

# 5

## Using Compiler Pragmas

---

This chapter documents High C/C++ compiler pragmas and describes how to use them. It contains the following sections:

§5.1: *Setting Pragmas*

§5.2: *Pragma Reference*

§5.3: *Customizing Your Program with Pragmas*

---

### 5.1 Setting Pragmas

Compiler pragmas control toggle settings, linkage to external functions, data storage, memory models, inclusion and listing of source text, and other features.

Each pragma has a default value. You can supply a profile to *override* the defaults and thereby customize the compiler to a local environment (see §2.5.4: *Using Profiles*). You can *change* the defaults of some pragmas by modifying the driver configuration file. See Appendix B: *Configuring the Driver* for details.

Compiler pragmas take one of the following general forms in your source:

<i>ANSI C Standard</i>	<b>#pragma</b> <i>Pragma_name</i> /* or */
<i>syntax</i>	<b>#pragma</b> <i>Pragma_name</i> ( <i>Pragma_parameters</i> )
<i>MetaWare syntax</i>	<b>pragma</b> <i>Pragma_name</i> ; /* or */
	<b>pragma</b> <i>Pragma_name</i> ( <i>Pragma_parameters</i> );

where *Pragma\_parameters* is a list of constant expressions separated by commas. The number and types of the expressions depend on the particular pragma. A pragma can appear anywhere a statement or declaration can appear. Pragma names are not case sensitive.

You can also specify pragmas on the command line with option **-Hpragma**:

```
-Hpragma=Pragma_name
-Hpragma=Pragma_name(Pragma_parameters)
```

For more information about setting pragmas on the command line, see option **-Hpragma** in §3.2: *Compiler Option Reference*.

---

**Note:** Although **#pragma** is the ANSI C Standard form, **pragma** already existed in High C prior to the ANSI Standard. Either form is acceptable.

An advantage of the MetaWare **pragma** syntax is that it can appear in macros; **#pragma** cannot. MetaWare's **pragma** statement must be terminated with a semicolon, and can appear only in a statement or declaration context.

---

## 5.2 Pragma Reference

**Alias**(*Internal\_name*, "*External\_name*") — **Assign an external name to an internal identifier**

Associates an external name with an internal name, for external or public purposes.

<i>Internal_name</i>	A variable or function identifier
<i>External_name</i>	A string constant denoting the alternate or external name; must be enclosed in double quotes

*External\_name* does not need to be a legal identifier, thus providing a way to define virtually any symbol (acceptable to the linker). See §5.3.3: *Aliasing External Names: Pragas Alias and Global\_aliasing\_convention*.

**Align\_members**(*n*) — **Set the maximum boundary for structure-member alignment**

Specifies the maximum byte boundary on which an (unpacked) structure member must be aligned. *n* is an integer constant denoting the maximum boundary. (The default alignment is the same as the default system

alignment, which varies by operating system.) Used without an argument, pragma `Align_members` restores the default.

See §10.5.1: *Aligning Structure Members: Pragma Align\_members*.

**`Alloc_text(seg_name, function[ , function... ])` — Group functions in specified text segment**

*OS/2 targets only* Allows you to group in the text segment *seg\_name* the functions you specify as the remaining arguments. You can specify any number of functions.

**`Called_infrequently(string[ , string ])` — Designate “called infrequently” all functions whose names contain a specified string**

If a function is declared within the scope of this pragma, and if the function’s name contains *string* as a substring, the compiler applies the calling-convention attribute `CALLED_INFREQUENTLY` to that function.

Whenever a function designated “called infrequently” is conditionally called within a loop construct, the compiler assumes that the basic block (node) containing the call is seldom executed. Based on that assumption, the compiler does the following:

- reorders the nodes so the node containing the function call is moved physically out of the loop
- sets a jump from the node containing the function call back into the loop

Pragma `Called_infrequently` must be specified *before* the declaration of the function(s).

---

**Note:** The configuration (`.cnf`) file *as distributed* automatically applies calling-convention attribute `CALLED_INFREQUENTLY` to functions with names containing any of these strings:

<code>fatal</code>	<code>fail</code>	<code>error</code>
<code>abort</code>	<code>trace</code>	

See Appendix B: *Configuring the Driver* for information about modifying the configuration file.

