Ultra C/C++ Processor Guide

Version 2.6

www.radisys.com

Revision A • October 2002

Copyright and publication information

This manual reflects version 2.6 of Ultra C/C++. Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Corporation.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

October 2002

Table of Contents

Chapter 1: 68K

This chapter contains information specific to the 68K family of processors. The following sections are included:

- **Executive and Phase Information**
- **C/C++ Application Binary Interface Information**
- **Span Dependent Optimizations**
- **fopen() Append Bit**
- **Using Math**
- **Assembler/ Linker**
- **Assembly Language Mnemonics**
- **Additional Information for 68020, 68030, 68040, 68060, and CPU32 Processors**
- **68881/68882/68040/68060 Floating Point Mnemonics**
- **Floating Point Condition Predicates used for <cc>**
- **Constant ROM Table**

Executive and Phase Information

The Executive `-tp` option, predefined macro names for the preprocessor, and 68K-unique phase option functionality is described in this section.

Executive -tp Option

Executive `-tp` enables options dependent upon executive mode. The options for each mode are identified following.

ucc and c89 Option Mode

`-tp[=]<target>[<suboptions>]`

Specify Target Processor and Target Processor Sub-Options.

Table 1-1 ucc Target Processors

Target	Target Processor
`68000 or 68K`	68000
`68010 or 010`	68010
`68070 or 070`	68070
`68301`	68301
`68302`	68302
`68303`	68303
`68306`	68306

Table 1-1 ucc Target Processors (continued)

Target	Target Processor
68307	68307
68322	68322
68328	68328
CPU32	CPU32
CPU32+	CPU32+
68330	68330
68331	68331
68332	68332
68f333	68F333
68334	68334
68340	68340
68341	68341
68349	68349
68360	68360
68020 or 020	68020
68030 or 030	680030
68ec030 or ec030	68EC030

Table 1-1 ucc Target Processors (continued)

Target	Target Processor
`68040` or `040`	68040
`68ec040` or `ec040`	68EC040
`68lc040` or `lc040`	68LC040
`68ec060` or `ec060`	68EC060
`68lc060` or `oc060`	68LC060
`68060` or `060`	68060

Sub-options are identified in the following table.

Table 1-2 Mode -tp Sub-Options

Suboptions	Description
`sc`	Use 16-bit code references (use jumptable if necessary)
`lc`	Use 32-bit code references (default)
`sd`	Use 16-bit data references (default)
`ld`	Use 32-bit data references

compat Option Mode

Options identified in **Table 1-3** for `tp_opts` are valid for the `-tp[=]<target>[<suboptions>]` option when `<target>` is `68K`, `68020`, or `020`.

Table 1-3 compat Mode -tp Sub-Options

Suboptions	Description
`cl`	Long word code access (default when `<target>` is `68020` or `020`)
`cw`	Word code access (default when `<target>` is `68K`)
`dl`	Long word data access (default when `<target>` is `68020` or `020`)
`dw`	Word data access (default when `<target>` is `68K`)
`i`	Do not emit 68881 instructions
`j`	Prevent linker from creating jumptable

For example:

```
cc -tp=020i
-k[=]<num>[w|l][cw|cl][f]
```

Specify Target Processor

<num> is the target machine:

0 = 68000 (default)

2 = 68020

`w` causes generation of 16-bit data offsets (default 68000).

`l` causes generation of 32-bit data offsets (default 68020).

`cw` causes generation of 16-bit code references (default 68000).

`cl` causes generation of 32-bit code references (default 68020).

`f` causes generation of 68881 instructions for float/double types.

`-tp[=]<target>[<suboptions>]`

Specify the target processor sub-options to use. **Table 1-4** identifies target processors.

Table 1-4 compat Target Processor Options

Target	Target Processor
68K	MC68000/08/10/12/70
68020	MC68020/30/40/60
020	MC68020/30/40/60
CPU32	CPU32-family

Predefined Macro Names for the Preprocessor

The macro names in **Table 1-5** are predefined in the preprocessor for target systems.

Table 1-5 Macros

Macro	Target Processor
_MPF68K	All supported 68000 family
_MPF68010	68010
_MPF68070	68070

Table 1-5 Macros (continued)

Macro	Target Processor
_MPF68301	68301
_MPF68302	68302
_MPF68303	68303
_MPF68306	68306
_MPF68307	68307
_MPF68322	68322
_MPF68328	68328
_MPFCPU32	CPU32 family
_MPFCPU32PLUS	CPU32+ family
_MPF68330	68330
_MPF68331	68331
_MPF68332	68332
_MPF68F333	68f333
_MPF68334	68334
_MPF68340	68340
_MPF68341	68341
_MPF68349	68349

Table 1-5 Macros (continued)

Macro	Target Processor
_MPF68360	68360
_MPF68020	68020
_MPF68030	68030
_MPF68EC030	68EC030
_MPF68040	68040
_MPF68EC040	68EC040
_MPF68LC040	68LC040
_MPF68EC060	68EC060
_MPF68LC060	68LC060
_MPF68060	68060

Target names specify the compiler to use when writing machine- and operating system-independent programs.

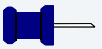
The executive defines the `_MPF` macro for the target processor as well as any processors that are generally considered subsets of the target processor. **Table 1-6** provides a few examples of this behavior.

For more information on which macros are defined for 68K target processors, run the executive in verbose and dry run modes stopping after the front end. For example, type the following line to check the defines for the 68349 target (source file not required):

```
cc -b -h -efe -tp=68349 t.c
```

This causes the executive to print a line similar to the following:

```
"cpfe -t=0 -x -v=/dd/MWOS/SRC/DEFS -v=/dd/MWOS/OS9/SRC/DEFS -n -d_UCC
  -d_SPACE_FACTOR=1 -d_TIME_FACTOR=1 -d_OSK -d_MPF68349 -d_MPFCPU32PLUS
  -d_MPF68K -d_FPF881 -d_BIG_END -o=t.i t.c"
```



Note

_MPF68349, _MPFCPU32PLUS, and _MPF68K macros are defined.

The _MPF68K macro indicates that a source file is being compiled for a Motorola 68000 family target.

Table 1-6 _MPFxxx Macro Behavior

Target	Microprocessor Family Macros Defined
68000	_MPF68K
CPU32	_MPF68K, _MPFCPU32
68020/030	_MPF68K, MPF68020
68040	_MPF68K, _MPF68020, _MPF68040
68060	_MPF68K, _MPF68020, _MPF68040, _MPF68060

68K-Unique Phase Option Functionality

Phases having unique phase option functionality on the 68K processor are:

- **Back End Options**
- **Assembly Optimizer Options**
- **Assembler Options**
- **Linker Options**

Back End Options

The back end orders the data area based static analysis of the data area objects and sorts the data based on use and size. This means that the most heavily used objects reside in the non-remote area. To do this, the back end requires information about how the object linker lays out the data area for the entire program. **Table 1-7** identifies options enabling information to be identified to the back end.

Table 1-7 Information Options

Option	Description
`-m=<non remote memory left>`	Specify that other files in the program have used some amount of the 64K data area
`-pa`	Notify the back end about using a jumptable
`-pl`	Cause references to external data to be long
`-ps`	Do not emit stack checking code
`-p68000`	Emit code for the 68000
`-p68020`	Emit code for the 68020
`-pcpu32`	Emit code for the CPU32
`-p68040`	Emit code for the 68040
`-p68060`	Emit code for the 68060

Assembly Optimizer Options

`-s<method>`

Set the peephole scheduling method

Table 1-8 Peephole Scheduling Methods

Method	Description
s	Spread dependent instructions
c	Compress floating point instructions
n	No reordering of instructions

`-t[=]<num>`

Specify Target Processor Family

Table 1-9 Assembly Optimizer Processor Numbers

Number	Assembly Optimizer Target
1	MC68000/08/10/12/70
2	CPU32-family
3	MC68020/30
4	MC68040
5	MC68060

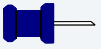
Assembler Options

`-b` Optimize Branch Sizing

Optimize sizes for span-dependent instructions. Span-dependent instructions are branches and instructions with an operand containing a PC-relative displacement.

When using this option, span-dependent instructions with branch targets or PC-relative displacements of the form `internal_label ± <constant>` cause the code produced for the instructions to be as small as possible.

Span-dependent instructions using internal labels that do not fit the above form are sized normally, according to the instruction extension. Span-dependent instructions using external labels are also sized according to the instruction extension.

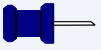


Note

`-b` changes the default size of the base displacement in extended addressing modes from long to word.

`-bt` Optimize Branch Sizing Making Branches to Externals Long

Use the largest size possible for branch or PC-relative displacements with external labels. This option assures the ability to reach the location independent of code size.



Note

`-bt` changes the default size of the base displacement in extended addressing modes from long to word.

`-j` Long Branches and PC-Relative References

Causes long branches and PC-relative references to access a jumptable when `-m0`. This option implies `-b`.

`-m<num>` Specify Machine Assembler to Use

Specify machine assembler to use:

Table 1-10 Machine Assemblers

Number	Machine Assembler
0	68000 (default)
1	68010
2	68020
3	68030
4	68040
6	68060

`-r` Require Use of Register Designator

Require register names to be distinguished from variable names by placing a percent sign (`%`) before each register name. If this option is on, a register designator (%) is necessary before a register name. The default is `off`.

`-y` Make all branches long.

Linker Options

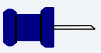
The following options are available for the linker.

`-a` Convert Out-of-Range `bsrs` and PC-Relative `leas` to Jumptable References

`bsrs` that address labels more than 32K distant are automatically converted to `jsrs` using a jumptable (in the initialized data area) that contains the desired destination address. `leas` are changed to move instructions that move the destination from a jump instruction in the jumptable. The linker automatically builds the required jumptables and includes them in the output file. This allows large programs to overcome the ±32K offset limit of `bsr` instructions without violating the operating system requirement for position independent code.

`-i` Allow Initialized Static Storage to be Included in a System Type Module

The module header used in this case is the program type module header but the type/language field reflects a type of system.



Note

Using this option does not imply that the 68K kernel initializes the static storage area for this module, only that the data structures are set in the module such that the 68K kernel or some other code could initialize the static storage area.

`-j` Print Jumptable Calculation Map

Refer to the description in `-a`.

`-t=<target>`Specify Target Module Type

Table 1-11 Target Module Type

Target	Module Type
`os9_68K`	68K

C/C++ Application Binary Interface Information

Register usage, passing arguments to functions, and language features are described in this section.

Register Usage

The compiler uses registers as identified in **Table 1-12**.

Table 1-12 Register Use

Register	Description
`d0/d1`	Parameter passing/return
`a5`	Frame/local pointer
`a6`	Static storage pointer
`a7`	Stack pointer

The compiler uses all other registers for temporaries and register variables.

Passing Arguments to Functions

When an argument is passed to a called function, the argument is in one of two places: in a register or on the stack. The called function determines the location of the argument by argument type and the order specified in the argument list. For this discussion:

- An **integral argument** is an argument of type `int`, a pointer, or a `char` or `short` converted to an `int`.
- A **double argument** is an argument of type `double` or a `float` converted to a `double`.

The first integral argument is passed in `d0`, and the second integral argument, if any, is passed in `d1`. A single double argument is passed in `d0` and `d1`, with the most significant half in `d0` and the least significant half in `d1`. Any remaining arguments are pushed on the stack. If the first argument is integral and the second is double, the integral argument is passed in `d0` and the double is passed entirely on the stack.

Parameters to functions taking variable arguments are passed differently. All variable arguments are passed on the stack.

Any `struct` arguments are copied to the next location on the stack.

The parameters are pushed on the stack in the reverse order that they appeared in the function call.

If a function is to return a value, the integral (or float) value is returned in `d0`. A double value is returned in `d0` and `d1`. If the returned value is a `struct`, then the address of the return area is passed as an argument to the callee in `a0`; the called function copies the returned `struct` to this location.

C Language Features

In conformance with the ANSI/ISO C specification, the implementation-defined areas are listed in this section. Each bulleted item contains one implementation-defined issue. The number in parentheses included with each bulleted item indicates the location in the ANSI/ISO specification where further information is provided.

For More Information

Other implementation-defined areas are included in **Language Features** chapter of the ***Using Ultra C/C++*** manual and the **Overview** chapter of the ***Ultra C Library Reference*** manual. Refer to an ANSI/ISO specification for more information.

Characters

- The number of bits in a character in the execution character set (5.2.4.2.1).

  There are 8 bits in a character in the execution character set.

Integers

- The representations and sets of values of the various integer types (6.1.2.5).

Table 1-13 Integer Type/Range

Type	Representation	Minimum / Maximum
`char`, `signed char`	8-bit 2's complement	-128 / 127
`unsigned char`	8-bit binary	0 / 255
`short int`	16-bit 2's complement	-32768 / 32767

Table 1-13 Integer Type/Range (continued)

Type	Representation	Minimum / Maximum
`unsigned short int`	16-bit binary	0 / 65535
`int`	32-bit 2's complement	-2147483648 / 2147483647
`unsigned int`	32-bit binary	0 / 4294967295
`long int`	32-bit 2's complement	-2147483648 / 2147483647
`unsigned long int`	32-bit binary	0 / 4294967295

- The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2).

  When converting a longer integer to a shorter signed integer, the least significant <n> bits of the longer integer are moved to the integer of <n> bits. The resulting value in the smaller integer is dictated by the representation. For example, if the conversion is from `int` to `short`, then the least significant 16 bits are moved from the `int` to the `short`. This value is then considered a 2's complement 16-bit integer.

  When conversion from unsigned to signed occurs with equally sized integers, the most significant bit becomes the sign bit. Therefore, if the unsigned integer is less than `0x80000000`, the conversion has no effect. Otherwise, a negative number results.

- The sign of the remainder on integer division (6.3.5).

  The result of an inexact division of one negative and one positive integer is the smallest integer greater than or equal to the algebraic quotient.

  The sign of the remainder on integer division is the same as that of the dividend.

Floating Point

- The representations and sets of values of the various types of floating-point numbers (6.1.2.5).

Table 1-14 Floating Point Number Characteristics

Type	Format	Minimum / Maximum
`float`	32 bit IEEE 754	`1.17549435e-38f / 3.40282347e38f`
`double`	64 bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`
`long double`	64 bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`

Refer to the `float.h` header file for other limits and values.

Arrays and Pointers

- The type of integer required to hold the maximum size of an array. Such as the type of the size of operator, `size_t` (6.3.3.4, 7.1.1).

  An `unsigned long int` is required to hold the maximum size of an array. `unsigned long int` is defined as `size_t` in `ansi_c.h`.

- The result of casting a pointer to an integer or vice versa (6.3.4).

  Since pointers are treated much like unsigned long integers, the integer is promoted using the usual promotion rules to an unsigned long. The sign bit propagates out to the full 32-bit width.

- The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (6.3.6, 7.1.1).

  A `signed long int` is required to hold the difference between two pointers to elements of the same array. `long int` is defined as `ptrdiff_t` in `ansi_c.h`.

Registers

- The extent to which objects can actually be placed in registers by use of the register storage-class specifier (6.5.1).

  The compiler automatically makes decisions regarding which objects are placed in registers, giving no special storage consideration to the register storage-class.

Structures, Unions, Enumerations, and Bit-Fields

- The padding and alignment of members of structures (6.5.2.1). This should present no problem unless binary data written by one implementation are read by another.

  Table 1-15 shows the alignment of the various objects within a structure. Required padding is supplied if the next available space is not at the correct alignment for the object. For example, a structure declared as:

  ```
  struct {
     char mem1;
     long mem2;
  };
  ```

  would be a four-byte structure (32-bit), including one byte for `mem1`, two bytes for `mem2`, and one byte of padding to complete the structure.

  Non-character structure members and sub-structures containing non-character members are aligned on an even byte boundary. Character structure members do not have alignment restrictions.

Table 1-15 Alignment Table

Type	Alignment Requirement
`char`	1
`short`	1
`int`	2
`long`	2
`float`	2
`double`	2

- Whether “plain” `int` bit field is treated as a signed `int` or as an unsigned `int` bit field (6.5.2.1).

  A plain `int` bit field is treated as a signed `int` bit field.
- The order of allocation of bit fields within a unit (6.5.2.1).

  The bit fields are allocated from most significant bit to least significant bit.
- Whether a bit field can straddle a storage-unit boundary (6.5.2.1).

  Bit fields may straddle a storage unit. Bit fields are allocated end-to-end until a non-bit field member is allocated or 32-bit size is executed.
- The integer type chosen to represent the values of an enumeration type (6.5.2.2).

  `Enum` values are represented in 32-bit two's complement integers.

Preprocessing Directives

- Whether the value of a single-character, character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (6.8.1).

  The value of a single-character, character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. This character constant may be a negative value.

- The method for locating includable source files (6.8.2).

  This method is described in the ***Using Ultra C/C++*** manual in the **Using the Executive** chapter.

- The support of quoted names for includable source files (6.8.2).

  Quoted names are supported for `#include` preprocessing directives.

- The mapping of source file character sequences (6.8.2).

  The mapping of source file character sequences is one-to-one. The case-sensitivity of file names depends on the file system being used.

_asm() Register Pseudo Functions

`_asm()` uses register pseudo functions as identified in **Table 1-16**.

Table 1-16 _asm() Register Pseudo Functions

Register	Description
`__reg_data`	Any data register
`__reg_addr`	Any non-dedicated address register
`__reg_gen`	Any data registers or non-dedicated address register
`__reg_float`	Any floating point register
`__reg_d[0-7]`	Individual data registers as indicated by name
`__reg_a[0-7]`	Individual address registers as indicated by name

The address registers `a5`, `a6`, and `a7` are defined by the compiler to be the frame pointer, static storage pointer, and stack pointer respectively. Their pseudo functions cannot have parameters and the user of `_asm()` must be responsible for any attempts to modify these registers.

Span Dependent Optimizations

The compiler performs **branch shortening and PC-relative addressing mode shortening**.

- Branch shortening reduces the instruction size on branch instructions when the distance to the destination is known to be within certain limits.
- PC-relative addressing mode shortening reduces the instruction size for the PC-relative addressing mode when the label is within certain limits.

Methods for Reducing Compiled Code Size

Methods of reducing compiled code size include overriding compiler size defaults, I-code linking, and use of provided, small library functions.

For More Information

For information on reducing compiled code size via I-code linking and use of provided small library functions, refer to the chapter describing compiling in the ***Using Ultra C/C++*** manual.

Overriding Compiler Size Defaults

When compiling a small model program with code references spanning references of less than 32K, the `sc` specification of the target processor option (`-tp`) directs the compiler to impose short external references to generate more efficient code. When compiling a program module (as opposed to a driver or system module) the assembler or linker may patch the code and create a jump table in the data area to enable

references spanning more than 32K. This is a reasonable way to use the compiler when compiling during development. Compile each source file to an ROF and link at the end with the object linker.

User Program Modules

An example of compiling a source file to an ROF (using `ucc` as the executive option mode) and linking user program modules follows.

```
Cc file1.c -tp=68K,sc -eas=RELS      **compile file #1 with short externs**
cc file2.c -tp=68K,sc -eas=RELS      **compile file #2 with short externs**
cc file3.c -tp=68K,sc -eas=RELS      **compile file #3 with short externs**
** link in default libraries with long externals
   and create a jumptable if necessary **
         cc -tp=68K,sc -f=test RELS/file1.r RELS/file2.r RELS/file3.r
```

System and Non-program Modules

Producing smaller code for system or non-program modules differs from the procedure used for user program modules. System and non-program module types disallow initialized data, therefore the object linker jumptable is disallowed. When compiling a program with references spanning less than 32K, the `sc` specification of the target processor option (`-tp`) directs the compiler to impose short external references to generate more efficient code. Use the `sc` specifier on each file in the module until the linker generates an **out of range** error on a file, then remove the `sc` specifier for this file.

fopen() Append Bit

The implementation of the append mode for `fopen()` requires that the device descriptor for the device that the program runs to have the append bit set. Most descriptors do not have this bit set.

The append bit is set in the device mode capabilities byte of an RBF descriptor. The append bit is bit number 4 (0x10).

For More Information

For more information on using `moded`, refer to the `moded` utility description in the user manual for your operating system.

`moded` can be used to set this bit. The following example illustrates setting the append bit:

```
$ moded h0

     OS-9/68000 Module Editor
     Copyright 1987 Microware Systems Corp.
     Type ? for editing help message
moded: e
      descriptor name              :        h0 =
      file manager name            :       RBF =
      device driver name           :    rbvccs =
      port address                 : $fff47000 =
      irq vector                   :       101 =
      irq level                    :         2 =
      irq priority                 :         5 =
      device mode capabilities     :       $a7 = $b7
      device class                 :       $01 = .
moded: w
moded: q
```

Once the capabilities byte is properly set, integrate the new descriptor into the boot procedure in place of the old descriptor by making a new bootfile. Refer to the `os9gen` utility in the ***Utilities Reference*** manual or, use `moded` directly on the 68K boot file:

```
moded h0 -f=/h0/os9boot
```

Using Math

Ultra C/C++ generates one model of floating point code. If the floating point coprocessor is absent in the system, a floating point emulation software module makes it appear as if it exists.

In K&R source mode, existing ROFs or libraries compiled to use the software math must link with the old `math.l` library when making applications that use them. Not linking with the old `math.l` library requires recompilation of any ROFs or libraries that used `math.l`, `math881.l`, or the math trap handler, `math` and `math881`.

To use the floating point math functions on a processor that does not have a math coprocessor, perform the following steps:

Step 1. Add the floating point module name to the bootfile as identified in **Table 1-17**.

Table 1-17 Floating Point Module

Target	68K Version	Floating Point Module
68000/20/30	2.4 or greater	`fpu`
68040	2.4	`fpu040`
68040	3.0 or greater	`fpsp040`
68060	3.0 or greater	`fpsp060`

Step 2. Add the floating point module name to the extension module list in the `Init` module and remake the extension module list.

Step 3. Remake the bootfile and reboot the system.

Assembler/ Linker

The assembler uses standard Motorola instruction syntax as modified in the **Assembly Language Mnemonics** section in this chapter.

ROF Edition Number

The `r68` assembler supports ROF Edition #15.

External References

ROF Edition #15 is only capable of representing limited expressions involving external references. These expressions can consist only of simple addition and subtraction operations involving two operands at most. The following expression forms involving external references are supported. All other forms are illegal.

```
External + Absolute
External - Absolute
External - External
```

The linker performs subtraction by negating one operand and adding it to the other operand. This method can cause problems on signed values of either word or byte length as the linker may report over/underflow errors. Therefore, expressions involving external names should not be too complex.

For More Information

Refer to the ROF Edition Number 9 Format section in the **Assembler and Object Code Linker Overview** chapter of the ***Using Ultra C/C++*** manual.

Symbol Biasing

The linker does not bias code symbols for the 68000 target. Data symbols are biased only for program and trap handler type modules. The bias value applied to data symbols is `-32768` (`-0x8000`). Neither code nor data symbols are biased for 68000 raw code.

Assembly Language Mnemonics

The Mnemonics table in this section lists the mnemonic names used on the 68K processors with their meanings. Many of the mnemonics include one or more optional symbols indicating conditions. Symbols, when present, modify the meaning of the mnemonic instructions. Additionally, instruction conventions are used in the mnemonics table. The following tables identify and define reserved registers and addressing modes, symbols, and instruction conventions used in the syntax of the mnemonics table.

Registers

Table 1-18 Reserved Register Names

Register Name	Description
`An`	Address register n
`Dn`	Data register n
`pc or pcr`	Program counter
`sr`	Status register
`ccr`	Condition codes
`ssp`	Supervisor stack pointer
`usp`	User stack pointer

Register notation n represents 0 through 7.

Addressing Mode Syntax Definitions

Table 1-19 Addressing Mode Syntax

Addressing Mode	Description
`Dn`	Data register direct
`An`	Address register direct
`Rn`	Data or address register direct
`Xn.s`	Index register n (either address or data) `.s` indicates the index register size. It is either `.w` (word) or `.l` (long, default)
`(An)`	Address register indirect
`(An)+`	Address register indirect with postincrement
`-(An)`	Address register indirect with predecrement
`d(An)`	Address register indirect with offset
`d(An,Xn.s)`	Address register indirect with index
`(xxx).w`	Absolute short
`(xxx).l`	Absolute long
`d(pc)`	Program counter indirect with offset
`d(pc,Xn.s)`	Program counter indirect with index
`#xxx`	Immediate data

Symbols

The symbol <cc> indicates condition codes as identified in the following table.

Table 1-20 Condition Codes Used with Assembler Instructions

Mnemonic	Condition	Description
cc	!C	Carry clear
cs	C	Carry set
eq	Z	Equal
ge	N.V+!N.!V	Greater than or equal
gt	N.V.Z+!N.!V.!Z	Greater than
hi	!C.!Z	Higher
hs	!C	Higher or the same
le	Z+N.!V+!N.V	Less than or equal
lo	C	Lower
ls	C+Z	Lower or the same
lt	N.!V+!N.V	Less than
mi	N	Minus
ne	!Z	Not equal

Table 1-20 Condition Codes Used with Assembler Instructions (continued)

Mnemonic	Condition	Description
pl	!N	Plus
vc	!V	Overflow clear
vs	V	Overflow set

N = negative
Z = zero
V = overflow
C = carry

Instruction Conventions

Instruction mnemonics shown in the following table are used in the Mnemonics table.

Table 1-21 Instruction Mnemonics

Mnemonic	Description
<cc>	Condition code
<data>	Immediate data of appropriate size
.s	Indicates .w, .l, or .b. The default is .w, if the size is not explicitly given.
<ea>	Any legal addressing mode for the instruction
<d>	Shift direction may be l for left or r right

Mnemonics Table

Table 1-22 Mnemonic Summary

Mnemonic	Description
`abcd Dy,Dx`	Add decimal with extend register
`abcd -(Ay),-(Ax)`	Add decimal with extend memory
`add.s <ea>,Dn`	Add binary register
`add.s Dn,<ea>`	Add binary memory
`adda.s <ea>,An`	Add address (`.w` or `.l` only)
`addi.s #<data>,<ea>`	Add immediate
`addq.s #<data>,<ea>`	Add quick
`addx.s Dy,Dx`	Add extended register
`addx.s -(Ay),-(Ax)`	Add extended memory
`and.s <ea>,Dn`	AND logical register
`and.s Dn,<ea>`	AND logical memory
`andi.s #<data>,<ea>`	AND immediate
`andi #<data>,ccr`	AND immediate to condition code
`andi #<data>,sr`	AND immediate to status register
`as<d>.s Dx,Dy`	Arithmetic shift register

Table 1-22 Mnemonic Summary (continued)

Mnemonic	Description
as<d>.s #<data>,Dy	Arithmetic shift immediate register
as<d> w <ea>	Arithmetic shift memory
bcc <label>	Conditional branch word displacement. cc represents the branch condition code.
bcc.s <label>	Conditional branch byte displacement. cc represents the branch condition code.
bchg.s Dn,<ea>	Test bit and change register (.b or .l)
bchg.s #<data>,<ea>	Test bit and change immediate (.b or .l)
bclr.s Dn,<ea>	Test bit and clear register (.b or .l)
bclr.s #<data>,<ea>	Test bit and clear immediate (.b or .l)
bra <label>	Branch word displacement
bra.s <label>	Branch byte displacement
bset.s Dn,<ea>	Test bit and set register (.b or .l)
bset.s #<data>,<ea>	Test bit and set immediate (.b or .l)
bsr <label>	Branch subroutine word displacement
bsr.s <label>	Branch subroutine byte displacement
btst.s Dn,<ea>	Test bit register (.b or .l)

Table 1-22 Mnemonic Summary (continued)

Mnemonic	Description
`btst.s #<data>,<ea>`	Test bit immediate (`.b` or `.l`)
`chk <ea>,Dn`	Check register against bounds
`clr.s <ea>`	Clear operand
`cmp.s <ea>,Dn`	Compare data register
`cmpa.s <ea>,An`	Compare address register
`cmpi.s #<data>,<ea>`	Compare immediate
`cmpm.s (Ay)+,(Ax)+`	Compare memory
`dbcc dn,<label>`	Test condition, decrement and branch
`divs <ea>,Dn`	Signed divide
`divu <ea>,Dn`	Unsigned divide
`eor.s Dn,<ea>`	Exclusive OR
`eori.s #<data>,<ea>`	Exclusive OR immediate
`eori #<date>,ccr`	Exclusive OR condition code
`eori #<data>,sr`	Exclusive OR status register
`exg Rx,Ry`	Exchange registers
`ext.s Dn`	Sign extend (`.w` or `.l`)
`jmp <ea>`	Jump

Table 1-22 Mnemonic Summary (continued)

Mnemonic	Description
`jsr <ea>`	Jump to subroutine
`lea <ea>,An`	Load effective address
`link An, #<displacement>`	Link and allocate
`ls<d>.s Dx,Dy`	Logical shift data
`ls<d>.s #<data>,Dy`	Logical shift immediate
`ls<d> <ea>`	Logical shift memory
`move.s <ea>,<ea>`	Move from source to destination
`move ccr,<ea>`	Move from condition codes
`move <ea>,ccr`	Move to condition codes
`move <ea>,sr`	Move to status register
`move sr,<ea>`	Move from status register. This is privileged. Avoid this instruction in programs that are to execute in user-state.
`move usp,An`	Move from user stack pointer
`move An,usp`	Move to user stack pointer
`movea.s <ea>,An`	Move address (`.w` or `.l`)

Table 1-22 Mnemonic Summary (continued)

Mnemonic	Description

`movem.s <ea>, <reg list>`	Move multiple
	`<reg list>:` `CrnRegisterrx-ry`
	Consecutive registers/Register delimiter `#<expr>` Register list mask
	Examples:
	`d0` `d0` only
	`d0/d4/a5` `d0,d4,a5`
	`d0-d7/a0-a5` `d0` through `d7`, `a0` through `a5`

`movep.s dx,d(Ay)`	Move peripheral data (`.w` or `.l`) from register to memory
`movep.s d(Ay),dx`	Move peripheral data (`.w` or `.l`) from memory to register
`moveq.l #<data>,dn`	Move quick
`muls <ea>,Dn`	`Signed multiply`
`mulu <ea>,Dn`	Unsigned multiply
`nbcd <ea>`	Negate decimal with extend
`neg.s <ea>`	Negate
`negx.s <ea>`	Negate with extend
`nop`	No operation
`not.s <ea>`	Logical complement

Table 1-22 Mnemonic Summary (continued)

Mnemonic	Description
`or.s <ea>,Dn`	Inclusive OR register
`or.d Dn,<ea>`	Inclusive OR memory
`ori.s #<data>,<ea>`	Inclusive OR immediate
`ori #<data>,ccr`	Inclusive OR condition codes
`ori #<data>,sr`	Inclusive OR status register
`pea <ea>`	Push effective address
`reset`	Reset external devices
`ro<d>.s Dx,Dy`	Rotate without extend register
`ro<d>.s #<data>,dy`	Rotate without extend immediate
`ro<d> <ea>`	Rotate without extend memory
`rox<d>.s Dx,Dy`	Rotate with extend register
`rox<d>.s #<data>,dy`	Rotate with extend immediate
`rox<d> <ea>`	Rotate with extend memory
`rte`	Return from exception
`rtr`	Return and restore condition codes
`rts`	Return from subroutine
`sbcd Dy,Dx`	Subtract decimal with extended register

Table 1-22 Mnemonic Summary (continued)

Mnemonic	Description
`sbcd -(Ay),-(Ax)`	Subtract decimal with extended memory
`s<cc> <ea>`	Set according to conditional registers. `cc` represents the branch condition code.
`stop #<data>`	Load and stop
`sub.s <ea>,Dn`	Subtract binary register
`sub.s Dn,<ea>`	Subtract binary memory
`suba.s <ea>,An`	Subtract address (`.w` or `.l`)
`subi.s #<data>,<ea>`	Subtract immediate
`subq.s #<data>,<ea>`	Subtract quick
`subx.s Dy,Dx`	Subtract with extend register
`swap Dn`	Swap register halves
`tas <ea>`	Test and set operand
`trap #<vector>`	Trap
`trapv`	Trap on overflow
`tst.s <ea>`	Test operand
`unlk An`	Unlink

Additional Information for 68020, 68030, 68040, 68060, and CPU32 Processors

The assembler can process all 680x0 instructions and syntax. However, there is a superset of 68020, 68030, 68040, 68060, and CPU32 instructions. The information presented in this section is in addition to the information in the preceding sections.

