PRE*f*ast Annotations

October 2, 2007

Abstract

Annotations in driver source code can provide information that makes it possible for PRE*f*ast to analyze code more accurately, with significantly fewer false positives and false negatives. This paper provides an introduction to PRE*f*ast annotations, with details about general-purpose and driver-specific annotations and best practices for using annotations in your code.

This information applies for the following operating systems:  
 Windows Server® 2008  
 Windows Vista®  
 Microsoft® Windows Server 2003  
 Microsoft Windows® XP  
 Microsoft Windows 2000 (KMDF only)

**Important**: PREfast driver-specific annotations require WDK build version 6001 or later.

The current version of this paper is maintained on the Web at:  
 <http://www.microsoft.com/whdc/DevTools/tools/annotations.mspx>

For comprehensive information about writing and testing WDF drivers, see *Developing Drivers with the Windows Driver Foundation*, by Penny Orwick and Guy Smith, available at <http://www.microsoft.com/MSPress/books/10512.aspx>.

Contents

[Introduction to PRE*f*ast Annotations 4](#_Toc165195380)

[How Annotations Improve PRE*f*ast Results 4](#_Toc165195381)

[Where to Place Annotations in Code 5](#_Toc165195382)

[Annotations on Functions and Function Parameters 5](#_Toc165195383)

[Annotations on typedef Declarations 7](#_Toc165195384)

[Annotations on Function Typedef Declarations 8](#_Toc165195385)

[Tips for Placing Annotations in Source Code 10](#_Toc165195386)

[General-Purpose Annotations 10](#_Toc165195387)

[Input and Output Parameter Annotations 11](#_Toc165195388)

[The \_\_in and \_\_out Contract 12](#_Toc165195389)

[\_\_in, \_\_out, and \_\_inout versus IN, OUT, and IN OUT 13](#_Toc165195390)

[Annotation Modifiers 13](#_Toc165195391)

[The \_opt Modifier 14](#_Toc165195392)

[The \_deref Modifier 14](#_Toc165195393)

[Buffer-Size Annotations 15](#_Toc165195394)

[Fixed-Size Buffer Annotations 15](#_Toc165195395)

[Summary of Annotations for Buffers 17](#_Toc165195396)

[Tips for Applying Annotations to Buffers 19](#_Toc165195397)

[Buffer Annotation Examples 19](#_Toc165195398)

[String Annotations 19](#_Toc165195399)

[\_\_nullterminated 20](#_Toc165195400)

[\_\_nullnullterminated 20](#_Toc165195401)

[\_\_possibly\_notnullterminated 20](#_Toc165195402)

[Reserved Parameters 21](#_Toc165195403)

[Function Return Values 21](#_Toc165195404)

[Driver Annotations 22](#_Toc165195405)

[Basic Driver Annotations and Conventions 25](#_Toc165195406)

[Conditional Annotations 27](#_Toc165195407)

[Examples of Nested Conditional Annotations 28](#_Toc165195408)

[Grammar for Conditional Expressions 29](#_Toc165195409)

[Special Functions in Conditions 29](#_Toc165195410)

[Function Result Annotations 30](#_Toc165195411)

[Type-Matching Annotations 31](#_Toc165195412)

[Pointer Annotations 33](#_Toc165195413)

[Constant and Non-Constant Parameter Annotations 34](#_Toc165195414)

[Format String Annotations 34](#_Toc165195415)

[Diagnostic Annotations 34](#_Toc165195416)

[Annotations for Preferred Functions 34](#_Toc165195417)

[Annotations for Error Messages 35](#_Toc165195418)

[Annotations for Functions in \_\_try Statements 35](#_Toc165195419)

[Memory Annotations 35](#_Toc165195420)

[Annotations for Allocating and Freeing Memory 36](#_Toc165195421)

[Annotations for Aliasing Memory 36](#_Toc165195422)

[Memory Annotation Examples 36](#_Toc165195423)

[Nonmemory Resource Annotations 37](#_Toc165195424)

[Annotations for Acquisition and Release of Nonmemory Resources 38](#_Toc165195425)

[Annotations for Global Nonmemory Resources 38](#_Toc165195426)

[Annotations for the Critical Region and Cancel Spin Lock 40](#_Toc165195427)

[Annotations for Holding and Not Holding Nonmemory Resources 41](#_Toc165195428)

[Composite Annotations for Resources 42](#_Toc165195429)

[Function Type Class Annotations 43](#_Toc165195430)

[Annotations for Identifying the Function Type Class of a Function 43](#_Toc165195431)

[Annotations for Checking a Function Type Class in a Conditional Expression 44](#_Toc165195432)

[Floating-Point Annotations 44](#_Toc165195433)

[IRQL Annotations 45](#_Toc165195434)

[Annotations for Specifying Maximum and Minimum IRQL 47](#_Toc165195435)

[Annotations for Specifying Explicit IRQL 47](#_Toc165195436)

[Annotations for Raising or Lowering IRQL 47](#_Toc165195437)

[Annotations for Saving and Restoring IRQL 47](#_Toc165195438)

[Annotations for Maintaining the Same IRQL 48](#_Toc165195439)

[Annotations for Saving and Restoring IRQL for I/O Cancellation Routines 48](#_Toc165195440)

[IRQL Annotation Examples 48](#_Toc165195441)

[Tips for Applying IRQL Annotations 50](#_Toc165195442)

[DO\_DEVICE\_INITIALIZING Annotation 50](#_Toc165195443)

[Annotations for Interlocked Operands 50](#_Toc165195444)

[Examples of Annotated System Functions 51](#_Toc165195445)

[How to Write and Debug Annotations 53](#_Toc165195446)

[Examples of Annotation Test Cases 53](#_Toc165195447)

[Tips for Writing Annotation Test Cases 54](#_Toc165195448)

[PRE*f*ast Annotation Best Practices 54](#_Toc165195449)

[Example: Osrusbfx2.h with Annotations 56](#_Toc165195450)

[Resources 59](#_Toc165195451)

Disclaimer

This is a preliminary document and may be changed substantially prior to final commercial release of the software described herein.

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2007 Microsoft Corporation. All rights reserved.

Microsoft, Windows, Windows Server, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

Change History

October 2, 2007 — Typo correction on **\_\_drv\_strictTypeMatch(***mode***)** definition on page 32.

# Introduction to PRE*f*ast Annotations

PREfast for Drivers is a compile-time static verification tool that detects basic coding errors in C and C++ programs and specialized errors in driver code. PREfast for Drivers is available as a stand-alone tool in the Microsoft® Windows® Driver Kit (WDK). PREfast can be extremely valuable as a driver development tool because it can find errors that are difficult to test and debug and it can identify assumptions that might not always be valid.

Annotations provide PREfast with information about global state and work that is performed outside the function being analyzed, plus specific information about the roles and possible values of function parameters. This information makes it possible for PREfast to analyze code more accurately, with significantly fewer false positives and false negatives.

With more specific information about the intended use of an annotated function, PREfast can better determine whether a particular bug exists. For example, if an annotation is added to indicate that a parameter represents a buffer of a particular size, PREfast can check for usage that would cause a buffer overrun.

Annotations do not interfere with normal compilation on any compiler because the annotation system for PRE*f*ast uses macros. When PRE*f*ast runs, these macros expand into meaningful definitions. When the code is compiled normally, these macros expand to nothing, yielding the original unmodified program. Annotations are visible only to static analysis tools such as PREfast and to human readers, who often find them highly informative.

This paper describes PRE*f*ast support for both the general-purpose annotations defined in %wdk%\inc\api\Specstrings.h, which can be applied to both drivers and general kernel-mode and user-mode code, and the driver-specific annotations defined in %wdk%\inc\ddk\Driverspecs.h, which are specifically designed for use in kernel-mode drivers.

For an overview of PREfast with details about how to run PREfast and how to analyze PREfast results, see PREfast Step-by-Step in “Resources” at the end of this paper.

## How Annotations Improve PRE*f*ast Results

Annotations represent the intended design of a function more precisely by extending the information supplied by function prototypes and by describing the contract between a function and its caller. The task of applying annotations can help the developer to refine a function’s design.

#### Annotations Extend Function Prototypes

Prototypes prevent many errors by establishing the type and number of function parameters, so that an incorrect call can be detected at compile time. However, prototypes do not provide enough information about the intended use of a function for a tool such as PRE*f*ast to identify or eliminate possible errors.

For example, C passes parameters by value, so the parameters themselves are always input parameters to a function. However, C can pass a pointer by value, so it is not possible to tell just by looking at a function prototype whether a passed value is intended as input to the function, output from the function, or both. PREfast cannot determine whether the parameter should be initialized before the call, and it can only flag an uninitialized parameter as a potential problem. Annotations help to clarify the intended purpose of function parameters.

#### Annotations Describe the Contract between a Function and its Caller

Annotations are like the clauses in a contract. As in any contract, both sides have obligations and expected results. For this kind of contract:

* The calling function (that is, the caller) meets its obligations by correctly providing the required inputs.
* The called function (that is, the callee) meets its obligations by correctly returning the expected results.

When checking the contract of a function call, PREfast checks that the caller has met its obligations and, for subsequent analysis, assumes that the callee has done its part correctly.

When checking the contract of a function body, PREfast does the opposite: it assumes that the caller has met its obligations and checks that the callee has done its part correctly. To the extent that annotations accurately represent the obligations of the caller and the callee, PREfast can check many interfunction relationships that would initially seem to be beyond the capabilities of a tool that analyzes one function at a time.

#### Annotations Help to Refine a Function’s Design

The task of applying annotations to a well-designed function is a straightforward activity. In contrast, if a function’s design is flawed or incomplete, applying annotations can help uncover significant issues for resolution early in development, as in the following examples:

* The function does not provide enough information to prevent buffer overruns, which shows a potential bug that should be fixed, even before PREfast is run.
* The annotation raises a design issue that should be resolved.

A common example of this is whether a parameter is truly optional (that is, whether the parameter can be NULL).

* The annotations for a function are hard to express, which can indicate that the function is poorly designed.

Even if it is not possible to change the function in the short term, attempting to annotate the function identifies issues to fix in a future release.

## Where to Place Annotations in Code

Annotations can be applied to functions as a whole, to individual function parameters, and to **typedef** declarations, including declarations of function types.

### Annotations on Functions and Function Parameters

Generally, annotations that apply to an entire function should be placed immediately before the beginning of the function definition. Annotations that apply to a function parameter can be placed either inline with the parameter or enclosed in a **\_\_drv\_arg** annotation before the beginning of the function.

The example in Listing 1 shows the placement of various general-purpose and driver annotations. These annotations will be explained in more detail later in this paper. This example is intended to show where annotations can appear rather than what they do.

**Note** In this example and many of the examples in the rest of this paper, annotations under discussion in source code examples are formatted in bold type to distinguish them from the surrounding code.

Listing 1 Placement of PRE*f*ast annotations on a function

\_\_checkReturn

\_\_drv\_allocatesMem(Pool)

\_\_drv\_when(PoolType&0x1f==2 || PoolType&0x1f==6,

\_\_drv\_reportError("Must succeed pool allocations are"

"forbidden. Allocation failures cause a system crash"))

\_\_bcount(NumberOfBytes)

PVOID

ExAllocatePoolWithTag(

\_\_in \_\_drv\_strictTypeMatch(\_\_drv\_typeExpr) POOL\_TYPE PoolType,

\_\_in SIZE\_T NumberOfBytes,

\_\_in ULONG Tag

);

In this example:

* The **\_\_checkReturn**, **\_\_drv\_allocatesMem**, \_\_**drv\_when**, and \_\_**drv\_reportError** annotations apply to the ExAllocatePoolWithTag function:

**\_\_checkReturn** instructs PRE*f*ast to issue a warning if subsequent code ignores the function return value.

**\_\_drv\_allocatesMem** indicates that the function allocates memory—in this case, pool memory.

**\_\_drv\_when** specifies a conditional expression: If the function is called with one of the “must succeed” pool types, PRE*f*ast should display the error message specified by the \_**\_drv\_reportError** annotation.

* The **\_\_bcount** annotation applies to the PVOID function return value. This annotation indicates that the return value should be NumberOfBytes long.
* The **\_\_in \_\_drv\_strictTypeMatch** annotation applies to the PoolType parameter. This annotation indicates that the parameter can take only the types implied by **\_\_drv\_typeExpr**, which can be either literal constants or expressions that involve only operands of a specific type.
* The other **\_\_in** annotations apply to the NumberOfBytes and Tag parameters, respectively. These annotations indicate that PRE*f*ast should check that these parameters are valid on entering the function.

Typically, simple annotations such as **\_\_in** are more readable when applied to parameters, but placing more complicated annotations inline can make code difficult to read. As an alternative to placing parameter annotations inline, you can enclose parameter annotations in **\_\_drv\_arg** annotations and place them with the other annotations before the start of the function, to help improve the readability of more complicated annotations.

For example, Listing 2 shows the **ExAllocatePoolWithTag** function with annotations on the *PoolType* parameter enclosed in a **\_\_drv\_arg** annotation at the beginning of the function, instead of inline.

Listing 2 Alternative placement of PRE*f*ast annotations on function parameters

\_\_checkReturn

\_\_drv\_allocatesMem(Pool)

\_\_drv\_when(PoolType&0x1f==2 || PoolType&0x1f==6,

\_\_drv\_reportError("Must succeed pool allocations are"

"forbidden. Allocation failures cause a system crash"))

\_\_drv\_arg(PoolType, \_\_in \_\_drv\_strictTypeMatch(\_\_drv\_typeExpr))

\_\_bcount(NumberOfBytes)

PVOID

ExAllocatePoolWithTag(

POOL\_TYPE PoolType,

\_\_in SIZE\_T NumberOfBytes,

\_\_in ULONG Tag

);

See “General-Purpose Annotations” later in this paper for more information about **\_\_checkReturn**, **\_\_bcount**, and **\_\_in**.

See “Driver Annotations” later in this paper for more information about **\_\_drv\_arg**, **\_\_drv\_allocatesMem**, **\_\_drv\_when**, **\_\_drv\_reportError**, and **\_\_drv\_strictTypeMatch**.

### Annotations on typedef Declarations

Annotations that are applied to **typedef** declarations are implicitly applied to functions and parameters of that type. If you apply annotations to a **typedef** declaration—including function **typedef** declarations, you do not need to apply annotations to uses of that type. PREfast interprets annotations on **typedef** declarations in the same way as it interprets annotations on functions.

The use of annotations on **typedef** declarations is both more convenient and safer than annotating each individual function parameter of a given type. For example, consider a function that takes a null-terminated string as a parameter. In the C programming language, there is no difference between an array of characters and a string, other than the programmer’s assumption that the string is null terminated. However, the semantics of many functions rely on the knowledge or assumption that a particular array of characters is null terminated and is thus semantically a string. If PREfast “knows” that an array of **char** or **wchar\_t** is intended to be a string, it can perform additional checks on the array to ensure that it is properly null terminated. The primitive annotation that expresses this is **\_\_nullterminated**.

In principle, you could explicitly apply **\_\_nullterminated** to every function parameter that takes a string as a parameter, as in the following example:

size\_t strlen(\_\_nullterminated const char \*s);

However, that quickly becomes tedious and error prone. Instead, if you declare a string parameter as a type that is already annotated with **\_\_nullterminated**, then you are not required to explicitly annotate all of the functions that use strings to ensure that their parameters are null terminated. The following **typedef** declaration for **PCSTR** is an example:

typedef \_\_nullterminated const char \*LPCSTR, \*PCSTR;

The function declaration becomes much simpler, as shown in the following example:

size\_t strlen(PCSTR s);

In this example, **PCSTR** *s* implies that s is null terminated because the PCSTR type is annotated with **\_\_nullterminated**. This declaration is easier to read than the previous example and expresses the intended use of the parameter more clearly to the programmer.

You should always use PCSTR or similar string types for strings in the functions you define. Use character types such as PCHAR only for strings that are not null terminated. If you use PCSTR and similar types for strings, you get the benefits of annotation without being required to explicitly apply them and it is easy to distinguish a function that takes a string from a function that takes an array of bytes as 8-bit numbers. However, if you use PCSTR or similar types to describe an array of bytes that might not be null terminated, the implicit **\_\_nullterminated** annotation on the string type causes PRE*f*ast to issue a false positive.

The functions in Listing 3 show the difference between a string and an array of bytes. FindChar relies on the PSTR parameter type’s implicit guarantee that the string is null terminated. The FindChar function cannot find zero as a character in the body of the string. A more realistic example would use annotations such as **\_\_drv\_when** and **\_\_drv\_reportError** to indicate that *c* must not be zero. See these annotations later in this paper for details.

FindByte relies on the PCHAR parameter type’s implicit guarantee that zero is not special in *arr* and that *len* defines the length of the array to search, so binary zero is a valid value to search for. When PRE*f*ast analyzes the FindChar function, it checks whether *str* is missing the required null terminator because the PSTR parameter type specifies that the buffer at *str* is intended as a string rather than an array.

Listing 3 Two functions that show the difference between a string and an array of bytes

PSTR FindChar(\_\_in PSTR str, \_\_in char c)

{

// str ends in '\0', so stop when we see a 0.

// Length is not needed.

// c cannot be 0, because it will never match.

}

PCHAR \* FindByte(\_\_in PCHAR \*arr, long len, \_\_in char b)

{

// We have no idea if the byte after the end of the buffer happens

// to be zero or not: we simply have to believe the length and quit

// after looking at len bytes.

// b could be zero: zero is just like any other value.