---

*Optimization benefits* `Called_infrequently` usually benefits error checking within a loop construct. When the portion of the `if` that contains the infrequently called function is moved out of the loop, the remaining code usually runs better,

because there are less jumps involved. Such jumps would normally bypass the portion of the **if** that contains the infrequently called function.

This optimization usually benefits code (like a benchmark) that has “hot spots.” “Hot spots” are portions of code, usually loops, that run most of the time the code is running. This optimization also benefits database code.

**Calling\_convention()**

**Calling\_convention(*Attributes*[ , \_DEFAULT ])** — **Allow High C/C++ modules to call functions not compiled with High C/C++**

Controls linkage to subroutines by allowing calls to modules compiled with other C or C++ compilers, or in other languages such as assembly, Pascal, PL/M, or FORTRAN, from modules compiled with High C/C++.

- *Attributes* is a bit pattern in which each bit specifies a particular calling convention attribute.
- `_DEFAULT` sets the default calling convention.

Pragma `Calling_convention()` cancels the effect of `Calling_convention(Attributes)`.

**Code()**

**Code("Section\_name")** — **Specify the code section name**

Specifies the name of the code section. *Section\_name*, a string constant, must be enclosed in double quotes.

Pragma `Code()` cancels pragma `Code("Section_name")`.

See §5.3.5: *Specifying an Alternate Code Section: Pragma Code*.

**Data()**

**Data([ *Sect\_type*, ] "Sect\_name")** — **Allocate named blocks for data storage**

Specifies the use of named blocks for data storage allocation.

- *Sect\_type*, an optional constant that specifies the type of the section, can be DATA, CODE, LIT, or COMMON; the default is DATA
- *Sect\_name*, a required string constant that denotes the name of the section, must be enclosed in double quotes.

Pragma `Data()` cancels `Data(Sect_type, "Sect_name")`.

Writable string literals used to initialize pointers are placed in the 32-bit data section `_DATA`, not in the data section specified with `Data`. To place a string in the data section specified with `Data`, initialize the string as an array:

```
char str[] = "This is a string.";
```

instead of as a pointer:

```
char *str = "This is a string.";
```

You cannot use pragma `Data` to specify the constant or block-storage section.

You can use pragma `Data` for laying out data in the specified section.

---

**Note:** If an uninitialized **extern** variable is declared inside the scope of pragma `Data`, the compiler issues a warning.

---

See §5.3.4: *Mapping Variables to Control Sections: Pragma Data*.

#### **Data\_seg()**

**Data\_seg(*Seg\_name*)** — **Put static and global variables in specified data segment**

*OS/2 targets only* Pragma `Data_seg` is a subset of pragma `Data`. It allows you to specify a data segment, but not segment type(s) or segment attribute(s).

**Ensure\_instantiation(*T1*, *T2*, ... *Tn*)** — **Ensure instantiation of specified template types**

*C++ only* Ensures instantiation of the specified template types, regardless of the setting of toggle `Auto_class_member_instantiation`. *T1*, *T2*, ... *Tn* are template type identifiers.

See Appendix E: *Manual Template Instantiation*.

**Exclude\_instantiation(*T1*, *T2*, ... *Tn*)** — **Prevent instantiation of specified template types**

*C++ only* Prevents instantiation of the specified template types, regardless of the setting of toggle `Auto_class_member_instantiation`. *T1*, *T2*, ... *Tn* are template type identifiers.

See Appendix E: *Manual Template Instantiation*.

**Exit\_expr(Expression [ ,Level ] )** — Evaluate the *Expression* argument at program termination

Evaluates the *Expression* argument when a program terminates. *Level* specifies the order in which expressions are evaluated. *Level* is a constant integral expression whose value can be between 0 (zero) and 255, inclusive. When you do not specify *Level*, it is set to 0 (zero), the default.

See §5.3.1: *Setting Module Initialization Priority: Pragma Initialization\_level*.

**Fini(func1, func2, ...)** — Place a call to each listed function in the program's .fini section

Places calls to functions *func1*, *func2*, ... in your program's .fini section.

The listed functions cannot take arguments. The function calls placed in the .fini section appear in the same order you give them in the pragma. You can call a function multiple times. There can be multiple occurrences of pragma **Fini**.

**Global\_aliasing\_convention()**

**Global\_aliasing\_convention("Form")** — Construct external object names for aliasing internal variable and function identifiers

Specifies automatic construction of external object names for internal identifiers. *Form* is a sequence of substring substitutions and designators; it must be enclosed in double quotes.

