User Guide to

SNC PPU C/C++ Compiler

SN Systems Limited
Version v470.1
February 27, 2015

Contents

1: Introduction

This manual provides information on how to use the SNC compiler.

Information provided in this manual includes:

- How to compile, assemble, and link programs.
- How to control compiler behavior during compilation.
- Which kinds of optimization are performed by the compiler.
- How the languages accepted by the SNC compiler compare with industry-standard definitions of those languages.
- Which programming restrictions apply when the SNC compiler is used, and how to deal with programs that violate these restrictions.
- How to use additional target-specific features unique to the SNC compiler.

Information that is not provided in this manual includes:

- How to write programs in general.
- How to use the various aspects of programming indirectly associated with compiling and executing programs, such as:
- preparing program files to be input to the compilers
- using tools to automate the compilation process
- using the debugger
- using a performance monitoring facility
- manipulating files output from program execution

For a quick start guide to the SNC compiler, see "Quick guide to using the SNC compiler".

Throughout this manual, C and C++ are collectively referred to as "C/C++". Cases specific to each language are noted by reference to the specific language. The addition of "SNC" to any of the above languages denotes the appropriate SNC compiler.

The following definitions apply to terms used throughout this manual:

Term	Definition
control-variable	A quantity that is similar in concept to a variable in a programming language, but that exists only during compilation and that is used to control the behavior of the compiler. Note the hyphen in "control-variable": this distinguishes it from more casual use of the term, e.g. "The loop control-variable ...".
host computer	The particular kind and model of computer on which the compiler is running. Usually (but not necessarily) the same as the *target computer*.
intrinsic-function	A *function* whose effect is defined as a part of the definition of the source language in use, so that it can be called by the user without needing to be supplied.
main-function	A function coded so that it is the starting point for program execution. In C/C++, a function named `main()`.
program	A collection of *functions* organized so as to execute together. Each program must have exactly one *main-function*, and may also have any number of non-main-functions.
program failure	Any behavior of the compiled program that causes it to produce the wrong answers, or to fail to complete execution properly. The concept of correctness

	against which program failure or success is measured can come from either a standard language definition or from the behavior of the program when compiled by some other compiler.
function	A subroutine, or procedure. The term "function" includes functions that may or may not return a value, which have either a single or multiple entry points, and which are either written by the user or are *intrinsic-functions*. In C/C++, the following are *functions*: a user-written function or a library function. A preprocessor macro is not a function.
target computer	The particular model of computer for which the compiler is to produce compiled code, and the operating environment on it.

Quick guide to using the SNC compiler

Read this section for a quick introduction to using the SNC compiler.

Option naming

The SNC compiler strongly follows UNIX tradition for option naming. Where there is no established tradition, options unique to the SNC compiler are used.

- The SNC compiler options may be specified using either the UNIX style '-' prefix, or the Windows style '/' prefix.

The common UNIX compiler options accepted by the SNC compiler are as follows: -c, -g, -l, -o, -w, -A, -C, -D, -E, -H, -I, -L, -O, -S, and -U. See "Compiler driver options" for details.

For non-traditional options, which permit very detailed control of the compiler, see the -X option in the table of "Compiler driver options" and the table of control-variables in "Control-variable reference".

Optimization control

Optimization control is done with the `-On` option, where `n` can range from `0` to `3`.

The default setting is `-O0`, which does no optimization and only forced inlining. Specifying just `-O` is equivalent to `-O2` and does full optimization and inlining. Level `-O3` adds the following additional optimizations:

- Aggressive auto-inlining (`-Xautoinlinesize`)
- Aggressive block-reordering (`-Xreorder`)
- Aggressive loop-unrolling (`-Xunrollssa`)
- Integral-comparison transformations (`-Xuseintcmp`)

Debugging optimized code is more challenging than debugging code compiled at `-O0`. SNC provides the `-Od` option that optimizes code with consideration for debugging. The debugging experience using `-Od` is substantially more productive compared to debugging at `-O2`. However, please be aware that debugging at `-Od` still presents certain challenges.

C/C++ language support

SNC-C offers several modes for dealing with the differences between traditional C and ANSI C. The default mode is ANSI compatible with some relaxed requirements. Other modes offer support for traditional C. See "Language definitions" for details.

SNC-C++ offers several modes for dealing with the differences between cfront-like C++ and ARM (*The Annotated C++ Reference Manual*) or ANSI C++. The default mode is ANSI compatible with a number of extensions. Other modes offer support for cfront-like C++. See "Language definitions" for details.

The SNC compiler and the optimizing preprocessors each require that certain restrictions on programming usage (as specified in the ANSI/ISO standards) be met in order to apply full optimization.

Some programs contain latent violations of these ANSI restrictions. Such programs may fail when high degrees of optimization are applied. A systematic process of fixing or working around any such violations may then be necessary to get the best program performance.

The SNC compiler uses the standard C calling sequence, so compiled modules from other compilers may be intermixed and linked.

Intrinsics vs inline asm

We prefer an intrinsics-based approach for low-level code so this release of the compiler does not provide support for GNU-style inline asm. Intrinsics have a higher level of integration with the compiler's optimizer and should enable it to produce better code than is often the case with mixed C/C++ and inline asm. If you have code which you feel cannot be implemented with the available set of intrinsics please let us know.

We do support 'raw' asm that is passed directly to the assembler without any further intervention by the compiler.

The intrinsics are documented in "Intrinsic function reference".

Source file encoding support

SNC supports ASCII and UTF-8 encoded source files. SNC automatically detects UTF-8 encoded files using the byte-order-mark (BOM) present at the start of the file.

On English-language versions of Windows, any file that does not have a BOM is assumed to be ASCII encoded, whereas on non-English-language versions of Windows any file that does not have a BOM is assumed to be UTF-8 encoded.

The compilation system

The compilation system can be used in a number of different ways to prepare a program such as this for execution, for example:

(1) The UNIX style utility *make* may be used to invoke the various build tools to produce an executable program.

(2) The SNC integration for Visual Studio .NET may be used to manage a project and then invoke the required build tools to produce an executable program.

The SNC compiler is part of a compilation system having several components that are used to compile and execute source programs. This collection of tools we term the *build tools*.

The build tool components are:

SN compiler driver	This program accepts and checks command line parameters and executes the relevant build tools components.
SNC C/C++ compiler	This program accepts C/C++ source files and compiles them to produce assembly files, which express the compiled program in symbolic machine language form. This compiler includes a preprocessor for the C/C++ preprocessing language.
SN assembler	This program takes assembly files and assembles them to produce object files. Object files express the compiled program in a binary machine language form.
SN linker	This program takes a number of object and library files and produces an executable program in the target format (ELF).
SN archive librarian (SNARL)	This utility is used to create and manage libraries of object files.

Normally one or more source files comprise a program. These source files can be written in C, C++ or in assembly language. In addition, each source file can contain one or more functions.

The root component of the compiler is called the *driver*. This is invoked by the command line that starts the system, and when invoked runs as a single process. The driver looks at the set of options passed to it, as well as the extension of each filename it is passed. These files and options direct the behavior of the driver as it invokes and/or directs the behavior of the remainder of the system. In general you should not call the compiler components directly, instead the driver should be used.

The components of the compilation system communicate with each other by writing and reading temporary files. For example, when a high-level language source file is compiled, the resulting assembly output is placed in a temporary file, and this is then processed by the assembler. By default, all temporary files are placed in the Windows temporary directory as specified by the environment variables `TMP`, `TEMP`, or `USERPROFILE`, checked in that order. If no valid path is found after checking these environment variables, the Windows directory is used.

Compiler driver usage scenarios

"Command-line syntax" contains a detailed discussion of the command line used to invoke the compilation system. The following examples are given as an introduction to that discussion.

Scenario 1

For the first scenario, the driver is passed a mixture of C, C++ and assembly source files with no command line options (thus invoking the default behavior of the driver). Under these circumstances the driver will:

- pass each of the given C source files to the compiler for C preprocessing and compilation
- pass each of the given C++ source files to the compiler for C++ preprocessing and compilation
- pass all of the resultant assembly files and all of the given assembly source files to the assembler for assembly
- pass all of the resultant relocatable object files to the linker to produce a single combined executable object file

Scenario 2

As a second scenario, the driver is passed a single C source file and is also given the `-c` command line option (the `-c` option directs the driver to stop prior to calling the linker). Under these circumstances the driver will:

- pass the given C source file to the C compiler for compilation
- pass the resultant assembly file to the assembler for assembly, leaving the resultant relocatable object file in the current directory

This is commonly used in makefiles to cause recompilation when a source file or any of its antecedents has changed.

Scenario 3

As a third scenario, suppose the driver is passed a set of relocatable object files and no options. Under these circumstances the driver will:

- pass the given files to the linker, which will bind them together to produce a single executable object ELF file.

This is commonly used in makefiles to create the executable object file when any of the relocatable object files have changed.

> **Tip:** if the desired final output is not an executable file, or for greater control when linking you should call the linker directly, rather than using the Compiler driver. For details about using the linker directly, see "User Guide to ProDG Linker for PlayStation®3"

Control of compiler behavior

The SNC compiler gives you much flexibility in controlling its behavior during compilation. Some of the aspects of compiler behavior that you can control are:

- The sort of progress that is made towards the production of an executable object file (as discussed above).
- The sorts of optimization that are applied by the compiler, and the extent to which they are applied.
- The dialect of the source language that was used in the source program, and/or the file format that was used to encode the source program.
- The sorts of listings, diagnostics, symbolic debugging information, or other output that the compiler should produce.
- The sorts of resource utilization limits that are to be placed on the compiler.

For some aspects of the control of compiler behavior, there is an established tradition that dictates the details of the control mechanism. This applies particularly to the organization of the compiler into the kind of compilation system discussed above and to the options that are used to direct the compilation system activity (such as the -c option, which stops activity before invoking the linker).

For other aspects of the control of the compiler's behavior, there is no established tradition, and mechanisms specific to the SNC compiler are used.

Control of the compiler's behavior can be exercised in two different places:

- On the command line, with command line options and with the nature of the files passed to the compilers.
- Within the source files, by way of a suitable construct that is an extension of the source language. For the SNC compiler, this construct takes the form of pragma directives.

Some aspects of the compiler's behavior need only be controlled at the level of the command line. For these aspects, there are suitable command line options. Most aspects of the compiler's behavior, however, are appropriately controlled from either the command line or the source file, depending on the circumstances. Consider, for example, the degree of optimization to be applied. This will usually be controlled most conveniently from the command line. However, if the nature of a certain function required that a certain sort of optimization be disabled for that function, it would be more convenient to put the disabling directive with the function. There are other examples with exactly the opposite properties (i.e. it is usually more convenient to place them in the source program, but sometimes better to place them on the command line).

To satisfy this dual usage need, a control scheme is used that permits control to be exercised on either the command line or in the source file, strictly at your discretion. The basic idea is that of a set of *control-variables*.

For each controllable aspect of the compilers' behavior, there is a control-variable, and the value currently assigned to this variable dictates the compiler's behavior. These variables can be given initial values on the command line, and their values can be changed at any point in a source file by a suitable pragma directive.

See "Controlling the compiler" for further information.

Differences between SNC and GCC

Please see the SNC Migration Guide for more information and advice on porting code from GCC to SNC.

C and C++ link compatibility

SNC uses the same C++ ABI as GCC (based on the IA64 C++ ABI). C and C++ code built by SNC and GCC may be mixed freely.

C++ class layout compatibility and vtable pointer placement

SNC uses the same C++ ABI as GCC (based on the IA64 C++ ABI). The layouts of classes and vtables produced by the two compilers are identical.

SDK include paths

The default include paths used by SNC are listed below. These are set by the compiler driver but can be suppressed using the -nostdinc option. Where relative paths are used, these are relative to the compiler executables. Because of this, it is essential that the toolchain is installed in the SDK. When correctly installed, the tool executables should be in the "{cell sdk}\host-win32\sn\bin" directory.

- ../../../host-win32/sn/ppu/include,
- ../../../host-win32/sn/ppu/include/sn,
- ../../../host-win32/sn/common/include,
- ../../../host-win32/sn/common/include/sn,
- ../../../target/ppu/include,
- ../../../target/common/include,
- $(SN_PS3_PATH)/ppu/include,
- $(SN_PS3_PATH)/ppu/include/sn,
- $(SN_PS3_PATH)/common/include,
- $(SN_PS3_PATH)/common/include/sn,

The SN_PS3_PATH environment variable is set by the installers for other SN products. The include directories referencing this variable are used to locate additional header files that may be shipped with these products.

Linking and libraries

Applications compiled with SNC should link with the SN runtime libraries as well as the GCC specific libraries supplied with the SDK.

The default linker script **$(CELL_SDK)/target/ppu/lib/elf64_lv2_prx.sn** now contains default settings for library search paths, libraries and other run-time components that must be linked into typical applications. There are separate settings for linking with or without C++ exceptions support. The default is to use the no-exceptions versions of the libraries and runtime. To select the exception handling versions, pass the `-exceptions` switch to the linker.

The defaults are specified in the following sections of the link script:

- LIB_SEARCH_PATHS - library search paths
- STANDARD_LIBRARIES - standard libraries
- REQUIRED_FILES - essential C and C++ runtime components

The default settings should be sufficient for the majority of applications linking against the SDK. Additional libraries and search paths can be added on the linker command line.

The SNC runtime support libraries are:

- libsn.a (utility and debug support)
- libsnc.a (C runtime support)

2: C++11 Support

This document describes the core language features that are supported by the SNC Compiler, with some examples of usage. It is not intended to be a reference manual for the C++11 features.

Scope

Language usability improvements:

- auto types. See "Type inference".
- constexpr. See "constexpr".
- decltype. See "Type inference".
- Delegating constructors. See "Delegating constructors".
- Inheriting constructors. See "Inheriting constructors".
- Inline namespace. See "Inline namespace".
- Lambdas and closures. See "Lambda expressions".
- nullptr. See "Null pointer constant".
- Opaque enumeration declarations. See "Opaque enumeration declarations".
- overrides and final specifiers. See "override and final specifiers".
- R-value references. See "Applying the reference qualifiers to *this*".
- Scoped enumerations. See "Scoped enumerations".
- Suffix return type syntax. See "Trailing return types".
- Uniform initialization. See "Uniform Initialization".

Language functionality improvements:

- alignas and alignof. See "alignas and alignof".
- Attribute (carries_dependency). See "Attribute "carries_dependency"".
- Attribute (noreturn). See "Attribute "noreturn"".
- noexcept specifier and operator. See "noexcept specifier and operator".
- Range-based for loops. See "Range-based for loops".
- Raw string literals. See "Raw string literals".
- static_assert . See "Static assertions.
- User defined literals. See "User defined literals".
- UTF8 string literals. See "UTF-8 string literals".
- Variadic templates. See "Variadic templates".

Miscellaneous improvements:

- Default template arguments for function templates
- Local and unnamed types as template arguments
- SFINAE Rules (Substitution Failure Is Not An Error)

Significant limitation

Note the following significant limitation:

- There is no C++11 library support. For example the compiler supports Rvalue references, but the runtime library does not provide the corresponding support.

Using C++11 mode

By default the compiler is in C++03 mode as in earlier releases. C++11 mode is enabled by a compiler switch `-Xstd` with these values:

- `-Xstd=cpp03`

  This is the current language mode and is the default.
- `-Xstd=cpp11`

  This is the C++11 mode

For consistency, `-Xstd` can also be used to specify the C dialect as follows:

- `-Xstd=c89 j`

  C language standard 1989
- `-Xstd=c99`

  C99 standard. This is the default

  For more information on specifying the language standard, see "std: C/C++ Language Standard".

Compile time performance improvements

The following compile time performance enhancing features are supported by the SNC Compiler.

Explicit instantiation declarations (extern template)

C++11 allows the programmer to declare an explicit instantiation without compelling the compiler to actually instantiate it, thereby saving compile time.

```
template <typename T>
inline T foo(T t)
{
      ...
}

// do not instantiate foo in this translation unit
extern template int foo<int>(int);
```

Language usability improvements

The following language usability features are supported by the SNC Compiler.

Alias templates

C++11 introduces template aliases, which are a way to introduce names for families of template types.

```
template<class T> struct Alloc { /* ... */ };

// Vec is a template alias
template<class T> using Vec = vector<T, Alloc<T>>;

// same as vector<int, Alloc<int>> v;
Vec<int> v;
```

Applying the reference qualifiers to *this*

C++11 allows the declaration of member functions that will operate only on lvalue or rvalue objects.

```
extern "C" int printf(const char *, ...);
struct A {
  void p() & { printf("&\n"); }
  void p() && { printf("&&\n"); }
```

```
};

int main() {
  A a;
  a.p();     // &
  A().p();  // &&
}
```

Braced initializers as default arguments

SNC supports the use of braced initializers in function default arguments.

```
struct S
{
  int i;
  float f;
};


S foo(S s = {1, 3.14})
{
  return s;
}
```

constexpr

C++11 supports generalized constant expressions by introducing the `constexpr` specifier. When applied to the definition of a variable or the declaration of a function, function template or static data member, the declared entity is designated to represent a value that is evaluated at compile time. It can also be used with constructors.

This feature allows the usage of expressions in contexts where they were previously not accepted:

```
constexpr int func() {return 4;}
float my_array[func()*20];

class A {
public:
  constexpr A():m(4){}
  constexpr int mem(){ return m; }
private:
  int m;
};

int my_array2[A().mem()];
```

Delegating constructors

C++11 adds the ability for constructors to call other constructors and effectively delegate part of the initialization of the object to them.

```
class A {
private:
  int mem;
public:
  A(int i) : mem(i) {}
  A() : A(7) {}
};
```

Inheriting constructors

C++11 lets classes inherit base class constructors with using declarations.

```
class Base {
public:
  Base(int m);
};

class Derived : public Base {
public:
  using Base::Base;
};

Derived d(4);
```

Inline namespace

SNC supports inline namespaces. Declaring a namespace to be inline makes its members available in the enclosing namespace.

```
namespace outer
{
  inline namespace inner
  {
    void foo();
  }

void bar()
{
  foo();
}
} // end namespace outer
```

Lambda expressions

Lambda expressions provide a concise way to create simple function objects. The syntax for lambda expressions is as follows:

[*lambda-capture*$_{opt}$ **]** **(** *parameter-declaration-clause* **)** **mutable**$_{opt}$
exception-specification$_{opt}$ *attribute-specifier-seq*$_{opt}$ *trailing-return-type*$_{opt}$
compound-statement

```
#include <algorithm>
#include <cmath>

void abssort(float *x, unsigned N) {
  std::sort(x, x + N,
    [](float a, float b) {
      return std::abs(a) < std::abs(b);
    });
}
```

Lambda expressions, when evaluated, yield a "closure object". Since the type of the closure object cannot be named, the "auto" specifier must be used if such an object is declared.

```
auto x1 = [](int i){ return i; };
…
int j = x1(4);
```

Lambdas have access to the variables of their enclosing block scopes, if their smallest enclosing scope is actually a block scope.

At the time of evaluation of the lambda expression the variables that the lambda expression requires to have access to are said to be *captured* in the lambda expression's *lambda capture.*

The lambda expression determines which variables it needs to capture and how it achieves the capture, for example by copy or by reference.

```
void foo(int i, int j, float f)
{
    auto m = [i, &j, &f]() {
        std::cout << I;
        j = 3; f = 3.14;
    }
}
```

Null pointer constant

The `nullptr` keyword is new in C++11. It represents a pointer literal of type `std::nullptr_t`.

```
void foo (void) {
    char *np = nullptr;

    // Compare pointers against 0 or nullptr
    if (np == 0)
        bar();
    if (np == nullptr)
        baz();
}
```

Opaque enumeration declarations

SNC supports opaque or "forward" enumerations:

```
enum E1: int;
enum class E2: short;

... // code using E1 and E2

enum E1: int {a, b, c};
enum class E2: short {a, b, c};
```

override and final specifiers

SNC supports the specifiers `override` and `final`. They can be added at the end of a virtual function declaration.

Adding `override` to a virtual function declaration will cause the compiler to issue an error message if the function does not override at least one base class member.

Adding `final` to a virtual function declaration will cause an error message to be issued if the function is overridden anywhere.

```
class A
{
public:
  virtual int foo() final;
  virtual void bar(int i);
};

class B: public A {
  virtual int foo();   // error: foo() cannot be overridden
  virtual void bar(float f) override;
                       // error: B::bar() is supposed to override
                       // A::bar() but the signatures don't match
};
```

Scoped enumerations

C++11 introduces enumerations whose enumerators are only visible in the scope of the enum declaration itself. Using them requires qualification.

```
enum class A {red, blue, green}; // "enum struct" would also be correct
…
A a = A::blue;
```

Sized enumerations

C++11 allows the programmer to specify a size for the enumeration type.

```
// the size of the type is the same size as unsigned short
enum B: unsigned short {Value0 = 10, Value1 = 256, Value2 = 45789};
```

Trailing return types

C++11 allows a function return type to be specified at the end of a function declarator instead of at the beginning.

```
auto foo(int i)->int;

template <typename T>
class A {
    typedef float Local_type;
    Local_type bar(T t);
};

template <typename T>

// no need to qualify Local_type
auto A::bar(T t)->Local_type;
```

Type inference

auto

The `auto` keyword has been introduced in C++11 as a type specifier. Its effect is that the type of the declared entity is inferred from its initializer, that is `auto` is a placeholder for a to-be-deduced type.

SNC supports this function with the exception of the usage of braced initializer lists (because the header file <initializer_list> is not provided).

```
auto intvar = 4;          // int
auto& refvar = intvar;  // reference to int
auto doublevar = 3.0;   // double
auto *pvar = &intvar;   // int *

template <typename T>
void foo(T t)
{
    auto lvar = t;
    …
}

// unsupported
auto arr = {1,2,3,4};
```

decltype

Using the `decltype` keyword is a new way to specify a type. The type denoted by a `decltype` specifier is, for practical purposes, the type of its argument.

```
int i;
```

```
struct A { double x; };
const A* a = new A();
decltype(i) x2;         // type is int
decltype(a->x) x3;      // type is double
```

Uniform initialization

SNC supports the use of braced initializers to the full extent of the C++11 standard, allowing braced initializers when either function-style initialization (with parentheses), or assignment-style initialization is allowed, for example:

```
int i{4};
float f[] {2.0, 3.14};

struct S {
  int i;
  float f;
};

S s {2, 3.14};

class A
{
  public:
    A(int i){ ... }
};

A a{5};
```

As an aside, uniform initialization removes the ambiguity between the declaration of an object and the declaration of a function prototype, for example:

```
class A;

void foo()
{
  int bar(A());
}
```

bar is interpreted to be a function, taking as parameter a function returning A, rather than as a variable of type int initialized with a temporary of type A, converted to an int. The new equivalent using uniform initialization:

```
int bar{A{}};
```

is unambiguously a variable.

Language functionality improvements

The following language functionality features are supported by the SNC Compiler.

alignas and alignof

C++11 introduces the `alignof` expression, which takes a type as an argument and returns its alignment.

```
class A {
  ...
};
std::size_t al = alignof(A);
```

Also available is the `alignas` qualifier, which can be added to a declaration. The qualifier takes an expression and instructs the compiler to align the declared variable at the argument expression type's alignment.

```
alignas(var) unsigned char myarray[N]; // align at the alignment of var
```

The alignas qualifier can also take a type (T), in which case it is equivalent to:

```
alignas(alignof(T))
```

Attribute "carries_dependency"

C++11 defines the carries_dependency attribute. SNC accepts this attribute, but ignores it.

```
struct foo { int* a; int* b; };
[[carries_dependency]] struct foo* f(int i);
```

Attribute "noreturn"

C++11 allows specification of the attribute `noreturn` with function declarations and definitions. It tells the function's caller that the function will not return to the point of call.

```
[[noreturn]] void foo(int i) { exit(1); }
```

Explicitly defaulted functions

In C++11 special member functions, such as default constructors, copy constructors, move constructors, copy assignment operators, move assignment operators, and destructors can be declared as "defaulted", that is the programmer intends to use the compiler-generated version of these functions.

A subsequent user-provided definition of such a member function is rejected.

```
class A {
public:
    A(const A&) = default;          // defaulted copy constructor
    A(A&&) = default;
    A& operator=(A&&) = default;
};
```

Explicitly deleted functions

Any function can be declared as "deleted", so that any reference to it is rejected by the compiler. This can be used to prevent unexpected copying or type conversions by declaring the copy constructor and/or conversion operator as deleted.

```
struct A {
    A(A&) = delete;                  // copying
    A(std::intmax_t) = delete;     // conversion, integer initialization
    A(double);                       // allow double initialization
};
```

noexcept specifier and operator

SNC supports the noexcept operator, which determines whether the evaluation of its operand can throw an exception. The operand is not evaluated.

```
void foo();

if (noexcept(foo())
  ...
```

The noexcept specifier takes a boolean constant expression and is used in function declarations. If the constant expression is *true,* the function is declared to not throw any exceptions.

```
void bar() noexcept(true);   // bar does not throw exceptions

template <class T>
  void foo() noexcept(noexcept(T())) {}
// declares a template function foo that can throw exceptions
// depending on whether the template type's constructor does
```

Range-based for loops

SNC supports the construct known as range-based for loops, a similar concept to *foreach* in other languages.

```
int my_array[5] = {1, 2, 3, 4, 5};
for (int &x : my_array) {
    x *= 2;
}
```

In addition to arrays, any class that supports begin() and end() member functions that return iterators, can be used as a range specifier. The container classes in the C++ standard library meet this requirement specifically.

```
#include <vector>

std::vector<int> vg;
int sum;

void foo()
{
  for (int x : vg)
  {
    sum += x;
  }
}
```

Raw string literals

C++11 allows string literals without needing to escape any special characters:

```
  R"(Special ? string \ " stuff)"
  R"delimiter(My special string ) ")delimiter"
```

In the first case, the characters between "(and)" are the contents of the string, in the second case the contents are the characters between delimiter(and)delimiter. Note that the second form allows the use of the character ')' as part of the string.

Right angle bracket

C++11 recognizes ">>" as 2 separate right angle brackets, instead of the right shift operator, in template declarations.

It makes it possible to write:

```
template <typename T> class MyType;
template<typename T> class SomeType;

// syntax error in C++03
MyType<SomeType<int>> *p;
```

Static assertions

C++11 introduces the `static_assert` keyword, which is syntactically treated as a declaration.

The first parameter is a boolean compile-time expression and the second parameter a string. If the first parameter evaluates to false, a diagnostic message containing the string parameter is issued.

```
static_assert(sizeof(long) >= 8, "long is required to be >= 64-bits");
```

User defined literals

C++11 allows users to specify their own literal suffixes and thereby write literals of complex types. The name of the suffix has to start with '_' (underscore).

```
class A;
```

```
A& operator "" _Aconstant(unsigned long long);

void foo()
{
  A& a = 4_Aconstant;
}
```

Preferably the type in question has `constexpr` constructors so that the literal can be generated at compile time. Note that the parameter type for the literal processing function has to be `unsigned long long` for integer literals.

```
class A
{
  int m;
public:
  constexpr A(unsigned long long ll):m(i){}
  constexpr A(const A& other):m(other.m) {}
};

constexpr A operator "" _Aconstant(unsigned long long ll)
{
  return A(ll);
}

void foo()
{
  A a = 4_Aconstant; //constant generated at compile time
}
```

Other available parameter types for literal processing functions are `long double` for floating point literals and `const char *` for strings.

```
A operator "" _doublesuffix(long double);
A operator "" _stringsuffix(const char *);

A a1 = 3.14159_doublesuffix;
A a2 = 123_stringsuffix;        // "123" is passed to the processing function
```

UTF-8 string literals

C++11 introduces string literals with UTF-8 encoding.

```
const char *s = u8"This UTF-8 string cost me 3000\u00A5."; // ¥
```

Variadic templates

C++11 allows the specification of templates with a variable number of parameters.

The parameter that accepts a variable number of arguments is referred to as a *parameter pack*. The pack can be used in various contexts within the template class, or template function declaration or definition.

```
template<class ... Types> void f(Types ... rest);
template<class ... Types> void g(Types ... rest) {
    f(&rest ...);
}

int a; float f;

// call f(&pi, &pf) with pi being g's first and pf being g's second parameter
g(a, f);
```

An example from the C++11 standard:

```
template<typename...> struct Tuple {};
```

```
template<typename T1, typename T2> struct Pair {};

template<class ... Args1> struct zip {
    template<class ... Args2> struct with {
        typedef Tuple<Pair<Args1, Args2> ... > type;
    };
};

// T1 is Tuple<Pair<short, unsigned short>, Pair<int, unsigned>>
typedef zip<short, int>::with<unsigned short, unsigned>::type T1;
```

3: Command-line syntax

The compiler driver command-line syntax is as follows:

```
ps3ppusnc [options] [files]
```

where [options] is a list of options and [files] is a list of filenames. See "Compiler driver options" for a list of compiler driver options. See "Filenames" for a discussion of the treatment of filenames.

Compiler driver options

The default compiler behavior can be modified by the use of options, which precede the filename list. The options described below can be given; any other options given are ignored by the compiler and passed through to the linker if it is invoked.

The following tables group the various compiler options according to type:

Help

Options	Actions
`[none]`	Typing the program name with no arguments causes a print of general usage information, including the main compiler options.

Pre-compiled headers

Options	Actions
`--pch`	Automatically use and/or create a pre-compiled header file. See "Pre-compiled headers" for further details. If --use_pch or --create_pch (manual pre-compiled header mode), appears on the command line following this option, its effect is erased.
`--create_pch= filename`	If other conditions are satisfied, create a pre-compiled header file with the specified name. If --pch (automatic pre-compiled header mode) or --use_pch appears on the command line following this option, its effect is erased.
`--use_pch=filename`	Use a pre-compiled header file of the specified name as part of the current compilation. If --pch (automatic pre-compiled header mode) or --create_pch appears on the command line following this option its effect is erased.
`--pch_dir= directory-name`	The directory in which to search for and/or create a pre-compiled header file. May be used with automatic (--pch) or manual (--create_pch or --use_pch) pre-compiled header mode.
`--pch_messages` `--no_pch_messages`	Enable or disable the display of a message indicating that a pre-compiled header file was created or used in the current compilation.
`--pch_verbose`	In automatic pch mode, for each pre-compiled header file that cannot be used for the current compilation, a message is displayed giving the reason that the file cannot be used.

Process control and output

Options	Actions

`-c`	Compile to an object file. If an output file is specified (via the -o option), all output is sent to this file. Otherwise the output file takes the input filename, with a new extension of .o.
`-C`	Retain comments in the C/C++ preprocessor output.
`-dryrun`	Write to stderr the name and arguments for each process that would be invoked during compilation, plus the name of each temporary file that would be unlinked, but do not actually do any compilation.
`-E`	Preprocess only. Write preprocessor output to stdout and stop the compilation. For C/C++ preprocessor output, comments are removed from this result by default (but see -C above), while line number information is included.
`-H`	Write the pathnames of included files to stdout and stop the compilation. Any source files to be preprocessed are passed through the preprocessor, but normal preprocessing result files are not produced. Instead, a list of the pathnames of all files included during the preprocessing is written to stdout. Also see the -M option, below.
`-keeptemp`	Do not remove any temporary files created during compilation.
`-M`	Output a rule, suitable for 'make', which shows the dependencies for each object file. The dependency information is written to stdout.
`-M1`	Same as the -M option, but the dependency information mentions only user header files, and not system header files. System header files are files not in the same directory as the source file, but which may be included without using a `-I` option (that is, header files that are in include directories implicitly known to the compiler, which is the set of directories inside $CELL_SDK).
`-MD`	Output a rule, suitable for 'make', which shows the dependencies for each object file. The dependency information is written to a file with the same name as the input file but with the extension '.d'. Note: If no output file (-o) is specified the dependency information file will be named after the input file, but with the extension '.d'.
`-MF <file>`	When used with -MD or -MMD, it specifies the file name for the dependency file to be created. If not used, the output file has the same name as the input file but with the extension '.d'.
`-MMD`	Same as the -MD option, but the dependency information mentions only user header files, and not system header files. System header files are files not in the same directory as the source file, but which may be included without using a `-I` option (that is, header files that are in include directories implicitly known to the compiler, which is the set of directories inside $CELL_SDK).
`-MP`	When used with -MD, -MMD, -M, or -M1 it causes generation of additional phony targets for dependent header files.
`-o <file>` `-o<file>`	Specify the output filename <file> rather than using the default. This option permits naming the output file to something other than the

	default rules would have generated. Certain restrictions on the extension of <file> are enforced if compilation is stopped before calling the linker. This is to prevent accidental overwriting of the source file, for instance.
`-P`	This option applies only to C/C++ source files. All C/C++ source files are only preprocessed, with the preprocessing result for each file written to a filename that has .i substituted for the filename extension of the source file. Comments are removed from this result by default (see -C above), also line number information is excluded (compare with -E above). The C/C++ compiler is not called on the preprocessed results.
`-S`	Compile to assembler source. Stop the compilation before invoking the assembler and leave all of the assembly source files produced by the compilation in the current directory.
`-td=<path>`	Specifies the path to a temporary directory.
`-Tc`	Specifies that the source file should be treated as a C source file, even if it does not have the normal .c filename extension.
`-Tp`	Specifies that the source file should be treated as a C++ source file, even if it does not have the normal .cpp filename extension.
`-V`	Write the version numbers of each process invoked to stdout.

`-v`
`-verbose`
`-#`
	Verbose mode – print all commands before execution. Write to stderr, the name and arguments for each process that is invoked during compilation, plus the name of each temporary file that is deleted. If using -# in a makefile, the # character must be escaped.

`-Xcprog`	Assign initial values for any number of control-variables, where *cprog* is a control-program. For example: `-Xdiag=2,inline=joe,unroll=8` assigns an initial value of 2 for control-variable diag, an initial value of "joe" for control-variable inline, and an initial value of 8 for control-variable unroll. -X options can be repeated. The full set of -X options, -XX options, and options that are abbreviations for -X options, are processed in left-to-right order, with the rightmost assignment prevailing in the case of duplicate assignments. (Note that -g is an exception to this rule: it is processed first, regardless of its relative position.) Particular care is needed when using control-groups because they contain implicit assignments to a number of control-variables. The initial values assigned in this fashion on the command line are established as the value in effect at the start of each source file processed by the compilation system. The value in effect can be changed for part or all of each source file by pragma directives that occur within the file. "Controlling the compiler" contains a complete description of control-programs. "Control-variable definitions" contains a complete

	description of the meaning of each control-variable. "Control-variable reference" contains quick reference tables for all control-variables.
-XXcprog	Assign non-changeable values for any number of control-variables, where *cprog* is a control-program. This option differs from -X above in that the values assigned by *cprog* do not change (in this compilation) regardless of whether pragma directives or other command-line options are seen. -XX options can be repeated, and can be mixed with -X options. See the left-to-right rule stated above under the -X option.
-Xastimeout=<n seconds>	Configure the compiler timeout <n seconds>, when waiting for the assembler to finish. The default value is 300 seconds. Specify 0 for an infinite timeout.
-Yc,dir	Specify a new pathname, <dir>, for the locations of the processes specified by *c*, where *c* is one or more of the following: p C/C++ preprocessor a assembler l (lowercase "L") linker S directory containing the startup functions. L first default library directory searched by the linker. U second default library directory searched by the linker. If a new pathname is specified for a process that would otherwise not have been invoked, this pathname will be ignored with one exception. In the SNC-C/C++ compiler, the C/C++ preprocessor is not implemented as a separate process; the preprocessing function is incorporated into the C/C++ front-end. However, if a -Yp,*dir* option is given, the driver will use a file named 'cpp' in directory *dir* as the preprocessor.
-##	Like -#, but do not actually do any compilation. In a makefile, the # characters must be escaped.

C/C++ language options

Options	Actions
-K	Accept the Kernighan & Ritchie (K&R) dialect of C. This is an abbreviation for -Xc=knr.
-noex	This is an abbreviation for -Xc-=exceptions.

Warning options

Options	Actions
-w	Disable all warnings. This is an abbreviation for -Xdiag=0. This serves to suppress warnings from preprocessors, but not from the assembler or linker.
-Werror	Treat all warnings as errors. If a warning occurs, the build terminates. This is equivalent to setting -Xquit=1 (see "quit: diagnostic quit level").
--diag_error=<list>	Set diagnostic codes or tag names listed in comma-delimited list <list> to

	be issued as errors.
`--diag_remark=<list>`	Set diagnostic codes or tag names listed in comma-delimited list <list> to be issued as remark-level messages.
`--diag_suppress=<list>`	Do not issue diagnostics for codes or tag names listed in comma-delimited list <list>.
`--diag_warning=<list>`	Set diagnostic codes or tag names listed in comma-delimited list <list> to be issued as warnings.

For an alternative method of controlling diagnostic messages, see "Diagnostic pragmas".

Debugging options

Options	Actions
`-g`	Generate debug information for source-level debugging. Include symbolic debugging information in the assembly files. The `-g` debug option generates symbolic debug information for types, variables, functions, namespaces etc which are used in the program. Unused program elements do not have any debug information generated by default (see `-gfull`). **Warning:** the `-g` option is required if using ProDG Debugger.
`-gfull`	As -g but generates symbolic information for all program elements.
`-Xoml=<n>, where <n> is 0 or 1`	The debugger's OML feature requires that a function consists of at least 3 PowerPC instructions so that it can plant the branches needed for OML to work. This switch forces all functions to have at least three instructions by inserting 'nop' instructions when necessary. `-Xoml=0` (or not specifying `-Xoml` at all) - Does not insert 'nop' instructions. `-Xoml=1` (or -Xoml) - Inserts 'nop' instructions when necessary to make all functions have at least 3 instructions.

Optimization options

Options	Actions
`-On`	Turn on optimization at level *n*, where *n* can be 0 (zero) to 3; also `d` or `s` (see below) This option is an abbreviation for `-XO=n`. See "Optimization group (O)" for the detailed meaning of control-group `O`. If no specification of optimization is given, the result is equivalent to `-O0`.
`-O0 [i.e. zero]`	No optimization and no inlining (except forced inlining).
`-O1`	No optimization, inlining allowed.
`-O2 (or -O)`	Full optimization.
`-O3`	Full optimization. `-O3` will enable more time consuming optimizations .
`-Od`	Debuggable optimized code (no scheduling etc.).
`-Os`	Optimize for both performance and code size, with code size considerations

	weighted more heavily than they are with `-O2`. For example, do less inlining at `-Os` than `-O2`.

Preprocessor options

Options	Actions
`-D<name>`	Define preprocessor symbol *<name>*. This option applies only to source files passed through the C/C++ preprocessor. *<name>* is defined with the value 1. The `-D` option has a lower precedence than the `-U` option, see below.
`-D<name>=<def>`	Define preprocessor symbol *<name>* with value *<def>*. This option applies only to source files passed through the C/C++ preprocessor. *<name>* is defined with value *<def>* exactly as if a corresponding `#define` statement had occurred as the first line of the program. The `-D` option has a lower precedence than the `-U` option, see below.
`-I<dir>` `-I <dir>`	Add this path to the list of directories searched for include files. Any include files found in *<dir>* will be viewed as "user" include files (i.e., "non-system" include files).
`-include <file>`	Include the source code of *<file>* at the beginning of the compilation. This can be used to establish standard macro definitions, etc. The file is searched for in the directories on the include search list.
`-iquote<dir>` `-iquote <dir>`	Add this path to the list of directories searched for include files specified using `#include "file"` and not `#include <file>`.
`-J<dir>` `-J <dir>`	Add this path to the list of directories searched for include files. Any include files found in *<dir>* will be viewed as "system" include files. Note that the generation of dependency information and warning diagnostics is done differently for "user" and "system" include files. See, for example '`-MMD`' and'`-Xnosyswarn`'. It is only necessary to use the `-J` switch when `-nostdinc` is used (and when instead of the "usual" system header file directories, there is another set of directories that you want to be viewed as system header directories).
`-nostdinc`	Suppress all `-I` options the driver would have generated related to the directories containing the "standard" include files. User-specified `-I` options are passed to the compiler as usual.
`-nostdinc++`	Suppress all `-I` options the driver would have generated related to the directories containing the "standard" C++ include files, but only the files related to C++. User-specified `-I` options are passed to the compiler as usual. If `-nostdinc` is specified, this option does nothing.
`-U<name>`	Undefine the symbol *<name>* before preprocessing. This option applies only to source files passed through the C/C++ preprocessor. Any initial definition of *<name>* is removed. Such an initial definition can be created by the `-D` option, or can be one of the symbols that are predefined in a particular environment. A `-U` option overrides a `-D` option for the same name regardless of the order of the options on the command line.

The `-I` option changes the search order used to find files named in the #include statement. For #include statements, this search order is as follows:

(1) For filenames that are absolute pathnames, use only the named file.

(2) For filenames that are not absolute pathnames and that are enclosed in quotation marks, search relative to the following directories, in the listed order:

(a) If control-variable `inclpath` has the value `absolute`, the directory containing the primary source file. If control-variable `inclpath` has the value `relative`, the directory containing the file that contains the #include statement (these two directories differ only for nested #include statements).

(b) The directories listed in any `-I` options, in the order the options occur on the command line.

(3) For filenames that are not absolute pathnames and that are enclosed in angle brackets, search relative to the following directories, in the listed order:

(a) The directories listed in any -I options, in the order the options occur on the command line.

(b) The directory where the SN-supplied include files have been installed.

Linker options

Options	Actions
`-l<library>` [lower case 'L']	Include specified library *<library>* when linking. This option is passed to the linker, where it directs the linker to search a library named *<library>*. Various extensions are applied to *<library>*, depending on whether dynamic or static libraries are to be searched. The `-l` option differs from the other options in that it can be intermixed with the filenames, and the relative placement among the filenames has significance.
`-L<dir>`	Add this path to the list of directories searched for libraries. This option is passed to the linker, where it directs the linker to look in *<dir>* for a library before looking in the standard library directories.
`-nolib` `-nostdlib`	Suppress all `-l` options the driver would otherwise have generated. User-specified `-l` options are passed to the linker as usual.
`-Wl,...`	Specify an option to be passed to the linker. See ProDG Linker documentation for linker command-line syntax. **Note:** options passed to the linker with `-Wl` override any options passed by default by the compiler driver.

