6.34.1 Common Variable Attributes

The following attributes are supported on most targets.

alias ("*target*")

The alias variable attribute causes the declaration to be emitted as an alias for another symbol known as an *alias target*. Except for top-level qualifiers the alias target must have the same type as the alias. For instance, the following

```
int var_target;
extern int __attribute__ ((alias ("var_target"))) var_alias;
```

defines var_alias to be an alias for the var_target variable.

It is an error if the alias target is not defined in the same translation unit as the alias.

Note that in the absence of the attribute GCC assumes that distinct declarations with external linkage denote distinct objects. Using both the alias and the alias target to access the same object is undefined in a translation unit without a declaration of the alias with the attribute.

This attribute requires assembler and object file support, and may not be available on all targets.

aligned
aligned (*alignment*)

The aligned attribute specifies a minimum alignment for the variable or structure field, measured in bytes. When specified, *alignment* must be an integer constant power of 2. Specifying no *alignment* argument implies the maximum alignment for the target, which is often, but by no means always, 8 or 16 bytes.

For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable x on a 16-byte boundary. On a 68040, this could be used in conjunction with an asm expression to access the move16 instruction which requires 16-byte aligned operands.

You can also specify the alignment of structure fields. For example, to create a double-word aligned int pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a `double` member, which forces the union to be double-word aligned.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the default alignment for the target architecture you are compiling for. The default alignment is sufficient for all scalar types, but may not be enough for all vector types on a target that supports vector operations. The default alignment is fixed for a particular target ABI.

GCC also provides a target specific macro `__BIGGEST_ALIGNMENT__`, which is the largest alignment ever used for any data type on the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned (__BIGGEST_ALIGNMENT__)));
```

The compiler automatically sets the alignment for the declared variable or field to `__BIGGEST_ALIGNMENT__`. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way. Note that the value of `__BIGGEST_ALIGNMENT__` may change depending on command-line options.

When used on a struct, or struct member, the `aligned` attribute can only increase the alignment; in order to decrease it, the `packed` attribute must be specified as well. When used as part of a typedef, the `aligned` attribute can both increase and decrease alignment, and specifying the `packed` attribute generates a warning.

Note that the effectiveness of `aligned` attributes for static variables may be limited by inherent limitations in the system linker and/or object file format. On some systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8-byte alignment, then specifying `aligned(16)` in an `__attribute__` still only provides you with 8-byte alignment. See your linker documentation for further information.

Stack variables are not affected by linker restrictions; GCC can properly align them on any target.

The `aligned` attribute can also be used for functions (see Common Function Attributes.)

warn_if_not_aligned (*alignment*)

This attribute specifies a threshold for the structure field, measured in bytes. If the structure field is aligned below the threshold, a warning will be issued. For example, the declaration:

```
struct foo
{
  int i1;
  int i2;
  unsigned long long x __attribute__ ((warn_if_not_aligned (16)));
};
```

causes the compiler to issue an warning on `struct foo`, like '`warning: alignment 8 of 'struct foo' is less than 16`'. The compiler also issues a warning, like '`warning: 'x' offset 8 in 'struct foo' isn't aligned to 16`', when the structure field has the misaligned offset:

```
struct __attribute__ ((aligned (16))) foo
{
  int i1;
  int i2;
  unsigned long long x __attribute__ ((warn_if_not_aligned (16)));
};
```

This warning can be disabled by `-Wno-if-not-aligned`. The `warn_if_not_aligned` attribute can also be used for types (see Common Type Attributes.)

`alloc_size (`*position*`)`
`alloc_size (`*position-1*`, `*position-2*`)`

The `alloc_size` variable attribute may be applied to the declaration of a pointer to a function that returns a pointer and takes at least one argument of an integer type. It indicates that the returned pointer points to an object whose size is given by the function argument at *position*, or by the product of the arguments at *position-1* and *position-2*. Meaningful sizes are positive values less than `PTRDIFF_MAX`. Other sizes are diagnosed when detected. GCC uses this information to improve the results of `__builtin_object_size`.

For instance, the following declarations

