Aphelion Application Binary Interface Specification Version 6

Seth Poulsen
sp@sandwichman.dev

Typeface **Io Serif** created by Echo Heo for Orbit Systems documentation.

Typeface **IBM Plex Mono** used in monospace/code-snippet contexts.

Contents

1. Introduction

This document specifies the elements of the standard Application Binary Interface (ABI) for the Aphelion ISA. Standard tools, system implementations, and software shall implement the ABI to build mutually-compatible Aphelion software and ensure a cohesive, reliable software ecosystem.

2. Register Convention

Aphelion has 32 general purpose registers (GPRs), each 64 bits wide.

#	Name	Preserved across calls?	Purpose
0	**`zr`**		Hardwired to zero
1..6	**`a0..a5`**	No	Function arguments/returns
7..20	**`l0..l13`**	Yes	Local variables
21..26	**`t0..t5`**	No	Temporary variables
27	**`tp`**		Thread pointer
28	**`fp`**	Yes	Frame pointer
29	**`sp`**	Yes	Stack pointer
30	**`lp`**	No	Link/return pointer
31	**`ip`**		Instruction pointer

Standard ABI-conforming software shall not modify the thread pointer **`tp`**.

3. Stack Convention

At procedure entrance and exit, the stack pointer **`sp`** shall be 8-byte aligned.

The call stack grows downward, meaning that:

- A procedure shall not use memory at addresses greater than or equal to the value of **`sp`** *at procedure entrance* for local value storage, with the exception of stack-allocated arguments.
- At any execution point in the procedure, it shall not access memory at addresses less than the value of **`sp`** *at that point*.

In short, procedures may only use memory they have allocated (by decrementing **`sp`**) for local storage, with the exception of stack-allocated arguments.

Use of the frame pointer **`fp`** is optional and may be used by procedures as a standard call-preserved register.

4. C/C++ Type Convention

Scalar types in C/C++ shall have size and alignment described in the following table:

Type	Size	Alignment
`bool`/**`_Bool`**	1	1
`char`	1	1
`short`	2	2
`int`	4	4
`long`	8	8
`long long`	8	8
`__int128`	16	16
`void*`	8	8
`_Float16`	2	2
`float`	4	4
`double`	8	8
`long double`	16	16
`_Complex float`	8	4
`_Complex double`	16	8
`_Complex long double`	32	16

Type **`char`** shall be unsigned.

Values of type **`bool`** shall always be either 0 (false) or 1 (true).

Values of type **`_Float16`** and **`long double`** shall be represented using the **`binary16`** and **`binary128`** formats respectively, defined in IEEE 754-2019.

Complex types shall have the same representation as the C structure below, where **`T`** is the real type component:

```
struct Complex {
    T real;
    T imaginary;
};
```

5. Calling Convention

The standard calling convention views procedure signatures as a list of typed parameters and a list of return values (called **returns** in this document).

5.1. Value Conversion

All values in the procedure signature undergo **value conversion**, which turns the procedure signature's richly-typed parameters and returns into lists of 64-bit words to pass and return.

For the purposes of this conversion, `bool` shall be treated as a 1-bit unsigned integer and `T*` shall be treated as a 64-bit unsigned integer.

Integer arguments and returns with size less than 64 bits shall be sign-extended (for signed integers) or zero-extended (for unsigned integers) to 64 bits.

Integer arguments and returns with size greater than 64 bits shall be treated as an aggregate of the same size.

Floating-point arguments and returns of any size shall be treated as an aggregate of the same size.

Aggregate arguments and returns with size less than or equal to 8 bytes shall be passed as though they are an integer argument, with identical byte layout. Padding and unused bits are undefined.

Aggregate arguments and returns with size greater than 8 bytes and less than or equal to 16 bytes shall be treated as two consecutive integer arguments, where the lower 8 bytes are passed first. Internal padding bits and bits not taken up by the aggregate value are undefined.

Aggregate arguments larger than 16 bytes shall be replaced in the argument list by a pointer. The storage backing the pointer shall not be mutated except through that pointer (equivalent to `restrict` qualification in C). Note that the aggregate may be mutated through this pointer, so it is recommended that a copy of this value is made on the caller side, though this copy may be optimized out if it would make no possible observable effect (e.g. the aggregate has no later uses or live references).

Aggregate returns larger than 16 bytes shall be removed from the return list and a pointer to write the aggregate return value shall be inserted into the beginning of the argument list. The storage backing the pointer shall not be mutated except through that pointer (equivalent to `restrict` qualification in C).

5.2. Value Passing

After value conversion, the procedure signature is a list of argument words and a list of return words.

Note that in this section, **`sp`** refers to the value of the stack pointer *at procedure entrance*.

At procedure exit, the first 6 return words are placed in order in registers **`a0`** to **`a5`**. If there are more than 6 return words, the rest shall be placed at consecutive positive offsets from **`sp`**, starting from 0.

At procedure entrance, the first 6 argument words are placed in order in registers **`a0`** to **`a5`**. If there are more than 6 argument words, the rest shall be placed *after* any stack-placed return words.

For example, if there are 8 argument words and 8 return words, the 7th return word would be placed at **`sp + 0`**, the 8th return word would be placed at **`sp + 8`**, the 7th argument word would be placed at **`sp + 16`**, and the 8th argument word would be placed at **`sp + 24`**.

