The C/C++ compiler supports the C/C++ language standard that was developed by a committee of the American National Standards Institute (ANSI/ISO) to standardize the C programming language.

The C++ language supported by the MSP430 is defined by the ANSI/ISO/IEC 14882-1998 standard with certain exceptions.
5.1 Characteristics of MSP430 C

The compiler supports the C language as defined by ISO 9899, which is equivalent to American National Standard for Information Systems-Programming Language C X3.159-1989 (C89). The compiler does not support C99.

Unsupported features of the C library are:

- The run-time library has minimal support for wide and multi-byte characters. The type wchar_t is implemented as int. The wide character set is equivalent to the set of values of type char. The library includes the header files <wchar.h> and <wctype.h>, but does not include all the functions specified in the standard. So-called multi-byte characters are limited to single characters. There are no shift states. The mapping between multi-byte characters and wide characters is simple equivalence; that is, each wide character maps to and from exactly a single multi-byte character having the same value.
- The run-time library includes the header file <locale.h>, but with a minimal implementation. The only supported locale is the C locale. That is, library behavior that is specified to vary by locale is hard-coded to the behavior of the C locale, and attempting to install a different locale by way of a call to setlocale() will return NULL.

5.2 Characteristics of MSP430 C++

The MSP430 compiler supports C++ as defined in the ANSI/ISO/IEC 14882:1998 standard, including these features:

- Complete C++ standard library support, with exceptions noted below.
- Templates
- Exceptions, which are enabled with the --exceptions option; see Section 5.5.
- Run-time type information (RTTI), which can be enabled with the --rtti compiler option.

The exceptions to the standard are as follows:

- The <complex> header and its functions are not included in the library.
- The library supports wide chars, in that template functions and classes that are defined for char are also available for wide char. For example, wide char stream classes wios, wiostream, wstreambuf and so on (corresponding to char classes ios, iostream, streambuf) are implemented. However, there is no low-level file I/O for wide chars. Also, the C library interface to wide char support (through the C++ headers <cwchar> and <cwctype>) is limited as described above in the C library.
- If the definition of an inline function contains a static variable, and it appears in multiple compilation units (usually because it’s a member function of a class defined in a header file), the compiler generates multiple copies of the static variable rather than resolving them to a single definition. The compiler emits a warning (#1369) in such cases.
- The reinterpret_cast type does not allow casting a pointer-to-member of one class to a pointer-to-member of another class if the classes are unrelated.
- Two-phase name binding in templates, as described in [tesp.res] and [temp.dep] of the standard, is not implemented.
- Template parameters are not implemented.
- The export keyword for templates is not implemented.
- A typedef of a function type cannot include member function cv-qualifiers.
- A partial specialization of a class member template cannot be added outside of the class definition.
### 5.3 Data Types

Table 5-1 lists the size, representation, and range of each scalar data type for the MSP430 compiler. Many of the range values are available as standard macros in the header file limits.h.

#### Table 5-1. MSP430 C/C++ Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Representation</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, signed char</td>
<td>8 bits</td>
<td>ASCII</td>
<td>-128</td>
<td>-127</td>
</tr>
<tr>
<td>unsigned char, bool</td>
<td>8 bits</td>
<td>ASCII</td>
<td>0</td>
<td>255</td>
</tr>
<tr>
<td>short, signed short</td>
<td>16 bits</td>
<td>2s complement</td>
<td>-32 768</td>
<td>32 767</td>
</tr>
<tr>
<td>unsigned short, wchar_t</td>
<td>16 bits</td>
<td>Binary</td>
<td>0</td>
<td>65 535</td>
</tr>
<tr>
<td>int, signed int</td>
<td>16 bits</td>
<td>2s complement</td>
<td>-32 768</td>
<td>32 767</td>
</tr>
<tr>
<td>unsigned int</td>
<td>16 bits</td>
<td>Binary</td>
<td>0</td>
<td>65 535</td>
</tr>
<tr>
<td>long, signed long</td>
<td>32 bits</td>
<td>2s complement</td>
<td>-2 147 483 648</td>
<td>2 147 483 647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32 bits</td>
<td>Binary</td>
<td>0</td>
<td>4 294 967 295</td>
</tr>
<tr>
<td>enum</td>
<td>16 bits</td>
<td>2s complement</td>
<td>-32 768</td>
<td>32 767</td>
</tr>
<tr>
<td>float</td>
<td>32 bits</td>
<td>IEEE 32-bit</td>
<td>1.175 495e-38(^{(1)})</td>
<td>3.40 282 35e+38</td>
</tr>
<tr>
<td>double</td>
<td>32 bits</td>
<td>IEEE 32-bit</td>
<td>1.175 495e-38(^{(1)})</td>
<td>3.40 282 35e+38</td>
</tr>
<tr>
<td>long double</td>
<td>32 bits</td>
<td>IEEE 32-bit</td>
<td>1.175 495e-308(^{(1)})</td>
<td>3.40 282 35e+38</td>
</tr>
<tr>
<td>pointers, references, pointer to data members</td>
<td>16 bits</td>
<td>Binary</td>
<td>0</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>MSP430X large-data model pointers, references, pointer to data members (^{(2)})</td>
<td>20 bits</td>
<td>Binary</td>
<td>0</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>MSP430 function pointers</td>
<td>16 bits</td>
<td>Binary</td>
<td>0</td>
<td>0xFFFF</td>
</tr>
<tr>
<td>MSP430X function pointers (^{(3)})</td>
<td>20 bits</td>
<td>Binary</td>
<td>0</td>
<td>0xFFFFFFFF</td>
</tr>
</tbody>
</table>

\(^{(1)}\) Figures are minimum precision.

