Intel® Memory Protection Extensions (Intel® MPX) Enabling Guide

Ramu Ramakesavan, Dan Zimmerman, Pavithra Singaravelu
April 2015

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

1 Contents .................................................................................................................................................. 1
2 License ...................................................................................................................................................... 3
3 Introduction ........................................................................................................................................... 3
4 Origins of buffer overflow vulnerabilities .................................................................................. 3
5 Other solutions ..................................................................................................................................... 4
6 Intel® MPX programming model .................................................................................................... 5
   6.1 Bounds registers and compare instructions ..................................................................... 5
      6.1.1 Instructions to compare bounds ............................................................................... 5
      6.1.2 Instructions to spill and fill bounds registers .......................................................... 6
6.2 CPUID settings ...................................................................................................................................... 7
6.3 Config and Status Registers ........................................................................................................... 9
7 Enabling software for Intel MPX ................................................................................................. 10
   7.1 Enabling the OS ......................................................................................................................... 10
   7.2 Intel® MPX Runtime ................................................................................................................. 11
      7.2.1 Enable Intel MPX for a thread ..................................................................................... 12
      7.2.2 Bound Directory and Bounds Tables .......................................................................... 12
      7.2.3 #BR exception handling ................................................................................................. 14
      7.2.4 Other environment variables used by Intel MPX runtime ......................................... 17
      7.2.5 De-allocation of Bounds Tables (Windows OS) ........................................................... 18
      7.2.6 De-allocation of Bounds Tables (Linux OS) ............................................................... 18
   7.3 Compiler Enabling .................................................................................................................... 20
      7.3.1 Bounds Computation ................................................................................................. 21
      7.3.2 Check pointer against bounds before use ................................................................. 23
      7.3.3 BND prefix for branch instructions ............................................................................. 24
      7.3.4 GCC modifications to ABI .............................................................................................. 25
      7.3.5 Compiler intrinsics ........................................................................................................... 26
      7.3.6 GCC Compiler Attributes ................................................................................................. 30
      7.3.7 GCC Compiler Options ................................................................................................. 31
      7.3.8 GCC Compiler macros .................................................................................................... 32
      7.3.9 Intel® C/C++ Compiler options ................................................................................... 32
7.4 Enabling Intel® MPX in other components

8 Building and Executing Intel® MPX applications

8.1 Enabling Intel® MPX an existing C/C++ application

8.2 Enabling Intel® MPX in new code

8.3 Validating an Intel® MPX application

8.4 Deploying Intel® MPX application

9 References
2 License

The code samples in the sections covering GCC implementation are governed by the GPLv2 license (https://www.gnu.org/licenses/gpl-2.0.txt) and sections covering Intel® C++ Compiler implementation are governed by the Intel Sample Code license (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 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 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 in 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 (https://nvd.nist.gov/) 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 Intel MPX-instrumented code 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® MPX programming model

Intel MPX is a set of processor features that, along with modifications in the OS and Compilers (Assemblers, Linkers) and a new run-time shared library, brings increased robustness to software by checking pointer references whose compile time normal intentions are usurped at run time 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, etc.) 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 bounds of first field in a structure and inner arrays) and missing bounds information (pointer returned from legacy code).

Intel MPX-enabled code 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](image-url)

