

Language Extensions

This chapter provides information about High C/C++ language extensions that supplements the **High C/C++ Language Reference**. It contains the following sections:

§7.1: *Semantics of Near and Far*

§7.2: *Special Type Qualifiers*

§7.3: *Generating Inline Assembly Code with `_ASM`*

§7.4: *The Enhanced asm Facility*

Note: For a more thorough presentation of High C/C++ language extensions, consult the **High C/C++ Language Reference**.

7.1 Semantics of Near and Far

The general notions of *near* and *far* are introduced in the **High C/C++ Language Reference**. Read that information, if necessary, before continuing here.

7.1.1 Near and Far Functions

The way a procedure is called depends on its distance from the `call` instruction. There are two possibilities:

- If the procedure is within $\pm 16\text{MB}$ bytes of the call, a single instruction suffices. Such a call is a *near* call.
- If the procedure is farther away, three instructions are required. Such a call is a *far* call.
- The keywords `_Near` and `_Far` are used to indicate that functions are near or far, respectively (see below).

The compiler assumes that calls to functions defined in the same module are *near* calls. If you use options **-MB** or **-Mb** (Big memory model), *far* calls are used to access imported (that is, **extern**) functions.

- By default, the compiler treats all calls as near calls.
- When using option **-Mb**, the compiler respects far calls, when specified.
- When using option **-MB**, the compiler treats all calls as far calls.

See Chapter 3: *Using Compiler Options* for information about these options.

You can explicitly declare functions to be near or far with the **_Near** or **_Far** qualifiers, as explained in the **High C/C++ Language Reference**.

For example:

```
_Far int ffar1();
char *(* _Far ffar2())();
_Near int fnear1();
char *(* _Near fnear2())();
int f1();
char *(* f2())();
```

Calls to `ffar1` and `ffar2` use the far linkage, while calls to `fnear1` and `fnear2` use the near linkage, and calls to `f1` and `f2` use the default linkage.

The linker can detect when the offset required for a near call exceeds the space within the instruction to place the offset. Therefore, if your program links successfully, near linkage worked.

7.2 Special Type Qualifiers

High C/C++ provides the special type qualifiers listed in Table 7.1 for generating PowerPC applications.

Table 7.1 Special Type Qualifiers

Type Qualifier	Description	See Page
_Alias	Variable can be accessed by indirect means.	146

Table 7.1 Special Type Qualifiers (Continued)

Type Qualifier	Description	See Page
<code>_Asm</code>	Function is an assembly language macro.	146
<code>_Noalias</code>	Variable cannot be accessed by indirect means.	146
<code>_Reversed_Endian</code>	Variables of opposite endianness are byte-reversed at the time of access.	146

Note: These type qualifiers are defined in the default profile, `hc.pro`, which typically resides in the same directory as the compiler executables.

If you explicitly define your own profile (via `-Hpro=fname`), the type qualifiers in the default profile will not be recognized.

7.2.1 Casting Type Qualifiers

Due to the characteristics of C and how type qualifiers (sometimes referred to as type attributes) work, you cannot use type qualifiers in casts according to the normal C mechanism for casting. For example, the following cast would not work as expected:

```
x = (volatile int) y ;
```

This would not work because the interpretation for the cast takes place after the access is completed.

To cast a type qualifier, you must cast the address of the desired variable, and then dereference to get the desired effect. For example:

```
x = * (volatile int *) &y ;
```

This requirement applies to all type qualifiers that affect external accesses.

7.2.2 Access-Related Type Qualifiers

The type qualifiers covered in this section enable you to specify whether variables can or cannot be accessed indirectly.

7.2.2.1 Type Qualifier `_Alias`

When you declare a variable with the `_Alias` type qualifier, you are telling the optimizer that the variable can be accessed by indirect means; for example, through a pointer dereference.

The `_Alias` type qualifier is used to override the effect of the `Noalias` toggle. Ordinarily, any global variable or pointer dereference is assumed aliasable.

Type qualifiers `_Alias` and `_Noalias` are mutually exclusive.