Pragma **Global\_aliasing\_convention()** cancels **Global\_aliasing\_convention("Form")**.

See §5.3.3: *Aliasing External Names: Pragma Alias and Global\_aliasing\_convention*

**Ident("Comment")** — Insert a comment in the generated object file

Specifies a string to be inserted into the generated object module in the .comment section. If you specify option **-s**, the string is added with the **.ident** assembler directive. The comment string must be enclosed in double quotes.

See §5.3.8: *Placing Comments in Object Code: Pragma Ident*.

**Init(*func1*, *func2*, ...)** — **Place a call to each listed function in the program's .init section**

Places calls to functions *func1*, *func2*, ... in your program's .init section.

The listed functions cannot take arguments. The function calls placed in the .init section appear in the same order you give them in the pragma. You can call a function multiple times. There can be multiple occurrences of pragma **Init**.

**Initialization\_level(*Level*)** — **Set initialization priority of modules**

Allows you to set the initialization priority for a module to *Level*, a constant integral expression whose value can be between 0 (zero) and 255, inclusive. The highest priority is 255; the default value of *Level* is 0 (zero).

See §5.3.1: *Setting Module Initialization Priority: Pragma Initialization\_level*.

**Literals("Section\_name")** — **Name a section for storing literals and static variables**

Specifies the name of the section to contain literals and read-only static variables declared with the **const** type qualifier.

The default literals section is .rdata.

*Section\_name*, a string constant denoting the section name, must be enclosed in double quotes.

See the descriptions of toggles *Const\_in\_code* and *Literals\_in\_code* in Chapter 4: *Using Compiler Toggles*. See also §5.3.7: *Specifying an Alternate Literals Section: Pragma Literals*.

**Loop\_factor(*Loop\_count*, *Size*)** — **Unroll loops**

Optimizes code by unrolling loops; that is, it replaces a loop with the code that makes up the loop body, replicated some number of loop iterations.

- *Loop\_count* is an integer constant specifying the maximum number of times a loop is unrolled; its default depends on the number of instructions in the loop.
- *Size* is an integer constant indicating the maximum size of loops to be unrolled.

See §G.2: *Selecting Loops for Unrolling: Pragma Loop\_factor* for more information.

---

**Note:** If a loop contains a function call or **switch** statement, it will not be unrolled.

---

**Off(Toggle\_name) — Turn Off a toggle**

Turns Off a compiler toggle. See §4.1: *Specifying Toggles*.

**Off\_inline(function\_name[, function\_name ...]) — Turn off inlining for specific functions**

Prevents calls to *function\_name* from being inlined when otherwise they would be inlined.

See §C.5: *Pragmas for Inlining* and §C.2.1: *How the Compiler Selects Routines for Inlining*.

**Offwarn(n1, n2, ...) — Disable specified warning messages**

Disables specified compiler warning messages (*n1*, *n2*, ...), so they are not displayed. (*n1*, *n2*, ...) are the numbers of the warning messages.

See §H.6.2: *Controlling Individual Warnings by Number*.

**On(Toggle\_name) — Turn On a toggle**

Turns On a compiler toggle. See §4.1: *Specifying Toggles*.

**On\_inline(function\_name[, function\_name ...]) — Turn on inlining for specific functions**

Directs the compiler to inline calls to *function\_name* even if the function does not otherwise meet the criteria for inlining.

See §C.5: *Pragmas for Inlining* and §C.2.1: *How the Compiler Selects Routines for Inlining*.

**Onwarn(n1, n2, ...) — Enable specified warning messages**

Enables specified compiler warning messages (*n1*, *n2*, ...), so they are displayed. (*n1*, *n2*, ...) are the numbers of the warning messages.

See §H.6.2: *Controlling Individual Warnings by Number*.



**Onwarn\_error(*n1*, *n2*, ...)** — Enable specified warning messages as compiler error messages

Enables specified compiler warning messages, so they are displayed, and upgrades them to be compiler errors. (*n1*, *n2*, ...) are the numbers of the warning messages.

See §H.6.2: *Controlling Individual Warnings by Number*.

**OS\_id(*n*)** — Generate .osinfo section

*OS/2 targets only* Generates a .osinfo section; *n* is the operating-system identification number. The default value of *n* is 4.