\(^{(2)}\) MSP430X large-data model is specified by --silicon_version=mspx --large_memory_model

\(^{(3)}\) MSP430X devices are specified by --silicon_version=mspx
Keywords

5.4 Keywords

The MSP430 C/C++ compiler supports the standard const, register, and volatile keywords. In addition, the MSP430 C/C++ compiler extends the C/C++ language through the support of the interrupt keyword.

5.4.1 The const Keyword

The C/C++ compiler supports the ANSI/ISO standard keyword `const`. This keyword gives you greater optimization and control over allocation of storage for certain data objects. You can apply the const qualifier to the definition of any variable or array to ensure that its value is not altered.

If you define an object as const, the .const section allocates storage for the object. The const data storage allocation rule has two exceptions:

- If the keyword volatile is also specified in the definition of an object (for example, volatile const int x). Volatile keywords are assumed to be allocated to RAM. (The program does not modify a const volatile object, but something external to the program might.)
- If the object has automatic storage (function scope).

In both cases, the storage for the object is the same as if the const keyword were not used.

The placement of the const keyword within a definition is important. For example, the first statement below defines a constant pointer p to a variable int. The second statement defines a variable pointer q to a constant int:

```
int * const p = &x;
const int * q = &x;
```

Using the const keyword, you can define large constant tables and allocate them into system ROM. For example, to allocate a ROM table, you could use the following definition:

```
const int digits[] = {0,1,2,3,4,5,6,7,8,9};
```

5.4.2 The interrupt Keyword

The compiler extends the C/C++ language by adding the interrupt keyword, which specifies that a function is treated as an interrupt function.

Functions that handle interrupts follow special register-saving rules and a special return sequence. When C/C++ code is interrupted, the interrupt routine must preserve the contents of all machine registers that are used by the routine or by any function called by the routine. When you use the interrupt keyword with the definition of the function, the compiler generates register saves based on the rules for interrupt functions and the special return sequence for interrupts.

You can only use the interrupt keyword with a function that is defined to return void and that has no parameters. The body of the interrupt function can have local variables and is free to use the stack or global variables. For example:

```c
interrupt void int_handler()
{
    unsigned int flags;
    ...
}
```

The name c_int00 is the C/C++ entry point. This name is reserved for the system reset interrupt. This special interrupt routine initializes the system and calls the function main. Because it has no caller, c_int00 does not save any registers.

Use the alternate keyword, __interrupt, if you are writing code for strict ANSI/ISO mode (using the --strict_ansi compiler option).

---

HWI Objects and the interrupt Keyword

**Note:** The interrupt keyword must not be used when BIOS HWI objects are used in conjunction with C functions. The HWI_enter/HWI_exit macros and the HWI dispatcher contain this functionality, and the use of the C modifier can cause catastrophic results.
5.4.3 The restrict Keyword

To help the compiler determine memory dependencies, you can qualify a pointer, reference, or array with the restrict keyword. The restrict keyword is a type qualifier that can be applied to pointers, references, and arrays. Its use represents a guarantee by you, the programmer, that within the scope of the pointer declaration the object pointed to can be accessed only by that pointer. Any violation of this guarantee renders the program undefined. This practice helps the compiler optimize certain sections of code because aliasing information can be more easily determined.

In Example 5-1, the restrict keyword is used to tell the compiler that the function func1 is never called with the pointers a and b pointing to objects that overlap in memory. You are promising that accesses through a and b will never conflict; therefore, a write through one pointer cannot affect a read from any other pointers. The precise semantics of the restrict keyword are described in the 1999 version of the ANSI/ISO C Standard.

**Example 5-1. Use of the restrict Type Qualifier With Pointers**

```c
void func1(int * restrict a, int * restrict b)
{
    /* func1's code here */
}
```

Example 5-2 illustrates using the restrict keyword when passing arrays to a function. Here, the arrays c and d should not overlap, nor should c and d point to the same array.

**Example 5-2. Use of the restrict Type Qualifier With Arrays**

```c
void func2(int c[restrict], int d[restrict])
{
    int i;
    for(i = 0; i < 64; i++)
    {
        c[i] += d[i];
        d[i] += 1;
    }
}
```

5.4.4 The volatile Keyword

The compiler analyzes data flow to avoid memory accesses whenever possible. If you have code that depends on memory accesses exactly as written in the C/C++ code, you must use the volatile keyword to identify these accesses. A variable qualified with a volatile keyword is allocated to an uninitialized section (as opposed to a register). The compiler does not optimize out any references to volatile variables.

