

# Assembly-Language Communication

---

This chapter describes the conventions the High C/C++ compiler uses to organize C and C++ functions and assembly-language functions. It contains the following sections:

§12.1: *Overview*

§12.2: *Coding Assembly Routines*

§12.3: *Function-Naming Conventions in C*

§12.4: *Function-Naming Conventions in C++*

§12.5: *Calling Assembly Routines from C and C++*

§12.6: *Calling C Functions from Assembly Routines*

§12.7: *Sharing Variables Across Modules*

---

## 12.1 Overview

This chapter documents the conventions the High C/C++ compiler uses to organize C and C++ functions and assembly-language functions. These conventions ensure that the ELF linker can correctly link the C and C++ language modules with any assembly-code modules you have written to optimize program performance.

By using these conventions, you ensure that global data can be shared among your program's functions: C and C++ functions can call assembly functions, and assembly functions can call C and C++ functions.

---

## 12.2 Coding Assembly Routines

To call a general assembly-language routine from a High C/C++ program, code the assembly as follows:

```
        .text
        .align 2
        .globl name
name:
```

where *name* is the function's name as referenced from C.

For a description of how arguments are passed and how function results are returned, see Chapter 11: *PowerPC Run-Time Organization*.

---

## 12.3 Function-Naming Conventions in C

A function that is global (that is, a function that can be called across module boundaries) must have information provided to the linker that associates the name of the function with its address. This is done by placing a corresponding name in the object file's external symbol table. This name does not need to be the same as the actual name of the function; it only needs to be derivable from it.

In assembly language, the name that is placed in the symbol table is lexically identical to the name of the corresponding symbol. In C, the entry in the symbol table is derived from the corresponding symbol by prefixing an underscore “\_”. For example, an external routine *fnc* has a symbol-table entry of *\_fnc*.

You can use `pragma Alias` to explicitly define how a public symbol is to appear in the symbol table. For example, the following code illustrates how to call an assembly function named *fnc* (with no underscore “\_” prefix) from a High C/C++ program:

```
extern fnc();
#pragma Alias(fnc, "fnc")
void main() {...
    fnc(); ...
}
```

---

## 12.4 Function-Naming Conventions in C++

In C++, function names in the symbol table are in mangled form. The compiler derives the mangled name from the function's source-code name, its argument types, and any enclosing class definitions. It is difficult to access a C++ function from assembly language because of the name mangling.

To access an assembly-language function from C++, declare the function as a C function using **extern "C"**. The compiler does not mangle C function names.

---

## 12.5 Calling Assembly Routines from C and C++

The following examples of High C and High C++ code demonstrate how to make a call to assembly language function `peek()`:

*High C:*

```
extern int peek(long adr);
#pragma Alias(peek, "peek")
void main(){
    char b; ...
    b = peek(0x8000); ...
}
```

*High C++:*

```
extern "C" int peek(long adr);
...
```

Assembly:

```

        .text
        .align 2
        .globl peek
peek:
        lbz    %r12,0(%r3) ! read *(char *)adr
        extsb  %r12,%r12   ! sign extend the byte
        cmpi   %cr0,%r12,0 ! if *(char *) adr is zero
        bne    ..LL33      ! then
        addi   %r3,%r0,0    ! set return value to 0
        b      ..LL34      ! else
..LL33:
        addi   %r3,%r0,1    ! set return value to 1
..LL34:
        bclr   20,0        ! return

```

---

## 12.6 Calling C Functions from Assembly Routines

To call a C function from an assembly routine, do the following *in the order given*:

1. Allocate a new stack frame on the local register stack.
2. Load arguments into registers %r3 through %r10 and %f1 through %f8.
3. Execute the **call** instruction.
4. Deallocate the stack frame after the call.

For example:

High C:

```

void write_string (char *s){
    printf("%s\n", s);
}

```

Assembly:

```

        .comm  string,4
        .globl call
call:
    ...
    addis    %r12,%r0,string@ha ! load the
    lwz      %r3,string@l(%r12) ! argument to
                                ! write_string
    bl       write_string      ! call write_string
    ...

```

---

## 12.7 Sharing Variables Across Modules

Global variables in High C/C++ programs are aliased in the same manner as public functions. A global variable “x” appears in the symbol table as “\_x”, unless specified otherwise with an `Alias` pragma.

Chapter 10: *Storage Mapping* explains how the various C data types are mapped into storage. Note that uninitialized global variables without the **extern** qualifier are actually defined as individual common blocks, unless `toggle Multiple_vardefs` is Off.

The following examples illustrate the sharing of variables across C and assembly modules.

High C:

```

int  alpha, beta;
char hextable[] = "0123456789ABCDEF";
extern char *names[]; /* Read-only table of names */
extern short status;

```

Assembly:

```

        .comm    alpha,4
        .comm    beta,4

        .data
        .align   2
        .globl   hextable      ! imported from C
hextable:
        .size    hextable,0x11
        .type    hextable,@object
        .byte    "0123456789ABCDEF"
        .byte    0
status:
        .word    0

        .rdata
        .align   2
        .globl   names         ! Read-only data
names:
        .word    0

```

*Mapping variables  
into a named block*

The High C/C++ compiler provides the ability to map more than one variable into a named block; for example, a common block as in FORTRAN. This facility is provided by the `Data` pragma and is documented in §5.3.4: *Mapping Variables to Control Sections: Pragma Data*.

The following code illustrates how such a common block can be accessed from assembly language:

*C Common Block Definition:*

```

#pragma Data(Common,"block_name")
static int   a,b;
static char  c,d;
static short e;
#pragma Data

```

*Assembly-Language Equivalent:*

```

        .align    2
a = 0x0                                ! static data
        .align    2
b = 0x4                                ! static data
        .align    0
c = 0x8                                ! static data
        .align    0
d = 0x9                                ! static data
        .align    1
e = 0xa                                ! static data

        .text
        .align    2
        .globl    func

func:
    ...
! load base address of common data
    addis    %r11,%r0,..block_name.0@h
    ori      %r11,%r11,..block_name.0@l

    lwz      %r10,a(%r11)    ! load a
    lwz      %r12,b(%r11)    ! load b
    add      %r9,%r12,%r10    ! a + b
    lbz      %r12,c(%r11)    ! load sign extended c
    extsb    %r10,%r12
    lbz      %r12,d(%r11)    ! load sign extended d
    extsb    %r12,%r12
    add      %r12,%r12,%r10    ! c + d
    subf     %r10,%r12,%r9    ! a + b - (c + d)
    lha      %r12,e(%r11)    ! load half word e
    add      %r12,%r12,%r10    ! a + b - (c + d) + e
    extsb    %r3,%r12        ! sign extend result
    stb      %r3,c(%r11)     ! store result in c
    ...
    .comm    ..block_name.0,0xc

```

Note that variables *a*, *b*, *c*, *d*, and *e* are *not* global. That is, they do not appear in the symbol table with the **global** attribute. The only name that appears in the symbol table is *block\_name*.

