

x64 exception handling

Article • 02/08/2022

An overview of structured exception handling and C++ exception handling coding conventions and behavior on the x64. For general information on exception handling, see [Exception Handling in Visual C++](#).

Unwind data for exception handling, debugger support

Several data structures are required for exception handling and debugging support.

struct RUNTIME_FUNCTION

Table-based exception handling requires a table entry for all functions that allocate stack space or call another function (for example, nonleaf functions). Function table entries have the format:


 Expand table

Size	Value
ULONG	Function start address
ULONG	Function end address
ULONG	Unwind info address

The RUNTIME_FUNCTION structure must be DWORD aligned in memory. All addresses are image relative, that is, they're 32-bit offsets from the starting address of the image that contains the function table entry. These entries are sorted, and put in the .pdata section of a PE32+ image. For dynamically generated functions [JIT compilers], the runtime to support these functions must either use RtlInstallFunctionTableCallback or RtlAddFunctionTable to provide this information to the operating system. Failure to do so will result in unreliable exception handling and debugging of processes.

struct UNWIND_INFO


The unwind data info structure is used to record the effects a function has on the stack pointer, and where the nonvolatile registers are saved on the stack:

 Expand table

Size	Value
UBYTE: 3	Version
UBYTE: 5	Flags
UBYTE	Size of prolog
UBYTE	Count of unwind codes
UBYTE: 4	Frame Register
UBYTE: 4	Frame Register offset (scaled)


Size	Value
USHORT * n	Unwind codes array
variable	Can either be of form (1) or (2) below

(1) Exception Handler

 Expand table

Size	Value
ULONG	Address of exception handler
variable	Language-specific handler data (optional)

(2) Chained Unwind Info

 Expand table

Size	Value
ULONG	Function start address
ULONG	Function end address
ULONG	Unwind info address


The UNWIND_INFO structure must be DWORD aligned in memory. Here's what each field means:

- **Version**

Version number of the unwind data, currently 1.

- **Flags**

Three flags are currently defined:

 Expand table

Flag	Description
UNW_FLAG_EHANDLER	The function has an exception handler that should be called when looking for functions that need to examine exceptions.
UNW_FLAG_UHANDLER	The function has a termination handler that should be called when unwinding an exception.
UNW_FLAG_CHAININFO	This unwind info structure is not the primary one for the procedure. Instead, the chained unwind info entry is the contents of a previous RUNTIME_FUNCTION entry. For information, see Chained unwind info structures . If this flag is set, then the UNW_FLAG_EHANDLER and UNW_FLAG_UHANDLER flags must be cleared. Also, the frame register and fixed-stack allocation fields must have the same values as in the primary unwind info.

- **Size of prolog**

Length of the function prolog in bytes.

- **Count of unwind codes**

The number of slots in the unwind codes array. Some unwind codes, for example, UWOP_SAVE_NONVOL, require more than one slot in the array.

• **Frame register**

If nonzero, then the function uses a frame pointer (FP), and this field is the number of the nonvolatile register used as the frame pointer, using the same encoding for the operation info field of UNWIND_CODE nodes.

• **Frame register offset (scaled)**

If the frame register field is nonzero, this field is the scaled offset from RSP that is applied to the FP register when it's established. The actual FP register is set to $RSP + 16 * \text{this number}$, allowing offsets from 0 to 240. This offset permits pointing the FP register into the middle of the local stack allocation for dynamic stack frames, allowing better code density through shorter instructions. (That is, more instructions can use the 8-bit signed offset form.)

• **Unwind codes array**

An array of items that explains the effect of the prolog on the nonvolatile registers and RSP. See the section on UNWIND_CODE for the meanings of individual items. For alignment purposes, this array always has an even number of entries, and the final entry is potentially unused. In that case, the array is one longer than indicated by the count of unwind codes field.

• **Address of exception handler**

An image-relative pointer to either the function's language-specific exception or termination handler, if flag UNW_FLAG_CHAININFO is clear and one of the flags UNW_FLAG_EHANDLER or UNW_FLAG_UHANDLER is set.