In the following example, the loop waits for a location to be read as 0xFF:

```c
unsigned int *ctrl;
while (*ctrl != 0xFF);
```

In this example, *ctrl is a loop-invariant expression, so the loop is optimized down to a single-memory read. To correct this, define *ctrl as:

```c
volatile unsigned int *ctrl;
```

Here the *ctrl pointer is intended to reference a hardware location, such as an interrupt flag.
5.5 C++ Exception Handling

The compiler supports all the C++ exception handling features as defined by the ANSI/ISO 14882 C++ Standard. More details are discussed in *The C++ Programming Language, Third Edition* by Bjarne Stroustrup.

The compiler --exceptions option enables exception handling. The compiler’s default is no exception handling support.

For exceptions to work correctly, all C++ files in the application must be compiled with the --exceptions option, regardless of whether exceptions occur in a particular file. Mixing exception-enabled object files and libraries with object files and libraries that do not have exceptions enabled can lead to undefined behavior. Also, when using --exceptions, you need to link with run-time-support libraries whose name contains _eh. These libraries contain functions that implement exception handling.

Using --exceptions causes code size to increase.

See Section 7.1 for details on the run-time libraries.

5.6 Register Variables and Parameters

The C/C++ compiler treats register variables (variables defined with the register keyword) differently, depending on whether you use the --opt_level (-O) option.

- **Compiling with optimization**
  The compiler ignores any register definitions and allocates registers to variables and temporary values by using an algorithm that makes the most efficient use of registers.

- **Compiling without optimization**
  If you use the register keyword, you can suggest variables as candidates for allocation into registers. The compiler uses the same set of registers for allocating temporary expression results as it uses for allocating register variables.

The compiler attempts to honor all register definitions. If the compiler runs out of appropriate registers, it frees a register by moving its contents to memory. If you define too many objects as register variables, you limit the number of registers the compiler has for temporary expression results. This limit causes excessive movement of register contents to memory.

Any object with a scalar type (integral, floating point, or pointer) can be defined as a register variable. The register designator is ignored for objects of other types, such as arrays.

The register storage class is meaningful for parameters as well as local variables. Normally, in a function, some of the parameters are copied to a location on the stack where they are referenced during the function body. The compiler copies a register parameter to a register instead of the stack, which speeds access to the parameter within the function.

For more information about register conventions, see Section 6.3.
5.7 The asm Statement

The C/C++ compiler can embed assembly language instructions or directives directly into the assembly language output of the compiler. This capability is an extension to the C/C++ language—the `asm` statement. The `asm` (or `__asm`) statement provides access to hardware features that C/C++ cannot provide. The `asm` statement is syntactically like a call to a function named `asm`, with one string constant argument:

```
asm(" assembler text ");
```

The compiler copies the argument string directly into your output file. The assembler text must be enclosed in double quotes. All the usual character string escape codes retain their definitions. For example, you can insert a `.byte` directive that contains quotes as follows:

```
asm("STR: .byte \"abc\"");
```

The inserted code must be a legal assembly language statement. Like all assembly language statements, the line of code inside the quotes must begin with a label, a blank, a tab, or a comment (asterisk or semicolon). The compiler performs no checking on the string; if there is an error, the assembler detects it. For more information about the assembly language statements, see the MSP430 Assembly Language Tools User's Guide.

The `asm` statements do not follow the syntactic restrictions of normal C/C++ statements. Each can appear as a statement or a declaration, even outside of blocks. This is useful for inserting directives at the very beginning of a compiled module.

Use the alternate statement `__asm("assembler text")` if you are writing code for strict ANSI/ISO C mode (using the `--strict_ansi` option).

---

**Note:** Avoid Disrupting the C/C++ Environment With asm Statements

Be careful not to disrupt the C/C++ environment with asm statements. The compiler does not check the inserted instructions. Inserting jumps and labels into C/C++ code can cause unpredictable results in variables manipulated in or around the inserted code. Directives that change sections or otherwise affect the assembly environment can also be troublesome.

Be especially careful when you use optimization with asm statements. Although the compiler cannot remove asm statements, it can significantly rearrange the code order near them and cause undesired results.

---
5.8 Pragma Directives

Pragma directives tell the compiler how to treat a certain function, object, or section of code. The MSP430 C/C++ compiler supports the following pragmas:

- BIS_IE1_INTERRUPT
- CODE_SECTION
- DATA_ALIGN
- DATA_SECTION
- FUNC_CANNOT_INLINE
- FUNC_EXT_CALLED
- FUNC_IS_PURE
- FUNC_NEVER_RETURNS
- FUNC_NO_GLOBAL_ASG
- FUNC_NO_IND_ASG
- INTERRUPT
- NO_HOOKS

For the pragmas that apply to functions or symbols, the syntax for the pragmas differs between C and C++. In C, you must supply the name of the object or function to which you are applying the pragma as the first argument. In C++, the name is omitted; the pragma applies to the declaration of the object or function that follows it.

