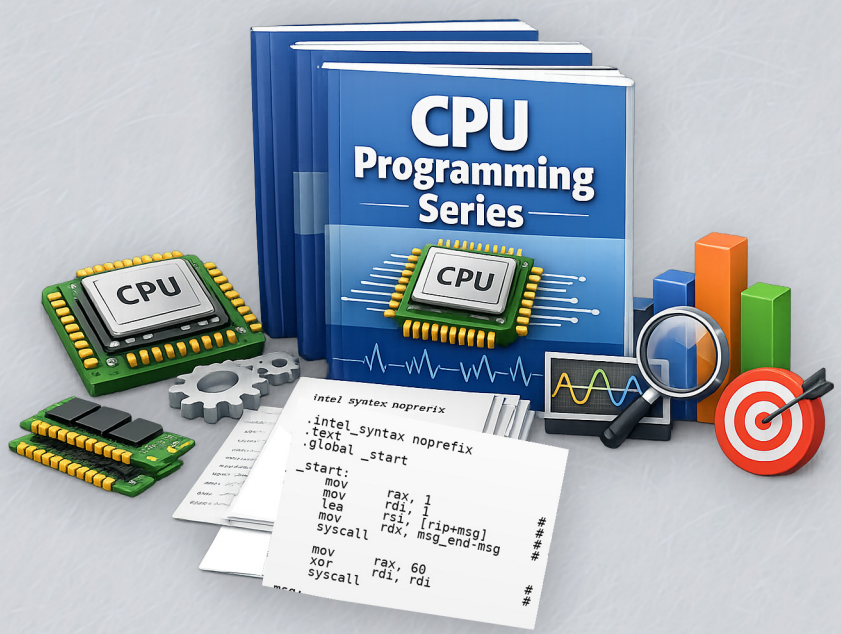
https://simplifycpp.org

CPU Programming Series

Windows x64 ABI

Calling Convention Differences That Break Code



8

Prepared by Ayman Alheraki

CPU Programming Series

Windows x64 ABI

Calling Convention Differences That Break Code

Prepared by Ayman Alheraki

simplifycpp.org

January 2026

Contents

Preface

Why This Booklet Exists

This booklet exists to address one of the most common and costly sources of low-level software failure: misunderstanding the *Windows x64 ABI*. Many programmers assume that calling conventions are a minor detail handled entirely by the compiler. In practice, ABI rules directly affect correctness, stability, interoperability, debugging, and security.

On Windows x64, small violations—such as incorrect shadow space handling, improper stack alignment, or incorrect register preservation—can silently corrupt execution state, cause intermittent crashes, or break code only under optimization or exception handling. These failures are frequently misdiagnosed as compiler bugs, optimizer issues, or undefined behavior unrelated to calling conventions.

This booklet focuses on the precise rules that actually break real programs when misunderstood, using concrete and minimal examples that reflect real compiler output and real-world failure modes.

Who This Booklet Is For

This booklet is intended for readers who already write or analyze low-level code and need ABI correctness rather than surface-level explanations.

It is written for:

- Systems programmers working close to the operating system or runtime
- C and C++ developers interfacing with assembly, foreign languages, or binary libraries
- Assembly programmers targeting Windows x64
- Developers debugging crashes using disassembly and stack traces
- Engineers maintaining cross-platform or cross-compiler codebases

The reader is expected to understand basic concepts such as registers, the stack, function calls, and compiler-generated code. This booklet does not assume prior knowledge of Windows-specific ABI rules.

What This Booklet Deliberately Does Not Cover

To maintain focus and correctness, this booklet deliberately excludes several topics that are commonly mixed with ABI discussions but require separate treatment.
This booklet does not cover:

- Instruction set details unrelated to calling conventions
- Microarchitectural optimization or performance tuning
- Operating system kernel internals
- Windows system calls and kernel transition mechanisms
- High-level language runtime design
- Debugger usage tutorials

Only ABI-relevant behavior that affects function calls, stack discipline, register preservation, and interoperability is included. Performance discussions appear only when they directly influence correctness.

How This Booklet Fits into the CPU Programming Series

This booklet is part of the architecture-specific phase of the CPU Programming Series.
Earlier booklets establish:

- How the CPU executes instructions
- How registers and flags behave
- How the stack works conceptually
- What an ABI represents at a theoretical level

This booklet applies those foundations specifically to the *Windows x64 ABI*. It assumes the reader already understands stack mechanics and function calls in principle, and now needs to understand why identical-looking code behaves differently across platforms and compilers. Later booklets build on this material to cover advanced interoperability, exception handling internals, and cross-platform ABI abstraction strategies.

How to Read This Booklet Effectively

This booklet is designed to be read sequentially.
Each chapter introduces ABI rules first, then demonstrates how violations occur, and finally explains how compilers enforce or rely on those rules. Code examples are intentionally small and focused, reflecting real compiler output rather than artificial demonstrations.
When reading assembly examples:

- Pay attention to stack pointer adjustments
- Track register preservation across calls
- Observe shadow space allocation and usage

Readers are encouraged to mentally simulate stack and register state rather than memorizing rules. The goal is to build ABI intuition that prevents bugs before they occur.

Chapter 1

Why ABI Differences Matter

1.1 What an ABI Really Defines (Beyond Function Calls)

An *Application Binary Interface (ABI)* defines the complete binary-level contract between independently compiled code units. While function calls are the most visible part, the ABI governs far more than argument passing.
An ABI defines, at minimum:

- How arguments and return values are passed
- Which registers must be preserved across calls
- Stack layout, alignment, and ownership rules
- How stack frames are constructed and destroyed
- How exceptions and stack unwinding operate
- How data types are laid out in memory

- How code interacts across object files, libraries, and languages

For example, even a simple call involves implicit ABI rules:

```
# Windows x64 ABI: caller-side responsibilities
# RCX, RDX, R8, R9 used for first four arguments
# 32 bytes of shadow space must be allocated

sub rsp, 40h        # 32 bytes shadow + 16-byte alignment
mov rcx, 1
mov rdx, 2
call target_func
add rsp, 40h
```

The function call itself is only one instruction, but correctness depends on stack alignment, shadow space allocation, and register state—none of which are visible in the function prototype.

1.2 Why Code That "Looks Correct" Still Breaks

Many ABI violations produce code that appears correct in isolation but fails under real execution conditions.
Common reasons include:

- The compiler assumes ABI rules were followed by the caller
- Violations may not fail immediately
- Bugs often surface only under optimization or exception handling

Example of code that looks correct but violates the ABI:

```
# Incorrect: missing shadow space allocation
mov rcx, 1
mov rdx, 2
call target_func      # ABI violation
ret
```

This may appear to work if:

- The callee does not use shadow space
- No exceptions occur
- Optimization level is low

However, once the callee spills registers or an exception is thrown, stack corruption occurs. The failure appears far from the original mistake, often misleading developers into suspecting unrelated code.

1.3 ABI vs ISA vs Compiler Behavior

These three concepts are frequently confused but serve different roles.

- **ISA** defines available instructions and registers
- **ABI** defines how compiled code must use those instructions together
- **Compiler behavior** is constrained by both

The ISA allows this instruction:

```
sub rsp, 8
```

But the ABI may require:

```
sub rsp, 40h      # shadow space + alignment
```

The compiler is not free to choose arbitrarily. It must generate code that obeys the ABI contract so that independently compiled modules can interoperate safely.
When programmers manually write assembly or mix languages, they step outside the compiler's safety net. At that point, ABI rules become mandatory knowledge rather than optional details.

1.4 Why Windows x64 Deserves Its Own Focused Study

Windows x64 is not a minor variation of other 64-bit ABIs. It introduces design choices that directly affect correctness and interoperability.
Key characteristics that justify a dedicated study include:

- Mandatory shadow space allocation by the caller
- No red zone below the stack pointer
- Strong coupling with structured exception handling
- Different register preservation rules from System V
- ABI rules enforced by the operating system and runtime

A direct comparison illustrates the danger of assumptions:

```
# System V style (invalid on Windows x64)
sub rsp, 8          # assumes red zone
call target_func
add rsp, 8
```

This code is valid on System V platforms but is incorrect on Windows x64. Reusing such code silently introduces instability, especially in cross-platform projects.

Windows x64 ABI rules are tightly integrated with the OS exception model, stack unwinding, and debugging infrastructure. Violating them affects not only function calls but also crash handling and diagnostics.

For these reasons, Windows x64 ABI correctness cannot be treated as a footnote. It requires explicit, focused study grounded in real execution behavior.

Chapter 2

Windows x64 ABI: Design Overview

2.1 Historical Background and Design Goals

The *Windows x64 ABI* was designed to provide a single, stable, and efficient binary contract for 64-bit Windows across compilers and languages, while avoiding the fragmentation of multiple 32-bit calling conventions.
Its practical design goals can be summarized as follows:

- **One primary calling convention for user-mode code:** reduce ambiguity and make interoperability the default.
- **High performance with predictable code generation:** pass the most common arguments in registers, minimize call overhead.
- **Reliable exception handling and stack unwinding:** enable correct unwinding through compiler-generated metadata rather than ad-hoc conventions.
- **Debuggability and tooling compatibility:** ensure stack walking, profiling, and crash dumps can reconstruct call stacks consistently.

- **Stable cross-module behavior:** allow independently compiled modules (EXEs, DLLs, third-party libraries) to interoperate without source-level coordination.

A key design choice is the mandatory **shadow space** (home space): the caller always reserves stack space so the callee can spill register arguments predictably.

```
.intel_syntax noprefix
# Windows x64: caller reserves 32 bytes shadow space (home space)
# plus additional space as needed to keep 16-byte alignment before
 ↪  CALL.

caller:
    sub  rsp, 40h           # 32 bytes shadow + 16 alignment padding
     ↪  (common pattern)
    mov  ecx, 1             # arg1 in RCX
    mov  edx, 2             # arg2 in RDX
    call target
    add  rsp, 40h
    ret
```

This predictable layout is not merely an optimization; it supports correctness across optimization levels and across language boundaries.

2.2 Relationship to MSVC and Windows OS Internals

On Windows x64, the ABI is tightly integrated with operating system mechanisms that depend on reliable stack frames and unwind information.

Unwinding, SEH, and "Unwindable" Prologues

Windows x64 relies on compiler-emitted unwind metadata to perform:

- stack unwinding during exceptions,
- stack walking for debuggers and profilers,
- reliable crash diagnostics.

This means that certain function prologues/epilogues are expected to follow patterns that the unwinder can interpret, and the compiler records how registers were saved and how the stack pointer changed.
A typical prologue that saves non-volatile registers and allocates locals:

```
.intel_syntax noprefix
# Typical Windows x64 prologue structure (conceptual)
# - allocate stack frame
# - save non-volatile registers if used
# - optionally establish frame pointer

target:
    sub  rsp, 60h          # locals + alignment, shadow space already
     ↪  provided by caller
    mov  [rsp+20h], rbx    # spill/save non-volatile register
     ↪  (example placement)
    mov  [rsp+28h], rsi
    # ... function body ...
    mov  rsi, [rsp+28h]
    mov  rbx, [rsp+20h]
    add  rsp, 60h
    ret
```

Important idea: even when a function does not “use exceptions”, the OS and tooling may still need to unwind through it correctly. ABI correctness therefore affects far more than just argument passing.

Stack Probing and Large Stack Allocations

Windows uses stack commitment rules that require safely touching stack pages when allocating large frames. Compilers may inject stack-probe sequences (or calls) to ensure a large allocation does not jump over guard pages.
You do not need to memorize the probing algorithm; the ABI-level lesson is:

- large stack allocations can introduce implicit code sequences,
- incorrect manual stack manipulation can break those assumptions,
- mixing handwritten assembly with compiler code requires extra discipline.

2.3 Why Windows x64 Diverges from System V

Although both are 64-bit ABIs on x86-64, *Windows x64* and *System V AMD64* make different trade-offs. Assuming one while targeting the other is a direct route to crashes and silent corruption.