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 & compare instructions

<table>
<thead>
<tr>
<th>New Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNDMK b, m</td>
<td>Creates LowerBound (LB) and UpperBound (UB) in bounds register b.</td>
</tr>
<tr>
<td>BNDCL b, r/m</td>
<td>Checks the address of a memory reference or address in r against the lower bound.</td>
</tr>
<tr>
<td>BNDCU b, r/m</td>
<td>Checks the address of a memory reference or address in r against the upper bound.</td>
</tr>
<tr>
<td>BNDCN b, r/m</td>
<td>Checks the address of a memory reference or address in r against the upper bound in one's compliment.</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>New Instruction</th>
<th>Function</th>
<th>Performance characteristic</th>
</tr>
</thead>
<tbody>
<tr>
<td>BNDMOV b, b/m</td>
<td>Copy/load LB and UB bounds from memory or a bounds register.</td>
<td>Move the BNDx registers from/to memory, mostly used to spill/fill.</td>
</tr>
<tr>
<td>BNDMOV b/m, b</td>
<td>Store LB and UB bounds in a bounds register to memory or another register.</td>
<td>Move the BNDx registers from/to memory, mostly used to spill/fill.</td>
</tr>
<tr>
<td>BNLDX b, mib</td>
<td>Load bounds using address translation using an sib-addressing expression mib.</td>
<td>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.</td>
</tr>
<tr>
<td>BNSTX mib, b</td>
<td>Store bounds using address translation using an sib-addressing expression mib.</td>
<td>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.</td>
</tr>
</tbody>
</table>

### 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 MPX bit**

<table>
<thead>
<tr>
<th>Intel MPX bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Processor does not support Intel MPX. All Intel MPX instructions are interpreted as NOPs.</td>
</tr>
</tbody>
</table>
| 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 XCRO enable save/restore (XSAVE/XRESTR) of 4 Intel MPX bounds registers, 2 config registers, and 1 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 MPX bits in XCR0 register

<table>
<thead>
<tr>
<th>XCR0 setting</th>
<th>Bit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BNDREGS</td>
<td>for saving and restoring BND0-BND3</td>
</tr>
<tr>
<td></td>
<td>BNDCSR</td>
<td>for saving and restoring the user-mode configuration (BNDCFGU) and the status register BNDESTATUS</td>
</tr>
</tbody>
</table>

Figure 3 Extended Processor State Components for MPX

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

Table 5 CPUID setting and interpretation

<table>
<thead>
<tr>
<th>CPUID Setting</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUID.(EAX=0DH, ECX=0):EAX[3]</td>
<td>XCR0.BNDREGS[bit 3] is supported. BND0-3 should be saved and restored at context switches.</td>
</tr>
<tr>
<td>CPUID.(EAX=0DH, ECX=0):EAX[4]</td>
<td>XCR0.BNDCSR[bit 4] is supported. BNDCFGx and BNDESTATUS should be saved and restored at context switches.</td>
</tr>
<tr>
<td>CPUID.(EAX=0DH, ECX=03H).EAX[31:0] &amp; CPUID.(EAX=0DH, ECX=03H).EBX[31:0].</td>
<td>Size and offset for the bounds registers BND0-3 on the XSAVE/XRESTR stack.</td>
</tr>
<tr>
<td>CPUID.(EAX=0DH, ECX=04H).EAX[31:0] &amp; CPUID.(EAX=0DH, ECX=04H).EBX[31:0]</td>
<td>Size and offset for BNDCFGU, BNDCFGS, and BNDESTATUS on the XSAVE/XRESTR stack.</td>
</tr>
</tbody>
</table>

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](image)

**Table 6 Bounds Config Register interpretation**

<table>
<thead>
<tr>
<th>BNDCFGS/U</th>
<th>Bits</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>En (bit 0)</td>
<td>1 – interpret the Intel MPX instructions as per the definitions 0 – interpret the Intel MPX instructions as NOPs</td>
<td></td>
</tr>
<tr>
<td>Bprv (BNDPRESERVE) (bit 1)</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>Base of Bound Directory (bits 12-63)</td>
<td>Linear address of Bound Directory</td>
<td></td>
</tr>
</tbody>
</table>

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.

```

<table>
<thead>
<tr>
<th>EC</th>
<th>Descriptions</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00b</td>
<td>No Intel MPX exception</td>
<td>Caused by Legacy BOUND instruction</td>
</tr>
<tr>
<td>01b</td>
<td>Bounds violation</td>
<td>#BR caused by BNDCL, BNDCU, or BNDCN instructions; ABD is 0</td>
</tr>
<tr>
<td>10b</td>
<td>Invalid BD entry</td>
<td>#BR caused by BNDLDX or BNDSTX instructions. BD will be set to the linear address of the invalid Bound directory entry</td>
</tr>
<tr>
<td>11b</td>
<td>Reserved</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
```

Figure 5 Layout of Bound Status Register

Table 7 describes the error codes.