All variadic argument words are placed on the stack as if there are no available registers, even if there are.

In the following table:
- n_{arg} represents the number of argument words placed on the stack;
- n_{ret} represents the number of return words placed on the stack;

Position	Contents
`sp + 8*(n_ret + n_arg - 1)`	Stack Argument Word $n_{\text{arg}} - 1$
…	…
`sp + 8*(n_ret + 1)`	Stack Argument Word 1
`sp + 8*(n_ret)`	Stack Argument Word 0
`sp + 8*(n_ret - 1)`	Stack Return Word $n_{\text{ret}} - 1$
…	…
`sp + 8`	Stack Return Word 1
`sp + 0`	Stack Return Word 0

Very few applications and languages take advantage of the multiple-return capabilities of the calling convention, so n_{ret} will almost always be 0.

6. Linker Relocations

Aphelion linkers, object files, and assemblers must support different types of **linker relocations** to be able to manipulate and relocate assembled code effectively.

Relocations must support signed addends of at least 16 bits.

In this section:
- **S** refers to the value of the symbol associated with the relocation.
- **A** refers to the addend associated with the relocation.
- **P** refers to the address of the relocation itself.

6.1. WORD

This relocation places the expression **`S + A`** into an aligned 64-bit word at **`P`**.

This relocation must be aligned to 8 bytes.

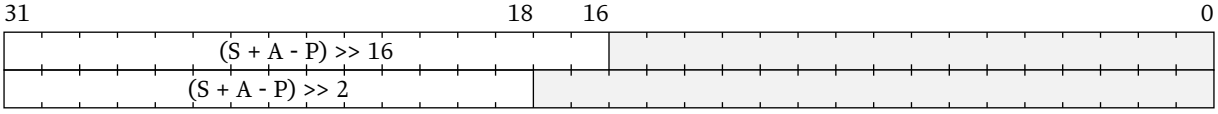
6.2. WORD_UNALIGNED

This relocation places the expression **`S + A`** into an unaligned 64-bit word at **`P`**.

6.3. CALL

This relocation corresponds to a **`call`** assembly sequence. This places:
- **`(S + A - P) >> 16`** into bits 16..31 of the instruction at **`P`**;
- **`(S + A - P) >> 2`** into bits 18..31 of the instruction at **`P + 4`**;

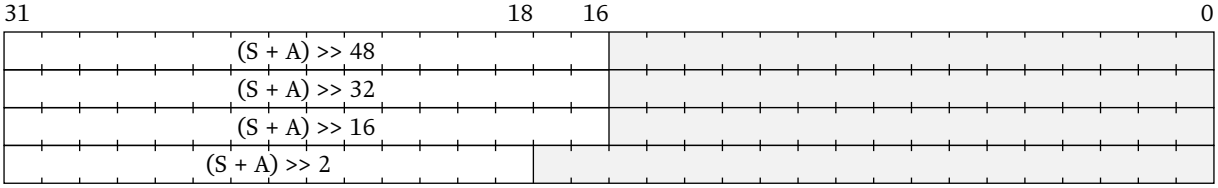
This relocation must be aligned to 4 bytes.



6.4. FCALL

This relocation corresponds to an **`fcall`** assembly sequence. This places:
- **`(S + A) >> 48`** into bits 16..31 of the instruction at **`P`**;
- **`(S + A) >> 32`** into bits 16..31 of the instruction at **`P + 4`**;
- **`(S + A) >> 16`** into bits 16..31 of the instruction at **`P + 8`**;
- **`(S + A) >> 2`** into bits 18..31 of the instruction at **`P + 12`**;

This relocation must be aligned to 4 bytes.

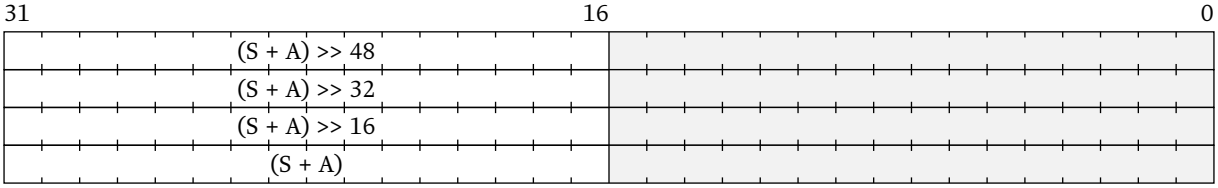


6.5. LI

This relocation corresponds to an **`li`** assembly sequence. This places:

- **`(S + A) >> 48`** into bits 16..31 of the instruction at **`P`**;
- **`(S + A) >> 32`** into bits 16..31 of the instruction at **`P + 4`**;
- **`(S + A) >> 16`** into bits 16..31 of the instruction at **`P + 8`**;
- **`(S + A)`** into bits 16..31 of the instruction at **`P + 12`**;

This relocation must be aligned to 4 bytes.



7. Glossary

- **word**: The size of a register. 8 bytes, 64 bits.
- **half-word**: 4 bytes, 32 bits.
- **quarter-word**: 2 bytes, 16 bits.
- **(linker) relocation**: An object file entry that inserts/replaces a value in a section such that symbol values that change during linking are accurately updated where they are referenced.