```
typedef __attribute__ ((alloc_size (1, 2))) void*
  (*calloc_ptr) (size_t, size_t);
typedef __attribute__ ((alloc_size (1))) void*
  (*malloc_ptr) (size_t);
```

specify that `calloc_ptr` is a pointer of a function that, like the standard C function `calloc`, returns an object whose size is given by the product of arguments 1 and 2, and similarly, that `malloc_ptr`, like the standard C function `malloc`, returns an object whose size is given by argument 1 to the function.

cleanup (*cleanup_function*)

The cleanup attribute runs a function when the variable goes out of scope. This attribute can only be applied to auto function scope variables; it may not be applied to parameters or variables with static storage duration. The function must take one parameter, a pointer to a type compatible with the variable. The return value of the function (if any) is ignored.

If -fexceptions is enabled, then *cleanup_function* is run during the stack unwinding that happens during the processing of the exception. Note that the cleanup attribute does not allow the exception to be caught, only to perform an action. It is undefined what happens if *cleanup_function* does not return normally.

common
nocommon

The common attribute requests GCC to place a variable in "common" storage. The nocommon attribute requests the opposite—to allocate space for it directly.

These attributes override the default chosen by the -fno-common and -fcommon flags respectively.

copy
copy (*variable*)

The copy attribute applies the set of attributes with which *variable* has been declared to the declaration of the variable to which the attribute is applied. The attribute is designed for libraries that define aliases that are expected to specify the same set of attributes as the aliased symbols. The copy attribute can be used with variables, functions or types. However, the kind of symbol to which the attribute is applied (either varible or function) must match the kind of symbol to which the argument refers. The copy attribute copies only syntactic and semantic attributes but not attributes that affect a symbol's linkage or visibility such as alias, visibility, or weak. The deprecated attribute is also not copied. See Common Function Attributes. See Common Type Attributes.

deprecated
deprecated (*msg*)

The deprecated attribute results in a warning if the variable is used anywhere in the source file. This is useful when identifying variables 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 variable, to enable users to easily find further information about why the variable is deprecated, or what they should do instead. Note that the warning only occurs for uses:

```
extern int old_var __attribute__ ((deprecated));
extern int old_var;
int new_fn () { return old_var; }
```

results in a warning on line 3 but not line 2. The optional *msg* argument, which must be a string, is printed in the warning if present.

The deprecated attribute can also be used for functions and types (see Common Function Attributes, see Common Type Attributes).

The message attached to the attribute is affected by the setting of the -fmessage-length option.

unavailable
unavailable (*msg*)

The unavailable attribute indicates that the variable so marked is not available, if it is used anywhere in the source file. It behaves in the same manner as the deprecated attribute except that the compiler will emit an error rather than a warning.

It is expected that items marked as deprecated will eventually be withdrawn from interfaces, and then become unavailable. This attribute allows for marking them appropriately.

The unavailable attribute can also be used for functions and types (see Common Function Attributes, see Common Type Attributes).

mode (*mode*)

This attribute specifies the data type for the declaration—whichever type corresponds to the mode *mode*. This in effect lets you request an integer or floating-point type according to its width.

See Machine Modes in *GNU Compiler Collection (GCC) Internals*, for a list of the possible keywords for *mode*. You may also specify a mode of byte or __byte__ to indicate the mode corresponding to a one-byte integer, word or __word__ for the mode of a one-word integer, and pointer or __pointer__ for the mode used to represent pointers.

nonstring

The nonstring variable attribute specifies that an object or member declaration with type array of char, signed char, or unsigned char, or pointer to such a type is intended to store character arrays that do not necessarily contain a terminating NUL. This is useful in detecting uses of such arrays or pointers with functions that expect NUL-terminated strings, and to avoid warnings when such an array or pointer is used as an argument to a bounded string manipulation function such as strncpy. For example, without the attribute, GCC will issue a warning for the strncpy call below because it may truncate the copy without appending the terminating NUL character. Using the attribute makes it possible to suppress the warning. However, when the array is declared with the attribute the call to strlen is diagnosed because when the array doesn't contain a NUL-terminated string the call is undefined. To copy, compare, of search non-string character arrays use the memcpy, memcmp, memchr, and other functions that operate on arrays of bytes. In

addition, calling strnlen and strndup with such arrays is safe provided a suitable bound is specified, and not diagnosed.

```
struct Data
{
  char name [32] __attribute__ ((nonstring));
};

int f (struct Data *pd, const char *s)
{
  strncpy (pd->name, s, sizeof pd->name);
  ...
  return strlen (pd->name);   // unsafe, gets a warning
}
```

packed

The packed attribute specifies that a structure member should have the smallest possible alignment—one bit for a bit-field and one byte otherwise, unless a larger value is specified with the aligned attribute. The attribute does not apply to non-member objects.

For example in the structure below, the member array x is packed so that it immediately follows a with no intervening padding:

```
struct foo
{
  char a;
  int x[2] __attribute__ ((packed));
};
```

Note: The 4.1, 4.2 and 4.3 series of GCC ignore the packed attribute on bit-fields of type char. This has been fixed in GCC 4.4 but the change can lead to differences in the structure layout. See the documentation of -Wpacked-bitfield-compat for more information.

section ("*section-name*")

Normally, the compiler places the objects it generates in sections like data and bss. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The section attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
int init_data __attribute__ ((section ("INITDATA")));

main()
{
  /* Initialize stack pointer */
  init_sp (stack + sizeof (stack));

  /* Initialize initialized data */
  memcpy (&init_data, &data, &edata - &data);

  /* Turn on the serial ports */
  init_duart (&a);
  init_duart (&b);
}
```

Use the section attribute with *global* variables and not *local* variables, as shown in the example.

