INTEL CORPORATION

Intel® Memory Protection Extensions Enabling Guide

Rev 1.00

Ramu Ramakesavan, Dan Zimmerman, Pavithra Singaravelu, George Kuan, Brian Vajda, Scott Gibbons, and Gautham Beeraka

3/30/2016

Abstract: This document describes Intel® Memory Protection Extensions, its motivation, and programming model. It also describes the enabling requirements and the current status of enabling in the supported operating systems: Linux* and Windows* and compilers: Intel ® C++ Compiler, GNU Compiler Collection*, and Visual Studio*. Finally, the paper describes how ISVs can incrementally enable bounds checking in their Intel MPX applications.

1 Contents

2 License

The code samples in the sections covering GNU Compiler Collection* (GCC) implementation are governed by the GNU* General Public License, version 2 (GPLv2), (https://www.gnu.org/licenses/gpl-2.0.txt) and sections covering Intel® C++ Compiler implementation are governed by the Intel Sample Source Code License Agreement (https://software.intel.com/en-us/articles/intel-sample-source-code-license-agreement).

3 Introduction

C/C++ pointer arithmetic is a convenient language construct often used to step through an array of data structures. If an iterative write operation does not take into consideration the bounds of the destination, then adjacent memory locations may get corrupted. Such modification of adjacent data not intended by the developer is referred as a buffer overflow. Similarly, uncontrolled reads could reveal cryptographic keys and passwords. Buffer overflows have been known to be exploited, causing denial of service (DoS) attacks and system crashes. More sinister attacks, which do not immediately draw the attention of the user or system administrator, alter the code execution path, such as modifying the return address in the stack frame, to execute malicious code or script.

Intel's Execute Disable Bit and similar HW features from other vendors have blocked all buffer overflow attacks that redirected the execution to malicious code stored as data. Various other techniques adopted by compiler vendors to mitigate buffer overflow problems can be found in the references.

Intel® Memory Protection Extensions (Intel® MPX) technology consists of new Intel® architecture instructions and registers that C/C++ compilers can use to check the bounds of a pointer before it is used. This new HW technology will be enabled in future Intel® processors. The supported compilers are the Intel® C/C++ compiler and GCC (GNU C/C++ compiler).

4 Origins of buffer overflow vulnerabilities

At its inception, the C programming language became very popular in the Unix* community. As a high-level language, developers found it easier to express application logic. The minimal type checking made it suitable for writing kernel-level components, and register variables helped write efficient code. However, another popular feature, pointer arithmetic, became a double-edged sword. Cybercriminals used it to create DoS attacks and malicious redirection of execution. Frequently used glibc routines, like gets and scanf, made it easier to cause buffer overflows because these routines were too simplistic and lacked data to check for buffer overflows.

DoS attacks resulting from corrupted data are human-noticeable events that are reported early and fixed, but attacks that redirect execution, such as altering the return address in a stack frame, are harder to detect before they have executed malicious code. A well-known buffer overflow attack was the Morris Worm that was introduced in 1988 by a grad student at Cornell. The program exploited various weaknesses in BSD Unix* systems including a buffer overflow weakness in the once popular "fingerd" daemon. It became well known not because of any damage it caused, but it was the first such attack where the creator was convicted under the new laws against computer fraud/abuse. Furthermore, DARPA (Defense Advanced Research Projects Agency) created the first response team to deal with such attacks on the Internet.

There were other notable buffer overflow vulnerabilities. A heap-based buffer overflow attack on Apple Safari* 7.0.2 caused an application to break out of the sandbox it was assigned to. A stack overflow attack on Microsoft Word* allowed users to execute malicious code through a crafted document in Microsoft Word 2003 SP3. NIST database on Internet vulnerabilities lists several known buffer overflow vulnerabilities in popular applications.

5 Other solutions

GCC has address sanitizer (https://code.google.com/p/address-sanitizer/) and mudflap pointer debugger (http://gcc.gnu.org/wiki/Mudflap_Pointer_Debugging), which were added to solve the problem with invalid memory accesses. The main difference of the described approach from these existing solutions is that each memory access is checked against pointer bounds, not against tables of valid addresses, providing the following advantages and drawbacks:

Advantages:

- When an overflow occurs and the pointer points out of the object, it may still point to a valid memory. Intel MPX instrumentation can detect such corruptions.
- Narrowing is possible for Intel MPX instrumentation only. For example, Intel MPX can catch an overflow in a static array that is a field of a structure.
- Intel MPX instrumentation allows manual bounds assignment for pointers.
- Compatibility with legacy code is much higher for code instrumented with Intel MPX because false violations are rare.
- Intel MPX instrumentation has hardware support.
- Intel MPX code may be disabled in runtime (all Intel MPX instructions are executed as NOPs), resulting in minimum overhead. Thus it is possible to use one binary file for both debug and release versions of a product.

Drawbacks:

- Intel MPX instrumentation does not detect dangling pointers.
- Intel MPX instrumentation is more complex. Compilers must track all pointer movements.

6 Intel® Memory Protection Extensions programming model

Intel MPX is a set of processor features that, along with modifications in the OS and compilers (assemblers, linkers) and a new runtime shared library, bring increased robustness to software by checking pointer references whose compile time normal intentions are usurped at runtime due to buffer overflow. The two primary goals of Intel MPX are to provide this capability at low performance overhead for newly compiled code and to maintain compatibility with legacy software components.

When executing software containing a mix of Intel MPX code and legacy code, the legacy code does not benefit from Intel MPX, but it also does not experience any change in functionality or performance. Intel MPX is designed such that enabled applications can link with, call into, or be called from legacy software (libraries, and so on) while maintaining existing application binary interfaces (ABIs). However, some source code changes may be required to take care of incorrect bounds computation due to adherence to C/C++ standards (like the bounds of the first field in a structure and inner arrays) and missing bounds information (pointer returned from legacy code).

Code enabled for Intel MPX running on processors that do not support Intel MPX or processors that support Intel MPX but the OS or runtime do not have it enabled results in performance similar to embedding NOPs in the instruction stream.

6.1 Bounds registers and compare instructions

Intel MPX introduces four new 128-bit-long bounds registers (BND0-3) and eight instructions that operate on the bounds registers. The bounds registers hold the lower and upper bounds for a pointer variable as shown in Figure 1.

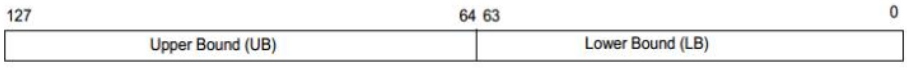

Figure 1. Layout of bounds registers BND0-3.

6.1.1 Instructions to compare bounds

The new Intel MPX instructions (BNDMK, BNDCL, BNDCU, BNDCN) are used to compare the value of a pointer against its lower and upper bound values stored in one of the bounds registers. A new #BR exception is raised if any of the bounds compare instructions fail (see Table 1 and Figure 2).

Table 1. MPX Set Bounds and Compare Instructions

New Instruction	Function
BNDMK b, m	Creates LowerBound (LB) and UpperBound (UB) in bounds register b.
BNDCL b, r/m	Checks the address of a memory reference or address in r against the lower bound.
BNDCU b, r/m	Checks the address of a memory reference or address in r against the upper bound.
BNDCN b, r/m	Checks the address of a memory reference or address in r against the upper bound in one's complement.

```
int A[100]; //assume the array A is allocated on the stack at 'offset' from RBP.
// the instruction to store starting address of array will be:
        LEA RAX, [RBP+offset]
// the instruction to create the bounds for array A will be:
        BNDMK BND0, [RAX+399]
// Store RAX into BND0.LB, and ~(RAX+399) into BND0.UB.
```

```
// similarly, for a library implementation of dynamic allocated memory
        int * k = malloc(100);
// assuming that malloc returns pointer k in RAX and holds (size - 1) in RCX
// the malloc implementation will execute the following instruction before returning:
        BNDMK BND0, [RAX+RCX]
// BND0.LB stores RAX, and BND0.UB stores ~(RAX+RCX)
```

Figure 2. Sample Code using BNDMK

If you write a 32-bit integer in 64-bit mode into a buffer at the address specified in RAX and the bounds are in register BND0, the instruction sequence will be:

```
BNDCL BND0, [RAX]
BNDCU BND0, [RAX+3] // operand size is 4
MOV DWORD PTR [RAX], RBX // RBX has the data to be written to the buffer.
```

6.1.2 Instructions to spill and fill bounds registers

Four bounds registers are obviously insufficient to store bounds for all pointers in a typical application. When a compiler has to bounds check a fifth pointer variable, it resorts to spilling a currently used pointer and its associated bounds register using various criteria. Spilling of bounds registers are also required when passing pointer parameters to a subroutine call. GCC's ABI has been modified to use the bounds

registers BND0 to BND3, in that order, to pass the bounds associated with the first four parameters of a subroutine call.

A bounds register could be spilled onto another bounds register, memory location, or a dynamically created *Bounds Table Entry*, using four new Intel MPX instructions as shown in Table 2.

Table 2. Bounds Register Spill/Fill Instructions

New Instruction	Function	Performance Characteristic
BNDMOV b, b/m	Copy/load LB and UB bounds from memory or a bounds register.	Move the BNDx registers from/to memory, mostly used to spill/fill.
BNDMOV b/m, b	Store LB and UB bounds in a bounds register to memory or another register.	Move the BNDx registers from/to memory, mostly used to spill/fill.
BNDLDX b, mib	Load bounds using address translation using an sib-addressing expression mib.	Access the Bound Table, a 2-layer cache-like data structure. Slower than the BNDMOV (accesses two different cache lines). Used mostly to spill/fill for parameter passing in subroutine call.
BNDSTX mib, b	Store bounds using address translation using an sib-addressing expression mib.	Access the Bound Table, a 2-layer cache-like data structure. Supposedly very slow (accesses two different cache lines). Used mostly to spill/fill for parameter passing in subroutine call.

6.2 CPUID settings

A new CPUID (EAX=07H, ECX-0H) bit14 (Intel MPX bit) indicates whether the CPU supports Intel MPX. CPUs that don't support Intel MPX interpret Intel MPX instructions as NOPs. See Table 3.

Table 3. CPUID Intel® Memory Protection Extensions (Intel® MPX) Bit

Intel MPX bit	
Bit	**Function**
0	Processor does not support Intel MPX. All Intel MPX instructions are interpreted as NOPs.
1	Processor is enabled for Intel MPX. The Intel MPX instructions may still be interpreted as NOPs if: • OS has not enabled XSAVE/XRESTR of bounds, config, and status registers. • Intel MPX runtime has not enabled Intel MPX in the mode (User/Kernel) specific config register.

Two new Intel MPX bits in the XCR0 enable save/restore (XSAVE/XRSTOR) of four Intel MPX bounds registers, two config registers, and one status register during context switches. The OS should set both bits to ONE to enable Intel MPX; otherwise the processor would interpret Intel MPX instructions as NOPs. See Table 4 and **Figure *3***.

