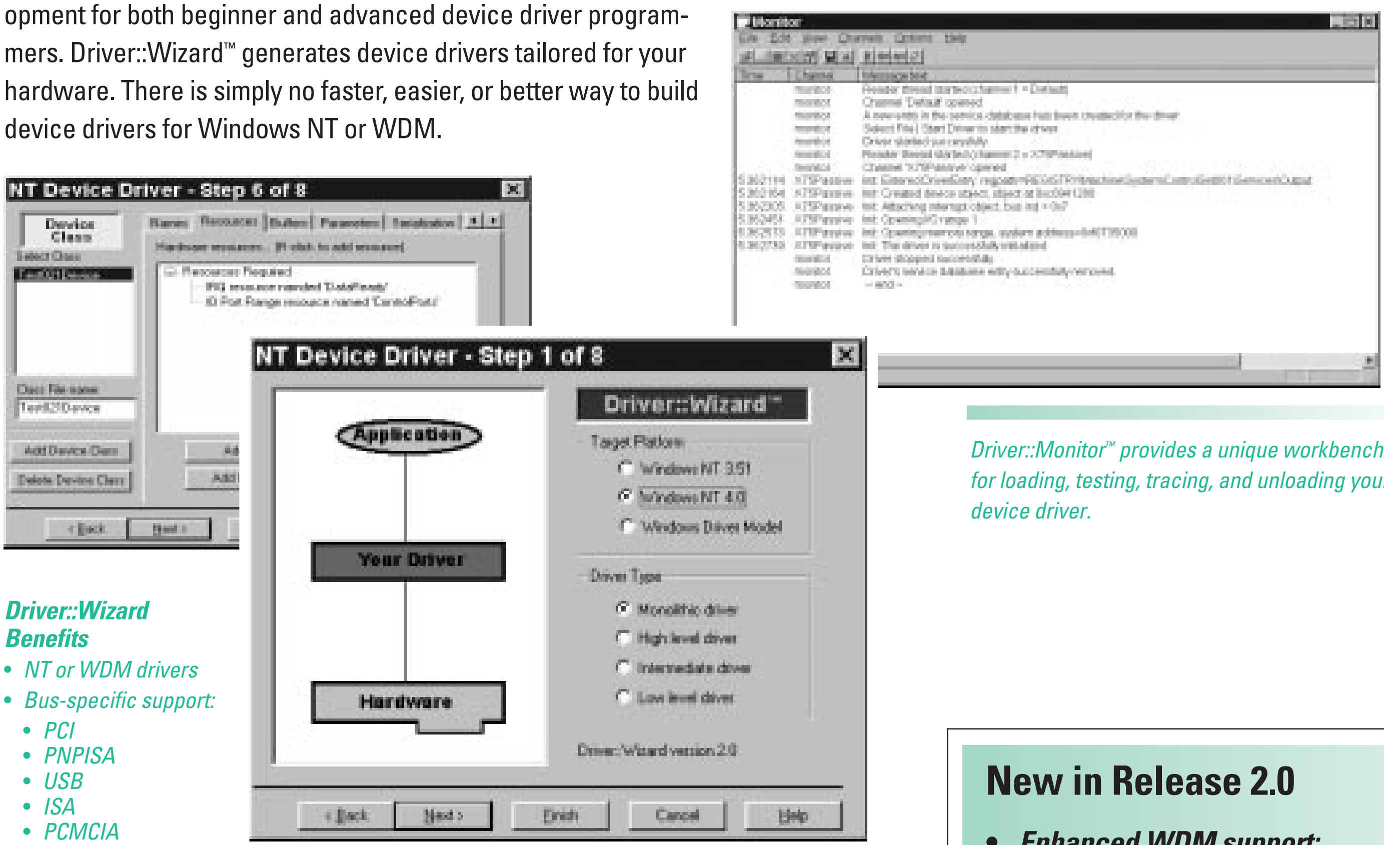
DRIVER::WORKS™ *Device Driver Development for Windows NT and Windows 98 (WDM)* **Version 2.0**

Driver::Works hides the complexity of Windows NT and WDM device driver development with a simple, easy-to-use Wizard, class library, documentation, sample code, and Vireo's renowned technical support.

Driver::Works exposes the object oriented design of Windows NT with a well-organized class library that closely models the operating system. Driver::Works hides the messy details of NT/WDM driver design, greatly simplifying driver design and development for both beginner and advanced device driver programmers. Driver::Wizard™ generates device drivers tailored for your hardware. There is simply no faster, easier, or better way to build device drivers for Windows NT or WDM.