}

### Annotations on Function Typedef Declarations

In C and C++, it is possible to declare a function type by using **typedef**. This is distinct from a function *pointer* type and historically has not been used very much in C code. However, with the addition of annotations and function type classes, which are described later in this paper, function types become very useful.

Although function **typedef** declarations might look unfamiliar, they are valid standard C—that is, they are correct and compilable. For example, Listing 4 shows the **DRIVER\_STARTIO** function **typedef** declaration, which is defined in %wdk%\inc\ddk\Wdm.h.

Listing 4 DRIVER\_STARTIO function typedef declaration

typedef

VOID

DRIVER\_STARTIO (

\_\_in struct \_DEVICE\_OBJECT \*DeviceObject,

\_\_in struct \_IRP \*Irp

);

typedef DRIVER\_STARTIO \*PDRIVER\_STARTIO;

It’s important to remember that **DRIVER\_STARTIO** defines a *function* **typedef**, not a *function pointer* **typedef**. This function **typedef** declaration is used to declare that MyStartIo is a function of type **DRIVER\_STARTIO**. A function that is declared with **DRIVER\_STARTIO** is assignment-compatible with the familiar **PDRIVER\_STARTIO** function pointer— that is, a pointer to one can be assigned to a pointer to the other.

Note also that the function parameters in **DRIVER\_STARTIO** are already annotated with **\_\_in**, which identifies them as input parameters. These annotations are implicit in the **typedef** declaration, so you are not required to annotate these parameters in your MyStartIo function unless you prefer, for readability.

If your MyStartIo function is intended to be a WDM StartIo function, you would declare MyStartIo as a function of type **DRIVER\_STARTIO** by placing the following declaration before the first use of MyStartIo in your driver:

DRIVER\_STARTIO MyStartIo;

In addition to declaring MyStartIo as a function of the type **DRIVER\_STARTIO**, this declaration applies to MyStartIo all of the system-supplied annotations on the **DRIVER\_STARTIO** type as defined in %wdk%\inc\Wdm.h.

You are not required to follow the function **typedef** declaration with a full prototype. Many developers find that the function **typedef** declaration alone is more readable. You would implement the MyStartIo function body (that is, the function definition) in the same way as any other function. See the Toaster sample driver in the WDK at %wdk%\src\general\toaster\func\shared\toaster.h for an example of a driver that uses function **typedef** declarations and omits the prototypes.

The function **typedef** declaration is useful in another way: it tells other programmers that the function is intended to be an actual StartIo function, rather than just looking like one. For example, Cancel functions are assignment-compatible with StartIo functions, so a poorly chosen function name can lead to ambiguities in the source code.

You can also use the **\_\_drv\_functionClass** annotation to indicate to PRE*f*ast that a function type belongs to a particular function type class. This significantly increases the checking that PREfast can do because PRE*f*ast understands that this function is a callback and knows the specific contract it must meet. See “Function Type Class Annotations” later in this paper for details.

When you use function **typedef** declarations, remember the following:

* The function must strictly match the type that the function **typedef** declares.
* The function **typedef** declaration should have the required annotations. System-provided function **typedef** declarations already have these.
* The declaration must precede the first mention of the function. If the function appears in a header file, the new declaration should precede—or simply replace—the mention in the header. If the first mention is the function definition itself, then the declaration should immediately precede the function definition.
* The function definition is not required to be annotated because annotations that are applied to function **typedef** declarations are implicitly applied to functions and parameters of that type, the same as for any **typedef**.
* Function **typedef** declarations can become difficult to read if you annotate each parameter individually. To make function **typedef** declarations more readable, consider whether to create and annotate a **typedef** for each parameter and then declare each parameter as the appropriate type in the function **typedef** declaration.

**Note** Driver function **typedef** declarations such as **DRIVER\_STARTIO** are intended for use in drivers written in C. For a discussion of using function **typedef** declarations in drivers written in C++, see “Using Function Typedef Declarations in C++ Driver Code to Improve PREfast Results”—online at <http://go.microsoft.com/fwlink/?LinkId=87238>.

### Tips for Placing Annotations in Source Code

Here are some tips for placing annotations in source code:

* If a function has both a declaration (that is, a prototype) and a definition, annotate both with identical annotations. Function **typedef** declarations help to satisfy this requirement.

If the annotations differ, PREfast issues a warning.

* If a function has only a definition and does not have a separate prototype, annotate the definition.

You do not need to create and annotate a separate prototype if you are only applying annotations and a prototype is not required for any other reason.

* Place annotations that apply to an entire function immediately before the beginning of the function.
* Place annotations that apply to a function parameter either inline, immediately before the function parameter, or immediately before the beginning of the function, enclosed in a **\_\_drv\_arg** annotation that identifies the parameter to which the annotation applies.
* Place annotations on **typedef** declarations to implicitly apply them to functions or parameters of that type.

# General-Purpose Annotations

General-purpose annotations provide PREfast with information about the information flow between a caller and the function that is being called, both in terms of direction of information flow and in providing size and type information that can be checked to detect potential buffer overflows.

You can use the general-purpose annotations in both driver and nondriver code. General-purpose annotations are defined in Specstrings.h and described with extensive comments in Specstrings\_strict.h. Both files are in %wdk%\inc\api.

This section provides guidelines and examples for using the general-purpose annotations and modifiers listed in Table 1.

Table 1 General-Purpose Annotations

| **Annotation** | **Usage** |
| --- | --- |
| \_\_in \_\_out \_\_inout | Input and output parameters |
| \_opt \_deref | Annotation modifiers |
| \_ecount(size) \_bcount(size) | Parameter size |
| \_full(size) \_part(siz*e*, length) | Partial parameter size |
| \_\_nullterminated \_\_nullnullterminated \_\_possibly\_notnullterminated | String annotations |
| \_\_reserved | Reserved parameters |
| \_\_checkReturn | Function return value |

PREfast does not interpret certain annotations such as **\_\_fallthrough**, but these can still be useful when they are applied as comments in code. See the comments in %wdk%\inc\api\Specstrings\_strict.h for a complete list of annotations.

Annotations can be either primitive or composite. A composite annotation is composed of two or more primitive annotations and other composite annotations. This paper explains some annotations in terms of primitive annotations; however, you should choose composite annotations instead of primitive annotations whenever possible because composite annotations are more resilient to future changes.

Important For simplicity, the examples in this paper show standard functions that are not completely annotated. You should apply all appropriate annotations to your code, as described in this paper, to ensure that your functions are completely annotated and can be fully analyzed by PREfast. Do not limit the annotations in your source code to only those shown in these examples.

## Input and Output Parameter Annotations

C passes parameters by value, so parameters themselves are always input parameters to a function. However, because C can pass a pointer by value, it is impossible to tell just by looking at the function prototype whether the value that is passed by using a pointer is intended as input to the function, output from the function, or both input and output.

The **\_\_in**, **\_\_out**, and **\_\_inout** annotations enable PRE*f*ast to check parameters that have these annotations and report any errors if it finds that uninitialized values are used incorrectly:

* If a parameter is marked **\_\_in**, then PREfast checks to be sure that the parameter is initialized before the call.

You can also annotate scalar parameters such as integers or enumerated types with **\_\_in**. The **\_\_in** annotation is optional for scalar parameters, but it helps to make code more consistent and readable.

* If a parameter is marked **\_\_out**, then PREfast does not check that it is initialized before the call, but assumes that it is initialized after the call and thus is safe to use as an **\_\_in** parameter to a subsequent function.
* If a parameter is marked **\_\_inout**, then PREfast checks to be sure that the parameter is initialized before the call and assumes that it is safe to use as an **\_\_in** parameter to a subsequent function.

Tip The **\_\_in** annotation is typically all that is needed for opaque types, including KMDF handles such as WDFDEVICE.

### The \_\_in and \_\_out Contract

The term “initialized” is used informally when discussing **\_\_in** and **\_\_out**, but these annotations actually describe what makes the annotated parameters valid in relation to pre-state and post-state:

* “Valid” means that all levels of the data structure are initialized and that pointers at all levels of dereference except for the last pointer are non-NULL values, unless the parameter has been explicitly annotated differently.
* “Pre-state” refers to the state of the analysis just before the call is made. If a variable has been given a value in a prior assignment, then it has that value in the pre-state.
* “Post-state” refers to the state just after the call has returned. For an **\_\_out** parameter, the post-state is the one in which the parameter has a new value or a value at all.

Formally, **\_\_in** means that the value that is being passed as an argument must be valid in pre-state and the function does not change it, and **\_\_out** means that the function has contracted to return a valid value in post-state and that PRE*f*ast can ignore the pre-state.

In pre-state, **\_\_out** implies that any intermediate pointers that lead to the final value to be modified are individually valid and not NULL, but the parameter that is annotated with **\_\_out** is not required to be recursively valid in pre-state. For example, suppose you want to pass a pointer to a pointer to a structure s (that is, you want to pass \*\**p*). In pre-state, this requires that both *p* and \**p* must be valid pointers to other pointers. However, for an **\_\_out** parameter, \*\**p* is not required to be valid—that is, in pre-state, \*\**p* is not required to be fully initialized. For the **\_\_out** annotation in post-state, the structure at \*\*s must be valid (that is, the structure is expected to be filled in.) The **\_opt** modifier can be used to indicate whether any intermediate levels can be omitted—for example, if \**p* can be NULL.

**\_\_inout** means that the parameter must be valid in pre-state and valid in post-state and that both should be checked. It also means that any assumptions that PREfast has made about the value of the parameter—beyond simple validity—are no longer true in post-state. That is, the value is presumed to have changed.

PREfast analyzes both sides of the contract. The **\_\_in**, **\_\_out**, and **\_\_inout** annotations are a definite illustration of that statement. For the function that is being analyzed—the callee—PREfast assumes that the pre-state is true at the entry point and checks that the post-state is achieved at the exit point. Specifically, the analysis of a function starts with the initial state of the parameters. PREfast assumes that any **\_\_in** parameters are valid at the beginning of the function and checks that **\_\_out** parameters are valid at the return from the function.

The following two examples show how PRE*f*ast analyzes contracts according to annotations. In the following example, the **\_\_in** annotation causes PREfast to check that the s input parameter is initialized before a **strlen** call in the following declaration for **strlen**:

size\_t strlen(\_\_in PCSTR s);

Another example of checking both sides of the contract is the **\_\_nullterminated** annotation that is implicit in the string types such as PSTR, PWSTR and so on, as shown in the following example:

PSTR substitute(\_\_inout\_ecount(len) PSTR str,

\_\_in int len, \_\_in PSTR oldstr, \_\_in PSTR newstr);

In this example, a function named substitute takes a string as input and substitutes all instances of *oldstr* with the value of *newstr*. The function never overruns *len* bytes, and it ensures that the resulting value of *str* is always null terminated. Assuming the substitute function does something, the **\_\_inout** part of the annotation applied to *str* indicates that the value of *str* before and after the call is different, but it is valid both before and after the call. PRE*f*ast performs the following analysis of this contract:

* For the caller part of the contract, PRE*f*ast checks that *str*, *oldstr*, and *newstr* are all null terminated at the point of the call, as far as can be determined statically. It also checks that the buffer at *str* is big enough for *len* bytes.
* For the callee part of the contract, PRE*f*ast checks that no access is made past *len* bytes into *str*. PRE*f*ast assumes that *str*, *oldstr*, and *newstr* are null terminated. The critical check that PRE*f*ast makes is that the final result value of *str* is null terminated.

Most of the annotations described later in this paper, including the IRQL, memory, and nonmemory resource driver annotations, have both caller and callee semantics to assure that both sides of the contract are met.

### \_\_in, \_\_out, and \_\_inout versus IN, OUT, and IN OUT

In general, you should replace all instances of **IN**, **OUT**, and **IN OUT** with **\_\_in**, **\_\_out**, and **\_\_inout**, respectively**.** However, do not simply redefine these older macros in terms of the newer annotations.

Although the **IN** and **OUT** macros often appear in source code, they have never been given a value and are never validated by any tool or compiler, so they do not always reflect the actual usage of the parameters and could be incorrect. Therefore, these macros might be incorrectly used or placed in existing source code. You should review functions that use **IN**, **OUT**, and **IN OUT** and make sure to place the correct **\_\_in**, **\_\_out**, and **\_\_inout** annotations in the appropriate locations.

## Annotation Modifiers

For various reasons related to implementation, many annotations that must be applied to function parameters must be represented as a single macro, rather than as a series of adjacent macros. In particular, this is true for most of the various basic annotations, which should appear as a single composite macro for each parameter.

This is accomplished by adding modifiers to the annotation to compose a more complete annotation. The two most common modifiers, **\_opt** and **\_deref**, are examples of how to create more complex annotations by combining simpler annotations.

### The \_opt Modifier

The **\_\_in** annotation does not allow null pointers, but often a function can take a NULL in the place of an actual parameter. The **\_opt** modifier indicates that the parameter is optional; that is, it can be NULL. For example, an optional input parameter—such as a pointer to a structure—would be annotated as **\_\_in\_opt**, whereas an optional output parameter would be coded as **\_\_out\_opt**.

Typically, **\_\_in\_opt** and **\_\_out\_opt** are used for pointers to structures with a fixed size. Additional modifiers can be applied to annotate variable-sized objects, as described in “Buffer-Size Annotations” later in this paper.

In general, you should replace all instances of the **OPTIONAL** macro with the **\_opt** modifier. However, because no tool or compiler validates **OPTIONAL**, check the code carefully to ensure that parameters labeled **OPTIONAL** actually are optional.

### The \_deref Modifier

User-defined types such as structures can be declared as parameter types, so it is sometimes necessary to annotate the dereferenced value of a parameter. The **\_deref** modifier indicates that an annotation should be applied to the dereferenced value of a parameter, and not the parameter itself.

For example, consider the following function:

int myFunction(struct s \*\*p);

When you pass a pointer such as **struct** s \**p* to a function, the memory that \**p* points to is passed by reference. *p* itself is passed by value. In this example, the *p* parameter is a variable of type pointer-to-s that is being passed by reference. \*\**p* is a variable of type **struct** s, \**p* is a pointer to that variable, and *p* is a pointer to that pointer.

In this example, the myFunction function is defined to modify \**p*, the pointer to the variable of type **struct** s. The function requires that *p* not be NULL. However, the function allows **\***p to be NULL—if \**p* is NULL, the function simply takes no action on \*p.

Annotating *p* as **\_\_inout** would require that **\***p be non-NULL. Annotating *p* as **\_\_inout\_opt** would allow p to be NULL. However, neither of these annotations correctly expresses the intended behavior of myFunction.

Adding the **\_deref** modifier to the annotation applies **\_\_inout\_opt** to the proper dereferenced value of *p*, as shown in the following example:

int myFunction(\_\_inout\_deref\_opt struct s \*\*p);

This annotation specifies that the **\_opt** annotation applies to \**p*, which is the dereferenced value of *p*; that is, \**p* can be NULL. The **\_opt** annotation does not apply to *p* itself; that is, *p* cannot be NULL. Put another way, **\_deref\_opt** applies to the parameter that is passed by reference—\**p*—instead of the address of the reference—*p*.

The **\_deref** modifier can appear more than once in an annotation, to indicate multiple levels of dereference. For example, **\_\_in\_deref\_deref\_opt** indicates that \*\**p* can be NULL. Many of the examples later in this paper show the use of **\_deref** with other annotations.

**Note** The **\_\_null** and **\_\_notnull** annotations, which explicitly indicate that a particular parameter can be NULL or must not be NULL, are built in to the composite general-purpose annotations such as **\_\_inout**. It is not necessary to include **\_\_null** and **\_\_notnull** in annotations such as the ones in this example.

## Buffer-Size Annotations

A variable-sized object is any object that does not carry its own size with it. Many bugs in code, particularly security bugs, are caused by buffer overflows in which a variable-sized object is being passed. The following annotations can be used to express the contract between the caller and the callee about the size of buffers:

**\_ecount(**size**)  
\_bcount(**size**)**  
**\_full(**size**)**  
**\_part(**size, length**)**

The contract must specify the size—or where to find it—and how it will be used. This information ensures that neither the caller nor the callee accesses data outside the bounds of the buffer, but the contract can also express the difference between available memory and initialized memory, so that access to uninitialized memory can be detected.

In C, buffers are typically arrays of something. When you describe the size of a buffer in your code, the size can be measured in two ways: as the number of bytes in the buffer or the number of elements in the buffer. For arrays of anything other than **char**, the size in elements differs from the size in bytes. In most cases—even for arrays of **char**—the size in elements is more useful and easier to express.

Note The sizes of wide character strings such as **wchar\_t** are usually expressed in elements, not bytes. UNICODE\_STRING is a notable exception.

### Fixed-Size Buffer Annotations

The \_**ecount(**size**)** and \_**bcount(**size**)** partial annotations are used to express the size of a buffer. Use \_**ecount(***size***)** to express the size of a buffer as a number of elements. Use \_**bcount(***size***)** to express the size of the buffer as a number of bytes. The *size* parameter can be any general expression that makes sense at compile time. It can be a number, but it is usually the name of some parameter in the function that is being annotated.

The following example of the **memset** function shows a typical buffer annotation:

void \* memset(

\_\_out\_bcount(s) char \*p,

\_\_in int v,

\_\_in size\_t s);

In this example, **\_\_out\_bcount(**s**)** specifies that the content of the memory at p is set by the function and that the value of s is the number of bytes to be set. Nothing in the C source code tells the compiler that p and s are related in this way. The annotation provides this useful information.

With this information provided by the annotation, PREfast can check the implementation of **memset** to be sure it never accesses past the end of the buffer—that is, it never accesses more than s bytes into the buffer. Often, PREfast can also check that the value of p+s is within the declared bounds of the array when **memset** is called. In this case, the buffer size is expressed in bytes because that is what **memset** expects.