5.8.1 The BIS_IE1_INTERRUPT

The BIS_IE1_INTERRUPT pragma treats the named function as an interrupt routine. Additionally, the compiler generates a BIS operation on the IE1 special function register upon function exit. The mask value, which must be an 8-bit constant literal, is logically OR’ed with the IE1 SFR, just before the RETI instruction. The compiler assumes the IE1 SFR is mapped to address 0x0000.

The syntax of the pragma in C is:

```
#pragma BIS_IE1_INTERRUPT (func, mask);
```

The syntax of the pragma in C++ is:

```
#pragma BIS_IE1_INTERRUPT (mask);
```

In C, the argument func is the name of the function that is an interrupt. In C++, the pragma applies to the next function declared.
5.8.2 The CODE_SECTION Pragma

The CODE_SECTION pragma allocates space for the symbol in a section named section name.

The syntax of the pragma in C is:

```c
#pragma CODE_SECTION ( symbol , " section name ");
```

The syntax of the pragma in C++ is:

```c
#pragma CODE_SECTION (" section name ");
```

The CODE_SECTION pragma is useful if you have code objects that you want to link into an area separate from the .text section.

The following examples demonstrate the use of the CODE_SECTION pragma.

**Example 5-3. Using the CODE_SECTION Pragma C Source File**

```c
#pragma CODE_SECTION(funcA,"codeA")
int funcA(int a)
{
    int i;
    return (i = a);
}
```

**Example 5-4. Generated Assembly Code From Example 5-3**

```assembly
.sect "codeA"
.align 2
.clink
.global funcA

;*****************************************************************************
;* FUNCTION NAME: funcA  *
;* * *
;* Regs Modified : SP,SR,r12  *
;* Regs Used : SP,SR,r12  *
;* Local Frame Size : 0 Args + 4 Auto + 0 Save = 4 byte  *
;*****************************************************************************
funcA:
    ;* -----------------------------------------------*
    SUB.W #4,SP
    MOV.W r12,0(SP) ; |4|
    MOV.W 0(SP),2(SP) ; |6|
    MOV.W 2(SP),r12 ; |6|
    ADD.W #4,SP
    RET
```

**Example 5-5. Using the CODE_SECTION Pragma C++ Source File**

```c
#pragma CODE_SECTION("codeB")
int i_arg(int x) { return 1; }
int f_arg(float x) { return 2; }
```
5.8.3 The DATA_ALIGNPragma

The DATA_ALIGN pragma aligns the symbol to an alignment boundary. The alignment boundary is the maximum of the symbol's default alignment value or the value of the constant in bytes. The constant must be a power of 2.

The syntax of the pragma in C is:

```c
#pragma DATA_ALIGN ( symbol, constant );
```

The syntax of the pragma in C++ is:

```cpp
#pragma DATA_ALIGN ( constant );
```
5.8.4 The DATA_SECTIONPragma

The DATA_SECTION pragma allocates space for the symbol in a section named section name.
The syntax of the pragma in C is:

```c
#pragma DATA_SECTION ( symbol , " section name ");
```

The syntax of the pragma in C++ is:

```c
#pragma DATA_SECTION (" section name ");
```

The DATA_SECTION pragma is useful if you have data objects that you want to link into an area separate from the .bss section. If you allocate a global variable using a DATA_SECTION pragma and you want to reference the variable in C code, you must declare the variable as extern far.

Example 5-7 through Example 5-9 demonstrate the use of the DATA_SECTION pragma.

**Example 5-7. Using the DATA_SECTION Pragma C Source File**

```c
#pragma DATA_SECTION(bufferB, "my_sect")
char bufferA[512];
char bufferB[512];
```

```c
char bufferA[512];
#pragma DATA_SECTION("my_sect")
char bufferB[512];
```

**Example 5-9. Using the DATA_SECTION Pragma Assembly Source File**

```
.global bufferA
.bss bufferA,512,2
.global bufferB
bufferB: .usect "my_sect",512,2
```
5.8.5 **The FUNC_CANNOT_INLINEPragma**

The FUNC_CANNOT_INLINE pragma instructs the compiler that the named function cannot be expanded inline. Any function named with this pragma overrides any inlining you designate in any other way, such as using the inline keyword. Automatic inlining is also overridden with this pragma; see Section 2.11.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument `func` is the name of the function that cannot be inlined. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_CANNOT_INLINE ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_CANNOT_INLINE;
```

5.8.6 **The FUNC_EXT_CALLEDPragma**

When you use the --program_level_compile option, the compiler uses program-level optimization. When you use this type of optimization, the compiler removes any function that is not called, directly or indirectly, by main. You might have C/C++ functions that are called by hand-coded assembly instead of main.

The FUNC_EXT_CALLED pragma specifies to the optimizer to keep these C functions or any other functions that these C/C++ functions call. These functions act as entry points into C/C++.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument `func` is the name of the function that you do not want removed. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_EXT_CALLED ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_EXT_CALLED;
```

Except for _c_int00, which is the name reserved for the system reset interrupt for C/C++ programs, the name of the interrupt (the `func` argument) does not need to conform to a naming convention.