7.2.2.2 Type Qualifier `_Noalias`

When you declare a variable with the `_Noalias` type qualifier, you are telling the optimizer to assume that the variable cannot be accessed or modified through any indirect means, such as through a pointer dereference.

When you apply `_Noalias` to a pointer-based variable (for example, `_Noalias char *p`), the optimizer assumes that no other pointer references the same memory location at the same time, and that the pointer does not contain the address of any variable that is accessed in the same function.

Caution: The compiler makes no attempt to confirm that these assumptions have not been violated. If the `_Noalias` qualifier is used incorrectly, the optimizer's assumption is wrong and the compiler could generate incorrect code.

7.2.3 Endianness-Related Type Qualifiers

The type qualifier covered in this section allows you to reverse the endian orientation of variables.

7.2.3.1 Type Qualifier `_Reversed_Endian`

With type qualifier `_Reversed_Endian`, you can declare variables that have an endianness opposite to that of the machine on which the program

runs. See §10.4: *Reversed Endianness* for more information about this type qualifier.

7.2.4 Function Type Qualifiers

The type qualifier covered in this section designates functions as being assembly language functions.

7.2.4.1 Type Qualifier `_Asm`

With the `_Asm` type qualifier, High C/C++ provides an enhanced `asm` capability through which you can insert several assembly instructions into the compiler's output. The `_Asm` type qualifier declares a set of assembly instructions to be a C function.

Unlike the `_ASM` function, which has one argument (see §7.3: *Generating Inline Assembly Code with _ASM*), a function qualified with `_Asm` can be defined or called with multiple arguments.

For more information, see §7.4: *The Enhanced asm Facility*.

Example: `_Asm` function with one argument

This `_Asm` function `CLZ()` takes one argument, which can be either a register or a constant:

```
_Asm int CLZ (long arg) {
% reg arg
    ori    %r14,arg,0
% con arg
    ori    %r14,%r0,arg
}
```

Function `CLZ()` can be called just like any other C function, as shown here:

```
int x = 20;
void func() {
    CLZ(40);
    CLZ(x+2);
}
```

The assembly code generated by the compiler for this example would be:

```
.data
.align 2
```

```

        .globl  x
x:
        .size   x,0x4
        .type   x,@object
        .long   20                                ! 0x14
        .text
        .align  2
        .globl  func
func:
        ...
        ori    %r14,%r0,+40                        ! _Asm inline for CLZ(40)
        addis   %r12,%r0,...data.0@h              !+c
        ori     %r12,%r12,...data.0@l             !+10
        lwz     %r12,x-...data.0(%r12)            !+14
        addi    %r12,%r12,2                        !+18
        ori     %r14,%r12,0                        ! _Asm inline for CLZ(x+2)
        ...

```

Example: `_Asm` function with multiple arguments You can write `_Asm` functions that specify more than one argument and account for different types, as shown here:

```

_Asml void multi_load(int arg1, int arg2, int arg3) {
% reg arg1, arg2 ; con arg3
    mtspr    20,arg1
    mtspr    21,arg2
    lmw      %r3,arg3(%r3)
% reg arg1, arg2, arg3
    mtspr    20,arg1
    mtspr    21,arg2
    lswz     %r3,%r3,arg3
}

```

In this example, function `multi_load()` can be called when `arg3` is either a register or a constant. This function is written to handle either case appropriately.

Function `multi_load()` can be called as shown here:

```

extern int x;
extern int y;
int count = 0;
int func() {
    multi_load(x+1,x+1,4);
    multi_load(x+1,y+1,count++);
}

```

The assembly code generated by the compiler for this example would be:

```

        .text
        .align 2
        .globl func
func:
        ...
        addis    %r31,%r0,x@ha                !+10
        lwz      %r12,x@l(%r31)                !+14
        addi     %r10,%r12,1                    !+18
        mtspr    20,%r10                        ! _Asm inline for
        mtspr    21,%r10                        ! multi_load(x+1,x+1,4)
        lmw      %r3,+4(%r3)

        lwz      %r12,x@l(%r31)                !+1c
        addi     %r10,%r12,1                    !+20
        addis    %r12,%r0,y@ha                !+24
        lwz      %r12,y@l(%r12)                !+28
        addi     %r8,%r12,1                    !+2c
        addis    %r9,%r0,...data.0@h          !+30
        ori      %r9,%r9,...data.0@l          !+34
        lwz      %r12,count-...data.0(%r9)    !+38
        addi     %r11,%r12,1                    !+3c
        stw      %r11,count-...data.0(%r9)    !+40
        mtspr    20,%r10                        ! _Asm inline for
        mtspr    21,%r8                        ! multi_load(x+1,y+1,count++)
        lswz     %r3,%r3,%r12

