Systems/C++
Compiler Version 2.25

Systems/C++
C++ Compiler
Version 2.25

Contents

How to use this book

This book describes the Systems/C++ compiler, **DCXX**. **DCXX** is used to compile C++ source code, producing assembly language source. This book describes how to run **DCXX**, how to assemble the generated output and special language features **DCXX** provides.

To learn more about the run-time environment, refer to the *Systems/C C Library* manual.

Systems/C++ also includes several utility programs used to manage the process of building OS/390 programs. For more information regarding these utilities, see the *Systems/C Utilities* manual.

For further information, contact Dignus, LLC at (919) 676-0847, or visit http://www.dignus.com.

The Systems/C++ C++ Compiler
DCXX

Systems/C++ Overview

Systems/C++ is a C++ language compiler for the 390 and zSeries architectures. It is unique in that its output is 390 assembly source code. Because of this, it provides features not typically found in C++ compilers for the mainframe.

Some of its features include:

- ANSI C++17 compliant compiler, including full support for templates, exceptions, RTTI and other modern C++ compiler features.
- Direct inline assembly source.
- IBM C++ compatibility mode when used with the Systems/ASM **DASM** assembler.
- GCC (g++) 4.x compatibility, including the C++ ABI used by g++.
- Recognized by IBM as supported for z/TPF.
- Support for 64-bit arithmetic (long long)
- C++ language extensions, including:
  - Support for Binary Floating Point (`_Ieee float`) and Hexadecimal Floating Point (`_Hexadec float`)
  - Built-in implementations for common C library functions
  - Support for remote function pointers (and the `__local` and `__remote` function pointer type qualifiers)
- Systems/C C++ library
  - The LLVM Project's libc++ STL and I/O library
  - Uses the distribution-provided C++ library under Linux or z/TPF

DCXX, the C++ compiler component of Systems/C++, generates assembly language ready to assemble on the mainframe.

Systems/C++ also supports cross-hosted development, where the compilation of C++ source occurs on a workstation. Systems/ASM can then be run on a workstation to translate the resulting assembly language source into mainframe object decks. Linking can also occur on a workstation (using **PLINK**), or the objects may be transferred to the mainframe for linking and binding.

Implementation Definitions

Implementation limits

What follows is a description of the compiler limits. In each case, the Systesm/C++ compiler, **DCXX**, meets or exceeds the requirements for ANSI C++. For most items the practical limit is imposed by the amount of memory available at compilation time.

When using the Systems/ASM assembler, **DASM**, to assemble the generated code, the character limit for external identifiers is 4096 characters. Note that C++ identifiers are typically augmented with type information to support symbol overloading. The 4096 character limit includes any characters added to the name. This limit is imposed by the Systems/ASM assembler.

The Linux assembler, *as*, has no limit on the length of external identifiers.

Basic Data Types and Alignments

The default signedness for `char` is `unsigned`, making `char` equivalent to `unsigned char`.

The type `char`, either `signed` or `unsigned`, is 8 bits long, and aligned on 8 bit (1 byte) boundaries.

The type `short`, either `signed` or `unsigned`, is 16 bits long, and aligned on 16 bit (2 byte) boundaries.

The type `int`, either `signed` or `unsigned`, is 32 bits long and aligned on 32 bit (fullword) boundaries.

The type `long`, either `signed` or `unsigned`, is 32 bits long and aligned on 32 bit (fullword) boundaries. If the *–mlp64* option is enabled, `long` is is 64 bits long and aligned on 64 bit (doubleword) boundaries.

The type `long long`, either `signed` or `unsigned`, is 64 bits long and aligned on 64 bit (doubleword) boundaries.

The type `float` is 32 bits long and aligned on 32 bit (fullword) boundaries. If the *–fieee* option is enabled, floating point constants and values are in IEEE format, otherwise they are in IBM HFP format.

The type `double` is 64 bits long and aligned on 64 bit (doubleword) boundaries, except in a formal parameter list, where *double* is aligned on 32 bit boundaries for MVS, z/OS, IBM compatibility -m ode, CMS and VSE. Linux/390, z/Linux and z/TPF alignments follow the rules of t he appropriate Application Binary Interface.

The type `long double` is 128 bits long and aligned on 64 bit (doubleword) boundaries. The *–mlong-double-64* option can be specified, in which case the `long double` type is treated the same as the `double` type.

Each of the floating point types can be modified by the keywords `_Ieee` or `_Hexadec`, to select either the IEEE BFP or IBM HFP format, respectively. For example, `_Ieee float` or `_Hexadec float`. If the format is not specified, then the default according to the *–fieee* command-line option is applied.

Bitfields are allocated left-to-right within the field, and may be `signed` or `unsigned`. A bitfield is considered `unsigned` unless explicitly declared `signed`. Bitfields may cross storage boundaries. The *–fansi_bitfield_packing* option can alter the packing of bitfields.

By default, enumerations have the type `signed int`. The *–fenum=size* and *–fc370=version* options can alter this behavior to tailor enumeration sizes.

If *–mlp64* is specified, pointers are 64 bits long and aligned on 64 bit (doubleword) boundaries. If *–milp32* is specified, pointers are 32 bits long and aligned on 32 bit (fullword) boundaries. Pointers are assumed to be "clean", in the sense that the upper bits are assumed to be zero as indicated by the runtime addressing mode.

Return values

When returning values in code compiled for z/TPF or Linux (the *–fztpf* or *–flinux* option was specified), the compiler follows the Linux (either 64-bit or 32-bit) conventions defined in the appropriate Linux Elf Applications Binary Interface (ABI). When returning values in code compiled in IBM Compatibility mode, the compiler follows the Language Environment return conventions.

The following describes the return conventions for the default mode of operation in the compiler, which are also the conventions used in the Systems/C runtime environment.

Integral values are returned in register 15 (`R15`.) When *–mlp64* is specified registers 15 and 0 (`R15`+`R0`) are used for the 64 bit `long long` data type, with the most significant bits placed in register 15.

Note that integral values smaller than 32 bits are promoted to the full 32 bit value for returning. The upper bits of register 15 will be appropriately set based on the signedness of the return type. Then the *–mlp64* option is specified, values smaller than 64 bits are promoted to the full 64 bit value for returning in the register.

`Float`, `double` and `long double` floating point values are returned in floating point register 0 (`FP0`) or the register pair 0,2 (`FP0`,`FP2`) for `long double` values. An HFP `float` value when (*–fieee* is not specified, or the type is explicitly `_Hexadec float`) will always be promoted to `double` to fill the entire register. An IEEE `float` value is returned as a 32-bit IEEE value in floating pointer register 0 (`FP0`).

Structure values are returned via a parameter inserted at the beginning of the parameter list. This parameter points to space allocated by the calling function.

C++ Language Features

1998 ANSI Standard C++

When *–fcpp98* is specified on the commandline, Systems/C++ accepts the entire C++ language as defined by the ISO/IEC 14882:1998 standard, with the following exceptions:

- The `export` keyword for templates is not implemented
- A partial specialization of a class member template cannot be added outside of the class definition.

The ANSI C++ standard adds many features above tradition C++ defined in "*The Annotated C++ Reference Manual*" by Ellis and Stroustrup, the ARM. Systems/C++ implements the ANSI C++ standard. The following features are implemented in Systems/C++ and are not found in the ARM definition:

- The dependent statement of an `if`, `while`, `do-while`, or `for` is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.
- The expression tested in an `if`, `while`, `do-while`, or `for`, as the first operand of a "`?`" operator, or as an operand of the "`&&`", "`||`", or "`!`" operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted in the ARM.
- Qualified names are allowed in elaborated type specifiers.
- A global-scope qualifier is allowed in member references of the form `x.::A::B` and `p->::A::B`.
- The precedence of the third operand of the "`?`" operator is changed.
- If control reaches the end of the `main()` routine, and `main()` has an integral return type, it is treated as if a `return 0;` statement were executed.

- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.
- A functional-notation cast of the form `A()` can be used even if `A` is a class without a (nontrivial) constructor. The temporary created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have nontype template parameters.
- A reference to `const volatile` cannot be bound to an rvalue.
- Qualification conversions such as conversion from `T**` to `T const * const *` are allowed.
- Digraphs are recognized.
- Operator keywords (e.g., `and`, `bitand`, etc.) are recognized.
- Static data member declarations can be used to declare member constants.
- `wchar_t` is recognized as a keyword and a distinct type.
- `bool` is recognized.
- RTTI (run-time type information), including `dynamic_cast` and the `typeid` operator, is implemented.
- Declarations in tested conditions (in `if`, `switch`, `for`, and `while` statements) are supported.
- Array `new` and `delete` are implemented.
- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`) are implemented.
- Definition of a nested class outside its enclosing class is allowed.
- `mutable` is accepted on nonstatic data member declarations.
- Namespaces are implemented, including `using` declarations and directives. Access declarations are broadened to match the corresponding `using` declarations.
- Explicit instantiation of templates is implemented.
- The `typename` keyword is recognized.

- `explicit` is accepted to declare non-converting constructors.
- The scope of a variable declared in the `for-init-statement` of a `for` loop is in the scope of the loop (not the surrounding scope.)
- Member templates are implemented.
- The new specialization syntax (using "`template <>`") is implemented.
- Cv-qualifiers are retained on rvalues (in particular, on function return values.)
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between PODs and non-PODs within trivial constructors.
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversion.)
- `extern inline` functions are supported, and the default linkage for `inline` functions is external.
- A typedef name may be used in an explicit destructor call.
- Placement `delete` is implemented.
- An array allocated via a placement `new` can be deallocated via `delete`.
- Covariant return types on overriding virtual functions are supported.
- `enum` types are considered to be non-integral types.
- Partial specialization of class templates is implemented.
- Partial ordering of function templates is implemented.
- Function declarations that match a function template are regarded as independent functions, not as "guiding declarations" that are instances of the template.
- It is possible to overload operators using functions that take `enum` types and no class types.
- Explicit specification of function template arguments is supported.
- Unnamed template parameters are supported.
- The new lookup rules for member references of the form `x.A::B` and `p->A::B` are supported.
- The notation `::template` (and `->template`, etc.) is supported.
- In a reference of the form `f()->g()`, with `g` a static member function, `f()` is evaluated, and likewise for a similar reference to a static data member. The ARM specifies that the left operand is not evaluated in such cases.

- `enum` types can contain values larger than can be contained in an `int`.
- Default arguments of function templates and member functions of class templates are instantiated only when the default argument is used in a call.
- String literals and wide string literals have `const` type.
- Class name injection is implemented.
- Argument-dependent (Koenig) lookup of function names is implemented.
- Class and function names declared only in unqualified friend declarations are not visible except for functions found by argument-dependent lookup.
- A `void` expression can be specified on a return statement in a `void` function.
- Function-try-blocks, i.e. try-blocks that are the top-level statements of functions, constructors, or destructors, are implemented.
- Universal character set escapes (e.g., `\uabcd`) are implemented.
- On a call in which the expression to the left of the opening parenthesis has class type, overload resolution looks for conversion functions that can convert the class object to pointer-to-function types, and each such pointed-to "surrogate function" type is evaluated alongside any other candidate functions.
- Dependent name lookup in templates is implemented. Nondependent names are looked up only in the context of the template definition. Dependent names are also looked up in the instantiation context, via argument-dependent lookup.

2011 ANSI Standard C++

Systems/C++ accepts the entire C++ language as defined by the ISO/IEC 14882:2011 standard, known as C++11. C++11 is accepted by default, but can be disabled with *–fcpp98* option.

The following enhancements over C++98 are supported:

- The `>>` token can be used to indicate two closing angle brackets, in addition to the right-shift operator.
- The `static_assert` declaration is supported.
- The `friend` specifier can indicate non-class types.
- Local and anonymous types can be template parameters.
- Mixed string literal concatenations are supported.

- C99 preprocessor extensions, including variadic macros, are supported.
- C99-style `_Pragma` is supported.
- `__func__` string is defined.
- Trailing comma in `enum` is accepted.
- Constants too large for `long` are now `long long`.
- `extern template` can be used to suppress instantiation.
- `typename` can appear outside of a template.
- `auto` is a valid type specifier.
- Trailing return type syntax (`func() -> type`) is supported.
- The `decltype` operator (a standardized replacement for the `typeof` extension) is supported.
- Scoped enumeration types (`enum class`) are supported.
- Lambda functions (closures) are supported (but `std::function` is not).
- Rvalue references (`&&`) are supported.
- Reference-qualifiers on implicit `this` argument are supported.
- Functions can be assigned to `delete` or to `default` to control automatic generation of special member functions and cast operations.
- Default move constructors and move assignment operators are generated.
- Explicit conversion functions can be supplied.
- The `sizeof`, `typeid`, and `decltype` operators can refer to non-static data members of classes.
- The `nullptr` keyword is supported.
- Certain declarator attributes can occur in double square brackets (`[[...]]`).
- The `final` keyword may appear on class types and virtual member functions.
- Alias declarations (with `using`) are supported.
- Variadic templates are supported.
- The `char16_t` and `char32_t` types are supported, along with their matching `u''/u""/U''/U""` literals.
- Access checking for base classes is performed in the context of the class being defined.
- `inline namespace` is supported.

- Initializer lists (`type var{...};`, etc.) are supported.
- The `noexcept` specifier and operator are supported (influenced by *–fimplicit_noexcept*).
- Range-based for loops (`for (auto i : container)`) are supported.
- Initializers can be provided for non-static class data members.
- The `constexpr` specifier is supported.
- Unrestricted unions are supported (controlled by *–funrestricted_unions*).
- Delegating constructors are supported.
- Raw strings (`R"(...)"`) and UTF-8 strings (`u8"..."`) are supported.

2014 ANSI Standard C++

Systems/C++ accepts the entire C++ language as defined by the ISO/IEC 14882:2014 standard, known as C++14. Use the *–fcpp14* option to enable support for C++14 (C++11 is accepted by default).

C++14 provides the following enhancements over C++11:

- Binary literals (such as `0b1010`) are accepted.
- Function return types can be deduced (`auto` as a return type).
- Lambdas can capture expressions as well as variables.
- Generic lambdas (`auto` as a parameter type) are accepted.
- Accept more statements inside of a `constexpr` function.
- Member initializers in classes.
- `[[deprecated]]` attribute is accepted.
- Multiple conversion functions in a class can be defined, with overload resolution selecting which one will be used.
- Apostrophes are ignored in numeric literals.
- Variable templates.
- Sized deallocation.

2017 ANSI Standard C++

Systems/C++ accepts the entire C++ language as defined by the ISO/IEC 14882:2017 standard, known as C++17. Use the *–fcpp17* option to enable support for C++17 (C++11 is accepted by default).

The following enhancements over C++14 are supported:

- `[[...]]` attributes on namespaces and enumerators.
- Nested namespace definitions.
- `u8` character literals.
- Extended `static_assert`.
- New rules for `auto` deduction from compound constant initializers.
- Allow `typename` in template template parameter list.
- Remove deprecated `register` keyword and `operator++(bool)`.
- `__has_include` to check for `#include` file availability.
- New attributes: `[[fallthrough]]`, `[[nodiscard]]`, `[[maybe_unused]]`.
- Generalized range-based `for` statement.
- Hexadecimal floating-point literals (were previously accepted by **DCXX** as an extension).
- `using` attribute namespaces without repetition.
- Standard and non-standard attributes.
- Structured bindings (returning a tuple).
- `if` and `switch` statements with embedded initializers.
- Fold expressions (apply a binary operator to a parameter pack).
- Lambda expressions within `constexpr` expressions.
- Lambda capture of `*this` by value.
- `if` statement with explicitly `constexpr` controlling expression.
- Exception specification as part of function type.
- `operator new` now accepts an alignment argument to honor C++11 `alignas` keyword.
- `inline` variables.

- Construction rules for `enum class` values.
- Constant evaluation for all non-type template arguments.
- Initialization from prvalues may elide copy or move constructors.
- Template deduction on constructors.

C++ language extensions accepted

As well as the ANSI C++ standard, Systems/C++ accepts the following common C++ language extensions:

- A `friend` declaration for a class may omit the `class` keyword:

```
class B;
class C {
   friend B; // Should be "friend class B"
};
```

- Constants of scalar type may be defined within classes (this is an old form; the modern form uses an initialized static data member):

```
class A {
   const int size = 10;
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A {
   int A::f(); // Should be int f()
};
```

- The preprocessing symbol `c_plusplus` is defined as well as `__cplusplus`.
- Except in IBM compatibility mode, implicit type conversion between a pointer to an `extern "C"` function and a pointer to an `extern "C++"` function is permitted. Here's an example:

```
extern "C" void f();    // f's type has extern "C"
                        //   linkage
void (*fp) ()           // fp points to an extern
                        //   "C++" function
        = &f;           // error in IBM mode.
```

  In z/TPF, Linux and Systems/C++ mode (non IBM compatible mode), this extension is allowed because C and C++ functions share the same calling convention.

  This extension is controlled by the *–fimplicit_extern_c_type_conversion* option.

- A "?" operator whose second and third operands are string literals or wide string literals can be implicitly converted to `char *` or `wchar_t *`. (Recall that in C++ string literals are `const`. There is a deprecated implicit conversion that allows conversion of a string literal to `char *`, dropping the `const`. That conversion, however, applies only to simple string literals. Allowing it for the result of a `?` operation is an extension.)

```
char *p = x ? "abc" : "def";
```

- Nonstatic local variables of an enclosing function can be referenced in a non-evaluated expression (e.g., a `sizeof` expression) inside a local class. A warning is issued.

- The `long long` and `unsigned long long` types are accepted.

- Integer constants suffixed by `LL` are given the type `long long`, and those suffixed by `ULL` are given the type `unsigned long long` (any of the suffix letters may be written in lower case.)

- The specifier `%lld` is recognized in `printf` and `scanf` formation strings.

- The `long long` types are accommodated in the usual arithmetic conversions.

- Bit fields may have base types that are enums or integral types besides `int` and `unsigned int`. This matches A.6.5.8 in the ANSI Common Extensions appendix.

- The last member of a `struct` may have an incomplete array type. It may not be the only member of the struct (otherwise, the struct would have zero size.) This is allowed only if the structure is "C-like".

- An extern array may have an incomplete `class`, `struct`, `union`, or `enum` type as its element type. The type must be completed before the array is subscripted (if it is).

- `enum` tags may be incomplete: one may define the tag name and resolve it (by specifying the brace-enclosed list) later.

- The values of enumeration constants may be given by expressions that evaluate to unsigned quantities that fit in the `unsigned int` range but not in the `int` range. A warning is issued for suspicious cases.

- An extra comma is allowed at the end of an `enum` list.

- The final semicolon preceding the closing } of a `struct` or `union` specifier may be omitted. A warning is issued.

- A label definition may be immediately followed by a right brace. (Normally, a statement must follow a label definition.) A warning is issued.