Floating Point Numbers

Specify floating point numbers in the following format (the exponent may be specified with either an uppercase or lowercase e):

```
[-]digits[.digits[e[±][digits]
```

The range for floating point numbers is `±2.2*10^-308` to `±1.8*10^308`. For example:

```
-1.             10.5          1e5
-1.36E-124      106352.671e4  123456789
```

Assembly Language Mnemonics

Table 1-23 Reserved Register Names

Register Name	Description
`sfc`	Source function code
`dfc`	Destination function
`cacr`	Cache control register

Table 1-23 Reserved Register Names (continued)

Register Name	Description
vbr	Vector base register
caar	Cache address register
msp	Master stack pointer
isp	Interrupt stack pointer
tc	MMU translation control register
itt0	Instruction transparent translation register 0
itt1	Instruction transparent translation register 1
dtt0	Data transparent translation register 0
dtt1	Data transparent translation register 1
mmusr	MMU status register
urp	User root pointer
srp	Supervisor root pointer
iacr0	Instruction access control register 0
iacr1	Instruction access control register 1
dacr0	Data access control register 0
dacr1	Data access control register 1

Table 1-23 Reserved Register Names (continued)

Register Name	Description
vbr	Vector base register
caar	Cache address register
msp	Master stack pointer
isp	Interrupt stack pointer
tc	MMU translation control register
itt0	Instruction transparent translation register 0
itt1	Instruction transparent translation register 1
dtt0	Data transparent translation register 0
dtt1	Data transparent translation register 1
mmusr	MMU status register
urp	User root pointer
srp	Supervisor root pointer
iacr0	Instruction access control register 0
iacr1	Instruction access control register 1
dacr0	Data access control register 0
dacr1	Data access control register 1

Table 1-23 Reserved Register Names (continued)

Register Name	Description
`buscr`	Bus control register
`pcr`	Data access control register 1

In the following definitions, `(disp)` is an expression. If `disp` is a symbol ending with `.w` or `.l`, the parentheses are required to distinguish the symbol name from the size extension. `*S` is an optional scale factor. If `*S` is used, it must be `*1`, `*2`, `*4`, or `*8`.

Table 1-24 Addressing Mode Syntax

Addressing Mode	Description
`((disp).w,An)`	Address register indirect with offset
`((disp).s,An,Xn.s*S)`	Address register indirect with index (base displacement)
`([(disp).s,An],Xn.s*S,(disp).s)`	Memory indirect post-indexed
`([(disp).s,An,Xn.s*S],(disp).s)`	Memory indirect pre-indexed

For the memory indirect addressing modes, all four parameters are optional. The assembler encodes the proper modes to indicate the suppression of the missing parameters. The assembler accepts the 68000 addressing modes `d(An)` and `d(An,Xn.s)`. In this case, the 68020 brief format extension format is generated. If the operand begins with a left parenthesis `(`, the 68020 full format extension format is always generated.

Table 1-25 Mnemonic Summary

Mnemonic	Description
b<cc>.b <label>	Conditional branch byte displacement
b<cc>.w <label>	Conditional branch word displacement
b<cc>.l <label>	Conditional branch long displacement
bfchg <ea>{offset:width}	Test bit field and change
bfclr <ea>{offset:width}	Test bit field and clear
bfexts <ea>{offset:width},Dn	Extract bit field signed
bfextu <ea>{offset:width},Dn	Extract bit field unsigned
bfffo <ea>{offset:width},Dn	Find first one in bit field
bfins Dn,<ea>{offset:width}	Insert bit field
bfset <ea>{offset:width}	Set bit field
bftst <ea>{offset:width}	Test bit field
bgnd	Enter background mode
bkpt #<data>	Breakpoint
bra.b <label>	Branch byte displacement
bra.w <label>	Branch word displacement
bra.l <label>	Branch long displacement
bsr.b <label>	Branch subroutine byte displacement

Table 1-25 Mnemonic Summary (continued)

Mnemonic	Description
`bsr.w <label>`	Branch subroutine word displacement
`bsr.l <label>`	Branch subroutine long displacement
`callm #<data>,<ea>`	Call module
`cas Dc,Du,<ea>`	Compare and swap with operand
`cas2 Dc1:Dc2,Du1:Du2,(Rn1):(Rn2)`	Compare and swap with operand
`chk.l <ea>`	Check register against bounds
`chk2.s <ea>,Rn`	Check register against bounds (`.b`, `.w`, or `.l`)
`cinv`	Invalidate cache lines
`cmp2.s <ea>,Rn`	Compare register against bounds (`.b`, `.w`, or `.l`)
`cpush`	Push and invalidate cache lines
`divs.w <ea>,Dn`	Signed divide - 32/16→16r:16q
`divs.l <ea>,Dq`	Signed divide - 32/32→32q
`divs.l <ea>,Dr:Dq`	Signed divide - 64/32→32r:32q
`divsl.l <ea>,Dr:Dq`	Signed divide - 32/32→32r:32q
`divu.w <ea>,Dn`	Unsigned divide - 32/16→16r:16q
`divu.l <ea>,Dq`	Unsigned divide - 32/32→32q
`divu.l <ea>,Dr:Dq`	Unsigned divide - 64/32→32r:32q

Table 1-25 Mnemonic Summary (continued)

Mnemonic	Description
`divul.l <ea>,Dr:Dq`	Unsigned divide - 32/32→32r:32q
`extb.l Dn`	Extend byte to longword
`link.l An, #<displacement>`	Link and allocate (long displacement)
`lpstop`	Low-power stop
`movec Rc,Rn`	Move from control register †
`movec Rn,Rc`	Move to control register †
`moves.s Rn,<ea>`	Move to address space
`moves.s <ea>,Rn`	Move from address space
`move16`	Move 16 byte block
`muls.w <ea>,Dn`	Signed multiply 16 x 16→32
`muls.l <ea>,Dn`	Signed multiply 32 x 32→32
`muls.l <ea>,Dh:Dl`	Signed multiply 32 x 32→64
`mulu.w <ea>,Dn`	Unsigned multiply 16 x 16→32
`mulu.l <ea>,Dn`	Unsigned multiply 32 x 32→32
`mulu.l <ea>,Dh:Dl`	Unsigned multiply 32 x 32→64
`pack -(Ax),-(Ay),#<adjust>`	Pack BCD
`pack Dx,Dy,#<adjust>`	Pack BCD
`pflush`	Flush entry in the ATC
`pload`	Load an entry into the ATC

Table 1-25 Mnemonic Summary (continued)

Mnemonic	Description
`plpar (An)`	Load physical address (read)
`plpaw (An)`	Load physical address (write)
`ptest`	Test a logical address
`rtd #<displacement>`	Return and deallocate
`rtm Rn`	Return from module
`trap <cc>`	Trap on condition
`trap <cc>.w #<data>`	Trap on condition
`trap <cc>.l #<data>`	Trap on condition
`unpk -(Ax),-(Ay),#<adjust>`	Unpack BCD
`unpk Dx,Dy,#<adjust>`	Unpack BCD

† Valid registers for `Rc: sfc, dfc, cacr, usp, vbr, caar, msp, isp, tc, itt0, itt1, dtt0, dtt1, mmusr, urp, buscr, pcr, srp, iacr0, iacr1, dacr0, and dacr1.`

68881/68882/68040/68060 Floating Point Mnemonics

The assembler recognizes instructions and addressing modes referencing the 68881 floating point coprocessor.

The following register names are reserved for referencing the 68881. They may not be redefined or used out of context, unless you use the assembler `-r` option:

`FPn` Floating point register (0-7)

`FPcr` Floating point control register

`FPsr` Floating point status register

`FPiar` Floating point instruction address register

The assembler recognizes the following floating operand data format extensions:

`B` Byte integer

`W` Word integer

`L` Longword integer

`S` Single-precision real

`D` Double-precision real

`X` Extended precision real

The `P` (packed decimal real) data format is not supported.

Floating point constants may be specified when a floating point instruction indicates immediate addressing. Floating point constants can be given in decimal format or left-justified hexadecimal format. The size of the immediate data value is determined from the data format extension given on the floating point instruction. Single-precision values are stored internally as double-precision and converted to single-precision before storing into the instruction. Extended precision constants can be given only as hexadecimal values.

Floating Point Examples

Table 1-26 Floating Point Examples

Example	Description
`fadd.l #10,fp0`	Long integer value of `10` is converted to extended and added to `fp0`
`fadd.l #0x10,fp0`	Same as above
`fadd.s #5,fp0`	Single-precision value of 5 is converted to extended and added to `fp0`
`fadd.s #0x40A0,fp0`	Same as above
`fadd.d #1.3e4,fp0`	Double-precision value of `130000` is converted to extended precision and added to `fp0`.
`fadd.d #0x40C964,fp0`	Same as above
`fadd.x #0x3ff,fp0`	Extended value of `3FF0000000000000` is added to `fp0`

Floating point expressions are not supported.

The 68881 instruction mnemonic summary uses the notation:

`<data>`Immediate data of appropriate size

`<ea>`Any legal addressing mode for the instruction

In the 68881 instruction mnemonic summary, the following format describes the instructions:

Mnemonic	Format	Syntax	Description
`<inst>`	`b,w,l,s,d or x`	`<syntax>`	`<description of instruction>`

For example:

`fadd`	`b,w,l,s,d,x` `x`	`<ea>,FPn` `FPm,FPn`	`Add`

The preceding example describes the `fadd` instruction. It shows that the `add` instruction may take the form `fadd.b`, `fadd.w`, or `fadd.l` and use the syntax `<ea>,FPn`. However, `fadd.x` may use the syntax `FPm,FPn`. For example:

```
fadd.x fp0,fp1
fadd.x #5,fp0
```

Dyadic Instructions

Dyadic floating point instructions require two source operands.

- The first source operand can be any effective address or a floating point register
- The second source operand must be a floating point register

The results of the operation are stored in this same register. The general format of the dyadic instructions is as follows:

Mnemonic	Format	Syntax
`<dyadic inst>`	`b,w,l,s,d,x` `x`	`<ea>,FPn` `FPm,FPn`

The following 68881 floating point instructions use the above dyadic syntax:

Table 1-27 Dyadic 68881 Floating Point Instructions

Mnemonic	Description
`fadd`	Add
`fcmp`	Compare
`fdiv`	Divide
`fmod`	Module remainder
`fmul`	Multiply
`frem`	IEEE remainder
`fscale`	Scale exponent
`fsgldiv`	Single precision divide
`fsglmul`	Single precision multiply
`fsub`	Subtract

Monadic Instructions

Monadic floating point instructions require only one source operand. These instructions can specify a source and destination operand. The source operand can be any effective address or a floating point register. The operation is performed on the source operand and the result is placed in the destination operand, which is always a floating point register. If the source operand is

- An effective address, any operand format can be given.
- A floating point register, only the x format is allowed.

If no destination floating point register is given, the operation is performed on the given register and the resulting value is stored in the same register.

The general format of the monadic instructions is as follows:

Mnemonic	Format	Syntax
`<monadic inst>`	`b,w,l,s,d,x`	`<ea>,FPn`
	`x`	`FPm,FPn`
	`x`	`FPn`

The following 68881 floating point instructions use this monadic syntax:

Table 1-28 Monadic 68881 Floating Point Instructions

Mnemonic	Description
`fabs`	Absolute value
`facos`	Arc cosine
`fasin`	Arc sine
`fatan`	Arc tangent
`fatanh`	Hyperbolic arc tangent
`fcos`	Cosine

Table 1-28 Monadic 68881 Floating Point Instructions (continued)

Mnemonic	Description
`fcosh`	Hyperbolic cosine
`fetox`	e^x
`fetoxm1`	$e^{(x-1)}$
`fgetexp`	Get exponent
`fgetman`	Get mantissa
`fint`	Integer part
`fintrz`	Integer part; round to zero
`flog10`	Log_{10}
`flog2`	Log_2
`flogn`	Log_e
`flognp1`	Log_{e-1}
`fneg`	Negate
`fsin`	Sine
`fsinh`	Hyperbolic sine
`fsqrt`	Square root
`ftan`	Tangent
`ftanh`	Hyperbolic tangent

Table 1-28 Monadic 68881 Floating Point Instructions (continued)

Mnemonic	Description
`ftentox`	10^x
`ftwotox`	2^x

Data Movement Instructions

Table 1-29 Data Movement Instructions

Mnemonic	Format	Syntax	Description
`fmove`	`x`	`FPm,FPn`	Floating move
	`b,w,l,s,d,x`	`<ea>,FPn`	
	`b,w,l,s,d,x`	`FPm,<ea>`	
	`l`	`<ea>,FPcr`	
	`l`	`FPcr,<ea>`	

Table 1-29 Data Movement Instructions (continued)

Mnemonic	Format	Syntax	Description
fmovecr		#ccc,FPn	Move from constant ROM
fmovem	l,x l,x x x	<flist>,<ea> <ea>,<flist> Dn,<ea> <ea>,Dn	Move multiple floating registers. <flist> is a sequence of floating registers. A slash (/) separates each register in the list. Consecutive registers may be grouped by using a hyphen (-) between the beginning and ending registers. If l format is given, only FPCR, FPSR, or FPIAR are allowed. If x is given, only FP0-FP7 are allowed.

Program Control Instructions

Table 1-30 Program Control Instructions

Mnemonic	Syntax	Description
fb<cc>	<label>	Branch on floating condition †
fdb<cc>	Dn,<label>	Decrement and branch on floating condition †
fnop		No operation
fs<cc>	<ea>	Set on floating condition †
ftst	<ea>	Test floating operand

† This instruction uses floating point condition predicates for <cc>.

System Control Operations

Table 1-31 System Control Operations

Mnemonic	Syntax	Description
`frestore`	`<ea>`	Restore internal state
`fsave`	`<ea>`	Save internal state
`ftrap<cc>`	`#<data>`	Trap on floating condition †

† This instruction uses floating point condition predicates for <*cc*>.

The `fsincos` instruction is a special dual monadic instruction. Consequently, two operands are given:

Mnemonic	Format	Syntax	Description
`fsincos`	`b,w,l,s,d,x`	`<ea>,FPc:FPs`	Simultaneous sine and cosine.
	`x`	`FPm,FPc:FPs`	FPc is the resulting cosine value, FPs is the resulting sine value.

Floating Point Condition Predicates used for <cc>

Table 1-32 Floating Point Condition Predicates Used for <cc>

Mnemonic	Description
`eq`	Equal
`f`	False
`ge`	Greater than or equal
`gl`	Greater or less than
`gle`	Greater or less than or equal
`gt`	Greater than
`le`	Less than or equal
`lt`	Less than
`ne`	Not equal
`nge`	Not greater than or equal
`ngl`	Not greater or less than
`ngle`	Not greater or less than or equal
`ngt`	Not greater than

Table 1-32 Floating Point Condition Predicates Used for <cc> (continued)

Mnemonic	Description
`nle`	Not less than or equal
`nlt`	Not less than
`oge`	Ordered greater than or equal
`ogl`	Ordered greater or less than
`ogt`	Ordered greater than
`ole`	Ordered less than or equal
`olt`	Ordered less than
`or`	Ordered
`seq`	Signaling equal
`sf`	Signaling false
`sne`	Signaling not equal
`st`	Signaling true
`t`	True
`ueq`	Unordered or equal
`uge`	Unordered or greater than or equal
`ugt`	Unordered or greater than
`ule`	Unordered or less than or equal

Table 1-32 Floating Point Condition Predicates Used for <cc> (continued)

Mnemonic	Description
`ult`	Unordered or less than
`un`	Unordered

Constant ROM Table

The following are offsets into the 68881 constant ROM that contain useful values:

Table 1-33 Constant ROM Table

Offset	Constant
$00	PI
$0B	$Log_{10}(2)$
$0C	e
$0D	$Log_2(e)$
$0E	$Log_{10}(e)$
$0F	0.0
$30	ln_2
$31	ln_{10}
$32	10^0
$33	10^1
$34	10^2
$35	10^4
$36	10^8

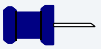
Table 1-33 Constant ROM Table (continued)

Offset	Constant
$37	10^{16}
$38	10^{32}
$39	10^{64}
$3A	10^{128}
$3B	10^{256}
$3C	10^{512}
$3D	10^{1024}
$3E	10^{2048}
$3F	10^{4096}

If stack checking is inappropriate for the module being created, define the following:

- 32-bit global called `_stklimit` (initialized to a large positive value if possible)
- Function called `_stkhandler` that just returns to its caller

A function called `_stkoverflow` is called when the stack appears to overflow. If a non-static function called `_stkoverflow` resides in an ROF that is linked to make the object module, that function is called instead of the default function. The default function writes a message to the standard error path and causes the program to exit with a status of one. `_stkoverflow` neither accepts parameters nor returns a value.

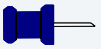


Note

If `_stkoverflow` is inappropriate for your application, consider writing a function to handle stack overflow.

The function that checks for stack overflow, `_stkhandler`, may be revised. This may be necessary if stack checking is inapplicable to the module that calls the library functions. `_stkhandler()` neither accepts parameters nor returns a value.

The following source files (Default Stack Handler Function and Default Stack Overflow Message and Exit) contain the code for the stack checking and error exiting routines for 68K.



Note

Stack handler code must be compiled with stack checking turned off (`-r` in ucc mode).

Default Stack Handler Function

```
/* typedef to the 1 byte unit so pointer arithmetic is easy */
typedef unsigned char byte;

static byte *__asm_get_stack();     /* get current stack pointer */
static void __asm_put_stack(byte *);/* set current stack pointer */

/*
   _stkhandler()
   Checks for stack overflow. _stklimit will be set with the negative
   value of the number of bytes that the function needs. This function
   does not take too much advantage of old information in the globals
   because old stack checking code does not update it.
*/
void _stkhandler()
{
   byte *sp;           /* stack pointer */
```

```
    /*
        Figure out what stack limit should really be.
        This is necessary because we may have gotten here after an
        arbitrary number of calls to the old stack checking code which
        only modifies _stbot.
    */
    if ((_stklimit = (sp = __asm_get_stack())) - (byte *)_stbot) < 0) {
        _stbot = sp;
        _stklimit = 0;

        if (sp <= (byte *)_mtop) {    /* overflow? */
            __asm_put_stack(sp - 256);
            _stklimit = 256;
            _stkoverflow();
        }

        _maxstack = (byte *)_sttop - sp;/* reset maximum so far */
    }
}

static byte *__asm_get_stack(void)
{
    register byte *stack_ptr;

    _asm(“ move.l %0,%1", __reg_a7(),
                          __reg_gen(__obj_assign(stack_ptr)));

    return stack_ptr;
}

static void __asm_put_stack(new_sp)
byte *new_sp;
{
    _asm(“ move.l %1,%0", __reg_a7(), __reg_gen(new_sp));
}
```

Default Stack Overflow Message and Exit

```
static const char ovf[] = "**** Stack Overflow ****\n";

/*
   _stkoverflow()
   print a message and exit
*/
void _stkoverflow()
{
   /* write message above to stderr and exit */
   u_int32 size = sizeof(ovf);

   if (stderr->_flag & _WRITE)
     _os_writeln(_fileno(stderr), (void *)ovf, &size);
   _os_exit(EOS_STKOVF);
}
```

Chapter 2: ARM

This chapter contains information specific to the ARM family of processors. The following sections are included:

- **Executive and Phase Information**
- **C/C++ Application Binary Interface Information**
- **_asm() Register Pseudo Functions**
- **Assembler/ Linker**
- **Stack Checking**

Executive and Phase Information

Executive `-tp` enables specific options dependent upon executive mode. Processors and sub-options for `ucc` and `c89` option modes are identified in this section.

Executive -tp Option

`-tp[=]<target>{[,]<suboptions>}` Specify Target Processor and Target Processor Options

Specify the target processor `<target>` and target processor sub-options. Target processors are identified in **Table 2-1** and `-tp` sub-options are identified in **Table 2-2**.

Table 2-1 Target Processor

Target	Target Processor
`ARM`	Generic ARM
`ARMV3`	ARM Version 3
`ARM710A`	ARM 710A Version 3
`ARMV4`	ARM Version 4
`ARM7TDMI`	ARM 7TDMI
`ARMV4BE`	ARM Version 4 big-endian
`ARMV5`	ARM Version 5 (big-endian)
`XScale`	Intel XScale Architecture (big-endian)

Table 2-2 Mode -tp Sub-Options

Suboptions	Description
`sd`	Use 12-bit data references
`ld`	Use 20-bit data references (default)
`fp`	Use static link library for floating-point support
`vld`	Use 28-bit data references
`scd`	Use 12-bit `const` data references
`lcd`	Use 20-bit `const` data references (default)
`vlcd`	Use 28-bit `const` data references

Predefined Macro Names for the Preprocessor

The macro names in **Table 2-3** are predefined in the preprocessor for target systems.

Table 2-3 Macros

Macro	Description
`_MPFARM`	Generic ARM processor
`_MPFARMBE`	Generic ARM processor (big-endian)
`_MPFARMV3`	ARM Version 3 processor
`_MPFARM710A`	ARM 710A processor

Table 2-3 Macros (continued)

Macro	Description
_MPFARMV4	ARM Version 4 processor
_MPFARM7TDMI	ARM 7TDMI processor
_MPFARMV4BE	ARM Version 4 processor, big-endian
_MPFARMV5	ARM Version 5 processor, big-endian
_MPFXSCALE	Intel XScale processor, big-endian
_FPFARM	ARM floating point processor
_FPFARMBE	ARM floating point processor, big-endian

Target names specify the compiler to use when writing machine- and operating system-independent programs.

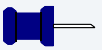
The executive defines the _MPF macro for the target processor as well as any processors that are generally considered subsets of the target processor. **Table 2-4** provides a few examples of this behavior.

For more information on exactly which macros are defined for the ARM processors, run the executive in verbose and dry run modes stopping after the front end. For example, to check the defines for the ARM7TDMI target (source file not required):

```
cc -b -h -efe -tp=ARM7TDMI t.c
```

This causes the executive to print a line similar to:

```
“cpfe -m --target=10 -I/dd/MWOS/SRC/DEFS -I/dd/MWOS/OS9000/SRC/DEFS
-I/dd/MWOS/OS9000/ARMV4/DEFS -D_UCC -D_SPACE_FACTOR=1 -D_TIME_FACTOR=1
-D_OS9000 -D_MPFARM7TDMI -D_MPFARMV4 -D_MPFARM -D_FPFARM -D_LIL_END
-w --Extended_ANSI --gen_c_file_name=t.i t.c"
```



Note

Note that _MPFARM7TDMI, MPFARMV4, and _MPFARM macros are defined.

The _MPFARM macro indicates that a source file is being compiled for an ARM little-endian family target. The _MPFARMBE macro indicates that a source file is being compiled for an ARM big-endian family target.

Table 2-4 identifies the relationship between the target processor and the preprocessor macros. Note that specific targets also define the subset macros (example: when targeting the ARM7TDMI, _MPFARM7TDMI, ARMV4, and _MPFARM are defined).

Table 2-4 _MPFxxx Macro Behavior

Target	Microprocessor Family Macros Defined
Generic ARM	_MPFARM _MPFARMV3
ARM Version 3	_MPFARM _MPFARMV3
710A	_MPFARM _MPFARMV3 _MPFARM710A
ARM Version 4	_MPFARM _MPFARMV4
7TDMI	_MPFARM _MPFARMV4 _MPFARM7TDMI
ARM Version 4, big-endian	_MPFARMBE _MPFARMV4BE

Table 2-4 _MPFxxx Macro Behavior (continued)

Target	Microprocessor Family Macros Defined
ARM Version 5, big-endian	_MPFARMBE _MPFARMV5
Intel XScale processor, big-endian	_MPFARMBE _MPFARMV5 _MPFXSCALE

ARM-Unique Phase Option Functionality

Phases having unique phase option functionality on the ARM processor are:

- **Back End Options**
- **Assembly Optimizer Options**
- **Linker Options**

Back End Options

`-m=<non remote memory left>`
: Informs the back end that other files in the program have used some amount of the 4K data area.

The back end orders the data area based on static analysis of the data area objects and sorts the data based on usage and size. This means that the most heavily used objects end up in the non-remote area. To do this, the back end needs information about how the object linker lays out the data area for the entire program.

Code generation options provide specifications for code generated by the back end.

Table 2-5 Code Generation Options

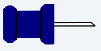
Option	Description
`-px`	Make references to external data extra long (28/32 bits)
`-pxc`	Make some references to code symbols extra long (28/32 bits)
`-pl`	Make references to external data long (20/24 bits)
`-plc`	Make some references to code symbols long (20/24 bits)
`-ps`	Disable stack checking code
`-pg`	Enable the back end to generate code to derive `r12` rather than relying on a globally set `r12` for each function that needs it. This option might be used for non-program modules that have multiple entry points.

Target architecture code generation options provide specifications unique to a target architecture for code generated by the back end.

-p<target architecture>	Identifies the target architecture for which to generate code. Availability of halfword and signed byte load and store instructions differ based upon target architecture.

Table 2-6 <target architecture> Code Generation Options

Option	Description
`-parmv3`	Generate code which does not use halfword or signed byte load and store instructions. (Not available in little-endian code generator)
`-parmv4`	Generate code which uses halfword and signed byte load and store instructions.
`-parmv5`	Generate code identical to -parmv4. (Not available in little-endian code generator)



Note

ARM Version 3 code executes correctly on ARM Version 4 architecture, however, there is a degradation in performance on signed byte and halfword data.

Because the ARM Version 3 lacks instructions to load and store halfwords as a unit, it is impossible to generate strictly correct code for manipulating objects of type volatile short, volatile signed short, and volatile unsigned short. This affects those writing code such as device drivers, since a two-byte memory-mapped I/O port cannot be properly accessed on an ARM Version 3.

Assembly Optimizer Options

`-t[=]<num>` Specify target processor family

Table 2-7 Assembly Optimizer Processor Numbers

<num>	Assembly Optimizer Target
1	ARM (default)

`-p=<X>` Selectively skip processor-specific optimizations

Table 2-8 Assembly Optimizer Processor-Specific Optimizations

<X>	Processor-Specific Optimization
l	Location tracking
c	Conditionalized execution
p	Pooling of PC-relative data
r	Copy/shift propagation

`-s<method>` Set the peephole scheduling method

Table 2-9 Peephole Scheduling Methods

<method>	Description
s	Spread dependent instructions
c	Compress floating point instructions

Table 2-9 Peephole Scheduling Methods

<method>	Description
n	No reordering of instructions
b	Both spread and compress

Linker Options

`-t=<target>` Linker, specify target module type

Table 2-10 Target Module Type

Target	Module Type
`os9k_arm`	OS-9 for ARM
`os9k_armbe`	OS-9 for ARM, big-endian

C/C++ Application Binary Interface Information

Register usage, passing arguments to functions, and language features are described in this section.

Register Usage

General purpose, floating point, and other registers are identified in this section.

Table 2-11 Register Classes

Register Class	Names Used
General Purpose Registers	`r0 - r15, gp, cp, sp, lr, pc`
Floating Point Registers	`f0 - f7`
Coprocessor Registers	`c0 - c15`
Program Status Registers	`cpsr, spsr`

General Purpose Registers.

Table 2-12 General Purpose Register Use for 32-bit Arguments

Register	Alternate Name	Description
r0-r5		Callee-saved register (for locals and temporaries).
r6	gp	Static storage pointer.
r7		1st integral argument passed; integral return value. For functions returning aggregates (e.g., structures) this points to the returned aggregate.*
r8		2nd integral argument passed. *
r9		3rd integral argument passed. *
r10		4th integral argument passed. *
r11		Caller-saved register (for locals and temporaries).
r12	cp	Constant storage pointer. †
r13	sp	Stack pointer.
r14	lr	Subroutine link register.
r15	pc	Program counter.

* If register is not in use as described in the table, it can be used for integral user register variables and compiler temporaries.

† Used to access `const` qualified data in the code area of the module. This is accomplished by using the register as a pointer to the code area data. The register is automatically initialized by the kernel for program modules. Non-program modules must either set the register up themselves or use the back end option `-pg` to generate the code to set up the register for each function that needs it.

The values in r7 through r11 need not be preserved across a function call. That is, a function is safe to use these registers without saving and restoring their values.

The compiler uses the remainder of the integral registers (r0 through r5) for integral user register variables and compiler temporaries.

ARMV4 only: Table 2-13 shows how registers r7-r10 are used for 64-bit long long integer arguments. Any consecutive registers from the set of R7-R10 can be used to hold a long long integer argument. The lower numbered register holds the least significant 32 bits of the argument and the higher numbered register holds the most significant 32 bits of the argument.

An interesting situation can occur when a function has both long long and other integral arguments. If three of the registers hold long arguments and the next argument is a long long, then the long long will be saved on the stack. Long long arguments will continue to be stored on the stack until a 32-bit integer argument is encountered and stored in R10. Therefore the argument in register R10 can not be assumed to be the argument that logically follows the argument in R9.

Table 2-13 General Purpose Register Use for 64-bit Arguments

Register	Alternate Name	Description
r7		32-bit segment of integral return value. For functions returning aggregates (e.g., structures) this points to the returned aggregate.*
r8		32-bit segment of integral argument passed. *
r9		32-bit segment of integral argument passed. *
r10		32-bit segment of integral argument passed. *

* If register is not in use as described in the table, it can be used for integral user register variables and compiler temporaries.

Floating Point Registers

Table 2-14 Floating Point Registers

Register	Description
`f0`	1st floating point argument passed, floating point return value
`f1`	2nd floating point argument passed
`f2`	3rd floating point argument passed
`f3`	4th floating point argument passed

When not in use, any of registers `f0` through `f3` can be used as temporary register variables.

Functions can use the values in registers `f0` through `f3` (caller saved registers) without saving/restoring them for the functions' callers. A calling function must save them if they expect to keep their values across the call to a function.

The compiler uses the remainder of the floating point registers (`f4` through `f7`) for floating point user register variables and compiler temporaries.

Passing Arguments to Functions

When an argument is passed to a called function, the argument is in one of two places, in a register or in the Output Parameter Area (OPA) of the calling function.

The called function determines the location of the argument by argument type and the order specified in the argument list.

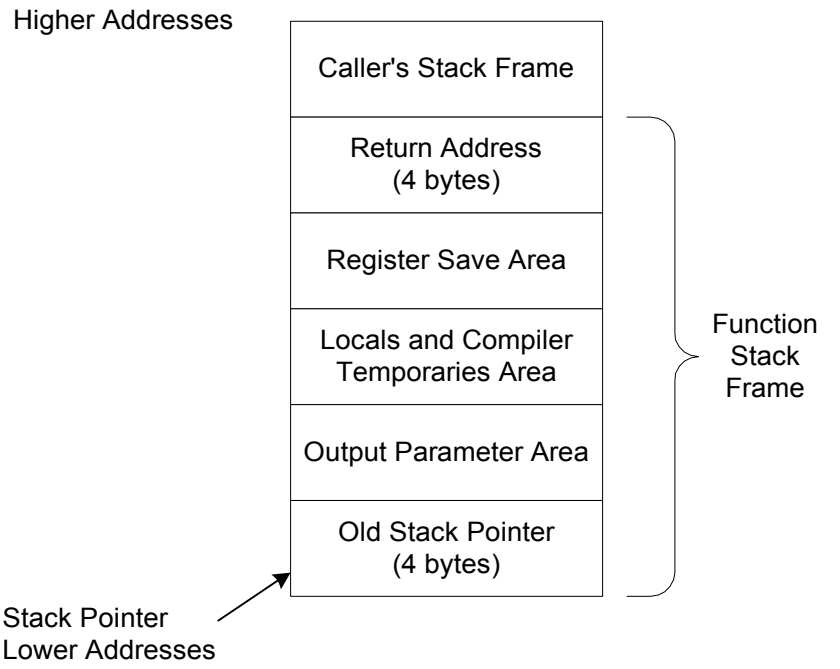
For this discussion:

- An **integral argument** is an argument of type `int`, a pointer, or a `char` or `short` converted to an `int`.
- A **floating point argument** is an argument of type `double` or a `float` converted to a `double`.

There are four integral registers used for parameter passing: `r7` through `r10` inclusive. Four floating point registers are available for floating point parameter passing: `f0` through `f3` inclusive.

The OPA is also used to pass arguments (when the registers have been exhausted). **Figure 2-1** illustrates a stack frame for a function.

Figure 2-1 Stack Frame for a Function



The basic algorithm the compiler uses to pass arguments is as follows:

```
if function returns a struct
    put address of struct return area into first integral passing register
while still more arguments
    if parameter is part of variable arguments
        put argument into next position in OPA
    else if argument is a struct
        copy struct into next position in OPA
    else
        if argument is integral
            if argument is 64-bit integer type
                if pair of integral passing registers are available
                    put argument into register pair
                else
                    put argument into next two 32-bit words of OPA
            else
                if an integral passing register is available
                    put argument into integral register
                else
                    put argument into next position in OPA
        else if argument is floating-point
            if a floating-point passing register is available
                put argument into floating-point register
            else
                put argument into next position in OPA
    advance to next argument
```

The OPA is filled from lowest address to highest address.

`Struct` arguments and parameters that comprise the variable argument to a variable argument function are always passed on the OPA. If a function is to return a value, an integral return value is returned in `r7` or a floating point return value is returned in `f0`. If a function is to return a `struct`, the address of a return area is passed as the first integral argument, in `r7`.

C Language Features

For More Information

Other implementation-defined areas are identified in the Language Features chapter of the ***Using Ultra C/C++*** manual and the Overview chapter of the ***Ultra C Library Reference*** manual. Refer to an ANSI/ISO specification for more information.

In conformance with the ANSI/ISO C specification, the implementation-defined areas of the compiler are listed in this section. Each bulleted item contains one implementation- defined issue. The number in parentheses included with each bulleted item indicates the location in the ANSI/ISO specification where further information is provided.

Characters

- The number of bits in a character in the execution character set (5.2.4.2.1).

  There are eight bits in a character in the execution character set.

Integers

- The representations and sets of values of the various integer types (6.1.2.5).