When you use program-level optimization, you may need to use the FUNC_EXT_CALLED pragma with certain options. See Section 3.3.2.
5.8.7 The **FUNC_IS_PURE** Pragma

The **FUNC_IS_PURE** pragma specifies to the compiler that the named function has no side effects. This allows the compiler to do the following:

- Delete the call to the function if the function's value is not needed
- Delete duplicate functions

The pragma must appear before any declaration or reference to the function. In C, the argument `func` is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_IS_PURE ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_IS_PURE;
```

5.8.8 The **FUNC_NEVER_RETURNS** Pragma

The **FUNC_NEVER_RETURNS** pragma specifies to the compiler that the function never returns to its caller.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument `func` is the name of the function that does not return. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_NEVER_RETURNS ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NEVER_RETURNS;
```

5.8.9 The **FUNC_NO_GLOBAL_ASG** Pragma

The **FUNC_NO_GLOBAL_ASG** pragma specifies to the compiler that the function makes no assignments to named global variables and contains no asm statements.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument `func` is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```
#pragma FUNC_NO_GLOBAL_ASG ( func );
```

The syntax of the pragma in C++ is:

```
#pragma FUNC_NO_GLOBAL_ASG;
```
5.8.10 **The FUNC_NO_IND_ASGPragma**

The FUNC_NO_IND_ASG pragma specifies to the compiler that the function makes no assignments through pointers and contains no asm statements.

The pragma must appear before any declaration or reference to the function that you want to keep. In C, the argument *func* is the name of the function that makes no assignments. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```c
#pragma FUNC_NO_IND_ASG (func);
```

The syntax of the pragma in C++ is:

```c
#pragma FUNC_NO_IND_ASG;
```

5.8.11 **The INTERRUPTPragma**

The INTERRUPT pragma enables you to handle interrupts directly with C code. In C, the argument *func* is the name of a function. In C++, the pragma applies to the next function declared.

The syntax of the pragma in C is:

```c
#pragma INTERRUPT (func);
```

The syntax of the pragma in C++ is:

```c
#pragma INTERRUPT;
```

The code for the function will return via the IRP (interrupt return pointer).

Except for _c_int00, which is the name reserved for the system reset interrupt for C programs, the name of the interrupt (the *func* argument) does not need to conform to a naming convention.

---

**Note:** The INTERRUPT pragma must not be used when BIOS HWI objects are used in conjunction with C functions. The HWI_enter/HWI_exit macros and the HWI dispatcher contain this functionality, and the use of the C modifier can cause catastrophic results.
5.8.12 The NO_HOOKSPragma

The NO_HOOKS pragma prevents entry and exit hook calls from being generated for a function.

The syntax of the pragma in C is:

```
#pragma NO_HOOKS (func);
```

The syntax of the pragma in C++ is:

```
#pragma NO_HOOKS;
```

See Section 2.13 for details on entry and exit hooks.

5.8.13 The vectorPragma

The vector pragma indicates that the function that follows is to be used as the interrupt vector routine for the listed vectors. The syntax of the pragma is:

```
#pragma vector = vec1, vec2, vec3, ...
```

The vector pragma requires linker command file support. The command file must specify output sections for each interrupt vector of the form .intxx where xx is the number of the interrupt vector. The output sections must map to the physical memory location of the appropriate interrupt vector. The standard linker command files are set up to handle the vector pragma.

The __even_in_range intrinsic provides a hint to the compiler when generating switch statements for interrupt vector routines. The intrinsic is usually used as follows:

```
switch (__even_in_range(x, NUM))
{
    ...
}
```

The __even_in_range intrinsic returns the value x to control the switch statement, but also tells the compiler that x must be an even value in the range of 0 to NUM, inclusive.
5.9 The _Pragma Operator

The MSP430 C/C++ compiler supports the C99 preprocessor _Pragma() operator. This preprocessor operator is similar to #pragma directives. However, _Pragma can be used in preprocessing macros (#defines).

The syntax of the operator is:

```
Pragma ("string_literal");
```

The argument string_literal is interpreted in the same way the tokens following a #pragma directive are processed. The string_literal must be enclosed in quotes. A quotation mark that is part of the string_literal must be preceded by a backward slash.

You can use the _Pragma operator to express #pragma directives in macros. For example, the DATA_SECTION syntax:

```
#pragma DATA_SECTION(func,"section");
```

Is represented by the _Pragma() operator syntax:

```
Pragma ("DATA_SECTION(func,"section")
```

The following code illustrates using _Pragma to specify the DATA_SECTION pragma in a macro:

```
... #define EMIT_PRAGMA(x) _Pragma(#x) #define COLLECT_DATA(var) EMIT_PRAGMA(DATA_SECTION(var,"mysection")) COLLECT_DATA(x) int x; ...
```

The EMIT_PRAGMA macro is needed to properly expand the quotes that are required to surround the section argument to the DATA_SECTION pragma.

