

ARM64 ABI 约定概述

2019/03/27 •  

本文内容

[定义](#)

[基本要求](#)

[字节排序方式](#)

[对齐方式](#)

[整数寄存器](#)

[浮点/SIMD 寄存器](#)

[系统寄存器](#)

[浮点异常](#)

[参数传递](#)

[返回值](#)

[堆栈](#)

[红色区域](#)

[内核堆栈](#)

[堆栈审核](#)

[异常展开](#)

[循环计数器](#)

[请参阅](#)

采用 64 位模式 (ARMv8 或更高版本体系结构) 在 ARM 处理器上进行编译和运行时, 适用于 Windows 的基本应用程序二进制接口 (ABI) 在大多数情况下都遵循 ARM 的标准 AArch64 EABI。本文重点介绍一些重要假设以及相对于 EABI 中所述内容的更改。有关 32 位 ABI 的信息, 请参阅 [ARM ABI 约定概述](#)。有关标准 ARM EABI 的详细信息, 请参阅[适用于 ARM 体系结构的应用程序二进制接口 \(ABI\)](#)。

定义

随着 64 位支持的引入, ARM 定义了若干术语:

- AArch32 – ARM 定义的旧 32 位指令集体系结构 (ISA), 包括 Thumb 模式执行。
- AArch64 – ARM 定义的新 64 位指令集体系结构 (ISA)。
- ARMv7 -“第 7 代”ARM 硬件的规范, 只包括对 AArch32 的支持。此版本的 ARM 硬件是支持 Windows 的第一个 ARM 版本。

- ARMv8 - “第 8 代”ARM 硬件的规范，包括对 AArch32 和 AArch64 的支持。

Windows 还使用以下术语：

- ARM – 指 32 位 ARM 体系结构 (AArch32)，有时称为 WoA（ARM 上的 Windows）。
- ARM32 – 与上面的 ARM 相同；为清楚起见，在本文档中使用。
- ARM64 – 指 64 位 ARM 体系结构 (AArch64)。没有 WoA64 这类术语。

最后，引用数据类型时，会引用 ARM 中的以下定义：

- 短矢量 – 直接在 SIMD 中表示的一种数据类型，是 8 字节或 16 字节元素组成的矢量。它与其大小（8 字节或 16 字节）对齐，其中每个元素可以为 1、2、4 或 8 字节。
- HFA（同类浮点聚合）– 具有 2 到 4 个相同浮点成员（浮点或双精度）的数据类型。
- HVA（同类短矢量聚合）– 具有 2 到 4 个相同短矢量成员的数据类型。

基本要求

Windows 的 ARM64 版本假设它始终在 ARMv8 或更高版本体系结构上运行。假设硬件中存在浮点和 NEON 支持。

ARMv8 规范描述适用于 AArch32 和 AArch64 的新可选加密和 CRC 帮助程序操作码。对它们的支持当前是可选的，但建议使用。若要利用这些操作码，应用应首先在进行运行时检查它们是否存在。

字节排序方式

与 Windows 的 ARM32 版本一样，在 ARM64 Windows 上以 little-endian 模式执行。如果没有 AArch64 中的内核模式支持，则切换字节排序方式难以实现，因此更易于强制执行。

对齐方式

在 ARM64 上运行的 Windows 使 CPU 硬件可以透明地处理未对齐访问。在相对于 AArch32 的改进中，此支持现在也适用于所有整数访问（包括多字访问）和浮点访问。

但是，对非缓存（设备）内存的访问仍必须始终对齐。如果代码可能对未缓存内存读取或写入未对齐数据，则必须确保对齐所有访问。

局部变量的默认布局对齐：

大小（以字节为单位）	对齐（以字节为单位）
------------	------------

大小（以字节为单位）	对齐（以字节为单位）
1	1
2	2
3、 4	4
> 4	8

全局变量和静态变量的默认布局对齐：

大小（以字节为单位）	对齐（以字节为单位）
1	1
2 - 7	4
8 - 63	8
> = 64	16

整数寄存器

AArch64 体系结构支持 32 个整数寄存器：

寄存器	是否易失？	角色
x0	易失的	参数寄存器/临时寄存器 1、结果寄存器
x1-x7	易失的	参数寄存器/临时寄存器 2-8
x8-x15	易失的	临时寄存器
x16-x17	易失的	过程内部调用临时寄存器
x18	非易失性的	平台寄存器：在内核模式下，指向当前处理器的 KPCR；在用户模式下，指向 TEB