Table 2-15 Integer Type/Range

Type	Representation	Minimum / Maximum
`char`, `signed char`	8 bit 2's complement	-128 / 127
`unsigned char`	8 bit binary	0 / 255
`short int *`	16 bit 2's complement	-32768 / 32767
`unsigned short int *`	16 bit binary	0 / 65535
`int`	32 bit 2's complement	-2147483648 / 2147483647
`unsigned int`	32 bit binary	0 / 4294967295
`long int`	32 bit 2's complement	-2147483648 / 2147483647
`unsigned long int`	32 bit binary	0 / 4294967295

Table 2-15 Integer Type/Range

Type	Representation	Minimum / Maximum
`long long`	64-bit 2's complement	-2^{63} / $2^{63}-1$
`unsigned long long`	64-bit binary	0 / $2^{64}-1$

* On ARM V3 architecture, it is not possible to access 16-bit values with a single instruction. Use of *volatile* on this integer type generates a warning.

- The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2).

  When converting a longer integer to a shorter signed integer, the least significant <n> bits of the longer integer are moved to the integer of <n> bits. The resulting value in the smaller integer is dictated by the representation. For example, if the conversion is from `int` to `short`, then the least significant 16 bits are moved from the `int` to the `short`. This value is then considered a 2's complement 16-bit integer.

  When conversion from unsigned to signed occurs with equally sized integers, the most significant bit becomes the sign bit. Therefore, if the unsigned integer is less than 0x80000000, the conversion has no affect. Otherwise, a negative number results.

- The sign of the remainder on integer division (6.3.5).

  The result of an inexact division of one negative and one positive integer is the smallest integer greater than or equal to the algebraic quotient.

  The sign of the remainder on integer division is the same as that of the dividend.

Floating Point

- The representations and sets of values of the various types of floating-point numbers (6.1.2.5).

Table 2-16 Floating Point Number Characteristics

Type	Format	Minimum / Maximum
`float`	32-bit IEEE 754	`1.17549435e-38f / 3.40282347e38f`
`double`	64-bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`
`long double`	64-bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`

Refer to the `float.h` header file for other limits and values.

Arrays and Pointers

- The type of integer required to hold the maximum size of an array. That is, the type of the size of operator, `size_t` (6.3.3.4, 7.1.1).

  An `unsigned long int` is required to hold the maximum size of an array. `unsigned long int` is defined as to `size_t` in `ansi_c.h`.

- The result of casting a pointer to an integer or vice versa (6.3.4).

  Since pointers are treated much like unsigned long integers, the integer is promoted using the usual promotion rules to an unsigned long. That is, the sign bit propagates out to the full 32-bit width.

- The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (6.3.6, 7.1.1).

A signed long int is required to hold the difference between two pointers to elements of the same array. long int is defined as `ptrdiff_t` in `ansi_c.h`.

Registers

- The extent to which objects can actually be placed in registers by use of the register storage-class specifier (6.5.1).

  The compiler automatically makes decisions about what objects are placed in registers, thus giving no special storage considerations for the register storage-class.

Structures, Unions, Enumerations, and Bit-Fields

- The padding and alignment of members of structures (6.5.2.1). This should present no problem unless binary data written by one implementation are read by another.

  Table 2-17 shows the alignment of the various objects within a structure. Required padding is supplied if the next available space is not at the correct alignment for the object. For example, a structure declared as:

  ```
  struct {
     char mem1;
     long mem2;
  };
  ```

  would be an eight-byte structure (64-bit), one byte for `mem1`, four bytes for `mem2`, and three bytes of padding to complete the structure.

Table 2-17 Alignment Table

Type	Alignment Requirement
`char`	1
`short`	2
`int`	4
`long`	4
`long long`	4
`pointers`	4
`float`	4
`double`	4
`long double`	4

- Whether “plain” `int` bit-field is treated as a signed `int` or as an unsigned `int` bit-field (6.5.2.1).

  A “plain” `int` bit-field is treated as a signed `int` bit-field.

- The order of allocation of bit fields within a unit (6.5.2.1).

  Bit fields are allocated from most significant bit to least significant bit.

- Whether a bit-field can straddle a storage-unit boundary (6.5.2.1).

  Bit fields are allocated end-to-end until a non-bit field member is allocated or until that positioning would cross an addressable boundary such that no object of an integral type could both contain the bit field and be correctly aligned.

- The integer type chosen to represent the values of an enumeration type (6.5.2.2).

  Enum values are represented in 32-bit two's complement integers.

Processing Directives

- Whether the value of a single-character, character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (6.8.1).

  The value of a single-character character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. This character constant may have a negative value.

- The method for locating includable source files (6.8.2).

  This method is described in the **Using the Executive** chapter of the ***Using Ultra C/C++*** manual.

- The support of quoted names for includable source files (6.8.2).

  Quoted names are supported for `#include` preprocessing directives.

- The mapping of source file character sequences (6.8.2).

  The mapping of source file character sequences is one-to-one. The case-sensitivity of file names depends on the file system being used.

ARM Processor-Specific Optimizations

The Ultra C/C++ ARM assembly optimizer (`optarm`), in addition to providing the standard generic assembly optimizations, provides processor-specific optimizations. These are:

- **Special Common Sub-Expressions**
- **Code Area Data Pooling and Consolidation**
- **Conditionalizing of Instructions**
- **Copy Propagation**

Special Common Sub-Expressions

On the ARM architecture, certain constants are more expensive to work with than others. To account for this, the common sub-expression elimination optimization is performed on expressions involving potentially expensive constants. These include the following:

- The computation of global variable addresses.
- The computation of integer constants that require several arithmetic instructions or one pc-relative load to compute. For example, the constant 0xaabbccdd. The constant 0xff00 would not be considered.
- The computation of all floating point constants, with the exception of 0.0, 1.0, 2.0, 3.0, 4,0, 5.0, 10.0, and 0.5.

Code Area Data Pooling and Consolidation

The ARM instruction set limits the size of immediate values to 8 bits rotated by 2n. Often times data can be built up using immediates exclusively. For example:

```
=x equ 0x333
   mov R7,0xff&=x
   add R7,0xff00&=x
```

At other times it is more appropriate to load the value from memory. This data can be stored in the code area and loaded into registers using PC-relative loads. The assembly optimizer attempts to pool this code area data together behind naturally occurring divisions (such as a return from subroutine) to limit the number of extra branches in code. For example, given the following C code:

```
    extern int x, y;
    func(&x, &y);
    func2(&x);
```

The backend might generate:

```
 ldr R7,=_$L1
 b =_$L2 **skip
=_$L1
```

```
 dc.l =x
=_$L2
 add R7,R6,R7
 ldr R8,=_$L3
 b =_$L4 **skip
=_$L3
 dc.l =y
=_$L4
 add R8,R6,R8
 bl =func
 ldr R7,=_$L5
 b =_$L6 **skip
=_$L5
 dc.l =x
=_$L6
 add R7,R6,R7
 bl =func2
 ...
 ldmfd R13!,{PC}
```

The assembly optimizer could change this to:

```
 ldr r7,=_$L1
=_$L2
 add r7,r6,r7
 ldr r8,=_$L3
=_$L4
 add r8,r6,r8
 bl =func
 ldr r7,=_$L5
=_$L6
 add r7,r6,r7
 bl =func2
 ...
 ldmfd r13!,{PC}
=_$L5
=_$L1 dc.l =x
=_$L3 dc.l =y
```

Conditionalizing of Instructions

In the ARM architecture, conditional execution is not limited to conditional branches. Every instruction can be conditionalized. The Ultra C/C++ ARM assembly optimizer attempts to take advantage of this capability to reduce code size and improve performance.

For example, given the following C code:

```
  b = x && y && c;
```

The backend might generate:

```
 cmp R7,0
 beq =_$l4e1
 cmp R8,0
 beq =_$l4e1
 cmp R9,0
 beq =_$l4e1
 mov R7,1
 b =_$l4e5
=_$l4e3
 mov R7,0
=_$l4e5
```

The assembly optimizer could then change this to:

```
 cmp r7,0
 cmpne r8,0
 cmpne r9,0
 movne r7,1
 bne =_$l4e5
=_$l4e3
 mov R7,0
=_$l4e5
```

This optimization pays close attention to the desired time/space ratio such that the number of instructions that are conditionalized depends on how important space is compared to time.

Copy Propagation

The assembly code optimizer tries to eliminate needless copies between register temporaries; resulting in smaller, more efficient code. For example:

```
ldr r7,[gp,x]
mov r8,lsl(r7,2)
add r8,r7,r8
```

This may be changed to the following:

```
ldr r7,[r6,x]
add r8,r7,lsl(r7,2)
```

As another example:

```
mov r8,lsl(r7,2)
and r8,r8,r7
```

This may be changed to the following:

```
and r8,r7,lsl(r7,2)
```

_asm() Register Pseudo Functions

`_asm()` uses registers pseudo functions as identified in **Table 2-18**.

Table 2-18 _asm() Register Pseudo Functions

Register	Description
`__reg_gen`	Any non-dedicated integer register
`__reg_float`	Any floating point register
`__reg_r<n>`	The integer register specified by `n (0 <= n < 16)`
`__reg_f<n>`	The floating point register specified by `n (0 <= n < 8)`

Assembler/ Linker

The assembler allows use of standard ARM assembly language mnemonics and syntax, modified as described in this section. For more specific information about individual instructions, consult the following documentation:

- ARM Architecture Reference
- ARM FPA10 Data Sheet

ROF Edition Number

The ARM assembler emits ROF Edition #15.

External References

The ARM assembler allows the use of external references with any operators within any expression fields not defined to be constant expression fields.

Symbol Biasing

There is no biasing of either code (`r12`) or static (`r6`) pointers for the ARM processor.

Assembler Syntax Extensions and Limitations

The Ultra C/C++ Compiler's adaptation of the ARM instruction syntax has a few notable differences from what is defined by the ARM Architecture Reference manuals.

- The assembler uses the space character as the comment delimiter. As a result, the operand stream must not include any spaces.
- Because of the above restriction, the **shifted register operand value** syntax was altered.

An example of ARM syntax and its equivalent Ultra C/C++ syntax are shown in **Table 2-19**.

Table 2-19 Equivalent ARM Instruction Syntax

Instruction	ARM Syntax	Ultra C/C++ Syntax
`mov`	`r2, r0, LSL #2`	`r2,lsl(r0,2)`
`add`	`r9, r5, r5, ASR #3`	`r9,r5,asr(r5,3)`
`rsb`	`r9, r5, r5, LSL #3`	`r9,r5,lsl(r5,3)`
`sub`	`r10, r9, r8, LSR #4`	`r10,r9,lsr(r8,4)`
`mov`	`r12, r4, ROR r3`	`r12,ror(r4,r3)`
`mvn`	`r1, r4, RRX`	`r1,rrx(r4)`

These equivalencies are due to restrictions in the Ultra C/C++ assembler and for ease in reading and understanding the syntax. To see other uses of Ultra C/C++ assembly, it is possible to stop the compilation process after the back end has finished (using the `-ebe` option) and view the resulting assembly file.

- The Ultra C/C++ ARM assembler does not require the # designator in front of immediate values. The value is known to be an immediate based on its context. In cases where the context is ambiguous (example: a value has been equated to the name `r3` (not recommended)), the assembler assumes the register is intended for use.
- The Ultra C/C++ assembler provides an `nop` instruction to do nothing. This is equivalent to:

```
andeq        r0,r0,r0
```

- The # character is used in place of the ^ character for load and store multiple instructions. For example, the first line is original ARM assembly syntax, the second line is the equivalent Ultra C/C++ ARM assembly syntax:

```
stmfd        sp!,{r7-r14}^
stmfd        sp!,{r7-r14}#
```

- The syntax accepted by the Ultra C/C++ ARM assembler for the move from general purpose register to **SR** instruction may be slightly different than that specified in ARM architecture manuals. The following syntax is accepted:

```
msr{<cond>}  <psr>_flg,<immediate>
msr{<cond>}  <psr>_flg,<rn>
msr{<cond>}  <psr>_all,<rn>
msr{<cond>}  <psr>,<rn>
msr{<cond>}  <psr>_f,<immediate>
msr{<cond>}  <psr>_[fsxc],<rn>
```

Table 2-20 Variables

Variable	Description
{ }	Optional Entry
[ ]	Enter one character only shown within the brackets

Table 2-20 Variables (continued)

Variable	Description
<cond>	ARM conditional execution code
<per>	One of cpsr or spsr
<immediate>	ARM 8-bit rotated immediate
<rn>	A general purpose register

- Coprocessor registers (as referred to by the coprocessor instructions cdp, ldc, stc, mrc, and mcr) are designated by their register number and prefixed by the letter c.
- The mnemonic extensions specifying conditional execution accepted by the assembler are identified in **Table 2-21**.

Table 2-21 Integer Type/Range

Mnemonic	Description
eq	Equal
ne	Not equal
cs/hs	Carry set / unsigned higher or same
cc/lo	Carry clear / unsigned lower
mi	Negative
pl	Positive or zero
vs	Overflow
vc	No overflow

Table 2-21 Integer Type/Range (continued)

Mnemonic	Description
hi	Unsigned higher
ls	Unsigned lower or same
ge	Signed greater than or equal
lt	Signed less than
gt	Signed greater than
le	Signed less than or equal
al	Always (equivalent to no extension)
nv	Never*

* Use of the nv conditional extension is not recommended as its effect is unpredictable. Acceptance of the nv conditional extension by the assembler is not assurance that it will work as expected.

Working with Immediate Data

The ARM instruction set limits the size of immediate data to 8 bits for data processing instructions. The ARM assembler does not implement constant explosion, leaving this responsibility to the assembly language programmer. Bitwise operations are useful for this purpose; specifically the `bitwise` and (`&`). The following code illustrates the addition of a 16-bit unsigned immediate to a register.

```
uimm16 set 0xf0f0
       add r7,r7,0xff&uimm16
       add r7,r7,0xff00&uimm16
```

Similarly, the ARM instruction set limits the size of immediate data to 12 bits for general load-store instructions. Again, the assembly language programmer must allow for this. The following code illustrates non-remote data access from the static data area.

```
      vsect
stuff ds.l 1
      ends
      add  r7,gp,0xfffff000&stuff
      ldr  r7,[r7,0xfff&stuff]
```

Note that the bitwise and operation in the addition statement masks only the lower twelve bits, leaving the upper twenty bits. This allows the object code linker to detect an error that would otherwise be difficult to track (such as a value being out of range). If, for instance, the linker symbol `stuff` evaluated to 0x00115004, the linker would display the following message:

```
The value $115000 cannot be encoded into a field
8-bits rotated by 2n bits.
```

Therefore, an additional add instruction would be necessary.

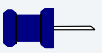
Stack Checking

This section provides ARM-specific information about stack checking. Refer to ***Using Ultra C/C++*** for more general information on stack checking.

If stack checking is inappropriate for the module being created, you will need to define the following items:

- a global pointer called `_stbot` (initialized to `ULONG_MAX` if possible)
- a function called `_stkhandler` (it returns to its caller)

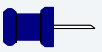
A function called `_stkoverflow` is called when the stack appears to overflow. If a non-static function called `_stkoverflow` resides in an ROF that is linked to make the object module, that function is called instead of the default function. The default function writes a message to the standard error path and causes the program to exit with a status of one. `_stkoverflow` neither accepts parameters nor returns a value.



Note

If `_stkoverflow` is inappropriate for your application, consider writing a function to handle stack overflow.

`_stkhandler`, the function that checks for stack overflow, can be revised. Revision may be necessary if stack checking is inapplicable to the module that calls the library functions. `_stkhandler()` is passed the desired stack pointer in `r3` and does not return a value.



Note

Stack handler code must be compiled with stack checking turned off (`-r` in `ucc` mode).

Chapter 3: SH-5

This chapter contains information specific to the SH-5 family of processors. The following sections are included:

- **Executive and Phase Information**
- **C/C++ Application Binary Interface Information**
- **_asm() Register Pseudo Functions**
- **SH-5 Processor-Specific Optimizations**
- **Assembler/ Linker**
- **Stack Checking**

Executive and Phase Information

Executive `-tp` enables specific options that are dependent upon executive mode. Processors and sub-options for ucc and c89 option modes are identified in this section.

Executive -tp Option

`-tp[=]<target>{[,]<suboptions>}`

Use this option when you want to specify the target processor `<target>` and target processor sub-options. Target processors are identified in **Table 3-1** and `-tp` sub-options are identified in **Table 3-2**.

Table 3-1 Target Processor

Target	Target Processor
SH5m	Generic SH-5 in media mode
SH8000	SH8000 family

Table 3-2 Mode -tp Sub-Options

Suboptions	Description
`sd`	Use 16-bit data references (default).
`ld`	Use 32-bit data references.

Table 3-2 Mode -tp Sub-Options (continued)

Suboptions	Description
`scd`	Use 16-bit code area data references (default).
`lcd`	Use 32-bit code area data references.
`sc`	Use 16-bit code function references (default).
`lc`	Use 32-bit code function references.
`sb`	Use 18-bit function-internal branches (default).
`lb`	Use 32-bit function-internal branches.

Predefined Macro Names for the Preprocessor

The macro names in **Table 3-3** are predefined in the preprocessor for target systems.

Table 3-3 Macros

Macro	Description
`_MPFSH5`	Generic SH-5 processor (media or compact)
`_MPFSH5M`	Generic SH-5media processor
`_MPFSH8000`	SH8000 series processor
`_FPFSH5M`	SH-5media floating-point processor

Target macros are used to place conditions on code so that machine- and operating system-independent programs can be created. Each target macro name specifies a particular compiler.

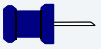
The executive defines the _MPF macro for the target processor as well as any processors that are generally considered subsets of the target processor. **Table 3-4** contains the complete list of macro combinations.

For more information on exactly which macros are defined for the SH-5 processors, run the executive in verbose and dry run modes stopping after the front end. For example, to check the defines for the SH-5media target (source file not required), enter the following on the command line:

```
xcc -b -h -efe -tp=sh5m t.c
```

This causes the executive to print something similar to the code below:

```
Include file paths:
    \mwos\SRC\DEFS
    \mwos\OS9000\SRC\DEFS
    \mwos\OS9000\SH5M\DEFS
"cpfe -m --target=8 -I\mwos\SRC\DEFS -I\mwos\OS9000\SRC\DEFS
-I\mwos\OS9000\SH5M\DEFS -D_UCC -D_MAJOR_REV=2 -D_MINOR_REV=4 -D_SPACE_FACTOR=1
-D_TIME_FACTOR=1 -D_OS9000 -D_MPFSH5M -D_MPFSH5 -D_FPFSH5M -D_BIG_END -w
--Extended_ANSI --gen_c_file_name=t.i t.c"
```



Note

Both _MPFSH5 and _MPFSH5M are defined.

The _MPFSH5 macro indicates that a source file is being compiled for a SH-5 family target.

Table 3-4 identifies the relationship between the target processor and the preprocessor macros. Note that specific targets also define the subset macros. (For example, when targeting the SH-8000, the `_MPFSH5` and `_MPFSH5M` macros are defined.)

Table 3-4 _MPFxxx Macro Behavior

Target	Microprocessor Family Macros Defined
SH5M	`_MPFSH5 _MPFSH5M`
SH8000	`_MPFSH5 _MPFSH5M _MPFSH8000`

SH-5m-Unique Phase Option Functionality

The following phases have unique phase option functionality on the SH-5m processor:

- Back end
- Assembly optimizer
- Linker

Back End Options

`-m=<non remote memory left>`

This option informs the back end that other files in the program have used some amount of the 64K data area.

The back end gives orders to the data area based on static analysis of the data area objects, and sorts the data based on usage and size. This means that the most heavily used objects end up in the non-remote area. To accomplish this, the back end needs information concerning how the object linker will lay out the data area for the entire program.

Code generation options provide specifications for code generated by the back end.

Table 3-5 Code Generation Options

Option	Description
`-pg`	Generate code to derive `cp` (r13) rather than relying on a globally set `cp` for each function that needs it. This option might be used for non-program modules that have multiple entry points.
`-pl`	Cause references to external data objects to be long.
`-pla`	Cause all non-function branches to be long.
`-plc`	Cause references to constant data objects to be long.
`-plf`	Cause references to functions to be long.
`-ps`	No stack checking code.

Target architecture code generation options provide specifications that are unique to target architecture (for code generated by the back end).

`-p<target architecture>`

Use this option to identify the target architecture for which to generate code. Implementation of multiply and divide instructions differs based on target architecture.

Table 3-6 <target architecture> Code Generation Options

Option	Description
`-pSH5m`	Generate code for the generic SH-5 processor in media mode.

Linker Options

`-t=<target>` This is a linker option to specify target module type.

Table 3-7 Target Module Type

Target	Module Type
`os9k_sh5m`	OS-9 for SH-5media

C/C++ Application Binary Interface Information

The following information is described in this section:

- **Register Usage**
- Arguments Passed to Functions
- Callee Saved Registers
- Language Features

Register Usage

General purpose, floating-point, and other registers are defined in this section. The register classes are listed and explained below.

- General Purpose Registers (GPRs)
- Floating-Point Registers (FPRs)

Table 3-8 General Purpose Registers

Register Names	Description
`r0, r1`	Caller save
`r2`	Integer or pointer return register; caller save
`r2 - r9`	Incoming integer and pointer arguments; caller save
`r10 / lr`	Linkage register; caller save
`r11 / at`	Reserved for long reference computation in assembler; caller save

Table 3-8 General Purpose Registers (continued)

Register Names	Description
`r12 / fp`	Local frame pointer; callee save
`r13 / cp`	Global constant data pointer; callee save
`r14 / gp`	Global data pointer; callee save
`r15 / sp`	Stack pointer; callee save
`r16 - r19`	Caller save
`r20 - r23`	Callee save
`r24 - r31`	Caller save
`r32 - r62`	Callee save
`r63 / zero`	Hard-coded 0 value register

* If the register is not used as stated above, it may be used for integral user register variables and compiler temporaries.

† `cp (r13)` is used to access `const`-qualified data in the code area of the module. This is accomplished by using the register `cp` as a biased (by `32K-16` [0x7ff0] bytes) pointer to the code area data. `cp` is automatically initialized by the kernel for program modules. Non-program modules must either set up `cp` themselves or use the back end option `-pg` to generate the code necessary to set up `cp` for each function that needs it.

The values in `r0` through `r9`, `r11`, `r16 - r19`, and `r24 - r31` do not need to be preserved across a function call; a function is safe to use these registers without saving and restoring the registers' values.

The compiler uses the remainder of the integral registers for integral user register variables and compiler temporaries.

floating-point Registers

Table 3-9 Floating-point Registers

Floating-point Register	Description
`fr0, fr1`	Floating-point return value; caller save
`fr0 - fr11`	Floating-point function arguments
`fr12 - fr15`	Callee save
`fr16 - fr35`	Caller save
`fr36 - fr63`	Callee save

When not in use for argument passing, any registers from `fr0` through `fr11` can be used as temporary register variables.

Functions can use values `fr0` through `fr11` and `fr16` through `fr35` without saving or restoring them for the functions' callers. However, a function must save them if they are expected to maintain their values across a call to a function.

Target Address Registers

Table 3-10 Target Address Register Usage

Registers	Description
`tr0 - tr4`	Caller save
`tr5 - tr7`	Callee save

Pointer and non-64-bit Integer Representation

Pointers and non-64-bit integers (including `char`) are held in registers in a 32-bit signed format. That is, the upper 32 bits of the 64-bit register contain a sign extension of the lower 32 bits regardless of the signedness of the 32-bit value. This results in a correct 64-bit representation of all types of integers except 32-bit unsigned values. Take this fact into consideration when writing assembly language that calls C functions. Do not assume that the result of a function returning a 32-bit unsigned value will always be positive when treated as a 64-bit value.

Be sure to use 32-bit instructions (commonly the .l extension) when writing assembly language that manipulates pointers or 32-bit or smaller integers. The use of 32-bit instructions is beneficial for pointers because it ensures that only valid addresses are generated by pointer arithmetic. The use of 32-bit instructions is also beneficial for integers because it keeps registers in the correct format for calling C functions.

Passing Arguments to Functions

When an argument is passed to a called function, the argument is in one of two places: in a register or in the Output Parameter Area (OPA) of the calling function.

The called function determines the location of the argument by argument type and the order specified in the argument list.

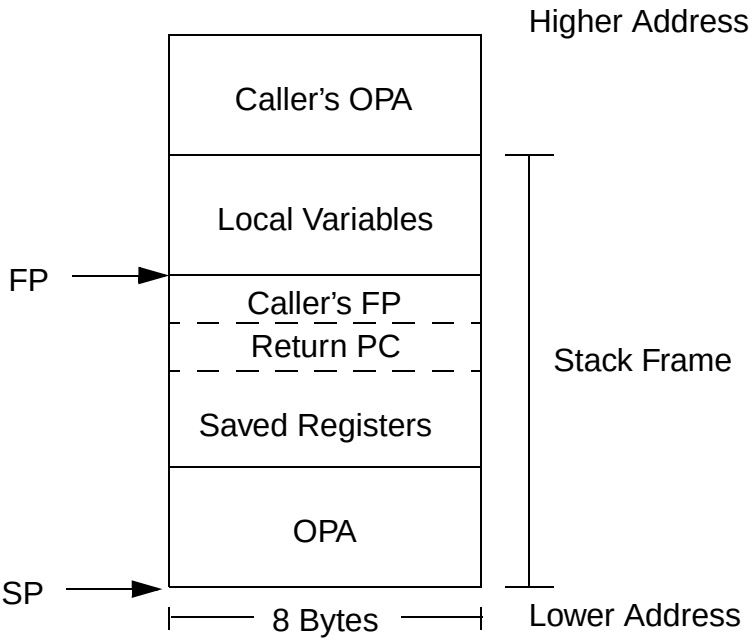
For the purposes of this discussion, the following statements are true:

- An **integral argument** is an argument of integer or pointer type.
- A **floating-point argument** is an argument of type `double` or `float`.

There are eight integral registers used for parameter passing: `r2` through `r9`. There are also twelve 32-bit registers available for floating-point argument passing: `fr0` through `fr11`. (This allows for up to six 64-bit floating-point arguments.)

The OPA is also used to pass arguments when the registers have been exhausted. **Figure 3-1** illustrates a stack frame for a function.

Figure 3-1 Stack Frame for a Function



It is possible for a function to have a zero-length stack frame. Furthermore, any or all of the areas of a stack frame can be omitted. For example, suppose local variables are omitted. Because no local variables have been allocated, the caller's `fp` register will not be saved and no new `fp` register value will be created.

Arguments are passed to functions as though the code generator performed the following steps at each call site:

1. If the called function returns a structure, the pointer to the location at which to return the structure is placed in the first parameter register (`r2`). `r2` is then no longer a candidate for further use.

2. For each argument, consider its type:

   Integral: Place the argument into the next available integral parameter register (`r2` through `r9`). If there are no available integral argument registers, the next position in the OPA is used.

   `double`: Place the argument in the next available pair of 32-bit floating-point parameter registers so that the first member of the pair is an even-numbered 32-bit floating-point register. This can leave a “hole” in the floating-point parameter registers (if the previous parameter was `float` and placed in an even-numbered register). If their are no more pairs of floating-point parameter registers, the next position of the OPA is used.

   `float`: Place the argument into the next available 32-bit floating-point parameter register. If a 32-bit floating-point parameter register had to be skipped because a `double` parameter needed an even-numbered floating-point register, the skipped one is used. If there are no more floating-point parameter registers, the next position of the OPA is used.

   Structure or Union: Copy the argument into the next position of the OPA.

3. For each variable argument, place the argument in the next available position in the OPA.

The OPA is filled from lowest address to highest address. Padding is added for arguments whose alignment requirements are not met by prior arguments. The following example demonstrate the above concepts:

```
void func(int p1, int p2)
                        r2 = p1
                        r3 = p2

void func(float p1, float p2)
                        fr0 = p1
                        fr1 = p2
```

```
void func(double p1, double p2)
                        dr0 (fr0, fr1) = p1
                        dr2 (fr2, fr3) = p2

void func(int p1, float p2, double p3, float p4)
                        r2 = p1
                        fr0 = p2
                        dr2 (fr2, fr3) = p3
                        fr1 = p4

void func(int p1, struct { int x, y; } p2, int p3)
                        r2 = p1
                        OPA + 0 = p2.x
                        OPA + 4 = p2.y
                        r3 = p3

struct { int x, y; } func(int p1, int p2)
                        r2 = <return struct address>
                        r3 = p1
                        r4 = p2

void func(char *p1, ...);

func(p1, int p2, double p3, float p4, long long p5)
                        r2 = p1
                        OPA + 0 = p2
                        OPA + 4 = <4 bytes padding>
                        OPA + 8 = p3
                        OPA + 16 = p4
                        OPA + 20 = <4 bytes padding>
                        OPA + 24 = p5
```

C Language Features

In conformance with the ANSI/ISO C specification, the implementation-defined areas of the compiler are listed in this section. Each bulleted item contains one implementation-defined issue. The number in parentheses included with each bulleted item indicates the location in the ANSI/ISO specification where more information is provided.

For More Information

Other implementation-defined areas are included in the ***Using Ultra C/C++*** manual and the ***Ultra C Library Reference*** manual. Refer to an ANSI/ISO specification for more information.

Characters

- The number of bits in a character in the execution character set (5.2.4.2.1)

  There are eight bits in a character in the execution character set.

Integers

- The representations and sets of values of the various integer types (6.1.2.5)

Table 3-11 Integer Type/Range

Type	Representation	Minimum / Maximum
`char`, `signed char`	8-bit 2's complement	`-128 / 127`
`unsigned char`	8-bit binary	`0 / 255`
`short int`	16-bit 2's complement	`-32768 / 32767`
`unsigned short int`	16-bit binary	`0 / 65535`
`int`	32-bit 2's complement	`-2147483648 / 2147483647`
`unsigned int`	32-bit binary	`0 / 4294967295`
`long int`	32-bit 2's complement	`-2147483648 / 2147483647`
`unsigned long int`	32-bit binary	`0 / 4294967295`

Table 3-11 Integer Type/Range (continued)

Type	Representation	Minimum / Maximum
`long long *`	64-bit 2's complement	-2^{63} / $2^{63}-1$
`unsigned long long *`	64-bit binary	0 / $2^{64}-1$

* `long long` is not a part of the current ANSI standard.

- The result of converting an integer to a shorter signed integer or the result of converting an unsigned integer to a signed integer of equal length (if the value cannot be represented) (6.2.1.2)

  When converting a longer integer to a shorter signed integer, the least significant <n> bits of the longer integer are moved to the integer of <n> bits. The resulting value in the smaller integer is dictated by the representation. For example, if the conversion is from `int` to `short`, the least significant 16 bits are moved from the `int` to the `short`. This value is then considered a 2's complement 16-bit integer.

  When conversion from unsigned to signed occurs with equally sized integers, the most significant bit becomes the sign bit. Therefore, if the unsigned integer is less than `0x80000000`, the conversion has no affect. Otherwise, a negative number results.

- The sign of the remainder on integer division (6.3.5)

  The result of an inexact division of one negative and one positive integer is the smallest integer greater than or equal to the algebraic quotient.

  The sign of the remainder on integer division is the same as that of the dividend.

floating-point

- The representations and sets of values of the various types of floating-point numbers (6.1.2.5)

Table 3-12 floating-point Number Characteristics

Type	Format	Minimum / Maximum
float	32 bit IEEE 754	`1.17549435e-38f / 3.40282347e38f`
double	64 bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`
long double	64 bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`

Refer to the `float.h` header file for other limits and values.

Arrays and Pointers

- The type of integer required to hold the maximum size of an array (the type of the size of operator, `size_t`) (6.3.3.4, 7.1.1)

  An `unsigned long int` is required to hold the maximum size of an array. `unsigned long int` is defined as `size_t` in `ansi_c.h`.

- The result of casting a pointer to an integer or vice versa (6.3.4)

  Since pointers are treated much like unsigned long integers, the integer will be promoted using the usual promotion rules to an unsigned long. That is, the sign bit propagates out to the full 32-bit width.

- The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (6.3.6, 7.1.1)

  A `signed long int` is required to hold the difference between two pointers to elements of the same array. `long int` is defined as `ptrdiff_t` in `ansi_c.h`.

Registers

- The extent to which objects can actually be placed in registers by use of the register storage-class specifier (6.5.1)

  The compiler automatically makes decisions about what objects are placed in registers, giving no special storage considerations for the register storage-class.

Structures, Unions, Enumerations, and Bit-Fields

- The padding and alignment of members of structures (6.5.2.1)

  This should present no problem unless binary data written by one implementation are read by another.

Table 3-13 shows the alignment of the various objects within a structure. Required padding is supplied if the next available space is not at the correct alignment for the object. For example, a structure declared as:

```
struct {
     char mem1;
     long mem2;
 };
```

would be an eight byte structure, including one byte for `mem1`, three bytes of padding to get `mem2` to four byte alignment, and four bytes for `mem2`.

Table 3-13 Alignment Table

Type	Alignment Requirement
`char`	1
`short`	2
`int`	4
`long`	4
`long long`	8
`pointers`	4
`float`	4
`double`	8
`long double`	8

- Whether “plain” `int` bit field is treated as a signed `int` or as an unsigned `int` bit field (6.5.2.1)

  A “plain” `int` bit field is treated as a signed `int` bit-field.
- The order of allocation of bit fields within a unit (6.5.2.1)

  Bit fields are allocated from most significant bit to least significant bit.