Compare **memset** with a similar function, **wmemset**:

wchar\_t \* wmemset(

\_\_out\_ecount(s) wchar\_t \*p,

\_\_in wchar\_t v,

\_\_in size\_t s);

This example uses **\_\_out\_ecount** to indicate that s is represented in elements of the **wchar\_t** type. If some incorrect code called this function with a byte count—which is an easy mistake to make—the value of s is likely to be twice as large as it should be. With the **\_\_out\_ecount** annotation, PREfast has a good chance of detecting a buffer overrun in the caller and identifying a probable bug.

Note that for **\_\_in** parameters, the definition of “valid” requires that the whole parameter being passed must be initialized. This also applies to arrays, which are passed by reference. Thus, when you use **\_\_in** for a parameter that is an array, the whole array must be initialized up to the limit specified by **\_bcount** or **\_ecount**. See “String Annotations” later in this paper for details about how this applies to null-terminated strings.

The **\_bcount** and **\_ecount** annotations are sufficient to describe **\_\_in** buffers that are not modified or **\_\_out** buffers that are fully initialized. For buffers that are partially initialized and that might have the initialized portion extended or reduced in place, you can combine these annotations with the **\_part** and **\_full** modifiers:

* The **\_full** modifier applies to the entire buffer. For an output buffer, the **\_full** modifier indicates that the function initializes the entire buffer. For an input buffer, the **\_full** modifier indicates that the buffer is already initialized, although this is redundant with other annotations.
* The **\_part** modifier indicates that the function initializes part of the buffer and explicitly indicates how much.

When you combine these modifiers with **\_\_inout** buffers and **\_full(**size**)** or **\_part(**size, length**)** annotations, you can use them to represent the “before” and “after” sizes of a buffer. *Size* and *length* can be constants, or they can be parameters of the function being annotated. The following examples show the use of *size* and *length* in buffer annotations:

* **\_\_inout\_bcount\_full(**cb**)** describes a buffer that is cb bytes in size, is fully initialized at entry and exit, and might be written to by this function.
* **\_\_out\_ecount\_part(**count**, \***countOut**)** describes a buffer that is *count* elements in size and is to be partially initialized by this function. The function indicates the number of elements it initialized by setting **\***countOut.

### Summary of Annotations for Buffers

This section summarizes the annotations that can be combined to describe a buffer. Table 2 lists these annotations.

Table 2 Annotations for Buffers

| **Level** | **Usage** | **Size** | **Output** | **Optional** | **Parameters** |
| --- | --- | --- | --- | --- | --- |
| *omitted*  \_deref  \_deref\_opt | *omitted*  \_\_in  \_\_out  \_\_inout | *omitted*  \_ecount  \_bcount  \_xcount(*expr*) | *omitted*  \_full  \_part | *omitted*  \_opt | *omitted*  (*size*)  (*size*, *length*) |

The headings in Table 2 are described in the following list.

**Level** in Table 2 describes the buffer pointer’s level of dereference from the parameter or return value p. Level can be one of the following:

*omitted*

*p* is the buffer pointer.

\_deref

\**p* is the buffer pointer. *p* must not be NULL.

\_deref\_opt

\**p* is the buffer pointer. *p* can be NULL, in which case the rest of the annotation is ignored.

**Usage** in Table 2 describes how the function uses the buffer. Usage can be one of the following:

*omitted*

The buffer is not accessed. If used on the return value or with **\_deref**, the function provides the buffer and the buffer is uninitialized at exit. Otherwise, the caller must provide the buffer. This should be used only for **alloc** and **free** functions.

\_\_in

The buffer is used only for input.. The caller must provide the buffer and initialize it.

\_\_out

The buffer is used only for output.. If used on the return value or with **\_deref**, the function provides the buffer and initializes it. Otherwise, the caller must provide the buffer and the function initializes it.

\_\_inout

The function may freely read from and write to the buffer. The caller must provide the buffer and initialize it. If used with **\_deref**, the buffer may be reallocated by the function.

**Size** in Table 2 describes the total size of the buffer. This can be less than the space that is actually allocated for the buffer, in which case it describes the accessible amount. Size can be one of the following:

*omitted*

No buffer size is given. If the type specifies the buffer size—such as with LPSTR and LPWSTR—that amount is used. Otherwise, the buffer is one element long. This must be used with **\_\_in**, **\_\_out**, or **\_\_inout**.

\_ecount

The buffer size is an explicit element count.

\_bcount

The buffer size is an explicit byte count.

\_xcount(*expr*)

The buffer size cannot be expressed as a simple byte or element count. For example, the count might be in a global variable, in a structure member, or implied by an enumeration. PRE*f*ast treats *expr* as a comment and does not use it to check buffer size. *expr* can be anything that is meaningful to the reader, such as an actual expression or a quoted string.

Important \_**xcount** satisfies the need to annotate a buffer, but causes PREfast to skip actual size checks. You can use **\_xcount** as a placeholder for annotations that become meaningful in future tools. However, use **\_xcount** with caution and restraint because it suppresses potential warnings and analysis.

**Output** in Table 2 describes how much of the buffer is initialized by the function. For **\_\_inout** buffers, this partial annotation also describes how much is initialized at entry. Omit this category for **\_\_in** buffers—they must be fully initialized by the caller.

Output can be one of the following:

*omitted*

The type specifies how much is initialized. For example, a function that is initializing an LPWSTR must null-terminate the string.

\_full

The function initializes the entire buffer.

\_part

The function initializes part of the buffer and explicitly indicates how much.

**Optional** in Table 2 describes whether the buffer itself is optional. This annotation modifier can be one of the following:

*omitted*

The pointer to the buffer must not be NULL.

\_opt

The pointer to the buffer might be NULL. It is checked by PRE*f*ast before being dereferenced.

**Parameters** in Table 2 gives explicit counts for the size and length of the buffer. Size and length can be either constant expressions or an expression that involves a parameter—usually other than the one being annotated. Length should refer to the resulting value of an **\_\_out** parameter. Parameters can be one of the following:

*omitted*

There is no explicit count. Use when neither **\_ecount** nor **\_bcount** is used.

(*size*)

This is the buffer’s total size. Use with **\_ecount** or **\_bcount** but not with **\_part**.

(*size*,*length*)

This is the buffer’s total size and initialized length. Use with **\_ecount\_part** and **\_bcount\_part**.

### Tips for Applying Annotations to Buffers

When applying annotations to buffers, remember the following:

* Each buffer annotation describes a single buffer with which the function interacts: where it is, how large it is, how much is initialized, and what the function does with it.

The buffer can be a string, a fixed-length or variable-length array, or just a pointer.

* You should use only a single buffer annotation for each parameter.
* Some combinations do not make sense as buffer annotations. See the buffer annotation definitions in Specstrings.h for a list of meaningful combinations.

### Buffer Annotation Examples

The examples in Listings 5 and 6 show uses of buffer annotations.

The example in Listing 5 does an in-place substitution on a counted array of characters. The old size is the input value for \*s, and the new size is the output. This function might be used to substitute UCS-8 for non-ASCII characters.

Listing 5 Example of annotations for in-place substitution on a counted array of characters

void substUCS8(

\_\_inout\_ecount\_part(\*s, \*s) wchar\_t \*buffer,

\_\_inout size\_t \*s);

The example in Listing 6 shows the use of **\_xcount** to annotate a buffer size that cannot be expressed as a simple expression. There are several better ways to implement this kind of function. This example simply illustrates the use of **\_xcount**. The function returns a string of one of three known lengths, depending on the input parameter *which.* This annotation causes PRE*f*ast to skip size checks on \**msgBuffer*.

Listing 6 Example of annotations for a buffer size that cannot be expressed as a simple expression

GetString(

\_\_out\_xcount("23, 42, or 26, depending on 'which'")

LPSTR \*msgBuffer,

\_\_in which);

Tip Specstrings.h defines a number of similar annotations to specify buffer sizes. If you do not find what you need, read the comments in Specstrings\_strict.h to see if the problem you are trying to solve has already been addressed.

## String Annotations

C null-terminated strings represent a special case of buffers. The following annotations describe null-terminated strings:

**\_\_nullterminated  
\_\_nullnullterminated  
\_\_possibly\_notnullterminated**

String annotations are useful when applied to **typedef** declarations. These annotations enable PREfast to check that the type is used correctly in a function without requiring the programmer to annotate every function parameter that uses the type. See “Annotations on Typedef Declarations” earlier in this paper for information about applying annotations to types.

**Note** The examples in this section are intended only to illustrate the use of annotations for null-terminated strings. You should always use the safe string functions declared in %wdk%\inc\ddk\ntstrsafe.h for string and UNICODE\_STRING manipulations instead of writing your own string manipulation functions.

### \_\_nullterminated

Many of the types declared in system header files are already annotated. If you use the appropriate STR type for all functions that take strings described as **char \*** or **wchar\_t \*** parameters, it is not necessary to apply the **\_\_nullterminated** annotations to these types. PSTR, PCSTR, and their “L” and “W” variations all imply that a string is null terminated. You should explicitly apply **\_\_nullterminated** only to types that are not already annotated with **\_\_nullterminated**.

The implied use of **\_\_nullterminated** through use of PSTR or PCSTR types is sufficient for input string buffers. If the parameter is strictly for input, use the CSTR forms because placement of the **const** modifier must be done inside the **typedef**.

If a function can create or add to a string, the function must have the actual size of the output buffer so it can avoid overruns. Output buffers should also have an additional **\_ecount** or **\_bcount** annotation that gives the actual buffer size because **\_\_nullterminated** by itself does not provide that information. For **\_\_out** parameters, the count annotation specifies that the resulting string is null terminated and that PREfast should check for buffer overruns. For **\_\_inout** parameters, the count annotation implies that the buffer is initialized up to the NULL and that the updated value is also null terminated.

For example, the **StringCchCopy** function copies up to cchDest elements. The **\_\_out\_ecount** annotation specifies that although the string is null terminated—as indicated by the LPSTR type—it does not overflow cchDest bytes, as shown in the following example:

StringCchCopyA(

\_\_out\_ecount(cchDest) LPSTR pszDest,

\_\_in size\_t cchDest,

\_\_in LPCSTR pszSrc);

### \_\_nullnullterminated

The **\_\_nullnullterminated** annotation is intended for the occasional “string of strings” that is terminated by a double null, such as a registry value whose type is REG\_MULTI\_SZ. Currently, PRE*f*ast does not check **\_\_nullnullterminated**, so this annotation should be considered advisory only. However, **\_\_nullnullterminated** might be enabled in a future version of PRE*f*ast. In the meantime, use a **#pragma warning** directive or the **\_xcount** annotation to silence PRE*f*ast noise related to strings terminated with a double NULL.

### \_\_possibly\_notnullterminated

Several older functions usually return null-terminated strings but occasionally do not. The classic examples are **snprintf** and **strncpy**, where the function omits the null terminator if the buffer is exactly full. These functions are considered deprecated and should not be used. Instead, you should use the equivalent functions declared in StrSafe.h for user-mode applications or NtStrSafe.h for kernel-mode code because they guarantee a null-terminated buffer on success.

However, it might not be possible to completely eliminate this kind of function in existing code, so you should annotate these functions by applying **\_\_possibly\_notnullterminated** to their output parameters, as shown in the following example:

int \_snprintf(

\_\_out\_ecount(count) \_\_possibly\_notnullterminated LPSTR buffer,

\_\_in size\_t count,

\_\_in LPCSTR \*format

[, argument] ...

);

When PREfast encounters a **\_\_possibly\_notnullterminated** annotation, it attempts to determine whether an action was taken to assure null termination of the output string. If it cannot find one, PRE*f*ast generates a warning.

## Reserved Parameters

Occasionally a function has a parameter that is intended for future use. The **\_\_reserved** annotation ensures that in future versions, old callers to a function can be reliably detected. This annotation insists that the provided parameter be 0 or NULL, as appropriate to the type.

For example, someday the following function will take a second parameter, but all current use of that parameter should be coded with NULL. The following annotation enables PREfast to check that current callers do not misuse the reserved parameter:

void do\_stuff(struct a \*pa, \_\_reserved void \*pb);

## Function Return Values

Many functions return a status that indicates whether the function was successful. However, it is common to find code that assumes that a function call is always successful and that does not check the return value. Memory allocators are often in this class, but there are quite a few others as well. For example, **malloc** is the classic function that should be marked with **\_\_checkReturn**, as shown in the following example:

\_\_checkReturn void \*malloc(\_\_in size\_t s);

The **\_\_checkReturn** annotation indicates that the function return value should be checked. PREfast can detect two different errors for a function annotated with **\_\_checkReturn**:

* The function return value is simply ignored.
* The function return value is placed into a variable and the variable is then ignored.

To avoid a warning when calling a function that is annotated with **\_\_checkReturn**, either use the return value directly in a conditional expression or assign it to a variable that is subsequently used in a conditional expression. Although **\_\_checkReturn** is traditionally applied to return values, PREfast can detect a **\_\_checkReturn** annotation that is applied to an **\_\_out** parameter to insist that the value be examined.

In the rare case when it might make sense to ignore the return value, call the function in an explicit void context: **(void)mustCheckReturn(**param**).**

Returning the value to a caller qualifies as successfully checking the return value; however, that parameter or the return value should itself be annotated with **\_\_checkReturn** so that the caller checks the value.

Kernel-mode drivers should be annotated to check all memory allocations, and the driver should attempt to fail gracefully if a memory allocation fails.

**Note** If you have used the **/analyze** option in Visual Studio, you might notice that PREfast behaves slightly differently when analyzing functions that are annotated with **\_\_checkReturn**. Both **/analyze** and PRE*f*ast issue a warning if the function’s return value is discarded at the point of the function call. However, PREfast also issues a warning if the function’s return value is assigned to a variable and that variable is not used in subsequent code.

# Driver Annotations

Driver annotations enhance the ability of PRE*f*ast to find some very specific kinds of errors in driver source code. Driver annotations are intended to complement the general-purpose annotations described earlier in this paper. The driver annotations are defined in %wdk%\inc\ddk\Driverspecs.h and begin with the prefix **\_\_drv**.

Although the driver annotations were initially designed for WDM kernel-mode drivers, with few exceptions they apply at a low enough level that it does not matter which driver model you’re using. Many of the driver annotations can also be used to annotate general kernel-mode code or user-mode drivers. Most of the examples in this section are drawn from WDM drivers, but the usages they illustrate should also work for WDF drivers. A few annotations make sense only for kernel-mode code and thus do not make sense for UMDF drivers. These annotations are noted in this paper.

Driver annotations use a slightly different syntax than the general-purpose annotations such as **\_\_in** and **\_\_out**. Some driver annotations identify the exact object that is being annotated. For example, **\_\_drv\_arg** applies a list of annotations to a named formal parameter to a function. Other driver annotations describe the semantics of a function. For example, **\_\_drv\_MaxIRQL** indicates the maximum interrupt level at which a function can be called.

All driver annotations either implicitly or explicitly apply to a specific element of the function that is being annotated. The specific element can be one of the following:

* The function itself (that is, the global state of the function).
* A function parameter.
* The function’s return value.
* The **this** pointer in C++ code.

A driver annotation can apply at any level of dereference, equivalent to the **\_deref** general-purpose partial annotation. The annotation can apply to the pre-state or post-state condition of the function or parameter.

Important You can combine lower-level general-purpose annotations such as **\_\_notnull** with driver annotations. However, you should not combine composite general-purpose annotations such as those built up from **\_\_in**, **\_\_out**, and **\_\_inout** with driver annotations. For example, a composite general-purpose annotation such as **\_\_inout\_ecount\_part** should appear separately from driver annotations, either adjacent to them or on a separate line, and should not be concatenated with a driver annotation or included in an annotation list.

Table 3 summarizes the driver annotations, in the order in which they are discussed in this paper.