Target Platform
○ Windows NT 3.51
● Windows NT 4.0
○ Windows Driver Model

Driver Type
● Monolithic driver
○ High level driver
○ Intermediate driver
○ Low level driver

Driver::Wizard version 2.0

Driver::Monitor™ provides a unique workbench for loading, testing, tracing, and unloading your device driver.

Driver::Wizard™ generates a device driver tailored to your needs. With support for many bus types including PCI, USB, ISA, PCMCIA, Driver::Wizard can generate more than a thousand lines of code to jump-start your development.

Driver::Wizard Benefits

- *NT or WDM drivers*
- *Bus-specific support:*
  - *PCI*
  - *PNPISA*
  - *USB*
  - *ISA*
  - *PCMCIA*
  - *and more*
- *Hardware resources support*
  - *DMA support*
  - *Mapped memory support*
  - *Port I/O support*
  - *Writes IRQ and DPC handlers*
- *Creates IRP handlers*
- *Optional trace code for debugging*
- *Integrated with MSVC*
- *Define IOCTL codes*
- *Select IRP queueing method*

According to Visual Developer in the Aug/Sep 1997 issue:

> ***"If you develop drivers for Windows NT and/or WDM, then this [Driver::Works] is the best way to develop them, period."***

New in Release 2.0

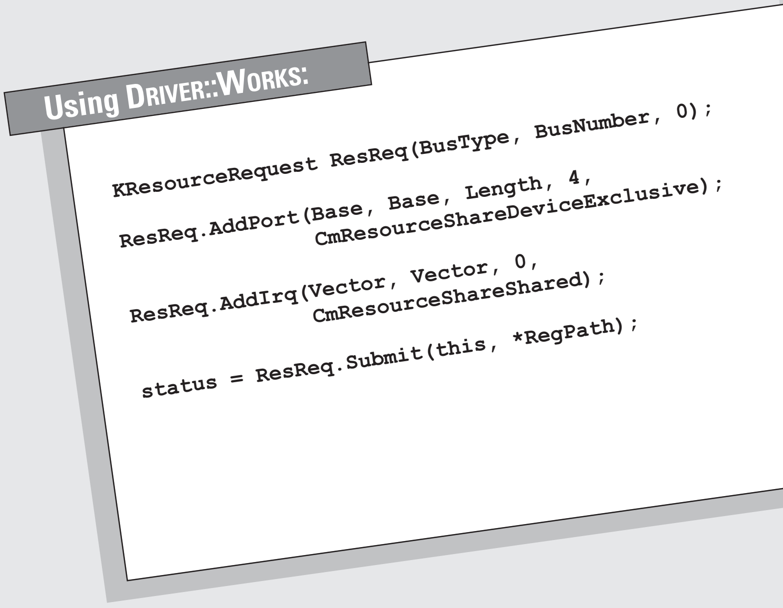
- ***Enhanced WDM support:***
  - ***Stream Minidriver classes***
  - ***Video capture stream sample***
  - ***WDM resource configuration***
  - ***Improved USB support***
- ***New Wizard supports:***
  - ***WDM***
  - ***Additional bus types***
  - ***Native MSVC projects***
- ***New Driver::Monitor supports Windows 98, timestamps, and more***
- ***New utilities***
- ***New sample drivers***

Here are just some of the sample drivers included in Driver::Works.

BASICPCI	A skeletal driver for a PCI device. Illustrates assignment of resources and access to the PCI configuration space. See MASTRDMA for a full PCI example.
BUSADDR	A demonstration of usage of class KPeripheralAddress and related classes. Shows how to map a physical region of a memory mapped device to process space, and how to do port I/O. This example accesses hardware specific to Intel platforms.
BASICUSB	Shows how to access the USB configuration descriptor, how to set up a configuration, and how to create interface and pipe objects to model a USB device.
COMMFILT	A filter driver for serial devices. Includes a GUI application.
CONTROLR	A demonstration of class KController. Illustrate serialization of IRPs for devices sharing common physical resources represented by a controller object.
DBGMSG	A driver written to support message passing from kernel mode drivers to utility program Driver::Monitor.
FILE	A demonstration of file I/O from a kernel mode driver.
HIDMOUSE	A WDM HID minidriver that enables control keys to simulate mouse movements.
INTRDEMO	A demonstration of handling hardware interrupts.
KBDCLASS	A keyboard class driver, based on the DDK sample.
KBFILTER	A filter driver for the keyboard.
MAPMEM	A demonstration of memory mapped devices.
MASTRDMA	A driver for a Datel PCI416L analog to digital board, demonstrating bus master DMA.
PARPORT	A contention handler (class driver) for the parallel port, based on the DDK sample.
PCIENUM	A driver to enumerate all the PCI devices in the system.
PCIWDM	Provides a framework for a monolithic driver of a PCI device in the WDM environment.
PORTIO	A driver to support a simple utility program that enables you to read and write ports interactively.
RAMDISK	Implements a RAM disk, under the control of a GUI application.
SERIAL	A driver for 8250 style UARTs, using an extensible SerialDevice class. This sample can be used to replace the standard system-provided serial driver.
SLAVEDMA	A driver for the National Instruments ATM-IO-16X analog to digital board, illustrating how to perform slave (system) DMA.
USBFILT	A driver that filters URBs directed to and from a USB device.
VIDCAP	A stream minidriver that implements video capture.