- Whether or not a bit field can straddle a storage-unit boundary (6.5.2.1)

  Bit fields are allocated end-to-end until a non-bit field member is allocated or until that positioning would cross an addressable boundary such that no object of an integral type could both contain the bit field and be correctly aligned.
- The integer type chosen to represent the values of an enumeration type (6.5.2.2)

  `Enum` values are represented in 32-bit two's complement integers.

Preprocessing Directives

- Whether or not the value of a single-character character constant, in a constant expression that controls inclusion, matches the value of the same character constant in the execution character set

  Whether or not such a character constant may have a negative value (6.8.1)

  The value of a single-character character constant, in a constant expression that controls inclusion, matches the value of the same character constant in the execution character set. This character constant may have a negative value.
- The method for locating includable source files (6.8.2)

  This method is described in the **Using the Executive** chapter of the ***Using Ultra C/C++*** manual.
- The support of quoted names for includable source files (6.8.2)

  Quoted names are supported for #include preprocessing directives.
- The mapping of source file character sequences (6.8.2)

  The mapping of source file character sequences is one-to-one. The case-sensitivity of file names depends on the file system being used.

_asm() Register Pseudo Functions

`_asm()` uses registers pseudo functions as identified in **Table 3-14**.

Table 3-14 _asm() Register Pseudo Functions

Register	Description
`__reg_gen`	Any non-dedicated integer register
`__reg_float`	Any 32-bit floating-point register
`__reg_double`	Any even-numbered 64-bit floating-point register pair
`__reg_r<n>`	The integer register specified by r<n>, <n> is 0 to 63
`__reg_fr<n>`	The 32-bit floating-point register specified by fr<n>, <n> is 0 to 63
`__reg_dr<n>`	The 64-bit floating-point register specified by dr<n>, <n> is an even number from 0 to 62
`__reg_zero`	Constant zero register, same as __reg_r63
`__reg_gp`	Global pointer register, same as __reg_r14
`__reg_sp`	Stack pointer register, same as __reg_r15
`__reg_fp`	Frame pointer register, same as __reg_r12
`__reg_lr`	Link register, same as __reg_r10. Request this register if you call functions via lr from within embedded assembly.
`__reg_cp`	Constant pointer register; same as `__reg_r13`
`__reg_at`	Assembler temporary register, same as __reg_r11

SH-5 Processor-Specific Optimizations

In addition to providing the standard generic assembly optimizations, the Ultra C/C++ SH-5media assembly optimizer (`optsh5m`), provides processor-specific optimizations. They are discussed in this section.

For More Information

The ***Using Ultra C/C++*** manual contains additional information on assembly level optimizations.

Special Common Sub-Expressions

On the SH-5m architecture, certain constants are more expensive to work with than others. To account for this, the common sub-expression elimination optimization is performed on expressions involving potentially expensive constants.

- The computation of all floating-point constants as none of the floating-point instructions allow for floating-point immediates
- The computation of integer constants that require two or more instructions

  An example of an integer constant included in this category is `0xffeeffee`. An example of a constant not included is `0xffffffee`.

Assembler/ Linker

The SH-5m assembler allows use of standard SH-5m assembly language mnemonics and syntax, modified as described in this section. For more specific information about individual instructions, consult the following books:

- ***SuperH TM (SH) 64-Bit RISC Series SH-5 CPU Core Volume 2: Shmedia***

ROF Edition Number

The SH-5m assembler emits ROF Edition #15.

External References

The SH-5m assembler allows the use of external references with any operators within expression fields not defined as constant expression fields.

Symbol Biasing

The linker biases both code and data symbols by `-32752 (-0x7ff0)`. Initialization routines for raw code should ensure that the static storage pointer (`r14`) and constant storage pointer (`r13`) are initialized with the proper base addresses (adjusted to account for this biasing).

The linker does not bias any symbols for system, file manager, device driver, device descriptor, or data modules.

Code Symbol Values

The SH-5media assembler (ash5m) creates ROF files such that all symbols that refer to the code area are off by one. This relieves most of the burden related to the SH-5's dual-mode architecture from the programmer. This requires that labels referring to code area data items be decremented by one. That is, the following code will compute the address of a constant string into `r0` (assuming `cp` is already set correctly):

```
movi string-1,r0
add.l cp,r0,r0
```

Map, debug (.dbg), and symbol table (.stb) files reflect the addition of one to code area symbols.

Assembler Syntax Extensions and Limitations

The Ultra C/C++ Compiler's adaptation of the SH-5media instruction syntax has a few notable differences from what is defined in SH-5media architecture manuals.

- All Microware assemblers use white space characters as comment delimiters. As a result, the arguments to mnemonics may not include any spaces.
- The SH-5media instruction set limits immediates to 16-bit signed values. To allow for manipulation of 32-bit immediate data, the following operators are available in the SH-5media assembler:

  ```
  hi(x)   = x >> 16               /* upper 16 bits */
  lo(x)   = x & 0xffff            /* lower 16 bits */
  ```

  The following code moves the 32-bit value represented by `Symbol` into GPR `r0`:

  ```
  movi    hi(Symbol),r0
  shori   lo(Symbol),r0
  ```

If Symbol is a 32-bit offset from GPR gp, then the following code moves the 32-bit address of Symbol int r0:

```
movi    hi(Symbol),r0
shori   lo(Symbol),r0
add.l   r0,gp,r0
```

Alternatively, the value of 32-bit integer data symbol Symbol can be loaded into r1 as follows:

```
movi    hi(Symbol),r0
shori   lo(Symbol),r0
ldx.l   r0,gp,r1
```

- The /l and /u modifiers in standard SH-5media assembly language are replaced with _l and _u in Microware's SH-5media assembler.
- The "$" character used to express the current program counter is replaced by the "*" character in Microware's SH-5media assembler. This assures some level of source-code compatibility with other Microware assemblers. $ is used to prefix hexadecimal constants. "0x" is also a valid hexadecimal constant prefix.
- In addition to the SH-5media instruction set, the Microware Ultra C/C++ SH-5media assembler also accepts the synthetic instructions specified in **Table 3-15**.

Table 3-15 SH-5m Assembler Synthetic Instructions

Synthetic Instruction	SH-5media Instruction Sequence
`move rS,rD`	`add rS,zero,rD`
`jump label`	`pta label,tr0` `blink tr0,zero`

`jump.l label`	`movi hi(label-pt),at`
	`shori lo(label-pt+1),at`
	`pt ptrel at,tr0`
	`blink tr0,zero`

Table 3-15 SH-5m Assembler Synthetic Instructions (continued)

Synthetic Instruction	SH-5media Instruction Sequence
`jump label,trD`	`pta label,trD` `blink trD,zero`

`jump.l label,trD`	`movi hi(label-pt),at`
	`shori lo(label-pt+1),at`
	`pt ptrel at,trD`
	`blink trD,zero`

`jlink label,rD`	`pta label,tr0` `blink tr0,rD`

`jlink.l label,rD`	`movi hi(label-pt),at`
	`shori lo(label-pt+1),at`
	`pt ptrel at,tr0`
	`blink tr0,rD`

`jlink label,rD,trD`	`pta label,trD` `blink trD,rD`

`jlink.l label,rD,trD`	`movi hi(label-pt),at`
	`shori lo(label-pt+1),at`
	`pt ptrel at,trD`
	`blink trD,rD`

`jabs[_l|_u] rS,rD`	`ptabs rS,tr0` `blink tr0,rD`
`jabs[_l|_u] rS,rD,trD`	`ptabs rS,trD` `blink trD,rD`

Table 3-15 SH-5m Assembler Synthetic Instructions (continued)

Synthetic Instruction	SH-5media Instruction Sequence
j<cc>[_l|_u] rM,rN,label where <cc> is: eq : == ne : != gt : > ge : >= gtu : > unsigned geu : >= unsigned	pta[_l|_u] label,tr0 b<cc>[_l|_u] rM,rN,tr0
j<cc>[_l|_u].l rM,rN,label	movi hi(label-pt),at shori lo(label-pt+1),at pt ptrel[_l|_u] at,tr0 b<cc>[_l|_u] rM,rN,tr0
j<cc>[_l|_u] rM,rN,label,trD	pta label,trD b<cc>[_l|_u] rM,rN,trD
j<cc>[_l|_u].l rM,rN,label,trD	movi hi(label-pt),at shori lo(label-pt+1),at pt ptrel[_l|_u] at,trD b<cc>[_l|_u] rM,rN,trD
j<cci>[_l|_u] rM,imm,label where <cci> is: eqi : == nei : !=	pta[_l|_u] label,tr0 b<cci>[_l|_u] rM,imm,tr0

Table 3-15 SH-5m Assembler Synthetic Instructions (continued)

Synthetic Instruction	SH-5media Instruction Sequence

`j<cci>[_l|_u].l rM,imm,label`	`movi hi(label-pt),at`
	`shori lo(label-pt+1),at`
	`pt ptrel[_l|_u] at,tr0`
	`b<cci>[_l|_u] rM,imm,tr0`

`j<cci>[_l|_u] rM,imm,label,trD`	`pta label,trD` `b<cci>[_l|_u] rM,imm,trD`

`j<cci>[_l|_u].l rM,imm,label,trD`	`movi hi(label-pt),at`
	`shori lo(label-pt+1),at`
	`pt ptrel[_l|_u] at,trD`
	`b<cci>[_l|_u] rM,imm,trD`

`tcall trap,func`	`movi trap,at`
	`trapa at`
	`dc.w trap`
	`dc.w func`

- When instructions call for a register specification, the assembler will accept two different names for the same register. These are called register aliases and are specified in **Table 3-16**.

Table 3-16 SH-5m Assembler Register Aliases

Alias Name	Real Name	Description
`lr`	`r10`	Link register
`at`	`r11`	Assembler/linker temporary
`fp`	`r12`	Frame pointer

Table 3-16 SH-5m Assembler Register Aliases (continued)

Alias Name	Real Name	Description
cp	r13	Constant data pointer
gp	r14	Global data pointer
sp	r15	Stack pointer
sr	cr0	Status register
ssr	cr1	Saved status register
pssr	cr2	Panic-saved status register
intevt	cr4	Interrupt event
expevt	cr5	Exception event
pexpevt	cr6	Panic-saved exception event
tra	cr7	Trap an exception
spc	cr8	Saved program counter
pspc	cr9	Panic-saved program counter
resvec	cr10	Reset vector
vbr	cr11	Vector base register
tea	cr13	Faulting effective address
dcr	cr16	Debug control
kcr0	cr17	Kernel register 0

Table 3-16 SH-5m Assembler Register Aliases (continued)

Alias Name	Real Name	Description
`kcr1`	`cr18`	Kernel register 1
`ctc`	`cr62`	Clock tick counter
`usr`	`cr63`	User-state status register

Stack Checking

This section provides SH5-specific information about stack checking. Refer to ***Using Ultra C/C++*** for more general information on stack checking.

If stack checking is inappropriate for the module being created, you will need to define the following items:

- a global pointer called `_stbot` (initialized to `ULONG_MAX` if possible)
- a function called `_stkhandler` (it returns to its caller)

A function called `_stkoverflow` is called when the stack appears to overflow. If a non-static function called `_stkoverflow` resides in an ROF that is linked to make the object module, that function is called instead of the default function. The default function writes a message to the standard error path and causes the program to exit with a status of one. `_stkoverflow` neither accepts parameters nor returns a value.

Note

If `_stkoverflow` is inappropriate for your application, consider writing a function to handle stack overflow.

`_stkhandler`, the function that checks for stack overflow, can be revised. Revision may be necessary if stack checking is inapplicable to the module that calls the library functions. `_stkhandler()` is passed the desired stack pointer in `r3` and does not return a value.

Note

Stack handler code must be compiled with stack checking turned off (`-r` in `ucc` mode).

Chapter 4: MIPS

This chapter contains information specific to the MIPS family of processors. The following sections are included:

- **Executive and Phase Information**
- **C/C++ Application Binary Interface Information**
- **_asm() Register Pseudo Functions**
- **MIPS Processor-Specific Optimizations**
- **Assembler/ Linker**
- **Assembler Syntax Extensions and Limitations**
- **Stack Checking**

Executive and Phase Information

Executive `-tp` enables specific options dependent upon executive mode. Processors and sub-options for **ucc** and **c89** option modes are identified in this section.

Executive -tp Option

`-tp[=]<target>{[,]<suboptions>}`
Specify Target Processor and Target Processor Options

Specify the target processor `<target>` and target processor sub-options. Target processors are identified in **Table 4-1** and `-tp` sub-options are identified in **Table 4-2**.

Table 4-1 Target Processor

Target	Target Processor
MIPS	Generic MIPS
MIPS3000	MIPS 3000
MIPS32	Generic MIPS32
IDT3081	IDT MIPS 3081
TX3900	Toshiba MIPS 3900
MIPS64	Generic MIPS64
MIPS64PFP	Generic MIPS64PFP (with paired FPU registers)
IDT4650	IDT MIPS 4650

Table 4-1 Target Processor (continued)

Target	Target Processor
MIPS64DFP	Generic MIPS64DFP (with 64-bit FPU registers)
IDT4700	IDT MIPS 4700

Table 4-2 Mode -tp Sub-Options

Suboptions	Description
`sd`	Use 16-bit data references (default)
`ld`	Use 32-bit data references
`scd`	Use 16-bit code area data references (default)
`lcd`	Use 32-bit code area data references
`fp`	Use static link library for floating-point support
`sc`	Use 16-bit code (functions only) references (default)
`lc`	Use 32-bit code (functions only) references
`sb`	Use 16-bit branches (default)
`lb`	Use 32-bit branches

Predefined Macro Names for the Preprocessor

The macro names in **Table 4-3** are predefined in the preprocessor for target systems.

Table 4-3 Macros

Macro	Description
`_MPFMIPS`	Generic MIPS processor
`_MPFMIPS3000`	MIPS 3000 processor
`_MPFMIPS32`	MIPS32 processor
`_MPFMIPS64`	MIPS64 processor
`_MPFMIPS64PFP`	MIPS64PFP processor
`_MPFMIPS64DFP`	MIPS64DFP processor
`_MPFIDT3081`	IDT 3081 processor
`_MPFIDT4650`	IDT 4650 processor
`_MPFIDT4700`	IDT 4700 processor
`_MPFTX3900`	Toshiba 3900 processor
`_FPFMIPS`	MIPS floating point processor

Target names specify the compiler to use when writing machine-independent and operating system-independent programs.

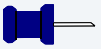
The executive defines the `_MPF` macro for the target processor as well as any processors that are generally considered subsets of the target processor. **Table 4-4** provides a few examples of this behavior.

For more information on exactly which macros are defined for the MIPS processors, run the executive in verbose and dry run modes stopping after the front end. For example, to check the defines for the MIPS3000 target (source file not required):

```
xcc -b -h -efe -tp=MIPS3000 t.c
```

This causes the executive to print a line similar to:

```
"cpfe -m --target=9 -Id:\MWOS\SRC\DEFS
-Id:\MWOS\OS9000\SRC\DEFS
-Id:\MWOS\OS9000\MIPS3000\DEFS
-Id:\MWOS\OS9000\MIPS\DEFS
-D_UCC -D_MAJOR_REV=2 -D_MINOR_REV=3 -D_SPACE_FACTOR=1 -D_TIME_FACTOR=1
-D_OS9000 -D_MPFMIPS3000 -D_MPFMIPS -D_FPFMIPS -D_BIG_END -w
--Extended_ANSI --gen_c_file_name=t.i t.c"
```



Note

Note that both `_MPFMIPS` and `_MPFMIPS3000` are defined.

The `_MPFMIPS` macro indicates that a source file is being compiled for a MIPS family target.

Table 4-4 identifies the relationship between the target processor and the preprocessor macros. Note that specific targets also define the subset macros (For example, when targeting the MIPS3000, the _MPFMIPS and _MPFMIPS3000 are defined).

Table 4-4 _MPFxxx Macro Behavior

Target	Microprocessor Family Macros Defined
Generic MIPS	`_MFPMIPS`
MIPS 3000	`_MPFMIPS _MPFMIPS3000`
MIPS32	`_MPFMIPS _MPFMIPS32`
MIPS64	`_MPFMIPS _MPFMIPS64`

Table 4-4 _MPFxxx Macro Behavior (continued)

Target	Microprocessor Family Macros Defined
MIPS64PFP	`_MPFMIPS _MPFMIPS64 _MPFMIPS64PFP`
MIPS64DFP	`_MPFMIPS _MPFMIPS64 _MPFMIPS64DFP`
IDT 3081	`_MPFMIPS _MPFMIPS3000 _MPFIDT3081`
IDT 4650	`_MPFMIPS _MPFMIPS64 _MPFMIPS64PFP _MPFIDT4650`
IDT 4700	`_MPFMIPS _MPFMIPS64 _MPFMIPS64DFP _MPFIDT4700`
Toshiba 3900	`_MPFMIPS _MPFMIPS3000 _MPFTX3900`

MIPS-Unique Phase Option Functionality

Phases having unique phase option functionality on the MIPS processor are:

- Back end
- Assembly optimizer
- Linker

Back End Options

`-m=<non remote memory left>`

Informs the back end that other files in the program have used some amount of the 64K data area.

The back end orders the data area based on static analysis of the data area objects and sorts the data based on usage and size. This means that the most heavily used objects end up in the non-remote area. To do this, the back end needs information about how the object linker will lay out the data area for the entire program.

Code generation options provide specifications for code generated by the back end.

Table 4-5 Code Generation Options

Option	Description
`-pg`	Generate code to derive `cp` ($30) rather than relying on a globally set `cp` for each function that needs it. This option might be used for non-program modules that have multiple entry points.
`-pl`	Cause references to data objects to be long
`-plc`	Cause references to constant data objects to be long
`-plb`	Cause references to functions to be long
`-pla`	Cause all branches to be long
`-pm=<n>`	Average memory latency for loads, in cycles (default 7)
`-pnbs`	Don’t perform bit shift elimination optimizations.
`-pnf`	Do not emit FP (`cop1`) operations
`-pnfm`	Do not use FP registers to copy data If this option is not selected, the back end looks for opportunities to copy data from one location to another using FP registers.

Table 4-5 Code Generation Options (continued)

Option	Description
`-pnp`	Do not store previous `sp` in prologue (hinders debugging)
`-pnr`	Do not add register liveness information as comments (for assembly optimizer)
`-ps`	No stack checking code
`-pu`	Use unordered (non-signalling) fp compares
`-pv`	Do not use `trunc`, `round`, `floor`, or `ceil` instructions.

Target architecture code generation options provide specifications unique to a target architecture for code generated by the back end.

`-p<target architecture>`

Identifies the target architecture for which to generate code. Implementation of multiply and divide instructions differs based upon target architecture.

Table 4-6 <target architecture> Code Generation Options

Option	Description
`-pmips`	Generate code for the generic MIPS target (default). This code should run on any processor (although `fpu` emulation may be necessary). Only MIPS-I instructions are generated with the exception of some floating point instructions (`double load/store`, `trunc`, and `round`) which provide for smaller code and allow for paired or 64-bit `fpu` registers. `Nop`'s fill delay slots (e.g., `loads` and `branches`). The back end does not choose instructions specific to a processor; if MIPS 3000-specific instructions appear in assembly language escapes, the compiled code does not port to other MIPS family processors.
`-pmips3000`	Generate code identical to that for the generic MIPS target except that only MIPS-I instructions are used.
`-pmips32`	Generate code for MIPS32 processors.
`-pmips64`	Generate code for MIPS64 processors. MIPS-III 64-bit integer instructions support the C `long long` data type.

Table 4-6 <target architecture> Code Generation Options (continued)

Option	Description
-pmips64pfp	Generate code identical to MIPS64 except that paired floating point registers are assumed.
-pmips64dfp	Generate code identical to MIPS64 except 64-bit floating point registers are assumed, so odd-numbered registers may be used.

Assembly Optimizer Options

-p=<X> Selectively skip processor-specific optimizations

Table 4-7 Processor-Specific Assembly Optimizations

<X>	Processor-Specific Optimization
d	Delay slot filling
l	Location tracking
n	Register renaming
r	Copy propagation

`-s<method>`	Set the peephole scheduling method

Table 4-8 Peephole Scheduling Methods

Method	Description
s	Spread dependent instructions
c	Compress floating point instructions
b	Both spread and compress (default except for 4700)
n	No reordering of instructions (not recommended)

`-t[=]<num>`	Specify target processor family

Table 4-9 Assembly Optimizer Processor Numbers

Number	Assembly Optimizer Target
1	Generic MIPS Processor (default)
2	MIPS3000
3	MIPS64
4	IDT4650
5	IDT4700
6	TX3900

For More Information

See **MIPS Processor-Specific Optimizations** on page 175. for more information.

Linker Options

`-t=<target>` Linker, specify target module type

Table 4-10 Target Module Type

Target	Module Type
`os9k_mips`	OS-9 for MIPS

C/C++ Application Binary Interface Information

Register usage, passing arguments to functions, callee saved registers, and language features are described in this section.

Register Usage

General purpose, floating point, and other registers are defined in this section. The register classes are listed and explained below.

- General Purpose Registers (GPRs)
- Floating Point Registers (FPRs)
- Multiply/Divide HI Register
- Multiply/Divide LO Register

Table 4-11 General Purpose Registers

Register Name	Description	$ Syntax
`zero`	Constant zero	$0
`AT`	Compiler temporary	$1
`v0, v1`	Function return (allows for 64-bit)	$2, $3
`a0 - a3`	Incoming args	$4 - $7
`t0 - t7`	Temporaries	$8 - $15
`s0 - s7`	Saved temporaries	$16 - $23

Table 4-11 General Purpose Registers (continued)

Register Name	Description	$ Syntax
t8, t9	Register variables and compiler temporaries	$24, $25
k0, k1	Exception handling	$26, $27
gp	Global data pointer	$28
sp	Stack pointer	$29
cp	Constant Data Pointer	$30
ra	Return Address	$31

* If register is not in use for above-stated use, may be used for integral user register variables and compiler temporaries.

† $30 is used to access **const** qualified data in the code area of the module. This is accomplished by using the register $30 as a biased (by 32K-16 [0x7ff0] bytes) pointer to the code area data. $30 is automatically initialized by the kernel for program modules. Non-program modules must either set $30 up themselves or use the back end option -pg to generate the code to set $30 up for each function that needs it.

The values in v0, v1, a0 through a3, t0 through t9, v0, and v1 need not be preserved across a function call. That is, a function is safe to use these registers without saving and restoring their values.

The compiler does not use at, k0, or k1.

The compiler uses the remainder of the integral registers for integral user register variables and compiler temporaries.

Floating Point Registers

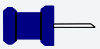
Table 4-12 Floating Point Registers

MIPS/MIPS32/ 4PFP Register	MIPS64/4DFP Register	Description
f0/f1	f0	Floating point return value
f2/f3-f10/f11	f2-f11	Floating point register variables and temporaries
f12/f13	f12	1st floating point argument passed
f14/f15	f14	2nd floating point argument passed
	f1	Floating point register variable and temporary
	f13	Floating point register variable and temporary
	f15	Floating point register variable and temporary
f16/f17-f18/f19	f16-f19	Floating point register variables and temporaries
f20/f21-f30/f31	f20-f31	Floating point register variable

Only the even-numbered registers are used for argument passing (`f12` and `f14`) and function return (`f0`). For targets that have 64-bit floating point registers, the following odd-numbered register is used as a temporary.

When not in use for argument passing, any of `f12` through `f15` may be used as temporary register variables.

Functions may use the values in `f0` through `f19` without saving/restoring them for the functions' callers; a function must save them if they are expected to maintain their values across a call to a function.



Note

If you are compiling a driver, file manager, ticker, or other system component that uses floating-point and you get unresolved references to the symbols below you can resolve the references by adding the following floating-point values to your module.

```
const double __zero[] = {0.0, 0.0};
const float __float_pos_1 = 1.0;
const float __float_neg_1 = -1.0;
const float __float_2_pow_31 = ((float)0x8000) * 0x10000;
const float __float_2_pow_32 = ((float)0x10000) * 0x10000;
const float __float_2_pow_63 = ((float)0x8000) * 0x10000 * 0x10000 *
0x10000;
const float __float_2_pow_64 = ((float)0x10000) * 0x10000 * 0x10000 *
0x10000;


const double __double_pos_1 = 1.0;
const double __double_neg_1 = -1.0;
const double __double_2_pow_31 = ((double)0x8000) * 0x10000;
const double __double_2_pow_32 = ((double)0x10000) * 0x10000;
const double __double_2_pow_63 = ((double)0x8000) * 0x10000 * 0x10000
* 0x10000;
const double __double_2_pow_64 = ((double)0x10000) * 0x10000 * 0x10000
* 0x10000;
```

These values should be compiled to a ROF (.r) file. This ROF file should be included on the link line directly after the root psect (e.g. fmstart.r, drvstart.r, etc.). Because these values are accessed with 16-bit offsets, they must appear early in the link statement to be easily accessible from the CP.

Special Purpose Registers

The multiply/divide `HI` and `LO` registers are used by the compiler when performing multiplication, division, and modulus operations.

Any co-processor registers are available for hand-written assembly language use, although values may need to be saved and restored.

Passing Arguments to Functions

When an argument is passed to a called function, the argument is in one of two places, in a register or in the Output Parameter Area (OPA) of the calling function.

The called function determines the location of the argument by argument type and the order specified in the argument list.

For this discussion:

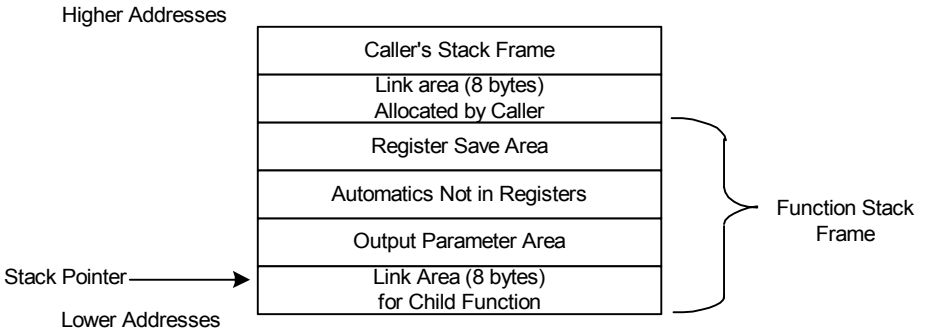
- An **integral argument** is an argument of type `int`, a pointer, or a `char` or `short` converted to an `int`.
- A **floating point argument** is an argument of type `double` or a `float` converted to a double.

There are four integral registers used for parameter passing: `$4` through `$7`, inclusive. Two registers are available for floating point argument passing: `f12` and `f14` (and their paired registers on appropriate processors).

The OPA is also used to pass arguments (when the registers have been exhausted). **Figure 4-1** illustrates a stack frame for a function.

The link area is allocated by the caller as a location where the function can store its link register and current stack pointer (before additional allocation occurs).

Figure 4-1 Stack Frame for a Function



The basic algorithm the compiler uses to pass arguments is as follows:

```
if function returns a struct
    put address of struct return area into first integral passing register
while still more arguments
    if parameter is part of variable arguments
        put argument into next position in OPA
    else if parameter is a struct
        copy struct into next position in OPA
    else
        if argument is integral
            if an integral passing register is available
                put argument into integral register
            else
            put argument into next position in OPA
        else if argument is floating-point
            if a floating-point passing register is available
                put argument into floating-point register
            else
                put argument into next position in OPA
    advance to next argument
```

The OPA is filled from lowest address to highest address.

Struct arguments and parameters that comprise the variable arguments to a variable argument function are always passed on the OPA. If a function is to return a value, an integral return value is returned in `$2` or a floating point return value is returned in `f0`. If a function is to return a struct, the address of a return area is passed as the first integral argument, in `$4`.

C Language Features

In conformance with the ANSI/ISO C specification, the implementation-defined areas of the compiler are listed in this section. Each bulleted item contains one implementation-defined issue. The number in parentheses included with each bulleted item indicates the location in the ANSI/ISO specification where further information is provided.

For More Information

Other implementation-defined areas are included in the ***Using Ultra C/C++*** manual and the ***Ultra C Library Reference*** manual. Refer to an ANSI/ISO specification for more information.

Characters

- The number of bits in a character in the execution character set (5.2.4.2.1).

  There are eight bits in a character in the execution character set.

Integers

- The representations and sets of values of the various integer types (6.1.2.5).

Table 4-13 Integer Type/Range

Type	Representation	Minimum / Maximum
`char`,`signed char`	8-bit 2's complement	`-128 / 127`
`unsigned char`	8-bit binary	`0 / 255`
`short int`	16-bit 2's complement	`-32768 / 32767`
`unsigned short int`	16-bit binary	`0 / 65535`
`int`	32-bit 2's complement	`-2147483648 / 2147483647`
`unsigned int`	32-bit binary	`0 / 4294967295`
`long int`	32-bit 2's complement	`-2147483648 / 2147483647`
`unsigned long int`	32-bit binary	`0 / 4294967295`

Table 4-13 Integer Type/Range (continued)

Type	Representation	Minimum / Maximum
`long long *`	64-bit 2's complement	-2^{63} / $2^{63}-1$
`unsigned long long *`	64-bit binary	0 / $2^{64}-1$

* MIPS64 targets only (MIPS-III ISA required). `long long` is not a part of the current ANSI standard.

- The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2).

  When converting a longer integer to a shorter signed integer, the least significant <n> bits of the longer integer are moved to the integer of <n> bits. The resulting value in the smaller integer is dictated by the representation. For example, if the conversion is from `int` to `short`, then the least significant 16 bits are moved from the `int` to the `short`. This value is then considered a 2's complement 16-bit integer.

  When conversion from unsigned to signed occurs with equally sized integers, the most significant bit becomes the sign bit. Therefore, if the unsigned integer is less than `0x80000000`, the conversion has no affect. Otherwise, a negative number results.

- The sign of the remainder on integer division (6.3.5).

  The result of an inexact division of one negative and one positive integer is the smallest integer greater than or equal to the algebraic quotient.

  The sign of the remainder on integer division is the same as that of the dividend.

Floating Point

- The representations and sets of values of the various types of floating-point numbers (6.1.2.5).

Table 4-14 Floating Point Number Characteristics

Type	Format	Minimum / Maximum
float	32 bit IEEE 754	`1.17549435e-38f / 3.40282347e38f`
double	64 bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`
long double	64 bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`

Refer to the `float.h` header file for other limits and values.

Arrays and Pointers

- The type of integer required to hold the maximum size of an array. That is, the type of the size of operator, `size_t` (6.3.3.4, 7.1.1).

  An `unsigned long int` is required to hold the maximum size of an array. `unsigned long int` is defined as `size_t` in `ansi_c.h`.
- The result of casting a pointer to an integer or vice versa (6.3.4).

  Since pointers are treated much like unsigned long integers, the integer will be promoted using the usual promotion rules to an unsigned long. That is, the sign bit propagates out to the full 32-bit width.
- The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (6.3.6, 7.1.1).

A `signed long int` is required to hold the difference between two pointers to elements of the same array. `long int` is defined as `ptrdiff_t` in `ansi_c.h`.

Registers

- The extent to which objects can actually be placed in registers by use of the register storage-class specifier (6.5.1).

  The compiler automatically makes decisions about what objects are placed in registers, thus giving no special storage considerations for the register storage-class.

Structures, Unions, Enumerations, and Bit-Fields

- The padding and alignment of members of structures (6.5.2.1). This should present no problem unless binary data written by one implementation are read by another.

Table 4-15 shows the alignment of the various objects within a structure. Required padding is supplied if the next available space is not at the correct alignment for the object. For example, a structure declared as:

```
struct {
   char mem1;
   long mem2;
};
```

would be an eight byte structure, one byte for mem1, three bytes of padding to get mem2 to four byte alignment, and four bytes for mem2.

Table 4-15 Alignment Table

Type	Alignment Requirement
char	1
short	2
int	4
long	4
long long	8
pointers	4
float	4
double	8
long double	8

- Whether “plain” `int` bit field is treated as a signed `int` or as an unsigned `int` bit field (6.5.2.1).

  A “plain” `int` bit field is treated as a signed `int` bit-field.
- The order of allocation of bit fields within a unit (6.5.2.1).

  Bit fields are allocated from most significant bit to least significant bit.
- Whether a bit field can straddle a storage-unit boundary (6.5.2.1).

  Bit fields are allocated end-to-end until a non-bit field member is allocated or until that positioning would cross an addressable boundary such that no object of an integral type could both contain the bit field and be correctly aligned.
- The integer type chosen to represent the values of an enumeration type (6.5.2.2).

  `Enum` values are represented in 32-bit two’s complement integers.

Preprocessing Directives

- Whether the value of a single-character character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (6.8.1).

  The value of a single-character character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. This character constant may have a negative value.
- The method for locating includable source files (6.8.2).

  This method is described in the **Using the Executive** chapter of the ***Using Ultra C/C++*** manual.
- The support of quoted names for includable source files (6.8.2).

  Quoted names are supported for #include preprocessing directives.
- The mapping of source file character sequences (6.8.2).

  The mapping of source file character sequences is one-to-one. The case-sensitivity of file names depends on the file system being used.

_asm() Register Pseudo Functions

_asm( ) uses registers pseudo functions as identified in Table 4-16.

