ISO/IEC JTC 1/SC 22/WG14 N1279

2008-01-23 Douglas Walls

Attributes Commonly Found in Open Source Applications

Introduction

This paper introduces some attributes commonly found in use by open source applications compiled with gcc on linux. It expands on those introduced in WG14 papers N1129, N1264, and N1273 discussed at the WG14 meeting in Kona in October 2007.

alias

syntax:
`declaration_specifiers __attribute__((alias(`*`alternative_name`*` ))) `*`original_name`*

constraints:
`original_name` must be defined in the same translation unit.
`alias` attribute may only be applied to functions.

semantics:
Defines *`alternative_name`* to be an alias of *`original_name`*.

Example of usage:

```
void abc () { /* Function body */; }
void z () __attribute__ ((weak, alias ("abc")));
```

defines 'z' to be a weak alias for 'abc'.

weak

syntax:
`declaration_specifiers __attribute__((weak))` *`name`*

constraints:
attribute `weak` may be applied to functions or global variables.

semantics:
Defines the symbol as a weak symbol to the linker. The linker won't complain if a definition for the symbol is not found.

always_inline

syntax:
`declaration_specifiers __attribute__((always_inline))`

constraints:
attribute `always_inline` may only be applied to functions.

semantics:
This attribute inlines the function even if no optimization level was specified.

noinline

syntax:
`declaration_specifiers __attribute__((noinline))`

constraints:
attribute `noinline` may only be applied to functions.
The attribute cannot be applied to a type definition

semantics:
The `noinline` attribute prevents a function from being considered for inlining by the compiler.

constructor

syntax:
`void func_name() __attribute__((constructor))`

constraints:
`func_name` must be defined as a void function.
The attribute can only be applied to a function.

semantics:
A function marked with the `constructor` attribute is called just before entering the function `main`.

destructor

syntax:
`void func_name() __attribute__((destructor))`

constraints:
`func_name` must be defined as a void function.
The attribute can only be applied to a function.

semantics:
A function marked with the `destructor` attribute is called just after main has returned or `exit()` has been called.

pure

As discussed in WG14 N1273

packed

As discussed in WG14 N1273

const

syntax:
`declaration_specifiers __attribute__((const))`

constraints:
The attribute can only be applied to a function.

semantics:
The `const` attribute is a more strict class of the `pure` attribute.
Function is not allowed to read global memory.

malloc

syntax:
`declaration_specifiers __attribute__((malloc))`

constraints:
The attribute can only be applied to a function.

semantics:
This attribute is used to tell the compiler that a function may be treated as if it were the `malloc` function. The compiler assumes that calls to `malloc` result in pointers that cannot alias anything. The compiler can use this information to optimize code.

aligned

As discussed in WG14 N1273

visibility

syntax:
`declaration_specifiers __attribute__((visibility("visibility_type_keyword")))`
where visibility_type_keywords are: `default`, `hidden`, `internal` or `protected`

constraints:
A redeclaration of a symbol with a visibility attribute must specify a visibility attribute that is equal or more strict than previous

semantics:

In case when no visibility attribute present: The symbol linker scoping is unchanged from any prior declarations. If the symbol has no prior declaration, the symbol has the default linker scoping.

- `default` The symbol has global linker scoping. All references to the symbol will bind to the definition in the first dynamic module that defines the symbol. This linker scoping is the current linker scoping for extern symbols.
- `hidden` The symbol has hidden linker scoping. All references within a dynamic module will bind to a definition within that module. The symbol will not be visible outside of the module.
- `internal` The same as `hidden` with the added semantics that the symbol can never be called from outside the module, for example via a function pointer. Allow for code optimizations, like omitting the load of a PIC register.
- `protected` The symbol has protected linker scoping. All references to the symbol from within the dynamic module being linked will bind to the symbol defined within the module. Outside of the module, the symbol appears as though it were global.