寄存器	是否易失?	角色
x19- x28	非易失性的	临时寄存器
x29/fp	非易失性的	帧指针
x30/lr	非易失性的	链接寄存器

每个寄存器可以作为完整 64 位值（通过 x0-x30）或作为 32 位值（通过 w0-w30）进行访问。32 位操作将结果零扩展到最多 64 位。

有关参数寄存器使用的详细信息，请参阅“参数传递”一节。

与 AArch32 不同，程序计数器 (PC) 和堆栈指针 (SP) 不是索引寄存器。它们的访问方式受到限制。另请注意，没有 x31 寄存器。该编码用于特殊用途。

与 ETW 和其他服务使用的快速堆栈浏览兼容需要帧指针 (x29)。它必须指向堆栈上的前一个 {x29, x30} 对。

浮点/SIMD 寄存器

AArch64 体系结构还支持 32 个浮点/SIMD 寄存器，下面进行了总结：

寄存器	是否易失?	角色
v0	易失的	参数寄存器/临时寄存器 1、结果寄存器
v1-v7	易失的	参数寄存器/临时寄存器 2-8
v8-v15	非易失性的	临时寄存器（仅低 64 位为非易失性）
v16-v31	易失的	临时寄存器

每个寄存器可以作为完整 128 位值（通过 v0-v31 或 q0-q31）进行访问。它可以作为 64 位值（通过 d0-d31）、作为 32 位值（通过 s0-s31）、作为 16 位值（通过 h0-h31）或作为 8 位值（通过 b0-b31）进行访问。小于 128 位的访问仅访问完整 128 位寄存器的较低位。除非另外指定，否则它们使其余位保持不变。（AArch64 与 AArch32 不同，其中较小寄存器在较大寄存器顶部打包。）

浮点控制寄存器 (FPCR) 对其中的各个位域具有特定要求：

Bits	含义	是否易失?	角色
26	AHP	非易失性	备选半精度控制。
25	DN	非易失性	默认 NaN 模式控制。
24	FZ	非易失性的	清零模式控制。
23-22	RMode	非易失性的	舍入模式控制。
15、12-8	IDE/IXE/等	非易失性	异常捕获启用位，必须始终为 0。

系统寄存器

与 AArch32 一样，AArch64 规范提供三个系统控制的“线程 ID”寄存器：

寄存器	角色
TPIDR_ELO	保留。
TPIDRRO_ELO	包含当前处理器的 CPU 编号。
TPIDR_EL1	包含当前处理器的 KPCR 结构。

浮点异常

在 AArch64 系统上，对 IEEE 浮点异常的支持是可选的。对于具有硬件浮点异常的处理器变量，Windows 内核将以静默形式捕捉这些异常并在 FPCR 寄存器中隐式地禁用它们。此陷阱可确保在处理器变量之间保持规范化行为。否则，当在不支持异常的平台开发的代码在提供支持的平台运行时，它可能会发现自己遇到意外异常。

参数传递

对于不可变参数函数，Windows ABI 遵循 ARM 指定的参数传递规则。这些规则直接摘录自 AArch64 体系结构的过程调用标准：

阶段 A – 初始化

在开始处理参数之前，此阶段只执行一次。

1. 下一个通用寄存器号编号 (NGRN) 设置为零。
2. 下一个 SIMD 和浮点寄存器号编号 (NSRN) 设置为零。
3. 下一个堆叠参数地址 (NSAA) 设置为当前堆栈指针值 (SP)。

阶段 B - 参数的预填充和扩展

对于列表中的每个参数，将从以下列表中应用第一个匹配规则。如果没有规则匹配，则使用未修改的参数。

1. 如果参数类型是其大小无法通过调用方和被调用方静态确定的复合类型，则参数会复制到内存，并且参数由一个指向副本的指针替换。（C/C++ 中没有此类类型，但它们存在于其他语言或语言扩展中）。
2. 如果参数类型为 HFA 或 HVA，则使用未修改的参数。
3. 如果参数类型是大于 16 字节的复合类型，则参数会复制到由调用方分配的内存，并且参数由一个指向该副本的指针替换。
4. 如果参数类型为复合类型，则参数大小将会舍入为 8 字节的最接近倍数。

