Citrix XenClient

OEM Features

Version 1.6

Contents

Goal 3

How Do OEMs Expose Value-Add Features In General? 3

OEM Features in Virtual Machines 4

Enabling OEM Features In Guest Environment 5

Guest Level OEM Features Implementation 5

Components Involved 5

MOF Data 8

Known Issues 8

Alternate Approaches 9

Dependencies 9

Installation Requirements 9

References 9

Appendix A: acpi-wmi.h file 10

Goal

Provide guest level support for OEM value-add features thus enabling OEM applications and OEM special buttons/hotkeys to work within user VMs.

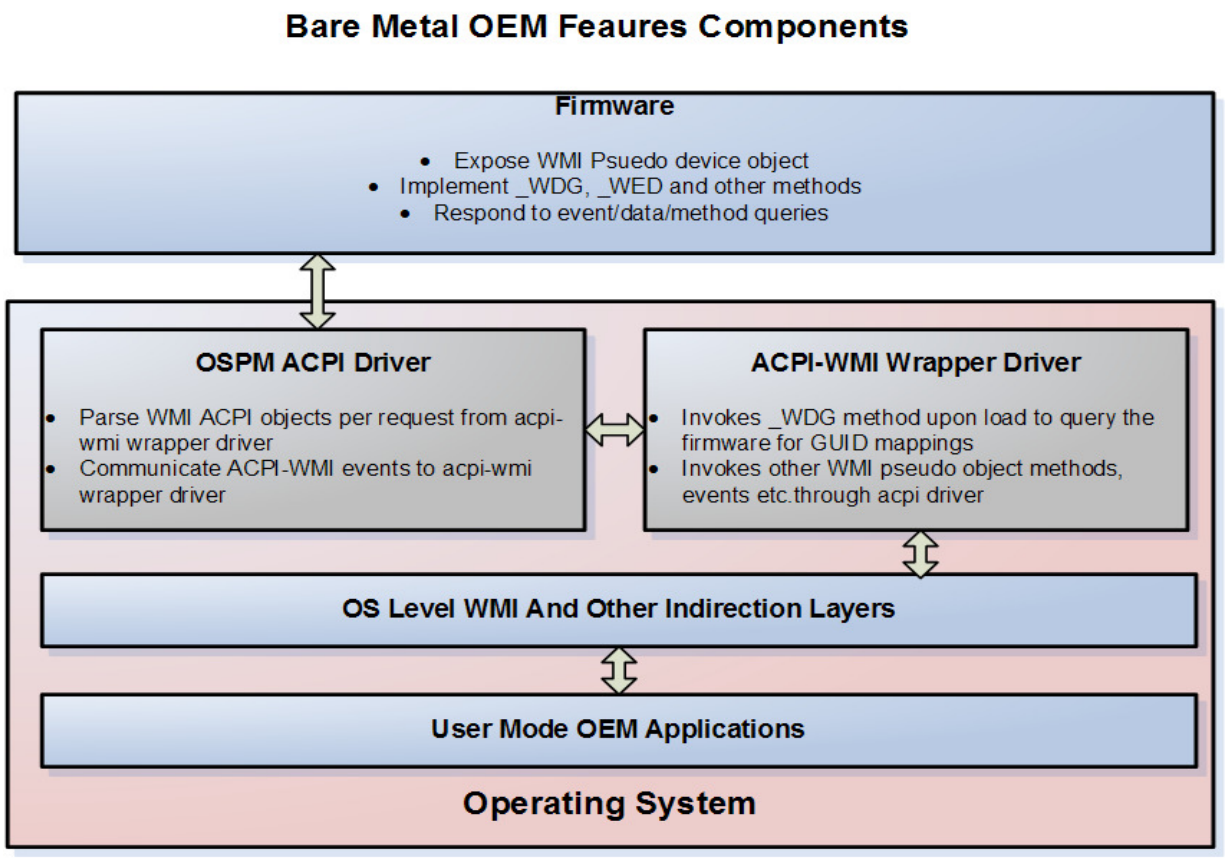
How Do OEMs Expose Value-Add Features In General?