```

7.3 Generating Inline Assembly Code with `_ASM`

Note: This feature is provided for backward compatibility.

High C/C++ provides a simple inline assembler directive, `_ASM`.

A function call of the form `_ASM("string")` places the string *string* in the assembly output of the compiler at the point of the `_ASM` call and appends a newline character. For example:

```

int func() {
    // Set count register to 10
    _ASM("\taddi    %r14,%r0,10");
    _ASM("\tmtspr   9,%r14");
    // Insert the following in the code stream:

```

```
// label:
//      addi    %r14,%r0,1
_ASM("label:\n\taddi    %r14,%r14,1");
}
```

(No attempt is made to check the contents of the string.)

The assembly code generated by the compiler for this example would be:

```
.text
.align 2
.globl func
func:
...
addi    %r14,%r0,10      ! *****
mtspr   9,%r14           ! These lines were inlined
label:   ! by the _ASM call
addi    %r14,%r14,1      ! *****
...
```

The `_ASM` directive can appear only where a function call is permitted; that is, within a function definition. `_ASM` cannot be used to insert assembly instructions outside of a function. (To generate inline assembly code outside a function, you must use `asm`; see §7.4: *The Enhanced asm Facility* for more information.)

Note: Use the `_ASM` directive only if you have also specified command-line option `-S`. Otherwise, an error diagnostic results.

The only machine registers that can be modified in an `_ASM` statement are those that can be modified by a call.

You cannot reliably use a target machine's branch instructions within `_ASM`, because you might introduce graph paths of which the compiler is unaware into the program flow. Such graph paths can cause the compiler to generate incorrect code.

Good example Branch instructions should be contained within a single section of `_ASM` code, not intermixed with C. For example:

```
/* * * * * * * * Good Example * * * * * */
int func() {
    _ASM(" \taddi    %r14,%r0,10");
    _ASM(" \tmtspr   9,%r14");
}
```



```

_ASM("loop:");
_ASM("\taddi    %r14,%r14,1");
_ASM("\tbdnz    loop");
}

```

The assembly code generated by the compiler for this example would be:

```

.text
.align 2
.globl func
func:
...
addi    %r14,%r0,10    ! *****
mtspr   9,%r14         ! Good example:
loop:   ! No C code intervenes. The
addi    %r14,%r14,1    ! loop is composed of _ASM
                        ! inlines only.
bdnz    loop           ! *****
...

```

Bad example Using a branch and label across C code, as shown here, is dangerous and should not be attempted:

```

/* * * * * * Bad Example * * * * */
int func() {
    int i = 0;
    extern int a[];
    _ASM("\taddi    %r14,%r0,10");
    _ASM("\tmtspr   9,%r14");
    _ASM("loop:");
    _ASM("\taddi    %r14,%r14,1");

    a[i] = i;           // C code interfering
    i++;                // with loop formed by
                        // _ASM inline code.

    _ASM("\tbdnz    loop");
}

```

The assembly code generated by the compiler for this example would be:

```

.text
.align 2
.globl func
func:
...
addi    %r14,%r0,10    ! *****

```

```

loop:    mtspr      9,%r14          ! Bad Example:
                                ! It is bad practice to let
                                ! C code inside loops
                                ! formed by _ASM inline.
        addi      %r14,%r14,1
        addis     %r12,%r0,a@ha !<--- Code from the block of
        stw       %r31,a@l(%r12)!<--- C statements
        bdnz     loop            ! *****

```

Dead code Furthermore, based on control-flow analysis, the compiler can discard `_ASM` statements that appear dead:

```

int func() {
    _ASM("\tb    label");
    return;
    _ASM("label:");
    _ASM("\taddi   %r13,%r0,1");
    _ASM("\taddi   %r14,%r14,1");
}

```

The three `_ASM` statements after the `return` are discarded because they are dead (code following a `return;` statement never executes).