Filenames

The compiler can accept any of the following types of file as input and applies the following actions according to the filename extensions:

File Type	Extensions	Actions
C source	.C	Preprocess, Compile, Assemble, Link
C++ source	.CC, .CPP, .CXX	Preprocess, Compile, Assemble, Link
Preprocessed C source	.I	Compile, Assemble, Link
Compiler-sourced assembler	.S	Assemble, Link
Compiler-sourced assembler	.SX	Preprocess, Assemble, Link
User-sourced	.ASM	Preprocess, Assemble

assembler		
Object files	.O, .OBJ	Link only

- Files with extensions that are not recognized as indicating any specific file type are treated as object files and passed only to the linker. This includes .o files, the standard object file extension.
- There is no restriction on how many different extensions can be used; the compiler can compile many C and C++ files in a single invocation and will apply the correct compiler to each.

The actions taken are also subject to control options such as –c that will omit automatic linking.

Examples:

```
ps3ppusnc -c -O2 main.c objects.c pluscode.cpp
```

This preprocesses, compiles and assembles main.c, objects.c and pluscode.cpp to produce three object files, compiled with optimizations and containing no debug information. The files main.c and objects.c are compiled with the C compiler; whereas pluscode.cpp is compiled with the C++ compiler.

The files need not be in the current directory. Absolute or relative path names can be used.

The default assumption of the compilation system is that the passed files together comprise a single program that you would like to prepare for execution. Thus the default behavior is to process all of the passed files appropriately according to their kind, and then to combine the results to yield a single executable object file.

The following specific steps are taken to achieve this:

(1) All high-level language source files are compiled to produce assembly source files, which are placed in a temporary directory. All appropriate source files are preprocessed by the appropriate preprocessor before compilation (this is redundant but harmless for .i files).

(2) All assembly source files, either produced in step 1 or given as input, are assembled to produce object files in the current directory, each of whose names have .o substituted for the previous filename extension. Any previous file with the same name is removed.

(3) All object files, either produced in step 2 or given as input, are passed to the linker, which combines them and links them to produce a single executable object file in the current directory named a.self.

(4) If only one file that was a source file was given to the compilation system, and no errors have occurred, then the .o file created from that source file is deleted. Note: this behavior is not Unix-like.

Compilation restrictions

A few restrictions in the use of command-line options and the like must be followed in order to ensure that files that are compiled separately but eventually linked together will be consistent. Some of these restrictions are enforced by the scope of individual control-variables, but others cannot be enforced by the compiler because they relate to entirely separate invocations of the compilation system.

4: Controlling the compiler

This section explains the different kinds of control that you have over the SNC compiler.

Control-variables

The basic idea of *control-variables* is to control the behavior of the compiler by assigning values to variables. This chapter discusses the concept of control-variables. "Control-variable definitions" contains a detailed discussion of the meanings for all control-variables"Control-variable reference" contains a quick reference table of the values of all control-variables.

There is a control-variable for each of a number of controllable aspects of the compiler's behavior. During compilation, the values currently assigned to these variables govern the compiler's behavior at that point. Control-variables are conceptually similar to variables in programming languages. Specifically, they have the following properties:

- Each control-variable has a unique name. (This name is case-sensitive and is always composed of lowercase letters.)
- The set of control-variable names is fixed (one example is the control-variable named diag). You cannot create new control-variables.
- Each control-variable has a particular type in the sense that there is a certain set of values that may be legally assigned to it.
- Each control-variable has a definite assigned value at all points throughout each C/C++ source file. This value can change at different points within the source file (depending on the occurrence of pragma directives in the source file).
- Each control-variable has a particular scope that governs the range of the source file over which changes to its value have effect.
- Control-variables exist only during compilation; they have no existence at run-time.

The value assigned to a control-variable at any point in a source file is established by the following rules:

- At the start of **each** source file, the value established by command line processing is assigned to the control-variable. The -X and the -XX command line options are used for this purpose.
- If this value was established via the -XX option, it does not change throughout the source file. Otherwise, proceeding sequentially through the source file, if a pragma directive that assigns a value to the control-variable is encountered, the newly assigned value is established until it is changed by another assignment or until the end of the source file is reached.

The notion of scope for a control-variable is similar to the notion of scope for a programming language variable in one important way: it governs the range of the program over which the variable has effect. However, the scope of a control-variable is quite different from the scope of a programming language variable in the way it accomplishes this purpose. Specifically, the notion of scope for a control-variable has the following properties:

- Each control-variable has one of five possible scopes:
  (a) Compilation Scope.
  (b) File Scope.
  (c) Function Scope.
  (d) Loop Scope.
  (e) Line Scope.
- The scope of a control-variable dictates a corresponding set of *scope-points* for the variable, as follows:
  (a) For control-variables with compilation scope, there is one scope-point: at the start of compilation, after the processing of the control-variable assignments on the command line, but before the processing of any source text from any source file.

(b) For control-variables with file scope, there is a scope-point at the start of each file, when the first token of non-pragma non-comment source language text is encountered, i.e., after the processing of any pragma directives that precede this first token of true source language text.

(c) For control-variables with function scope, there is a scope-point at the start of each function, when the first token of text that defines a new function is encountered.

(d) For control-variables with loop scope, there is a scope-point when the first token of an iterative source language statement is encountered. (For C/C++, the for, while and do statements.)

(e) For control-variables with line scope, there is a scope-point at the start of each source line, after the processing of any pragma directive on the preceding line is completed.

- The scope-points for a control-variable are the only points at which the compiler reads the value currently assigned to the control-variable and uses this value to govern the compiler's (future) behavior.

These rules about control-variable assignment and control-variable scope have the following effect:

- Pragma directives assigning values to control-variables may be written at any point in any source file where pragma directives can be written, regardless of the scope of the control value.
- Pragma directives assigning values to control-variables will have the effect of doing the specified assignment and establishing the current value of the control-variable, regardless of the scope of the involved control-variable. The only exception to this is: if a value was established for the control-variable via the -XX option on the command line, the assignment will be ignored.
- The current value established for a control-variable will have no effect on the behavior of the compiler until the next scope-point for that control-variable. At this next scope-point, the currently established value will be read by the compiler and saved for use in governing the behavior of the compiler until the succeeding scope point is encountered.
- A control-value with compilation scope behaves as if it were always set via the -XX option.
- A value assigned to a control-variable with a pragma directive that occurs after the last scope-point in the file for that control-variable will never be applied by the compiler.

Control-groups

There is a rich set of individual control-variables, with each control-variable governing a particular detailed aspect of the compiler's behavior. This arrangement provides flexibility for those circumstances that need it, but it can also place an undue burden on you to set large numbers of control-variable values. A facility for grouping control-variables is provided to ease this burden. See "Control-group reference tables" for control-group reference tables.

Control-groups have the following properties:

- Each control-group has a unique name. (This name is case-sensitive and is always a single uppercase letter.)
- There is a particular set of control-variables that are said to be in the control-group.
- Each control-group has a particular type in the sense that there is a certain set of values that may be legally assigned to it.
- Control-groups do not possess values in the same way that control-variables do. The assignment of a particular value to a control-group is interpreted as an abbreviation for the assignment of some particular set of values to the control-variables in the group.

Control-groups do not have default values. If a control-group is not mentioned, and there is no other assignment to a control-variable in that control-group, then the default value of that control-variable applies.

Control-expressions

Control-variables may be assigned values of any of the following types: integers, names, pairs, or lists of names or pairs. Values of these types are created by the evaluation of control-expressions. Control-expressions may be formed as follows:

- Integer constants written using decimal notation may be used as integer values. For example "1" and "47" are possible integer values.
- Integer expressions may be formed using plus and minus operators. For example, "1+4+8-2" yields "11". Parentheses may not be used, and evaluation is strictly left-to-right. For example, "5-1+3" and "11-1-3" both yield "7".
- Name values may be written using any characters except equal ("="), comma (","), plus ("+"), minus ("-"), and colon (":"). The first character must not be percent ("%") or a numeric digit. Note that this permits names formed by the identifier rules of most high-level languages as well as permitting most filenames or path names. For example, "simple3" and "gorp/foo_bar" are possible name values.
- The pair operator, written using the colon character (":") as a separator, may be used to form pair values. Pair values must have a name as the first part of the value, but may have either a name or an integer as the second part of the value. For example, "a:2", "b:1", and "c:joe" are possible pair values.
- The list addition operator, written using the plus character ("+"), may be used to form list values. Items in the list may be either names, pairs or a mixture. For example "a+b+c" is a list containing three names, while "a:2+b:1+c:joe+d" is a list containing three pairs and one (unpaired) name. If a name is duplicated in a list, the rightmost one prevails. For example, "a:10+b+a:5" yields "b+a:5" and "a:10+b+a" yields "b+a".
- The list subtraction operator, written using the minus character ("-"), may be used to form list values by deleting elements from a list value. For example, "a+b+c-a" yields "b+c". If an item is not in the list, the deletion is ignored, for example "a+b-c" yields "a+b". Parentheses may not be used, and evaluation is strictly left-to-right. For example, "a-a+b" yields "b" and "a-b-c+b" yields "a+b".
- The "value of" operator, written using the percent character ("%"), may be used to extract the current value of any control-variable. For example, "%inline+a" yields the list currently assigned to control-variable inline with the name "a" added to the list.
- The special token "%all" may be used as a name to denote the set of all possible names that are applicable for the control-variable to which it is assigned.

Control-assignments

A value is assigned to a control-variable or a control-group by writing a control-assignment, which is of the form:

```
control-variable=control-expression
```

or

```
control-group=control-expression
```

From the compiler-invocation command line, such a control-assignment may accomplished with the use of the -X switch:

```
-Xcontrol-variable=control-expression
```

or

```
-Xcontrol-group=control-expression
```

To take a simple example, if you wish to assign the value 2 to the control-variable named diag, either of the following forms are permissible:

```
diag=2  diag2
```

Note that from the compiler-invocation command line, the -X switch may be used to specify either of these control-assignments:

```
-Xdiag=2-Xdiag2
```

Similarly, if you wish to assign the value 4 to the control-group named O, either of the following forms are permissible:

```
O=4          O4
```

If the control-expression starts with a name value, the equal sign is required. For example, to assign the name ansi to the control-variable named c, only the following form is permissible:

```
c=ansi
```

Control-programs

A *control-program* is written as a sequence of control-assignments.

Within control-programs, blanks may be inserted as desired, with these restrictions:

- Blanks can not be inserted within names or numbers, and
- blanks may not be used at all in control-programs that appear on the command line, because blank is a separator of command line options.
- Within control-programs, the control-assignments are usually separated by commas. However:
- blanks may be used instead of commas when not on the command line, and
- the commas are optional after a control-assignment that ends with an integer constant.

For example, to assign 3 to diag and to assign joe+pete to inline, any of the following forms are permissible (blank is denoted by "Δ"):

```
diag=3,inline=joe+pete      inline=joe+pete,diag=3
diag=3Δinline=joe+pete      inline=joe+peteΔdiag=3
diag=3inline=joe+pete       inline=joe+peteΔdiag3
diag3inline=joe+pete        inline=joe+pete,diag3
```
etc.

To extend this example, if an assignment of 3 to control-group O was also needed, any of the following forms are permissible:

```
O=3,diag=3,inline=joe+pete  inline=joe+pete,diag=3,O=3
O3diag=3Δinline=joe+pete    inline=joe+pete,diag=3O=3
diag=3O=3inline=joe+pete    inline=joe+pete,O=3diag=3
O3diag3inline=joe+pete      inline=joe+pete,diag3O3
```
etc.

All control-programs are processed left to right, so if duplication of assignment occurs, the rightmost assignment takes precedence. For example, assume alias is assigned a value of 3 by the assignment of 3 to O. Then:

```
O=3,alias=1
```

assigns 1 to alias, while

```
alias=1,O=3
```

assigns 3 to alias.

These rules concerning the permissible forms for writing a control-program are summarized in the following grammar:

control-program:	:=	control-assign-list
control-assign-list:	:=	control-assignment | control-assign-list [separator] control-assignment {Constraint: the separator is optional only when the control-assignment list ends with an integer value.}
separator:	:=	"," | "Δ"{Note: Δ denotes the blank character.}
control-assignment:	:=	control-variable-name ["="] control-expression | control-group-name ["="] control-expression {Constraint: the "=" is optional only when the

		control-expression begins with an integer value.}
control-variable-name:	:=	{One of the control-variable names listed in "Control-variable definitions" and "Control-variable reference".}
control-group-name:	:=	{One of the control-variable names listed in "Control-variable definitions" and "Control-variable reference".}
control-expression:	:=	term |control-expression plus-op term
term:	:=	integer-value | name | pair
integer-value:	:=	digit | integer-value digit
digit:	:=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
name:	:=	name-value | "%all" | "%none" |"%"control-variable-name
name-value:	:=	{any string not containing equal, comma, plus, minus or colon, and not starting with percent or with a numeric digit.}
pair:	:=	name ":" name | name ":" integer-value
plus-op:	:=	"+" | "-"

Attributes

ps3ppusnc supports the following attributes which are based upon those provided for gcc4.1.1.

Function attributes

Additional descriptions for function attributes can be obtained from:

http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Function-Attributes.html

constructor, destructor	Implemented.
dllexport, dllimport	Ignored with a warning.
fastcall	Ignored with a warning but the fastcall functionality is implemented with a different fastcall syntax.
alias("target") , warn_unused_result,unused, sentinel,section,always_inline, noinline,deprecated,naked, format_arg,format	Implemented.
pure,const,malloc	Accepted but silently ignored.
no_instrument_function	Accepted but invalid because the _instrument_functions option is not implemented.
used	Accepted but ignored.
nothrow, noreturn	Accepted but ignored.
weak	Implemented. The weak attribute causes the declaration to be emitted as a weak symbol rather than a global. This is primarily useful in defining

	library functions which can be overridden in user code, though it can also be used in non-function declarations. Weak symbols are supported for ELF targets and also for a.out targets when using the GNU assembler and linker.
`weakref`	Does not work as described in gcc4.1.1 documentation.

Variable attributes

Additional descriptions for variable attributes can be obtained from:

http://gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html

`aligned`	Implemented.
`cleanup`	Silently ignored.
`common`	Not recognized. Ignored with a warning.
`nocommon`	Implemented.
`deprecated`	Implemented.
`mode`	Not yet in use.
`packed`	Implemented.
`section`	Implemented.
`unused`	Implemented.

Type attributes

Additional descriptions for type attributes can be obtained from:

http://gcc.gnu.org/onlinedocs/gcc/Type-Attributes.html

`aligned`	Implemented.
`packed`	Implemented.
`transparent_union`	Implemented.
`unused`	Implemented.
`deprecated`	Implemented.
`may_alias`	Implemented.
`vecreturn`	Implemented. This can be tagged onto the end of a class/struct declaration for handling vectors. When an object of this type is then returned from a function, it is returned in a single vector register rather than via memory and is therefore more efficient. Note that this attribute will only apply to structs/classes containing a single vector element.

Pragma directives

A *pragma directive*, or simply "pragma," is a statement in the source code of the program which is syntactically equal to a comment, but which can communicate information to the compilation system. One use of pragmas in the SNC compiler is to manipulate the values of control-variables and/or control-groups.

Pragma directives can be written in either C or C++.

Syntax

```
#pragma <directive>
```

or

```
_Pragma ("<directive>")
```

The _Pragma form can only be used if C99 or GNU extensions (gnu_ext) mode is enabled. Note that gnu_ext mode is enabled by default. See "-Xc".

Both forms are interchangeable except when you need to use a pragma directive in a macro, in which case you need to use the `_Pragma` form.

> **Tip:** By default, unrecognized pragma directives are ignored. Use -Xdiag=2 to display remarks, or to have them displayed as warnings, use the --diag_warning=162 diagnostic warning.

Library search

```
#pragma comment (lib,"<library>")
```

This pragma places a library search request in the object file, where `<library>` is the name of the library that you want the linker to include. The linker will try multiple file names for a single pragma. For example:

```
#pragma comment (lib,"foo")
```

This is will first search for a library named "`libfoo.a`" in accordance with the `-l` linker option. If it fails to find that library it will then search for "`foo`".

You can use multiple '`lib`' comments in an object file to cause multiple libraries to be automatically included by the linker.

Segment control pragmas

The SNC compiler provides pragmas to control the segment (i.e. section) where certain entities are generated. Note that it is the responsibility of the programmer to ensure extra sections generated in this way are laid out correctly in the program ELF (usually by means of the linker script).

Code segment control

The code segment pragma takes the form:

```
#pragma code_seg ("<segname>")
```

where <segname> is the desired section name. This pragma is declared at the function level and describes the desired segment for all following functions.

Generation into the normal code segment can be specified by leaving the segment name blank, i.e.:

```
#pragma code_seg ("")
```

String segment control

The string segment pragma takes the form:

```
#pragma str_seg ("<segname>")
```

where <segname> is the desired section name. This pragma is declared at the statement level and describes the desired segment for all following constant strings. This is frequently used to separate string constants used for debug code (e.g. asserts etc.) from the normal program string generation.

Generation into the normal string segment can be specified by leaving the segment name blank, i.e.:

```
#pragma str_seg ("")
```

Bit field implementation control

The following pragmas affect the way bit fields are implemented in SNC:

`#pragma ms_struct on`	bit fields may not share a word with a non-bit field (use Microsoft compiler bit field allocation rules)
`#pragma ms_struct off`	turns off Microsoft compiler bit field rules
`#pragma reverse_bitfields on`	turns on reverse bit fields
`#pragma reverse_bitfields off | #pragma reverse_bitfields reset`	turns off reverse bit fields

Normally bit fields are allocated in order left-to-right (most to least significant bit). With reverse bit fields turned on, bit fields are allocated in order right-to-left (least to most significant bit). The default state for #pragma reverse_bitfields and #pragma ms_struct is OFF.

Limitation:

Following the PPU Lv2 GCC compiler implementation, turning on reverse bit fields in SNC is only effective if used in conjunction with #pragma ms_structs, i.e. if reverse_bitfields is on then ms_struct must also be on, and vice versa.

Example:

```
#include <stdio.h>

#pragma ms_struct on
#pragma reverse_bitfields on

union u01 {
  struct s {
    /**
       [reverse bit-field]
        msb                                   lsb
        00000000 00000000 00000000 000 00 00 0
                                 |s4 |s3|s2|s1|
    **/
    unsigned int s1: 1;
    unsigned int s2: 2;
    unsigned int s3: 2;
    unsigned int s4: 3;
  } u1;
  unsigned int i;
};

#pragma ms_struct off
#pragma reverse_bitfields off

union u02 {
  struct ss {
    /**
       [normal bit-field]
        msb                                 lsb
        0  00 00 000 00000000 00000000 00000000
       |s1|s2|s3|s4 |
    **/
    unsigned int s1: 1;
    unsigned int s2: 2;
    unsigned int s3: 2;
    unsigned int s4: 3;
  } u1;
  unsigned int i;
};

int main(void) {
  union u01 uu1; /* reversed */
  union u02 uu2; /* normal */
```

```
  uu1.i = 0;
  uu2.i = 0;

  uu1.u1.s1 = 0x1;
  uu1.u1.s2 = 0x2;
  uu1.u1.s3 = 0x3;
  uu1.u1.s4 = 0x4;
  /**
     [reverse bit-field]
      msb                                    lsb
      00000000 00000000 00000000 100 11 10 1
                                |s4 |s3|s2|s1|
  **/

  printf("%x\n", uu1.i); /* should print 9d */

  uu2.u1.s1 = 0x1;
  uu2.u1.s2 = 0x2;
  uu2.u1.s3 = 0x3;
  uu2.u1.s4 = 0x4;
  /**
     [normal bit-field]
      msb                                  lsb
      1  10 11 100 00000000 00000000 00000000
     |s1|s2|s3|s4 |
  **/

  printf("%x\n", uu2.i); /* should print dc000000 */

  return(0);
}
```

Template instantiation pragmas

Three pragmas aid in control of template instantiations.

```
#pragma instantiate argument
```

The above pragma causes the compiler to instantiate *argument* in this compilation.

```
#pragma do_not_instantiate argument
```

The above pragma causes the compiler to not instantiate *argument* in this compilation.

```
#pragma can_instantiate argument
```

The above pragma tells the compiler that the *argument* may be instantiated in the current translation unit if needed.

In each case *argument* may be a template class name, a member function name, a static data member name, a member function declaration, or a function declaration. When a class name is specified, the directive is applied to all member functions and static data members of the class.

Inline pragmas

Two pragmas can be used for explicit control over inlining. These are:

`#pragma inline`	Forces a function to be inlined wherever it is called
`#pragma noinline`	Ensures compiler never inlines the function.

The pragmas are function-level bound and therefore must precede a function declaration. They only come into effect when automatic inlining is enabled (-Xautoinlinesize > 0).

Diagnostic pragmas

Many of the diagnostic messages that the compiler can produce can also have their category changed via the use of source pragmas. This allows the severity of individual messages to be raised or lowered on a line by line basis. Warnings and remarks can be reassigned any diagnostic level. Only certain errors, called *discretionary errors*, may have their diagnostic level lowered. Discretionary errors are denoted by a '-D' postfix to the error code number in the diagnostic display line. Note that some error codes are only

discretionary in certain contexts. Non-discretionary errors may not have their diagnostic level lowered as this would introduce unworkable changes into the source language or its processing by the compiler.

The diagnostic pragma takes the form:

```
#pragma diag_<category>=<idlist>
```

Where <category> is the desired diagnostic category to set the diagnostic messages to. It may be one of the following:

`suppress`	do not issue diagnostic
`remark`	set diagnostic to issue a remark level message
`warning`	set diagnostic to issue a warning
`error`	set diagnostic to issue an error
`default`	set diagnostic to default category

<idlist> is a comma-separated list of either diagnostic numbers or diagnostic tag names (see the error documentation file help\err_doc.htm for a list of diagnostic tags and their numbers).

You can also temporarily save the state of the entire warning set onto a local stack, and then restore from the stack. The diagnostic stack pragmas take the form:

```
#pragma diag_push           // saves state of entire warning set
#pragma diag_pop            // restores state of entire warning set
```

Using diagnostic pragmas to disable warnings

Use of the diagnostic pragma:

```
#pragma diag_default=942
```

would return the diagnostic level to 'warning' for missing returns from non-void functions.

It is possible to suppress an individual warning for a selected code block using these pragmas:

```
#pragma diag_push
#pragma diag_suppress=942
     <code block>
#pragma diag_pop
```

This will disable warning 942 for the duration of the code block and then restore the previous state of that message.

Using control pragmas to suppress all warnings

A related, but more dramatic, technique would be to suppress all of the compiler's warnings and remarks using the control pragma stack and the diag control:

```
#pragma control %push diag
#pragma control diag=0
     <code block>
#pragma control %pop diag
```

This will suppress all remarks and warnings for the duration of the code block, and then restore the previous state of the diag control-variable at the end of the block. Note that the "control %push" and "control %pop" pragmas apply to the control pragma and hence do not affect the state of the diagnostic pragma.

Control pragmas

The syntax of a control pragma directive is:

```
#pragma control <cprog>
```

where:

- `control` is case-sensitive,

- <cprog> is a control-program, and
- these fields are separated by whitespace. Whitespace is a sequence of one or more space (blank) and/or tab characters.

The compiler issues diagnostics as follows:

- if the pragma token (#pragma) is recognized, but the control token is not present, a remark diagnostic message is issued (see "Diagnostic control-variables").
- if the pragma token (#pragma) is recognized and the control token is present, but the control-program is malformed, an error is issued.

When using control pragmas, the values can be stored and restored from a stack. The syntax for adding a value to the stack is:

```
#pragma control %push <cprog>
```

where <cprog> is a control-program.

The push pragma can be extended to set the new value as well as saving the previous, with the following syntax:

```
#pragma control %push <cprog>=0
```

The syntax for restoring a value from the stack is:

```
#pragma control %pop <cprog>
```

The stack is useful when you wish to momentarily alter a value and then restore it to the previous value.

The following example shows how to disable a divide-by-zero warning for one line of code:

```
#pragma control %push diag=0
        return 1/0;
#pragma control %pop diag
```

Structure-packing pragmas

The following #pragma directives are supported. They change the maximum alignment of members of structures (other than zero-width bitfields), unions, and classes subsequently defined. The n value specifies the new alignment in bytes.

`#pragma pack(n)`	Sets the new alignment.
`#pragma pack()`	Sets the alignment to the one that was in effect when compilation started.
`#pragma pack(push[,n])`	Pushes the current alignment setting on an internal stack and then optionally sets the new alignment.
`#pragma pack(pop)`	Restores the alignment setting to the one saved at the top of the internal stack (and removes that stack entry). **Tip:** `#pragma pack([n])` does not influence this internal stack; thus it is possible to have `#pragma pack(push)` followed by multiple `#pragma pack(n)` instances and finalized by a single `#pragma pack(pop)`.

Example:

```
#pragma pack(push,1)
  struct s1 {
    short  d16; // offset 0
    int    d32; // offset 2
    char   d8;  // offset 6
  } a1;         // next offset 7
#pragma pack(pop)
#pragma pack(push,2)
  struct s2 {
```

```
    short  d16; // offset 0
    int    d32; // offset 2
    char   d8;  // offset 6
  } a2;         // next offset 8
#pragma pack(pop)
#pragma pack(push,4)
  struct s4 {
    short  d16; // offset 0
    int    d32; // offset 4
    char   d8;  // offset 8
  } a4;         // next offset 12
#pragma pack(pop)
void TestPack()
{
  printf("sizeof( a1 ) = %d\n", sizeof( a1 ) ); // 7
  printf("sizeof( a2 ) = %d\n", sizeof( a2 ) ); // 8
  printf("sizeof( a4 ) = %d\n", sizeof( a4 ) ); // 12
}
```

Using predefined macros

The compiler declares a number of predefined macros internally. To obtain a list of these macros and their current values, use the -Xpredefinedmacros control-variable:

Example usage:

```
ps3ppusnc -c test.cpp -Xpredefinedmacros
```

Example output:

```
#define __SIGNED_CHARS__ 1
#define __DATE__ "Feb 18 2009"
#define __TIME__ "14:51:11"
...
```

See "-Xpredefinedmacros".

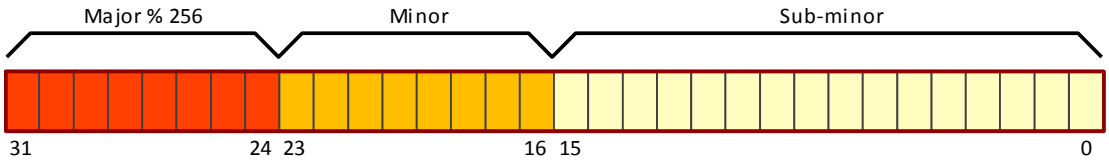
Obtaining the compiler version

Use the __SN_FULL_VER__ predefined macro to determine the version of SNC. This can be used for conditional compilation.

To obtain the version of SNC:

(1) Convert the value of __SN_FULL_VER__ to binary.

(2) Convert sections of the result to integers based on the following diagram:



Warning: The build number is not included in the __SN_FULL_VER__ macro.

Example:

__SN_FULL_VER__	1409425240		
Converted to binary	01010100	00000010	0001111101011000
Converted to integer	84	2	8024
SNC version	340.2.8024		

Testing the value of a control-variable

You can test the compile-time value of a control-variable which takes integer assignments by using the __option pre-processor macro in your code. If an invalid (non-integer) control-variable is specified, the compiler will emit a compilation error.

Syntax:

`__option(`*`control-variable`*`)`

where *control-variable* is the name of a control-variable that takes integer values, minus the '-X' prefix.

Example:

```
#if __option(notocrestore)
... // do code that depends on -Xnotocrestore being non-zero
#else
... // do code that depends on -Xnotocrestore being zero
#endif
```

Limitation

- The __option pre-processor macro is not compatible with pre-compiled headers. See "Pre-compiled headers".

Support for -Xc control-variable options

The __option macro can also take as an argument a single C/C++ language mode as specified by the -Xc control-variable. See "c: C/C++ language modes".

Example:

```
#if __option(rtti)
... // do code that depends on -Xc+=rtti
#else
... // do code that depends on -Xc-=rtti
#endif
```

5: Control-variable definitions

This section defines and explains the meaning of each control-variable. The explanations are grouped by classes of control-variables that have related functions.

"Control-variable reference" contains reference tables that list all the control-variables alphabetically and give the important properties of each one, i.e. the name, abbreviation, scope, type and/or range of values, default value, and a brief (one or two sentence) explanation of the meanings of the various values that can be assigned to the control-variable.

Read this section if you need an *explanation* of what a control-variable does; consult "Control-variable reference" if you need a *reminder* of what a particular control-variable does. Both chapters also cover control-groups.

Optimization control-variables

The control-variables discussed in this section govern the kind and the extent of the optimizations performed by the compiler. A subset of these control-variables forms the variables in the control-group named O.

Introduction to optimization

The SNC compiler is a highly optimizing compiler, which can apply a high degree of optimization to compiled programs. A non-optimizing compiler does a straightforward, "obvious" job of translating the source program to a machine language program, so that the structure of the program (as represented by the computational operations performed and the logical flow of control) remains unchanged during translation. On the other hand, an optimizing compiler:

(1) does a careful analysis of the source program in order to discover various basic properties of the variables and statements of the program, and then

(2) performs a series of transformations called optimizations on the program so that the resulting machine language program runs faster but still produces the same answers. However, the structure of the resulting optimized program may differ very significantly from the structure of the original source program.

These analyses and optimizations are performed in a sequence determined by the design of the compiler. In the SNC compiler, they are intermixed, i.e. some analyses are performed after some optimizations. Furthermore, some analyses and optimizations are repeated because other optimizations open up new opportunities.

When being compiled, a function is divided into units called basic-blocks before analysis or optimization is applied. Specifically, a basic block is a sequence of computational operations which is entered only at the beginning of the sequence and which is exited only at the end of the sequence (where it can transfer to zero, one, or more than one other basic blocks). For the optimizer, the important property of a basic block is that if any of the operations of the block are executed, all of them are executed. Basic blocks are not the same thing as statements in the source program — each basic block may contain only part of a source statement, or a basic block may contain several source statements. Analyses or optimizations that are applied by examining and/or changing each basic block separately are called local analyses or optimizations. Analyses or optimizations that involve more than one basic block are called global analyses or optimizations. Analyses or optimizations that involve more than one function are called interprocedural analyses or optimizations.

alias: alias analysis

Alias analysis is concerned with the issue of deciding whether two memory references in the program may possibly reference the same object at run-time. The results of alias analysis are used at many points throughout the optimizer and affect the results of many different optimizations. To illustrate this point, consider the following two statements:

```
X = 4*A*B/(2*C-D)+E*F
Y = M/(N+O-P)-5*Q
```

If it can be determined that X is a completely different object in memory than any of M, N, O, P, or Q, then the optimizer is free to compile code which starts the evaluation of the second expression before the result is stored into X. Also, if Y is different than any of A, B, C, D, E, or F, then the two statements can be completely interchanged, or if other conditions were met, one might be hoisted out of a loop even if the other could not be hoisted, etc.

If two memory references can be determined to always refer to distinct objects in memory, we say the references are independent. If that determination cannot be made, we say the references possibly interfere. To illustrate the different factors that go into such decisions, consider the following C fragment:

```
float x,y;
union p{float u[10], v[5]};
float a,b,c,d,e,f,g,h;
int i,j;
...
x = a + b;
y = c - d;
...
p.u[5] = e*f;
p.v[j] = g*h;
...
p.u[i] = g/h;
p.u[i+1] = e/f;
```

In this program, it can be seen that:

- the references to x and y are independent by simply examining the declarations of x and y.
- the references to u[5] and v[j] are independent by examining both the declarations of u and v and the fact that the u subscript is the constant value 5 (with the implicit assumption that the value of j does not over index v, which is a restriction applied by the ANSI/ISO C language standard).
- the references to u[i] and u[i+1] are independent if the flow of the program is examined to establish that the two subscripts have distinct values (for example, there cannot be an i=i-1 statement between the two statements).

The alias control-variable is used to govern the degree of alias analysis performed. Specifically:

-Xalias=0	Alias analysis is not performed. The compiler assumes that all memory references possibly interfere with all of the other memory references within the section of code to which the option has been applied. This has a severe impact on optimization, inhibiting most optimizations.
-Xalias=1	Alias analysis based on declarations is performed, i.e. the declarations of the variables used in the references are examined to determine if interference is possible. The majority of cases of independence are detected by this level of analysis.
-Xalias=2	Alias analysis based on declarations and on constant subscripts is performed, i.e. non-pointer array references that have differing constant subscript values in a common subscript position are marked as independent. Analysis at this level is important for numerically based functions, particularly those that use multidimensional arrays in inner loops; it is less important or not useful for other functions.
-Xalias=3	Alias analysis is augmented by use of flow sensitive considerations in subscripts. Analysis at this level is important for numerically based functions, particularly those that use arrays in inner loops; it is less important or not useful for other functions.

The alias control-variable has function scope, accepts values of 0 to 3, and is a member of the O control-group. The default value is alias=0.

debuglocals: improve debugability of local variables when optimizing

When optimizing, variables with automatic storage duration (variables local to a function), are often kept in registers, rather than in memory. See reg: register allocation for more details. There are usually more quantities that could be allocated to registers than the number of available registers. Therefore, the register allocator attempts to re-use registers for different quantities as often as possible. For example, when optimization is enabled for the following code:

```
int* foo(int* p_dst, int* p_src)
{
  int i, j;

  i = p_src[0];          // line 5
  p_dst[0] = i + 10;     // line 6

  j = p_src[1];          // line 8
  p_dst[1] = j - 10;     // line 9

  return p_src;
}                        // line 12
```

the variables 'i' and 'j' will be kept in registers. In this case, the "lifetime" of 'i' goes from line 5 to line 6 (the last use of 'i' is on line 6), and the lifetime of 'j' goes from line 8 to line 9 (the last use of 'j' is on line 9). These distinct lifetimes allow the register allocator to use the same register for both 'i' and 'j'. Note however, that re-using registers will degrade debugging performance when debugging optimized code. For example, after stepping over line 6, the register for 'i' will get re-used, and 'i' will not be displayed in the debugger even though it is still in scope after line 6.

Therefore, to improve optimized debugging, the lifetimes of local variables can be extended to the points where they go out of scope. The lifetime of 'i' will then go from line 5 to line 12, and the lifetime of 'j' will go from line 8 to line 12. The register allocator will then use different registers for 'i' and 'j', and the debugger will be able to display the values of 'i' and 'j' until the end of the function, where they go out of scope.

> **Note:** This improvement in the optimized debugging experience uses more registers and also generates spill-code more frequently. This usually reduces run-time performance by about 2 or 3%.

`-Xdebuglocals=0`	Do not extend the lifetimes of local variables to the points where they go out of scope.
`-Xdebuglocals=1`	Extend the lifetimes of local variables to the points where they go out of scope.

The debuglocals control-variable has function scope, accepts the values of 0 and 1, and is a member of the O control-group. The default value is debuglocals=0.

flow: control flow optimization

Control flow optimization improves the control flow of the program. Unreachable code is eliminated, GOTOs that transfer to GOTOs are collapsed, adjacent basic blocks are merged when possible, and branches are simplified. Usually these optimizations are only applicable after other optimizations on the program have occurred, such as the propagation of a constant into a test condition.

An important disadvantage of control flow optimization is that it changes the control structure of the program so that it may become very difficult to debug the program.

The control flow optimization is governed by the flow control-variable. Specifically:

`-Xflow=0`	Do not do control flow optimization.
`-Xflow=1`	Do control flow optimization.

The flow control-variable has function scope, accepts values of 0 and 1, and is a member of the O control-group. The default value is flow=0.

fltedge: floating point limits

Floating point values may be either numeric or non-numeric (NaN, INF, etc.). Furthermore, floating point computations involving non-numeric values may be "signaling" or may be "quiet", i.e. they may or may not result in the raising of exceptions, as determined by the rules of the IEEE Standard 754 and the rules of the target processor.

Optimization may change the behavior of programs that deal with non-numeric values. For example, if a signaling computation is "dead", i.e. its result is never used, optimization will eliminate the computation and the exception will not get raised. As another example, the IEEE rules require that quiet comparisons involving non-numeric values always yield false, so an option that eliminated the comparison "A is equal to A" would change the result if A contained a non-numeric value. In addition, an optimization that changed "A not greater than B" to "A less than or equal to B" would also change the result if either A or B contained a non-numeric value.

Some control of this situation is offered by the fltedge control-variable. Specifically:

`-Xfltedge=1`	Do no optimization that changes the behavior of the program if non-numeric values occur and are used in quiet computations. (The implementation of this mode is not perfect in the SNC compiler. In some cases, comparisons are modified in a way that changes their behavior. For example, expression (!(a>b)) is changed to (a<=b), which is incorrect if a and b are unordered.)
`-Xfltedge=2`	Do optimizations that may change the behavior of the program if non-numeric values occur and are used in quiet computations, but do not optimize the special case of testing a variable for equality or non-equality to itself. (This mode is provided to permit normal optimization, but also to provide the ability to program a test for non-numeric values).
`-Xfltedge=3`	Do optimizations that may change the behavior of the program if non-numeric values occur and are used in quiet computations.

The fltedge control-variable has function scope and accepts values of 1 to 3. The default value is fltedge=2.

> **Note:** the fltedge control-variable has no effect unless fastmath is enabled using -Xfastmath=1. See "-Xfastmath".

fltfold: floating point constant folding

Constant folding is an optimization that evaluates expressions involving constants during compilation rather than during execution. This optimization may be controlled for floating-point constants by the fltfold control-variable, as follows:

`-Xfltfold=0`	During compilation, do not evaluate expressions involving floating-point constants.
`-Xfltfold=1`	During compilation, evaluate expressions involving floating-point constants and arithmetic operators, but do not evaluate expressions involving intrinsic functions applied to floating point constants.
`-Xfltfold=2`	During compilation, evaluate expressions involving floating-point constants.

The fltfold control-variable has function scope and accepts values of 0 to 2. The default value is fltfold=2.

intedge: integer limits

Some optimizations can only be done if it is known that the values are not near the "edge" of the permissible range of values for the variables involved. For integer variables, this factor is governed by the intedge control-variable. Specifically:

`-Xintedge=0`	Assume that integer overflow can occur during integer operations. Do no

	optimization that would change the program behavior if it does occur.
`-Xintedge=1`	Assume that the effects of integer overflow during integer operations can be ignored in applying optimizations.

The intedge control-variable has function scope and accepts values of 0 and 1. The default value is intedge=0.

notocrestore: eliminate TOC overhead

The PS3 PPU ABI defines and uses a feature called the TOC (Table of Contents) which is used to support the calling of functions. A consequence of the ABI is that function calls must have the following properties:

- A call to a function must have room after the call instruction itself for the linker to patch up the code.
- A call through a pointer to a function must use an intermediate structure: the ".opd" entry. This structure consists of the address of the TOC region used by the target code, and the address of the target code itself.

As the SN compiler does not use the TOC, we can tell the compiler to omit the nop instructions after a function call. We can also tell the compiler to omit the code for loading the TOC for a call through a pointer to function. This is achieved by specifying the -Xnotocrestore control-variable.

Object files built with notocrestore enabled must be linked with SN linker 240.0.2992.0 or later, and with the --notocrestore and --no-multi-toc linker command-line switches specified. It is completely safe to use this option when mixing SNC and GCC compiled code with the single proviso that the total amount of TOC data does not exceed 64 KB, a limit that the linker will rigorously enforce.

Note that there are limitations on the use of the notocrestore control-variable if your program uses indirect calls to PRX functions. See "TOC information" in *User Guide to ProDG Linker for PlayStation 3*.

The notocrestore optimization setting can be changed on a per function basis by using the #pragma control %push option. See "Optimizing on a per-function basis" for more information.

reg: register allocation

Register allocation is concerned with optimizing the use of the fast registers of the target processor. This is important because referencing a quantity from a register takes only a fraction of the time required to reference a quantity from memory. It requires careful attention because there are typically many more quantities that could be usefully placed in registers than there are registers to hold them, and the selection of the best subset of these quantities to actually place in the registers is a very difficult problem.

A few quantities must be allocated to registers, such as the first few arguments when a function is called. Beyond that, there are two kinds of quantities that are candidates for being held in a register:

- Any intermediate value involved in the evaluation of expressions or the execution of the statements of the language. These include all the sub expressions of the evaluated expressions, all the quantities involved in addressing expressions, etc.
- *Register-candidate* variables.
- In C/C++, a variable is a register-candidate variable if it is a scalar, has automatic storage duration, and its address is not taken with the & operator. (The register declaration is ignored by the SNC-C and SNC-C++ compilers.)

Register allocation in the SNC compiler occurs at three levels: interprocedural (between functions), global (within one function), and local (within one basic block). Interprocedural register allocation is done by a bottom-up traversal of the call-graph tree. (Where possible, i.e., where caller-callee relationships are known, callees are processed before callers.) This permits the global and local allocation around a call site to take into account the allocation that has already occurred for the callee. Thus the caller may be able to use registers that would normally be scratch registers according to the calling convention.

Global register allocation is done using a priority-based graph coloring algorithm. A register interference graph is built to guide the allocation algorithm. Local register allocation is done after global register allocation.

See the discussion of the sched control-variable ("sched: scheduling") for a discussion of the relationship between register allocation and scheduling.

Many times there will not be enough registers available to hold all of the candidate values. In this case, spill code will be inserted to move register values to and from memory.

The allocation of registers is governed by the reg control-variable. Specifically:

`-Xreg=0`	Do not allocate register-candidate variables to registers.
`-Xreg=1`	Allocate register-candidate variables to registers, and do global and local register allocation only.
`-Xreg=2`	Allocate register-candidate variables to registers, and do global and local register allocation. Perform more aggressive register optimizations.

The reg control-variable has function scope, accepts values of 0 to 2, and is a member of the O control-group. The default value is reg=0.

sched: scheduling

Most modern RISC processors have some degree of instruction-level parallelism, i.e. the execution of certain instructions overlaps in time with the execution of other nearby instructions. The degree and circumstances of this parallelism vary widely from processor to processor, but they all have the property that some particular choice of ordering of the instructions will run faster than other choices. Scheduling is the optimization that reorders instructions to take advantage of whatever instruction-level parallelism exists on the target machine. This optimization is naturally very dependent on the specified target machine.