- An empty declaration (a semicolon with nothing before it) is allowed. A remark is issued.

- An initializer expression that is a single value and is used to initialize an entire static array, struct or union need not be enclosed in braces. ANSI C++ requires the braces.
- Benign redeclarations of `typedef` names are allowed. That is, a typedef name may be redeclared in the same scope as the same type.
- A pointer to `void` may be converted to or from a pointer to a function type.
- The nonstandard preprocessing directive `#include_next` is supported. This is a variant of the `#include` directive. It searches for the named file only in the locations on the search path that follow the location in which the current source file (the one containing the `#include_next` directive) is found.

Namespace Support

Namespaces are enabled by default, unless the *–fno_namespaces* option is used. The Systems/C++ library also assumes namespaces are supported.

When doing name lookup in a template instantiation, some names must be found in the context of the template definition while others may also be found in the context of a template instantiation. Systems/C++ implemented two different instantiation lookup algorithms: the one mandated by the C++ standard (referred to as "dependent name lookup"), and the one that existed before dependent name lookup was implemented.

Dependent name lookup is done when the *–fdep_name* option is specified.

Dependent Name Processing

When doing dependent name lookup, Systems/C++ implements the instantiation name lookup rules specified in the ANSI C++ standard. This processing requires that nonclass prototype instantiations be done. This in turn requires that the code be written using the `typename` and `template` keywords as required by the standard.

Lookup Using the Reference Context

When not using dependent name lookup, Systems/C++ uses a name lookup algorithm that approximates the two-phase lookup rule of the standard, but does so in such a way that allows most previously existing code to compile.

When searching for a name as part of a template instantiation, but not in the local context of the instantiation, the search is performed in a synthesized instantiation context that uses names from the context of the template definition and names from the context of the template instantiation. For example:

```
namespace N {
   int g(int);
   template <class T> struct A {
      T f(T t)  return g(t);
      T f()  return x;
   };
}

namespace M {
   int x = 99;
   double g(double);

   N::A<int> ai;
   int i = ai.f(0);      // N::A<int>::f(int)
                         //    calls N::g(int)
   int i2 = ai.f();      // N::A<int>::f()
                         //    returns 0 (= N::x)

   N::A<double> ad;
   double d = ad.f(0);   // N::A<double>::f(double)
                         //    calls M::g(double)
   double d2 = ad.f();   // N::A<double>::f()
                         //    also returns 0 (= N::x)
}
```

The search of names in template instantiation does not conform to the rules in the standard in the following respects:

- Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point at which the template was defined.
- Functions from the context in which the template was referenced are considered for all function calls in the template. Functions from the reference context should only visible for "dependent" function calls.

Argument-Dependent Lookup

When argument-dependent lookup is enabled, functions made visible using argument-dependent lookup overload with those made visible by normal lookup. The standard requires that this overloading occur even when the name found by normal lookup is a block `extern` declaration. Systems/C++ performs this overloading, but in default mode, argument-dependent lookup is suppressed when the normal lookup finds a block `extern`.

This means a program can have different behavior, depending on whether it is compiled with or without argument-dependent lookup, even if the program makes no use of namespaces. For example:

```
struct A { };
A operator+(A, double);
void f() {
   A a1;
   A operator+(A, int);
   a1 + 1.0;   // calls operator+(A, double) with
}              // arg-dependent lookup enabled,
               // but otherwise calls
               // operator+(A, int);
```

Template Instantiation

The C++ language defines the idea of templates. A template is a description of a data structure, or a function that is an example for an entire group of possible data types or functions. The template descriptions allows the type or function to be altered based on the specified template parameters. For example, in C++, a generic linked-list data structure could be implemented as templates — with the types of the values the linked-list contains parameterized. In the source, there could be `List<int>`, for a linked list of integers and `List<double>` for a linked list of floating point values. Based on the definition supplied in the source, the compiler provides each actual function or data type of a template — *instantiating* the template.

The instantiation of a class template is always done as soon as it is needed during the compilation. However, the instantiation of template functions, member functions of template classes, and static data members of template classes can be deferred. Allowing the deferral of these template entities is preferable, to reduce redundant definitions of data or functions, or to allow the program to specify which source should contain the instantiation (via a *specialization*.) Also, the C++ standard specifies that unreferenced template functions should not be compiled.

These instantiation requirements dictate that the compiler cannot be responsible for instantiation, as the decision has to be performed after examination of the entire program.

Systems/C++ provides mechanisms that can perform automatic instantiation at link time for Linux/390 and IBM compatibility mode. In normal mode, Systems/C++ allows for various options to assist the programmer. There are four instantiation modes, specified by the *–finstantiate=mode* option:

`none` Do not automatically create instantiations of any template entities.

`used` Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default when *–flinux* is specified. The compiler will generate the appropriate linker directives to ensure that any duplicate code is eliminated from the resulting program. In this manner, for Linux programs all template entities can be considered to be automatically instantiated.

`all` Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Nonmember template functions will be instantiated even if the only reference was a declaration. This is the default mode when *–fc370* is used, as it is how the IBM compiler operates. The generated assembler source in IBM mode will have specifications that instruct the linker not to declare duplicate definition errors, etc. Unfortunately, the IBM linker does not remove duplicate definitions, which can cause significant code expansion.

`local` Similar to `used`, except that the functions are given internal linkage. This mode can be used for non-Linux and non-IBM mode to provide "automatic" instantiation of templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly. However, more than one source module can have a local copy of the function, causing some increase in code. This is the default when neither *–flinux* nor *–fc370* is specified.

When *–flinux* is enabled, the default mode is `used`. The compiler will instantiate all referenced template entities and produce code that causes the linker to remove duplicate definitions.

When *–fc370* is enabled, the default mode is `all`, as it is with the IBM compiler. The compiler will instantiate all possible template entities. However, when *–fc370* is enabled, the compiler will also generate instructions that cause the IBM linker to ignore any duplicate definitions.

__int8, __int16, __int32, __int64

DCXX supports the `__int8`, `__int16`, `__int32` and `__int64` builtin data types similar to that offered by the Microsoft C++ compiler. These can be used to declare integers of 8-bits, 16-bits, 32-bits and 64-bits respectively.

These types are synonyms for types that have the same size. The __int8 type is the same as `char`, the `__int16` type is the same as `short`, __int32 is the same as `int`

and __int64 is the same as the `long long` type. When *–mlp64* is specified , `__int64` is the same as `long`.

Note that by default, `char` is `unsigned`, and thus `__int8` is `unsigned`, while the other types are `signed`, unless otherwise qualified.

These types are provided for compatibility with Microsoft.

Compiling, Linking and Running Programs

This chapter describes how to invoke the Systems/C++ C++ compiler, **DCXX**. It also explains how to assemble the resulting assembly language source, how to link modules to build an executable and how to run the resulting program. It is not intended to be a complete description of compilers or the C++ language; for that, one must consult other texts.

DCXX translates C++ source code into native IBM 390 assembly source, either in HLASM format or GNU GAS format. This source is ready to be assembled into object decks for linking on the mainframe.

Running DCXX

The **DCXX** command is used to compile programs and generate assembly source code.

In OS/390 and z/OS

In the OS/390 and z/OS environments, the compiler is executed by invoking the DCXX member of the Systems/C++ installation PDS, as installed. The options are specified in the *PARM* statement. Each option is separated by a comma and preceded with a dash. The first option that does not contain a dash names the input file to be compiled. An option which begins with the commercial at-sign (`@`), specifies a DDN from which to read other options.

For example, the following `EXEC` card will execute the compiler, directing the generated assembly source to the DD named *ASM*, as specified by the *–oASM* value in the *PARM* statement, as well as producing a compile listing on the DD named *LIST*.

```
//COMP EXEC PGM=DCXX,PARM='-oASM,-flisting=LIST'
```

The compiler reads from the DD *STDIN* if no other file is specified as the input file. Note that a comma is required to separate the arguments.

Another way to specify the same command using the –@ redirection option would be:

```
//COMP EXEC PGM=DCXX,PARM='-@PARMS'
//PARMS DD *
-oASM,-flisting=LIST
/*
```

In this example, the `-oASM,-flisting=LIST` options are specified in the file named `//DDN:PARMS` from the `PARM` option to the compiler. Using this technique allows for arbitrarily long command line options.

In Windows

In the Windows operating systems, the compiler is named **dcxx** and may be found in the installation directory. The command line is

`dcxx` *[options] input-file*

Options, if any, are preceded with a dash, –.

Unless otherwise specified, the generated assembly source is written to a default file. On OS/390 and z/OS, the default output assembler file is named `//DDN:ASMOUT`, on all other systems it is named `asm.out`.

The Windows version of **DCXX** supports the –`@`*filename* option. –`@`*filename* causes the compiler to read the file filename and insert its contents in the command line. This provides a mechanism for supporting arbitrarily long command line parameter lists.

In the UNIX/LINUX environment

In the UNIX environment, the compiler is named **dcxx**, and can be found in the installation directory. The command line is

`dcxx` *[options] input-file*

Options, if any, are preceded with a dash, –.

Unless otherwise specified, the generated assembly source is written to the file named `asm.out`.

Include File Processing

On OS/390 or z/OS

On OS/390 or z/OS, the C++ preprocessor follows typical rules when searching for `#include`d files. First, for names specified with double quotes, unless *–fnosearchlocal* is specified, an attempt is made to open the included file in the same "location" as the including file. If that fails, any include search list, specified with the *–I* or *–S* options is examined, in the order it was specified. The system include location is then examined.

When checking the system include locations, the same PDS but with "`.LIBCXX`" appended onto its name is searched first (i.e. "*SYSINCL*`.LIBCXX`".) This is so that any files provided by the C++ library to wrap around the C library (for example, to manage namespaces) will be found before the C library headers.

`#include` names are mapped so as to make the typical UNIX-style include names operate in a reasonable fashion on OS/390. First, if the name to be included begins with a *style-prefix* (i.e. `//DSN`, `//DDN`, ...) then no other processing is performed and the preprocessor attempts to open the specified file. If no *style-prefix* is specified, then the name is searched for UNIX-style directory delimiters (the forward-slash.) The last component of the file name is translated into a member name. The remaining forward-slash characters are translated into periods. Then, the search location is prefixed on the result to form the file name the preprocessor will attempt to open.

If the resulting prefix is `//DSN` or `//DDN`, any underscore characters (_) are then translated into pound-signs (`#`).

The resulting member name is further processed. All letters are transformed to upper-case, any period is transformed to the commercial at-sign (`@`). The name is truncated to 8 characters.

For example, if the systems `#include` directory (specified during installation) was set to `//DSN:SYSC.SYSINC`, then the line:

```
#include <machine/ansi.h>
```

would attempt to open

```
//DSN:SYSC.SYSINC.MACHINE(ANSI@H)
```

On UNIX, Linux and Windows

On UNIX, Linux and Windows, the C++ preprocessor follows typical UNIX rules when searching for `#include`d files. First, for names specified with double quotes, unless *–fnosearchlocal* is specified, an attempt is made to open the included file in the same directory as the including file. If that fails, the directory search list is examined, in the order it was specified on the command line with the *–I* or *–S* options. The system include directories are then examined.

For system includes (`#include <...>`), directories are searched in the following order:

1. The directories specified by any *–I* or *–S* options, in the order listed on the command line.
2. Any directories specified by *–isystem*.
3. The system include directory directory specified by `System Include` in the `dignus.inf` license file.
4. Any directories specified by *–idirafter*.

For local includes (`#include "..."`), the following directories are searched first:

1. The directory containing the current source.
2. Any directories specified by *–iquote*.

Then the search continues as if it was a system include file.

Header filename mapping ($$HDRMAP)

Header filename mapping is a facility which can map `#include` names specified in the original source file to other names, without changing the original source. This facility can be useful when moving source from mainframe to cross environments, or vice-versa. When the compiler first begins execution, it looks for mapping files in the `#include` search list named `$$HDRMAP`, which are used to specify this translation. `$$HDRMAP` processing occurs before any other host-specific mappings are applied.

DCXX maintains two separate mappings, one for system headers and another for user headers. The system header mapping consists of all of the `$$HDRMAP` files (not just the first one) that would be found if `#include <$$HDRMAP>` was encountered, and is used for resolving any headers included with `#include <...>`. The user header mapping consists of all of the `$$HDRMAP` files that would be found if `#include`

"$$HDRMAP" was encountered, and is used for resolving any headers included with #include "...".

Note that even though **DCXX** maintains two separate mappings, by default the *–fuser_sys_hdrmap* option is in effect, causing the user header map to be applied to all #includes, even system ones. If *–fnouser_sys_hdrmap* is specified, then only the system header map will be applied to system include directives.

Every name specified in a #include statement is searched for in the appropriate $$HDRMAP file. If the name is located in a $$HDRMAP file, the specified alternative name will be used instead of the original name from the source.

In a $$HDRMAP file, any line that begins with a pound sign (#), will be considered a comment line and ignored. Other lines specify the source and destination mapping and are of the form:

source destination

The specified source and destination are delimited by white space, and can optionally be enclosed within double quote characters. When enclosed within double quote characters, the names are treated as string literals.

Special lines specifying directory mappings are of the form:

DIR *source destination*

"DIR" is a keyword to indicate that this mapping applies to directories. Directory mappings apply to all of the text up to the last forward slash ("/"), allowing you to remap all files from one directory into another directory. Useful on systems like CMS where there is no nested directory heirarchy. Directory mappings are applied after any normal mappings.

For example, consider the following $$HDRMAP file, placed in the current working directory on a UNIX host:

```
# redirect header files to their UNIX filenames
DIR special very_special
SPEC#HDR special/header.h
LONGNAME "special/long name.h"
```

Then, if the original source contains

```
#include "SPEC#HDR"
```

the compiler will act as if the source contained

```
#include "very_special/header.h"
```

First it will map "`SPEC#HDR`" to "`special/header.h`", then it will convert that to "`very_special/header.h`" according to the `DIR` rule.

Note that the Systems/C++ library for OS/390 uses a `$$HDRMAP` file to translate the long header file names associated with typical C++ installations into unique short ones that can reside in a PDS.

Description of options

The options available to **DCXX** are summarized in the following table:

–D*name[=value]*	Add *name* to the list of C preprocessor definitions, optionally assigning it a value
–I*dir*	Add *dir* to the list of directories to examine for include files
–iquote *dir*	Add *dir* to the list of directories to examine for local include files.
–isystem *dir*	Add *dir* to the list of system include directories.
–idirafter *dir*	Add *dir* to the list of directories to search after the system include directories.
–S*dir*	Add *dir* to the list of directories to examine for include files, honoring IBM's `SEARCH` semantics.
–o*file*	Write the generated assembly language source to *file*
–E	Perform only the preprocessing step of compilation
–femitdefs	Include #define values in preprocessor (–E) output
–M*[=filename]*	Generate a source dependence list.
–fdep*[=filename]*	Generate a source dependence list during regular compilation.
–g	Generate debuggable code and provide extra debugging information
–g0	Disable debuggable code and debugging information
–gdwarf	Generate DWARF debugging information
–gstabs	Generate STABS debugging information
–gisd	Generate ISD debugging information
–fansi_bitfield_packing –fno_ansi_bitfield_packing	Follow/Ignore ANSI rules for bitfield allocation in structures
–fc370=*version*	Compile in C/370 compatibility mode

–fexportall	Provide DLL definitions for all defined data/functions
–fep=*name*	Specify a name that will be placed on the generated **END** card to denote a program entry point
–fprol=*macro*	Specify an alternate name for the function prologue macro
–fprv=*macro*	Specify an alternate name for the macro which supplies the address of the Pseudo-Register Vector
–fepil=*macro*	Specify an alternate name for the function epilogue macro
–fopts=*macro*	Request interesting options noted at top of generated assembly
–lnameaddr –fno_lnameaddr	Enable or disable generation of Logical Name Address info
–fendmacro=*macro*	Specify text to appear before the END statement
–finstrument_functions	Request function beginning/ending instrumentation
–fcode_base=*N*	Specify a register to use as the base register for executable code
–fframe_base=*N*	Specify a register to use as the base register for automatic data
–freserve_reg=*N*	Instruct the compiler that register *N* is reserved
–fwarn_disable=*N[,N,N-M,...]*	Disable particular warnings
–fwarn_enable=*N[,N,N-M,...]*	Re-enable particular warnings
–fwarn_promote=*N[,N,N-M,...]*	Promote a warning to an error
–ftrim	Remove trailing blanks from input
–faddh	Add "`.h`" to `#include` file names
–flowerh	Lower-case characters in `#include` file names
–fnosearchlocal	Specify that "local" searches for `#include` files should not be performed
–fpreinclude=*filename*	`#include` the named file before compiling the C++ source file

–flisting*[=filename]* –fno_listing	Generate a listing of the compilation
–fvtable_listing –fno_vtable_listing	Enable/disable virtual function table information to listing
–fpagesize=*n*	Set the listing page size to *n* lines
–fshowinc –fno_showinc	Enable/disable including source files from `#include` files in the listing
–fstructmap –fno_structmap	Enable/disable including struct layout information in the listing
–fstructmaphex –fno_structmaphex	Structure layout information should/shouldn't be displayed in hex
–frent	Generate re-entrant code
–fmaxerrcount	Limit the number of reported errors
–U*name*	Undefine `#define` macros
–fincstripdir	Remove directory components from `#include` names
–fincstripsuf	Conditionally remove suffixes from `#include` names
–fincrepsuf	Conditionally replace suffixes from `#include` names
–fmargins*[=m,n]*	Specify margins for source lines
–fmesg=*style*	Specify message style
–fasciiout –fnoasciiout	String and character constants will be in ASCII instead of EBCDIC
–fdollar	Allow the dollar-sign character ($) in identifiers
–fwchar_ucs	Indicate that wide character constants are UCS-2 or UCS-4
–fwchar=*n*	Specify the size of `wchar_t`
–fsname*[=name]*	Specify unique CSECT name
–fnosname	Tell the compiler to let **PLINK** assign CSECT names
–fsnameprefix=*char*	Explicitly set the section name prefix
–fieee	Enable support for IEEE floating point format