Three instances can raise a #BR exception:

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

Intel MPX runtime takes suitable action in each case.

**7 Enabling software for Intel MPX**

Intel MPX should be enabled in the OS and C/C++ compiler components (compiler, assembler, linker, libraries). An 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 instruction BOUNDS.

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

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows* (Desktop)</td>
<td>Enabled in Windows 8.1</td>
</tr>
<tr>
<td>Windows* (Modern UI)</td>
<td>Expected to be enabled in future release</td>
</tr>
<tr>
<td>Linux*</td>
<td>Enabled. Not yet released</td>
</tr>
</tbody>
</table>

### 7.2 Intel® MPX Runtime

Intel MPX runtime is a new SW component required to enable Intel MPX. It is a collection of Intel MPX functions that is usually implemented as a static or shared library, but could be part of an OS.

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. De-allocate 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 MPX Feature Enabling

<table>
<thead>
<tr>
<th>CR4</th>
<th>XCR0</th>
<th>IA32_BNDCFGS</th>
<th>BNDCFGU</th>
<th>Intel® MPX Instruction Behavior</th>
<th>Load/Store to BNDO-BND3, BNDCFGU, BNDSTATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>OXSAVE</td>
<td>BndRegs</td>
<td>BndCSR</td>
<td>Bit0</td>
<td>Bit0</td>
<td>CPL0-2</td>
</tr>
<tr>
<td>0</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NOP</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>NOP</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>NOP</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>NOP</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>MPX</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>MPX</td>
</tr>
</tbody>
</table>

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: 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 Intel MPX-enabled application is launched (see Figure 7):

- Kernel creates 1st thread.
- MpxRuntime.DLL receives DLL_PROCESS_ATTACH.
  - 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.DLL receives DLL_THREAD_ATTACH.

2. Update BNDConfig register to enable Intel MPX for the thread.

![Diagram of MPX initialization for threads]

*Figure 7 MPX initialization for threads*

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.

### 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 BOUNDS command.

#### 7.2.3.1 Exception caused by BNDCL/U/N instructions

Intel MPX runtime for the Intel® C/C++ compiler allows you to declare a handler routine using the following API (similar runtime is being developed for Linux):

```c
Void __chkp_report_control (__chkp_report_option_t option, __chkp_callback_t callback);
```

Instead of a handler routine, you can specify one of the several standard actions defined in Table 10. One of the standard actions, __CHKP_REPORT_USE_ENV_VAR, can be
used to determine the action based on the environment/session variable INTEL_CHKP_REPORT_MODE.

Table 10 Configurable standard handler functions

<table>
<thead>
<tr>
<th>_chkp_report_option_t</th>
<th>How #BR error is reported</th>
</tr>
</thead>
<tbody>
<tr>
<td>_CHKP_REPORT_NONE</td>
<td>Do nothing</td>
</tr>
<tr>
<td>_CHKP_REPORT_BPT</td>
<td>Execute a breakpoint interrupt</td>
</tr>
<tr>
<td>_CHKP_REPORT_LOG</td>
<td>Log the error and continue</td>
</tr>
<tr>
<td>_CHKP_REPORT_TERM</td>
<td>Log the error and exit the program</td>
</tr>
<tr>
<td>_CHKP_REPORT_CALLBACK</td>
<td>Call a user-defined function</td>
</tr>
<tr>
<td>_CHKP_REPORT_TRACE_BPT</td>
<td>Print a traceback and emit Break Point Interrupt</td>
</tr>
<tr>
<td>_CHKP_REPORT_TRACE_LOG</td>
<td>Print a trace back and continue</td>
</tr>
<tr>
<td>_CHKP_REPORT_TRACE_TERM</td>
<td>Print a trace back and exit the program</td>
</tr>
<tr>
<td>_CHKP_REPORT_TRACE_CALLBACK</td>
<td>Print a trace back and call a user function</td>
</tr>
<tr>
<td>_CHKP_REPORT_OOB_STATS</td>
<td>Emit statistics on OOB errors (currently this is just a count of the OOB errors)</td>
</tr>
<tr>
<td>_CHKP_REPORT_USE_ENV_VAR</td>
<td>Use value in environment variable INTEL_CHKP_REPORT_MODE to set the report mode. If the environment variable is not set, the default report mode is used.</td>
</tr>
</tbody>
</table>