A classic dilemma for compilers is whether to do scheduling before or after register allocation. Doing scheduling after register allocation has the effect of constraining the extent of the reorderings the scheduler can perform. Doing scheduling before register allocation increases the register pressure and causes more spill code. To deal with this, the SNC compiler splits up the job. Global register allocation is done first, then one pass of scheduling which takes register pressure into account, then local register allocation, and finally a second pass of scheduling (if needed).

Scheduling is governed by the sched control-variable. Specifically:

`-Xsched=1`	Schedule instructions using pass 1 only.
`-Xsched=2`	Schedule instructions using both passes.

The sched control-variable has function scope, accepts values of 0 to 2, and is a member of the O control-group. The default value is sched=0.

unroll: loop unrolling

The loop unrolling optimization takes certain loops and replicates the code in them several times. This increases the code size, but it also lowers the overhead of testing the loop conditions and it increases the scope for the application of other optimizations, such as scheduling.

Only certain loops can be unrolled.

The loop unrolling optimization is governed by the unroll control-variable. Specifically:

`-Xunroll=0`	Do not unroll loops.
`-Xunroll=1`	Unroll loops under automatic control.

`-Xunroll=n`	where $n > 1$. Always unroll loops (that can be unrolled), and unroll them n times.

The unroll control-variable has loop scope, accepts integer values, and is a member of the O control-group. The default value is unroll=0.

Control-group O: optimization

For a discussion of control-groups, see "Control-groups". The O control-group provides a convenient way to set values for those control-variables which govern optimization.

Six levels of optimization are available with O:

`-XO=0`	No optimization and no inlining (except forced inlining).
`-XO=1`	No optimization, inlining allowed.
`-XO=2`	Full optimization.
`-XO=3`	O=2, plus more time consuming optimizations (none currently).
`-XO=d`	Debuggable optimized code (no scheduling etc.).
`-XO=s`	O=2, but with less inlining.

The values assigned to the member control-variables for each these O values is given in the O control-group table in "Optimization group (O)".

Function inlining: inline, noinline, deflib

The inlining optimization takes the code of a called function and inserts it directly into the code of the calling function in place of the call. This usually increases the overall code size, but it also:

- saves the overhead of a function call and return and the passing of arguments.
- frequently opens up additional optimization opportunities. For example, a function may be coded to deal with several different cases, with a constant value being passed to distinguish the cases. Once such a function is inlined, optimizations such as constant propagation and control flow optimization may be able to reduce the code to be executed. As another example, the call may be inside a loop, and once inlined, it may be possible to move portions of the function outside the loop.

In SNC-C/C++, both intrinsic and user functions may be inlined.The point in the program at which a function is called is termed a *call site*. The decision as to whether to inline the called function is made separately at each call site. In other words, the same function may be inlined at some call sites, and not inlined at others. The factors that influence this decision are as follows:

(1) Factors from the nature of the called function:

  (a) Some intrinsic-functions are always inlined, others are not available to the compiler in an expandable form.

  (b) For a user function, the function source code may or may not be available. (It is only available if it is in the same file as the calling function.)

  (c) The size of the called function.

  (d) The number of places from which this function is called.

(2) Factors from the nature of the call site:

  (a) The call may be an indirect one, via pointers in C/C++ so that it is not possible during compilation to determine the actual function called.

  (b) The loop nesting level of the call site.

  (c) The size of the calling function, including any previously inlined functions.

(3) The value of the inline control-variable at the call site (see "inline").

(4) The value of the noinline control-variable at the call site (see "noinline").

(5) For intrinsic functions, the value of the deflib control-variable at the call site (see "deflib").

inline

The **inline** control-variable accepts a list such that each list member may be either a name or a pair, where the first member of the pair is a name and the second member of the pair is an integer. If the name of the called function occurs in the call site value of the list, it is a candidate for inlining at this point according to automatic rules in the compiler, which use the integer value, if it is given, as a priority for the named function. Larger *n* increase the likelihood that the function will be inlined.

The inline control-variable has line scope and accepts list values as described above. The default value is the empty list.

noinline

The **noinline** control-variable accepts a list of names. If the name of the called function occurs in the call site value of the list, the function is not inlined.

The noinline control-variable has line scope and accepts list values as described above. The default value is the empty list.

deflib

The **deflib** control-variable accepts the following values:

`-Xdeflib=0`	By default, do not inline intrinsic-functions.
`-Xdeflib=1`	By default, inline intrinsic-functions under automatic control.
`-Xdeflib=2`	By default, inline intrinsic-functions whenever possible.

The deflib control-variable has line scope and accepts values of 0 to 2. The default value is deflib=1.

Diagnostic control-variables

Each of the diagnostic messages that the compiler can produce is classified into one of the following categories:

`Remark`	A remark message diagnoses some language usage that the compiler will accept, but that the compiler regards as unconventional usage.
`Warning`	A warning message diagnoses some language usage that the compiler will accept, but that the compiler regards as questionable usage.
`Error`	An error message diagnoses a violation of the syntax or semantics of the language being compiled. Object code will not be produced, but compilation will continue past the point of the error for the purpose of possibly diagnosing additional errors.
`Fatal Error`	A fatal error message diagnoses a problem of such severity that the compilation cannot continue past the point of the error. Object code will not be produced.
`Internal Error`	An internal error message diagnoses a problem with the logic of the compiler itself. Internal errors should be reported to the appropriate support personnel (contact SN Systems). The source code that created the internal error will need to be reported to the support personnel so the message can be reproduced.

The response to the messages is controlled by the diag and the quit control-variables.

diag: diagnostic output level

The level of diagnostic messages output by the compiler is controlled by the value assigned to control-variable diag, as follows:

`-Xdiag=0`	Output only error and fatal error messages. Do not output remark or warning messages.
`-Xdiag=1`	Output only warning, error, and fatal error messages. Do not output remark messages.
`-Xdiag=2`	Output remark, warning, error, and fatal error messages.

These messages are output on stderr. Note that the error and fatal error messages cannot be suppressed.

The diag control-variable has line scope and accepts values of 0 to 2. The default value is diag=1.

> **Tip:** A -w option on the command line is an abbreviation for -Xdiag=0.

diaglimit: limit number of diagnostic messages

SNC by default tends to issue more complete warnings than some other compilers. When initially porting source from these other compilers the number of extra warnings generated can obscure the porting process. This switch allows you to set a maximum number of diagnostics to be issued for each specific diagnostic.

`-Xdiaglimit=n`	Output only the first *n* messages for each diagnostic.

The diaglimit control-variable has file scope and accepts integer values. The default value is 0 (unlimited).

quit: diagnostic quit level

The compiler will exit normally (with an exit status of 0) at the end of compilation if it does not encounter any situation that generates a message. The exit status when diagnostic messages are encountered is controlled by the value assigned to control-variable `quit`, as follows:

`-Xquit=0`	Exit abnormally (exit status=1) if error or fatal error message situations were encountered, exit normally otherwise.
`-Xquit=1`	Exit abnormally (exit status=1) if warning or error or fatal error message situations were encountered, exit normally otherwise. Any messages that would usually be printed as "warning:" are instead printed as "error:".
`-Xquit=2`	Exit abnormally (exit status=1) if remark or warning or error or fatal error message situations were encountered, exit normally otherwise. Any messages that would usually be printed as "warning:" or "remark:" are instead printed as "error:".

Notice that these exits are defined in terms of whether the message situation was encountered, not whether the message was output. The effect of this control is independent of the setting of control-variable diag.

The quit control-variable has compilation scope and accepts values of 0 to 2. The default value is quit=0.

C/C++ compilation

This section describes control-variables that relate to C/C++.

std: C/C++ Language Standard

The `-Xstd` switch allows you to override the default C/C++ dialect values. The default dialect for C++ is cpp03 and the default C dialect is c99.

This switch does not change the interpretation of whether a compilation is in C++ or C; the underlying language selection is determined by the file extension. By default, `foo.c` is compiled as a C source file, and `foo.cpp` is compiled as a C++ source file, but the `-Tc` and `-Tp` switches can be used respectively to force a file to be compiled as C or C++.

When `-Xstd` is used to set the C++ dialect:

-Xstd	file	cpp03 or cpp11	cpp03
cpp03	The C++ dialect is C++03 (the default value).		
cpp11	The C++ dialect is C++11.		

When `-Xstd` is used to set the C dialect:

-Xstd	file	c89 or c99	c99
c89	The C dialect is c89.		
c99	The C dialect is c99 (the default value).		

Note that `-Xstd` allows two separate values to be specified independently; one for the C++ dialect value and one for the C dialect value.

Example:

```
Ps3ppusnc -c -Xstd=cpp11 -Xstd=c89 foo.cpp bar.c
```

The above example will set the C++ dialect and the C dialect to C++11 and c89 respectively, where `foo.cpp` will be compiled in C++11 mode and `bar.c` will be compiled in c89 mode.

c: C/C++ language features

SNC C/C++ has six basic modes, three of which specify and govern the C dialect, while three specify and govern the C++ dialect accepted by the compiler. These modes are controlled by the c control-variable, as follows:

`-Xc=ansi`	In this mode, the compiler complies completely with the ANSI and ISO C standard (ANSI X3.159-1989 and ISO/IEC 9899:1990(E)) as a "conforming hosted implementation", i.e. it supports all of the language and standard header files.
`-Xc=knr`	In this mode, the compiler is largely compatible with the definition of the C language as given in *The C Programming Language* by Kernighan & Ritchie and is closely compatible with the UNIX pcc compiler.
`-Xc=mixed`	In this mode, the compiler is essentially an ANSI compiler, except that a few extensions are added to ease the job of porting existing K&R code to ANSI. See "Language definitions" for a discussion of these extensions.
`-Xc=arm`	In this mode the compiler accepts the C++ language as defined in *The Annotated C++ Reference Manual* by Margaret A. Ellis & Bjarne Stroustrup, and the C++ standard (ISO/IEC 14882:2003).
`-Xc=cp`	Similar to arm, except that it allows for several anachronisms and is less restrictive. Programs that compile under both arm and cp modes will behave identically.
`-Xc=cfront`	In this mode, the compiler accepts the C++ language accepted by AT&T Cfront Compiler and generates compatible object code. This option can take an additional value of either :21 or :30. -Xc=cfront:21 enables compatibility with AT&T Cfront 2.1, while -Xc=cfront:30 enables compatibility with AT&T Cfront 3.0. -Xc=cfront is equivalent to -Xc=cfront:30.

The c control-variable has file scope and accepts name values of ansi, knr, mixed, arm, cp, cfront, c99, const, volatile, signed, noknr, inline, c_func_decl, array_nd, rtti, wchar_t, bool, old_for_init, exceptions, tmplname, gnu_ext and msvc_ext. The default value for the SNC C compiler is c=mixed. The default value for the SNC C++ compiler is c=cp. Note that -K on the command line is an abbreviation for -Xc=knr.

const, volatile and signed

In addition to these seven basic modes, any subset of the three names const, volatile, and signed may be added to the value of control-variable c, forming a list value. When the basic mode is c=knr, the use of any of these names indicates that the corresponding qualifier in the ANSI C language is to be recognized. For example, c=knr+const+volatile indicates K&R compatibility, but with the const and volatile type qualifiers of ANSI C also recognized.

noknr

An additional value, noknr, can be added to the mixed or ansi C modes (for example, -Xc=mixed+noknr). This value causes the compiler to emit warnings on declarations and definitions of any function without a prototype. When noknr mode is enabled, a warning is also emitted when the compiler encounters a use of a function that has not been previously declared or defined.

inline

An additional value, inline, may be given with the C modes ansi, knr and mixed. For these modes the default is off, which tells the compiler to not recognize inline as a keyword. To enable recognition of inline in C programs as a keyword, add inline to control-variable c (e.g. -Xc =mixed+inline). Note that inline is always recognized as a keyword in C++ modes. In c99 mode inline is on by default.

Note the distinction between inline as a value for the c control-variable (described here) and the separate inline control-variable; see "Function inlining: inline, noinline, deflib, inllev, sinllev".

c_func_decl

An additional value, c_func_decl, can be given along with all of the C++ modes. This value relaxes the prototype requirements of the C++ language to those of the C language for functions declared within an extern "C" block. This value is not meant for direct use in user code, but is meant to enable the use of C style system include files in the C++ environment.

array_nd

An additional value, array_nd, can be given with all of the C++ modes. The default is on, which tells the compiler to recognize array new and array delete operators. To disable recognition of array new and array delete, subtract array_nd from control-variable c (e.g. -Xc-=array_nd or -Xc=arm-array_nd).

rtti

An additional value, rtti, can be given with all of the C++ modes. When set to on, the compiler recognizes RTTI (runtime type identification) keywords, thus enabling RTTI behavior. To disable RTTI after it has been enabled, subtract rtti from control-variable c (e.g. -Xc-=rtti or -Xc=cp-rtti). The default setting is "on".

wchar_t

An additional value, wchar_t, may be given with all of the C++ modes. The default is on, which tells the compiler to recognize wchar_t as a keyword, and also to add -D_WCHAR_T and -D__WCHAR_T_IS_KEYWORD as built-in predefined preprocessor macros. The former macro is used in various include files provided by the compiler to ensure that at most one definition of the wchar_t type is seen. The latter macro is provided for you to protect your code which depends on wchar_t being a distinct C++ type, such as when instantiating a template for all built-in types. To disable recognition of wchar_t as a keyword (and distinct type) subtract wchar_t from control-variable c (e.g. -Xc-=wchar_t).

Note the distinction between wchar_t as a value for the c control-variable (described here) and the separate wchart control-variable; see "sizet and wchart: C/C++ type definitions of size_t and wchar_t".

bool

An additional value bool may be given with all of the C++ modes. The default is on, which tells the compiler to recognize bool as a keyword, and also add -D_BOOL_DEFINED and -D__BOOL_IS_KEYWORD as built-in predefined preprocessor macros. The former macro can be used to protect your own definition of bool, perhaps as follows:

```
#ifndef _BOOL_DEFINED
    typedef unsigned char bool;
    #define _BOOL_DEFINED 1
#endif
```

The latter macro is provided for you to protect your code which depends on bool being a distinct C++ built-in type, such as when instantiating a template for all built-in types. To disable recognition of bool as a keyword, subtract bool from control-variable c (e.g. -Xc=cp-bool).

old_for_init

An additional value old_for_init may be given with all of the C++ modes. The default in cfront mode is on, but in cp and arm modes the default is off, which tells the compiler to limit the scope of variables declared in init statements of for loops to the scope of the loop itself (limitation of the scope is required by the ISO/IEC 14882:2003 C++ standard). If your code assumes the larger scope of the variables, and you otherwise want to use cp or arm modes, you will need to add this value to control-variable c (e.g. -Xc=cp+old_for_init).

exceptions

An additional value, exceptions, may be given with all of the C++ modes. The default in all modes is off. When set to on, exceptions tells the compiler to generate all necessary data structures to support the use of C++ exceptions. This protects your code (even if it does not use exceptions) if other code throws an exception across your code. For further discussion of exception handling, see "Exception handling".

tmplname

An additional value, tmplname, may be given with all of the C++ modes. This creates templates with mangled names that are distinct from the names given to non-templated functions.

gnu_ext

An additional value, gnu_ext, may be given with all of the C and C++ modes. This enables the use of GNU GCC extensions to the C/C++ languages (where supported). These include the 'attribute' feature commonly used in legacy GCC code. This switch is on by default.

msvc_ext

An additional value, msvc_ext, may be given with all of the C and C++ modes. This enables use of Microsoft Visual Studio extensions to the C/C++ language (where supported).

char: signedness of plain char in C/C++

ANSI C/C++ has three different character types: char, signed char, and unsigned char. It is clear from the standard that these are three distinct types for purposes such as determining if two expressions have the same type. However, the standard leaves as "implementation-defined" the issue of whether quantities declared as type char are to be implemented with a representation that has a sign bit or not. In SNC-C/C++, this choice is governed by the value of the control-variable char. Specifically:

`-Xchar=signed`	The representation for char is the same as signed char.
`-Xchar=unsigned`	The representation for char is the same as unsigned char.

The char control-variable has file scope and accepts name values of signed or unsigned. The default value is signed.

sizet and wchart: C/C++ type definitions of size_t and wchar_t

There are situations in C and C++ where the compiler must know information about the types size_t or wchar_t, even if they are not defined. Therefore, the compiler has a built-in expectation of the manner in which these types are going to be defined. A mismatch between the compiler's expectation and the definition in a program can cause incorrect behavior.

Normally, these types are defined in one or more system include files. The compiler's built in expectations have been set to match the definitions in standard system include files under the expected environment. However, if for any reason a non-standard set of system include files is being used, the options below can change the compiler's built-in expectations to match the setting in the include files.

The sizet and wchart control-variables can have the following values:

`-Xsizet=uint`	Definition for size_t is unsigned int.
`-Xsizet=ulong`	Definition for size_t is unsigned long.
`-Xsizet=ushort`	Definition for size_t is unsigned short.
`-Xwchart=uint`	Definition for wchar_t is unsigned int.
`-Xwchart=ulong`	Definition for wchar_t is unsigned long.
`-Xwchart=ushort`	Definition for wchar_t is unsigned short.
`-Xwchart=uchar`	Definition for wchar_t is unsigned char.
`-Xwchart=int`	Definition for wchar_t is int.
`-Xwchart=long`	Definition for wchar_t is long.
`-Xwchart=short`	Definition for wchar_t is short.
`-Xwchart=char`	Definition for wchar_t is char.
`-Xwchart=schar`	Definition for wchar_t is signed char.

Warning: sizet is not allowed to have signed types.

Note: the distinction between the control-variable wchart (described here), and wchar_t as a value for the control-variable c; see "c: C/C++ language modes".

Both control-variables have compilation scope, and accept name values as described above. The default values are sizet=uint and wchart=ushort.

inclpath: include file searching

There is a strongly established UNIX tradition for the order in which directories are searched for files named in #include statements, except for one strange case. This case arises when a relative filename is used inside quotation marks in an #include statement that is itself in a file already being included via another #include statement. Recent UNIX implementations usually start this search with the directory containing the file that contains the #include statement being processed. Earlier UNIX implementations started with the directory containing the original (top-level) source file. Also, there is a comment in the ANSI C standard that the standards committee favored "in principle" the earlier approach, but did not actually specify it in the standard.

In SNC-C, this choice is governed by the value of the inclpath control-variable. Specifically:

`-Xinclpath=relative`	In C, while searching for files specified in a #include statement that uses a relative filename delimited with quotation marks, look first in the directory containing the file that contains the #include statement

	being processed.
`-Xinclpath=absolute`	In C, while searching for files specified in a #include statement that uses a relative filename delimited with quotation marks, look first in the directory containing the original (top-level) source file.

The inclpath control-variable has file scope and accepts values of relative or absolute. The default value is inclpath=relative.

C++ compilation

This section describes control-variables that relate specifically to C++.

C++ dialect

The dialect of C++ recognized by the compiler is controlled by the control-variable c, which is described in "c: C/C++ language modes". Also see "C++ language definition".

General code control

This section describes control-variables that relate to general control of your program code.

bss: use of .bss section

Data items that do not require link-time initialization to specific non-zero values may be placed in either the .data or .bss section. Binary program files are smaller if such items are placed in .bss, but compatibility reasons may dictate placement in .data. This is governed by the bss control-variable, as follows:

`-Xbss=0`	All data will be placed in the .data section.
`-Xbss=1`	Static uninitialized data and data initialized to zero may be placed in either the .data section or .bss section according to automatic rules within the compiler.
`-Xbss=2`	Static uninitialized data will be placed in the .bss section; data initialized to zero will be placed in the .bss section where possible.

The bss control-variable has function scope, and accepts values of 0 to 2. The default value is bss=1.

<reg>reserve: reserve machine registers

The SNC compiler allows you to remove individual registers from its allocation pool. This will prevent the compiler from generating code that uses these registers. This leaves them free for use with preset values in asm statements (such as register numbers).

> **Warning:** This feature only covers the current compilation unit. Calling other units or libraries may execute code that uses these reserved registers.

Only 'general' usage registers may be reserved in this way. According to the target architecture, some registers have fixed uses and cannot be reserved in this manner. For example you cannot reserve the GPR associated with the stack pointer as its use is intrinisc to the code generation for the target architecture. Any attempt to reserve a special purpose register will result in an error message of the form:

```
Command line error: Illegal attempt to reserve <regclass> register number <num>
```

where <regclass> is {fpr | gpr} and <num> specifies the register number.

The following control-variables are available:

`-Xfprreserve=list`	Reserve floating point registers defined in *list*.
`-Xgprreserve=list`	Reserve general purpose integer registers defined in *list*.

where *list* is a list of register numbers or register number range pairs. Range pairs use the ':' separator and multiple elements in the list are distinguished by the '+' separator. For example -Xgprreserve=4+21:27 means reserve general purpose registers 4 and 21-27 inclusive.

The <reg>reserve control-variables have file scope. The default value is <empty list> i.e. no registers are reserved.

g: symbolic debugging

Symbolic debugging information may be included in the assembly files produced by the compiler by use of the g control-variable, as follows:

`-Xg=0`	Do not include symbolic debugging information in the assembly files.
`-Xg=1,2`	Include symbolic debugging information in the assembly file for use by the SN symbolic debugger.

The g control-variable has compilation scope and accepts values of 0 to 2. The default value is g=0.

writable_strings: are strings read-only?

Different languages and different targets have different ways of treating strings, particularly in terms of what section of memory strings are placed into, and whether that section is marked writable or read-only. In addition, on hosts with "small data" sections, there may be further interaction with relevant controls governing placement of data into any such "small data" section. Control-variable writable_strings gives you additional control over where strings will be placed.

Like all control-variables, writable_strings can be used on the command line and/or in pragmas inserted into the code. Use on the command line is particularly convenient as a means to make all strings read-only or writable. Use in pragmas permits precise control over each individual string's writability, which allows most strings to be placed in read-only memory (particularly useful on some systems), but some strings can be marked writable to avoid memory faults if the code does modify them.

`-Xwritable_strings=0`	Force strings to be allocated in a read-only data section (usually called something like .rdata).
`-Xwritable_strings=1`	Strings will be placed into a target-dependent data section. Usually this is .data when control-variable c has value knr, otherwise strings are placed in the .string section when it exists on that target, otherwise strings are placed in the .data or .rdata section, based on custom for that target.
`-Xwritable_strings=2`	Force strings to be allocated in a writable data section (usually called something like .data).

The writable_strings control-variable has line scope, and accepts as values 0 to 2. The default value is writable_strings=0.

Miscellaneous controls

This section describes control-variables that provide general control of the compilation system.

mserrors: suppress display of source lines in errors/warnings

The default behavior of the SNC compiler is to print the source line for every error and warning. However in Visual Studio this is not needed and makes the errors harder to read in the Visual Studio task list. The control-variable mserrors may be used to suppress the display of source lines in errors and warnings. Specifically:

`-Xmserrors=0`	Do print source lines for errors and warnings.
`-Xmserrors=1`	Do not print source lines for errors and warnings.

This control-variable is automatically enabled for all SNC compiler builds in Visual Studio, but if you create any custom build steps that call the SNC compiler then you should add this switch by hand.

For example, a warning without the switch would look like this:

```
test.c(11,6): warning: variable "a" was declared but never referenced
    int a =1 ;
        ^
```

But the same warning with the -Xmserrors switch enabled would look like this:

```
test.c(11,6): warning: variable "a" was declared but never referenced
```

progress: status of compilation

You can request additional information about the status of the compilation as well as information about which optimizations are being done. These extra messages are affected by the setting of the progress control-variable.

There are two basic types of these extra messages: the first type involves printing an additional message as a phase of the compiler begins processing a portion of source code; the second type involves providing information about optimizations that have been executed on portions of source code. Specifically:

`-Xprogress=files`	Announce progress at the start of compiling each file.
`-Xprogress=functions`	Announce progress at the start of compiling each function.
`-Xprogress=phases`	Announce progress at the start of each phase of the compiler.
`-Xprogress=subphases`	Announce progress at the start of each subphase of the compiler.
`-Xprogress=actions`	Announce progress at each major action (e.g. inlining) done by the compiler.
`-Xprogress=failures`	Announce the failure of each major action (e.g. failed to inline a function) attempted by the compiler.
`-Xprogress=templates`	Announce instantiations of template functions.
`-Xprogress=memory`	Include compiler memory-usage information in progress announcements.
`-Xprogress=sizes`	Include information on the size of internal compiler data structures in progress announcements.
`-Xprogress=realtime`	Include the realtime used by the compiler in progress announcements.
`-Xprogress=rtime`	Include the realtime used by the compiler in progress announcements (same as realtime).
`-Xprogress=usertime`	Include the usertime used by the compiler in progress announcements.
`-Xprogress=utime`	Include the usertime used by the compiler in progress announcements (same as usertime).
`-Xprogress=%all`	Announce progress at all possible points of compilation.

`-Xprogress=%none`	Do not announce compiler progress.

As with all controls of "name" type, this control can take a list of more than one value, for example: -Xprogress=actions+failures+files.

The progress control-variable has compilation scope and accepts values as shown in the list above. The default value is progress=files.

show: output values of control-variables

The values of specified control-variables may be placed onto stdout by using the command line switch -Xshow. This control-variable may only be specified on the command line; it has no effect when specified in a pragma.

The show control-variable has compilation scope and accepts as values a list of names of other control-variables whose values are to be displayed. The default value is the empty list.

6: Language definitions

This section describes the control that the SNC compiler provides over the definition of the C and C++ programming languages.

C language definition

SNC-C has three modes that govern the dialect of the C language accepted by the compiler, depending on the value of the control-variable c:

- ANSI mode. In this mode, the compiler complies completely with the ANSI C standard (ANSI X3.159-1989 and ISO/IEC 9899:1990(E)) as a "conforming hosted implementation", i.e. it supports all of the standard header files.
- K&R mode. In this mode, the compiler is largely compatible with the definition of the C language as given in *The C Programming Language* by Kernighan & Ritchie and is closely compatible with the UNIX pcc compiler.
- Mixed mode. In this mode, the compiler is essentially an ANSI compiler, except that a few extensions are added to ease the job of porting existing K&R code to ANSI. See below for a discussion of these extensions.

The mixed mode is the default mode and has the following changes from the ANSI mode:

- A number of messages are demoted from errors to warnings.
- The alloca function is recognized as an intrinsic function, and is implemented using its normal K&R definition.

The value c99 can be added to ansi, K&R and mixed modes, to enable C language features that were added in the ISO/IEC 9899:1999 C programming standard. This switch is on by default.

The values const, volatile or signed can be added to K&R mode, causing the compiler to recognize them as keywords. The value inline can be added to any C mode, causing the compiler to recognize it as a keyword and treat it just like C++.

The pragma statements defined in "Controlling the compiler" can be used in all three C modes.

The value noknr can be added to ansi and mixed modes (e.g. -Xc= ansi+noknr) to cause the compiler to give warnings at each definition or declaration of non-prototyped functions. When noknr mode is enabled, a warning is also emitted when the compiler encounters a use of a function that has not been previously declared or defined. This mode can be used to "clean" the code by finding and changing all functions to prototyped versions.

Bit fields of type int are left "implementation-defined" in ANSI C/C++. The behavior of bit fields follows the PlayStation®3 PPU ABI specification.

The representation of char is left implementation-defined in ANSI C/C++. SNC-C/C++ provides the char control-variable to switch between signed and unsigned chars.

C++ language definition

This section describes how to control the definition of the C++ language.

The compiler front end accepts the C++ language as defined by the ISO/IEC 14882:1998 standard and modified by TC1 for that standard. The front end also has four 'dialect' compatibility modes so that programmers using those dialects can continue to compile their existing code. Note however that complete compatibility is not guaranteed or intended. In particular, a compiler error generated natively may result in a different error, or no error at all, when using the SNC compiler.

Dialect

SNC-C++ has four modes that govern the C++ dialect accepted by the compiler, which is determined by the value of control-variable c:

- ARM mode. This is the strict ANSI mode of SNC-C++. This mode initially accepted and implemented the language as described in *The Annotated C++ Reference Manual*, by Margaret A. Ellis & Bjarne Stroustrup (the ARM), which was the base document of the C++ standard ISO/IEC 14882:2003. This mode is invoked by the -Xc=arm option for the compiler driver.
- CP mode. This is same as ARM but with several restrictions relaxed. This mode is the default value for the SNC C compiler (.c and .C files) and for the SNC C++ compiler (.cpp files). This mode is invoked by the -Xc=cp option.
- Cfront 2.1 mode. In this mode the compiler is compatible with AT&T Cfront version 2.1. This mode is invoked by the -Xc=cfront:21 option.
- Cfront 3.0 mode. In this mode the compiler is compatible with AT&T Cfront version 3.0. This mode is invoked by -Xc=cfront or -Xc=cfront:30.

All of these dialects will use a non-templatized version of the library by default.

The C++ compiler is nearly current with the standard. It supports exceptions, RTTI (runtime type identification), templates, namespaces, and libraries including STL (the Standard Template Library). It also recognizes the keywords for bool, and wchar_t. For details on how to control and/or change the language definition recognized, see "c: C/C++ language modes".

Several of the newer features are on by default, but can be selectively disabled by altering the value of control-variable c. All possible values are discussed in the above reference, but additional details are given in the next few paragraphs.

Exception handling

In the SNC compiler exception handling is disabled by default for all modes. If you wish to use exceptions, then you can turn them on by specifying -Xc=*mode*+exceptions on the command line. You may want to refer to the discussion of "c: C/C++ language modes".

Significant comments

There are three different comments that may be placed in C source code to affect warning messages generated by the compiler, as follows:

`/*NOTREACHED*/`	When inserted at the start of a block of code that appears unreachable to the compiler, this comment will suppress the warning message.
`/*VARARGSn*/`	This comment suppresses the usual checking for a variable number of arguments in the following function declaration. The data types of the first n arguments are checked. If n is omitted, a value of zero is used.
`/*ARGSUSED*/`	This directive suppresses warnings about unused arguments in functions.

All three of these comments are case-sensitive.

Predefined symbols

Certain preprocessor symbols are predefined by the compiler (see "Using predefined macros"). Some of these symbols (for example, `__STDC__`) are defined regardless of target computer environment. Others are only defined in the appropriate environment. For a description of language modes, see "c: C/C++ language modes".

Predefined in all language modes (C and C++):

- All of the predefined symbols that are specified by the ANSI C standard except the symbol `__STDC__`, and the following additional symbol: `__SNC__`

Pre-defined in all modes except knr C mode:

- `__STDC__` (defined as the value 1 in ansi C mode, arm C++ mode, and cp C++ mode, and the value 0 in mixed C mode, and cfront C++ mode).

Pre-defined in all C++ modes:

- `__cplusplus`

Controlling global static instantiation order

It is possible to control the order in which global objects that require instantiation at startup have their constructors called. This is done by using the *init_priority* attribute. The usable range is 101-65535; the lower the value assigned the higher will be the construction priority. 0-100 are reserved values, and the default priority (i.e. no init_priority attribute) is 65535.

The syntax is:

```
<object type> <object name> __attribute__((init_priority( x )));
```

e.g.

```
foo myfoo1 __attribute__((init_priority( 110 )));
foo myfoo2 __attribute__((init_priority( 101 )));
foo myfoo3;       // effective priority is 65535
```

These objects will be instantiated at startup is the following order: myfoo2, myfoo1, myfoo3.

The __restrict keyword

SNC features support for an extended form of the __restrict keyword. This allows control of aliasing issues when using pointers in C or C++ source code.

For example:

```
void VectorAdd ( float *Result, const float *Src1, const float *Src2 )
{
  Result[0] = Src1[0] + Src2[0] ;
  Result[1] = Src1[1] + Src2[1] ;
  Result[2] = Src1[2] + Src2[2] ;
} ;
```

This function would appear simple. However, the language would allow this code to be called with Result pointing to the same, or overlapping, memory as Src1 and/or Src2. For this reason the compiler will have to generate each add operation separately, as storing the partial result may overwrite one or both of the source arrays. In these terms, Result is said to possibly alias Src1 and Src2. Faster code could be generated if the compiler knows that storing Result does not alter the inputs.

SNC allows the use of the keyword '__restrict' in both C and C++ source to control pointer aliasing. In addition, a compiler control allows the automatic tagging of function parameters as having an implied __restrict qualifier. The use of the __restrict keyword follows the syntax and conventions of the C99 standard 'restrict' keyword. It is a qualifier that may be added to pointer declarations. For example:

```
float * __restrict pParams;
void VectorAdd ( float * __restrict Result,
const float * __restrict Src1,
const float * __restrict Src2 )
```

The operation performed by the __restrict keyword is controlled by the -Xrestrict control-variable.

`-Xrestrict=0`	Do not act on __restrict keywords. Assume pointer aliases with other pointers.
`-Xrestrict=1`	Assume pointers qualified with __restrict do not alias other __restrict qualified pointers.
`-Xrestrict=2`	Assume pointers qualified with __restrict do not alias any other pointers.

Automatic qualification of function parameters can be controlled via the -Xparamrestrict control-variable:

-Xparamrestrict=0	Does not decorate function parameters of pointer type with the __restrict qualifier.
-Xparamrestrict=1	Automatically decorates function parameters of pointer type with the __restrict qualifier.

When set this control will automatically decorate function parameters of pointer type with the __restrict qualifier. This control can also be modified in source via the pragma feature. For example:

```
#pragma control %push paramrestrict
#pragma control paramrestrict=1
// this function will assume __restrict on its parameters
void qaz ( float * dest, float * src, float * src2 )
{
  dest[0] = src[0] + src2[0] ;
  dest[1] = src[1] + src2[1] ;
  dest[2] = src[2] + src2[2] ;
}
#pragma control %pop paramrestrict
// this function will not assume __restrict
void qaz0 ( float * dest0, float * src0, float * src02 )
{
  dest0[0] = src0[0] + src02[0] ;
  dest0[1] = src0[1] + src02[1] ;
  dest0[2] = src0[2] + src02[2] ;
}
```

For C++ code, SNC also allows the __restrict keyword to be specified with member function declarations and definitions, including declarations and definitions of constructors and destructors:

```
class A
{
public:
  A() __restrict;
  ~A() __restrict
  {
    ...
  }
  int foo() __restrict
  {
    ...
  }

  void bar() __restrict;
}

A::A() __restrict
{
 ...
}

void A::bar() __restrict
{
  ...
}
```

The effect is to associate the __restrict keyword with the 'this' pointer. An error will be generated if __restrict is specified with static member functions. You can use __restrict with member function

definitions even when __restrict has not been specified in the corresponding declaration, but you cannot specify __restrict with a member function declaration without the corresponding definition.

The __unaligned keyword

The __unaligned keyword is a type modifier in pointer definitions; when a pointer is declared with the __unaligned modifier, the compiler assumes that the pointer addresses data that is not correctly aligned. When data is accessed through a pointer declared __unaligned, the compiler generates the additional code necessary to read or write the data without causing alignment errors. Note that there is a performance penalty for the use of this additional code, so it is obviously best to ensure that data is correctly aligned whenever possible.

On the Cell PPU processor, read and write of floating point and vector data types must be correctly aligned or an exception occurs. The hardware will successfully access misaligned integer types.

This modifier describes the alignment of the addressed data only; the pointer itself is assumed to be aligned.

For example, the code snippet below deliberately creates a misaligned access, but the use of the __unaligned keyword allows the code to run successfully.

```
float read (float __unaligned * f)
{
    return *f;
}
char x [5];
float f;
void foo (void)
{
    f = read (&x [1]);
}
```

The __may_alias__ attribute

Accesses to objects with types marked with the __may_alias__ attribute are not subjected to type-based alias analysis, but are instead assumed to be able to alias any other type of objects, just like the char type. This is effectively the opposite of __restrict. The __may_alias__ attribute can be used to mark deliberately aliasing pointers when compiling with the stricter type-based (C99) aliasing rules enabled by -Xrelaxalias=2.

Example:

```
typedef int __attribute__((__may_alias__))  int_a;
int main ()
{
    int local;
    int_a *local_ptr = &local;
}
```

The Microsoft __fastcall and __stdcall extensions

The SNC compiler supports the __fastcall and __stdcall modifiers. Example syntax:

```
//  Function Prototype
void __fastcall foo();
//  fast call function pointer
void(__fastcall *call_to_foo)();
```

For the PS3 target, the fastcall directive bypasses the use of the procedure descriptor and does a direct function call via a pointer to the function address. By using the fastcall directive we can potentially increase performance by removing a level of indirection to the call, and reducing code size by eliminating the instructions required to restore the TOC.

A function descriptor is a two-word data structure that contains a word describing the entry point address of a function, and a second word that describes the TOC base address for the function. These function descriptors are located in the .opd section of an object file. Further information on function descriptors can be found in the PS3 PPU ABI document (located at cell\SDK_doc\en\pdf\OS_lowlevel\PPU_ABI-Specifications_e.pdf).

Warning: The __fastcall calling convention is not compatible with GCC, and fastcall function pointers cannot be passed to GCC-compiled code.

- 'stdcall' calls a function indirectly through the procedure descriptor
- 'fastcall' calls a function directly via a pointer

Example of standard function call:

```
Code:
int bar();
typedef int (*func_ptr)();
func_ptr ptr;
int foo()
{
  return ptr();
}
Output:
.Z8foov:
...
  std    %rtoc,40(%sp)  # save the current TOC value
  lwz    %r5,0(%r4)     # load the function address from the
                        # function descriptor
  lwz    %rtoc,4(%r4)   # load the TOC value from the function
                        # descriptor
  mtctr  %r5            # set CTR to function address
  bcctrl 20,30          # branch to function
  ld     %rtoc,40(%sp)  # restore the TOC
...
```

Example of __fastcall function call:

```
Code:
int __fastcall bar();
typedef int (__fastcall *func_ptr)();
func_ptr ptr;
int foo()
{
  return ptr();
}
Output:
.Z8foov:
...
  lwz    %r4,fastptr@l(%r4)  # load function address
  mtctr  %r4                 # set CTR to function address
  bcctrl 20,30               # branch to function
...
```

The virtual_fastcall and all_fastcall attributes

The virtual_fastcall and all_fastcall attributes can only be attached to C++ classes, and are SNC-specific, i.e. they are not standard or GNU attributes.

virtual_fastcall specifies that all member virtual functions in the class use the __fastcall calling convention, as if you had manually written __fastcall in front of all those functions.

all_fastcall has a similar effect, but it affects all member functions, not just the virtual ones.

The attribute behaviour can be overridden for individual functions, for example, you can use all_fastcall on the class, but then __stdcall on one of the functions.

Warning: In case of class inheritance all derived classes must use the same calling convention for virtual functions. The compiler will issue an error if, for example, virtual_fastcall is specified on a base class but not on a derived class and there are virtual functions being overridden.

Example of the virtual_fastcall attribute:

```
class __attribute__((virtual_fastcall)) Base
{
public:
  Base();
  Base(int c);
  virtual ~Base();

  virtual int foo(int a, int b);

  int getC();
};
```

Note that this is equivalent to:

```
class Base
{
public:
  Base();          // Ctor and dtor calling convention
  Base(int c);     // is determined by ABI and it is
  virtual ~Base(); // not affected by the attribute

  virtual int __fastcall foo(int a, int b);

  int getC();
};
```

Similarly, if a class is defined with the all_fastcall attribute:

```
class __attribute__((all_fastcall)) Base
{
public:
  Base();
  Base(int c);
  virtual ~Base();

  virtual int foo(int a, int b);

  int getC();
};
```

Then this is equivalent to:

```
class Base
{
public:
  Base();          // Ctor and dtor calling convention
  Base(int c);     // is determined by ABI and it is
  virtual ~Base(); // not affected by the attribute

  virtual int __fastcall foo(int a, int b);

  int __fastcall getC();
};
```

7: Pre-compiled headers

It is often desirable to avoid recompiling a set of header files, especially when they introduce many lines of code and the primary source files that #include them are relatively small. The EDG front end provides a mechanism for, in effect, taking a snapshot of the state of the compilation at a particular point and writing it to a disk file before completing the compilation; then, when recompiling the same source file or compiling another file with the same set of header files, it can recognize the "snapshot point", verify that the corresponding pre-compiled header (PCH) file is reusable, and read it back in. Under the right circumstances, this can produce a dramatic improvement in compilation time; the trade-off is that PCH files can take a lot of disk space.

For a complete list of the --pch switches see "Pre-compiled headers".

Automatic pre-compiled header processing

When --pch appears on the command line, automatic PCH processing is enabled. This means the front end will automatically look for a qualifying PCH file to read in and/or will create one for use on a subsequent compilation.

The PCH file will contain a snapshot of all the code preceding the header stop point. The header stop point is usually the first token in the primary source file that does not belong to a pre-processing directive, but it can also be specified directly by #pragma hdrstop if that comes first.

For example:

```
#include "xxx.h"
#include "yyy.h"
int i;
```

The header stop point is int (the first non-preprocessor token) and the PCH file will contain a snapshot reflecting the inclusion of xxx.h and yyy.h. If the first non-preprocessor token or the #pragma hdrstop appears within a #if block, the header stop point is the outermost enclosing #if.

For example:

```
#include "xxx.h"
#ifndef YYY_H
#define YYY_H 1
#include "yyy.h"
#endif
#if TEST
    int i;
#endif
```

Here, the first token that does not belong to a pre-processing directive is again int, but the header stop point is the start of the #if block containing it. The PCH file will reflect the inclusion of xxx.h and conditionally the definition of YYY_H and inclusion of yyy.h; it will not contain the state produced by #if TEST.

A PCH file will be produced only if the header stop point and the code preceding it (mainly, the header files themselves) meet certain requirements:

(1) The header stop point must appear at file scope. It may not be within an unclosed scope established by a header file. For example, a PCH file will not be created in this case:

```
// xxx.h
class A {
// xxx.C
#include "xxx.h"
int i; };
```

(2) The header stop point may not be inside a declaration started within a header file, nor (in C++) may it be part of a declaration list of a linkage specification. For example, in the following case the header stop point is int, but since it is not the start of a new declaration, no PCH file will be created:

```
// yyy.h
static
// yyy.C
#include "yyy.h"
int i;
```

(3) Similarly, the header stop point may not be inside a #if block or a #define started within a header file.

(4) The processing preceding the header stop must not have produced any errors.

Warning: warnings and other diagnostics will not be reproduced when the PCH file is reused. No references to predefined macros __DATE__ or __TIME__ may have appeared.

(5) No use of the #line pre-processing directive may have appeared.