–fmrc –fnomrc	Enable/disable use of mainframe-style return code
–fpatch –fno_patch	Specify if a patch area should be generated
–fpatchmul=*n*	Alter the size of any generated patch area
–flinux	Enable Linux/390 code generation
–fvisibility=*setting*	Set ELF object symbol visibility
–fsigned_char –funsigned_char	Control if `char` is signed or unsigned by default
–fsuppress_vtbl	Suppress generation of C++ vtable information
–fforce_vtbl	Force generation of C++ vtable information
–finstantiate=*mode*	Set the template instantiation mode
–fdistinct_template_signatures –fno_distinct_template_signatures	Enable or disable distinct template signatures
–fimplicit_include –fno_implicit_include	Enable/disable implicit inclusion for template libraries
–ftempinc*[=directory]* –fno_tempinc	Enable/disable template instantiation method
–fnonstd_qualifier_deduction –fno_nonstd_qualifer_deduction	Enable or disable support for non-standard qualifier deduction
–fexceptions –fno_exceptions	Enable or disable support for C++ exceptions
–fguiding_decls –fno_guiding_decls	Imply template instantation with a specific declaration
–frtti –fno_rtti	Enable or disable support for C++ Run-Time Type Information
–farray_new_and_delete –fno_array_new_and_delete	Enable or disable support for special `new[]` and `delete[]` operators for arrays
–fexplicit –fno_explicit	Enable or disable support for the `explicit` keyword
–fnamespaces –fno_namespaces	Enable or disable support for C++ namespaces
–fold_for_init	Variables declared in `for()` initialization statements follow the pre-ANSI standard semantics

–fnew_for_init	Variables declared in `for()` initialization statements follow the ANSI standard semantics
–fold_specializations –fno_old_specializations	Enable or disable support for old-style template specializations
–fextern_inline –fno_extern_inline	Enable or disable support for `extern inline` functions
–fshort_lifetime_temps –flong_lifetime_temps	Enable or disable support for cfront-style long lifetime temporaries
–fbool –fno_bool	Enable or disable support for the `bool` keyword
–fwchar_t_keyword –fno_wchar_t_keyword	Enable or disable support for the `wchar_t` keyword
–ftypename –fno_typename	Enable or disable support for the `typename` keyword
–fimplicit_typename –fno_implicit_typename	Enable or disable support for determining if a template parameter is a type
–fdep_name –fno_dep_name	Enable or disable support for dependent name processing
–fparse_templates –fno_parse_templates	Enable or disable the parsing of templates
–fspecial_subscript_cost –fno_special_subscript_cost	Enable or disable the preferential use of an overloaded `[]` operator over a `*` operator for array access
–falternative_tokens –fno_alternative_tokens	Enable or disable support for the C++ alternative keywords tokens in comparisons
–fenum_overloading –fno_enum_overloading	Enable or disable support for overloading operations on enum-type operands
–fconst_string_literals –fno_const_string_literals	Treat string literals as `const` or non-`const`
–fimplicit_extern_c_type_conversion –fno_implicit_extern_c_type_conversion	Allow implicit conversions between C and C++ functions
–fclass_name_injection –fno_class_name_injection	Enable or disable injection of the name of a class into the class's scope
–farg_dependent_lookup –fno_arg_dependent_lookup	Enable or disable argument-dependent symbol lookup

–ffriend_injection –fno_friend_injection	Controls whether or not the name of a class or function declared only in friend declarations is visible
–flate_tiebreaker	`const` and/or `volatile` qualifiers are not used to distinguish function overloading
–fearly_tiebreaker	`const` and/or `volatile` qualifiers are used to distinguish function overloading
–fnonstd_using_decl –fno_nonstd_using_decl	Enable or disable the use of a nonmember using-declaration that specifies an unqualified name
–fvariadic_macros –fno_variadic_macros	Enable or disable support for C99 variadic macros
–fextended_variadic_macros –fno_extended_variadic_macros	Enable or disable support for GCC-style variadic macros
–fbase_assign_op_is_default –fno_base_assign_op_is_default	Enable or disable the use of a copy assignment operator that takes a reference to a base class as the default
–fignore_std –fno_ignore_std	Enable or disable the treatment of the `std` namespace as a synonym for the global namespace
–version	Print the compiler version number on STDOUT and exit
–famode=*val*	Specify the runtime addressing mode
–march=esa390,–march=esa390z	Enable ESA/390 architecture compilation
–march=z*N*	Enable use of the edition *N* of z/Architecture instructions
–mlp64	Enable 64-bit compilation, implies -march=z0
–milp32	Enable 32-bit compilation
–mfp16 –mfp4	Enable/disable use of extended FP registers
–mlong-double-128 –mlong-double-64	Enable/disable 128-bit long double type
–mmvcle –mno-mvcle	Enable/disable use of the MVCLE/CLCLE instructions
–mdistinct-operands –mno-distinct-operands	Enable/disable use of distinct-operands facility instructions

–mextended-immediate –mno-extended-immediate	Enable/disable use of distinct-operands facility instructions
–mload-store-on-condition –mno-load-store-on-condition	Enable/disable use of load/store-on-condition facility instructions
–mhfp-multiply-add –mno-hfp-multiply-add	Enable/disable use of HFP multiply-and-add facility instructions
–mlong-displacement –mno-long-displacement	Enable/disable use of long-displacement facility instructions
–mgeneral-instructions-extension –mno-general-instructions-extension	Enable/disable use of general-instructions-extension facility instructions
–mhigh-word-facility –mno-high-word-facility	Enable/disable use of high-word facility instructions
–mhfp-extensions –mno-hfp-extensions	Enable/disable use of HFP extensions facility instructions.
–finline*[=x[:y:z]]* –fnoinline	Control inlining optimization
–O*[n]*	Set optimization level
–fasmcomm=*mode*	Control the comments in the assembly output
–fasmlnno –fnoasmlnno	Include line numbers in C source comments in generated assembly
–fmin_lm_reg=*val*	Set the minimum number of registers in one `LM` instruction
–fmin_stm_reg=*val*	Set the minimum number of registers in one `STM` instruction
–fflex	Enable FLEX/ES-specific optimizations
–fpic	Generate position independent code, small GOT
–fPIC	Generate position independent code for Linux & z/TPF, large GOT
–ffpremote –ffplocal	Function pointers are remote/local
–fxplink	Use eXtra Performance Linkage
–fc370_extended	Enable C/370 LANGLVL(EXTENDED) compatibility mode

–fdll=cba –fdll=nocba	Enable/disable LE `DLL(CALLBACKANY)` support
–frsa=*size*	Specify the amount of space the compiler reserves for the Register Save Area
–fpack=*val*	Specify a default maximum structure alignment
–fuser_sys_hdrmap –fnouser_sys_hdrmap	Use user `$$HDRMAP` for system `#include`s
–fevents=*filename*	Emit an IBM-compatible events listing
–fnamemangling=*mode*	Select the name mangling mode to use for IBM compatibility
–fenum=*val*	Specify a default enumeration size
–ftest*[=name]*	Enable a separate test csect
–fprolkey=*key*	Append a global prologue key
–fsave_dsa_over_call –fno_save_dsa_over_call	Control if DSA bytes are saved and restored over alternate linkage call
–fcommon –fnocommon	Enable/disable common linkage for uninitialized globals
–fdfe –fnodfe	Enable/disable dead function elimination
–fmapat –fnomapat	Enable/disable mapping '@' to '_' in external symbol names
–fat –fnoat	Support @-operator in expressions
–fctrlz_is_eof –fno_ctrlz_is_eof	Enable/disable treating control-Z as an EOF character
–fpermissive_friend –fno_permissive_friend	Enable/disable friend declarations on private members
–ffnio –fno_fnio	Enable/disable function names in objects for debugging
–fhide_skipped –fshow_skipped	Enable/disable omission of preprocessor-skipped lines in listing
–fsigned_bitfields –funsigned_bitfields	Set default signedness of bitfields with bare types
–v	Print version information

–fsched_inst –fsched_inst2 –fno_sched_inst	Control the behavior of the instruction scheduler
–fxref –fno_xref	Enable/disable cross-reference listing
–frestrict –fno_restrict	Enable/disable C99-style `restrict` keyword
–fcpp98	Specify only C++98 will be accepted
–fcpp11	Enable support for C++11 language features
–fcpp14	Enable support for C++14 language features
–fcpp17	Enable support for C++17 language features
–funrestricted_unions –fno_unrestricted_unions	Enable/disable the C++11 unrestricted unions feature
–fimplicit_noexcept –fno_implicit_noexcept	Enable/disable the implicit C++11 exception specifications
–fstatic_anon_names –fno_static_anon_names	Enable/disable forcing members of the unnamed namespace to static
–fsource_enc=utf8 –fsource_enc=ascii	Select source character encoding
–fdwarf_extern –fno_dwarf_extern	Enable/disable generation of DWARF data for extern variables

Detailed description of the options

The –D option (define a macro)

The *–D* option defines a symbol in the same way as a *#define* preprocessor directive in C source code. It's usage is:

```
dcxx -Dmacro[=text]
```

For example:

```
dcxx -DMAXLEN=1024 prog.cxx
```

is equivalent to inserting the following C++ source line at the beginning of the program `prog.cxx`:

```
#define MAXLEN 1024
```

Because the *–D* option causes a C++ preprocessor symbol to be defined for the compilation, it can be used in conjunction with other preprocessor directives, such as `#ifdef` or `#if defined()` to implement conditional compilation. A common example of conditional compilation is:

```
#ifdef DEBUG
    printf("Entered function func()\n");
#endif
```

This debugging code would be included in the compiled object by adding *–DDEBUG* on the **DCXX** command line.

The –I option (specify additional locations to look for included files)

The *–I* option adds a specified location (a directory on UNIX and Windows) to the search list examined when source is `#include`d. The name of the directory immediately follows the *–I*, with no intervening spaces.

See the section on include file processing on page 26 for more details.

The –iquote *dir* option (Add *dir* to the list of directories to examine for local include files)

The *–iquote dir* option provides a local include search path, which is searched just for directives specified with a double quote character (`#include "..."`).

The –isystem *dir* option (Add *dir* to the list of system include directories)

The *–isystem dir* option provides a system search path, which is searched for any `#include` directives which are still not resolved after looking in the *–Idir* paths. These system search paths are treated like the `System Include` path in the `dignus.inf` license file, meaning that first the search is tried with `libcxx` appended to the path before the bare path. This way it will find C++ headers that wrap C ones (like `#include <stdio.h>`).

The –idirafter *dir* option (Add *dir* to the list of directories to search after the system include directories)

The *–idirafter dir* option provides an "after-system search path", which is searched for any `#include` directives which are still not resolved after looking in the *–isystem dir* paths or in the `System Include` path specified in the license file. These paths are also treated like system search paths, with the `libcxx` directory checked first.

The –S*dir* option (Add *dir* to the list of directories to examine for include files, honoring IBM's `SEARCH` semantics)

The *–Sdir* option is provided for compatibility with IBM's `SEARCH` parameter, and is useful for looking up headers in PDS-style datasets, especially when they have been transfered to a PC. The *–S* paths are searched in the order they were specified, just like *–I* paths.

When searching for a `#include` filename in a *–S* path, first the `#include` filename is uppercased and any underscores (`_`) are converted into at-signs (`@`). Then the filename is split into directory, member, and extension. Then the member name is truncated to 8 characters. Finally, the filename components are combined with the *–S* path in one of three ways, depending on how the *–S* is specified. If "`.*`" is used as a suffix, then the member and extension names are appended to it as if it was a DSN. If "`.+`" is the suffix, then directory and extension are appended to the DSN and the member name is treated as a member name. If there is no special suffix, then the member name is used as a member name, and the directory and extension are ignored.

For example, if `#include <dir/longfilename.ext>` is encountered, then here are some possible searches:

`-Spath.*`	Searches `path.EXT.LONGFILE`.
`-Spath.+`	Searches `path.DIR.EXT/LONGFILE` (or `path.DIR.EXT(LONGFILE)` on MVS).
`-Spath`	Searches `path/LONGFILE` (or `path(LONGFILE)` on MVS).

The –o*file* option (specify the name of the generated output file)

The *–ofile* option specifies the name of the generated assembly output file. If the file cannot be opened for writing, the compiler writes the generated assembly output to *stdout*. The usage of *–o* is as follows:

```
dcxx -ofilename prog.c
```

For example:

```
dcxx -omyfile.asm myfile.cxx
```

will compile the C++ source file *myfile.cxx* placing the generated assembly language source in the file *myfile.asm*.

If *file* is the single dash character (`-`), then the output is written to *stdout.*

The –E option (preprocess only)

The *–E* option instructs the compiler to execute the preprocessing phase of compilation only. No attempt is made to generate code. The output of the preprocessor is written to *stdout*.

The –femitdefs option (include #define values in preprocessor output)

The *–femitdefs* option causes the compiler to generate `#define` lines in the preprocessor output for every defined macro.

This can be utilized in complex configurations to determine where a `#define` was processed, and how it was defined. It can also be helpful in determining which macros were predefined or defined on the command line.

If the *–E* option isn't specified, *–femitdefs* is ignored.

The –M*[=filename]* option (generate a source dependence list)

The *–M* option causes the compiler to perform the preprocessing step only, and generate a dependency list suitable for including in a "makefile" on UNIX platforms.

The compiler will generate lines of the form:

target: *source*

where *target* is generated from the source file name (replacing any extension with `.o`). The *source* is the any source file the compiler read while performing the preprocessing step.

If the optional filename is provided, then the dependency list will be output to that file, otherwise it will be output to `stdout`.

The –fdep*[=filename]* option (generate a source dependence list during regular compilation)

The *–fdep* option has the same effect as *–M*, except that the compiler also performs the compilation step. Using *–fdep*, it is possible to generate the dependency list with every compilation, instead of as a separate step.

The –g option (debuggable code)

The *–g* option instructs the compiler to generate more information in the generated assembly language file. This extra information is generally helpful when debugging the generated code.

When *–flinux* or *–fztpf* is also specified, this option causes the compiler to generate DWARF version 3 debugging information suitable for use with the Linux debugger, *gdb*.

When *–fc370* is also specified, this causes the compiler to generate ISD format debugging information (referenced from PPA3/PPA4) for IBM compatibility.

Otherwise, DWARF version 3 debug information will be generated for processing by **PLINK**. Run **PLINK** with *–dbg=filename* to specify the side file that will be loaded by **DDBG**.

By default, when *–g* is specified, some optimizations (such as inlining) are disabled. However, debugging information may still be generated on optimized output. For this to happen, you must ensure that *–g* comes before *–O* (or *–finline*) on the command line, or the *–O* will be ignored.

The –g0 option (Disable debuggable code and debugging information)

The *–g0* option disables generation of debugging information, and re-enables inlining, as if the *–g* option had not been specified.

The –gdwarf option (generate DWARF debugging information)

The *–gdwarf* option is used to instruct the compiler to generate debugging information using the DWARF format instead of the STABS or ISD formats.

When *–flinux* is also enabled, the information is embedded within the object file.

When *–fc370* is also used, the DWARF information is placed in a side file. The filename defaults to the source file name with the extension replaced with "`.dbg`". The name of the side file may be manually specified with *–gdwarf=filename.dbg*. The name provided here is the name that the debugger will ultimately use to find the side file, so it may be necessary to manually specify a filename with a path that is valid on your debugging host. The side file is actually created by **DASM**, so If you want the side file to be placed in a different intermediate location on your build host than on your debugging host, you can specify a separate *–fdwarf=filename.dbg* on the **DASM** commandline to override the name used by **DCXX**.

Otherwise, the DWARF information is encoded in a special CSECT in each compilation unit which **PLINK** will put in a side file specified by its *–dbg=filename* option.

Note that *–gdwarf*, like *–g* must occur before any *–O* or *–finline* options which occur on the commandline in order to generate debugging information for optimized code.

The –gstabs option (generate STABS debugging information)

The *–gstabs* option is used to instruct the compiler to output legacy STABS debugging information.

The –gisd option (generate ISD debugging information)

The *–gisd* option is used to instruct the compiler to output legacy ISD debugging information, only for use in LE mode (*–fc370*) for compatibility with IBM tools.

The –fansi_bitfield_packing/–fno_ansi_bitfield_packing options (ANSI rules for bitfield allocation)

The *–fansi_bitfield_packing* option instructs the compiler to allocate bitfields in structures according to ANSI rules. This typically results in smaller structures, as it allows the compiler to pack bitfields as tightly as possible. When this option isn't enabled, the compiler will follow more traditional bitfield allocation rules. Enabling this option causes the compiler to allocate bitfields as the IBM C++ compiler does when the **LANGLVL** is set to `ANSI`. When the option is not enabled, the compiler will allocate bitfields in a manner compatible with IBM C++ when the **LANGLVL** is set to **COMMONC**.

When *–fansi_bitfield_packing* is not enabled, the compiler will pad structures to the bitfield type alignment requirements if the bitfield is the last element of the structure.

When *–fansi_bitfield_packing* is enabled, structures are not padded. The structure size is packed to a byte boundary sufficient to contain the number of bits specified by the bitfield.

The *–fno_ansi_bitfield_packing* option negates the effect of the *–fansi_bitfield_packing* option and is the default.

The –fc370=*version* option (specify IBM C++ compatibility)

The *–fc370* option specifies that the generated assembly language source is to be compatible with IBM C++ objects. In this mode, the compiler will generate function prologues and epilogues, data offsets, alignments and initializers that interoperate with IBM C++. Note that the generated assembly source must be processed by the Systems/ASM assembler to produce a correct object file. For more information, see the chapter on IBM C compatibility mode (page 147).

The value of *version* is used to indicate which version of IBM C++ is desired. Current supported *version* specifications are `v2r4`, `v2r6`, `v2r10`, and `z1r2` to `z1r13`. Language features, pre-defined macros, and prologue/epilogue function linkage conventions all change to be compatible with the specified version of IBM C++.

The –fexportall option (Provide DLL definitions for data/functions)

The –fexportall option causes the compiler to provide IBM DLL definitions for all defined data and functions in the compilation, in IBM compatibility mode.

Typically, to cause a datum or function to be visible to other code that uses an IBM DLL, the `#pragma export` pragma must be employed, or the `export` keyword applied to a class, function or data.

The –fexportall option removes the need for that, making all defined data, classes and functions visible.

The –fexportall option is only meaningful when the –fc370 option are also enabled.

The –fdll=cba and –fdll=nocba options (Enable/disable LE `DLL(CALLBACKANY)` support)

Language Environment always uses FASTLINK DLL linkage (or XPLINK) for C++ modules. However, it is possible to call a function pointer provided by an LE C module, which will not use FASTLINK. In the default case (*–fdll=nocba*), **DCXX** assumes that such function pointers are DLL function pointers and provide both a PRV and an entry point. However, if the C module which provided the function pointer was not itself compiled with *–fdll* then it is necessary to use *–fdll=cba* on the **DCXX** command line. In that case, **DCXX** will generate a call to a run-time helper (`@@FXCLBK`) which will automatically detect whether the function pointer provides a PRV or not.

The –fep=*name* option (specify entry point)

The *–fep=name* option provides a symbol that will be placed on the **END** statement in generated assembly language source. It is used to specify the entry point of a module.