Some OEMs expose their value add functionality by exposing a WMI-mapping pseudo device object in their ACPI space. This device object is expected to have a pre-defined PNP ID of PNP0c14. This device object is also expected to expose some well known methods as outlined in the ACPI-WMI specification.

The operating system that supports such ACPI-WMI implementation will first look for a pseudo device object with PNP ID PNP0c14 in the ACPI space and load it's ACPI-WMI wrapper driver. In case of Windows it would be wmiacpi.sys. This wmiacpi.sys relies on windows ACPI driver acpi.sys to navigate through the ACPI space and execute ACPI methods.

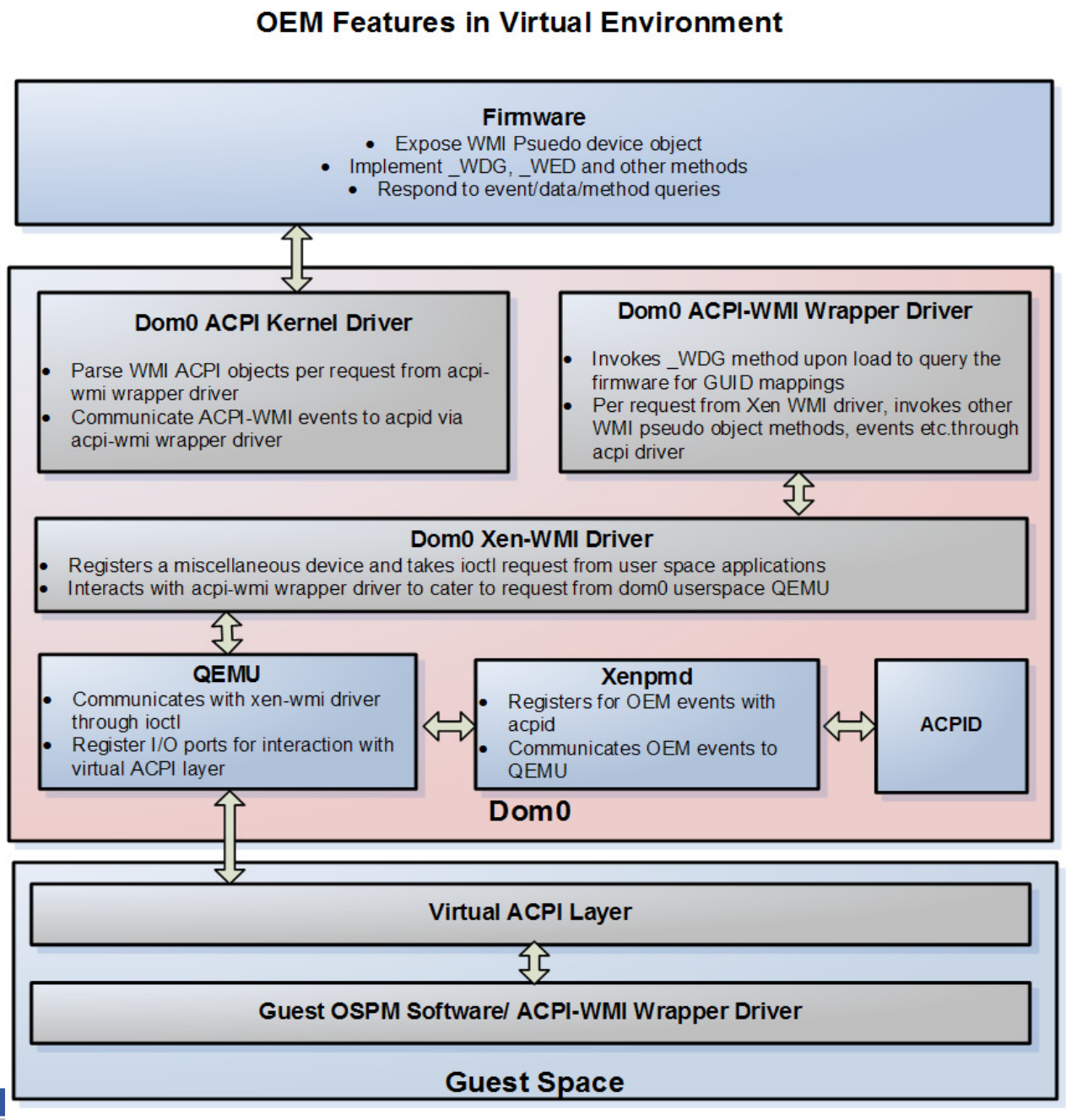
Upon loading, one of the first step that wmiacpi.sys does is to look for and call a well known ASL control method - Data Block GUID Mapping control method (_WDG) exposed through the pseudo device object. This method returns a buffer with GUID mapping information for the data blocks, events and methods exposed by the pseudo object. With this information in hand, OEM applications can query relevant data, call appropriate methods implemented in the firmware by referring to its GUID and expose OEM features to the end user.