Table 4. bits in XCRO register

XCR0 setting	
Bit	**Function**
BNDREGS	for saving and restoring BND0-BND3
BNDCSR	for saving and restoring the user-mode configuration (BNDCFGU) and the status register BNDSTATUS

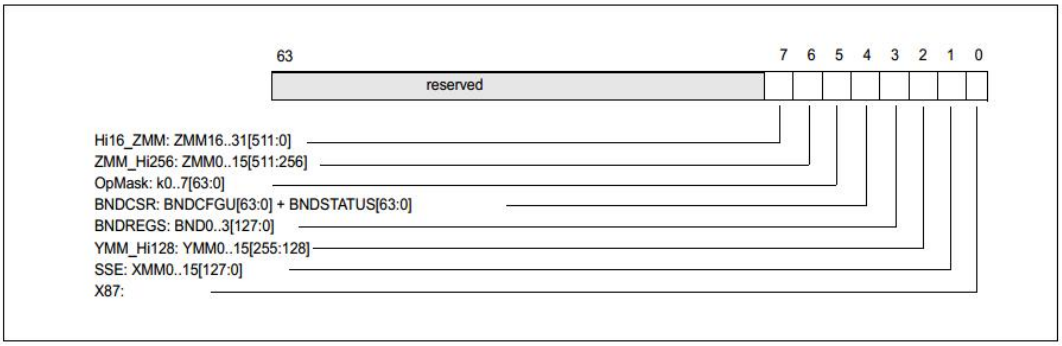

Figure 3. Extended Processor State Components for Intel® Memory Protection Extensions

The settings in XCR0 and offsets of the components relative to the beginning of XSAVE/XRSTOR area can be read from CPUID and interpreted as shown in Table 5.

Table 5. CPUID Setting and Interpretation

CPUID Setting	**Interpretation**
CPUID.(EAX=0DH, ECX=0):EAX[3]	XCR0.BNDREGS[bit 3] is supported. BND0-3 should be saved and restored at context switches.
CPUID.(EAX=0DH, ECX=0):EAX[4]	XCR0.BNDCSR[bit 4] is supported. BNDCFGx and BNDSTATUS should be saved and restored at context switches.
CPUID.(EAX=0DH, ECX=03H).EAX[31:0] & CPUID.(EAX=0DH, ECX=03H).EBX[31:0].	Size and offset for the bounds registers BND0-3 on the XSAVE/XRSTOR stack.

CPUID.(EAX=0DH, ECX=04H).EAX[31:0] & CPUID.(EAX=0DH, ECX=04H).EBX[31:0]	Size and offset for BNDCFGU, BNDCFGS, and BNDSTATUS on the XSAVE/XRSTOR stack.

Note: The OS should set both bits in XCR0 to enable correct interpretation of Intel MPX instructions. It is not sufficient to set one bit.

Note: Setting the two XCR0 bits and the Enable bit in the config registers are necessary to enable Intel MPX; otherwise, Intel MPX instructions would be interpreted as NOPs.

6.3 Config and status registers

There are two config registers (one for kernel mode and another for User mode) and one status register. The config and status registers are RW registers spilled and filled by the OS on thread context switches. The Intel MPX runtime library, one of the enabling SW components, initializes the config registers. See Figure 4 and Table 6.

Note: Intel MPX can be enabled for user mode apps, kernel mode apps, or both depending on the Enable bit setting in the config registers.

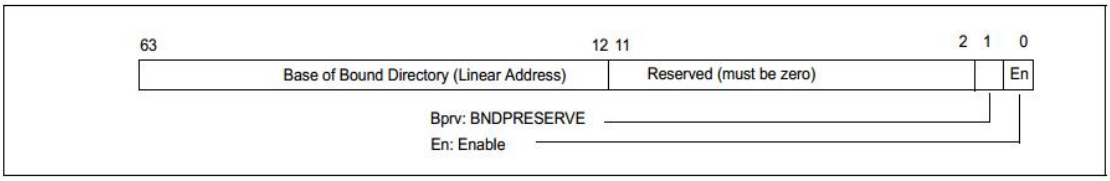

Figure 4. Layout of bounds configuration registers

Table 6. Bounds Config Register Interpretation

BNDCFGS/U	
Bits	**Function**
En (bit 0)	1– interpret the Intel® MPX instructions as per the definitions 0 – interpret the Intel MPX instructions as NOPs
Bprv (BNDPRESERVE) (bit 1)	1 – legacy instructions: CALL, RET, JMP, Jcc, will NOT INIT bounds registers 0 – legacy instructions: CALL, RET, JMP, Jcc, will INIT bounds registers This setting is significant for coexistence of Intel MPX and legacy code in an application.

Base of Bound Directory (bits 12-63)	Linear address of Bound Directory

Intel MPX runtime is expected to initialize the config register to the same value for all threads in an application.

Note: All threads are Intel MPX-enabled or no thread is Intel MPX-enabled by the Intel MPX runtime.

The BNDSTATUS register provides two fields to indicate the status of #BR exception (See Figure 5):

- EC (bits 1:0): source of #BR exception. See Table 7 for details.
- ABD: (bits 63:2): The address field of a bound directory entry when exception is caused by BNDSTX failure.

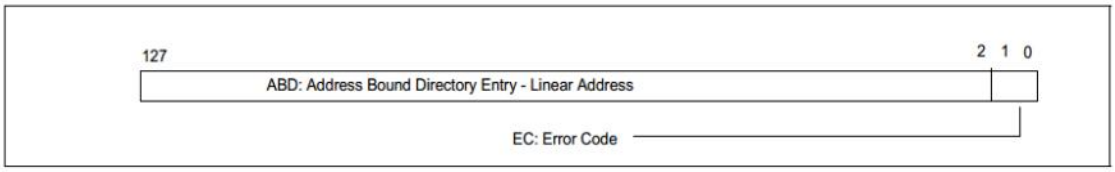

Figure 5. Layout of Bound Status Register

Table 7 describes the error codes.

Table 7. Error Code Definitions of BNDSTATUS

EC	Descriptions	Meaning
00b	No Intel® Memory Protection Extensions exception	Caused by Legacy BOUND instruction
01b	Bounds violation	#BR caused by BNDCL, BNDCU, or BNDCN instructions; ABD is 0
10b	Invalid BD entry	#BR caused by BNDLDX or BNDSTX instructions. BD will be set to the linear address of the invalid Bound directory entry
11b	Reserved	Reserved

Three instances can raise a #BR exception:

- A bounds pointer comparison instruction—BNDCL, BNDCU, or BNDCN—failed.
- BNDSTX failed because the Bound directory entry was invalid.
- Legacy BOUND instruction, which is unrelated to Intel MPX, is executed.

Intel MPX runtime takes suitable action in each case.

7 Enabling software for Intel Memory Protection Extensions

Intel MPX should be enabled in the OS and C/C++ compiler components (compiler, assembler, linker, libraries). An Intel MPX runtime library is also required.

7.1 Enabling the OS

Modify OS initialization to do the following:

- If the processor supports Intel MPX (read CPUID), then set the XCR0 register bits to enable spilling and filling Intel MPX registers during context switches as described in section 6.2, “CPUID settings”.

The developer and validation engineer should keep the following points in mind:

- OS enabling is necessary for Intel MPX buffer overflow protection to take effect; however, an Intel MPX-enabled application may still not be protected if Intel MPX runtime does not set the *Enable* bit in the BNDCFGU/S config registers for each thread.
- Even if the HW, OS, and Intel MPX are enabled in the BNDCFGx register, the bounds register may be set to INIT under low memory conditions, thus depriving the application of the benefits of Intel MPX.
- The bounds, config, and status registers are spilled and filled with each context switch whether an Intel MPX application is running or not.
- A #BR exception can be raised by the legacy BOUND instruction.

Table 8 shows the status of enabling Intel MPX in various OSs.

Table 8. Status of OS Enabling

Operating System	Status
Microsoft Windows* (Desktop)	Enabled in Microsoft Windows 10
Microsoft Windows (Modern UI)	Enabled in Microsoft Windows 10
Linux*	Enabled. Not yet released

7.2 Intel Memory Protection Extensions runtime

Intel MPX runtime is a new SW component required to enable Intel MPX. It is a collection of Intel MPX functions that can be implemented as a static or shared library, a kernel driver, or could be part of an OS. In Microsoft Windows* 10, Intel MPX runtime support is implemented as a kernel driver.

If a processor supports Intel MPX and the OS has enabled Intel MPX in the SW, then it implements the following functions:

1. At application initialization
    a. Install #BR handler
    b. Allocate memory for Bound Directory
    c. Create a background daemon to clean up Bounds Tables in the background
    d. Enable Intel MPX in the BNDCFGx register for the first thread
2. At application runtime
    a. Handle #BR exception
        i. Allocate Bounds Table on demand
        ii. Process buffer overflow
    b. Enable Intel MPX in the BNDCFGx register for each thread
    c. Modify the actions based on user session variables
3. Deallocate Bound Tables when the entries are no longer in use.

The following sections discuss the details about the functions listed above.

7.2.1 Enable Intel MPX for a thread

Intel MPX is enabled for a specific thread by setting the En:Enable bit in the BNDCFGx config register to ONE; otherwise, all Intel MPX instructions would be executed as NOPs. You can find details of the config register setting in Section 6.3, “Config and status registers”.

Theoretically, the three fields could be set differently for each thread, but current implementations of the runtime set the same value for all threads.

The cumulative effects of enabling Intel MPX in the processor, OS, and individual threads are shown in Table 9.

Table 9. Intel® Memory Protection Extensions (Intel® MPX) Feature Enabling

CR4	XCR0		**IA32_BNDCFGS**	**BNDCFGU**	**Intel® MPX Instruction Behavior**		**Load/Store to BND0-BND3, BNDCFGU, BNDSTATUS**	
OXSAVE	**BndRegs**	**BndCSR**	**Bit0**	**Bit 0**	**CPL0-2**	**CPL3**	**XSAVE**	**XRSTOR**
0	NA	NA	NA	NA	NOP	NOP	#UD	#UD
1	0	0	X	X	NOP	NOP	No	No
1	1	1	0	0	NOP	NOP	Yes	Yes
1	1	1	0	1	NOP	MPX	Yes	Yes
1	1	1	1	0	MPX	NOP	Yes	Yes

1	1	1	1	1	MPX	MPX	Yes	Yes

The config registers also contain two other important fields:

- BNDPRESERVE bit determines how the BND registers behave on short CALL, RET, and Near Jcc. For details, see Section 7.3.3, “BND prefix for branch instructions”.
- Bound Directory Base: This is the base of the Bound Directory allocated by the runtime during initialization. For details, see Section 7.2.2, “Bound directory and bounds tables”.

7.2.2 Bound directory and bounds tables

An application can contain hundreds of pointers, and four BND registers are insufficient to store bounds for this number of pointers. In 64-bit mode, a 2GB Bound Directory with its entries pointing to a 4MB Bounds Table is used to store its bounds and additional pointer values. A Bounds Table is an array of Bounds Table Entries, which are 64-bit long 4-tuples storing the pointer value, lower bound of the buffer, upper bound of the buffer, and a reserved field. At initialization, all Bound Directory entries are flagged INVALID as the corresponding Bounds Table has not yet been allocated (see Figure 6).

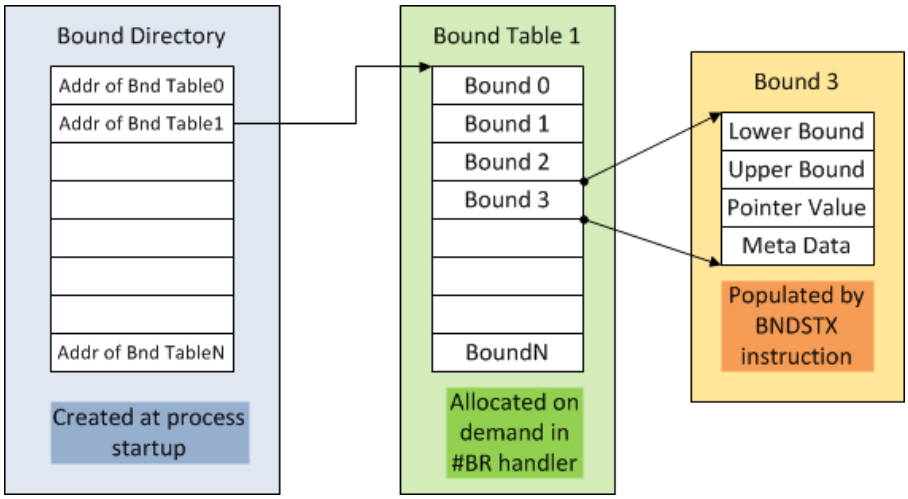

Figure 6. MPX Bound Directory and Bound Table entries

The following operations are executed on thread creation in a Windows OS.