The –fprol=*macro* option (specify alternate prologue macro)

The *–fprol=macro* option specifies an assembly language macro that will be issued at the start of each function in this compilation, instead of the default **DCCPRLG** macro. This option is not valid in combination with the *–fc370* (IBM C++ compatibility) option. The macro is responsible for function startup, stack management, saving registers, etc. The macro is passed these arguments:

ARCH=ZARCH	Added to the prologue parms when *–mlp64* is specified on the compiler command line.
BASER=*n*	The register number used as the code base register for this function. If the value of *n* is 0, then this function does not require a code base register. In this case, the function prologue does not need to set up a base register or worry about code addressibility.
CINDEX=n	The unique function number for this function.
ENTRY=[YES|NO]	Whether an ENTRY statement should be generated for this function.

FRAME=*n* The size of the automatic data required by this function. The prologue macro must allocate this many bytes. Note that if `SAVEAREA=NO` is specified, the function prologue is not required to allocate these bytes. A reasonable size will still be specified when `SAVEAREA=NO` is present for backwards compatibility.

FRAMER=*n* The register number used as the automatic storage area base register for this function. If the value of *n* is 0, then this function does not use any automatic storage, and the functio prologue need not allocate any.

IEEEFP=[YES|NO] Indicates the function was compiled with IEEE floating point or not (HFP.)

LNAMEADDR=*label* Prior to each function, the compiler generates a function name block that contains the "logical" name for the function. This block is a 4-byte length, followed by a `NUL`-terminated string. The compiler passes the *label* for this block to the prologue macro for any use there. The name specified in the function descriptor block is the "C++" name of the function, and does not reflect any application of a `#pragma map` or other compiler-assigned value.

This function name block label is also passed to the `DCCENTR` and `DCCEXIT` macros generated when the –finstrument_functions option is enabled.

SAVEAREA=NO If `SAVEAREA=NO` is specified on the prologue macro, the prologue expansion does not need to create local save area for this function. This indicates that the function is a "leaf" function (it doesn't invoke any other functions) and did not reference any local memory.

Note that the compiler assumes the registers are saved/restored by the prologue and epilogue; but that saving and restoring is typically accomplished in the register save-area of the calling function.

Note that no bytes need to be allocated for the function is `SAVEAREA=NO` is specified, even though the `FRAME=` option may have a value.

The name of the function is provided as the name on the macro invocation. Also note that use of `#pragma prolkey` can add arguments to the macro invocation.

The –fprv=*macro* option (specify alternate PRV address macro)

The *–fprv=macro* option specifies an assembly language macro that will be issued to acquire the base address of the Pseudo-Register Vector (PRV), instead of the default **DCCPRV**. The compiler will specify one argument on the macro:

REG=nn	Specify the register which should contain the address of the PRV at the end of the macro.

The macro will be generated once for each function that needs to reference data in the Pseudo-Register Vector. The compiler will then save the returned address locally in the function's stack frame for future reference.

The –fepil=*macro* option (specify alternate epilogue macro)

The *–fepil=macro* option specifies an assembly language macro that will be issued at the end of each function in this compilation, instead of the default **DCCEPIL** macro. This option is not valid in combination with the *–fc370* (IBM C compatibility) option. The macro is responsible for function termination, stack management and return to the calling function.

The –lnameaddr and –fno_lnameaddr macros (Enable or disable generation of Logical Name Address info)

Normally, the compiler generates a Logical Name Address block for each function. This block of memory contains the logical (C++) name for each function. The address of this memory is passed on the generated prologue macro as the `&LNAMEADDR` parameter.

If *–fnolnameaddr* is specified, the compiler will not generate the Logical Name Address block, and will not provide a `&LNAMEADDR` parameter.

The *–flnameaddr* option is enabled by default.

The –fopts[=*macro*] option (Request interesting options noted at top of generated assembly)

The *–fopts[=macro]* option specifies requests that the compiler invoke the `DCCOPTS` (or other specified *macro*) at the top of the generated assembly language source. Parameters to the macro will describe some of the code-generation options specified on the **DCXX** command line.

The purpose of this macro is to provide a mechanism to direct any macro-generated source based on **DCXX** compiler options. This can be used to alter the expansion of other macros. For example, the prolog macro could expand differently if *–fieee* were specified on the **DCXX** command line. Or, various runtime flags could be set as appropriate.

If *–fieee* is specified on the command line, then `FP=IEEE` will be added to the `DCCOPTS` invocation.

If *–fasciiout* is specified on the command line, then `CHARSET=ASCII` will be added to the `DCCOPTS` invocation.

The *–fopts* macro may not be used if the *–flinux* or *–fc370* options are also used.

The –fendmacro[=*text*] option (Specify text to appear before the END statement)

The *–fendmacro[=text]* option cause the compiler to invoke the `DCCEND` (or other specified *text*) just before the `END` statement in the the generated assembly language source. The compiler will not add any arguments to the invocation of the macro. Thus, the text could be any valid assembly language text.

Any valid assembly language text can be specified.

The *–fendmacro* option may not be used if the *–flinux* or *–fc370* option is also used.

The –finstrument_functions option (Request function beginning /ending instrumentation)

The *–finstrument_functions* option causes the compiler to generate instrumentation code that denotes the start and end of a function.

When the *–flinux* option is enabled, *–finstrument_functions* causes the compiler to generate the appropiate code for use with Linux profiling tools.

When *–flinux* is not specified, the compiler generates references to the `DCCENTR` and `DCCEXIT` macros. `DCCENTR` and `DCCEXIT` are invoked with two parameters `ADDR=`*`reg`* and `LNAMEADDR=`*`label`*. The `ADDR` parm is a register that contains the starting address of the current function. The `LNAMEADDR` parm is a label for a name structure generated by the compiler. The name structure is a 4-byte length, followed by a `NUL`-terminated string containing the function's name. The name will be the "logical" C++ name for the function similar to how appears in the C++ source, and does not reflect any application of `#pragma map` or any other compiler-assigned name.

The compiler saves and restores registers `R0`, `R1`, `R14` and `R15` across invocations of these macros, so they can be used in the macro. Furthermore, when *–finstrument_functions*

is enabled, the compiler guarantees a register save area for the current function will be requested. That is, when *–finstrument_functions* is enabled the `SAVEAREA=NO` parameter will not be specified on the prologue macro; ensuring a register save area will be available in the function.

Although example `DCCENTR` and `DCCEXIT` macros are provided with the Systems/C library, these are essentially empty and will need to be altered for use. The following example assumes the presence of a function named TRACE, which accepts a pointer to the function's entry point in `R1` and a pointer to the name of the function in `R0`.

```
macro
DCCENTR &ADDR=none,&LNAMEADDR=none
L   1,=A(&LNAMEADDR)
LA  0,4(0,1)  R0 points to NUL-terminated name
LR  1,&ADDR   R1 points to function address
L   15,=V(TRACE)  Call "TRACE"
BALR 14,15
B   *+8
LTORG
mend
```

The compiler allocates up to 128 bytes for the expansion of the DCCENTR and DCCEXIT macros. If the macro expansion results in more than 128 bytes, the generated code may encounter addressability errors.

Note that the function instrumentation is not the same as the function prologue and epilogue. Because of the possibility of inlined functions, the instrumentation can actually appear anywhere in the code. The compiler will note the entry and exit from an "inlined" function by generating the instrumentation at the proper location.

The –fcode_base=*N* option (specify register to use for addressing code)

The *–fcode_base=N* option specifies a different base register for executable code. The default base register is **R12**. *N* is an integer, in the range 2 to 13. Registers 0, 1, 14 and 15 may not be specified as the code base register. If register 13 is specified as the code base register, then prologue and epilogue macros must also be specified. The default prologue and epilogue macros assume register 13 is the frame base register. The value specified here becomes the value of the **BASER** argument to the prologue macro.

The –fframe_base=*N* option (specify register to use for addressing automatic data)

The *–fframe_base=N* option specifies a different register to use for addressing automatic data. The default frame base register is **R13**. Automatic data is allocated for each function on a dynamic basis during program execution. *N* is an integer, in the range 2 to 13. That is, one may not specify registers 0, 1. 14 or 15 may not be specified as the frame base register. The default prologue and epilogue macros assume register 13 is the frame base register. Prologue and epilogue macros must be provided if a value other than 13 is specified in the *–fframe_base* option. If register 13 is specified as the *code_base* register, then a different register must be specified as the *frame_base* register.

The –freserve_reg=*N* option (reserve register #*N*)

The *–freserve_reg=N* option instructs the compiler that register #*N* is reserved and should not be used in code generation. The compiler will reserve that register for the entire compilation and not generate code that alters the register. This can be useful for particular inline assembly sequences, or when using a prologue/epilogue sequence that assumes a register remains unaltered throughout execution.

The –fwarn_disable=*N[,N,N-M,...]* option (disable emission of warning(s))

The *–fwarn_disable=N[,N,N-M,...]* option disables emission of the named warning(s). A range of warnings can be specified separated by the hyphen character. More than one warning may be specified, separated by commas or colons. The option may appear multiple times.

A disabled warning may be re-enabled with the *–fwarn_enable* option.

The –fwarn_enable=*N[,N,N-M,...]* option (reenable disabled warning(s))

The *–fwarn_enable=N[,N,N-M,...]* option enables emission of the named warning(s). A range of warnings can be specified separated by the hyphen character. More than one warning may be specified, separated by commas or colons. The option may appear multiple times.

An enabled warning may be disabled with the *–fwarn_disable* option.

The –fwarn_promote=*N[,N,N-M,...]* option (promote warning(s) to error status)

The *–fwarn_promote=N[,N,N-M,...]* option promotes emission of the named warning(s). A range of warnings can be specified separated by the hyphen character. More than one warning may be specified, separated by commas or colons. The option may appear multiple times. Once a warning has been promoted, it remains an error.

The –ftrim option (remove trailing blanks from source)

The *–ftrim* option removes trailing blanks from input source lines. This can be useful on cross-platform hosts if the source has been copied from a mainframe fixed record length data set. When copying such a file to a cross-platform host, the record length is typically preserved, causing extra blanks to be appended. These blanks can cause problems if they occur after a backslash (\). Using *–ftrim* will remove these trailing blanks, allowing the source to be compiled on the cross-platform host as it was on the mainframe.

The –faddh option (add ".h" to #include names)

The *–faddh* option causes the compiler to examine each `#include` name from the source file. If the specified string does not end in "`.h`", a "`.h`" will be added. This option can be useful when moving program source from an OS/390 environment where PDS names sometimes don't include "`.h`".

The –flowerh option (convert #include names to lower case)

The *–flowerh* option causes the compiler to convert characters in `#include` file names to lower case. This conversion is applied before any other modifications are made to the file names. This option can help in a multi-OS environment, where sources are shared between file systems which are case-sensitive (i.e. UNIX) and not case-sensitive (i.e. Windows.) On the case-sensitive system, convert all the file names to lower case in the file system, and use the *–flowerh* option to ensure the compiler uses all lower-case names for `#include` file lookup.

The –fnosearchlocal option (don't look in "local" directories)

The *–fnosearchlocal* option causes the compiler to not examine "local" directories when doing `#include` lookup for file names that start with a double quote.

The –fpreinclude=*filename* option (#include the named file before compiling the C++ source file)

The *–fpreinclude=file* option causes the compiler to behave as if

```
#include "filename"
```

were the first line in the C++ source file. That is, the compiler will look for the named file on the `#include` list. If found, it will be processed before the primary C++ source file.

The –flisting*[=filename]* option (generate a listing)

The *–flisting[=filename]* option will cause the compiler to produce a listing of the compilation. The listing shows such items as the source line, the file name table, C preprocessor expanded lines, and the structure map. If the *–fshowinc* option is enabled, source lines which originate in `#include`d files will be included in the listing. Otherwise, only the source from the primary file will be listed.

A filename for the listing may be optionally specified. If no filename is specified, the listing is written to `stdout`.

This option can be disabled with a subsequent *–fno_listing* option.

The –fvtable_listing and –fno_vtable_listing options ((enable/disable virtual function table information to listing)

Classes that contain virtual functions define a virtual function table, which is a hidden compiler-generated variable initialized with a list of function pointers. At times, it is convenient to know the layout of this table, especially to determine if the layout may have changed as a result of maintenance. The *–fvtable_listing* option instructs the compiler to emit a listing of the members in the virtual function table after each class's description in the structure map section of the listing. The default is *–fno_vtable_listing.*

It is important to note that the virtual function table listing does not include base class tables, even though the complete class will depend on these as well. So to test if two builds are truly compatible, base class virtual function tables must be checked independently.

The –fpagesize=*n* option (set the listing page size to *n* lines)

By default, the number of lines listed on each page of the listing is sixty (60) lines per page. The *–fpagesize=n* option can reduce or increase that as needed. The value of *n* should not be less than twenty (20).

The –fshowinc and –fno_showinc options (enable/disable inc luding source from #include files in listing)

If a listing of the compilation is requested, the *–fshowinc* option may be used to request that source lines from `#include` files be included in the listing.

–fshowinc is the default.

–fno_showinc can be used to reduce the size of the listing file by not including source from `#include` files.

The –fstructmap and –fno_structmap options (enable/disable including struct layout information in the listing)

If a listing of the compilation is requested, the *–fstructmap* option may be used to request that a "structure map" appear at the end of the listing. This structure map will contain information regarding the layout of the structures defined in the program source, including field offsets and lengths.

–fstructmap is the default.

–fno_structmap can be used to reduce the size of the listing file by not producing the structure map.

The –fstructmaphex and –fno_structmaphex options (structure layout information should/shouldn't be displayed in hex)

If the *–fstructmap* option is in effect, –fstructmaphex will cause the offsets to be displayed using hexadecimal values instead of decimal ones. *–fno_structmaphex* indicates the values should be displayed in decimal.

The –frent option (generate re-entrant code)

The *–frent* option instructs the compiler to generate re-entrant code. When *–frent* is enabled, file-scoped external and static variables will be re-entrant by default. The *–frent* option is enabled by default.

The –fmaxerrcount=*N* option (limit the number of reported errors)

The *–fmaxerrcount=N* option places a limit on the number of errors the compiler will report. When the specified number of errors have been encountered, compilation stops.

The –U*name* option (undefine predefined #define values)

DCXX predefines the following values as well as the standard ANSI ones:

Macro Name	Replacement Value
__COUNTER__	A unique value at each reference, begining with 0
__SYSC__	1
__SYSC_VER__	Compiler version number
__SYSC_ASCIIOUT__	Defined if *–fasciiout* enabled
__SYSC_ANSI_BITFIELD_PACKING__	Defined if *–fansi_bitfield_packing* enabled
__BFP__	Defined if *–fieee* is enabled
__I390__	1
__370__	1
__SYSC_LP64__	Defined if *–mlp64* enabled
_LP64	Defined if *–mlp64* enabled
__SYSC_ILP32__	Defined if *–milp32* enabled
_ILP32	Defined if *–milp32* enabled
__ptr31__	Defined to be __ptr32, equivalent to __ptr31
__PTR31	Defined to 1
__PTR32	Defined if __PTR31 is defined

The *–U* option can selectively remove these definitions, or any others that were added via the command line.

The –fincstripdir option (remove directory components from #include names)

The *–fincstripdir* option will cause the compiler to remove any directory components from a `#include` file name before any other processing occurs. This option is useful for compiling source with Systems/C and other compilers which act similarly. For example, if the source contains:

```
#include <sys/parm.h>
```

and the *–fincstripdir* option is enabled, the result would be same as if the source contained

```
#include <parm.h>
```

The –fincstripsuf option (conditionally remove suffixes from #include names)

The *–fincstripsuf* option causes the compiler to retry failed open attempts for `#include` files. As the compiler is searching for a `#include` file, it will first try to open the file with the given suffix. If *–fincstripsuf* is specified, the compiler will then remove any suffix and try again to open that file. This option is helpful on OS/390 and z/OS when moving from other C compilers to Systems/C.

The –fincrepsuf option (conditionally replace suffixes from #include names)

The *–fincrepsuf* option is similar to the *–fincstripsuf* option in that it causes the compiler to first try to locate `#include` files using the given suffix. If this attempt fails, it is replaced with "`.h`", as if *–faddh* were specified.

The –fmargins*[=m,n]* option (specify margins for source lines)

The *–fmargins* option specifies columns from the input file which are examined for input to the compiler. The compiler ignores text that does not fall in the specified range.

The *–fnomargins* option is the default option, and specifies that each entire source line is to be considered as input.

The *–fmargins* option, with no arguments is equivalent to *–fmargins=1,72*.

The *–fmargins=m,n* form of the option specifies the starting and ending column to be considered as input. *m* must be greater than `0` and less than `32761`. If *,n* is specified, *n* must be greater than *m* and less than `32761`. If *,n* is not specified, the compiler uses the remainder of the input line.

–fmargins can be useful when copying source from a mainframe environment where sequence numbers are preserved in the input lines.

–fmargins implies *–ftrim*.

The –fmesg=*style* option (specify message style)

The *–fmesg=style* option is used to indicate which style of message format the compiler should employ. Currently, three message styles are supported, `microsoft`, `sysc` and `ext`.

If the `microsoft` style is specified, as in *–fmesg=microsoft*, the messages produced by the compiler will look similar to those produced by the Microsoft VC++ compiler and are suitable for using with Microsoft's DevStudio integrated development environment. This is the default style on Windows hosts.

If the `sysc` style is specified, the message format will be the Systems/C message format. This is the default format on UNIX, OS/390 and z/OS hosts.

If the `ext` style is specified, messages will be in an extended format, and will include extra information regarding the error or warning.

The –fasciiout option (char and string constants are ASCII)

Normally, the character set employed for character and string constants is EBCDIC. Specifying the *–fasciiout* option causes the compiler to use ASCII values for character and string constants. Note that the Systems/C and Systems/C++ libraries don't support ASCII values for character-specific functions. Also, the *–fasciiout* option does not affect character or string constants specified in the C++ preprocessor or `#pragma` statements.

If *–fasciiout* is specified, the C++ preprocessor will predefine the `__SYSC_ASCIIOUT__` macro to the value `1`. Otherwise, `__SYSC_ASCIIOUT__` will not be defined.

–fasciiout is enabled by default when *–flinux* is specified.

If *–fnoasciiout* is present after *–flinux* on the commandline then **DCXX** will generate EBCDIC string constants on Linux.

The –fdollar option (alloc dollar sign character in identifier s)

According to standard C++, the dollar sign character (`$`) is not allowed in C++ identifiers or preprocessor macro identifiers. When the *–fdollar* option is specified **DCXX** will allow the dollar sign character in identifiers and macros.

Use *–fno_dollar* to disable this option.

The –fwchar_ucs option (indicate that wide character constants are UCS-2 or UCS-4.)

The *–fwchar_ucs* option indicates that wide character string and character constants are to be generated in the UCS (Universal Character Set) encoding rather than the target ASCII or EBCDIC encoding.