Shadow Space vs Red Zone

Windows x64:

- Caller must reserve 32 bytes of shadow space for the callee.
- No red zone: memory below `RSP` is not safe to use.

System V AMD64:

- No shadow space requirement.
- Red zone exists (a region below `RSP` that leaf functions may use without adjusting `RSP`).

A leaf function style that is valid on System V can be invalid on Windows x64:

```
.intel_syntax noprefix
# System V style idea: use red zone without moving RSP (NOT valid on
↪  Windows x64)
leaf_like:
    mov  qword ptr [rsp-08h], rbx   # writes below RSP
    # ... do work ...
    ret
```

On Windows x64, code that touches below `RSP` can be overwritten by asynchronous events or system mechanisms, and it violates the platform expectation that `RSP` defines the valid stack extent.

Argument Registers and Preservation Rules

Both ABIs pass early arguments in registers, but the details differ, including which registers are considered non-volatile and which vector registers must be preserved.
Windows x64 emphasizes:

- predictable shadow space for register-argument homing,
- a defined set of non-volatile general-purpose registers,
- preservation requirements that include a subset of SIMD registers.

A common interoperability pitfall: mixing code that assumes System V argument registers with Windows x64 code.

```
.intel_syntax noprefix
# WRONG on Windows x64:
# System V passes arg1 in RDI, arg2 in RSI.
```

```
# Windows x64 expects arg1 in RCX, arg2 in RDX.

call_with_sysv_assumptions:
    mov  rdi, 111          # programmer thinks: arg1
    mov  rsi, 222          # programmer thinks: arg2
    call target            # target reads RCX/RDX, not RDI/RSI
    ret
```

Correct Windows x64 register setup:

```
.intel_syntax noprefix
call_with_win64_rules:
    sub  rsp, 40h
    mov  rcx, 111          # arg1
    mov  rdx, 222          # arg2
    call target
    add  rsp, 40h
    ret
```

Why These Divergences Exist

Windows x64 chooses:

- **shadow space** to standardize argument homing/spilling and simplify certain code-generation and debugging patterns,
- **no red zone** to keep stack semantics robust under asynchronous system activity and tooling,
- **strong unwind integration** to support reliable exception handling and stack walking in the Windows ecosystem.

System V chooses:

- a red zone to optimize leaf functions,
- different register allocation priorities aligned with Unix-like ecosystem conventions and toolchains.

Neither is “better” in isolation; each is optimized for its platform’s conventions and runtime assumptions. The danger is assuming they are interchangeable.

2.4 ABI Guarantees vs Compiler Freedoms

An ABI defines what **must** hold true across separately compiled boundaries. Compilers may choose different strategies internally, but only as long as the externally visible contract is preserved.

ABI Guarantees (Non-Negotiable Rules)

Examples of hard ABI requirements on Windows x64:

- **Caller provides shadow space** before making a call.
- **Stack alignment must meet the platform rule at call boundaries.**
- **Non-volatile registers must be preserved by the callee** if it uses them.
- **Return values use defined registers and layouts.**
- **Unwinding must be possible** through compiler-generated metadata for non-leaf functions and functions that alter `RSP` in non-trivial ways.

Violation example: clobbering a non-volatile register in the callee.

```
.intel_syntax noprefix
# WRONG: clobbers RBX (non-volatile) without saving/restoring.
bad_callee:
    mov  rbx, 12345678h
    xor  eax, eax
    ret
```

Correct pattern:

```
.intel_syntax noprefix
good_callee:
    push rbx
    mov  rbx, 12345678h
    xor  eax, eax
    pop  rbx
    ret
```

Compiler Freedoms (Allowed Variations)

Within the ABI rules, compilers have significant freedom:

- omit or use a frame pointer depending on optimization and debugging settings,
- choose how to allocate locals and where to spill registers,
- reorder instructions, inline calls, and perform tail-call optimizations when legal,
- keep values in registers instead of memory, or spill aggressively under register pressure,
- choose different prologue/epilogue sequences as long as they remain unwind-correct.

Two different compilers (or the same compiler under different options) can produce different assembly for the same source code. That is normal. What must remain invariant is the **external ABI contract** at module boundaries.
A practical rule for the reader:

- If your code crosses a boundary (DLL, library, language runtime, callback, hand-written assembly), obey the ABI strictly.
- If your code stays within one compilation unit, the compiler may transform it heavily while still preserving the ABI at every external call site.

Chapter 3

Register Usage and Calling Rules

3.1 Argument Passing Registers

Integer and Pointer Arguments (General-Purpose Registers)

The Windows x64 ABI passes the first four integer or pointer arguments in general-purpose registers:

- Argument 1 → `RCX`
- Argument 2 → `RDX`
- Argument 3 → `R8`
- Argument 4 → `R9`

Additional arguments are passed on the stack (right-to-left in memory as laid out by the caller), while the caller also reserves the mandatory 32-byte shadow space.

```
.intel_syntax noprefix
# call: f(a,b,c,d,e,f)
# a,b,c,d in RCX,RDX,R8,R9
# e,f on stack
# caller must allocate 32-byte shadow space

caller:
    sub  rsp, 40h                 # 32 shadow + alignment (typical)
    mov  rcx, 11                  # a
    mov  rdx, 22                  # b
    mov  r8,  33                  # c
    mov  r9,  44                  # d
    mov  qword ptr [rsp+20h], 55  # e (stack arg 5) placed above
     ↪  shadow space
    mov  qword ptr [rsp+28h], 66  # f (stack arg 6)
    call f
    add  rsp, 40h
    ret
```

Floating-Point Arguments (XMM Registers)

For floating-point (`float`/`double`) arguments, the ABI uses XMM registers for the first four FP arguments:

- FP Arg 1 → `XMM0`
- FP Arg 2 → `XMM1`
- FP Arg 3 → `XMM2`
- FP Arg 4 → `XMM3`

A key rule: the ABI distinguishes integer-class and floating-class arguments; the registers used depend on the parameter types as seen by the compiler and the callee's signature.

```
.intel_syntax noprefix
# call: g(int a, double x, int b, double y)
# a -> RCX, x -> XMM1? (see note below), b -> R8, y -> XMM3? depends
 ↪  on signature classification
# Practical rule: argument registers are assigned by position and
 ↪  class rules as the ABI defines.

caller_g:
    sub  rsp, 40h
    mov  ecx, 7                  # a
    mov  r8d, 9                  # b (3rd integer arg position)
    # load doubles into xmm registers (example uses memory constants)
    movsd xmm1, qword ptr [rip+val_x]
    movsd xmm3, qword ptr [rip+val_y]
    call g
    add  rsp, 40h
    ret

val_x: .quad 0x400921FB54442D18    # pi as double bits (example)
val_y: .quad 0x4005BF0A8B145769    # e  as double bits (example)
```

Practical guidance: for mixed signatures, do not hand-assign XMM registers unless you are matching the exact compiled signature and have verified with disassembly. The safest approach is to let the compiler generate the call sequence, then mirror it in assembly if needed.

3.2 Return Value Registers

Integer and Pointer Returns

Integer/pointer return values are delivered in:

- `RAX` (primary return)

For small integer types, the value is in `AL/AX/EAX` as appropriate, but logically it is `RAX`.

```
.intel_syntax noprefix
# int64_t h(void) returns in RAX

h:
    mov  rax, 123456789
    ret
```

Floating-Point Returns

Floating-point returns:

- `XMM0` (float/double return)

```
.intel_syntax noprefix
# double k(void) returns in XMM0

k:
    movsd xmm0, qword ptr [rip+val_pi]
    ret

val_pi: .quad 0x400921FB54442D18
```

Aggregates and "Hidden" Return Mechanisms

Some aggregates/structs are returned indirectly using a hidden pointer provided by the caller (a **hidden sret** mechanism). Whether a struct is returned in registers or indirectly depends on ABI classification rules and size/layout.

Practical rule: when interoperability matters, avoid returning large aggregates across module/language boundaries unless you have validated the exact ABI lowering, or use an explicit out-parameter.

```
.intel_syntax noprefix
# Conceptual pattern: caller provides pointer to return storage
 ↪  (hidden)
# callee writes result into that storage
# (Exact register used for hidden pointer is defined by ABI lowering
 ↪  rules.)

# Safer cross-boundary alternative:
# void make_result(Result* out, ...);
```

3.3 Volatile vs Non-Volatile Registers

Windows x64 classifies registers into:

- **Volatile (caller-saved):** may be clobbered by the callee.
- **Non-volatile (callee-saved):** must be preserved by the callee if it uses them.

General-Purpose Registers

Common volatile GPRs:

- `RAX, RCX, RDX, R8, R9, R10, R11`

Common non-volatile GPRs:

- `RBX, RBP, RSI, RDI, R12, R13, R14, R15`

XMM Registers

Volatile XMM:

- `XMM0--XMM5`

Non-volatile XMM:

- `XMM6--XMM15`

Example: correct preservation of a non-volatile register (`RBX`) and a non-volatile XMM register (`XMM6`):

```
.intel_syntax noprefix
callee:
    push rbx
    sub  rsp, 20h                 # align / local space (example)
    movdqu xmmword ptr [rsp], xmm6 # save XMM6 (example save)
    # ... use RBX and XMM6 ...
    movdqu xmm6, xmmword ptr [rsp] # restore XMM6
    add  rsp, 20h
    pop  rbx
    ret
```

Note: the exact stack allocation must still maintain required alignment rules at call boundaries. Use compiler output as a reference pattern when writing hand assembly.

3.4 Hidden ABI Assumptions Programmers Often Miss

Shadow Space Must Exist Even If You "Do Not Use It"

The caller must reserve 32 bytes of shadow space for every call, regardless of the number of arguments. Some callees will use it for homing/spilling register arguments.

```
.intel_syntax noprefix
# WRONG: call without shadow space
bad_call:
    mov  rcx, 1
    call target
    ret
```

```
.intel_syntax noprefix
# CORRECT: always reserve shadow space
good_call:
    sub  rsp, 40h
    mov  rcx, 1
    call target
    add  rsp, 40h
    ret
```

Stack Alignment at Call Sites Is Not Optional

If the stack is misaligned at a call site, a callee that uses aligned SIMD instructions or assumes ABI alignment may fault or behave unpredictably.

```
.intel_syntax noprefix
# Example of a common alignment bug: subtracting wrong amount
```

```
misaligned_call:
    sub  rsp, 20h      # may break required alignment depending on
     ↪  entry alignment
    call target
    add  rsp, 20h
    ret
```

Practical rule: replicate the compiler's standard allocation pattern for call sites (often `sub rsp, 40h` for simple calls) unless you have proven your alignment math is correct for every path.

Callee May Assume Correct Prologue for Unwinding

Even if you never "throw", the platform's unwinder and tools may walk through frames. Unwind correctness relies on standardized stack manipulation patterns recorded in unwind metadata.
Practical rule: avoid writing hand-crafted prologues in mixed C/C++ code unless you fully understand unwind requirements for Windows x64.

3.5 Common Mistakes When Mixing Inline Assembly

Mistake 1: Clobbering Non-Volatile Registers Without Saving

Inline assembly that overwrites `RBX, RBP, RSI, RDI, R12--R15` (or `XMM6--XMM15`) without saving/restoring breaks the ABI contract.

```
.intel_syntax noprefix
# WRONG: clobbers RBX (non-volatile)
    mov rbx, 0
```

Correct approach: save/restore or avoid non-volatile registers.

```
.intel_syntax noprefix
# CORRECT: preserve RBX if used
    push rbx
    mov  rbx, 0
    # ...
    pop  rbx
```

Mistake 2: Assuming Inline Assembly Is a "Barrier"

Optimizing compilers may reorder surrounding code unless the inline-asm form explicitly tells the compiler about clobbers, memory effects, and dependencies. If the compiler does not know what your assembly touches, it may keep values in registers you silently overwrite.
Practical rule: inline assembly must declare:

- all clobbered registers,
- whether memory is read/written,
- outputs and inputs accurately.

