Next: Attribute Syntax, Previous: Mixed Declarations, Up: C Extensions

---

5.27 Declaring Attributes of Functions

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls and check your code more carefully.

The keyword `__attribute__` allows you to specify special attributes when making a declaration. This keyword is followed by an attribute specification inside double parentheses. The following attributes are currently defined for functions on all targets: `aligned`, `alloc_size`, `noreturn`, `returns_twice`, `noinline`, `always_inline`, `flatten`, `pure`, `const`, `nothrow`, `sentinel`, `format`, `format_arg`, `no_instrument_function`, `section`, `constructor`, `destructor`, `used`, `unused`, `deprecated`, `weak`, `malloc`, `alias`, `warn_unused_result`, `nonnull`, `gnu_inline`, `externally_visible`, `hot`, `cold`, `artificial`, `error` and `warning`. Several other attributes are defined for functions on particular target systems. Other attributes, including `section` are supported for variables declarations (see Variable Attributes) and for types (see Type Attributes).

You may also specify attributes with `__' preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use `__noreturn__` instead of `noreturn`.

See Attribute Syntax, for details of the exact syntax for using attributes.

`alias ("`*target*`")`
: The `alias` attribute causes the declaration to be emitted as an alias for another symbol, which must be specified. For instance,

```
void __f () { /* Do something. */; }
void f () __attribute__ ((weak, alias ("__f")));
```

defines `f' to be a weak alias for `__f'. In C++, the mangled name for the target must be used. It is an error if `__f' is not defined in the same translation unit.

Not all target machines support this attribute.

`aligned (`*alignment*`)`
: This attribute specifies a minimum alignment for the function, measured in bytes.

You cannot use this attribute to decrease the alignment of a function, only to increase it. However, when you explicitly specify a function alignment this will override the effect of the `-falign-functions` (see Optimize Options) option for this function.

Note that the effectiveness of `aligned` attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for functions to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) See your linker documentation for further information.

The `aligned` attribute can also be used for variables and fields (see Variable Attributes.)

`alloc_size`
: The `alloc_size` attribute is used to tell the compiler that the function return value points to memory,

where the size is given by one or two of the functions parameters. GCC uses this information to improve the correctness of `__builtin_object_size`.

The function parameter(s) denoting the allocated size are specified by one or two integer arguments supplied to the attribute. The allocated size is either the value of the single function argument specified or the product of the two function arguments specified. Argument numbering starts at one.

For instance,

```
void* my_calloc(size_t, size_t) __attribute__((alloc_size(1,2)))
void my_realloc(void*, size_t) __attribute__((alloc_size(2)))
```

declares that my_calloc will return memory of the size given by the product of parameter 1 and 2 and that my_realloc will return memory of the size given by parameter 2.

`always_inline`
Generally, functions are not inlined unless optimization is specified. For functions declared inline, this attribute inlines the function even if no optimization level was specified.

`gnu_inline`
This attribute should be used with a function which is also declared with the `inline` keyword. It directs GCC to treat the function as if it were defined in gnu89 mode even when compiling in C99 or gnu99 mode.

If the function is declared `extern`, then this definition of the function is used only for inlining. In no case is the function compiled as a standalone function, not even if you take its address explicitly. Such an address becomes an external reference, as if you had only declared the function, and had not defined it. This has almost the effect of a macro. The way to use this is to put a function definition in a header file with this attribute, and put another copy of the function, without `extern`, in a library file. The definition in the header file will cause most calls to the function to be inlined. If any uses of the function remain, they will refer to the single copy in the library. Note that the two definitions of the functions need not be precisely the same, although if they do not have the same effect your program may behave oddly.

In C, if the function is neither `extern` nor `static`, then the function is compiled as a standalone function, as well as being inlined where possible.

This is how GCC traditionally handled functions declared `inline`. Since ISO C99 specifies a different semantics for `inline`, this function attribute is provided as a transition measure and as a useful feature in its own right. This attribute is available in GCC 4.1.3 and later. It is available if either of the preprocessor macros `__GNUC_GNU_INLINE__` or `__GNUC_STDC_INLINE__` are defined. See An Inline Function is As Fast As a Macro.

In C++, this attribute does not depend on `extern` in any way, but it still requires the `inline` keyword to enable its special behavior.

`artificial`
This attribute is useful for small inline wrappers which if possible should appear during debugging as a unit, depending on the debug info format it will either mean marking the function as artificial or using the caller location for all instructions within the inlined body.

`flatten`
Generally, inlining into a function is limited. For a function marked with this attribute, every call inside this function will be inlined, if possible. Whether the function itself is considered for inlining depends on its size and the current inlining parameters.

`error ("`*message*`")`
: If this attribute is used on a function declaration and a call to such a function is not eliminated through dead code elimination or other optimizations, an error which will include *message* will be diagnosed. This is useful for compile time checking, especially together with `__builtin_constant_p` and inline functions where checking the inline function arguments is not possible through `extern char [(condition) ? 1 : -1];` tricks. While it is possible to leave the function undefined and thus invoke a link failure, when using this attribute the problem will be diagnosed earlier and with exact location of the call even in presence of inline functions or when not emitting debugging information.

`warning ("`*message*`")`
: If this attribute is used on a function declaration and a call to such a function is not eliminated through dead code elimination or other optimizations, a warning which will include *message* will be diagnosed. This is useful for compile time checking, especially together with `__builtin_constant_p` and inline functions. While it is possible to define the function with a message in `.gnu.warning*` section, when using this attribute the problem will be diagnosed earlier and with exact location of the call even in presence of inline functions or when not emitting debugging information.

