skeleton.c, isa-skeleton.c*
3. *tulip directory*
4. *3c515.c*
5. *eepro.c*
6. *3c59x.c*