OEM special buttons and hotkeys are generally wired to generate ACPI interrupt. The OS ACPI driver would then notify acpi-wmi driver which would then call a well known entry point _WED to get further information about the triggered event and act accordingly.



OEM Features in Virtual Machines

As OEM specific "value-add" knowledge is provided through the firmware, our virtual firmware has to expose to the guest the same level of information provided through the base firmware to provide a bare metal comparable experience. Once that is done, the virtual firmware need to respond to queries and method invocations the same way the platform firmware would respond. Following explains how we achieve that in our implementation.



Enabling OEM Features In Guest Environment

OEM features support is enabled within a guest by setting "oem-features = 1" in the configuration file. For an unsupported platform, this option is ignored.

Guest Level OEM Features Implementation

When the OEM value add feature is enabled for the guest through the configuration file option, our toolstack would set the appropriate value in xenstore to indicate to the rest of the components that OEM features should be enabled for the relevant domain. HVMLoader will then check for OEM name/model of the platform it is running on and if support for the platform exists, it loads the appropriate vendor/model specific SSDT table to extend our vACPI to expose OEM features to guest. The guest OS in the user VM will then load the acpi-wmi wrapper driver and further communicate with our virtual ACPI layer through well known entry points. Our vACPI layer would write to appropriate ports thus delegating QEMU to get the information it needs. QEMU in turn relies on the dom0 Linux ACPI-WMI wrapper driver through our Xen WMI wrapper driver to get the information it needs from the actual firmware and returns it to our vACPI layer which would then communicate it to the guest.

When a special button or hotkey is pressed thus generating an ACPI event, we have tweaked the Linux ACPI-WMI driver to route those events to ACPID. Our Xen power management daemon which is already registered to receive ACPID events will update appropriate xenstore node watched by QEMU which would then inject the SCI to guest space. Guest ACPI-WMI wrapper driver would then through its ACPI driver invoke _WED method exposed by the pseudo object in the virtual ACPI space. This will result in a call to the base firmware through QEMU, Xen and Linux WMI drivers resulting in the event data information passed back to the guest and the guest takes appropriate action based on that event data.

Components Involved

Toolstack

If the "oem-features" option is specified in the configuration file, the toolstack will add the following entry in xenstore for QEMU and other components to enable OEM features at its level -

/local/domain/0/device-model/<domid>/oem_features

HVMLoader, Virtual Firmware

Each vendor/model specific OEM ACPI-WMI pseudo object is implemented within a SSDT. HVMLoader while building the guest space will first communicate with QEMU (by writing to and then reading from port 0x96) to determine whether or not OEM features is enabled. If enabled, it will then determine whether or not OEM features is support for the current vendor and model and if supported, loads SSDT created for the specific vendor/model.

