Supporting Multiple Pointing Devices in Microsoft Windows*

Michael Westergaard

Department of Computer Science, University of Aarhus
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
mw@daimi.au.dk

Abstract. In this paper the implementation of a Microsoft Windows driver including APIs supporting multiple pointing devices is presented. Microsoft Windows does not natively support multiple pointing devices controlling independent cursors, and a number of solutions to this have been implemented by us and others. Here we motivate and describe a general solution, and how user applications can use it by means of a framework.
The device driver and the supporting APIs will be made available free of charge. Interested parties can contact the author for more information.

1 Introduction

Interaction techniques for modern desktop applications are changing. Earlier, it was common to use a single pointing device (typically a mouse or a track-ball) and a single keyboard for interaction. However, it is no longer rare to encounter applications providing more sophisticated interaction techniques, for example by using multiple pointing devices each controlling its own cursor. Microsoft Windows does not fully support more than one pointing device natively. If multiple pointing devices are attached, they all control the same cursor. A number of solutions have been made in order to overcome this problem, and be able to have multiple independent cursors. Here we describe a general implementation allowing programs to use multiple pointing devices with little effort. The solution is designed for Microsoft Windows 2000/XP. In the following we will refer to the supported platforms just as “Windows” and take “mouse” to mean any pointing device (mouse, track-ball, pen etc.).

Multiple mice can be advantageous in different settings. For instance one can use a mouse in one hand and a track-ball in the other, using the mouse for precision navigation, and the track-ball for larger moves. This could be used to move a window while resizing it, which has proven useful for customising the layout of windows on the desktop. This is called two-hand resize. This interaction can also be used in other settings, for example to zoom and move elements in one action. One can also imagine using two or more mice, each having a different purpose. This could be one mouse for drawing rectangles and another for drawing circles. The user then has a direct, physical notion of switching from one instrument to another. This is useful when drawing Coloured Petri Nets (CPN) [11], which is a graphical modelling language. For the purposes of this

* Supported by Microsoft Research Limited by a grant to the CPN Tools project

example, a CPN is a directed graph with two different kinds nodes, some are drawn as rectangles and some are drawn as circles. Also, several modern interaction techniques, such as tool-glasses [3] or floating palettes, require at least two mice; one mouse for moving the tool palette and another for selecting the tool. We believe that using multiple mice will cause the user to switch the load from one hand to the other, thereby reducing the risk of obtaining cumulative trauma disorders. One can also imagine more people each having a mouse, working together to solve a task on a large screen. It is one of the purposes of the WorkSPACE [17] project to investigate this.

The CPN Tools [2,4] application is an editor and simulator for Coloured Petri Nets, which makes use of advanced interaction techniques. From the very beginning one of the goals of the project has been to use advanced interaction techniques, including, but not limited to, investigating advanced interaction techniques using multiple pointer devices. At an early stage of the project the limitations of the support for multiple mice in Windows became apparent, and a custom solution was implemented. This custom solution was to spawn an external application, which would directly probe the serial port to which the mouse was attached, decode the signals from the attached mouse, and send events to the main application. As the need for support for arbitrary many mice arose, this first solution proved to be insufficient. First of all, it became increasingly difficult to obtain serial mice, and we had to support USB mice as a minimum, and preferably also serial mice, PS/2 mice and "any future mouse", while at the same time providing a common interface for all mice. This was not possible with the first solution, where the primary mouse was controlled by Windows and the secondary by the spawned application. Second, the solution required disabling the secondary mouse in Windows, which forced the user to have an extra mouse on the desk, which was essentially useless outside the CPN Tools application. Third, the solution was not generic; a new implementation had to be made if the protocol of the mouse changed, for example to USB. For these reasons we decided to implement a new solution. As we had a great deal of experience from the earlier implementation, we decided to make an implementation independent of CPN Tools, generic enough for other uses. One goal of the implementation is to make an API that is at least as easy to use as the standard mouse interface provided by Windows.