**Pack()****Pack(*n*)** — Control default alignment of data structures

Specifies the maximum boundary on which an (unpacked) structure member must be aligned. *n* is an integer constant denoting the boundary.

Pragma `Pack()` — without arguments — is equivalent to an “unpack” instruction: it sets the alignment to the default. Pragma `Pack()` does *not* revert to the previous setting given with a pragma `Pack(n)`.

See also §10.5.2: *Aligning Data Structures: Pragma Pack*.

---

**Note:** The default alignment of data structures in a program is the natural alignment specified by the appropriate application binary interface.

---



---

**Caution:** If you use pragma `Pack` indiscriminately, you can severely impair run-time performance by trying to access **struct** members that are not properly aligned.

---

**Page(*n*)** — Insert page ejects in a source-code listing

Causes page ejects to be inserted into a source-code listing. *n* is the number of form feeds (pages to eject). This pragma takes effect only when the `List` toggle is On.

See §A.2: *Customizing the Listing with Pragmas Page, Skip, and Title*.

**Pop(*Toggle\_name*)** — Reinstate prior status of a toggle

Reinstates a prior status of a compiler toggle.

See §4.1: *Specifying Toggles*.

**Pop\_align\_members** — **Restore alignment prior to earlier Push\_align\_members**

Cancels previous Push\_align\_members and restores prior alignment.

See §10.5.4: *Restoring Alignment: Pragma Pop\_align\_members*.

**Pop\_ignore(Pragma\_name, Pragma\_name ...)** — **Reinstate prior ignore status of listed pragmas**

Restores the compiler's previous state of awareness of any pragma listed as one of its arguments. (Once a pragma has been set, you can make the compiler ignore it or become “unaware” of it by listing it as an argument of pragma Push\_ignore.)

The compiler issues a warning if you “un-ignore” a pragma it is already aware of. The names of pragma arguments can be listed with or without quotes.

**Pop\_inline(function\_name[, function\_name ...])** — **Pop topmost stack entry for each specified function**

Pops the topmost stack entry for each function specified, and restores the previous On\_inline or Off\_inline state.

See §C.5: *Pragmas for Inlining* and §C.2.1: *How the Compiler Selects Routines for Inlining*.

**Pop\_small\_data()** — **Reinstate prior small-data specification**

Pops the small-data specification set by the last Push\_small\_data pragma, and reinstates the previous specification.

Refer to §11.5.4: *Small-Data Sections* for more information about using small-data sections.

**Popwarn** — **Cancel previous pragma Offwarn or Onwarn**

Cancels the previous Offwarn or Onwarn; these pragmas are “stacked.”

See §H.6.2: *Controlling Individual Warnings by Number*.

**Push\_align\_members(n)** — **Set maximum boundary for structure-member alignment and save state**

Specifies that *n* bytes is the maximum boundary for structure member alignment, then saves the state for a subsequent call to pragma Pop\_align\_members.

Used without an argument, pragma `Push_align_members` saves the current state for pop, then restores the default.

See §10.5.3: *Aligning Structures and Saving Maximum Alignment: Pragma `Push_align_members`*.

**`Push_ignore(Pragma_name, Pragma_name ...)` — Ignore pragmas listed**

This pragma causes the compiler to ignore (lose awareness of) any previously set pragma listed as one of its arguments. It also suppresses any warning message the compiler would otherwise emit about the pragma.

`Push_ignore` is useful mainly for ignoring pragmas not supported by High C/C++. The names of pragmas to ignore can be listed with or without quotes.

**`Push_small_data([non_const_section, ]n[; [const_section, ]m])` — Assign data to small-data section**

Establishes a small-data specification for subsequently declared non-`auto` variables. Pragma `Push_small_data` specifies two things:

- the small-data sections to which the compiler will assign variables declared in a given block of code
- the maximum size of variables in these small-data sections

The syntax allows you to set independent specifications for `const` and non-`const` data.

Pragma `Push_small_data` “pushes” the specification for small data processing, and pragma `Pop_small_data` pops the current specification, reinstating the previous state of small-data processing.

*non\_const\_section* The *non\_const\_section* argument is optional. It can take as values either `sdata` or `sdata0`. If you do not specify an argument, the default is `sdata`.

`sdata0` Specifies small-data sections `.PPC.EMB.sdata0` and `.PPC.EMB.sbss0`.

`sdata` Specifies small-data sections `.sdata` and `.sbss`.