Table 4-16 _asm() Register Pseudo Functions

Register	Description
__reg_gen	Any non-dedicated integer register
__reg_float	Any non-dedicated floating point register (single or pair)
__reg_<n>	The register specified by $<n>, <n> is 1 to 31
__reg_zero	Constant zero register; equivalent to __reg_0
__reg_at	Compiler temporary register; equivalent to __reg_1
__reg_v<n>	The register specified by v<n>, <n> is 0 or 1; equivalent to __reg_<N> where <N> is 2 or 3
__reg_a<n>	The register specified by a<n>, <n> is 0 to 3; equivalent to __reg_<N> when N is 4 to 7
__reg_t<n>	The register specified by t<n>, <n> is 0 to 9; equivalent to __reg_<N> where N is 8 to 15, 24, or 25
__reg_s<n>	The register specified by s<n>, <n> is 0 to 7; equivalent to __reg_<N> where N is 16 to 23
__reg_k<n>	The register specified by k<n>, <n> is 0 or 1; equivalent to __reg_<N> where N is 26 or 27

Table 4-16 _asm() Register Pseudo Functions

Register	Description
`__reg_gp`	Global pointer register; equivalent to `__reg_28`
`__reg_sp`	Stack pointer register; equivalent to `__reg_29`
`__reg_cp`	Constant pointer register; equivalent to `__reg_30`
`__reg_ra`	Return address register; equivalent to `__reg_31`
`__reg_recall`	Retrieves the register object specified by a previous pseudo-function.
`__reg_hi`	The register containing the most-significant bits of a register pair (odd register); used in conjunction with `__reg_recall` and `__reg_float`
`__reg_lo`	The register containing the least-significant bits of a register pair (even register); used in conjunction with `__reg_recall` and `__reg_float`
`__call`	Used to inform the compiler of the intention to call a subroutine from within an `_asm()`; example: `"__reg_ra(__call())"`
`__reg_callee_save[d]`	Any callee-saved general purpose register
`__reg_caller_save[d]`	Any caller-saved general purpose register

The `__reg_recall`, `__reg_hi`, and `__reg_lo` pseudo-functions are used in a different manner than the other pseudo-functions.

The user may recall a previous `_asm` argument whose value is a register using the `__reg_recall()` pseudo-function, in conjunction with one of `__reg_gen()` or `__reg_float()`. `__reg_recall()` takes one integer argument, the ordinal number of the argument it is to duplicate; it should be specified as the argument to `__reg_gen()` if it is to return an integer register or `__reg_float()` if it is to return a floating point register. See the next item for an example of its usage.

`__reg_hi()` or `__reg_lo()` may be used to extract either the most significant half or the least significant half, respectively, of a paired floating point register. The user normally calls `__reg_float()` to reserve the register in question, then uses `__reg_hi()` or `__reg_lo()` in conjunction with `__reg_recall()` to reference either half. The following example uses `__reg_float()` to allocate a register pair and `__reg_hi()` and `__reg_lo()` to access the individual registers.

```
_asm("
        swc1    %1,0(%3)
        swc1    %2,4(%3)
",
        __reg_float(a),
        __reg_hi(__reg_float(__reg_recall(0))),
        __reg_lo(__reg_float(__reg_recall(0))),
        __reg_gen(&b));
```

MIPS Processor-Specific Optimizations

In addition to providing the standard generic assembly optimizations, the Ultra C/C++ MIPS assembly optimizer (`optmips`), provides processor-specific optimizations. These are:

- **Special Common Sub-Expressions**
- **Delay Slot Filling**
- **Copy Propagation**
- **Register Renaming**
- **Instruction Scheduling**

For More Information

The ***Using Ultra C/C++*** manual contains additional information on assembly level optimizations.

Special Common Sub-Expressions

On the MIPS architecture, certain constants are more expensive to work with than others. To account for this, the common sub-expression elimination optimization is performed on expressions involving potentially expensive constants. These include the following:

- The computation of all floating point constants as none of the floating point instructions allow for floating point immediates.
- The computation of integer constants that require two or more instructions. For example the constant 0xffeeffee. The constant `0xffffffee` would not be considered.

Delay Slot Filling

The assembly optimizer attempts to fill the delay slots of those instructions that have them with useful instructions (in an attempt to reduce code size and/or increase code efficiency). To do this, it looks for a movable instruction in the series of preceding instructions or, in some cases, the following or destination instructions. If the delay slot cannot be filled with a useful instruction, it is either left alone or, in the case of unconditional branches, the delay slot is removed altogether.

Copy Propagation

The assembly code optimizer tries to eliminate needless copies between register temporaries; resulting in smaller, more efficient code. For example:

```
lw $4,=x($28)
nop
move $8,$4
addiu $8,$8,-1
bne $8,$0,=$L1
nop
```

This may be changed to the following:

```
lw $8,=x($28)
nop
addiu $8,$8,-1
bne $8,$0,=$L1
nop
```

Register Renaming

In an effort to conserve registers, the back end often generates code using the same temporary register in sequential instructions, much to the dismay of the instruction scheduler. The assembly optimizer uses different registers when possible to help itself schedule better. For example:

```
* compute w += x + y + z
lw $8,=x+0($28)
nop
lw $9,=y+0($28)
nop
addu $8,$8,$9
lw $9,=z+0($28)
nop
addu $8,$9,$8
lw $9,=w+0($28)
nop
addu $9,$9,$8
sw $9,=w+0($28)
```

This may be changed to the following:

```
lw $8,=x+0($28)
lw $3,=y+0($28)
lw $1,=z+0($28)
lw $9,=w+0($28)
addu $8,$8,$3
addu $8,$1,$8
addu $9,$9,$8
sw $9,=w+0($28)
```

The use of different registers allows for more freedom in instruction scheduling.

Instruction Scheduling

Instruction scheduling is usually performed in an effort to increase generated code speed. This involves spreading dependent instructions to eliminate any latencies between them. For the MIPS, instruction scheduling is also important to ensure proper execution. Some MIPS targets contain pipeline hazards which are not guarded by hardware interlocks. For example, on many MIPS3000 architectures there is a one-cycle delay on the availability of the destination register for a load. The backend inserts `nop` instructions into these delay slots. The assembly optimizer attempts to fill these delay slots with useful instructions when possible.

Other recognized hazards may include: multiply/divide instructions and those that access the `HI` and `LO` registers, floating point comparisons, integer/floating point moves, and integer/control register moves.

Assembler/ Linker

The assembler allows use of standard MIPS assembly language mnemonics and syntax, modified as described in this section. For more specific information about individual instructions, consult the following books:

- ***IDT R30xx Family Software Reference Manual***
- ***IDT79R4640 and IDT79R4650 RISC Processor Hardware User's Manual***
- ***IDT79R4600 and IDT79R4700 RISC Processor Hardware User's Manual***

ROF Edition Number

The MIPS assembler emits ROF Edition #15.

External References

The MIPS assembler allows the use of external references with any operators within any expression fields not defined to be constant expression fields.

Symbol Biasing

The linker does not bias code or data symbols for system, file managers, device drivers, device descriptors, or data modules. For all other types of modules, or for raw code, the linker biases both code and data symbols by `-32752 (-0x7ff0)`. Initialization routines for raw code should ensure that the static storage pointer (`$28`) and constant storage pointer (`$30`) are initialized with the proper base addresses, adjusted to account for this biasing.

Assembler Syntax Extensions and Limitations

The Ultra C Compiler's adaptation of the MIPS instruction syntax has a few notable differences from what is defined in MIPS architecture manuals.

- All Microware assemblers use the space character as the comment delimiter. As a result, the operand stream must not include any spaces.
- The MIPS instruction set limits immediates to 16-bit signed and unsigned values. To allow for manipulation of 32-bit immediate data, the following operators are available in the MIPS assembler:

```
hi(x)   = x >> 16                    /* upper 16 bits */
high(x) = (x >> 16) + ((x >> 15) & 1)  /* upper 16 bits + sign bit of lower 16 */
lo(x)   = x & 0x0000ffff             /* lower 16 bits */
```

- The following code moves the 32-bit value represented by `Symbol` into `GPR a0`:

```
lui     a0,hi(=Symbol)
ori     a0,a0,lo(=Symbol)
```

- If `Symbol` represents the offset of `x` from `GPR gp`, then the following code moves the 32-bit address of `x` int `a0`:

```
lui     a0,high(=Symbol)
addu    a0,a0,gp
add     a0,a0,lo(=Symbol)
```

  Alternatively, the value of `x` can be loaded into `a0` as follows:

```
lui     a0,high(=Symbol)
addu    a0,a0,gp
lw      a0,lo(=Symbol)(a0)
```

- In addition to the MIPS instruction set, the Microware Ultra C/C++ MIPS assembler also accepts the synthetic instructions specified in **Table 4-17**.

Table 4-17 MIPS Assembler Synthetic Instructions

Synthetic Instruction	MIPS Instruction
`move reg^d,reg^1`	`or reg^d,reg^1,zero`
`b label`	`beq zero,zero,label`
`bal label`	`bgezal zero,label`

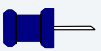
Stack Checking

This section provides MIPS-specific information about stack checking. Refer to ***Using Ultra C/C++*** for more general information on stack checking.

If stack checking is inappropriate for the module being created, you will need to define the following items:

- a global pointer called `_stbot` (initialized to `ULONG_MAX` if possible)
- a function called `_stkhandler` (it returns to its caller)

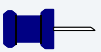
A function called `_stkoverflow` is called when the stack appears to overflow. If a non-static function called `_stkoverflow` resides in an ROF that is linked to make the object module, that function is called instead of the default function. The default function writes a message to the standard error path and causes the program to exit with a status of one. `_stkoverflow` neither accepts parameters nor returns a value.



Note

If `_stkoverflow` is inappropriate for your application, consider writing a function to handle stack overflow.

`_stkhandler`, the function that checks for stack overflow, can be revised. Revision may be necessary if stack checking is inapplicable to the module that calls the library functions. `_stkhandler()` is passed the desired stack pointer in `r3` and does not return a value.



Note

Stack handler code must be compiled with stack checking turned off (`-r` in ucc mode).

Chapter 5: Pentium and 80x86

This chapter provides information specific to the Pentium processor and the 80x86 family of processors. The following sections are included:

- **Executive and Phase Information**
- **C/C++ Application Binary Interface Information**
- **_asm() Register Pseudo Functions**
- **Span Dependent Optimizations**
- **Assembler/ Linker**
- **Assembly Language Mnemonics**

Executive and Phase Information

This section describes the Executive `-tp` option, predefined macro names for the preprocessor, and Pentium- and 80x86-unique phase option functionality.

Executive -tp Option

`-tp[=]<target>` Specify the target processor.

Table 5-1 Target Processors

<target>	Target Processor
80386	I80386
386	I80386
80486	I80486
486	I80486
p5	Pentium

Table 5-2 -tp Sub-Options

<suboptions>	Description
`sd`	Use 8 bit data references
`ld`	Use 32 bit data references (default)

Predefined Macro Names for the Preprocessor

The macro names in **Table 5-3** are predefined in the preprocessor for target systems.

Table 5-3 Macros

Macro	Description
_FPF387	80387 floating point target coprocessor family
_MPF386	80386 target processor
_MPF486	80486 target processor
_MPFP5	Pentium target processor

Target names specify the compiler to use when writing machine- and operating system-independent programs.

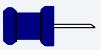
The executive defines the _MPF macro for the target processor as well as any processors that are generally considered subsets of the target processor. **Table 5-4** provides a few examples of this behavior.

For more information on which macros are defined for Pentium and 80x86 target processors, run the executive in verbose and dry run modes, stopping after the front end. For example, to check the defines for the p5 (Pentium) target (source file not required):

```
cc -b -h -efe -tp=p5 t.c
```

This causes the executive to print a line similar to:

```
"cpfe -m --target=1 -I\mwos\SRC\DEFS
-I\mwos\OS9000\SRC\DEFS -I\mwos\OS9000\80386\DEFS
--translate_names_2 -D_UCC -D_MAJOR_REV=2
-D_MINOR_REV=5 -D_SPACE_FACTOR=1 -D_TIME_FACTOR=1
-D_OS9000 -D_MPFP5 -D_MPF486 -D_MPF386 -D_FPF387
-D_LIL_END -D__LONGLONG_BIT=64 -w --Extended_ANSI
--gen_c_file_name=t.i t.c"
```



Note

Note that _MPFP5, _MPF486, and _MPF386 macros are defined.

The _MPF386 macro indicates that a source file is being compiled for an Intel 80x86 family target.

Table 5-4 identifies the relationship between the target processor and the preprocessor macros.

Table 5-4 _MPFxxx Macro Behavior

Target	Microprocessor Macros Defined
80386	_MPF386
80486	_MPF486, _MPF386
Pentium	_MPFP5, _MPF386

80x86-Unique Phase Option Functionality

Phases having unique phase option functionality on the 80x86 processor are:

- **Back End Options**
- **Assembly Optimizer Options**
- **Assembler Options**
- **Linker Options**

Back End Options

Options identified in the following table are available for the back end.

Table 5-5 Information Options

Option	Description
`-pd`	Clear the direction flag at function entry
`-pg`	Do not use data area to calculate code area addresses
`-ps`	Do not emit stack checking code
`-p5`	Emit code for Pentium processor

Assembly Optimizer Options

`-t[=]<num>` Specify Target Processor Family

Table 5-6 Assembly Optimizer Processor Numbers

Number	Assembly Optimizer Target
1	80386
2	486
3	Pentium

`-s<method>` Set the peephole scheduling method

Table 5-7 Peephole Scheduling Methods

Method	Description
s	Spread dependent instructions
c	Compress floating point instructions
n	No reordering of instructions

Assembler Options

`-b` Optimize Branch Sizing

`-m<num>` Specify Microprocessor

Table 5-8 Microprocessor

Number	Description
80386	80386 processor (default)
8086	8086 processor

Linker Options

`-t=<target>` Specify target module type

Table 5-9 Target Module Type

Target	Module Type
os9k_386	OS-9 for x86

C/C++ Application Binary Interface Information

This section describes the register usage, passing arguments to functions, and language features.

Register Usage

The compiler uses registers identified in **Table 5-10**.

Table 5-10 Register Usage

Register	Description
`eax`	Function argument/return register, temporaries, and register variables
`ebp`	Frame pointer
`ebx`	Global static storage pointer
`ecx`	Temporaries and register variables
`edi`	Temporaries and register variables
`edx`	Temporaries and register variables
`esi`	Temporaries and register variables
`esp`	Stack pointer
`st0`	80387 stack top; float/double return register

Passing Arguments to Functions

When arguments are passed to a called function, the argument resides in one of two places, in a register or on the stack.

The called function determines the location of the argument by the following argument type and the order specified in the argument list.

For this section, the following statements are assumed:

- An **integral argument** is an argument of type `int` or `pointer` or `char`, or `short` converted to an `int`.
- A **double argument** is an argument of type `double` or `float` converted to `double`.

The first integral argument is passed in `eax`, and the second integral argument, if any, is pushed on the stack. A single double argument is pushed on the stack as are remaining arguments. If the first argument is integral and the second is double, the integral argument is passed in `eax` and the double is pushed on the stack.

Any `struct` or `long long` arguments are copied to the next location on the stack. A `long long` is pushed such that the most signifcant 32 bits appear at a higher address than the least significant 32 bits.

If a function returns a value, the integral value is returned in `eax`. A `double` (or `float`) value is returned in `st0`. If the returned value is a `struct`, then the address of the return area is passed as an argument to the callee in `edi`; the called function copies the returned `struct` to this location. If the return value is a `long long`, the least significant 32 bits are returned in `eax` and the most significant 32 bits are returned in `edx`.

C Language Features

In conformance with the ANSI/ISO C specification, the implementation-defined areas of Ultra C/C++ are listed in this section. Each bulleted item contains one implementation-defined issue. The

number in parentheses included with each bulleted item indicates the location in the ANSI/ISO specification where further information is provided.

For More Information

Other implementation- defined areas are defined in the **Language Features** chapter of the ***Using Ultra C/C++*** manual and the **Overview** chapter of the ***Ultra C Library Reference*** manual. Refer to an ANSI/ISO specification for more information.

Characters

- The number of bits in a character in the execution character set (5.2.4.2.1).

  There are eight bits in a character in the execution character set.

Integers

- The representations and sets of values of the various integer types (6.1.2.5).

Table 5-11 Integers

Type	Representation	Minimum / Maximum
`char`, `signed char`	8-bit 2's complement	`-128` / `127`
`unsigned char`	8-bit binary	`0` / `255`

Table 5-11 Integers

Type	Representation	Minimum / Maximum
`short int`	16-bit 2's complement	`-32768 / 32767`
`unsigned short int`	16-bit binary	`0 / 65535`
`int`	32-bit 2's complement	`-2147483648 / 2147483647`
`unsigned int`	32-bit binary	`0 / 4294967295`
`long int`	32-bit 2's complement	`-2147483648 / 2147483647`
`unsigned long int`	32-bit binary	`0 / 4294967295`
`long long`	64-bit 2's complement	-2^{63} / $2^{63}-1$
`unsigned long long`	64-bit binary	`0` / $2^{64}-1$

- The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2).

  When converting a longer integer to a shorter signed integer, the least significant <n> bits of the longer integer are moved to the integer of <n> bits. The resulting value in the smaller integer is dictated by the representation. For example, if the conversion is from `int` to `short`, the least significant 16 bits are moved from the `int` to the `short`. This value is then considered a 2's complement 16-bit integer.

When conversion from unsigned to signed occurs with equally sized integers, the most significant bit becomes the sign bit. Therefore, if the unsigned integer is less than `0x80000000`, the conversion has no effect. Otherwise, a negative number results.

The Sign of the Remainder on Integer Division (6.3.5)

- The result of an inexact division of one negative and one positive integer is the smallest integer greater than or equal to the algebraic quotient.

  The sign of the remainder on integer division is the same as that of the dividend.

Floating Point

- The representations and sets of values of the various types of floating-point numbers (6.1.2.5).

Table 5-12 Floating Point Number Characteristics

Type	Format	Minimum / Maximum
`float`	32 bit IEEE 754	`1.17549435e-38f / 3.40282347e38f`
`double`	64 bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`
`long double`	64 bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`

Refer to the `float.h` header file for other limits and values.

Arrays and Pointers

- The type of integer required to hold the maximum size of an array. That is, the type of the `sizeof` operator, `size_t` (6.3.3.4, 7.1.1).

An `Unsigned long int` is required to hold the maximum size of an array. `unsigned long int` is defined as `size_t` in `ansi_c.h`.

- The result of casting a pointer to an integer or vice versa (6.3.4).

  Since pointers are treated much like `unsigned long` integers, the integer is promoted using the usual promotion rules to an `unsigned long`. That is, the sign bit propagates out to the full 32-bit width.

- The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (6.3.6, 7.1.1).

  A `signed long int` is required to hold the difference between two pointers to elements of the same array. `long int` is defined as `ptrdiff_t` in `ansi_c.h`.

Registers

- The extent to which objects can actually be placed in registers by use of the register storage-class specifier (6.5.1).

  The compiler automatically makes decisions about which objects are placed in registers, giving no special storage considerations for the register storage-class.

Structures, Unions, Enumerations, and Bit-Fields

- The padding and alignment of members of structures (6.5.2.1). This should present no problem unless binary data written by one implementation are read by another.

  Non-character structure members and sub-structures containing non-character members are aligned on an even byte boundary. Character structure members have no alignment restrictions.

- Whether “plain” `int` bit field is treated as a signed `int` or as an unsigned `int` bit field (6.5.2.1).

  A plain `int` bit field is treated as a signed `int` bit field.

- The order of allocation of bit fields within a unit (6.5.2.1).

  The bit fields are allocated from least significant bit to most significant bit.

- Whether a bit field can straddle a storage-unit boundary (6.5.2.1).

  Bit fields can straddle a storage unit. That is, bit fields are allocated end-to-end until a non-bit-field member is allocated or 32-bit size is executed.

- The integer type chosen to represent the values of an enumeration type (6.5.2.2).

  `Enum` values are represented in 32-bit two's complement integers.

Preprocessing Directives

- Whether the value of a single-character, character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (6.8.1).

  The value of a single-character character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. This character constant may have a negative value.

- The method for locating includable source files (6.8.2).

  This method is described in the **Using the Executive** chapter of the ***Using Ultra C/C++*** manual.

- The support of quoted names for includable source files (6.8.2).

  Quoted names are supported for `#include` preprocessing directives.

- The mapping of source file character sequences (6.8.2).

  The mapping of source file character sequences is one-to-one. The case-sensitivity of file names depends on the file system being used.

_asm() Register Pseudo Functions

`_asm()` uses registers pseudo functions as identified in **Table 5-13**.

Table 5-13 _asm() Register Pseudo Functions

Register	Description
`__reg_gen`	Any non-dedicated 32 bit integer register
`__reg_bit16`	Any 16-bit integer register with an accessible 8 bit
`__reg_bit8`	Any 8-bit integer register not part of a dedicated 32-bit register
`__reg_eax`	Individual 32-bit register
`__reg_ebx`	Individual 32-bit integer register
`__reg_ecx`	Individual 32-bit register
`__reg_edx`	Individual 32-bit register
`__reg_esi`	Individual 32-bit register
`__reg_edi`	Individual 32-bit register
`__reg_ebp`	Individual 32-bit register
`__reg_esp`	Individual 32-bit register
`__reg_ax`	Individual 16-bit integer registers
`__reg_cx`	Individual 16-bit integer registers
`__reg_dx`	Individual 16-bit integer registers

Table 5-13 _asm() Register Pseudo Functions (continued)

Register	Description
`__reg_al`	AL
`__reg_cl`	CL

Span Dependent Optimizations

The compiler performs **branch shortening**. Branch shortening reduces the instruction size on branch instructions when the distance to the destination is known to be within certain limits.

Assembler/ Linker

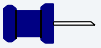
The assembler allows use of standard Intel assembly language. However, the order of operands accepted by the assembler is:

```
<instruction>   <source>,<destination>
```

For more specific information about individual instructions, refer to the appropriate hardware manuals.

ROF Edition #9

The Pentium and 80x86 assembler supports ROF Edition #9.2.



Note

The 386 linker will not accept ROFs/libraries created with pre-UltraC2.4 assembler/libgen.

External References

ROF Edition Number 9.2 is only capable of representing limited expressions involving external references. These expressions can consist only of simple addition and subtraction operations involving two operands at most. The following expression forms involving external references are supported. All other forms are illegal.

```
External + Absolute
External - Absolute
External - External
```

The linker performs subtraction by negating one operand and adding it to the other operand. This method can cause problems on signed values of either word or byte length as the linker may report over/underflow errors. Therefore, expressions involving external names should not be too complex.

For More Information

Refer to the ROF Edition Number 9 Format section in the Assembler and Object Code Linker Overview chapter of the ***Using Ultra C/C++*** manual.

Symbol Biasing

The linker does not bias code symbols for the 80x86/Pentium targets. Data symbols are biased only for system, file manager, device driver, device descriptor, and data modules. For all other module types, the linker biases data symbols by -128 (-0x80) bytes. When generating raw code for these processors, the linker biases data symbols by -128 bytes as well. Initialization routines for such code should be sure to add 128 to the base address loaded into the global static storage pointer (ebp) to accommodate this biasing.

Assembly Language Mnemonics

Table 5-14 Assembly Language Mnemonics

Mnemonic	Description
`aaa`	ASCII adjust after addition
`aad`	ASCII adjust before division
`aam`	ASCII adjust after multiplication
`aas`	ASCII adjust after subtraction
`adc`	Add with carry
`add`	Add integers
`and`	Logical AND
`arpl`	Adjust requested privilege level
`bound`	Check array index against register and then bounds in memory
`bsf`	Bit scan forward
`bsr`	Bit scan reverse
`bt`	Bit test
`btc`	Bit test and complement
`btr`	Bit test and reset

Table 5-14 Assembly Language Mnemonics (continued)

Mnemonic	Description
`bts`	Bit test and set
`call`	Call a procedure
`cbw`	Convert byte to word
`cdq`	Convert dword to qword
`clc`	Clear the carry flag
`cld`	Clear the direction flag
`cli`	Clear the interrupt flag
`clts`	Clear the task-switched flag
`cmc`	Complement the carry flag
`cmp`	Compare. The first operand must be a register or immediate. The second operand must be a register or memory.
`cmps`	Compare string
`cwd`	Convert word to `dword`
`cwde`	Convert word to `dword`
`daa`	Decimal adjust after addition
`das`	Decimal adjust after subtraction
`dec`	Decrement

Table 5-14 Assembly Language Mnemonics (continued)

Mnemonic	Description
`div`	Unsigned divide
`enter`	Create stack frame
`hlt`	Halt
`idiv`	Signed divide
`imul`	Signed multiplication
`in`	Input from a port
`inc`	Increment
`ins`	Input string
`int`	Call to interrupt procedure
`into`	Interrupt on overflow
`iret`	Return from interrupt
`ja`	Jump if above
`jae`	Jump if above or equal
`jb`	Jump if below
`jbe`	Jump if below or equal
`jc`	Jump if carry
`jcxz`	Jump if `CS == 0`

Table 5-14 Assembly Language Mnemonics (continued)

Mnemonic	Description
`je`	Jump if equal
`jecxz`	Jump if `ECX == 0`
`jg`	Jump if greater than
`jge`	Jump if greater than or equal
`jl`	Jump if less than
`jle`	Jump if less than or equal
`jmp`	Jump
`jna`	Jump if not above
`jnae`	Jump if not above or equal
`jnb`	Jump if not below
`jnbe`	Jump if not below or equal
`jnc`	Jump if not carry
`jne`	Jump if not equal
`jng`	Jump if not greater than
`jnge`	Jump if not greater or equal
`jnl`	Jump if not less than
`jnle`	Jump if not less than or equal

Table 5-14 Assembly Language Mnemonics (continued)

Mnemonic	Description
`jno`	Jump if no overflow
`jnp`	Jump if not parity
`jns`	Jump if not sign
`jnz`	Jump if not zero
`jo`	Jump if overflow
`jp`	Jump if parity
`jpe`	Jump if parity even
`jpo`	Jump if parity odd
`js`	Jump if sign
`jz`	Jump if zero
`lahf`	Load flags into AH register
`lar`	Load access rights
`lds`	Load DS segment register
`lea`	Load effective address
`leave`	Procedure exit
`les`	Load ES segment register
`lfs`	Load FS segment register

Table 5-14 Assembly Language Mnemonics (continued)

Mnemonic	Description
`lgdt`	Load global descriptor table
`lgs`	Load GS segment register
`lidt`	Load interrupt descriptor table
`lldt`	Load local descriptor table
`lmsw`	Load machine status word
`lock`	Bus lock
`lods`	Load string
`loop`	Loop control while ECX counter not zero
`loope`	Loop while equal
`loopne`	Loop while not equal
`loopnz`	Loop while not zero
`loopz`	Loop while zero
`lsl`	Load segment limit
`lss`	Load SS segment register
`ltr`	Load task register
`mov`	Move data
`movs`	Move string

Table 5-14 Assembly Language Mnemonics (continued)

Mnemonic	Description
`movsx`	Move with sign extension
`movzx`	Move with zero extension
`mul`	Unsigned multiplication
`neg`	Negate (two's complement)
`nop`	No operation
`not`	Negate (one's complement)
`or`	Logical inclusive OR
`out`	Write to port
`outs`	Output string
`pop`	Pop a word from the stack
`popa`	Pop all registers off stack
`popf`	Pop from stack into flags
`push`	Push onto the stack
`pusha`	Push all onto stack
`pushf`	Push flags onto stack
`rcl`	Rotate left through carry
`rcr`	Rotate right through carry

Table 5-14 Assembly Language Mnemonics (continued)

Mnemonic	Description
`rep`	Repeat
`repe`	Repeat while equal
`repne`	Repeat while not equal
`repnz`	Repeat while not zero
`repz`	Repeat while zero
`ret`	Return
`rol`	Rotate left
`ror`	Rotate right
`sahf`	Store AH register into flags
`sal`	Shift arithmetic left
`sar`	Shift arithmetic right
`sbb`	Subtract with borrow
`scas`	Scan string
`seta`	Set byte above
`setae`	Set byte above or equal
`setb`	Set byte below
`setbe`	Set byte below or equal

Table 5-14 Assembly Language Mnemonics (continued)

Mnemonic	Description
`sete`	Set byte equal
`setg`	Set byte greater than
`setge`	Set byte greater than or equal
`setl`	Set byte less
`setle`	Set byte less than or equal
`setz`	Set byte zero
`setna`	Set byte not above
`setnae`	Set byte not above or equal
`setnb`	Set byte not below
`setnbe`	Set byte not below or equal
`setne`	Set byte not equal
`setng`	Set byte not greater than
`setnge`	Set byte not greater than or equal
`setnl`	Set byte not less than
`setnle`	Set byte not less than or equal
`setno`	Set byte no overflow
`setnp`	Set byte not parity

Table 5-14 Assembly Language Mnemonics (continued)

Mnemonic	Description
`setns`	Set byte not sign
`setnz`	Set byte not zero
`seto`	Set byte overflow
`setp`	Set byte parity
`setpe`	Set byte parity even
`setpo`	Set byte parity odd
`sets`	Set byte sign
`sgdt`	Store global descriptor table
`shl`	Shift logical left
`shld`	Double precision shift left
`shr`	Shift logical right
`shrd`	Double precision shift right
`sidt`	Store interrupt descriptor table
`sldt`	Store local descriptor table
`smsw`	Store machine status word
`stc`	Set carry flag
`std`	Set direction flag

Table 5-14 Assembly Language Mnemonics (continued)

Mnemonic	Description
`sti`	Set interrupt flag
`stos`	Store string
`str`	Store task register
`sub`	Subtract
`test`	Logical compare
`verr`	Verify a segment for reading
`verw`	Verify a segment for writing
`wait`	Wait for coprocessor
`xchg`	Exchange
`xlat`	Table look-up translation
`xor`	Logical exclusive OR

Table 5-15 Floating Point Mnemonics

Mnemonic	Description
`fabs`	Absolute value
`fadd`	Addition
`faddp`	Addition

Table 5-15 Floating Point Mnemonics (continued)

Mnemonic	Description
`fbld`	BCD load
`fbstp`	BCD store and pop
`fchs`	Change sign
`fcom`	Compare
`fcomp`	Compare
`fcompp`	Compare
`fcos`	Cosine
`fdecstp`	Decrement stack pointer
`fdiv`	Division
`fdivp`	Division
`fdivr`	Division reverse
`fdivrp`	Division reverse
`ffree`	Free register
`fincstp`	Increment stack pointer
`fld`	Real load
`fld1`	Load 1
`fldc`	Load control word

Table 5-15 Floating Point Mnemonics (continued)

Mnemonic	Description
`fldenv`	Load environment
`fldl2e`	Load $\log_2 e$
`fldl2t`	Load $\log_2 10$
`fldlg2`	Load $\log_{10} 2$
`fldln2`	Load $\log_e 2$
`fldpi`	Load π
`fldz`	Load zero
`fmul`	Multiply
`fmulp`	Multiply
`fnclex`	Clear exceptions
`fninit`	Initialize processor
`fnop`	No operation
`fnsave`	Save state
`fnstc`	Store control word
`fnstenv`	Store environment
`fnsts`	Store status word
`fpatan`	Partial arctangent

Table 5-15 Floating Point Mnemonics (continued)

Mnemonic	Description
`fprem`	Partial remainder
`fprem1`	Partial remainder (IEEE)
`fptan`	Partial tangent
`frndint`	Round to integer
`frstor`	Restore state
`fscale`	Power of two scaling
`fsin`	Sine
`fsincos`	Sine and cosine
`fsqrt`	Square root
`fst`	Real store
`fstp`	Real store and pop
`fsub`	Subtraction
`fsubp`	Subtraction
`fsubr`	Subtraction reverse
`ftst`	Test
`fucom`	Unordered compare
`fucomp`	Unordered compare

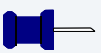
Table 5-15 Floating Point Mnemonics (continued)

Mnemonic	Description
`fucompp`	Unordered compare
`fxam`	Examine
`fxch`	Exchange registers
`fxtract`	Extract exponent and significand
`fyl2x`	$y * \log_2 x$
`fyl2xp1`	$y * \log_2(x + 1)$
`f2xm1`	$2^x - 1$

If stack checking is inappropriate for the module being created, define the following:

- 32-bit global called `_stklimit` (initialized to a large positive value if possible)
- Function called `_stkhandler` that just returns to its caller

A function called `_stkoverflow` is called when the stack appears to overflow. If a non-static function called `_stkoverflow` resides in an ROF that is linked to make the object module, that function is called instead of the default function. The default function writes a message to the standard error path and causes the program to exit with a status of one. `_stkoverflow` neither accepts parameters nor returns a value.

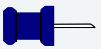


Note

If `_stkoverflow` is inappropriate for your application, consider writing a function to handle stack overflow.

The function that checks for stack overflow, `_stkhandler`, may be revised. This may be necessary if stack checking is inapplicable to the module that calls the library functions. `_stkhandler()` neither accepts parameters nor returns a value.

The following source files (Default Stack Handler Function and Default Stack Overflow Message and Exit) contain the code for the stack checking and error exiting routines for 80x86.



Note

Stack handler code must be compiled with stack checking turned off (`-r` in `ucc` mode).