(6) #pragma no_pch must not have appeared.

(7) The code preceding the header stop point must have introduced a sufficient number of declarations to justify the overhead associated with PCHs.

When a PCH file is produced, it contains, in addition to the snapshot of the compiler state, some information that can be checked to determine under what circumstances it can be reused. This includes:

- The compiler version, including the date and time the compiler was built.
- The current directory (i.e., the directory in which the compilation is occurring).
- The command line options.
- The initial sequence of pre-processing directives from the primary source file, including #include directives.
- The date and time of the header files specified in `#include` directives.

This information comprises the PCH prefix. The prefix information of a given source file can be compared to the prefix information of a PCH file to determine whether the latter is applicable to the current compilation. As an illustration, consider two source files:

```
// a.C
#include "xxx.h"
... // Start of code
// b.C
#include "xxx.h"
... // Start of code
```

When a.C is compiled with --pch, a PCH file named a.pch is created. Then, when b.C is compiled (or when a.C is recompiled), the prefix section of a.pch is read in for comparison with the current source file. If the command line options are identical, if xxx.h has not been modified, and so forth, then, instead of opening xxx.h and processing it line by line, the front end reads in the rest of a.pch and thereby establishes the state for the rest of the compilation. It may be that more than one PCH file is applicable to a given compilation. If so, the largest (i.e., the one representing the most pre-processing directives from the primary source file) is used. For example, consider a primary source file that begins with:

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

If there is one PCH file for xxx.h and a second for xxx.h and yyy.h, the latter will be selected (assuming both are applicable to the current compilation). Moreover, after the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers may be created.

When a PCH file is created, it takes the name of the primary source file, and the extension will be replaced by .pch. Unless --pch_dir is specified it is created in the directory of the primary source file. When a PCH file is created or used, a message such as:

```
"test.C": creating precompiled header file "test.pch"
```

is issued. You can suppress the message by using the command-line option --no_pch_messages.

When the --pch_verbose option is used the front end will display a message for each PCH file that cannot be used giving the reason for this.

In automatic mode (i.e. when --pch is used) the front end will deem a PCH file obsolete and delete it under the following circumstances:

- if the PCH file is based on at least one out-of-date header file but is otherwise applicable for the current compilation; or
- if the PCH file has the same base name as the source file being compiled (e.g., xxx.pch and xxx.C) but is not applicable for the current compilation (e.g., because of different command-line options).

This handles some common cases; other PCH file clean-up must be dealt with by the user. Support for PCH processing is not available when multiple source files are specified in a single compilation. An error will be issued and the compilation aborted if the command line includes a request for PCH processing and specifies more than one primary source file.

Manual pre-compiled header processing

- Command-line option --create_pch=*filename* specifies that a PCH file of the specified name should be created.
- Command-line option --use_pch=*filename* specifies that the indicated PCH file should be used for this compilation; if it is invalid (i.e., if its prefix does not match the prefix for the current primary source file), a warning will be issued and the PCH file will not be used.

When either of these options is used in conjunction with --pch_dir, the indicated file name (which may be a path name) is tacked on to the directory name, unless the file name is an absolute path name.

The --create_pch, --use_pch, and --pch options may not be used together. If more than one of these options is specified, only the last one will apply. Nevertheless, most of the description of automatic PCH processing applies to one or the other of these modes. Header stop points are determined the same way, PCH file applicability is determined the same way etc.

Overriding the check that PCH files must be in the same directory

- The control-variable -Xpch_override will disable the compiler's check that the file used to generate the PCH file is in the same directory as the file being compiled.

The compiler implements a number of checks to ensure that a compilation using PCH files will behave exactly as a compilation without the use of PCH files. See "Automatic pre-compiled header processing" for more information on these checks. However it has been found that one specific coherency check prevents a commonly used idiom for PCH files.

This check is that the file used to generate the PCH file must be in the same directory as the file being compiled. This prevents files in different directories from sharing the PCH information. In this circumstance the compiler will generate the following warning message:

```
"the file being compiled needs to be in the same directory as the file used to create the PCH file"
```

and will not use the PCH file specified but the compilation will continue without the use of PCHs. This check is needed if header files with the same name but different contents are used in different subdirectories.

Consider this directory example:

```
src\
    src1\
          A.h
          foo1.cpp
    src2\
          A.h
          foo2.cpp
```

If the two files foo1.cpp & foo2.cpp both include A.h via this #include line:

```
#include "A.h"
```

then, if we compile the files *without* PCH files, foo2.cpp will pick up the local A.h header file in `src\src2`.

However if we issue the following commands:

```
ps3ppusnc -c src1\foo1.cpp –create_pch=foo1.pch
ps3ppusnc -c src2\foo2.cpp --use_pch=foo1.pch
```

then the second compilation will not use the PCH file and a warning message will be generated (as above) since foo1.cpp—the file used to construct the PCH file—is not in the same directory as the file being compiled (foo2.cpp).

You can override this check by using the control-variable -Xpch_override=1 as follows:

```
ps3ppusnc -c src2\foo2.cpp --use_pch=foo1.pch -Xpch_override=1
```

This compilation will succeed but the A.h used will be the one from the `src1\` directory, and not from `src2\`.

- If a project does *not* make use of the idiom of having header files of the same name in different subdirectories then the pch_override control-variable may be safely used.

Controlling pre-compiled headers

There are several ways in which you can control and/or tune how PCHs are created and used.

#pragma hdrstop may be inserted in the primary source file before the first token that does not belong to a pre-processing directive. It enables you to specify where the set of header files subject to pre-compilation ends.

For example,

```
#include "xxx.h"
#include "yyy.h"
#pragma hdrstop
#include "zzz.h"
```

Here, the PCH file will include processing state for xxx.h and yyy.h but not zzz.h. This is useful if you decide that the information added by what follows the #pragma hdrstop does not justify the creation of another PCH file.

- #pragma no_pch may be used to suppress PCH processing for a given source file.
- Command-line option --pch_dir *directory-name* is used to specify the directory in which to search for and/or create a PCH file.

Performance issues

The relative overhead incurred in writing out and reading back in a PCH file is quite small for reasonably large header files.

In general, it does not cost much to write a PCH file out even if it does not end up being used, and if it *is* used it invariably speeds up compilation. The problem is that the PCH files can be quite large, from a minimum of about 250 KB to several megabytes or more.

Thus, despite the faster re-compilations, PCH processing is not likely to be justified for an arbitrary set of files with nonuniform initial sequences of pre-processing directives. Rather, the greatest benefit occurs when a number of source files can share the same PCH file. The more sharing, the less disk space is consumed. With sharing, the disadvantage of large PCH files can be minimized, without giving up the advantage of a significant speedup in compilation times.

Consequently, to take full advantage of header file pre-compilation, you should expect to reorder the #include sections of your source files and/or to group #include directives within a commonly used header file.

Different environments and different projects will have different needs, but in general, you should be aware that making the best use of the PCH support will require some experimentation and probably some minor changes to source code.

8: Optimization strategies

The SNC compiler features a number of optimization phases, many of which are specifically designed for PlayStation®3. The optimizer controls are designed to be immediately familiar to users of the GNU toolchain, however there are some differences. To get the most out of SNC's optimizer we recommend becoming familiar with its specific features and controls. This chapter is designed to aid in this familiarization process.

Most optimizations can be controlled either on a per-file basis with the use of command line switches, or on a per-function basis with the use of control pragmas. SNC also allows users to provide some additional annotations to the code in order to give the optimizer more information via the use of attributes or pragmas. Many of these will be familiar to users of either the GNU toolchain or previous versions of SNC, however some are related to new features for SNC for PlayStation®3 and so may not be familiar.

> **Tip:** We recommend reading the Release Notes and the Important Changes document whenever moving to a newer version of SNC, as improvements are always being made that may be controlled via new switches or code annotations. This chapter will also be updated to reflect these changes.

As with GCC, when compiling with -O0 or without specifying an optimization level, the optimizer will not be run. The only inlining that will be performed is forced inlining. Compiling with -O2 enables most optimizations, however there are a number of optimizations that must be enabled specifically as they may rely on certain assumptions that the compiler makes about the code or they may be only useful in specific circumstances. SNC also features an "debuggable optimized" mode by specifying -Od. This enables a number of optimizations but should still allow a good level of source correspondence and other debug information. Compiling with -Os will optimize for size, although again certain optimizations are disabled by default due to relying on assumptions about the code.

We recommend the following baseline settings for optimized builds to get the best combinations of size/performance:

```
-O2 -Xfastmath=1 -Xassumecorrectsign=1
```

or

```
-Os -Xassumecorrectsign=1
```

These switches are described in more detail below.

Main optimization level

The main optimization level is controlled with the -O<n> switch where <n> specifies the level:

<n>	Optimization
0	No optimization. No inlining is performed except forced inlining.
1	Some basic optimization. Some inlining is performed.
2	Full conservative optimization with inlining.
3	Full conservative optimization with inlining. Currently -O3 enables the same optimization set as -O2 but in future releases may also feature some more time-consuming optimizations.
d	Debuggable optimized mode. Optimizations that should not affect the quality of the debug information are performed.
s	Size optimized mode. Optimizations that will increase code size are not performed and there is a reduced amount of inlining.

Inlining controls

There are three main switches to control inlining in SNC. Making adjustments to these values can yield massive improvements to the size and execution speed of compiled code. Unfortunately it is not possible to have default values that are best suited to all styles of code. **Therefore, we highly recommend that you experiment with these values to find the optimum ones for your code.**

The parameters to these controls express the maximum size of a function in terms of 'instructions'. These are internal compiler instructions and are not necessarily the same as individual processor instructions.

-Xautoinlinesize - controls automatic inlining

This switch limits the maximum size of functions that will be automatically inlined by the compiler without them having been marked as inline in the source code. This does not apply to explicitly inline functions such as C++ methods defined inside classes in header files (see "-Xinlinesize - controls inlining of explicitly inline functions").

See "-Xautoinlinesize".

-Xinlinesize - controls inlining of explicitly inline functions

This switch limits the maximum size of explicitly inline functions that will be inlined by the compiler. Explicitly inline functions include C++ methods defined inside classes in header files.

See "-Xinlinesize".

-Xinlinemaxsize - controls the maximum amount of inlining into any one function

This switch controls the maximum amount of inlining into any one function. It is used to prevent individual functions from becoming too large and slowing down other stages in the optimizers. Increasing this from the default value may increase the amount of inlining (and therefore possibly performance) at the expense of compilation speed.

See "-Xinlinemaxsize".

Forced inlining

If a function is marked with the `__attribute__((always_inline))` attribute, it will be inlined even at -O0 when no other inlining is performed.

Example:

```
#define FORCE_INLINE __attribute__((always_inline))
FORCE_INLINE int timesTwo( int x )
{
    return ( 2 * x );
}

int main()
{
    return timesTwo( 3 );
}
```

Finding the optimal inlining settings

In our own testing we have found that massive improvements to code performance and size can be achieved by finding the optimal inlining settings for a project. Unfortunately, the optimal settings vary wildly between different codebases. The defaults have been set at a level to get good performance over as large a cross-section of code as possible.

- *In code that makes extensive use of the inline keyword, or functions defined inside class definitions*, benefits may be had by increasing the value for -Xinlinesize=<n>. The default at -O2 is 256. A good starting point for experimentation would be 512 or even 1024.
- *For code that is designed to rely on the compiler to make the decision of whether to perform inlining* we suggest increasing the value for -Xautoinlinesize=<n>. The default at -O2 is 32. Increasing this value

will allow more automatic inlining. A good starting point for experimentation would be to increase this value to 128.

By adjusting these values, measuring the impact on code size and performance, and then increasing or lowering the values to find the best combination of size against performance, a 'sweet-spot' can often be found which give improvements over both of the above default values.

Additional optimizations

The optimizer provides a number of additional optimizations that are enabled by the `-Xfastmath=1` switch.

These include:

- Automatic use of VMX registers to avoid conversion between floating point, integer and VMX registers. This will reduce the number of Load Hit Store penalties.
- Conversion of if statements to '`fsel`'.
- Replacement of '`fdiv`' with an approximate divide and refinement.

The `-Xfastmath` switch is not enabled by default because these optimizations may not work correctly with code that relies on the edge behavior of floating point values such as 'denormal' numbers. This should not affect the vast majority of code.

The optimizations enabled by `-Xfastmath=1` are extremely sensitive to floating point divisions by extremely small values below the value of FLT_EPSILON. We recommend always checking that the divisor is greater than the value of FLT_EPSILON when `-Xfastmath=1` is specified.

Example:

```
#include <float.h>

float divide( float x, float y )
{
    if ( y < FLT_EPSILON )
    {
          y = FLT_EPSILON;
    }

    float z = x / y;

    return z;
}
```

It is also important when using the `-Xfastmath` switch to be careful with data residing in video memory, which has stricter alignment constraints than standard memory.

> **Tip:** The base address of video memory is determined by using the macro `RSX_FB_BASE_ADDR`.

Alignment traps in video memory can be avoided by marking the variables resident in video memory as volatile.

Example:

```
// This function is safe to be used with a pointer to video
// memory even when -Xfastmath=1 is specified.

void videoFunc(short int volatile *somewhereInVideoMemory)
{
    // (some code)
}
```

> **Tip:** We highly recommend that `-Xfastmath` is enabled on optimized builds wherever possible and that code relying on the above edge conditions is modified to work with it.

Pointer arithmetic assumptions

Pointer arithmetic on the PS3 is quite difficult as we are running a 32-bit address model in a 64-bit address space. This means that the compiler must emit extra instructions to ensure that the top 32 bits of a final address are zero, otherwise the code will crash with an address exception.

This puts a huge burden on the compiler and increases code size considerably. By enabling the following switch, we assume that the code follows a few simple rules, which in most cases will be true. The C99 standard documents these rules, and in order to obtain the benefit of this switch your code must adhere to these rules.

- If -Xassumecorrectalignment=1 the compiler is able to generate more efficient memory access sequences but these will fail at run-time if alignment of pointers is not declared correctly. Therefore, when porting, initially use -Xassumecorrectalignment=0, fix the bugs and then set -Xassumecorrectalignment=1. See "Assume correct pointer alignment".

> **Tip:** This switch is on by default and we highly recommend that you leave it enabled in optimized builds and that where necessary your code is modified to conform to the assumptions made.

Assume correct pointer alignment

On the PPU architecture all data types have default alignments in memory and the compiler will place data in memory to follow these rules. For example doubles are always 8-byte aligned, ints are 4-byte aligned. -Xassumecorrectalignment allows the compiler to assume these rules for all objects accessed via pointers. For example:

```
double * dbl_pointer; // dbl_pointer will always contain 8-byte aligned addresses
int * int_pointer;   // int_pointer will always contain 4-byte aligned addresses
```

However, it is possible to create unaligned pointers by casting from smaller sizes or intptr_t.

Example:

```
char x[ 10 ];
int main()
{
  int *p = (int*)( x + 5 );
  *p = 0;
}
```

Here we have used a cast to create an incorrectly aligned pointer and have written a four-byte zero to the array "x".

This will not work on some targets and may cause some optimizations to fail.

On the PPU, for example, it is possible, albeit inefficient, to read from and write to unaligned integer pointers, but not to unaligned floating point pointers.

So this will fail:

```
char x[ 10 ];
int main(){
  float *p = (float*)( x + 5 );
  *p = 0;
}
```

as will this:

```
char x[ 10 ];
float f;
int main()
{
  int *p = (int*)( x + 5 );

  union { int i; float f; } u;
```

```
  u.i = *p;
  f = u.f;
}
```

Here, if the optimizer assumes that p is aligned correctly, it may replace the load > store > load sequence with a direct float load, which is illegal.

So to enable this optimization on the PPU target, we must use -Xassumecorrectalignment and we have to make sure that all our pointers are aligned.

Most importantly, when vectorizing, the optimizer can use a plain lvlx instruction to load a scalar into a vmx register.

Otherwise, vectorization code will use the sequence:

```
add tmp1, addr, 16
lvlx tmp2, addr
lvrx tmp3, tmp1
vor result, tmp2, tmp3
```

which is much longer.

If sections of your code contain potentially misaligned pointers, you can either disable -Xassumecorrectalignment by setting it to 0, in which case the optimizer will not perform any transformations that assume knowledge of a pointer's alignment when this cannot be determined by the compiler. Alternatively, you can use the `__unaligned` keyword as follows:

```
char x[10];
           ...
        __unaligned int *p = (int *)(x + 5)
```

See "The __unaligned keyword".

Use of –Xassumecorrectsign

The PPU processor is a 64-bit machine but the programming model is a 32-bit model where 32-bit calculations are carried out in 64-bit registers. The compiler therefore needs to add mask instructions to clamp calculations into 32 bits when the calculation overflows 32 bits.

A simple scalar example is:

```
unsigned x = 0xffffffff;
f( x + 1 ); // expecting x + 1 = 0, but 0xffffffff + 1 = 0x100000000 in 64-bit
arithmetic.
```

Here we are exploiting the ANSI C property of calculating addition modulo 2^32. Without an extra mask to zero the upper 32 bits the overflow to zero will not happen.

These masking instructions can lead to an increase in code size but cases may be rare in practice.

The compiler therefore provides the assumecorrectsign option which if enabled will help reduce code size and improve performance by making assumptions about calculations involving pointers and integers, i.e. POINTER +/ Integer. The basic assertion is that the code is not relying on any overflow semantics for pointer and integer calculations.

assumecorrectsign is especially useful in address calculations in loops which use the loop variable. The flag allows the compiler to assume that the base address + offset calculations will not overflow in the loop and hence can be hoisted out of the loop.

It is not possible for the compiler to warn about dangerous situations where assumecorrectsign should not be used and you are advised to experiment with the option to see if the code still runs and that positive benefits can be seen. If the program does crash with a page fault and the faulting address has a high order bit set (upper 32 bits) then it is likely that some address overflow semantics were assumed and therefore assumecorrectsign cannot be used in this case.

Handling pointer relocation

When relocating pointers stored in files, make sure that the base of the pointer is an unsigned int and the offset is a signed int. This way negative offsets will not overflow.

Example:

```
struct RelocateMe
{
  RelocateMe *next; // when loaded from a file, this is an offset.
};

void relocate( void *base, RelocateMe *ptr )
{
  if( ptr->next != NULL )
  {
    ptr->next = (RelocateMe*)( (unsigned) base + (int)ptr->next );
    relocate( base, ptr->next );
  }
}
```

Virtual call speculation

Virtual function calls are often useful in Object Oriented design, however on the PPU they can impose a very large performance penalty over normal function calls.

After analysis of large amounts of game code we have observed that a single virtual function is often the target of the majority of virtual function calls made. The virtual call speculation feature allows users to tell SNC when this is the case so that it can eliminate the virtual function overhead in the majority of cases. This is done with the use of the `__attribute__((likely_target))` attribute.

Example:

```
#include <stdio.h>

#if defined ( USE_LIKELY )
#    define LIKELY_TARGET __attribute__((likely_target))
#else
#    define LIKELY_TARGET
#endif

class Base
{
    public:
        virtual int foo();
};

class Wibble : public Base
{
    public:
        LIKELY_TARGET virtual int foo();
};

int Base::foo()
{
    printf( "Base foo\n" );
    return 0;
}

int Wibble::foo()
```

```
{
    printf( "Wibble foo\n" );
    return 1;
};

int bar()
{
    Wibble* w = new Wibble();

    return w->foo();
}
```

When this code is compiled with USE_LIKELY defined, the attrribute is applied to the virtual function Wibble::foo(). SNC is then able to assume that Wibble::foo will be called in more cases than any other versions of foo in the vtable (in this case, Base::foo).

When a call to foo is made, rather than immediately going via the vtable and incurring the associated performance penalty, a compare is generated. This compares whether the target is the marked function and if so a direct branch is taken. If the target of the call is not the marked function then the normal virtual call mechanism is used.

If the marked function meets the normal inlining criteria then the direct branch will be replaced with an inlined copy, further improving performance.

This means that in the case where the target is the marked function it should be substantially faster. In the case where the target is another function there will be a small penalty due to the extra compare.

See also "Marking a function as 'hot'".

Marking a function as 'hot'

If SNC knows that a particular function accounts for a large amount of time in a frame it can take this into account when optimizing by performing transformations that increase the size of the function and increasing inlining of the function.

A function can be marked as "hot" with the use of __attribute__((hot)). Marking a virtual function as being hot will also have the same effect as marking it with __attribute__((likely_target)) for virtual call speculation (see "Virtual call speculation").

Inlining of "hot" functions can be controlled with the switch -Xinlinehotfactor=<n> where <n> is the factor that the inlining settings (controlled by the inlining switches, see "Inlining controls") are increased by in the case of hot functions. See "-Xinlinehotfactor".

In future releases of SNC, further optimizations will be enabled for 'hot' functions.

Alias analysis

If a pointer refers to the same location as another it is said to *alias* that other pointer. In order to try and produce the optimal code scheduling, SNC performs alias analysis to find when a pointer aliases another pointer.

At -O2 by default SNC assumes that code does not break the C99 strict aliasing rule. By making this assumption it is possible to perform much more aggressive scheduling and therefore generate more efficient code. By relying on this assumption, however it is possible to write code that does not comply with it.

This assumption is controlled by the switch -Xrelaxalias control-variable. See "-Xrelaxalias".

> **Tip:** We recommend that this value is set to at least 2 for optimized builds and any code that violates the strict aliasing rule is modified to conform with it.

If the code cannot be modified easily to work at -Xrelaxalias=2, we recommend lowering the value to -Xrelaxalias=1. The vast majority of code should work at this level. -Xrelaxalias=0 should only be used in the case where even this fails.

Optimizing on a per-function basis

SNC fully supports enabling and disabling of optimizations on a per-function basis with the use of pragmas within the code. This can be done to all optimizations at once or just to individual optimizations. See "Control pragmas".

A typical use might be to turn off optimization on a function that is being debugged in a file that is otherwise being compiled at -O2:

```
#define START_NOT_OPTIMIZING _Pragma("control %push O=0")
#define END_NOT_OPTIMIZING   _Pragma("control %pop O")

void aFunctionIWantOptimized()
{
    //...
}

START_NOT_OPTIMIZING

void aFunctionIAmTryingToDebug()
{
    //...
}

END_NOT_OPTIMIZING

void anotherFunctionIWantOptimized()
{
    //...
}
```

Note that this will not work in reverse, by trying to enable optimization on a single function in a file that is being compiled at -O0. This is because when a file is compiled at -O0 the optimizer is not even enabled in order to speed up compilations.

Another use of this feature might be to enable certain optimizations on specific functions. Loop unrolling for example (enabled on the command line via -Xunroll=1) might not be suitable to be enabled on an entire project as it tends to increase overall code size. However it might be useful for specific functions where performance is critical.

Example:

```
#define START_UNROLLING _Pragma("control %push unroll=1")
#define END_UNROLLING   _Pragma("control %pop unroll")

START_UNROLLING

int functionToUnRoll( int x )
{
    for ( int i = 0; i < 3; ++i )
    {
        x += 7;
    }

    return x;
}
```

```
END_UNROLLING
```

Debugging optimized code

In optimized code, local variables, including function parameters, are generally kept in registers when possible. This tends to make examining these variables in the debugger difficult, because as soon as the last use of a variable is reached, the register holding that variable may be re-used for a different variable, even though the original variable is still in scope.

The SNC compiler provides the -Od switch to overcome some of these difficulties. The -Od switch enables optimizations that will not adversely affect the ability to debug the code. In addition it treats variables as though they are used at the point where they go out of scope, to improve debugging.

Example:

```
1  extern int printf(const char *, ...);
2  int glb1 = 10;
3  int glb2 = -10;
4  int glb3 = 0;
5
6  int
7  main(void)
8  {
9    int loc1, loc2;
10   loc1 = glb1;
11   ++glb1;
12   printf("loc1 = %d\n", loc1);    // prints 10
13   loc2 = glb2;
14   ++glb2;
15   if (glb2 < 0) {
16     int loc3;
17     loc3 = glb1 + glb2;
18     glb2 = 0;
19     printf("loc3 = %d\n", loc3);  // prints 2
20     ++glb2;
21   } // 'loc3' goes out of scope here
22   printf("loc2 = %d\n", loc2);    // prints -10
23   glb3 = glb1 / glb2;
24   printf("glb1/glb2 = glb3 = %d\n", glb3);  // prints 11
25   return 0;
26 } // 'loc1' and 'loc2' go out of scope here
```

In this example, the last use of variable 'loc2' is line 22, where it is passed to `printf()`, but it does not go out of scope until line 26, the end of the function. The `-Od` switch causes the code to appear to make additional use of 'loc2' at the end of the function. Compiling with `-Od` means the debugger can view the variables from their point of definition until they go out of scope, rather than their last actual use.

Limitation when compiling with –Od

There is a known limitation when compiling with `-Od`: Unneeded restore instructions are produced.

Compiling with `-Od` extends the lifetime of variables, from when they were last used to when they go out of scope instead, and therefore requires the use of many registers. When all available registers are used up, spill and restore code will be generated for any additional variables. Generally the restore is unneeded since the lifetime was artificially extended to the point where the variable went out of scope (that is, when the lifetime was artificially extended, the variable is not actually used at the point where the variable went out of scope, so there is no need to do the restore). This unneeded restore code will increase the executable size, and reduce the execution speed.

9: Control-variable reference

All of the control-variables are listed alphabetically by name in the following tables. See "Control-variable definitions" for a more complete discussion of the meaning of each control-variable.

For each control-variable the following properties are listed:

- the name of the definition
- the scope of the variable (compilation, file, line, loop or function)
- the type and/or range of values
- the default value
- a brief (one or two sentence) explanation of the meanings of the various values that can be assigned to the control-variable.

For a detailed explanation of control-variable scope, see "Control-variables".

The format of each table is as follows:

-Xcontrol-variable	scope	type/values	default
Value #1	Explanation #1.		
Value #2	Explanation #2.		
...	...		

-Xalias

-Xalias	function	0..3	0
0	No alias analysis, assume each memory reference interferes with everything.		
1	Alias analysis based on declarations only.		
2	Alias analysis based on declarations and on use of constant subscripts.		
3	Previous analysis plus use of flow-sensitive considerations.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xalignfunctions

-Xalignfunctions	file	1..32768	4
N	Cause functions to be aligned to the next power of 2 greater than or equal to *n*. For example, -Xalignfunctions=8 causes functions to be aligned on an 8-byte boundary.		

-Xassumecorrectalignment

-Xassumecorrectalignment	function	0..1	1

0	Do not assume that pointers have correct alignment.
1	Assume that pointers have correct alignment (the default).

-Xassumecorrectsign

See "Use of –Xassumecorrectsign".

-Xassumecorrectsign	function	0..1	0
0	Assumes that only simply proven values are valid.		
1	Assumes that if pointer is valid then adding -32768..32767 results in a valid pointer also.		

-Xautoinlinesize

This switch limits the maximum size of functions that will be automatically inlined by the compiler without them having been marked as inline in the source code. This does not apply to explicitly inline functions such as C++ methods defined inside classes in header files (see "-Xinlinesize").

-Xautoinlinesize	function	0..50000	0
0	No automatic inlining.		
N	Allow automatic inlining of unmarked functions up to a maximum size of *n* instructions.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

See also -Xinlinesize and -Xinlinemaxsize.

-Xautovecreg

-Xautovecreg	function	0..2	0
0	Do not automatically perform vmx optimizations.		
1	Use vmx registers to avoid LHS dependencies on conversion and integer/float casts.		
2	Use vmx registers to avoid variable shift, small variable multiplies and other expensive operations.		

-Xbranchless

-Xbranchless	function	0..2	0
0	Do not use branchless compares.		
1	Use branchless compares for ternary operators only, e.g. a > b ? a : b		
2	Use branchless compares for all possible integer comparisons.		

-Xbss

-Xbss	function	0..2	1
0	All data will be placed in the .data section.		
1	Static uninitialized data and data initialized to zero may be placed in either the .data section or .bss section according to automatic rules within the compiler.		
2	Static uninitialized data will be placed in the .bss section; data initialized to zero will be placed in the .bss section where possible.		

-Xc

-Xc	file	list of names	mixed+gnu_ext+c99
ansi	For C the compiler complies completely with the ANSI and ISO C standard (ANSI X3.159-1989 and ISO/IEC 9899:1990(E)) as a "conforming hosted implementation", i.e. it supports all of the language and standard header files.		
knr	For C the compiler is largely compatible with the definition of the C language as given in *The C Programming Language* by Kernighan & Ritchie and is closely compatible with the UNIX pcc compiler.		
mixed	(default: on) For C the compiler is essentially an ANSI compiler, except that a few extensions are added to ease the job of porting existing K&R code to ANSI. See "Language definitions" for a discussion of these extensions.		
knr+x	In addition to the above three basic modes, any subset of the three names const, volatile, and signed may be added to the value of control-variable c, forming a list value. When the basic mode is c=knr, the use of any of these names indicates that the corresponding qualifier in the ANSI C language is to be recognized. For example, c=knr+const+volatile indicates K&R compatibility, but with the const and volatile type qualifiers of ANSI C also recognized. An additional value, noknr, can be added to the mixed or ansi C modes (for example, Xc=mixed+noknr). This value causes the compiler to emit warnings on declarations and definitions of any function without a prototype. When noknr mode is enabled, a warning is also emitted when the compiler encounters a use of a function that has not been previously declared or defined. Another additional value inline can be added to C modes ansi, knr and mixed to make inline be a keyword as in C++ (for example, Xc+=inline).		
c99	(default: on) The value c99 can be added to ansi, K&R and mixed modes, to enable C language features that were added in the ISO/IEC 9899:1999 C programming standard.		
cfront:21	For C++, the compiler is compatible to AT&T Cfront 2.1.		
cfront cfront:30	For C++, the compiler is compatible to AT&T Cfront 3.0		
arm	For C++, the compiler implements the language described in *The Annotated C++ Reference Manual* by Margaret A. Ellis & Bjarne Stroustrup, modified by changes made in the C++ standard (ISO/IEC 14882:2003.		

cp	For C++, similar to arm, except that it allows for several anachronisms and is less restrictive.
Several additional values may be added to, or subtracted from, any of the C++ language modes.	
c_func_decl	(default: off) permits C-style function prototypes to support inclusion of non-C++ include files.
array_nd	(default: off) enables recognition of array new and delete operators.
rtti	(default: on) enables RTTI behavior.
wchar_t	(default: off) makes wchar_t a keyword declaring a distinct type.
bool	(default: off) makes bool a keyword declaring a distinct type.
old_for_init	(default: off) increases the scope of variables declared in for loop init statements.
exceptions	(default: off) permits use of exception handling constructs and behavior.
tmplname	(default:off) creates templates with mangled names that are distinct from the names given to non-templated functions.
gnu_ext	(default: on) allows use of GNU GCC extensions to the C/C++ languages.
msvc_ext	(default: off) allows use of Microsoft Visual Studio® extensions to the C/C++ languages.

-Xcallprof

-Xcallprof	function	0..1	0
0	Do not generate extra code for Tuner callprof hierarchical profiling.		
1	Generate extra code for function entry and exit to allow profiling via Tuner. This extra code has a very small impact on the performance of the application. See the Tuner for PS3 user guide for more information on this new callprof feature.		

-Xchar

-Xchar	file	name	signed
signed	In C/C++, type char is signed by default.		
unsigned	In C/C++, type char is unsigned by default.		

-Xconstpool

This optimization groups together constants used by each function into a contiguous cache aligned block of memory. This improves cache locality and simplifies the code required to load the constants into registers (they share a common high address).

-Xconstpool	function	0..1	0
0	No pooling.		

1	Create per function constant pools (the default at O2).

-Xdebuglocals

-Xdebuglocals	function	0..1	0
0	Do not extend the lifetimes of local variables to the points where they go out of scope.		
1	Extend the lifetimes of local variables to the points where they go out of scope.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xdebugvtbl

-Xdebugvtbl	function	0..1	0
0	Do not generate debug data for the C++ virtual tables that may be contained in classes.		
1	Generate debug data for the C++ virtual tables that may be contained in classes. Thus when examining classes in the debugger's watch pane the virtual table pointer may be also examined.		

-Xdeflib

-Xdeflib	line	0..2	1
0	Do not inline intrinsic functions.		
1	Inline intrinsic functions under automatic control.		
2	Inline intrinsic functions whenever it is possible to do so.		

-Xdepmode

-Xdepmode	file	0..1	1
0	Use GCC 2 style dependency filenames.		
1	Use GCC 3/4 style dependency filenames.		

-Xdiag

-Xdiag	line	0..2	1
0	Output diagnostics at the error and fatal error levels. Do not output remark or warning messages.		
1	Output diagnostics at the warning, error and fatal error levels. Do not output remark messages.		

2	Output diagnostics at the remark, warning, error and fatal error levels.

-Xdiaglimit

-Xdiaglimit	file	0..1000000	0
N	Limit number of messages issued for each diagnostic to the first *n* occurrences. A value of 0 means unlimited.		

-Xdivstages

Used in conjunction with -Xfastmath, this switch controls the number of iterations used when refining the results of approximated floating point divides.

For typical 'game' applications -Xdivstages=3 should give the right balance of speed and accuracy.

See also "-Xfastmath".

-Xdivstages	function	0..5	0
0	Disable fast approximation (use fdiv)		
1	Just fre instruction		
2	fre + one stage of newton-raphson (~10 bits)		
3	fre + two stages of newton-raphson (~20 bits)		
4	fre + three stages of newton-raphson (~30 bits)		
5	fre + four stages of newton-raphson (roughly the same as using fdiv)		

-Xfastlibc

-Xfastlibc	compilation	0..1	0
0	Do not replace libc.a by libcs.a.		
1	When linking use libcs.a (the compact C library) rather than libc.a (the standard C library).		

-Xfastmath

-Xfastmath	function	0..1	0
0	No additional optimization.		
1	Enable additional floating point optimizations that may affect precision. Conversion of if statements to 'fsel'. Automatic use of VMX registers to avoid conversion between floating point, integer and VMX registers. This will reduce the number of Load Hit Store penalties. Replacement of 'fdiv' with an approximate divide and refinement. **Note:** This switch is similar to the GCC --fast-math switch, but the		

	optimizations it controls in SNC are different to those implemented by GCC.

Warning: If the last word in memory contains a scalar load or store that may be converted to an unaligned vector load or store, then this may cause a page fault. If you encounter this problem either do not use floating point optimization by setting -Xfastmath=0 or ensure that the end of the last block in memory given to the program is more than 16 bytes from the end of system allocated memory.

Note that this problem is only likely to occur if -Xassumecorrectalignment=0, indicating that there may be mis-aligned pointers used in the application. Correcting the mis-aligned pointer issues will enable the use of -Xassumecorrectalignment=1 and -Xfastmath=1 without the page fault problem.

-Xflow

-Xflow	function	0..1	0
0	Do not do control flow optimization.		
1	Do control flow optimization.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xfltconst

-Xfltconst	file	0|4|8	0
0	For C implement single precision floating point constants as indicated by fltconst=4.		
4	Implement single precision floating point constants with single precision accuracy.		
8	Implement single precision floating point constants with double precision accuracy when they are used in a context in which the value would be converted to double precision before being used, such as assignment to a double precision variable, or use as one operand of an operator whose other operand is a double precision variable, etc.		

-Xfltdbl

-Xfltdbl	function	0..2	2
0	For C in c=knr mode, perform arithmetic on float objects in single-precision, and do not convert float function arguments and return values to double. Files compiled in this mode cannot be correctly linked with files compiled using the normal K&R floating-point model.		
1	For C in c=knr mode, perform arithmetic on float objects in single-precision, but convert float function arguments and return values to double. Files compiled in this mode can be linked correctly with files compiled using the normal K&R floating-point model.		
2	For C in c=knr mode, perform arithmetic on float objects in double-precision,		

	and convert float function arguments and return values to double. This is the normal K&R floating-point model.

-Xfltedge

-Xfltedge	function	1..3	2
0	Reserved for future use.		
1	Do no optimization that changes the behavior of the program if non-numeric values occur and are used in quiet computations. (The implementation of this mode is not perfect in the SNC compiler. In some cases, comparisons are modified in a way that changes their behavior. For example, the expression (!(a>b)) is changed to (a<=b), which is incorrect if a and b are unordered.)		
2	Do optimizations that may change the behavior of the program if non-numeric values occur and are used in quiet computations, but do not optimize the special case of testing a variable for equality or non-equality to itself. (This mode is provided to permit normal optimization, but also to provide the ability to program a test for non-numeric values).		
3	Do optimizations that may change the behavior of the program if non-numeric values occur and are used in quiet computations.		

-Xfltfold

-Xfltfold	function	0..2	2
0	During compilation do not evaluate expressions involving floating-point constants.		
1	During compilation evaluate expressions involving floating-point constants and arithmetic operators, but do not evaluate expressions involving intrinsic functions applied to floating point constants.		
2	During compilation evaluate expressions involving floating-point constants.		

-Xforcevtbl

-Xforcevtbl	file	0..1	0
0	Do not force generation of C++ vtables.		
1	Forces generation of C++ vtables.		

-Xfprreserve

-Xfprreserve	line	list	empty list
list	Reserve floating point registers defined in *list*.		

-Xfusedmadd

-Xfusedmadd	routine	0..1	0
0	Do not use the floating point multiply-accumulate instructions.		
1	Enables the use of the floating point multiply-accumulate instructions, e.g. fmadds, fmsubs, fnmadds and fnmsubs. When multiply-accumulate instructions are used, the intermediate product is calculated to infinite precision and certain status bits in FPSCR are not set for the intermediate result. These may be undesirable in some circumstances. This control-variable provides similar functionality to the GCC switch -mfused-madd.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xg

-Xg	compilation	0..2	0
0	Do not include symbolic debugging information in the output files.		
1,2	Include symbolic debugging information in the output file for use by the SN symbolic debugger.		

-Xgnuversion

-Xgnuversion	compilation	400..500	411
N	Determine which version of GNU compiler SNC is compatible with (default is compatible with GCC 4.1.1). Use -Xgnuversion=402 for compatibility with GCC 4.0.2.		

-Xgprreserve

-Xgprreserve	line	list	empty list
list	Reserve general purpose integer registers defined in *list*.		

-Xhookentry

This switch allows you to specify the function name of the entry-profile calls generated by -Xhooktrace.

-Xhookentry	file	name	
name	Name of entry-profile call generated by -Xhooktrace.		

-Xhookexit

This switch allows you to specify the function name of the exit-profile calls generated by -Xhooktrace.

-Xhookexit	file	name	
name	Name of exit-profile call generated by –Xhooktrace.		

-Xhooktrace

This switch generates entry/exit-profile calls.

-Xhooktrace	file	0..3	0
0	No entry/exit calls generated (default).		
1	Generate entry calls to the entry-profile routine, but no exit calls.		
2	Generate exit calls to the exit-profile routine, but no entry calls.		
3	Generate entry and exit calls to the entry and exit-profile routines.		

By default the entry/exit functions that are called are:

```
extern void __cyg_profile_func_enter
                  (void *this_fn, void *call_site);
extern void __cyg_profile_func_exit
                  (void *this_fn, void *call_site);
```

These are the same names (and prototypes) as used by GCC for -finstrument-functions. The switch '-Xhooktrace=3' is equivalent to simply '-Xhooktrace', so the following two commands will generate analogous entry/exit-profile calls:

```
ps3ppusnc -S -Xhooktrace foo.c
ppu-lv2-gcc -S -finstrument-functions foo.c
```

You can also specify the names of the entry/exit functions via the -Xhookentry and -Xhookexit switches.

For example:

```
-Xhookentry=My_Profile_Entry_Routine
-Xhookexit=My_Profile_Exit_Routine
```

will generate calls on entry and exit to:

```
extern void My_Profile_Entry_Routine
                  (void *this_fn, void *call_site);
extern void My_Profile_Exit_Routine
                  (void *this_fn, void *call_site);
```

-Xhostarch

-Xhostarch	file	0..65536	32
32	Use 32-bit compiler.		
64	Use 64-bit compiler. Requires a 64-bit host operating system.		

-Xignoreeh

-Xignoreeh	function	0..1	0
0	Do not ignore exception handling constructs.		
1	Ignore exception handling constructs on the assumption that no exception will be thrown. If a program compiled with -Xignoreeh=1 actually throws an exception, it will not be caught. • The construct try { body .. } catch() {} is compiled assuming that the		

	try block cannot throw exceptions, so the exception handling is avoided. • Exception specifications are ignored. • No cleanup code is generated. • However explicit throw statements are compiled as usual. Note that -Xignoreeh=1 does not syntactically enable exception handling constructs. Therefore if a file contains explicit exception handling constructs, such as try, throw, etc., it needs -Xc+=exceptions before you can use -Xignoreeh=1. However, -Xignoreeh=1 can change the behavior of files with no explicit exception handling constructs, for example suppressing cleanup code. -Xignoreeh has no effect on the _NO_EX preprocessor symbol. This is defined without -Xc+=exceptions and not defined with -Xc+=exceptions; -Xignoreeh=1 does not change that rule.

-Xindexaddr

This switch can be used to suppress the use of base_reg+index_reg addressing mode in contexts where the compiler cannot guarantee that the sum of the two 64-bit registers will produce an effective address that has all zeros in the top 32 bits.

The following example shows where it can be useful:

```
extern int printf (const char *, ...);
unsigned int x = 0xffffffff;
char array[ 100 ];
int main()
{
  array[0] = 42;
  printf( "%d\n", array[ x + 1 ] );
  return 0;
}
```

In the default -Xindexaddr=1 mode, a base-register holding array+1 is used and x is loaded into an index register. That base+index addressing reference then adds array+1+4294967295 to compute the effective address, and ultimately accesses array[4294967296], causing a memory fault. According to the C/C++ Standards this is undefined behavior but if -Xindexaddr=0 is used instead, the two registers will be added together, the top 32 bits will be cleared, and the reference will be to the array[0], as expected.

Alternatively, this code could be made Standard-compliant by casting the unsigned int variable x in the array-reference to an int, as in:

```
array[ ((int) x) + 1 ]
```

With this cast, the reference will be to array[0] irrespective of the setting of -Xindexaddr.

-Xindexaddr	routine	0..1	1
0	Suppress the use of base+index addressing mode unless the compiler can guarantee that the sum will have all zeros in the top 32 bits.		
1	Allow the use of base+index addressing mode (default).		