*const\_section* The *const\_section* argument is optional, and if it is used takes a value `sdata2`.

`sdata2` Specifies sections `.sdata2` and `.sbss2`.

For information about the characteristics of small-data sections, see §11.5.4.2: *More About Small-Data Sections*.

*n* and *m* *n* is an integer that represents the maximum size of non-**const** data in the `sdata` section. *m* is an integer that represents the maximum size of **const** data in the `sdata` section.

See §11.5.4: *Small-Data Sections* for more information on small data processing.

**skip(*n*)** — **Insert blank lines in a source-code listing**

Causes blank lines to be inserted into a source-code listing. *n* is the number of blank lines to insert. This pragma takes effect only when toggle `List` is `On`.

See §A.2: *Customizing the Listing with Pragas Page, Skip, and Title*.

**startup\_expr(*Expression* [ , *Level* ] )** — **Evaluate the *Expression* argument at start-up**

Evaluates the *Expression* argument at run-time start-up, before `main()` is invoked. If you use pragma `Startup_expr`, the compiler generates an initialization routine when it generates the object file.

*Level* specifies the order in which expressions are evaluated. *Level* is a constant integral expression whose value can be between 0 (zero) and 255, inclusive. When you do not specify *Level*, it is set to 0 (zero), the default.

See §5.3.2: *Evaluating Expressions at Run Time: Pragas Startup\_expr and Exit\_expr*.

**static\_segment("Section\_name")** — **Name a section for storing initialized static variables**

Specifies the section in which initialized static variables and writable string constants are stored.

*Section\_name*, a string constant denoting the name of the section, must be enclosed in double quotes. This pragma is useful for laying out data in the specified section.

See §5.3.6: *Specifying an Alternate Data Section: Pragma Static\_segment*.

**Title("Listing\_title")** — **Put a title at the top of each page of the source listing**

Causes a title to appear at the top of each page of the source listing. This pragma takes effect only when toggle `List` is On. *Listing\_title*, a string constant, must be enclosed in double quotes.

See §A.2: *Customizing the Listing with Pragmas Page, Skip, and Title*.

**Warning\_level(*n*)** — **Set the level of warning diagnostics**

Sets the level of warning diagnostics to *n*, where *n* is a decimal integer between 0 (zero) and 4, inclusive.

See §H.6: *Controlling Warning Messages*.

**Weak(*x*, *y*)**

**Weak *x* = *y*** — **Designate a weak global symbol and assign it a value**

Designates identifier *x* as a weak global symbol whose value is the same as identifier *y*. *y* must be a global symbol defined in the current module with *x* otherwise undefined.

---

## 5.3 Customizing Your Program with Pragmas

This section provides more information about using compiler pragmas to perform the following customizations:

Customization	Pragma(s)	Page
Set module initialization priority	<code>Initialization_level</code>	108
Evaluate expressions at run time	<code>Startup_expr</code> and <code>Exit_expr</code>	108
Alias external names	<code>Alias</code> and <code>Global_aliasing_convention</code>	109
Map variables to control sections	<code>Data</code>	113
Specify an alternate code section	<code>Code</code>	115
Specify an alternate data section	<code>Static_segment</code>	116
Specify an alternate literals section	<code>Literals</code>	117
Place comments in object code	<code>Ident</code>	117

---

### 5.3.1 Setting Module Initialization Priority: `Pragma Initialization_level`

`Pragma Initialization_level( Level )` sets the initialization level for the module to the specified value.

The program start-up code calls each of the initialization routines in order of decreasing initialization level, where *Level* is a constant integral expression whose value can be between 0 (zero) and 255, inclusive.

If *Level* is not specified, the compiler assumes a value of 0 (zero — the lowest priority).

See §D.3.2: *Setting Module Initialization Priority: Pragma Initialization\_level* for an example of using `pragma Initialization_level` for embedded PowerPC applications.

---

**Note:** Initialization priorities in the range 120 through 130 are reserved for MetaWare internal use.

---

---

### 5.3.2 Evaluating Expressions at Run Time: `Pragas startup_expr` and `Exit_expr`

You can specify expressions to be evaluated at program start-up or termination with pragmas `Startup_expr` and `Exit_expr`.

Use these initialization pragmas if you want some code to be executed before `main()` is called or after `main()` exits. Such code could perform initialization of data structures and could free memory to undo the initialization process.