Table 3 Driver Annotations

| **Annotation** | **Usage** |
| --- | --- |
| **\_\_drv\_arg(***arg*,*anno\_list***)  \_\_drv\_arg(\_\_param(***n***)**,*anno\_list***) \_\_drv\_deref(***anno\_list***)  \_\_drv\_fun(***anno\_list***)  \_\_drv\_in(***anno\_list***) \_\_drv\_in\_deref(***anno\_list***)  \_\_drv\_out(***anno\_list***)  \_\_drv\_out\_deref(***anno\_list***)  \_\_drv\_ret(***anno\_list***)** | Basic driver annotations |
| **\_\_drv\_when**(cond, anno\_list) | Conditional annotations |
| **\_\_drv\_valueIs**(list) | Function result annotations |
| **\_\_drv\_strictTypeMatch(**mode**) \_\_drv\_strictType(**typename, mode**)** | Type annotations |
| **\_\_drv\_notPointer \_\_drv\_isObjectPointer** | Pointer annotations |
| **\_\_drv\_constant \_\_drv\_nonConstant** | Constant and nonconstant parameter annotations |
| **\_\_drv\_formatString(**kind**)** | Format string annotations |
| **\_\_drv\_preferredFunction(**name**,** reason**) \_\_drv\_reportError(**string**)** | Diagnostic annotations |
| **\_\_drv\_inTry \_\_drv\_notInTry** | Annotations for functions in **\_\_try** statements |
| **\_\_drv\_allocatesMem(**type**) \_\_drv\_freesMem(**type**) \_\_drv\_aliasesMem** | Memory annotations |
| **\_\_drv\_acquiresResource(**kind**) \_\_drv\_releasesResource(**kind**) \_\_drv\_acquiresResourceGlobal(**kind**,** param**) \_\_drv\_releasesResourceGlobal(**kind**,** param**) \_\_drv\_acquiresCriticalRegion \_\_drv\_releasesCriticalRegion \_\_drv\_acquiresCancelSpinLock \_\_drv\_releasesCancelSpinLock \_\_drv\_mustHold(**kind**) \_\_drv\_neverHold(**kind**) \_\_drv\_mustHoldGlobal(**kind**,** param**) \_\_drv\_neverHoldGlobal(**kind**,** param**) \_\_drv\_mustHoldCriticalRegion \_\_drv\_neverHoldCriticalRegion \_\_drv\_mustHoldCancelSpinLock \_\_drv\_neverHoldCancelSpinLock \_\_drv\_acquiresExclusiveResource(**kind**) \_\_drv\_releasesExclusiveResource(**kind**) \_\_drv\_acquiresExclusiveResourceGlobal(**kind**,** param**) \_\_drv\_releasesExclusiveResourceGlobal(**kind**,** param**)** | Nonmemory resource annotations |
| **\_\_drv\_functionClass** | Function class annotations |
| **\_\_drv\_floatSaved \_\_drv\_floatRestored** | Floating point annotations |
| **\_\_drv\_maxIRQL(***value***) \_\_drv\_minIRQL(***value***) \_\_drv\_setsIRQL(***value***) \_\_drv\_requiresIRQL(***value***) \_\_drv\_raisesIRQL(***value***) \_\_drv\_savesIRQL \_\_drv\_restoresIRQL \_\_drv\_savesIRQLGlobal(***kind*, *param***) \_\_drv\_restoresIRQLGlobal(***kind*, *param***) \_\_drv\_minFunctionIRQL(***value***) \_\_drv\_maxFunctionIRQL(***value***) \_\_drv\_sameIRQL \_\_drv\_isCancelIRQL** | IRQL annotations |
| **\_\_drv\_clearDoInit** | DO\_DEVICE\_INITIALIZING annotation |
| **\_\_drv\_interlocked** | Annotations for interlocked operands |

## Basic Driver Annotations and Conventions

You can use the basic driver annotations in Table 4 throughout a driver. The anno\_list argument consists of one or more annotations in a space-separated list.

Table 4 Basic Driver Annotations

| **Annotation** | **Description** |
| --- | --- |
| **\_\_drv\_arg(***arg***,** *anno\_list***)**  **\_\_drv\_arg(\_\_param(***n***),** *anno\_list***)** | Indicates that the annotations in *anno\_list* apply to *arg*, which is a named formal parameter to the function. The *arg* can be a parameter name or the C++ **this** pointer. You can use the indirection operator such as “\*” to specify the level of dereference that is being annotated.  You can use the annotation **\_\_param(***n***)** in place of *arg* to specify a 1-based parameter position. For example, **\_\_param(3)** indicates that the annotation applies to the third parameter to the function. |
| **\_\_drv\_deref(***anno\_list***)** | Indicates that an additional level of dereference should be applied to the annotation. This annotation is equivalent to **\_\_drv\_arg(**\***\_\_param(***n***)**, *anno\_list***)**; that is, it applies to the dereferenced value of *n*. |
| **\_\_drv\_fun(***anno\_list***)** | Indicates that the annotations in *anno\_list* apply to the function as a whole. That is, the annotation applies to some property of the global state of the calling function. |
| **\_\_drv\_in(***anno\_list***)** | Indicates that *anno\_list* applies on input—a precondition. |
| **\_\_drv\_in\_deref(***anno\_list***)** | Is equivalent to **\_\_drv\_in(\_\_drv\_deref(***anno\_list***))**. This annotation is provided as a convenience. |
| **\_\_drv\_out(***anno\_list***)** | Indicates that *anno\_list* applies on output—a postcondition. |
| **\_\_drv\_out\_deref(***anno\_list***)** | Is equivalent to **\_\_drv\_out(\_\_drv\_deref(***anno\_list***))**. This annotation is provided as a convenience. |
| **\_\_drv\_ret(***anno\_list***)** | Indicates that the annotations in *anno\_list* apply to the function return value. |

The following conventions are used for annotation lists and for nesting annotations.

#### Annotation Lists for Drivers

An annotation list can contain one or more of the following:

* Positioning annotations, such as **\_\_deref**, **\_\_drv\_deref**, and **\_\_drv\_arg**.
* Conditional annotations, specified with **\_\_drv\_when**.
* General-purpose annotations, such as **\_\_notnull**.
* Other annotations that make sense for the function or parameter.

Annotation lists are frequently used with the **\_\_drv\_arg** annotation, which you can use to place the annotations that apply to a function parameter at the beginning of a function, instead of inline with the parameter. For example, the following **\_\_drv\_arg** annotation applies to a function parameter named *NumberOfBytes*. The annotation list consists of a single annotation, **\_\_in**, as shown in the following example:

\_\_drv\_arg(NumberOfBytes, \_\_in)

The following example shows another **\_\_drv\_arg** annotation that applies to a function parameter named \**pBuffer*:

\_\_drv\_arg(\*pBuffer,

\_\_drv\_neverHold(EngFloatState)

\_\_drv\_notPointer)

The annotation list consists of two annotations: **\_\_drv\_neverHold(**EngFloatState**)**, which specifies that the floating-point state should not be held when the function is called, and **\_\_drv\_notPointer**, which specifies that \**pBuffer* should point directly to memory. These annotations are explained in more detail in “Examples of Annotated System Functions,” later in this paper (see also Listing 29).

Other driver annotations that take annotation lists include **\_\_drv\_in** and **\_\_drv\_out**. For example, the following annotation list indicates that both the annotated output parameter and its first level of dereference must not be NULL. This annotation list would be placed inline with a function parameter, as shown in the following example:

\_\_drv\_out(\_\_notnull \_\_deref(\_\_notnull))

Note the use of **\_\_deref** to apply the second **\_\_notnull** annotation to the dereferenced value of the parameter.

Annotation lists can become unwieldy, especially when you are applying annotations at various levels of dereference. For example, suppose you want to annotate a \*\*\**p* parameter as an input parameter that must not be NULL.

If you place the annotations inline, the annotation would look like the following, which is extremely difficult to read:

\_\_drv\_in(\_\_drv\_deref(\_\_drv\_deref(\_\_drv\_deref(\_\_notnull)))) char \*\*\*p

Notice the three nested **\_\_drv\_deref** annotations, which are necessary to apply the **\_\_notnull** annotation to \*\*\**p*.

If you use a **\_\_drv\_arg** annotation instead, the annotation becomes much simpler to read, as shown in the following example:

\_\_drv\_arg(\*\*\*p, \_\_drv\_in(\_\_notnull)) char \*\*\*p

This **\_\_drv\_arg** annotation associates **\_\_drv\_in(\_\_notnull))** with \*\*\**p*, instead of using **\_\_drv\_deref** three times to dereference \*\*\**p*.

#### Nesting Annotations

You can combine the basic driver annotations by nesting them. Because there is often more than one way to express the same annotation, the choice is usually based on readability.

Each of the following five examples does exactly the same thing: applies the annotations in *anno\_list* to the *p1* parameter of xyz type at one level of dereference. These examples also show how to compose the annotation and how to separate a complex annotation from the parameter to which it applies.

The following two examples apply the annotations in anno\_list to parameter p1of type xyz at one level of dereference. Both of these annotations must be placed inline with the parameter declaration:

\_\_drv\_in\_deref(anno\_list) xyz p1;

\_\_drv\_in(\_\_drv\_deref(anno\_list)) xyz p1;

The following three examples are equivalent to the two preceding examples, but they can be placed anywhere that an annotation for that function is valid. They are not required to be near the declaration of p1:

\_\_drv\_arg(p1, \_\_drv\_in(\_\_drv\_deref(anno\_list)))

\_\_drv\_arg(\*p1, \_\_drv\_in(anno\_list))

\_\_drv\_arg(p1, \_\_drv\_in\_deref(anno\_list))

See “Examples of Annotated System Functions” later in this paper for additional examples of nesting annotations.

## Conditional Annotations

Some functions have complex interfaces in which some aspect of the interface is applicable only in certain circumstances. For such functions, applying annotations unconditionally can create a situation in which, no matter how the annotations are written, PREfast issues some false positives for perfectly good code.

You can use the **\_\_drv\_when(**cond**,** anno\_list**)** annotation to conditionally apply annotations. This annotation specifies that the annotations in anno\_list should be checked only if the cond conditional expression is true. In a **\_\_drv\_when** annotation:

* Cond is a condition that is evaluated according to C syntax.

If the condition cannot be evaluated to a constant, then PREfast simulates the function with both the true and false possibilities.

* Anno\_list is a list of appropriate annotations, as described in “Annotation Lists for Drivers” earlier in this paper.

You can freely mix **\_\_drv\_when** with the annotations that are described in “Basic Driver Annotations and Conventions” earlier in this paper.

For example, a primary use of a condition is to indicate that, if a called function is successful, it acquires some resource—either memory or another resource type. If the called function is unsuccessful, it does not acquire the resource and thus the resource cannot leak. This information is particularly important for functions that return NTSTATUS to indicate success because the resource that is acquired might not change in any detectable way. Only the function’s status value indicates success or failure. See “Memory Annotations” later in this paper for more about annotations for resources.

One of the more common examples of a function that benefits from conditional annotation is **ExAcquireResourceExclusiveLite**, as shown in the following example:

BOOLEAN ExAcquireResourceExclusiveLite(

\_\_in PERESOURCE Resource,

\_\_in BOOLEAN Wait);

If Wait is true, then the function’s BOOLEAN return value is not required to be checked. If Wait is false, however, the return value must always be checked. One approach would be to write code that checks when Wait is true, but this is both irritating and confusing. However, failing to check when Wait is false could result in a serious bug.

For the case in which Wait is false, PRE*f*ast requires the **\_\_checkReturn** annotation for good analysis. To avoid checking the return value when Wait is true, the example in Listing 7 uses **\_\_drv\_when** to make the **\_\_checkReturn** conditional.

Listing 7 Example of ExAcquireResourceExclusiveLite with conditional annotation

\_\_drv\_when(!Wait, \_\_checkReturn)

BOOLEAN ExAcquireResourceExclusiveLite(

\_\_in PERESOURCE Resource,

\_\_in BOOLEAN Wait);

### Examples of Nested Conditional Annotations

The following example shows nesting of driver annotations in a single condition. The annotations indicate that when Wait is true, p must not be NULL:

\_\_drv\_when(Wait, \_\_drv\_arg(p, \_\_drv\_in(\_\_drv\_deref(\_\_notnull)))

Starting from the end of the annotation in this example:

* **\_\_drv\_in**(**\_\_drv\_deref**(**\_\_notnull**)) is the list of annotations to apply to *p*. These annotations specify that *p* should be checked on input and that the dereferenced value of *p* must not be NULL.
* The enclosing **\_\_drv\_arg** annotation identifies *p* as the argument associated with the list of annotations.
* The enclosing **\_\_drv\_when** specifies that, when *Wait* is true, the annotations in **\_\_drv\_arg** should be applied to *p*.

**\_\_drv\_when** annotations can be nested. The inner annotation is applied when both the inner and outer conditions are true. The following example shows nesting of two conditional annotations. These annotations indicate that that when Wait is true, p must not be NULL and, if both Wait and p2 are true, p3 must not be NULL:

\_\_drv\_when(Wait, \_\_drv\_arg(p, \_\_drv\_in(\_\_notnull) )

\_\_drv\_when(p2, \_\_drv\_arg(p3, \_\_drv\_in(\_\_notnull) )

Starting from the end of the second line of this example—the inner condition:

* **\_\_drv\_arg(***p3*, **\_\_drv\_in(\_\_notnull) )** specifies the annotations to apply to *p3.*
* **\_\_drv\_in**(**\_\_notnull**) is the annotation list, which describes an input argument that must not be NULL.

The enclosing **\_\_drv\_arg(***p3* ... **)** annotation associates *p3* with this annotation list.

* The enclosing **\_\_drv\_when(***p2***, ... )** annotation specifies that the annotations should be applied to *p3* only when *p2* is true.

Starting from the end of the first line of this example—the outer condition:

* **\_\_drv\_arg(***p*, **\_\_drv\_in(\_\_notnull) )** specifies the annotations to apply to *p*.
* **\_\_drv\_in**(**\_\_notnull**) is the annotation list. The annotations in this list describe an input argument that must not be NULL.

The enclosing **\_\_drv\_arg(***p* ...**)** annotation associates *p* with this annotation list.

* The enclosing **\_\_drv\_when(***Wait* ...**)** annotation specifies that the annotations should be applied to *p* only when both *Wait* and *p2* are true (that is, *Wait* && *p2*).

### Grammar for Conditional Expressions

The grammar for conditional expressions in annotations is a subset of the grammar that the C programming language supports.

#### Conditional Grammar Supported

The grammar for conditional expressions supports the following:

* The **+**, **-**, **\***, **/**, unary **-**, **<<**, **>>**, **<**, **<=**, **==**, **!=**, **>**, **>=**, **!**, **&&**, **||**, **~**, **&**, **|**, **?:**, **(**, and **)** operators.

PREfast supports these operators with the usual operator precedence, along with the **$** functions that are discussed in “Special Functions in Conditions” later in this paper.

* All forms of integer constants.
* Parameter names and the unary \* operator on parameter names.

#### Evaluation of Expressions

When PREfast evaluates a condition expression, it:

* Performs constant folding.

That is, when PRE*f*ast can determine the value of a parameter, PRE*f*ast uses that value. As in C, PRE*f*ast converts expressions in a Boolean context to zero or nonzero. Expressions can use #defined constant names.

* Carries out all calculations as wide signed integers.

PRE*f*ast does not rely on implied truncation to a narrower word size.

* Evaluates the expressions in each **\_\_drv\_when** annotation independently.

For a given function, all, some, or none of the conditional expressions might be true, depending on context. If two conditions are meant to be mutually exclusive, they should be coded so that they actually are.

#### Unsupported Expressions

PREfast does not support the following:

* Enumerated constants
* Consts (in C++)
* **sizeof** operator in conditions

Depending on the circumstances, it is often possible to work around these limitations by combining existing annotations.

### Special Functions in Conditions

The following functions can be called from within a condition. PRE*f*ast executes these functions when it analyzes code with annotations that contain them:

**strstr$(**p1**,** p2**)  
macroDefined$(**p1**)  
isFunctionClass$(**name**)**

#### strstr$(*p1*, *p2*)

This function computes the same value as the **strstr** C function, except that the result is the offset in the p1 string where the first instance of p2 is found. If p2 is not found in p1, **strstr$** returns -1. The p1 and p1 arguments can be either parameter expressions or literals.

The usual usage is to determine—to the extent that PREfast is capable—whether p1 contains the p2 constant. For example, the following function call determines whether path contains any backslash characters:

strstr$(path, "\\")>=0

The following function call determines whether path begins with “C:”:

strstr$(path, "C:")==0

#### macroDefined$(*p1*)

This function determines whether the p1 string is a symbol that is defined with **#define**. p1 should be a quoted string. If p1 is not a defined symbol, the function returns 0. If p1 is a defined symbol, the function returns 1. You can use defined symbols directly in **\_\_drv\_when** conditions, but **macroDefined$** is the only way for **\_\_drv\_when** to determine whether the symbol is defined.

**macroDefined$** is functionally equivalent to the **\_\_drv\_defined** macro, which has the same syntax as **macroDefined$** but handles quoted strings more reliably. Like **macroDefined$**, **\_\_drv\_defined(**XYZ**)** returns a Boolean value that indicates whether XYZ is a defined symbol in the program.

You can often express an annotation without using either **\_\_drv\_defined** or **macroDefined$**, simply by assuming that the macro is expanded by the compiler. If that does not work, use **\_\_drv\_defined**.

#### isFunctionClass$(*name*)

This function determines whether the annotated function belongs to the class that is identified by name. See “Special Functions in Conditions” later in this paper for more information about **isFunctionClass$**.

## Function Result Annotations

Many functions have a limited set of possible results for an output parameter or the function’s return value. Informing PREfast of this often makes its analysis much more accurate because PREfast can avoid following impossible paths. For example, a function that returns a BOOLEAN is returning an integer, which might be any value. The BOOLEAN type is simply a convention that indicates that the value should be only TRUE or FALSE.

You can use the **\_\_drv\_valueIs(**list**)** annotation to indicate a set of possible result values for a function. The result for the output parameter or function return value must be one of the values in list, which consists of a series of partial expressions in the form <relational operator><constant>, separated by semicolons.

For example, consider the following code:

BOOLEAN b = boolfunc(...);

if (b == TRUE) {

// do something

}

...

if (b == FALSE) {

// do something else

}

PREfast interprets b as an integer, but it cannot determine that b can have only two possible values. Therefore, when analyzing this function, PREfast might simulate situations in which both statements are skipped—for example, if PREfast assumed that b was 3. This kind of situation can lead to both false positives and false negatives. For this example, the b parameter can be annotated with **\_\_drv\_valueIs(**==0; ==1**),** which limits b to FALSE or TRUE.

Functions that return NTSTATUS are typically annotated with **\_\_drv\_valueIs(**<0; ==0**)**, which indicates the value range for failure and success.

A few functions might be better annotated with **\_\_drv\_valueIs(**<0;>=0**)**. These functions can return a strictly positive value for NTSTATUS, which means that the set of possible results is all integers—those less than zero and those greater than or equal to zero. This annotation, which is effectively no annotation at all, is included here for completeness.