When an application enabled with Intel MPX is launched (see Figure 7):

- Kernel creates first thread.
- MpxRuntime.sys driver receives a process create notification from the OS.

- Creates BD (Bound Directory).
  - BD is allocated as Demand Zero pages.
  - No Bounds Tables are created yet, so all BD entries are invalid.
- Register exception handler for #BR.

- Update BNDConfig register to enable Intel MPX for thread.

For each subsequent thread creation:

1. MpxRuntime.sys receives a thread create notification from the OS.
2. Update BNDConfig register to enable Intel MPX for the thread.

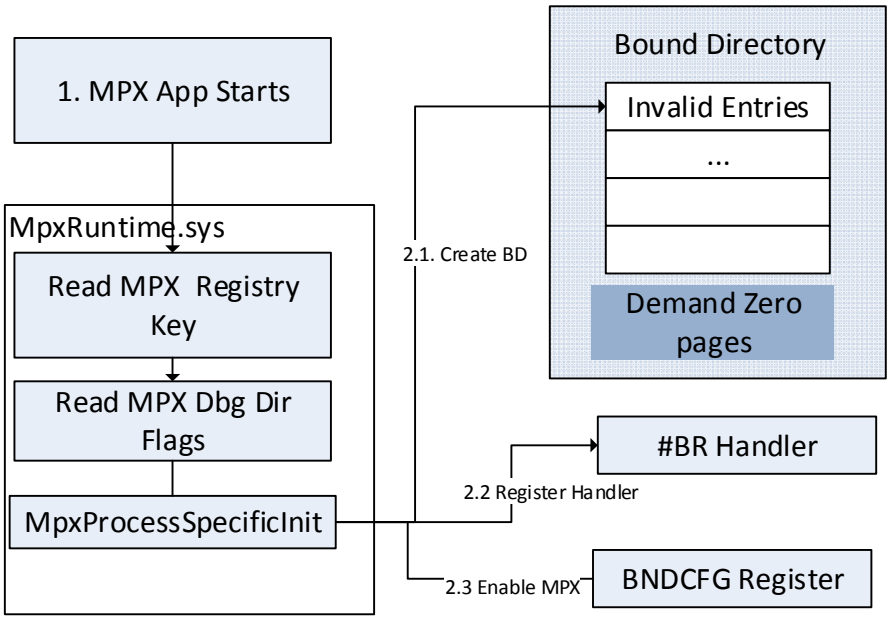

Figure 7. Intel® Memory Protection Extensions initialization

The Bound Directory is allocated in virtual memory, so physical memory is allocated only when a page fault occurs. A Bounds Table is allocated only when a BNDSTX instruction results in a #BR exception. For further details, see Section 7.2.3.2, “Exceptions caused by BNDSTX failure”.

7.2.3 #BR exception handling

#BR exceptions are generated by the HW under three conditions:

- One of the BNDCL, BNDCU, BNDCN instructions fails the comparison test.
- BNDSTX could not find a free Bounds Table Entry to copy a pointer and its bounds.
- Raised by legacy BOUND instruction.

7.2.3.1 Exception caused by BNDCL/U/N instructions

Intel MPX runtime supports two modes of operation for handling exceptions caused by the BNDCL/U/N instructions. The driver can immediately terminate the application or it can allow the application's exception handler to catch the exception and perform any needed operations. This behavior is controlled by the IMAGE_MPX_ENABLE_FAST_FAIL_ON_BND_EXCEPTION flag which can be set in the PE header debug directory information or in the registry. Setting this flag will cause the driver to immediately terminate the application on a bound check failure. (See Section 7.2.4, "Control of the Intel Memory Protection Extensions runtime driver (Windows OS)", for control flag details.)

7.2.3.2 Exceptions caused by BNDSTX failure

The BNDSTX and BNDLDX instructions have two operands each:

BNDSTX mib, b

BNDLDX b, mib

Operand *b* is one of the bounds registers: BND0-3. Operand *mib* contains two values: the address of the stored pointer variable and an index register storing the value of the pointer variable, as the example in Figure 8 shows.

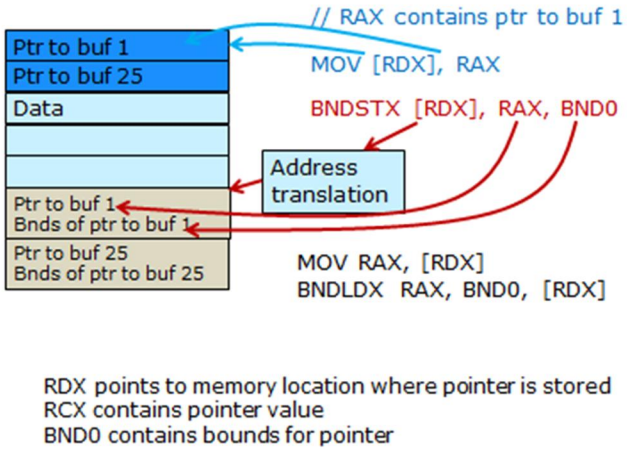

Figure 8. Storing and loading bounds

The bit representation of the address of the pointer is used to compute two offsets:

- Offset1: points to the Bound Directory entry, which contains the pointer to the Bounds Table that should be written/read

- Offset2: points to the Bounds Table entry that should be written/read

Figure 9 shows how the offsets are computed.

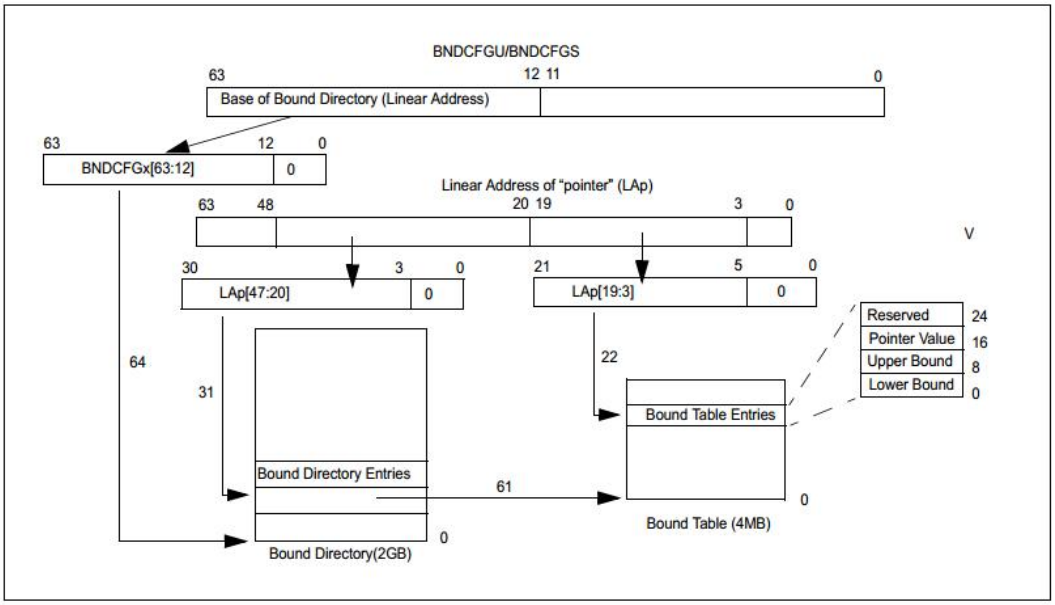

Figure 9. Bound paging structure and address translation in 64-bit mode

Any attempt to write the first entry into a Bounds Table fails with a #BR exception because the corresponding Bound Directory entry is invalid due to a Bounds Table having not been created. The #BR exception handler allocates a Bounds Table and sets the Bound Directory entry to VALID. The BNDSTX command is retried at this point and succeeds. The instruction writes the triplet (pointer value, lower bound, upper bound) into the Bounds Table Entry determined by the computed offset2, as Figure 10 shows:

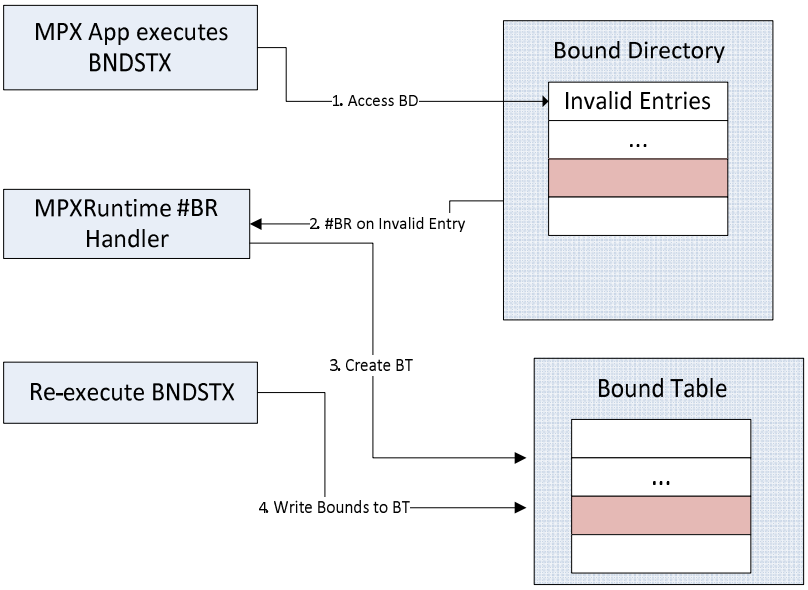

Figure 10. BNDSTX operation

It is important to note the following:

- The Bounds Table entry contains the lower bound, upper bound, and the value of the pointer variable. The address of the pointer variable is only used to locate the specific Bounds Table entry to be used. Hence, two or more pointer variables pointing to the same buffer will have distinct entries in the Bounds Table.

- If a legacy-called routine modifies a pointer variable, it will be detected on return by the calling code. The BNDLDX will detect a mismatch with the pointer value it saved in the Bounds Table entry. In this case, the BNDLDX instruction will set the value of the Bounds Register to INIT, making the subsequent bounds checking ineffective.

7.2.4 Control of the Intel Memory Protection Extensions runtime driver (Windows OS)

Applications that have been compiled with Intel MPX support will have a new section in their Portable Executable (PE) header debug directory information. A 'Flags' dword for controlling the behavior of the Intel MPX runtime driver is included in this section at compile time. The following control flags have been defined:

Table 10. MPX Runtime Control Flags

Flag	Bit	Description

IMAGE_MPX_ENABLE	0	This flag is used to determine whether Intel Memory Protection Extensions (MPX) should be enabled for this application.
IMAGE_MPX_ENABLE_DRIVER_RUNTIME	1	This flag determines whether the driver or a runtime DLL will be providing the Intel MPX runtime support.
IMAGE_MPX_ENABLE_FAST_FAIL_ON_BND_EXCEPTION	2	This flag determines the behavior on a #BR for a bounds violation. If set, the driver will immediately terminate the application. If cleared then the exception will be passed up to the application's exception handler.

