Linux Wmi Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

ACPI WMI INTERFACE

The ACPI WMI interface is a proprietary extension of the ACPI specification made by Microsoft to allow hardware vendors to embed WMI (Windows Management Instrumentation) objects inside their ACPI firmware. Typical functions implemented over ACPI WMI are hotkey events on modern notebooks and configuration of BIOS options.

1.1 PNP0C14 ACPI device

Discovery of WMI objects is handled by defining ACPI devices with a PNP ID of `PNP0C14`. These devices will contain a set of ACPI buffers and methods used for mapping and execution of WMI methods and/or queries. If there exist multiple of such devices, then each device is required to have a unique ACPI UID.

1.2 _WDG buffer

The `_WDG` buffer is used to discover WMI objects and is required to be static. Its internal structure consists of data blocks with a size of 20 bytes, containing the following data:

Offset	Size (in bytes)	Content
0x00	16	128 bit Variant 2 object GUID.
0x10	2	2 character method ID or single byte notification ID.
0x12	1	Object instance count.
0x13	1	Object flags.

The WMI object flags control whether the method or notification ID is used:

- 0x1: Data block usage is expensive and must be explicitly enabled/disabled.
- 0x2: Data block contains WMI methods.
- 0x4: Data block contains ASCIZ string.
- 0x8: Data block describes a WMI event, use notification ID instead of method ID.

Each WMI object GUID can appear multiple times inside a system. The method/notification ID is used to construct the ACPI method names used for interacting with the WMI object.

1.3 WQxx ACPI methods

If a data block does not contain WMI methods, then its content can be retrieved by this required ACPI method. The last two characters of the ACPI method name are the method ID of the data block to query. Their single parameter is an integer describing the instance which should be queried. This parameter can be omitted if the data block contains only a single instance.

1.4 WSxx ACPI methods

Similar to the `WQxx` ACPI methods, except that it is optional and takes an additional buffer as its second argument. The instance argument also cannot be omitted.

1.5 WMxx ACPI methods

Used for executing WMI methods associated with a data block. The last two characters of the ACPI method name are the method ID of the data block containing the WMI methods. Their first parameter is a integer describing the instance which methods should be executed. The second parameter is an integer describing the WMI method ID to execute, and the third parameter is a buffer containing the WMI method parameters. If the data block is marked as containing an ASCIZ string, then this buffer should contain an ASCIZ string. The ACPI method will return the result of the executed WMI method.

1.6 WExx ACPI methods

Used for optionally enabling/disabling WMI events, the last two characters of the ACPI method are the notification ID of the data block describing the WMI event as hexadecimal value. Their first parameter is an integer with a value of 0 if the WMI event should be disabled, other values will enable the WMI event.

1.7 WCxx ACPI methods

Similar to the `WExx` ACPI methods, except that it controls data collection instead of events and thus the last two characters of the ACPI method name are the method ID of the data block to enable/disable.

1.8 _WED ACPI method

Used to retrieve additional WMI event data, its single parameter is a integer holding the notification ID of the event.

CHAPTER

TWO

DRIVER-SPECIFIC DOCUMENTATION

This section provides information about various devices supported by the Linux kernel, their protocols and driver details.

2.1 Dell DDV WMI interface driver (dell-wmi-ddv)

2.1.1 Introduction

Many Dell notebooks made after ~2020 support a WMI-based interface for retrieving various system data like battery temperature, ePPID, diagostic data and fan/thermal sensor data.

This interface is likely used by the *Dell Data Vault* software on Windows, so it was called *DDV*. Currently the `dell-wmi-ddv` driver supports version 2 and 3 of the interface, with support for new interface versions easily added.

Warning: The interface is regarded as internal by Dell, so no vendor documentation is available. All knowledge was thus obtained by trial-and-error, please keep that in mind.