Wide character char/string literals are specified with the `L'x'` and `L"xxx"` prefixes. Unicode literals can be defined – regardless of the setting of *–fwchar_ucs* – using

`u'x'` (UCS-2), `U'x'` (UCS-4), `u8"xxx"` (UTF-8), `u"xxx"` (UTF-16), and `U"xxx"` (UCS-4).

The UCS-2 character set is used when the *–fwchar=2* option is specified, UCS-4 will be used when *–fwchar=4* is specified.

–fwchar_ucs is enabled by default when the *–fztpf* option is specified. On the z/TPF platform, normal character strings are EBCDIC, but wide character strings are UCS-4.

–fwchar_ucs can be disabled using the *–fnowchar_ucs* option.

The –fwchar=*n* option (specify the size of wchar_t)

The *–fwchar=n* option specifies the size, in bytes, of the wide character type, `wchar_t`. By default, the size of `wchar_t` is assumed to be 4 bytes. Allowed values for *n* are 2 and 4 (for `unsigned short` or `unsigned long` declarations of `wchar_t`.)

The Systems/C library uses a size of 4 for `wchar_t`. If another size is selected, the wide character related functions in the Systems/C library may not operate correctly.

The –fsname=*name* option (specify section names)

Each compilation requires section names for the various code and sections the compiler will produce. These names must be unique for the load module in which the generated object will participate. By default, the various section names are taken from the source file name; which can produce duplicate section names in some circumstances.

The *–fsname=name* option is used to specify what the section name should be, allowing for the unique specification of section names and avoiding duplicates. *Name* must begin with an alphabetic letter. *Name* must be 1023 characters or less.

If the specified *name* is too long, the compiler will truncate it.

The compiler generates both upper-and-lower case versions of the name for various CSECTs, so the name should not be considered case-specific.

The *–fsname* option is ignored for linux and z/TPF compilations.

The –fnosname option (allow PLINK to choose unique section names)

When the *–fnosname* option is specified, the compiler produces assembler source that uses names **PLINK** later recognizes at pre-link time. In this case, **PLINK** maps these names to a name that is unique for the load module. In this way,

individual compilations need not be concerned over the choice of section name. **PLINK** guarantees this compilation will have a unique name in the resulting load module.

Using *–fnosname* requires the use of **PLINK** before final linking of the load module to properly map the various section names.

–fnosname is enabled by default in Systems/C++. If the C++ compiler is being used without **PLINK** and CSECT names should be automatically generated based on the filename then *–fsname* (giving no section name) should be specified.

The –fsnameprefix=*char* option (specify section name prefix)

When section names are generated, a prefix character is added. The default prefix character is "`@`", so that the code `CSECT` for a source named "`test.c`" will be "`@TEST`". Using the *–fsnameprefix=char* option, you can specify an alternative prefix character. If no character is provided (i.e., *–fsnameprefix=*) then the section names are generated without a prefix.

The –fieee option (BFP format floating point values and constants)

The *–fieee* option instructs the compiler to use the newer Binary Floating Point format for floating point constants and use the new BFP-related instructions for floating point arithmetic calculations. Binary Floating Point format is equivalent to the IEEE floating point format used in many other hardware implementations.

When the *–fieee* option is enabled, **DCXX** will convert floating point constant values into their IEEE format for emission in the generated assembler source. Also, **DCXX** will use IEEE arithmetic operations for any floating point operations performed by the compiler. Lastly, **DCXX** will generate the BFP instructions for any arithmetic performed at run-time.

If *–fieee* is enabled, **DCXX** will define the macro `__BFP__` to `"1"`. C programs may test for the use of IEEE instructions and constants by testing for the `__BFP__` macro.

The floating point format can be explicitly specified on a per-variable basis using the type modifiers `_Ieee` and `_Hexadec`. For example, the type `_Ieee double` will hold IEEE values regardless of the setting of *–fieee*. Be aware that intermediate operations will generally be performed using the floating point format specified on the command line, regardless of the type of variable the value will ultimately be stored in. So there will be a potential for rounding error and poor performance from excessive conversions when using a different format than the one specified on the command line.

The Systems/C and Systems/C++ libraries have been designed (since version 2.10) to support IEEE and HFP floating point formats at the same time. Most C++ functions are defined as two variants, one that uses `_Ieee float`, and another that uses

`_Hexadec float`, and standard C++ overload resolution selects the right function. For C library functions, the C library provides a wrapper function that detects if the calling function was compiled with *–fieee* or not and calls the appropriate variant based on that determination.

The *–flinux* option enables *–fieee* by default.

The –fmrc/–fnomrc options (mainframe or UNIX-style return codes)

The *–fmrc* and *–fnomrc* options alter the return code returned by **DCXX**.

Normally, on cross-platform (UNIX and Windows) hosts, **DCXX** returns a typical UNIX-style return code, 0 for success or warnings, 1 for errors. And, on OS/390 and z/OS, **DCXX** returns a mainframe-style return code, 0 for no warnings, 4 for warnings, 8 for errors and 12 for catastrophic failure.

These defaults can be altered by using the *–fmrc* and *–fnomrc* options. When *–fmrc* is enabled, **DCXX** will return mainframe-style return codes; allowing for the use of mainframe-style return codes on a cross-platform host. When *–fnomrc* is enabled, **DCXX** will return UNIX-style return codes, allowing for the use of UNIX-style return codes on OS/390 or z/OS.

The –fpatch/–fno_patch options (generate a patch area)

The *–fpatch* and *–fno_patch* options control the generation of a per-compilation patch area. If *–fpatch* is enabled, the compiler will generate a patch area named `@@PATCH_AREA`, which appears at the end of the `CODE` section. Each 4K region of text in the generated assembler code will contain an A-CON reference to the patch area, so it can be readily addressed. Typically, it will appear with other constant definitions, and will look similar to:

```
DC    A(@@PATCH_AREA)
```

The size of the generated patch area is determined by computing a percentage of the size of the generated code, with a minimum size of 32 bytes and a maximum size of 4096 bytes. The default percentage is 10%, but can be altered by the *–fpatchmul* option.

The –fpatchmul=*n* option (alter the size of the patch area)

The *–fpatchmul=n* option changes the percentage multiplier used in the computation of the size of a generated patch area. The size of the generated patch area is computed as a percentage of the size of the generated code. The default percentage

is 10%. To increase the size of the generated patch area, increase the *–fpatchmul* value, to decrease it, decrease the *–fpatchmul* value. Note that the minimum size for a patch area is 32 bytes, and the maximum is 4096 bytes. The *–fpatchmul=n* option implies the *–fpatch* option.

The –flinux option (enable Linux/390 and z/Linux code generation)

The *–flinux* option instructs **DCXX** to generate assembler source suitable for use on Linux/390 or z/Linux. The assembler source will be generated and formatted to be assembled by the Linux/390 or z/Linux assembler, *as*. Furthermore, some HLASM specific features and related options will be disabled and may produce warnings if used.

This option operates on any host supported by Systems/C++, thus, it is possible to generate Linux/390 assembler source on any supported platform.

The *–flinux* option implies the *–fieee* option. On Linux/390 and z/Linux floating point values and constants are in BFP (IEEE) format.

The *–flinux* option implies the *–fasciiout* option. On Linux, character values are in ASCII.

If the *–mlp64* option is enabled, the generate assembler source is intended for use on z/Linux, and should be assembled with the z/Linux *as* assembler.

The –fvisibility=*setting* option (set ELF object symbol visibility)

When generating code for either Linux, z/Linux or z/TPF, the compiler produces assembly source to be assembled with the GNU GAS assembler. That assembler, in turn, produces `ELF` object files.

An `ELF` object file contains symbols that have a visibility attribute. This attribute controls the visibility of the symbols during linking. For example, a symbol can be "hidden", which means that it is internal to the object and can't be referenced during linking.

There are four valid values for the visibility, `default`, `internal`, `hidden` and `protected`.

This feature should be employed for building shared objects, to manage the symbols exported by the shared objects, avoiding symbol clashes.

Unless otherwise specified in the source, the value of the *–fvisibility* setting applies to all the symbols in a compilation. The `__attribute__((visibility ("setting")))` attribute can be used to specifically set a symbol's visibility.

The `default` visibility indicates that the symbol is visible to other modules.

The `hidden` visibility indicates the symbol is "hidden" within a shared object. Two symbols of the same name with "hidden" visibility refer to the same data if they are linked into the same shared object.

The `internal` visibility is similar to `hidden`, but in some ELF environments can have other special meaning, as afforded by the hardware processor. `internal` also indicates that a function can never be invoked from "outside" a shared object, which allows the compiler some flexibility in optimizations.

The `protected` visibility indicates that references to a symbol will only be resolved within the defining module. The declared symbol cannot be overridden by a same-named symbol in another module.

The –fsigned_char/–funsigned_char options (Control if `char` is signed or unsigned by default)

The *–fsigned_char* option instructs the compiler to treat the `char` data type as signed (range −128 to 127) unless the keyword `unsigned` is explicitly specified. The *–funsigned_char* option instructs the compiler to treat the `char` data type as unsigned (range 0 to 255) unless the keyword `signed` is explicitly specified. The default is *–funsigned_char*.

The –fsuppress_vtbl option (suppress generation of C++ vtable information)

In most situations there is a heuristic for deciding which compilation unit provides the virtual function table for a class to avoid duplicate definitions. However, in the case of a class with no virtual functions which inherits from a base class with virtual functions, this heuristic breaks down.

By default the compiler generates the virtual function table but marks it as private to this compilation unit so there are no conflicts. The *–fsuppress_vtbl* option instructs the compiler to not generate virtual function table information at all in this compilation unit for classes where it is unclear which compilation unit should provide the virtual function table.

The –fforce_vtbl option (force generation of C++ vtable information)

The *–fforce_vtbl* option instructs the compiler to generate (and export as external data) virtual function table information for all defined classes where it is unclear which compilation unit should provide the virtual function table.

The –finstantiate=*mode* option (set the template instantiation mode)

The *–finstantiate=mode* option selects which heuristic will be used by the compiler to decide which templates should be instantiated in this compilation unit. See the section Template Instantiation (page 20) for a detailed discussion.

The –fdistinct_template_signatures/–fno_distinct_template_signatures options (enable/disable distinct template signatures)

The *–fdistinct_template_signatures* option is the default in most modes and instructs the compiler to generate unique symbols for functions generated through template instantiation. *–fno_distinct_template_signatures* causes the compiler to treat a template instantiation as though it were a normal function and is in effect by default if the *–fc370* option is specified.

For example, consider the following two functions:

```
void func(int x) {
   . . .
}

template <class A>
void func(A x) {
   . . .
}
```

If *–fno_distinct_template_signatures* is in effect then the non-template version can serve as a specialization of the template `func<int>` even if the compiler does not know that `func` is a template, otherwise `func<int>` is distinct from the non-template version.

The –fimplicit_include/–fno_implicit_include options (enable/disable implicit inclusion for template libraries)

Some template libraries are distributed as a collection of `.h` files that contain the template declarations and `.c` files that contain the template definitions, with no explicit connection between the two files other than their filenames. That is, there is no `#include "foo.c"` directive within `foo.h`. The compiler is then responsible for finding any needed template definitions.

When *–fimplicit_include* is specified, any time a definition for a template that was declared in a `.h` file is requested, **DCXX** will search for a file in the same directory with the same name, but an extension of `.c` instead. Then that file will be processed using the regular template instantiation technique, as if it had been `#included`.

The –ftempinc*[=directory]*/–fno_tempinc options (enable/disable template instantiation method

–ftempinc is an alternative to *–fimplicit_include* for some template libraries. It honors roughly the same language-level semantics but has a different approach to template instantiation.

Instead of processing the `.c` file as if it had been `#included`, *–ftempinc* will cause a file to be created in the tempinc directory that has the same name as the `.h` file, but the extension of `.C`, containing C++-language source that explicitly references the `.h` and `.c` files to instantiate the templates. The tempinc directory can be specified with *–ftempinc=directory* or it will default to "`tempinc`".

After the compiler has run on all of the source files within your project, you will need to run the compiler on each of the generated source files within the tempinc directory. This second pass generates the definitions for templates which were requested in your project's source.

During this second pass, additional template dependencies may be detected. If you compile the second pass with *–ftempinc* as well, then you must run **DCXX** again on any tempinc file that changes during the second pass. Or you can run the second pass with *–fimplicit_include* instead, and it will use regular template instantiation methods within the tempinc pass, which may result in some duplicate definitions.

The –fnonstd_qualifier_deduction/–fno_nonstd_qualifier_deduction options (enable/disable non-standard qualifier deduction)

The *–fnonstd_qualifier_deduction* and *–fno_nonstd_qualifier_deduction* options control whether nonstandard template argument deduction should be performed in the qualifier portion of a qualified name.

The –guiding_decls/–fno_guiding_decls ioptions (Imply template instantation with a specific declaration)

The *–fguiding_decls* option indicates that the compiler should support "guiding declarations" of template functions. A guiding declaration is the declaration of a function that matches a specific instance of a template and no function definition is provided. Such a declaration indicates that the compiler should generate the template function within the current compilation unit.

For example:

```
template <class T> void func(T)  ...
 ...
void func(int);
```

The declaration of `void func(int);` indicates that the compiler should instantiate the template function `func` with an `int` argument within the current compilation unit.

If *–fno_guiding_decls* is enabled, then the function declaration is considered a separate function not associated with the template which would require its own function definition.

The –fexceptions/–fno_exceptions options (enable/disable support for C++ exceptions)

The C++ language provides features for passing exceptions in a program to bypass the regular flow of execution for certain events. Most importantly are the `try`, `catch` and `throw` keywords. The *–fexceptions* and *–fno_exceptions* options control the compiler's support for these features. When *–fno_exceptions* is specified these features are disabled and, if used, will generate an error message.

The –frtti/–fno_rtti options (enable/disable C++ Run-Time Type Info)

C++ provides Run-Time Type Information through the keyword `typeid`, which evaluates to a `typeinfo` class which can be used to determine certain information about the class that may change at run-time. Also provided is the `dynamic_cast` operator which implements run-time casts from a base class to a derived class with type checking. This feature is useful for polymorphic classes where it is necessary to recover the type of the derived class from a pointer to a base class.

The *–frtti* and *–fno_rtti* options control support for C++ Run-Time Type Information. When *–fno_rtti* is specified, use of the `typeid` keyword is an error.

The –farray_new_and_delete/–fno_array_new_and_delete options (enable/disable `new[]` and `delete[]`)

Ordinarily when an array `new` (i.e., `new int[10]`) is processed by the compiler a call to `void *operator new[](size_t)` is generated. Similarly, a call to `void operator delete[](void *)` is generated for array `delete` (i.e., `delete[]`) expressions. *–fno_array_new_and_delete* can be specified to instruct the compiler that there are no separate `new[]` or `delete[]` functions. The source code may still contain array `new` and `delete` expressions but the compiler will generate calls to the regular `void *operator new(size_t)` and `void operator delete(void *)` functions rather than specialized array versions.

The –fexplicit/–fno_explicit options (enable/disable `explicit` keyword)

The *–fexplicit* and *–fno_explicit* options control support for the C++ `explicit` keyword. The `explicit` keyword is applied to a constructor to mark that it may not be implicitly invoked. For example, if `class A` has an `explicit` constructor that takes an integer as its only argument then `A x(123);` will generate an explicit constructor call, but `A x = 123;` will be considered implicit and disallowed. *–fno_explicit* causes the compiler to no longer recognize the `explicit` keyword for compatibility with older compilers that do not support it.

The –fnamespaces/–fno_namespaces options (enable/disable namespace support)

C++ provides a namespace feature to facilitate the separation of a program into logical sections (namespaces) which can use the same name to refer to separate constructs. The keyword `namespace` defines a new namespace and the keyword `using` instructs the compiler to use a certain namespace for symbol lookup. The *–fno_namespaces* option disables this feature. See the section Namespace Support (page 18) for more information.

The –fold_for_init option (variables in `for()` inits follow pre-ANSI semantics)

When the *–fold_for_init* option is in effect any variables declared in the initialization of a `for` statement will be treated as if they occurred in the scope surrounding the `for` statement. For example,

```
{ . . .
  for (int i = 0; i < 10; i++)
      . . .
}
```

is equivalent to

```
{ int i;
  . . .
  for (i = 0; i < 10; i++)
      . . .
}
```

The –fnew_for_init option (variables in `for()` inits follow ANSI semantics)

The *–fnew_for_init* option instructs the compiler to treat any variables declared in the initialization of a `for` statement to be put in a scope specific to the `for` statement (as per the ANSI standard). The example from *–fold_for_init* would then be equivalent to:

```
{ . . .
   {
      int i;
      for (i = 0; i < 10; i++)
         . . .
   }
}
```

The –fold_specializations/–fno_old_specializations options (enable/disable old-style template specializations)

Older C++ compilers would accept specializations of a template function without the `template<>` syntax. The *–fold_specializations* option instructs the compiler to accept these definitions as template specializations. The *–fno_old_specializations* option instructs the compiler to treat them as regular functions distinct from the template rather than template specializations (in which case they may be unreachable).

An old-style template specialization takes the following form:

```
template <class A>
void func(A x) { ... }

void func(int x) { ... }
```

The new template specialization syntax takes the following form:

```
template <class A>
void func(A x) { ... }

template<>
void func(int x) { ... }
```

The –fextern_inline/–fno_extern_inline options (enable/disable `extern inline`)

The `extern inline` combination on a function tells the compiler to generate an externally-visible symbol for the function and also to try to `inline` it where appropriate. In order for a function to be inlined it has to be defined in every compilation unit so typically it is implicitly `static` to avoid any duplicate symbols. `extern inline` instructs the compiler to attempt to generate the externally-visible symbols even if there may be duplicate symbols. If *–flinux* is specified then the linker is instructed to discard all but one of the definitions. On other platforms multiple definitions and/or linker warnings may result.

`extern inline` is obviated by the modern practice of weak linking (available on all supported platforms now in some way or another). The compiler generally produces an externally-visible symbol for an inline function, but marks that symbol as a "weak definition." In this way the linker is exposed to all of the multiple definitions and it can remove the redundant ones, so in the end all non-inlined calls will point to the same function, without the use of explicit extern inlining.

Also note that this option has no effect if `extern inline` does not explicitly occur in the source code. The default behavior is always applied to `inline` functions that are not also marked `extern`.

If the *–fno_extern_inline* option is specified then the compiler will generate an error message if `extern inline` is used.

The –fshort_lifetime_temps/–flong_lifetime_temps options (enable/disable long lifetime temporaries)

In cfront and older C++ compilers temporaries would not be destroyed until the next end of scope. The standard, however, specifies that temporaries be destroyed at the end of the full expression. *–flong_lifetime_temps* instructs the compiler to treat temporaries in a fashion compatible with cfront, whereas *–fshort_lifetime_temps* instructs the compiler to treat temporaries according to the standard.