You may use the section attribute with initialized or uninitialized global variables but the linker requires each object be defined once, with the exception that uninitialized variables tentatively go in the common (or bss) section and can be multiply "defined". Using the section attribute changes what section the variable goes into and may cause the linker to issue an error if an uninitialized variable has multiple definitions. You can force a variable to be initialized with the -fno-common flag or the nocommon attribute.

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.

tls_model ("*tls_model*")

The tls_model attribute sets thread-local storage model (see Thread-Local) of a particular __thread variable, overriding -ftls-model= command-line switch on a per-variable basis. The *tls_model* argument should be one of global-dynamic, local-dynamic, initial-exec or local-exec.

Not all targets support this attribute.

unused

This attribute, attached to a variable or structure field, means that the variable or field is meant to be possibly unused. *GCC* does not produce a warning for this variable or field.

used

This attribute, attached to a variable with static storage, means that the variable must be emitted even if it appears that the variable is not referenced.

When applied to a static data member of a C++ class template, the attribute also means that the member is instantiated if the class itself is instantiated.

retain

For ELF targets that support the GNU or FreeBSD OSABIs, this attribute will save the variable from linker garbage collection. To support this behavior, variables that have not been placed in specific sections (e.g. by the `section` attribute, or the `-fdata-sections` option), will be placed in new, unique sections.

This additional functionality requires Binutils version 2.36 or later.

uninitialized

This attribute, attached to a variable with automatic storage, means that the variable should not be automatically initialized by the compiler when the option `-ftrivial-auto-var-init` presents.

With the option `-ftrivial-auto-var-init`, all the automatic variables that do not have explicit initializers will be initialized by the compiler. These additional compiler initializations might incur run-time overhead, sometimes dramatically. This attribute can be used to mark some variables to be excluded from such automatical initialization in order to reduce runtime overhead.

This attribute has no effect when the option `-ftrivial-auto-var-init` does not present.

vector_size (*bytes*)

This attribute specifies the vector size for the type of the declared variable, measured in bytes. The type to which it applies is known as the *base type*. The *bytes* argument must be a positive power-of-two multiple of the base type size. For example, the declaration:

```
int foo __attribute__ ((vector_size (16)));
```

causes the compiler to set the mode for `foo`, to be 16 bytes, divided into `int` sized units. Assuming a 32-bit `int`, `foo`'s type is a vector of four units of four bytes each, and the corresponding mode of `foo` is `V4SI`. See Vector Extensions, for details of manipulating vector variables.

This attribute is only applicable to integral and floating scalars, although arrays, pointers, and function return values are allowed in conjunction with this construct.

Aggregates with this attribute are invalid, even if they are of the same size as a corresponding scalar. For example, the declaration:

```
struct S { int a; };
struct S  __attribute__ ((vector_size (16))) foo;
```

is invalid even if the size of the structure is the same as the size of the `int`.

visibility ("*visibility_type*")

This attribute affects the linkage of the declaration to which it is attached. The `visibility` attribute is described in Common Function Attributes.

weak

The `weak` attribute is described in Common Function Attributes.

noinit

Any data with the `noinit` attribute will not be initialized by the C runtime startup code, or the program loader. Not initializing data in this way can reduce program startup times.

This attribute is specific to ELF targets and relies on the linker script to place sections with the `.noinit` prefix in the right location.

persistent

Any data with the `persistent` attribute will not be initialized by the C runtime startup code, but will be initialized by the program loader. This enables the value of the variable to 'persist' between processor resets.

This attribute is specific to ELF targets and relies on the linker script to place the sections with the `.persistent` prefix in the right location. Specifically, some type of non-volatile, writeable memory is required.

objc_nullability (*nullability kind*) (Objective-C and Objective-C++ only)

This attribute applies to pointer variables only. It allows marking the pointer with one of four possible values describing the conditions under which the pointer might have a `nil` value. In most cases, the attribute is intended to be an internal representation for property and method nullability (specified by language keywords); it is not recommended to use it directly.

When *nullability kind* is `"unspecified"` or `0`, nothing is known about the conditions in which the pointer might be `nil`. Making this state specific serves to avoid false positives in diagnostics.

When *nullability kind* is `"nonnull"` or `1`, the pointer has no meaning if it is `nil` and thus the compiler is free to emit diagnostics if it can be determined that the value will be `nil`.

When *nullability kind* is `"nullable"` or `2`, the pointer might be `nil` and carry meaning as such.

When *nullability kind* is "resettable" or 3 (used only in the context of property attribute lists) this describes the case in which a property setter may take the value nil (which perhaps causes the property to be reset in some manner to a default) but for which the property getter will never validly return nil.

Next: ARC Variable Attributes, Up: Variable Attributes [Contents][Index]