We rely on SMBIOS information passed through to the guest to figure out the vendor/model name. So OEM features implementation has a dependency on SMBIOS pass-through.

Our vendor/model specific SSDT has a single ACPI WMI pseudo object whose implementation is compliant with the ACPI WMI specification and mirrors the underlying base firmware's implementation for the critical well known method - _WDG. By mirroring the base implementation for _WDG, we are exposing the same metadata the underlying firmware exposes thus providing the same set of data blocks, methods and events the underlying firmware provides. The rest of the ASL methods, query/set data blocks provided through our SSDT are based on what is exposed through _WDG and thus same as the underlying firmware though the ASL code implemented within those methods is different than that of the underlying firmware. Our ASL implementation in the virtual space with the exception of _WDG simply writes to I/O ports in a specified order to communicate to QEMU what method was invoked by the guest. QEMU would then, through our dom0 Xen and Linux WMI driver communicate with base firmware, invoking the same methods invoked by guest but at platform level and returns actual firmware data back to the guest. The I/O ports and sequence of commands used to communicate are covered under QEMU section.

QEMU

QEMU registers the following I/O ports and take control when the virtual firmware reads from/writes to the below ports -

- Command port 0x96
- Data Port 0x98 for byte data transfer and 0x9a for DWORD data transfer

Following are the commands written by the virtual firmware to the command port -

- XEN_ACPI_WMI_CMD_INIT – This is the first command written to the command port to indicate start of a request sequence. When this command port is written to, the corresponding data port is expected to hold on of the below to indicate the type of invocation -
  - XEN_ACPI_WMI_EXEC_METHOD
  - XEN_ACPI_WMI_QUERY_OBJECT
  - XEN_ACPI_WMI_SET_OBJECT
  - XEN_ACPI_WMI_GET_EVENT_DATA

- XEN_ACPI_WMI_CMD_GUID – This command is issued by the virtual firmware through the command port to indicate to QEMU that the GUID data would now be written through the data port
- XEN_ACPI_WMI_CMD_OBJ_INSTANCE – This command is written to command port before passing the object instance number through the data port
- XEN_ACPI_WMI_CMD_METHOD_ID – Indicates a method ID is available through data port
- XEN_ACPI_WMI_CMD_IN_BUFFER – This command is issued by the virtual firmware before it populates the data port with argument buffer

- XEN_ACPI_WMI_CMD_IN_BUFFER_SIZE – Indicates method parameter length is available through data port
- XEN_ACPI_WMI_CMD_EVENT_ID – This command is issued by virtual firmware before writing a event ID to data port
- XEN_ACPI_WMI_CMD_OUT_BUFFER – This command is issued before a command execution to request QEMU to populate the output buffer through data port
- XEN_ACPI_WMI_CMD_OUT_BUFFER_SIZE – This command is issued by the virtual firmware to request the size of output buffer before issuing XEN_ACPI_WMI_CMD_OUT_BUFFER command
- XEN_ACPI_WMI_CMD_EXECUTE – This is the concluding command of a sequence of commands indicating to QEMU that it has all the information it needs to communicate with the base firmware

A sample sequence of commands executed by our virtual firmware to request QEMU to invoke equivalent base firmware command, let us say for a method would be -

1) Issue XEN_ACPI_WMI_CMD_INIT command before writing XEN_ACPI_WMI_EXEC_METHOD to data port
2) Issue XEN_ACPI_WMI_CMD_GUID command before communicating the GUID associated with the method through data port
3) Issue XEN_ACPI_WMI_CMD_OBJ_INSTANCE before writing object instance number through data port
4) Issue XEN_ACPI_WMI_CMD_METHOD_ID command before writing method ID to data port.
5) Issue XEN_ACPI_WMI_CMD_IN_BUFFER_SIZE and XEN_ACPI_WMI_CMD_IN_BUFFER before writing input buffer size and input buffer through the data port
6) Lastly, issue XEN_ACPI_WMI_CMD_EXECUTE followed by XEN_ACPI_WMI_CMD_OUT_BUFFER_SIZE and XEN_ACPI_WMI_CMD_OUT_BUFFER when appropriate