The –fbool/–fno_bool options (enable/disable `bool` support)

The *–fbool* and *–fno_bool* options control compiler support for the `bool` keyword. `bool` specifies a data type that can hold just two values, `true`, and `false`. C++ code written before the `bool` keyword was available often included the code `typedef enum { false, true } bool;`, which conflicts with the compiler's keyword.

The –fwchar_t_keyword/–fno_wchar_t_keyword options (enable/disable `wchar_t` support)

The *–fwchar_t_keyword* and *–fno_wchar_t_keyword* options control compiler support for the `wchar_t` keyword. `wchar_t` specifies a data type that can hold characters that are several bytes wide (depending upon the *–fwchar=n* setting). There exists code which provides its own definition of `wchar_t`, similar to `typedef unsigned short wchar_t;`, which may require *–fno_wchar_t_keyword* to compile without changes.

The –ftypename/–fno_typename options (enable/disable `typename` support)

The *–ftypename* and *–fno_typename* options control compiler support for the `typename` keyword. The C++ standard provides the `typename` keyword to help make it clear which template parameters are used as types.

The –fimplicit_typename/–fno_implicit_typename options (enable/disable implicit template param type determination)

The *–fimplicit_typename* and *–fno_implicit_typename* options control whether the compiler will attempt to determine, from context, if a template parameter is a type or nontype. This may be necessary for older code which does not use the `typename` keyword explicitly to mark type template parameters.

The –fdep_name/–fno_dep_name options (enable/disable dependent name processing)

When *–fdep_name* is specified, dependent name processing is enabled and the compiler will perform a separate lookup of names in templates when the template is instantiated and when it is parsed (if *–fparse_templates* is specified). For details see the section Dependent Name Processing (page 18.)

The –fparse_templates/–fno_parse_templates options (enable/disable parsing templates)

The *–fparse_templates* and *–fno_parse_templates* options control the parsing of templates in their generic form (i.e., even if they are not instantiated). If *–fparse_templates* is specified then the templates will be parsed even if they are never instantiated, otherwise the function will be ignored until instantiated. This can have an effect on dependent name processing and error reporting.

The –fspecial_subscript_cost/–fno_special_subscript_cost options (enable/disable special operator costs)

When *–fspecial_subscript_cost* is specified the compiler will preferentially use an overloaded `operator[]` over an `operator*` for array accesses.

The –falternative_tokens/–fno_alternative_tokens options (enable/disable C++ alternative tokens)

The *–falternative_tokens* and *–fno_alternative_tokens* options control support for C++ alternative tokens. These include digraphs such as `:>` and operator keywords such as `and`, `bitand`, etc.

The –fenum_overloading/–fno_enum_overloading options (enable/disable enum overloading)

The *–fenum_overloading* and *–fno_enum_overloading* options control support for overloading operations on enum-type operands. If enum overloading is disabled then the compiler will always use generic built-in operators for operations on enums.

The –fconst_string_literals/–fno_const_string_literals options (enable/disable `const` strings)

If *–fconst_string_literals* is specified then the compiler will treat string literals as `const` by default. If *–fno_const_string_literals* is specified then the compiler will treat string literals as non-`const` (mutable) by default.

The –fimplicit_extern_c_type_conversion/–fno_implicit_extern_c_type_conversion options (Allow implicit conversions between C and C++ functions)

In many implementations, the function linkage for C++ functions is different than `extern "C"` functions. Because of this, conversions between those two functions would not be allowed.

On platforms where C++ and `extern "C"` linkage is the same, the compiler can implicitly convert between the two function type.

The *–fimplicit_extern_c_type_conversion* option enables this conversion, and is set for z/TPF, Linux and Systems/C++ runtime environments.

In the LE runtime environment (IBM compatibility), the linkages are not the same, so the *–fno_implicit_extern_c_type_conversion* option is set.

The –fclass_name_injection/–fno_class_name_injection options (enable/disable class name injection)

The *–fclass_name_injection* and *–fno_class_name_injection* options control whether the class's name will be injected into its own scope. Older implementations of C++ did not inject the class name into its own scope.

The –farg_dependent_lookup/–fno_arg_dependent_lookup options (enable/disable arg-dependent lookup)

The *–farg_dependent_lookup* and *–fno_arg_dependent_lookup* options control whether argument-dependent lookup is used for unqualified names in function calls of overloaded functions. For more information see the section Argument Dependent Lookup (page 19.)

The –ffriend_injection/–fno_friend_injection options (enable/disable friend namespace injection)

If *–ffriend_injection* is specified then a function or class declared only in friend declarations is visible. If *–fno_friend_injection* is specified then a function or class declared only in friend declarations is not visible.

The –flate_tiebreaker option (avoid the use of qualifiers in overload resolution)

The *–flate_tiebreaker* option instructs the compiler to ignore `const` and/or `volatile` qualifiers during overload resolution unless more than one candidate is found and the qualifiers are the only means of distinguishing.

The –fearly_tiebreaker option (use qualifiers in overload resolution)

The *–fearly_tiebreaker* option instructs the compiler to use `const` and/or `volatile` qualifiers during overload resolution, causing an overloaded function to match only if the qualifiers match. This is the standard behavior.

The –fnonstd_using_decl/–fno_nonstd_using_decl options (enable/disable non-standard using)

The *–fnonstd_using_decl* and *–fno_nonstd_using_decl* options control whether the compiler will accept a nonmember using-declaration that specifies an unqualified name.

The –fvariadic_macros/–fno_variadic_macros options (enable/disable C99 variadic macros)

The *–fvariadic_macros* and *–fno_variadic_macros* options control support for C99-style variadic macros. For example,

```
#define printf(...) fprintf(new_output, __VA_ARGS__)
```

The –fextended_variadic_macros/–fno_extended_variadic_macros options (enable/disable GCC variadic macros)

The *–fextended_variadic_macros* and *–fno_extended_variadic_macros* options control support for special GCC extensions to variadic macros. GCC accepts "`args...`"

to specify that `args` is the name of the variadic argument, rather than `__VA_ARGS__`. GCC also accepts an empty variadic macro argument (the standard requires at least one element in its list). In addition, they have an extension to the paste operator (`##`) if it occurs between a comma and a variadic argument, then the comma will be elided if the variadic argument is empty. So the macro in the following example will emit proper syntax even if called with only one argument:

```
#define FOO(x, ...) bar(x, ## __VA_ARGS__)
```

Note that GCC variadic macros are enabled by default if *–flinux* or *–fztpf* is specified.

The –fbase_assign_op_is_default/–fno_base_assign_op_is_default options (enable/disable copy assignment from base)

The *–fbase_assign_op_is_default* and *–fno_base_assign_op_is_default* options control whether the compiler will accept a copy constructor which takes a base class as its argument as the default copy assignment operator for the derived class.

The –fignore_std/–fno_ignore_std options (enable/disable `std` namespace special treatment)

The *–fignore_std* and *–fno_ignore_std* options control whether the `std` namespace will be ignored (treated as a synonym for the global namespace).

The –version option (print the compiler version number on STDOUT and exit)

The *–version* option causes the compiler print the version number on the STDOUT output stream and then exit. When *–version* is encountered, no other processing occurs.

The –famode=val option (specify runtime addressing mode)

The *–famode* option is used to indicate to the compiler what the runtime addressing mode (AMODE) environment will be. Valid options for *val* are `24`, `31`, `any` and `64`.

This option is most meaningful when *–mlp64* is also specified. When *–mlp64* is specified, by default, the compiler generates code which assumes the runtime AMODE will be 64. Thus, the compiler can safely employ the LOAD-ADDRESS (`LA`) instruction to evaluate pointer arithmetic.

If *–famode* is set to anything other than `64`, the compiler will not use LOAD-ADDRESS for pointer arithmetic when *–mlp64* is enabled. This allows the compiler to generate z/Architecture code which can be executed in any runtime environment.

Also - when *–mlp64* is specified for Systems/C++ compiles, the compiler decorates the prologue macro for the `main()` function to indicate to the Systems/C runtime library that the program should run in an AMODE=64 environment. If *–famode* specifies an *val* other than `64`, the compiler will not indicate that the program should be run in an AMODE=64 environment.

When *–mlp64* is enabled, the `__SYSC_LP64` preprocessing symbol will be defined.

The –march=z*N* option (enable z/Architecture compilation)

The *–march=zN* allows the compiler to generate code that employs instructions available on the edition *N* of the z/Architecture hardware architecture.

The *–march=z0* option is implied when *–mlp64* or *–fztpf* is specified.

However, for situations where *–milp32* is specified, this option allows the compiler to take advantage of the architecture improvements provided in the z/Architecture specifications for 32-bit programs. These include all of the improvements made available in ESA/390 architectures as well as those provided in the specified z/Archtecture definition.

The *–march=zN* option should not be specified if your program is intended to operate on older (pre-z/Architecture) hardware.

For given *–march=zN* settings, the following table shows which facilities will be enabled:

z0	–msquare-root
	–mhfp-extensions
	–mfp-support-extension
	–mfp16
z3	–mhfp-multiply-add
	–mlong-displacement
z5	–mextended-immediate
z6	–mdecimal-floating-point-facility
	–mpfpo-facility
	–mfloating-point-support-sign-handling-facility
	–mfpr-gr-transfer-facility
z7	–mgeneral-instructions-extension
z9	–mload-store-on-condition
	–mdistinct-operands
	–mhigh-word-facility
	–mfp-extensions

z10	–mmisc-instruction-extensions-facility-1 –mtransaction-facility
z11	–mdecimal-floating-point-packed-conversion-facility
z12	–mmisc-instruction-extensions-facility-2
z13	–mmisc-instruction-extensions-facility-3

The –march=esa390 and –march=esa390z options (enable ESA/390 compilation)

The *–march=esa390* option allows the compiler to generate code that employs instructions available on ESA/390 architectures.

If no other *–march* option is specified, the compiler generates code suitable for a 370-class machine.

When the *–march=esa390* option is specified, the compiler will generate code that makes use of the immediate operand instructions and the string-assist instructions. It will also assume there are 16 floating-pt registers available.

The *–march=esa390z* option enables supprot of "ESA/390 mode under z/Architecture" instructions. These instructions were added to the ESA/390 specification when operating in "ESA/390 mode" on z/Architecture hardware. This includes support for the MULTIPLY LOGICAL, DIVIDE LOGICAL, ADD LOGICAL WITH CARRY and SUBTRACT LOGICAL WITH CARRY as well as other instructions.

Depending on your runtime architecture environment, specifying *–march=esa390* may allow your programs to execute faster.

The *–march=esa390* option should not be specified if your program is intended to operate on older (pre-ESA/390) hardware.

The –milp32 option (32-bit compilation)

When *–milp32* is specified, the compiler treats `int`, `long` and pointer data types as 32-bit data types, the ILP32 compilation model.

This is the default, and is historically the compilation model used in mainframe environemnts.

The –mlp64 option (64-bit compilation)

When *–mlp64* is specified, the compiler treats `long` and pointer data types as 64-bit data types, the LP64 compilation model.

The *–mlp64* option implies the *–march=z* option.

For the Systems/C prologue macro, the compiler will add the `ARCH=ZARCH` option to the prologue macro invocation, indicating the generate prologue and epilogue should assume z/Architecture instructions and 64-bit values. If the `main()` function is compiled with the *–mlp64* option enabled, and no other *–famode* is specified, the Systems/C runtime environment will enable a 64-bit AMODE.

The code generated when *–mlp64* is specified can be controlled using the *–famode* option. If *–famode=any*, *–famode=31* or *–famode=24* is specified along with *–mlp64*, the compiler will not use the LOAD-ADDRESS (`LA`) instruction for pointer arithmetic. The `LA` instruction is dependent on the AMODE at runtime, and thus can't be used to perform 64-bit addressing calculations. If any of these *–famode* options is specified, the compiler will use 64-bit logical arithmetic instructions to perform addressing arithmetic. This allows the resulting code to operate in any runtime environment.

If *–flinux* is specified, the assembler source produced by the compiler should be assembled with the 64-bit z/Linux version of the *as* assembler.

The chapter on z/Architecture programming contains more detailed information about the compiler's z/Architecture support.

The –mfp16 and –mfp4 options (enable/disable use of extended FP registers)

The *–mfp16* and *–mfp4* options allow you to override the setting for whether or not the extended FP registers are available. When *–mfp16* is used, FP registers numbered 0 to 15 are assumed to be available. When *–mfp4* is used, only FP registers 0, 2, 4, and 6 will be used. *–mfp4* is the default, but many of the settings such as *–march=z* will automatically set *–mfp16* because the host platform can be assumed to support these options. To override this setting, the *–mfp4* must occur after any other architecture specifications on the commandline.

The –mlong-double-128 and –mlong-double-64 options (enable/disable 128-bit long double characteristics)

When *–mlong-double-128* is specified, the compiler treats a `long double` data type as 128 bits in size with the characteristics associated with the extended floating point data type.

When *–mlong-double-64* is specified, the compiler treats the `long double` data type as 64 bits, with the same characteristics as the `double` data type.

The *–mlong-double-128* option is the default mode of operating.

The *–fztpf* option enables *–mlong-double-64* to match the configuration of the environment there.

If *–mlong-double-128* is specified, the compiler predefines the `__LONGDOUBLE128` preprocessor macro. If *–mlong-double-64* is specified, `__LONGDOUBLE64` will be predefined.

The –mmvcle and –mno-mvcle options (enable/disable use of the MVCLE/CLCLE instruction)

The `MVCLE (MOVE LONG EXTENDED)` and `CLCLE (COMPARE LOGICAL LONG EXTENDED)` instructions where introduced as part of the "Compare-and-Move-Extended Facility" for the ESA/390 architecture.

By default, the `MVCLE` and `CLCLE` instructions are not used, instead a loop of `MVC` or `CLC` instructions is generated. Enabling the *–mmvcle* option indicates that the compiler can use the `MVCLE` and `CLCLE` instructions in generated code.

The –mdistinct-operands and –mno-distinct-operands options (enable/disable use of distinct-operands facility instructions)

The 9th edition of the z/Architecture hardware architecture introduced the *distinct-operands facility* instructions. These instructions typically have 3 operands, a target and two source operands.

Because of the flexibility this format provides, the compiler can generate better code if it can take advantage of these instructions.

The *–mdistinct-operands* option allows the compiler to use the instructions from the *distinct-operands facilty.*

The –mextended-immediate and –mno-extended-immediate options (enable/disable use of extended-immediate facility instructions)

The 5th edition of the z/Architecture hardware architecture introduced the *extended-immediate facility* which provides several instructions to improve the use of immediate operand values.

The *–extended-immediate* option enables the use of these instructions.

The *–mno-extended-immediate* option can be used to disable the use of these instructions.

The –mload-store-on-condition and –mno-load-store-on-condition options (enable/disable use of load-store-on-condition facility instructions)

The 9th edition of the z/Architecture architecture introduced the *load-store-on-condition facility* instructions, which are `LOCR`, `LOCGR`, `LOC`, `LOCG`, `STOC`, `STOCG`.

If *–mload-store-on-condition* is enabled, the compiler will take advantage of those instructions where it can.

The –mhfp-multiply-add and –mno-hfp-multiply-add options (enable/disable use of HFP multiply-and-add facility instructions)

The *–mhfp-multiply-add* option tells **DCXX** it can use the instructions in the *HFP multiply-and-add/subtract facility*, which was added to the 3rd edition of z/Architecture. These instructions are also enabled by *–march=z3* and above, and can be disabled by *–mno-hfp-multiply-add*.

The –mlong-displacement and –mno-long-displacement options (enable/disable use of long-displacement facility instructions)

The *–mlong-displacement* option tells **DCXX** it can use the instructions in the *long-displacement facility*, which was added to the 3rd edition of z/Architecture. These instructions are also enabled by *–march=z3* and above, and can be disabled by *–no-long-displacement*.

The –mgeneral-instructions-extension and –mno-general-instructions-extension options (enable/disable use of general-instructions-extension facility instructions)

The *–mgeneral-instructions-extension* option tells **DCXX** it can use the instructions in the *general-instructions-extension facility*, which was added to the 7th edition of z/Architecture. These instructions are also enabled by *–march=z7* and above, and can be disabled by *–mno-general-instructions-extension*.

The –mhigh-word-facility and –mno-high-word-facility options (enable/disable use of high-word facility instructions)

The *–mhigh-word-facility* option tells **DCXX** it can use the instructions in the *high-word facility*, which was added to the 9th edition of z/Architecture. These instructions are also enabled by *–march=z9* and above, and can be disabled by *–mno-mhigh-word-facility*.

The –mhfp-extensions and –mno-hfp-extensions options (enable/disable use of HFP extensions facility instructions)

The *–mhfp-extensions* option tells **DCC** it can use the instructions in the *HFP extensions facility*, which was added to ESA/390. These instructions are also enabled by any z/Architecture setting, and can be disabled by *–mno-hfp-extensions*.

The –finline*[=x[:y:z]]* and –fnoinline options (Control inlining optimization)

DCXX features an inliner which can optimize the output code by expanding a function "inline" at its call point. Inlining operates by replacing a call to a function with the operations contained in the function itself. For small functions this can decrease the execution time required by eliminating the function call linkage code. It can also allow optimizations to be performed inside of the inlined function that are aware of the context from which it was called. Inlined functions can actually generate significantly larger code, though, as more than one copy of a function may be generated for all the contexts it is called from.

The –finline*[=x[:y:z]]* option enables inlining. The value *x* specifies the inlining mode. The value *y* specifies a maximum size (approximately number of opcodes) for a function to be a candidate for inlining. The value *z* specifies a maximum stack size for a function to be a candidate for inlining. In mode 0, the inliner is disabled. In mode 1, all functions that are marked inline (with the `inline` keyword, or are defined inline to a class definition) and are smaller than *y:z* are candidates for inlining. In mode 2, all functions that are smaller than *y:z* are candidates for inlining (whether they are explicitly marked or not). For values of *x* greater than 2, extra passes of the inliner are performed, essentially providing greater inlining depth, but this is not recommended.

The inliner now proceeds in a different way than in past versions. It starts with the smallest function and inlines all candidate calls from that function. It then proceeds to larger functions, up until it reaches functions that are larger than *y:z*. It then starts over at the smallest function and continues to repeat this process until either all candidate calls from small functions are inlined, or until there are no more small functions left (because they have all been made big). In this way, anything which is advantageous to inline (i.e., a call to a small function from a small function) will be inlined regardless of its depth in the call tree. Then once it is done with small functions, the inliner performs a global pass (or several passes, in the case where *x* is greater than 2) where it visits every single function call and inlines it if the called function is a candidate for inlining.

The default behavior is now `-finline=1:128:256`, which means to inline any function which is marked inline and is smaller than approximately 128 opcodes long and uses less than 256 bytes of stack space.