阶段 C - 将参数分配给寄存器和堆栈

对于列表中的每个参数，将依次应用以下规则，直到参数被分配。将参数分配给寄存器时，寄存器中任何未使用的位都具有未指定的值。如果将参数分配给堆栈槽，则任何未填充字节都具有未指定的值。

1. 如果参数为半精度、单精度、双精度或四精度浮点数或是短矢量类型，并且 NSRN 小于 8，则会将参数分配给寄存器 $v[NSRN]$ 的最小有效位。NSRN 会按 1 递增。参数现在已分配。
2. 如果参数是 HFA 或 HVA，且有足够的未分配 SIMD 和浮点寄存器 ($NSRN + \text{成员数} \leq 8$)，则会将参数分配给 SIMD 和浮点寄存器，HFA 或 HVA 的每个成员一个寄存器。NSRN 按所使用的寄存器的数量递增。参数现在已分配。
3. 如果参数是 HFA 或 HVA，则 NSRN 设置为 8，并且参数的大小会向上舍入为 8 字节的最接近倍数。

4. 如果参数是 HFA、HVA、四精度浮点或短矢量类型，则 NSAA 会向上舍入为 8 或参数类型的自然对齐中的较大者。
5. 如果参数为半精度或单精度浮点类型，则参数的大小会设置为 8 字节。效果如同参数已复制到 64 位寄存器的最低有效位，并且剩余位使用未指定的值填充。
6. 如果参数是 HFA、HVA、半精度、单精度、双精度或四精度浮点或短矢量类型，则参数会复制到已调整 NSAA 处的内存。NSAA 按参数的大小递增。参数现在已分配。
7. 如果参数为整型或指针类型，参数的大小小于或等于 8 字节，并且 NGRN 小于 8，则参数会复制到 $x[NGRN]$ 中的最低有效位。NGRN 会按 1 递增。参数现在已分配。
8. 如果参数的对齐为 16，则 NGRN 会向上舍入为下一个偶数。
9. 如果参数为整型类型，参数的大小等于 16，并且 NGRN 小于 7，则参数会复制到 $x[NGRN]$ 和 $x[NGRN+1]$ 。 $x[NGRN]$ 应包含参数内存表示形式的较低地址双字。NGRN 会按 2 递增。参数现在已分配。
10. 如果参数是复合类型，并且参数双字中的大小不大于 8 减 NGRN，则参数会复制到连续通用寄存器中，从 $x[NGRN]$ 开始。参数的传递如同从双字对齐地址加载到寄存器中一样（通过从内存加载连续寄存器的适当 LDR 指令序列）。本标准不指定寄存器的任何未使用部分的内容。NGRN 按所使用的寄存器的数量递增。参数现在已分配。
11. NGRN 设置为 8。
12. NSAA 向上舍入为 8 或参数类型的自然对齐中的较大者。
13. 如果参数是复合类型，则参数会复制到已调整 NSAA 处的内存。NSAA 按参数的大小递增。参数现在已分配。
14. 如果参数的大小小于 8 字节，则参数的大小会设置为 8 字节。效果如同参数已复制到 64 位寄存器的最低有效位，并且剩余位使用未指定的值填充。
15. 参数会复制到已调整 NSAA 处的内存。NSAA 按参数的大小递增。参数现在已分配。

附录：可变参数函数

采用可变数量参数的函数的处理方式与以上内容不同，如下所示：

1. 所有复合类型都进行相似处理；对 HFA 或 HVA 不进行特殊处理。
2. 不使用 SIMD 和浮点寄存器。

实际上，它与将参数分配给虚堆栈的以下规则 C.12–C.15 相同，其中堆栈的前 64 字节会加载到 x0-x7 中，所有剩余堆栈参数会正常放置。

返回值

整数值在 x0 中返回。

浮点值以适当方式在 s0、d0 或 v0 中返回。

如果满足下列所有项，则类型会视为 HFA 或 HVA：

- 它不为空，
- 它没有任何重要的默认或复制构造函数，或赋值运算符，
- 它的所有成员具有相同的 HFA 或 HVA 类型，或者是与其他成员的 HFA 或 HVA 类型匹配的 float、float 或 neon 类型。

