Windows Platform Binary Table (WPBT)

July 9, 2015

Abstract

A platform can be provisioned with the Windows operating system by entities including an enterprise, a system reseller, or an end-user customer. If the platform has drivers, system services, or executable files that are integral to the platform, the platform binaries must either be distributed as part of the Windows image or they must be injected into the Windows image by each of the possible provisioning entities. A rich set of tools exist to aid Windows provisioning, ranging from driver injection and offline registry management to sysprep imaging tools. However, there is a small set of software where the tools are not enough. The software is absolutely critical for the execution of Windows but for one reason or another, the vendor is unable to distribute the software to every provisioning entity. This paper describes a mechanism for a platform, via the boot firmware, to publish a binary to Windows for execution. The mechanism leverages a boot firmware component to publish a binary in physical memory described to Windows using a fixed ACPI table.

The information provided here was originally published in conjunction with the availability of Windows 8. The guidance and requirements to use WPBT functionality has been updated for the Windows 10 timeframe.

The current version of this paper is maintained on the web at:
 <http://go.microsoft.com/fwlink/p/?LinkId=234840>

**Disclaimer**: This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet website references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2015 Microsoft. All rights reserved.



Document History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date | Change |  |  |  |
| November 29, 2011 | First publication |
| July 8, 2015, 2015 | Revision to include security guidance and requirements  |

Contents

[Introduction 3](#_Toc424199068)

[Windows Platform Binary Table (WPBT) format 4](#_Toc424199069)

[Binary handoff layout 5](#_Toc424199070)

[Binary content type 5](#_Toc424199071)

[Execution status method 6](#_Toc424199072)

[Code-signing policy 6](#_Toc424199073)

[Detailed description of handoff mechanism 7](#_Toc424199074)

[How to allocate memory 7](#_Toc424199075)

[How to update the RSDT/XSDT 8](#_Toc424199076)

[Security Considerations and Requirements 10](#_Toc424199077)

[Continued servicing for WPBT publisher and client applications 10](#_Toc424199078)

[Resources 12](#_Toc424199079)

# Introduction

This paper describes the format of a Windows Platform Binary Table (WPBT). The WPBT is a fixed Advanced Configuration and Power Interface (ACPI) table that enables boot firmware to provide Windows with a platform binary that the operating system can execute. The binary handoff medium is physical memory, allowing the boot firmware to provide the platform binary without modifying the Windows image on disk. In the initial version, the WPBT simply contains a physical address pointer to a flat, Portable Executable (PE) image that has been copied to physical memory. The WPBT is extensible, allowing the layout of published platform binaries to be more complex in future versions and allowing the support of more than one binary type.

It is expected that the binary pointed to by the WPBT is part of the boot firmware ROM image. The binary can be shadowed to physical memory as part of the initial bootstrap of the boot firmware, or it can be loaded into physical memory by extensible boot firmware code prior to executing any operating system code. A boot firmware component would create the WPBT based on the location of the platform binary. During operating system initialization, Windows will read the WPBT to obtain the physical memory location of the platform binary. In the first version, the binary is required to be a native, user-mode application that is executed by the Windows Session Manager during operating system initialization. Windows will write the flat image to disk, and the Session Manager will launch the process. Windows may reclaim the physical memory described in the WPBT.

If Windows observes a WPBT during operating system initialization, it will attempt to use an ACPI control method to communicate binary execution status back to the platform.

## Windows Platform Binary Table (WPBT) format

Table 1 details the layout of the WPBT. The WPBT is published by a boot firmware component, describing the location and usage of platform binaries in memory.