The –O*[n]* option (Set optimization level)

The *–O[n]* sets the optimization level. The default setting is to do a minimal level of optimization and inlining, with the intent to produce acceptable code while still having fast compiles. *–O0* disables even these basic optimizations. *–O1* enables a slightly larger set of optimizations, including block-local versions of common subexpression elimination, constant propagation, and dead code elimination. *–O* without an explicit level indicator is the same as *–O2*, and adds more aggressive inlining as well as global versions of common subexpression elimination and constant propagation. The highest setting, *–O3*, enables even more inlining, permits an unlimited number of passes of all of the optimizations, and causes instruction scheduling to reduce latency even at the expense of generating more instructions. For large compilation units, *–O3* could potentially cause the compiler to take a very long time to execute.

Note that the *–g* option (enable debugging information) disables certain optimizations, especially inlining. To override this default (to reenable optimizations), make sure that *–O* occurs after *–g* on the commandline. Also, *–finline* may be specified after *–O* to override the inliner settings, as some code bases perform best with a specific inliner configuration.

The –fasmcomm=*mode* option (control the comments in the assembly output)

The *–fasmcomm=mode* option controls the output of comments in the assembly source which represent lines from the C source code. *mode* can be one of `none`, `source`, `expanded`, or `both`. If it is `none` then no comments are generated for C source lines. If *mode* is `source` then comments are generated which reflect the unprocessed C source code, prefixed with "`---`". When `expanded` is specified comments are generated which reflect the preprocessed (macro expanded) source lines, prefixed with "`***`". If `both` is specified then the unprocessed C source lines are present prefixed with "`---`" and the processed source (when it is different) is present prefixed with "`+++`".

The default is `-fasmcomm=expanded`.

The –asmlnno option (Include line numbers in C source comments in generated assembly)

The *–fasmlnno* option causes the compiler to include line numbers in the C source comments in the generated assembly.

The default is *–fno_asmlnno*.

The –fmin_lm_reg=*val* option (Set the minimum number of registers in one `LM` instruction)

The *–fmin_lm_reg=val* option determines the minimum number of consecutive load instructions which will be collected into a single `LM` or `LMG` instruction by the compiler's peephole optimizer. The default value is 2.

The –fmin_stm_reg=*val* option (Set the minimum number of registers in one `STM` instruction)

The *–fmin_stm_reg=val* option determines the minimum number of consecutive store instructions which will be collected into a single `STM` or `STMG` instruction by the compiler's peephole optimizer. The default value is 2.

The –fflex option (Enable FLEX/ES-specific optimizations)

The *–fflex* option tells the compiler it is targetting a FLEX/ES platform and should make the appropriate optimizations. Currently this option has the same effect as `-fmin_lm_reg=4 -fmin_stm_reg=8`.

The –frsa[=*size*] option (Specify the amount of space the compiler reserves for the Register Save Area)

The *–frsa=size* option causes the compiler to reserve the specified *size* bytes at the beginning of each per-function local stack area as the "Register Save Area" size.

This option is useful when using custom prologue/epilogue macros that may want to apply different techniques for saving/restoring the register values at function entry and exit.

In Systems/C mode (*–flinux* and *–fc370* not specified), the compiler reserves 80 bytes of space for 31-bit programs and when *–mlp64* option is enabled (in 64-bit mode), the compiler reserves 168 bytes. This space is used by the default `DCCPRLG` and `DCCEPIL` macros to save and restore register values.

The *–rsa=size* option can specify any positive value for the register-save-area size, including zero. If it is specified as some value other than the default, then the default `DCCPRLG` and `DCCEPIL` macros should not be used.

The *size* parameter is automatically adjusted to a multiple of 8 bytes, to enforce C memory allocation requirements.

This option should not be used in conjunction with the *–fc370* or *–flinux* options, as the register save area in those situations is architected by the runtime environment.

The –fpack=*val* option (Specify a default maximum structure alignment)

The *–fpack*=val provides a default maximum structure alignment. Specifying this parameter is functionally equivalent to specifying `pragma pack(`*val*`)`.

The –fpic option (Generate position independent code, small GOT)

When *–flinux* or *–fztpf* options are specified, the *–fpic* option can be used to cause the compiler to generate position independent code. The resulting object can then become part of a Linux or z/TPF shared library. The *–fpic* option causes the compiler to generate code assuming a small Global Offset Table (GOT), where it uses 12 bits of displacement to index into the table. If the GOT grows too large at link time, then the *–fPIC* option can be used to indicate that the generated code should assume a large GOT.

When building for use with the Systems/C runtime, *–fpic* causes the creation of code suitable for linking into a shared library. It also enables *–frent* and *–ffpremote*, so that each library can have its own PRV. External symbols will be encoded to use an extra level of indirection. A reference to external symbol "`foo`" generates a Q-con named "`&foo`", which will be filled in by the dynamic linker with the address of the variable, whereever it is resolved from. Likewise, a definition of the symbol causes a definition of the "`&foo`" Q-con as well. Special reentrant initializer scripts are emitted so that **PLINK** and the runtime know what to do with these indirect symbols.

The –fPIC option (Generate position independent code for Linux & z/TPF, large GOT)

When *–flinux* or *–fztpf* options are specified, the *–fPIC* option can be used to cause the compiler to generate position independent code.

The resulting object can then become part of a Linux or z/TPF shared library.

The *–fPIC* option causes the compiler to use complete displacements into the Global Offset Table (GOT), allowing for the largest program to be built as a shared library.

The –ffpremote/–ffplocal options (function pointers are remote/local)

By default, function pointers are local. If *–ffpremote* is specified, then they will be remote. A remote function pointer contains the PRV to be used for the function, and it is often needed for shared library situations (where more than one PRV may be in play at a time). See the section on remote function pointers on page 132 for more details.

The –fxplink option (Use eXtra Performance Linkage)

The *–fxplink* option instructs the compiler to use IBM's eXtra Performance Linkage (XPLINK). The compiler generates appropriate re-entrant DLL references to XPLINK variables and code. It also uses the optimized XPLINK function calling conventions.

Note that *–fxplink* is only effective when *–fc370* is also applied. The *–fxplink* option does imply *–frent*.

The –fc370_extended option (Enable C/370 LANGLVL(EXTENDED) compatibility mode)

The *–fc370_extended* option causes **DCXX** to generate objects compatible with IBM's C++ compiler run with the `LANGLVL(EXTENDED)` option. For now this only has an effect when *–fxplink* is specified. It enables Run-Time Type Information.

The –fuser_sys_hdrmap option (Use user `$$HDRMAP` for system `#includes`)

When a `#include` directive is processed, the file name may be altered depending on rules in `$$HDRMAP` files. The system `$$HDRMAP` files are found as if a `#include <$$HDRMAP>` was processed, and the user ones are found as if `#include "$$HDRMAP"` was used instead. When *–fuser_sys_hdrmap* is specified, **DCXX** searches for system headers using the rules from both the user and system `$$HDRMAP` files. When *–fnouser_sys_hdrmap* is specified, searches for system headers use only the rules from the system `$$HDRMAP` files. In either case, searches for user headers use just the user `$$HDRMAP` rules.

–fuser_sys_hdrmap is the default.

The –fevents=*filename* option (Emit an IBM-compatible events listing)

The *–fevents=filename* option causes **DCXX** to generate an event listing in the named file. Several IBM products use event listings of this format to communicate error message information between compilers and user interfaces. Using this option, you may generate an events file for use with any products that share this format.

The events file contains 3 types of single-line records:

`ERROR 0` *A* `0 0` *B C* `0 0 DCX`*D E F G H*

A The number of the file where the error occurred.

B The line number at which the error occurred.

C The column number at which the error occurred.

D The error code.

E A severity, one of `I` for information, `W` for warnings, `E` for errors, or `U` for unrecoverable errors.

F The mainframe return code for the error.

G The length of the error message.

H The error message.

`FILEID 0` *A B C D*

A The number of the file that is beginning.

B The line number of the `#include` that caused this file to be listed.

C The length of the file name.

D The file name for the file that is beginning.

`FILEEND 0` *A B*

A The number of the file that is ending.

B The number of lines processed in that file.

The –fnamemangling=*mode* option (Select the name mangling mode to use for IBM compatibility)

The *–fnamemangling=mode* option can be used to select one of two modes:

ansi ANSI mangling mode is the default when *–fc370* is not specified and causes **DCXX** to use a name-mangling that is consistent with the ANSI standard.

compat The default when *–fc370* is specified. Instructs the compiler to generate mangled symbol names compatible with IBM's compiler. In specific, this causes type qualifiers like `const` to be represented in parameter type encodings, even though the ANSI standard does not allow overleading a function differing only by type qualifiers.

The –fenum=*val* option (Specify default enumeration size)

The *–fenum*=val specifies the default enumeration size when compiling in IBM compatibilty mode (when the *–fc370* option is enabled.)

Specifying this parameter is functionally equivalent to specifying `pragma enum(`*val*`)`.

The value *val* can be specified as `SMALL`, `INT`, `1`, `2` or `4`.

The *–fenum*=val is only useful when the *–fc370* option is enabled.

The –ftest*[=name]* option (Enable a separate test csect)

The *–ftest* option enables the creation of a separate CSECT for test (debugging) data. It only has an effect when combined with the *–fc370* and *–g* options (LE370 ISD debugging). The name of the section must be specified either as an argument to *–ftest* or with a `#pragma csect(test, "name")` statement. Most ISD-related debugging information is put in the test CSECT.

The –fprolkey=*key* option (Append a global prologue key)

The *–fprolkey=key* option causes **DCXX** to append *key* to all `DCCPRLG` invocations, as if it had been specified on each function using `#pragma prolkey`. If `#pragma prolkey` and *–fprolkey* are both specified, they are concatenated.

The –fcommon and –fnocommon options (Enable/disable common linkage for uninitialized globals)

In Linux/390, z/Linux, and z/TPF modes (the *–flinux* or *–fztpf* options are specified, all defined global data is by default placed in `.data`, which is the behavior when *–fnocommon* is specified. However, if *–fcommon* is specified then any uninitialized global data is placed in `.bss` instead. Definitions in `.bss` take up less space in the object files and, more importantly, do not generate linker messages for duplicate definitions.

The –fsave_dsa_over_call/–fno_save_dsa_over_call options (Control if DSA bytes are saved and restored over alternate linkage call)

The *–fsave_dsa_over_call* option indictes that, for Systems/C mode, the save-chain area of the DSA should be saved and restored across linkage-OS and linkage-ASM function calls. These areas are used in the Dignus runtime and can be overwritten by linkage-OS and linkage-ASM functions.

By default the linkage areas are saved and restored across calls to these alternative linkage functions.

This option is only meaningful for Systems/C mode, and not applied when the *–flinux* or *–fc370* options are specified.

The –fdfe and –fnodfe options (Enable/disable dead function elimination.)

Normally the compiler does not generate code for unreferenced `static` functions or class members. If the function is declared `static` but not invoked, or referenced via its address, then it cannot be reached and thus does not need to be present in the resulting code.

This optimization is called "dead function elimination".

The *–fnodfe* option defeats dead function elimination, so that those functions will appear in the generated code.

The default if *–fdfe*. If the *–g* option is enabled, requesting debuggable code, then *–fnodef* will be enabled in case the user wishes to reference such functions during a debug sessions. *–fdfe* can be used to re-enable it.

Keeping unreferenced functions in the generated code can dramatically increase the size of resulting objects and programs.

The –fmapat and –fnomapat options (Enable/disable mapping '@' to '_' in external symbol names)

If *–fmapat* is specified then any at signs ('`@`') in `#pragma map` directives will be replaced with underscores ('_'). This option is especially useful in Linux or z/TPF modes where at signs are not valid in symbol names.

The –fat option (Support @-operator in expressions)

The `@` operator is an extension provided by **DCXX** to assist in passing arguments by-reference to assembly language functions.

The `@` operator is similar to the `&` operator in standard C, in that it produces the address of the following expression, but can be used on rvalue expressions as well as lvalue expressions.

See the section on the `@` operator on page 113 for more information about the use of `@`.

The default is *–fnoat*.

The –fctrlz_is_eof and –fno_ctrlz_is_eof options (Enable/disable treating control-Z as an EOF character)

On Windows hosts, the character associated with control-Z (`0x1A`) has traditionally (since DOS) been used to indicate the end of file. So on Windows hosts we default to *–fctrlz_is_eof* so that any files with a control-Z in them will be terminated at that point. Contents of the file after the control-Z will then be ignored. On all non-Windows hosts the default is *–fno_ctrlz_is_eof*, meaning that control-Z will be treated like any other character in the source code. Note that the C language assigns no meaning to control-Z so if it occurs outside of a comment it may still generate a language-level error message.

The –fpermissive_friend and –fno_permissive_friend options (Enable/disable friend declarations on private members)

According to the ISO C++ standard, `friend` declarations may only refer to accessible members of other classes. However, older compilers would commonly permit `friend` declarations to refer to `private` members of other classes. The default is the modern behavior, *–fno_permissive_friend*. For compatibility with the older behavior, use the *–fpermissive_friend* option.

The –ffnio/–fno_fnio options (enable/disable function names in objects for debugging)

Often it is necessary to be able to determine which function you are looking at when reading a memory dump. Some linkages (such as the `DCCPRLG` macro) provide this information by default, and others provide it via indirect pointers to debug information. But if neither of those options is convenient, use *–ffnio* (function name in object) to guarantee that a string containing the name of the function will be present in memory just before the entry point of the function. The default behavior is to not emit the string, corresponding to *–fno_fnio*.

The –fhide_skipped/–fshow_skipped options (enable/disable omission of preprocessor-skipped lines in listing)

The preprocessor will skip certain source lines, due to constructs like `#if 0`. By default (*–fshow_skipped*), these skipped lines will be output in the compiler listing. However, if *–fhide_skipped* is specified then they will be omitted from the listing. In some situations this can make a much more readable *–flisting* output. These options only affect the informational listing, and not the generated code.

The –fsigned_bitfields and –funsigned_bitfields options (set default signedness of bitfields with bare types)

If a bitfield declaration does not specify an explicit `signed` or `unsigned` keyword and *–fsigned_bitfields* is specified then the compiler will use the signedness inherent in the type. For example `int` is a signed type, so `int x:1` will define a 1-bit signed bitfield.

However, if *–funsigned_bitfields* is specified then **DCXX** will use an unsigned type for bitfields unless the `signed` keyword is explicitly specified in the declaration.

If *–fztpf* or *–flinux* is specified then *–fsigned_bitfields* is the default, for compatibility with **gcc**. Otherwise, *–funsigned_bitfields* is the default, as is typical for other mainframe compilers.

The –v option (print version information)

The *–v* option causes **DCXX** to print the version information on the STDERR stream and exit with a return code of 0.

The –fsched_inst, –fsched_inst2 and –fno_sched_inst options (control the behavior of the instruction scheduler)

DCXX has an instruction scheduler which will attempt to reorganize the instruction sequence so that any instruction which reads a value in a register is separated from the instruction which initializes that register. On modern architectures such as z/Series, this can cause a substantial performance improvement by minimizing pipeline stalls. The reordered code can be hard to debug, because the point where one expression ends and another begins is effectively blurred.

By default the compiler uses the setting of *–fsched_inst*, meaning a single pass of scheduling is completed just before the compiler is done. In this case, the exact same instructions are generated as without scheduling, but their order may be changed.

The *–fno_sched_inst* option disables instruction scheduling, to produce more readable code. It is the default if *–g* is specified.

The *–fsched_inst2* option causes the compiler to perform an additional pass of scheduling before register allocation and peephole optimization. This way, scheduling can have a more substantial impact on the generated code. It has the general effect of making register contention higher, as each register is in use over a longer span of time. Thus it may result in slightly larger code with more spilled registers. Because of the high cost of a pipeline stall, it is often faster even so. If you specify *–O3* then that will imply *–fsched_inst2*.

The –fxref and –fno_xref options (enable/disable cross-reference listing

If *–fxref* is specified, then the **DCXX** listing will contain an extra section with cross reference information, indicating where each symbol is read or modified.

The –frestrict and –fno_restrict options (enable/disable C99-style `restrict` keyword)

If *–frestrict* is specified, then the C99-style `restrict` keyword will be supported. By default, `restrict` is not supported, as it is not part of standard C++. If *–fc370* is specified (compatibility with IBM's Language Environment), then `restrict` is enabled automatically.

The –fcpp98 option (specify only C++98 will be accepted)

By default, **DCXX** accepts source code that complies with the ISO/IEC 14882:2011 standard, known as C++11. If *–fcpp98* is specified, then **DCXX** will require the source code to comply with the earlier ISO/IEC 14882:1998 standard, known as C++98.

The –fcpp11 option (enable support for C++11 language features)

The *–fcpp11* option is the default behavior for **DCXX**, indicating that code conforming to the ISO/IEC 14882:2011 standard will be accepted.

To use *–fcpp11* when compiling code that was developed for C++03, it may be necessary to also specify *–fno_parse_templates* to restore the C++98 default of not parsing generic template code until it is being instantiated. It may also be necessary to specify *–fno_implicit_noexcept* because the implicit `noexcept` specifier present on some destructors will often cause incompatibilities in inherited virtual functions.

See the chapter "C++ Language Features" on page 12 for more details.

The –fcpp14 option (enable support for C++14 language features)

If *–fcpp14* is specified, **DCXX** will accept code using the new features found in the ISO/IEC 14882:2014 standard. See the chapter "C++ Language Features" on page 14 for more details.

The –fcpp17 option (enable support for C++17 language features)

If *–fcpp17* is specified, **DCXX** will accept code using the new features found in the ISO/IEC 14882:2017 standard. See the chapter "C++ Language Features" on page 15 for more details.

The –funrestricted_unions and –fno_unrestricted_unions options (Enable/disable the C++11 unrestricted unions feature)

Unrestricted unions are a C++11 feature that allows you to specify classes that require construction or destruction as members of a union. Construction using placement `new` must then be manually paired with the correct explicit destructor calls. The *–funrestricted_unions* option can be used to enable unrestricted unions even in *–fcpp98* mode.

The –fimplicit_noexcept and –fno_implicit_noexcept options (Enable/disable the implicit C++11 exception specifications)

The `noexcept` function specifier is a new C++11 that allows the programmer to tell the compiler that it may issue a non-recoverable call to `std::terminate()` if any exceptions are thrown. The C++11 standard says that `noexcept` will be assumed on certain destructors and deallocators. This can cause trouble for C++98-conforming code, because compatibility between `noexcept` specifiers is necessary in inherited virtual functions. If *–fcpp11* or *–fimplicit_noexcept* is specified, then the implicit `noexcept` specifiers will be applied. However, if *–fno_implicit_noexcept* is specified, then code containing C++98-conforming virtual destructors will compile cleanly even in C++11 mode.