`cdecl`
: On the Intel 386, the `cdecl` attribute causes the compiler to assume that the calling function will pop off the stack space used to pass arguments. This is useful to override the effects of the `-mrtd` switch.

`const`
: Many functions do not examine any values except their arguments, and have no effects except the return value. Basically this is just slightly more strict class than the `pure` attribute below, since function is not allowed to read global memory.

  Note that a function that has pointer arguments and examines the data pointed to must *not* be declared `const`. Likewise, a function that calls a non-`const` function usually must not be `const`. It does not make sense for a `const` function to return `void`.

  The attribute `const` is not implemented in GCC versions earlier than 2.5. An alternative way to declare that a function has no side effects, which works in the current version and in some older versions, is as follows:

```
typedef int intfn ();

extern const intfn square;
```

  This approach does not work in GNU C++ from 2.6.0 on, since the language specifies that the `const' must be attached to the return value.

`constructor`
`destructor`
`constructor (`*priority*`)`
`destructor (`*priority*`)`
: The `constructor` attribute causes the function to be called automatically before execution enters `main ()`. Similarly, the `destructor` attribute causes the function to be called automatically after `main ()` has completed or `exit ()` has been called. Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.

  You may provide an optional integer priority to control the order in which constructor and destructor functions are run. A constructor with a smaller priority number runs before a constructor with a larger priority number; the opposite relationship holds for destructors. So, if you have a constructor that allocates a resource and a destructor that deallocates the same resource, both functions typically have the same priority. The priorities for constructor and destructor functions are the same as those specified

for namespace-scope C++ objects (see C++ Attributes).

These attributes are not currently implemented for Objective-C.

`deprecated`

The `deprecated` attribute results in a warning if the function is used anywhere in the source file. This is useful when identifying functions that are expected to be removed in a future version of a program. The warning also includes the location of the declaration of the deprecated function, to enable users to easily find further information about why the function is deprecated, or what they should do instead. Note that the warnings only occurs for uses:

```
int old_fn () __attribute__ ((deprecated));
int old_fn ();
int (*fn_ptr)() = old_fn;
```

results in a warning on line 3 but not line 2.

The `deprecated` attribute can also be used for variables and types (see Variable Attributes, see Type Attributes.)

`dllexport`

On Microsoft Windows targets and Symbian OS targets the `dllexport` attribute causes the compiler to provide a global pointer to a pointer in a DLL, so that it can be referenced with the `dllimport` attribute. On Microsoft Windows targets, the pointer name is formed by combining `_imp__` and the function or variable name.

You can use `__declspec(dllexport)` as a synonym for `__attribute__ ((dllexport))` for compatibility with other compilers.

On systems that support the `visibility` attribute, this attribute also implies "default" visibility. It is an error to explicitly specify any other visibility.

Currently, the `dllexport` attribute is ignored for inlined functions, unless the `-fkeep-inline-functions` flag has been used. The attribute is also ignored for undefined symbols.

When applied to C++ classes, the attribute marks defined non-inlined member functions and static data members as exports. Static consts initialized in-class are not marked unless they are also defined out-of-class.

For Microsoft Windows targets there are alternative methods for including the symbol in the DLL's export table such as using a `.def` file with an `EXPORTS` section or, with GNU ld, using the `--export-all` linker flag.

`dllimport`

On Microsoft Windows and Symbian OS targets, the `dllimport` attribute causes the compiler to reference a function or variable via a global pointer to a pointer that is set up by the DLL exporting the symbol. The attribute implies `extern`. On Microsoft Windows targets, the pointer name is formed by combining `_imp__` and the function or variable name.

You can use `__declspec(dllimport)` as a synonym for `__attribute__ ((dllimport))` for compatibility with other compilers.

On systems that support the `visibility` attribute, this attribute also implies "default" visibility. It is an error to explicitly specify any other visibility.

Currently, the attribute is ignored for inlined functions. If the attribute is applied to a symbol *definition*, an error is reported. If a symbol previously declared `dllimport` is later defined, the attribute is ignored in subsequent references, and a warning is emitted. The attribute is also overridden by a subsequent declaration as `dllexport`.

When applied to C++ classes, the attribute marks non-inlined member functions and static data members as imports. However, the attribute is ignored for virtual methods to allow creation of vtables using thunks.

On the SH Symbian OS target the `dllimport` attribute also has another affect—it can cause the vtable and run-time type information for a class to be exported. This happens when the class has a dllimport'ed constructor or a non-inline, non-pure virtual function and, for either of those two conditions, the class also has a inline constructor or destructor and has a key function that is defined in the current translation unit.

For Microsoft Windows based targets the use of the `dllimport` attribute on functions is not necessary, but provides a small performance benefit by eliminating a thunk in the DLL. The use of the `dllimport` attribute on imported variables was required on older versions of the GNU linker, but can now be avoided by passing the `--enable-auto-import` switch to the GNU linker. As with functions, using the attribute for a variable eliminates a thunk in the DLL.

One drawback to using this attribute is that a pointer to a *variable* marked as `dllimport` cannot be used as a constant address. However, a pointer to a *function* with the `dllimport` attribute can be used as a constant initializer; in this case, the address of a stub function in the import lib is referenced. On Microsoft Windows targets, the attribute can be disabled for functions by setting the `-mnop-fun-dllimport` flag.

`eightbit_data`
Use this attribute on the H8/300, H8/300H, and H8S to indicate that the specified variable should be placed into the eight bit data section. The compiler will generate more efficient code for certain operations on data in the eight bit data area. Note the eight bit data area is limited to 256 bytes of data.

You must use GAS and GLD from GNU binutils version 2.7 or later for this attribute to work correctly.

`exception_handler`
Use this attribute on the Blackfin to indicate that the specified function is an exception handler. The compiler will generate function entry and exit sequences suitable for use in an exception handler when this attribute is present.