5.10 Generating Linknames

The compiler transforms the names of externally visible identifiers when creating their linknames. The algorithm used depends on the scope within which the identifier is declared. For objects and C functions, an underscore (_) is prefixed to the identifier name. C++ functions are prefixed with an underscore also, but the function name is modified further.

Mangling is the process of embedding a function's signature (the number and types of its parameters) into its name. Mangling occurs only in C++ code. The mangling algorithm used closely follows that described in The Annotated Reference Manual (ARM). Mangling allows function overloading, operator overloading, and type-safe linking.

For example, the general form of a C++ linkname for a function named func is:

```
_func__F parmcodes
```

Where parmcodes is a sequence of letters that encodes the parameter types of func.

For this simple C++ source file:

```
int foo(int i){ } //global C++ function
```

This is the resulting assembly code:

```
_Fo_Fi
```

The linkname of foo is _foo__Fi, indicating that foo is a function that takes a single argument of type int. To aid inspection and debugging, a name demangling utility is provided that demangles names into those found in the original C++ source. See Chapter 8 for more information.
5.11 Initializing Static and Global Variables

The ANSI/ISO C standard specifies that global (extern) and static variables without explicit initializations must be initialized to 0 before the program begins running. This task is typically done when the program is loaded. Because the loading process is heavily dependent on the specific environment of the target application system, the compiler itself makes no provision for preinitializing variables at run time. It is up to your application to fulfill this requirement.

5.11.1 Initializing Static and Global Variables With the Linker

If your loader does not preinitialize variables, you can use the linker to preinitialize the variables to 0 in the object file. For example, in the linker command file, use a fill value of 0 in the .bss section:

```plaintext
SECTIONS
{
    ...
    .bss: {} = 0x00;
    ...
}
```

Because the linker writes a complete load image of the zeroed .bss section into the output COFF file, this method can have the unwanted effect of significantly increasing the size of the output file (but not the program).

If you burn your application into ROM, you should explicitly initialize variables that require initialization. The preceding method initializes .bss to 0 only at load time, not at system reset or power up. To make these variables 0 at run time, explicitly define them in your code.

For more information about linker command files and the SECTIONS directive, see the linker description information in the MSP430 Assembly Language Tools User’s Guide.

5.11.2 Initializing Static and Global Variables With the const Type Qualifier

Static and global variables of type `const` without explicit initializations are similar to other static and global variables because they might not be preinitialized to 0 (for the same reasons discussed in Section 5.11). For example:

```plaintext
const int zero; /* may not be initialized to 0 */
```

However, the initialization of const global and static variables is different because these variables are declared and initialized in a section called .const. For example:

```plaintext
const int zero = 0 /* guaranteed to be 0 */
```

This corresponds to an entry in the .const section:

```plaintext
.sect .const
_zero        .word 0
```

This feature is particularly useful for declaring a large table of constants, because neither time nor space is wasted at system startup to initialize the table. Additionally, the linker can be used to place the .const section in ROM.

You can use the DATA_SECTION pragma to put the variable in a section other than .const. For example, the following C code:

```plaintext
#pragma DATA_SECTION (var, ".mysect");
const int zero=0;
```

is compiled into this assembly code:

```plaintext
.sect .mysect
_zero        .word 0
```
5.12 Changing the ANSI/ISO C Language Mode

The --kr_compatible, --relaxed_ansi, and --strict_ansi options let you specify how the C/C++ compiler interprets your source code. You can compile your source code in the following modes:

- Normal ANSI/ISO mode
- K&R C mode
- Relaxed ANSI/ISO mode
- Strict ANSI/ISO mode

The default is normal ANSI/ISO mode. Under normal ANSI/ISO mode, most ANSI/ISO violations are emitted as errors. Strict ANSI/ISO violations (those idioms and allowances commonly accepted by C/C++ compilers, although violations with a strict interpretation of ANSI/ISO), however, are emitted as warnings. Language extensions, even those that conflict with ANSI/ISO C, are enabled.

K&R C mode does not apply to C++ code.

5.12.1 Compatibility With K&R C (--kr_compatible Option)

The ANSI/ISO C/C++ language is a superset of the de facto C standard defined in Kernighan and Ritchie's *The C Programming Language*. Most programs written for other non-ANSI/ISO compilers correctly compile and run without modification.

There are subtle changes, however, in the language that can affect existing code. Appendix C in *The C Programming Language* (second edition, referred to in this manual as K&R) summarizes the differences between ANSI/ISO C and the first edition's C standard (the first edition is referred to in this manual as K&R C).

To simplify the process of compiling existing C programs with the ANSI/ISO C/C++ compiler, the compiler has a K&R option (--kr_compatible) that modifies some semantic rules of the language for compatibility with older code. In general, the --kr_compatible option relaxes requirements that are stricter for ANSI/ISO C than for K&R C. The --kr_compatible option does not disable any new features of the language such as function prototypes, enumerations, initializations, or preprocessor constructs. Instead, --kr_compatible simply liberalizes the ANSI/ISO rules without revoking any of the features.