You can combine conditions with the **\_\_drv\_valueIs** annotation to limit the result values to those that are made possible by the input parameters. For example, the annotations that are applied to the return value for **ExAcquireResourceExclusiveLite** indicate that if Wait is 0, the function can return either 0 or 1, but if Wait is nonzero, the function can return only 1, as shown in the following example:

\_\_drv\_when(!Wait, \_\_drv\_valueIs(==0; ==1))

\_\_drv\_when(Wait, \_\_drv\_valueIs(==1))

An alternative annotation would be as follows:

\_\_drv\_when(!Wait, \_\_drv\_valueIs(==0;==1) \_\_checkReturn)

This annotation indicates that if Wait is false, the function can return 0 or 1 and the result must be checked. If Wait is nonzero, it does not matter what the function returns because the return value is not required to be checked.

## Type-Matching Annotations

The C and C++ languages permit some mixing of integer types. In particular, it is acceptable to pass an enumerated type where a generic integer type is expected. However, for a number of functions, it is easy to pass the wrong enumerated type.

You can use the **\_\_drv\_strictTypeMatch(**mode**)** and **\_\_drv\_strictType(**typename, mode**)** annotations to ensure that PRE*f*ast checks whether a function is called with exactly the right type:

* For **\_\_drv\_strictTypeMatch**, the actual parameter must be the same type as the formal parameter, within the limits set by mode.
* For **\_\_drv\_strictType**, the parameter must be the type that is specified by typename, within the limits set by mode.

The mode argument can be one of the following:

\_\_drv\_typeConst

The parameter takes a single simple operand that must match exactly. This annotation is typically used where the value is a single constant.

\_\_drv\_typeCond

The annotated expression can use the **?:** operator to select between single operands. Parentheses are also permitted. This annotation is typically used to switch dynamically among a few constant arguments.

\_\_drv\_typeBitset

The parameter can take expressions that involve only operands of that type. This annotation is typically used for bitsets, but is not limited to bit operations.

\_\_drv\_typeExpr

The parameter can take the same operands as a **\_\_drv\_typeBitset** annotation, plus literal constants. For example, the POOL\_TYPE parameter to **ExAllocatePool** uses this annotation.

The **\_\_drv\_strictType** annotation is useful for functions that take integer parameters whose value should be limited to members of a particular enumerated type. If either a variable or a constant might be passed to the function, it is helpful to specify both the **typedef** name for variables and an enumerated type for constants. This can be done by giving the **typedef** name and the enumerated type name, separated by a slash, as shown in the following example:

\_\_drv\_strictType(KPROCESSOR\_MODE/enum \_MODE, \_\_drv\_typeCond)

The **KeWaitForMultipleObjects** function provides several examples of type annotations. This function has three parameters that are different enumerated types. However, the parameters are easily confused, and the C compiler does not check for correctness. With appropriate annotations, PREfast can check that the parameters are of the correct type. See the notes after Listing 8 for an explanation.

Listing 8 Example of type annotations applied to KeWaitForMultipleObjects

NTSTATUS KeWaitForMultipleObjects(

\_\_in ULONG Count,

\_\_in PVOID Object[],

\_\_in

\_\_drv\_strictTypeMatch(\_\_drv\_typeConst) // 1

WAIT\_TYPE WaitType,

\_\_in

\_\_drv\_strictTypeMatch(\_\_drv\_typeConst) // 2

KWAIT\_REASON WaitReason,

\_\_in

\_\_drv\_strictType(KPROCESSOR\_MODE/enum \_MODE,

\_\_drv\_typeCond) // 3

KPROCESSOR\_MODE WaitMode,

\_\_in BOOLEAN Alertable,

\_\_in\_opt PLARGE\_INTEGER Timeout,

\_\_in\_opt PKWAIT\_BLOCK WaitBlockArray);

In the example in Listing 8:

1. The **\_\_drv\_strictTypeMatch(\_\_drv\_typeConst)** annotation indicates that WaitType must be a member of the **WAIT\_TYPE** enumerated type.

2. The **\_\_drv\_strictTypeMatch(\_\_drv\_typeConst)** annotation indicates that WaitReason must be a member of the **KWAIT\_REASON** enumerated type.

3. The **\_\_drv\_strictType(**KPROCESSOR\_MODE/enum \_MODE, **\_\_drv\_typeCond)** annotation indicates that WaitMode must be either a variable of a **KPROCESSOR\_MODE** type or a member of the **\_MODE** enumerated type and can be passed as an expression that uses the **?:** operator.

Some additional considerations apply to WaitMode:

* Constants of **enum \_MODE** type are semantically reasonable, and it is semantically reasonable to want to select between them with the **?:** operator. However, it is not reasonable to allow arithmetic on those constants. **\_\_drv\_typeCond** allows use of the **?:** operator but does not allow the use of any other operators.
* **KPROCESSOR\_MODE** is defined to be a **char**, and enumeration values are defined by C to be the same as **int**. Thus, there cannot be a symbolic name for a constant of type **KPROCESSOR\_MODE**. Instead, **enum \_MODE** is the closest match.

## Pointer Annotations

Certain driver functions take PVOID as a parameter type, so they can accept any one of a number of different types. It is common to mistakenly pass the wrong type, for example, to pass &pStruct instead of pStruct as intended. For any type but PVOID, the compiler would diagnose this as an error. For the PVOID type, the compiler cannot detect the error because it has no type information to check.

The **\_\_drv\_notPointer** and **\_\_drv\_isObjectPointer** annotations enable PREfast to diagnose errors when a parameter must not be a pointer:

* **\_\_drv\_notPointer** indicates that the parameter must be a struct or scalar value.

For example, a **\_\_drv\_notPointer** annotation specifies that (PVOID) 1 is acceptable but (PVOID) &*var* is not acceptable.

The **\_\_drv\_notPointer** annotation typically appears as **\_\_drv\_deref(\_\_drv\_notPointer)**, which indicates that the parameter should be a pointer to a nonpointer object, usually a structure, and not a pointer to a pointer. A common error that this annotation helps PRE*f*ast to detect is passing &*pStruct* when *pStruct* was intended because you have forgotten you were passed a pointer to a structure rather than the structure.

* **\_\_drv\_isObjectPointer** indicates that the parameter must be a pointer to a nonpointer object.

This annotation is a simpler equivalent to **\_\_drv\_deref(\_\_drv\_notPointer)**.

For example, in Listing 9, the annotation on the Object parameter of **KeWaitForSingleObject** causes PREfast to issue a warning if the function is called with Object as a pointer to a pointer.

Listing 9 Example of annotations for a function that must be called with a pointer to a nonpointer object

NTSTATUS

KeWaitForSingleObject(

\_\_in \_\_drv\_isObjectPointer PVOID Object,

\_\_in

\_\_drv\_strictTypeMatch(\_\_drv\_typeConst)

KWAIT\_REASON WaitReason,

\_\_in

\_\_drv\_strictType(KPROCESSOR\_MODE/enum \_MODE,

\_\_drv\_typeCond)

KPROCESSOR\_MODE WaitMode,

\_\_in BOOLEAN Alertable,

\_\_in\_opt PLARGE\_INTEGER Timeout

);

## Constant and Non-Constant Parameter Annotations

You can use the **\_\_drv\_constant** and **\_\_drv\_nonConstant** annotations for functions that either require or prohibit the use of literal constants as parameters.

For example, a device driver should not assume that any port address is a constant. Therefore, the various READ\_PORT\_Xxx functions should all be annotated with **\_\_drv\_nonConstant** for the address of the port being read. For the occasional exception to this, either ignore or suppress the PREfast warning.

Another example is the Wait parameter to **KeSetEvent**. Although theoretically that parameter might be a variable, the requirements for what must be done before and after the call make it difficult and possibly confusing to call **KeSetEvent** with a variable. Therefore, Wait is annotated with **\_\_drv\_constant**. Again, the occasional exception can be handled by ignoring or suppressing the PREfast warning.

## Format String Annotations

The **\_\_drv\_formatString(**kind**)** annotation indicates that the annotated parameter is a format string.Kind can be **printf** or **scanf**, which refers to the type of format string that is allowed, not the specific function being called. That is, the format string follows the rules for either **printf** or **scanf**. **\_\_drv\_formatString** can be used to annotate any function that is similar to **printf** or **scanf**.

This annotation causes PREfast to check that the argument list matches the format string and that potentially dangerous combinations are avoided.

The following annotation indicates that format is a format string for **printf**:

int \_snprintf(

\_\_out\_ecount(count) \_\_possibly\_notnullterminated LPSTR buffer,

\_\_in size\_t count,

\_\_in \_\_drv\_in(\_\_drv\_formatString(printf)) LPCSTR \*format

[, argument] ...

);

## Diagnostic Annotations

Occasionally, a particular combination of parameters is either dangerous or can be done better in some other way. PREfast can check for many such usage errors.

You can use the **\_\_drv\_preferredFunction(**name**,** reason**)** and **\_\_drv\_reportError(**string**)** annotations to generate error messages. These annotations are typically used in combination with the **\_\_drv\_when** conditional annotation. You should use these annotations for recommendations and for annotating specific usages to avoid.

If a function must never be used under any circumstances, you should mark it with **#pragma \_\_deprecated** or **\_\_declspec(deprecated)** so that the compiler can generate a compile-time error.

### Annotations for Preferred Functions

You can use the **\_\_drv\_preferredFunction(**name**,** reason**)** annotation to generate error messages. Name can be anything, but is usually the name of a preferred function, whereas reason is an additional explanation of why that function is preferred.

For example, consider two hypothetical functions, GetResource and TryToGetResource. GetResource takes a Wait parameter. When Wait is TRUE (that is, nonzero), a call should wait until the resource is acquired. When Wait is FALSE, GetResource can still be used to acquire the resource, but TryToGetResource is more efficient. The following annotations would cause PREfast to flag this circumstance:

\_\_drv\_when(!Wait,

\_\_drv\_preferredFunction(TryToGetResource,

"When calling GetResource with Wait==FALSE,

"TryToGetResource performs better."))

### Annotations for Error Messages

The **\_\_drv\_reportError(**string**)** annotation causes PREfast to generate a warning that it has encountered the error that the annotation describes. The **\_\_drv\_reportError** annotation is similar to **\_\_drv\_preferredFunction** except that it generates a warning to fix the problem that the *string* parameter describes.

For example, you can use **\_\_drv\_reportError** for unacceptable usage such as an attempt to use a must-succeed allocation from the **ExAllocatePool** family of functions, as in the following example:

\_\_drv\_when(PoolType&0x1f==2 || PoolType&0x1f==6,

\_drv\_reportError("Must succeed pool allocations are"

"forbidden. Allocation failures cause a system crash"))

## Annotations for Functions in \_\_try Statements

Certain functions must always be called from inside a structured exception handling (SEH) **\_\_try** statement, whereas other functions must never be called from inside a **\_\_try** statement. The **\_\_drv\_inTry** and **\_\_drv\_notInTry** annotations cause PREfast to check for proper usage within a function:

* **\_\_drv\_inTry** indicates that the function must be called from inside a **\_\_try** statement.
* **\_\_drv\_notInTry** indicates that the function must not be called from inside a **\_\_try** statement.

For example, use **\_\_drv\_inTry** with **ProbeForRead** and **ProbeForWrite** so that failed attempts to access memory are caught by a **\_\_try** statement.

## Memory Annotations

Drivers often use special-purpose functions to allocate and free memory. Drivers also often pass memory out of a function in a way that causes the memory to be aliased, where it will be dealt with later. Annotations can help PREfast more accurately detect leaks and other problems with allocations of both memory and nonmemory resources such as spin locks.

You can use the memory annotations described in the following sections to help PREfast more accurately check functions that allocate memory.

### Annotations for Allocating and Freeing Memory

The **\_\_drv\_allocatesMem(**type**)** annotation indicates that the output value is allocated, either through a parameter or through the function result. The type parameter indicates the type of memory allocator used. This parameter is advisory—PREfast does not check it. However, this parameter might be checked by a future version of PREfast. The following are recommended values for the type parameter:

* For **malloc** and **free**, type should be **mem**.
* For the **new** operator, type should be **object**.

If a function that allocates memory indicates failure by returning NULL, you should also apply the **\_\_checkReturn** annotation to the function.

**\_\_drv\_freesMem(**type**)** indicates that the memory that is passed as an input parameter is freed. In post-state, PRE*f*ast assumes that the annotated parameter is in an uninitialized state and, until the parameter is changed, PRE*f*ast treats further access through the actual parameter as an access to an uninitialized variable. Type should match the type used in **\_\_drv\_allocatesMem**.

### Annotations for Aliasing Memory

You can apply the **\_\_drv\_aliasesMem** annotation to input parameters—including **\_\_in** and **\_\_out** parameters—to indicate that the called function saves the value of the parameter in some location where it will be found later and presumably freed.

In general, PREfast cannot confirm whether memory that is aliased is actually freed. The memory might continue to be accessed after a call to a function with this annotation. PREfast tries to identify when memory is aliased in several different ways, but it cannot determine whether memory is aliased for called functions without this annotation.

**\_\_drv\_aliasesMem** helps to suppress false “possibly leaking memory” warnings from PREfast. It does not take a type parameter because it operates in the same way on all kinds of memory.

The **\_\_drv\_freesMem** and **\_\_drv\_aliasesMem** annotations are mutually exclusive. **\_\_drv\_freesMem** indicates that the memory is discarded (that is, the memory is no longer accessible), and PREfast enforces this by invalidating the variable that contains the pointer to the freed memory. **\_\_drv\_aliasesMem** indicates simply that there is no longer a risk of that memory leaking, but the memory continues to exist and can be accessed subsequently.

### Memory Annotation Examples

The examples in this section show the use of memory annotations.

The example in Listing 10 shows some of the annotations that are applied to the **ExAllocatePool** and **ExFreePool** functions, which are the classic examples of functions that allocate and free memory. These functions have additional parameter checks that are not shown in this listing.

Listing 10 Example of annotations for functions that allocate and free memory

\_\_checkReturn

\_\_drv\_allocatesMem(Pool)

\_\_drv\_when(PoolType&0x1f==2 || PoolType&0x1f==6,

\_\_drv\_reportError("Must succeed pool allocations are"

"forbidden. Allocation failures cause a system crash"))

\_\_bcount(NumberOfBytes)

PVOID

ExAllocatePoolWithTag(

\_\_in \_\_drv\_strictTypeMatch(\_\_drv\_typeExpr)

POOL\_TYPE PoolType,

\_\_in SIZE\_T NumberOfBytes,

\_\_in ULONG Tag

);

NTKERNELAPI

VOID

ExFreePoolWithTag(

\_\_in \_\_drv\_in(\_\_drv\_freesMem(Pool))

PVOID P,

\_\_in ULONG Tag

);

In the example in Listing 11, the **InsertHeadList** function takes and holds Entry. That is, it aliases the memory that is occupied by Entry. The **\_\_drv\_aliasesMem** annotation in this example suppresses a potential memory leak warning from PREfast, but leaves Entry accessible.

Listing 11 Example of annotations for a function that aliases memory

VOID

InsertHeadList(

\_\_in PLIST\_ENTRY ListHead,

\_\_in \_\_drv\_in(\_\_drv\_aliasesMem) PLIST\_ENTRY Entry

);

## Nonmemory Resource Annotations

A number of nonmemory resource types, such as critical regions and spin locks, can leak. PREfast can detect leaks of nonmemory resources just as it can detect memory leaks. However, PRE*f*ast cannot apply memory semantics to nonmemory objects because semantic differences between the two types of objects would lead to various kinds of incorrect analysis. For example, there is no concept of aliasing for nonmemory resources. In general, if a resource must be aliased, it is better modeled as memory.

The following sections describe annotations related to nonmemory resource types, as summarized in Table 5.

Table 5 Nonmemory Resources Annotations

| **Annotation** | **Usage** |
| --- | --- |
| **\_\_drv\_acquiresResource(**kind**) \_\_drv\_releasesResource(**kind**)** | Annotations for acquisition and release of nonmemory resources |
| **\_\_drv\_acquiresResourceGlobal(**kind**,** param**) \_\_drv\_releasesResourceGlobal(**kind**,** param**)** | Annotations for global nonmemory resources |
| **\_\_drv\_acquiresCriticalRegion \_\_drv\_releasesCriticalRegion \_\_drv\_acquiresCancelSpinLock \_\_drv\_releasesCancelSpinLock** | Annotations for the critical region and cancel spin lock |
| **\_\_drv\_mustHold(**kind**) \_\_drv\_neverHold(**kind**) \_\_drv\_mustHoldGlobal(**kind**,** param**) \_\_drv\_neverHoldGlobal(**kind**,** param**) \_\_drv\_mustHoldCriticalRegion \_\_drv\_neverHoldCriticalRegion \_\_drv\_mustHoldCancelSpinLock \_\_drv\_neverHoldCancelSpinLock** | Annotations for holding and not holding nonmemory resources |
| **\_\_drv\_acquiresExclusiveResource(**kind**) \_\_drv\_releasesExclusiveResource(**kind**) \_\_drv\_acquiresExclusiveResourceGlobal(**kind**,** param**) \_\_drv\_releasesExclusiveResourceGlobal(**kind**,** param**)** | Composite annotations for resources |

### Annotations for Acquisition and Release of Nonmemory Resources

You can use the **\_\_drv\_acquiresResource(**kind**)** and **\_\_drv\_releasesResource(**kind**)** annotations to indicate the acquisition and release of nonmemory resources in a function parameter.

