A Domain-Specific Language for Device Drivers

Christopher Conway

16 December 2002

Abstract

Device drivers are difficult to write and error-prone. They are usually written in low-level languages with minimal type safety and little support for device driver semantics. As a result, they have become a major source of instability in operating system code.

This paper presents NDL, a type-safe, platform-independent Network Device Language. NDL provides high-level abstractions of device resources and constructs tailored for the expression of common device driver operations. We show that NDL allows for the coding of a robust and efficient driver with a code size reduction of approximately 45%.

1 Introduction

Device drivers have been noted as a major source of faults in operating system code [2]. Largely for efficiency, device drivers and other systems code have historically been written in low-level languages like C. Unfortunately, these languages do not provide the type safety and robustness one would expect in critical systems code. Work has been done to augment the type safety of low-level languages [3, 6], but the efficacy of this work is limited by both fundamental and practical concerns.

In this paper, I will describe NDL, a domain-specific language for network interface device drivers. The language has been used to implement a driver for NE2000 network cards, a widely available class of inexpensive LAN adaptors. The language includes direct support for the operational semantics of device drivers and provides a high level of type safety. Concurrency semantics are included for the description of devices with multiple independent operational units. Though the language has been built for and tested on network drivers, it is flexible enough to describe a wider class of drivers. The compiler is also designed to be readily ported to a wider class of operating systems.

2 Related Work

A variety of approaches have been suggested to improve the reliability of low-level software and device driver software in particular. Crary and Morrissett proposed typed assembly language (TAL) as a compiler target for preserving type information from higher-level languages [3]. Unfortunately, C, the most common systems programming language, is not much more strongly typed than a traditional assembly language and there is little the compiler can do to improve the type safety of C code.

Deline and Fähndrich use a similar typing system in the C-like programming language VAULT [4]. The use of variables is controlled through type guards which describe when an operation on a variable is valid. In order for the compiler to accept the program, it must respect the type guards' access specifications and types must match at program join points. VAULT has shown some success in preventing common programmer errors, but its limitations on alias types prevent it from being generally applicable to device driver development.

A more pragmatic approach is static analysis of traditional C systems code. Ball and Rajamani developed SLAM, a system that is currently in use in the Microsoft Windows group [1]. SLAM operates on a specification for correct behavior developed separately from the driver code. The result is very good error detection at compile time for the properties captured by the specification. However, the analysis can be slow and may take many iterations to complete. In addition, the types of errors that may be detected are restricted in principle, and limited as well by the correctness of the behavior specification.

A group at the University of Rennes has done work on domain-specific languages for device drivers. Thibault, et al., developed GAL, a domain-specific language for X Windows video drivers [7]. The project combines a partial evaluation framework with a language tailored to video driver operations to produce driver code that is nearly 90% smaller than the equivalent C code and just as fast. This work is

promising, but the methodology may not be applicable to device drivers as a whole.

Mérillon, et al., also of the University of Rennes, designed a more general solution for device driver development: the Devil interface definition language [5]. A Devil specification describes entities exposed for interaction with a hardware device (e.g., I/O ports, memory-mapped registers). The specification is compiled into a C module for manipulating the device, allowing the driver programmer to write to a clean API and avoid writing low-level code. This approach prevents certain common low-level programming errors, but it does not fully specify the protocol for using the device, and it does not provide the type safety of a higher-level solution.

3 NDL

The Network Device Language (NDL) is a domain-specific language for network device drivers. Its interface definition syntax is based on the Devil IDL [5], but it provides significant additional facilities for device manipulation and behavioral specification. The hardware interface is described using *ports*, *registers* and *device variables*. The behavior of the device is defined using *member variables*, *device operations*, *device functions*, *platform functions* and a *protocol specification*. Complex driver semantics are supported with language constructs for *device inheritance*, *synchronization* and *interrupt handling*. We will demonstrate the features of NDL using the example of a driver for the National Semiconductor NE2000 Ethernet controller and the 8390 chipset on which it is based.