Mistake 3: Forgetting Shadow Space Before Calling from Assembly

Calling a C/C++ function from inline/hand assembly without allocating shadow space is a classic source of sporadic crashes.

```
.intel_syntax noprefix
# WRONG: missing shadow space
    mov rcx, 123
    call some_c_function
```

```
.intel_syntax noprefix
# CORRECT
    sub rsp, 40h
    mov rcx, 123
    call some_c_function
    add rsp, 40h
```

Mistake 4: Mixing System V Assumptions on Windows

Some developers reuse snippets that place arg1 in `RDI` and arg2 in `RSI`. That is System V, not Windows x64.

```
.intel_syntax noprefix
# WRONG on Windows x64:
    mov rdi, 1
    mov rsi, 2
    call f
```

```
.intel_syntax noprefix
# CORRECT on Windows x64:
    sub rsp, 40h
    mov rcx, 1
    mov rdx, 2
    call f
    add rsp, 40h
```

Mistake 5: Confusing “volatile” in C++ With “volatile” Registers

C++ `volatile` is not a calling convention rule. It does not substitute for proper clobber declarations, stack discipline, or ABI preservation. The ABI definition of volatile/non-volatile registers is purely about call preservation.

Bottom line: inline assembly is safest when it is minimized, fully declares clobbers, preserves non-volatile registers, and follows shadow space and alignment rules exactly.

Chapter 4

Shadow Space (Home Space)

4.1 What Shadow Space Is and Why It Exists

Shadow space (also called **home space**) is a mandatory stack area reserved by the **caller** on every call under the Windows x64 ABI.

- Size: **32 bytes** (4 slots of 8 bytes each)
- Purpose: provides a fixed, always-available place where the callee can **home** (spill) the first four register arguments (`RCX`, `RDX`, `R8`, `R9`) if needed.

Even if the callee never uses it, the ABI requires it to exist so that:

- code generation is simpler and predictable across optimization levels,
- the callee can spill arguments without first adjusting `RSP`,
- debugging and instrumentation can rely on a stable call-frame convention,
- interoperability across modules and languages is consistent.

A minimal call that obeys the ABI looks like:

```
.intel_syntax noprefix
# Windows x64 ABI: caller allocates 32-byte shadow space for every
 ↪  call.

caller:
    sub  rsp, 40h          # 20h shadow + 20h alignment padding
     ↪  (common pattern)
    mov  rcx, 1
    mov  rdx, 2
    call callee
    add  rsp, 40h
    ret
```

Important: the `40h` allocation above is a common pattern because it both provides `20h` shadow space and keeps the stack aligned at the call site. The ABI requirement is the `20h` shadow space; alignment requirements determine the total adjustment.

4.2 Mandatory Allocation Rules

The ABI rule is strict:

- The caller must reserve **exactly at least 32 bytes** of shadow space **before executing `call`**.
- This applies even if the callee takes zero parameters.
- This applies even if all parameters are passed on the stack (more than four arguments).

Shadow space sits at the top of the caller's call frame, directly above the return address as seen by the callee after the call transfers control.
Conceptual layout from the callee's perspective on entry:

```
.intel_syntax noprefix
# On entry to callee (conceptual stack view):
# [rsp+00h] : return address
# [rsp+08h] : shadow slot for RCX
# [rsp+10h] : shadow slot for RDX
# [rsp+18h] : shadow slot for R8
# [rsp+20h] : shadow slot for R9
```

Practical caution: offsets above are conceptual and used for understanding. Real compiler output may establish a frame and move `RSP`; always validate offsets against the final `RSP` value in the actual prologue.

4.3 Caller vs Callee Responsibilities

Caller Responsibilities

The caller must:

- allocate 32 bytes shadow space for every call,
- ensure required stack alignment at the call boundary,
- place additional arguments (5th and beyond) on the stack above the shadow space,
- assume volatile registers may be clobbered by the callee.

Example: calling a function with 6 integer arguments:

```
.intel_syntax noprefix
# f(a,b,c,d,e,f)
# a..d in RCX,RDX,R8,R9
# e,f on stack above shadow space.

call_f:
    sub  rsp, 40h
    mov  rcx, 11                  # a
    mov  rdx, 22                  # b
    mov  r8,  33                  # c
    mov  r9,  44                  # d
    mov  qword ptr [rsp+20h], 55 # e (5th)
    mov  qword ptr [rsp+28h], 66 # f (6th)
    call f
    add  rsp, 40h
    ret
```

Callee Responsibilities

The callee may:

- use shadow space to store (home) argument registers,
- overwrite those shadow slots freely,
- treat those slots as temporary storage for spilling those arguments.

The callee must:

- preserve non-volatile registers if it uses them,
- maintain unwind-correct stack manipulations (important for Windows tooling and exceptions),

- not assume shadow space contains valid values unless it explicitly stores them.

A common pattern is homing register arguments early:

```
.intel_syntax noprefix
callee:
    # home the register arguments (example pattern)
    mov  qword ptr [rsp+08h], rcx
    mov  qword ptr [rsp+10h], rdx
    mov  qword ptr [rsp+18h], r8
    mov  qword ptr [rsp+20h], r9
    # ... now arguments are in memory if needed ...
    ret
```

Key idea: the callee can do this immediately because the ABI guarantees the space exists.

4.4 Interaction with Leaf and Non-Leaf Functions

Leaf Functions

A **leaf function** makes no calls. It may not need to allocate stack space at all, and it may not need shadow space for itself because it does not call others.
However:

- leaf functions can still be called by others, so their *callers* still must provide shadow space,
- a leaf function may still use the caller-provided shadow space to home arguments if desired.

Example leaf that homes one argument and returns:

```
.intel_syntax noprefix
# leaf: returns (arg1 + 1)
# caller already provided shadow space

leaf_add1:
    mov  qword ptr [rsp+08h], rcx   # optional home
    lea  rax, [rcx+1]
    ret
```

Non-Leaf Functions

A **non-leaf function** calls other functions. It must allocate **its own** shadow space before each call it performs.
This is a critical point:

- Shadow space is not inherited across calls.
- Each call site must allocate its own shadow space.

Example: non-leaf function calling another function:

```
.intel_syntax noprefix
non_leaf:
    # ... do work ...
    sub  rsp, 40h
    mov  rcx, 100
    call helper
    add  rsp, 40h
    ret
```

4.5 Bugs Caused by Incorrect Shadow Space Handling

Shadow space violations create some of the hardest bugs to diagnose because the crash often happens far away from the mistake.

Bug 1: Calling Without Allocating Shadow Space

```
.intel_syntax noprefix
# WRONG: missing shadow space
bad_call:
    mov  rcx, 1
    call target
    ret
```

Possible outcomes:

- immediate crash if the callee homes arguments to `[rsp+08h]` etc. and overwrites unrelated stack data,
- silent corruption that triggers later,
- works in debug but fails in release due to different spill patterns.

Bug 2: Allocating Shadow Space but Forgetting Alignment

Allocating 32 bytes is necessary but not sufficient if your stack alignment at the call boundary is wrong.

```
.intel_syntax noprefix
# WRONG pattern: reserves 20h shadow but may violate alignment in
 ↪  some contexts
```

```
sub  rsp, 20h
call target
add  rsp, 20h
```

Practical safe approach: use the compiler-like pattern at call sites (commonly `sub rsp, 40h`) unless you have proven the alignment math for every path.

Bug 3: Treating Shadow Space as Persistent Storage

Shadow space belongs to the caller's frame and is intended as temporary homing/spill area for the callee. It is not a stable place to store values across calls.

```
.intel_syntax noprefix
# WRONG idea: store something in shadow space, then call another
 ↪  function,
# assuming it will still be there.
sub  rsp, 40h
mov  qword ptr [rsp+08h], 1234
call other            # other may overwrite its own shadow space, and
 ↪  your assumptions may break
mov  rax, qword ptr [rsp+08h]
add  rsp, 40h
ret
```

Correct approach: if you need persistent locals, allocate a real local stack frame (beyond shadow) or use callee-saved registers with proper preservation.

Bug 4: Mixing System V Snippets that Use the Red Zone

Some System V assembly uses memory below `RSP` without adjusting `RSP`. On Windows x64, there is no red zone. Combined with shadow space misunderstandings, this produces severe instability.

```
.intel_syntax noprefix
# WRONG on Windows x64: writing below RSP (red-zone assumption)
mov  qword ptr [rsp-08h], rbx
```

Bug 5: Forgetting Shadow Space on Indirect Calls and Callbacks

Indirect calls (through function pointers) still require shadow space, and callback boundaries are a common place where violations appear.

```
.intel_syntax noprefix
# WRONG: indirect call without shadow space
mov  rax, qword ptr [rip+fp]
mov  rcx, 7
call rax
ret
fp: .quad 0
```

```
.intel_syntax noprefix
# CORRECT
mov  rax, qword ptr [rip+fp]
sub  rsp, 40h
mov  rcx, 7
call rax
add  rsp, 40h
ret
fp: .quad 0
```

Operational Checklist: Shadow Space Correctness

- Every `call` site reserves at least `20h` bytes shadow space.

- Shadow space is reserved **even for zero-argument calls**.
- Stack alignment is correct at the call boundary.
- Do not treat shadow space as persistent local storage.
- Indirect calls and callbacks also obey the shadow-space rule.

Chapter 5

Stack Layout and Alignment

5.1 Required Stack Alignment Rules

Windows x64 imposes a strict stack alignment rule that must hold at **every call boundary**. The goal is to guarantee that callees can safely use aligned SIMD instructions and predictable stack layouts.

Core Rule (Call-Site Contract)

- The stack pointer `RSP` must be **16-byte aligned at the point of a call boundary as defined by the ABI contract**.
- The `call` instruction pushes an 8-byte return address, so the alignment relationship differs between:
  - the caller *before* `call`,
  - the callee *on entry* (after return address is pushed).

Practical safe rule for hand-written call sites: use the same pattern compilers use for simple calls:

```
.intel_syntax noprefix
# Safe call-site pattern used widely by compilers:
# - allocate 32-byte shadow space
# - include padding so the call-site alignment is correct

sub  rsp, 40h
call target
add  rsp, 40h
```

Why This Matters

If alignment is wrong, a callee that uses aligned loads/stores to the stack (or assumes aligned local variables) can fault or behave unpredictably under optimization.

5.2 Stack Frame Structure in Windows x64

Windows x64 stack frames are designed to be compatible with:

- register argument passing,
- mandatory shadow space,
- unwind metadata for exceptions and tooling.

A typical stack frame has these conceptual regions (from high addresses down toward `RSP`):

- **Caller-passed stack arguments** (5th and beyond)
- **Shadow space (32 bytes)** reserved by the caller for the callee

- **Return address** pushed by `call`
- **Callee local frame** (locals, spills, saved non-volatiles, alignment padding)

A conceptual view from the callee on entry (before it changes `RSP`):

```
.intel_syntax noprefix
# Callee entry stack view (conceptual):
# [rsp+00h] return address
# [rsp+08h] shadow slot 0 (home for RCX)
# [rsp+10h] shadow slot 1 (home for RDX)
# [rsp+18h] shadow slot 2 (home for R8)
# [rsp+20h] shadow slot 3 (home for R9)
# [rsp+28h] stack arg 5 (if present)
# [rsp+30h] stack arg 6 (if present)
```

Typical Non-Leaf Frame Pattern

A non-leaf function that uses locals and saves non-volatile registers commonly:

- allocates local space,
- saves non-volatiles it will modify,
- may home args to shadow space or to its own locals,
- makes calls (each call requires its own shadow space allocation).

```
.intel_syntax noprefix
# Example: typical shape (conceptual) of a non-leaf function frame
# Note: exact offsets depend on the compiler and options.
```

```
func:
    push rbx                    # save non-volatile
    sub  rsp, 50h               # locals + alignment (example)
    # ... function body ...
    sub  rsp, 40h               # shadow space for a nested call (plus
     ↪  padding if needed)
    call helper
    add  rsp, 40h
    add  rsp, 50h
    pop  rbx
    ret
```

Key point: shadow space is tied to each call site, not to the function as a whole. A function may allocate multiple shadow spaces across its execution (or reuse a reserved region if it manages alignment correctly).