Sample Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include "plrt-chkp.h"

void boundViolation(char *addr, char *lb, char *ub, char *ref, size_t size){
    printf("Lower bound is: 0x%p\n", lb);
    printf("Upper bound is: 0x%p\n", ub);
    exit(255);
}

int main(){
    char inputArray[11]="Intel MPXexample";
    char buffer[4];
    /*call user-defined function on a #BR exception*/
    plrt_report_control(_CHKP_REPORT_CALLBACK,boundViolation);
    /*Bounds violation on copying*/
    strcpy(buffer, inputArray);
    return 1;
}
```

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. See Figure 8 for an example.

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 below 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 because a Bounds Table has 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](image)

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 Other environment variables used by Intel MPX runtime

Table 11 lists the other environment variables you can use to control the Intel MPX runtime.
Table 11 Environment Variables to control MPX runtime

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHKP_RT_OUT_FILE</td>
<td>Set output file for info and debug [default: stdout]</td>
</tr>
<tr>
<td>CHKP_RT_ERR_FILE</td>
<td>Set output file for error [default: stderr]</td>
</tr>
<tr>
<td>CHKP_RT_VERBOSE</td>
<td>Set verbosity type [default: 2]</td>
</tr>
<tr>
<td></td>
<td>0 - print only internal run-time errors</td>
</tr>
<tr>
<td></td>
<td>1 - just print summary</td>
</tr>
<tr>
<td></td>
<td>2 - print summary and bounds violation information</td>
</tr>
<tr>
<td></td>
<td>3 - print debug information</td>
</tr>
<tr>
<td>CHKP_RT_MODE</td>
<td>Set Intel MPX runtime behavior on #BR exception. [default: stop]</td>
</tr>
<tr>
<td>CHKP_RT_ADDPID</td>
<td>Generate out, err file for each process. The generated file will be CHKP_RT_{OUT,ERR}_FILE.pid [default: no]</td>
</tr>
<tr>
<td>CHKP_RT_BNDPRESERVE</td>
<td>Set value for the BNDPRESERVE bit:</td>
</tr>
<tr>
<td></td>
<td>BNDPRESERVE = 0 flush bounds on unprefixed call/ret/jmp</td>
</tr>
<tr>
<td></td>
<td>BNDPRESERVE = 1 do NOT flush bounds [default: 0]</td>
</tr>
<tr>
<td>CHKP_RT_PRINT_SUMMARY</td>
<td>Print summary at the end of the run [default: no].</td>
</tr>
<tr>
<td>CHKP_RT_HELP</td>
<td>Print this help and exit</td>
</tr>
</tbody>
</table>

7.2.5 De-allocation of Bounds Tables (Windows OS)

The Intel MPX runtime creates a daemon that walks the Bound Directory entries and de-allocates 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 de-allocation algorithm may change as memory requirements for Intel MPX are better understood.

7.2.6 De-allocation of Bounds Tables (Linux OS)

Bounds Tables are de-allocated by the OS kernel in response to application de-allocation. 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(). De-allocation 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
      }
  }
}

PL App Frees buffer
Release critSection
}  
}  
Sleep for 'x' seconds
}  

7.3 Compiler Enabling

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

<table>
<thead>
<tr>
<th>Compiler</th>
<th>OS</th>
<th>Intel® MPX</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® C++ Composer XE 2013 SP1 (Update 1)</td>
<td>Windows*</td>
<td>Validated</td>
<td>loadrt32.obj/loadrt64.obj</td>
</tr>
<tr>
<td>Intel C++ Composer XE 2013 SP1 (Update 1)</td>
<td>Linux*</td>
<td>Not Validated</td>
<td></td>
</tr>
<tr>
<td>GCC 5.1</td>
<td>Windows</td>
<td>Not validated</td>
<td>Not planned</td>
</tr>
<tr>
<td>GCC 5.1</td>
<td>Linux</td>
<td>Validated</td>
<td>Runtime WIP</td>
</tr>
</tbody>
</table>

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.
  ```c
  Char *p = get_str();
  ```

- **Load**. If pointer is loaded from the memory, then bounds are loaded from the Bounds Table.
  ```c
  Char *str = strings[i];
  ```

- **Input function argument**. If pointer is an input argument, then the caller should pass its bounds.
  ```c
  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 for the x86 target uses size relocation to obtain the object size in runtime and compute the corresponding bounds.
  ```c
  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:

**Example 1:**