For these annotations, the kind parameter indicates the kind of resource. The value of kind in acquisitions and releases of the resource must match. The *kind* parameter can be any name. A few names already in use include **SpinLock**, **QueuedSpinLock**, **InterruptSpinLock**, **CancelSpinLock**, **Resource**, **ResourceLite**, **FloatState**, **EngFloatState**, **FastMutex**, **UnsafeFastMutex**, and **Critical-Region**. Unrelated usages of the same *kind* value do not conflict.

For example, in Listing 12, the functions acquire and release a resource that is named **SpinLock**. The value is held in a variable named SpinLock. The symbols are in different name spaces, so they do not conflict.

Listing 12 Example of annotations for functions that acquire and release a resource

VOID

KeAcquireSpinLock(

\_\_inout

\_\_drv\_deref(\_\_drv\_acquiresResource(SpinLock))

PKSPIN\_LOCK SpinLock,

\_\_out PKIRQL OldIrql

);

VOID

KeReleaseSpinLock(

\_\_inout

\_\_drv\_deref(\_\_drv\_releasesResource(SpinLock))

PKSPIN\_LOCK SpinLock,

\_\_in KIRQL NewIrql

);

### Annotations for Global Nonmemory Resources

Operations on nonmemory resources must consider that the resource is often not held “in” some variable, but consists of global state information that is accessed by context or selected by some kind of identifier.

You can use the **\_\_drv\_acquiresResourceGlobal(**kind**,** param**)** and **\_\_drv\_releasesResourceGlobal(**kind**,** param**)** annotations to indicate acquisition and release of this kind of resource for a function.

These annotations apply to the function as a whole, rather than to an individual function parameter. The kind and param parameters specify the kind of resource and the instance of the resource, respectively.

#### Annotations for Naming a Resource

You can use the **\_\_drv\_acquiresResourceGlobal** annotation to create a name for annotating a nonmemory resource that is not held in a variable. The annotation takes the following two parameters:

kind

A known string constant (that is, the resource class name) for the kind of object (that is, the resource) to be tracked.

param

The specific instance to be tracked. The function call typically provides the name of the specific instance (that is, *param).*

#### Annotations for Identifying an Instance of a Resource

If a resource is not held in a variable, another variable is often used to identify the instance of the resource that is wanted. Many of the resource annotation macros take a parameter that identifies the instance that is being allocated. This parameter is similar to the *size* parameter to the **\_ecount(***size***)** annotation modifier in that the annotation does not apply to the parameter itself. Instead, the value of the parameter modifies the annotation of some other parameter—or the function as a whole.

For example, consider a resource that has no associated data, such as “the right to use I/O register n.” In principle, if some function acquires that resource but does not release it, the resource can leak. For PREfast to detect a leak of the resource, it must be able to identify the instance of the resource that is being requested or is owned by a function. That is, for the I/O register, it is necessary to annotate “the right to use I/O register n” as a resource.

In terms of the function that acquires the right to use that resource, n is a parameter to the acquiring function, and it is necessary to create an object that represents “the right to use I/O register *n*” to PREfast. However, no object in the function that is being analyzed holds “the right to use I/O register n,” so there is nothing PREfast can overload during simulation to hold that concept.

The example in Listing 13 shows how to map the identification of a specific instance to a class name by using **\_\_drv\_acquiresResourceGlobal**. In this example, IORegister is the class name and the regnum parameter identifies the instance of the resource.

Listing 13 Example of annotations for a function that acquires a global resource

\_\_checkReturn

\_\_drv\_acquiresResourceGlobal(IORegister, regnum)

NTSTATUS acquireIORegister(int regnum);

The acquireIORegister function puts a name into some private name space that is not accessible to PREfast where the right to use IORegister regnum is held. When the right to use the register is relinquished, the symbol then indicates that the right is no longer held. This serves the same kind of purpose as the pointer to memory returned by **malloc**, without introducing anything into the actual source code of the function that is being analyzed.

Note that **\_\_drv\_acquiresResourceGlobal** does not annotate the regnum parameter in any way. Instead, regnum—or, rather, its value as simulated by PREfast—is a parameter to the annotation, just as the size parameter of **\_bcount(**size**)** is a parameter to the **\_bcount** annotation.

### Annotations for the Critical Region and Cancel Spin Lock

The following annotations can be used to annotate functions that acquire or release the critical region or cancel spin lock:

**\_\_drv\_acquiresCriticalRegion  
\_\_drv\_releasesCriticalRegion  
\_\_drv\_acquiresCancelSpinLock  
\_\_drv\_releasesCancelSpinLock**

Certain resources, such as the critical region and the cancel spin lock, have no programmatic name—they simply exist. Only one instance of these resources can exist at one time. The same concept that was used in **\_\_drv\_acquiresResourceGlobal**, as described in the previous section, can be used to annotate these resources, but the specific instance part of the name (that is, the second parameter) is implied by the macro name. In practice, special-purpose macros are used for this kind of resource.

As a special case for most drivers, the critical region and the cancel spin lock are two global resources that are accessed by context. Neither resource has any name or parameter value. **KeEnterCriticalReqion** has no parameters and no result value—it simply blocks until it succeeds.

You could express this with the more generic annotations, but it is recommended that you use the critical region and cancel spin lock annotations. These annotations also specify that the resource cannot already be held when the acquire operation occurs and the resource must already be held when the release operation occurs. For more information, see “Annotations for Holding and Not Holding Nonmemory Resources” later in this paper.

The example in Listing 14 indicates that the function is acquiring the critical region. This annotation is the preferred way of expressing **\_\_drv\_acquiresResourceGlobal(**CriticalRegion**,** ""**)**.

Listing 14 Example of annotations for a function that acquires the critical region

\_\_drv\_acquiresCriticalRegion

VOID

KeEnterCriticalRegion(

);

You can use conditional annotations to indicate that acquisition of a resource might fail, as described in “Conditional Annotations” earlier in this paper. For nonmemory resources, the indication of success or failure is usually separated from the resource. For example, as shown in Listing 15, it is only when **KeTryToAcquireSpinLockAtDpcLevel** returns TRUE that the spin lock is acquired.

Listing 15 Example of annotations for a function that might fail to acquire a resource

NTKERNELAPI

BOOLEAN

\_\_drv\_valueIs(==0;==1)

FASTCALL

KeTryToAcquireSpinLockAtDpcLevel (

\_\_inout

\_\_drv\_when(return==1,

\_\_drv\_deref(\_\_drv\_acquiresResource(SpinLock)))

PKSPIN\_LOCK SpinLock

);

### Annotations for Holding and Not Holding Nonmemory Resources

The following annotations can be used to specify resources that either should or should not be held by a function:

**\_\_drv\_mustHold(**kind**)  
\_\_drv\_neverHold(**kind**)  
\_\_drv\_mustHoldGlobal(**kind**,** param**)  
\_\_drv\_neverHoldGlobal(**kind**,** param**)  
\_\_drv\_mustHoldCriticalRegion  
\_\_drv\_neverHoldCriticalRegion  
\_\_drv\_mustHoldCancelSpinLock  
\_\_drv\_neverHoldCancelSpinLock**

A number of functions require that a resource be held—or not be held—before the function is called. The simplest examples are the special functions for critical regions and the cancel spin lock, in which the resource is about to be acquired or released. However, a number of other functions do not operate correctly unless the proper resources are held—or not held. For example, **ExAcquireResourceExclusiveLite** must be called with the critical region held.

#### Annotations for Holding Nonmemory Resources in a Function

The **\_\_drv\_mustHold(**kind**)** and **\_\_drv\_neverHold(**kind**)** annotationscan be applied to a function as a whole, rather than to a particular parameter. These annotations indicate that the function must hold at least one resource of kind or no resource of kind, respectively.

For example, **IoCompleteRequest** cannot be called with any spin lock held, so **\_\_drv\_neverHold(SpinLock)** is used to indicate that no spin lock can be held when that function is called. The **\_\_drv\_mustHold(Memory)** and **\_\_drv\_neverHold(Memory)** annotationsare special cases that indicate that the object being held is a memory object, from a function such as **malloc** or **new**.

#### Annotations for Holding a Global Nonmemory Resource

The **\_\_drv\_mustHoldGlobal(**kind**,** param**)** and **\_\_drv\_neverHoldGlobal(**kind**,** param**)** annotationsare used to indicate that a global resource must be held or not held. *Kind* and *param* have the same meaning as described earlier for acquiring and releasing global resources.

#### Annotations for Holding the Critical Region or Cancel Spin Lock

The following annotations apply to the critical region and cancel spin lock, respectively:

**\_\_drv\_mustHoldCriticalRegion  
\_\_drv\_neverHoldCriticalRegion  
\_\_drv\_mustHoldCancelSpinLock  
\_\_drv\_neverHoldCancelSpinLock**

The examples in Listing 16, 17, and 18 show the use of these annotations.

Listing 16 Example of annotations for a function that must hold the critical region

\_\_drv\_mustHoldCriticalRegion

BOOLEAN

ExAcquireResourceExclusiveLite(

\_\_in PERESOURCE Resource,

\_\_in BOOLEAN Wait

);

Listing 17 Example of annotations for a function that must never hold a spin lock

\_\_drv\_neverHold(SpinLock)

VOID

IoCompleteRequest(

\_\_in PIRP Irp,

\_\_in CCHAR PriorityBoost

);

The example in Listing 18 includes the annotations that are necessary to prevent taking or releasing a resource more than once, which was not shown in earlier examples.

Listing 18 Example of annotations for functions that must not take or release a resource more than once

VOID

KeAcquireSpinLock(

\_\_inout

\_\_deref(\_\_drv\_acquiresResource(SpinLock)

\_\_drv\_neverHold(SpinLock))

PKSPIN\_LOCK SpinLock,

\_\_out PKIRQL OldIrql

);

VOID

KeReleaseSpinLock(

\_\_inout

\_\_deref(\_\_drv\_releasesResource(SpinLock)

\_\_drv\_mustHold(SpinLock))

PKSPIN\_LOCK SpinLock,

\_\_in KIRQL NewIrql

);

### Composite Annotations for Resources

The following annotations can be used to annotate functions that allocate resources:

**\_\_drv\_acquiresExclusiveResource(**kind**)  
\_\_drv\_releasesExclusiveResource(**kind**)  
\_\_drv\_acquiresExclusiveResourceGlobal(**kind**,** param**)  
\_\_drv\_releasesExclusiveResourceGlobal(**kind**,** param**)**

These composite annotations are used to annotate resource allocation functions which are similar to spin lock wrapper functions in which the resource can have only one owner:

* The “acquires” forms combine the **\_\_drv\_neverHold** and **\_\_drv\_acquiresResource** annotations.
* The“releases” forms combine the **\_\_drv\_mustHold** and **\_\_drv\_releasesResource** annotations.

The following list summarizes the behavior that these annotations specify:

\_\_drv\_acquiresExclusiveResource(*kind*)

The *kind* resource is being acquired and cannot already be held.

\_\_drv\_releasesExclusiveResource(*kind*)

The *kind* resource is being released and must already be held.

\_\_drv\_acquiresExclusiveResourceGlobal(*kind*, *param*)

The *param* instance of the *kind* global resource is being acquired and cannot already be held.

\_\_drv\_releasesExclusiveResourceGlobal(*kind*, *param*)

The *param* instance of the *kind* global resource is being released and must already be held.

For example, the annotation in Listing 19 indicates that the function is acquiring MySpinLock, which must not already be held.

Listing 19 Example of a composite annotation for acquiring a spin lock

VOID

GetMySpinLock(

\_\_inout

\_\_deref(\_\_drv\_acquiresExclusiveResource(MySpinLock))

PKSPIN\_LOCK SpinLock,

\_\_out PKIRQL OldIrql

);

## Function Type Class Annotations

Drivers commonly define functions that the system calls by using a function pointer that the driver passes to the system. A driver’s AddDevice, StartIo, and Cancel functions are examples of this kind of callback function. Many of the annotations that are discussed in this paper apply to these callback functions.

Properly annotated callback functions significantly help PREfast to check for proper usage. This can be done by declaring functions to be members of a function type class.

Most common system-defined callback types have a function class. System-defined function type classes for WDM drivers are defined in %wdk%\inc\ddk\Wdm.h. PRE*f*ast can detect function type class annotations for WDM drivers. Class annotations for KMDF drivers are defined in %wdk%\inc\wdf\kmdf\wdfroletypes.h. The ability to detect these role types is enabled in Static Driver Verifier and planned for a future version of PRE*f*ast.

You can define your own function type classes by using typedef declarations, as described in “Annotations on Function Typedef Declarations” earlier in this paper.

### Annotations for Identifying the Function Type Class of a Function

The **\_\_drv\_functionClass(**name**)** annotation indicates that the function or function **typedef** declaration is a member of the named class of functions that is represented by name. This annotation is most useful when applied to both the function and the function pointer type. It can be most easily applied through the use of function **typedef** declarations.

The **\_\_drv\_functionClass** annotation propagates to the corresponding function pointer **typedef** declaration and causes PREfast to check that function assignments to and from function pointers use the same function class.

PREfast issues a warning in the following cases:

* If a function does not have a function class.
* If a function of the wrong function class is assigned to a function pointer that does have a function pointer class.

To fix this warning, add the appropriate function **typedef** declaration to your code. The PREfast error message text usually lists the function **typedef** declaration that you should use.

For example, in Listing 20, the **\_\_drv\_functionClass** annotation indicates that this is the function **typedef** declaration for the **DRIVER\_INITIALIZE** function class. The **typedef** name and the class name are in different name spaces. The pointer **PDRIVER\_INITIALIZE** is of the class **DRIVER\_INITIALIZE** because it is derived from the **DRIVER\_INITIALIZE** **typedef** declaration.

Listing 20 Example of annotation for a function of a specific function type class

typedef

\_\_drv\_functionClass(DRIVER\_INITIALIZE)

NTSTATUS

DRIVER\_INITIALIZE (

\_\_in struct \_DRIVER\_OBJECT \*DriverObject,

\_\_in PUNICODE\_STRING RegistryPath

);

typedef DRIVER\_INITIALIZE \*PDRIVER\_INITIALIZE;

### Annotations for Checking a Function Type Class in a Conditional Expression

A few functions have annotations that apply only when the function is—or is not—called from a particular class of function that is specified by a **\_\_drv\_functionClass** annotation. This case can be annotated by adding a call to **isFunctionClass$(**name**)** as part of a conditional expression in a **\_\_drv\_when** annotation.

The **isFunctionClass$(**name**)** function determines whether the function belongs to the class that is identified by name:

* If the function does not belong to the class, **isFunctionClass$** returns 0.
* If the function does belong to the class, **isFunctionClass$** returns 1.

For example, **isFunctionClass$("DRIVER\_INITIALIZE")** determines whether the function is a driver initialization routine type. See “Special Functions in Conditions” earlier in this paper for information about other functions that you can use in conditions.

**IoCreateDevice** is an example of a special case that applies only to legacy drivers. In a legacy driver, the system allows **IoCreateDevice** to be called from within the **DRIVER\_INITIALIZE** function and does not require it to explicitly keep track of the resulting device object. The system puts the device object in a location that the driver’s *Unload* function can find, but PREfast cannot detect this and issues a warning.

The annotations in the following example prevent noise that is related to this special case when you run PREfast on a legacy driver:

\_\_drv\_when(!isFunctionClass$("DRIVER\_INITIALIZE")

&& return == 0,\_\_deref(\_\_drv\_allocatesMem(DeviceObject))

The return==0 clause is an example of a check for successful execution.

## Floating-Point Annotations

For some processor families, particularly x86 processors, using floating point from within a kernel code must be done only within the scope of functions that save and restore floating-point state. Violations of this rule can be difficult to find because they cause problems only sporadically at runtime. With the proper use of annotations, PREfast can detect the use of floating point in kernel-mode code and report an error if floating-point state is not properly protected. Floating-point rules are checked only for kernel-mode code.

You can apply the **\_\_drv\_floatSaved** and **\_\_drv\_floatRestored** annotations to a function parameter to indicate what it does with floating-point state. These annotations are already applied to **KeSaveFloatingPoint** state and **KeRestoreFloatingPointState**, along with annotations for acquiring and releasing resources to prevent leaks. The similar **Eng**Xxx functions are also annotated in this way. However, functions that wrap these functions should also use these annotations.

When PREfast discovers an apparently unprotected use of floating point, it issues a warning. If the entire function is called safely by some calling function, you can annotate the function with **\_\_drv\_floatUsed**, which suppresses the warning and also causes PREfast to check that the caller is safely using the function. Additional levels of **\_\_drv\_floatUsed** can be added as required. PREfast automatically provides the **\_\_drv\_floatUsed** annotation when either the function result or one of the function’s parameters is a floating-point type, but you might find it useful to apply the annotation explicitly, as documentation.

For example, in Listing 21 the **\_\_drv\_floatSaved** annotation indicates that the floating-point state is stored in the FloatSave parameter of the **KeSaveFloatingPointState** system function.

Listing 21 Example of annotations that indicate where floating-point state is stored

NTSTATUS

KeSaveFloatingPointState(

\_\_out

\_\_drv\_deref(\_\_drv\_floatSaved)

PKFLOATING\_SAVE FloatSave

);

In the example in Listing 22, the **\_\_drv\_floatUsed** annotation suppresses PREfast warnings about the use of floating-point state by the MyDoesFloatingPoint function. The annotation also causes PREfast to check that any calls to MyDoesFloatingPoint occur in a floating-point-safe context.

Listing 22 Example of annotation for a function that uses floating point

\_\_drv\_floatUsed

void

MyDoesFloatingPoint(arguments);

## IRQL Annotations

All kernel-mode drivers must consider IRQLs. When PRE*f*ast analyzes unannotated driver code, it attempts to infer the range of IRQL at which a function could be running and identify any inconsistencies.