3.1 Ports

Ports abstract the details of device access (e.g., the base address of the device, whether it is accessed using I/O or memory operations) and provide a window for communicating with a device. Consider the port declarations for the 8390 chipset:

```
port registers = 0..0x0f ;
port dataport = 0x10..0x17, littleendian ;
port reset = 0x18..0x1f ;
```

Port ranges indicate offsets from the base address of the device. `dataport` is declared `littleendian`, meaning that any multi-byte read from or write to that port should use little-endian byte ordering. The compiler can use this information to automatically translate from or to the CPU byte ordering scheme as needed.

3.2 Registers

Registers define the basic hardware-level granularity of device access. They are defined by their offset within a port window and may have separate ports for reading and writing. A bitmask may be associated with a register to specify bit constraints. A '`.`' in a bitmask denotes a bit that is relevant and '`*`' denotes a bit that can be ignored. '`0`' or '`1`' denote a bit that can be ignored when read but has a fixed value when written. The command register on the 8390 is defined as:

```
register CommandReg = registers(0),
        read mask '......**' : bit[8];
```

`CommandReg` is an 8-bit register at the beginning of the `registers` port. Only bits 2 through 7 are considered relevant when the register is read.

3.3 Device Variables

Device registers often contain several semantically distinct values. To provide independent access to these values, a device specification may contain device variables. Device variable values may be taken from individual bits of a single register, an entire register, or multiple registers concatenated together. Device variables are strongly typed. Consider a variable defined on the command register of the 8390:

```
variable registerPage =
        CommandReg[7..6] : int{0..2};
```

`registerPage` takes its value from the high-order two bits of `CommandReg`. It is of type `int` with a range of values from 0 to 2.

3.4 Member Variables

Member variables are used for storing driver state. They are not associated with a register or port; they are statically allocated or created on the stack at runtime, just as variables in C or C++. They may be integers, booleans, structures or arrays.

3.5 Device Operations

NDL's data types and operators simplify the expression of common device driver operations. Device variables on a single register can be grouped into structures to efficiently write their values without the use of bit operations. Additionally, device variables that are spread over multiple registers can be read and written with just one assignment statement.

Take, for example, a typical sequence of register writes in C:

```
outb(E8390_NODMA+E8390_PAGE0+E8390_START,
        nic_base+ NE_CMD);
outb(count & 0xff, nic_base + EN0_RCNTLO);
outb(count >> 8, nic_base + EN0_RCNTHI);
```

Bit flags are combined and the integer `count` is shifted and masked to force its value into a disjoint register format. The write operations are a series of low-level I/O calls, using predefined indexes on the device's base address. The equivalent NDL code would use simple assignment and structure syntax:

```
command = { nicState=START,
            remoteDmaState=DISABLED } ;
remoteDmaByteCount = count ;
```

Another frequent device operation is copying data to or from buffers and ports. There are common idioms for a buffer-to-buffer copy in C and C++, but they are complicated by the need to support compatible devices with different data bus widths in the same driver. NDL provides an extraction operator that handles these differences automatically:

```
var buffer: byte[count] ;
buffer << dataport ;
```

In this example, the buffer will be completely filled from the dataport using the appropriate I/O operations.

3.6 Device Functions

Device functions expose an interface for interacting with the device. They combine low-level device operations to provide access to high-level actions like device initialization or data transmission. Device functions may take parameters and return a value of any valid type.

3.7 Platform functions

Certain functions necessary for the proper operation of the device may be platform-dependent. For example, Ethernet cards usually have a packet buffer which is filled from an operating system queue of outgoing packets. When the packet buffer is full, the card must issue a signal to stop pulling packets off the queue until there is room in the buffer. The nature of this signal will vary between operating systems.