```c
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:**

```c
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:**

```c
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:**

```c
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. It happens 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):**

```c
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, BNDUCU/BNDUCN 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:

```c
// 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>
<thead>
<tr>
<th>Instruction</th>
<th>Branch Instruction Opcodes</th>
<th>BNDPRESERVE=0</th>
<th>BNDPRESERVE=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>EB, FF/2</td>
<td>Init BND0-BND3</td>
<td>BND0-BND3 unchanged</td>
</tr>
<tr>
<td>BND+CALL</td>
<td>F2 E8, F2 FF/2</td>
<td>BND0-BND3 unchanged</td>
<td>BND0-BND3 unchanged</td>
</tr>
<tr>
<td>RET</td>
<td>C2, C3</td>
<td>Init BND0-BND3</td>
<td>BND0-BND3 unchanged</td>
</tr>
</tbody>
</table>
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

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

```c
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

<table>
<thead>
<tr>
<th>Bound Registers</th>
<th>Stack Frame Offset for</th>
<th>General Purpose Registers</th>
<th>Stack Frame Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>%bnd0: p1</td>
<td>-16:</td>
<td>%rdi: p1</td>
<td>0: p6</td>
</tr>
<tr>
<td>%bnd1: p2</td>
<td>-24:</td>
<td>%rsi: p2</td>
<td></td>
</tr>
<tr>
<td>%bnd2: p3</td>
<td></td>
<td>%rdx: p3</td>
<td></td>
</tr>
<tr>
<td>%bnd3: p4</td>
<td></td>
<td>%rcx: p4</td>
<td></td>
</tr>
</tbody>
</table>

7.3.5 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.

7.3.5.1 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 and Intel® C/C++ Compiler

<table>
<thead>
<tr>
<th>GCC (Linux*)</th>
<th>Intel® C/C++ Compiler (Linux/ Windows*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>__bnd_set_ptr_bounds</td>
<td>__chkp_make_bounds</td>
</tr>
<tr>
<td>__bnd_narrow_ptr_bounds</td>
<td></td>
</tr>
<tr>
<td>__bnd_copy_ptr_bounds</td>
<td></td>
</tr>
<tr>
<td>__bnd_init_ptr_bounds</td>
<td>__chkp_kill_bounds</td>
</tr>
<tr>
<td>__bnd_null_ptr_bounds</td>
<td></td>
</tr>
<tr>
<td>__bnd_store_ptr_bounds</td>
<td></td>
</tr>
<tr>
<td>__bnd_chk_ptr_lbounds</td>
<td></td>
</tr>
<tr>
<td>__bnd_chk_ptr_ubounds</td>
<td></td>
</tr>
<tr>
<td>__bnd_chk_ptr_bounds</td>
<td></td>
</tr>
<tr>
<td>__bnd_get_ptr_lbound</td>
<td>__chkp_lower_bound</td>
</tr>
<tr>
<td>__bnd_get_ptr_ubound</td>
<td>__chkp_upper_bound</td>
</tr>
</tbody>
</table>

Table 16 describes the function of the various GCC intrinsics.
### Table 16 GCC intrinsics and their functions