2.1.2 Dell ePPID (electronic Piece Part Identification)

The Dell ePPID is used to uniquely identify components in Dell machines, including batteries. It has a form similar to *CC-PPPPPP-MMMMM-YMD-SSSS-FFF* and contains the following information:

- Country code of origin (CC).
- Part number with the first character being a filling number (PPPPPP).
- Manufacture Identification (MMMMM).
- Manufacturing Year/Month/Date (YMD) in base 36, with Y being the last digit of the year.
- Manufacture Sequence Number (SSSS).
- Optional Firmware Version/Revision (FFF).

The eppidtool python utility can be used to decode and display this information.

All information regarding the Dell ePPID was gathered using Dell support documentation and this website.

2.1.3 WMI interface description

The WMI interface description can be decoded from the embedded binary MOF (bmof) data using the bmfdec utility:

```
[WMI, Dynamic, Provider("WmiProv"), Locale("MS\\0x409"), Description("WMI␣
↪Function"), guid("{8A42EA14-4F2A-FD45-6422-0087F7A7E608}")]
class DDVWmiMethodFunction {
  [key, read] string InstanceName;
  [read] boolean Active;

  [WmiMethodId(1), Implemented, read, write, Description("Return Battery␣
↪Design Capacity.")] void BatteryDesignCapacity([in] uint32 arg2, [out]␣
↪uint32 argr);
  [WmiMethodId(2), Implemented, read, write, Description("Return Battery Full␣
↪Charge Capacity.")] void BatteryFullChargeCapacity([in] uint32 arg2, [out]␣
↪uint32 argr);
  [WmiMethodId(3), Implemented, read, write, Description("Return Battery␣
↪Manufacture Name.")] void BatteryManufactureName([in] uint32 arg2, [out]␣
↪string argr);
  [WmiMethodId(4), Implemented, read, write, Description("Return Battery␣
↪Manufacture Date.")] void BatteryManufactureDate([in] uint32 arg2, [out]␣
↪uint32 argr);
  [WmiMethodId(5), Implemented, read, write, Description("Return Battery␣
↪Serial Number.")] void BatterySerialNumber([in] uint32 arg2, [out] uint32␣
↪argr);
  [WmiMethodId(6), Implemented, read, write, Description("Return Battery␣
↪Chemistry Value.")] void BatteryChemistryValue([in] uint32 arg2, [out]␣
↪string argr);
  [WmiMethodId(7), Implemented, read, write, Description("Return Battery␣
↪Temperature.")] void BatteryTemperature([in] uint32 arg2, [out] uint32 argr);
  [WmiMethodId(8), Implemented, read, write, Description("Return Battery␣
↪Current.")] void BatteryCurrent([in] uint32 arg2, [out] uint32 argr);
  [WmiMethodId(9), Implemented, read, write, Description("Return Battery␣
↪Voltage.")] void BatteryVoltage([in] uint32 arg2, [out] uint32 argr);
  [WmiMethodId(10), Implemented, read, write, Description("Return Battery␣
↪Manufacture Access(MA code).")] void BatteryManufactureAceess([in] uint32␣
↪arg2, [out] uint32 argr);
  [WmiMethodId(11), Implemented, read, write, Description("Return Battery␣
↪Relative State-Of-Charge.")] void BatteryRelativeStateOfCharge([in] uint32␣
↪arg2, [out] uint32 argr);
  [WmiMethodId(12), Implemented, read, write, Description("Return Battery␣
↪Cycle Count")] void BatteryCycleCount([in] uint32 arg2, [out] uint32 argr);
  [WmiMethodId(13), Implemented, read, write, Description("Return Battery ePPID
↪")] void BatteryePPID([in] uint32 arg2, [out] string argr);
  [WmiMethodId(14), Implemented, read, write, Description("Return Battery Raw␣
↪Analytics Start")] void BatteryeRawAnalyticsStart([in] uint32 arg2, [out]␣
↪uint32 argr);
  [WmiMethodId(15), Implemented, read, write, Description("Return Battery Raw␣
↪Analytics")] void BatteryeRawAnalytics([in] uint32 arg2, [out] uint32␣
↪RawSize, [out, WmiSizeIs("RawSize") : ToInstance] uint8 RawData[]);
```

```
  [WmiMethodId(16), Implemented, read, write, Description("Return Battery␣
↪Design Voltage.")] void BatteryDesignVoltage([in] uint32 arg2, [out] uint32␣
↪argr);
  [WmiMethodId(17), Implemented, read, write, Description("Return Battery Raw␣
↪Analytics A Block")] void BatteryeRawAnalyticsABlock([in] uint32 arg2, [out]␣
↪uint32 RawSize, [out, WmiSizeIs("RawSize") : ToInstance] uint8 RawData[]);
  [WmiMethodId(18), Implemented, read, write, Description("Return Version.")]␣
↪void ReturnVersion([in] uint32 arg2, [out] uint32 argr);
  [WmiMethodId(32), Implemented, read, write, Description("Return Fan Sensor␣
↪Information")] void FanSensorInformation([in] uint32 arg2, [out] uint32␣
↪RawSize, [out, WmiSizeIs("RawSize") : ToInstance] uint8 RawData[]);
  [WmiMethodId(34), Implemented, read, write, Description("Return Thermal␣
↪Sensor Information")] void ThermalSensorInformation([in] uint32 arg2, [out]␣
↪uint32 RawSize, [out, WmiSizeIs("RawSize") : ToInstance] uint8 RawData[]);
};
```