Table 1. Windows Platform Binary Table

|  |  |  |  |
| --- | --- | --- | --- |
| Field | Byte length | Byte offset | Description |
| ***ACPI Standard Header*** |
| Signature | 4 | 0 | A signature identifying the WPBT. This value must be “WPBT”, written from low-order bytes to high-order bytes (that is, byte 0 must be ‘W’ and byte 3 must be ‘T’). |
| Length | 4 | 4 | Length, in bytes, of the WPBT. For Revision 1, the length must be at least 52 bytes. |
| Revision | 1 | 8 | 1 |
| Checksum | 1 | 9 | Entire table, which must sum to zero. |
| OEMID | 6 | 10 | Original equipment manufacturer (OEM) identifier (ID). |
| OEM Table ID | 8 | 16 | Manufacturer model ID. |
| OEM Revision | 4 | 24 | OEM revision for supplied OEM table ID. |
| Creator ID | 4 | 28 | Vendor ID of the utility that created the table. |
| Creator Revision | 4 | 32 | Revision of the utility that created the table. |
| ***WPBT-Specific Fields*** |
| Handoff Memory Size | 4 | 36 | The size of the handoff memory buffer containing a platform binary. |
| Handoff Memory Location | 8 | 40 | The 64-bit physical address of a memory buffer containing a platform binary.  |
| Content Layout | 1 | 48 | Description of the layout of the handoff memory buffer. Possible values include:1 – Image location points to a single Portable Executable (PE) image at offset 0 of the specified memory location. The image is a flat image where sections have not been expanded and relocations have not been applied.  |
| Content Type | 1 | 49 | Description of the content of the binary image and the usage model of the platform binary. Possible values include:1 – The platform binary is a native user-mode application that should be executed by the Windows Session Manager during operating system initialization.  |
| ***Content Type–Specific Fields*** |
| ***Type 1–Specific (native user-mode application)*** |
| Command-line Arguments Length | 2 | 50 | Supplies the length, in bytes, of an input string for the native user-mode application. Must be a multiple of two bytes. If no input parameters exist, this field should be set to zero. |
| Command-line Arguments | Variable | 52 | An optional Unicode string, which is passed as an input parameter string to the native user-mode application. |

The WPBT provides a pointer to a physical memory location that contains a Windows binary for execution. The initial version of the WPBT is limited, supporting Autochk.exe-type applications executed by the Windows Session Manager during operating system initialization. Additionally, only one binary is supported, which leads to a very simple handoff format where the memory buffer simply contains the executable file. The table definition allows for new handoff layout formats and binary types to be added if necessary.

### Binary handoff layout

The binary handoff layout is described by the Content Layout field of the WPBT. Currently only one layout is supported. The binary handoff layout is required to contain a single, flat PE binary. The handoff memory buffer contains only the PE binary, with the binary starting at offset zero of the memory buffer. The binary is a flat image, identical to the binary as it would exist after it had been compiled and could be found on a file system.

### Binary content type

The binary content type is described by the Content Type field of the WPBT. The content type describes the type of Windows binary contained in the handoff memory buffer. The initial version of the WPBT supports only native, user-mode applications that are executed by the Windows Session Manager during operating system initialization. A native application refers to an application that does not have a dependency on the Windows API (Win32). Ntdll.dll is the only DLL dependency of a native application. A native application has a PE subsystem type of 1 (IMAGE\_SUBSYSTEM\_NATIVE).

The applications supported by the BootExecute and SetupExecute[[1]](#footnote-2) registry directives are examples of native applications. Any application supported by BootExecute or SetupExecute would be supported by the WPBT. After executing native applications from the BootExecute and SetupExecute directives, the Session Manager will launch a native application from the WPBT. If the WPBT contains a command-line input parameter string, the Session Manager will pass the string to the executable file when starting its process. Session Manager will not execute any platform binaries if the system is booted into safe mode or the Windows Recovery Environment (WinRE).

The Session Manager writes platform binaries to disk and launches them from files on disk. The file is written every boot that the Session Manager attempts to execute the binary. The on-disk file location is \Windows\System32\Wpbbin.exe on the operating system volume.

64-bit Windows will support a 32-bit application, allowing a platform to support a 32-bit native application for both 32-bit and 64-bit Windows. A native UEFI system, only supporting 64-bit operating systems, should provide a 64-bit native application. As the industry moves to 64-bit and UEFI, Windows can add enforcement that the native application be 64-bit.

The boot process will be blocked when executing a native application from the WPBT. Care should be taken to avoid negatively affecting boot performance. If the binary is not required every boot, a WPBT should not be published every boot.

## Execution status method

Firmware that provides a WPBT may also provide an ACPI control method named \_PBS that is placed under the root of the ACPI namespace. If Windows detects the presence of a WPBT during operating system initialization, it will attempt to invoke this method to communicate platform binary execution status back to the platform.