In addition to this PE header information, these same flags can be set in registry. The registry key is a per application key located in the existing 'Image File Execution Options' key. The MpxOptionsDrv REG_DWORD value under HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\<appName> contains these flags. The registry values act as an override on the values included in the PE as shown in the following table:

Table 11. PE bit vs. Registry Behavior

PE bit value	Registry bit value	Result (value used by driver)
0	0 or key not present	0
0	1	1
1	0	0
1	1 or key not present	1

7.2.5 Deallocation of bounds tables (Windows OS)

The Intel MPX runtime creates a background thread that periodically walks the Bound Directory entries and deallocates the Bounds Tables that are not committed. Currently, the Windows API 'VirtualQuery' (retrieves information about a range of pages in the virtual address space of the calling process. See MSDN for details) is used to determine if the pages associated with the Bounds Table are committed. The deallocation algorithm may change as memory requirements for Intel MPX are better understood.

7.2.6 Deallocation of bounds tables (Linux OS)

Bounds Tables are deallocated by the OS kernel in response to application deallocation. For example, if an application calls malloc(), populates the Bounds Tables for that malloc(), and later calls free(), the OS kernel will find and free all Bounds Tables associated with the malloc(). Deallocation occurs during system calls like munmap() or when using brk() to shrink the heap.

Figure 11 and the pseudo-code sample that follows provide more details:

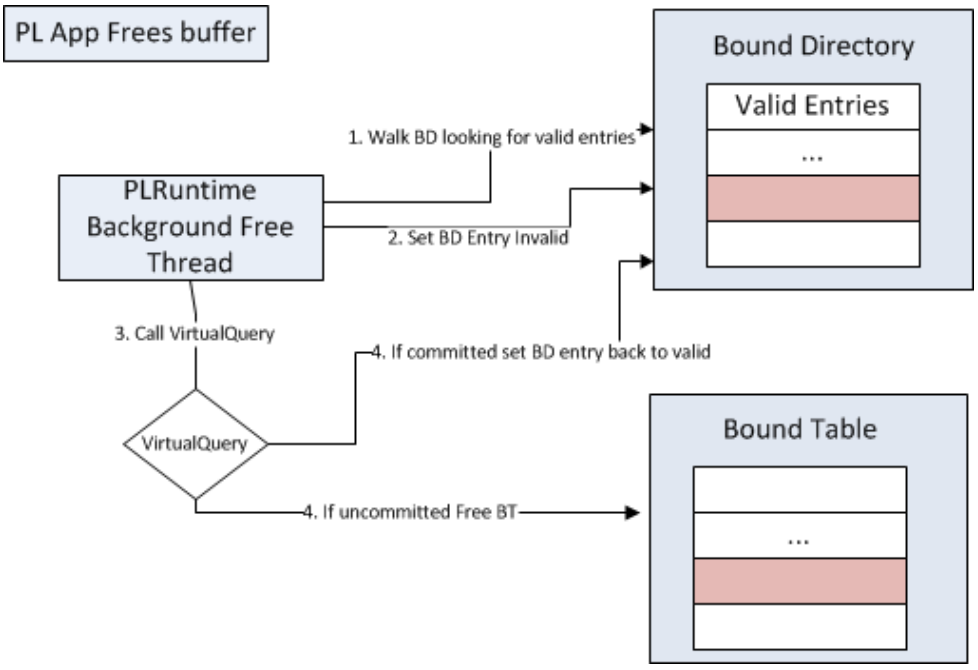

Figure 11. Deallocation of bounds tables

Pseudo-code

```
While (!terminating)
    {
```

```
        Loop through all Directory Entries
        {
                If(entry not valid)
                {
                        Nothing to do, continue entry loop
                }
                Else
                {
                        Acquire CritSection
                        Zero BD entry - to prevent BNDSTX,BNDLDX from
                        accessing this table
                        Use virtualQuery to determine if pages are committed
                        for this table
                        If (committed pages)
                        {
                                Reset BD entry to original value
                        }
                        Else
                        {
                                Free the table if no committed pages
                        }
                        Release critSection
                }
        }
        Sleep for 'x' seconds
}
```

7.3 Compiler support

Table 12 shows the status of enabling for the supported compilers:

Table 12. Supported Compilers - Enabling Status

Compiler	OS	Intel® Memory Protection Extensions	Library
Intel® C++ Composer XE 2013 SP1 (Update 1)	Microsoft Windows*	Validated	loadrt32.obj/ loadrt64.obj

Intel C++ Composer XE 2013 SP1 (Update 1)	Linux*	Not Validated	
GCC 5.1	Microsoft Windows	Not validated	Not planned
GCC 5.1	Linux	Validated	Runtime WIP
Microsoft Visual Studio* 2015 Update 1	Microsoft Windows	Validated	Not supported

Compilers that are enabled for Intel MPX implement the following features:

- Compute the bounds for each pointer used in the code in accordance with C/C++ standards. Some of the bounds may have to be computed at runtime.
- Check the pointer values against their bounds for overflow before using them to access data.
- Manage the Bounds Table entry for each pointer.
- Compute bounds consistent with C/C++ standards. For example, the first variable in a data structure could be used to reference the entire data structure.
- Use the BND prefix for short subroutine calls; return from subroutines and short jumps.
- Pass bounds data along with the pointer parameter for an API call per the enhanced ABI specification.
- Implement various options to
    - Instrument the code for Intel MPX.
    - Omit the lower or upper bound comparison if compiler has sufficient information to conclude that the omitted bound check would not have caused a bounds overflow.
- Allow intrinsics and attributes for custom computation and manipulation of bounds to handle missing or incorrect bounds values usually in the presence of legacy code.
- Modify assembler and linker to support Intel MPX new instructions.

7.3.1 Bounds computation

The following definitions are often used in the context of bounds computation.

- **INIT bounds**. Bounds that allow full memory access. Checks against these bounds always pass for all pointers.
- **NULL bounds**. Bounds that do not allow access to memory. Checks against these bounds always fail for all pointers.

The main problem in enabling Intel MPX in a compiler is the implementation of a data flow analysis for the determination of correct bounds for each dereferenced pointer.

The logic looks for a valid pointer source since pointer arithmetic and type casts do not affect bounds values. The five pointer sources are:

- **Function call**. If the call returns pointer, then the call also returns its bounds.

  Char *p = get_str();

- **Load**. If pointer is loaded from the memory, then bounds are loaded from the Bounds Table.

  Char *str = strings[i];

- **Input function argument**. If pointer is an input argument, then the caller should pass its bounds.

  Void print(char *str);

- **Object address**. If pointer is an address of an object, then the compiler creates bounds using the object address as a low bound and uses its size to compute the upper bound. For objects with incomplete type the compiler must use dynamic bounds computation. The current implementation in GCC for the x86 target uses size relocation to obtain the object size in runtime and compute the corresponding bounds.

  Struc S *p = &obj;

- Field address. When the address of a field in an object is taken, then the compiler applies a narrowing procedure.

  Int *p = &pobj-> int_field;

Other examples of bounds computation:

Example1:

```
    int foo (void *p)
    {
        int *ip = (int *)p; // ip gets bounds passed for input
argument p
        ip += 10; // ip bounds are not changed
        return *ip; // Checks should be made here
```

```
    }
```

Example 2:

```
    int foo (void **p)
    {
        int *ip = (int *)(*p); // ip gets bounds loaded from
bounds table for pointer *p
        return *ip; // Checks should be made here
    }
```

Example 3:

```
    int buf[100];
    int foo (int i)
    {
        int *p = buf; // Bounds [buf, buf + 399] are used
        return p[i]; // Checks should be made here
    }
```

Example 4:

```
    int *getptr ();
    int foo (int i)
    {
        int *p = getptr (); // returned bounds are associated
with p
        return p[i]; // Checks should be made here
    }
```

In some cases the compiler cannot find valid sources of a pointer, such as when non-pointer type is cast into a pointer. By default the compiler does not check such pointers (uses INIT bounds for them). The Intel MPX team is considering having a compiler switch to change this behavior.

7.3.1.1 Narrowing

When an object's field address is taken, the compiler narrows the object's bounds to the field. No narrowing is applied when the address of the array element is taken. The following rules are applied when narrowing takes place (including nested field accesses):

- If there are static array accesses, then bounds of the outermost array are taken.
- If there are no static array accesses, then bounds of the innermost field, which is not the first in the outer object, are taken.

Some compilation flags may affect narrowing.

Example of narrowing result (default behavior):

```
    struct S1
    {
      int f11; //size is 4
      int f12; //size is 4
    };

    struct S2
    {
      S1 f21; //size is 8
      S1 f22[10]; //size is 80
      S1 f23; //size is 8
    };

    struct S2
    {
      S1 f31; //size is 8
      S2 f32[10]; //size is 960
      S2 f33; //size is 96
    };

    S2 s; //size is 1064

    &s.f31.f12; // Bounds are [&s.f31.f12, &s.f31.f12 + 3]
    &s.f31.f11; // Bounds are [&s, &s + 1063]
    &s.f32[5].f22[4].f12; //Bounds are [&s.f32, &s.f32 + 959]
    &s.f33.f22[4].f12; //Bounds are [&s.f33.f22, &s.f33.f22 +
79]
    &s.f33.f23.f11; //Bounds are [&s.f33.f23, &s.f33.f23 + 7]
```

7.3.2 Check pointer against bounds before use

The basic instrumentation for buffer overflow detection consists of the following steps:

- Allocate the buffer and load the start and end addresses of the buffer into a bounds register using the BNDMK instruction.
- Bounds check the pointer value against the lower or upper bound using BNDCL, BNDCU/BNDCN instructions before the pointer value is used to write.
- Bounds violations, if any, are caught by the HW and #BR (Boundary Range) exceptions are raised.

Example:

// s2 is RDX, and s1 in RCX, bounds for s1 in BND0 by calling convention

```
Strcpy(chr *s1, char *s2) {
        While (*s1++ = *s2++) {}
}
        BNDCL       BND0, [rcx]; CHECK S1 (rcx) LB against bounds in BND0
L1      MOVB        RAX, [RDX]; load a char
        INC         RDX
        BNDCU       BND0, [RCX]; check UB for s1 before write
        MOVEB       [RCX], AL
        INC         RCX

        TESTB       AL, AL
        BND  JNE    L1
        BND  RET            ;BND (0Xf2_ prefix is NOP in MPX enabled code
```

7.3.3 BND prefix for branch instructions

An application compiled to use Intel MPX will use the REPNE (0xF2) prefix (denoted by BND) for all forms of near CALL, near RET, near JMP, short and near Jcc instructions (BND+CALL, BND+RET, BND+JMP, BND+Jcc). All far CALL, RET, JMP, and short JMP (JMP rel 8, opcode EB) instructions will never cause bound registers to be initialized.