If data items must be initialized at run time, or expressions must be evaluated at start-up or termination, the compiler generates an initialization routine when it generates an object module.

#### 5.3.2.1 `Pragma startup_expr`

`Pragma Startup_expr( Expression, Level )` evaluates the expression argument at run time before `main()` is invoked. The order in which

expressions are evaluated is specified by the integer constant expression *Level*.

If *Level* is not specified, the initialization level of the entire compilation unit is used, or the default value 0 (zero) is used. A compilation unit's initialization level is set with `pragma Initialization_level`, which is 0 (zero) by default. (See §5.3.1: *Setting Module Initialization Priority: Pragma Initialization\_level*.)

Typically, the *Expression* argument is a function call:

```
void initialize_this_module() { ... }
#pragma Startup_expr( initialize_this_module(), 10 )
```

With High C/C++, the full power of the expression is available, and can be used to disambiguate between two functions or to execute other non-function code at start-up:

```
void init(int), init(double); int x, y;

/* C++ code here */

#pragma Startup_expr( (init(3), init(2.71),
                      x = 4, y = x*x), 10)
```

To specify more than one expression, enclose them in parentheses — or use multiple pragmas.

### 5.3.2.2 Pragma Exit\_expr

`Pragma Exit_expr(Expression, Level)` evaluates the *Expression* argument when the program terminates, at the time `atexit()` gains control.

The integer constant expression *Level* specifies the order in which expressions are evaluated. If *Level* is not specified, the default value 0 (zero) is used.

---

## 5.3.3 Aliasing External Names: Pragas Alias and Global\_aliasing\_convention

An *aliasing convention* is a method for aliasing internal names to external names. The names of variables and functions that are communicated across

module boundaries are normally made global in the resultant object module. In large programs there can be hundreds or even thousands of such names, so name conflicts are likely to occur.

Unfortunately, C compilers and most linkers do not provide for a structured name space — for named packages of resources, for example. Thus the well-chosen *internal* names in a program might not be usable as *external* names (those known to the linker). Some method of aliasing internal names to external names is needed, and High C/C++ provides it.

*Aliasing versus  
perverting names*

It is important to be able to alias such names to avoid conflicts in the linker's external symbol dictionary, rather than being forced to pervert the internal names themselves. The internal names must be well chosen “containers of meaning,” for program maintainability.

The High C/C++ compiler provides two pragmas for aliasing names: `Alias` and `Global_aliasing_convention`.

### 5.3.3.1 Pragma Alias

Pragma `Alias` specifies, for a given internal name, another name for external or public purposes. This external name is the alternate name that appears in the object module. This is the syntax of the `Alias` pragma:

```
#pragma Alias(Internal_name, "External_name")
```

where the arguments have the following meanings:

<i>Internal_name</i>	A variable or function identifier
<i>External_name</i>	A string constant denoting the alternate or external name; must be enclosed in double quotes

*External\_name* does not need to be a legal identifier, thus providing a way to define virtually any symbol (acceptable to the linker).

The `Alias` pragma must appear *in the scope* of the declaration of the internal name. For example:

```
void Initialize();
#pragma Alias(Initialize, "x_initialize")
/* The function Initialize is referenced in the */
/* object-module symbol table as "x_initialize" */
int BA;
```



```
#pragma Alias(BA, "Ph.D.")
/* "BA" is referenced in the */
/* symbol table as "Ph.D." */
```

### 5.3.3.2 Pragma Global\_aliasing\_convention

Pragma `Global_aliasing_convention` specifies the aliasing of external names not already aliased by the `Alias` pragma. This is the syntax of the `Global_aliasing_convention` pragma:

```
#pragma Global_aliasing_convention("Form")
```

where *Form* is a string that specifies the way each external name *Ext\_name* is derived from its internal name *Int\_name*.

*Ext\_name* is the actual text of *Form* except for the following substitutions for substrings within *Form*:

- `%r` (resource) denotes *Int\_name*
- `%%` denotes `%`
- `%C`, `%c`, and `%a` affect the casing of the external name

For example, if you specify

```
#pragma Global_aliasing_convention("_%r")
```

a function named `fred` will be known as `_fred` in the object module. (See §: *Arguments to Global\_aliasing\_convention* for additional information.)

By default, the `Global_aliasing_convention` setting is `"_%r"`; external names match internal names, but with an `"_"` prefix.