IRQL annotations help PREfast make more accurate inferences about the range of IRQL at which a function should run. For example, a function can be annotated with the maximum IRQL at which it can be called. If that function is called at a higher IRQL, PREfast reports an error. The more annotations that are applied to driver functions, the better PREfast can make those inferences and the more accurately it can find errors.

IRQL parameter annotations interact with each other more than other annotations because the IRQL value is set, reset, saved, and restored by various function calls.

Table 6 lists the annotations that you can use to indicate correct IRQL for a function and its parameters.

Table 6 IRQL Annotations

| **Annotation** | **Description** |
| --- | --- |
| **\_\_drv\_maxIRQL(***value***)** | IRQL value is the maximum IRQL at which the function can be called. |
| **\_\_drv\_minIRQL(***value***)** | IRQL value is the minimum IRQL at which the function can be called. |
| **\_\_drv\_setsIRQL(***value***)** | The function returns at IRQL value. |
| **\_\_drv\_requiresIRQL(***value***)** | The function must be entered at IRQL value. |
| **\_\_drv\_raisesIRQL(***value***)** | The function exits at IRQL value, but it can only be called to raise—not lower—the current IRQL. |
| **\_\_drv\_savesIRQL** | The annotated parameter saves the current IRQL to restore later. |
| **\_\_drv\_restoresIRQL** | The annotated parameter contains an IRQL value from **\_\_drv\_savesIRQL** that is to be restored when the function returns. |
| **\_\_drv\_savesIRQLGlobal(***kind*, *param***)** | The current IRQL is saved into a location that is internal to PRE*f*ast from which the IRQL is to be restored. This annotation is used to annotate a function. The location is identified by *kind* and further refined by *param*. |
| **\_\_drv\_restoresIRQLGlobal(***kind*, *param***)** | The IRQL saved by the function annotated with **\_\_drv\_savesIRQLGlobal** is restored from a location that is internal to PRE*f*ast. |
| **\_\_drv\_minFunctionIRQL(***value***)** | IRQL value is the minimum value to which the function can lower the IRQL. |
| **\_\_drv\_maxFunctionIRQL(***value***)** | IRQL value is the maximum value to which the function can raise the IRQL. |
| **\_\_drv\_sameIRQL** | The annotated function must enter and exit at the same IRQL. The function can change the IRQL, but it must restore the IRQL to its original value before exiting. |
| **\_\_drv\_isCancelIRQL** | The annotated parameter is the IRQL passed in as part of the call to a DRIVER\_CANCEL callback function. This annotation indicates that the function is a utility that is called from Cancel routines and that completes the requirements for DRIVER\_CANCEL functions, including release of the cancel spin lock.  **\_\_drv\_isCancelIRQL** is a composite annotation that consists of **\_\_drv\_useCancelIRQL** plus several other annotations that ensure correct behavior of a DRIVER\_CANCEL callback utility function. |
| **\_\_drv\_useCancelIRQL** | The annotated parameter is the IRQL value that should be restored by a DRIVER\_CANCEL callback function.  \_**\_drv\_useCancelIRQL** by itself is only occasionally useful, for example, if the rest of the obligations described by **\_\_drv\_isCancelIRQL** have already been fulfilled in some other way. |

### Annotations for Specifying Maximum and Minimum IRQL

The **\_\_drv\_maxIRQL(**value**)** and **\_\_drv\_minIRQL(**value**)** annotations simply indicate that the function should not be called from an IRQL that is higher or lower than the specified value. For example, when PREfast sees a sequence of calls that do not change the IRQL, if it finds one with a **\_\_drv\_maxIRQL** value that is lower than a nearby **\_\_drv\_minIRQL**, PRE*f*ast reports a warning on the second call that it encounters. The error might actually occur in the first call—the warning indicates where the other half of the conflict occurred.

If the annotations on a function mention IRQL and do not explicitly apply **\_\_drv\_maxIRQL**, PRE*f*ast implicitly applies the **\_\_drv\_maxIRQL(DISPATCH\_LEVEL)** annotation, which is typically correct with rare exceptions. Implicitly applying this annotation as the default eliminates a lot of annotation clutter and makes the exceptions more visible.

The **\_\_drv\_minIRQL(PASSIVE\_LEVEL)** annotation is always implied because the IRQL can go no lower; consequently, there is no corresponding explicit rule about minimum IRQL. Very few functions have both an upper bound other than **DISPATCH\_LEVEL** and a lower bound other than **PASSIVE\_LEVEL**.

Some functions are called in a context in which the called function cannot safely raise the IRQL above some maximum or, more often, cannot safely lower it below some minimum. The **\_\_drv\_maxFunctionIRQL** and **\_\_drv\_minFunctionIRQL** annotations help PREfast find cases where this occurs unintentionally.

For example, functions of the **DRIVER\_STARTIO** type are annotated with **\_\_drv\_minFunctionIRQL(DISPATCH\_LEVEL)**. This means that, during the execution of a **DRIVER\_STARTIO** function, it is an error to lower the IRQL below **DISPATCH\_LEVEL**. Other annotations indicate that the function must be entered and exited at **DISPATCH\_LEVEL**.

### Annotations for Specifying Explicit IRQL

A **\_\_drv\_setsIRQL** or **\_\_drv\_requiresIRQL** annotation helps PREfast better report an inconsistency that is discovered with \_**\_\_drv\_maxIRQL** or **\_\_drv\_minIRQL** because PRE*f*ast then knows the IRQL.

### Annotations for Raising or Lowering IRQL

The **\_\_drv\_raisesIRQL** annotation is similar to **\_\_drv\_setsIRQL**, but indicates that the function must be used only to raise IRQL and must not be used to lower IRQL, even if the syntax of the function would allow it. **KeRaiseIrql** is an example of a function that should not be used to lower IRQL.

### Annotations for Saving and Restoring IRQL

This section discusses the following annotations for saving and restoring IRQL:

**\_\_drv\_savesIRQL  
\_\_drv\_restoresIRQL  
\_\_drv\_savesIRQLGlobal(**kind**,** param**)  
\_\_drv\_restoresIRQL Global(**kind**,** *param***)**

The **\_\_drv\_savesIRQL** and **\_\_drv\_restoresIRQL** annotations indicate that the current IRQL—whether it is known exactly or only approximately—is saved to or restored from the annotated parameter. The locations that are associated with **\_\_drv\_savesIRQL** and **\_\_drv\_restoresIRQL** are presumed to be some form of integer—that is, any integral type that the compiler allows. PREfast attempts to deal with that value as an integer where possible.

Some functions save and restore the IRQL implicitly. For example, **ExAcquireFastMutex** saves IRQL in an opaque location that is associated with the fast mutex object that the first parameter identifies. The saved IRQL is restored by the corresponding **ExReleaseFastMutex** for that fast mutex object. You can indicate these actions explicitly by using the **\_\_drv\_savesIRQLGlobal** and **\_\_drv\_restoresIRQLGlobal** annotations. The kind and param parameters indicate where the IRQL value is saved, similar to the resource annotations discussed in “Nonmemory Resource Annotations” earlier in this paper. You do not need to precisely specify the location where the value is saved as long as the annotations that save and restore the value are consistent.

### Annotations for Maintaining the Same IRQL

User-defined functions that change IRQL should be annotated either with **\_\_drv\_sameIRQL** or with one of the other IRQL annotations to indicate that the change in IRQL is expected. In the absence of annotations that indicate any net change in IRQL, PREfast issues a warning for any function that does not exit at the same IRQL at which the function was entered. If the change in IRQL is intended, add the appropriate annotation to suppress the error. If the change in IRQL is not intended, you should correct the code.

The addition of **\_\_drv\_sameIRQL** indicates to other programmers that the original developer consciously considered this behavior to be correct. For example, almost all of the system-defined callback functions are annotated with **\_\_drv\_sameIRQL** because they are expected to exit at the same IRQL at which they were entered. The exceptions are marked appropriately for the action that they take.

### Annotations for Saving and Restoring IRQL for I/O Cancellation Routines

The **\_\_drv\_useCancelIRQL** annotation indicates that the annotated parameter is the IRQL value that should be restored by a DRIVER\_CANCEL callback. The presence of this annotation indicates that the function is a utility that is called from *Cancel* routines and that it completes the requirements for DRIVER\_CANCEL functions (that is, it discharges the obligation for the caller).

For example, the MyCompleteCurrent function is a utility function that is called from many places to implement cancel functionality. One of the parameters is the IRQL that should be restored by this function. The annotations indicate that the function should meet the requirements of a *Cancel* function, as shown in the following example:

VOID

MyCompleteCurrent(

\_\_in PDEVICE\_EXTENSION Extension,

\_\_in\_opt PKSYNCHRONIZE\_ROUTINE SynchRoutine,

\_\_in \_\_drv\_in(\_\_drv\_useCancelIRQL) KIRQL IrqlForRelease,

);

Note that **\_\_drv\_useCancelIRQL** is a low-level annotation. For many uses, **\_\_drv\_isCancelIRQL** is a better choice.

### IRQL Annotation Examples

The examples in this section show IRQL annotations applied to various system functions.

The maximum IRQL at which a fast mutex can be acquired is APC\_LEVEL. Acquiring a fast mutex raises the current IRQL to APC\_LEVEL. When the fast mutex is released, the caller must still be at APC\_LEVEL and the previous IRQL is restored. The example in Listing 23 shows the annotations that enforce these rules.

Listing 23 Example of annotations for enforcing maximum IRQL

\_\_drv\_maxIRQL(APC\_LEVEL)

\_\_drv\_setsIRQL(APC\_LEVEL)

VOID

ExAcquireFastMutex(

\_\_inout

\_\_drv\_out(\_\_drv\_savesIRQL

\_\_drv\_acquiresResource(FastMutex))

PFAST\_MUTEX FastMutex

);

\_\_drv\_requiresIRQL(APC\_LEVEL)

VOID

ExReleaseFastMutex(

\_\_inout

\_\_drv\_in(\_\_drv\_restoresIRQL

\_\_drv\_releasesResource(FastMutex))

PFAST\_MUTEX FastMutex

);

In the example in Listing 24, the annotations override the default **\_\_drv\_maxIRQL(DISPATCH\_LEVEL)** for **KeRaiseIrql** and specify that **KeRaiseIrql** can be used only to raise the IRQL.

Listing 24 Example of annotations for overriding the default maximum IRQL

\_\_drv\_maxIRQL(HIGH\_LEVEL)

VOID

KeRaiseIrql(

\_\_in \_\_drv\_in(\_\_drv\_raisesIRQL) KIRQL NewIrql,

\_\_out \_\_drv\_out\_deref(\_\_drv\_savesIRQL) PKIRQL OldIrql

);

In the example in Listing 25, note the association of the saved and restored IRQL with LockHandle, which has no direct association with the saved IRQL value.

Listing 25 Example of annotations for saving and restoring IRQL

\_\_drv\_maxIRQL(DISPATCH\_LEVEL)

\_\_drv\_savesIRQLGlobal(QueuedSpinLock, LockHandle)

\_\_drv\_setsIRQL(DISPATCH\_LEVEL)

VOID

FASTCALL

KeAcquireInStackQueuedSpinLock (

\_\_in PKSPIN\_LOCK SpinLock,

\_\_in \_\_drv\_in\_deref(

\_\_drv\_acquiresExclusiveResource(QueuedSpinLock))

PKLOCK\_QUEUE\_HANDLE LockHandle

);

\_\_drv\_restoresIRQLGlobal(QueuedSpinLock, LockHandle)

\_\_drv\_requiresIRQL(DISPATCH\_LEVEL)

VOID

FASTCALL

KeReleaseInStackQueuedSpinLock (

\_\_in \_\_drv\_in\_deref(

\_\_drv\_releasesExclusiveResource(QueuedSpinLock))

PKLOCK\_QUEUE\_HANDLE LockHandle

);

The minimum and maximum levels to which the IRQL can be changed are typically applied to callbacks. In the example in Listing 26, the annotations specify that the AddDevice function cannot raise the IRQL at all.

Listing 26 Example of annotations that prevent a function from raising IRQL

typedef

\_\_drv\_maxFunctionIRQL(0)

\_\_drv\_sameIRQL

\_\_drv\_clearDoInit(yes)

\_\_drv\_functionClass(DRIVER\_ADD\_DEVICE)

NTSTATUS

DRIVER\_ADD\_DEVICE (

\_\_in struct \_DRIVER\_OBJECT \*DriverObject,

\_\_in struct \_DEVICE\_OBJECT \*PhysicalDeviceObject

);

typedef DRIVER\_ADD\_DEVICE \*PDRIVER\_ADD\_DEVICE;

### Tips for Applying IRQL Annotations

Here are some tips for applying IRQL annotations to functions:

* Annotate the function with whatever IRQL information might be appropriate.

The additional information helps PRE*f*ast in subsequent checking of both the caller and callee. In some cases, adding an annotation is a good way to suppress a false positive.

* If a function’s annotations do not mention IRQL at all, it is likely a utility function that can be called at any IRQL and thus explicitly having no IRQL annotation is the proper annotation.
* When annotating a function for IRQL, consider how the function might evolve, not just its current implementation.

For example, a function as implemented might work correctly at a higher IRQL than the designer intended. Although it is tempting to annotate the function based upon what the code actually does, the designer might be aware of future requirements, such as the need to lower maximum IRQL for some future enhancement or pending system requirement. The annotation should be derived from the intention of the function designer, not from the actual implementation of the function.

## DO\_DEVICE\_INITIALIZING Annotation

The **\_\_drv\_clearDoInit** annotation specifies that the annotated function is expected to clear the DO\_DEVICE\_INITIALIZING bit in the Flags word of the device object. Calling a function that is annotated with **\_\_drv\_clearDoInit** discharges that obligation for the caller.

This annotation should be used in a conditional context when the function returns success, unless the annotation is applied to a function **typedef** declaration. For an example, see “IRQL Annotation Examples” earlier in this paper.

## Annotations for Interlocked Operands

A large family of functions takes as one of their parameters the address of a variable that should be accessed by using an interlocked processor instruction. These are cache read-through atomic instructions. If the operands are used incorrectly, very subtle bugs can result.

You can apply the **\_\_drv\_interlocked** annotation to a function parameter to identify it as an interlocked operand. System-supplied functions are already annotated for interlocked operands.

PREfast assumes that, if a variable is accessed by any interlocked function, the developer intends to share the variable between threads that could be running on different processors. Thus, any attempt to access or modify that variable without an interlocked operation might be done only in the local processor’s cache, which would be potentially incorrect code. Variables in the local stack frame that is used as the interlocked operand are both very unusual and often dangerous, and usually indicate a misuse of the function.

The following example shows the annotation for an InterlockedExchange function. This annotation specifies that the Target parameter must always be accessed by using an interlocked operation:

LONG

InterlockedExchange(

\_\_inout \_\_drv\_in(\_\_drv\_interlocked) PLONG Target,

\_\_in LONG Value

);

## Examples of Annotated System Functions

The examples in this section show annotations that are applied to commonly used system functions.

In Listing 27, the **IoGetDmaAdapter** function returns a value through a parameter that should be checked, as well as having other annotations.

Listing 27 Example of annotations for IoGetDmaAdapter

PDMA\_ADAPTER

\_\_drv\_maxIRQL(DISPATCH\_LEVEL)

IoGetDmaAdapter(

\_\_in PDEVICE\_OBJECT PhysicalDeviceObject,

\_\_in PDEVICE\_DESCRIPTION DeviceDescription,

\_\_checkReturn

\_\_deref\_inout PULONG NumberOfMapRegisters

);

The **IoCreateDevice** function is generally very simple to use, except that the driver must properly dispose of the created device object—usually by calling **IoAttachDeviceToDeviceStack**. However, when a legacy driver calls **IoCreateDevice** from within the driver’s DRIVER\_INITIALIZE or DRIVER\_DISPATCH functions, the device object is put in a location where it can be found, usually during unload. PREfast is unaware of this location.

The annotations in Listing 28 help to prevent false positives in PREfast. The device object is similar to memory in that it can be aliased, so the example uses a memory annotation.

Listing 28 Example of annotations for IoCreateDevice

\_\_drv\_maxIRQL(APC\_LEVEL)

NTSTATUS

\_\_drv\_valueIs(<0;==0)

IoCreateDevice(

\_\_in PDRIVER\_OBJECT DriverObject,

\_\_in ULONG DeviceExtensionSize,

\_\_in\_opt PUNICODE\_STRING DeviceName,

\_\_in DEVICE\_TYPE DeviceType,

\_\_in ULONG DeviceCharacteristics,

\_\_in BOOLEAN Exclusive,

\_\_out \_\_drv\_out(\_\_drv\_when(return<0, \_\_null)

\_\_drv\_when(return==0, \_\_notnull

\_\_drv\_when(!inFunctionClass$("DRIVER\_INITIALIZE")

&& !inFunctionClass$("DRIVER\_DISPATCH"),

\_\_acquiresMemory(Memory))))

PDEVICE\_OBJECT \*DeviceObject

);

For symmetry with **IoCreateDevice**, the example in Listing 29 shows the annotations for **IoAttachDeviceToDeviceStack**. Note that this example aliases the device object only when the function is successful. Correct code should check the return value for **IoAttachDeviceToDeviceStack** and typically should call **IoDeleteDevice** if it fails. These annotations cause PREfast to enforce those rules.