5.3 Why the Red Zone Does Not Exist

Some ABIs (notably System V AMD64) define a **red zone**: a fixed region below `RSP` that leaf functions may use without adjusting `RSP`.
Windows x64 defines **no red zone**.
Practical implications:

- Memory below `RSP` must be treated as **unsafe**.
- Leaf functions must not assume they can store temporaries at `[rsp-XX]`.
- Reusing System V assembly snippets that rely on the red zone is a frequent source of crashes.

Example of a red-zone assumption that is invalid on Windows x64:

```
.intel_syntax noprefix
# WRONG on Windows x64: writing below RSP (red-zone assumption)
leaf_bad:
    mov  qword ptr [rsp-08h], rbx
    xor  eax, eax
    ret
```

Correct Windows x64 approach: allocate explicit space if needed:

```
.intel_syntax noprefix
leaf_ok:
    sub  rsp, 20h
    mov  qword ptr [rsp+00h], rbx
    xor  eax, eax
    add  rsp, 20h
    ret
```

5.4 Stack Probing and Large Allocations

Windows uses a committed-stack model with guard pages. Large stack allocations must ensure the program touches pages in a controlled way so that guard pages can be triggered safely and the stack can be committed incrementally.
Modern compilers therefore insert stack probing logic for large allocations. This can appear as:

- an inlined probing loop, or
- a call to a helper routine that performs probing.

The ABI-level lesson is not the exact probing algorithm, but the constraint it creates:

- If you move `RSP` by a large amount without probing, you can jump over a guard page and fault in a way that looks unrelated.
- If you write hand assembly that allocates large frames, you must follow the platform's probing expectations.

A conceptual (simplified) probing pattern looks like:

```
.intel_syntax noprefix
# Conceptual probing: touch one address per page while moving RSP
 ↪  down.
# This is illustrative only (exact step/page size matters).

sub  rsp, 20000h          # large allocation (example)
mov  rax, rsp
# touch memory in steps to ensure pages are committed
# (real probing would touch at page granularity)
mov  byte ptr [rax], 0
```

Practical guidance:

- Avoid large stack allocations in hand-written assembly.
- Prefer heap or caller-provided buffers for large objects at ABI boundaries.
- If unavoidable, mirror the compiler-generated probing pattern used by your toolchain.

5.5 Crash Patterns Caused by Misalignment

Misalignment bugs are notorious because they often:

- occur only in release builds,

- disappear when debugging (due to different prologues),
- crash inside unrelated library code.

Pattern 1: Crash Inside Aligned SIMD Instruction

A callee may use an aligned instruction that faults if the address is not properly aligned. Example concept (aligned store to stack-local):

```
.intel_syntax noprefix
# If RSP is misaligned, an aligned store/load to [rsp+offset] can
 ↪  fault.
# (Instruction choice depends on compiler and CPU features.)

# Example conceptual aligned store (do not rely on instruction
 ↪  selection here)
# movaps xmmword ptr [rsp+20h], xmm0
```

Symptom: crash appears in a function that uses SIMD, not in the caller that caused misalignment.

Pattern 2: Crash Only When Optimization Is Enabled

At low optimization, the compiler may:

- spill less,
- use different instructions,
- insert a frame pointer.

At high optimization, it may introduce aligned SIMD spills or vectorized code that assumes correct alignment, exposing the bug.

Pattern 3: "Random" Corruption After Returning

If the caller miscalculates stack adjustment (e.g., allocates shadow space but restores incorrectly), the function may return to the wrong address or restore wrong saved registers.

```
.intel_syntax noprefix
# WRONG: allocate 40h but restore only 20h
sub  rsp, 40h
call target
add  rsp, 20h      # stack imbalance
ret                # returns using a corrupted stack
```

Pattern 4: Failures in Callbacks and Indirect Calls

Callbacks often cross module boundaries. If alignment and shadow space rules are violated in the trampoline or callback wrapper, crashes occur inside the callback or only after it returns.

```
.intel_syntax noprefix
# WRONG: indirect call without preserving alignment/shadow rules
mov  rax, qword ptr [rip+fp]
call rax
ret
fp: .quad 0
```

Operational Checklist: Stack Correctness at Call Boundaries

- Every call site reserves at least `20h` bytes shadow space.
- Ensure call-site stack alignment is correct for every path.
- Never write below `RSP` on Windows x64 (no red zone).

- Restore `RSP` exactly (no imbalance).
- Treat large stack allocations as a special case (probing requirements).

Chapter 6

Prologues, Epilogues, and Unwinding

6.1 Standard Function Prologue Structure

On Windows x64, function prologues are not merely “style”. They are closely tied to correctness because the platform relies on unwind metadata to walk the stack during exceptions and diagnostics. Compilers therefore generate prologues that follow well-defined patterns.

A typical prologue performs some subset of:

- allocate the local stack frame (`sub rsp, imm`)
- save non-volatile GPRs that will be modified (`push` or `mov [rsp+off], reg`)
- save non-volatile XMM registers if used (`movdqu` / `movaps` depending on alignment guarantees)
- optionally establish a frame pointer (`rbp`) in some builds/configurations
- optionally home incoming argument registers to memory (often using the caller-provided shadow space)

Minimal Leaf Prologue (No Locals, No Saves)

A leaf function that uses only volatile registers may require no explicit prologue:

```
.intel_syntax noprefix
leaf_add:
    lea  eax, [ecx+1]      # arg1 in ECX, return in EAX
    ret
```

Typical Prologue With Locals and Saved Non-Volatiles

```
.intel_syntax noprefix
# Example shape: saves RBX (non-volatile), allocates locals.
# Offsets are illustrative; real compilers choose specific layouts.

func:
    push rbx
    sub  rsp, 30h          # locals + alignment padding (example)
    # ... body ...
```

Using Shadow Space to Home Arguments

Even if a callee does not allocate locals, it may home register arguments into shadow space:

```
.intel_syntax noprefix
# On entry, caller-provided shadow space exists.
# Callee may store RCX/RDX/R8/R9 there immediately.

callee_home_args:
    mov  qword ptr [rsp+08h], rcx
    mov  qword ptr [rsp+10h], rdx
```

```
    mov  qword ptr [rsp+18h], r8
    mov  qword ptr [rsp+20h], r9
    # ... body ...
    ret
```

This is one of the reasons shadow space is mandatory: it lets the callee spill arguments without first moving `RSP`.

6.2 Epilogue Rules and Stack Cleanup

The epilogue must exactly reverse what the prologue did, restoring non-volatile state and returning with a correct stack pointer.
Typical epilogue responsibilities:

- restore any saved non-volatile registers
- deallocate locals (undo `sub rsp, imm` via `add rsp, imm`)
- return using `ret`

Correct Symmetric Epilogue

```
.intel_syntax noprefix
func:
    push rbx
    sub  rsp, 30h
    # ... body ...
    add  rsp, 30h
    pop  rbx
    ret
```

Classic Stack Imbalance Bug

```
.intel_syntax noprefix
# WRONG: allocates 30h but frees only 20h
bad_func:
    push rbx
    sub  rsp, 30h
    # ... body ...
    add  rsp, 20h          # imbalance
    pop  rbx               # now pops wrong value
    ret                    # returns to wrong address or corrupts
     ↪  state
```

This class of bug often produces:

- crashes far away from the real mistake,
- corrupted return addresses,
- “random” failures under optimization.

6.3 Structured Exception Handling (SEH) Implications

Windows x64 uses a structured exception mechanism where correct stack unwinding depends on compiler-provided unwind metadata rather than ad-hoc conventions.
Key implications for ABI correctness:

- The OS unwinder must be able to determine how `RSP` changes and which non-volatiles were saved.
- The OS expects prologues/epilogues to be representable by unwind codes recorded in metadata.

- Hand-written assembly that does not provide correct unwind information can break:
  - exceptions (C++ exceptions, OS exceptions),
  - stack walking in crash dumps,
  - debugging and profiling tools.

Why "It Works Until an Exception" Happens

A function can appear to run correctly in normal execution but fail during an exception because the unwinder cannot correctly restore the caller context.
Example scenario:

- Your function modifies `RSP` and saves `RBX`.
- Normal return path restores correctly.
- An exception occurs inside a nested call.
- Unwinder tries to walk through your frame.
- Without accurate unwind data, the unwinder restores the wrong `RSP`/registers.

Result: the exception handling path corrupts execution state, often crashing in runtime code or in an unrelated handler.

6.4 Why Unwind Metadata Affects Correctness

Unwind metadata is not optional decoration; it is part of the platform correctness story.
Unwind metadata encodes facts such as:

- how much stack space was allocated,

- where non-volatile registers were saved,
- whether a frame pointer is used,
- how to restore the caller context.

This affects correctness in at least four ways:

- **Exception safety:** stack unwinding must restore the correct register state.
- **Crash reliability:** crash dumps need accurate stack traces.
- **Debugger correctness:** stepping and backtraces depend on unwindable frames.
- **Profiler accuracy:** sampling profilers rely on stack walking.

Practical Consequence for Hand-Written Assembly

If you write assembly that:

- changes `RSP`,
- saves non-volatiles,
- makes calls,

you are effectively responsible for ensuring that the platform can unwind through it correctly.
Practical rule: keep hand-written assembly leaf-only when possible, or use compiler-supported mechanisms for emitting correct unwind info.

6.5 ABI Impact on Debugging and Crash Analysis

Many “mysterious” crash reports are actually ABI violations that manifest as:

- incorrect or truncated stack traces,
- return addresses pointing into the middle of instructions,
- crashes during unwinding rather than at the original fault,
- exceptions that cannot be caught as expected,
- debugger showing impossible local variables or call frames.

Pattern 1: Broken Stack Trace

If the stack pointer is misaligned or imbalanced, stack walking yields nonsense frames:

```
.intel_syntax noprefix
# Typical cause: mismatched sub/add on RSP, or missing shadow space
 ↪  on calls
# Consequence: backtrace shows wrong function names or stops early.
```

Pattern 2: Crash in a Library Routine After Returning

A function returns with corrupted non-volatile registers (e.g., `RBX`):

```
.intel_syntax noprefix
# WRONG: modifies RBX without preserving it
bad_preserve:
    mov  rbx, 0
    ret
```

The crash may occur later in a different function that relies on `RBX` being preserved.

Pattern 3: Crash Only When Exceptions Are Enabled

Without unwindable frames, exceptions can fail during stack unwinding, producing crashes that disappear when exceptions are disabled or when code is compiled differently.

Operational Checklist: Prologue/Epilogue and Unwind Safety

- Preserve all non-volatile registers you modify.
- Restore `RSP` exactly (no imbalance).
- Allocate shadow space at every call site.
- Avoid using memory below `RSP` (no red zone).
- Treat unwindability as a correctness requirement, not a debugging convenience.

Chapter 7

Interoperability Pitfalls

7.1 Mixing MSVC with Clang or GCC on Windows

On Windows x64, the *calling convention* is mostly unified, but interoperability problems still arise because compilers differ in:

- name mangling (C++),
- object layout rules influenced by compilation modes and pragmas,
- exception model and runtime libraries,
- structure/aggregate passing and return lowering details,
- vector types and alignment assumptions,
- linker/runtime expectations (CRT selection, initialization order, TLS).