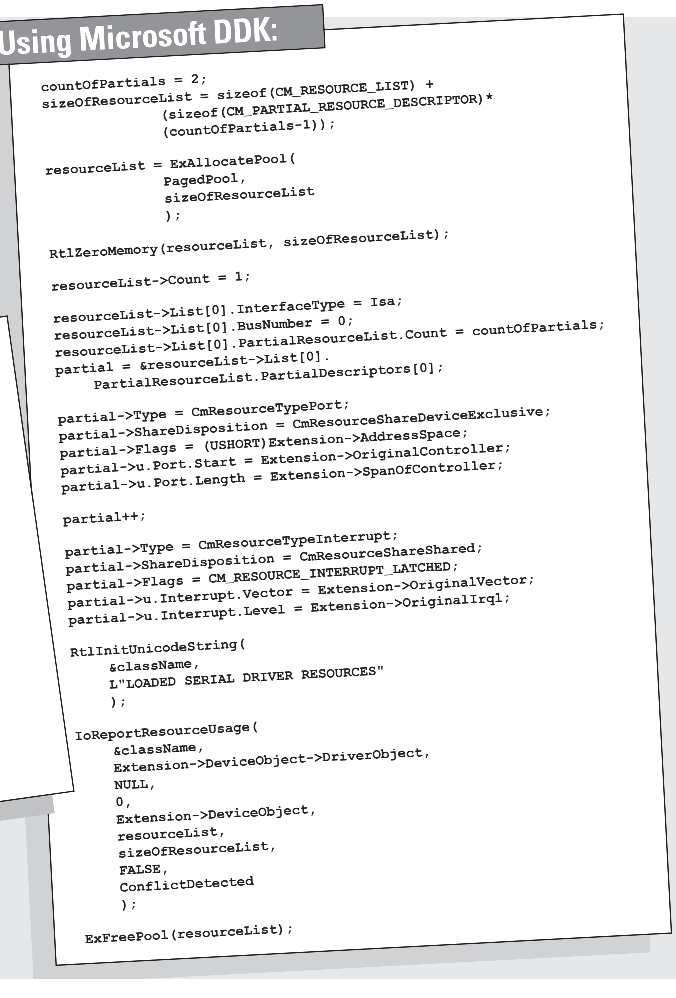
How much time and work does Driver::Works save?

The first code sample below shows how to use Driver::Works to report the use of one interrupt line and one I/O port to Windows. The second sample shows how to perform the same task using the DDK.



Using Driver::Works:

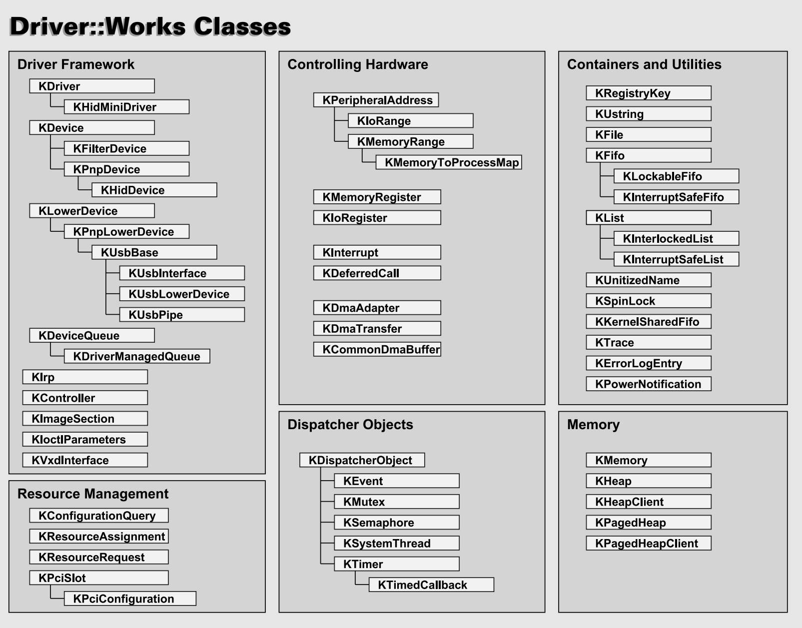
```
KResourceRequest ResReq(BusType, BusNumber, 0);

ResReq.AddPort(Base, Base, Length, 4,
               CmResourceShareDeviceExclusive);

ResReq.AddIrq(Vector, Vector, 0,
              CmResourceShareShared);

status = ResReq.Submit(this, *RegPath);
```