Listing 29 Example of annotations for IoAttachDeviceToDeviceStack

PDEVICE\_OBJECT

\_\_drv\_maxIRQL(2)

\_\_drv\_valueIs(==0;!=0)

\_\_checkReturn

IoAttachDeviceToDeviceStack(

\_\_in PDEVICE\_OBJECT SourceDevice,

\_\_in

\_\_drv\_in(\_\_drv\_mustHold(Memory)

\_\_drv\_when(return!=0, \_\_drv\_aliasesMem))

PDEVICE\_OBJECT TargetDevice

);

The example in Listing 30 shows annotations for the **EngSaveFloatingPointState** function, which saves the current kernel floating-point state. Although the function prototype appears simple, this function has fairly complex semantics at the contract level and is easy to misuse. Consequently, the annotations that specify correct usage are also fairly complex.

This example uses the **\_\_drv\_arg** annotation to move the more complex annotations from the parameter list to a block of annotations above the beginning of the function prototype. The simple annotations such as **\_\_in** are left in their usual position.

The numbers in comments correspond to explanations after the example.

Listing 30 Example of annotations for EngSaveFloatingPointState

// EngSaveFloatingPointState

\_\_checkReturn // 1

\_\_drv\_arg(\*pBuffer,

\_\_drv\_neverHold(EngFloatState) // 2

\_\_drv\_notPointer) // 3

\_\_drv\_when(pBuffer==0 || cjBufferSize==0,

\_\_drv\_ret(\_\_drv\_valueIs(>=0))) // 4

\_\_drv\_when(pBuffer!=0 && cjBufferSize!=0, // 5

\_\_drv\_ret(\_\_drv\_valueIs(==0;==1))

\_\_drv\_when(return==1,

\_\_drv\_ret(\_\_drv\_floatSaved)

\_\_drv\_arg(pBuffer, \_\_bcount\_opt(cjBufferSize)

\_\_deref \_\_drv\_acquiresResource(EngFloatState))

)

)

ULONG

EngSaveFloatingPointState(

\_\_out\_opt VOID \*pBuffer, // 6

\_\_in ULONG cjBufferSize // 6

);

Starting from the beginning of the example in Listing 30:

1. The result value should always be checked or used.

2. The floating-point state should not be held when the function is called.

3. The buffer pointer should point directly to memory.

4. If either parameter is zero, the function returns a positive number (that is, the size of the buffer required to save the floating-point state) or zero (that is, the processor having no hardware floating-point capability).

5. If both parameters are nonzero, the function returns a BOOLEAN. If it is successful, it saves the floating-point state, fills the buffer, and acquires the floating-point state in the buffer.

6. The usual **\_\_in** and **\_\_out** annotations still apply.

# How to Write and Debug Annotations

Many annotations are straightforward and obvious to write, but some can take a little effort to get right. PREfast attempts to diagnose any errors it sees in annotations, but it cannot check and report every conceivable error. It is always a good idea to write a small test case to confirm that annotations behave as you expect.

A good test case should report any expected errors and should not report instances of correct usage. Simple annotations such as **\_\_in** do not benefit from test cases, but annotations that involve sizes—in particular, annotations that use the **\_\_part** modifier—often benefit from test cases because writing the test cases often forces you to think about corner cases.

## Examples of Annotation Test Cases

The example in Listing 31 shows a set of very simple test cases for the myfun function, which takes three parameters: mode, p1, and p2. The value of mode determines valid values for p1 and p2, so the test cases use the **\_\_drv\_when** conditional annotation to specify how to enforce correct usage of myfun.

Listing 31 Example of annotation test cases for a function

// Annotate the prototype.

\_\_drv\_when(mode==1, drv\_arg(p1, \_\_null))

\_\_drv\_when(mode==2, drv\_arg(p2, \_\_null))

\_\_drv\_when(mode <= 0 || mode > 2,

\_\_drv\_reportError("bad mode value"))

void myfun(\_\_in int mode,

\_\_in struct s \*p1,

\_\_in struct s \*p2);

Starting with the first annotation, the prototype establishes the following usage requirements:

* If mode is 1, p1 must be NULL.
* If mode is 2, p2 must be NULL.
* If mode is negative or greater than 2, issue a “bad mode value” error message.

In Listing 32, the test cases call the function correctly and incorrectly. Incorrect calls cause PREfast to issue a warning.

Listing 32 Example of code that exercises annotation test cases

// A correct use

void dummy1(\_\_in struct s \*a, \_\_in struct s \*b)

{

myfun(0, a, b);

}

// An incorrect use: expect a warning

void dummy2(\_\_in struct s \*a, \_\_in struct s \*b)

{

myfun(1, a, b);

}

// An incorrect use: expect a warning

void dummy3(\_\_in struct s \*a, \_\_in struct s \*b)

{

myfun(2, a, b);

}

// A correct use

void dummy4(\_\_in struct s \*a, \_\_in struct s \*b)

{

myfun(1, NULL, b);

}

// A correct use

void dummy5(\_\_in struct s \*a, \_\_in struct s \*b)

{

myfun(2, a, NULL);

}

// An incorrect use: expect a warning

void dummy6(\_\_in struct s \*a, \_\_in struct s \*b)

{

myfun(14, a, b);

}

## Tips for Writing Annotation Test Cases

Consider these tips when writing annotation test cases:

* It is usually not necessary for the test case code to actually do anything. Often a single function call inside a dummy function is sufficient.
* If you need pointers to structures, it is often enough to simply have the dummy function take those pointers as parameters, with the appropriate **\_\_in**, **\_\_out**, or other annotations that are required for the test.
* It is not necessary to link or run annotation test cases. It is only necessary to use PRE*f*ast to compile them.
* If you are writing several test cases, it is usually a good idea to put each test case in a separate dummy function. PREfast tries to minimize noise by suppressing duplicate errors within a function. Often when you are checking for multiple ways to trigger the error, the duplicate suppression logic suppresses the additional instances.
* Functions that return **void** are usually better for test cases because they are not required to meet the compiler’s requirement that the function return a value.
* Remember to give each function a distinct name.
* Remember that annotations are independent. It is usually unnecessary to check for unexpected interactions of unrelated annotations.

# PRE*f*ast Annotation Best Practices

The following guidelines represent best practices for applying annotations:

#### Use annotations in a way that makes sense for your project.

The “right” approach to using annotations is the one that works best for your project:

* For some development projects, it might make sense to exercise PREfast capabilities to the fullest, by proactively examining every function in the program and applying the appropriate annotations and other recommended changes.

This takes some effort but helps to assure that every bug that PREfast could find is found and identified. Done correctly, this minimizes false positives as well.

* If time and resources are limited, it might make sense to take a reactive approach by running PREfast on existing code and applying whatever annotations are needed to suppress false positives.

This does not necessarily find all of the problems that the proactive approach might find, but is a valid way to start using PREfast.

Rather than postpone using PREfast until there is “time to do it right,” it is better to follow a reactive model—or even to apply no annotations and ignore the noise. Typically, the time that is saved when PREfast finds even a few problems is enough to compensate for the effort of applying annotations, compared to the time that is required to debug those problems by using conventional methods. Over time, transitioning to a more proactive approach is a good idea, and the value of being more proactive should become obvious.

#### Use annotations to describe a successful function call.

Annotations should reflect a successful call to the function. PREfast should catch inherently ill-formed or unsuccessful calls, not just calls that cause the system to crash. Annotations should reflect the intent of the function as reflected by its interface, not the actual implementation of the function.

A common example is optional parameters. A lot of code is written that checks to see if a parameter is NULL. But is a NULL parameter genuinely optional? For example:

* If the function ignores the NULL parameter and proceeds to do something useful, then the parameter is optional.
* If the function returns an error when it is called with a NULL parameter, the parameter is not optional. Instead, the function defends itself from bad usage.

If PREfast can tell the caller that a potentially NULL parameter will cause the function to fail, then PREfast can find the potential bug rather than having the bug surface at runtime in some obscure and hard-to-test-for circumstance. Having a function defend itself against bad parameters is good practice, but the fact that it does so does not make the parameter “optional” in this sense.

#### Consider how a function might evolve.

When annotating a function, it is important to consider how the function might evolve. The annotation should reflect the intention of the function designer, not necessarily the current implementation of the function.

For example, a function as implemented might work correctly with parameter values that are different from those that the designer intended. Although it is tempting to annotate the function based upon what the code actually does, the designer might be aware of future requirements, such as the need to maintain some restriction to support some future enhancement or pending system requirement.

#### Resolve conflicts between a function and its documentation.

Annotations can expose two kinds of “conflicts” between a function’s implementation and its documentation:

* The code and the documentation are factually inconsistent—one or the other must be fixed.
* The documentation describes as required something that is true about the implementation but is not enforced—that is, an “incorrect” function call would succeed.

In this case, the function designer must decide whether to revise the documentation or use annotations to enforce the documented behavior of the function, to ensure that the function is used correctly as intended.

#### Use the /W4, /WX, and /Wp64 compiler flags.

The compiler also detects a number of potential errors, not all of which are detected by PREfast. It is good practice to use the following compiler flags:

|  |  |
| --- | --- |
| **/W4** | Warn at level 4 |
| **/WX** | Make warnings fatal |
| **/Wp64** | Warn on 64-bit portability issues |

Minimize the scope of any **#pragma** warning annotations that are used to suppress false-positive warnings from the compiler.

#### Use only the annotations described in this paper for annotating driver code.

Annotations are implemented in different ways, depending on the exact version of the analysis tools you are using. Documentation and header file comments that mention **\_\_declspec** or annotations that use a square-bracket notation similar to those used in C# can be ignored for the purposes of using PREfast. Only the annotations that are discussed in this paper are officially supported for annotating driver code.

Some annotations are currently only partially implemented. Some annotations might be sufficient to suppress a spurious warning in PREfast, but they do not enable the additional checks that the annotation name might imply. In other cases, they are simply structured comments. These annotations are being considered for future versions of PREfast.

**Note** Annotations can also be provided through a model file. For various reasons, many Microsoft-provided system functions are annotated in the model file and not yet annotated in the source code. The model file is no more powerful than the annotations that are described here and is more difficult to use, so its use is neither documented nor recommended for new annotations.

# Example: Osrusbfx2.h with Annotations

Listing 33 shows an example of a header file that is annotated with the PRE*f*ast **\_drv\_requiresIRQL**, **\_\_in**, and **\_\_out** annotations.

This header file also shows KMDF callback role type annotations, such as **EVT\_WDF\_DRIVER\_DEVICE\_ADD**, which Static Driver Verifier can interpret. For details, see Static Driver Verifier in “Resources” at the end of this paper.

Listing 33 Osrusbfx2.h with annotations

#pragma warning(disable:4200) // nameless struct/union

#pragma warning(disable:4201) // nameless struct/union

#pragma warning(disable:4214) // bit field types other than int

#include <initguid.h>

#include <ntddk.h>

#include "usbdi.h"

#include "usbdlib.h"

#include "public.h"

#include "driverspecs.h"

#pragma warning(default:4200)

#pragma warning(default:4201)

#pragma warning(default:4214)

#include <wdf.h>

#include <wdfusb.h>

#define NTSTRSAFE\_LIB

#include <ntstrsafe.h>

#include "trace.h"

#ifndef \_PRIVATE\_H

#define \_PRIVATE\_H

#define POOL\_TAG (ULONG) 'FRSO'

#define \_DRIVER\_NAME\_ "OSRUSBFX2:"

#define TEST\_BOARD\_TRANSFER\_BUFFER\_SIZE (64\*1024)

#define DEVICE\_DESC\_LENGTH 256

//

// Define the vendor commands supported by our device

//

#define USBFX2LK\_READ\_7SEGMENT\_DISPLAY 0xD4

#define USBFX2LK\_READ\_SWITCHES 0xD6

#define USBFX2LK\_READ\_BARGRAPH\_DISPLAY 0xD7

#define USBFX2LK\_SET\_BARGRAPH\_DISPLAY 0xD8

#define USBFX2LK\_IS\_HIGH\_SPEED 0xD9

#define USBFX2LK\_REENUMERATE 0xDA

#define USBFX2LK\_SET\_7SEGMENT\_DISPLAY 0xDB

//

// Define the features that we can clear

// and set on our device

//

#define USBFX2LK\_FEATURE\_EPSTALL 0x00

#define USBFX2LK\_FEATURE\_WAKE 0x01

//

// Order of endpoints in the interface descriptor

//

#define INTERRUPT\_IN\_ENDPOINT\_INDEX 0

#define BULK\_OUT\_ENDPOINT\_INDEX 1

#define BULK\_IN\_ENDPOINT\_INDEX 2

//

// A structure representing the instance information associated with

// this particular device.

//

typedef struct \_DEVICE\_CONTEXT {

WDFUSBDEVICE UsbDevice;

WDFUSBINTERFACE UsbInterface;

WDFUSBPIPE BulkReadPipe;

WDFUSBPIPE BulkWritePipe;

WDFUSBPIPE InterruptPipe;

UCHAR CurrentSwitchState;

WDFQUEUE InterruptMsgQueue;

} DEVICE\_CONTEXT, \*PDEVICE\_CONTEXT;

WDF\_DECLARE\_CONTEXT\_TYPE\_WITH\_NAME(DEVICE\_CONTEXT, GetDeviceContext)

extern ULONG DebugLevel;

DRIVER\_INITIALIZE DriverEntry;

EVT\_WDF\_OBJECT\_CONTEXT\_CLEANUP OsrFxEvtDriverContextCleanup;

EVT\_WDF\_DRIVER\_DEVICE\_ADD OsrFxEvtDeviceAdd;

EVT\_WDF\_DEVICE\_PREPARE\_HARDWARE OsrFxEvtDevicePrepareHardware;

EVT\_WDF\_IO\_QUEUE\_IO\_READ OsrFxEvtIoRead;

EVT\_WDF\_IO\_QUEUE\_IO\_WRITE OsrFxEvtIoWrite;

EVT\_WDF\_IO\_QUEUE\_IO\_DEVICE\_CONTROL OsrFxEvtIoDeviceControl;

EVT\_WDF\_REQUEST\_COMPLETION\_ROUTINE EvtRequestReadCompletionRoutine;

EVT\_WDF\_REQUEST\_COMPLETION\_ROUTINE EvtRequestWriteCompletionRoutine;

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

ResetPipe(

\_\_in WDFUSBPIPE Pipe

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

ResetDevice(

\_\_in WDFDEVICE Device

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

SelectInterfaces(

\_\_in WDFDEVICE Device

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

AbortPipes(

\_\_in WDFDEVICE Device

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

ReenumerateDevice(

\_\_in PDEVICE\_CONTEXT DevContext

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

GetBarGraphState(

\_\_in PDEVICE\_CONTEXT DevContext,

\_\_out PBAR\_GRAPH\_STATE BarGraphState

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

SetBarGraphState(

\_\_in PDEVICE\_CONTEXT DevContext,

\_\_in PBAR\_GRAPH\_STATE BarGraphState

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

GetSevenSegmentState(

\_\_in PDEVICE\_CONTEXT DevContext,

\_\_out PUCHAR SevenSegment

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

SetSevenSegmentState(

\_\_in PDEVICE\_CONTEXT DevContext,

\_\_in PUCHAR SevenSegment

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

GetSwitchState(

\_\_in PDEVICE\_CONTEXT DevContext,

\_\_in PSWITCH\_STATE SwitchState

);

\_\_drv\_requiresIRQL(DISPATCH\_LEVEL)

VOID

OsrUsbIoctlGetInterruptMessage(

\_\_in WDFDEVICE Device

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

OsrFxSetPowerPolicy(

\_\_in WDFDEVICE Device

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

NTSTATUS

OsrFxConfigContReaderForInterruptEndPoint(

\_\_in PDEVICE\_CONTEXT DeviceContext

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

VOID

OsrFxEvtUsbInterruptPipeReadComplete(

\_\_in WDFUSBPIPE Pipe,

\_\_in WDFMEMORY Buffer,

\_\_in size\_t NumBytesTransferred,

\_\_in WDFCONTEXT Context

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

BOOLEAN

OsrFxEvtUsbInterruptReadersFailed(

\_\_in WDFUSBPIPE Pipe,

\_\_in NTSTATUS Status,

\_\_in USBD\_STATUS UsbdStatus

);

EVT\_WDF\_IO\_QUEUE\_IO\_STOP OsrFxEvtIoStop;

EVT\_WDF\_DEVICE\_D0\_ENTRY OsrFxEvtDeviceD0Entry;

EVT\_WDF\_DEVICE\_D0\_EXIT OsrFxEvtDeviceD0Exit;

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

BOOLEAN

OsrFxReadFdoRegistryKeyValue(

\_\_in PWDFDEVICE\_INIT DeviceInit,

\_\_in PWCHAR Name,

\_\_out PULONG Value

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

VOID

OsrFxEnumerateChildren(

\_\_in WDFDEVICE Device

);

\_\_drv\_requiresIRQL(PASSIVE\_LEVEL)

PCHAR

DbgDevicePowerString(

\_\_in WDF\_POWER\_DEVICE\_STATE Type

);

#endif

# Resources

PREfast on the WHDC Web site  
<http://www.microsoft.com/whdc/DevTools/tools/PREfast.mspx>

Static Driver Verifier  
<http://www.microsoft.com/whdc/devtools/tools/sdv.mspx>

Windows Driver Kit (WDK)  
<http://www.microsoft.com/whdc/DevTools/ddk/default.mspx>

#### Books

*Developing Drivers with the Windows Driver Foundation*, by Penny Orwick and Guy Smith  
 <http://www.microsoft.com/MSPress/books/10512.aspx>

#### White Papers

PREfast Step-by-Step

<http://www.microsoft.com/whdc/DevTools/tools/PREfast_steps.mspx>

#### WDK documentation

PREfast for Drivers