Xen WMI Wrapper Driver

Xen WMI wrapper driver is a dom0 kernel driver that sits between QEMU and Linux ACPI-WMI wrapper driver. Its main functionality is to expose IOCTL interface for user mode dom0 component - QEMU to communicate with the Linux ACPI-WMI driver. It provides the following IOCTLs -

- XEN_WMI_IOCTL_CALL_METHOD
- XEN_WMI_IOCTL_QUERY_OBJECT
- XEN_WMI_IOCTL_SET_OBJECT
- XEN_WMI_IOCTL_GET_EVENT_DATA

Additional data structures provided by the Xen WMI driver can be found in Appendix A: acpi-wmi.h file at this end of this document.

QEMU upon receiving XEN_ACPI_WMI_CMD_EXECUTE command will build appropriate parameter list and invoke relevant IOCTL which will then be communicated to the Linux ACPI-WMI driver.

We provide a XEN_ACPI_WMI_WRAPPER kernel build configuration option to choose whether or not to build this driver. It has dependency on the Linux ACPI-WMI driver and is configured to build by default.

Linux ACPI-WMI Wrapper Driver

We have taken the Linux ACPI-WMI driver as a base and modified it work for our use

case without making too many major changes to it. This driver exports some methods that are called by our Xen WMI driver as appropriate. Upon invocation of such methods, this driver would communicate with the rest of the acpi layer to invoke firmware ASL methods and returns the result to Xen WMI driver which is then communicated to the guest through QEMU and virtual firmware.

This driver is built when ACPI_WMI kernel build configuration option is selected and in our environment it is selected by default.

Xenpmd/ACPID

We have modified the Linux ACPI-WMI driver to route WMI events through ACPID. Since our xenpmd daemon already watches for ACPID events, we have modified it to look for the following ACPID events -

- WMID
- AMW0

and update the below xenstore node -

- /oem/event

Since this node is watched by QEMU when OEM features is enabled, upon receiving a xenstore notification, QEMU will inject an SCI to guest to let it know about the platform event. The guest ACPI-WMI wrapper driver will then request our virtual firmware layer for event data using the well known _WED entry point and act accordingly.

MOF Data

OEMs that choose to expose their MOF data through control methods, with GUID mapping for such control methods exposed through _WDG can expect their MOF data to be exposed in the virtual space through similar GUID mapping.

However, if OEMs choose other means to expose their private MOF data but from within their base firmware, for us to expose the same data in our virtual space, we would have to work out a standard interface for OEMs to provide us that data. Such standard interface can be created once we establish a need for it based on input from OEMs.

Known Issues

- We require a per OEM, in some cases per model SSDT. Though it is easy enough to create such SSDT, we are going to investigate mechanisms to generate them on the fly.
- Firmware level bugs can affect the results we pass back to the guest VM and we currently have workaround for at least one OEM.
- If OEMs do not use ACPI-WMI wrapper or uses it in an unconventional way, we may not be able to support OEM value-add feature for such OEMs at the moment without additional work.
- We currently do not support more than one WMI pseudo device object in our virtual space. If the underlying firmware provides more than one WMI pseudo object (rare), our implementation won't be complete in such environment.
- Special buttons, hotkeys that are not wired to generate ACPI interrupt would simply not fall under the radar of our oem-features support. This could be a problem where it is expected that our oem-features cover features outside the scope of WMI.
- Whether or not it is necessary to expose OEM features to multiple VMs at the same time need to be further discussed and assumed to be not supported for V1 (though the design would allow for it and most likely to work as is).
- Headers between QEMU and vACPI layer and between QEMU and Xen WMI driver is not shared at the moment resulting in multiple copies of same headers.