The difficulty of the task can be appreciated by looking at other attempts of using multiple mice. The MAME:Analog+ [1] (a gaming library supporting multiple mice) and Multiple Input Devices (MID) [7,8] (a more generic Java library for multiple mice) projects try to use multiple mice, but fail to do so on Windows 2000. Both cite the same message we also received from Microsoft: The inability to distinguish the input from multiple mice on Windows NT/2000/XP is a design decision, and both projects state that it is impossible use multiple mice on these operating systems. In addition MAME:Analog+ states that you must use USB devices in order to get multiple mice. Furthermore, both libraries focus on using one mouse per user, and do not acknowledge the potential of using multiple mice for each user.

2 Design Overview and Requirements

The device driver and the supporting APIs are designed to provide an interface between the hardware and the user application. The overall architecture can be seen in Fig. 1.

The device driver takes care of filtering the mouse events as received from the hardware dependent driver. The low-level API takes care of communicating with the device driver and provides a callback mechanism to the upper layers. The high-level API allows user applications to receive mouse events in an efficient way, either by subscribing to events or by polling. This abstraction is useful for a framework supporting multiple mice. Octopus [5] is an example of such a framework. The integration with the Octopus will be described in a little more detail in Sect. 3. CPN Tools is an example of an application building on top of Octopus.

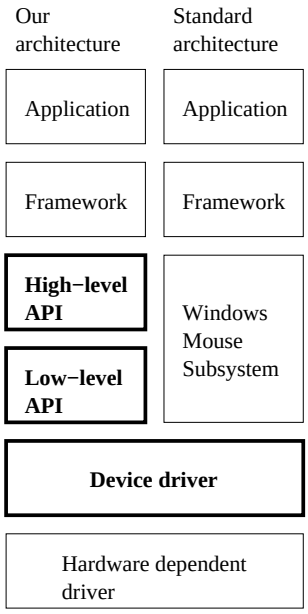

Fig. 1. The architecture of the device driver and the supporting APIs (left) contrasted to how mouse input normally works (right). These two can coexist for compatibility with old applications. This paper focuses on the boldfaced components.

2.1 Device Driver

Even though a mouse is meant for use in an application using the device driver and APIs described in this paper, it must be available as a normal mouse in Windows, because otherwise it would be impossible to use it in applications not using the device driver.

In the following we will refer to this normal mouse and the associated cursor as the Windows mouse/cursor. As we would like to support a great deal of different mice, it is not fruitful to implement a device driver for every different kind of mouse in existence. The described driver can and should be installed on all mice on a system in order to let applications use a single interface to all available pointing devices.

For these reasons, it was decided to implement the device driver as a filter driver [15]. This allows us to hook into the device stack of any mouse, installing our own device interface in addition to the mouse interface and filter what hardware events we want to pass on to the Windows mouse. Furthermore, a filter driver can be installed at a convenient point, where the differences in hardware have been abstracted away, but it is still possible to distinguish the different mice.

The following three paragraphs assumes some knowledge about Windows driver development and parts of the Windows API, but can be skipped without losing the context.

In order to make the implementation as simple as possible, we started out with the *moufiltr* driver from the Microsoft Windows Driver Development Kit [14]. This driver is a good starting point when trying to develop mouse filter drivers; basically one only has to change one function in order to filter out mouse events. The moufiltr driver is installed as a filter on top of the standard mouse driver *mouclass*, making it easy to support serial, PS/2 and USB devices with a single driver. It is easy to add our own device interface using *IoRegisterDeviceInterface*.