Rule 1: Cross-compiler boundaries must be C-compatible

The safest interop contract is:

- `extern "C"` functions,
- fixed-width integer types,
- explicit pointer-based APIs (no C++ templates/overloads across boundaries),
- avoid passing/returning non-trivial C++ objects across DLL boundaries.

```
.intel_syntax noprefix
# ABI-safe boundary design (conceptual):
# Prefer: extern "C" void api_do_work(Context* ctx, int32_t x);
# Avoid: passing std::string / exceptions / C++ class types across
 ↪  compilers/DLLs.
```

Rule 2: Do not mix exception runtimes across boundaries

Even when the calling convention matches, crossing module boundaries with C++ exceptions can fail if modules use different runtimes or exception handling models. The result is typically:

- uncaught exceptions,
- termination in unexpected locations,
- unwinding failures that look like stack corruption.

Rule 3: Structure layout must be identical

A struct that “looks the same” in source may differ due to:

- packing pragmas,
- alignment directives,

- compiler flags that affect ABI layout,
- different definitions in different translation units.

Practical enforcement: define shared structs in a single header used by all compilers, and keep them plain-old-data with explicit padding when needed.

7.2 Calling Windows x64 ABI from Hand-Written Assembly

When calling into C/C++ from hand-written assembly, you must obey **all** call-site rules:

- arguments in `RCX, RDX, R8, R9` (and XMM registers for FP),
- allocate 32-byte shadow space for every call,
- maintain stack alignment at the call boundary,
- assume volatile registers are clobbered by the callee.

Correct Minimal Call Site

```
.intel_syntax noprefix
# void f(int64_t a, int64_t b);

call_f:
    sub  rsp, 40h          # shadow + alignment padding
    mov  rcx, 111          # a
    mov  rdx, 222          # b
    call f
    add  rsp, 40h
    ret
```

Classic Bug: Missing Shadow Space

```
.intel_syntax noprefix
# WRONG: missing shadow space (may corrupt stack when callee homes
 ↪  args)
bad_call_f:
    mov  rcx, 111
    mov  rdx, 222
    call f
    ret
```

This often appears to work in simple tests, then fails in release builds when the callee spills or when instrumentation/unwinding paths execute.

Classic Bug: Treating RDI/RSI as arg registers

Some developers reuse System V snippets:

```
.intel_syntax noprefix
# WRONG on Windows x64
mov  rdi, 111
mov  rsi, 222
call f
```

Correct Windows x64 register usage:

```
.intel_syntax noprefix
sub  rsp, 40h
mov  rcx, 111
mov  rdx, 222
call f
add  rsp, 40h
```

7.3 Interfacing with Foreign Languages and Runtimes

Interfacing with other languages typically introduces additional ABI layers:

- language runtime calling convention wrappers,
- different object representations and lifetimes,
- different exception and stack-unwinding expectations,
- different alignment/packing defaults.

Rule 1: Use a C ABI surface

Export C-callable functions and keep them narrow:

- pointers and integers,
- explicit buffer lengths,
- explicit ownership rules (`create/free` pairs).

```
.intel_syntax noprefix
# Example boundary pattern (conceptual):
# extern "C" void* lib_create();
# extern "C" void  lib_destroy(void*);
# extern "C" int32_t lib_process(void* ctx, const uint8_t* data,
 ↪  uint32_t len);
```

Rule 2: Never pass C++ exceptions across language boundaries

Foreign runtimes generally cannot unwind C++ frames correctly unless explicitly integrated. Treat the boundary as **no-throw**:

- catch all exceptions inside the C++ side,
- return error codes/status objects.

Rule 3: Avoid passing non-trivial objects and STL types

Even within C++, passing `std::string`, `std::vector`, or class types across module boundaries is risky due to allocator and runtime coupling. Across foreign languages it is almost always incorrect.

7.4 Callback Mismatches and Silent Memory Corruption

Callbacks are the most common place where ABI errors become **silent corruption** rather than immediate crashes.
A callback mismatch occurs when:

- the caller expects one signature/calling convention,
- the callee implements another.

Mismatch Type 1: Wrong Prototype (Argument Count/Types)

If a callback is declared with fewer arguments than the caller supplies, the callee may ignore registers but may still clobber state unpredictably if it uses a different interpretation of registers or stack.

```
.intel_syntax noprefix
# Conceptual mismatch:
# expected: void cb(void* ctx, int32_t code)
# actual:   void cb(void* ctx)    # wrong prototype
#
```

```
# Symptom: appears to work until optimization or until the callback
↪  uses registers differently.
```

Mismatch Type 2: Mixed Integer/FP Classification

If a callback signature differs in whether an argument is integer or floating-point, the runtime will pass it in different registers (GPR vs XMM). This produces values that look like random bits.

```
.intel_syntax noprefix
# Conceptual mismatch:
# expected: void cb(double x)   # expects XMM0
# actual:   void cb(int64_t x)  # reads RCX
#
# Result: callee reads garbage.
```

Mismatch Type 3: Stack Alignment Violation in Callback Trampolines

Callbacks often go through trampolines/wrappers. If the wrapper fails to allocate shadow space or maintain alignment, the crash occurs inside the callback, not in the wrapper.

```
.intel_syntax noprefix
# WRONG callback trampoline: no shadow space
trampoline_bad:
    mov  rcx, rdi           # random wrong assumption (also wrong
    ↪  register choice)
    call rax                # indirect call without shadow/alignment
    ↪  discipline
    ret
```

```
.intel_syntax noprefix
```

```
# Correct trampoline pattern: shadow + correct arg registers
trampoline_ok:
    sub  rsp, 40h
    # set RCX/RDX/R8/R9 as needed
    call rax
    add  rsp, 40h
    ret
```

Why it becomes silent corruption: the callback may return successfully, but non-volatile registers or stack state may be corrupted, and the crash happens later.

7.5 DLL Boundary ABI Hazards

Crossing a DLL boundary introduces additional correctness hazards beyond the calling convention.

Hazard 1: Different CRTs and Allocators

If one module allocates memory and another frees it using a different runtime allocator, the result ranges from leaks to heap corruption.

Safe pattern:

- allocate and free within the same module, or
- export paired functions: `lib_alloc`/`lib_free`.

Hazard 2: Passing C++ Objects Across DLL Boundaries

Passing objects with:

- non-trivial destructors,

- inline methods compiled differently,
- vtables and RTTI,
- STL members,

is extremely fragile across DLL boundaries, especially when compilers/runtimes differ.
Safe pattern: opaque handles.

```
.intel_syntax noprefix
# Opaque-handle boundary model (conceptual):
# typedef struct Handle Handle;
# extern "C" Handle* create();
# extern "C" void destroy(Handle*);
```

Hazard 3: Different Alignment/Packing or Header Drift

If two modules compile with different packing/alignment, structs shared across the boundary will not match. The bug often appears as:

- wrong field values,
- crashes when dereferencing pointers that were read from the wrong offset,
- subtle corruption in large structs.

Hazard 4: Inlining and ODR-Style Mismatches

Even if the binary interface is “the same”, differences in inline code or macro-controlled layouts can produce inconsistent assumptions across modules.

Operational Checklist: Interop That Does Not Break

- Use `extern "C"` for exported/imported APIs.
- Use only plain data types across boundaries; prefer pointers + sizes.
- Never let C++ exceptions cross module or language boundaries.
- Allocate and free in the same module (or export `free`).
- Ensure shadow space and stack alignment at every call site, including callbacks.
- Avoid non-trivial C++ objects and STL types across DLL boundaries.
- Validate mixed compiler output with disassembly for boundary functions.

Chapter 8

Windows x64 vs System V ABI

8.1 Side-by-Side Calling Convention Comparison

Although both ABIs target the same ISA (*x86-64*), they define different binary contracts. Treating them as "almost the same" is a direct path to stack corruption, wrong arguments, and broken callbacks.

High-Impact Differences (Summary Table)

Category	**Windows x64 ABI**	**System V AMD64 ABI**
Integer / pointer argument registers	RCX, RDX, R8, R9	RDI, RSI, RDX, RCX, R8, R9
Floating-point argument registers	XMM0–XMM3	XMM0–XMM7
Shadow space	Mandatory 32 bytes reserved by the caller for every call	None

Category	Windows x64 ABI	System V AMD64 ABI
Red zone	None (memory below RSP is not usable)	128-byte red zone available below RSP
Stack arguments	Placed on stack above shadow space	Placed directly on stack without shadow space
Caller-saved general-purpose registers (typical)	RAX, RCX, RDX, R8–R11	RAX, RCX, RDX, RSI, RDI, R8–R11
Callee-saved general-purpose registers (typical)	RBX, RBP, RSI, RDI, R12–R15	RBX, RBP, R12–R15
Non-volatile XMM registers	XMM6–XMM15	None (all XMM registers are caller-saved)
Unwinding integration	OS-centric unwind metadata tightly coupled with the Windows exception model	DWARF-based CFA rules used by platform tooling

Note: “caller-saved” and “callee-saved” here mean ABI-level volatility rules that must hold across module boundaries.

8.2 Register Allocation Differences

Integer / Pointer Argument Registers

Windows x64 passes up to four integer/pointer arguments in:

$$(\texttt{RCX}, \texttt{RDX}, \texttt{R8}, \texttt{R9})$$

System V passes up to six integer/pointer arguments in:

$$(\texttt{RDI}, \texttt{RSI}, \texttt{RDX}, \texttt{RCX}, \texttt{R8}, \texttt{R9})$$

Example: call `f(a,b,c,d,e,f)` with 6 integers.

Windows x64 call setup

```
.intel_syntax noprefix
# Windows x64: first four in RCX,RDX,R8,R9 ; remaining on stack
# Caller must allocate 32-byte shadow space.

call_f_win64:
    sub  rsp, 40h
    mov  rcx, 1              # a
    mov  rdx, 2              # b
    mov  r8,  3              # c
    mov  r9,  4              # d
    mov  qword ptr [rsp+20h], 5   # e
    mov  qword ptr [rsp+28h], 6   # f
    call f
    add  rsp, 40h
    ret
```

System V call setup

```
.intel_syntax noprefix
# System V: first six in RDI,RSI,RDX,RCX,R8,R9
# No shadow space requirement.
```

```
call_f_sysv:
    mov  rdi, 1             # a
    mov  rsi, 2             # b
    mov  rdx, 3             # c
    mov  rcx, 4             # d
    mov  r8,  5             # e
    mov  r9,  6             # f
    call f
    ret
```

A direct reuse of one sequence on the other platform produces wrong arguments immediately.

Floating-Point Argument Registers

Windows x64 uses XMM0–XMM3 for up to four FP args. System V uses XMM0–XMM7 for up to eight FP args.
This difference matters for:

- FP-heavy APIs,
- callbacks,
- varargs,
- mixed integer/FP signatures where register assignment is type-class dependent.

8.3 Stack Discipline Differences

Shadow Space Presence vs Absence

Windows x64 requires the caller to reserve 32 bytes (shadow space) at every call site. System V has no such requirement.

Consequence: the stack layout observed by callees differs. Even when both pass the same number of parameters, stack offsets for additional arguments are not interchangeable.

Alignment and Call Boundaries

Both ABIs require careful alignment for correctness, but the compiler-generated call-site patterns differ due to:

- shadow space reservation on Windows,
- red zone availability on System V,
- different prologue/epilogue conventions aligned with platform unwinding models.

A common Windows call-site pattern:

```
.intel_syntax noprefix
sub  rsp, 40h
call target
add  rsp, 40h
```

System V often uses smaller adjustments or none for leaf calls (because of red zone), which is not valid to transplant to Windows.

8.4 Shadow Space vs Red Zone

Windows x64: Shadow Space

- mandatory 32 bytes reserved by caller,
- callee may home RCX/RDX/R8/R9 there,
- exists on every call, even for zero-argument calls.

System V: Red Zone

- 128 bytes below `RSP` are usable by leaf functions without changing `RSP`,
- improves leaf function performance by avoiding stack pointer updates,
- not available on Windows x64.