-Xinline

-Xinline	line	list of names or pairs	empty list

name	Inline the named function, which can be either a source function or an intrinsic-function.
name:*n*	Inline the named function, applying *n* as a priority. The function can be either a source function or an intrinsic-function.

-Xinlinedebug

-Xinlinedebug	function	0..1	0
0	Hide the debugger's inlined function display (default).		
1	Generate information to display inlined functions. This will increase the size of the debug information.		

-Xinlinehotfactor

This switch is used in conjunction with __attribute__((hot)). If the calling function is tagged as hot, the value of this switch will control how much extra inlining is performed within it. The value of n specifies the factor by which the values of autoinlinesize and inlinesize are multiplied when considering inlining within the 'hot' function.

See "Marking a function as 'hot'".

-Xinlinehotfactor	function	1..100	5
1	No effect.		
N	Factor for multiplying autoinlinesize and inlinesize when inlining within the 'hot' function.		

-Xinlinemaxsize

This switch controls the maximum amount of inlining into any one function. It is used to prevent individual functions from becoming too large and slowing down other stages in the optimizers. Increasing this from the default value may increase the amount of inlining (and therefore possibly performance) at the expense of compilation speed.

The parameter to this control expresses the maximum size of a function in terms of 'instructions'. These are internal compiler instructions and are not necessarily the same as individual processor instructions.

-Xinlinemaxsize	function	0..50000	1000
0	No inlining.		
N	Allow inlining into functions up to a maximum size of *n* instructions.		

See also -Xinlinesize and -Xautoinlinesize.

> **Note:** this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xinlinesize

This switch limits the maximum size of explicitly inline functions that will be inlined by the compiler. Explicitly inline functions include C++ methods defined inside classes in header files.

The parameter to this control expresses the maximum size of a function in terms of 'instructions'. These are internal compiler instructions and are not necessarily the same as individual processor instructions.

-Xinlinesize	function	0..50000	0
0	No explicit inlining.		
n	Allow automatic inlining of explicitly inline functions up to a maximum size of *n* instructions.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

See also -Xautoinlinesize and -Xinlinemaxsize.

-Xintedge

-Xintedge	function	0..1	0
0	Assume that integer overflow can occur during integer operations. Do no optimization that would change the program behavior if it does occur.		
1	Assume that the effects of integer overflow during integer operations can be ignored in applying optimizations.		

-Xlinkoncesafe

-Xlinkoncesafe	file	0..1	0
0	Do not assume all link-once implementations are the same.		
1	Assume all link-once implementations are the same.		

-Xmathwarn

-Xmathwarn	line	off .. on	off
off	Do not warn if the compiler uses calls to maths emulation libraries.		
on	Warn if the compiler uses calls to maths emulation libraries.		

-Xmemlimit

-Xmemlimit	file	0..max int	512
n	Tells the optimizer how much memory should be assumed to be available (in KB).		

-Xmserrors

-Xmserrors	compilation	0..1	0

0	Do print source lines for errors and warnings.
1	Do not print source lines for errors and warnings.

-Xmultibytechars

-Xmultibytechars	file	0..1	0
0	No support for multibyte encoded source files.		
1	Allow the use of source files containing multibyte character sequences encoded using the UTF8 standard.		

-Xnewalign

-Xnewalign	function	0..64	16
n	This value determines the point at which the compiler will use the two-argument (aligned) form of operator new. Allocating an instance of a type with an alignment less than or equal to this threshold will use the single-argument standard "operator new(std::size_t)" function, whereas types with an alignment greater than this value will use the two-argument SCE extension "operator new (std::size_t, std::size_t)" function.		

-Xnoident

-Xnoident	compilation	0..1	0
0	Generate an entry for compiler version in the .comment section.		
1	Do not generate an entry for compiler version in the .comment section.		

-Xnoinline

-Xnoinline	line	list of names	empty list
name	Do not inline the named function, which can be either a source function or an intrinsic-function.		

-Xnosyswarn

-Xnosyswarn	file	0..1	1
0	System header files are header files that are in include directories implicitly known to the compiler, which is the set of directories inside $CELL_SDK. An include directory that is not implicitly known to the compiler may explicitly be identified as a 'system' include directory using a -J option, rather than a -I option.		
1	Suppress warnings issued from 'system' header files (this is roughly equivalent to GCC's -Wsystem-headers switch).		

-Xnotocrestore

-Xnotocrestore	function	0..2	0
0	The compiler generates fully ABI compliant code. The code to call a function through a pointer assumes that the value of the TOC register at the callee may be different from that of the caller. A nop instruction is generated after a call to an external function to allow the linker to restore the TOC pointer if the callee code resides in a different TOC region at link time. No special linker switches are necessary for code built with this option to run correctly. This is the default value of the notocrestore control.		
1	The compiler elides the nop instruction after a call to an external function but calls through pointers are guaranteed to be TOC-safe. The program must be linked with the SN linker --notocrestore switch.		
2	The compiler elides both the nop instruction after a call to an external function and assumes that a call through a pointer will always use the same TOC region. The program must be linked with the SN linker --notocrestore switch.		

-Xoptintrinsics

SNC will apply its optimization passes to code that is written using intrinsics, in the same way as it will with code written using C or C++. This allows the compiler to make a number of optimizations, combining intrinsics with surrounding C or C++ code in order to produce the most optimal sequence.

In some cases this will result in different instructions being generated to those expected from the intrinsics that have been used. If this behavior is not desired, it can be controlled with this switch.

-Xoptintrinsics	function	0..1	1
0	Disables optimization of code generated for intrinsics.		
1	Allows optimization of code generated for intrinsics.		

-Xparamrestrict

-Xparamrestrict	function	0..1	0
0	Does not decorate function parameters of pointer type with the __restrict qualifier.		
1	Automatically decorates function parameters of pointer type with the __restrict qualifier.		

-Xpch_override

-Xpch_override	file	0..1	0
0	Do not override the compiler's check that the file used to generate the pre-compiled header file is in the same directory as the file being compiled.		

1	Override the compiler's check that the file used to generate the pre-compiled header file is in the same directory as the file being compiled.

-Xpostopt

This switch controls a number of new optimizations based on data flow analysis.

-Xpostopt	function	0..6	0
0	No optimization (the default).		
1	(not used)		
2	Enables: - additional global constant folding and propagation		
3	Also enables: - elimination of zero/sign extends - simplification of load/store addresses - improved propagation of alias information		
4	Also enables: - collapsing of chains of loads and stores - removal of LHS dependencies - conversion of floating point comparisons to integer operations with shorter latency - removal of empty loops		
5	Also enables: - more aggressive load and store elimination		
6	Also enables: - further optimizations		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xpredefinedmacros

-Xpredefinedmacros	function	0..1	0
0	Do not display predefined macros, that is, defined by the SNC compiler front-end, during compilation.		
1	Display predefined macros during compilation.		

-Xpreprocess

-Xpreprocess	file	0..2	0
0	Do not generate pre-processed output.		
1	Generate pre-processed output. The compiler will output a file with a .i filename extension. You should either rename the file to match the input filename		

	extension, or else use the -Tp or -Tc command-line switches to indicate to the compiler that the file should be treated as either C++ or C source respectively.
2	As for value=1 but generate pre-processed output with source line information.

-Xprogress

-Xprogress	function	list of names	files
Files	Announce progress at the start of compiling each file.		
functions	Announce progress at the start of compiling each function.		
Phases	Announce progress at the start of each phase of the compiler.		
subphases	Announce progress at the start of each subphase of the compiler.		
Actions	Announce progress at each major action (e.g. inlining) done by the compiler.		
Failures	Announce progress at each major action (e.g. inlining) done by the compiler.		
templates	Announce instantiations of template functions.		
Memory	Include compiler memory-usage information in progress announcements.		
Sizes	Include information on the size of internal compiler data structures in progress announcements.		
Realtime	Include the realtime used by the compiler in progress announcements.		
Rtime	Same as realtime.		
Usertime	Include the usertime used by the compiler in progress announcements.		
Utime	Same as usertime.		
%all	Announce progress at all possible points of compilation.		
%none	Do not announce compiler progress.		

-Xquit

-Xquit	compilation	0..2	0
0	Exit abnormally (exit status=1) if error or fatal error messages were printed; exit normally otherwise.		
1	Exit abnormally (exit status=1) if warning or error or fatal error messages were printed; exit normally otherwise. Any messages that would usually be printed as "warning:" are instead printed as "error:".		
2	Exit abnormally (exit status=1) if remark or warning or error or fatal error messages were printed; exit normally otherwise. Any messages that would usually be printed as "warning:" or "remark:" are instead printed as "error:".		

-Xreg

-Xreg	function	0..2	0
0	Do not allocate register-candidate variables to registers.		
1	Allocate register-candidate variables to registers, and do global and local register allocation.		
2	Allocate register-candidate variables to registers, and do interprocedural and global and local register allocation, but without reordering functions for interprocedural allocation.		

Note: this control-variable is affected by the setting of the optimization control-group (`-O`). See "Optimization group (O)".

-Xrelaxalias

This switch controls the alias analysis rules.

Note:

- `-Xrelaxalias=0` is equivalent to GCC's `-fno-strict-aliasing`
- `-Xrelaxalias=2` is roughly equivalent to GCC's `-fstrict-aliasing`

We recommend that code be written or adapted to work with at least `-Xrelaxalias=2` (the C99 aliasing rules). The stricter aliasing rules enable the compiler to generate much better code in some cases.

Note that the `__may_alias` attribute may be used to mark deliberately aliasing pointers even when using the stricter aliasing rules. See "The __may_alias__ attribute".

-Xrelaxalias	function	0..3	0
0	Alias checking is not relaxed.		
1	Assume that type instances do not partially overlap.		
2	Use strict language aliasing rules according to section 6.5 of the ISO/ANSI C99 specification. That is, types that are not 'similar' to one another do not alias.		
3	Strict language aliasing rules, but additionally, const and non-const variables do not alias.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xreorder

-Xreorder	function	0..2	1
0	Do not reorder basic blocks.		
1	Reorder basic blocks only using `__builtin_expect` hints (if any).		
2	Full reordering of basic blocks based on `__builtin_expect` hints and heuristics for optimal execution flow.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xreserve

-Xreserve	file	list of registers	empty list
reg{+*reg*...}	SNC allows registers to be removed from normal register allocation and saving. These registers can be specified by specifying a list of registers in the following format: *reg1*{+*reg2*...} where *reg1*, *reg2* etc. are register identifiers, e.g. -Xreserve=r14 reserves register r14 for use as a global register. Functions will then avoid using r14. **Warning:** reserving registers that have specific uses under the PPU ABI will cause undefined results.		

-Xrestrict

-Xrestrict	function	0..2	1
0	Do not act on `__restrict` keywords. Assume pointer aliases with other pointers.		
1	Assume pointers qualified with `__restrict` do not alias other `__restrict` qualified pointers.		
2	Assume pointers qualified with `__restrict` do not alias any other pointers.		

-Xretpts

-Xretpts	function	0..1	1
n	Control number of return points in functions. For functions that have multiple 'return' statements, the default mode is to generate the function epilogue code inline to every return statement. Setting this switch to 1 selects an alternate mode where each return statement branches to a common function epilogue. Inlined epilogue code results in quicker execution but also results in larger overall code size.		

-Xretstruct

This is an optional ABI extension to return the results of functions returning classes or structs wrapping a single primitive type (int, float vector etc) in registers.

This optimization can have a dramatic effect on math libraries that use C++ classes to wrap VMX vectors.

-Xretstruct	function	0..2	0
0	No optimization (the default) - This is the standard ABI.		
1	Return wrapped vector types in registers.		
2	Return all wrapped primitive types in registers.		

Wrappers are structs or classes with no virtual functions that contain a single data member of a primitive type.
Example:

```
struct MyInt
{
    int mN;
};
class MyVector
{
    vector float mVec;
};
```

Warning: This optimization is not compatible with the standard ABI and must be used consistently across code that returns wrapper structs. Code that calls SDK or other library functions that return wrapper structs must be compiled with -Xretstruct=0.

-Xsaverestorefuncs

-Xsaverestorefuncs	function	0..1	0
0	Do not use save/restore millicode functions.		
1	Use save/restore millicode functions (similar to GCC's -muse-save-restore-funcs switch). This will replace the standard save/restore code at the beginning of certain functions with a branch to a standard function containing the necessary code sequence. This will generally produce smaller code at the expense of performance. Save and restore millicode functions are used automatically for functions marked __attribute__((cold)) regardless of the setting of -Xsaverestorefuncs. They are never used for functions marked __attribute__((hot)) regardless of the setting of -Xsaverestorefuncs or for functions marked __attribute__((inlinesrf)) regardless of any other factors mentioned above.		

-Xsched

-Xsched	function	0..2	0
0	Do not schedule instructions.		
1	Schedule instructions using pass 1 only.		
2	Schedule instructions using both passes.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xshow

-Xshow	line	list of names	empty list
names of control-variables	Display the value of each control-variable listed.		

-Xsingleconst

-Xsingleconst	file	0..1	0

0	Treat floating point constants without a 'f' postfix as double precision type (i.e. double).
1	Treat floating point constants without a 'f' postfix as single precision type (i.e. float).

-Xsizet

-Xsizet	compilation	name	uint
uint	size_t is unsigned int.		
ulong	size_t is unsigned long.		
ushort	size_t is unsigned short.		

-Xswbr

-Xswbr	compilation	0..1	1
0	For switch statements having consecutive case labels, generate a table of jump addresses and do an indirect jump for switch statements having at least 5 labels (provided the labels are close enough).		
1	For switch statements having consecutive case labels, forces the generation of compare-branch trees.		

-Xswmaxchain

-Xswmaxchain	function	0..100	8
n	Determines the maximum length of a decision tree, i.e. series of compare/goto instructions, generated for switch statements containing a large number of case labels. At higher values of -Xswmaxchain, the compiler will generate longer sequences of compare/goto instructions.		

-Xtrigraphs

-Xtrigraphs	file	0..1	0
0	Do not support use of trigraphs.		
1	Support use of trigraphs in code.		

-Xuninitwarn

-Xuninitwarn	file	0..1	1
0	Do not generate warning for potentially uninitialized variable usage from compiler backend.		

1	Generate warning for potentially uninitialized variable usage from compiler backend.

-Xunroll

-Xunroll	loop	0..max int	0
0	Do not unroll loops.		
1	Unroll loops under automatic control.		
n>1	Always unroll loops (that can be unrolled), and unroll them *n* times.		

-Xunrollssa

-Xunrollssa	function	0..100	0
0	Do not convert loops.		
n	Unroll loops to have a final body comprising only one basic block (where possible), which is *n* instructions long.		
10	Convert very small loops.		
30	Convert larger loops.		
100	Extreme loop unrolling. Unlikely to be useful for real code but may be of use for benchmarks. Real, non-benchmark code may run slower due to the additional checks introduced by loop unrolling.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xuseatexit

-Xuseatexit	function	0..1	0
0	Use static tables for calling destructors for static variables.		
1	Use a dynamically created linked list for calling destructors for static variables. Equivalent to GNU -fuse-cxa-atexit switch. This is required for fully standard compliant reverse order of invocation of destructors for static variables. This is marginally more expensive in terms of code space and run time, but is required to get the correct semantics, particularly in situations where the constructor for one static variable initializes another static variable before it exits. It is permissible to intermix object files compiled with and without -Xuseatexit. However in that case the destructors for all files compiled with -Xuseatexit will be called before those of files compiled without it. In particular SDK and middleware libraries are not compiled with this option. Therefore destructors for static variables in files compiled with -Xuseatexit will be called before such libraries.		

-Xuseintcmp

-Xuseintcmp	function	0..1	0
0	Do not convert compares.		
1	Convert compares to integer operations.		

Note: this control-variable is affected by the setting of the optimization control-group (-O). See "Optimization group (O)".

-Xwchart

-Xwchart	compilation	name	ushort
char	wchar_t is char.		
int	wchar_t is int.		
long	wchar_t is long.		
schar	wchar_t is signed char.		
short	wchar_t is short.		
uchar	wchar_t is unsigned char.		
uint	wchar_t is unsigned int.		
ulong	wchar_t is unsigned long.		
ushort	wchar_t is unsigned short.		

-Xwritable_strings

-Xwritable_ strings	line	0..2	0
0	Place strings into a read-only data section (e.g., .rodata).		
1	Place strings into a target- and language-dependent data section.		
2	Place strings into a writable data section (e.g., .data).		

-Xzeroinit

-Xzeroinit	function	0..1	0
0	Disables zero initialization of classes with compiler generated constructor before calls to constructor.		
1	Enables zero initialization of classes with compiler generated constructor before calls to constructor.		

Control-group reference tables

The following subsections each define one control-group.

Optimization group (O)

Group name = O, values = 0..3, d or s, default: same as O=2.

Control Name	Optimization Level					
	-O0	-O1	-O2	-O3	-Os	-Od
Alias	0	1	4	4	4	3
autoinlinesize	0	0	32	64	16	0
debuglocals	0	0	0	0	0	1
Flow	0	0	1	1	1	1
fusedmadd	0	1	1	1	1	1
inlinemaxsize	1000	1500	1500	1500	32	1000
inlinesize	0	16	256	256	64	32
postopt	0	0	6	6	6	6
Reg	0	0	3	3	3	1
relaxalias	0	0	2	2	2	0
reorder	0	0	1	2	1	0
Sched	0	0	2	2	2	0
unrollssa	0	0	10	16	10	0
useintcmp	0	0	0	0	0	0

These control-variables are discussed in "Optimization control-variables".

10: Intrinsic function reference

JSRE intrinsics

Note 1 - Support for JSRE intrinsics

Notes	Returns	Function	Defined in
Specify the address and direction of a read data stream. Will continue to load cache blocks starting at address	void	__dcbt_TH1000(void *address, unsigned direction, unsigned unlimited, unsigned id);	ppu_intrinsics.h
Control a stream started by __dcbt_TH1000. start and stop the stream, specify count and other flags.	void	__dcbt_TH1010(unsigned go, unsigned stop, unsigned unit_count, unsigned transient, unsigned unlimited, unsigned id);	ppu_intrinsics.h
Read the time base. Skip values with zero lower 32 bits.	long long	__mftb();	ppu_intrinsics.h
Invalidate a L2 instruction cache block	void	__icbi(void *ptr);	ppu_intrinsics.h
Invalidate a L2 data cache block	void	__dcbi(void *ptr);	ppu_intrinsics.h
Flush a L2 data cache block	void	__dcbf(void *ptr);	ppu_intrinsics.h
Zero a L2 data cache block.	void	__dcbz(void *ptr);	ppu_intrinsics.h
Write out a L2 data cache block	void	__dcbst(void *ptr);	ppu_intrinsics.h
Read a L2 data cache block ready for a store.	void	__dcbtst(void *ptr);	ppu_intrinsics.h
Read a L2 data cache block, non-streamed form.	void	__dcbt(void *ptr);	built in
Load and reserve atomic value. Used with stwcx for atomic operations.	unsigned	__lwarx(void *base);	ppu_intrinsics.h
Load and reserve atomic value. Used with stdcx for atomic operations.	unsigned long long	__ldarx(void *base);	ppu_intrinsics.h
Store atomic value only if another thread has not already done so.	bool	__stwcx(void *base, unsigned value);	ppu_intrinsics.h
Store atomic value only if another thread has not	bool	__stdcx(void *base, unsigned long long value);	ppu_intrinsics.h

already done so.			
Load 16 bit value and reverse bytes	unsigned int	__lhbrx(void *base);	ppu_intrinsics.h
Load 32-bit value and reverse bytes	unsigned int	__lwbrx(void *base);	ppu_intrinsics.h
Load 64-bit value and reverse bytes	unsigned long long	__ldbrx(void *base);	ppu_intrinsics.h
Store 16 bit value and reverse bytes	void	__sthbrx(void *base, unsigned short value);	ppu_intrinsics.h
Store 32-bit value and reverse bytes	void	__stwbrx(void *base, unsigned int value);	ppu_intrinsics.h
Store 64-bit value and reverse bytes	void	__stdbrx(void *base, unsigned long long value);	ppu_intrinsics.h
Count leading zeros, 64-bit	unsigned long long	__cntlzd(long long a);	built in
Count leading zeros, 32-bit	long long	__cntlzw(long long a);	built in
Heavyweight data sync ensures all writes complete	void	__sync();	ppu_intrinsics.h
Instruction sync used before modifying code.	void	__isync();	ppu_intrinsics.h
Light weight memory sync.	void	__lwsync();	ppu_intrinsics.h
Heavyweight sync for memory mapped I/O	void	__eieio();	ppu_intrinsics.h
Convert double value to 64- bit integer	long long	__fctid(double a);	built in
Convert double value to 32- bit integer	long long	__fctiw(double a);	built in
Convert 64-bit value to double	double	__fcfid(long long a);	built in
Move from floating point status	double	__mffs();	ppu_intrinsics.h
Move to floating point status with mask	void	__mtfsf(int mask, double value);	ppu_intrinsics.h
Move to floating point status, immediate	void	__mtfsfi(int bits, int field);	ppu_intrinsics.h
Clear floating point status bit	void	__mtfsb0(int bit);	ppu_intrinsics.h
Set floating point status bit	void	__mtfsb1(int bit);	ppu_intrinsics.h
Set floating point status,	double	__setflm(double a);	ppu_intrinsics.h

return old value			
Rotate left and insert, 64-bit	long long	__rldimi(long long a, long long b, unsigned char sh, unsigned char mb);	ppu_intrinsics.h
Rotate left and clear, 64-bit	long long	__rldic(long long a, unsigned char sh, unsigned char mb);	ppu_intrinsics.h
Rotate left and clear left, 64- bit	long long	__rldicl(long long a, unsigned char sh, unsigned char mb);	ppu_intrinsics.h
Rotate left and clear right, 64-bit	long long	__rldicr(long long a, unsigned char sh, unsigned char me);	ppu_intrinsics.h
Rotate left and clear right, 64-bit microcoded version	long long	__rldcr(long long a, long long sh, unsigned char me);	ppu_intrinsics.h
Rotate left and clear left, 64- bit microcoded version	long long	__rldcl(long long a, long long sh, unsigned char mb);	ppu_intrinsics.h
Rotate left and insert, 32-bit	unsigned	__rlwimi(long long a, long long b, unsigned char sh, unsigned char mb, unsigned char me);	ppu_intrinsics.h
Rotate left and insert, 32-bit	unsigned	__rlwinm(long long a, unsigned char sh, unsigned char mb, unsigned char me);	ppu_intrinsics.h
Rotate left and insert, 32-bit microcoded version	unsigned	__rlwnm(long long a, long long sh, unsigned char mb, unsigned char me);	ppu_intrinsics.h
Note 1	void	__cctph( );	built in
Note 1	void	__cctpl( );	built in
Note 1	void	__cctpm( );	built in
Note 1	unsigned long long	__cntlzd(unsigned long long);	built in
Note 1	unsigned long long	__cntlzw(unsigned long long);	built in
Note 1	void	__db10cyc( );	built in
Note 1	void	__db12cyc( );	built in
Note 1	void	__db16cyc( );	built in
Note 1	void	__db8cyc( );	built in
Note 1	void	__dcbt(const void *);	built in
Note 1	double	__fabs(double);	built in
Note 1	float	__fabsf(float);	built in

Note 1	double	__fctid(double);	built in
Note 1	double	__fctiw(double);	built in
Note 1	double	__fsel(double, double, double);	built in
Note 1	float	__fsqrts(float);	built in
Note 1	unsigned long long	__mfspr(int);	built in
Note 1	unsigned long long	__mftb( );	built in
Note 1	void	__nop( );	built in

SNC/GCC intrinsics

Note 2 - PPU instruction for asm translation. See 64-bit PEM.

Note 3 - Internal intrinsics used by GCC to implement altivec.h.

Notes	Returns	Function	Defined in
Note 2	long long	__addc(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__adde(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__addic(long long a, short b);	ppu_asm_intrinsics.h
Note 2	long long	__addme(long long a);	ppu_asm_intrinsics.h
Note 2	long long	__addze(long long a);	ppu_asm_intrinsics.h
Note 2	long long	__subfc(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__subfe(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__subfme(long long a);	ppu_asm_intrinsics.h
Note 2	long long	__subfze(long long a);	ppu_asm_intrinsics.h
Note 2	long long	__subfic(long long a, const short b);	ppu_asm_intrinsics.h
Note 2	long long	__srad(long long a, long long b);	ppu_asm_intrinsics.h
Note	long long	__sradi(long long a, unsigned char b);	ppu_asm_intrinsics.h

2			
Note 2	long long	__sraw(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__srawi(long long a, unsigned char b);	ppu_asm_intrinsics.h
Note 2	long long	__add(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__addi(long long a, short b);	ppu_asm_intrinsics.h
Note 2	long long	__addis(long long a, short b);	ppu_asm_intrinsics.h
Note 2	long long	__subf(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__neg(long long a);	ppu_asm_intrinsics.h
Note 2	long long	__divd(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__divdu(unsigned long long a, unsigned long long b);	ppu_asm_intrinsics.h
Note 2	long long	__divw(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__divwu(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__mulhd(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__mulhdu(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__mulhw(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__mulhwu(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__mulld(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__mulli(long long a, short b);	ppu_asm_intrinsics.h
Note 2	long long	__mullw(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__extsb(long long a);	ppu_asm_intrinsics.h

Note 2	long long	__extsh(long long a);	ppu_asm_intrinsics.h
Note 2	long long	__extsw(long long a);	ppu_asm_intrinsics.h
Note 2	long long	__and(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__andc(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__eqv(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__nand(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__nor(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__or(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__orc(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__ori(long long a, unsigned short b);	ppu_asm_intrinsics.h
Note 2	long long	__oris(long long a, unsigned short b);	ppu_asm_intrinsics.h
Note 2	long long	__xor(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__xori(long long a, const unsigned short b);	ppu_asm_intrinsics.h
Note 2	long long	__xoris(long long a, const unsigned short b);	ppu_asm_intrinsics.h
Note 2	double	__fadd(double a, double b);	ppu_asm_intrinsics.h
Note 2	double	__fadds(double a, double b);	ppu_asm_intrinsics.h
Note 2	double	__fdiv(double a, double b);	ppu_asm_intrinsics.h
Note 2	double	__fdivs(double a, double b);	ppu_asm_intrinsics.h
Note 2	double	__fmadd(double a, double b, double c);	ppu_asm_intrinsics.h
Note 2	double	__fmadds(double a, double b, double c);	ppu_asm_intrinsics.h

Note 2	double	__fmr(double b);	ppu_asm_intrinsics.h
Note 2	double	__fmsubs(double a, double b, double c);	ppu_asm_intrinsics.h
Note 2	double	__fmsub(double a, double b, double c);	ppu_asm_intrinsics.h
Note 2	double	__fmul(double a, double b);	ppu_asm_intrinsics.h
Note 2	double	__fmuls(double a, double b);	ppu_asm_intrinsics.h
Note 2	double	__fnabs(double a);	ppu_asm_intrinsics.h
Note 2	double	__fnabsf(double a);	ppu_asm_intrinsics.h
Note 2	double	__fneg(double a);	ppu_asm_intrinsics.h
Note 2	double	__fnmadd(double a, double b, double c);	ppu_asm_intrinsics.h
Note 2	double	__fnmadds(double a, double b, double c);	ppu_asm_intrinsics.h
Note 2	double	__fnmsub(double a, double b, double c);	ppu_asm_intrinsics.h
Note 2	double	__fnmsubs(double a, double b, double c);	ppu_asm_intrinsics.h
Note 2	float	__fres(float a);	ppu_asm_intrinsics.h
Note 2	double	__fsqrt(double a);	ppu_asm_intrinsics.h
Note 2	double	__frsp(double a);	ppu_asm_intrinsics.h
Note 2	float	__fsels(float a, float b, float c);	ppu_asm_intrinsics.h
Note 2	double	__frsqrte(double x);	ppu_asm_intrinsics.h
Note 2	double	__fsub(double a, double b);	ppu_asm_intrinsics.h
Note 2	double	__fsubs(double a, double b);	ppu_asm_intrinsics.h
Note 2	long long	__fctiwz(double a);	ppu_asm_intrinsics.h

Note 2	long long	__lbz(const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	long long	__lbzx(void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	long long	__ld(const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	long long	__ldx(void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	double	__lfd(const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	double	__lfdx(void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	double	__lfs(const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	double	__lfsx(void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	long long	__lha(const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	long long	__lhax(void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	long long	__lhz(const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	long long	__lhzx(void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	long long	__lwa(const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	long long	__lwax(void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	long long	__lwz(const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	long long	__lwzx(void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	long long	__sld(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__sldi(long long a, unsigned char b);	ppu_asm_intrinsics.h
Note 2	long long	__slw(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__slwi(long long a, unsigned char b);	ppu_asm_intrinsics.h

Note 2	long long	__srd(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__srdi(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__srw(long long a, long long b);	ppu_asm_intrinsics.h
Note 2	long long	__srwi(long long a, unsigned char b);	ppu_asm_intrinsics.h
Note 2	void	__stb(long long a, const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	void	__stbx(long long a, void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	void	__std(long long a, const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	void	__stdx(long long a, void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	void	__stfd(double a, const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	void	__stfdx(double a, void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	void	__stfs(double a, const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	void	__stfsx(double a, void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	void	__sth(long long a, const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	void	__sthx(long long a, void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	void	__stw(long long a, const short offset, void *p);	ppu_asm_intrinsics.h
Note 2	void	__stwx(long long a, void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	void	__stfiwx(double a, void *p, long long offset);	ppu_asm_intrinsics.h
Note 2	unsigned	__lbzu(int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	unsigned	__ldu(int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	double	__lfdu(int offset, void *&p);	ppu_asm_intrinsics.h

Note 2	float	__lfsu(int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	unsigned	__lhau(int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	unsigned	__lhzu(int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	unsigned	__lwau(int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	unsigned	__lwzu(int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	void	__stbu(long long value, int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	void	__stdu(long long value, int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	void	__stfdu(long long value, int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	void	__stfsu(long long value, int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	void	__sthu(long long value, int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	void	__stwu(long long value, int offset, void *&p);	ppu_asm_intrinsics.h
Note 2	unsigned	__lbzux(void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	unsigned	__ldux(void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	double	__lfdux(void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	float	__lfsux(void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	unsigned	__lhaux(void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	unsigned	__lhzux(void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	unsigned	__lwaux(void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	unsigned	__lwzux(void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	void	__stbux(long long value, void *&p, int offset);	ppu_asm_intrinsics.h

Note 2	void	__stdux(long long value, void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	void	__stfdux(long long value, void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	void	__stfsux(long long value, void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	void	__sthux(long long value, void *&p, int offset);	ppu_asm_intrinsics.h
Note 2	void	__stwux(long long value, void *&p, int offset);	ppu_asm_intrinsics.h
Note 3	vector signed int	__builtin_altivec_vaddcuw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vaddfp(vector float a, vector float b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vaddsbs(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vaddshs(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vaddsws(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vaddubm(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vaddubs(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vadduhm(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vadduhs(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vadduwm(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vadduws(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vand(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vandc(vector signed int a, vector signed int b);	ppu_altivec_internals.h

Note 3	vector signed char	__builtin_altivec_vavgsb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vavgsh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vavgsw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vavgub(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vavguh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vavguw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vcfsx(vector signed int a, const int b);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vcfux(vector signed int a, const int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vcmpbfp(vector float a, vector float b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vcmpeqfp(vector float a, vector float b);	ppu_altivec_internals.h
Note 3	vector bool char	__builtin_altivec_vcmpequb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector bool short	__builtin_altivec_vcmpequh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector bool int	__builtin_altivec_vcmpequw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vcmpgefp(vector float a, vector float b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vcmpgtfp(vector float a, vector float b);	ppu_altivec_internals.h
Note 3	vector bool char	__builtin_altivec_vcmpgtsb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector bool short	__builtin_altivec_vcmpgtsh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector bool int	__builtin_altivec_vcmpgtsw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note	vector bool	__builtin_altivec_vcmpgtub(vector signed char	ppu_altivec_internals.

3	char	a, vector signed char b);	h
Note 3	vector bool short	__builtin_altivec_vcmpgtuh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector bool int	__builtin_altivec_vcmpgtuw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vctsxs(vector float a, unsigned char b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vctuxs(vector float a, unsigned char b);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vexptefp(vector float a);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vlogefp(vector float a);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vmaddfp(vector float a, vector float b, vector float c);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vmaxfp(vector float a, vector float b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vmaxsb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmaxsh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmaxsw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vmaxub(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmaxuh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmaxuw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmhaddshs(vector signed short a, vector signed short b, vector signed short c);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmhraddshs(vector signed short a, vector signed short b, vector signed short c);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vminfp(vector float a, vector float b);	ppu_altivec_internals.h

Note 3	vector signed char	__builtin_altivec_vminsb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vminsh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vminsw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vminub(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vminuh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vminuw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmladduhm(vector signed short a, vector signed short b, vector signed short c);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vmrghb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmrghh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmrghw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vmrglb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmrglh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmrglw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmsummbm(vector signed char a, vector signed char b, vector signed int c);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmsumshm(vector signed short a, vector signed short b, vector signed int c);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmsumshs(vector signed short a, vector signed short b, vector signed int c);	ppu_altivec_internals.h

Note 3	vector signed int	__builtin_altivec_vmsumubm(vector signed char a, vector signed char b, vector signed int c);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmsumuhm(vector signed short a, vector signed short b, vector signed int c);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmsumuhs(vector signed short a, vector signed short b, vector signed int c);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmulesb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmulesh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmuleub(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmuleuh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmulosb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmulosh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vmuloub(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vmulouh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vnmsubfp(vector float a, vector float b, vector float c);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vnor(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vor(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vperm_4si(vector signed int a, vector signed int b, vector signed char c);	ppu_altivec_internals.h
Note 3	vector pixel	__builtin_altivec_vpkpx(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vpkshss(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed	__builtin_altivec_vpkshus(vector signed short a, vector signed short b);	ppu_altivec_internals.h

	char		
Note 3	vector signed short	__builtin_altivec_vpkswss(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vpkswus(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vpkuhum(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vpkuhus(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vpkuwum(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vpkuwus(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vrefp(vector float a);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vrfim(vector float a);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vrfin(vector float a);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vrfip(vector float a);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vrfiz(vector float a);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vrlb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vrlh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vrlw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vrsqrtefp(vector float a);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsel_4si(vector signed int a, vector signed int b, vector signed int c);	ppu_altivec_internals.h
Note	vector	__builtin_altivec_vsl(vector signed int a, vector	ppu_altivec_internals.

3	signed int	signed int b);	h
Note 3	vector signed char	__builtin_altivec_vslb(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsldoi_4si(vector signed int a, vector signed int b, unsigned char c);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vslh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vslo(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vslw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vspltb(vector signed char a, unsigned char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vsplth(vector signed short a, unsigned char b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vspltisb(signed char a);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vspltish(signed char a);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vspltisw(signed char a);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vspltw(vector signed int a, unsigned char b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsr(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vsrab(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vsrah(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsraw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vsrb(vector signed char a, vector signed char b);	ppu_altivec_internals.h

Note 3	vector signed short	__builtin_altivec_vsrh(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsro(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsrw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsubcuw(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_vsubfp(vector float a, vector float b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vsubsbs(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vsubshs(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsubsws(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vsububm(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_vsububs(vector signed char a, vector signed char b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vsubuhm(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vsubuhs(vector signed short a, vector signed short b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsubuwm(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsubuws(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsum2sws(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsum4sbs(vector signed char a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vsum4shs(vector signed short a, vector signed int b);	ppu_altivec_internals.h
Note	vector	__builtin_altivec_vsum4ubs(vector signed char	ppu_altivec_internals.

3	signed int	a, vector signed int b);	h
Note 3	vector signed int	__builtin_altivec_vsumsws(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vupkhpx(vector signed short a);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vupkhsb(vector signed char a);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vupkhsh(vector signed short a);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vupklpx(vector signed short a);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_vupklsb(vector signed char a);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vupklsh(vector signed short a);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_vxor(vector signed int a, vector signed int b);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_lvebx(long long offset, void *p);	ppu_altivec_internals.h
Note 3	vector signed short	__builtin_altivec_lvehx(long long offset, void *p);	ppu_altivec_internals.h
Note 3	vector signed int	__builtin_altivec_lvewx(long long offset, void *p);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_lvlx(long long offset, void *p);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_lvlxl(long long offset, void *p);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_lvrx(long long offset, void *p);	ppu_altivec_internals.h
Note 3	vector float	__builtin_altivec_lvrxl(long long offset, void *p);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_lvsl(long long offset, void *p);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_lvsr(long long offset, void *p);	ppu_altivec_internals.h

Note 3	vector signed char	__builtin_altivec_lvx(long long offset, void *p);	ppu_altivec_internals.h
Note 3	vector signed char	__builtin_altivec_lvxl(long long offset, void *p);	ppu_altivec_internals.h
Note 3	void	__builtin_altivec_stvebx(vector signed char a, long long offset, void *p);	ppu_altivec_internals.h
Note 3	void	__builtin_altivec_stvehx(vector signed short a, long long offset, void *p);	ppu_altivec_internals.h
Note 3	void	__builtin_altivec_stvewx(vector signed int a, long long offset, void *p);	ppu_altivec_internals.h
Note 3	void	__builtin_altivec_stvlx(vector signed char a, long long offset, void *p);	ppu_altivec_internals.h
Note 3	void	__builtin_altivec_stvlxl(vector signed char a, long long offset, void *p);	ppu_altivec_internals.h
Note 3	void	__builtin_altivec_stvrx(vector signed char a, long long offset, void *p);	ppu_altivec_internals.h
Note 3	void	__builtin_altivec_stvrxl(vector signed char a, long long offset, void *p);	ppu_altivec_internals.h
Note 3	void	__builtin_altivec_stvx(vector signed int a, long long offset, void *p);	ppu_altivec_internals.h
Note 3	void	__builtin_altivec_stvxl(vector signed int a, long long offset, void *p);	ppu_altivec_internals.h

SNC intrinsics

The following SNC intrinsics are all built in to the compiler.

Note 4 - Gcc __builtin equivalent

Notes	Returns	Function
Note 4	unsigned int	__builtin_cellAtomicAdd32(unsigned int *, unsigned int);
Note 4	unsigned long long	__builtin_cellAtomicAdd64(unsigned long long *, unsigned long long);
Note 4	unsigned int	__builtin_cellAtomicAnd32(unsigned int *, unsigned int);
Note 4	unsigned long long	__builtin_cellAtomicAnd64(unsigned long long *, unsigned long long);
Note 4	unsigned int	__builtin_cellAtomicCompareAndSwap32(unsigned int *, unsigned int, unsigned int);
Note 4	unsigned long long	__builtin_cellAtomicCompareAndSwap64(unsigned

		long long *, unsigned long long, unsigned long long);
Note 4	unsigned int	__builtin_cellAtomicDecr32(unsigned int *);
Note 4	unsigned long long	__builtin_cellAtomicDecr64(unsigned long long *);
Note 4	unsigned int	__builtin_cellAtomicIncr32(unsigned int *);
Note 4	unsigned long long	__builtin_cellAtomicIncr64(unsigned long long *);
Note 4	unsigned int	__builtin_cellAtomicLockLine32(unsigned int *);
Note 4	unsigned long long	__builtin_cellAtomicLockLine64(unsigned long long *);
Note 4	unsigned int	__builtin_cellAtomicNop32(unsigned int *);
Note 4	unsigned long long	__builtin_cellAtomicNop64(unsigned long long *);
Note 4	unsigned int	__builtin_cellAtomicOr32(unsigned int *, unsigned int);
Note 4	unsigned long long	__builtin_cellAtomicOr64(unsigned long long *, unsigned long long);
Note 4	unsigned int	__builtin_cellAtomicStore32(unsigned int *, unsigned int);
Note 4	unsigned long long	__builtin_cellAtomicStore64(unsigned long long *, unsigned long long);
Note 4	unsigned int	__builtin_cellAtomicStoreConditional32(unsigned int *, unsigned int);
Note 4	unsigned int	__builtin_cellAtomicStoreConditional64(unsigned long long *, unsigned long long);
Note 4	unsigned int	__builtin_cellAtomicSub32(unsigned int *, unsigned int);
Note 4	unsigned long long	__builtin_cellAtomicSub64(unsigned long long *, unsigned long long);
Note 4	unsigned int	__builtin_cellAtomicTestAndDecr32(unsigned int *);
Note 4	unsigned long long	__builtin_cellAtomicTestAndDecr64(unsigned long long *);
Note 4	int	__builtin_clz(int);
Note 4	unsigned long long	__builtin_clzl(unsigned long long);
Note 4	unsigned long long	__builtin_clzll(unsigned long long);
Note 4	int	__builtin_constant_p(int);
Note 4	void	__builtin_dcbf(const void *, int);
Note 4	void	__builtin_dcbi(void *, int);
Note 4	void	__builtin_dcbst(const void *, int);
Note 4	void	__builtin_dcbt(const void *, int);

Note 4	void	__builtin_dcbt3(unsigned int, int, int);
Note 4	void	__builtin_dcbtst(void *, long long);
Note 4	void	__builtin_dcbz(void *, int);
Note 4	void	__builtin_eieio( );
Note 4	int	__builtin_expect(int, int);
Note 4	double	__builtin_fabs(double);
Note 4	float	__builtin_fabsf(float);
Note 4	double	__builtin_fcfid(double);
Note 4	double	__builtin_fctid(double);
Note 4	double	__builtin_fctidz(double);
Note 4	double	__builtin_fctiw(double);
Note 4	double	__builtin_fctiwz(double);
Note 4	void	__builtin_fence( );
Note 4	double	__builtin_fmadd(double, double, double);
Note 4	float	__builtin_fmadds(float, float, float);
Note 4	double	__builtin_fmsub(double, double, double);
Note 4	float	__builtin_fmsubs(float, float, float);
Note 4	double	__builtin_fnabs(double);
Note 4	float	__builtin_fnabsf(float);
Note 4	double	__builtin_fnmadd(double, double, double);
Note 4	float	__builtin_fnmadds(float, float, float);
Note 4	double	__builtin_fnmsub(double, double, double);
Note 4	float	__builtin_fnmsubs(float, float, float);
Note 4	void *	__builtin_frame_address( );
Note 4	double	__builtin_fre(double);
Note 4	double	__builtin_frsqrte(double);
Note 4	float	__builtin_frsqrtes(float);
Note 4	double	__builtin_fsel(double, double, double);
Note 4	float	__builtin_fsels(float, float, float);
Note 4	double	__builtin_fsqrt(double);
Note 4	float	__builtin_fsqrts(float);

Note 4	long	__builtin_get_toc( );
Note 4	void	__builtin_icbi(void *, long long);
Note 4	void	__builtin_isync( );
Note 4	long long	__builtin_ldarx(void *, long long);
Note 4	unsigned int	__builtin_ldbrx(const void *, int);
Note 4	unsigned int	__builtin_lhbrx(const void *, int);
Note 4	unsigned int	__builtin_lwarx(void *, long long);
Note 4	unsigned int	__builtin_lwbrx(const void *, int);
Note 4	void	__builtin_lwsync( );
Note 4	void	__builtin_mb( );
Note 4	volatile double	__builtin_mffs( );
Note 4	unsigned long long	__builtin_mftb( );
Note 4	long long	__builtin_mtfsb0(int);
Note 4	long long	__builtin_mtfsb1(int);
Note 4	void	__builtin_mtfsf(int, double);
Note 4	void	__builtin_mtfsfi(int, int);
Note 4	long long	__builtin_mulhd(long long, long long);
Note 4	long long	__builtin_mulhdu(long long, long long);
Note 4	long long	__builtin_mulhw(long long, long long);
Note 4	long long	__builtin_mulhwu(long long, long long);
Get lower 32 bits of time base register	unsigned int	__builtin_raw_mftb( );
Note 4	void *	__builtin_return_address( );
Note 4	double	__builtin_setflm(double);
Note 4	void	__builtin_snpause( );
Note 4	void	__builtin_stdbrx(unsigned int, void *, int);
Note 4	int	__builtin_stdcx(unsigned long long, void *, long long);
Note 4	void	__builtin_stfiwx(double, void *, int);
Note 4	void	__builtin_sthbrx(unsigned short, void *, int);
Note 4	void	__builtin_stop( );
Note 4	void	__builtin_stwbrx(unsigned int, void *, int);

Note 4	int	__builtin_stwcx(unsigned int, void *, long long);
Note 4	void	__builtin_sync( );
Note 4	void	__builtin_trap( );
Note 4	void	__cctph( );
Note 4	void	__cctpl( );
Note 4	void	__cctpm( );
Note 4	unsigned long long	__cntlzd(unsigned long long);
Note 4	unsigned long long	__cntlzw(unsigned long long);
Note 4	void	__db10cyc( );
Note 4	void	__db12cyc( );
Note 4	void	__db16cyc( );
Note 4	void	__db8cyc( );
Note 4	void	__dcbt(const void *);
Note 4	double	__fabs(double);
Note 4	float	__fabsf(float);
Note 4	double	__fctid(double);
Note 4	double	__fctiw(double);
Note 4	double	__fsel(double, double, double);
Note 4	float	__fsqrts(float);
Note 4	unsigned long long	__mfspr(int);
Note 4	unsigned long long	__mftb( );
Note 4	void	__nop( );
Get register value immediately after call / system call	unsigned long long	__reg(int);

Altivec intrinsics

See Altivec Programming Interface Manual (PIM). The following Altivec intrinsics are all built in to the compiler.