`externally_visible`
This attribute, attached to a global variable or function, nullifies the effect of the `-fwhole-program` command-line option, so the object remains visible outside the current compilation unit.

`far`
On 68HC11 and 68HC12 the `far` attribute causes the compiler to use a calling convention that takes care of switching memory banks when entering and leaving a function. This calling convention is also the default when using the `-mlong-calls` option.

On 68HC12 the compiler will use the `call` and `rtc` instructions to call and return from a function.

On 68HC11 the compiler will generate a sequence of instructions to invoke a board-specific routine to switch the memory bank and call the real function. The board-specific routine simulates a `call`. At the end of a function, it will jump to a board-specific routine instead of using `rts`. The board-specific return routine simulates the `rtc`.

`fastcall`

On the Intel 386, the `fastcall` attribute causes the compiler to pass the first argument (if of integral type) in the register ECX and the second argument (if of integral type) in the register EDX. Subsequent and other typed arguments are passed on the stack. The called function will pop the arguments off the stack. If the number of arguments is variable all arguments are pushed on the stack.

`format (`*archetype, string-index, first-to-check*`)`

The `format` attribute specifies that a function takes `printf`, `scanf`, `strftime` or `strfmon` style arguments which should be type-checked against a format string. For example, the declaration:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
      __attribute__ ((format (printf, 2, 3)));
```

causes the compiler to check the arguments in calls to `my_printf` for consistency with the `printf` style format string argument `my_format`.

The parameter *archetype* determines how the format string is interpreted, and should be `printf`, `scanf`, `strftime`, `gnu_printf`, `gnu_scanf`, `gnu_strftime` or `strfmon`. (You can also use `__printf__`, `__scanf__`, `__strftime__` or `__strfmon__`.) On MinGW targets, `ms_printf`, `ms_scanf`, and `ms_strftime` are also present. *archtype* values such as `printf` refer to the formats accepted by the system's C run-time library, while `gnu_` values always refer to the formats accepted by the GNU C Library. On Microsoft Windows targets, `ms_` values refer to the formats accepted by the `msvcrt.dll` library. The parameter *string-index* specifies which argument is the format string argument (starting from 1), while *first-to-check* is the number of the first argument to check against the format string. For functions where the arguments are not available to be checked (such as `vprintf`), specify the third parameter as zero. In this case the compiler only checks the format string for consistency. For `strftime` formats, the third parameter is required to be zero. Since non-static C++ methods have an implicit `this` argument, the arguments of such methods should be counted from two, not one, when giving values for *string-index* and *first-to-check*.

In the example above, the format string (`my_format`) is the second argument of the function `my_print`, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3.

The `format` attribute allows you to identify your own functions which take format strings as arguments, so that GCC can check the calls to these functions for errors. The compiler always (unless `-ffreestanding` or `-fno-builtin` is used) checks formats for the standard library functions `printf`, `fprintf`, `sprintf`, `scanf`, `fscanf`, `sscanf`, `strftime`, `vprintf`, `vfprintf` and `vsprintf` whenever such warnings are requested (using `-Wformat`), so there is no need to modify the header file `stdio.h`. In C99 mode, the functions `snprintf`, `vsnprintf`, `vscanf`, `vfscanf` and `vsscanf` are also checked. Except in strictly conforming C standard modes, the X/Open function `strfmon` is also checked as are `printf_unlocked` and `fprintf_unlocked`. See Options Controlling C Dialect.

The target may provide additional types of format checks. See Format Checks Specific to Particular Target Machines.

`format_arg (`*string-index*`)`

The `format_arg` attribute specifies that a function takes a format string for a `printf`, `scanf`, `strftime` or `strfmon` style function and modifies it (for example, to translate it into another language), so the result can be passed to a `printf`, `scanf`, `strftime` or `strfmon` style function (with the remaining arguments to the format function the same as they would have been for the unmodified string). For example, the declaration:

```
extern char *
```

```
my_dgettext (char *my_domain, const char *my_format)
      __attribute__ ((format_arg (2)));
```

causes the compiler to check the arguments in calls to a `printf`, `scanf`, `strftime` or `strfmon` type function, whose format string argument is a call to the `my_dgettext` function, for consistency with the format string argument `my_format`. If the `format_arg` attribute had not been specified, all the compiler could tell in such calls to format functions would be that the format string argument is not constant; this would generate a warning when `-Wformat-nonliteral` is used, but the calls could not be checked without the attribute.

The parameter *string-index* specifies which argument is the format string argument (starting from one). Since non-static C++ methods have an implicit `this` argument, the arguments of such methods should be counted from two.

The `format-arg` attribute allows you to identify your own functions which modify format strings, so that GCC can check the calls to `printf`, `scanf`, `strftime` or `strfmon` type function whose operands are a call to one of your own function. The compiler always treats `gettext`, `dgettext`, and `dcgettext` in this manner except when strict ISO C support is requested by `-ansi` or an appropriate `-std` option, or `-ffreestanding` or `-fno-builtin` is used. See Options Controlling C Dialect.

`function_vector`

Use this attribute on the H8/300, H8/300H, and H8S to indicate that the specified function should be called through the function vector. Calling a function through the function vector will reduce code size, however; the function vector has a limited size (maximum 128 entries on the H8/300 and 64 entries on the H8/300H and H8S) and shares space with the interrupt vector.

In SH2A target, this attribute declares a function to be called using the TBR relative addressing mode. The argument to this attribute is the entry number of the same function in a vector table containing all the TBR relative addressable functions. For the successful jump, register TBR should contain the start address of this TBR relative vector table. In the startup routine of the user application, user needs to care of this TBR register initialization. The TBR relative vector table can have at max 256 function entries. The jumps to these functions will be generated using a SH2A specific, non delayed branch instruction JSR/N @(disp8,TBR). You must use GAS and GLD from GNU binutils version 2.7 or later for this attribute to work correctly.

Please refer the example of M16C target, to see the use of this attribute while declaring a function,

In an application, for a function being called once, this attribute will save at least 8 bytes of code; and if other successive calls are being made to the same function, it will save 2 bytes of code per each of these calls.

On M16C/M32C targets, the `function_vector` attribute declares a special page subroutine call function. Use of this attribute reduces the code size by 2 bytes for each call generated to the subroutine. The argument to the attribute is the vector number entry from the special page vector table which contains the 16 low-order bits of the subroutine's entry address. Each vector table has special page number (18 to 255) which are used in `jsrs` instruction. Jump addresses of the routines are generated by adding 0x0F0000 (in case of M16C targets) or 0xFF0000 (in case of M32C targets), to the 2 byte addresses set in the vector table. Therefore you need to ensure that all the special page vector routines should get mapped within the address range 0x0F0000 to 0x0FFFFF (for M16C) and 0xFF0000 to 0xFFFFFF (for M32C).

In the following example 2 bytes will be saved for each call to function `foo`.