The first problem one encounters is the fact the Windows registers itself as the only user of any mouse class device, including the ones we install the filter on top of. Normally one communicates with an arbitrary driver by looking up the correct device (an instance of the driver). Then one gets a handle to use for communicating using *CreateFile*, creating a "file" in the drivers name-space. This handle can be used for communicating with the device driver using Device Input and Output Control (IOCTL) [13] using the *DeviceIoControl* function. As Windows registers itself for exclusive access to the mouse, it is not possible to acquire a handle with permissions to perform IOCTLs. However, it is possible to get a handle with permissions to query device attributes. When we try do do this, the device driver is informed and passed information about what file we try to query information on. We exploit this to encode function calls in the filename, getting filenames of the form:

```
<device-name>\execute\<function>(\<parameter>)*
```

This might seem like an inelegant solution, but it has two nice properties: it is simple and it makes it easy to have an inverse function. As simplicity was one goal, this in itself justifies the solution. The value of the inverse function should be obvious, if one considers the fact that applications crash and application programmers forget to close open files. As Windows automatically closes all open files belonging to an application when the application terminates or crashes, we obtain a similar effect with this approach. If an application has told the driver it wishes to use its services on startup, the application automatically tells the device driver that it does not want to use them anymore on termination and even on crashes.

The second problem is that we would like to perform callback to user-mode when a mouse event occurs. As this probably will happen frequently, we want a fairly light-

weight mechanism, and settle on performing a callback. We have considered using named events, but find this to be a bit heavy for our purposes. We also would like to use the undocumented function *KeUserModeCallback*, which provides very fast callbacks to user-mode, but is impossible, since we do not know if we are in the context of the thread. We finally decided to use Asynchronous Procedure Calls (APC). APCs can perform a call from one part of the system to another, but in the thread of the receiver. We can thus make a call from the device driver to a user-mode callback, in the context of the user-mode program. The APC functionality is included in the standard libraries of Windows, but it is not well documented. Fortunately [6, 16] contains enough information to get started.

The device driver interface offers four functions to user-mode applications: *Get*, *UnGet*, *Suspend*, and *UnSuspend*. The Get function is for hooking on to the mouse and registering a user-mode callback. When this function is called on a device it stops passing on events, so the Windows mouse will not know of them. In effect we tear the physical mouse away from the logical Windows mouse and into our own logical mouse. The UnGet is the reverse function, drop the callback and give the mouse back to Windows. The Suspend function is for temporarily suspending the mouse from an application. That is used when an application has called the Get function, and wants Windows to temporarily receive events from the mouse, but at the same time receive the events itself and prevent other applications from getting the mouse. This is interesting if the mouse is moved outside of the application window, and the user should have a normal Windows cursor again.

All in all the device driver can be in one of the three states depicted in Fig. 2. The text in parentheses indicate which processes may call a given function, only the indicated transitions are legal.

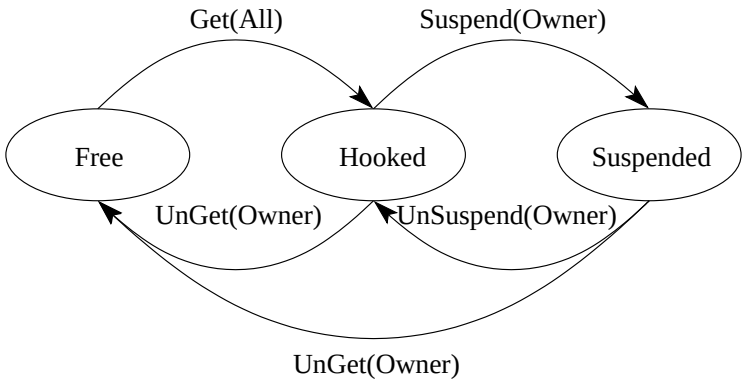

Fig. 2. The device driver can be in one of three states. In the free state the mouse behaves like a normal mouse. In the hooked state, the mouse is grabbed by an application, and does not control the Windows mouse. In the suspended state, the mouse behaves to the user as if it is in the free state, but the application, which has called the Get function, is still in control of it. The names in the parentheses indicate which processes are allowed to call the function to perform the transition.

The device driver ensures that foreign applications cannot get the mouse when an application is using it. It does so by only allowing calls to certain functions from different states, and only from certain processes. When a process has called the Get function, it becomes the owner of the device, and until the UnGet function has been called, only the owner process can call functions for this device. This is done in order to prevent foreign processes from terminating access to the device driver on behalf of another application.