Returns	Function
vector float	vec_abs(vector float);

vector signed short	vec_abs(vector signed short);
vector signed int	vec_abs(vector signed int);
vector signed char	vec_abs(vector signed char);
vector signed short	vec_abss(vector signed short);
vector signed int	vec_abss(vector signed int);
vector signed char	vec_abss(vector signed char);
vector float	vec_add(vector float, vector float);
vector signed char	vec_add(vector bool char, vector signed char);
vector unsigned char	vec_add(vector bool char, vector unsigned char);
vector signed char	vec_add(vector signed char, vector bool char);
vector signed char	vec_add(vector signed char, vector signed char);
vector unsigned char	vec_add(vector unsigned char, vector bool char);
vector unsigned char	vec_add(vector unsigned char, vector unsigned char);
vector signed short	vec_add(vector bool short, vector signed short);
vector unsigned short	vec_add(vector bool short, vector unsigned short);
vector signed short	vec_add(vector signed short, vector bool short);
vector signed short	vec_add(vector signed short, vector signed short);
vector unsigned short	vec_add(vector unsigned short, vector bool short);
vector unsigned short	vec_add(vector unsigned short, vector unsigned short);
vector signed int	vec_add(vector bool long, vector signed int);
vector unsigned int	vec_add(vector bool long, vector unsigned int);
vector signed int	vec_add(vector signed int, vector bool long);
vector signed int	vec_add(vector signed int, vector signed int);
vector unsigned int	vec_add(vector unsigned int, vector bool long);
vector unsigned int	vec_add(vector unsigned int, vector unsigned int);
vector unsigned int	vec_addc(vector unsigned int, vector unsigned int);
vector signed char	vec_adds(vector bool char, vector signed char);
vector signed char	vec_adds(vector signed char, vector bool char);
vector signed char	vec_adds(vector signed char, vector signed char);
vector signed short	vec_adds(vector bool short, vector signed short);
vector signed short	vec_adds(vector signed short, vector bool short);

vector signed short	vec_adds(vector signed short, vector signed short);
vector signed int	vec_adds(vector bool long, vector signed int);
vector signed int	vec_adds(vector signed int, vector bool long);
vector signed int	vec_adds(vector signed int, vector signed int);
vector unsigned char	vec_adds(vector bool char, vector unsigned char);
vector unsigned char	vec_adds(vector unsigned char, vector bool char);
vector unsigned char	vec_adds(vector unsigned char, vector unsigned char);
vector unsigned short	vec_adds(vector bool short, vector unsigned short);
vector unsigned short	vec_adds(vector unsigned short, vector bool short);
vector unsigned short	vec_adds(vector unsigned short, vector unsigned short);
vector unsigned int	vec_adds(vector bool long, vector unsigned int);
vector unsigned int	vec_adds(vector unsigned int, vector bool long);
vector unsigned int	vec_adds(vector unsigned int, vector unsigned int);
int	vec_all_eq(vector bool short, vector bool short);
int	vec_all_eq(vector bool short, vector signed short);
int	vec_all_eq(vector bool short, vector unsigned short);
int	vec_all_eq(vector bool long, vector bool long);
int	vec_all_eq(vector bool long, vector signed int);
int	vec_all_eq(vector bool long, vector unsigned int);
int	vec_all_eq(vector bool char, vector bool char);
int	vec_all_eq(vector bool char, vector signed char);
int	vec_all_eq(vector bool char, vector unsigned char);
int	vec_all_eq(vector float, vector float);
int	vec_all_eq(vector pixel, vector pixel);
int	vec_all_eq(vector signed short, vector bool short);
int	vec_all_eq(vector signed short, vector signed short);
int	vec_all_eq(vector signed int, vector bool long);
int	vec_all_eq(vector signed int, vector signed int);
int	vec_all_eq(vector signed char, vector bool char);
int	vec_all_eq(vector signed char, vector signed char);
int	vec_all_eq(vector unsigned short, vector bool short);

int	vec_all_eq(vector unsigned short, vector unsigned short);
int	vec_all_eq(vector unsigned int, vector bool long);
int	vec_all_eq(vector unsigned int, vector unsigned int);
int	vec_all_eq(vector unsigned char, vector bool char);
int	vec_all_eq(vector unsigned char, vector unsigned char);
int	vec_all_ge(vector bool short, vector signed short);
int	vec_all_ge(vector bool short, vector unsigned short);
int	vec_all_ge(vector bool long, vector signed int);
int	vec_all_ge(vector bool long, vector unsigned int);
int	vec_all_ge(vector bool char, vector signed char);
int	vec_all_ge(vector bool char, vector unsigned char);
int	vec_all_ge(vector float, vector float);
int	vec_all_ge(vector signed short, vector bool short);
int	vec_all_ge(vector signed short, vector signed short);
int	vec_all_ge(vector signed int, vector bool long);
int	vec_all_ge(vector signed int, vector signed int);
int	vec_all_ge(vector signed char, vector bool char);
int	vec_all_ge(vector signed char, vector signed char);
int	vec_all_ge(vector unsigned short, vector bool short);
int	vec_all_ge(vector unsigned short, vector unsigned short);
int	vec_all_ge(vector unsigned int, vector bool long);
int	vec_all_ge(vector unsigned int, vector unsigned int);
int	vec_all_ge(vector unsigned char, vector bool char);
int	vec_all_ge(vector unsigned char, vector unsigned char);
int	vec_all_gt(vector bool short, vector signed short);
int	vec_all_gt(vector bool short, vector unsigned short);
int	vec_all_gt(vector bool long, vector signed int);
int	vec_all_gt(vector bool long, vector unsigned int);
int	vec_all_gt(vector bool char, vector signed char);
int	vec_all_gt(vector bool char, vector unsigned char);
int	vec_all_gt(vector float, vector float);

int	vec_all_gt(vector signed short, vector bool short);
int	vec_all_gt(vector signed short, vector signed short);
int	vec_all_gt(vector signed int, vector bool long);
int	vec_all_gt(vector signed int, vector signed int);
int	vec_all_gt(vector signed char, vector bool char);
int	vec_all_gt(vector signed char, vector signed char);
int	vec_all_gt(vector unsigned short, vector bool short);
int	vec_all_gt(vector unsigned short, vector unsigned short);
int	vec_all_gt(vector unsigned int, vector bool long);
int	vec_all_gt(vector unsigned int, vector unsigned int);
int	vec_all_gt(vector unsigned char, vector bool char);
int	vec_all_gt(vector unsigned char, vector unsigned char);
int	vec_all_in(vector float, vector float);
int	vec_all_le(vector bool short, vector signed short);
int	vec_all_le(vector bool short, vector unsigned short);
int	vec_all_le(vector bool long, vector signed int);
int	vec_all_le(vector bool long, vector unsigned int);
int	vec_all_le(vector bool char, vector signed char);
int	vec_all_le(vector bool char, vector unsigned char);
int	vec_all_le(vector float, vector float);
int	vec_all_le(vector signed short, vector bool short);
int	vec_all_le(vector signed short, vector signed short);
int	vec_all_le(vector signed int, vector bool long);
int	vec_all_le(vector signed int, vector signed int);
int	vec_all_le(vector signed char, vector bool char);
int	vec_all_le(vector signed char, vector signed char);
int	vec_all_le(vector unsigned short, vector bool short);
int	vec_all_le(vector unsigned short, vector unsigned short);
int	vec_all_le(vector unsigned int, vector bool long);
int	vec_all_le(vector unsigned int, vector unsigned int);
int	vec_all_le(vector unsigned char, vector bool char);

int	vec_all_le(vector unsigned char, vector unsigned char);
int	vec_all_lt(vector bool short, vector signed short);
int	vec_all_lt(vector bool short, vector unsigned short);
int	vec_all_lt(vector bool long, vector signed int);
int	vec_all_lt(vector bool long, vector unsigned int);
int	vec_all_lt(vector bool char, vector signed char);
int	vec_all_lt(vector bool char, vector unsigned char);
int	vec_all_lt(vector float, vector float);
int	vec_all_lt(vector signed short, vector bool short);
int	vec_all_lt(vector signed short, vector signed short);
int	vec_all_lt(vector signed int, vector bool long);
int	vec_all_lt(vector signed int, vector signed int);
int	vec_all_lt(vector signed char, vector bool char);
int	vec_all_lt(vector signed char, vector signed char);
int	vec_all_lt(vector unsigned short, vector bool short);
int	vec_all_lt(vector unsigned short, vector unsigned short);
int	vec_all_lt(vector unsigned int, vector bool long);
int	vec_all_lt(vector unsigned int, vector unsigned int);
int	vec_all_lt(vector unsigned char, vector bool char);
int	vec_all_lt(vector unsigned char, vector unsigned char);
int	vec_all_nan(vector float);
int	vec_all_ne(vector bool short, vector bool short);
int	vec_all_ne(vector bool short, vector signed short);
int	vec_all_ne(vector bool short, vector unsigned short);
int	vec_all_ne(vector bool long, vector bool long);
int	vec_all_ne(vector bool long, vector signed int);
int	vec_all_ne(vector bool long, vector unsigned int);
int	vec_all_ne(vector bool char, vector bool char);
int	vec_all_ne(vector bool char, vector signed char);
int	vec_all_ne(vector bool char, vector unsigned char);
int	vec_all_ne(vector float, vector float);

int	vec_all_ne(vector pixel, vector pixel);
int	vec_all_ne(vector signed short, vector bool short);
int	vec_all_ne(vector signed short, vector signed short);
int	vec_all_ne(vector signed int, vector bool long);
int	vec_all_ne(vector signed int, vector signed int);
int	vec_all_ne(vector signed char, vector bool char);
int	vec_all_ne(vector signed char, vector signed char);
int	vec_all_ne(vector unsigned short, vector bool short);
int	vec_all_ne(vector unsigned short, vector unsigned short);
int	vec_all_ne(vector unsigned int, vector bool long);
int	vec_all_ne(vector unsigned int, vector unsigned int);
int	vec_all_ne(vector unsigned char, vector bool char);
int	vec_all_ne(vector unsigned char, vector unsigned char);
int	vec_all_nge(vector float, vector float);
int	vec_all_ngt(vector float, vector float);
int	vec_all_nle(vector float, vector float);
int	vec_all_nlt(vector float, vector float);
int	vec_all_numeric(vector float);
vector bool short	vec_and(vector bool short, vector bool short);
vector signed short	vec_and(vector bool short, vector signed short);
vector unsigned short	vec_and(vector bool short, vector unsigned short);
vector bool long	vec_and(vector bool long, vector bool long);
vector float	vec_and(vector bool long, vector float);
vector signed int	vec_and(vector bool long, vector signed int);
vector unsigned int	vec_and(vector bool long, vector unsigned int);
vector bool char	vec_and(vector bool char, vector bool char);
vector signed char	vec_and(vector bool char, vector signed char);
vector unsigned char	vec_and(vector bool char, vector unsigned char);
vector float	vec_and(vector float, vector bool long);
vector float	vec_and(vector float, vector float);
vector signed short	vec_and(vector signed short, vector bool short);

vector signed short	vec_and(vector signed short, vector signed short);
vector signed int	vec_and(vector signed int, vector bool long);
vector signed int	vec_and(vector signed int, vector signed int);
vector signed char	vec_and(vector signed char, vector bool char);
vector signed char	vec_and(vector signed char, vector signed char);
vector unsigned short	vec_and(vector unsigned short, vector bool short);
vector unsigned short	vec_and(vector unsigned short, vector unsigned short);
vector unsigned int	vec_and(vector unsigned int, vector bool long);
vector unsigned int	vec_and(vector unsigned int, vector unsigned int);
vector unsigned char	vec_and(vector unsigned char, vector bool char);
vector unsigned char	vec_and(vector unsigned char, vector unsigned char);
vector bool short	vec_andc(vector bool short, vector bool short);
vector signed short	vec_andc(vector bool short, vector signed short);
vector unsigned short	vec_andc(vector bool short, vector unsigned short);
vector bool long	vec_andc(vector bool long, vector bool long);
vector float	vec_andc(vector bool long, vector float);
vector signed int	vec_andc(vector bool long, vector signed int);
vector unsigned int	vec_andc(vector bool long, vector unsigned int);
vector bool char	vec_andc(vector bool char, vector bool char);
vector signed char	vec_andc(vector bool char, vector signed char);
vector unsigned char	vec_andc(vector bool char, vector unsigned char);
vector float	vec_andc(vector float, vector bool long);
vector float	vec_andc(vector float, vector float);
vector signed short	vec_andc(vector signed short, vector bool short);
vector signed short	vec_andc(vector signed short, vector signed short);
vector signed int	vec_andc(vector signed int, vector bool long);
vector signed int	vec_andc(vector signed int, vector signed int);
vector signed char	vec_andc(vector signed char, vector bool char);
vector signed char	vec_andc(vector signed char, vector signed char);
vector unsigned short	vec_andc(vector unsigned short, vector bool short);
vector unsigned short	vec_andc(vector unsigned short, vector unsigned short);

vector unsigned int	vec_andc(vector unsigned int, vector bool long);
vector unsigned int	vec_andc(vector unsigned int, vector unsigned int);
vector unsigned char	vec_andc(vector unsigned char, vector bool char);
vector unsigned char	vec_andc(vector unsigned char, vector unsigned char);
int	vec_any_eq(vector bool short, vector bool short);
int	vec_any_eq(vector bool short, vector signed short);
int	vec_any_eq(vector bool short, vector unsigned short);
int	vec_any_eq(vector bool long, vector bool long);
int	vec_any_eq(vector bool long, vector signed int);
int	vec_any_eq(vector bool long, vector unsigned int);
int	vec_any_eq(vector bool char, vector bool char);
int	vec_any_eq(vector bool char, vector signed char);
int	vec_any_eq(vector bool char, vector unsigned char);
int	vec_any_eq(vector float, vector float);
int	vec_any_eq(vector pixel, vector pixel);
int	vec_any_eq(vector signed short, vector bool short);
int	vec_any_eq(vector signed short, vector signed short);
int	vec_any_eq(vector signed int, vector bool long);
int	vec_any_eq(vector signed int, vector signed int);
int	vec_any_eq(vector signed char, vector bool char);
int	vec_any_eq(vector signed char, vector signed char);
int	vec_any_eq(vector unsigned short, vector bool short);
int	vec_any_eq(vector unsigned short, vector unsigned short);
int	vec_any_eq(vector unsigned int, vector bool long);
int	vec_any_eq(vector unsigned int, vector unsigned int);
int	vec_any_eq(vector unsigned char, vector bool char);
int	vec_any_eq(vector unsigned char, vector unsigned char);
int	vec_any_ge(vector bool short, vector signed short);
int	vec_any_ge(vector bool short, vector unsigned short);
int	vec_any_ge(vector bool long, vector signed int);
int	vec_any_ge(vector bool long, vector unsigned int);

int	vec_any_ge(vector bool char, vector signed char);
int	vec_any_ge(vector bool char, vector unsigned char);
int	vec_any_ge(vector float, vector float);
int	vec_any_ge(vector signed short, vector bool short);
int	vec_any_ge(vector signed short, vector signed short);
int	vec_any_ge(vector signed int, vector bool long);
int	vec_any_ge(vector signed int, vector signed int);
int	vec_any_ge(vector signed char, vector bool char);
int	vec_any_ge(vector signed char, vector signed char);
int	vec_any_ge(vector unsigned short, vector bool short);
int	vec_any_ge(vector unsigned short, vector unsigned short);
int	vec_any_ge(vector unsigned int, vector bool long);
int	vec_any_ge(vector unsigned int, vector unsigned int);
int	vec_any_ge(vector unsigned char, vector bool char);
int	vec_any_ge(vector unsigned char, vector unsigned char);
int	vec_any_gt(vector bool short, vector signed short);
int	vec_any_gt(vector bool short, vector unsigned short);
int	vec_any_gt(vector bool long, vector signed int);
int	vec_any_gt(vector bool long, vector unsigned int);
int	vec_any_gt(vector bool char, vector signed char);
int	vec_any_gt(vector bool char, vector unsigned char);
int	vec_any_gt(vector float, vector float);
int	vec_any_gt(vector signed short, vector bool short);
int	vec_any_gt(vector signed short, vector signed short);
int	vec_any_gt(vector signed int, vector bool long);
int	vec_any_gt(vector signed int, vector signed int);
int	vec_any_gt(vector signed char, vector bool char);
int	vec_any_gt(vector signed char, vector signed char);
int	vec_any_gt(vector unsigned short, vector bool short);
int	vec_any_gt(vector unsigned short, vector unsigned short);
int	vec_any_gt(vector unsigned int, vector bool long);

int	vec_any_gt(vector unsigned int, vector unsigned int);
int	vec_any_gt(vector unsigned char, vector bool char);
int	vec_any_gt(vector unsigned char, vector unsigned char);
int	vec_any_le(vector bool short, vector signed short);
int	vec_any_le(vector bool short, vector unsigned short);
int	vec_any_le(vector bool long, vector signed int);
int	vec_any_le(vector bool long, vector unsigned int);
int	vec_any_le(vector bool char, vector signed char);
int	vec_any_le(vector bool char, vector unsigned char);
int	vec_any_le(vector float, vector float);
int	vec_any_le(vector signed short, vector bool short);
int	vec_any_le(vector signed short, vector signed short);
int	vec_any_le(vector signed int, vector bool long);
int	vec_any_le(vector signed int, vector signed int);
int	vec_any_le(vector signed char, vector bool char);
int	vec_any_le(vector signed char, vector signed char);
int	vec_any_le(vector unsigned short, vector bool short);
int	vec_any_le(vector unsigned short, vector unsigned short);
int	vec_any_le(vector unsigned int, vector bool long);
int	vec_any_le(vector unsigned int, vector unsigned int);
int	vec_any_le(vector unsigned char, vector bool char);
int	vec_any_le(vector unsigned char, vector unsigned char);
int	vec_any_lt(vector bool short, vector signed short);
int	vec_any_lt(vector bool short, vector unsigned short);
int	vec_any_lt(vector bool long, vector signed int);
int	vec_any_lt(vector bool long, vector unsigned int);
int	vec_any_lt(vector bool char, vector signed char);
int	vec_any_lt(vector bool char, vector unsigned char);
int	vec_any_lt(vector float, vector float);
int	vec_any_lt(vector signed short, vector bool short);
int	vec_any_lt(vector signed short, vector signed short);

int	vec_any_lt(vector signed int, vector bool long);
int	vec_any_lt(vector signed int, vector signed int);
int	vec_any_lt(vector signed char, vector bool char);
int	vec_any_lt(vector signed char, vector signed char);
int	vec_any_lt(vector unsigned short, vector bool short);
int	vec_any_lt(vector unsigned short, vector unsigned short);
int	vec_any_lt(vector unsigned int, vector bool long);
int	vec_any_lt(vector unsigned int, vector unsigned int);
int	vec_any_lt(vector unsigned char, vector bool char);
int	vec_any_lt(vector unsigned char, vector unsigned char);
int	vec_any_nan(vector float);
int	vec_any_ne(vector bool short, vector bool short);
int	vec_any_ne(vector bool short, vector signed short);
int	vec_any_ne(vector bool short, vector unsigned short);
int	vec_any_ne(vector bool long, vector bool long);
int	vec_any_ne(vector bool long, vector signed int);
int	vec_any_ne(vector bool long, vector unsigned int);
int	vec_any_ne(vector bool char, vector bool char);
int	vec_any_ne(vector bool char, vector signed char);
int	vec_any_ne(vector bool char, vector unsigned char);
int	vec_any_ne(vector float, vector float);
int	vec_any_ne(vector pixel, vector pixel);
int	vec_any_ne(vector signed short, vector bool short);
int	vec_any_ne(vector signed short, vector signed short);
int	vec_any_ne(vector signed int, vector bool long);
int	vec_any_ne(vector signed int, vector signed int);
int	vec_any_ne(vector signed char, vector bool char);
int	vec_any_ne(vector signed char, vector signed char);
int	vec_any_ne(vector unsigned short, vector bool short);
int	vec_any_ne(vector unsigned short, vector unsigned short);
int	vec_any_ne(vector unsigned int, vector bool long);

int	vec_any_ne(vector unsigned int, vector unsigned int);
int	vec_any_ne(vector unsigned char, vector bool char);
int	vec_any_ne(vector unsigned char, vector unsigned char);
int	vec_any_nge(vector float, vector float);
int	vec_any_ngt(vector float, vector float);
int	vec_any_nle(vector float, vector float);
int	vec_any_nlt(vector float, vector float);
int	vec_any_numeric(vector float);
int	vec_any_out(vector float, vector float);
vector signed char	vec_avg(vector signed char, vector signed char);
vector signed short	vec_avg(vector signed short, vector signed short);
vector signed int	vec_avg(vector signed int, vector signed int);
vector unsigned char	vec_avg(vector unsigned char, vector unsigned char);
vector unsigned short	vec_avg(vector unsigned short, vector unsigned short);
vector unsigned int	vec_avg(vector unsigned int, vector unsigned int);
vector float	vec_ceil(vector float);
vector signed int	vec_cmpb(vector float, vector float);
vector bool long	vec_cmpeq(vector float, vector float);
vector bool char	vec_cmpeq(vector signed char, vector signed char);
vector bool char	vec_cmpeq(vector unsigned char, vector unsigned char);
vector bool short	vec_cmpeq(vector signed short, vector signed short);
vector bool short	vec_cmpeq(vector unsigned short, vector unsigned short);
vector bool long	vec_cmpeq(vector signed int, vector signed int);
vector bool long	vec_cmpeq(vector unsigned int, vector unsigned int);
vector bool long	vec_cmpge(vector float, vector float);
vector bool long	vec_cmpgt(vector float, vector float);
vector bool char	vec_cmpgt(vector signed char, vector signed char);
vector bool short	vec_cmpgt(vector signed short, vector signed short);
vector bool long	vec_cmpgt(vector signed int, vector signed int);
vector bool char	vec_cmpgt(vector unsigned char, vector unsigned char);
vector bool short	vec_cmpgt(vector unsigned short, vector unsigned short);

vector bool long	vec_cmpgt(vector unsigned int, vector unsigned int);
vector bool long	vec_cmple(vector float, vector float);
vector bool long	vec_cmplt(vector float, vector float);
vector bool short	vec_cmplt(vector signed short, vector signed short);
vector bool long	vec_cmplt(vector signed int, vector signed int);
vector bool char	vec_cmplt(vector signed char, vector signed char);
vector bool short	vec_cmplt(vector unsigned short, vector unsigned short);
vector bool long	vec_cmplt(vector unsigned int, vector unsigned int);
vector bool char	vec_cmplt(vector unsigned char, vector unsigned char);
vector float	vec_ctf(vector signed int, int);
vector float	vec_ctf(vector unsigned int, int);
vector signed int	vec_cts(vector float, int);
vector unsigned int	vec_ctu(vector float, int);
void	vec_dss(int);
void	vec_dssall( );
void	vec_dst(float *, int, int);
void	vec_dst(int *, int, int);
void	vec_dst(long *, int, int);
void	vec_dst(short *, int, int);
void	vec_dst(char *, int, int);
void	vec_dst(unsigned char *, int, int);
void	vec_dst(unsigned int *, int, int);
void	vec_dst(unsigned long *, int, int);
void	vec_dst(unsigned short *, int, int);
void	vec_dst(vector bool short *, int, int);
void	vec_dst(vector bool long *, int, int);
void	vec_dst(vector bool char *, int, int);
void	vec_dst(vector float *, int, int);
void	vec_dst(vector pixel *, int, int);
void	vec_dst(vector signed short *, int, int);
void	vec_dst(vector signed long *, int, int);

void	vec_dst(vector signed char *, int, int);
void	vec_dst(vector unsigned short *, int, int);
void	vec_dst(vector unsigned long *, int, int);
void	vec_dst(vector unsigned char *, int, int);
void	vec_dstst(float *, int, int);
void	vec_dstst(int *, int, int);
void	vec_dstst(long *, int, int);
void	vec_dstst(short *, int, int);
void	vec_dstst(char *, int, int);
void	vec_dstst(unsigned char *, int, int);
void	vec_dstst(unsigned int *, int, int);
void	vec_dstst(unsigned long *, int, int);
void	vec_dstst(unsigned short *, int, int);
void	vec_dstst(vector bool short *, int, int);
void	vec_dstst(vector bool long *, int, int);
void	vec_dstst(vector bool char *, int, int);
void	vec_dstst(vector float *, int, int);
void	vec_dstst(vector pixel *, int, int);
void	vec_dstst(vector signed short *, int, int);
void	vec_dstst(vector signed long *, int, int);
void	vec_dstst(vector signed char *, int, int);
void	vec_dstst(vector unsigned short *, int, int);
void	vec_dstst(vector unsigned long *, int, int);
void	vec_dstst(vector unsigned char *, int, int);
void	vec_dststt(float *, int, int);
void	vec_dststt(int *, int, int);
void	vec_dststt(long *, int, int);
void	vec_dststt(short *, int, int);
void	vec_dststt(char *, int, int);
void	vec_dststt(unsigned char *, int, int);
void	vec_dststt(unsigned int *, int, int);

void	vec_dststt(unsigned long *, int, int);
void	vec_dststt(unsigned short *, int, int);
void	vec_dststt(vector bool short *, int, int);
void	vec_dststt(vector bool long *, int, int);
void	vec_dststt(vector bool char *, int, int);
void	vec_dststt(vector float *, int, int);
void	vec_dststt(vector pixel *, int, int);
void	vec_dststt(vector signed short *, int, int);
void	vec_dststt(vector signed long *, int, int);
void	vec_dststt(vector signed char *, int, int);
void	vec_dststt(vector unsigned short *, int, int);
void	vec_dststt(vector unsigned long *, int, int);
void	vec_dststt(vector unsigned char *, int, int);
void	vec_dstt(float *, int, int);
void	vec_dstt(int *, int, int);
void	vec_dstt(long *, int, int);
void	vec_dstt(short *, int, int);
void	vec_dstt(char *, int, int);
void	vec_dstt(unsigned char *, int, int);
void	vec_dstt(unsigned int *, int, int);
void	vec_dstt(unsigned long *, int, int);
void	vec_dstt(unsigned short *, int, int);
void	vec_dstt(vector bool short *, int, int);
void	vec_dstt(vector bool long *, int, int);
void	vec_dstt(vector bool char *, int, int);
void	vec_dstt(vector float *, int, int);
void	vec_dstt(vector pixel *, int, int);
void	vec_dstt(vector signed short *, int, int);
void	vec_dstt(vector signed long *, int, int);
void	vec_dstt(vector signed char *, int, int);
void	vec_dstt(vector unsigned short *, int, int);

void	vec_dstt(vector unsigned long *, int, int);
void	vec_dstt(vector unsigned char *, int, int);
vector float	vec_expte(vector float);
vector float	vec_floor(vector float);
vector float	vec_ld(int, float *);
vector signed int	vec_ld(int, int *);
vector signed int	vec_ld(int, long *);
vector signed short	vec_ld(int, short *);
vector signed char	vec_ld(int, char *);
vector unsigned char	vec_ld(int, unsigned char *);
vector unsigned int	vec_ld(int, unsigned int *);
vector unsigned int	vec_ld(int, unsigned long *);
vector unsigned short	vec_ld(int, unsigned short *);
vector bool short	vec_ld(int, vector bool short *);
vector bool long	vec_ld(int, vector bool long *);
vector bool char	vec_ld(int, vector bool char *);
vector float	vec_ld(int, vector float *);
vector pixel	vec_ld(int, vector pixel *);
vector signed short	vec_ld(int, vector signed short *);
vector signed int	vec_ld(int, vector signed long *);
vector signed char	vec_ld(int, vector signed char *);
vector unsigned short	vec_ld(int, vector unsigned short *);
vector unsigned int	vec_ld(int, vector unsigned long *);
vector unsigned char	vec_ld(int, vector unsigned char *);
vector signed char	vec_lde(int, char *);
vector unsigned char	vec_lde(int, unsigned char *);
vector signed short	vec_lde(int, short *);
vector unsigned short	vec_lde(int, unsigned short *);
vector float	vec_lde(int, float *);
vector signed int	vec_lde(int, int *);
vector signed int	vec_lde(int, long *);

vector unsigned int	vec_lde(int, unsigned int *);
vector unsigned int	vec_lde(int, unsigned long *);
vector float	vec_ldl(int, float *);
vector signed int	vec_ldl(int, int *);
vector signed int	vec_ldl(int, long *);
vector signed short	vec_ldl(int, short *);
vector signed char	vec_ldl(int, char *);
vector unsigned char	vec_ldl(int, unsigned char *);
vector unsigned int	vec_ldl(int, unsigned int *);
vector unsigned int	vec_ldl(int, unsigned long *);
vector unsigned short	vec_ldl(int, unsigned short *);
vector bool short	vec_ldl(int, vector bool short *);
vector bool long	vec_ldl(int, vector bool long *);
vector bool char	vec_ldl(int, vector bool char *);
vector float	vec_ldl(int, vector float *);
vector pixel	vec_ldl(int, vector pixel *);
vector signed short	vec_ldl(int, vector signed short *);
vector signed int	vec_ldl(int, vector signed long *);
vector signed char	vec_ldl(int, vector signed char *);
vector unsigned short	vec_ldl(int, vector unsigned short *);
vector unsigned int	vec_ldl(int, vector unsigned long *);
vector unsigned char	vec_ldl(int, vector unsigned char *);
vector float	vec_loge(vector float);
vector signed char	vec_lvebx(int, char *);
vector unsigned char	vec_lvebx(int, unsigned char *);
vector signed short	vec_lvehx(int, short *);
vector unsigned short	vec_lvehx(int, unsigned short *);
vector float	vec_lvewx(int, float *);
vector signed int	vec_lvewx(int, int *);
vector signed int	vec_lvewx(int, long *);
vector unsigned int	vec_lvewx(int, unsigned int *);

vector unsigned int	vec_lvewx(int, unsigned long *);
vector float	vec_lvlx(int, float *);
vector signed int	vec_lvlx(int, int *);
vector signed int	vec_lvlx(int, long *);
vector signed short	vec_lvlx(int, short *);
vector signed char	vec_lvlx(int, char *);
vector unsigned char	vec_lvlx(int, unsigned char *);
vector unsigned int	vec_lvlx(int, unsigned int *);
vector unsigned int	vec_lvlx(int, unsigned long *);
vector unsigned short	vec_lvlx(int, unsigned short *);
vector bool short	vec_lvlx(int, vector bool short *);
vector bool long	vec_lvlx(int, vector bool long *);
vector bool char	vec_lvlx(int, vector bool char *);
vector float	vec_lvlx(int, vector float *);
vector pixel	vec_lvlx(int, vector pixel *);
vector signed short	vec_lvlx(int, vector signed short *);
vector signed int	vec_lvlx(int, vector signed long *);
vector signed char	vec_lvlx(int, vector signed char *);
vector unsigned short	vec_lvlx(int, vector unsigned short *);
vector unsigned int	vec_lvlx(int, vector unsigned long *);
vector unsigned char	vec_lvlx(int, vector unsigned char *);
vector float	vec_lvlxl(int, float *);
vector signed int	vec_lvlxl(int, int *);
vector signed int	vec_lvlxl(int, long *);
vector signed short	vec_lvlxl(int, short *);
vector signed char	vec_lvlxl(int, char *);
vector unsigned char	vec_lvlxl(int, unsigned char *);
vector unsigned int	vec_lvlxl(int, unsigned int *);
vector unsigned int	vec_lvlxl(int, unsigned long *);
vector unsigned short	vec_lvlxl(int, unsigned short *);
vector bool short	vec_lvlxl(int, vector bool short *);

vector bool long	vec_lvlxl(int, vector bool long *);
vector bool char	vec_lvlxl(int, vector bool char *);
vector float	vec_lvlxl(int, vector float *);
vector pixel	vec_lvlxl(int, vector pixel *);
vector signed short	vec_lvlxl(int, vector signed short *);
vector signed int	vec_lvlxl(int, vector signed long *);
vector signed char	vec_lvlxl(int, vector signed char *);
vector unsigned short	vec_lvlxl(int, vector unsigned short *);
vector unsigned int	vec_lvlxl(int, vector unsigned long *);
vector unsigned char	vec_lvlxl(int, vector unsigned char *);
vector float	vec_lvrx(int, float *);
vector signed int	vec_lvrx(int, int *);
vector signed int	vec_lvrx(int, long *);
vector signed short	vec_lvrx(int, short *);
vector signed char	vec_lvrx(int, char *);
vector unsigned char	vec_lvrx(int, unsigned char *);
vector unsigned int	vec_lvrx(int, unsigned int *);
vector unsigned int	vec_lvrx(int, unsigned long *);
vector unsigned short	vec_lvrx(int, unsigned short *);
vector bool short	vec_lvrx(int, vector bool short *);
vector bool long	vec_lvrx(int, vector bool long *);
vector bool char	vec_lvrx(int, vector bool char *);
vector float	vec_lvrx(int, vector float *);
vector pixel	vec_lvrx(int, vector pixel *);
vector signed short	vec_lvrx(int, vector signed short *);
vector signed int	vec_lvrx(int, vector signed long *);
vector signed char	vec_lvrx(int, vector signed char *);
vector unsigned short	vec_lvrx(int, vector unsigned short *);
vector unsigned int	vec_lvrx(int, vector unsigned long *);
vector unsigned char	vec_lvrx(int, vector unsigned char *);
vector float	vec_lvrxl(int, float *);

vector signed int	vec_lvrxl(int, int *);
vector signed int	vec_lvrxl(int, long *);
vector signed short	vec_lvrxl(int, short *);
vector signed char	vec_lvrxl(int, char *);
vector unsigned char	vec_lvrxl(int, unsigned char *);
vector unsigned int	vec_lvrxl(int, unsigned int *);
vector unsigned int	vec_lvrxl(int, unsigned long *);
vector unsigned short	vec_lvrxl(int, unsigned short *);
vector bool short	vec_lvrxl(int, vector bool short *);
vector bool long	vec_lvrxl(int, vector bool long *);
vector bool char	vec_lvrxl(int, vector bool char *);
vector float	vec_lvrxl(int, vector float *);
vector pixel	vec_lvrxl(int, vector pixel *);
vector signed short	vec_lvrxl(int, vector signed short *);
vector signed int	vec_lvrxl(int, vector signed long *);
vector signed char	vec_lvrxl(int, vector signed char *);
vector unsigned short	vec_lvrxl(int, vector unsigned short *);
vector unsigned int	vec_lvrxl(int, vector unsigned long *);
vector unsigned char	vec_lvrxl(int, vector unsigned char *);
vector unsigned char	vec_lvsl(int, float *);
vector unsigned char	vec_lvsl(int, int *);
vector unsigned char	vec_lvsl(int, long *);
vector unsigned char	vec_lvsl(int, short *);
vector unsigned char	vec_lvsl(int, char *);
vector unsigned char	vec_lvsl(int, unsigned char *);
vector unsigned char	vec_lvsl(int, unsigned int *);
vector unsigned char	vec_lvsl(int, unsigned long *);
vector unsigned char	vec_lvsl(int, unsigned short *);
vector unsigned char	vec_lvsr(int, float *);
vector unsigned char	vec_lvsr(int, int *);
vector unsigned char	vec_lvsr(int, long *);

vector unsigned char	vec_lvsr(int, short *);
vector unsigned char	vec_lvsr(int, char *);
vector unsigned char	vec_lvsr(int, unsigned char *);
vector unsigned char	vec_lvsr(int, unsigned int *);
vector unsigned char	vec_lvsr(int, unsigned long *);
vector unsigned char	vec_lvsr(int, unsigned short *);
vector float	vec_lvx(int, float *);
vector signed int	vec_lvx(int, int *);
vector signed int	vec_lvx(int, long *);
vector signed short	vec_lvx(int, short *);
vector signed char	vec_lvx(int, char *);
vector unsigned char	vec_lvx(int, unsigned char *);
vector unsigned int	vec_lvx(int, unsigned int *);
vector unsigned int	vec_lvx(int, unsigned long *);
vector unsigned short	vec_lvx(int, unsigned short *);
vector bool short	vec_lvx(int, vector bool short *);
vector bool long	vec_lvx(int, vector bool long *);
vector bool char	vec_lvx(int, vector bool char *);
vector float	vec_lvx(int, vector float *);
vector pixel	vec_lvx(int, vector pixel *);
vector signed short	vec_lvx(int, vector signed short *);
vector signed int	vec_lvx(int, vector signed long *);
vector signed char	vec_lvx(int, vector signed char *);
vector unsigned short	vec_lvx(int, vector unsigned short *);
vector unsigned int	vec_lvx(int, vector unsigned long *);
vector unsigned char	vec_lvx(int, vector unsigned char *);
vector float	vec_lvxl(int, float *);
vector signed int	vec_lvxl(int, int *);
vector signed int	vec_lvxl(int, long *);
vector signed short	vec_lvxl(int, short *);
vector signed char	vec_lvxl(int, char *);

vector unsigned char	vec_lvxl(int, unsigned char *);
vector unsigned int	vec_lvxl(int, unsigned int *);
vector unsigned int	vec_lvxl(int, unsigned long *);
vector unsigned short	vec_lvxl(int, unsigned short *);
vector bool short	vec_lvxl(int, vector bool short *);
vector bool long	vec_lvxl(int, vector bool long *);
vector bool char	vec_lvxl(int, vector bool char *);
vector float	vec_lvxl(int, vector float *);
vector pixel	vec_lvxl(int, vector pixel *);
vector signed short	vec_lvxl(int, vector signed short *);
vector signed int	vec_lvxl(int, vector signed long *);
vector signed char	vec_lvxl(int, vector signed char *);
vector unsigned short	vec_lvxl(int, vector unsigned short *);
vector unsigned int	vec_lvxl(int, vector unsigned long *);
vector unsigned char	vec_lvxl(int, vector unsigned char *);
vector float	vec_madd(vector float, vector float, vector float);
vector signed short	vec_madds(vector signed short, vector signed short, vector signed short);
vector float	vec_max(vector float, vector float);
vector signed char	vec_max(vector bool char, vector signed char);
vector signed char	vec_max(vector signed char, vector bool char);
vector signed char	vec_max(vector signed char, vector signed char);
vector signed short	vec_max(vector bool short, vector signed short);
vector signed short	vec_max(vector signed short, vector bool short);
vector signed short	vec_max(vector signed short, vector signed short);
vector signed int	vec_max(vector bool long, vector signed int);
vector signed int	vec_max(vector signed int, vector bool long);
vector signed int	vec_max(vector signed int, vector signed int);
vector unsigned char	vec_max(vector bool char, vector unsigned char);
vector unsigned char	vec_max(vector unsigned char, vector bool char);
vector unsigned char	vec_max(vector unsigned char, vector unsigned char);
vector unsigned short	vec_max(vector bool short, vector unsigned short);