```
void foo (void) __attribute__((function_vector(0x18)));
void foo (void)
{
}

void bar (void)
{
    foo();
}
```

If functions are defined in one file and are called in another file, then be sure to write this declaration in both files.

This attribute is ignored for R8C target.

`interrupt`

Use this attribute on the ARM, AVR, CRX, M32C, M32R/D, m68k, and Xstormy16 ports to indicate that the specified function is an interrupt handler. The compiler will generate function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

Note, interrupt handlers for the Blackfin, H8/300, H8/300H, H8S, and SH processors can be specified via the `interrupt_handler` attribute.

Note, on the AVR, interrupts will be enabled inside the function.

Note, for the ARM, you can specify the kind of interrupt to be handled by adding an optional parameter to the interrupt attribute like this:

```
void f () __attribute__ ((interrupt ("IRQ")));
```

Permissible values for this parameter are: IRQ, FIQ, SWI, ABORT and UNDEF.

On ARMv7-M the interrupt type is ignored, and the attribute means the function may be called with a word aligned stack pointer.

`interrupt_handler`

Use this attribute on the Blackfin, m68k, H8/300, H8/300H, H8S, and SH to indicate that the specified function is an interrupt handler. The compiler will generate function entry and exit sequences suitable for use in an interrupt handler when this attribute is present.

`interrupt_thread`

Use this attribute on fido, a subarchitecture of the m68k, to indicate that the specified function is an interrupt handler that is designed to run as a thread. The compiler omits generate prologue/epilogue sequences and replaces the return instruction with a `sleep` instruction. This attribute is available only on fido.

`isr`

Use this attribute on ARM to write Interrupt Service Routines. This is an alias to the `interrupt` attribute above.

`kspisusp`

When used together with `interrupt_handler`, `exception_handler` or `nmi_handler`, code will be generated to load the stack pointer from the USP register in the function prologue.

`l1_text`

This attribute specifies a function to be placed into L1 Instruction SRAM. The function will be put into a specific section named `.l1.text`. With `-mfdpic`, function calls with a such function as the callee or caller will use inlined PLT.

`long_call/short_call`

This attribute specifies how a particular function is called on ARM. Both attributes override the `-mlong-calls` (see ARM Options) command line switch and `#pragma long_calls` settings. The `long_call` attribute indicates that the function might be far away from the call site and require a different (more expensive) calling sequence. The `short_call` attribute always places the offset to the function from the call site into the `BL' instruction directly.

`longcall/shortcall`

On the Blackfin, RS/6000 and PowerPC, the `longcall` attribute indicates that the function might be far away from the call site and require a different (more expensive) calling sequence. The `shortcall` attribute indicates that the function is always close enough for the shorter calling sequence to be used. These attributes override both the `-mlongcall` switch and, on the RS/6000 and PowerPC, the `#pragma longcall` setting.

See RS/6000 and PowerPC Options, for more information on whether long calls are necessary.

`long_call/near/far`

These attributes specify how a particular function is called on MIPS. The attributes override the `-mlong-calls` (see MIPS Options) command-line switch. The `long_call` and `far` attributes are synonyms, and cause the compiler to always call the function by first loading its address into a register, and then using the contents of that register. The `near` attribute has the opposite effect; it specifies that non-PIC calls should be made using the more efficient `jal` instruction.

`malloc`

The `malloc` attribute is used to tell the compiler that a function may be treated as if any non-`NULL` pointer it returns cannot alias any other pointer valid when the function returns. This will often improve optimization. Standard functions with this property include `malloc` and `calloc`. `realloc`-like functions have this property as long as the old pointer is never referred to (including comparing it to the new pointer) after the function returns a non-`NULL` value.

`mips16/nomips16`

On MIPS targets, you can use the `mips16` and `nomips16` function attributes to locally select or turn off MIPS16 code generation. A function with the `mips16` attribute is emitted as MIPS16 code, while MIPS16 code generation is disabled for functions with the `nomips16` attribute. These attributes override the `-mips16` and `-mno-mips16` options on the command line (see MIPS Options).

When compiling files containing mixed MIPS16 and non-MIPS16 code, the preprocessor symbol `__mips16` reflects the setting on the command line, not that within individual functions. Mixed MIPS16 and non-MIPS16 code may interact badly with some GCC extensions such as `__builtin_apply` (see Constructing Calls).

`model (`*model-name*`)`

On the M32R/D, use this attribute to set the addressability of an object, and of the code generated for a function. The identifier *model-name* is one of `small`, `medium`, or `large`, representing each of the code models.

Small model objects live in the lower 16MB of memory (so that their addresses can be loaded with the `ld24` instruction), and are callable with the `bl` instruction.

Medium model objects may live anywhere in the 32-bit address space (the compiler will generate `seth/add3` instructions to load their addresses), and are callable with the `bl` instruction.

Large model objects may live anywhere in the 32-bit address space (the compiler will generate `seth/add3` instructions to load their addresses), and may not be reachable with the `bl` instruction (the compiler will generate the much slower `seth/add3/jl` instruction sequence).

On IA-64, use this attribute to set the addressability of an object. At present, the only supported identifier for *model-name* is `small`, indicating addressability via "small" (22-bit) addresses (so that their addresses can be loaded with the `addl` instruction). Caveat: such addressing is by definition not position independent and hence this attribute must not be used for objects defined by shared libraries.

`ms_abi/sysv_abi`
On 64-bit x86_64-*-* targets, you can use an ABI attribute to indicate which calling convention should be used for a function. The `ms_abi` attribute tells the compiler to use the Microsoft ABI, while the `sysv_abi` attribute tells the compiler to use the ABI used on GNU/Linux and other systems. The default is to use the Microsoft ABI when targeting Windows. On all other systems, the default is the AMD ABI.

Note, This feature is currently sorried out for Windows targets trying to

`naked`
Use this attribute on the ARM, AVR, IP2K and SPU ports to indicate that the specified function does not need prologue/epilogue sequences generated by the compiler. It is up to the programmer to provide these sequences. The only statements that can be safely included in naked functions are `asm` statements that do not have operands. All other statements, including declarations of local variables, `if` statements, and so forth, should be avoided. Naked functions should be used to implement the body of an assembly function, while allowing the compiler to construct the requisite function declaration for the assembler.

`near`
On 68HC11 and 68HC12 the `near` attribute causes the compiler to use the normal calling convention based on `jsr` and `rts`. This attribute can be used to cancel the effect of the `-mlong-calls` option.

`nesting`
Use this attribute together with `interrupt_handler`, `exception_handler` or `nmi_handler` to indicate that the function entry code should enable nested interrupts or exceptions.

`nmi_handler`
Use this attribute on the Blackfin to indicate that the specified function is an NMI handler. The compiler will generate function entry and exit sequences suitable for use in an NMI handler when this attribute is present.

`no_instrument_function`
If `-finstrument-functions` is given, profiling function calls will be generated at entry and exit of most user-compiled functions. Functions with this attribute will not be so instrumented.

`noinline`
This function attribute prevents a function from being considered for inlining. If the function does not have side-effects, there are optimizations other than inlining that causes function calls to be optimized away, although the function call is live. To keep such calls from being optimized away, put