Default Stack Handler Function

```
/* typedef to the 1 byte unit so pointer arithmetic is easy */
typedef unsigned char byte;

static byte *__asm_get_stack();     /* get current stack pointer */
static void __asm_put_stack(byte *);/* set current stack pointer */

/*
   _stkhandler()
   Checks for stack overflow. _stklimit will be set with the negative
   value of the number of bytes that the function needs. This function
   does not take too much advantage of old information in the globals
   because old stack checking code does not update it.
*/
void _stkhandler()
{
   byte *sp;           /* stack pointer */

   /*
      Figure out what stack limit should really be.
      This is necessary because we may have gotten here after an
      arbitrary number of calls to the old stack checking code which
      only modifies _stbot.
   */
   if ((_stklimit = (sp = __asm_get_stack())) - (byte *)_stbot) < 0) {
      _stbot = sp;
      _stklimit = 0;
```

```
        if (sp <= (byte *)_mtop) {    /* overflow? */
            __asm_put_stack(sp - 256);
            _stklimit = 256;
            _stkoverflow();
        }

        _maxstack = (byte *)_sttop - sp;/* reset maximum so far */
    }
}

static byte *__asm_get_stack(void)
{
    register byte *stack_ptr;

    _asm(" mov.l %0,%1",  __reg_esp(),
                          __reg_gen(__obj_assign(stack_ptr)));

    return stack_ptr;
}

static void __asm_put_stack(new_sp)
byte *new_sp;
{
    _asm(" mov.l %1,%0", __reg_esp(), __reg_gen(new_sp));
}
```

Default Stack Overflow Message and Exit

```
static const char ovf[] = "**** Stack Overflow ****\n";

/*
   _stkoverflow()
   print a message and exit
*/
void _stkoverflow()
{
   /* write message above to stderr and exit */
   u_int32 size = sizeof(ovf);

   if (stderr->_flag & _WRITE)
     _os_writeln(_fileno(stderr), (void *)ovf, &size);
   _os_exit(EOS_STKOVF);
}
```

Chapter 6: PowerPC

This chapter contains information specific to the PowerPC family of processors. The following sections are included:

Executive and Phase Information

Executive `-tp` enables specific options dependent upon executive mode. Processors and sub-options for *ucc* and *c89* option modes are identified in this section.

Executive -tp Option

`-tp[=]<target>{[,]<suboptions>}` Specify Target Processor and Target Processor Options

Specify the target processor `<target>` and target processor sub-options. Target processors are identified in **Table 6-1** and `-tp` sub-options are identified in **Table 6-2**.

Table 6-1 Target Processor

Target	Target Processor
PPC	Generic PPC
403	PPC403
405	PPC405
505	MPC505
555	MPC555
601	MPC601
602	MPC602
603	MPC603
604	MPC604

Table 6-1 Target Processor (continued)

Target	Target Processor
750	MPC750
821	MPC821
860	MPC860
8240	MPC8240
8260	MPC8260

Table 6-2 Mode -tp Sub-Options

Suboptions	Description
`sd`	Use 16-bit data references (default)
`ld`	Use 32-bit data references
`scd`	Use 16-bit `const` data references (default)
`lcd`	Use 32-bit `const` data references
`fp`	Use static link library for floating-point support

Predefined Macro Names for the Preprocessor

The macro names in **Table 6-3** are predefined in the preprocessor for target systems.

Table 6-3 Macros

Macro	Description
_MPFPOWERPC	Generic PowerPC processor
_MPFPPC403	PowerPC 403 processor
_MPFPPC405	PowerPC 405 processor
_MPFPPC505	PowerPC 505 processor
_MPFPPC555	PowerPC 555 processor
_MPFPPC601	PowerPC 601 processor
_MPFPPC602	PowerPC 602 processor
_MPFPPC603	PowerPC 603 processor
_MPFPPC604	PowerPC 604 processor
_MPFPPC750	PowerPC 750 processor
_MPFPPC821	PowerPC 821 processor
_MPFPPC860	PowerPC 860 processor
_MPFPPC8240	PowerPC 8240 processor

Table 6-3 Macros (continued)

Macro	Description
_MPFPPC8260	PowerPC 8260 processor
_FPFPOWERPC	PowerPC floating point processing

Target names specify the compiler to use when writing machine- and operating system-independent programs.

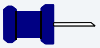
The executive defines the _MPF macro for the target processor as well as any processors that are generally considered subsets of the target processor. **Table 6-4** provides a few examples of this behavior.

For more information on exactly which macros are defined for the PowerPC processors, run the executive in verbose and dry run modes stopping after the front end. For example, to check the defines for the 601 target (source file not required):

```
cc -b -h -efe -tp=601 t.c
```

This causes the executive to print a line similar to:

```
"cpfe -t=4 -x -v=/dd/MWOS/SRC/DEFS
  -v=/dd/MWOS/OS9000/SRC/DEFS
  -v=/dd/MWOS/OS9000/PPC/DEFS -d_UCC -d_SPACE_FACTOR=1
  -d_TIME_FACTOR=1 -d_OS9000 -d_MPFPPC601 -d_MPFPOWERPC
  -d_FPFPOWERPC -d_BIG_END -o=t.i t.c"
```



Note

Note that both _MPFPPC601 and _MPFPOWERPC macros are defined.

The _MPFPOWERPC macro indicates that a source file is being compiled for a PowerPC family target.

Table 6-4 identifies the relationship between the target processor and the preprocessor macros. Note that specific targets also define the subset macros (e.g., when targeting the *601*, both _MPF601 and _MPFPOWERPC are defined).

Table 6-4 _MPFxxx Macro Behavior

Target	Microprocessor Family Macros Defined
Generic PowerPC	_MPFPOWERPC
403	_MPFPOWERPC _MPFPPC403
405	_MPFPOWERPC _MPFPPC405
505	_MPFPOWERPC _MPFPPC505
555	_MPFPOWERPC _MPFPPC555
601	_MPFPOWERPC _MPFPPC601
602	_MPFPOWERPC _MPFPPC602

Table 6-4 _MPFxxx Macro Behavior

Target	Microprocessor Family Macros Defined
603	`_MPFPOWERPC _MPFPPC603`
604	`_MPFPOWERPC _MPFPPC604`
750	`_MPFPOWERPC _MPFPPC750`
821	`_MPFPOWERPC _MPFPPC821`
860	`_MPFPOWERPC _MPFPPC860`
8240	`_MPFPOWERPC _MPFPPC8240`
8260	`_MPFPOWERPC _MPFPPC8260`

PowerPC-Unique Phase Option Functionality

Phases having unique phase option functionality on the PowerPC processor are:

- **Back End Options**
- **Assembly Optimizer Options**
- **Linker Options**

Back End Options

`-m=<non remote memory left>`

Informs the back end that other files in the program have used some amount of the 64K data area.

The back end orders the data area based on static analysis of the data area objects and sorts the data based on usage and size. This means that the most heavily used objects end up in the non-remote area. To do this, the back end needs information about how the object linker will lay out the data area for the entire program.

Code generation options provide specifications for code generated by the back end.

Table 6-5 Code Generation Options

Option	Description
`-pl`	Cause references to external data to be long
`-plc`	Causes some references to code symbols to be long
`-ps`	No stack checking code
`-pnd`	Use non-destructive stack checking, not for use with -ps nor -pc

Table 6-5 Code Generation Options (continued)

Option	Description
`-pnr`	Do not emit register liveness information for the assembly language optimizer.
`-pc`	Force all registers to be callee-saved
`-pg`	Causes the back end to generate code to derive `r13` rather than relying on a globally set `r13` for each function that needs it. This option might be used for non-program modules that have multiple entry points.

Target architecture code generation options provide specifications unique to a target architecture for code generated by the back end.

`-p<target architecture>`

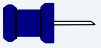
Identifies the target architecture for which to generate code. Implementation of string instructions in hardware and alignment of memory accesses for load and store instructions differs based upon target architecture.

Table 6-6 <target architecture> Code Generation Options

Option	Description
`-p403,p405`	Generate code specifically for PowerPC architectures that implement string instructions in hardware but require memory references to be aligned.

Table 6-6 <target architecture> Code Generation Options (continued)

Option	Description
`-p601`	Generate code specifically for PowerPC architectures that implement string instructions in hardware and allow misaligned accesses for load and store instructions.
`-p602`	Generate code specifically for PowerPC architectures that do not implement string instructions in hardware but allow misaligned accesses for load and store instructions.



Note

The executive instructs the back end to emit the most appropriate code for the target named in the executive `-tp` option. In the default case (`-tp=PPC`), a target architecture option is not sent to the back end. This implies code generation for an architecture which does not implement string instructions in hardware and that requires alignment of memory accesses for load and store instructions.

Assembly Optimizer Options

`-s<method>`	Set the peephole scheduling method

Table 6-7 Peephole Scheduling Methods

Method	Description
s	Spread dependent instructions
c	Compress floating point instructions
t	Target-driven scheduling
w	Target-driven scheduling, with compression of floating point instructions
n	No reordering of instructions

`-t[=]<num>`	Specify target processor family

Table 6-8 Assembly Optimizer Processor Numbers

Number	Assembly Optimizer Target
1	Generic PowerPC
2	PPC403
3	MPC505
4	MPC601
5	MPC603
6	PPC602

Table 6-8 Assembly Optimizer Processor Numbers (continued)

Number	Assembly Optimizer Target
7	MPC604
8	MPC821
9	MPC860
10	MPC750
11	MPC8260

`-p=<X>`	Selectively skip processor-specific optimizations

Table 6-9 Assembly Optimizer Processor-Specific Optimizations

<X>	Processor-Specific Optimization
l	Location Tracking
r	Copy propagation
n	Register naming

Linker Options

`-t=<target>`	Linker, specify target module type

Table 6-10 Target Module Type

Target	Module Type
os9k_ppc	OS-9 for PowerPC

C/C++ Application Binary Interface Information

Register usage, passing arguments to functions, callee saved registers, and language features are described in this section.

Register Usage

General purpose, floating point, and other registers are defined in this section.

Table 6-11 Register Classes

Register Class	Names Used
General Purpose Registers (GPRs)	r0 - r31
Floating Point Registers (FPRs)	f0 - f31
Condition Register Fields (CRFs)	cr0 - cr7
Special Purpose Registers (SPRs)	Refer to **Table 6-19**
Time Base Registers (TBRs)	Refer to **Table 6-21**
Device Control Registers (DCRs)	Refer to **Table 6-22**

General Purpose Registers

Table 6-12 General Purpose Registers

Register	Description
`r0`	Function prologue/epilog, compiler temporary *
`r1`	Stack pointer
`r2`	Static storage pointer
`r3`	1st integral argument passed, integral return value*
`r4`	2nd integral argument passed *
`r5`	3rd integral argument passed *
`r6`	4th integral argument passed *
`r7`	5th integral argument passed *
`r8`	6th integral argument passed *
`r9`	7th integral argument passed *
`r10`	8th integral argument passed *
`r13`	Constant storage pointer †

* If register is not in use for above stated use, may be used for integral user register variables and compiler temporaries.

† `r13` is used to access `const` qualified data in the code area of the module. This is accomplished by using the register `r13` as a biased (by 32K-4 bytes) pointer to the code area data. `r13` is automatically initialized by the kernel for program modules. Non-program modules must either set `r13` up themselves or use the back end option `-pg` to generate the code to set `r13` up for each function that needs it.

The values in `r0` and `r3` through `r12` need not be preserved across a function call. That is, a function is safe to use these registers without saving and restoring their values.

The compiler uses the remainder of the integral registers for integral user register variables and compiler temporaries.

Floating Point Registers

Table 6-13 Floating Point Registers

Register	Description
`f1`	1st floating point argument passed, floating point return value
`f2`	2nd floating point argument passed
`f3`	3rd floating point argument passed
`f4`	4th floating point argument passed
`f5`	5th floating point argument passed
`f6`	6th floating point argument passed
`f7`	7th floating point argument passed
`f8`	8th floating point argument passed
`f9`	9th floating point argument passed
`f10`	10th floating point argument passed
`f11`	11th floating point argument passed

Table 6-13 Floating Point Registers (continued)

Register	Description
`f12`	12th floating point argument passed
`f13`	13th floating point argument passed

When not in use, any of registers `f1` through `f13` may be used as temporary register variables.

Functions may use the values in `f1` through `f13` without saving/restoring them for the functions' callers, a function must save them if they expect to keep their values across a call to a function.

The compiler uses the remainder of the floating point registers for floating point user register variables and compiler temporaries.

Condition Registers

Registers `cr0` through `cr7`, `ctr`, `xer`, `fpscr`, and `lr` are available for hand-written assembly language use although the value must be saved and restored.

WARNING

The compiler currently uses some of the above named registers and may, in future versions, use any or all of the above named registers.

Following are descriptions of how the compiler uses four of the registers listed above.

Table 6-14 Other Registers

Register	Description
`ctr`	Compiler temporary used for structure assignments
`cr0`	Used by the compiler for all integer and comparisons assignments
`cr1`	Used by the compiler for all floating point assignments
`lr`	Used by the compiler to store caller return value

Passing Arguments to Functions

When an argument is passed to a called function, the argument is in one of two places, in a register or in the Output Parameter Area (OPA) of the calling function.

The called function determines the location of the argument by argument type and the order specified in the argument list.

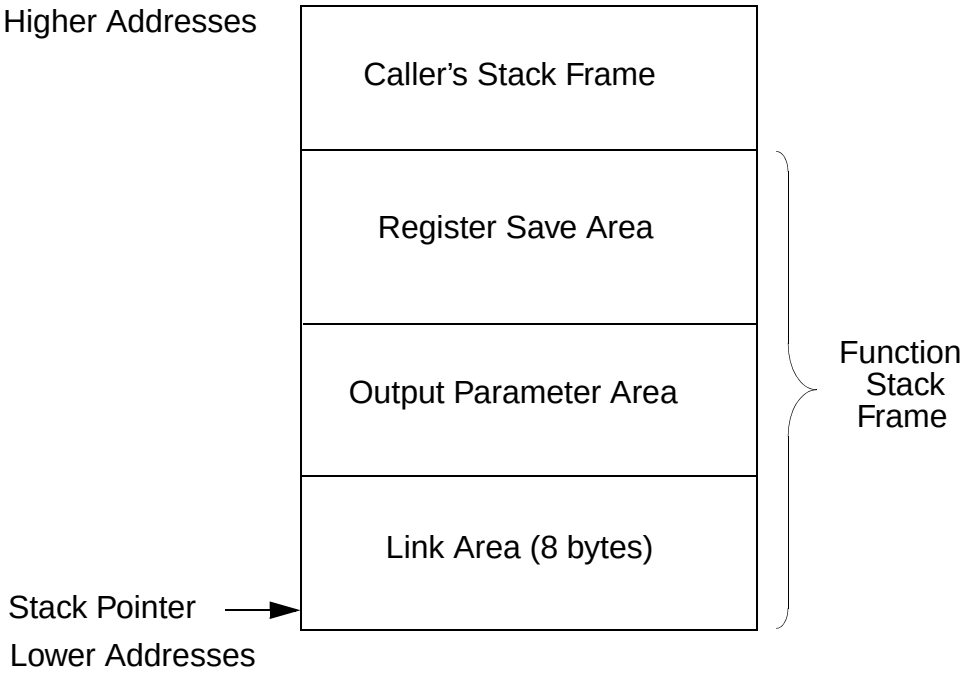
For this discussion:

- An **integral argument** is an argument of type `int`, a pointer, or a `char` or short converted to an `int`.
- A **floating point argument** is an argument of type double or a float converted to a double.

There are eight integral registers used for parameter passing: `r3` through `r10` inclusive. Thirteen floating point registers are available for floating point parameter passing: `f1` through `f13` inclusive.

The OPA is also used to pass arguments (when the registers have been exhausted). **Figure 6-1** illustrates a stack frame for a function.

Figure 6-1 Stack Frame for a Function



The basic algorithm the compiler uses to pass arguments is as follows:

```
if function returns a struct
    put address of struct return area into first integral passing register
while still more arguments
    if argument is part of variable arguments
        put argument into next position in OPA
    else if argument is a struct
        copy struct into next position in OPA
    else
        if argument is integral
            if argument is 64-bit integer type
                if pair of integral passing registers are available
                    put argument into register pair
                else
                    put argument into next two 32-bit words of OPA
            else
                if an integral passing register is available
                    put argument into integral register
                else
                    put argument into next position in OPA
```

```
        else if argument is floating-point
            if a floating-point passing register is available
                put argument into floating-point register
            else
                put argument into next position in OPA
    advance to next argument
```

The OPA is filled from lowest address to highest address.

Struct arguments and arguments that comprise the variable arguments to a variable argument function are always passed on the OPA. If a function is to return a value, an integral return value is returned in `r3` or a floating point return value is returned in `f1`. If a function is to return a struct, the address of a return area is passed as the first integral argument, in `r3`.

Callee Saved Registers

The back end for PowerPC is capable of supporting two calling conventions:

- Caller saved registers — Registers `r0` through `r10` and `f0` through `f13` are volatile and do not require saving by a function if they are modified. This is the default code generation model.
- Callee saved registers — All registers modified by a function must be saved upon entry and restored prior to returning to the caller. This model can be used by using the back end `-pc` option.

The callee saved registers convention is implemented by saving all modified registers at the beginning of the function and restoring the modified registers before returning to the caller. If function calls appear inside a function compiled with callee saved registers, it is assumed that the called functions use the callee saved registers convention also. Caller saved register functions may call callee saved register functions without adverse effects.

The compiler libraries are not compiled with callee saved register conventions. Therefore, functions compiled with callee saved registers may not call compiler library functions without the extra overhead of saving all the volatile registers before the call and restoring them after the call returns.

C Language Features

In conformance with the ANSI/ISO C specification, the implementation-defined areas of the compiler are listed in this section. Each bulleted item contains one implementation- defined issue. The number in parentheses included with each bulleted item indicates the location in the ANSI/ISO specification where further information is provided.

For More Information

Other implementation-defined areas are included in the **Language Features** chapter of the ***Using Ultra C/C++*** manual and the **Overview** chapter of the ***Ultra C Library Reference*** manual. Refer to an ANSI/ISO specification for more information.

Characters

- The number of bits in a character in the execution character set (5.2.4.2.1).

  There are eight bits in a character in the execution character set.

Integers

- The representations and sets of values of the various integer types (6.1.2.5).

Table 6-15 Integer Type/Range

Type	Representation	Minimum / Maximum
`char`, `signed char`	8-bit 2's complement	-128 / 127
`unsigned char`	8-bit binary	0 / 255
`short int`	16-bit 2's complement	-32768 / 32767
`unsigned short int`	16-bit binary	0 / 65535
`int`	32-bit 2's complement	-2147483648 / 2147483647
`unsigned int`	32-bit binary	0 / 4294967295
`long int`	32-bit 2's complement	-2147483648 / 2147483647
`unsigned long int`	32-bit binary	0 / 4294967295

Table 6-15 Integer Type/Range

Type	Representation	Minimum / Maximum
`long long`	64-bit 2's complement	-2^{63} / $2^{63}-1$
`unsigned long long`	64-bit binary	0 / $2^{64}-1$

- The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2).

  When converting a longer integer to a shorter signed integer, the least significant <n> bits of the longer integer are moved to the integer of <n> bits. The resulting value in the smaller integer is dictated by the representation. For example, if the conversion is from `int` to `short`, then the least significant 16 bits are moved from the `int` to the `short`. This value is then considered a 2's complement 16-bit integer.

  When conversion from unsigned to signed occurs with equally sized integers, the most significant bit becomes the sign bit. Therefore, if the unsigned integer is less than `0x80000000`, the conversion has no affect. Otherwise, a negative number results.

- The sign of the remainder on integer division (6.3.5).

  The result of an inexact division of one negative and one positive integer is the smallest integer greater than or equal to the algebraic quotient.

  The sign of the remainder on integer division is the same as that of the dividend.

Floating Point

- The representations and sets of values of the various types of floating-point numbers (6.1.2.5).

Table 6-16 Floating Point Number Characteristics

Type	Format	Minimum / Maximum
`float`	32 bit IEEE 754	`1.17549435e-38f / 3.40282347e38f`
`double`	64 bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`
`long double`	64 bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`

Refer to the `float.h` header file for other limits and values.

Arrays and Pointers

- The type of integer required to hold the maximum size of an array. That is, the type of the size of operator, `size_t` (6.3.3.4, 7.1.1).

  An `unsigned long int` is required to hold the maximum size of an array. `unsigned long int` is defined as `size_t` in `ansi_c.h`.

- The result of casting a pointer to an integer or vice versa (6.3.4).

  Since pointers are treated much like `unsigned long` integers, the integer is promoted using the usual promotion rules to an `unsigned long`. That is, the sign bit propagates out to the full 32-bit width.

- The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (6.3.6, 7.1.1).

  A `signed long int` is required to hold the difference between two pointers to elements of the same array. `long int` is defined as `ptrdiff_t` in `ansi_c.h`.

Registers

- The extent to which objects can actually be placed in registers by use of the register storage-class specifier (6.5.1).

  The compiler automatically makes decisions about what objects are placed in registers, thus giving no special storage considerations for the register storage-class.

Structures, Unions, Enumerations, and Bit-Fields

- The padding and alignment of members of structures (6.5.2.1). This should present no problem unless binary data written by one implementation are read by another.

  Table 6-17 shows the alignment of the various objects within a structure. Required padding is supplied if the next available space is not at the correct alignment for the object. For example, a structure declared as:

```
struct {
     char mem1;
     long mem2;
  };
```

  would be an 8-byte structure, one byte for `mem1`, three bytes of padding to get `mem2` to 4-byte alignment, and four bytes for `mem2`.

Table 6-17 Alignment Table

Type	Alignment Requirement
`char`	1
`short`	2
`int`	4
`long`	4
`long long`	4

Table 6-17 Alignment Table (continued)

Type	Alignment Requirement
`pointers`	4
`float`	4
`double`	8
`long double`	8

- Whether “plain” `int` bit-field is treated as a signed `int` or as an unsigned `int` bit-field (6.5.2.1).

  A “plain” `int` bit-field is treated as a signed `int` bit-field.

- The order of allocation of bitfields within a unit (6.5.2.1).

  Bit fields are allocated from most significant bit to least significant bit.

- Whether a bit-field can straddle a storage-unit boundary (6.5.2.1).

  Bit fields are allocated end-to-end until a non-bit field member is allocated or until that positioning would cross an addressable boundary such that no object of an integral type could both contain the bit field and be correctly aligned.

- The integer type chosen to represent the values of an enumeration type (6.5.2.2).

  `Enum` values are represented in 32-bit two's complement integers.

Preprocessing Directives

- Whether the value of a single-character, character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (6.8.1).

  The value of a single-character character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. This character constant may have a negative value.

- The method for locating includable source files (6.8.2).

  This method is described in the **Using the Executive** chapter of the ***Using Ultra C/C++*** manual.

- The support of quoted names for includable source files (6.8.2).

  Quoted names are supported for `#include` preprocessing directives.

- The mapping of source file character sequences (6.8.2).

  The mapping of source file character sequences is one-to-one. The case-sensitivity of file names depends on the file system being used.

_asm() Register Pseudo Functions

`_asm()` uses registers pseudo functions as identified in **Table 6-18**.

Table 6-18 _asm() Register Pseudo Functions

Register	Description
`__reg_gen`	Any non-dedicated integer register
`__reg_base`	Any (non-dedicated) integer register except `r0`
`__reg_float`	Any floating point register
`__reg_r<n>`	The integer register specified by `n (0 <= n < 32)`
`__reg_f<n>`	The floating point register specified by `n (0 <= n < 32)`

PowerPC Processor-Specific Optimizations

In addition to providing the standard generic assembly optimizations, the Ultra C/C++ PowerPC assembly optimizer (`optppc`) provides the following processor-specific optimizations:

- **Special Common Sub-Expressions**
- **Copy Propagation**
- **Target-Driven Instruction Scheduling**
- **Register Renaming**

Special Common Sub-Expressions

On the PowerPC architecture, certain constants are more expensive to work with than others. To account for this, the common sub-expression elimination optimization is performed on expressions involving potentially expensive constants. These include the following:

- The computation of all floating point constants as none of the floating point instructions allow for floating point immediates.
- The computation of integer constants that require two or more instructions. For example the constant 0xffeeffee. The constant 0xffffffee would not be considered.

Copy Propagation

The assembly code optimizer tries to eliminate needless copies between register temporaries, resulting in smaller, more efficient code. For example:

```
lwz r4,=x(r2)
mr r5,r4
addi r4,r5,1
```

This may be changed to the following:

```
lwz r5,=x(r2)
addi r4,r5,1
```

Target-Driven Instruction Scheduling

Using target-driven instruction scheduling, the assembly optimizer can factor in what it knows about the target architecture when scheduling instructions. Following are variables that can be taken into account:

- Number and types of instruction units
- Number of instructions that can be executed at a time
- Time required for an instruction to produce its results
- Time it takes an instruction to keep an instruction unit from executing other instructions

Another important variable is whether a target processor has hardware floating point or not. If it does, then floating point instructions should be scheduled just like any other instructions; mixed in with other instructions to improve throughput. However, if the target processor does not have hardware floating point support and the instructions must be emulated, it is best to compress the floating point instructions. When compressed, the instructions can be emulated much faster.

Register Renaming

It is possible to use the same temporary register in sequential instructions. However, this can confuse the instruction scheduler. The assembly optimizer uses different registers when possible to help itself perform schedule instructions. For example:

```
* compute sum = a + b + c + d
  lwz r3,a(r2)
  lwz r4,b(r2)
  add r3,r3,r4
  lwz r4,c(r2)
  add r3,r3,r4
  lwz r4,d(r2)
  add r3,r3,r4
  stw r3,sum(r2)
```

This may be changed to the following:

```
  lwz r3,a(r2)
  lwz r5,b(r2)
  lwz r1,c(r2)
  add r3,r3,r5
  lwz r4,d(r2)
  add r3,r3,r1
  add r3,r3,r4
  stw r3,sum(r2)
```

The use of different registers allowed the spreading of register loads from their subsequent uses.

Assembler/ Linker

The assembler allows use of standard PowerPC assembly language mnemonics and syntax, modified as described in this section. For more specific information about individual instructions, consult the following books:

PowerPC Microprocessor Family: The Programming Environments

IBM PowerPC 403GA User's Manual

Technical Summary — PowerPC MPC505 RISC Microcontroller Motorola Semiconductor Technical Data

PowerPC 601 RISC Microprocessor User's Manual Motorola/IBM, Revision 1

PowerPC 602 RISC Microprocessor User's Manual

PowerPC 603 Microprocessor User's Manual

PowerPC 604 — RISC Microprocessor User's Manual

MPC821 — Functional Design Specification Rev. 0.4

ROF Edition Number

The PowerPC assembler emits ROF Edition #15.

External References

The PowerPC assembler allows the use of external references with any operators within any expression fields not defined to be constant expression fields.

Symbol Biasing

The linker does not bias code or data symbols for system, file managers, device drivers, device descriptors, or data modules. For all other types of modules, or for raw code, the linker biases both code and data symbols by `-32764 (-0x7ffc)`. Initialization routines for raw code should ensure that the static storage pointer (`r2`) and constant storage pointer (`r13`) are initialized with the proper base addresses, adjusted to account for this biasing.

Assembler Syntax Extensions and Limitations

The PowerPC instruction set limits the size of immediate data to 16 bits. To allow for the manipulation of 32-bit immediate data, the following operators are available in the PowerPC assembler:

```
high(x)= ((x >> 16) & 0xffff) + ((x & 0x8000) >> 15)
hi(x)= x >> 16
lo(x)= x & 0xffff
```

The following code moves the 32-bit value represented by `Symbol` into GPR `r4`.

```
addisr4,r0,hi(Symbol)
ori  r4,r4,lo(Symbol)
```

If `Symbol` represents the offset of `X` from GPR `r2`, then the following code moves 32-bit address of `X` into `r4`.

```
addisr4,r2,high(Symbol)
addi r4,r4,lo(Symbol)
```

The following code uses the `high()` operand to load the word `X` into `r4`. These operators may be used to manipulate external symbols.

```
addisr4,r0,high(Symbol)
lwz  r4,lo(Symbol)(r4)
```

For load and store instructions using the register indirect with immediate index addressing mode, it is not necessary to explicitly indicate an index of `0`. For example, the following are equivalent:

```
lwz  r4,0(r1)
lwz  r4,(r1)
```

The special symbols `cr0-cr7`, `lt`, `gt`, `eq`, `so`, and `un` used in many examples in the ***PowerPC 601 Microprocessor User's Manual*** are not directly supported by the PowerPC assembler. For example, the following examples generate errors:

```
bdnzt eq,<target>
bdnze 4*cr5+eq,<target>
crnot eq,cr5*4+eq
crclr so
```

These instruction forms may be successfully used and translated if the following definitions are in the source file or in an external file included using the use assembler directive.

```
cr0: equ0         * condition register field
             definitions
cr1: equ1
...
cr7: equ7

lt:  equ0         * condition code definitions
gt:  equ1
eq:  equ2
so:  equ3
un:  equ3
```

GPRs, FPRs, and CRFs must be referenced by name. The use of numbers to denote register objects is not supported except where specifically noted.

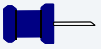
When the GPR `r0` is used in a read operation in certain instructions on the PowerPC family of processors, the value used by the instruction is `0` rather than the content of `r0`. For these instructions, the assembler allows the use of the numeral `0` in place of the register name `r0`. For example, in each of the following instruction pairs, the instructions are encoded identically.

```
addisr5,r0,312
addisr5,0,312
subi r0,r0,$7fff
subi r0,0,$7fff
la   r10,$56(r0)
la   r10,$56(0)
lbzx r8,r0,r12
lbzx r8,0,r12
```

Special Purpose Registers

The PowerPC assembler accepts either the appropriate register number or name as identified in **Table 6-19** for SPRs used in the `mtspr` and `mfspr` instructions. For example, the following two instructions are equivalent:

```
mfsprr4,ctr
mfsprr4,9
```



Note

The SPRs shown in the following table may not be valid for a specific processor. Consult your hardware documentation for valid SPR numbers.

Table 6-19 SPRs by Name

SPR Name	SPR Number
asr	280
bar	159

Table 6-19 SPRs by Name

SPR Name	SPR Number
cdbcr	983
cmpa	144
cmpb	145
cmpc	146
cmpd	147
cmpe	152
cmpf	153
cmpg	154
cmph	155
counta	150
countb	151
ctr	9
dabr	1013
dac1	1014
dac2	1015
dar	19
dbat0l	537

Table 6-19 SPRs by Name

SPR Name	SPR Number
dbat0u	536
dbat1l	539
dbat1u	538
dbat2l	541
dbat2u	540
dbat3l	543
dbat3u	542
dbcr	1010
dbsr	1008
dc_adr	569
dc_cst	568
dc_dat(ro)	570
dccr	1018
dcmp	977
dcwr	954
dear(ro)	981
dec	22

Table 6-19 SPRs by Name

SPR Name	SPR Number
der	149
dmiss	976
dpdr	630
dsisr	18
ear	282
ecr(ro)	148
eid(wo)	81
eie(wo)	80
esasrr	987
esr	980
evpr	982
fpecr	1022
hash1	978
hash2	979
hid0	1008
hid1	1009
hid15	1023

Table 6-19 SPRs by Name

SPR Name	SPR Number
hid2	1010
hid5	1013
iabr	1010
iac1	1012
iac2	1013
ibat0l	529
ibat0u	528
ibat1l	531
ibat1u	530
ibat2l	533
ibat2u	532
ibat3l	535
ibat3u	534
ibr	986
ic_adr	561
ic_cst	560
ic_dat(ro)	562

Table 6-19 SPRs by Name

SPR Name	SPR Number
icadr	561
iccr	1019
iccst	560
icdat(ro)	562
icdbdr	979
icmp	981
icr(ro)	148
ictc	1019
ictrl	158
imiss	980
immr	638
l2cr	1017
lctrl1	156
lctrl2	157
lr	8
lt	1022
m_casid	793

Table 6-19 SPRs by Name

SPR Name	SPR Number
m_tw	799
m_twb	796
md_ap	794
md_ctr	792
md_dbcam	824
md_dbram0	825
md_dbram1	826
md_epn	795
md_rpn	798
md_twc	797
mi_ap	786
mi_ctr	784
mi_dbcam	816
mi_dbram0	817
mi_dbram1	818
mi_epn	787
mi_rpn	790

Table 6-19 SPRs by Name

SPR Name	SPR Number
mi_twc	789
mmcr0	952
mmcr1	956
mq	0
nre(wo)	82
nri(wo)	82
pbl1	1020
pbl2	1022
pbu1	1021
pbu2	1023
pid	945
pir	1023
pit	987
pmc1	953
pmc2	954
pmc3	957
pmc4	958

Table 6-19 SPRs by Name

SPR Name	SPR Number
pvr(ro)	287
rpa	982
rtcl(ro)	5
rtcl(wo)	21
rtcu(ro)	4
rtcu(wo)	20
sda	959
sdr1	25
sebr	990
ser	991
sgr	953
sia	955
sp	1021
sprg0	272
sprg1	273
sprg2	274
sprg3	275

Table 6-19 SPRs by Name

SPR Name	SPR Number
srr0	26
srr1	27
srr2	990
srr3	991
tbhi	988
tbhu(ro)	972
tbl(wo)	284
tblo	989
tblu(ro)	973
tbu(wo)	285
tcr	986
thrm1	1020
thrm2	1021
thrm3	1022
tsr	984
tsrs	985
ummcr0	936

Table 6-19 SPRs by Name

SPR Name	SPR Number
ummcr1	940
upmc1	937
upmc2	938
upmc3	941
upmc4	942
usia	939
xer	1
zpr	944

Table 6-20 shows the SPRs by number. Some SPRs may not be valid for a specific processor. Consult your hardware documentation for valid SPR numbers.