2.2 Low-level API

Even though the device driver provides a fairly simple interface to the mice, it is a bit tedious to communicate with the device driver, because one has to enumerate all mice and search for the mice to use and manually call the functions provided by the device driver. The low-level API makes this task simpler by providing a simple interface to setting up a callback. In addition it simplifies the identification of the mice to simply being natural numbers (as opposed to generic device identifiers as provided by Windows) and adds extra error handling. Basically the low-level API hides the operating system support for multiple mice, so when higher levels are based on the low-level API, it is easy to port them if, for example, Microsoft decides to fully support multiple mice. An implementation of the low-level API currently exists in Standard C and BETA [10, 12]. The BETA implementation builds upon the C implementation.

The low-level API offers five essential functions: *RegisterCallback*, *GetMice*, *UnGetMouse*, *SuspendMouse*, and *UnSuspendMouse*. The low-level API includes a few more functions, but they are not essential for understanding the API. RegisterCallback takes care of registering a callback to the application. The callback is a function, which is called whenever one of the hooked mice sends events. Only one callback can be registered for all mice, and this must multiplex between all the hooked mice. If this is not acceptable, another low-level API has to be implemented. GetMice gets a given number of mice, or all available mice when given 0 as parameter. The user of the API does not have any control of which mice will be hooked. This is not seen as a major problem, as an application will probably always hook all available mice, and, eventually supported by a framework, let the user decide which mice are used for what. GetMice returns the number of mice it actually got. The mice will be numbered from one to and including the number of mice we actually got. Any bookkeeping needed when UnGetting mice later on must be taken care of by some higher level. When a mouse has been hooked by calling this function, events are sent via the registered callback. If no callback is registered, incoming events are ignored. The UnGetMouse, SuspendMouse, and UnSuspendMouse function calls are passed on to the specified mouse if it has been hooked by calling the GetMice function.

The API is not thread-safe, i.e. calls of functions must be protected by a mutex if they are to be done in different threads. This is acceptable as it is assumed that the functions will be called in a single thread or during application initialisation. The API makes sure to serialise calls to the registered callback, so it does not have to be thread-safe either.

2.3 High-Level API

The low-level API is callback based. As the BETA language does not support this sufficiently well (it does not use true operating system threads), it is decided to implement an even higher level API. This API must process the information gained from callbacks and offer it to the application by means of standard events. Furthermore, as already mentioned, the low-level API is not thread safe, so the high-level also adds this feature. The high-level API also takes care of acceleration of all the input devices as well as offering to draw a mouse cursor for each mouse. Finally the high-level API offers calls for obtaining the absolute and relative position of each mouse. Currently the high-level API is designed to be statically linked to an application and manage the access to the driver for that application. Multiple applications, each statically linked to the high-level API can run simultaneously, but any given mouse can only be hooked by one application at a time; if, for example, a user has three mice installed, and uses an application that requires only 2 mice, the last mouse is available to other applications. All mice are available on a "First come, first served" basis. The high-level API is used and designed for the Octopus framework, and may not suit the needs of other frameworks or applications. In the future one might develop a Dynamic Link Library (DLL) implementing the high-level API.

The high-level API spawns a thread to listen for incoming callbacks. The received information can then be processed and sent to the main application window using the *PostMessage* function. If desired, the cursors are updated.

The high-level API offers ten functions: *Initialise*, *Cleanup*, *SuspendMouse*, *UnSuspendMouse*, *GetRelativePosition*, *GetAbsolutePosition*, *SetAbsolutePosition*, *SetCursor*, *LockCanvas* and *UpdateCursors*. Initialise and Cleanup are meant to start up and close down the high-level API, SuspendMouse and UnSuspendMouse are just thread safe versions of the low-level API functions with the same names, and will not be described further. GetRelativePosition, GetAbsolutePosition, and SetAbsolutePosition are for applications that prefer polling for mouse information over receiving the information by events. SetCursor, LockCanvas, and UpdateCursors are for setting up and drawing cursors.