• **Language-specific handler data**


The function's language-specific exception handler data. The format of this data is unspecified and completely determined by the specific exception handler in use.

• **Chained Unwind Info**

If flag UNW_FLAG_CHAININFO is set, then the UNWIND_INFO structure ends with three UWORDS. These UWORDS represent the RUNTIME_FUNCTION information for the function of the chained unwind.

struct UNWIND_CODE

The unwind code array is used to record the sequence of operations in the prolog that affect the nonvolatile registers and RSP. Each code item has this format:

 Expand table

Size	Value
UBYTE	Offset in prolog
UBYTE: 4	Unwind operation code
UBYTE: 4	Operation info

The array is sorted by descending order of offset in the prolog.

Offset in prolog

Offset (from the beginning of the prolog) of the end of the instruction that performs this operation, plus 1 (that is, the offset of the start of the next instruction).

Unwind operation code

Note: Certain operation codes require an unsigned offset to a value in the local stack frame. This offset is from the start, that is, the lowest address of the fixed stack allocation. If the Frame Register field in the UNWIND_INFO is zero, this offset is from RSP. If the Frame Register field is nonzero, this offset is from where RSP was located when the FP register was established. It equals the FP register minus the FP register offset ($16 * \text{the scaled frame register offset in the UNWIND_INFO}$). If an FP register is used, then any unwind code taking an offset must only be used after the FP register is established in the prolog.

For all opcodes except `UWOP_SAVE_XMM128` and `UWOP_SAVE_XMM128_FAR`, the offset is always a multiple of 8, because all stack values of interest are stored on 8-byte boundaries (the stack itself is always 16-byte aligned). For operation codes that take a short offset (less than 512K), the final USHORT in the nodes for this code holds the offset divided by 8. For operation codes that take a long offset ($512\text{K} \leq \text{offset} < 4\text{GB}$), the final two USHORT nodes for this code hold the offset (in little-endian format).

For the opcodes `UWOP_SAVE_XMM128` and `UWOP_SAVE_XMM128_FAR`, the offset is always a multiple of 16, since all 128-bit XMM operations must occur on 16-byte aligned memory. Therefore, a scale factor of 16 is used for `UWOP_SAVE_XMM128`, permitting offsets of less than 1M.

The unwind operation code is one of these values:

- `UWOP_PUSH_NONVOL` (0) 1 node

Push a nonvolatile integer register, decrementing RSP by 8. The operation info is the number of the register. Because of the constraints on epilogs, `UWOP_PUSH_NONVOL` unwind codes must appear first in the prolog and correspondingly, last in the unwind code array. This relative ordering applies to all other unwind codes except `UWOP_PUSH_MACHFRAME`.


- `UWOP_ALLOC_LARGE` (1) 2 or 3 nodes

Allocate a large-sized area on the stack. There are two forms. If the operation info equals 0, then the size of the allocation divided by 8 is recorded in the next slot, allowing an allocation up to $512\text{K} - 8$. If the operation info equals 1, then the unscaled size of the allocation is recorded in the next two slots in little-endian format, allowing allocations up to $4\text{GB} - 8$.

- `UWOP_ALLOC_SMALL` (2) 1 node

Allocate a small-sized area on the stack. The size of the allocation is the operation info field $\times 8 + 8$, allowing allocations from 8 to 128 bytes.

The unwind code for a stack allocation should always use the shortest possible encoding:

 Expand table

Allocation Size	Unwind Code
8 to 128 bytes	<code>UWOP_ALLOC_SMALL</code>

Allocation Size	Unwind Code
136 to 512K-8 bytes	UWOP_ALLOC_LARGE, operation info = 0
512K to 4G-8 bytes	UWOP_ALLOC_LARGE, operation info = 1

• UWOP_SET_FPREG (3) 1 node

Establish the frame pointer register by setting the register to some offset of the current RSP. The offset is equal to the Frame Register offset (scaled) field in the UNWIND_INFO * 16, allowing offsets from 0 to 240. The use of an offset permits establishing a frame pointer that points to the middle of the fixed stack allocation, helping code density by allowing more accesses to use short instruction forms. The operation info field is reserved and shouldn't be used.