Each WMI method takes an ACPI buffer containing a 32-bit index as input argument, with the first 8 bit being used to specify the battery when using battery-related WMI methods. Other WMI methods may ignore this argument or interpret it differently. The WMI method output format varies:

- if the function has only a single output, then an ACPI object of the corresponding type is returned
- if the function has multiple outputs, when an ACPI package containing the outputs in the same order is returned

The format of the output should be thoroughly checked, since many methods can return malformed data in case of an error.

The data format of many battery-related methods seems to be based on the *Smart Battery Data Specification*, so unknown battery-related methods are likely to follow this standard in some way.

WMI method GetBatteryDesignCapacity()

Returns the design capacity of the battery in mAh as an u16.

WMI method BatteryFullCharge()

Returns the full charge capacity of the battery in mAh as an u16.

WMI method BatteryManufactureName()

Returns the manufacture name of the battery as an ASCII string.

WMI method BatteryManufactureDate()

Returns the manufacture date of the battery as an u16. The date is encoded in the following manner:

- bits 0 to 4 contain the manufacture day.
- bits 5 to 8 contain the manufacture month.
- bits 9 to 15 contain the manufacture year biased by 1980.

Note: The data format needs to be verified on more machines.

WMI method BatterySerialNumber()

Returns the serial number of the battery as an u16.

WMI method BatteryChemistryValue()

Returns the chemistry of the battery as an ASCII string. Known values are:

- “Li-I” for Li-Ion

WMI method BatteryTemperature()

Returns the temperature of the battery in tenth degree kelvin as an u16.

WMI method BatteryCurrent()

Returns the current flow of the battery in mA as an s16. Negative values indicate discharging.

WMI method BatteryVoltage()

Returns the voltage flow of the battery in mV as an u16.

WMI method BatteryManufactureAccess()

Returns a manufacture-defined value as an u16.

WMI method BatteryRelativeStateOfCharge()

Returns the capacity of the battery in percent as an u16.

WMI method BatteryCycleCount()

Returns the cycle count of the battery as an u16.

WMI method BatteryePPID()

Returns the ePPID of the battery as an ASCII string.

WMI method BatteryeRawAnalyticsStart()

Performs an analysis of the battery and returns a status code:

- `0x0`: Success
- `0x1`: Interface not supported
- `0xfffffffe`: Error/Timeout

Note: The meaning of this method is still largely unknown.