A platform-specific function is declared using the keyword `platform`. Platform functions will be defined or mapped to existing system functions in a platform-specific library included with the compiler.

```
protocol {
  NetworkDevice :
      init ( start DevFunc* stop )*
      ;
  DevFunc :
      set_multicast_list
      | start_transmit
      | timeout
      | hard_header
      | rebuild_header
      | set_mac_address
      | interrupt
      ;
}
```

Figure 1: Protocol specification for an Ethernet adaptor.

3.8 Protocol Specification

In order to avoid entering an unknown state, every driver has a protocol for accessing device resources. Dependencies between device functions can be defined using a protocol declaration. Protocol declarations begin with the `protocol` keyword, followed by a series of yacc-like extended Backus-Naur form grammar productions. Only functions included in the protocol declaration are visible outside of the driver.

Figure 1 shows the protocol declaration for `NetworkDevice`, the description of a generic Ethernet adaptor interface. The function `init` must be called first, to initialize the device. Then `start` may be called to put the device in a running state, followed by any of the functions denoted by `DevFunc`. The `DevFunc` functions represent typical Ethernet interface operations like `start_transmit` and `set_mac_address`. (The function `interrupt` is special and will be discussed in Section 3.12.) The `DevFunc` functions may called in any order and repeated any number of times, as indicated by the use of the Kleene closure operator '`*`'. The protocol concludes with a call to `stop`. The sequence "`start DevFunc* stop`" may be repeated any number of times, indicating that the device may be restarted after it has been stopped.

3.9 Device Inheritance

A device may belong to a class that presents a generic interface to the operating system, or it may belong to a sub-class of devices that share a usage protocol (e.g., devices based on a common chipset). NDL pro-

```
abstract device NetworkDevice {
    abstract function init() ;
    ...
}

abstract device NS8390
extends NetworkDevice {
    abstract function hard_reset() ;
    function init() {
        ...
        hard_reset() ;
        ...
    }
    function timeout() {
    ...
}

device NE2000 extends NS8390 {
    function hard_reset() { ... }
    ...
}
```

Figure 2: A device that inherits from abstract devices.

vides an interface abstraction and a limited form of inheritance that allow device drivers to be built in a modular way, minimizing duplicate code in related devices.

An NDL specification may represent an abstract device encapsulating the behavior of a broad class of devices. A device is considered abstract when one or more of its functions is declared using the `abstract` keyword. An abstract function has no body and represents a device-specific function that will be defined in a specification that extends the abstract device.

A device may inherit the behavior, interface and use protocol of another device using the keyword `extends`. The new device has access to all of the original interface's variables and functions, and exposes the same interface to the operating system. If the original device was abstract, then the new device must define all of its abstract functions, or itself become abstract. The new device may redefine any of the inherited functions and alter the protocol specification as needed.

Figure 2 shows several device specifications with abstraction and inheritance. `NetworkDevice` declares several abstract functions, including `init`. `NS8390` extends `NetworkDevice`, defines the function `init` and declares abstract functions like `hard_reset` to

```
identification {
  REALTEK { name="RealTek RTL-8029",
            id=0x802910ec },
  HOLTEK32 { name="Holtek HT80232",
             id=0x005812c3,
             ioBits=16 },
  HOLTEK29 { name="Holtek HT80229",
             id=0x559812c3,
             ioBits=32 }
  ...
}
```

Figure 3: An identification block.

modularize board-specific behavior. Finally, `NE2000` extends `NS8390` and defines all of its abstract functions.

3.10 Device Identification

A device driver must include a method for identifying the hardware devices it can control to the underlying system. The PCI bus identifies hardware using a 32-bit unique ID. The IDs of the devices supported by the driver can be enumerated using an identification block. In addition, the block may be used to set model-specific variables to handle minor inconsistencies between compatible devices. A common example is the display name of the device, which should include the precise vendor and model number rather than the generic device class.