• UWOP_SAVE_NONVOL (4) 2 nodes

Save a nonvolatile integer register on the stack using a MOV instead of a PUSH. This code is primarily used for *shrink-wrapping*, where a nonvolatile register is saved to the stack in a position that was previously allocated. The operation info is the number of the register. The scaled-by-8 stack offset is recorded in the next unwind operation code slot, as described in the note above.

• UWOP_SAVE_NONVOL_FAR (5) 3 nodes

Save a nonvolatile integer register on the stack with a long offset, using a MOV instead of a PUSH. This code is primarily used for *shrink-wrapping*, where a nonvolatile register is saved to the stack in a position that was previously allocated. The operation info is the number of the register. The unscaled stack offset is recorded in the next two unwind operation code slots, as described in the note above.

• UWOP_SAVE_XMM128 (8) 2 nodes


Save all 128 bits of a nonvolatile XMM register on the stack. The operation info is the number of the register. The scaled-by-16 stack offset is recorded in the next slot.

• UWOP_SAVE_XMM128_FAR (9) 3 nodes

Save all 128 bits of a nonvolatile XMM register on the stack with a long offset. The operation info is the number of the register. The unscaled stack offset is recorded in the next two slots.

• UWOP_PUSH_MACHFRAME (10) 1 node

Push a machine frame. This unwind code is used to record the effect of a hardware interrupt or exception. There are two forms. If the operation info equals 0, one of these frames has been pushed on the stack:

 Expand table

Location	Value
RSP+32	SS
RSP+24	Old RSP
RSP+16	EFLAGS
RSP+8	CS
RSP	RIP

If the operation info equals 1, then one of these frames has been pushed:

 Expand table

Location	Value
RSP+40	SS
RSP+32	Old RSP
RSP+24	EFLAGS
RSP+16	CS
RSP+8	RIP
RSP	Error code


This unwind code always appears in a dummy prolog, which is never actually executed, but instead appears before the real entry point of an interrupt routine, and exists only to provide a place to simulate the push of a machine frame. `UWOP_PUSH_MACHFRAME` records that simulation, which indicates the machine has conceptually done this operation:

1. Pop RIP return address from top of stack into *Temp*
2. Push SS
3. Push old RSP
4. Push EFLAGS
5. Push CS
6. Push *Temp*
7. Push Error Code (if op info equals 1)

The simulated `UWOP_PUSH_MACHFRAME` operation decrements RSP by 40 (op info equals 0) or 48 (op info equals 1).

Operation info

The meaning of the operation info bits depends upon the operation code. To encode a general-purpose (integer) register, this mapping is used:

 Expand table

Bit	Register
0	RAX
1	RCX
2	RDX
3	RBX
4	RSP

Bit	Register
5	RBP
6	RSI
7	RDI
8 to 15	R8 to R15

Chained unwind info structures

If the UNW_FLAG_CHAININFO flag is set, then an unwind info structure is a secondary one, and the shared exception-handler/chained-info address field contains the primary unwind information. This sample code retrieves the primary unwind information, assuming that `unwindInfo` is the structure that has the UNW_FLAG_CHAININFO flag set.

C++

```
PRUNTIME_FUNCTION primaryUnwindInfo = (PRUNTIME_FUNCTION)&(unwindInfo->UnwindCode[(unwindInfo->CountOfCodes + 1) & ~1]);
```

Chained info is useful in two situations. First, it can be used for noncontiguous code segments. By using chained info, you can reduce the size of the required unwind information, because you do not have to duplicate the unwind codes array from the primary unwind info.

You can also use chained info to group volatile register saves. The compiler may delay saving some volatile registers until it is outside of the function entry prolog. You can record them by having primary unwind info for the portion of the function before the grouped code, and then setting up chained info with a non-zero size of prolog, where the unwind codes in the chained info reflect saves of the nonvolatile registers. In that case, the unwind codes are all instances of UWOP_SAVE_NONVOL. A grouping that saves nonvolatile registers by using a PUSH or modifies the RSP register by using an additional fixed stack allocation is not supported.

