3 *iWarp C Extensions*

In addition to the widely-recognized extensions noted in Chapter 1, the iWarp C compiler also supports the following extensions to the C programming language:

- communication support (send and receive primitives)
- condition code checking support (iWarp condition codes)
- assembly code inlining facility
- pragma support for inlining specification

Communication Support (send and receive)

The iWarp C compiler recognizes send and receive pseudo-functions for sending and receiving data from either of two logical gates. The logical gates are named GATE0 and GATE1 by stdio.h, which you must #include in your program. There are send and receive pseudo-functions corresponding to each data type as shown in the following list:

receive

int _receivei(*channel*)

float _receivef(*channel*)

double _received(*channel*)

send

void _sendi(*channel, int-data*)

void _sendf(*channel, float-data*)

void _sendd(*channel, double-data*)

channel is either GATE0 or GATE1
int-data is an integer expression
float-data is a float expression
double-data is a double expression

NOTE

Integer and float communication each use a single word (32 bits) to transfer data between cells. Double communication uses two words (64 bits).

You must write programs so that data sent using _sendi is received using _receivei, data sent using _sendf is received using _receivef, and data sent using _sendd is received using _received. The compiler does not check for conflicting data types in send and receive pseudo-functions.

Condition Code Checking Support (cond and condcall)

You can check iWarp condition codes and use them for the following:

- creating a boolean value (0 or 1)
- conditional branching
- conditionally calling a function

The iWarp C compiler provides two pseudo-functions to allow this functionality; _cond and _condcall.

_cond

The syntax of the _cond pseudo-function is as follows:

int _cond(*condition-code*)

condition-code is a symbol defined by either #include<iwarp_cc.h> or #include<sys/iwarp_math_cc.h>, depending on the symbol.

The _cond pseudo-function tests a single iWarp condition and can be used for any of the purposes previously listed. The value of _cond is one if the condition tested is true and zero if the condition is false.

_condcall

The syntax of the _condcall pseudo-function is as follows:

void _condcall (*function-name, condition-code*)

function-name is the name of a parameterless function which is defined to return void. You cannot use a pointer to a function.

condition-code is a symbol defined by either #include<iwarp_cc.h> or #include<sys/iwarp_math_cc.h>, depending on the symbol.

The _condcall pseudo-function places a call to the named function if the condition tested is true.

The _condcall pseudo-function allows you to write a more compact call to a parameterless procedure (a function returning void) to generate more efficient code. For example, consider the following:

```
_condcall(example,code);
```

This use of _condcall is more compact and will probably generate more efficient code than the otherwise equivalent:

```
if (_cond(code)) example();
```

Unsafe Condition Codes

You should not use _cond or _condcall to check any unsafe condition codes. Unsafe conditions are conditions that occur synchronous to the code being executed. This includes all conditions generated by arithmetic instructions.

Appendix B contains complete lists of the condition codes which are associated with safe (asynchronous) and unsafe (synchronous) conditions. Only the asynchronous condition codes are safe for use in _cond and _condcall constructs. The safe condition codes are defined in <iwarp_cc.h>. The unsafe condition codes are defined in <sys/iwarp_math_cc.h>.

Assembly Code Inlining Facility

The assembly code inlining (asm-inline) facility allows you to define assembly code templates that the compiler can insert into the code it generates. Asm-inlines have any number of input parameters and an optional return value. An asm-inline is generally expanded in the code rather than as a subroutine, thus eliminating the overhead associated with the subroutine call. Asm-inline parameters are matched as one of the following:

- constants
- values in registers (modifiable or non-modifiable)
- register numbers (control space or general register file)

There is a library of asm-inline templates available for use. The template definitions are found in the asm subdirectory of the default include area (reached by <asm/*.h>).

NOTE

> You may need Control Space Register name definitions to invoke certain asm-inline routines. You can include these definitions by specifying #include <regnums.h>.

Pragma Support for Inlining Specification

iWarp C has been extended to include the #pragma directive to allow assembly language inlining to be passed to the optimizer. The syntax of the #pragma directive for assembly language inlining is as follows:

```
#pragma inline function
```

function is the name of a function.

The #pragma inline directive specifies that calls to *function* be expanded inline when possible instead of implemented as an actual function call.

NOTE

> Actual inlining is done by the PCO optimizer at compile time, so #pragma directives are ignored unless you use the –Q option to invoke the PCO optimizer when compiling the program.

You can also use the #pragma directive to prevent inlining which may otherwise occur. The syntax for preventing inlining is as follows:

```
#pragma noinline function
```

function is the name of a function.

Both #pragma inline and #pragma noinline apply to the entire file you are compiling, regardless of the position of the directive in the file. Specifying a function name which is not defined in the file in a #pragma directive will not result in an error .

Refer to Chapter 4, *Enhancing Program Performance,* for discussions of inter-file and intra-file inlining.