```
asm ("");
```

(see Extended Asm) in the called function, to serve as a special side-effect.

`nonnull (`*arg-index*`, ...)`
The `nonnull` attribute specifies that some function parameters should be non-null pointers. For instance, the declaration:

```
extern void *
my_memcpy (void *dest, const void *src, size_t len)
        __attribute__((nonnull (1, 2)));
```

causes the compiler to check that, in calls to `my_memcpy`, arguments *dest* and *src* are non-null. If the compiler determines that a null pointer is passed in an argument slot marked as non-null, and the

`-Wnonnull` option is enabled, a warning is issued. The compiler may also choose to make optimizations based on the knowledge that certain function arguments will not be null.

If no argument index list is given to the `nonnull` attribute, all pointer arguments are marked as non-null. To illustrate, the following declaration is equivalent to the previous example:

```
extern void *
my_memcpy (void *dest, const void *src, size_t len)
        __attribute__((nonnull));
```

`noreturn`

A few standard library functions, such as `abort` and `exit`, cannot return. GCC knows this automatically. Some programs define their own functions that never return. You can declare them `noreturn` to tell the compiler this fact. For example,

```
void fatal () __attribute__ ((noreturn));

void
fatal (/* ... */)
{
  /* ... */ /* Print error message. */ /* ... */
  exit (1);
}
```

The `noreturn` keyword tells the compiler to assume that `fatal` cannot return. It can then optimize without regard to what would happen if `fatal` ever did return. This makes slightly better code. More importantly, it helps avoid spurious warnings of uninitialized variables.

The `noreturn` keyword does not affect the exceptional path when that applies: a `noreturn`-marked function may still return to the caller by throwing an exception or calling `longjmp`.

Do not assume that registers saved by the calling function are restored before calling the `noreturn` function.

It does not make sense for a `noreturn` function to have a return type other than `void`.

The attribute `noreturn` is not implemented in GCC versions earlier than 2.5. An alternative way to declare that a function does not return, which works in the current version and in some older versions, is as follows:

```
typedef void voidfn ();

volatile voidfn fatal;
```

This approach does not work in GNU C++.

`nothrow`

The `nothrow` attribute is used to inform the compiler that a function cannot throw an exception. For example, most functions in the standard C library can be guaranteed not to throw an exception with the notable exceptions of `qsort` and `bsearch` that take function pointer arguments. The `nothrow` attribute is not implemented in GCC versions earlier than 3.3.

`optimize`

The `optimize` attribute is used to specify that a function is to be compiled with different optimization options than specified on the command line. Arguments can either be numbers or strings. Numbers are assumed to be an optimization level. Strings that begin with `O` are assumed to be an optimization option, while other options are assumed to be used with a `-f` prefix. You can also use the `` `#pragma GCC ``

`optimize' pragma to set the optimization options that affect more than one function. See Function Specific Option Pragmas, for details about the `#pragma GCC optimize' pragma.

This can be used for instance to have frequently executed functions compiled with more aggressive optimization options that produce faster and larger code, while other functions can be called with less aggressive options.

`pure`

Many functions have no effects except the return value and their return value depends only on the parameters and/or global variables. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared with the attribute `pure`. For example,