An UNWIND_INFO item that has UNW_FLAG_CHAININFO set can contain a RUNTIME_FUNCTION entry whose UNWIND_INFO item also has UNW_FLAG_CHAININFO set, sometimes called *multiple shrink-wrapping*. Eventually, the chained unwind info pointers arrive at an UNWIND_INFO item that has UNW_FLAG_CHAININFO cleared. This item is the primary UNWIND_INFO item, which points to the actual procedure entry point.

Unwind procedure

The unwind code array is sorted into descending order. When an exception occurs, the complete context is stored by the operating system in a context record. The exception dispatch logic is then invoked, which repeatedly executes these steps to find an exception handler:

1. Use the current RIP stored in the context record to search for a RUNTIME_FUNCTION table entry that describes the current function (or function portion, for chained UNWIND_INFO entries).
2. If no function table entry is found, then it's in a leaf function, and RSP directly addresses the return pointer. The return pointer at [RSP] is stored in the updated context, the simulated RSP is incremented by 8, and step 1 is repeated.
3. If a function table entry is found, RIP can lie within three regions: a) in an epilog, b) in the prolog, or c) in code that may be covered by an exception handler.

- Case a) If the RIP is within an epilog, then control is leaving the function, there can be no exception handler associated with this exception for this function, and the effects of the epilog must be continued to compute the context of the caller function. To determine if the RIP is within an epilog, the code stream from RIP onward is examined. If that code stream can be matched to the trailing portion of a legitimate epilog, then it's in an epilog, and the remaining portion of the epilog is simulated, with the context record updated as each instruction is processed. After this processing, step 1 is repeated.
 - Case b) If the RIP lies within the prologue, then control hasn't entered the function, there can be no exception handler associated with this exception for this function, and the effects of the prolog must be undone to compute the context of the caller function. The RIP is within the prolog if the distance from the function start to the RIP is less than or equal to the prolog size encoded in the unwind info. The effects of the prolog are unwound by scanning forward through the unwind codes array for the first entry with an offset less than or equal to the offset of the RIP from the function start, then undoing the effect of all remaining items in the unwind code array. Step 1 is then repeated.
 - Case c) If the RIP isn't within a prolog or epilog, and the function has an exception handler (UNW_FLAG_EHANDLER is set), then the language-specific handler is called. The handler scans its data and calls filter functions as appropriate. The language-specific handler can return that the exception was handled or that the search is to be continued. It can also initiate an unwind directly.
4. If the language-specific handler returns a handled status, then execution is continued using the original context record.
 5. If there's no language-specific handler or the handler returns a "continue search" status, then the context record must be unwound to the state of the caller. It's done by processing all of the unwind code array elements, undoing the effect of each. Step 1 is then repeated.

When chained unwind info is involved, these basic steps are still followed. The only difference is that, while walking the unwind code array to unwind a prolog's effects, once the end of the array is reached, it's then linked to the parent unwind info and the entire unwind code array found there is walked. This linking continues until arriving at an unwind info without the UNW_CHAINED_INFO flag, and then it finishes walking its unwind code array.

The smallest set of unwind data is 8 bytes. This would represent a function that only allocated 128 bytes of stack or less, and possibly saved one nonvolatile register. It's also the size of a chained unwind info structure for a zero-length prolog with no unwind codes.

Language-specific handler

The relative address of the language-specific handler is present in the UNWIND_INFO whenever flags UNW_FLAG_EHANDLER or UNW_FLAG_UHANDLER are set. As described in the previous section, the language-specific handler is called as part of the search for an exception handler or as part of an unwind. It has this prototype:

C++

```
typedef EXCEPTION_DISPOSITION (*PEXCEPTION_ROUTINE) (
    IN PEXCEPTION_RECORD ExceptionRecord,
    IN ULONG64 EstablisherFrame,
    IN OUT PCONTEXT ContextRecord,
    IN OUT PDISPATCHER_CONTEXT DispatcherContext
);
```

ExceptionRecord supplies a pointer to an exception record, which has the standard Win64 definition.