The Initialise function starts up a thread to listen for callbacks from the low-level API. It takes as arguments the number of mice desired, the window to send events to, a Display Context to draw cursors on and some flags describing the behaviour of the API. The flags can turn on or off features of the API. For instance the programmer can select whether he desires events to be sent to a main window (if not, it is not necessary to give a window as a parameter), whether he wants a cursor to be drawn (if not, it is not necessary to pass a Display Context as a parameter), whether the cursor should be clipped to a passes Display Context, whether the mouse should be accelerated, and whether events should be sent when the mouse is suspended. The Cleanup function UnGets all hooked mice, kills the thread started by Initialise and releases all resources obtained while running the API.

As already mentioned, the GetRelativePosition, GetAbsolutePosition and SetAbsolutePosition functions are primarily meant for applications that prefer polling for mouse information, but they can also be used in combination with events if desired. All of these functions take as parameters the number of a mouse. The SetAbsolutePosition function

also takes a point as a parameter. The two Get functions return a point. The GetRelativePosition function gets the relative movement of the specified mouse since the last call to the function. The outcome is affected by acceleration, but is not affected by clipping. The GetAbsolutePosition and SetAbsolutePosition functions get/set the absolute position. The position is affected by acceleration and clipping.

The SetCursor function sets the cursor for a given mouse. It takes as parameters the number of the mouse and a cursor and the hot-spot of the cursor. The UpdateCursors function redraws the cursors for the mice. This is called automatically whenever a mouse is moved, but it can also be called manually when the screen has been updated. If LockCanvas is called, UpdateCursors will not be called automatically until UpdateCursors has been called manually from within the thread that called LockCanvas. This function should be called just before the application redraws the Display Context and UpdateCursors should be called just after the update.

3 Applications and Future Work

The device driver and the corresponding APIs are being integrated with the Octopus framework and thus the CPN Tools application. Octopus is an example of a framework from the left-hand column in Fig. 1, and CPN Tools is an application, also from the left column. The general considerations in the following are primarily based on experience from the Octopus framework.

The framework must take care of completely abstracting the notion of position and button clicks from the user application. It must provide a service very similar to the service provided by any other application framework today. It must thus have a set of widgets and send events to these whenever a mouse action happened. For instance in the Lidskjalv User Interface Framework [9] for BETA, a pushButton has an eventHandler, which again has an onClicked event. This can by inheritance be bound to receive notification when the user clicks with the mouse on the button. Similarly the Octopus framework offers some widgets and services for using multiple mice. It offers a mechanism similar to the onClicked event, but it must also inform the receiver of an event what the other mice are currently doing, in order to allow the application programmer to specify some multi-hand interactions, like the two-hand resize described in Sect. 1. As the operating system does not support multiple mice, the framework must also make a higher level mechanism available for handing over control of the mouse to the operating system when needed. We propose that when any mouse leaves the application window, it is suspended (i.e. it now controls the Windows mouse), and when any previously hooked mouse enters the application window, it is unsuspended, and the Windows mouse is left alone. Finally a framework might also support the addition and removal of mice while running, but as it is not expected that adding or removing mice is a very common action, this can be left out.

Octopus currently uses the driver and the APIs described in this article. The device driver is independent of the Octopus framework and the CPN Tools application, but the implemented APIs only implement the features we see a need for in Octopus and CPN Tools at the moment. For this reason there is plenty room for improvement. An example of this is the fact that the application programmer has no control of which mice will be

hooked by the GetMice function. An obvious improvement is to offer some functions letting the application select mice based on a textual description, the capabilities of the mouse (2 or 3 buttons, scroll-wheel etc.) or position on the bus.

The current implementation of the device driver and the APIs does not have very good support for multiple applications using mice concurrently. As described, the device driver actually prevents multiple applications from using the same mouse at once. This behaviour is acceptable as long as only few applications make use of the device driver and the APIs, but is not acceptable when using multiple mice becomes more common. We thus propose that it might be fruitful to look into making a sharing mechanism. On possibility would be to install a system service and simply send the mouse events to the window the mouse is currently over.