```
int square (int) __attribute__ ((pure));
```

says that the hypothetical function `square` is safe to call fewer times than the program says.

Some of common examples of pure functions are `strlen` or `memcmp`. Interesting non-pure functions are functions with infinite loops or those depending on volatile memory or other system resource, that may change between two consecutive calls (such as `feof` in a multithreading environment).

The attribute `pure` is not implemented in GCC versions earlier than 2.96.

`hot`

The `hot` attribute is used to inform the compiler that a function is a hot spot of the compiled program. The function is optimized more aggressively and on many target it is placed into special subsection of the text section so all hot functions appears close together improving locality.

When profile feedback is available, via `-fprofile-use`, hot functions are automatically detected and this attribute is ignored.

The `hot` attribute is not implemented in GCC versions earlier than 4.3.

`cold`

The `cold` attribute is used to inform the compiler that a function is unlikely executed. The function is optimized for size rather than speed and on many targets it is placed into special subsection of the text section so all cold functions appears close together improving code locality of non-cold parts of program. The paths leading to call of cold functions within code are marked as unlikely by the branch prediction mechanism. It is thus useful to mark functions used to handle unlikely conditions, such as `perror`, as cold to improve optimization of hot functions that do call marked functions in rare occasions.

When profile feedback is available, via `-fprofile-use`, hot functions are automatically detected and this attribute is ignored.

The `cold` attribute is not implemented in GCC versions earlier than 4.3.

`regparm (`*number*`)`

On the Intel 386, the `regparm` attribute causes the compiler to pass arguments number one to *number* if they are of integral type in registers EAX, EDX, and ECX instead of on the stack. Functions that take a variable number of arguments will continue to be passed all of their arguments on the stack.

Beware that on some ELF systems this attribute is unsuitable for global functions in shared libraries

with lazy binding (which is the default). Lazy binding will send the first call via resolving code in the loader, which might assume EAX, EDX and ECX can be clobbered, as per the standard calling conventions. Solaris 8 is affected by this. GNU systems with GLIBC 2.1 or higher, and FreeBSD, are believed to be safe since the loaders there save EAX, EDX and ECX. (Lazy binding can be disabled with the linker or the loader if desired, to avoid the problem.)

`sseregparm`
On the Intel 386 with SSE support, the `sseregparm` attribute causes the compiler to pass up to 3 floating point arguments in SSE registers instead of on the stack. Functions that take a variable number of arguments will continue to pass all of their floating point arguments on the stack.

`force_align_arg_pointer`
On the Intel x86, the `force_align_arg_pointer` attribute may be applied to individual function definitions, generating an alternate prologue and epilogue that realigns the runtime stack if necessary. This supports mixing legacy codes that run with a 4-byte aligned stack with modern codes that keep a 16-byte stack for SSE compatibility.

`resbank`
On the SH2A target, this attribute enables the high-speed register saving and restoration using a register bank for `interrupt_handler` routines. Saving to the bank is performed automatically after the CPU accepts an interrupt that uses a register bank.

The nineteen 32-bit registers comprising general register R0 to R14, control register GBR, and system registers MACH, MACL, and PR and the vector table address offset are saved into a register bank. Register banks are stacked in first-in last-out (FILO) sequence. Restoration from the bank is executed by issuing a RESBANK instruction.

`returns_twice`
The `returns_twice` attribute tells the compiler that a function may return more than one time. The compiler will ensure that all registers are dead before calling such a function and will emit a warning about the variables that may be clobbered after the second return from the function. Examples of such functions are `setjmp` and `vfork`. The `longjmp`-like counterpart of such function, if any, might need to be marked with the `noreturn` attribute.

`saveall`
Use this attribute on the Blackfin, H8/300, H8/300H, and H8S to indicate that all registers except the stack pointer should be saved in the prologue regardless of whether they are used or not.

`section ("`*section-name*`")`
Normally, the compiler places the code it generates in the `text` section. Sometimes, however, you need additional sections, or you need certain particular functions to appear in special sections. The `section` attribute specifies that a function lives in a particular section. For example, the declaration:

```
extern void foobar (void) __attribute__ ((section ("bar")));
```

puts the function `foobar` in the `bar` section.

Some file formats do not support arbitrary sections so the `section` attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

`sentinel`
This function attribute ensures that a parameter in a function call is an explicit `NULL`. The attribute is only valid on variadic functions. By default, the sentinel is located at position zero, the last parameter of the function call. If an optional integer position argument P is supplied to the attribute, the sentinel must be located at position P counting backwards from the end of the argument list.

```
__attribute__ ((sentinel))
is equivalent to
__attribute__ ((sentinel(0)))
```

The attribute is automatically set with a position of 0 for the built-in functions `execl` and `execlp`. The built-in function `execle` has the attribute set with a position of 1.

A valid `NULL` in this context is defined as zero with any pointer type. If your system defines the `NULL` macro with an integer type then you need to add an explicit cast. GCC replaces `stddef.h` with a copy that redefines NULL appropriately.

The warnings for missing or incorrect sentinels are enabled with `-Wformat`.

`short_call`
See long_call/short_call.

`shortcall`
See longcall/shortcall.

`signal`
Use this attribute on the AVR to indicate that the specified function is a signal handler. The compiler will generate function entry and exit sequences suitable for use in a signal handler when this attribute is present. Interrupts will be disabled inside the function.

`sp_switch`
Use this attribute on the SH to indicate an `interrupt_handler` function should switch to an alternate stack. It expects a string argument that names a global variable holding the address of the alternate stack.