EstablisherFrame is the address of the base of the fixed stack allocation for this function.

ContextRecord points to the exception context at the time the exception was raised (in the exception handler case) or the current "unwind" context (in the termination handler case).

DispatcherContext points to the dispatcher context for this function. It has this definition:

```
C++

typedef struct _DISPATCHER_CONTEXT {
    ULONG64 ControlPc;
    ULONG64 ImageBase;
    PRUNTIME_FUNCTION FunctionEntry;
    ULONG64 EstablisherFrame;
    ULONG64 TargetIp;
    PCONTEXT ContextRecord;
    PEXCEPTION_ROUTINE LanguageHandler;
    PVOID HandlerData;
} DISPATCHER_CONTEXT, *PDISPATCHER_CONTEXT;
```

ControlPc is the value of RIP within this function. This value is either an exception address or the address at which control left the establishing function. The RIP is used to determine if control is within some guarded construct inside this function, for example, a `__try` block for `__try/__except` or `__try/__finally`.

ImageBase is the image base (load address) of the module containing this function, to be added to the 32-bit offsets used in the function entry and unwind info to record relative addresses.

FunctionEntry supplies a pointer to the `RUNTIME_FUNCTION` function entry holding the function and unwind info image-base relative addresses for this function.

EstablisherFrame is the address of the base of the fixed stack allocation for this function.

TargetIp Supplies an optional instruction address that specifies the continuation address of the unwind. This address is ignored if **EstablisherFrame** isn't specified.

ContextRecord points to the exception context, for use by the system exception dispatch/unwind code.

LanguageHandler points to the language-specific language handler routine being called.

HandlerData points to the language-specific handler data for this function.

Unwind helpers for MASM

In order to write proper assembly routines, there's a set of pseudo-operations that can be used in parallel with the actual assembly instructions to create the appropriate .pdata and .xdata. And, there's a set of macros that provide simplified use of the pseudo-operations for their most common uses.

Raw pseudo-operations

 Expand table

Pseudo operation	Description
PROC FRAME [: <i>ehandler</i>]	<p>Causes MASM to generate a function table entry in .pdata and unwind information in .xdata for a function's structured exception handling unwind behavior. If <i>ehandler</i> is present, this proc is entered in the .xdata as the language-specific handler.</p> <p>When the FRAME attribute is used, it must be followed by an .ENDPROLOG directive. If the function is a leaf function (as defined in Function types) the FRAME attribute is unnecessary, as are the remainder of these pseudo-operations.</p>
.PUSHREG <i>register</i>	<p>Generates a UWOP_PUSH_NONVOL unwind code entry for the specified register number using the current offset in the prologue.</p> <p>Only use it with nonvolatile integer registers. For pushes of volatile registers, use an .ALLOCSTACK 8, instead</p>
.SETFRAME <i>register, offset</i>	<p>Fills in the frame register field and offset in the unwind information using the specified register and offset. The offset must be a multiple of 16 and less than or equal to 240. This directive also generates a UWOP_SET_FPREG unwind code entry for the specified register using the current prologue offset.</p>
.ALLOCSTACK <i>size</i>	<p>Generates a UWOP_ALLOC_SMALL or a UWOP_ALLOC_LARGE with the specified size for the current offset in the prologue.</p> <p>The <i>size</i> operand must be a multiple of 8.</p>
.SAVEREG <i>register, offset</i>	<p>Generates either a UWOP_SAVE_NONVOL or a UWOP_SAVE_NONVOL_FAR unwind code entry for the specified register and offset using the current prologue offset. MASM chooses the most efficient encoding.</p> <p><i>offset</i> must be positive, and a multiple of 8. <i>offset</i> is relative to the base of the procedure's frame, which is generally in RSP, or, if using a frame pointer, the unscaled frame pointer.</p>
.SAVEXMM128 <i>register, offset</i>	<p>Generates either a UWOP_SAVE_XMM128 or a UWOP_SAVE_XMM128_FAR unwind code entry for the specified XMM register and offset using the current prologue offset. MASM chooses the most efficient encoding.</p> <p><i>offset</i> must be positive, and a multiple of 16. <i>offset</i> is relative to the base of the procedure's frame, which is generally in RSP, or, if using a frame pointer, the unscaled frame pointer.</p>
.PUSHFRAME [<i>code</i>]	<p>Generates a UWOP_PUSH_MACHFRAME unwind code entry. If the optional <i>code</i> is specified, the unwind code entry is given a modifier of 1. Otherwise the modifier is 0.</p>
.ENDPROLOG	<p>Signals the end of the prologue declarations. Must occur in the first 255 bytes of the function.</p>