This method takes the following arguments:

* **Argument 1**: A 32-bit unsigned integer describing the version of the fourth argument used by this version of Windows, as described later in this document.
* **Argument 2**: A 32-bit unsigned integer describing the mechanism by which a platform binary was provided by the operating system. In the initial version, this value will always be 1, specifying that the binary was provided to Windows via the WPBT.
* **Argument 3**: A 32-bit unsigned integer describing the execution status of the binary. In the initial version, this argument may have one of the following values:

0: The binary was read, written, and launched successfully.

1: Windows could not successfully execute the platform binary provided by the WPBT, either because the WPBT was itself invalid or because of an internal operating system error.

2: Windows detected the presence of a WPBT, but did not try to execute the binary because of operating system policy. Currently, this status is used when the current boot environment is safe mode or WinRE.

3: Windows detected the presence of a WPBT and attempted to execute the associated binary, but the binary did not execute because a code integrity check failed. See “Code-signing policy,” later in this document.

* **Argument 4**: A variable-length buffer that Windows uses to provide information about the binary it actually ran. The contents of this buffer depend on the value of argument 1. In the initial version, the possible values of argument 1 and their corresponding argument 4 values are as follows:

Argument 1 = 1: Argument 4 is the 64-bit physical address from which Windows copied and executed a platform binary, or 0 if the binary failed to execute or was not executed.

Windows also logs an event to its system event log that describes the platform binary execution status.

## Code-signing policy

All binaries published to Windows using the WPBT mechanism outlined in this paper must be embedded signed and timestamped. These images should be linked with the **/INTEGRITYCHECK** option and signed using the [SignTool](http://msdn.microsoft.com/library/aa387764%28VS.85%29.aspx) command-line tool with the **/nph** switch to suppress page hashes.

## Detailed description of handoff mechanism

This section describes the handoff mechanism implied by the WPBT. Included in this section is a description of the handoff mechanism on both UEFI and older BIOS boot firmware. For both boot firmware types, this section describes how the WPBT should be published and how to describe physical memory allocations used to publish the WPBT.

Like all other ACPI tables, the location of the WPBT is provided by the Root System Description Table or Extended System Description Table (RSDT/XSDT). The RSDT/XSDT is a special ACPI table, which contains an array describing the physical address of all other ACPI tables. During operating system initialization, Windows will enumerate the RSDT/XSDT to obtain the location of ACPI tables. As needed, Windows may copy the contents of an ACPI table into operating system memory. Based on these fundamental properties of ACPI tables, the initialization and publishing of the WPBT can require up to three physical allocations:

* **RSDT/XSDT**: Array containing the physical address of all platform ACPI tables.
* **WPBT**: The Windows Platform Binary Table, describing the physical memory handoff location of a Windows platform binary.
* **Handoff Memory Buffer**: A physical memory region containing a flat, PE Windows platform binary.

Firmware design can simplify the requirements of the code that initializes and publishes the WPBT. If the WPBT is published by the core firmware, then the firmware can pre-allocate space for the WPBT table and the RSDT/XSDT, eliminating the need to explicitly allocate memory for their purposes. However, if an extensible firmware component publishes the WPBT, the code will need to allocate the WPBT and it may additionally need to allocate a new RSDT/XSDT to store the physical address pointer of the WPBT.

The next few sections describe how firmware can allocate memory and publish a new RSDT/XSDT, assuming firmware design necessitates these actions.

### How to allocate memory

The ACPI specification describes system memory-map-reporting interfaces that boot firmware must implement in order to describe the physical address map to an operating system. ACPI specifies that an older BIOS implement the **int 15h**, **E820h** interface to report the physical address map to an operating system. And ACPI specifies that UEFI firmware implements the UEFI **GetMemoryMap()** boot service to report the physical address map to an operating system. Both interfaces are used by the Windows boot loader to obtain an accurate description of physical memory prior to initializing the operating system. They each describe the physical address map as a series of memory ranges, each with a memory type indicating the usage of the memory. Some of the examples of memory type include operating system available memory, unusable memory, and platform reserved memory. Windows uses the description to determine the ranges of physical memory that are available to the operating system and the ranges of physical memory that are in use by the firmware.