```
void *alt_stack;
void f () __attribute__ ((interrupt_handler,
                          sp_switch ("alt_stack")));
```

`stdcall`
On the Intel 386, the `stdcall` attribute causes the compiler to assume that the called function will pop off the stack space used to pass arguments, unless it takes a variable number of arguments.

`syscall_linkage`
This attribute is used to modify the IA64 calling convention by marking all input registers as live at all function exits. This makes it possible to restart a system call after an interrupt without having to save/restore the input registers. This also prevents kernel data from leaking into application code.

`target`
The `target` attribute is used to specify that a function is to be compiled with different target options than specified on the command line. This can be used for instance to have functions compiled with a different ISA (instruction set architecture) than the default. You can also use the `` `#pragma GCC target' `` pragma to set more than one function to be compiled with specific target options. See Function Specific Option Pragmas, for details about the `` `#pragma GCC target' `` pragma.

For instance on a 386, you could compile one function with `target("sse4.1,arch=core2")` and another with `target("sse4a,arch=amdfam10")` that would be equivalent to compiling the first function with `-msse4.1` and `-march=core2` options, and the second function with `-msse4a` and `-march=amdfam10` options. It is up to the user to make sure that a function is only invoked on a machine that supports the particular ISA it was compiled for (for example by using `cpuid` on 386 to determine what feature bits and architecture family are used).

```
int core2_func (void) __attribute__ ((__target__ ("arch=core2")));
int sse3_func (void) __attribute__ ((__target__ ("sse3")));
```

On the 386, the following options are allowed:

`abm'
`no-abm'
Enable/disable the generation of the advanced bit instructions.

`aes'
`no-aes'
Enable/disable the generation of the AES instructions.

`mmx'
`no-mmx'
Enable/disable the generation of the MMX instructions.

`pclmul'
`no-pclmul'
Enable/disable the generation of the PCLMUL instructions.

`popcnt'
`no-popcnt'
Enable/disable the generation of the POPCNT instruction.

`sse'
`no-sse'
Enable/disable the generation of the SSE instructions.

`sse2'
`no-sse2'
Enable/disable the generation of the SSE2 instructions.

`sse3'
`no-sse3'
Enable/disable the generation of the SSE3 instructions.

`sse4'
`no-sse4'
Enable/disable the generation of the SSE4 instructions (both SSE4.1 and SSE4.2).

`sse4.1'
`no-sse4.1'
Enable/disable the generation of the sse4.1 instructions.

`sse4.2'
`no-sse4.2'
Enable/disable the generation of the sse4.2 instructions.

`sse4a'
`no-sse4a'
Enable/disable the generation of the SSE4A instructions.

`sse5'
`no-sse5'
Enable/disable the generation of the SSE5 instructions.

`ssse3'
`no-ssse3'
Enable/disable the generation of the SSSE3 instructions.

`cld'
`no-cld'
Enable/disable the generation of the CLD before string moves.

`fancy-math-387'
`no-fancy-math-387'
Enable/disable the generation of the `sin`, `cos`, and `sqrt` instructions on the 387 floating point unit.

`fused-madd'
`no-fused-madd'
Enable/disable the generation of the fused multiply/add instructions.

`ieee-fp'
`no-ieee-fp'
Enable/disable the generation of floating point that depends on IEEE arithmetic.

`inline-all-stringops'
`no-inline-all-stringops'
Enable/disable inlining of string operations.

`inline-stringops-dynamically'
`no-inline-stringops-dynamically'
Enable/disable the generation of the inline code to do small string operations and calling the library routines for large operations.

`align-stringops'
`no-align-stringops'
Do/do not align destination of inlined string operations.

`recip'
`no-recip'
Enable/disable the generation of RCPSS, RCPPS, RSQRTSS and RSQRTPS instructions followed an additional Newton-Raphson step instead of doing a floating point division.

`arch=*ARCH*'
Specify the architecture to generate code for in compiling the function.

`tune=*TUNE*'
Specify the architecture to tune for in compiling the function.

`fpmath=*FPMATH*'
Specify which floating point unit to use. The `target("fpmath=sse,387")` option must be specified as `target("fpmath=sse+387")` because the comma would separate different options.

On the 386, you can use either multiple strings to specify multiple options, or you can separate the option with a comma (`,`).

On the 386, the inliner will not inline a function that has different target options than the caller, unless the callee has a subset of the target options of the caller. For example a function declared with `target("sse5")` can inline a function with `target("sse2")`, since `-msse5` implies `-msse2`.

The `target` attribute is not implemented in GCC versions earlier than 4.4, and at present only the 386 uses it.

`tiny_data`
Use this attribute on the H8/300H and H8S to indicate that the specified variable should be placed into the tiny data section. The compiler will generate more efficient code for loads and stores on data in the tiny data section. Note the tiny data area is limited to slightly under 32kbytes of data.

`trap_exit`
Use this attribute on the SH for an `interrupt_handler` to return using `trapa` instead of `rte`. This attribute expects an integer argument specifying the trap number to be used.

`unused`
This attribute, attached to a function, means that the function is meant to be possibly unused. GCC will not produce a warning for this function.

`used`
This attribute, attached to a function, means that code must be emitted for the function even if it appears that the function is not referenced. This is useful, for example, when the function is referenced only in inline assembly.

`version_id`

This IA64 HP-UX attribute, attached to a global variable or function, renames a symbol to contain a version string, thus allowing for function level versioning. HP-UX system header files may use version level functioning for some system calls.