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>void * __bnd_set_ptr_bounds (const void * q, size_t size)</strong></td>
<td>Return a new pointer with the value of q, and associate it with the bounds [q, q+size-1].</td>
<td>$p = __bnd_set_ptr_bounds (q, 8);$ //Associate p with bounds [q, q + 7] and value q $//Equal to p = q when instrumentation is disabled</td>
</tr>
<tr>
<td>**void * __bnd_narrow_ptr_bounds (const void <em>p, const void <em>q, size_t size)</em></em></td>
<td>Return a new pointer with the value of p and associate it with the narrowed bounds formed by the intersection of bounds associated with q and the [p, p + size - 1].</td>
<td>$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 $//Equal to q = r when instrumentation is disabled</td>
</tr>
<tr>
<td>**void * __bnd_copy_ptr_bounds (const void <em>q, const void <em>r)</em></em></td>
<td>Return a new pointer with the value of q and associate it with the bounds already associated with pointer r (essentially BNDMOV with pointer association).</td>
<td>$p = __bnd_copy_ptr_bounds (q, r);$ //Associate pointer p with bounds of r and the value q $//Equal to p = q when instrumentation is disabled</td>
</tr>
<tr>
<td>*<em>void * __bnd_init_ptr_bounds (const void <em>q)</em></em></td>
<td>Return a new pointer with the value of q and associate it with INIT bounds.</td>
<td>$p = __bnd_init_ptr_bounds (q);$ //Associate pointer p with INIT bounds and the value q $//Equal to p = q when instrumentation is disabled</td>
</tr>
<tr>
<td>*<em>void * __bnd_null_ptr_bounds (const void <em>q)</em></em></td>
<td>Return a new pointer with the value of q and associate it with NULL bounds.</td>
<td>$p = __bnd_null_ptr_bounds (q);$ //Associate pointer p with NULL bounds and the value q $//Equal to p = q when instrumentation is disabled</td>
</tr>
<tr>
<td>**void __bnd_store_ptr_bounds (const void *<em>ptr_addr, const void <em>ptr_val)</em></em></td>
<td>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.</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void * __chkp_lower_bound(void **)</td>
<td>Returns the lower bound associated with the pointer.</td>
</tr>
<tr>
<td>void * __chkp_upper_bound(void **)</td>
<td>Returns the upper bound associated with the pointer.</td>
</tr>
<tr>
<td>void * __chkp_kill_bounds(void *p)</td>
<td>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.</td>
</tr>
<tr>
<td>void * __chkp_make_bounds(void *p, size_t size)</td>
<td>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:</td>
</tr>
</tbody>
</table>

\[
p = __chkp_make_bounds(q, size)
\]

// 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.

7.3.6 GCC Compiler Attributes

Table 18 is the list of attributes added to GCC for additional guidance to the compiler.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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. |
### 7.3.7 GCC Compiler Options

Table 19 describes the Intel MPX options supported in GCC.