*Changing the  
default convention*

You can change the default aliasing convention by modifying the driver configuration (`.cnf`) file to supply a value for the `-Hgac` option on the `ARGS=` line. If you set the `Global_aliasing_convention` by configuring the driver, all external names (explicitly declared or otherwise) are affected.

---

**Note:** Changing the default aliasing convention has implications for the use of existing libraries.

---

*Setting the  
convention in code*

If you set the aliasing convention *in the source code or in a profile*, only subsequently declared names are affected. Functions that are used but not

declared are *not* affected by the convention; their external names have the “\_” prefix.

*Overriding an aliasing convention* You can override the `Global_aliasing_convention` setting for a particular name by specifying `pragma Alias`; that is, `pragma Alias` takes precedence.

*Turning off the pragma* You can turn off `pragma Global_aliasing_convention` by omitting the argument:

```
#pragma Global_aliasing_convention()
```

This restores the original default (“\_%r”).

### Arguments to `Global_aliasing_convention`

*Substring designators* The `%r` argument can be followed by substring designators of the form `:Start:Len` to indicate the substring starting at character *Start* and going for *Len* characters.

The character positions are numbered starting at 1. If *Start* is negative, it denotes the starting position from the *right end* of the string where the last character is numbered -1. If *:Len* is omitted, it means “to the end of the string.”

These substitutions within *Form* affect the casing of the external name:

- `%C` means to shift subsequent letters in the external name to uppercase.
- `%c` means to shift them to lowercase.
- `%a` means not to convert subsequent letters at all, but to use them as is, each with its given case. This mode holds at the beginning of the specification.

**Examples** This example changes all global names (`%r`) to uppercase (`%C`) and truncates them to eight characters (`:1:8`):

```
#pragma Global_aliasing_convention("%C%r:1:8")
```

This same example would appear in the configuration file as follows:

```
ARGS= -Hgac="%C%r:1:8"
```

See Chapter 3: *Using Compiler Options* and Appendix B: *Configuring the Driver* for details about `-Hgac` and the configuration file.

### 5.3.4 Mapping Variables to Control Sections: Pragma Data

To define your own control sections for mapping static or exported variables, use `pragma Data`. `Pragma Data` has two forms, which must be used in pairs to bracket the relevant source code, as follows:

```
#pragma Data ( [sect_type, ] "sect_name" )
/* Static and global variable declarations
   go here */
#pragma Data /* Turns off the prior pragma Data */
```

where the arguments have the following meanings:

<i>sect_type</i>	type of the block; DATA, CODE, LIT, or COMMON
<i>sect_name</i>	a constant string expression that denotes the name of the section

If you do not specify a *sect\_type*, the section is defined as a writable data section. The section types and section names can appear in any order.

*CODE type* The `CODE` type denotes an executable code section. Variables mapped into code sections must be qualified **const** (that is, must be read-only). If a variable is not qualified **const**, the compiler issues a compile-time warning and applies **const** to the variable's type.

Use the `CODE` type to map read-only variables directly into executable code, immediately before or after a function definition. For example:

```
#pragma Data (CODE, ".text")
static const char func[] = "func starts here";
#pragma Data /* No parameters here */

static void func() {...}
```

---

**Note:** It is not possible to embed variables within the body of a function.

If a `Data` pragma appears within a function definition, subsequently defined local variables within the scope of the pragma will be mapped into the indicated section.

If the section name corresponds to that of the function (in this case, if the name corresponds to `.text`), the variables will be placed immediately before the start of the function.

---

*COMMON type* The `COMMON` type defines a common block. Technically, a common block is not a control section, but is a named global symbol that has an associated length.

When you map a variable into a common block, the variable must be **static** and it must not be initialized. Each variable is mapped to consecutive offsets within the section or common block, subject to boundary alignment.

*LIT type* The `LIT` type designates a read-only section. Variables mapped into read-only sections must be qualified **const**. If the variables are not qualified **const**, you will get a compile-time warning.

*DATA type* The `DATA` type designates a writable data section.

*Closing pragma* The closing pragma `Data` has no parameters. For example:

```
Data
#pragma Data (LIT, "RODATA")
const int x = 100;
const char string [] = "some string";
#pragma Data /* No parameters here */
```

In this example, the variable `x` and the string `some string` are placed in a read-only data section named `RODATA`.

Each pragma `Data` must be terminated or “turned off,” as illustrated, *in the same scope* in which it is turned on.