Allocations for the RSDT/XSDT, the WPBT, and the WPBT’s handoff memory buffer must be described in the respective BIOS or UEFI physical address maps. Each allocation must be described as memory that can be reclaimed by the operating system after its contents are read. The process for allocating memory differs on UEFI platforms and older BIOS platforms. The next two subsections outline the process in further detail.

#### UEFI

The core UEFI boot firmware provides memory allocation services to firmware components. A firmware component allocating memory for the WPBT or the binary handoff memory buffer must use the firmware service, **AllocatePages()**with the allocation type **AllocateAnyPages**. For all three allocations — the RSDT/XSDT, the WPBT, and the handoff memory buffer allocation — the firmware must use the **EfiACPIReclaimMemory** memory type. Windows may reclaim the physical memory for these allocations after copying the contents into operating system memory.

#### BIOS

Unlike UEFI, BIOS does not provide a service for memory allocations. Allocations made by a firmware component require the firmware to hook **int 15h**, **E820h** to ensure the memory map returned to the operating system describes all platform allocations made by the firmware. The hooking process is not described in detail in this paper. It is assumed that a firmware developer writing this code is familiar with the process of hooking a BIOS interrupt service. The process for hooking the BIOS service is not trivial and should be taken into consideration when designing a BIOS solution that publishes a WPBT. In general, it is advantageous to minimize allocations made by the publishing component. If possible, it is better to pre-reserve additional memory for the RSDT/XSDT, WPBT, and handoff memory buffer.

For the RSDT/XSDT, WPBT, and handoff memory buffer allocations, the firmware must use the **AddressRangeACPI**memory type. Windows may reclaim the physical memory for these allocations after copying the contents into operating system memory.

### How to update the RSDT/XSDT

If the WPBT is published by the core firmware, then the firmware should pre-allocate space in the RSDT/XSDT for the WPBT physical address pointer and pre-reserve memory in an ACPI memory region for the WPBT table. When the WPBT is initialized, the RSDT/XSDT is initialized to point to the WPBT.

Platform design does not always allow for the RSDT/XSDT to be pre-allocated to include the WPBT. In that case, more complex steps must be followed. An extensible firmware component must allocate physical memory for the WPBT and it must additionally publish the table in a newly allocated RSDT/XSDT that gets exposed to Windows. The process to accomplish the publishing task is outlined in the following subsections.

#### UEFI

UEFI firmware explicitly provides the RSDT/XSDT to an operating system through the EFI system table. The system table contains an array of configuration tables, one of which is the RSDT/XSDT and is identified by the table GUID EFI\_ACPI\_TABLE\_GUID. On a 64-bit UEFI system, the RSDT/XSDT is always an XSDT. In this section, XSDT will be exclusively used for correctness.

During its initialization, an operating system walks the configuration table list to obtain the root of the ACPI tables. Firmware that complies with UEFI 2.1 or later supplies the boot service, **InstallConfigurationTable()**, which enables an extensible firmware component to modify the XSDT that is exposed to an operating system.

To modify the XSDT, an extensible firmware component must allocate a new XSDT buffer that is large enough for the original table contents as well as an additional entry for the WPBT. A firmware component must obtain the original XSDT the same way that an operating system would obtain the root ACPI table. The firmware must walk the list of configuration tables in the EFI system table, finding the XSDT by locating the table with the GUID EFI\_ACPI\_TABLE\_GUID. The firmware must allocate a new buffer that is large enough for at least one new physical address pointer. An XSDT contains an array of 64-bit physical addresses of each ACPI table. A firmware component that allocates a new XSDT would need to allocate a buffer at least 8 bytes larger than the current size (which can be obtained by the Length field in the original table’s header). The firmware must copy the original table to the new table’s buffer, update the new table’s size, add the pointer to the WPBT at the end of the table, and update the XSDT checksum.

After a new XSDT is initialized, the firmware would call **InstallConfigurationTable()** with EFI\_ACPI\_TABLE\_GUID and the new table address as parameters. After the table is updated using the **InstallConfigurationTable()** service, Windows will obtain the updated table when it performs its search for the table.