Here's a sample function prolog with proper usage of most of the opcodes:

MASM

```
sample PROC FRAME
    db      048h; emit a REX prefix, to enable hot-patching
    push rbp
    .pushreg rbp
    sub rsp, 040h
    .allocstack 040h
    lea rbp, [rsp+020h]
    .setframe rbp, 020h
    movdqa [rbp], xmm7
    .savexmm128 xmm7, 020h ;the offset is from the base of the frame
                           ;not the scaled offset of the frame
    mov [rbp+018h], rsi
    .savereg rsi, 038h
    mov [rsp+010h], rdi
    .savereg rdi, 010h ; you can still use RSP as the base of the frame
                      ; or any other register you choose
    .endprolog
```

```
; you can modify the stack pointer outside of the prologue (similar to alloca)
; because we have a frame pointer.
; if we didn't have a frame pointer, this would be illegal
; if we didn't make this modification,
; there would be no need for a frame pointer

    sub rsp, 060h

; we can unwind from the next AV because of the frame pointer

    mov rax, 0
    mov rax, [rax] ; AV!

; restore the registers that weren't saved with a push
; this isn't part of the official epilog, as described in section 2.5

    movdqa xmm7, [rbp]
    mov rsi, [rbp+018h]
    mov rdi, [rbp-010h]

; Here's the official epilog

    lea rsp, [rbp+020h] ; deallocate both fixed and dynamic portions of the frame
    pop rbp
    ret
sample ENDP
```

For more information about the epilog example, see [Epilog code](#) in [x64 prolog and epilog](#).

MASM macros

In order to simplify the use of the [Raw pseudo-operations](#), there's a set of macros, defined in `ksamd64.inc`, which can be used to create typical procedure prologues and epilogues.

 Expand table

Macro	Description
<code>alloc_stack(n)</code>	Allocates a stack frame of <code>n</code> bytes (using <code>sub rsp, n</code>), and emits the appropriate unwind information (<code>.allocstack n</code>)
<code>save_reg reg, loc</code>	Saves a nonvolatile register <code>reg</code> on the stack at RSP offset <code>loc</code> , and emits the appropriate unwind information. (<code>.savereg reg, loc</code>)
<code>push_reg reg</code>	Pushes a nonvolatile register <code>reg</code> on the stack, and emits the appropriate unwind information. (<code>.pushreg reg</code>)
<code>rex_push_reg reg</code>	Saves a nonvolatile register on the stack using a 2-byte push, and emits the appropriate unwind information (<code>.pushreg reg</code>). Use this macro if the push is the first instruction in the function, to ensure that the function is hot-patchable.
<code>save_xmm128 reg, loc</code>	Saves a nonvolatile XMM register <code>reg</code> on the stack at RSP offset <code>loc</code> , and emits the appropriate unwind information (<code>.savexmm128 reg, loc</code>)
<code>set_frame reg, offset</code>	Sets the frame register <code>reg</code> to be the RSP + <code>offset</code> (using a <code>mov</code> , or an <code>lea</code>), and emits the appropriate unwind information (<code>.set_frame reg, offset</code>)
<code>push_eflags</code>	Pushes the eflags with a <code>pushfq</code> instruction, and emits the appropriate unwind information (<code>.alloc_stack 8</code>)