The impacts of the BNDPRESERVE bit setting on the bounds register after a short branch (Jcc), CALL, and RET are shown in Table 13:

Table 13. Bounds Register INIT Behavior Due To BND Prefix with Branch Instruction

Instruction	Branch Instruction Opcodes	BNDPRESERVE=0	BNDPRESERVE=1
CALL	EB, FF/2	Init BND0-BND3	BND0-BND3 unchanged
BND+CALL	F2 E8, F2 FF/2	BND0-BND3 unchanged	BND0-BND3 unchanged
RET	C2, C3	Init BND0-BND3	BND0-BND3 unchanged
BND+RET	F2 C2, F2 C3	BND0-BND3 unchanged	BND0-BND3 unchanged
JMP	E9, FF/4	Init BND0-BND3	BND0-BND3 unchanged
BND+JMP	F2 E9, FE FF/4	BND0-BND3 unchanged	BND0-BND3 unchanged
Jcc	70 through 7F 0F 80 through 0F 8F	Init BND0-BND3	BND0-BND3 unchanged
BND+Jcc	F2 70 through F2 7F, F2 0F 80 through F2 0F 8F	BND0-BND3 unchanged	BND0-BND3 unchanged

7.3.4 GCC modifications to ABI

Intel MPX is designed to give you the ability to mix instrumented and legacy code. Legacy code does not experience any change in its functionality. Instrumented

applications can link with, call into, or be called from legacy software. You have granular control to provide protection to higher priority modules first. When mixing instrumented and legacy code, keep the following rules in mind:

- When instrumented code is called from legacy code, it gets INIT bounds for all incoming arguments.
- When legacy code returns pointer, INIT bounds are returned for that pointer.
- When legacy code changes pointer in memory, instrumented code gets INIT bounds when this pointer is loaded.

Your code will meet the first two conditions by using a proper ABI that is backward compatible with the legacy one. In GCC, the bounds registers BND0-3 are used to pass the bounds associated with the first four pointer parameters. From fifth pointer parameter onwards, the pointer and bounds are spilled onto a Bounds Table entry using the BNDSTX instruction and a pointer to the entry is passed on the stack.

For the last condition, use a Bounds Table. If legacy code changes a pointer, when you request bounds for new pointer value, the BNDLDX instruction detects pointer change and returns INIT bounds.

The following is an example of bounds passing

```
Extern void func (int *p1, int *p2, int *p3, int *p4, int *p5, intx, int *p6);


func (p1, p2, p3, p4, p5, x, p6);
```

Table 14 shows how the various bounds are passed to the function call.

Table 14. Bounds Passing For The Above Call

Bound Registers	Stack Frame Offset for BOUND_MAP_STORE	General Purpose Registers	Stack Frame Offset
%bnd0: p1	-16: p5 [17]	%rdi: p1	0: p6
%bnd1: p2	-24: p6	%rsi: p2	
%bnd2: p3		%rdx: p3	
%bnd3: p4		%rcx: p4	
		%r8: p5	
		%r9: x	

7.3.5 Microsoft Visual C++* modifications to ABI

The x86 and x64 ABI calling conventions need to be extended to account for passing bounds arguments. For x86 we have to consider the various calling conventions, while

for x64 we have a unified *FASTCALL* calling convention. All the bounds registers, BND0 through BND3, are considered volatile and those that are live through the call need to be saved and restored by the caller.

Bounds arguments are conveyed either through bound registers, or through the Bounds Table, so as not to affect the layout of the non-bounds parameters. Updates to the Bounds Table are made using the BNDSTX and BNDLDX instructions.

Function prototypes must be used for functions that use variable number of arguments and are compiled with MPX enabling. This is needed in order to pass the bounds correctly for pointer arguments.

Please consult the MSDN Library articles “Calling Convention” and “Overview of x64 Calling Conventions” listed in section 9, “References”, for more background.

7.3.5.1 X86 calling convention

Only up to two integer or pointer arguments may be passed in registers using any calling convention. Thus the associated bounds of a pointer argument can always be conveyed using a bounds register if the pointer argument is itself conveyed through an integer register, though the converse is not necessarily true.

It must be noted that both (> 8 bytes) large and (<= 8 bytes) small multi-byte arguments are conveyed by value via the stack, and so the bounds for the nested pointers in them thus becoming in-memory pointers need to be conveyed using the Bounds Table as described below.

The ABI extensions for the *CDECL*, *STDCALL*, *THISCALL*, *FASTCALL,* and *VECTORCALL* calling conventions are as follows:

1. The bounds for the first four scalar (hidden or explicit) pointer arguments taken in order are conveyed via the bounds registers. The bound registers BND0 through BND3 are assigned in order.

    a. The hidden arguments are the first set of arguments, thereby offsetting the bound registers assigned to the explicit arguments. For the *THISCALL* calling convention, the first argument is the hidden “*this*” pointer, the conveyed bound for which will be the bound associated with the “*this*” argument. A hidden pointer argument may also be used to return large multi-bytes, the conveyed bound for which will be the bound associated with the large multi-byte.

2. The nested bounds, if any, for multi-byte parameters are conveyed using the Bounds Table. The Bounds Table entry is associated with the address and value of the argument’s stack location containing the associated pointer field. If the

associated nested pointer field partially overlaps with another (as can be created by unions containing unaligned pointer fields) then in order to conservatively ensure correctness, INIT bounds are associated with all such partially overlaying fields that would get mapped to the same Bounds Table entry by BNDSTX address translation.

3. The overflow bound arguments (that do not fit in BND0 through BND3) are conveyed using the Bounds Table. The Bounds Table entry is associated with the address and value of the argument's stack location containing the associated pointer.

4. The return bounds are returned in BND0 (and BND1).

    a. In the case of returning small multi-bytes, BND1 may be additionally used if there is more than one pointer; otherwise, if only one pointer is returned only BND0 is used. The following rules are followed in order to determine the return bounds.

        i. If it contains a single (or multiple exactly overlapping pointers), then the bounds associated with that pointer(s) is conveyed.

        ii. If first four bytes of the return value may contain more than one partially overlapping pointer, then BND0 will contain INIT bounds.

        iii. If first four bytes of the return value may contain any part of one (or multiple exactly overlapping) pointer member(s), then register BND0 will hold bounds for such pointer member(s).

        iv. If first four bytes of the return value may contain any part of a pointer member and second four bytes of the return value contains a pointer member, then register BND1 will contain bounds for pointer members located in the second four bytes of the return value.

        v. Otherwise, BND0 and BND1 are stated to be undefined.

    b. Large multi-bytes are returned via reference with the caller allocating storage and passing the pointer to the storage as a hidden argument. The returned bound in this case will simply be the bounds associated with this hidden argument. The bounds associated with any nested

pointers within large multi-byte return values are conveyed via the Bounds Table as any for any in-memory pointers using BNDTSX. If the associated nested pointer field partially overlaps with another (as can be created by unions), then in order to conservatively ensure correctness, INIT bounds are associated with all such partially overlaying fields that would get mapped to the same Bounds Table entry by BNDSTX address translation.

5. The bound arguments for vararg functions are conveyed like regular function arguments. Additionally, the first four bound arguments are additionally conveyed through the Bounds Table as well. We do this to handle some common but non-standard practices such as using mismatched vararg function declaration with respect to its definition.

The *CLRCALL* calling convention is used for calling managed code that is not supported by MPX and hence there are no ABI extensions defined for this.

7.3.5.2 X64 calling convention

Only up to four integer or pointer arguments may be passed in registers using the unified *FASTCALL* calling convention for x64. Thus the associated bound of a pointer argument can always be conveyed using a bounds register if the pointer argument is itself conveyed through a register, though the converse is not necessarily true.

In contrast to x86, any large multi-byte argument (larger than 8 bytes) is passed via reference, while others are passed via registers if available. As a consequence, any structure containing more than one partially overlapping or non-overlapping native 64-bit pointers will be passed via reference.

It is the responsibility of the caller is responsible for allocating space for parameters to the callee, and must always allocate sufficient space (called the backing storage) for the four register parameters, even if the callee doesn't have that many parameters. Any parameters above the first four must be stored on the stack, above the backing-store for the first four, prior to the call. This aids in the simplicity of conveying bounds vararg C/C++ function arguments (even though the first four pointer arguments are passed in integer registers, we have caller-assigned stack slots to associate the bounds within the Bounds Table).

The ABI extensions for the x64 calling convention are as follows:

1. The bounds for the first four (hidden or explicit) scalar pointers, or small multi-byte nested pointers, or pointers to large multi-bytes arguments taken in order

are conveyed via the bounds registers. The bound registers BND0 through BND3 are assigned in order.

   a. The hidden arguments are the first set of arguments, thereby offsetting the bound registers assigned to the explicit arguments. For the THISCALL calling convention, the first argument is the hidden "*this*" pointer, the conveyed bound for which will be the bound associated with the "*this*" argument. A hidden pointer argument may also be used to return large multi-bytes, the conveyed bound for which will be the bound associated with the large multi-byte.

   b. The following rules are followed, in order, for conveying bounds for nested bounds in small multi-bytes.

      i. If it contains a single (or multiple exactly overlapping pointers), then the bounds associated with that pointer(s) is conveyed.

      ii. If it contains non-native 32-bit pointer(s), then a single INIT bounds is conveyed for the whole small multi-byte.

      iii. Otherwise no nested bounds exist for it.

2. The nested bounds, if any, for large multi-byte parameters are conveyed using the Bounds Table, as these are passed by reference and the nested pointers are thus treated as in-memory pointers. The Bounds Table entry is associated with the address and value of the argument value's pointer field. If the associated nested pointer field partially overlaps with another (as can be created by unions) then, in order to conservatively ensure correctness, INIT bounds are associated with all such partially overlaying fields that would get mapped to the same Bounds Table entry by BNDSTX address translation.

3. The overflow bound arguments (that do not fit in BND0 through BND3) are conveyed using the Bounds Table. The Bounds Table entry is associated with the address and value of the argument's stack location containing the associated pointer.

4. The return bounds are returned in BND0.

   a. The following rules are followed, in order, for conveying bounds for return bounds for small multi-bytes.

i. If it contains a single (or multiple exactly overlapping) pointer(s), then BND0 will contain the bounds associated with that pointer(s).

ii. If it contains a non-native 32-bit pointer, the BND0 will contain INIT bounds.

iii. Otherwise BND0 is stated to be undefined.

b. Large multi-bytes are returned via reference with the caller allocating storage and passing the pointer to the storage as a hidden argument. The returned bound in this case will simply be the bound associated with this hidden argument. The bounds associated with any nested pointers within large multi-byte return values are conveyed via the Bounds Table for any in-memory pointer. If the associated nested pointer field partially overlaps with another (as can be created by unions), then in order to ensure correctness, INIT bounds are associated with all such partially overlaying fields that would get mapped to the same Bounds Table entry by BNDSTX address translation.

5. The bound arguments for vararg functions are conveyed like regular function arguments. Additionally, the first four bound arguments are additionally conveyed through the Bounds Table associating the stack slots assigned to the arguments in the shadow space/overflow area to the conveyed bounds. We do this to handle some common but non-standard practices such as using mismatched vararg function declaration with respect to its definition.

In Microsoft Visual Studio* 2015 Update 1, the ABI extensions related to handling multi-byte arguments are not supported.

7.3.6 Compiler intrinsics

When an application is a mix of instrumented and legacy code, the compiler and linker are unable to compute the bounds values for a pointer in all instances. Intrinsics provide a method of reading, setting, and manipulating the bounds values.

7.3.6.1 Comparison of intrinsics in various compilers

Currently, each compiler has a different set of intrinsics implemented with some of them having the same function. The functionality could merge in the future, but Table 15 shows a comparison of the intrinsics.