WMI method BatteryeRawAnalytics()

Returns a buffer usually containing 12 blocks of analytics data. Those blocks contain:

- a block number starting with 0 (u8)
- 31 bytes of unknown data

Note: The meaning of this method is still largely unknown.

WMI method BatteryDesignVoltage()

Returns the design voltage of the battery in mV as an u16.

WMI method BatteryeRawAnalyticsABlock()

Returns a single block of analytics data, with the second byte of the index being used for selecting the block number.

Supported since WMI interface version 3!

Note: The meaning of this method is still largely unknown.

WMI method ReturnVersion()

Returns the WMI interface version as an u32.

WMI method FanSensorInformation()

Returns a buffer containing fan sensor entries, terminated with a single `0xff`. Those entries contain:

- fan type (u8)
- fan speed in RPM (little endian u16)

WMI method ThermalSensorInformation()

Returns a buffer containing thermal sensor entries, terminated with a single `0xff`. Those entries contain:

- thermal type (u8)
- current temperature (s8)
- min. temperature (s8)
- max. temperature (s8)
- unknown field (u8)

Note: TODO: Find out what the meaning of the last byte is.

2.1.4 ACPI battery matching algorithm

The algorithm used to match ACPI batteries to indices is based on information which was found inside the logging messages of the OEM software.

Basically for each new ACPI battery, the serial numbers of the batteries behind indices 1 till 3 are compared with the serial number of the ACPI battery. Since the serial number of the ACPI battery can either be encoded as a normal integer or as a hexadecimal value, both cases need to be checked. The first index with a matching serial number is then selected.

A serial number of 0 indicates that the corresponding index is not associated with an actual battery, or that the associated battery is not present.

Some machines like the Dell Inspiron 3505 only support a single battery and thus ignore the battery index. Because of this the driver depends on the ACPI battery hook mechanism to discover batteries.

Note: The ACPI battery matching algorithm currently used inside the driver is outdated and does not match the algorithm described above. The reasons for this are differences in the handling of the ToHexString() ACPI opcode between Linux and Windows, which distorts the serial number of ACPI batteries on many machines. Until this issue is resolved, the driver cannot use the above algorithm.

2.1.5 Reverse-Engineering the DDV WMI interface

1. Find a supported Dell notebook, usually made after ~2020.
2. Dump the ACPI tables and search for the WMI device (usually called “ADDV”).
3. Decode the corresponding bmof data and look at the ASL code.
4. Try to deduce the meaning of a certain WMI method by comparing the control flow with other ACPI methods (_BIX or _BIF for battery related methods for example).
5. Use the built-in UEFI diagostics to view sensor types/values for fan/thermal related methods (sometimes overwriting static ACPI data fields can be used to test different sensor type values, since on some machines this data is not reinitialized upon a warm reset).

Alternatively:

1. Load the `dell-wmi-ddv` driver, use the `force` module param if necessary.
2. Use the debugfs interface to access the raw fan/thermal sensor buffer data.
3. Compare the data with the built-in UEFI diagnostics.

In case the DDV WMI interface version available on your Dell notebook is not supported or you are seeing unknown fan/thermal sensors, please submit a bugreport on bugzilla so they can be added to the `dell-wmi-ddv` driver.

See Documentation/admin-guide/reporting-issues.rst for further information.

2.2 WMI embedded Binary MOF driver

2.2.1 Introduction

Many machines embed WMI Binary MOF (Managed Object Format) metadata used to describe the details of their ACPI WMI interfaces. The data can be decoded with tools like bmfdec to obtain a human readable WMI interface description, which is useful for developing new WMI drivers.

The Binary MOF data can be retrieved from the `bmof` sysfs attribute of the associated WMI device. Please note that multiple WMI devices containing Binary MOF data can exist on a given system.

2.2.2 WMI interface

The Binary MOF WMI device is identified by the WMI GUID `05901221-D566-11D1-B2F0-00A0C9062910`. The Binary MOF can be obtained by doing a WMI data block query. The result is then returned as an ACPI buffer with a variable size.