A red-zone leaf pattern (valid System V, invalid Windows x64):

```
.intel_syntax noprefix
# System V red zone usage: writing below RSP without sub rsp, ...
leaf_sysv_ok:
    mov  qword ptr [rsp-08h], rbx
    xor  eax, eax
    ret
```

Correct Windows x64 approach:

```
.intel_syntax noprefix
leaf_win64_ok:
    sub  rsp, 20h
    mov  qword ptr [rsp+00h], rbx
    xor  eax, eax
    add  rsp, 20h
    ret
```

8.5 Why Direct Code Reuse Fails Across Platforms

Directly copying assembly or “low-level” C/C++ assumptions across Windows x64 and System V fails for five recurring reasons:

1) Wrong argument registers

System V uses `RDI/RSI` first; Windows x64 uses `RCX/RDX`. Reusing call setup code passes garbage to the callee.

2) Missing shadow space on Windows

System V code does not allocate shadow space. If reused on Windows, the callee may overwrite stack memory when homing arguments:

```
.intel_syntax noprefix
# WRONG on Windows x64: call without shadow space
mov  rcx, 1
call target
```

3) Red zone assumptions

System V leaf code may use memory below `RSP`. On Windows, this is unsafe and violates the ABI.

4) Different non-volatile register sets

On Windows x64, `RSI` and `RDI` are non-volatile and must be preserved by the callee. On System V, `RSI` and `RDI` are volatile (caller-saved). Copying a callee that clobbers `RSI/RDI` from System V into Windows breaks callers.

```
.intel_syntax noprefix
# Valid as a SysV callee behavior (RSI/RDI volatile),
# but WRONG on Windows x64 if not preserved.

bad_on_win64:
```

```
    mov  rsi, 0
    mov  rdi, 0
    ret
```

5) Different unwinding ecosystems

Windows x64 stack walking and exception handling depend heavily on unwind metadata and representable prologue/epilogue patterns. System V uses a different tooling model (DWARF/CFA rules). Hand-written assembly that ignores unwind requirements tends to fail in:

- crash dumps,
- debugging,
- exception unwinding paths,
- profilers.

Practical Cross-Platform Rule

- Do not copy raw assembly across Windows x64 and System V.
- Treat the ABI as part of the platform, just like the OS and toolchain.
- When portability is required, implement separate ABI-specific assembly files and keep the boundary C-compatible.

Chapter 9

Real-World Failure Patterns

9.1 Stack Corruption That Appears Unrelated

ABI bugs often corrupt the stack in a way that does not crash immediately. The program continues until:

- a later function returns using a corrupted return address,
- a saved non-volatile register is restored from the wrong location,
- an exception triggers unwinding through a broken frame,
- a library routine uses an aligned stack local and faults.

Pattern 1: Stack Imbalance (Mismatched `sub/add rsp`)

```
.intel_syntax noprefix
# WRONG: allocates 40h but restores 20h
bad_stack_cleanup:
```

```
    sub  rsp, 40h
    call target
    add  rsp, 20h       # imbalance
    ret                 # return address is now wrong
```

Observed symptoms:

- crash in an unrelated function after several returns,
- corrupted call stack in debugger,
- access violation at a seemingly random address.

Pattern 2: Missing Shadow Space Corrupts Caller's Frame

```
.intel_syntax noprefix
# WRONG: missing 32-byte shadow space
bad_shadow_call:
    mov  rcx, 1
    mov  rdx, 2
    call callee
    ret
```

If `callee` homes arguments into `[rsp+08h..20h]`, it overwrites the caller's stack data (saved registers, locals, or even the return chain). The crash often occurs later, far from the call.

Pattern 3: Clobbering Non-Volatile Registers

```
.intel_syntax noprefix
# WRONG on Windows x64: RBX must be preserved
```

```
bad_preserve_rbx:
    mov  rbx, 0
    ret
```

Observed symptoms:

- caller data structure pointer "mysteriously" changes,
- crash in code that uses RBX long after the bad function returned,
- failures that disappear if you add logging (changes register pressure / spills).

9.2 Random Crashes After Seemingly Correct Calls

Some calls look correct (arguments in registers, correct symbol, correct return type), yet crash randomly. This is typical when only *some* ABI rules are followed.

Pattern 4: Call Looks Correct but Alignment Is Wrong

```
.intel_syntax noprefix
# WRONG: may break alignment depending on entry state
sub  rsp, 20h
mov  rcx, 123
call target
add  rsp, 20h
ret
```

Why it looks correct: shadow space exists (`20h`). **Why it still breaks:** call-site alignment may be wrong for the callee's assumptions, leading to crashes when aligned SIMD spills or locals are used.

Pattern 5: Indirect Call / Callback Without ABI Discipline

```
.intel_syntax noprefix
# WRONG: indirect call without shadow space
mov  rax, qword ptr [rip+fp]
mov  rcx, 7
call rax
ret
fp: .quad 0
```

The crash may occur:

- inside the callback,
- when the callback returns,
- later when unwinding or returning through the corrupted frame.

9.3 Bugs Triggered Only Under Optimization

Optimization changes register allocation, spilling behavior, prologue/epilogue form, and inlining. ABI violations that are latent in debug builds become fatal in release builds.

Pattern 6: Release Build Starts Homing Arguments

Debug build callee might not spill arguments; release build might home arguments immediately:

```
.intel_syntax noprefix
# callee in release may do:
mov  qword ptr [rsp+08h], rcx
mov  qword ptr [rsp+10h], rdx
```

If the caller forgot shadow space, release build overwrites the caller's stack.

Pattern 7: Vectorization Introduces Alignment-Sensitive Spills

Under optimization, compilers may:

- use wider vector registers,
- spill XMM registers to stack locals,
- assume ABI-mandated alignment.

A misaligned stack can now crash when the compiler emits aligned stack stores/loads. The fault appears in the callee even though the caller caused it.

Pattern 8: Inlining Hides the True Boundary

Inlining can remove a function boundary that previously "accidentally" repaired damage. In release builds, the ABI violation becomes exposed because:

- the call disappears,
- stack layout changes,
- register lifetimes extend.

9.4 ABI Bugs Mistaken for Compiler Bugs

ABI violations frequently look like "the compiler is broken" because:

- symptoms change with optimization flags,
- adding a print statement makes the bug disappear,

- crash address points into valid code but with a wrong stack,
- the same source behaves differently across compilers.

Red Flags That Indicate ABI Violation, Not a Compiler Bug

- crash disappears when compiling with no optimization
- stack trace is corrupted or ends abruptly
- fault occurs on return (`ret`) or shortly after a return
- fault occurs inside a callee that uses SIMD or exception handling
- only specific call sites trigger the problem

Example: "Compiler Bug" That Is Actually Wrong Register Preservation

```
.intel_syntax noprefix
# Caller assumes RSI is preserved (Windows rule), but callee clobbers
 ↪  it:
bad_callee:
    mov  rsi, 0
    ret
```

The caller later uses RSI and fails. This is not a compiler bug; it is an ABI contract violation.

9.5 Diagnosing ABI Issues Using Disassembly

Disassembly is the most reliable way to confirm ABI correctness. The goal is not to reverse-engineer everything, but to verify a small set of invariants at each boundary.

Step 1: Verify the Call Site

At the call instruction, check:

- shadow space allocation exists (`sub rsp, ...` with at least `20h`)
- alignment at the call boundary is consistent with compiler patterns
- correct argument registers are loaded (`RCX, RDX, R8, R9`)
- for FP args, check XMM0–XMM3 usage

```
.intel_syntax noprefix
# Healthy call-site pattern to recognize:
sub  rsp, 40h
mov  rcx, ...
mov  rdx, ...
call target
add  rsp, 40h
```

Step 2: Inspect the Callee Entry

At callee entry, look for:

- immediate homing of args into shadow space
- saving of non-volatile registers if they will be modified
- stack allocation for locals

```
.intel_syntax noprefix
# Common callee entry patterns:
push rbx
```

```
sub  rsp, 30h
# or
mov  [rsp+08h], rcx
```

If the callee writes to `[rsp+08h..20h]`, the caller *must* have allocated shadow space.

Step 3: Validate Register Preservation

If a callee uses a non-volatile register (RBX, RBP, RSI, RDI, R12–R15, XMM6–XMM15), confirm you see save/restore pairs.

```
.intel_syntax noprefix
# Example preservation shape:
push rbx
# ... uses rbx ...
pop  rbx
ret
```

Absence of preservation is a correctness defect, not an optimization choice.

Step 4: Validate Stack Cleanup Symmetry

Confirm that the function's net `RSP` delta is restored exactly on all exits.

```
.intel_syntax noprefix
# Good symmetry:
sub rsp, 50h
# ...
add rsp, 50h
ret
```

Multiple exits are a common source of imbalance bugs. Check every return path.

Step 5: Confirm "No Red Zone" Assumptions

Search for memory references below `RSP` without allocating stack space:

```
.intel_syntax noprefix
# Suspicious on Windows x64:
mov qword ptr [rsp-08h], rbx
```

This pattern is a strong indicator of transplanted System V assumptions.

Practical Diagnostic Checklist

- Every call reserves shadow space.
- The call-site alignment matches compiler patterns.
- Argument registers match Windows x64 rules.
- Callee preserves non-volatile registers it modifies.
- `RSP` is restored exactly on every exit path.
- No memory below `RSP` is used (no red zone).
- Indirect calls and callbacks follow the same rules as direct calls.

Chapter 10

Writing ABI-Correct Code

10.1 Safe Rules for Cross-ABI Development

Cross-ABI work is any situation where code crosses a boundary that may have a different binary contract:

- Windows x64 vs System V
- MSVC vs Clang/GCC on Windows
- DLL boundary vs static linkage
- C/C++ vs foreign languages/runtimes
- compiler-generated code vs hand-written assembly

Rule 1: Treat the boundary as a C ABI surface

Make the boundary explicit and simple:

- `extern "C"` exported/imported functions
- fixed-width integers (`int32_t`, `uint64_t`)
- pointers + lengths for buffers
- opaque handles for objects
- error codes / status structs (no exceptions crossing boundary)

```
.intel_syntax noprefix
# Boundary contract model (conceptual, ABI-safe):
# extern "C" Handle* api_create();
# extern "C" void    api_destroy(Handle*);
# extern "C" int32_t api_process(Handle*, const uint8_t* data,
 ↪  uint32_t len);
```

Rule 2: Never pass non-trivial C++ objects across boundaries

Avoid across ABI boundaries:

- STL containers and strings
- exceptions
- RTTI and virtual classes
- objects with non-trivial destructors or custom allocators

Use:

- POD structs with explicit layout,
- handles and explicit lifetime functions,
- caller-allocated buffers.

Rule 3: Do not reuse assembly across Windows and System V

Implement separate ABI-specific files. Shared logic may exist, but the call/stack/register glue must be per-ABI.

10.2 Assembly Guidelines for Windows x64

When you write assembly that calls C/C++ or is called by C/C++, the ABI rules are the correctness rules.

1) Always allocate shadow space at every call site

```
.intel_syntax noprefix
# Correct call site: reserve shadow + keep alignment
sub  rsp, 40h
mov  rcx, 1
mov  rdx, 2
call target
add  rsp, 40h
```

2) Use the correct argument registers

- integer/pointer args: `RCX, RDX, R8, R9`
- floating-point args: `XMM0--XMM3` (for the first four FP args)
- return values: `RAX` (integer/pointer), `XMM0` (FP)

```
.intel_syntax noprefix
# f(int64_t a, double x) -> returns int64_t
sub  rsp, 40h
```

```
mov  rcx, 123
movsd xmm1, qword ptr [rip+val_x]   # FP arg (illustrative pattern;
 ↪  verify signature)
call f
add  rsp, 40h
# RAX now holds return
ret
val_x: .quad 0x400921FB54442D18
```

Practical warning: mixed integer/FP signatures require exact ABI classification. When in doubt, generate a reference call in C/C++ and match the compiler output.