Attempt to consolidate this should be added in future.

Alternate Approaches

Before researching WMI approach and implementing it for our guests, considerable effort was devoted to researching whether or not we could pass-through the firmware pseudo WMI object as is and wire appropriate ports to simplify the implementation. As there were too many dependencies in the base firmware space, this approach turned out to be infeasible. However, in future if the firmware implementation changes in favor of such pass-through, such an implementation should be highly considered for its comparative efficiency and simplicity.

Dependencies

- Linux ACPI-WMI Wrapper driver
- ACPID
- SMBIOS-pt

Installation Requirements

- ACPID

References

http://www.microsoft.com/whdc/system/pnppwr/wmi/wmi-acpi.mspx

Appendix A: acpi-wmi.h file

```
/*****************************************************************************
 * acpi-wmi.h
 *
 * Interface to /proc/misc/xen-acpi-wmi
 *
 * Copyright (c) 2009 Kamala Narasimhan
 * Copyright (c) 2009 Citrix Systems, Inc.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License version 2
 * as published by the Free Software Foundation; or, when distributed
 * separately from the Linux kernel or incorporated into other
 * software packages, subject to the following license:
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this source file (the "Software"), to deal in the Software without
 * restriction, including without limitation the rights to use, copy, modify,
 * merge, publish, distribute, sublicense, and/or sell copies of the Software,
 * and to permit persons to whom the Software is furnished to do so, subject to
 * the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
 * FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
 * IN THE SOFTWARE.
 */


#ifndef _XEN_WMI_ACPI
#define _XEN_WMI_ACPI

/*
 * Userspace Interface
 */

#define XEN_WMI_DEVICE_NAME             "xen-acpi-wmi"
#define XEN_WMI_GUID_SIZE               16

#define XEN_WMI_SUCCESS                  0
#define XEN_WMI_UNSUPPORTED_TYPE        -1
#define XEN_WMI_BUFFER_TOO_SMALL        -11
#define XEN_WMI_NOT_ENOUGH_MEMORY       -12
#define XEN_WMI_EFAULT                  -14
```

```
50 #define XEN_WMI_INVALID_ARGUMENT        -22
51 #define XEN_WMI_ENOIOCTLCMD             -515
52
53 #define XEN_WMI_IOCTL_CALL_METHOD        100
54 #define XEN_WMI_IOCTL_QUERY_OBJECT       101
55 #define XEN_WMI_IOCTL_SET_OBJECT         102
56 #define XEN_WMI_IOCTL_GET_EVENT_DATA     103
57
58 typedef unsigned char byte;
59
60 typedef struct xen_wmi_buffer {
61     size_t       length;
62     void        *pointer;
63     size_t      *copied_length;
64 } xen_wmi_buffer_t;
65
66 typedef struct xen_wmi_obj_invocation_data {
67     byte                        guid[XEN_WMI_GUID_SIZE];
68     union {
69         struct {
70             ushort              instance;
71             uint                method_id;
72             xen_wmi_buffer_t    in_buf;
73             xen_wmi_buffer_t    out_buf;
74         } xen_wmi_method_arg;
75
76         struct {
77             ushort              instance;
78             xen_wmi_buffer_t    out_buf;
79         } xen_wmi_query_obj_arg;
80
81         struct {
82             ushort              instance;
83             xen_wmi_buffer_t    in_buf;
84         } xen_wmi_set_obj_arg;
85
86         struct {
87             ushort              event_id;
88             xen_wmi_buffer_t    out_buf;
89         } xen_wmi_event_data_arg;
90     } xen_wmi_arg;
91 } xen_wmi_obj_invocation_data_t;
92
93 #endif /* _XEN_WMI_ACPI */
94
95
```