Table 6-20 SPRs by Number

SPR Number	SPR Name
0	mq
1	xer
4	rtcu(ro)
5	rtcl(ro)
8	lr

Table 6-20 SPRs by Number

SPR Number	SPR Name
9	ctr
18	dsisr
19	dar
20	rtcu(wo)
21	rtcl(wo)
22	dec
25	sdr1
26	srr0
27	srr1
80	eie(wo)
81	eid(wo)
82	nre(wo)
82	nri(wo)
144	cmpa
145	cmpb
146	cmpc
147	cmpd

Table 6-20 SPRs by Number

SPR Number	SPR Name
148	icr(ro)
148	ecr(ro)
149	der
150	counta
151	countb
152	cmpe
153	cmpf
154	cmpg
155	cmph
156	lctrl1
157	lctrl2
158	ictrl
159	bar
272	sprg0
273	sprg1
274	sprg2
275	sprg3

Table 6-20 SPRs by Number

SPR Number	SPR Name
280	asr
282	ear
284	tbl(wo)
285	tbu(wo)
287	pvr(ro)
528	ibat0u
529	ibat0l
530	ibat1u
531	ibat1l
532	ibat2u
533	ibat2l
534	ibat3u
535	ibat3l
536	dbat0u
537	dbat0l
538	dbat1u
539	dbat1l

Table 6-20 SPRs by Number

SPR Number	SPR Name
540	dbat2u
541	dbat2l
542	dbat3u
543	dbat3l
560	ic_cst
560	iccst
561	icadr
561	ic_adr
562	ic_dat(ro)
562	icdat(ro)
568	dc_cst
569	dc_adr
570	dc_dat(ro)
630	dpdr
638	immr
784	mi_ctr
786	mi_ap

Table 6-20 SPRs by Number

SPR Number	SPR Name
787	mi_epn
789	mi_twc
790	mi_rpn
792	md_ctr
793	m_casid
794	md_ap
795	md_epn
796	m_twb
797	md_twc
798	md_rpn
799	m_tw
816	mi_dbcam
817	mi_dbram0
818	mi_dbram1
824	md_dbcam
825	md_dbram0
826	md_dbram1

Table 6-20 SPRs by Number

SPR Number	SPR Name
936	ummcr0
937	upmc1
938	upmc2
939	usia
940	ummcr1
941	upmc3
942	upmc4
944	zpr
945	pid
952	mmcr0
953	pmc1
953	sgr
954	pmc2
954	dcwr
955	sia
956	mmcr1
957	pmc3

Table 6-20 SPRs by Number

SPR Number	SPR Name
958	pmc4
959	sda
972	tbhu(ro)
973	tblu(ro)
976	dmiss
977	dcmp
978	hash1
979	icdbdr
979	hash2
980	esr
980	imiss
981	icmp
981	dear(ro)
982	rpa
982	evpr
983	cdbcr
984	tsr

Table 6-20 SPRs by Number

SPR Number	SPR Name
985	tsrs
986	tcr
986	ibr
987	esasrr
987	pit
988	tbhi
989	tblo
990	srr2
990	sebr
991	ser
991	srr3
1008	dbsr
1008	hid0
1009	hid1
1010	dbcr
1010	iabr
1010	hid2

Table 6-20 SPRs by Number

SPR Number	SPR Name
1012	iac1
1013	dabr
1013	iac2
1013	hid5
1014	dac1
1015	dac2
1017	l2cr
1018	dccr
1019	ictc
1019	iccr
1020	pbl1
1020	thrm1
1021	thrm2
1021	pbu1
1021	sp
1022	pbl2
1022	thrm3

Table 6-20 SPRs by Number

SPR Number	SPR Name
1022	lt
1022	fpecr
1023	pbu2
1023	pir
1023	hid15

Time Based Registers

Table 6-21 Time Base Register (TBR) Support

SPR Name	505 SPR #	603 SPR #
tbu	269	269
tbl	268	268

Device Control Registers

Table 6-22 403 DCRs Sorted Alphabetically

DCR Name	DCR #
bear	144
besr	145
br0	128
br1	129
br2	130
br3	131
br4	132
br5	133
br6	134
br7	135
dmacc0	196
dmacc1	204
dmacc2	212
dmacc3	220
dmacr0	192

Table 6-22 403 DCRs Sorted Alphabetically (continued)

DCR Name	DCR #
dmacr1	200
dmacr2	208
dmacr3	216
dmact0	193
dmact1	201
dmact2	209
dmact3	217
dmada0	194
dmada1	202
dmada2	210
dmada3	218
dmasa0	195
dmasa1	203
dmasa2	211
dmasa3	219
dmasr	224
exisr	64

Table 6-22 403 DCRs Sorted Alphabetically (continued)

DCR Name	DCR #
exier	66
iocr	160

Table 6-23 403 DCRs Sorted Numerically

DCR #	DCR Name
64	exisr
66	exier
128	br0
129	br1
130	br2
131	br3
132	br4
133	br5
134	br6
135	br7
144	bear
145	besr

Table 6-23 403 DCRs Sorted Numerically (continued)

DCR #	DCR Name
160	iocr
192	dmacr0
193	dmact0
194	dmada0
195	dmasa0
196	dmacc0
200	dmacr1
201	dmact1
202	dmada1
204	dmacc1
196	dmacc0
208	dmacr2
209	dmact2
210	dmada2
211	dmasa2
212	dmacc2
216	dmacr3

Table 6-23 403 DCRs Sorted Numerically (continued)

DCR #	DCR Name
217	dmact3
218	dmada3
219	dmasa3
220	dmacc3
224	dmasr

Assembly Language Mnemonics

Table 6-28 in this section lists the mnemonic names used on the PowerPC along with their meanings. Many of the PowerPC mnemonics include one or more optional suffixes. Symbols indicating conditions and SPR codes may also be present in a mnemonic. Suffixes and symbols, when present, modify the meaning of the mnemonic instructions. The next two subsections identify and define suffixes and symbols used in the syntax of **Table 6-28**.

Suffixes

`.`	Update the condition register to reflect the instruction result. `cr1` is updated in the case of floating point instructions, otherwise, `cr0` is updated.
o	Enable setting of `OV` and `SO` in the Fixed Point Exception Register (`xer`).
s	For floating point instructions, execute instruction using single precision.
`l`	For branch instructions, cause the effective address of the next instruction to be placed in the Link Register (`lr`).
a	For branch instructions, cause target address to be interpreted as absolute rather than PC-relative.
+	For branch instructions, indicates branch is predicted to be taken.
-	For branch instructions, indicates branch is predicted not to be taken.

In the Mnemonics table, optional suffixes are denoted using square brackets ([]). For example, the mnemonic add is listed as:

```
add[o][.]
```

indicating the four valid mnemonics:

```
add
add.
addo
addo.
```

Since the branch mnemonic suffixes + and - are mutually exclusive, these suffixes appear as [+|-] indicating the use of no greater than one of these characters. For example,

```
bclr[+|-]
```

indicates the three valid mnemonics:

```
bclr
bclr+
bclr-
```

Symbols

The symbols <bc>, <cc>, and <tc> indicate condition codes, branch conditions, and trap conditions respectively as identified in the following listings. The symbol <spr> indicates any one of the special purpose registers identified in the <spr> listing.

Table 6-24 Branch Conditions

<bc>	Description
t	Branch if condition true
f	Branch if condition false
dnz	Decrement CTR, branch if non-zero

Table 6-24 Branch Conditions (continued)

<bc>	Description
dnzt	Decrement CTR, branch if non-zero and condition true
dnzf	Decrement CTR, branch if non-zero and condition false
dz	Decrement CTR, branch if zero
dzt	Decrement CTR, branch if zero and condition true
dzf	Decrement CTR, branch if zero and condition false

Table 6-25 Condition Codes

<cc>	Description
lt	Less than
le	Less than or equal (not greater than)
eq	Equal
ge	Greater than or equal (not less than)
gt	Greater than
nl	Not less than
ne	Not equal
ng	Not greater than
so	Summary overflow
ns	Not summary overflow

Table 6-25 Condition Codes (continued)

<cc>	Description
un	Unordered (after floating-point compare)
nu	Not unordered (after floating-point compare)

Table 6-26 Trap Condition Codes

<tc>	Description
lt	Less than
le	Less than or equal
eq	Equal
ge	Greater than or equal
gt	Greater than
nl	Not less than
ne	Not equal
ng	Not greater than
llt	Logically less than
lle	Logically less than or equal
lge	Logically greater than or equal
lgt	Logically greater than

Table 6-26 Trap Condition Codes (continued)

<tc>	Description
`lnl`	Logically not less than
`lng`	Logically not greater than

Table 6-27 Special Purpose Registers

<spr>	Description
`xer`	Fixed point exception register
`lr`	Link register
`ctr`	Count register
`dsisr`	Data storage interrupt status register
`dar`	Data address register
`dec`	Decrementer
`sdr1`	Storage description register 1
`srr0`	Save/restore register 0
`srr1`	Save/restore register 1
`ear`	External access register

Mnemonics Table

The mnemonics listed in **Table 6-28** are either native to the PowerPC architecture or extended mnemonics accepted by the PowerPC assembler as aliases for native instructions, providing a simpler syntax for the programmer. All extended mnemonics are identified by an asterisk in the column labeled **E** in the listing.

Table 6-28 Mnemonics

Mnemonic	Description	E
`add[o][.]`	Add	
`addc[o][.]`	Add carrying	
`adde[o][.]`	Add extended	
`addi`	Add immediate	
`addic[.]`	Add immediate carrying	
`addis`	Add immediate shifted	
`addme[o][.]`	Add to minus one extended	
`addze[o][.]`	Add to zero extended	
`and[.]`	And logical	
`andc[.]`	And logical with complement	
`andi.`	And logical immediate (alters `cr0`)	
`andis.`	And logical immediate shifted (alters `cr0`)	

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
b[l][a]	Branch unconditional	
bc[l][a][+|-]	Branch conditionally	
b<bc>[l][a][+|-]	Branch conditionally (implicit)	*
b<cc>[l][a][+|-]	Branch if <cc>	*
bcctr[l][+|-]	Branch conditionally to count register	
b<cc>ctr[l][+|-]	Branch if <cc> to count register	*
bclr[l][+|-]	Branch conditionally to link register	
b<bc>lr[l][+|-]	Branch conditionally to link register (implicit)	*
b<cc>lr[l][+|-]	Branch if <cc> to link register	*
bctr[l]	Branch unconditionally to count register	*
bfctr[l][+|-]	Branch if condition false to count register	*
blr[l]	Branch unconditionally to link register	*
btctr[l][+|-]	Branch if condition true to count register	*
clrlwi[.]	Clear left immediate	*
clrrwi[.]	Clear right immediate	*
clrlslwi[.]	Clear left and shift left immediate	*

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
cmp	Compare	
cmpi	Compare immediate	
cmpl	Compare logical	
cmpli	Compare logical immediate	
cmplw	Compare logical word	*
cmplwi	Compare logical word immediate	*
cmpw	Compare word	*
cmpwi	Compare word immediate	*
cntlzw[.]	Count leading zeros in word	
crand	Condition register and	
crandc	Condition register and with complement	
crclr	Condition register clear	*
creqv	Condition register equivalent	
crmove	Condition register move	*
crnand	Condition register nand	
crnor	Condition register nor	
crnot	Condition register not	*

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
cror	Condition register or	
crorc	Condition register or with complement	
crset	Condition register set	*
crxor	Condition register xor	
dcbf	Data cache block flush	
dcbi	Data cache block invalidate	
dcbst	Data cache block store	
dcbt	Data cache block touch	
dcbtst	Data cache block touch for store	
dcbz	Data cache block set to zero	
divw[o][.]	Divide word	
divwu[o][.]	Divide word unsigned	
eciwx	External control input word indexed	
ecowx	External control output word indexed	
eieio	Enforce in-order execution of I/O	
eqv[.]	Equivalent logical	
extlwi[.]	Extract and left justify immediate	*

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
`extrwi[.]`	Extract and right justify immediate	*
`extsb[.]`	Extend sign byte	
`extsh[.]`	Extend sign half word	
`fabs[.]`	Floating point absolute value	
`fadd[s][.]`	Floating point add	
`fcmpo`	Floating point compare ordered	
`fcmpu`	Floating point compare unordered	
`fctiw[.]`	Floating point convert to integer word	
`fctiwz[.]`	Floating point convert to integer word round toward zero	
`fdiv[s][.]`	Floating point divide	
`fmadd[s][.]`	Floating point multiply and add	
`fmr[.]`	Floating point move register	
`fmsub[s][.]`	Floating point multiply and subtract	
`fmul[s][.]`	Floating point multiply	
`fnabs[.]`	Floating point negative absolute value	
`fneg[.]`	Floating point negate	
`fnmadd[s][.]`	Floating point negative multiply and add	

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
`fnmsub[s][.]`	Floating point negative multiply and subtract	
`frsp[.]`	Floating point round to single precision	
`fsub[s][.]`	Floating point subtract	
`icbi`	Instruction cache block invalidate	
`inslwi[.]`	Insert from left immediate	*
`insrwi[.]`	Insert from right immediate	*
`isync`	Instruction synchronize	
`la`	Load address	*
`lbz`	Load byte and zero	
`lbzu`	Load byte and zero with update	
`lbzux`	Load byte and zero with update indexed	
`lbzx`	Load byte and zero indexed	
`lfd`	Load floating-point double	
`lfdu`	Load floating-point double with update	
`lfdux`	Load floating-point double with update indexed	
`lfdx`	Load floating-point double indexed	

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
lfs	Load floating-point single	
lfsu	Load floating-point single with update	
lfsux	Load floating-point single with update indexed	
lfsx	Load floating-point single indexed	
lha	Load half word algebraic	
lhau	Load half word algebraic with update	
lhaux	Load half word algebraic with update indexed	
lhax	Load half word algebraic indexed	
lhbrx	Load half word byte-reversed indexed	
lhz	Load half word and zero	
lhzu	Load half word and zero with update	
lhzux	Load half word and zero with update indexed	
lhzx	Load half word and zero indexed	
li	Load immediate	*
lis	Load immediate shifted	*
lmw	Load multiple word	

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
`lswi`	Load string word immediate	
`lswx`	Load string word indexed	
`lwarx`	Load word and reverse indexed	
`lwbrx`	Load word byte-reverse indexed	
`lwz`	Load word and zero	
`lwzu`	Load word and zero with update	
`lwzux`	Load word and zero with update indexed	
`lwzx`	Load word and zero indexed	
`mcrf`	Move condition register field	
`mcrfs`	Move to condition register from `fpscr`	
`mcrxr`	Move to condition register from *xer*	
`mfcr`	Move from condition register	
`mffs[.]`	Move from `fpscr`	
`mfibatl`	Move from one of `ibat0 - ibat3` lower	*
`mfibatu`	Move from one of `ibat0 - ibat3` upper	*
`mfmsr`	Move from machine state register	

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
mfpvr	Move from pvr	*
mfspr	Move from SPR	
mfsr	Move from SPR	
mf<spr>	Move from the SPR indicated by <spr>	*
mtsprg	Move from one of sprg0 - sprg3	*
mfsrin	Move from segment register indirect	
mftb	Move from time base register	
mftbl	Move from time base register lower	
mftbu	Move from time base register upper	
mr[.]	Move general purpose register	*
mtcrf	Move to condition register fields	
mtfsb0[.]	Move to fpscr bit 0	
mtfsb1[.]	Move to fpscr bit 1	
mtfsf[.]	Move to fpscr fields	
mtfsfi[.]	Move to fpscr field immediate	
mtibatl	Move to one of ibat0 - ibat3 upper	*
mtibatu	Move to one of ibat0 - ibat3 upper	*

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
mtmsr	Move to machine state register	
mtspr	Move to SPR	
mt<spr>	Move to the SPR indicated by <spr>	*
mtsprg	Move to one of sprg0 - sprg3	*
mtsr	Move to segment register	
mtsrin	Move to segment register indirect	
mulhw[.]	Multiply high word	
mulhwu[.]	Multiply high word unsigned	
mulli	Multiply low word immediate	
mullw[o][.]	Multiply low word	
nand[.]	Not and logical	
neg[o][.]	Negate	
nop	No operation	*
nor[.]	Not or logical	
not[.]	Not logical	*
or[.]	Or logical	
orc[.]	Or logical with complement	

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
ori	Or logical immediate	
oris	Or logical immediate shifted	
rfi	Return from interrupt	
rlwimi[.]	Rotate left word immediate then mask insert	
rlwinm[.]	Rotate left word immediate then and with mask	
rlwnm[.]	Rotate left word then and with mask	
rotlwi[.]	Rotate left word immediate	*
rotrwi[.]	Rotate right word immediate	*
rotlw[.]	Rotate left word	*
sc	System call	
slw[.]	Shift left word	
slwi[.]	Shift left word immediate	*
sraw[.]	Shift right algebraic word	
srawi[.]	Shift right algebraic word immediate	
srw[.]	Shift right word	
srwi[.]	Shift right word immediate	*

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
stb	Store byte	
stbu	Store byte with update	
stbux	Store byte with update indexed	
stbx	Store byte indexed	
stfd	Store floating-point double	
stfdu	Store floating-point double with update	
stfdux	Store floating-point double with update indexed	
stfdx	Store floating-point double indexed	
stfs	Store floating-point single	
stfsu	Store floating-point single with update	
stfsux	Store floating-point single with update indexed	
stfsx	Store floating-point single indexed	
sth	Store half word	
sthbrx	Store half word byte-reversed indexed	
sthu	Store half word with update	
sthux	Store half word with update indexed	

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
`sthx`	Store half word indexed	
`stmw`	Store multiple word	
`stswi`	Store string word immediate	
`stswx`	Store string word indexed	
`stw`	Store word	
`stwbrx`	Store word byte-reversed indexed	
`stwcx.`	Store word conditional indexed (alters `cr0`)	
`stwu`	Store word with update	
`stwux`	Store word with update indexed	
`stwx`	Store word indexed	
`sub[o][.]`	Subtract	*
`subc[o][.]`	Subtract carrying	*
`subf[o][.]`	Subtract from	
`subfc[o][.]`	Subtract from carrying	
`subfe[o][.]`	Subtract from extended	
`subfic`	Subtract from immediate carrying	
`subfme[o][.]`	Subtract from minus one extended	

Table 6-28 Mnemonics (continued)

Mnemonic	Description	E
subfze[o][.]	Subtract from zero extended	
subi	Subtract immediate	*
subic[.]	Subtract immediate carrying	*
subis	Subtract immediate shifted	*
sync	Synchronize	
tlbie	Translation lookaside buffer invalidate entry	
trap	Trap unconditionally	*
tw	Trap word	
tw<tc>	Trap word if <tc>	*
twi	Trap word immediate	
tw<tc>i	Trap word if <tc> immediate	*
xor[.]	Exclusive or logical	
xori	Exclusive or logical immediate	
xoris	Exclusive or logical immediate shifted	

Extended Mnemonics

This section provides the equivalent forms for each of the extended mnemonics listed in **Table 6-28**. The subsections following are organized according to mnemonic function.

Subtract Immediate

Table 6-29 Subtract Immediate

Extended Mnemonic	Equivalent To
`subi rX,rY,value`	`addi rX,rY,-value`
`subis rX,rY,value`	`addis rX,rY,-value`
`subic[.]rX,rY,value`	`addic[.]rX,rY,-value`
`subic. rX,rY,value`	`addic. rX,rY,-value`

Subtract

Table 6-30 Subtract

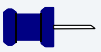
Extended Mnemonics	Equivalent To
`sub[o][.] rX,rY,rZ`	`subf[o][.] rX,rZ,rY`
`subc[o][.] rX,rY,rZ`	`subfc[o][.] rX,rZ,rY`

Word Compare

Table 6-31 Word Compare

Extended Mnemonics	Equivalent To
`cmpwi crfD,rA,si`	`cmpi crfD,0,rA,si`
`cmpw crfD,rA,rB`	`cmp crfD,0,rA,rB`
`cmplwi crfD,rA,ui`	`cmpli crfD,0,rA,ui`
`cmplw crfD,rA,rB`	`cmpl crfD,0,rA,rB`

Extract, Insert, Rotate, Shift, and Clear



Note

All expressions used above are in unsigned `mod 32` arithmetic.

Table 6-32 Extract, Insert, Rotate, Shift, and Clear

Extended Mnemonics	Equivalent To
`extlwi[.] rA,rS,n,b`	`rlwinm[.] rA,rS,b,0,n-1`
`extrwi[.] rA,rS,n,b`	`rlwinm[.] rA,rS,b+n,32-n,31`

Table 6-32 Extract, Insert, Rotate, Shift, and Clear (continued)

Extended Mnemonics	Equivalent To
`inslwi[.] rA,rS,n,b`	`rlwimi[.] rA,rS,32-b,b,(b+n)-1`
`insrwi[.] rA,rS,n,b`	`rlwimi[.] rA,rS,32-(b+n),b,(b+n)-1`
`rotlwi[.] rA,rS,n`	`rlwinm[.] rA,rS,n,0,31`
`rotrwi[.] rA,rS,n`	`rlwinm[.] rA,rS,32-n,0,31`
`rotlw[.] rA,rS,rB`	`rlwnm[.] rA,rS,rB,0,31`
`slwi[.] rA,rS,n`	`rlwinm[.] rA,rS,n,0,31-n`
`srwi[.] rA,rS,n`	`rlwinm[.] rA,rS,32-n,n,31`
`clrlwi[.] rA,rS,n`	`rlwinm[.] rA,rS,0,n,31`
`clrrwi[.] rA,rS,n`	`rlwinm[.] rA,rS,0,0,31-n`
`clrlslwi[.] rA,rS,b,n`	`rlwinm[.] rA,rS,n,b-n,31-n`

Move to/from Special Purpose Registers

Table 6-33 Move to/from Special Purpose Registers

Extended Mnemonics	Equivalent To
mfpvr rX	mfspr rX,287
mfsprg rX,<n>	mfspr rX,272+<n>
mfibatu rX,<n>	mfspr rX,528+2<n>
mfibatl rX,<n>	mfspr rX,529+2<n>
mt<spr> rX	mtspr <num>,rX
mtsprg <n>,rX	mtspr 272+<n>,rX
mtibatu <n>,rX	mtspr 528+2<n>,rX
mtibatl <n>,rX	mtspr 529+2<n>,rX

Where <n> is one of 0, 1, 2, or 3 and the relation between <spr> and <num> is given by the following table.

Table 6-34 <spr> to <num> Relationship

<spr>	<num>
xer	1
lr	8
ctr	9

Table 6-34 <spr> to <num> Relationship

<spr>	<num>
dsisr	18
dar	19
dec	22
sdr1	25
srr0	26
srr1	27
ear	282

Move to/from Time Base Registers

Table 6-35 Move to/from Time Base Registers

Extended Mnemonics	Equivalent To
mftb rX	mftb rX,268
mftbl rX	mftb rX,268
mftbu rX	mftb rX,269
mttbl rX	mtspr 284,rX
mttbu rX	mtspr 285,rX

Conditional Branch

Table 6-36 Conditional Branch

Extended Mnemonics	Equivalent To
b<bc>[l][a] bi,target16	bc[l][a] <num>,bi,target16
bctr[l]	bcctr[l] 20,0
bfctr[l] bi	bcctr[l] 4,bi
btctr[l] bi	bcctr[l] 12,bi
blr[l]	bclr[l] 20,0
b<bc>lr[l] bi	bclr[l] <num>,bi

Where <bc> and <num> are related in the following table.

Table 6-37 <bc> to <num> Relationship

<bc>	<num>
dnzf	0
dzf	2
f	4
dnzt	8
dzt	10

Table 6-37 <bc> to <num> Relationship (continued)

<bc>	<num>
t	12
dnz	16
dz	18

Branch Mnemonics Incorporating Conditions

Table 6-38 Branch Mnemonics Incorporating Conditions

Extended Mnemonics	Equivalent To
b<cc>[l][a] crf,target16	bc[l][a] <t_f>,crf+<cond>,target16
b<cc>[l][a] target16	bc[l][a] <t_f>,0+<cond>,target16
b<cc>ctr[l] crf	bcctr[l] <t_f>,crf+<cond>
b<cc>ctr[l]	bcctr[l] <t_f>,0+<cond>
b<cc>lr[l] crf	bclr[l] <t_f>,crf+<cond>
b<cc>lr[l]	bclr[l] <t_f>,0+<cond>

Where <cc>, <t_f>, and <cond> are related as shown in **Table 6-39**.

Table 6-39 <cc>, <t_f>, and <cond> Relationships

<cc>	<t_f>	<cond>	Description
lt	12 (t)	0 (lt)	Less than
le	4 (f)	1 (gt)	Not greater than
eq	12 (t)	2 (eq)	Equal
ge	4 (f)	0 (lt)	Not less than
gt	12 (t)	1 (gt)	Greater than
nl	4 (f)	0 (lt)	Not less than
ne	4 (f)	2 (eq)	Not equal
ng	4 (f)	1 (gt)	Not greater than
so	12 (t)	3 (so)	Summary overflow
ns	4 (f)	3 (so)	Not summary overflow
un	12 (t)	3 (un)	Unordered
nu	4 (f)	3 (un)	Not unordered

Branch Prediction Suffixes

The + and – branch prediction suffixes are available for use with any conditional branch mnemonic. If the following equivalences can be inferred from the preceding branch mnemonic mappings:

Table 6-40 Branch Prediction Suffixes

Extended Mnemonic	Equivalent To
`b<xxx>[l][a] <operand>`	`bc[l][a] bo,bi,target16`
`b<xxx>ctr[l] <operand>`	`bcctr[l] bo,bi`
`b<xxx>lr[l] <operand>`	`bclr[l] bo,bi`

Then the following instructions are also equivalent:

Table 6-41 Equivalent Instructions

Extended Mnemonic	Equivalent To
`b<xxx>[l][a]+ <operand>`	`bc[l][a] bo+<x>,bi,target16`
`b<xxx>[l][a]- <operand>`	`bc[l][a] bo+<y>,bi,target16`
`b<xxx>ctr[l]+ <operand>`	`bcctr[l] bo+1,bi`
`b<xxx>ctr[l]- <operand>`	`bcctr[l] bo,bi`
`b<xxx>lr[l]+ <operand>`	`bclr[l] bo+1,bi`
`b<xxx>lr[l]- <operand>`	`bclr[l] bo,bi`

Where:

If `target16 < 0`, `<x> = 0` and `<y> = 1`

If `target16 Š 0`, `<x> = 1` and `<y> = 0`

Traps

Table 6-42 Traps

Extended Mnemonics	Equivalent To
`trap rA,rB`	`tw 31,rA,rB`
`tw<tc> rA,rB`	`tw <num>,rA,rB`
`tw<tc>i rA,target16`	`twi <num>,rA,target16`

Where `<tc>` and `<num>` are related as shown.

Table 6-43 <tc> and <num> Relationship

<tc>	<num>
`lt`	`16`
`le`	`20`
`eq`	`4`
`ge`	`12`
`gt`	`8`

Table 6-43 <tc> and <num> Relationship (continued)

<tc>	<num>
nl	12
ne	24
ng	20
llt	2
lle	6
lge	5
lgt	1
lnl	5
lng	6

Miscellaneous

Table 6-44 Miscellaneous

Extended Mnemonics	Equivalent To
`nop`	`ori r0,r0,0`
`li rX,value`	`addi rX,r0,value`
`lis rX,value`	`addis rX,r0,value`
`la rX,D(rY)`	`addi rX,rY,D`
`mr[.] rX,rY`	`or[.] rX,rY,rY`
`not[.] rX,rY`	`nor[.] rX,rY,rY`

Power Mnemonics Supported by PowerPC 601

Table 6-45 Power Mnemonics Supported by PowerPC 601

Mnemonic	Description
`abs[o][.]`	Absolute value
`clcs`	Cache line compute size
`div[o][.]`	Divide
`divs[o][.]`	Divide short
`doz[o][.]`	Difference or zero
`dozi`	Difference or zero immediate
`lscbx[.]`	Load string and compare byte indexed
`maskg[.]`	Mask generate
`maskir[.]`	Mask insert from register
`mul[o][.]`	Multiply
`nabs[o][.]`	Negative absolute value
`rlmi[.]`	Rotate left then mask insert
`rrib[.]`	Rotate right and insert bit
`sle[.]`	Shift left extended

Table 6-45 Power Mnemonics Supported by PowerPC 601 (continued)

Mnemonic	Description
`sleq[.]`	Shift left extended with `mq`
`sliq[.]`	Shift left immediate with `mq`
`slliq[.]`	Shift left long immediate with `mq`
`sllq[.]`	Shift left long with `mq`
`slq[.]`	Shift left with `mq`
`sraiq[.]`	Shift right algebraic immediate with `mq`
`sraq[.]`	Shift right algebraic with `mq`
`sre[.]`	Shift right extended
`srea[.]`	Shift right extended algebraic
`sreq[.]`	Shift right extended with `mq`
`sriq[.]`	Shift right immediate with `mq`
`srliq[.]`	Shift right long immediate with `mq`
`srlq[.]`	Shift right long with `mq`
`srq[.]`	Shift right with `mq`

PowerPC 403-Specific Mnemonics

Table 6-46 PowerPC 403-Specific Mnemonics

Mnemonic	Description
`rfci`	Return from critical interrupt
`dccci`	Data cache congruence class invalidate
`icbt`	Load instruction cache block
`iccci`	Instruction cache congruence class invalidate
`wrtee`	Write external enable
`wrteei`	Write external enable immediate
`mfdcr`	Move from device control register
`mtdcr`	Move to device control register

PowerPC 603-Specific Mnemonics

Table 6-47 PowerPC 603-Specific Mnemonics

Mnemonic	Description
`tlbld`	Load data TLB entry
`tlbli`	Load instruction TLB entry

PowerPC 602-Specific Mnemonics

Table 6-48 PowerPC 602-Specific Mnemonics

Mnemonic	Description
`mfrom`	Move from ROM
`esa`	Enable special access
`dsa`	Disable special access

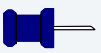
Stack Checking

This section provides PowerPC-specific information about stack checking. Refer to ***Using Ultra C/C++*** for more general information on stack checking.

If stack checking is inappropriate for the module being created, you will need to define the following items:

- a global pointer called `_stbot` (initialized to `ULONG_MAX` if possible)
- a function called `_stkhandler` (it returns to its caller)

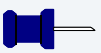
A function called `_stkoverflow` is called when the stack appears to overflow. If a non-static function called `_stkoverflow` resides in an ROF that is linked to make the object module, that function is called instead of the default function. The default function writes a message to the standard error path and causes the program to exit with a status of one. `_stkoverflow` neither accepts parameters nor returns a value.



Note

If `_stkoverflow` is inappropriate for your application, consider writing a function to handle stack overflow.

`_stkhandler`, the function that checks for stack overflow, can be revised. Revision may be necessary if stack checking is inapplicable to the module that calls the library functions. `_stkhandler()` is passed the desired stack pointer in `r3` and does not return a value.



Note

Stack handler code must be compiled with stack checking turned off (`-r` in ucc mode).

Chapter 7: SuperH

This chapter contains information specific to the Hitachi SuperH family of processors. The following sections are included:

- **Executive and Phase Information**
- **C/C++ Application Binary Interface Information**
- **Assembly Language with SH-4 Target**
- **SuperH Processor-Specific Optimizations**
- **_asm() Register Pseudo Functions**
- **Assembler/ Linker**
- **Working with PC-Relative Data**
- **Stack Checking**

Executive and Phase Information

This section describes various features of the compiler executive and phases that are SuperH specific.

Executive -tp Option

`-tp[=]<target>{[,]<suboptions>}`

Specify Target Processor and Target Processor Options

Specify the target processor and target processor sub-options. Target processors are identified in **Table 7-1** and target processor sub-options are identified in **Table 7-2**.

Table 7-1 Target Processor

SuperH Family Targets	Target Processor
`sh, sh3`	SH7708/SH7709
`sh4`	SH7750

Table 7-2 Mode -tp Sub-Options

Suboptions	Description	Processor
`fp`	Use a statically linked library to implement floating-point operations (default)	SH-3
`fpd`	Enable denormalized number support.	SH-4
`fps`	Use dynamically linked shared library to implement floating-point operations.	SH-3
`sd`	Use 16-bit data references (default)	SH-3, SH-4
`ld`	Use 32-bit data references	SH-3, SH-4
`scd`	Use 16-bit code area data reference (default)	SH-3, SH-4
`lcd`	Use 32-bit code area data references	SH-3, SH-4
`sc`	Use 12-bit code references	SH-3, SH-4
`mc`	Use 16-bit code references (default)	SH-3, SH-4
`lc`	Use 32-bit code references	SH-3, SH-4
`sb`	Use 8-bit branches	SH-3, SH-4
`mb`	Use 12-bit branches (default)	SH-3, SH-4
`lb`	Use 32-bit branches	SH-3, SH-4

Predefined Macro Names for the Preprocessor

The macro names in **Table 7-3** are predefined in the preprocessor for target systems.