3) Preserve non-volatile registers if you modify them

Non-volatile GPRs include:

RBX, RBP, RSI, RDI, R12, R13, R14, R15

Non-volatile XMM registers include:

XMM6--XMM15

```
.intel_syntax noprefix
# Correct preservation pattern for RBX
use_rbx:
    push rbx
    mov  rbx, 0
    # ...
    pop  rbx
    ret
```

4) Never rely on a red zone

Windows x64 has no red zone. Do not write below `RSP` unless you explicitly allocated space.

```
.intel_syntax noprefix
# WRONG on Windows x64:
mov qword ptr [rsp-08h], rbx
```

5) Keep stack changes representable and symmetric

A function must restore `RSP` exactly on all exit paths.

```
.intel_syntax noprefix
# Good symmetry
sub rsp, 50h
# ...
add rsp, 50h
ret
```

10.3 C and C++ Interoperability Best Practices

1) Prefer `extern "C"` for inter-module APIs

- avoids C++ name mangling,
- stabilizes linkage,
- reduces cross-compiler fragility.

```
.intel_syntax noprefix
# Conceptual pattern:
# extern "C" int32_t api_sum(int32_t a, int32_t b);
# Use plain types and explicit calling surface.
```

2) Make ownership explicit

Across boundaries:

- allocate/free in the same module, or
- provide paired functions from the same module.

```
.intel_syntax noprefix
# Safe: library owns allocations
# extern "C" void* lib_alloc(uint32_t n);
# extern "C" void  lib_free(void* p);
```

3) Do not pass exceptions across boundaries

Catch inside the boundary and translate to:

- error codes,
- status structs,
- out-parameters.

4) Avoid varargs across boundaries

Varargs create additional ABI complexity. Prefer explicit parameter lists or structured payloads.

5) Freeze struct layout intentionally

If you must pass a struct:

- keep it trivially copyable,

- use explicit fixed-width fields,
- control padding explicitly if needed,
- avoid compiler-dependent types.

10.4 How to Reason About ABI Correctness

Reasoning about ABI correctness is a discipline of invariants at boundaries. For any function boundary, ask:

Boundary Invariants (Must Hold)

- Are arguments placed in the correct registers / stack locations for this ABI?
- Is shadow space present for every call site? (Windows x64)
- Is stack alignment correct at the call boundary?
- Are non-volatile registers preserved if modified?
- Is `RSP` restored exactly on every exit path?
- Are stack writes limited to allocated stack space? (no red zone on Windows)
- If exceptions/unwinding can occur, can the platform unwind through this frame correctly?

Use the compiler as a reference oracle

For critical boundaries:

- write a tiny C/C++ wrapper that makes the call,

- compile with the target toolchain,
- compare your hand-written assembly to compiler output.

```
.intel_syntax noprefix
# Practical workflow (conceptual):
# 1) Implement: extern "C" int64_t f(int64_t, int64_t);
# 2) Compile a wrapper call in C/C++.
# 3) Inspect generated call sequence.
# 4) Mirror it exactly in hand assembly.
```

10.5 Checklist Before Blaming the Compiler

When a crash looks random or optimization-dependent, first assume an ABI violation until proven otherwise.

Call-Site Checklist

- Every `call` has at least 32-byte shadow space reserved.
- Stack alignment is correct at every call boundary (all paths).
- Correct argument registers are used (`RCX`, `RDX`, `R8`, `R9`).
- Stack arguments (5th+) are placed at correct offsets above shadow space.
- Indirect calls and callbacks follow the same rules as direct calls.

Callee Checklist

- Non-volatile registers are preserved if modified (GPR and XMM6–XMM15).

- `RSP` is restored exactly on every exit.
- No memory below `RSP` is used (no red zone).
- Prologue/epilogue patterns are consistent with unwind expectations (especially if non-leaf).

Boundary Design Checklist

- Cross-module APIs are `extern "C"` with plain types.
- No STL, no exceptions, no RTTI/vtables across boundaries.
- Allocation/free are paired within the same module.
- Struct layouts are shared, frozen, and trivially copyable.

Disassembly Confirmation Checklist

- Verify call-site `sub/add rsp` symmetry.
- Verify homing/spills in callee do not overwrite unexpected areas.
- Verify preservation save/restore pairs for every non-volatile used.
- If the issue disappears with small source changes, suspect register pressure and ABI misuse, not compiler instability.

Conclusion: In practice, the majority of "compiler bug" reports around calls, crashes after returns, and optimization-only faults are ABI violations at boundaries. Establish ABI correctness first; then evaluate toolchain behavior.

Appendices

Appendix A Windows x64 Calling Convention Summary

Argument Passing Quick Reference

General-purpose (integer / pointer) arguments:

- Arg1 → `RCX`
- Arg2 → `RDX`
- Arg3 → `R8`
- Arg4 → `R9`
- Arg5+ → passed on the stack (caller places them in memory above shadow space)

Floating-point arguments (`float/double`):

- FP Arg1 → `XMM0`
- FP Arg2 → `XMM1`
- FP Arg3 → `XMM2`
- FP Arg4 → `XMM3`

Return values:

- Integer / pointer return → `RAX` (or `EAX/AX/AL` for smaller types)
- Floating-point return → `XMM0`
- Some aggregates may return indirectly via a hidden pointer (use out-parameters for stable interop)

Mandatory call-site rule (Windows x64 specific): The caller must allocate **32 bytes of shadow space** for *every* call.

```
.intel_syntax noprefix
# Minimal correct call site (shadow space + common alignment padding)
sub  rsp, 40h
mov  rcx, 1
mov  rdx, 2
call target
add  rsp, 40h
```

Register Preservation Rules

Volatile (caller-saved) GPRs:

- `RAX, RCX, RDX, R8, R9, R10, R11`

Non-volatile (callee-saved) GPRs:

- `RBX, RBP, RSI, RDI, R12, R13, R14, R15`

Volatile XMM registers:

- `XMM0--XMM5`

Non-volatile XMM registers:

- `XMM6--XMM15`

Example: correct preservation of `RBX` and `XMM6`:

```
.intel_syntax noprefix
callee_preserve:
    push rbx
    sub  rsp, 20h
    movdqu xmmword ptr [rsp+00h], xmm6   # save XMM6 (unaligned-safe
     ↪  form)
    # ... modify RBX and XMM6 ...
    movdqu xmm6, xmmword ptr [rsp+00h]   # restore XMM6
    add  rsp, 20h
    pop  rbx
    ret
```

Conceptual Stack Layout Overview

Windows x64 stack behavior is defined around two critical ideas:

- **No red zone:** do not use memory below `RSP`.
- **Shadow space:** 32 bytes reserved by the caller for every call.

Callee entry view (conceptual, before callee adjusts `RSP`):

```
.intel_syntax noprefix
# [rsp+00h] return address
# [rsp+08h] shadow slot 0 (home for RCX)
# [rsp+10h] shadow slot 1 (home for RDX)
```

```
# [rsp+18h] shadow slot 2 (home for R8)
# [rsp+20h] shadow slot 3 (home for R9)
# [rsp+28h] stack arg 5 (if present)
# [rsp+30h] stack arg 6 (if present)
```

Caller responsibilities at call sites:

- allocate shadow space,
- ensure call-boundary alignment,
- place stack arguments above shadow space.

Practical Do's and Don'ts

Do:

- Allocate **32 bytes shadow space** at every call site (even for zero-arg calls).
- Use `RCX`, `RDX`, `R8`, `R9` for the first four integer/pointer args.
- Use `XMM0`–`XMM3` for the first four FP args; return FP in `XMM0`.
- Preserve all non-volatile registers you modify (GPR and XMM6–XMM15).
- Keep `sub/add rsp` perfectly symmetric on all exit paths.
- Keep ABI boundaries C-compatible: `extern "C"`, plain types, explicit ownership.

Don't:

- Do *not* omit shadow space on direct calls, indirect calls, or callbacks.
- Do *not* write below `RSP` (no red zone on Windows x64).

- Do *not* reuse System V assembly call setups (`RDI/RSI` first args).
- Do *not* clobber `RSI` or `RDI` without saving them (they are non-volatile on Windows).
- Do *not* pass STL types, exceptions, or non-trivial C++ objects across DLL/compiler boundaries.
- Do *not* assume debug success implies ABI correctness; release builds expose latent ABI violations.

Appendix B Common Mistakes and How to Detect Them

Missing or Incorrect Shadow Space

What goes wrong: Windows x64 requires the caller to reserve **32 bytes** of shadow (home) space for *every* call. If omitted, the callee may overwrite the caller's stack when it homes register arguments or uses the space for spills.

Classic incorrect pattern (no shadow space):

```
.intel_syntax noprefix
# WRONG: missing 32-byte shadow space
mov  rcx, 1
mov  rdx, 2
call target
ret
```

Correct minimal pattern (shadow space + common alignment padding):

```
.intel_syntax noprefix
sub  rsp, 40h
mov  rcx, 1
```

```
mov  rdx, 2
call target
add  rsp, 40h
ret
```

How to detect:

- In disassembly, search for call sites that do `call` without a preceding `sub rsp, ...` that reserves at least `20h`.
- In the callee, if you see stores to `[rsp+08h..20h]` early, the caller *must* have allocated shadow space.

Typical symptoms:

- crashes far away from the call site,
- corruption that appears only in release builds,
- broken stack traces (debugger cannot unwind).

Stack Misalignment Bugs

What goes wrong: even if shadow space exists, the stack may be misaligned at the call boundary. This can crash in code that uses aligned SIMD stack locals or expects ABI alignment.

Incorrect call-site pattern (shadow exists but alignment may be wrong):

```
.intel_syntax noprefix
# WRONG risk pattern: uses 20h only, may break alignment depending on
 ↪  entry state
sub  rsp, 20h
call target
```

```
add  rsp, 20h
ret
```

Preferred safe call-site pattern:

```
.intel_syntax noprefix
# Common compiler-like safe pattern
sub  rsp, 40h
call target
add  rsp, 40h
ret
```

How to detect:

- Verify that all call sites follow a consistent compiler-like allocation pattern.
- Look for crashes inside functions that use SIMD/vector spills or large local frames.
- If the crash disappears when disabling optimization, suspect alignment at call boundaries.

Typical symptoms:

- crash in a callee on an instruction that touches stack locals,
- crash appears “random” and depends on optimization level,
- changing small unrelated code changes the crash location.

Register Clobbering Errors

What goes wrong: clobbering non-volatile registers without preserving them violates the ABI. The caller assumes these registers survive the call.

Non-volatile GPRs on Windows x64:

`RBX, RBP, RSI, RDI, R12, R13, R14, R15`

Non-volatile XMM registers:

`XMM6--XMM15`

Incorrect callee (clobbers `RBX`):

```
.intel_syntax noprefix
# WRONG: RBX is non-volatile and must be preserved
bad_callee:
    mov  rbx, 0
    xor  eax, eax
    ret
```

Correct callee (preserves `RBX`):

```
.intel_syntax noprefix
good_callee:
    push rbx
    mov  rbx, 0
    xor  eax, eax
    pop  rbx
    ret
```

How to detect:

- In disassembly, if a function writes `RBX/RSI/RDI/R12--R15`, verify save/restore exists.
- For XMM6–XMM15 usage, verify spill/restore exists when the function modifies them.
- If a failure occurs long after a call, suspect non-volatile clobbering.

Typical symptoms:

- data structure pointers become invalid “later”,
- failures disappear when adding logging or debug prints,
- crash appears in unrelated code that simply uses the corrupted register value.

Wrong Function Prototype Assumptions

What goes wrong: the ABI assigns registers based on the callee signature. If the declared prototype does not match the actual implementation, the call passes the wrong registers and/or wrong stack layout.
Common causes:

- incorrect declaration of parameter types (integer vs floating-point),
- incorrect parameter count,
- mismatch in struct passing/return rules,
- mismatch in pointer width or signedness assumptions,
- calling a function pointer with the wrong signature.

