- Not a number (NaN) is represented by setting the exponent to the highest positive value and the mantissa to a non-zero value. The value of the sign bit is ignored.
- Subnormal numbers are used for representing values smaller than what can be represented by normal values. The drawback is that the precision will decrease with smaller values. The exponent is set to 0 to signify that the number is subnormal, even though the number is treated as if the exponent was 1. Unlike normal numbers, subnormal numbers do not have an implicit 1 as the most significant bit (the MSB) of the mantissa. The value of a subnormal number is:

  $(-1)^S * 2^{(1-BIAS)} * 0.Mantissa$

  where BIAS is 127 and 1023 for 32-bit and 64-bit floating-point values, respectively.

Note: The IAR CLIB Library does not fully support the special cases of floating-point numbers, such as infinity, NaN, and subnormal numbers. A library function which gets one of these special cases of floating-point numbers as an argument might behave unexpectedly.

Pointer types

The compiler has two basic types of pointers: function pointers and data pointers.

FUNCTION POINTERS

The size of function pointers is always 16 or 24 bits, and they can address the entire memory. The internal representation of a function pointer is the actual address it refers to divided by two.

These function pointers are available:

Keyword	Address range	Pointer size	Description
__nearfunc	0-0x1FFFE	16 bits	Can be called from any part of the code memory, but must reside in the first 128 Kbytes of that space.
__farfunc	0-0x7FFFFE	24 bits	No restrictions on code placement.

Table 32: Function pointers

DATA POINTERS

Data pointers have three sizes: 8, 16, or 24 bits. These data pointers are available:

Keyword	Pointer size	Memory space	Index type	Address range
__tiny	8 bits	Data	signed char	0x-0xFF

Table 33: Data pointers

Keyword	Pointer size	Memory space	Index type	Address range
`__near`	16 bits	Data	signed int	`0x0-0xFFFF`
`__far`	24 bits	Data	signed int	`0x0-0xFFFFFF` (16-bit arithmetics)
`__huge`	24 bits	Data	signed long	`0x0-0xFFFFFF`
`__tinyflash`	8 bits	Code	signed char	`0x0-0xFF`
`__flash`	16 bits	Code	signed int	`0x0-0xFFFF`
`__farflash`	24 bits	Code	signed int	`0x0-0xFFFFFF` (16-bit arithmetic)
`__hugeflash`	24 bits	Code	signed long	`0x0-0xFFFFFF`
`__eeprom`	8 bits	EEPROM	signed char	`0x0-0xFF`
`__eeprom`	16 bits	EEPROM	signed int	`0x0-0xFFFF`
`__generic`	16 bits 24 bits	Data/Code	signed int signed long	The most significant bit (MSB) determines whether `__generic` points to CODE (1) or DATA (0). The small generic pointer is generated for the processor options `-v0` and `-v1`.

Table 33: Data pointers (Continued)

Segmented data pointer comparison

Note that the result of using relational operators (<, <=, >, >=) on data pointers is only defined if the pointers point into the same object. For segmented data pointers, only the offset part of the pointer is compared when using these operators. For example:

```
void MyFunc(char __far * p, char __far * q)
{
  if (p == q)  /* Compares the entire pointers. */
    ...
  if (p < q) /* Compares only the low 16 bits of the pointers. */
    ...
}
```

CASTING

Casts between pointers have these characteristics:

- Casting a *value* of an integer type to a pointer of a smaller type is performed by truncation
- Casting a value of an integer type to a pointer of a larger type is performed by zero extension
- Casting a pointer type to a smaller integer type is performed by truncation
- Casting a *pointer type* to a larger integer type is performed via casting to the largest possible pointer that fits in the integer
- Casting a *data pointer* to a function pointer and vice versa is illegal
- Casting a *function pointer* to an integer type gives an undefined result
- Casting from a smaller pointer to a larger pointer is performed by zero extension
- Casting from a larger pointer to a smaller pointer gives an undefined result.

size_t

`size_t` is the unsigned integer type of the result of the `sizeof` operator. In the IAR C/C++ Compiler for AVR, the type used for `size_t` depends on the processor option used:

Generic processor option	Typedef
`-v0` and `-v1`	`unsigned int`
`-v2`, `-v3`, `-v4`, `-v5`, and `-v6`	`unsigned long`

Table 34: size_t typedef

When using the Large or Huge memory model, the typedef for `size_t` is `unsigned long int`.

Note that some data memory types might be able to accommodate larger, or only smaller, objects than the memory pointed to by default pointers. In this case, the type of the result of the `sizeof` operator could be a larger or smaller unsigned integer type. There exists a corresponding `size_t` typedef for each memory type, named after the memory type. In other words, `__near_size_t` for `__near` memory.

ptrdiff_t

`ptrdiff_t` is the signed integer type of the result of subtracting two pointers. This table shows the `typedef` of `ptrdiff_t` depending on the processor option:

Generic processor option	Typedef
`-v0` and `-v1`	`unsigned int`
`-v2`, `-v3`, `-v4`, `-v5`, and `-v6`	`unsigned long`

Table 35: ptrdif_t typedef

Note that subtracting pointers other than default pointers could result in a smaller or larger integer type. In each case, this integer type is the signed integer variant of the corresponding `size_t` type.

Note: It is sometimes possible to create an object that is so large that the result of subtracting two pointers in that object is negative. See this example:

```
char buff[60000];           /* Assuming ptrdiff_t is a 16-bit */
char *p1 = buff;            /* signed integer type. */
char *p2 = buff + 60000;
ptrdiff_t diff = p2 - p1;   /* Result: -5536 */
```

intptr_t

`intptr_t` is a signed integer type large enough to contain a `void *`. In the IAR C/C++ Compiler for AVR, the type used for `intptr_t` is `long int` when using the Large or Huge memory model and `int` in all other cases.

uintptr_t

`uintptr_t` is equivalent to `intptr_t`, with the exception that it is unsigned.

Structure types

The members of a `struct` are stored sequentially in the order in which they are declared: the first member has the lowest memory address.

ALIGNMENT OF STRUCTURE TYPES

The `struct` and `union` types inherit the alignment requirements of their members. In addition, the size of a `struct` is adjusted to allow arrays of aligned structure objects.

GENERAL LAYOUT

Members of a `struct` are always allocated in the order specified in the declaration. Each member is placed in the `struct` according to the specified alignment (offsets).