具有 4 个或更少元素的 HFA 和 HVA 值以适当方式在 s0-s3、d0-d3 或 v0-v3 中返回。

通过值返回的类型的处理方式因它们是否具有某些属性而异，且因函数是否为非静态成员函数而异。具有所有这些属性的类型，

- 按照 C++14 标准定义，它们是聚合，即它们不包含用户提供的构造函数、专用或受保护的静态数据成员、基类，也不包含虚拟函数，并且
- 它们具有普通复制赋值运算符，并且
- 它们具有普通析构函数，

由非成员函数或静态成员函数返回，使用以下返回样式：

- 小于或等于 8 字节的类型在 x0 中返回。
- 小于或等于 16 字节的类型在 x0 和 x1 中返回，其中 x0 包含低序 8 字节。
- 对于大于 16 字节的类型，调用方应保留具有足够大小和对齐的内存块来保存结果。内存块的地址应在 x8 中作为附加参数传递给函数。调用方可能会在子例程执行过程中的任何时间点修改结果内存块。被调用方不需要保留存储在 x8 中的值。

所有其他类型都使用以下约定：

- 调用方应保留具有足够大小和对齐的内存块来保存结果。内存块的地址应在 x0 或 x1（如果在 x0 中传递 \$this）中作为附加参数传递给函数。调用方可能会在子例程执行过程中的任何时间点修改结果内存块。被调用方在 x0 中返回内存块的地址。

堆栈

按照 ARM 发布的 ABI，堆栈必须始终保持 16 字节对齐。AArch64 包含一种硬件功能，该功能可在每当 SP 不是 16 字节对齐且已进行了 SP 相关加载或存储时生成堆栈对齐错误。Windows 运行时始终启用此功能。

在堆栈上分配 4k 或更大容量的函数必须确保在最后一页之前每页都需要按顺序进行处理。此操作会确保任何代码都不能“跳过”Windows 用于扩展堆栈的保护页。通常，接触由 `__chkstk` 帮助程序进行，它具有自定义调用约定，会在 x15 中传递总堆栈分配除以 16 的值。

红色区域

保留紧跟在当前堆栈指针下方的 16 字节区域以供分析和动态修补方案使用。此区域允许小心地插入生成的代码，它在 `[sp, #-16]` 中存储了 2 个寄存器并临时将其用于任意目的。Windows 内核保证在用户模式和内核模式下出现异常或中断时不覆盖这 16 个字节。

内核堆栈

Windows 中的默认内核模式堆栈为 6 页 (24k)。请额外注意内核模式下具有大堆栈缓冲区的函数。不合时宜的中断会随着非常小的空余空间一起出现，并且会形成堆栈应急 bug 检查。

堆栈审核

Windows 中的代码会在启用了帧指针的情况 (`/Oy-`) 中进行编译，以实现快速堆栈审核。通常，x29 (fp) 指向链中的下一个链接，它是指向指向堆栈上前一个帧的指针和返回地址的 {fp, lr} 对。建议第三方代码也启用帧指针以改进分析和跟踪。

异常展开

在异常处理期间，通过使用展开代码来帮助展开。展开代码是存储在可执行文件的 .xdata 节中的字节序列。它们以抽象的方式描述了序言和尾声的操作，以便可以撤消函数序言的效果，从而准备备份调用方的堆栈帧。有关展开代码的详细信息，请参阅 [ARM64 异常处理](#)。

ARM EABI 还指定了使用展开代码的异常展开模式。但是，所提供的规范不足以用于在 Windows 中进行展开，此时必须处理电脑处于函数序言或尾声中间的情况。

动态生成的代码应通过 `RtlAddFunctionTable` 和关联函数，使用动态函数表进行描述，以便生成的代码可以参与异常处理。

循环计数器

所有 ARMv8 CPU 都需要支持循环计数器寄存器，这是 64 位寄存器，Windows 会将它配置为在任何异常级别都可读（包括用户模式）。可以通过特殊 `PMCCNTR_ELO` 寄存器，使用程序集代码中的 MSR 操作码或 C/C++ 代码中的 `_ReadStatusReg` 内部函数来访问它。



此处的循环计数器为 true 循环计数器，而不是时钟。计数频率因处理器频率而异。如果认为必须知道循环计数器的频率，则不应使用循环计数器。而是要度量时钟时间，对此应使用 `QueryPerformanceCounter`。

请参阅

[Visual C++ ARM 迁移的常见问题](#)

[ARM64 异常处理](#)

此页面有帮助吗？

 是  否