Using Microsoft DDK:

```
countOfPartials = 2;
sizeOfResourceList = sizeof(CM_RESOURCE_LIST) +
               (sizeof(CM_PARTIAL_RESOURCE_DESCRIPTOR)*
               (countOfPartials-1));

resourceList = ExAllocatePool(
               PagedPool,
               sizeOfResourceList
               );

RtlZeroMemory(resourceList, sizeOfResourceList);

resourceList->Count = 1;

resourceList->List[0].InterfaceType = Isa;
resourceList->List[0].BusNumber = 0;
resourceList->List[0].PartialResourceList.Count = countOfPartials;
partial = &resourceList->List[0].
     PartialResourceList.PartialDescriptors[0];

partial->Type = CmResourceTypePort;
partial->ShareDisposition = CmResourceShareDeviceExclusive;
partial->Flags = (USHORT)Extension->AddressSpace;
partial->u.Port.Start = Extension->OriginalController;
partial->u.Port.Length = Extension->SpanOfController;

partial++;

partial->Type = CmResourceTypeInterrupt;
partial->ShareDisposition = CmResourceShareShared;
partial->Flags = CM_RESOURCE_INTERRUPT_LATCHED;
partial->u.Interrupt.Vector = Extension->OriginalVector;
partial->u.Interrupt.Level = Extension->OriginalIrql;

RtlInitUnicodeString(
     &className,
     L"LOADED SERIAL DRIVER RESOURCES"
     );

IoReportResourceUsage(
     &className,
     Extension->DeviceObject->DriverObject,
     NULL,
     0,
     Extension->DeviceObject,
     resourceList,
     sizeOfResourceList,
     FALSE,
     ConflictDetected
     );

ExFreePool(resourceList);
```

Driver::Works Classes



The Driver::Works classes are carefully designed and constructed for two primary purposes:

1) To help you better understand NT and WDM device drivers

2) To simplify the development of NT and WDM device drivers

The Framework Classes

The **Framework** classes capture the single driver, multiple device architecture defined by the NT/WDM device driver model. These classes handle all of the bookkeeping required to pass messages to the correct handler within a driver. Special support is provided for WDM minidrivers to simplify the callbacks defined by Microsoft for different device classes.

Resource Management Classes

The **Resource Management** classes create a single, simple interface used to request and manage hardware resources. The simple nature of these classes hides a confusing tangle of data structures and calls that are provided by the Microsoft DDK.

Hardware Control Classes

The **Hardware Control** classes simplify the interface to your hardware. Once again, thousands of lines of Driver::Works library code create a simple and consistent interface to an otherwise messy set of calls and structures.

Dispatcher Objects

The **Dispatcher Object** classes unify the *waitable* objects defined by the NT/WDM system.

Memory Classes

The **Memory Classes** encapsulate memory management, simplifying heap management and the use of Memory Descriptor Lists (MDLs).

Container Objects and Utilities Classes

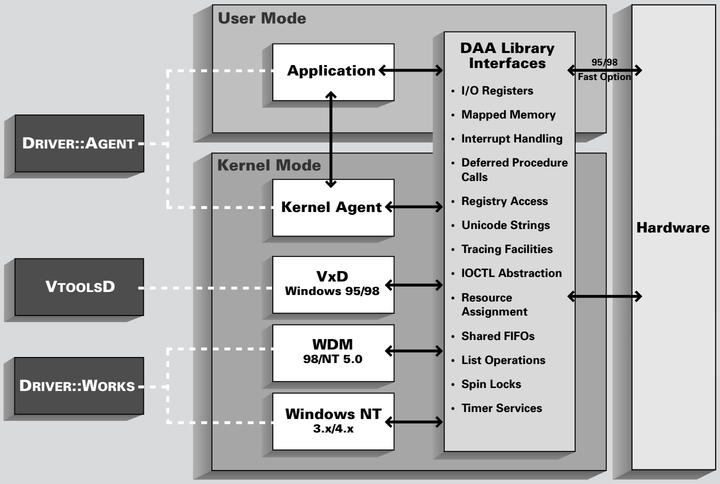
Take advantage of thousands of lines of source code designed to eliminate repetitive coding. The **Container** and **Utility** classes let you use Vireo's bag of tricks, including common data structures such as Lists and FIFOs, debug tracing, string management, and easy and direct access to values stored in the system registry.

