Exactly what is a driver?

If you come from the world of application programming, you probably have a fuzzy idea of what a driver is. You probably know that it's some piece of software that runs in kernel mode that controls hardware. But beyond that…

The easiest way that we know of to describe a driver to application writers is that it is pretty like a Dynamic Link Library (DLL). It has a main entry point like "DllMain" named "DriverEntry". In the case of a Driver, "DriverEntry" it is only called once, when the driver is loaded by the Operating System, whereas "DLLMain" can get called enumerable times, depending on how many applications and threads are using the DLL.

A DLL defines its own interfaces, which it exports via a ".LIB" file that applications can link to; A Driver exports a standard set of entry points in its "DriverEntry" routine by filling in a data structure created by the Operating System called a DRIVER_OBJECT. The Operating System looks at this data structure to retrieve a pointer to the appropriate function to call whenever a request is targeted to a Device, described by a DEVICE_OBJECT, which is created by the Driver to represent either the "physical" or "virtual" device that the Driver supports.

A DRIVER_OBJECT is a block of memory allocated and partially initialized by the Operating System. This object describes where the Driver Code is loaded into memory, the name of the Driver, and it contains Function Address Pointers that must be filled in by the Driver to indicate which functions the Driver supports. One of the Function Address Pointers that must be filled in is a table of Pointers called the "Function Dispatch Table". This table contains one entry for each Major Function Code that the Operating System supports. There are currently 28 functions that the driver can elect to support however most driver usually support about 8 functions; IRP_MJ_CREATE, IRP_MJ_CLOSE, IRP_MJ_READ, IRP_MJ_WRITE, IRP_MJ_PNP, IRP_MJ_POWER, IRP_MJ_DEVICE_CONTROL, and IRP_MJ_SYSTEM_CONTROL (By default the Operating System initializes all the entries in this table to a routine which indicates that the Major Function is not supported). Whenever a user makes a request to the Operating System to perform a function, the Operating System determines which Major Function Code corresponds to the operation and delivers the operation to the Device that the user specified.

As mentioned above, a Driver creates DEVICE_OBJECTs to describe either "physical" or "virtual" devices that the Driver supports. These DEVICE_OBJECTs are targets of I/O requests and the DEVICE_OBJECTs must have names created for them so that user applications or other drivers can "Open" those devices to communicate with them. So, for example, when a user application does a "CreateFile" Win32 request specifying the name "COM1", what the operating system is doing is performing an open of the DEVICE_OBJECT whose name is "COM1". Upon a successful completion of the "CreateFile" request, the application is returned a HANDLE which it uses in all subsequent requests that it is making to the Device.

Remember that Drivers are processing requests sent by programs running in user mode and can also sometimes be processing requests coming from other drivers in the Operating System. These requests are targeted at DEVICE_OBJECTs that the Driver has created. In the case of user mode applications, the request is built into a packet called an I/O Request Packet (IRP). The IRP is built by the I/O Manager, and Operating System Component, and contains the Major Function Code indicating to the Driver what it is supposed to do on the Device. Once the IRP has been built, the I/O Manager calls the Driver at the entry point it has exported via the "Function Dispatch Table" contained within the DRIVER_OBJECT. In the case of one driver sending a request to another driver, the driver that communicates with your driver is responsible for building an IRP and using the Operating System routines to pass the IRP to your Driver.

Once the Driver receives the IRP it has 3 options on how to process the request. It can either immediately do the operation that the user has requested, in which case it can immediately tell the Operating System that the IRP is complete; It can hold onto the IRP and tell the Operating System that its processing is Pending and will be completed later on (probably because it has to program it's hardware device to do the requested operation); or finally, the Driver can determine that it cannot process the request itself and must pass the IRP to some other driver who will process the request.

If the Driver is controlling some actual physical device, then it is the Drivers responsibility to program the device to perform the requested operation. Typically, these devices "Interrupt" when the requested operation is complete. A Driver whose device "Interrupts", registers an "Interrupt Service Routine" (ISR) with the Operating System. This registration associates the "Interrupt" with the ISR. So, when the Device Interrupts the Operating System calls the appropriate ISR. It is the ISRs responsibility to determine if it is its device interrupting, since the device could be on a shared interrupt bus. If it is not its device interrupting then it returns and indicates to the Operating System that the interrupt was not handled, in which case the Operating System calls the next registered interrupt handler. If the Interrupt was for this device, however, then the ISR must save away any context it needs about the interrupt, acknowledge the interrupt (i.e. indicate to the device that it has seen the interrupt), and finally queue a "Deferred Procedure Call" (DPC).

A DPC routine is a kernel mode callback and it is responsible for completing the IRP whose work was performed by the signaled interrupt and for propagating the execution of the driver. In other words, it is responsible for making sure that any IRP that was received by the Driver and queued because the device was busy working on a previous request, is taken off the queue and started on the device.

So, that is a driver in a nutshell. There is one other critical thing that a potential Driver write needs to understand: Drivers can't make mistakes! Unlike user mode programming, where a bug in the user program causes just the application to abort, in kernel mode, a Driver making an error causes the system to abort (i.e. a system crash, the blue screen of death), hideously corrupts something, or hang. Therefore, driver writing requires good design skills, attention to detail, good testing skills, paranoia, and patience. Oh, and it also requires understanding how the operating system works.