The solution presented in this paper is layered on purpose. It is very easy to build other APIs on top of the device driver as well as porting the current APIs to other programming languages. The high-level API presented in this paper is the one that best satisfies the requirements of Octopus, but other frameworks may have other requirements. For example one might implement a framework compatible with the Windows widget system that simply adds multiple cursors to Windows, which can be used with existing programs, allowing multiple users to use different applications at the same time. Furthermore, should Microsoft decide to make it possible to distinguish between different mice, we can simply drop the device driver, change the low-level API to use the new functionality and the rest of the implementation will work with no problems.

It might also be interesting to change the MAME:Analog+ and MID implementations to use the device driver, extending their usability a lot, by allowing to use multiple mice in more settings, such as under Windows 2000 or without the limitation of only using USB devices.

4 Conclusion

In this paper we have presented the main problem with having separate cursors for multiple mice, the fact that Windows does not distinguish between the different mice at the level of the application program.

We have described the architecture of a solution, using a device driver and two APIs, each of which removes some of the dirty work, and provides the programmer with a cleaner interface to multiple mice.

Finally we have shortly described the road ahead: some of the considerations which must be put into the development of a framework supporting multiple mice. Basically it must support primitives similar to the current (widgets with events), but it must also provide information about what the other mice are doing.

The device driver and the supporting APIs will be made available free of charge. Interested parties can contact the author for more information.

Acknowledgements. The author would like to thank Kurt Jensen, Jens Bæk Jørgensen, Thomas Mailund, and the reviewers for constructive comments on this paper.

References

1. A MAME site For Improved Analog Input. `http://www.urebelscum.speedhost.com/`.
2. M. Beaudouin-Lafon and M. Lassen. The Architecture and Implementation of CPN2000, A Post-WIMP Graphical Application. In *Proceedings of ACM Symposium on User Interface Software and Technology*. ACM Press, November 2000.
3. L. Bier, M. Stone, K. Pier, and W. Buxton T. De Rose. Toolglass and magic lenses: the see-through interface. In *Proc. ACM SIGGRAPH*, pages 73–80. ACM Press, 1993.
4. University of Aarhus CPN group. CPN Tools homepage. `http://www.daimi.au.dk/CPNTools/`.
5. M.B. Enevoldsen and M. Lassen. Octopus: A PostWIMP Framework for New Interaction Techniques. `http://www.ecoop2002.lcc.uma.es/tpDemonstrations.htm#d13`.
6. J. Finnegan. Nerditorium. `http://msdn.microsoft.com/library/en-us/dnmsj99/html/nerd0799.asp`, July 1999.
7. J.P. Hourcade and B. Bederson. Multiple input devices: Mid. `http://www.cs.umd.edu/hcil/mid/`.
8. J.P. Hourcade and B.B. Bederson. Architecture and Implementation of a Java Package for Multiple Input Devices (MID). Technical report, Institute for Advanced Computer Studies, Computer Science Department, University of Maryland, May 1999.
9. Mjølner Informatics. Lidskjalv: User Interface Framework – Reference Manual. Mjølner Informatics Report, MIA 94-27, available at `http://www.mjolner.dk/mjolner-system/documentation/lidskjalv-ref/index.html`.
10. Mjølner Informatics. The Mjølner System: BETA Language. `http://www.mjolner.dk/mjolner-system/beta_en.php`.
11. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1992.
12. O.L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.
13. Microsoft Corporation. Device Input and Output Control (IOCTL). `http://msdn.microsoft.com/library/en-us/devio/devio_7pb3.asp`.
14. Microsoft Corporation. Microsoft Windows Driver Development Kits. `http://www.microsoft.com/ddk/W2kDDK.asp`.
15. P.G. Viscarola and W.A Mason. *Windows NT Device Driver Development*, pages 235–239. Macmillan Computer Publishing, 1999.
16. Anatoly Vorobey. User mode APCs. `http://www.cmkrnl.com/arc-userapc.html`, May 1997.
17. WorkSPACE: Distributed Work support through component based SPAtial Computing Environments. `http://www.daimi.au.dk/workspace/index.shtml`.