The specific differences between the ANSI/ISO version of C and the K&R version of C are as follows:

- The integral promotion rules have changed regarding promoting an unsigned type to a wider signed type. Under K&R C, the result type was an unsigned version of the wider type; under ANSI/ISO, the result type is a signed version of the wider type. This affects operations that perform differently when applied to signed or unsigned operands; namely, comparisons, division (and mod), and right shift:

  ```c
  unsigned short u;
  int i;
  if (u < i) /* SIGNED comparison, unless --kr_compatible used */
  ```

- ANSI/ISO prohibits combining two pointers to different types in an operation. In most K&R compilers, this situation produces only a warning. Such cases are still diagnosed when --kr_compatible is used, but with less severity:

  ```c
  int *p;
  char *q = p; /* error without --kr_compatible, warning with --kr_compatible */
  ```

- External declarations with no type or storage class (only an identifier) are illegal in ANSI/ISO but legal in K&R:

  ```c
  a; /* illegal unless --kr_compatible used */
  ```

- ANSI/ISO interprets file scope definitions that have no initializers as tentative definitions. In a single module, multiple definitions of this form are fused together into a single definition. Under K&R, each definition is treated as a separate definition, resulting in multiple definitions of the same object and usually an error. For example:

  ```c
  int a;
  int a; /* illegal if --kr_compatible used, OK if not */
  ```

  Under ANSI/ISO, the result of these two definitions is a single definition for the object a. For most K&R compilers, this sequence is illegal, because int a is defined twice.

- ANSI/ISO prohibits, but K&R allows objects with external linkage to be redeclared as static:

  ```c
  extern int a;
  static int a; /* illegal unless --kr_compatible used */
  ```
5.12.2 Enabling Strict ANSI/ISO Mode and Relaxed ANSI/ISO Mode (--strict_ansi and --relaxed_ansi Options)

Use the --strict_ansi option when you want to compile under strict ANSI/ISO mode. In this mode, error messages are provided when non-ANSI/ISO features are used, and language extensions that could invalidate a strictly conforming program are disabled. Examples of such extensions are the inline and asm keywords.

Use the --relaxed_ansi option when you want the compiler to ignore strict ANSI/ISO violations rather than emit a warning (as occurs in normal ANSI/ISO mode) or an error message (as occurs in strict ANSI/ISO mode). In relaxed ANSI/ISO mode, the compiler accepts extensions to the ANSI/ISO C standard, even when they conflict with ANSI/ISO C.

5.12.3 Enabling Embedded C++ Mode (--embedded_cpp Option)

The compiler supports the compilation of embedded C++. In this mode, some features of C++ are removed that are of less value or too expensive to support in an embedded system. When compiling for embedded C++, the compiler generates diagnostics for the use of omitted features.

Embedded C++ is enabled by compiling with the --embedded_cpp option.

Embedded C++ omits these C++ features:
- Templates
- Exception handling
- Run-time type information
- The new cast syntax
- The keyword mutable
- Multiple inheritance
- Virtual inheritance

Under the standard definition of embedded C++, namespaces and using-declarations are not supported. The MSP430 compiler nevertheless allows these features under embedded C++ because the C++ run-time-support library makes use of them. Furthermore, these features impose no run-time penalty.
5.13 GNU C Compiler Extensions

The GNU compiler, GCC, provides a number of language features not found in the ANSI standard C. The definition and official examples of these extensions can be found at [http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/C-Extensions.html#C-Extensions](http://gcc.gnu.org/onlinedocs/gcc-3.4.6/gcc/C-Extensions.html#C-Extensions). To enable GNU extension support, use the --gcc compiler option.

The extensions that the TI C compiler supports are listed in Table 5-2.