Figure 3 shows a portion of the identification block for the NE2000. Each supported board has a handle, a display name and a unique ID number. The handles become boolean constants that can be tested in device functions sensitive to incompatibilities. `ioBits` is an implicit member variable that is used by the compiler to generate correct data operations.

3.11 Synchronization

Device drivers often contain precisely defined actions on device memory. The code defining these actions is vulnerable to race conditions. Unpredictable errors can be avoided by protecting critical sections of code. A critical section is defined with the keyword `critical` followed by a block of code to be protected. Critical blocks prevent another process from entering any other critical section while the code within the block is being executed—only one critical section may be executing in the driver at one time.

```
critical function receive()
@( interruptStatus.packetRxIrq
   || interruptStatus.rxErrorIrq ) {
    ...
    interruptStatus = {
        packetRxIrq=ACKNOWLEDGED,
        rxErrorIrq=ACKNOWLEDGED
    } ;
}
```

Figure 4: An interrupt handling routine, declared critical.

By itself, `critical` defines a simple lock; it does not prevent the driver from being interrupted by another process. Under certain circumstances, this may lead to deadlock. To prevent an interrupt from occuring on the device's IRQ line, the declaration is followed by an optional parameter: `irq`. To disable all interrupts, the parameter takes the macro `ALL_IRQ`. An entire function may be treated as a critical section by placing the declaration `critical` before the function definition, as in Figure 4.

3.12 Interrupt Handlers

A function is marked as an interrupt handling routine using the notation "`@(`*`boolean-expr`*`)`". Interrupt handling routines are invoked by a compiler-generated top-level interrupt handler associated with the device IRQ. (This function may be referred to as `interrupt` in the protocol specification, as in Figure 1, but it may not be explicitly invoked elsewhere in the driver.) When an interrupt occurs, each routine's boolean expression is evaluated and, if true, the associated function is executed.

Figure 4 shows an interrupt handling routine. If either the `packetRxIrq` or `rxErrorIrq` fields of the `interruptStatus` structure is true at the time of an interrupt, `receive` will be invoked and the condition will be handled. The function ends by acknowledging the interrupts, clearing the execution condition for future evaluation.

4 Conclusions

NDL demonstrates the benefits of a domain-specific language approach to device driver development and reliability. NDL specifications are more robust, perspicuous and concise than their C language equivalents. Figure 5 shows the number of lines of code for the full NE2000 device driver (including the separate 8390 chipset code) written in C and NDL. In total, the NDL code is approximately 45% shorter, a significant increase in expressiveness.

	C	NDL
8390	1000	684
NE2000	507	142
Total	1507	826

Figure 5: A comparison of lines of code in C vs. NDL.

The next step is to demonstrate that NDL is a practical, flexible and efficient choice for real-world driver code. We intend to develop a Linux compiler for the language and test it in comparison with existing C drivers. The compiler will incorporate static verification techniques and the language may need to be altered to accomodate more sophisticated model checking. Work also needs to be done to improve NDL's application to a more general class of drivers and ensure it is able to specify the driver interface on a wide variety of platforms.

References

[1] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, January 2002. ACM SIGPLAN-SIGACT.

[2] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Banff, Alberta, Canada, October 2001. ACM.

[3] Karl Crary and Greg Morrisett. Type structure for low-level programming languages. In *International Colloquium on Automata, Languages, and Programming 1999*, volume 1644 of *Lecture Notes in Computer Science*, pages 40–54, Prague, Czech Republic, July 1999. Springer Verlag.

[4] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 59–69, Snowbird, Utah, June 2001. ACM SIGPLAN.

[5] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil:

An IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating System Design and Implementation*, pages 17–30, San Diego, California, October 2000. USENIX.

[6] Greg Morrisett. Type checking systems code. In *European Symposium on Programming*, volume 2305 of *Lecture Notes on Computer Science*, pages 1–5, Grenoble, France, April 2002. Springer Verlag.

[7] Scott Thibault, Renaud Marlet, and Charles 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.