Mismatch example: FP vs integer classification

```
.intel_syntax noprefix
# Conceptual mismatch:
# expected: void cb(double x)   -> runtime passes x in XMM0
# actual:   void cb(int64_t x)  -> implementation reads RCX
#
# Result: callee reads garbage bits.
```

Mismatch example: wrong argument count in callback

```
.intel_syntax noprefix
# Conceptual mismatch:
# expected: void cb(void* ctx, int32_t code)
# actual:   void cb(void* ctx)
#
# Result: may appear to work until the callee uses registers
↪  differently.
```

How to detect:

- Validate headers: ensure the declaration used at the call site matches the definition exactly.
- For function pointers/callbacks, ensure the typedef matches the actual function.
- Use disassembly to confirm which registers the callee reads early:
  - reads `RCX/RDX/R8/R9` for integer args,
  - reads `XMM0--XMM3` for FP args.

Debugging and Validation Techniques

ABI validation is about checking boundary invariants.

Technique 1: Confirm the call-site skeleton

For each suspicious call site, confirm the presence of:

- shadow space reservation,
- correct argument registers,

- symmetric stack restore.

```
.intel_syntax noprefix
# Healthy call-site skeleton
sub  rsp, 40h
mov  rcx, ...
mov  rdx, ...
call target
add  rsp, 40h
```

Technique 2: Inspect callee entry for homing/spills

If a callee writes to `[rsp+08h..20h]` early, missing shadow space is the first suspect.

```
.intel_syntax noprefix
# Common homing pattern (callee expects shadow space)
mov  qword ptr [rsp+08h], rcx
mov  qword ptr [rsp+10h], rdx
```

Technique 3: Verify non-volatile preservation

For any function that modifies non-volatile registers, confirm save/restore pairs exist on all paths.

```
.intel_syntax noprefix
# Save/restore shape
push rbx
# ... uses rbx ...
pop  rbx
ret
```

Technique 4: Verify stack symmetry on all exits

Multiple return paths are a frequent source of imbalance. Ensure each exit restores `RSP` exactly.

```
.intel_syntax noprefix
# Good: every return path must have the same net RSP restoration.
sub  rsp, 50h
# ...
add  rsp, 50h
ret
```

Technique 5: Treat optimization-only failures as ABI suspects

If:

- debug build works,
- release build crashes,
- adding a print changes the crash,

then assume an ABI boundary is violated until proven otherwise (shadow space, alignment, preservation, wrong prototype).

Operational Checklist: Quick ABI Triage

- Every call site reserves at least `20h` shadow space (commonly `40h` total).
- Stack restore matches stack allocate exactly on all paths.
- Correct argument registers are loaded (Windows x64: `RCX,RDX,R8,R9`).

- Callee preserves all non-volatiles it modifies (including XMM6–XMM15).
- No memory below `RSP` is used (no red zone).
- Function pointer and callback signatures match exactly, especially FP vs integer types.

Appendix C Preparation for Advanced ABI Topics

This appendix prepares you for advanced ABI work where "basic calling convention knowledge" is not enough. These topics require strict correctness under exceptions, asynchronous events, runtime code generation, and cross-platform portability constraints.

Exception Handling Internals

Advanced ABI work on Windows x64 requires understanding that exception handling is not only a language feature; it is a platform mechanism tightly integrated with stack unwinding and metadata.

What you must be ready to reason about:

- how the OS unwinder walks frames using unwind metadata,
- how saved registers and stack pointer deltas are restored during unwinding,
- how non-standard prologues/epilogues break unwinding,
- how exceptions interact with callbacks, foreign frames, and mixed toolchains.

Minimal correctness model:

- Every non-leaf function must be unwindable by the platform.
- Any code that changes `RSP` in non-trivial ways must have unwind-correct structure.

- Hand-written assembly that is not unwind-aware can “work” until an exception occurs, then fail catastrophically during unwinding.

Failure pattern to recognize:

```
.intel_syntax noprefix
# Symptom pattern (conceptual):
# - code runs normally
# - exception thrown inside nested call
# - crash occurs during unwinding or in a handler
# Root cause: frame cannot be unwound correctly due to ABI/unwind
 ↪  mismatch
```

Preparation tasks:

- Learn to identify compiler prologues/epilogues that are “unwindable”.
- Practice validating that saved non-volatiles can be restored correctly on all exit paths.
- Treat unwind correctness as part of ABI correctness, not as “debug info”.

Syscall Boundaries

A system call boundary is not a normal function-call boundary. It introduces additional constraints:

- privilege transition,
- kernel-controlled clobber rules,
- restricted calling sequences,
- different expectations for register preservation.

ABI-ready mindset:

- Do not assume user-mode calling convention rules apply to kernel transitions.
- Treat system-call entry as a boundary with its own register clobber set.
- Ensure stack alignment and shadow space discipline is correct *before and after* the syscall wrapper, even if the syscall mechanism itself does not use shadow space.

Common interop hazard: mixing low-level syscall stubs with high-level code without a strict wrapper contract.

```
.intel_syntax noprefix
# Correct design concept:
# user-mode code calls a normal ABI-correct wrapper
# wrapper performs syscall transition using the required mechanism
# wrapper returns with ABI preserved for callers
```

Preparation tasks:

- Learn to separate: **user-mode ABI** vs **syscall ABI**.
- Design syscall wrappers that preserve the Windows x64 contract for their callers.
- Validate register preservation around wrappers (non-volatiles must survive).

JIT and Runtime-Generated Code

JIT engines and runtime code generation introduce ABI challenges because the compiler is no longer the sole authority producing correct prologues/epilogues and call sites.

What changes in a JIT world:

- you generate call sequences manually,

- you must allocate shadow space and maintain alignment yourself,
- you must preserve non-volatiles according to ABI,
- you must ensure unwindability if exceptions or stack-walking are expected,
- you must obey memory protection and code-cache rules (W^X discipline and instruction cache coherence policies).

Minimal ABI-correct JIT call stub shape:

```
.intel_syntax noprefix
# JIT-generated call stub (conceptual, must be emitted as machine
 ↪  code by the JIT):
# - reserve shadow space
# - set args
# - call target
# - restore stack

sub  rsp, 40h
mov  rcx, 111
mov  rdx, 222
call rax           # rax holds target address
add  rsp, 40h
ret
```

Preparation tasks:

- Learn to validate JIT stubs using disassembly of generated code.
- Create a strict internal ABI spec for your runtime calling sequences.
- Use small, proven templates for stubs; avoid ad-hoc code emission.

Cross-Platform ABI Abstraction Layers

Cross-platform libraries and runtimes often need to support:

- Windows x64 ABI,
- System V AMD64 ABI,
- other ABIs (ARM64, etc.).

A correct abstraction layer does not hide ABI differences by pretending they do not exist. Instead, it isolates them in a narrow, testable boundary.

Core design principles:

- Keep ABI-specific code in separate translation units (or separate assembler files).
- Use a single C-compatible internal interface for the boundary.
- Avoid sharing raw assembly across ABIs; share algorithms, not calling glue.
- Prefer opaque handles and pointer-based APIs for stable interop.

Boundary pattern (portable):

```
.intel_syntax noprefix
# Portable boundary model (conceptual):
# - common header defines C ABI surface
# - per-platform implementation provides ABI-correct glue
#
# Example:
# extern "C" int32_t abi_call_bridge(void* fn, void* ctx);
```

Preparation tasks:

- Build a two-ABI test harness: one implementation for Windows x64 and one for System V.
- Validate the same logical call through both bridges with disassembly checks.
- Enforce a strict “no C++ objects across boundary” rule to avoid layout/runtime coupling.

Final Readiness Checklist for Advanced ABI Work

Before proceeding to advanced topics, ensure you can do all of the following reliably:

- Inspect a call site and verify shadow space, alignment, and argument registers.
- Inspect a callee and verify preservation of non-volatiles and stack symmetry.
- Detect wrong prototypes by observing which registers are read (GPR vs XMM).
- Explain why unwind metadata affects correctness, not only debugging.
- Write a minimal ABI-correct call stub in assembly for direct and indirect calls.
- Design a C-compatible boundary that remains stable across compilers and DLLs.

References

Official Windows x64 ABI Documentation

The Windows x64 ABI is defined by the platform contract enforced by the operating system, toolchains, and runtime infrastructure. The authoritative sources establish:

- the unified calling convention for user-mode code,
- mandatory shadow (home) space requirements,
- register volatility and preservation rules,
- stack alignment guarantees,
- unwindability requirements integrated with the OS exception model.

These specifications are stable across modern Windows releases and are treated as non-negotiable contracts for interoperability. Any code that crosses module, compiler, or language boundaries is expected to obey these rules exactly. Deviations are considered correctness bugs, not optimizations or implementation choices.
This booklet is aligned strictly with the current Windows x64 ABI behavior as enforced by modern Windows runtimes and debuggers.

Compiler Calling Convention Documentation

Modern Windows toolchains converge on the same ABI contract while differing internally in code generation strategies. Compiler documentation consistently describes:

- argument classification into general-purpose and SIMD registers,
- mandatory caller-reserved shadow space,
- callee-saved versus caller-saved registers,
- stack alignment rules at call boundaries,
- prologue/epilogue patterns that support unwinding.

Important practical conclusions drawn from compiler documentation:

- Differences between compilers do not change the ABI contract.
- Internal instruction selection and frame layout may vary, but the externally visible behavior must remain ABI-correct.
- Hand-written assembly must match the ABI expectations of the compiler-generated code it interacts with.

Throughout this booklet, compiler behavior is treated as a reference oracle for validating ABI correctness rather than as an authority to redefine the ABI itself.

ABI Behavior Observed in Generated Machine Code

In practice, the most reliable confirmation of ABI rules comes from inspecting real compiler output. Across optimization levels and compilers, consistent patterns appear:

- Call sites reserve shadow space even when arguments are passed entirely in registers.
- Callees frequently home argument registers into shadow space early.
- Non-volatile registers are saved and restored in predictable patterns.
- Stack pointer adjustments are symmetric and unwind-friendly.
- No code relies on memory below `RSP` (no red zone).

Representative call-site pattern consistently observed:

```
.intel_syntax noprefix
sub  rsp, 40h
mov  rcx, ...
mov  rdx, ...
call target
add  rsp, 40h
```

Representative callee behavior consistently observed:

```
.intel_syntax noprefix
push rbx
sub  rsp, 30h
mov  qword ptr [rsp+08h], rcx
# ...
add  rsp, 30h
pop  rbx
ret
```

These patterns are not stylistic preferences; they are manifestations of ABI guarantees required for correctness, optimization, and unwind support. This booklet's rules are derived from these stable, observable behaviors.

Cross-References to Earlier Booklets in This Series

This booklet builds directly on concepts introduced earlier in the CPU Programming Series. Readers are expected to be familiar with:

- **Instruction execution and control flow** — how calls, returns, and branches are executed by the CPU.
- **Registers and flags** — general-purpose versus SIMD registers and their roles.
- **Stack fundamentals** — call frames, return addresses, and stack growth.
- **Conceptual ABI foundations** — why ABIs exist and how they enable binary interoperability.

Specifically:

- The explanation of stack discipline and calling mechanics relies on earlier stack-focused booklets.
- The discussion of register volatility assumes familiarity with register roles and lifetimes.
- The analysis of failure patterns depends on understanding how control flow and stack state interact.

This booklet serves as the first architecture-specific deep dive where all those foundations are applied to a real, widely deployed ABI. Subsequent booklets extend this knowledge to advanced topics such as exception internals, syscalls, and cross-platform ABI abstraction. **Position in the series:** this booklet marks the transition from conceptual ABI theory to strict, platform-enforced ABI correctness.