System V ABI

The **System V Application Binary Interface** is a set of specifications that detail calling conventions, object file formats, executable file formats, dynamic linking semantics, and much more for systems that complies with the *X/Open Common Application Environment Specification* and the *System V Interface Definition*. It is today the standard ABI used by the major Unix operating systems such as Linux, the BSD systems, and many others. The Executable and Linkable Format (ELF) is part of the System V ABI.

The ABI is organized as a portable base document and platform-specific supplements that fill in the blank gaps. Unofficial new architecture processor supplements have been published as the format has been adapted to new platforms such as X86-64. The standard is extensible and the format continues to evolve as Unix vendors add new features. Due to the many unofficial supplement specifications and the chaotic history of the Unix operating systems, the current situation is that the System V ABI has become a family of unofficial draft specifications with no real central governing body.

Many of the advanced feature such as dynamic linking are optional and loading a simple statically linked ELF program is straightforward. Earlier versions of the standard were more ambitious and attempted to standardize software package installation formats and X11 details, while these obsolete details are disregarded today. The ABI is well-understood by common operating system development tools like Binutils and GCC. Toolchains such as `i686-elf-gcc` generate code and executable files according to this ABI.

Contents [hide]

Executable and Linkable Format

Main article: Executable and Linkable Format

The Executable and Linkable Format is standardized as an adaptable file format in the System V ABI. Each processor supplement subtly changes the file format by declaring the size of abstract types used in the ELF format structures as well as the endianness. This allows the skeleton file format to be adapted to multiple processor architectures, where the difference between 32-bit and 64-bit systems are handled by simply increasing the size of various header fields. The format is powerful enough to contain auxiliary information such as debugging information, relocations for dynamic libraries and other vendor-specific miscellaneous information. This allows using the same format for both object files and linked executables.

Calling Convention

This is a short overview of the important calling convention details for the major System V ABI architectures. This is an incomplete account and you should consult the relevant processor supplement document for the details. Additionally, you can use the `-S` compiler option to stop the compilation process before the assembler is invoked, which lets you study how the compiler translates code to assembly following the relevant calling convention.

i386

This is a 32-bit platform. The stack grows downwards. Parameters to functions are passed on the stack in reverse order such that the first parameter is the last value pushed to the stack, which will then be the lowest value on the stack. Parameters passed on the stack may be modified by the called function. Functions are called using the `call` instruction that pushes the address of the next instruction to the stack and jumps to the operand. Functions return to the caller using the `ret` instruction that pops a value from the stack and jump to it. The stack is 4-byte aligned all the time, on older systems and those honouring the SYSV psABI. On some newer systems, the stack is additionally 16-byte aligned just before the call instruction is called (usually those that want to support SSE instructions); consult your manual (GNU/Linux on i386 has recently become such a system, but code mixing with 4-byte stack alignment-assuming code is possible).

Functions preserve the registers `ebx`, `esi`, `edi`, `ebp`, and `esp`; while `eax`, `ecx`, `edx` are scratch registers. The return value is stored in the `eax` register, or if it is a 64-bit value, then the higher 32-bits go in `edx`. Functions push `ebp` such that the `caller-return-eip` is 4 bytes above it, and set `ebp` to the address of the saved ebp. This allows iterating through the existing stack frames. This can be eliminated by specifying the `-fomit-frame-pointer` GCC option.

As a special exception, GCC assumes the stack is not properly aligned and realigns it when entering `main` or if the `attribute ((force_align_arg_pointer))` is set on the function.

x86-64

This is a 64-bit platform. The stack grows downwards. Parameters to functions are passed in the registers `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, and further values are passed on the stack in reverse order. Parameters passed on the stack may be modified by the called function. Functions are called using the `call` instruction that pushes the address of the next instruction to the stack and jumps to the operand. Functions return to the caller using the `ret` instruction that pops a value from the stack and jump to it. The stack is 16-byte aligned just before the call instruction is called.

Functions preserve the registers `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, and `r15`; while `rax`, `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`, `r10`, `r11` are scratch registers. The return value is stored in the `rax` register, or if it is a 128-bit value, then the higher 64-bits go in `rdx`. Optionally, functions push `rbp` such that the `caller-return-rip` is 8 bytes above it, and set `rbp` to the address of the saved rbp. This allows iterating through the existing stack frames. This can be eliminated by specifying the `-fomit-frame-pointer` GCC option.

Signal handlers are executed on the same stack, but 128 bytes known as the red zone is subtracted from the stack before anything is pushed to the stack. This allows small leaf functions to use 128 bytes of stack space without reserving stack space by subtracting from the stack pointer. The red zone is well-known to cause problems for x86-64 kernel developers, as the CPU itself doesn't respect the red zone when calling interrupt handlers. This leads to a subtle kernel breakage as the ABI contradicts the CPU behavior. The solution is to build all kernel code with `-mno-red-zone` or by handling interrupts in kernel mode on another stack than the current (and thus implementing the ABI).

See Also

Documents

TODO: *Ensure whether these are the latest official links. These documents are simply what I could find through a quick online search.*

- base document and addons
  - System V ABI - Latest Base Document
  - System V ABI - Older Base Document
  - ELF Handling For Thread-Local Storage
- x86 (i8086, i386, amd64, x32, k1om)
  - System V ABI - Intel386 Architecture Processor Supplement
  - System V ABI - AMD64 Architecture Processor Supplement
  - System V ABI - K1OM Architecture Processor Supplement
- MIPS
  - System V ABI - MIPS RISC Processor Supplement
  - System V ABI - MIPSpro™ 64-Bit
  - System V ABI - MIPSpro™ N32 ABI Handbook
- PowerPC
  - System V ABI - PowerPC Processor Supplement
  - System V ABI - 64-bit PowerPC
- SPARC
  - System V ABI - SPARC® Processor Supplement
  - System V ABI - SPARC® Version 9 Processor Supplement
- Itanium
  - System V ABI - IA-64 Architecture Processor Supplement
- ⚠ not SYSV ABI, but at least the architecture-official calling standards:
  - DEC Alpha calling standard

External Links

- System V ABI Website
- Linux Foundation—Referenced Specifications
- Introduction to Computer Architecture—Technical Documents

Categories: ABI | Executable Formats | Object Files | Standards