Here's a sample function prolog with proper usage of the macros:

MASM

```

sampleFrame struct
    Fill    dq ?; fill to 8 mod 16
    SavedRdi dq ?; Saved Register RDI
    SavedRsi dq ?; Saved Register RSI
sampleFrame ends

sample2 PROC FRAME
    alloc_stack(sizeof sampleFrame)
    save_reg rdi, sampleFrame.SavedRdi
    save_reg rsi, sampleFrame.SavedRsi
    .end_prolog

; function body

    mov rsi, sampleFrame.SavedRsi[rsp]
    mov rdi, sampleFrame.SavedRdi[rsp]

; Here's the official epilog

    add rsp, (sizeof sampleFrame)
    ret
sample2 ENDP

```

Unwind data definitions in C

Here's a C description of the unwind data:

C

```

typedef enum _UNWIND_OP_CODES {
    UWOP_PUSH_NONVOL = 0, /* info == register number */
    UWOP_ALLOC_LARGE, /* no info, alloc size in next 2 slots */
    UWOP_ALLOC_SMALL, /* info == size of allocation / 8 - 1 */
    UWOP_SET_FPREG, /* no info, FP = RSP + UNWIND_INFO.FPRegOffset*16 */
    UWOP_SAVE_NONVOL, /* info == register number, offset in next slot */
    UWOP_SAVE_NONVOL_FAR, /* info == register number, offset in next 2 slots */
    UWOP_SAVE_XMM128 = 8, /* info == XMM reg number, offset in next slot */
    UWOP_SAVE_XMM128_FAR, /* info == XMM reg number, offset in next 2 slots */
    UWOP_PUSH_MACHFRAME /* info == 0: no error-code, 1: error-code */
} UNWIND_CODE_OPS;

typedef unsigned char UBYTE;

typedef union _UNWIND_CODE {
    struct {
        UBYTE CodeOffset;
        UBYTE UnwindOp : 4;
        UBYTE OpInfo : 4;
    };
    USHORT FrameOffset;
} UNWIND_CODE, *PUNWIND_CODE;

#define UNW_FLAG_EHANDLER 0x01
#define UNW_FLAG_UHANDLER 0x02
#define UNW_FLAG_CHAININFO 0x04

typedef struct _UNWIND_INFO {
    UBYTE Version : 3;
    UBYTE Flags : 5;
    UBYTE SizeOfProlog;

```

```

    UBYTE CountOfCodes;
    UBYTE FrameRegister : 4;
    UBYTE FrameOffset : 4;
    UNWIND_CODE UnwindCode[1];
    /* UNWIND_CODE MoreUnwindCode[((CountOfCodes + 1) & ~1) - 1];
    * union {
    *     OPTIONAL ULONG ExceptionHandler;
    *     OPTIONAL ULONG FunctionEntry;
    * };
    * OPTIONAL ULONG ExceptionData[]; */
} UNWIND_INFO, *PUNWIND_INFO;

typedef struct _RUNTIME_FUNCTION {
    ULONG BeginAddress;
    ULONG EndAddress;
    ULONG UnwindData;
} RUNTIME_FUNCTION, *PRUNTIME_FUNCTION;

#define GetUnwindCodeEntry(info, index) \
    ((info)->UnwindCode[index])

#define GetLanguageSpecificDataPtr(info) \
    ((PVOID)&GetUnwindCodeEntry((info),((info)->CountOfCodes + 1) & ~1))

#define GetExceptionHandler(base, info) \
    ((PEXCEPTION_HANDLER)((base) + *(PULONG)GetLanguageSpecificDataPtr(info)))

#define GetChainedFunctionEntry(base, info) \
    ((PRUNTIME_FUNCTION)((base) + *(PULONG)GetLanguageSpecificDataPtr(info)))

#define GetExceptionDataPtr(info) \
    ((PVOID)((PULONG)GetLanguageSpecificDataPtr(info) + 1))

```

See also

[x64 software conventions](#)

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)