Table 15. Intrinsics in GCC, Intel® C/C++ Compiler, and Microsoft Visual C++*

GCC (Linux*)	Intel® C/C++ Compiler (Linux/ Windows*)	Microsoft Visual C++* (Windows)
__bnd_set_ptr_bounds	__chkp_make_bounds	__bnd_set_ptr_bounds
__bnd_narrow_ptr_bounds		__bnd_narrow_ptr_bounds
__bnd_copy_ptr_bounds		__bnd_copy_ptr_bounds
__bnd_init_ptr_bounds	__chkp_kill_bounds	__bnd_init_ptr_bounds
__bnd_null_ptr_bounds		
__bnd_store_ptr_bounds		_bnd_store_ptr_bounds
__bnd_chk_ptr_lbounds		_bnd_chk_ptr_lbounds
__bnd_chk_ptr_ubounds		_bnd_chk_ptr_ubounds
__bnd_chk_ptr_bounds		_bnd_chk_ptr_bounds
__bnd_get_ptr_lbound	__chkp_lower_bound	_bnd_get_ptr_lbound
__bnd_get_ptr_ubound	__chkp_upper_bound	_bnd_get_ptr_ubound
		_bnd_load_ptr_bounds

Table 16 describes the function of the various GCC intrinsics.

Table 16. GCC Intrinsics and Their Functions

void * __bnd_set_ptr_bounds (const void * q, size_t size) Return a new pointer with the value of q, and associate it with the bounds [q, q+size-1]
Example: p = __bnd_set_ptr_bounds (q, 8); //Associate p with bounds [q, q + 7] and value q //Equal to p = q when instrumentation is disabled
void * __bnd_narrow_ptr_bounds (const void *p, const void *q, size_t size) Return a new pointer with the value of p and associate it with the narrowed bounds formed by the intersection of bounds associated with q and the [p, p + size - 1].
Example: r = __bnd_narrow_ptr_bounds (p, q, 8); //Associate pointer r with bounds formed by the intersection (bnd(q), [p, p + 7]) and the value p //Equal to q = r when instrumentation is disabled
void * __bnd_copy_ptr_bounds (const void *q, const void *r)

Return a new pointer with the value of q and associate it with the bounds already associated with pointer r (essentially BNDMOV with pointer association).
Example: p = __bnd_copy_ptr_bounds (q, r); //Associate pointer p with bounds of r and the value q //Equal to p = q when instrumentation is disabled
void * __bnd_init_ptr_bounds (const void *q) Return a new pointer with the value of q and associate it with INIT bounds
Example: p = __bnd_init_ptr_bounds (q); //Associate pointer p with INIT bounds and the value q //Equal to p = q when instrumentation is disabled
void * __bnd_null_ptr_bounds (const void *q) Return a new pointer with the value of q and associate it with NULL bounds
Example: p = __bnd_null_ptr_bounds (q); //Associate pointer p with NULL bounds and the value q //Equal to p = q when instrumentation is disabled
void __bnd_store_ptr_bounds (const void **ptr_addr, const void *ptr_val) Store the bounds associated with pointer ptr_val and location ptr_addr into Bounds Table. This can be useful to propagate bounds from legacy code without touching the associated pointer's memory when pointers were copied as integers.
Example: __bnd_store_ptr_bounds (p, q); //Store the bounds associated with pointer q and location p to Bounds Table. //Ignored when instrumentation is disabled
void __bnd_chk_ptr_lbounds (const void *q) Check if the pointer is within the lower bounds of its associated bounds.
Example: __bnd_chk_ptr_lbounds (q); //Get the bounds associated with q and do lower bound check on it with q //Ignored when instrumentation is disabled
void __bnd_chk_ptr_ubounds (const void *q) Check if the pointer is within the upper bounds of its associated bounds.
Example: __bnd_chk_ptr_ubounds (q); //Get the bounds associated with q and do upper bound check on it with q //Ignored when instrumentation is disabled
void __bnd_chk_ptr_bounds (const void *q, size_t size)

Check that [q, q + size - 1] is within the lower and upper bounds of its associated bounds.
Example: __bnd_chk_ptr_bounds (q, 8); //Get the bounds associated with q and do bounds check on it with [q, q + 7] //Ignored when instrumentation is disabled
const void * __bnd_get_ptr_lbound (const void * q) Return the lower bound (which is a pointer) associated with the pointer q. This is useful for debugging using printf.
Example: lb = __bnd_get_ptr_lbound (q); printf ("lb(q)=%p", lb);
const void * __bnd_get_ptr_ubound (const void * q) Return the upper bound (which is a pointer) associated with the pointer q. This is useful for debugging using printf.
Example: ub = __bnd_get_ptr_ubound (q); printf ("ub(q)=%p", ub);

The pointer checker implemented in the Intel C/C++ compiler supports the following intrinsics as shown in Table 17.

Table 17. Intrinsics Supported in Intel® C/C++ Compiler

`void * __chkp_lower_bound(void **)` Returns the lower bound associated with the pointer.
`void * __chkp_upper_bound(void **)` Returns the upper bound associated with the pointer.
`void * __chkp_kill_bounds(void *p)` Removes the bounds information to allow the pointer specified in the argument to access all memory. Use this function for a pointer from a non-enabled module that will be used in an enabled module where you cannot determine the bounds of the pointer.

The function ensures that the pointer created from a non-enabled module does not inherit the bounds from another pointer that was in the same memory address. The return value is a pointer without bounds information.
`void * __chkp_make_bounds(void *p, size_t size)` Creates new bounds information within the allocated memory address for the pointer in the argument, replacing any previously associated bounds information. The new bounds are: `p = __chkp_make_bounds(q, size)` `// lower_bound(p) = (char *)q` `// upper_bound(p) = lower_bound(p) + size`

The file chkp.h defines intrinsic and reporting functions. The header file is located in the `<install-dir>\include` directory.

`void plrt_report_control( __chkp_report_option_t,__chkp_callback_t)` determines how errors are reported.

Table 18 describes the intrinsics supported in Microsoft Visual C++.

Table 18. Intrinsics Supported by the Visual C++* Compiler

void* _bnd_set_ptr_bounds(const void* q, size_t size) *Synopsis*: It returns a pointer with the value q associated with the bounds [q, q+size-1]. It will generate a BNDMK instruction. Generally, it can be used after a pointer is allocated memory or an array is defined. *Example*: `p = _bnd_set_ptr_bounds(q, 8)` will associate p with bounds [q,q+7] and value q.
void* _bnd_init_ptr_bounds(const void* q) *Synopsis*: It returns a pointer with the value q and makes it unbounded. It generates a BNDMK instruction.

Example: `p = _bnd_init_ptr_bounds(q)` will make p an unbounded pointer and assign the value q to it.

void* _bnd_copy_ptr_bounds(const void *q, const void *r)

Synopsis: It returns a pointer with the value q and associates it with the bounds corresponding to pointer r. It is useful for propagating bounds.
Example: `p = _bnd_copy_ptr_bounds(q, r)` will associate p with bounds corresponding to r and value q.

void* _bnd_narrow_ptr_bounds(const void* q, const void* r, size_t size)

Synopsis: It returns a pointer with the value q and associates it with the bounds formed by the intersection of bounds associated with r and [q, q+size-1].
Example: `p = _bnd_narrow_ptr_bounds(q, r, 8)` will associate p with bounds corresponding to r intersected with [q, q+7] and the value q.

void _bnd_chk_ptr_bounds(const void *q, size_t size)

Synopsis: It checks if memory address range [q, q+size-1] is within the lower and upper bounds of the bounds associated with pointer q using BNDCL and BNDCU instructions.
Example: `_bnd_chk_ptr_bounds(q, 8)` will verify that [q, q+7] is within the bounds associated with q. It does BNDCL with q and BNDCU with q+7.

void _bnd_chk_ptr_lbounds(const void *q)

Synopsis: It checks that the address pointed to by q is at or above the lower bound of its associated bounds using the BNDCL instruction.
Example: `_bnd_chk_ptr_lbounds(q)` will verify that [q] is at or above the lower bound associated with q.

void _bnd_chk_ptr_ubounds(const void *q)

Synopsis: It checks that the address pointed to by q is at or below the upper bound of its associated bounds using the BNDCU instruction.
Example: `_bnd_chk_ptr_ubounds(q)` will verify that [q] is at or below the upper bound associated with q.

void _bnd_store_ptr_bounds(const void **ptr_addr, const void *ptr_val)

Synopsis: Store the bounds associated with in-memory pointer ptr_val stored at address ptr_addr into the Bounds Table using BNDSTX instruction. The Bounds Table entry address is calculated using ptr_addr.
Example: `_bnd_store_ptr_bounds(p, q)` will generate a BNDSTX to store bounds associated with q to Bounds Table. Here *p = q.

void* _bnd_load_ptr_bounds(const void **ptr_addr, const void *ptr_val) *Synopsis:* Load the bounds associated with in-memory pointer ptr_val stored at address ptr_addr from Bounds Table using BNDLDX instruction. It returns a pointer with value ptr_val and associates it with the loaded bounds. The Bounds Table entry address is calculated using ptr_addr. *Example*: `p = _bnd_load_ptr_bounds(q, r)` will generate a BNDLDX to load bounds associated with r from Bounds Table and associates p with the loaded bounds and the value r. Here *q=r.
const void* _bnd_get_ptr_lbound(const void* q) *Synopsis:* It returns a pointer whose value is the lower bound associated with q. It also associates the returning pointer with the bounds corresponding to q. *Example*: `lb = _bnd_get_ptr_lbound(q)` retrieves the lower bound associated with q and stores it in lb. This involves storing the bounds onto the stack (BNDMOV) and retrieving the lower bound (MOV).
const void* _bnd_get_ptr_ubound(const void* q) *Synopsis:* Get the upper bound from the bounds register regnum and return a pointer with that value. *Example*: `ub = _bnd_get_ptr_ubound(q)` retrieves the upper bound associated with q and stores it in ub. This involves storing the bounds onto the stack (BNDMOV), but the upper bound is stored as one's complement of the actual upper bound address. It needs to be inverted (NOT) after retrieving (MOV) it.

7.3.7 GCC compiler attributes

Table 19 is the list of attributes added to GCC for additional guidance to the compiler.

Table 19. Attributes Supported by GCC for Intel®

bnd_legacy Used to prevent generating BND prefix and parameter passing code for a legacy function call. Also used to prevent instrumentation for selected functions. Example: `__attribute__((bnd_legacy)) int* legacy_function (int*);` `<some code>`

```
      int *p = legacy_function (q); //No BND prefix for
CALL and no bounds arguments passed
```

Example:

```
      __attribute__((bnd_legacy)) int* legacy_function
(int* p) //No incoming bounds
      {
          <some_code>
          *p = 0; //No bounds checks
          <some code>
          return p; //No BND prefix for RET and no bounds
returned
      }
```

bnd_variable_size

This attribute is used to mark variable-sized fields in objects.

Example:

```
struct dyn_data
{
  int additional_data_length;
  char contents[4] __attribute__((bnd_variable_size));
//No narrowing for this field
};
```

bnd_instrument

This attribute is used to mark functions to instrument in case -fchkp-instrument-marked-only is used.

7.3.8 GCC Compiler Options

Table 20 describes the Intel MPX options supported in GCC.