vector unsigned short	vec_max(vector unsigned short, vector bool short);
vector unsigned short	vec_max(vector unsigned short, vector unsigned short);
vector unsigned int	vec_max(vector bool long, vector unsigned int);
vector unsigned int	vec_max(vector unsigned int, vector bool long);
vector unsigned int	vec_max(vector unsigned int, vector unsigned int);
vector bool char	vec_mergeh(vector bool char, vector bool char);
vector signed char	vec_mergeh(vector signed char, vector signed char);
vector unsigned char	vec_mergeh(vector unsigned char, vector unsigned char);
vector bool short	vec_mergeh(vector bool short, vector bool short);
vector pixel	vec_mergeh(vector pixel, vector pixel);
vector signed short	vec_mergeh(vector signed short, vector signed short);
vector unsigned short	vec_mergeh(vector unsigned short, vector unsigned short);
vector bool long	vec_mergeh(vector bool long, vector bool long);
vector float	vec_mergeh(vector float, vector float);
vector signed int	vec_mergeh(vector signed int, vector signed int);
vector unsigned int	vec_mergeh(vector unsigned int, vector unsigned int);
vector bool char	vec_mergel(vector bool char, vector bool char);
vector signed char	vec_mergel(vector signed char, vector signed char);
vector unsigned char	vec_mergel(vector unsigned char, vector unsigned char);
vector bool short	vec_mergel(vector bool short, vector bool short);
vector pixel	vec_mergel(vector pixel, vector pixel);
vector signed short	vec_mergel(vector signed short, vector signed short);
vector unsigned short	vec_mergel(vector unsigned short, vector unsigned short);
vector bool long	vec_mergel(vector bool long, vector bool long);
vector float	vec_mergel(vector float, vector float);
vector signed int	vec_mergel(vector signed int, vector signed int);
vector unsigned int	vec_mergel(vector unsigned int, vector unsigned int);
volatile vector unsigned short	vec_mfvscr( );
vector float	vec_min(vector float, vector float);
vector signed char	vec_min(vector bool char, vector signed char);
vector signed char	vec_min(vector signed char, vector bool char);

vector signed char	vec_min(vector signed char, vector signed char);
vector signed short	vec_min(vector bool short, vector signed short);
vector signed short	vec_min(vector signed short, vector bool short);
vector signed short	vec_min(vector signed short, vector signed short);
vector signed int	vec_min(vector bool long, vector signed int);
vector signed int	vec_min(vector signed int, vector bool long);
vector signed int	vec_min(vector signed int, vector signed int);
vector unsigned char	vec_min(vector bool char, vector unsigned char);
vector unsigned char	vec_min(vector unsigned char, vector bool char);
vector unsigned char	vec_min(vector unsigned char, vector unsigned char);
vector unsigned short	vec_min(vector bool short, vector unsigned short);
vector unsigned short	vec_min(vector unsigned short, vector bool short);
vector unsigned short	vec_min(vector unsigned short, vector unsigned short);
vector unsigned int	vec_min(vector bool long, vector unsigned int);
vector unsigned int	vec_min(vector unsigned int, vector bool long);
vector unsigned int	vec_min(vector unsigned int, vector unsigned int);
vector signed short	vec_mladd(vector signed short, vector signed short, vector signed short);
vector signed short	vec_mladd(vector signed short, vector unsigned short, vector unsigned short);
vector signed short	vec_mladd(vector unsigned short, vector signed short, vector signed short);
vector unsigned short	vec_mladd(vector unsigned short, vector unsigned short, vector unsigned short);
vector signed short	vec_mradds(vector signed short, vector signed short, vector signed short);
vector signed int	vec_msum(vector signed char, vector unsigned char, vector signed int);
vector signed int	vec_msum(vector signed short, vector signed short, vector signed int);
vector unsigned int	vec_msum(vector unsigned char, vector unsigned char, vector unsigned int);
vector unsigned int	vec_msum(vector unsigned short, vector unsigned short, vector unsigned int);
vector signed int	vec_msums(vector signed short, vector signed short, vector signed int);
vector unsigned int	vec_msums(vector unsigned short, vector unsigned short, vector unsigned int);

void	vec_mtvscr(vector bool short);
void	vec_mtvscr(vector bool long);
void	vec_mtvscr(vector bool char);
void	vec_mtvscr(vector pixel);
void	vec_mtvscr(vector signed short);
void	vec_mtvscr(vector signed int);
void	vec_mtvscr(vector signed char);
void	vec_mtvscr(vector unsigned short);
void	vec_mtvscr(vector unsigned int);
void	vec_mtvscr(vector unsigned char);
vector signed short	vec_mule(vector signed char, vector signed char);
vector signed int	vec_mule(vector signed short, vector signed short);
vector unsigned short	vec_mule(vector unsigned char, vector unsigned char);
vector unsigned int	vec_mule(vector unsigned short, vector unsigned short);
vector signed short	vec_mulo(vector signed char, vector signed char);
vector signed int	vec_mulo(vector signed short, vector signed short);
vector unsigned short	vec_mulo(vector unsigned char, vector unsigned char);
vector unsigned int	vec_mulo(vector unsigned short, vector unsigned short);
vector float	vec_nmsub(vector float, vector float, vector float);
vector bool short	vec_nor(vector bool short, vector bool short);
vector bool long	vec_nor(vector bool long, vector bool long);
vector bool char	vec_nor(vector bool char, vector bool char);
vector float	vec_nor(vector float, vector float);
vector signed short	vec_nor(vector signed short, vector signed short);
vector signed int	vec_nor(vector signed int, vector signed int);
vector signed char	vec_nor(vector signed char, vector signed char);
vector unsigned short	vec_nor(vector unsigned short, vector unsigned short);
vector unsigned int	vec_nor(vector unsigned int, vector unsigned int);
vector unsigned char	vec_nor(vector unsigned char, vector unsigned char);
vector bool short	vec_or(vector bool short, vector bool short);
vector signed short	vec_or(vector bool short, vector signed short);

vector unsigned short	vec_or(vector bool short, vector unsigned short);
vector bool long	vec_or(vector bool long, vector bool long);
vector float	vec_or(vector bool long, vector float);
vector signed int	vec_or(vector bool long, vector signed int);
vector unsigned int	vec_or(vector bool long, vector unsigned int);
vector bool char	vec_or(vector bool char, vector bool char);
vector signed char	vec_or(vector bool char, vector signed char);
vector unsigned char	vec_or(vector bool char, vector unsigned char);
vector float	vec_or(vector float, vector bool long);
vector float	vec_or(vector float, vector float);
vector signed short	vec_or(vector signed short, vector bool short);
vector signed short	vec_or(vector signed short, vector signed short);
vector signed int	vec_or(vector signed int, vector bool long);
vector signed int	vec_or(vector signed int, vector signed int);
vector signed char	vec_or(vector signed char, vector bool char);
vector signed char	vec_or(vector signed char, vector signed char);
vector unsigned short	vec_or(vector unsigned short, vector bool short);
vector unsigned short	vec_or(vector unsigned short, vector unsigned short);
vector unsigned int	vec_or(vector unsigned int, vector bool long);
vector unsigned int	vec_or(vector unsigned int, vector unsigned int);
vector unsigned char	vec_or(vector unsigned char, vector bool char);
vector unsigned char	vec_or(vector unsigned char, vector unsigned char);
vector bool char	vec_pack(vector bool short, vector bool short);
vector signed char	vec_pack(vector signed short, vector signed short);
vector unsigned char	vec_pack(vector unsigned short, vector unsigned short);
vector bool short	vec_pack(vector bool long, vector bool long);
vector signed short	vec_pack(vector signed int, vector signed int);
vector unsigned short	vec_pack(vector unsigned int, vector unsigned int);
vector pixel	vec_packpx(vector unsigned int, vector unsigned int);
vector signed char	vec_packs(vector signed short, vector signed short);
vector signed short	vec_packs(vector signed int, vector signed int);

vector unsigned char	vec_packs(vector unsigned short, vector unsigned short);
vector unsigned short	vec_packs(vector unsigned int, vector unsigned int);
vector unsigned char	vec_packsu(vector signed short, vector signed short);
vector unsigned short	vec_packsu(vector signed int, vector signed int);
vector unsigned char	vec_packsu(vector unsigned short, vector unsigned short);
vector unsigned short	vec_packsu(vector unsigned int, vector unsigned int);
vector bool short	vec_perm(vector bool short, vector bool short, vector unsigned char);
vector bool long	vec_perm(vector bool long, vector bool long, vector unsigned char);
vector bool char	vec_perm(vector bool char, vector bool char, vector unsigned char);
vector float	vec_perm(vector float, vector float, vector unsigned char);
vector pixel	vec_perm(vector pixel, vector pixel, vector unsigned char);
vector signed short	vec_perm(vector signed short, vector signed short, vector unsigned char);
vector signed int	vec_perm(vector signed int, vector signed int, vector unsigned char);
vector signed char	vec_perm(vector signed char, vector signed char, vector unsigned char);
vector unsigned short	vec_perm(vector unsigned short, vector unsigned short, vector unsigned char);
vector unsigned int	vec_perm(vector unsigned int, vector unsigned int, vector unsigned char);
vector unsigned char	vec_perm(vector unsigned char, vector unsigned char, vector unsigned char);
vector float	vec_re(vector float);
vector signed char	vec_rl(vector signed char, vector unsigned char);
vector unsigned char	vec_rl(vector unsigned char, vector unsigned char);
vector signed short	vec_rl(vector signed short, vector unsigned short);
vector unsigned short	vec_rl(vector unsigned short, vector unsigned short);
vector signed int	vec_rl(vector signed int, vector unsigned int);
vector unsigned int	vec_rl(vector unsigned int, vector unsigned int);
vector float	vec_round(vector float);
vector float	vec_rsqrte(vector float);
vector bool short	vec_sel(vector bool short, vector bool short, vector bool short);
vector bool short	vec_sel(vector bool short, vector bool short, vector unsigned short);

vector bool long	vec_sel(vector bool long, vector bool long, vector bool long);
vector bool long	vec_sel(vector bool long, vector bool long, vector unsigned int);
vector bool char	vec_sel(vector bool char, vector bool char, vector bool char);
vector bool char	vec_sel(vector bool char, vector bool char, vector unsigned char);
vector float	vec_sel(vector float, vector float, vector bool long);
vector float	vec_sel(vector float, vector float, vector unsigned int);
vector signed short	vec_sel(vector signed short, vector signed short, vector bool short);
vector signed short	vec_sel(vector signed short, vector signed short, vector unsigned short);
vector signed int	vec_sel(vector signed int, vector signed int, vector bool long);
vector signed int	vec_sel(vector signed int, vector signed int, vector unsigned int);
vector signed char	vec_sel(vector signed char, vector signed char, vector bool char);
vector signed char	vec_sel(vector signed char, vector signed char, vector unsigned char);
vector unsigned short	vec_sel(vector unsigned short, vector unsigned short, vector bool short);
vector unsigned short	vec_sel(vector unsigned short, vector unsigned short, vector unsigned short);
vector unsigned int	vec_sel(vector unsigned int, vector unsigned int, vector bool long);
vector unsigned int	vec_sel(vector unsigned int, vector unsigned int, vector unsigned int);
vector unsigned char	vec_sel(vector unsigned char, vector unsigned char, vector bool char);
vector unsigned char	vec_sel(vector unsigned char, vector unsigned char, vector unsigned char);
vector signed char	vec_sl(vector signed char, vector unsigned char);
vector unsigned char	vec_sl(vector unsigned char, vector unsigned char);
vector signed short	vec_sl(vector signed short, vector unsigned short);
vector unsigned short	vec_sl(vector unsigned short, vector unsigned short);
vector signed int	vec_sl(vector signed int, vector unsigned int);
vector unsigned int	vec_sl(vector unsigned int, vector unsigned int);
vector float	vec_sld(vector float, vector float, int);
vector pixel	vec_sld(vector pixel, vector pixel, int);
vector signed short	vec_sld(vector signed short, vector signed short, int);
vector signed int	vec_sld(vector signed int, vector signed int, int);
vector signed char	vec_sld(vector signed char, vector signed char, int);

vector unsigned short	vec_sld(vector unsigned short, vector unsigned short, int);
vector unsigned int	vec_sld(vector unsigned int, vector unsigned int, int);
vector unsigned char	vec_sld(vector unsigned char, vector unsigned char, int);
vector bool short	vec_sll(vector bool short, vector unsigned short);
vector bool short	vec_sll(vector bool short, vector unsigned int);
vector bool short	vec_sll(vector bool short, vector unsigned char);
vector bool long	vec_sll(vector bool long, vector unsigned short);
vector bool long	vec_sll(vector bool long, vector unsigned int);
vector bool long	vec_sll(vector bool long, vector unsigned char);
vector bool char	vec_sll(vector bool char, vector unsigned short);
vector bool char	vec_sll(vector bool char, vector unsigned int);
vector bool char	vec_sll(vector bool char, vector unsigned char);
vector pixel	vec_sll(vector pixel, vector unsigned short);
vector pixel	vec_sll(vector pixel, vector unsigned int);
vector pixel	vec_sll(vector pixel, vector unsigned char);
vector signed short	vec_sll(vector signed short, vector unsigned short);
vector signed short	vec_sll(vector signed short, vector unsigned int);
vector signed short	vec_sll(vector signed short, vector unsigned char);
vector signed int	vec_sll(vector signed int, vector unsigned short);
vector signed int	vec_sll(vector signed int, vector unsigned int);
vector signed int	vec_sll(vector signed int, vector unsigned char);
vector signed char	vec_sll(vector signed char, vector unsigned short);
vector signed char	vec_sll(vector signed char, vector unsigned int);
vector signed char	vec_sll(vector signed char, vector unsigned char);
vector unsigned short	vec_sll(vector unsigned short, vector unsigned short);
vector unsigned short	vec_sll(vector unsigned short, vector unsigned int);
vector unsigned short	vec_sll(vector unsigned short, vector unsigned char);
vector unsigned int	vec_sll(vector unsigned int, vector unsigned short);
vector unsigned int	vec_sll(vector unsigned int, vector unsigned int);
vector unsigned int	vec_sll(vector unsigned int, vector unsigned char);
vector unsigned char	vec_sll(vector unsigned char, vector unsigned short);

vector unsigned char	vec_sll(vector unsigned char, vector unsigned int);
vector unsigned char	vec_sll(vector unsigned char, vector unsigned char);
vector float	vec_slo(vector float, vector signed char);
vector float	vec_slo(vector float, vector unsigned char);
vector pixel	vec_slo(vector pixel, vector signed char);
vector pixel	vec_slo(vector pixel, vector unsigned char);
vector signed short	vec_slo(vector signed short, vector signed char);
vector signed short	vec_slo(vector signed short, vector unsigned char);
vector signed int	vec_slo(vector signed int, vector signed char);
vector signed int	vec_slo(vector signed int, vector unsigned char);
vector signed char	vec_slo(vector signed char, vector signed char);
vector signed char	vec_slo(vector signed char, vector unsigned char);
vector unsigned short	vec_slo(vector unsigned short, vector signed char);
vector unsigned short	vec_slo(vector unsigned short, vector unsigned char);
vector unsigned int	vec_slo(vector unsigned int, vector signed char);
vector unsigned int	vec_slo(vector unsigned int, vector unsigned char);
vector unsigned char	vec_slo(vector unsigned char, vector signed char);
vector unsigned char	vec_slo(vector unsigned char, vector unsigned char);
vector bool char	vec_splat(vector bool char, int);
vector signed char	vec_splat(vector signed char, int);
vector unsigned char	vec_splat(vector unsigned char, int);
vector bool short	vec_splat(vector bool short, int);
vector pixel	vec_splat(vector pixel, int);
vector signed short	vec_splat(vector signed short, int);
vector unsigned short	vec_splat(vector unsigned short, int);
vector bool long	vec_splat(vector bool long, int);
vector float	vec_splat(vector float, int);
vector signed int	vec_splat(vector signed int, int);
vector unsigned int	vec_splat(vector unsigned int, int);
vector signed short	vec_splat_s16(int);
vector signed int	vec_splat_s32(int);

vector signed char	vec_splat_s8(int);
vector unsigned short	vec_splat_u16(int);
vector unsigned int	vec_splat_u32(int);
vector unsigned char	vec_splat_u8(int);
vector signed char	vec_sr(vector signed char, vector unsigned char);
vector unsigned char	vec_sr(vector unsigned char, vector unsigned char);
vector signed short	vec_sr(vector signed short, vector unsigned short);
vector unsigned short	vec_sr(vector unsigned short, vector unsigned short);
vector signed int	vec_sr(vector signed int, vector unsigned int);
vector unsigned int	vec_sr(vector unsigned int, vector unsigned int);
vector signed char	vec_sra(vector signed char, vector unsigned char);
vector unsigned char	vec_sra(vector unsigned char, vector unsigned char);
vector signed short	vec_sra(vector signed short, vector unsigned short);
vector unsigned short	vec_sra(vector unsigned short, vector unsigned short);
vector signed int	vec_sra(vector signed int, vector unsigned int);
vector unsigned int	vec_sra(vector unsigned int, vector unsigned int);
vector bool short	vec_srl(vector bool short, vector unsigned short);
vector bool short	vec_srl(vector bool short, vector unsigned int);
vector bool short	vec_srl(vector bool short, vector unsigned char);
vector bool long	vec_srl(vector bool long, vector unsigned short);
vector bool long	vec_srl(vector bool long, vector unsigned int);
vector bool long	vec_srl(vector bool long, vector unsigned char);
vector bool char	vec_srl(vector bool char, vector unsigned short);
vector bool char	vec_srl(vector bool char, vector unsigned int);
vector bool char	vec_srl(vector bool char, vector unsigned char);
vector pixel	vec_srl(vector pixel, vector unsigned short);
vector pixel	vec_srl(vector pixel, vector unsigned int);
vector pixel	vec_srl(vector pixel, vector unsigned char);
vector signed short	vec_srl(vector signed short, vector unsigned short);
vector signed short	vec_srl(vector signed short, vector unsigned int);
vector signed short	vec_srl(vector signed short, vector unsigned char);

vector signed int	vec_srl(vector signed int, vector unsigned short);
vector signed int	vec_srl(vector signed int, vector unsigned int);
vector signed int	vec_srl(vector signed int, vector unsigned char);
vector signed char	vec_srl(vector signed char, vector unsigned short);
vector signed char	vec_srl(vector signed char, vector unsigned int);
vector signed char	vec_srl(vector signed char, vector unsigned char);
vector unsigned short	vec_srl(vector unsigned short, vector unsigned short);
vector unsigned short	vec_srl(vector unsigned short, vector unsigned int);
vector unsigned short	vec_srl(vector unsigned short, vector unsigned char);
vector unsigned int	vec_srl(vector unsigned int, vector unsigned short);
vector unsigned int	vec_srl(vector unsigned int, vector unsigned int);
vector unsigned int	vec_srl(vector unsigned int, vector unsigned char);
vector unsigned char	vec_srl(vector unsigned char, vector unsigned short);
vector unsigned char	vec_srl(vector unsigned char, vector unsigned int);
vector unsigned char	vec_srl(vector unsigned char, vector unsigned char);
vector float	vec_sro(vector float, vector signed char);
vector float	vec_sro(vector float, vector unsigned char);
vector pixel	vec_sro(vector pixel, vector signed char);
vector pixel	vec_sro(vector pixel, vector unsigned char);
vector signed short	vec_sro(vector signed short, vector signed char);
vector signed short	vec_sro(vector signed short, vector unsigned char);
vector signed int	vec_sro(vector signed int, vector signed char);
vector signed int	vec_sro(vector signed int, vector unsigned char);
vector signed char	vec_sro(vector signed char, vector signed char);
vector signed char	vec_sro(vector signed char, vector unsigned char);
vector unsigned short	vec_sro(vector unsigned short, vector signed char);
vector unsigned short	vec_sro(vector unsigned short, vector unsigned char);
vector unsigned int	vec_sro(vector unsigned int, vector signed char);
vector unsigned int	vec_sro(vector unsigned int, vector unsigned char);
vector unsigned char	vec_sro(vector unsigned char, vector signed char);
vector unsigned char	vec_sro(vector unsigned char, vector unsigned char);

void	vec_st(vector bool short, int, vector bool short *);
void	vec_st(vector bool long, int, vector bool long *);
void	vec_st(vector bool char, int, vector bool char *);
void	vec_st(vector float, int, float *);
void	vec_st(vector float, int, vector float *);
void	vec_st(vector pixel, int, vector pixel *);
void	vec_st(vector signed short, int, short *);
void	vec_st(vector signed short, int, vector signed short *);
void	vec_st(vector signed int, int, int *);
void	vec_st(vector signed int, int, long *);
void	vec_st(vector signed int, int, vector signed long *);
void	vec_st(vector signed char, int, char *);
void	vec_st(vector signed char, int, vector signed char *);
void	vec_st(vector unsigned short, int, unsigned short *);
void	vec_st(vector unsigned short, int, vector unsigned short *);
void	vec_st(vector unsigned int, int, unsigned int *);
void	vec_st(vector unsigned int, int, unsigned long *);
void	vec_st(vector unsigned int, int, vector unsigned long *);
void	vec_st(vector unsigned char, int, unsigned char *);
void	vec_st(vector unsigned char, int, vector unsigned char *);
void	vec_ste(vector signed char, int, char *);
void	vec_ste(vector unsigned char, int, unsigned char *);
void	vec_ste(vector signed short, int, short *);
void	vec_ste(vector unsigned short, int, unsigned short *);
void	vec_ste(vector float, int, float *);
void	vec_ste(vector signed int, int, int *);
void	vec_ste(vector signed int, int, long *);
void	vec_ste(vector unsigned int, int, unsigned int *);
void	vec_ste(vector unsigned int, int, unsigned long *);
void	vec_stl(vector bool short, int, vector bool short *);
void	vec_stl(vector bool long, int, vector bool long *);

void	vec_stl(vector bool char, int, vector bool char *);
void	vec_stl(vector float, int, float *);
void	vec_stl(vector float, int, vector float *);
void	vec_stl(vector pixel, int, vector pixel *);
void	vec_stl(vector signed short, int, short *);
void	vec_stl(vector signed short, int, vector signed short *);
void	vec_stl(vector signed int, int, int *);
void	vec_stl(vector signed int, int, long *);
void	vec_stl(vector signed int, int, vector signed long *);
void	vec_stl(vector signed char, int, char *);
void	vec_stl(vector signed char, int, vector signed char *);
void	vec_stl(vector unsigned short, int, unsigned short *);
void	vec_stl(vector unsigned short, int, vector unsigned short *);
void	vec_stl(vector unsigned int, int, unsigned int *);
void	vec_stl(vector unsigned int, int, unsigned long *);
void	vec_stl(vector unsigned int, int, vector unsigned long *);
void	vec_stl(vector unsigned char, int, unsigned char *);
void	vec_stl(vector unsigned char, int, vector unsigned char *);
void	vec_stvebx(vector signed char, int, char *);
void	vec_stvebx(vector unsigned char, int, unsigned char *);
void	vec_stvehx(vector signed short, int, short *);
void	vec_stvehx(vector unsigned short, int, unsigned short *);
void	vec_stvewx(vector float, int, float *);
void	vec_stvewx(vector signed int, int, int *);
void	vec_stvewx(vector signed int, int, long *);
void	vec_stvewx(vector unsigned int, int, unsigned int *);
void	vec_stvewx(vector unsigned int, int, unsigned long *);
void	vec_stvlx(vector bool short, int, vector bool short *);
void	vec_stvlx(vector bool long, int, vector bool long *);
void	vec_stvlx(vector bool char, int, vector bool char *);
void	vec_stvlx(vector float, int, float *);

void	vec_stvlx(vector float, int, vector float *);
void	vec_stvlx(vector pixel, int, vector pixel *);
void	vec_stvlx(vector signed short, int, short *);
void	vec_stvlx(vector signed short, int, vector signed short *);
void	vec_stvlx(vector signed int, int, int *);
void	vec_stvlx(vector signed int, int, long *);
void	vec_stvlx(vector signed int, int, vector signed long *);
void	vec_stvlx(vector signed char, int, char *);
void	vec_stvlx(vector signed char, int, vector signed char *);
void	vec_stvlx(vector unsigned short, int, unsigned short *);
void	vec_stvlx(vector unsigned short, int, vector unsigned short *);
void	vec_stvlx(vector unsigned int, int, unsigned int *);
void	vec_stvlx(vector unsigned int, int, unsigned long *);
void	vec_stvlx(vector unsigned int, int, vector unsigned long *);
void	vec_stvlx(vector unsigned char, int, unsigned char *);
void	vec_stvlx(vector unsigned char, int, vector unsigned char *);
void	vec_stvlxl(vector bool short, int, vector bool short *);
void	vec_stvlxl(vector bool long, int, vector bool long *);
void	vec_stvlxl(vector bool char, int, vector bool char *);
void	vec_stvlxl(vector float, int, float *);
void	vec_stvlxl(vector float, int, vector float *);
void	vec_stvlxl(vector pixel, int, vector pixel *);
void	vec_stvlxl(vector signed short, int, short *);
void	vec_stvlxl(vector signed short, int, vector signed short *);
void	vec_stvlxl(vector signed int, int, int *);
void	vec_stvlxl(vector signed int, int, long *);
void	vec_stvlxl(vector signed int, int, vector signed long *);
void	vec_stvlxl(vector signed char, int, char *);
void	vec_stvlxl(vector signed char, int, vector signed char *);
void	vec_stvlxl(vector unsigned short, int, unsigned short *);
void	vec_stvlxl(vector unsigned short, int, vector unsigned short *);

void	vec_stvlxl(vector unsigned int, int, unsigned int *);
void	vec_stvlxl(vector unsigned int, int, unsigned long *);
void	vec_stvlxl(vector unsigned int, int, vector unsigned long *);
void	vec_stvlxl(vector unsigned char, int, unsigned char *);
void	vec_stvlxl(vector unsigned char, int, vector unsigned char *);
void	vec_stvrx(vector bool short, int, vector bool short *);
void	vec_stvrx(vector bool long, int, vector bool long *);
void	vec_stvrx(vector bool char, int, vector bool char *);
void	vec_stvrx(vector float, int, float *);
void	vec_stvrx(vector float, int, vector float *);
void	vec_stvrx(vector pixel, int, vector pixel *);
void	vec_stvrx(vector signed short, int, short *);
void	vec_stvrx(vector signed short, int, vector signed short *);
void	vec_stvrx(vector signed int, int, int *);
void	vec_stvrx(vector signed int, int, long *);
void	vec_stvrx(vector signed int, int, vector signed long *);
void	vec_stvrx(vector signed char, int, char *);
void	vec_stvrx(vector signed char, int, vector signed char *);
void	vec_stvrx(vector unsigned short, int, unsigned short *);
void	vec_stvrx(vector unsigned short, int, vector unsigned short *);
void	vec_stvrx(vector unsigned int, int, unsigned int *);
void	vec_stvrx(vector unsigned int, int, unsigned long *);
void	vec_stvrx(vector unsigned int, int, vector unsigned long *);
void	vec_stvrx(vector unsigned char, int, unsigned char *);
void	vec_stvrx(vector unsigned char, int, vector unsigned char *);
void	vec_stvrxl(vector bool short, int, vector bool short *);
void	vec_stvrxl(vector bool long, int, vector bool long *);
void	vec_stvrxl(vector bool char, int, vector bool char *);
void	vec_stvrxl(vector float, int, float *);
void	vec_stvrxl(vector float, int, vector float *);
void	vec_stvrxl(vector pixel, int, vector pixel *);

void	vec_stvrxl(vector signed short, int, short *);
void	vec_stvrxl(vector signed short, int, vector signed short *);
void	vec_stvrxl(vector signed int, int, int *);
void	vec_stvrxl(vector signed int, int, long *);
void	vec_stvrxl(vector signed int, int, vector signed long *);
void	vec_stvrxl(vector signed char, int, char *);
void	vec_stvrxl(vector signed char, int, vector signed char *);
void	vec_stvrxl(vector unsigned short, int, unsigned short *);
void	vec_stvrxl(vector unsigned short, int, vector unsigned short *);
void	vec_stvrxl(vector unsigned int, int, unsigned int *);
void	vec_stvrxl(vector unsigned int, int, unsigned long *);
void	vec_stvrxl(vector unsigned int, int, vector unsigned long *);
void	vec_stvrxl(vector unsigned char, int, unsigned char *);
void	vec_stvrxl(vector unsigned char, int, vector unsigned char *);
void	vec_stvx(vector bool short, int, vector bool short *);
void	vec_stvx(vector bool long, int, vector bool long *);
void	vec_stvx(vector bool char, int, vector bool char *);
void	vec_stvx(vector float, int, float *);
void	vec_stvx(vector float, int, vector float *);
void	vec_stvx(vector pixel, int, vector pixel *);
void	vec_stvx(vector signed short, int, short *);
void	vec_stvx(vector signed short, int, vector signed short *);
void	vec_stvx(vector signed int, int, int *);
void	vec_stvx(vector signed int, int, long *);
void	vec_stvx(vector signed int, int, vector signed long *);
void	vec_stvx(vector signed char, int, char *);
void	vec_stvx(vector signed char, int, vector signed char *);
void	vec_stvx(vector unsigned short, int, unsigned short *);
void	vec_stvx(vector unsigned short, int, vector unsigned short *);
void	vec_stvx(vector unsigned int, int, unsigned int *);
void	vec_stvx(vector unsigned int, int, unsigned long *);

void	vec_stvx(vector unsigned int, int, vector unsigned long *);
void	vec_stvx(vector unsigned char, int, unsigned char *);
void	vec_stvx(vector unsigned char, int, vector unsigned char *);
void	vec_stvxl(vector bool short, int, vector bool short *);
void	vec_stvxl(vector bool long, int, vector bool long *);
void	vec_stvxl(vector bool char, int, vector bool char *);
void	vec_stvxl(vector float, int, float *);
void	vec_stvxl(vector float, int, vector float *);
void	vec_stvxl(vector pixel, int, vector pixel *);
void	vec_stvxl(vector signed short, int, short *);
void	vec_stvxl(vector signed short, int, vector signed short *);
void	vec_stvxl(vector signed int, int, int *);
void	vec_stvxl(vector signed int, int, long *);
void	vec_stvxl(vector signed int, int, vector signed long *);
void	vec_stvxl(vector signed char, int, char *);
void	vec_stvxl(vector signed char, int, vector signed char *);
void	vec_stvxl(vector unsigned short, int, unsigned short *);
void	vec_stvxl(vector unsigned short, int, vector unsigned short *);
void	vec_stvxl(vector unsigned int, int, unsigned int *);
void	vec_stvxl(vector unsigned int, int, unsigned long *);
void	vec_stvxl(vector unsigned int, int, vector unsigned long *);
void	vec_stvxl(vector unsigned char, int, unsigned char *);
void	vec_stvxl(vector unsigned char, int, vector unsigned char *);
vector float	vec_sub(vector float, vector float);
vector signed char	vec_sub(vector bool char, vector signed char);
vector unsigned char	vec_sub(vector bool char, vector unsigned char);
vector signed char	vec_sub(vector signed char, vector bool char);
vector signed char	vec_sub(vector signed char, vector signed char);
vector unsigned char	vec_sub(vector unsigned char, vector bool char);
vector unsigned char	vec_sub(vector unsigned char, vector unsigned char);
vector signed short	vec_sub(vector bool short, vector signed short);

vector unsigned short	vec_sub(vector bool short, vector unsigned short);
vector signed short	vec_sub(vector signed short, vector bool short);
vector signed short	vec_sub(vector signed short, vector signed short);
vector unsigned short	vec_sub(vector unsigned short, vector bool short);
vector unsigned short	vec_sub(vector unsigned short, vector unsigned short);
vector signed int	vec_sub(vector bool long, vector signed int);
vector unsigned int	vec_sub(vector bool long, vector unsigned int);
vector signed int	vec_sub(vector signed int, vector bool long);
vector signed int	vec_sub(vector signed int, vector signed int);
vector unsigned int	vec_sub(vector unsigned int, vector bool long);
vector unsigned int	vec_sub(vector unsigned int, vector unsigned int);
vector unsigned int	vec_subc(vector unsigned int, vector unsigned int);
vector signed char	vec_subs(vector bool char, vector signed char);
vector signed char	vec_subs(vector signed char, vector bool char);
vector signed char	vec_subs(vector signed char, vector signed char);
vector signed short	vec_subs(vector bool short, vector signed short);
vector signed short	vec_subs(vector signed short, vector bool short);
vector signed short	vec_subs(vector signed short, vector signed short);
vector signed int	vec_subs(vector bool long, vector signed int);
vector signed int	vec_subs(vector signed int, vector bool long);
vector signed int	vec_subs(vector signed int, vector signed int);
vector unsigned char	vec_subs(vector bool char, vector unsigned char);
vector unsigned char	vec_subs(vector unsigned char, vector bool char);
vector unsigned char	vec_subs(vector unsigned char, vector unsigned char);
vector unsigned short	vec_subs(vector bool short, vector unsigned short);
vector unsigned short	vec_subs(vector unsigned short, vector bool short);
vector unsigned short	vec_subs(vector unsigned short, vector unsigned short);
vector unsigned int	vec_subs(vector bool long, vector unsigned int);
vector unsigned int	vec_subs(vector unsigned int, vector bool long);
vector unsigned int	vec_subs(vector unsigned int, vector unsigned int);
vector signed int	vec_sum2s(vector signed int, vector signed int);

vector signed int	vec_sum4s(vector signed char, vector signed int);
vector signed int	vec_sum4s(vector signed short, vector signed int);
vector unsigned int	vec_sum4s(vector unsigned char, vector unsigned int);
vector signed int	vec_sums(vector signed int, vector signed int);
vector float	vec_trunc(vector float);
vector signed int	vec_unpack2sh(vector unsigned short, vector unsigned short);
vector signed short	vec_unpack2sh(vector unsigned char, vector unsigned char);
vector signed int	vec_unpack2sl(vector unsigned short, vector unsigned short);
vector signed short	vec_unpack2sl(vector unsigned char, vector unsigned char);
vector unsigned int	vec_unpack2uh(vector unsigned short, vector unsigned short);
vector unsigned short	vec_unpack2uh(vector unsigned char, vector unsigned char);
vector unsigned int	vec_unpack2ul(vector unsigned short, vector unsigned short);
vector unsigned short	vec_unpack2ul(vector unsigned char, vector unsigned char);
vector unsigned int	vec_unpackh(vector pixel);
vector bool short	vec_unpackh(vector bool char);
vector signed short	vec_unpackh(vector signed char);
vector bool long	vec_unpackh(vector bool short);
vector signed int	vec_unpackh(vector signed short);
vector unsigned int	vec_unpackl(vector pixel);
vector bool short	vec_unpackl(vector bool char);
vector signed short	vec_unpackl(vector signed char);
vector bool long	vec_unpackl(vector bool short);
vector signed int	vec_unpackl(vector signed short);
vector unsigned int	vec_vaddcuw(vector unsigned int, vector unsigned int);
vector float	vec_vaddfp(vector float, vector float);
vector signed char	vec_vaddsbs(vector bool char, vector signed char);
vector signed char	vec_vaddsbs(vector signed char, vector bool char);
vector signed char	vec_vaddsbs(vector signed char, vector signed char);
vector signed short	vec_vaddshs(vector bool short, vector signed short);
vector signed short	vec_vaddshs(vector signed short, vector bool short);
vector signed short	vec_vaddshs(vector signed short, vector signed short);

vector signed int	vec_vaddsws(vector bool long, vector signed int);
vector signed int	vec_vaddsws(vector signed int, vector bool long);
vector signed int	vec_vaddsws(vector signed int, vector signed int);
vector signed char	vec_vaddubm(vector bool char, vector signed char);
vector unsigned char	vec_vaddubm(vector bool char, vector unsigned char);
vector signed char	vec_vaddubm(vector signed char, vector bool char);
vector signed char	vec_vaddubm(vector signed char, vector signed char);
vector unsigned char	vec_vaddubm(vector unsigned char, vector bool char);
vector unsigned char	vec_vaddubm(vector unsigned char, vector unsigned char);
vector unsigned char	vec_vaddubs(vector bool char, vector unsigned char);
vector unsigned char	vec_vaddubs(vector unsigned char, vector bool char);
vector unsigned char	vec_vaddubs(vector unsigned char, vector unsigned char);
vector signed short	vec_vadduhm(vector bool short, vector signed short);
vector unsigned short	vec_vadduhm(vector bool short, vector unsigned short);
vector signed short	vec_vadduhm(vector signed short, vector bool short);
vector signed short	vec_vadduhm(vector signed short, vector signed short);
vector unsigned short	vec_vadduhm(vector unsigned short, vector bool short);
vector unsigned short	vec_vadduhm(vector unsigned short, vector unsigned short);
vector unsigned short	vec_vadduhs(vector bool short, vector unsigned short);
vector unsigned short	vec_vadduhs(vector unsigned short, vector bool short);
vector unsigned short	vec_vadduhs(vector unsigned short, vector unsigned short);
vector signed int	vec_vadduwm(vector bool long, vector signed int);
vector unsigned int	vec_vadduwm(vector bool long, vector unsigned int);
vector signed int	vec_vadduwm(vector signed int, vector bool long);
vector signed int	vec_vadduwm(vector signed int, vector signed int);
vector unsigned int	vec_vadduwm(vector unsigned int, vector bool long);
vector unsigned int	vec_vadduwm(vector unsigned int, vector unsigned int);
vector unsigned int	vec_vadduws(vector bool long, vector unsigned int);
vector unsigned int	vec_vadduws(vector unsigned int, vector bool long);
vector unsigned int	vec_vadduws(vector unsigned int, vector unsigned int);
vector bool short	vec_vand(vector bool short, vector bool short);

vector signed short	vec_vand(vector bool short, vector signed short);
vector unsigned short	vec_vand(vector bool short, vector unsigned short);
vector bool long	vec_vand(vector bool long, vector bool long);
vector float	vec_vand(vector bool long, vector float);
vector signed int	vec_vand(vector bool long, vector signed int);
vector unsigned int	vec_vand(vector bool long, vector unsigned int);
vector bool char	vec_vand(vector bool char, vector bool char);
vector signed char	vec_vand(vector bool char, vector signed char);
vector unsigned char	vec_vand(vector bool char, vector unsigned char);
vector float	vec_vand(vector float, vector bool long);
vector float	vec_vand(vector float, vector float);
vector signed short	vec_vand(vector signed short, vector bool short);
vector signed short	vec_vand(vector signed short, vector signed short);
vector signed int	vec_vand(vector signed int, vector bool long);
vector signed int	vec_vand(vector signed int, vector signed int);
vector signed char	vec_vand(vector signed char, vector bool char);
vector signed char	vec_vand(vector signed char, vector signed char);
vector unsigned short	vec_vand(vector unsigned short, vector bool short);
vector unsigned short	vec_vand(vector unsigned short, vector unsigned short);
vector unsigned int	vec_vand(vector unsigned int, vector bool long);
vector unsigned int	vec_vand(vector unsigned int, vector unsigned int);
vector unsigned char	vec_vand(vector unsigned char, vector bool char);
vector unsigned char	vec_vand(vector unsigned char, vector unsigned char);
vector bool short	vec_vandc(vector bool short, vector bool short);
vector signed short	vec_vandc(vector bool short, vector signed short);
vector unsigned short	vec_vandc(vector bool short, vector unsigned short);
vector bool long	vec_vandc(vector bool long, vector bool long);
vector float	vec_vandc(vector bool long, vector float);
vector signed int	vec_vandc(vector bool long, vector signed int);
vector unsigned int	vec_vandc(vector bool long, vector unsigned int);
vector bool char	vec_vandc(vector bool char, vector bool char);

vector signed char	vec_vandc(vector bool char, vector signed char);
vector unsigned char	vec_vandc(vector bool char, vector unsigned char);
vector float	vec_vandc(vector float, vector bool long);
vector float	vec_vandc(vector float, vector float);
vector signed short	vec_vandc(vector signed short, vector bool short);
vector signed short	vec_vandc(vector signed short, vector signed short);
vector signed int	vec_vandc(vector signed int, vector bool long);
vector signed int	vec_vandc(vector signed int, vector signed int);
vector signed char	vec_vandc(vector signed char, vector bool char);
vector signed char	vec_vandc(vector signed char, vector signed char);
vector unsigned short	vec_vandc(vector unsigned short, vector bool short);
vector unsigned short	vec_vandc(vector unsigned short, vector unsigned short);
vector unsigned int	vec_vandc(vector unsigned int, vector bool long);
vector unsigned int	vec_vandc(vector unsigned int, vector unsigned int);
vector unsigned char	vec_vandc(vector unsigned char, vector bool char);
vector unsigned char	vec_vandc(vector unsigned char, vector unsigned char);
vector signed char	vec_vavgsb(vector signed char, vector signed char);
vector signed short	vec_vavgsh(vector signed short, vector signed short);
vector signed int	vec_vavgsw(vector signed int, vector signed int);
vector unsigned char	vec_vavgub(vector unsigned char, vector unsigned char);
vector unsigned short	vec_vavguh(vector unsigned short, vector unsigned short);
vector unsigned int	vec_vavguw(vector unsigned int, vector unsigned int);
vector float	vec_vcfsx(vector signed int, int);
vector float	vec_vcfux(vector unsigned int, int);
vector signed int	vec_vcmpbfp(vector float, vector float);
vector bool long	vec_vcmpeqfp(vector float, vector float);
vector bool char	vec_vcmpequb(vector signed char, vector signed char);
vector bool char	vec_vcmpequb(vector unsigned char, vector unsigned char);
vector bool short	vec_vcmpequh(vector signed short, vector signed short);
vector bool short	vec_vcmpequh(vector unsigned short, vector unsigned short);
vector bool long	vec_vcmpequw(vector signed int, vector signed int);