#### Table 19 Compiler options supported by GCC

<table>
<thead>
<tr>
<th>Compiler Options to control Intel® MPX instrumentation</th>
<th>Compiler Switches</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>-fcheck-pointer-bounds</td>
<td>Enables pointer bounds checking. At the moment there is only Intel MPX implementation on Intel® architecture (IA). Other implementation for non-IA may be available in the future.</td>
<td></td>
</tr>
<tr>
<td>-fchkp-check-incomplete-type</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>-fchkp-zero-input-bounds-for-main</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>-fchkp-treat-zero-size-reloc-as-infinite</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>-fchkp-first-field-has-own-bounds</td>
<td>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.</td>
<td></td>
</tr>
<tr>
<td>-fchkp-narrow-bounds</td>
<td>Controls how Pointer Bounds Checker handles pointers to object fields. When narrowing is on, field bounds are used. Otherwise full object bounds are used.</td>
<td></td>
</tr>
<tr>
<td>-fchkp-narrow-to-innermost-array</td>
<td>Forces instrumentation to use bounds of the innermost arrays in the case of nested static arrays access. By default outermost array is used.</td>
<td></td>
</tr>
<tr>
<td>-fchkp-optimize</td>
<td>Allows optimizations of instrumentation. By</td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td><code>-fchkp-use-fast-string-functions</code></td>
<td>Allows usage of *_nobnd versions (which assumes there is no bounds copying) of string functions when optimizing instrumentation. Disabled by default.</td>
<td></td>
</tr>
<tr>
<td><code>-fchkp-use-nochk-string-functions</code></td>
<td>Allows usage of *_nochk versions (which do not check memory accesses) of string functions when optimizing instrumentation. Disabled by default.</td>
<td></td>
</tr>
<tr>
<td><code>-fchkp-use-static-bounds</code></td>
<td>Creates and uses static bounds objects instead of creating them dynamically each time it is required. Enabled when <code>-fcheck-pointer-bounds</code> is specified.</td>
<td></td>
</tr>
<tr>
<td><code>-fchkp-use-static-const-bounds</code></td>
<td>Uses static variables for constant bounds rather than generate them each time it is required.</td>
<td></td>
</tr>
<tr>
<td><code>-fchkp-check-read</code></td>
<td>Generates checks for all read accesses to memory. Enabled by default when <code>-fcheck-pointer-bounds</code> is specified.</td>
<td></td>
</tr>
<tr>
<td><code>-fchkp-check-write</code></td>
<td>Generates checks for all write accesses to memory. Enabled by default when <code>-fcheck-pointer-bounds</code> is specified.</td>
<td></td>
</tr>
<tr>
<td><code>-fchkp-store-bounds</code></td>
<td>Generates bounds stores for pointer writes. Enabled by default when <code>-fcheck-pointer-bounds</code> is specified.</td>
<td></td>
</tr>
<tr>
<td><code>-fchkp-instrument-calls</code></td>
<td>Generates bounds passing for calls. Enabled by default when <code>-fcheck-pointer-bounds</code> is specified.</td>
<td></td>
</tr>
<tr>
<td><code>-fchkp-instrument-marked-only</code></td>
<td>Instruments only functions marked with bnd_instrument attribute.</td>
<td></td>
</tr>
</tbody>
</table>

### 7.3.8 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

### 7.3.9 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 per shown in Table 20.

**Table 20 Options to select Hardware-based Intel® MPX and SW-based Pointer Checker**

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Q]check-pointers</td>
<td>Enables the pointer checker and adds the associated libraries (SW-based bounds checking).</td>
</tr>
<tr>
<td>[Q]check-pointers-mpx</td>
<td>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</td>
</tr>
</tbody>
</table>

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: https://software.intel.com/node/7ac14276-fa70-4c9b-ae18-c07ba0e83e79.

---

### 7.4 Enabling Intel® MPX in other components

---
Assemblers and debuggers should be modified to process the new Intel MPX registers and instructions. The Linker requires new flag to process BND prefix for Jmp instructions to set proper bounds values.

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

8 Building and Executing Intel® MPX 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® MPX 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.
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 susceptible 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
   ```
8.2 Enabling Intel® MPX in new code

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

1. Ensure allocations, reallocations, and de-allocations 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:**
   ```c
   void *__wrap_malloc (size_t n)
   {
       void *p = (void *)__real_malloc (n);
       if (!p) return __bnd_null_ptr_bounds(p); //bnd: [-1, 0]
       return __bnd_set_ptr_bounds (p, n); //bnd: [p, p+n-1]
   }
   ```
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® MPX 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.
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® MPX 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. Microsoft Windows* 8.1 systems require a register setting for Intel MPX to be 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.
- Create a launcher for the application, which sets the session variables to influence the runtime behavior.
- The installer should ensure that sufficient free memory is available on the system.

**9 References**

8. Address Sanitizer and Intel MPX: https://code.google.com/p/address-sanitizer/wiki/IntelMemoryProtectionExtensions
9. Discussion page for Intel MPX: http://www.reddit.com/comments/1iw9lx

Notices

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

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.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others

© 2015 Intel Corporation.