Table 20. Compiler Options Supported by GCC

Compiler Options to Control Intel® Memory Protection Extensions (Intel® MPX) Instrumentation	
Compiler Switches	**Function**
-fcheck-pointer-bounds	Enables pointer bounds checking. At the moment there is only Intel MPX implementation on Intel® architecture. Other implementation for non-Intel architecture may be available in the future.
-fchkp-check-incomplete-type	Adds instrumentation for use of variables with incomplete type (otherwise INIT

	bounds are used for such objects). By default is enabled when -fcheck-pointer-bounds is specified.
-fchkp-zero-input-bounds-for-main	Uses zero bounds for all incoming arguments in 'main' function. Used for debugging purposes. Your setup is probably wrong if you need it. Disabled by default.
-fchkp-treat-zero-size-reloc-as-infinite	Forces dynamic zero size to be treated as infinite. Can be useful if you link with legacy objects with incorrect size information. Disabled by default.
-fchkp-first-field-has-own-bounds	Forces instrumentation to use narrowed bounds for address of the first field in the structure. By default pointer to the first field has the same bounds as pointer to the whole structure.
-fchkp-narrow-bounds	Controls how Pointer Bounds Checker handles pointers to object fields. When narrowing is on, field bounds are used. Otherwise full object bounds are used.
-fchkp-narrow-to-innermost-array	Forces instrumentation to use bounds of the innermost arrays in the case of nested static arrays access. By default outermost array is used.
-fchkp-optimize	Allows optimizations of instrumentation. By default allowed on optimization levels >0.
-fchkp-use-fast-string-functions	Allows usage of *_nobnd versions (which assumes there is no bounds copying) of string functions when optimizing instrumentation. Disabled by default.
-fchkp-use-nochk-string-functions	Allows usage of *_nochk versions (which do not check memory accesses) of string functions when optimizing instrumentation. Disabled by default.
-fchkp-use-static-bounds	Creates and uses static bounds objects instead of creating them dynamically each time it is required. Enabled when -fcheck-pointer-bounds is specified.

-fchkp-use-static-const-bounds	Uses static variables for constant bounds rather than generate them each time it is required.
-fchkp-check-read	Generates checks for all read accesses to memory. Enabled by default when -fcheck-pointer-bounds is specified.
-fchkp-check-write	Generates checks for all write accesses to memory. Enabled by default when -fcheck-pointer-bounds is specified.
-fchkp-store-bounds	Generates bounds stores for pointer writes. Enabled by default when -fcheck-pointer-bounds is specified.
-fchkp-instrument-calls	Generates bounds passing for calls. Enabled by default when -fcheck-pointer-bounds is specified.
-fchkp-instrument-marked-only	Instruments only functions marked with bnd_instrument attribute.

7.3.9 GCC compiler macros

The option **-fcheck-pointer-bounds** enables pointer bounds checking. Currently, Intel MPX is only implemented on Intel architecture. The compiler defines the following macros:

__CHKP__ is defined when the -fcheck-pointer-bounds flag is passed

__Intel MPX__ is defined when the -mmpx flag is passed

7.3.10 Intel® C/C++ compiler options

Intel C/C++ Compiler supports two types of bounds checking: SW-based bounds checking and HW-accelerated bounds checking for Intel MPX. They are selected as shown in Table 21.

Table 21. Options to Select Hardware-Based Intel® Memory Protection Extensions (Intel® MPX) and SW-Based Pointer Checker

Element	Description
`[Q]check-pointers` `[Q]check-pointers-mpx`	Enables the pointer checker and adds the associated libraries (SW-based bounds checking).

	These compiler options enable checking of all indirect accesses through pointers and accesses to arrays. The `[Q]check-pointers-mpx` option generates code that uses the Intel MPX for performance acceleration. On systems without Intel MPX, the pointer checker features will not function, but the application will execute normally. If both options are set, `[Q]check-pointers-mpx` will take preference. The option keywords are `[none | write | rw]`. The pointer checker is off by default. Specify `none` to disable the pointer checker. Specify `write` to check bounds for writes through pointers only. Specify `rw` to check bounds for both reads and writes through pointers. If the compiler determines that an access is safe during optimization, then the compiler removes the pointer checking code. See Checking Bounds.

7.3.11 Microsoft Visual C++ extended compiler attribute for Intel Memory Protection Extensions

Microsoft Visual Studio 2015 Update 1 supports Intel MPX intrinsic functions. These intrinsic functions also bring the ABI extensions with them. This causes each and every function to pass around bounds (mostly init bounds) even if they do not use Intel MPX intrinsics. This can have a detrimental impact on code size and performance. To mitigate this impact, we add a __declspec(mpx) whose functionality is explained in the table below.

The syntax and behavior of __declspec(mpx) is explained in Table 22.

Table 22. Microsoft Visual C++* Extended Compiler Attribute for Intel® Memory Protection Extensions

Syntax

__declspec(mpx) declarator
Usage __declspec(mpx) int func(formal_parameters) {}

A function is marked with __declspec(mpx) 1. This function receives bounds from other functions calling it as parameters. 2. This function passes bounds on calls and receives bounds returned only if the callee is also marked with __declspec(mpx). This assumes that we have headers of all library functions that are MPX enabled and marked with __declspec(mpx). 3. This function returns bounds to the calling functions. The above actions happen irrespective of the presence of MPX intrinsics in the function.	**Example:** __declspec(mpx) void* bar(void* ptr) { …… } void* bar2(void* ptr) {} __declspec(mpx) void* foo(void* ptr) { *// Will receive bounds* void* p; …. p = bar(ptr); *// Will pass and receive bounds* bar2(ptr); *// Will not pass bounds* …. return p; *// Will return bounds* }
A function is not marked with __declspec(mpx) 1. This function does not receive bounds from other functions calling it as parameters, irrespective of the presence of MPX intrinsics. 2. This function passes bounds on calls and receives bounds returned only if the callee is also marked with __declspec(mpx). This assumes that we have headers of all library functions that are MPX enabled and marked with __declspec(mpx). This function does not return bounds to the function calling it.	**Example:** __declspec(mpx) bar(void* ptr) { …… } void* bar2(void* ptr) {} foo(void* ptr) { *// Will not receive bounds* void* p; …. p = bar(ptr); *// Will **pass** and receive bounds* bar2(ptr); *// Will **not pass** bounds* …. return p; *// Will not return bounds* }

7.3.12 Microsoft Visual C++ compiler options

Microsoft Visual Studio 2015 Update 1 introduced an experimental compiler option for Intel MPX. Table 23 describes the options supported in Microsoft Visual C++.

Table 23. Microsoft Visual C++* Compiler Options

Compiler Option	Description
`/d2MPX`	This compiler option enables 1. Checks for all memory writes through pointers and arrays for buffer overflows. This provides protection for local and global pointers and arrays. 2. ABI extensions to propagate bounds associated with pointer arguments.

If the user wants to build project files from the command line, then specify the compiler option (/d2MPX) to add MPX instrumentation as follows:

cl /d2MPX sample.cpp

To build project files within Visual Studio 2015, make the following change in project's property pages: click through the sequence:

PROJECT|Properties|Configuration Properties|C/C++|Command Line|Additional Options. In the resulting window, add /d2MPX

The compiler attribute __declspec(mpx) does not have any effect on the MPX flag. Automatic Intel MPX code generation will still happen in a function marked with __declspec(mpx).

The compiler option /d2MPX does not currently include support for C/C++ libraries enabled for Intel MPX, narrowing and protection against buffer overflows through memory reads. Users may use intrinsic functions to wrap library routines and add read checks.

7.4 Enabling Intel Memory Protection Extensions in other components

Assemblers and debuggers should be modified to process the new Intel MPX registers and instructions. The linker requires a new flag to switch on processing of BND-prefixed JMP instructions to enable setting proper bounds values.

GDB is implemented as part of binutils. Debuggers are not supported for the Intel C/C++ Compiler.

Microsoft Visual Studio debugger supports the display and manipulation of the Intel MPX registers via both the register and watch windows as well as within expressions when running on a machine enabled with Intel MPX.

8 Building and executing Intel Memory Protection Extensions applications

This section shows you how to build, validate, and deploy Intel MPX applications and has the following subsections:

- Enabling Intel MPX in an existing C/C++ application
- Enabling Intel MPX in a new application
- Validating an Intel MPX application
- Deploying an Intel MPX application

8.1 Enabling Intel Memory Protection Extensions in an existing C/C++ application

Intel MPX is designed to give you the ability to mix instrumented and legacy code. Legacy code does not experience any change in its functionality. Instrumented applications can link with, call into, or be called from legacy software.

Here are the recommended steps:

1. Build and enable all modules with Intel MPX.
2. Set the Intel MPX session variables to obtain the desired behavior of Intel MPX runtime. On Windows, set the registry settings described in section 7.2.4, “Control of the Intel Memory Protection Extensions runtime driver (Windows OS)”, for an application to control the runtime behavior.
3. Check the following metrics for the application:
    a. Memory bloat resulting from enabling Intel MPX
    b. Performance degradation of key features
    c. Change in power consumption
4. If the above metrics are within acceptable levels, there is no requirement to selectively enable Intel MPX; otherwise, you should enable Intel MPX only in modules susceptive to bounds violations to mitigate the above degradations.
5. Follow these criteria for selecting modules for enabling Intel MPX

a. Modules that (de)allocates memory
b. Modules that alter pointer variables

6. Inspect all cases of false positive reports. They may require source code changes. For example, in some cases the address of the structure's field is used to access the whole structure. In instrumented code it may cause failures due to narrowing. You should use a different way to obtain the field's address in such cases.

Example:

```
    struct S
    {
      int a;
      int b[10];
      int c;
    };

    #define OFFSETOF(s,f) (((char *)(&s)) +
__builtin_offsetof (typeof (s), f))

    void print (int *p, int i)
    {
      printf ("%d\n", p[i]);
    }

    int foo (S &s)
    {
      print ((int *)s.b, 10); //Bounds violation
      print (&s.c, -1); //Bounds violation
      print ((int *)OFFSETOF (s, b), 10); //OK
      print ((int *)OFFSETOF (s, c), -1); //OK
    }
```

8.2 Enabling Intel Memory Protection Extensions in new code

Here are some tips to ensure the compiler accurately sets the BND registers.

1. Ensure allocations, reallocations, and deallocations of memory are done in the same code module. This ensures BND registers are set correctly.
2. Ensure bounds are assigned correctly if custom memory allocation routines are used. Use compiler intrinsic functions to set the bounds, if required. Add test cases to validate bounds settings.

   Example:

```
void *__wrap_malloc (size_t n)
{
    void *p = (void *) malloc (n);
    if (p)
    {
```

```
            return __bnd_set_ptr_bounds (p, n); //bnd: [p, p+n-1]
        }
        return p;
    }
```

3. Do not assign a pointer using cast of an absolute integer. The compiler uses INIT bounds for such pointers, where all bounds checking passes.
4. Avoid having legacy code call instrumented code because the compiler sets INIT bounds, making your bounds checking code ineffective.
5. Avoid casting a field in a structure back to the structure to prevent incorrect bounds value computation at compile time or runtime.
6. If you pass pointers to a subroutine, make them the initial parameters. This ensures that the maximum number of pointers and their bounds are passed using registers and BND registers.

8.3 Validating an Intel Memory Protection Extensions application

No JIT compilers are enabled for Intel MPX; hence all Intel MPX applications should be fully compiled to machine instructions.

The following recommendations will help improve the quality of validation:

1. Ensure the session variables used by Intel MPX runtime are set correctly. On Windows, if the registry settings described in section 7.2.4, “Control of the Intel Memory Protection Extensions runtime driver (Windows OS)”, for an application exist, check if they are set correctly.
2. Some bounds violations could be false positives caused by the compiler improperly narrowing the bounds. The app developer should fix these issues using intrinsics.

8.4 Deploying Intel Memory Protection Extensions applications

Successfully executing Intel MPX applications depends on the following conditions being satisfied; otherwise, the applications execute, but are not protected from buffer overflows:

- The installer should validate the following:
  - Every processor in a multiprocessor system has Intel MPX enabled.
  - The OS has Intel MPX enabled.
  - The installer should ensure that an appropriate Intel MPX runtime is available on the system. If the runtime is not available, the installer should install one. On Windows, check that the Intel MPX Runtime Driver is installed on your Microsoft Windows 10 November 2015 Update or greater system by verifying its presence in Device

Manager under System devices (Figure 12). If it is absent, please download and install the driver from the Intel® Memory Protection Extensions Enabling Guide.