```
extern int foo () __attribute__((version_id ("20040821")));
```

Calls to *foo* will be mapped to calls to *foo{20040821}*.

`visibility ("`*visibility_type*`")`

This attribute affects the linkage of the declaration to which it is attached. There are four supported *visibility_type* values: default, hidden, protected or internal visibility.

```
void __attribute__ ((visibility ("protected")))
f () { /* Do something. */; }
int i __attribute__ ((visibility ("hidden")));
```

The possible values of *visibility_type* correspond to the visibility settings in the ELF gABI.

default

Default visibility is the normal case for the object file format. This value is available for the visibility attribute to override other options that may change the assumed visibility of entities.

On ELF, default visibility means that the declaration is visible to other modules and, in shared libraries, means that the declared entity may be overridden.

On Darwin, default visibility means that the declaration is visible to other modules.

Default visibility corresponds to “external linkage” in the language.

hidden

Hidden visibility indicates that the entity declared will have a new form of linkage, which we'll call “hidden linkage”. Two declarations of an object with hidden linkage refer to the same object if they are in the same shared object.

internal

Internal visibility is like hidden visibility, but with additional processor specific semantics. Unless otherwise specified by the psABI, GCC defines internal visibility to mean that a function is *never* called from another module. Compare this with hidden functions which, while they cannot be referenced directly by other modules, can be referenced indirectly via function pointers. By indicating that a function cannot be called from outside the module, GCC may for instance omit the load of a PIC register since it is known that the calling function loaded the correct value.

protected

Protected visibility is like default visibility except that it indicates that references within the defining module will bind to the definition in that module. That is, the declared entity cannot be overridden by another module.

All visibilities are supported on many, but not all, ELF targets (supported when the assembler supports the `.visibility' pseudo-op). Default visibility is supported everywhere. Hidden visibility is supported on Darwin targets.

The visibility attribute should be applied only to declarations which would otherwise have external linkage. The attribute should be applied consistently, so that the same entity should not be declared with different settings of the attribute.

In C++, the visibility attribute applies to types as well as functions and objects, because in C++ types have linkage. A class must not have greater visibility than its non-static data member types and bases, and class members default to the visibility of their class. Also, a declaration without explicit visibility is limited to the visibility of its type.

In C++, you can mark member functions and static member variables of a class with the visibility attribute. This is useful if you know a particular method or static member variable should only be used from one shared object; then you can mark it hidden while the rest of the class has default visibility. Care must be taken to avoid breaking the One Definition Rule; for example, it is usually not useful to mark an inline method as hidden without marking the whole class as hidden.

A C++ namespace declaration can also have the visibility attribute. This attribute applies only to the particular namespace body, not to other definitions of the same namespace; it is equivalent to using `#pragma GCC visibility' before and after the namespace definition (see Visibility Pragmas).

In C++, if a template argument has limited visibility, this restriction is implicitly propagated to the template instantiation. Otherwise, template instantiations and specializations default to the visibility of their template.

If both the template and enclosing class have explicit visibility, the visibility from the template is used.

`warn_unused_result`

The `warn_unused_result` attribute causes a warning to be emitted if a caller of the function with this attribute does not use its return value. This is useful for functions where not checking the result is either a security problem or always a bug, such as `realloc`.

```
int fn () __attribute__ ((warn_unused_result));
int foo ()
{
  if (fn () < 0) return -1;
  fn ();
  return 0;
}
```

results in warning on line 5.

`weak`

The `weak` attribute causes the declaration to be emitted as a weak symbol rather than a global. This is primarily useful in defining library functions which can be overridden in user code, though it can also be used with non-function declarations. Weak symbols are supported for ELF targets, and also for a.out targets when using the GNU assembler and linker.

`weakref`

`weakref ("`*target*`")`

The `weakref` attribute marks a declaration as a weak reference. Without arguments, it should be accompanied by an `alias` attribute naming the target symbol. Optionally, the *target* may be given as an argument to `weakref` itself. In either case, `weakref` implicitly marks the declaration as `weak`. Without a *target*, given as an argument to `weakref` or to `alias`, `weakref` is equivalent to `weak`.

```
static int x() __attribute__ ((weakref ("y")));
/* is equivalent to... */
static int x() __attribute__ ((weak, weakref, alias ("y")));
/* and to... */
static int x() __attribute__ ((weakref));
static int x() __attribute__ ((alias ("y")));
```

A weak reference is an alias that does not by itself require a definition to be given for the target symbol. If the target symbol is only referenced through weak references, then the becomes a `weak` undefined

symbol. If it is directly referenced, however, then such strong references prevail, and a definition will be required for the symbol, not necessarily in the same translation unit.

The effect is equivalent to moving all references to the alias to a separate translation unit, renaming the alias to the aliased symbol, declaring it as weak, compiling the two separate translation units and performing a reloadable link on them.

At present, a declaration to which `weakref` is attached can only be `static`.

You can specify multiple attributes in a declaration by separating them by commas within the double parentheses or by immediately following an attribute declaration with another attribute declaration.

Some people object to the `__attribute__` feature, suggesting that ISO C's `#pragma` should be used instead. At the time `__attribute__` was designed, there were two reasons for not doing this.

1. It is impossible to generate `#pragma` commands from a macro.
2. There is no telling what the same `#pragma` might mean in another compiler.

These two reasons applied to almost any application that might have been proposed for `#pragma`. It was basically a mistake to use `#pragma` for *anything*.

The ISO C99 standard includes `_Pragma`, which now allows pragmas to be generated from macros. In addition, a `#pragma GCC` namespace is now in use for GCC-specific pragmas. However, it has been found convenient to use `__attribute__` to achieve a natural attachment of attributes to their corresponding declarations, whereas `#pragma GCC` is of use for constructs that do not naturally form part of the grammar. See Miscellaneous Preprocessing Directives.