vector bool long	vec_vcmpequw(vector unsigned int, vector unsigned int);
vector bool long	vec_vcmpgefp(vector float, vector float);
vector bool long	vec_vcmpgtfp(vector float, vector float);
vector bool char	vec_vcmpgtsb(vector signed char, vector signed char);
vector bool short	vec_vcmpgtsh(vector signed short, vector signed short);
vector bool long	vec_vcmpgtsw(vector signed int, vector signed int);
vector bool char	vec_vcmpgtub(vector unsigned char, vector unsigned char);
vector bool short	vec_vcmpgtuh(vector unsigned short, vector unsigned short);
vector bool long	vec_vcmpgtuw(vector unsigned int, vector unsigned int);
vector signed int	vec_vctsxs(vector float, int);
vector unsigned int	vec_vctuxs(vector float, int);
vector float	vec_vexptefp(vector float);
vector float	vec_vlogefp(vector float);
vector float	vec_vmaddfp(vector float, vector float, vector float);
vector float	vec_vmaxfp(vector float, vector float);
vector signed char	vec_vmaxsb(vector bool char, vector signed char);
vector signed char	vec_vmaxsb(vector signed char, vector bool char);
vector signed char	vec_vmaxsb(vector signed char, vector signed char);
vector signed short	vec_vmaxsh(vector bool short, vector signed short);
vector signed short	vec_vmaxsh(vector signed short, vector bool short);
vector signed short	vec_vmaxsh(vector signed short, vector signed short);
vector signed int	vec_vmaxsw(vector bool long, vector signed int);
vector signed int	vec_vmaxsw(vector signed int, vector bool long);
vector signed int	vec_vmaxsw(vector signed int, vector signed int);
vector unsigned char	vec_vmaxub(vector bool char, vector unsigned char);
vector unsigned char	vec_vmaxub(vector unsigned char, vector bool char);
vector unsigned char	vec_vmaxub(vector unsigned char, vector unsigned char);
vector unsigned short	vec_vmaxuh(vector bool short, vector unsigned short);
vector unsigned short	vec_vmaxuh(vector unsigned short, vector bool short);
vector unsigned short	vec_vmaxuh(vector unsigned short, vector unsigned short);
vector unsigned int	vec_vmaxuw(vector bool long, vector unsigned int);

vector unsigned int	vec_vmaxuw(vector unsigned int, vector bool long);
vector unsigned int	vec_vmaxuw(vector unsigned int, vector unsigned int);
vector signed short	vec_vmhaddshs(vector signed short, vector signed short, vector signed short);
vector signed short	vec_vmhraddshs(vector signed short, vector signed short, vector signed short);
vector float	vec_vminfp(vector float, vector float);
vector signed char	vec_vminsb(vector bool char, vector signed char);
vector signed char	vec_vminsb(vector signed char, vector bool char);
vector signed char	vec_vminsb(vector signed char, vector signed char);
vector signed short	vec_vminsh(vector bool short, vector signed short);
vector signed short	vec_vminsh(vector signed short, vector bool short);
vector signed short	vec_vminsh(vector signed short, vector signed short);
vector signed int	vec_vminsw(vector bool long, vector signed int);
vector signed int	vec_vminsw(vector signed int, vector bool long);
vector signed int	vec_vminsw(vector signed int, vector signed int);
vector unsigned char	vec_vminub(vector bool char, vector unsigned char);
vector unsigned char	vec_vminub(vector unsigned char, vector bool char);
vector unsigned char	vec_vminub(vector unsigned char, vector unsigned char);
vector unsigned short	vec_vminuh(vector bool short, vector unsigned short);
vector unsigned short	vec_vminuh(vector unsigned short, vector bool short);
vector unsigned short	vec_vminuh(vector unsigned short, vector unsigned short);
vector unsigned int	vec_vminuw(vector bool long, vector unsigned int);
vector unsigned int	vec_vminuw(vector unsigned int, vector bool long);
vector unsigned int	vec_vminuw(vector unsigned int, vector unsigned int);
vector signed short	vec_vmladduhm(vector signed short, vector signed short, vector signed short);
vector signed short	vec_vmladduhm(vector signed short, vector unsigned short, vector unsigned short);
vector signed short	vec_vmladduhm(vector unsigned short, vector signed short, vector signed short);
vector unsigned short	vec_vmladduhm(vector unsigned short, vector unsigned short, vector unsigned short);
vector bool char	vec_vmrghb(vector bool char, vector bool char);

vector signed char	vec_vmrghb(vector signed char, vector signed char);
vector unsigned char	vec_vmrghb(vector unsigned char, vector unsigned char);
vector bool short	vec_vmrghh(vector bool short, vector bool short);
vector pixel	vec_vmrghh(vector pixel, vector pixel);
vector signed short	vec_vmrghh(vector signed short, vector signed short);
vector unsigned short	vec_vmrghh(vector unsigned short, vector unsigned short);
vector bool long	vec_vmrghw(vector bool long, vector bool long);
vector float	vec_vmrghw(vector float, vector float);
vector signed int	vec_vmrghw(vector signed int, vector signed int);
vector unsigned int	vec_vmrghw(vector unsigned int, vector unsigned int);
vector bool char	vec_vmrglb(vector bool char, vector bool char);
vector signed char	vec_vmrglb(vector signed char, vector signed char);
vector unsigned char	vec_vmrglb(vector unsigned char, vector unsigned char);
vector bool short	vec_vmrglh(vector bool short, vector bool short);
vector pixel	vec_vmrglh(vector pixel, vector pixel);
vector signed short	vec_vmrglh(vector signed short, vector signed short);
vector unsigned short	vec_vmrglh(vector unsigned short, vector unsigned short);
vector bool long	vec_vmrglw(vector bool long, vector bool long);
vector float	vec_vmrglw(vector float, vector float);
vector signed int	vec_vmrglw(vector signed int, vector signed int);
vector unsigned int	vec_vmrglw(vector unsigned int, vector unsigned int);
vector signed int	vec_vmsummbm(vector signed char, vector unsigned char, vector signed int);
vector signed int	vec_vmsumshm(vector signed short, vector signed short, vector signed int);
vector signed int	vec_vmsumshs(vector signed short, vector signed short, vector signed int);
vector unsigned int	vec_vmsumubm(vector unsigned char, vector unsigned char, vector unsigned int);
vector unsigned int	vec_vmsumuhm(vector unsigned short, vector unsigned short, vector unsigned int);
vector unsigned int	vec_vmsumuhs(vector unsigned short, vector unsigned short, vector unsigned int);
vector signed short	vec_vmulesb(vector signed char, vector signed char);

vector signed int	vec_vmulesh(vector signed short, vector signed short);
vector unsigned short	vec_vmuleub(vector unsigned char, vector unsigned char);
vector unsigned int	vec_vmuleuh(vector unsigned short, vector unsigned short);
vector signed short	vec_vmulosb(vector signed char, vector signed char);
vector signed int	vec_vmulosh(vector signed short, vector signed short);
vector unsigned short	vec_vmuloub(vector unsigned char, vector unsigned char);
vector unsigned int	vec_vmulouh(vector unsigned short, vector unsigned short);
vector float	vec_vnmsubfp(vector float, vector float, vector float);
vector bool short	vec_vnor(vector bool short, vector bool short);
vector bool long	vec_vnor(vector bool long, vector bool long);
vector bool char	vec_vnor(vector bool char, vector bool char);
vector float	vec_vnor(vector float, vector float);
vector signed short	vec_vnor(vector signed short, vector signed short);
vector signed int	vec_vnor(vector signed int, vector signed int);
vector signed char	vec_vnor(vector signed char, vector signed char);
vector unsigned short	vec_vnor(vector unsigned short, vector unsigned short);
vector unsigned int	vec_vnor(vector unsigned int, vector unsigned int);
vector unsigned char	vec_vnor(vector unsigned char, vector unsigned char);
vector bool short	vec_vor(vector bool short, vector bool short);
vector signed short	vec_vor(vector bool short, vector signed short);
vector unsigned short	vec_vor(vector bool short, vector unsigned short);
vector bool long	vec_vor(vector bool long, vector bool long);
vector float	vec_vor(vector bool long, vector float);
vector signed int	vec_vor(vector bool long, vector signed int);
vector unsigned int	vec_vor(vector bool long, vector unsigned int);
vector bool char	vec_vor(vector bool char, vector bool char);
vector signed char	vec_vor(vector bool char, vector signed char);
vector unsigned char	vec_vor(vector bool char, vector unsigned char);
vector float	vec_vor(vector float, vector bool long);
vector float	vec_vor(vector float, vector float);
vector signed short	vec_vor(vector signed short, vector bool short);

vector signed short	vec_vor(vector signed short, vector signed short);
vector signed int	vec_vor(vector signed int, vector bool long);
vector signed int	vec_vor(vector signed int, vector signed int);
vector signed char	vec_vor(vector signed char, vector bool char);
vector signed char	vec_vor(vector signed char, vector signed char);
vector unsigned short	vec_vor(vector unsigned short, vector bool short);
vector unsigned short	vec_vor(vector unsigned short, vector unsigned short);
vector unsigned int	vec_vor(vector unsigned int, vector bool long);
vector unsigned int	vec_vor(vector unsigned int, vector unsigned int);
vector unsigned char	vec_vor(vector unsigned char, vector bool char);
vector unsigned char	vec_vor(vector unsigned char, vector unsigned char);
vector bool short	vec_vperm(vector bool short, vector bool short, vector unsigned char);
vector bool long	vec_vperm(vector bool long, vector bool long, vector unsigned char);
vector bool char	vec_vperm(vector bool char, vector bool char, vector unsigned char);
vector float	vec_vperm(vector float, vector float, vector unsigned char);
vector pixel	vec_vperm(vector pixel, vector pixel, vector unsigned char);
vector signed short	vec_vperm(vector signed short, vector signed short, vector unsigned char);
vector signed int	vec_vperm(vector signed int, vector signed int, vector unsigned char);
vector signed char	vec_vperm(vector signed char, vector signed char, vector unsigned char);
vector unsigned short	vec_vperm(vector unsigned short, vector unsigned short, vector unsigned char);
vector unsigned int	vec_vperm(vector unsigned int, vector unsigned int, vector unsigned char);
vector unsigned char	vec_vperm(vector unsigned char, vector unsigned char, vector unsigned char);
vector pixel	vec_vpkpx(vector unsigned int, vector unsigned int);
vector signed char	vec_vpkshss(vector signed short, vector signed short);
vector unsigned char	vec_vpkshus(vector signed short, vector signed short);
vector signed short	vec_vpkswss(vector signed int, vector signed int);
vector unsigned short	vec_vpkswus(vector signed int, vector signed int);
vector bool char	vec_vpkuhum(vector bool short, vector bool short);

vector signed char	vec_vpkuhum(vector signed short, vector signed short);
vector unsigned char	vec_vpkuhum(vector unsigned short, vector unsigned short);
vector unsigned char	vec_vpkuhus(vector unsigned short, vector unsigned short);
vector bool short	vec_vpkuwum(vector bool long, vector bool long);
vector signed short	vec_vpkuwum(vector signed int, vector signed int);
vector unsigned short	vec_vpkuwum(vector unsigned int, vector unsigned int);
vector unsigned short	vec_vpkuwus(vector unsigned int, vector unsigned int);
vector float	vec_vrefp(vector float);
vector float	vec_vrfim(vector float);
vector float	vec_vrfin(vector float);
vector float	vec_vrfip(vector float);
vector float	vec_vrfiz(vector float);
vector signed char	vec_vrlb(vector signed char, vector unsigned char);
vector unsigned char	vec_vrlb(vector unsigned char, vector unsigned char);
vector signed short	vec_vrlh(vector signed short, vector unsigned short);
vector unsigned short	vec_vrlh(vector unsigned short, vector unsigned short);
vector signed int	vec_vrlw(vector signed int, vector unsigned int);
vector unsigned int	vec_vrlw(vector unsigned int, vector unsigned int);
vector float	vec_vrsqrtefp(vector float);
vector bool short	vec_vsel(vector bool short, vector bool short, vector bool short);
vector bool short	vec_vsel(vector bool short, vector bool short, vector unsigned short);
vector bool long	vec_vsel(vector bool long, vector bool long, vector bool long);
vector bool long	vec_vsel(vector bool long, vector bool long, vector unsigned int);
vector bool char	vec_vsel(vector bool char, vector bool char, vector bool char);
vector bool char	vec_vsel(vector bool char, vector bool char, vector unsigned char);
vector float	vec_vsel(vector float, vector float, vector bool long);
vector float	vec_vsel(vector float, vector float, vector unsigned int);
vector signed short	vec_vsel(vector signed short, vector signed short, vector bool short);
vector signed short	vec_vsel(vector signed short, vector signed short, vector unsigned short);
vector signed int	vec_vsel(vector signed int, vector signed int, vector bool long);
vector signed int	vec_vsel(vector signed int, vector signed int, vector unsigned int);

vector signed char	vec_vsel(vector signed char, vector signed char, vector bool char);
vector signed char	vec_vsel(vector signed char, vector signed char, vector unsigned char);
vector unsigned short	vec_vsel(vector unsigned short, vector unsigned short, vector bool short);
vector unsigned short	vec_vsel(vector unsigned short, vector unsigned short, vector unsigned short);
vector unsigned int	vec_vsel(vector unsigned int, vector unsigned int, vector bool long);
vector unsigned int	vec_vsel(vector unsigned int, vector unsigned int, vector unsigned int);
vector unsigned char	vec_vsel(vector unsigned char, vector unsigned char, vector bool char);
vector unsigned char	vec_vsel(vector unsigned char, vector unsigned char, vector unsigned char);
vector bool short	vec_vsl(vector bool short, vector unsigned short);
vector bool short	vec_vsl(vector bool short, vector unsigned int);
vector bool short	vec_vsl(vector bool short, vector unsigned char);
vector bool long	vec_vsl(vector bool long, vector unsigned short);
vector bool long	vec_vsl(vector bool long, vector unsigned int);
vector bool long	vec_vsl(vector bool long, vector unsigned char);
vector bool char	vec_vsl(vector bool char, vector unsigned short);
vector bool char	vec_vsl(vector bool char, vector unsigned int);
vector bool char	vec_vsl(vector bool char, vector unsigned char);
vector pixel	vec_vsl(vector pixel, vector unsigned short);
vector pixel	vec_vsl(vector pixel, vector unsigned int);
vector pixel	vec_vsl(vector pixel, vector unsigned char);
vector signed short	vec_vsl(vector signed short, vector unsigned short);
vector signed short	vec_vsl(vector signed short, vector unsigned int);
vector signed short	vec_vsl(vector signed short, vector unsigned char);
vector signed int	vec_vsl(vector signed int, vector unsigned short);
vector signed int	vec_vsl(vector signed int, vector unsigned int);
vector signed int	vec_vsl(vector signed int, vector unsigned char);
vector signed char	vec_vsl(vector signed char, vector unsigned short);
vector signed char	vec_vsl(vector signed char, vector unsigned int);
vector signed char	vec_vsl(vector signed char, vector unsigned char);
vector unsigned short	vec_vsl(vector unsigned short, vector unsigned short);

vector unsigned short	vec_vsl(vector unsigned short, vector unsigned int);
vector unsigned short	vec_vsl(vector unsigned short, vector unsigned char);
vector unsigned int	vec_vsl(vector unsigned int, vector unsigned short);
vector unsigned int	vec_vsl(vector unsigned int, vector unsigned int);
vector unsigned int	vec_vsl(vector unsigned int, vector unsigned char);
vector unsigned char	vec_vsl(vector unsigned char, vector unsigned short);
vector unsigned char	vec_vsl(vector unsigned char, vector unsigned int);
vector unsigned char	vec_vsl(vector unsigned char, vector unsigned char);
vector signed char	vec_vslb(vector signed char, vector unsigned char);
vector unsigned char	vec_vslb(vector unsigned char, vector unsigned char);
vector float	vec_vsldoi(vector float, vector float, int);
vector pixel	vec_vsldoi(vector pixel, vector pixel, int);
vector signed short	vec_vsldoi(vector signed short, vector signed short, int);
vector signed int	vec_vsldoi(vector signed int, vector signed int, int);
vector signed char	vec_vsldoi(vector signed char, vector signed char, int);
vector unsigned short	vec_vsldoi(vector unsigned short, vector unsigned short, int);
vector unsigned int	vec_vsldoi(vector unsigned int, vector unsigned int, int);
vector unsigned char	vec_vsldoi(vector unsigned char, vector unsigned char, int);
vector signed short	vec_vslh(vector signed short, vector unsigned short);
vector unsigned short	vec_vslh(vector unsigned short, vector unsigned short);
vector float	vec_vslo(vector float, vector signed char);
vector float	vec_vslo(vector float, vector unsigned char);
vector pixel	vec_vslo(vector pixel, vector signed char);
vector pixel	vec_vslo(vector pixel, vector unsigned char);
vector signed short	vec_vslo(vector signed short, vector signed char);
vector signed short	vec_vslo(vector signed short, vector unsigned char);
vector signed int	vec_vslo(vector signed int, vector signed char);
vector signed int	vec_vslo(vector signed int, vector unsigned char);
vector signed char	vec_vslo(vector signed char, vector signed char);
vector signed char	vec_vslo(vector signed char, vector unsigned char);
vector unsigned short	vec_vslo(vector unsigned short, vector signed char);

vector unsigned short	vec_vslo(vector unsigned short, vector unsigned char);
vector unsigned int	vec_vslo(vector unsigned int, vector signed char);
vector unsigned int	vec_vslo(vector unsigned int, vector unsigned char);
vector unsigned char	vec_vslo(vector unsigned char, vector signed char);
vector unsigned char	vec_vslo(vector unsigned char, vector unsigned char);
vector signed int	vec_vslw(vector signed int, vector unsigned int);
vector unsigned int	vec_vslw(vector unsigned int, vector unsigned int);
vector bool char	vec_vspltb(vector bool char, int);
vector signed char	vec_vspltb(vector signed char, int);
vector unsigned char	vec_vspltb(vector unsigned char, int);
vector bool short	vec_vsplth(vector bool short, int);
vector pixel	vec_vsplth(vector pixel, int);
vector signed short	vec_vsplth(vector signed short, int);
vector unsigned short	vec_vsplth(vector unsigned short, int);
vector signed char	vec_vspltisb(int);
vector signed short	vec_vspltish(int);
vector signed int	vec_vspltisw(int);
vector bool long	vec_vspltw(vector bool long, int);
vector float	vec_vspltw(vector float, int);
vector signed int	vec_vspltw(vector signed int, int);
vector unsigned int	vec_vspltw(vector unsigned int, int);
vector bool short	vec_vsr(vector bool short, vector unsigned short);
vector bool short	vec_vsr(vector bool short, vector unsigned int);
vector bool short	vec_vsr(vector bool short, vector unsigned char);
vector bool long	vec_vsr(vector bool long, vector unsigned short);
vector bool long	vec_vsr(vector bool long, vector unsigned int);
vector bool long	vec_vsr(vector bool long, vector unsigned char);
vector bool char	vec_vsr(vector bool char, vector unsigned short);
vector bool char	vec_vsr(vector bool char, vector unsigned int);
vector bool char	vec_vsr(vector bool char, vector unsigned char);
vector pixel	vec_vsr(vector pixel, vector unsigned short);

vector pixel	vec_vsr(vector pixel, vector unsigned int);
vector pixel	vec_vsr(vector pixel, vector unsigned char);
vector signed short	vec_vsr(vector signed short, vector unsigned short);
vector signed short	vec_vsr(vector signed short, vector unsigned int);
vector signed short	vec_vsr(vector signed short, vector unsigned char);
vector signed int	vec_vsr(vector signed int, vector unsigned short);
vector signed int	vec_vsr(vector signed int, vector unsigned int);
vector signed int	vec_vsr(vector signed int, vector unsigned char);
vector signed char	vec_vsr(vector signed char, vector unsigned short);
vector signed char	vec_vsr(vector signed char, vector unsigned int);
vector signed char	vec_vsr(vector signed char, vector unsigned char);
vector unsigned short	vec_vsr(vector unsigned short, vector unsigned short);
vector unsigned short	vec_vsr(vector unsigned short, vector unsigned int);
vector unsigned short	vec_vsr(vector unsigned short, vector unsigned char);
vector unsigned int	vec_vsr(vector unsigned int, vector unsigned short);
vector unsigned int	vec_vsr(vector unsigned int, vector unsigned int);
vector unsigned int	vec_vsr(vector unsigned int, vector unsigned char);
vector unsigned char	vec_vsr(vector unsigned char, vector unsigned short);
vector unsigned char	vec_vsr(vector unsigned char, vector unsigned int);
vector unsigned char	vec_vsr(vector unsigned char, vector unsigned char);
vector signed char	vec_vsrab(vector signed char, vector unsigned char);
vector unsigned char	vec_vsrab(vector unsigned char, vector unsigned char);
vector signed short	vec_vsrah(vector signed short, vector unsigned short);
vector unsigned short	vec_vsrah(vector unsigned short, vector unsigned short);
vector signed int	vec_vsraw(vector signed int, vector unsigned int);
vector unsigned int	vec_vsraw(vector unsigned int, vector unsigned int);
vector signed char	vec_vsrb(vector signed char, vector unsigned char);
vector unsigned char	vec_vsrb(vector unsigned char, vector unsigned char);
vector signed short	vec_vsrh(vector signed short, vector unsigned short);
vector unsigned short	vec_vsrh(vector unsigned short, vector unsigned short);
vector float	vec_vsro(vector float, vector signed char);

vector float	vec_vsro(vector float, vector unsigned char);
vector pixel	vec_vsro(vector pixel, vector signed char);
vector pixel	vec_vsro(vector pixel, vector unsigned char);
vector signed short	vec_vsro(vector signed short, vector signed char);
vector signed short	vec_vsro(vector signed short, vector unsigned char);
vector signed int	vec_vsro(vector signed int, vector signed char);
vector signed int	vec_vsro(vector signed int, vector unsigned char);
vector signed char	vec_vsro(vector signed char, vector signed char);
vector signed char	vec_vsro(vector signed char, vector unsigned char);
vector unsigned short	vec_vsro(vector unsigned short, vector signed char);
vector unsigned short	vec_vsro(vector unsigned short, vector unsigned char);
vector unsigned int	vec_vsro(vector unsigned int, vector signed char);
vector unsigned int	vec_vsro(vector unsigned int, vector unsigned char);
vector unsigned char	vec_vsro(vector unsigned char, vector signed char);
vector unsigned char	vec_vsro(vector unsigned char, vector unsigned char);
vector signed int	vec_vsrw(vector signed int, vector unsigned int);
vector unsigned int	vec_vsrw(vector unsigned int, vector unsigned int);
vector unsigned int	vec_vsubcuw(vector unsigned int, vector unsigned int);
vector float	vec_vsubfp(vector float, vector float);
vector signed char	vec_vsubsbs(vector bool char, vector signed char);
vector signed char	vec_vsubsbs(vector signed char, vector bool char);
vector signed char	vec_vsubsbs(vector signed char, vector signed char);
vector signed short	vec_vsubshs(vector bool short, vector signed short);
vector signed short	vec_vsubshs(vector signed short, vector bool short);
vector signed short	vec_vsubshs(vector signed short, vector signed short);
vector signed int	vec_vsubsws(vector bool long, vector signed int);
vector signed int	vec_vsubsws(vector signed int, vector bool long);
vector signed int	vec_vsubsws(vector signed int, vector signed int);
vector signed char	vec_vsububm(vector bool char, vector signed char);
vector unsigned char	vec_vsububm(vector bool char, vector unsigned char);
vector signed char	vec_vsububm(vector signed char, vector bool char);

vector signed char	vec_vsububm(vector signed char, vector signed char);
vector unsigned char	vec_vsububm(vector unsigned char, vector bool char);
vector unsigned char	vec_vsububm(vector unsigned char, vector unsigned char);
vector unsigned char	vec_vsububs(vector bool char, vector unsigned char);
vector unsigned char	vec_vsububs(vector unsigned char, vector bool char);
vector unsigned char	vec_vsububs(vector unsigned char, vector unsigned char);
vector signed short	vec_vsubuhm(vector bool short, vector signed short);
vector unsigned short	vec_vsubuhm(vector bool short, vector unsigned short);
vector signed short	vec_vsubuhm(vector signed short, vector bool short);
vector signed short	vec_vsubuhm(vector signed short, vector signed short);
vector unsigned short	vec_vsubuhm(vector unsigned short, vector bool short);
vector unsigned short	vec_vsubuhm(vector unsigned short, vector unsigned short);
vector unsigned short	vec_vsubuhs(vector bool short, vector unsigned short);
vector unsigned short	vec_vsubuhs(vector unsigned short, vector bool short);
vector unsigned short	vec_vsubuhs(vector unsigned short, vector unsigned short);
vector signed int	vec_vsubuwm(vector bool long, vector signed int);
vector unsigned int	vec_vsubuwm(vector bool long, vector unsigned int);
vector signed int	vec_vsubuwm(vector signed int, vector bool long);
vector signed int	vec_vsubuwm(vector signed int, vector signed int);
vector unsigned int	vec_vsubuwm(vector unsigned int, vector bool long);
vector unsigned int	vec_vsubuwm(vector unsigned int, vector unsigned int);
vector unsigned int	vec_vsubuws(vector bool long, vector unsigned int);
vector unsigned int	vec_vsubuws(vector unsigned int, vector bool long);
vector unsigned int	vec_vsubuws(vector unsigned int, vector unsigned int);
vector signed int	vec_vsum2sws(vector signed int, vector signed int);
vector signed int	vec_vsum4sbs(vector signed char, vector signed int);
vector signed int	vec_vsum4shs(vector signed short, vector signed int);
vector unsigned int	vec_vsum4ubs(vector unsigned char, vector unsigned int);
vector signed int	vec_vsumsws(vector signed int, vector signed int);
vector unsigned int	vec_vupkhpx(vector pixel);
vector bool short	vec_vupkhsb(vector bool char);

vector signed short	vec_vupkhsb(vector signed char);
vector bool long	vec_vupkhsh(vector bool short);
vector signed int	vec_vupkhsh(vector signed short);
vector unsigned int	vec_vupklpx(vector pixel);
vector bool short	vec_vupklsb(vector bool char);
vector signed short	vec_vupklsb(vector signed char);
vector bool long	vec_vupklsh(vector bool short);
vector signed int	vec_vupklsh(vector signed short);
vector bool short	vec_vxor(vector bool short, vector bool short);
vector signed short	vec_vxor(vector bool short, vector signed short);
vector unsigned short	vec_vxor(vector bool short, vector unsigned short);
vector bool long	vec_vxor(vector bool long, vector bool long);
vector float	vec_vxor(vector bool long, vector float);
vector signed int	vec_vxor(vector bool long, vector signed int);
vector unsigned int	vec_vxor(vector bool long, vector unsigned int);
vector bool char	vec_vxor(vector bool char, vector bool char);
vector signed char	vec_vxor(vector bool char, vector signed char);
vector unsigned char	vec_vxor(vector bool char, vector unsigned char);
vector float	vec_vxor(vector float, vector bool long);
vector float	vec_vxor(vector float, vector float);
vector signed short	vec_vxor(vector signed short, vector bool short);
vector signed short	vec_vxor(vector signed short, vector signed short);
vector signed int	vec_vxor(vector signed int, vector bool long);
vector signed int	vec_vxor(vector signed int, vector signed int);
vector signed char	vec_vxor(vector signed char, vector bool char);
vector signed char	vec_vxor(vector signed char, vector signed char);
vector unsigned short	vec_vxor(vector unsigned short, vector bool short);
vector unsigned short	vec_vxor(vector unsigned short, vector unsigned short);
vector unsigned int	vec_vxor(vector unsigned int, vector bool long);
vector unsigned int	vec_vxor(vector unsigned int, vector unsigned int);
vector unsigned char	vec_vxor(vector unsigned char, vector bool char);

vector unsigned char	vec_vxor(vector unsigned char, vector unsigned char);
vector bool short	vec_xor(vector bool short, vector bool short);
vector signed short	vec_xor(vector bool short, vector signed short);
vector unsigned short	vec_xor(vector bool short, vector unsigned short);
vector bool long	vec_xor(vector bool long, vector bool long);
vector float	vec_xor(vector bool long, vector float);
vector signed int	vec_xor(vector bool long, vector signed int);
vector unsigned int	vec_xor(vector bool long, vector unsigned int);
vector bool char	vec_xor(vector bool char, vector bool char);
vector signed char	vec_xor(vector bool char, vector signed char);
vector unsigned char	vec_xor(vector bool char, vector unsigned char);
vector float	vec_xor(vector float, vector bool long);
vector float	vec_xor(vector float, vector float);
vector signed short	vec_xor(vector signed short, vector bool short);
vector signed short	vec_xor(vector signed short, vector signed short);
vector signed int	vec_xor(vector signed int, vector bool long);
vector signed int	vec_xor(vector signed int, vector signed int);
vector signed char	vec_xor(vector signed char, vector bool char);
vector signed char	vec_xor(vector signed char, vector signed char);
vector unsigned short	vec_xor(vector unsigned short, vector bool short);
vector unsigned short	vec_xor(vector unsigned short, vector unsigned short);
vector unsigned int	vec_xor(vector unsigned int, vector bool long);
vector unsigned int	vec_xor(vector unsigned int, vector unsigned int);
vector unsigned char	vec_xor(vector unsigned char, vector bool char);
vector unsigned char	vec_xor(vector unsigned char, vector unsigned char);

Atomic memory access

SNC provides intrinsic functions for atomic memory access.

The following builtin functions return the value of the variable before performing the operations suggested by the name:

```
type __sync_fetch_and_add (type *ptr, type value, ...);
type __sync_fetch_and_sub (type *ptr, type value, ...);
type __sync_fetch_and_or (type *ptr, type value, ...);
type __sync_fetch_and_and (type *ptr, type value, ...);
type __sync_fetch_and_xor (type *ptr, type value, ...);
type __sync_fetch_and_nand (type *ptr, type value, ...);
```

Examples:

```
{ tmp = *ptr; *ptr op= value; return tmp; }
{ tmp = *ptr; *ptr = ~tmp & value; return tmp; } // nand
```

The following builtin functions return the value of the variable after performing the operations suggested by the name:

```
type __sync_add_and_fetch (type *ptr, type value);
type __sync_sub_and_fetch (type *ptr, type value);
type __sync_or_and_fetch (type *ptr, type value);
type __sync_and_and_fetch (type *ptr, type value);
type __sync_xor_and_fetch (type *ptr, type value);
type __sync_nand_and_fetch (type *ptr, type value);
```

Examples:

```
{ *ptr op= value; return *ptr; }
{ *ptr = ~*ptr & value; return *ptr; } // nand
```

The following builtin functions perform an atomic compare, and swap. If `oldval` matches the current value of `*ptr`, then write `newval` into `*ptr`. The boolean version of this function returns `true` if the swap occurred and `newval` was written to `*ptr`.

```
type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)
bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...)
```

The following builtin function issues a full memory barrier.

```
void __sync_synchronize (...)
```

The following builtin function is an atomic exchange operation. It writes `value` into `*ptr`, and returns the previous contents of `*ptr`.

```
type __sync_lock_test_and_set (type *ptr, type value, ...)
```

The following builtin function releases the lock acquired by `__sync_lock_test_and_set`.

```
void __sync_lock_release (type *ptr, ...)
```

In each of the functions, `type` can be one of the following types:

- int
- unsigned int
- long
- unsigned long
- long long
- unsigned long long

> **Note:** Support for 8 and 16-bit data types will be added at a later date.

11: Type traits pseudo-function reference

The compiler supports type traits, which indicate various characteristics of a type at compile time.

Type traits pseudo-functions

The table below lists the type traits supported by SNC Compiler. All type traits return **false** if the condition specified by the type trait name is not met.

For practical purposes when applied to a class, the term trivial means that the class does not have any virtual functions or virtual base classes , that its base classes are also trivial, and that its non-static members are of trivial types. When trivial is applied to constructors, it means that they are not user-defined and that their classes are trivial.

Please refer to the GNU documentation for an exact definition for Type Traits.

Type traits	Description
`__has_nothrow_assign (type)`	Returns **true** if a copy assignment operator's exception specification is empty.
`__has_nothrow_constructor (type)`	Returns **true** if the default constructor's exception specification is empty.
`__has_nothrow_copy (type)`	Returns **true** if the copy constructor's exception specification is empty.
`__has_nothrow_move_assign(type)`	Returns **true** if the *type* has a move assignment operator with an empty exception specification.
`__has_trivial_assign (type)`	Returns **true** if the *type* has a trivial, compiler-generated assignment operator.
`__has_trivial_constructor (type)`	Returns **true** if the *type* has a trivial, compiler-generated constructor.
`__has_trivial_copy (type)`	Returns **true** if the *type* has a trivial, compiler-generated copy constructor.
`__has_trivial_destructor (type)`	Returns **true** if the *type* has a trivial, compiler-generated destructor.
`__has_trivial_move_assign(type)`	Returns **true** if the *type* has a trivial, move assignment operator.
`__has_trivial_move_constructor(type)`	Returns **true** if the *type* has a trivial, move constructor.
`__has_virtual_destructor(type)`	Returns **true** if the *type* has a virtual destructor.
`__is_abstract (type)`	Returns **true** if the *type* is abstract. In addition, `__is_abstract` works for platform types. An interface with at least one member is an abstract type, as is a reference type with at least one abstract member.
`__is_base_of (base_type, derived_type)`	Returns **true** if the first *type* is a base class of the second *type*, if both types are the same.

Type traits	Description
	In addition, `__is_base_of` works on platform types. For example, this function will return **true** if the first *type* is an interface class and the second *type* implements the interface.
`__is_class (type)`	Returns **true** if the *type* is a native class or struct.
`__is_constructible(type)`	Returns **true** if a variable of the argument type can be defined.
`__is_convertible_to (type, type)`	Returns **true** if the first *type* can be converted to the second *type*.
`__is_destructible(type)`	Returns **true** if the argument *type's* destructor can be called.
`__is_empty (type)`	Returns **true** if there is no instance data members for the *type*.
`__is_enum (type)`	Returns **true** if the *type* is a native enum.
`__is_literal_type(type)`	Returns **true** if the *type* is a literal type (scalar type, reference type or trivial class type).
`__is_nothrow_assignable(type1, type2)`	Returns **true** if *type1* and *type2* are assignable and the assignment is known not to throw exceptions.
`__is_nothrow_constructible(type)`	Returns **true** if a variable of the argument *type* can be defined and its construction does not throw exceptions.
`__is_nothrow_destructible(type)`	Returns **true** if the argument *type's* destructor can be called and does not throw exceptions.
`__is_pod (type)`	Returns **true** if the *type* consists of Plain Old Data (POD). For example, a class or union which has no private or protected non-static members.
`__is_polymorphic (type)`	Returns **true** if a native *type* has virtual functions.
`__is_standard_layout (type)`	Returns **true** if a *type* is a standard-layout type.
`__is_trivial (type)`	Returns **true** if a *type* is a trivial type.
`__is_trivially_assignable(type1, type2)`	Returns **true** if *type1* and *type2* are assignable and the assignment performs only trivial operations.
`__is_trivially_constructible(type)`	Returns **true** if a variable of the argument *type* can be defined and its constructions only involve trivial operations.
`__is_trivially_copyable(type)`	Returns **true** if a *type* can be copied by performing only trivial operations.
`__is_trivially_destructible(type)`	Returns **true** if the argument *type's* destructor can be called and only performs trivial operations.

Type traits	Description
`__is_union (type)`	Returns **true** if a *type* is a union.

Example

This feature can be used in template metaprogramming:

```
template<typename _T> class MyClass
{
public:
    /// Return true if the template parameter was a POD type.
    bool isPodBased()
    {
        return __is_pod(_T);
    }

    // ...

};
```

12: Predefined macro reference

General predefined symbols

Name	Default Value	Description
__SNC__	1	Always enabled. Indicates that the program is being compiled by SNC.
__SN_VER__	varies	Version of the SN Compiler in the versioning format of the specific target.
__DATE__	"Mmm dd yyyy"	Date string in following format: "Feb 19 2009".
__TIME__	"hh:mm:ss"	Time string in following format: "15:38:03".
__EDG__	0	The __EDG__ is disabled by default for the PS3 PPU compiler by default as SNC now uses the GCC runtime libraries to allow link compatibility.
__EDG_RUNTIME_NAMESPACES	1	Indicates that the EDG front end uses namespaces.
__EDG_IA64_ABI	1	Defined as 1 to indicate that the compiler is using the IA-64 ABI.
__EDG_VERSION__	310	EDG version number.
__VERSION__	"EDG gcc 4.1.1 mode"	EDG version string.
__BOOL_IS_KEYWORD	1	Defined if bool is a keyword.
_BOOL_DEFINED	1	Defined if bool is a keyword.
__SIGNED_CHARS__	1	Used to modify the definition of CHAR_MIN and CHAR_MAX definition in limit.h.
__cplusplus	1	Defined if compilation is in C++ mode.
__WCHAR_T_IS_KEYWORD	1	Defined if wchar_t is a keyword.
_WCHAR_T_DEFINED	1	Defined if wchar_t is a keyword.
_NO_EX	1	Defined when Exception handling is disabled.
__EXCEPTIONS	undefined	Defined when exception handling is enabled.
__PLACEMENT_DELETE	undefined	Defined when exception handling is enabled.
__RTTI	1	Defined when RTTI is enabled in the compiler.
_M_IX86	undefined	Defined when Microsoft mode is specified.
_INTEGRAL_MAX_BITS	64	Defined when Microsoft mode is specified.
__STDC__	0	Defined in ANSI C mode and in C++ mode. In C++ mode the value may be redefined. Not defined in

		Microsoft compatibility mode.
__STDC_VERSION__	199901L	Defined in ANSI C mode with the value 199409L. The name of this macro, and its value, are specified in Normative Addendum 1 of the ISO C89 Standard. In C99 mode, defined with the value 199901L.
__STDC_HOSTED__	1	Indicates that SNC is a hosted implementation.

GNU mode symbols

Note that the GNU version symbol values are governed by the -Xgnuversion control-variable (see "-Xgnuversion"). The default values are "411" reflecting the fact that the compiler emulates GCC 4.1.1 by default.

Name	Default Value	Description
__GNUC__	4	Major GNUC version dialect accepted by the SN Compiler.
__GNUG__	4	Major GNUG version dialect accepted by the SN Compiler. Equivalent to (__GNUC__ && __cplusplus).
__GNUC_MINOR__	1	Minor GNUC version dialect accepted by the SN Compiler.
__GNUC_PATCHLEVEL__	1	Patch level macro defined by GCC from version 3.0.
__ELF__	1	Defined if the target uses the ELF object file format.

Target-specific symbols

Name	Default Value	Description
__PPU__	1	Application is targeted to run on the PPU.
__PPC__	1	Target architecture is PowerPC.
__PPC64__	1	Target architecture is PowerPC and that 64bit compilation mode is enabled.
__CELLOS_LV2__	1	Required for Havok libraries.
_ARCH_PPC64	1	Application is targeted to run on PowerPC processors with 64-bit support. (Required for SCE atomic header file).
__LP32__	1	Target platform uses 32 bits for int, long int, and pointer types.
__STRICT_ALIGNED	1	Required for SCE "aligned new" language extension where a variant of operator new is provided that adds an alignment parameter for types with non-standard alignment.
__thread	__declspec	Keyword __thread.

	(thread)	
__VEC__	10205	Support for vector data types.
__BIG_ENDIAN__	1	Target platform is big endian.
__ALTIVEC__	1	Support for vector data types.

Special macros

Name	Default Value	Description
__TIMESTAMP__	string constant	
__FILE__	string constant	Expands to a string constant of the name of the file that is under compilation.
__LINE__	string constant	Expands to the line number of the source file under compliation.
__COUNTER__	integer constant	Expands to an integer starting with 0 and incrementing by 1 every time it is used in a compilation.
__BASE_FILE__		Expands to a string constant of the name of the primary source file that is under compilation.
__SN_FILE__	string constant	Same as __FILE__.
__SN_BASE_FILE__	string constant	Same as __BASE_FILE__.

__has_feature Pseudo-macro

Use the __has_feature pseudo-macro within #if preprocessor directives to determine if the compiler supports a specified feature or attribute. __has_feature takes a single identifier argument, which is the name of a feature or attribute, and returns 1 if it is standardized in the current language standard, or 0 if not.

Note: The __has_feature identifiers are part of the C++11 standard and are therefore only available when enabled with the -Xstd=cpp11 compiler switch and when compiling C++ code.

__has_feature identifiers

The table below lists the feature identifiers recognized by the __has_feature pseudo-macro.

Feature identifier	Determines
cxx_access_control_sfinae	Whether access-control errors (for example, calling a private constructor), are considered to be template argument deduction errors, also known as 'substitution failure is not an error' (SFINAE) errors.
cxx_attributes	If support for attribute parsing with the square bracket notation of C++11 is enabled.
cxx_auto_type	Whether C++11 type inference is supported using the auto specifier. If this is disabled,

	`auto` will be a storage class specifier instead, as in C or C++98.
`cxx_decltype`	If support for the `decltype()` specifier is enabled. The decltype of C++11 does not require type-completeness of a function call expression. Use `__has_feature(cxx_decltype_incomplete_return_types)` to determine if support for this feature is enabled.
`cxx_defaulted_functions`	If support for defaulted function definitions (with = default) is enabled.
`cxx_default_function_template_args`	If support for default template arguments in function templates is enabled.
`cxx_deleted_functions`	If support for deleted function definitions (with = delete) is enabled.
`cxx_lambdas`	If support for `lambdas` is enabled.
`cxx_local_type_template_args`	If support for local and unnamed types as template arguments is enabled.
`cxx_nullptr`	If support for `nullptr` is enabled.
`cxx_rvalue_references`	If support for `rvalue` references is enabled.
`cxx_static_assert`	If support for compile-time assertions using `static_assert` is enabled.
`cxx_strong_enums`	If support for strongly typed, scoped enumerations is enabled.
`cxx_trailing_return`	If support for the alternate function declaration syntax with trailing return type is enabled.
`cxx_variadic_templates`	If support for variadic templates is enabled.

The attribute identifiers currently recognized by the `__has_feature` pseudo-macro are listed below.

Attribute identifier	**Determines**
`cxx_align_attribute`	If support is enabled for the `align_attribute`.
`cxx_base_check_attribute`	If support is enabled for the `base_check_attribute`.
`cxx_carries_dependency_attribute`	If support is enabled for the `carries_dependency_attribute`.
`cxx_final_attribute`	If support is enabled for the `final_attribute`.
`cxx_hiding_attribute`	If support is enabled for the `hiding_attribute`.

`__cxx_noreturn_attribute__`	If support is enabled for the `noreturn_attribute`.
`__cxx_override_attribute__`	If support is enabled for the `override_attribute`.

For more information on the C++11 support provided by the SNC compiler, see "C++11 Support".

Example

In this example `__has_feature` is used to detect whether `nullptr` is available:

```
#if __has_feature (cxx_nullptr)
    // Some code
#elsef
    // Some other code
#endif
```

Useful links

- http://publib.boulder.ibm.com/infocenter/cellcomp/v9v111/index.jsp?topic=/com.ibm.xlcpp9.cell.doc/compiler_ref/platform_related.htm - describes many of the target-specific predefined macros.
- http://gcc.gnu.org/onlinedocs/cpp/Common-Predefined-Macros.html - describes the GCC predefined macros.

13: Index