The assembly code generated by the compiler for this example would be:

```

        .text
        .align 2
        .globl func
func:
    ...
    b      label          ! The rest of the _ASM
                        ! inlines got discarded.

    lwz    %r0,+20(%r1)
    mtlr   %r0
    addi   %r1,%r1,16
    bclr   20,0

```

It is not reasonable to turn off dead-code elimination, because the compiler uses it to eliminate extra compiler-inserted code that assists in other optimizations.

It is also not reasonable to assume that each `_ASM` breaks up a basic block, because the compiler would have to further assume that each `_ASM` can be branched to from every other `_ASM`, and the connectivity in the flow graph would result in terrible code for the C constructs near the `_ASMs`.

Caution: Use **_ASM** sparingly, and *never* use it to alter control flow across C constructs.

7.4 The Enhanced **asm** Facility

High C/C++ supports the enhanced **asm** facility as defined in the **AT&T UNIX System V Programmer's Guide: ANSI C and Programming Support Tools**.

Note: In High C/C++ the keyword **asm** has been renamed to **_Asm** to avoid corrupting the ANSI Standard C name space.

For example:

```
#define asm _Asm
...
asm char setfc(int count) {
% con count;
    mtsrim fc,count
% reg count;
    mtsr    fc,count
}
```

Note: A source module using the enhanced **asm** facility must be compiled with the **-s** option so the assembler will be invoked to produce the object file.

7.4.1 **_Asm** Macros

Programs use the enhanced **asm** facility by way of **_Asm** macros. This is the syntax of an **_Asm** macro:

```
_Asm [ type_spec ] identifier ( [ formal_param_list ] ) {
    storage_mode_spec [ ; storage_mode_spec ]
        asm_body
    ...
}
```

```

    [ storage_mode_spec [ ; storage_mode_spec ]
      asm_body
      ... ]
  }

```

Table 7.2 lists the meanings of the individual components of the **_Asm** macro.

Table 7.2 Components of an **_Asm Macro**

Component	Meaning
<i>type_spec</i>	Any valid C type (optional)
<i>identifier</i>	Any valid C identifier (required)
<i>formal_param_list</i>	One or more valid C parameters (optional)
<i>storage_mode_spec</i>	Storage-mode specification line (one required); a <i>pattern</i> for the actual parameter list to match. See Table 7.3.
<i>asm_body</i>	Portion of assembly code to be expanded when the actual parameter list matches the pattern in <i>storage_mode_spec</i> .

storage_mode_spec This is the syntax of each *storage_mode_spec* (also known as a *pattern*):

```

    % storage_mode identifier [ , identifier ]

```

The “%” character defines a case for the argument(s) to the macro. Table 7.3 lists the different cases (storage modes) the compiler recognizes.

Table 7.3 Storage Modes for **_Asm Macro Arguments**

Mode	Meaning
con	The argument is a compile-time constant.
error	Generate a compiler error.
lab	The argument is a unique label generated by the compiler.
mem	Matches any allowed machine-addressing mode, including con and reg .
reg	The argument is a treg or ureg .
treg	Temporary register selected by the compiler.
ureg	C register variable, allocated by the compiler in a machine register.

Note: Any identifier with storage mode **lab** must be unique. Such an identifier does not appear as a formal parameter in the **asm** macro definition.

Use storage mode **error** to flag errors at compile time if a set of arguments does not match any of the **asm** macro's specified patterns.

For more information A complete discussion of this **asm** facility is beyond the scope of this Programmer's Guide. For a more detailed explanation, refer to the *Enhanced **asm** Facility* appendix in the **AT&T UNIX System V Programmer's Guide: ANSI C and Programming Support Tools**.

For more information about High C/C++ **_Asm** macros, see §7.2.4.1: *Type Qualifier _Asm*.