#### BIOS

The ACPI specification defines a Root System Description Pointer (RSDP), which points to the RSDT/XSDT. The RSDP is defined to be located in relatively small, traversable regions, in the first megabyte (MB) of physical memory. The search criterion for the RSDP limits the amount of time it takes to search and find the RSDT/XSDT, which is allocated towards the top of physical address space. One of the regions where the RSDP can exist is the shadowed BIOS ROM in RAM. To prevent modification of BIOS settings and code, several BIOSs program the shadowed memory as read-only. This would prevent a RSDT/XSDT publishing solution that requires a firmware component to modify the RSDP to point to a new RSDT/XSDT. Instead of publishing a new RSDT/XSDT or RSDP for the WPBT, Windows will support searching and finding the WPBT and appending the table to the ACPI namespace. This mechanism is needed if the core firmware does not pre-allocate space for the WPBT in the RSDT/XSDT and the publishing firmware component needs to extend the ACPI namespace. The rest of this section describes this mechanism in detail.

An extensible boot firmware component publishing the WPBT should allocate memory for the WPBT in low memory (below 0xA0000). During operating system bootstrap, the Windows boot loader will search for the WPBT in memory regions allocated by the firmware below 0xA0000. The Windows boot loader will scan low memory looking for a table with the WPBT’s table signature and a valid table length and checksum. The Windows boot loader will first search for the WPBT table signature. If the boot loader finds a match, the boot loader will ensure that the Length field is valid for revision 1 (at least 52 bytes) and finally check that the checksum sums to zero. If a table is found, Windows will treat the WPBT as if it were part of the RSDT/XSDT.

The low memory region is a shared resource across all boot components. Any allocations made in this region should be made as high as possible (as close to 0xA0000 as possible) and the allocation should be reflected in the **int 15h**, **E820h** physical memory map. Only the allocation for the WPBT should be in low memory. The allocation for the handoff memory buffer containing the platform binary should be in high memory, to prevent the exhaustion of memory allocated in low memory. Furthermore, the WPBT low memory requirement is a requirement only if the firmware requires republishing the table. If the firmware pre-allocates space for the table in the RSDT/XSDT, eliminating the republishing requirement, then the WPBT table should not use low memory resources.

## Security Considerations and Requirements

The primary purpose of WPBT is to allow critical software to persist even when the operating system has changed or been reinstalled in a “clean” configuration. One use case for WPBT is to enable anti-theft software which is required to persist in case a device has been stolen, formatted, and reinstalled. In this scenario WPBT functionality provides the capability for the anti-theft software to reinstall itself into the operating system and continue to work as intended. This functionality is powerful and provides the capability for independent software vendors (ISV) and original equipment manufacturers (OEM) to have their solutions stick to the device indefinitely. Because this feature provides the ability to persistently execute system software in the context of Windows, it becomes critical that WPBT-based solutions are as secure as possible and do not expose Windows users to exploitable conditions. In particular, WPBT solutions must not include malware (i.e., malicious software or unwanted software installed without adequate user consent).

To ensure the security profile of Windows users, Microsoft strongly recommends that WPBT only be used for critical functionality where persistence is a core requirement. Microsoft recommends that the following security best practices, processes, and engineering guidelines be followed to minimize exploitable conditions for Windows users. Microsoft recommends following security development lifecycle [(SDL)](https://www.microsoft.com/en-us/sdl/default.aspx) practices to help minimize security risks and exposure. Please refer to the [SDL site](https://www.microsoft.com/en-us/sdl/default.aspx) for more information.

The security guidance provided here is not intended as a cumulative or exhaustive list of security considerations but rather as a starting point for security best practices. Microsoft encourages OEMs and ISVs to go above and beyond these recommendations to provide as robust a security profile for Windows users as possible.