Table 7-3 Macros

Macro	Description
_MPFSH	Generic SuperH processor
_MPFSH3	SH-3 (SH7708/SH7709) processor
_MPFSH4	SH-4 (SH7750) processor
_FPFSH	SH-3 Floating-point support
_FPFSH4	SH-4 Floating-point support

Target macros are used to conditionalize code so that machine- and operating system-independent programs can be created.

The executive automatically defines the _MPF and _FPF macro for the target processor. **Table 7-4** provides an example of this behavior.

For more information on exactly which macros are defined for the SuperH processors, run the executive in verbose and dry run modes stopping after the front end. For example, to check the defines on for the SH-3 target (source file need not actually exist):

```
xcc -b -h -efe -tp=sh3 t.c
```

This causes the executive to print a line for SH-3 similar to:

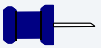
```
"cpfe -m --target=11 -Id:\MWOS\SRC\DEFS -I\MWOS\OS9000\SRC\DEFS
-I\MWOS\OS9000\SH3\DEFS -D_UCC -D_MAJOR_REV=2 -D_MINOR_REV=2
-D_SPACE_FACTOR=1 -D_TIME_FACTOR=1 -D_OS9000 -D_MPFSH3 -D_MPFSH
-D_FPFSH -D_BIG_END -w --Extended_ANSI --gen_c_file_name=t.i t.c"
```

To check the defines for an SH-4 target:

```
xcc -b -h -efe -tp=sh4 t.c
```

The executive prints a line for SH-4 similar to:

```
"cpfe -m --target=14 -Id:\MWOS\SRC\DEFS -Id:\MWOS\OS9000\SRC\DEFS
-Id:\MWOS\OS9000\SH4\DEFS -D_UCC -D_MAJOR_REV=2 -D_MINOR_REV=2
-D_SPACE_FACTOR=1 -D_TIME_FACTOR=1 -D_OS9000 -D_MPFSH4 -D_MPFSH
-D_FPFSH4 -D_BIG_END -w --Extended_ANSI --gen_c_file_name=t.i t.c"
```



Note

Note that _MPFSH, _MPFSH4 macros are defined.

The _MPFSH macro indicates that a source file is being compiled for a SuperH family target.

Table 7-4 identifies the relationship between the target processor and the preprocessor macros. Note that specific targets also define the general macro _MPFSH.

Table 7-4 _MPFxxx and _FPFxxx Macro Behaviors

Target	Microprocessor Family Macros Defined
sh3	_MPFSH3 _MPFSH _FPFSH
sh4	_MPFSH4 _MPFSH _FPFSH4

SuperH-Unique Phase Option Functionality

Phases having unique phase option functionality on the SuperH processor are:

- **Back End Options**
- **Assembly Optimizer Options**
- **Linker Options**

Back End Options

`-m=<non remote memory left>`
: Informs the back end that other files in the program have used some amount of the 64K non-remote data.

`-pd`
: Enables denormalized number support. (SH-4)

The back end orders the data area based on static analysis of the data area objects and sorts the data based on usage and size. This means that the most heavily used objects end up in the non-remote area. To do this, the back end needs information about how the object-code linker lays out the data area for the entire program. This option may be used to inform the back end that the files it is unable to process will use a specified amount of data space.

Code generation options provide specifications for code generated by the back end. Most of these options can be controlled via executive command line options, thus there should not be a need to explicitly use them.

Table 7-5 Code Generation Options

Option	Description
`-pg`	Use stand-alone code area address calculation (This option is obsolete. It is provided for backwards compatibility, but it has no effect on code generation.)
`-pl`	Cause long references to global/static data objects
`-pla`	Cause branches to be long
`-plb`	Cause function calls to be long
`-plc`	Cause some references to code objects to be long
`-pmb`	Cause function calls to be medium (default)
`-pnp`	Save the previous stack pointer on the stack
`-ps`	Do not emit stack checking code

Target architecture code generation options provide specifications unique to a target architecture for code generated by the back end.

Table 7-6 SH-3 Code Generation Options

Option	Description
`-pSH3`	Does not have floating-point hardware support. Supports dynamic shift operations. (besh only)
`-pSH4`	Has single and double-precision hardware floating-point support; supports dynamic shift operations. (besh4 only)

Assembly Optimizer Options

`-b`	Do not perform long/medium branch simplification
`-t[=]<num>`	Specify target processor family

Table 7-7 Assembly Optimizer Processor Numbers

<num>	Assembly Optimizer Target
1	SH-3 family processor
2	SH-4 family processor

`-p=<X>`	Selectively skip processor-specific optimizations

Table 7-8 Assembly Optimizer Processor-Specific Optimizations

<X>	Processor-Specific Optimization
`l`	Location (memory and register contents) tracking
`d`	Branch delay slot filling
`p`	Pooling of PC-relative data
`r`	Copy propagation
`n`	Register renaming

`-s<method>`	Set the peephole scheduling method

Table 7-9 Peephole Scheduling Methods

<method>	Description
`s`	Spread dependent instructions (default)
`n`	No reordering of instructions

Assember Options

`-m`	Accept a target number.

Linker Options

`-t=<target>` Linker, specify target module type

Table 7-10 Target Module Type

Target	Module Type
`os9k_sh`	OS-9 for SH-3 Family Processors
`os9k_sh4`	OS-9 for SH-4 Family Processors

C/C++ Application Binary Interface Information

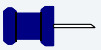
Register usage, passing arguments to functions, and language features are described in this section.

Register Usage

General purpose and other registers are identified in this section.

Table 7-11 Register Classes

Register Class	Names Used	Processor
General Purpose Registers	`r0 - r15, gp, sp`	SH-3, SH-4
System Registers	`mach, macl, pr, pc, fpul`	SH-3, SH-4
Control Registers	`sr, gbr, vbr, ssr, spc`	SH-3, SH-4
Control Register	`fpscr`	SH-4
Floating point Registers	`fr0 - fr15` `dr0, dr2, ..., dr14`	SH-4



Note

The single and double precision registers for SH-4 are not separate. Each double precision register is formed from two single precision registers: fr0 and fr1 comprise dr0, fr2 and fr3 comprise dr2, and so forth. Loading dr0 will change fr0 and fr1 and vice versa.

General Purpose Registers

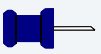
The following table describes names and meanings of the general purpose register set used by the C/C++ Compiler and supported by the Ultra C assembler. The assembler supports both the register number and the alternate name. The alternate name is used for convenience since the name gives the user a hint to its intended purpose.

Figure 7-1 General Purpose Register

Register #	Alias	Description	Processor
`r0`		Caller-saved; function integer return register; for functions returning aggregates, this points to the returned aggregate.	SH-3, SH-4
`r1-r3`		Caller-saved/volatile.	SH-3, SH-4
`r4-r7`		Caller-saved; Up to four arguments placed in these registers	SH-3, SH-4
		`r4/r5` pair for floating point return	SH-3
`r8-r13`		Callee-saved; for locals or temporaries	SH-3, SH-4
`r14`	`gp`	Biased pointer to static data area	SH-3, SH-4
`r15`	`sp`	Stack pointer	SH-3, SH-4

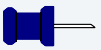
Figure 7-1 General Purpose Register

Register #	Alias	Description	Processor
macl mach gbr		Callee saved	SH-3, SH-4
fpul		Caller saved	SH-4
fr0		Caller saved; float point function return register	SH-4
fr1-fr3		Caller saved	SH-4
fr4-fr11		Caller saved/volatile. Up to eight single-precision arguments are placed in these registers.	SH-4
fr12-fr15		Callee saved	SH-4
dr0		Caller saved Double function return register	SH-4
dr2		Caller saved	SH-4
dr4-dr10		Caller saved; up to four double precision arguments are placed in these registers	SH-4
dr12, dr14		Callee saved registers	SH-4



Note

No register is dedicated to point at constant data. Refer to a later section regarding code area data item references.



Note

If a floating point needs to be performed in interrupt service routines, all of the caller-saved floating point registers, including FPUL and FPSCR registers, will have to be saved manually. You do not have to save the callee-saved floating point registers because the Compiler will save them.

Passing Arguments to Functions

When an argument is passed to a called function, the argument is in one of two places, in a register or in the Output Parameter Area (OPA) of the calling function.

The called function determines the location of the argument by argument type and the order specified in the argument list. Standard argument promotions are assumed to be used where applicable (example: for optional parameters and parameters in the absence of a prototype). Optional parameters and parameters of aggregate types are passed on the stack regardless of position.

With the SH-4 processor, single precision floating point parameters are passed in single precision registers when available. If all such registers set aside for parameter passing are in use, single precision floating point parameters are passed on the stack. Double precision floating point parameters use double precision registers, which are comprised of consecutive pairs of single precision registers. If a pair is not available, the value is passed on the stack. If an odd number of single precision parameters precede a double precision parameter passed in a register, a gap is left in the single precision registers, which may be filled by a later single precision parameter. For example, in a function

```
double indemnity (float f1, double d, float f2)
```

the parameter f1 will be passed in fr4, d will be passed in dr6 (which consists of fr6 and fr7), and f2 will be passed in fr5.

The SH-3 processor does not have floating point registers. Therefore, single precision floating point parameters use integer registers when available. If one is not available, it is passed on the stack. Double precision floating point parameters use a consecutive pair of integer registers. If a pair is not available, the value is passed on the stack. If there is a single parameter register available, it is used for a later argument if it fits.

For SH-3, if a function is to return a value, an integral return value is returned in `r0` or a single precision in `r4`; double precision in `r4-r5` pair.

For SH-4, if a function is to return a value, an integral return value is returned in `r0` or a single precision in fr0; double precision in `dr0`.

If a function is to return a struct, the address of a return area is passed as the first integral argument (r4) and is returned in the integral return register (r0).

SH-3 Floating Point Parameter Example

This is an example function for an SH-3 target.

```
func(int, int, float double, float);
func1() {
    int a, b;
    float c, d;
    double d1;

    func(a, b, c, d1, d);
}
```

This function generates the following code:

```
...
*   func(a, b, c, d1, d);
 mov.l @(16,r15),r4  <--  first parameter (int) in r4
 mov.l @(20,r15),r5  <-- second parameter (int) in r5
 mov.l @(24,r15),r6  <-- third parameter (float) in r6
 mov.l @(28,r15),r1  <-- fourth parameter (double) needs two consecutive
 mov.l @(32,r15),r2        registers, but we only have r7 left, so it is passed
 mov.l r1,@(8,sp)          on the stack, in the OPA
 mov.l r2,@(12,sp)
 mov.l @(36,r15),r7  <-- fifth parameter (float) uses the remaining register r7
 mov.w @(=_$L59,pc),r1
 bra =_$L60 **skip
 nop
=_$L59
 dc.sw =func-=_$L60-4
=_$L60
 bsrf r1
 nop
...
```

SH-4 Floating Point Parameter Example

This is an example function for an SH-4 target.

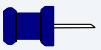
```
extern void     func(int, int, float, double, float, int, int,
int);

func1()
{
        int     a, b, c, d, e;
        float   f1, f2;
        double  d1;

        func(a, b, f1, d1, f2, c, d, e);
}

*     func(a, b, f1, d1, f2, c, d, e);
 mov.l @(16,r15),r4     <-- first parameter (int) in r4
 mov.l @(20,r15),r5     <-- second parameter (int) in r5
 mov #24,r1
 add r15,r1
 sts fpscr,r2 **PR=0
 mov #-9,r3
 swap.w r3,r3
 and r3,r2
 lds r2,fpscr **PR
 fmov.s @r1,fr4         <-- third parameter (float) in fr4
 mov #32,r1
 add r15,r1
 fmov.s @r1+,fr6        <-- fourth parameter (double) in dr6
(fr6/fr7)
 fmov.s @r1,fr7
 mov #40,r1
 add r15,r1
 fmov.s @r1,fr5         <-- fifth parameter (float) in fr5
 mov.l @(44,r15),r6     <-- sixth parameter (int) in r6
 mov.l @(48,r15),r7     <-- seventh parameter (int) in r7
 mov.l @(52,r15),r1
 mov.l r1,@(8,sp)       <-- eighth parameter (int) on the stack
 mov.l @(=_$L22,pc),r1
 bra =_$L23 **skip
```

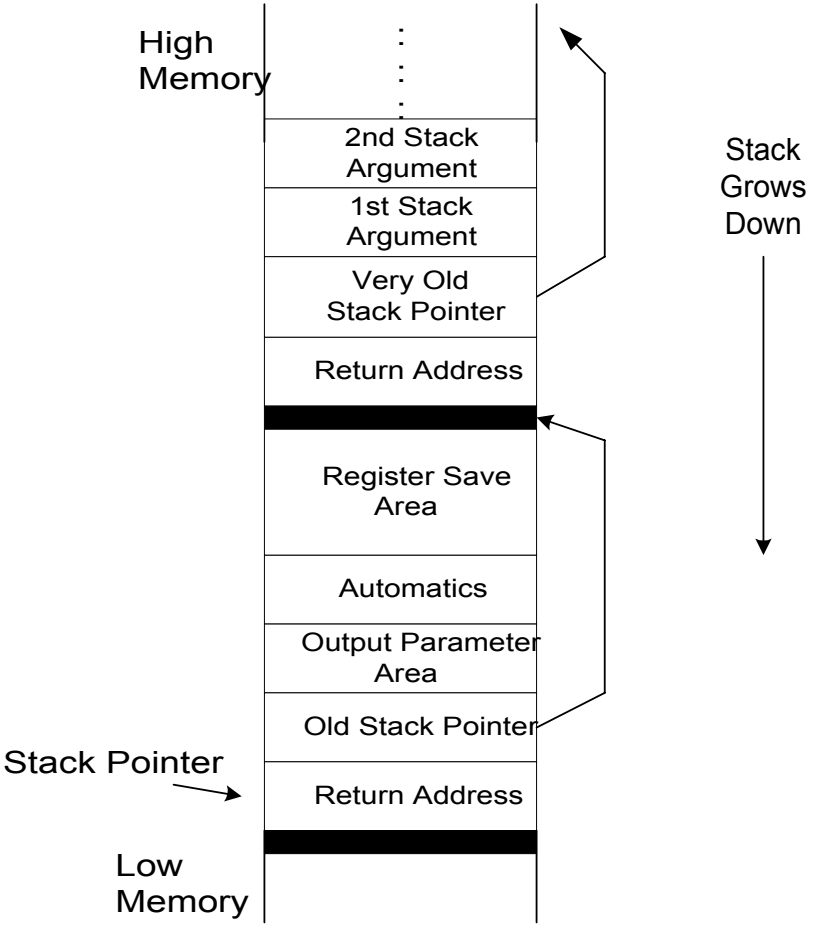
```
 nop
 align 4
=_$L22
 dc.l =func-=_$L23-4
=_$L23
 bsrf r1
 nop
```



Note

Individual parameters passed on the stack are aligned to the maximum alignment requirement of any data type on a target (8 bytes for SH-4).

Figure 7-2 Stack Frame for a Function



Stack Pointer	Points to the one-time allocated stack for the function
Old Stack Pointer	Contains the calling function's stack pointer. It is only used for "unwinding" the stack by the debugger.
Output Parameter Area	The area for stack arguments that are passed to functions that this function calls
Automatics	The area for function locals (and compiler generated temporaries)
Register Save Area	The area the compiler saves the callee-saved registers that it uses
Return Address	This place is reserved for non-leaf functions to save their return address
Very Old Stack Pointer	The saved stack pointer for the function that called the calling function
Stack Arguments	These are the argument that were passed on the stack to this function

SH-3 Stack Alignment

The **register save area**, **automatic area**, **output parameter area** and **stack pointer** are all 4-byte aligned. Padding is added where needed.

SH-4 Stack Alignment

The **register save area**, **automatic area**, **output parameter area** and **stack pointer** are all 8-byte aligned. Padding is added where needed.

C Language Features

For More Information

Other implementation-defined areas are identified in the Language Features chapter of the ***Using Ultra C/C++*** manual and the **Overview** chapter of the ***Ultra C Library Reference*** manual. Refer to an ANSI/ISO specification for more information.

In conformance with the ANSI/ISO C specification, the implementation-defined areas of the compiler are listed in this section. Each bulleted item contains one implementation-defined issue. The number in parentheses included with each bulleted item indicates the location in the ANSI/ISO specification where further information is provided.

Characters

- The number of bits in a character in the execution character set (5.2.4.2.1).

  There are 8 bits in a character in the execution character set.

Integers

- The representations and sets of values of the various integer types (6.1.2.5).

Table 7-12 Integer Type/Range

Type	Representation	Minimum / Maximum
`char`, `signed char`	8-bit 2's complement	`-128 / 127`
`unsigned char`	8-bit binary	`0 / 255`
`short int`	16-bit 2's complement	`-32768 / 32767`
`unsigned short int`	16-bit binary	`0 / 65535`
`int`	32-bit 2's complement	`-2147483648 / 2147483647`
`unsigned int`	32-bit binary	`0 / 4294967295`
`long int`	32-bit 2's complement	`-2147483648 / 2147483647`
`unsigned long int`	32-bit binary	`0 / 4294967295`

- The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented (6.2.1.2).

  When converting a longer integer to a shorter signed integer, the least significant <n> bits of the longer integer are moved to the integer of <n> bits. The resulting value in the smaller integer is dictated by the representation. For example, if the conversion is from `int` to `short`, the least significant 16 bits are moved from the `int` to the `short`. This value is then considered a 2's complement 16-bit integer.

  When conversion from unsigned to signed occurs with equally sized integers, the most significant bit becomes the sign bit. Therefore, if the unsigned integer is less than 0x80000000, the conversion has no affect. Otherwise, a negative number results.

- The sign of the remainder on integer division (6.3.5).

  The result of an inexact division of one negative and one positive integer is the smallest integer greater than or equal to the algebraic quotient.

  The sign of the remainder on integer division is the same as that of the dividend.

Floating-Point

- The representations and sets of values of the various types of floating-point numbers (6.1.2.5).

Table 7-13 Floating-Point Number Characteristics

Type	Format	Minimum / Maximum
`float`	32-bit IEEE 754	`1.17549435e-38f / 3.40282347e38f`
`double`	64-bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`
`long double`	64-bit IEEE 754	`2.2250738585072016e-308 / 1.7976931348623157e308`

Refer to the `float.h` header file for other limits and values.

Arrays and Pointers

- The type of integer required to hold the maximum size of an array. That is, the type of the size of operator, `size_t` (6.3.3.4, 7.1.1).

  An `unsigned long int` is required to hold the maximum size of an array. `unsigned long int` is defined as to `size_t` in `ansi_c.h`.

- The result of casting a pointer to an integer or vice versa (6.3.4).

  Since pointers are treated much like unsigned long integers, the integer is promoted using the usual promotion rules to an unsigned long. That is, the sign bit propagates out to the full 32-bit width.

- The type of integer required to hold the difference between two pointers to elements of the same array, `ptrdiff_t` (6.3.6, 7.1.1).

  A `signed long int` is required to hold the difference between two pointers to elements of the same array. `long int` is defined as `ptrdiff_t` in `ansi_c.h`.

Registers

- The extent to which objects can actually be placed in registers by use of the register storage-class specifier (6.5.1).

  The compiler automatically makes decisions about what objects are placed in registers, thus giving no special storage considerations for the register storage-class.

Structures, Unions, Enumerations, and Bit-Fields

- The padding and alignment of members of structures (6.5.2.1). This should present no problem unless binary data written by one implementation are read by another.

Table 7-14 shows the alignment of the various objects within a structure. Required padding is supplied if the next available space is not at the correct alignment for the object. For example, a structure declared as:

```
struct {
   char mem1;
   long mem2;
};
```

would be an 8-byte structure: one byte for `mem1`, three bytes of padding to get `mem2` to 4-byte alignment, and four bytes for `mem2`.

Table 7-14 Alignment Table

Type	Alignment Requirement
`char`	1
`short`	2
`int`	4
`long`	4
`pointers`	4
`float`	4
`double`	SH-3 target: 4 SH-4 target: 8
`long double`	SH-3 target: 4 SH-4 target: 8

- Whether “plain” `int` bit-field is treated as a signed `int` or as an unsigned `int` bit-field (6.5.2.1).

  A “plain” `int` bit-field is treated as a signed `int` bit-field.
- The order of allocation of bitfields within a unit (6.5.2.1).

  Bit fields are allocated from most-significant bit to least-significant bit.

- Whether a bit-field can straddle a storage-unit boundary (6.5.2.1).

  Bit fields are allocated end-to-end until a non-bit field member is allocated or until that positioning would cross an addressable boundary such that no object of an integral type could both contain the bit field and be correctly aligned.

- The integer type chosen to represent the values of an enumeration type (6.5.2.2).

  `Enum` values are represented in 32 bit two's complement integers.

Processing Directives

- Whether the value of a single-character, character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. Whether such a character constant may have a negative value (6.8.1).

  The value of a single-character character constant in a constant expression that controls inclusion matches the value of the same character constant in the execution character set. This character constant may have a negative value.

- The method for locating includable source files (6.8.2).

  This method is described in the **Using the Executive** chapter of the ***Using Ultra C/C++*** manual.

- The support of quoted names for includable source files (6.8.2).

  Quoted names are supported for `#include` preprocessing directives.

- The mapping of source file character sequences (6.8.2).

  The mapping of source file character sequences is one-to-one. The case-sensitivity of file names depends on the file system being used.

Assembly Language with SH-4 Target

Because there are few instruction encodings with fixed-length 16-bit instructions, integer and floating point behaviors are affected. The permissible immediate operands range and the displacement sizes are limited and r0 is widely used as an implicit operand or component of instructions. The greatest effects with floating point are:

4. The bank-switching nature of the floating point registers under control of bits in the FPSCR
5. The overloading of particular bit patterns so that the same instruction encoding behaves differently under control of bits in the FPSCR

When working with the FPSCR register and assembly language escapes that do floating point, ensure that the SZ and FR enable bits come out the same way they go in.

At the time of this writing, the SH-4 back end does not allow values to be encouraged into the xd registers as is possible with the fr and dr registers.

SuperH Processor-Specific Optimizations

In addition to providing the standard generic assembly optimizations, the Ultra C/C++ icode optimizer (`iopt`) and SuperH assembly optimizer (`optsh`) provide processor-specific optimizations. These are:

Special Common Sub-Expressions

Delay Slot Filling

Long/Medium Branch Simplification

Code Area Data Pooling and Consolidation

Copy Propagation

Special Common Sub-Expressions

On the SuperH processors, loading certain constants (< -128 or >= 128 for signed constants, >= 128 for unsigned constants, all floating point constants on SH-3 and all floating point constants except single precision 0 and single precision 1 on the SH-4 processor) and computing the addresses of global variables and functions are very expensive due to the limited size of displacement in most instructions. So, the icode optimizer creates common sub-expressions associated with the above so they can be computed just once into a register and the contents of the register can be reused every time a reference to the expression is made.

Delay Slot Filling

In order to reduce code size and/or increase code efficiency, the assembly optimizer attempts to fill the delay slots of those instructions that have them with useful instructions. To do this, it looks for a movable instruction in the series of preceding instructions or, in some cases, the following or destination instructions. If the delay slot cannot be filled with a useful instruction, it is either left alone or, in the case of conditional branches, the delay slot is removed altogether.

Long/Medium Branch Simplification

During the code generation phase, the compiler is unaware of the distances between different sections of code. As a result, the back end is somewhat conservative in the branch instructions that it emits. For example, the back end produces sub-optimal code for the following C code:

```
* if (x) y *= x;

 tst r4,r4
 bf =_$L1
 bra =_$138e
 nop
=_$L1
 mul.l r4,r5
 sts macl,r5
=_$138e
 ...
```

The assembly optimizer attempts to shorten these branching sequences. Given the above example, the assembly optimizer outputs the following:

```
 tst r4,r4
 bt =_$138e
=_$L1
 mul.l r4,r5
 sts macl,r5
=_$138e
```

Here is another example of a long branch to subroutine:

```
 ...
 mov.w @(=_$L3,pc),r0
 bra =_$L4 **skip
 nop
 align 2
=_$L3
 dc.sw =func-=_$L4-4
=_$L4
```

```
 bsrf r0
 nop
 ...
```

This can be converted to a simple branch to subroutine, if the assembly optimizer can determine that the branch target is in range:

```
...
bsr =func
nop
...
```

Code Area Data Pooling and Consolidation

The SuperH instruction set limits the size of immediate values to 12 bits (in the case of BRA and BSR); generally fewer. As a result, it is often necessary to load data into registers. This is what is done for long branches and subroutine calls as well as large numerical values. This data can be stored in the code area and copied into registers using PC-relative loads. The assembly optimizer attempts to pool this code area data together beyond naturally occurring divisions (such as a return from subroutine) to limit the number of extra branches in code. For example, given the following code:

```
extern int x;
func(&x);
```

The back end might generate:

```
 mov.w @(=_$L6,pc),r4
 bra =_$L7 **skip
 nop
=_$L6
 dc.sw =w
=_$L7
 add r14,r4
 mov.w @(=_$L3,pc),r0
 bra =_$L4 **skip
 nop
 align 2
```

```
=_$L3
 dc.sw =func-=_$L4-4
=_$L4
 bsrf r0
 nop
 ...
 rts
 nop
```

The assembly optimizer could change this to:

```
 mov.w @(=_$L6,pc),r4
=_$L7
 add r14,r4
 mov.w @(=_$L3,pc),r0
=_$L4
 bsrf r0
 nop
 ...
 rts
 nop
=_$L6 dc.sw =x
 align 2
=_$L3 dc.sw =func2-=_$L4-4
```

Copy Propagation

The assembly code optimizer tries to eliminate needless copies between register temporaries; resulting in smaller, more efficient code. For example:

```
mov.l @r0,r8
mov r8,r7
exts.b r7,r7
```

This may be changed to the following:

```
mov.l @r0,r8
exts.b r8,r7
```

As another example:

```
mov.l @r0,r8
mov r8,r7
exts.b r8,r8
mov #0,r7
```

This may be changed to the following:

```
mov.l @r0,r7
exts.b r7,r8
mov #0,r7
```

_asm() Register Pseudo Functions

`_asm()` uses register pseudo functions as identified in **Table 7-15**.

Table 7-15 _asm() Register Pseudo Functions

Register	Description	Processor
`__reg_gen`	Any non-dedicated integer register	SH-3
`__reg_r<n>`	The register specified by `n (0 <= n <= 13)`	SH-3
`__reg_single`	Any single precision register	SH-4
`__reg_double`	Any double precision register	SH-4
`__reg_fr<n>`	The single precision register specified by n (0 <= n <= 15)	SH-4
`__reg_dr<n>`	The double precision register specified by n (n in {0, 2, 4, 6, 8, 10, 12, 14})	SH-4

Assembler/ Linker

The assembler allows use of standard SuperH assembly language mnemonics and syntax, modified as described in this section. For more specific information about individual instructions, consult the following documentation:

- ***SH7000/SH7600 Series Programming Manual***
- ***SH7700 Series SuperH RISC Engine Programming Manual***
- ***SH-3E Single-Chip RISC Microprocessor Programming Manual***

ROF Edition Number

The SuperH assembler emits ROF Edition #15.

External References

The SuperH assembler allows the use of external references with any operators within any expression fields not defined to be constant expression fields.

Symbol Biasing

The linker does not bias code or data symbols for system, file managers, device drivers, device descriptors, or data modules. For all other types of modules, or for raw code, the linker biases both code and data symbols by `-32764 (0x7ffc)`. Initialization routines for raw code should ensure that the static storage pointer (`r14`) is initialized with the proper base address, adjusted to account for the biasing.

Assembler Syntax Extensions and Limitations

The Ultra C/C++ compiler's adaptation of the SuperH instruction syntax has a few notable differences from what is defined in SuperH programming reference manuals:

- The assembler uses white space as the comment delimiter. As a result, the operand stream must not include any white space.
- The forward slash character ("/") is not allowed within instruction mnemonics. As a result, those mnemonics that contain a forward slash (such as `BF/S` and `CMP/EQ`) contain a period (".") instead.

An example of SuperH syntax and its equivalent Ultra C/C++ syntax are shown in **Table 7-16**.

Table 7-16 SuperH and Ultra C/C++ Syntax Equivalents

SuperH Syntax	Ultra C/C++ Syntax
`bf/s label`	`bf.s label`
`cmp/eq r1,r2`	`cmp.eq r1,r2`

To see other uses of Ultra C/C++ assembly, it is possible to stop the compilation process after the back end has finished (using the `-ebe` compiler switch) and view the resulting assembly file.

Global Data Accessing

Global data for a process is stored in a single region of memory, accessed via a dedicated register. Since the SuperH's load with displacement only has a 4-bit unsigned displacement field, this instruction is not practical to use for global data accesses and to use it would also cause other modes to be too limited. A similar situation exists for the use of the following code sequence since this only accesses a maximum of 256 bytes, words and/or longs (less if larger structures are involved).

```
            mov         lo8(_symb),r0
            mov.l       @(r0,gp),rd
```

Therefore, we recommend adopting two global data accessing modes, one for accessing up to 64K of data and another that allows the entire 32-bit address space to be accessed.

The following code can be emitted for short data accesses:

```
            mov.w       @(_symb_addr,PC),r0
            bra         _around
            nop
_symb_addr  dc.sw       _symb
_around     mov.l       @(r0,gp),rd
```

It can handle offsets from -32768 to 32764, thereby allowing access to approximately 64K of data. For long data accesses, the following PC-relative addressing mode should be used.

```
            mov.l       @(_symb_addr,PC),r0
            bra         _around
            nop
            align4
_symb_addr  dc.l        _symb
_around     mov.l       @(r0,gp),rd
```

The above examples assume:

- `rd` is the destination register
- `_symb` is a referenced value to be replaced by the linker
- `gp` is the name of the dedicated register “pointing” to the global data area

The assembly code optimizer eliminates as many of the branches around displacements in code as possible.

Code Accessing

Ultra C/C++ takes advantage of the SuperH PC-relative load and mova instructions to perform code accesses. Therefore, you can adopt two code accessing modes: one for accessing up to 64K of code and another for accessing the entire 32-bit address space.

The following code can be emitted for short code accesses:

```
L1          mova        @(0,PC),r0
            mov.w       @(L2,PC),rd
            bra         L3
            nop
L2          dc.sw       -((L1&0xfffffffc)+4)-_symb
L3          add         r0,rd
```

It can handle offsets from -32768 to 32764, thereby allowing access to approximately 64K of data. For long code accesses, the following PC-relative addressing mode should be used.

```
L1          mova        @(0,PC),r0
            mov.l       @(L2,PC),rd
            bra         L3
            nop
            align4
L2          dc.l        -((L1&0xfffffffc)+4)-_symb
L3          add         r0,rd
```

The above examples assume:

- `rd` is the destination register
- `_symb` is a referenced value to be replaced by the linker

Calling Functions

The SuperH processor supports three different models for calling a function. The short model works for programs under 4K. Calls to a function look like this:

```
            bsr  _printf
```

In the medium model, programs is guaranteed to work if their size is less than 32K. Code for these calls look like this:

```
          mov.w   @(_pntf_ofst,PC),r0
          bra     _call_pt
          nop
_pntf_ofstdc.sw   _printf-_call_pt-4
_call_pt  bsfr    r0
          nop
```

For any other programs, the code looks like this:

```
          mov.l   @(_pntf_ofst,PC),r0
          bra     _call_pt
          nop
          align 4
_pntf_ofstdc.l    _print-_call_pt-4
_call_pt  bsrf    r0
          nop
```

In this way, the user can minimize the size of the code.

Calls through pointers to functions should be done in the following manner:

```
          <get the value in register rd>
          jsr     @rd
```

Working with PC-Relative Data

The SuperH instruction set limits branch labels to 12 bits. It is possible to perform branches using 16 or 32 bits by loading a PC-relative displacement into a register. Following are examples of medium and long branches to a destination symbol `=dest`:

Medium branch to dest

```
        mov.w   @(disp,pc),r0
baddr   braf    r0
        nop
        slign   2
disp    dc.sw   dest-baddr-4
```

Long branch to dest

```
        mov.l   @(disp,pc),r0
baddr   braf    r0
        nop
        align   4
disp    dc.l    dest-baddr-4
```

It is important to note the `.sw` extension used on the define constant pseudo-instruction in the medium branch example. This is necessary because the value is to be signed displacement and the linker should give an error if it does not fit within `-32768` and `32767`.

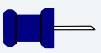
Stack Checking

This section provides SuperH-specific information about stack checking. Refer to ***Using Ultra C/C++*** for more general information on stack checking.

If stack checking is inappropriate for the module being created, you will need to define the following items:

- a global pointer called `_stbot` (initialized to `ULONG_MAX` if possible)
- a function called `_stkhandler` (it returns to its caller)

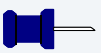
A function called `_stkoverflow` is called when the stack appears to overflow. If a non-static function called `_stkoverflow` resides in an ROF that is linked to make the object module, that function is called instead of the default function. The default function writes a message to the standard error path and causes the program to exit with a status of one. `_stkoverflow` neither accepts parameters nor returns a value.



Note

If `_stkoverflow` is inappropriate for your application, consider writing a function to handle stack overflow.

`_stkhandler`, the function that checks for stack overflow, can be revised. Revision may be necessary if stack checking is inapplicable to the module that calls the library functions. `_stkhandler()` is passed the desired stack pointer in `r3` and does not return a value.



Note

Stack handler code must be compiled with stack checking turned off (`-r` in ucc mode).