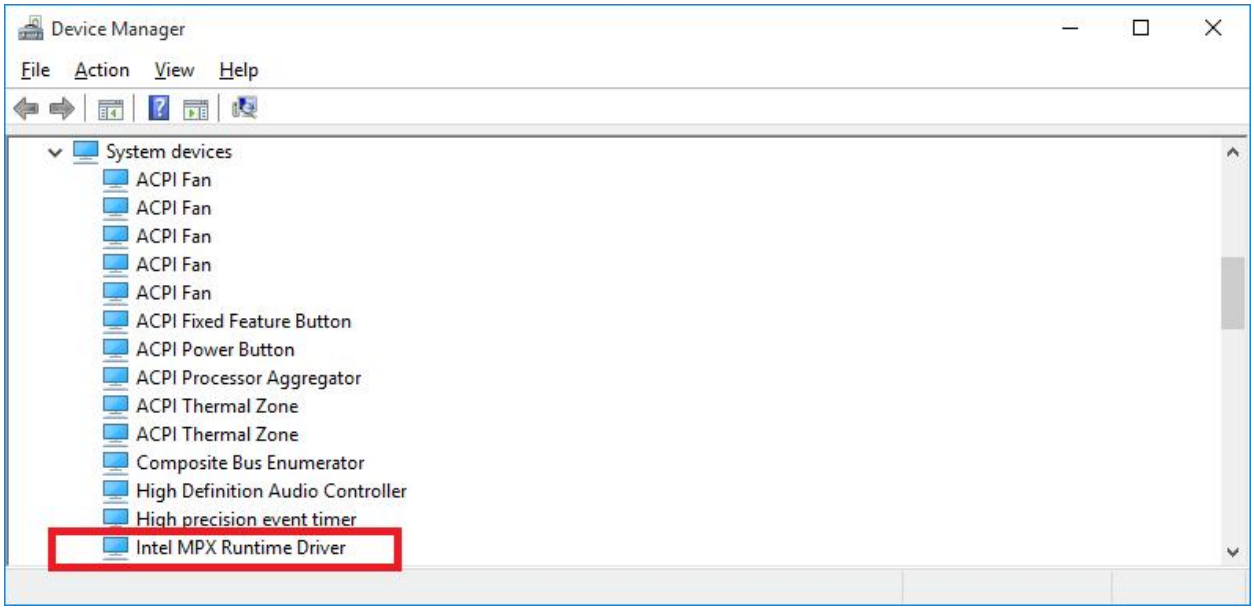

Figure 12. Verify that the Intel® Memory Protection Extensions Runtime Driver is installed via Device Manager

- Create a launcher for the application, which sets the session variables to influence the runtime behavior. This is not required on Microsoft Windows 10 but the registry settings described in section 7.2.4, “Control of the Intel Memory Protection Extensions runtime driver (Windows OS)”, can be set to control the runtime behavior.
- The installer should ensure that sufficient free memory is available on the system.

8.5 Microsoft Windows-specific details for deploying Intel Memory Protection Extensions applications

This section describes the end-to-end process of building sample code containing a buffer overflow with the Microsoft Visual C++ compiler, running it with Microsoft Windows 10 on hardware enabled with Intel MPX and observing the #BR exception in the Visual Studio debugger. We also cover Microsoft Windows-specific details for deploying Intel MPX applications.

The source code of the example is as below:

```
                                        /* sample.cpp */
// mpxexample.cpp
// compile with: /d2MPX
#include "stdafx.h"
#include <windows.h>

const int OUTBUFSIZE = 42;
```

```
wchar_t out[OUTBUFSIZE];
void copyUpper(wchar_t* str, size_t size) {
	__try {
		for (unsigned int i = 0; i < size; i++) {
			// buffer overflow when attempting to write the 43rd wchar
			out[i] = towupper(str[i]);
		}
	}
	__except (GetExceptionCode() == STATUS_ARRAY_BOUNDS_EXCEEDED) {
		wprintf(L"Caught array bounds exceeded exception\n");
	}
}
int main(int argc, char* argv[]) {
	wchar_t str[] = L"the quick brown fox jumps over the lazy dog";
	copyUpper(str, wcsnlen_s(str, 255));
	wprintf(L"%s\n", out);
	return 0;
}
```

Setup required:

- A machine with hardware enabled with Intel MPX.
- Microsoft Windows 10 November 2015 Update or greater. Verify the presence of the Intel MPX runtime driver in the Device Manager under System devices (Figure 12). If it is absent, please download and install the driver from the Intel® Memory Protection Extensions Enabling Guide.
- Microsoft Visual Studio 2015 Update 1.
    - Installing Visual Studio 2015 Update 1 with device emulators based on Microsoft Hyper-V* may mask the Intel MPX state, causing Intel MPX instructions to be treated as NOPs. To verify this, after installing Visual Studio, the user should check the hypervisor settings. To do this, type **bcdedit** in an administrative cmd prompt. Make sure that the **hypervisorlaunchtype** setting is off. If it is set to auto- do **bcdedit /set hypervisorlaunchtype off** and reboot the machine. This issue will be addressed in future.

To build and run this example:

- Build the test with your favorite set of compiler flags either from the command line or within Visual Studio 2015 Update 1. Add the Intel MPX compiler option as explained in section 7.3.12, “Microsoft Visual C++ Compiler Options”.
- To verify if a binary generated by Microsoft Visual C++ compiler is instrumented with Intel MPX:
    - Run the dumpbin tool with /HEADERS option on the binary.
    - Look for an MPX debug directory entry as shown below for a sample binary:

```
Debug Directories

    Time Type        Size      RVA  Pointer
-------- ------- -------- -------- --------
```

```
5633F55B coffgrp    35C 00019F3C   1853C
5633F55B mpx        14 0001A298    18898   Flags:  5
```

  - To double check if the binary has Intel MPX instructions, run the dumpbin tool with /DISASM option on the binary.

- If required, control the runtime behavior by setting the registry flags described in section 7.2.4 , “Control of the Intel Memory Protection Extensions runtime driver (Windows OS)”. For example, to make the application fastfail, set the IMAGE_MPX_ENABLE_FAST_FAIL_ON_BND_EXCEPTION bit.
- Execute the binary with Microsoft Windows 10 on hardware enabled with Intel MPX, which has the OS support for Intel MPX.
- If the #BR exception is not handled using exception blocks, the application will crash. You can debug it in the Visual Studio debugger. Screenshots of the debug capability are shown in Figure 13 and Figure 14. Note that when debugging in Visual Studio, Array bounds exceeded exception should be turned on in Win32 exceptions under Debug|Windows|Exception Settings.

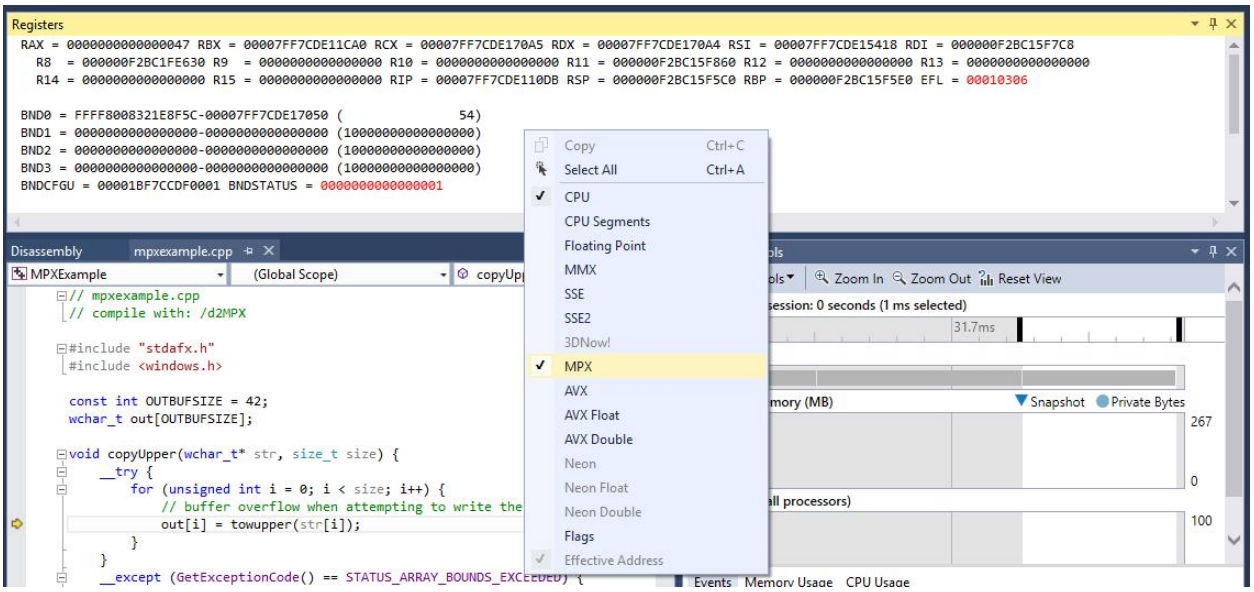

Figure 13. Intel® Memory Protection Extensions bounds registers in the Debugger Register window

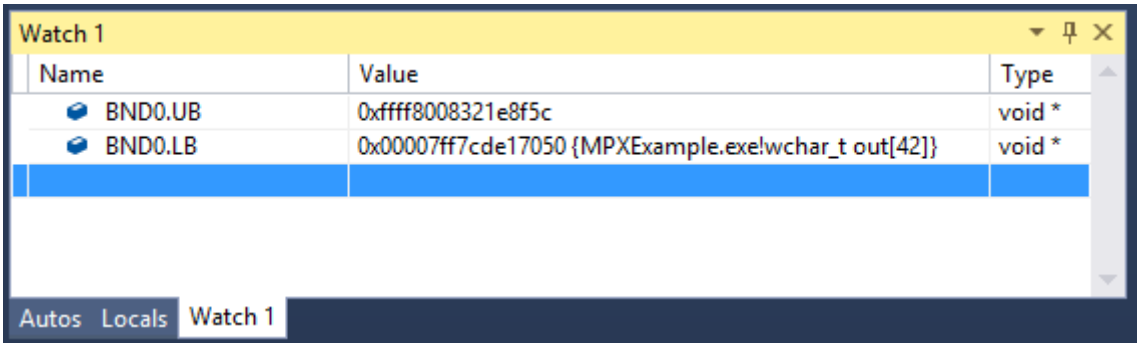

Figure 14. Bounds register to the Debugger Watch window

- The debugger supports display and manipulation of the Intel MPX registers via both the register and watch windows as well as within expressions when running on an Intel MPX-enabled machine. For example, BND0.UB in the watch window refer to upper bound in BND0 register and BND0.LB refers to lower bound. Note that the upper bound is displayed in 2's complement form as show in the images above.
- Note that Visual Studio 2015 Update 1 does not support C/C++ libraries enabled with Intel MPX. The user should wrap the required routines similar to the example shown in section 8.2, "Enabling Intel Memory Protection Extensions in new code", for malloc.

9 References

- Buffer Overflow Protection for buffer on stacks
- Intel Software Developer Manuals
- Intel MPX ABI documentation
- MPX GCC Wiki
- Support of Intel MPX in binutils
- Interactive intrinsics guide
- Pointer Checker in Intel C/C++ Compiler
- Discussion page for Intel MPX
- Intel's instruction set architecture extensions
- MSDN Library: Calling Conventions
- MSDN Library: Overview of x64 Calling Conventions

Notices

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.