*Data pragmas cannot be nested within one function* The compiler issues a warning if a `Data` pragma is specified when a prior `Data` pragma is still active (in which case the subsequent pragma applies), or if a `Data` pragma is active at the end of a function declaration or at the end of a compilation unit. Thus `Data` pragmas cannot be nested within a

single function, though they can be nested if they apply to the local variables of distinct functions.

Pragma Data does *not* apply to static variables declared within embedded function definitions. However, the Data pragma can be specified within a function definition to apply to locally declared static variables; any pending pragma Data will be resumed at the end of the function definition.

---

### 5.3.5 Specifying an Alternate Code Section: Pragma Code

*Code segmentation* means grouping code into named control sections that are manipulated by a linker. The concept of segmenting generated code is useful whether or not a program is modularized and separately compiled.

The concept applies only to top-level or “level-one” functions: level-two and greater functions (that is, *nested functions*) must be in the same section as their containing level-one function. Code for one function cannot be split across sections.

---

**Note:** Nested functions are a High C/C++ extension; ANSI Standard C permits only level-one functions.

---

By default, compiler-generated code is placed in the `.text` section. The default code section name can be overridden by using the `Code` pragma. The `Code` pragma explicitly specifies the name of the section in which the code of the subsequent functions is to reside.

Pragma Code has two forms that must be used in pairs to bracket the relevant source code:

```
#pragma Code(section_name)
...           /* Affected source code goes here */
#pragma Code
```

where *section\_name* is a constant string expression that denotes the name of the section.

*Closing pragma Code* The closing pragma `Code` has no parameters. You use it to terminate the effect of the previously specified `Code` pragma so that subsequent code is placed in the default section.

*Code pragmas cannot be nested* Pragma Code applies only at the outer level of the program, so the pairs cannot be nested. For example:

```
/* Assume default section ".text" */
void A(){
    ...
} /* A's code put in section ".text" */
#pragma Code("SEC1")

void B(){
    ...
} /* B's code put in section "SEC1" */
#pragma Code /* Ends pragma Code("SEC1") */

void C(){
    ...
} /* C's code put in section ".text" */
...
```

---

**Note:** The Code pragma that is active over the body of a function applies to the function's code. Any Code pragma active at a prior declaration of the function is irrelevant.

---



---

### 5.3.6 Specifying an Alternate Data Section: `Pragma Static_segment`

Associated with every compilation unit is a default data section into which initialized static variables are mapped (unless overridden with a Data pragma specification). This section is named `.data`.

The name of the default data section can be altered with the `Static_segment` pragma:

```
#pragma Static_segment("section_name")
```

where `section_name` is a string constant that names the section.

Static variables declared with the **const** type qualifier are ordinarily placed in the default literals section. See §10.6: *How the Compiler Maps Variables to Storage*.

---

### 5.3.7 Specifying an Alternate Literals Section: `Pragma Literals`

The *literals* section is where read-only static variables and read-only string constants are ordinarily placed.

*Read-only variables* A variable is designated as read-only if it is declared with the **const** type qualifier.

*Read-only string constants* A string constant is considered read-only if it is cast to a (**const char \***) type (implicitly or explicitly), or if toggle `Read_only_strings` is On. A string constant is *implicitly* cast by assigning it to a pointer of type (**const char \***) or passing it as a parameter that is so typed.

The default name for the literals section is `.rdata`. Use the `Literals` pragma to assign an alternate section name:

```
#pragma Literals("section_name")
```

where *section\_name* is a string constant that names the section.

See §10.6: *How the Compiler Maps Variables to Storage* for more information.

---

### 5.3.8 Placing Comments in Object Code: `Pragma Ident`

Use `pragma Ident` to place notes in a comment section within a generated object module. The `Ident` pragma has this form:

```
#pragma Ident("comment string")
```

You can use this pragma multiple times. The name of the comment section is typically `.comment`. When the compiler generates an assembly file, the pragma translates to an **.ident** statement.

The compiler normally emits a comment string that identifies the version of the compiler and the selected command-line options used to produce the object file, even when you do not use the `Ident` pragma. This string can be suppressed by turning Off the `Command_line_ident` toggle. See Chapter 4: *Using Compiler Toggles*.

These are some example uses of the `Ident` pragma:

```
#pragma Ident("Source version 3.4.1\n")
#pragma Ident("Date is " __DATE__)
```

