Synthesizing Device Drivers

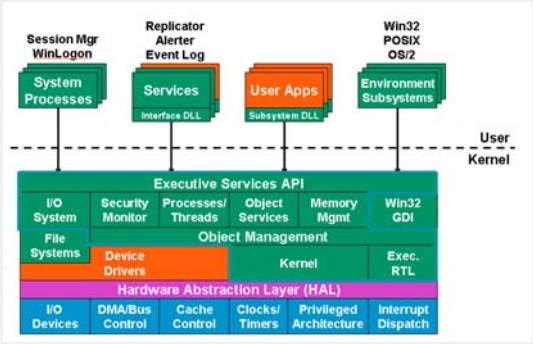
Roumen Kaiabachev and Walid Taha
Department of Computer Science, Rice University




What Makes Device Driver Development Hard

Device drivers:

- Are written in low-level languages with minimal support for modularity and code reuse
- Execute in the operating system kernel. A driver code crash takes the whole system down



The Windows 2000 Architecture

How to Improve the Development Process

We want to use high level declarative specifications to generate device drivers which model operating system processes and correctly communicate with the hardware.

We are designing and implementing a domain-specific language (DSL) **RIDL** which will provide the formal theory and software engineering medium for development of device drivers.

Case Studies (Jointly With National Instruments and Schlumberger)

Existing software high-level tools claim to help:

- Cyclone – statically typed safe dialect of C
- Devil – DSL, high-level definition of bit-mapped I/O
- NDL – DSL, state-machine-based model of device resources and common device driver operations
- BLAST – software model checking

We wrote a tiny device driver in each of the tools to study how much and to what extent. While there were several useful techniques to consider, neither tool offers a systematic way of writing device drivers.

Problems of More Complex Driver For Continuous Data Acquisition

{1} Resource tracking. It is illegal to start an acquisition at the same time another one is already running. Similarly, it is invalid to read data or stop an acquisition if none is running.

{2} Buffering. Data acquired continuously is transferred in the background from a small hardware buffer on the device to a larger buffer in computer memory. There are several ways to buffer data until an application can read it.

{3} Wait strategy. There a number of ways to wait when requested data is not yet available.

{4} Page locking. Memory accessed from an interrupt-service routine (ISR) or as part of a direct memory access transfer must be "page-locked" to prevent pages from being swapped out.

{5} Atomic device access. The device can be concurrently accessed from both the ISR and from a user process calling device code. The driver has to guarantee the ISR atomic access until it is done unless it is interrupted by a higher priority interrupt.

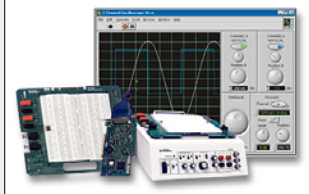
{6} Buffer overflow. In the case of a buffer overflow, the driver has to indicate missed data to the user application.

{7} Bad parameters. The driver should detect bad parameters passed to the device.

{8} Handling device reset. The driver should support a reset function that stops any ongoing acquisition and returns the device to an initial state.

How RIDL Will Solve These Problems

- We will use a higher-level representation of the physical device state to tackle {1}, {2}, {3}, {4}, {6}, {8}
- We will use types for safe locking and will explicitly model an interrupt scheduler (constraints will be applied from defined scheduling requirements). {5}
- We will use Devil-like interface checking and a BLAST-like approach to guarantee that certain user-specified states cannot be reached. {7}



Acknowledgements

This work is supported by the National Science Foundation, a Texas ATP grant National Instruments, and Schlumberger

Handling Other Driver Problems We will expand RIDL to handle other interesting device driver paradigms and make it a useful tool for driver writers.