Q. What does DAA mean to the Driver::Works developer?

A. Driver::Works implements Vireo's Device Access Architecture (DAA) for use in Windows NT and WDM drivers. Because the DAA interfaces are also provided in VtoolsD for use in Windows 95/98 VxDs, and in Driver::Agent for use in applications, porting is a snap!

Use DAA to take advantage of easy portability between platforms without sacrificing access to the full, native device interfaces available on each system. Vireo's Device Access Architecture is designed to provide optimal performance on each platform while at the same time offering a simple, common set of objects and interfaces that offer source code portability with no limitations or overhead.

Device Access Architecture



FAQs

Q. How does Driver::Works speed up device driver development?

A. Driver::Works is a next-generation environment for device driver development based on a powerful and flexible C++ class library coupled with a powerful code generation wizard.

Over time, Windows application development has evolved to class libraries such as MFC and development tools such Microsoft's Application Wizard. Vireo provides a similar environment for Windows NT and WDM device driver development.

The Driver::Works class library offers thousands of lines of tested code that reduce many complex tasks to simple library calls. In fact, Driver::Works offers by far the most complete device driver library available.

Driver::Works also ships with complete examples that are designed to be used as a basis for further development.

Driver::Works also includes Vireo's unique Driver::Wizard technology. Driver::Wizard guides you through a series of steps that identify many characteristics of your device. Driver::Wizard then generates source code tailored to your driver.

Q. Does Driver::Works incur any performance penalty?

A. No.

The Driver::Works libraries take extensive advantage of C++ inline members to eliminate overhead. In the few cases where even 2-3 instructions of overhead may be too much (for example, the interrupt handler for a high-frequency interrupt) the programmer can choose between a class member or stand-alone implementation.

In real-world tests, we have found no perceptible difference in performance between Driver::Works drivers and drivers written using only the Microsoft DDK.

Q. Is it safe to use C++ in a device driver?

A. Yes.

Vireo has been supplying C++ device driver tools since the release of VtoolsD in 1993. Vireo's libraries provide the infrastructure required to safely use C++ within device drivers. Whether writing in C or C++, developers should avoid writing drivers that require large amounts of stack space. Vireo uses C++ features carefully and selectively to provide the advantages of the language without overhead.

Q. What kind of drivers is Driver::Works designed for?

A. Driver::Works is well suited for both monolithic and layered drivers, NDIS protocol drivers, filter drivers, USB devices, PCI and ISA hardware devices, and many others. New examples are made available on a regular basis.

Q. Why choose Driver::Works?

A. Vireo has an unequaled reputation for providing the tools and support you need to develop device drivers. Our job isn't finished until your driver is working well!

Vireo provides free bug fixes available for immediate download. Timely new versions provide support for new compiler versions and operating system revisions. Vireo also provides new examples and bug fixes on a regular basis.

Driver::Works incorporates years of class-library design experience into a clean, object-oriented system that accurately reflects the underlying system architecture while avoiding the use of arcane C++ language features.

Driver::Works never leaves you in the dark! Full library source code is included in every package, along with support for both Intel and Alpha platforms.

Q. What is included in the Driver::Works package?

A. Driver::Works includes:

- Driver::Wizard — practically writes your driver for you.
- C++ Class Library for NT/WDM driver development
- Windows NT support
- Windows 98 support
- Driver::Monitor — monitor driver activity without a debugger
- Full technical support
- 30-day money back guarantee
- Complete library source code
- RISC platform support
- Full integration with Microsoft Visual C++
- Working examples
- Over 19,000 lines of sample code
- More than 20,000 lines of library source code
- More than 700 pages of printed and online documentation

Q. What is required to use Driver::Works?

A. Driver::Works requires Microsoft Visual C++ version 4.2 or later, and the Microsoft NT DDK, or the Windows 98 DDK. Driver::Works drivers have been tested on both Alpha and Intel single and dual processor platform.

Vireo Software, Inc.
30 Monument Square, Suite 135, Concord, MA 01742
Phone (978) 369-3380 Fax (978) 318-6946 Email sales@vireo.com Web www.vireo.com

Vireo Software