### Table 5-2. GCC Extensions Supported

<table>
<thead>
<tr>
<th>Extensions</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement expressions</td>
<td>Putting statements and declarations inside expressions (useful for creating 'safe' macros)</td>
</tr>
<tr>
<td>Local labels</td>
<td>Labels local to a statement expression</td>
</tr>
<tr>
<td>Naming types</td>
<td>Giving a name to the type of an expression</td>
</tr>
<tr>
<td>typeof operator</td>
<td>typeof referring to the type of an expression</td>
</tr>
<tr>
<td>Generalized lvalues</td>
<td>Using question mark (?) and comma (,) and casts in lvalues</td>
</tr>
<tr>
<td>Conditionals</td>
<td>Omitting the middle operand of a ? expression</td>
</tr>
<tr>
<td>Hex floats</td>
<td>Hexadecimal floating-point constants</td>
</tr>
<tr>
<td>Zero length</td>
<td>Zero-length arrays</td>
</tr>
<tr>
<td>Macro varargs</td>
<td>Macros with a variable number of arguments</td>
</tr>
<tr>
<td>Subscripting</td>
<td>Any array can be subscripted, even if it is not an lvalue.</td>
</tr>
<tr>
<td>Pointer arithmetic</td>
<td>Arithmetic on void pointers and function pointers</td>
</tr>
<tr>
<td>Initializers</td>
<td>Nonconstant initializers</td>
</tr>
<tr>
<td>Cast constructors</td>
<td>Constructor expressions give structures, unions, or arrays as values</td>
</tr>
<tr>
<td>Labeled elements</td>
<td>Labeling elements of initializers</td>
</tr>
<tr>
<td>Cast to union</td>
<td>Casting to union type from any member of the union</td>
</tr>
<tr>
<td>Case ranges</td>
<td>'Case 1 ... 9' and such</td>
</tr>
<tr>
<td>Function attributes</td>
<td>Declaring that functions have no side effects, or that they can never return</td>
</tr>
<tr>
<td>Function prototypes</td>
<td>Prototype declarations and old-style definitions</td>
</tr>
<tr>
<td>C++ comments</td>
<td>C++ comments are recognized.</td>
</tr>
<tr>
<td>Dollar signs</td>
<td>A dollar sign is allowed in identifiers.</td>
</tr>
<tr>
<td>Character escapes</td>
<td>The character ESC is represented as 'e'</td>
</tr>
<tr>
<td>Alignment</td>
<td>Inquiring about the alignment of a type or variable</td>
</tr>
<tr>
<td>Variable attributes</td>
<td>Specifying the attributes of variables</td>
</tr>
<tr>
<td>Type attributes</td>
<td>Specifying the attributes of types</td>
</tr>
<tr>
<td>Inline</td>
<td>Defining inline functions (as fast as macros)</td>
</tr>
<tr>
<td>Assembly labels</td>
<td>Specifying the assembler name to use for a C symbol</td>
</tr>
<tr>
<td>Alternate keywords</td>
<td>Header files can use <strong>const</strong>, <strong>asm</strong>, etc</td>
</tr>
<tr>
<td>Incomplete enums</td>
<td>enum foo??</td>
</tr>
<tr>
<td>Function names</td>
<td>Printable strings which are the name of the current function</td>
</tr>
<tr>
<td>Return address</td>
<td>Getting the return or frame address of a function</td>
</tr>
<tr>
<td>Other built-ins</td>
<td>Other built-in functions include:</td>
</tr>
<tr>
<td></td>
<td>__builtin_return_address</td>
</tr>
<tr>
<td></td>
<td>__builtin_frame_address</td>
</tr>
<tr>
<td></td>
<td>__builtin_constant_p</td>
</tr>
<tr>
<td></td>
<td>__builtin_expect</td>
</tr>
</tbody>
</table>
5.13.1 Function Attributes

The GNU extension support provides a number of attributes about functions to help the C compiler’s optimization. The TI compiler accepts only three of these attributes. All others are simply ignored. Table 5-3 lists the attributes that are supported.

Table 5-3. TI-Supported GCC Function Attributes

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deprecated</td>
<td>This function exists but the compiler generates a warning if it is used.</td>
</tr>
<tr>
<td>section</td>
<td>Place this function in the specified section.</td>
</tr>
<tr>
<td>unused</td>
<td>The function is meant to be possibly not used.</td>
</tr>
</tbody>
</table>

5.13.2 Built-In Functions

TI provides support for only the four built-in functions in Table 5-4.

Table 5-4. TI-Supported GCC Built-In Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__builtin_constant_p(expr)</td>
<td>Returns true only if expr is a constant at compile time.</td>
</tr>
<tr>
<td>__builtin_expect(expr, CONST)</td>
<td>Returns expr. The compiler uses this function to optimize along paths determined by conditional statements such as if-else. While this function can be used anywhere in your code, it only conveys useful information to the compiler if it is the entire predicate of an if statement and CONST is 0 or 1. For example, the following indicates that you expect the predicate &quot;a == 3&quot; to be true most of the time:</td>
</tr>
<tr>
<td>if (__builtin_expect(a == 3, 1))</td>
<td></td>
</tr>
<tr>
<td>__builtin_return_address(int level)</td>
<td>Returns 0.</td>
</tr>
<tr>
<td>__builtin_frame_address(int level)</td>
<td>Returns 0.</td>
</tr>
</tbody>
</table>

5.14 Compiler Limits

Due to the variety of host systems supported by the C/C++ compiler and the limitations of some of these systems, the compiler may not be able to successfully compile source files that are excessively large or complex. In general, exceeding such a system limit prevents continued compilation, so the compiler aborts immediately after printing the error message. Simplify the program to avoid exceeding a system limit.

Some systems do not allow filenames longer than 500 characters. Make sure your filenames are shorter than 500.

The compiler has no arbitrary limits but is limited by the amount of memory available on the host system. On smaller host systems such as PCs, the optimizer may run out of memory. If this occurs, the optimizer terminates and the shell continues compiling the file with the code generator. This results in a file compiled with no optimization. The optimizer compiles one function at a time, so the most likely cause of this is a large or extremely complex function in your source module. To correct the problem, your options are:

- Don't optimize the module in question.
- Identify the function that caused the problem and break it down into smaller functions.
- Extract the function from the module and place it in a separate module that can be compiled without optimization so that the remaining functions can be optimized.