Continued servicing for WPBT publisher and client applications
The ability to update the WPBT publisher (firmware) as well as the OS side applications is critical and is the sole responsibility of the OEM or solution provider. OEMs and solution providers that leverage WPBT are expected to remediate security issues in a timely, holistic, and cumulative manner. It is critical that this remediation include updates to in-field devices and solutions. Update solutions must be secure end-to-end. This includes securing the update payload (signing), securing the update pipeline (encryption), and protections against rollback to insecure versions. For more information on firmware update mechanisms available from Microsoft please see the [UEFI Firmware Update Platform document](http://www.microsoft.com/en-us/download/details.aspx?id=38405) for more information. Some examples are as follows:

#### WPBT publisher

* OEMs that include a firmware WPBT publisher should have agreements in place with solution providers to service this component post release. This may require that OEMs ship a firmware update to remediate vulnerabilities in the WPBT publisher.
* The authenticated device owner should have the ability to disable or remove this functionality if desired. Note that device owner in this case could mean that it’s not the user that is using the device. For example in a corporate environment the owner maybe the IT admin but not the end user using the device.
* Ensure that update channels are only available to the OEM or solution provider that is responsible for that software. Ensure that an attacker cannot tamper with, block, or manipulate the update channel or the payloads that is distributes. Ensuring the integrity of the update channel can be accomplished in many ways for an example of how to leverage code signing or certificate pinning as potential solutions please see the [Introductions to code signing MSDN page](https://msdn.microsoft.com/en-us/library/ms537361%28v%3Dvs.85%29.aspx).

Client Side Application and Firmware Considerations
All software in a WPBT-based solution, including firmware, OS applications, OS services, OS drivers, and services, must be secure to maintain the integrity and intended functionality of the application but also to ensure that vulnerabilities are not introduced that could affect Windows users.

* Solutions must be security reviewed and not contain vulnerabilities that allow an attacker to elevate privileges, leak data, or weakens or bypasses security features provided by the OS.
* All components provided in the solution should require strong digital signing (code signing) and integrity checks. Please visit the [Microsoft code-signing best practices guide](https://msdn.microsoft.com/en-us/library/windows/hardware/dn653556%28v%3Dvs.85%29.aspx) for more information.
* Communication from the client side applications to backend severs should be encrypted. Please visit [What is TLS/SSL?](https://technet.microsoft.com/en-us/library/cc784450%28v%3Dws.10%29.aspx?f=255&MSPPError=-2147217396) on TechNet for more information.
* The solution must not tamper with or disable Windows Update.
* The solution must not weaken the integrity of Windows security features.
* Operations that only apply to a subset of devices, should include this targeting information inside a signed payload. For example, per-device-unique identifiers would be placed inside a signed file such that it cannot be used to modify an unintended system.
* Operations that require freshness should include a random nonce inside the signed payload. For example, a payload that locks a device would use a nonce to ensure that the payload could not be reused to lock the device at a future date when it has not been requested.
* Data consumed by the firmware from the OS is considered untrusted. The data must either be authenticated (by way of a digital signature) or input validated if authentication is not required (i.e. check for buffer overflows or invalid requests)
* All client side software must be updateable and is the sole responsibility of the OEM or solution provider to provide updates when security issues are identified in their product.
* Scrutiny should be used when adding certificates to the windows certificate store. <Research key management practices> <add verbiage on cert revocation>

Microsoft believes a defense-in-depth approach is a better way to resolve security issues in software, because issues that cannot be leveraged by themselves to successfully conduct an attack can be chained together to conduct a successful attack. In order to reduce the exposure for Windows users, Microsoft’s defense-in-depth philosophy seeks to address multiple vulnerabilities in order to break exploit chains and block successful attacks. Accordingly, Microsoft will provide reasonable defense-in-depth assistance to help partners identify and remediate their security issues that could expose Windows users to an attack resulting from partner-provided solutions.

Removal of Malware
If partners intentionally or unintentionally introduce malware or unwanted software though the WPBT, Microsoft may remove such software through the use of antimalware software. Software that is determined to be malicious may be subject to immediate removal without notice.

# Resources

ACPI Specification

[www.acpi.info](http://www.acpi.info)

UEFI Specification

[www.uefi.org](http://www.uefi.org)

Kernel-Mode Signing Policy

<http://msdn.microsoft.com/library/ff548231.aspx>

1. HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\BootExecute, HKLM\SYSTEM\CurrentControlSet\Conrol\SessionManager\SetupExecute [↑](#footnote-ref-2)