The –fstatic_anon_names and –fno_static_anon_names options (Enable/disable forcing members of the unnamed namespace to static)

By default, members of the unnamed namespace can be externally-visible symbols with unique compiler-generated names. But it is equally valid to treat them as static, because nothing in another compilation unit could need to reference them. *–fstatic_anon_names* causes the compiler to treat all of them as static local-only definitions. *–fno_static_anon_names* is the default.

The –fsource_enc=utf8 and –fsource_enc=ascii options (Select source character encoding)

The *–fsource_enc=utf8* option causes **DCXX** to treat the source input files as UTF-8. Multi-byte characters will be decoded to the appropriate unicode code point. This

allows unicode to be used in string literals such as `u'...'` and `u"..."`. The default is *–fsource_enc=ascii*, which treats each byte as a single code point.

These options are only available on ASCII hosts. EBCDIC hosts always use an 8-bit character encoding.

The –fdwarf_extern and –fno_dwarf_extern options (enable/disable generation of DWARF data for extern variables)

The *–fdwarf_extern* option enables the generation of full DWARF location info for `extern` variables. The default (*–fno_dwarf_extern*) is to only generate location info for locally-defined variables. Note that non-referenced variables will still not have any debug information generated for them.

Assembling the output

Using HLASM

For traditional mainframe operating systems (MVS, z/OS, etc...) **DCXX** generates HLASM-style assembly code which is assembled with the Dignus **DASM** program.

For Linux, z/Linux and z/TPF, the compiler generates output in the GNU GAS style, and the GAS assembler is used. For information about how to use GAS to create object files, see the chapter "Compiling for Linux/390, z/Linux and z/TPF" (page 161).

This section describes the programs for building programs for traditional mainframe operating systems.

DCXX generates assembler source that generally requires the use of the Dignus assembler, Systems/ASM, V1.50 or later. The generated assembler source may contain features and extensions which are not recognized by the IBM HLASM assembler. Using HLASM to assemble **DCXX**-generated source is not supported.

Using Systems/ASM

The Systems/ASM assembler, **DASM** version 1.50 or later, can be used on cross-platform hosts or natively on OS/390, z/OS and z/VM. The Systems/ASM assembler will generate either OMF, Extended C/370 or GOFF object files. Extended C/370 object files use XSD cards instead of ESD cards allowing for external identifiers longer than 8 characters.

DCXX generates assembler source that contains some extensions supported by **DASM** that support the advanced linking features required to implement the C++ language.

When assembling source generated by **DCXX**, the **DASM** *–fdupalias* option should be used.

The IBM pre-linker and binder examine the IDR information on END cards to determine the version of the C compiler which generated the object. The section on IBM C++ compatibility in this document describes those requirements in more detail. The compiler-generated code will properly set the IDR value.

A typical **DASM** command line on a cross-platform UNIX host is shown below. This would be similar for Windows.

```
dasm -Lc:/dignus/maclib -macext . -fdupalias -o my.obj myfile
```

Note that the **DASM** command specifies where to search for the dignus macros via the —*L* option.

For more information, consult the Systems/ASM documentation.

Linking Assembled Objects

For traditional operating system targets, the assembled object decks can be linked into an executable load module.

Once the compiler-generated assembly source has been assembled, the disparate objects can be linked into an executable load module. If the Systems/ASM assembler was used to cross-assemble the assembly source, the object decks should be transferred to OS/390 via FTP or some other binary-mode transfer mechanism.

Systems/C contains two versions of the C libraries — the RENT version for generating re-entrant programs and the non-rent version for generating non-re-entrant programs. The Systems/C++ C++ library is currently only available in RENT form, so the *–frent* option (default) should be used with the RENT libraries.

Systems/C++ programs require the use of the Systems/C++ pre-linker, **PLINK**.

A note on re-entrant (RENT) programs

Re-entrant (RENT) programs are programs which can safely be linked with the RENT option applied to the IBM LINKER, and can be placed in the OS/390 LINKLST, etc. They are, generally speaking, programs which do not modify their own loaded sections, but instead allocate memory to contain program variables at program start-up.

All C++ source is compiled in "RENT" mode, thus the compiler will place all of the `extern` and `static` variables in the pseudo-register vector, the **PRV**. These variables are referred to by `Q-CON` references in the generated assembly source.

By default, the Systems/C pre-linker **PLINK** gathers all of the `DXD` definitions together allocating an entry for each in the **PRV**, and adjusts the `Q-CON` references accordingly. Alternatively, this step can be deferred, allowing the IBM or PLINK binding steps.

At start-up, the Systems/C library allocates the appropriate space for the **PRV**, and retains a pointer to the **PRV** at a known location. (Note that for z/Architecture programs, this allocated space may be above the 2-gigabyte "bar".)

At run-time, a reference to a variable in the **PRV** uses the **PRV** pointer and the value the linker has substituted for the `Q-CON`, adding them together to produce the run-time offset for the variable.

An issue arises because of variable initialization allowed by the ANSI C standard. For example, the address of a variable in the **PRV** isn't known until run-time, when the **PRV** is allocated, but is a valid file-scoped initialization value.

Because of this, the Systems/C++ compiler, **DCXX**, and the Systems/C compiler, **DCC**, produce run-time initialization scripts which the Systems/C library processes

at program start up, after the **PRV** has been allocated. It is the job of the pre-linker, **PLINK**, to locate the start of these scripts in each object and gather them together. **PLINK** then places a list of these at the end of the resulting object, in a known section. The run-time library walks the list, interpreting the scripts it finds.

Thus, RENT programs must be processed with the Systems/C and Systems/C++ pre-linker, **PLINK**, to ensure proper run-time initialization of variables located in the **PRV**.

Using PLINK

PLINK gathers the input objects together, performing AUTOCALL resolution where appropriate, handling **PRV** allocations and references, and produces a single file which can then be processed by the IBM BINDER or older IEWL linker. Alternatively, **PLINK** can also directly produce the load module, avoiding the IBM BINDER step.

As **PLINK** gathers objects, it examines the defined symbols, looking for a Systems/C++ initialization script section and other object file processing that may need to be performed.

For detailed information on **PLINK**, see the **PLINK** section in the *Systems/C Utilities* manual.

On cross-hosted platforms (Windows and UNIX), **PLINK** is typically executed with the object files listed on the command line; and a *–S* option or **DAR** archive names to locate any required library objects.

For example, on a Windows platform the command:

```
plink prog.obj C:\sysc\lib\libstdcxx_mvs.a C:\sysc\lib\libcxx_mvs.a
     "-SC:\sysc\lib\objs_rent\&M"
```

will read `prog.obj`, using the `libstdcxx_mvs.a` and `libcxx_mvs.a` **DAR** archives and then the `C:\sysc\lib\objs_rent` directory for any AUTOCALL references. Because no *–o* option was specified, the resulting object file is written to the file `p.out`.

This command, on UNIX platforms:

```
plink t1.obj t2.obj libone.a -L/usr/local/sysc -lstdcxx_mvs -lcxx_mvs
```

will read the two primary input objects `t1.obj` and `t2.obj`. It will try and resolve references from the **DAR** archive `libone.a`, then the two **DAR** archives `/usr/local/sysc/libstdcxx_mvs.a` and `/usr/local/sysc/libcxx_mvs.a`.

On OS/390 and z/OS, under TSO or batch JCL, **PLINK** operates similar to the IBM pre-linker. The resulting gathered object is written to the file `//DDN:SYSLMOD` unless otherwise specified. **PLINK** has a default library template of `-S//DDN:SYSLIB(&M)` which causes it to look in the `SYSLIB` PDS for autocall references. Other input objects, `-S` library templates or **DAR** archives may be added in the PARMS option on the **PLINK** step. **PLINK** reads the file `//DDN:SYSIN` as the initial input file. Typically, this file contains `INCLUDE` cards to include the primary objects for the program. Other primary input files may be included in the PARMS for **PLINK**. For example, the following JCL reads the object `INDD(PROG)` and uses `DIGNUS.LIBSTDCX`, `DIGNUS.LIBCXX` and `DIGNUS.LIBCR.OBJ` as the autocall libraries:

```
//PLINK EXEC PGM=PLINK
//STEPLIB DD DSN=Systems/C load library,DISP=SHR
//STDERR  DD SYSOUT=A
//STDOUT  DD SYSOUT=A
//SYSLIB  DD DSN=DIGNUS.LIBSTDCX,DISP=SHR
//         DD DSN=DIGNUS.LIBCXX,DISP=SHR
//         DD DSN=DIGNUS.LIBCR,DISP=SHR
//INDD DD DSN=mypds,DISP=SHR
//SYSIN DD *
 INCLUDE INDD(PROG)
//SYSMOD DD DSN=myoutput.obj,DISP=NEW
```

Note that the `STDERR` and `STDOUT` DDs were specified for **PLINK**'s message output. Also, the `ARLIBRARY` control card could have been used to add additional **DAR** archive files for resolving external references.

Systems/C++ programs can also be pre-linked and linked for the OpenEdition shell. Under the OpenEdition shell, **PLINK** operates as it would under any other UNIX platform. After pre-linking, the final link can be accomplished using the

```
cc -e // -oprogram  plinked-file
```

command. Where *program* is the resulting load-module and *plinked-file* is the previous **PLINK** output.

For more detailed information regarding **PLINK** and the other Systems/C utilities, see the *Systems/C Utilities* manual. Also, the *Systems/C C Library* manual contains more information on linking with the Systems/C C library and the OpenEdition runtime environment.

Other useful utilities

Systems/C++ provides other useful utilities. More details and examples of their use can be found in the *Systems/C Utilities* manual.

DAR — the Systems/C Archive utility

The Systems/C archive utility, **DAR**, creates and maintains groups of files combined into an archive. Once an archive has been created, new files can be added and existing files can be extracted, deleted or replaced. Files gathered together with DAR can be used to resolve AUTOCALLed references from **PLINK**.

DRANLIB — the Systems/C Archive index utility

DRANLIB is used to index a Systems/C archive to allow for AUTOCALL references to longer names, or to names which are not dependent on the archive member name. **DRANLIB** will create a `__SYMDEF` member in the Systems/C archive which **PLINK** will consult when looking for symbolic resolutions.

DPDSLIB — the Systems/C PDS library utility

DPDSLIB is used to index a PDS library on OS/390 or z/OS to allow for AUTOCALL references to longer names, or to names which are not dependent on the PDS member names. **DPDSLIB** will create a `##SYMDEF` member in the PDS which **PLINK** will consult when looking for symbolic references.

The Systems/C++ C++ library PDSs on z/OS and OS/390 have been processed with the **DPDSLIB** utility, to allow `PLINK` to AUTOCALL the longer C++ names present in the C++ library.

GOFF2XSD — convert GOFF format objects to XSD format

GOFF2XSD is used to convert GOFF format objects to XSD format. GOFF format objects are created by the IBM HLASM assembler when the `XOBJECT` option is enabled or **DASM** when the `-goff` option is enabled.

DCCPC — Dignus CICS Command Processor

DCCPC takes as input C source code containing `EXEC CICS` commands and generates pure C source that interfaces with the CICS run-time.

Linking programs on z/OS and OS/390

Before execution, programs must be prepared using the Systems/C++ pre-linker, **PLINK**, then the IBM BINDER.

Systems/C provides two versions of the C library, one for RENT programs and one for non-RENT programs. The Systems/C++ C++ library is currently only available in RENT form. It is important to link with the appropriate version. If any source programs reference variables found in the Systems/C++ or Systems/C libraries (e.g., `errno`) and that program was compiled with the *–frent* option (the default), then the re-entrant version of the C and C++ libraries should be used.

Using the incorrect version of the library will cause strange run-time errors. The installation instructions for your particular host platform will detail where to find the correct library. Normally the Systems/C++ and then the Systems/C library is specified as the last library to use for AUTOCALL resolution in the **PLINK** step. **PLINK** must be used for all C++ programs to properly support C++ language features. Also **PLINK** can take advantage of **DAR** archive libraries and **DPDSLIB** processed PDS libraries for external reference resolution.

In the following example JCL, there are three objects to link together to form the resulting executable, *MAIN*, *SUB1*, and *SUB2*, representing a main module and two supporting sub-modules. These are found in the PDS *MY.PDS.OBJ.* The resulting executable is written to *MY.PDS.LOAD(MYPROG).*

```
//LINK JOB
//PLINK EXEC PGM=PLINK,REGION=2048K
//STEPLIB DD DSN=Systems/C load library,DISP=SHR
//STDOUT  DD SYSOUT=*
//STDERR  DD SYSOUT=*
//SYSLIB  DD DSN=DIGNUS.LIBCXX,DISP=SHR
//        DD DSN=DIGNUS.LIBSTDCX,DISP=SHR
//        DD DSN=DIGNUS.LIBCR,DISP=SHR
//SYSMOD  DD DSN=&&PLKDD,UNIT=VIO,DISP=(NEW,PASS),
//        SPACE=(32000,(30,30)),
//        DCB=(RECFM=FB,LRECL=80,BLKSIZE=3200)
//INDD    DD DSN=MY.PDS.OBJ,DISP=SHR
//SYSIN   DD *
  INCLUDE INDD(MAIN)
  INCLUDE INDD(SUB1)
  INCLUDE INDD(SUB2)
//STDIN   DD *
//LINK EXEC PGM=IEWL,REGION=2M,PARM=('LIST',
//  'MAP,XREF,LET',
//  'ALIASES=NO,UPCASE=NO,MSGLEVEL=4,EDIT=YES')
//SYSPRINT DD SYSOUT=*
//SYSUT1   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSUT2   DD UNIT=SYSDA,SPACE=(CYL,(1,1))
//SYSLIN   DD DSN=*.PLINK.SYSMOD,DISP=(OLD,DELETE)
//SYSLMOD  DD DSN=MY.PDS.LOAD(MYPROG)
```

First, the Systems/C++ pre-linker, **PLINK** is invoked, specifying the inclusion of the three object modules and the Systems/C++ and Systems/C reentrant libraries. This step could have been performed on a cross-platform host, running **PLINK** there. Then the IBM BINDER is invoked for final linking and generation of the resulting load module.

Running programs on z/OS and OS/390

Once a program has been successfully linked, it is a typical OS/390 or z/OS load module and may be executed via JCL or the TSO CALL command as any other load module.

The Systems/C++ and Systems/C libraries contain no modules that are loaded during program execution, meaning it is "all-resident." As such, there are no run-time library concerns, and no particular modules which must be present in a *STEPLIB* DD.

The I/O portion of the Systems/C C library reserves file descriptors #0, #1 and #2 for association with `//DDN:STDIN`, `//DDN:STDOUT` and `//DDN:STDERR`. Thus, the DD-names `STDIN`, `STDOUT` and `STDERR` must be properly allocated. The *Systems/C C Library* manual contains more information regarding file descriptors and I/O.

For more information about the Systems/C++ run-time environment, consult the *Systems/C C Library* manual.

DCXX Advanced Features and C++ Extensions

The Systems/C++ compiler, **DCXX**, provides many advanced features. These features combine to produce a programming environment which is perfectly suited for many systems programming tasks.

Predefined macros

The following predefined macros are defined by the Systems/C++ compiler.

`__BFP__`	`__BFP__` is defined to `1` if the *–fieee* option was specified. This indicates that IEEE floating point values will be generated by default.
`__COUNTER__`	`__COUNTER__` is initially defined to `0` and incremented each time it is referenced. `__COUNTER__` can be useful to create unique variable names, or within inline assembly language macros.
`__CHAR_UNSIGNED__`	`__CHAR_UNSIGNED__` is defined to `1` if the `char` data type is unsigned by default. This is the typical mode of compilation.
`__SYSC__`	`__SYSC__` is always defined to the value `1`, indicating the source is being compiled with the Systems/C compiler.
`__I390__`	`__I390__` is always defined to the value `1`.
`__SYSC_VER__`	`__SYSC_VER__` is defined to a string containing the Systems/C compiler version number.
`__SYSC_ASCIIOUT__`	`__SYSC_ASCIIOUT__` is defined to `1` if the *–fasciiout* option was enabled. This indicates that character and string constants will be generated as ASCII values.
`__SYSC_ANSI_BITFIELD_PACKING__`	`__SYSC_ANSI_BITFIELD_PACKING__` is defined to `1` if the *–fansi_bitfield_packing* option was specified.

`__SYSC_LP64`	`__SYSC_LP64` is defined to the value `1` if the *–mlp64* option is enabled. This indicates that pointers and the `long` data type are 64-bits wide.
`_LP64`	`_LP64` is defined to the value `1` when the *–mlp64* option is enabled. This indicates that pointers and the `long` data type are 64-bits wide.
`__SYSC_ILP32`	`__SYSC_IPL32` is defined to the value `1` when the *–milp32* option is not enabled. This indicates that pointers and the types, `int` and `long` are 32-bits wide.
`_ILP32`	`_IPL32` is defined to the value `1` when the *–milp32* option is not enabled. This indicates that pointers and the types, `int` and `long` are 32-bits wide.
`__ptr31__`	`__ptr31__` is defined to be `__ptr32` which is recognized as equivalent to the Systems/C `__ptr31` keyword.
`__PTR31`	`__PTR31` is defined to the value `1`, indicating this compiler recognizes the `__ptr31` and `__ptr64` keywords.
`__PTR32`	`__PTR32` is defined whenever `__PTR31` is defined.

__attribute__

DCXX supports the `__attribute__` extension found in the g++ compiler, but only for *–flinux* or *–fztpf*. This extension is used to provide attributes on declarations outside of the scope of the C standard. Attribute-clauses may be placed at the end of structure/union definitions, within structure member lists, after variable declarations and within function declarations, or anywhere a type qualifier/specifier can be used.

An attribute-clause has the form:

```
__attribute__((value))
```

notice that two parenthesis are required.

Unrecognized `__attribute__` clauses are silently ignored.

`constructor/destructor` attributes

`__attribute__((constructor))` applies to function definitions, and indicates that the given function is a constructor-type function and should be executed when C++ constructors are executed, prior to the invocation of the `main` function.

`__attribute__((destructor))` applies to function definitions, and indicates that the given function is a destructor-type function and should be executed when C++ destructors are executed, after the `main` function has returned or `exit` has been called.

For example, the following source declares two functions, `construct` and `destruct`, which will be executed along with C++ constructors and destructors appropriately:

```
void __attribute__((__constructor__)) construct(void)
{
   printf("I am executed along with C++ constructors\n");
}

void __attribute__((__destructor__)) destruct(void)
{
   printf("I am executed along with C++ destructors\n");
}
```

`packed` attribute

`__attribute__((packed))` applies to `struct` and/or `union` definitions. If `__attribute__((packed))` appears after the structure or union definition, it indicates that the elements within the structure should be allocated without regard for their alignment requirements. Thus, the elements in the structure are "packed" together without any alignment bytes. Consider, for example, this structure

```
struct unpacked {
     char c;
     int i;
};
```

The `sizeof` operator applied to `struct unpacked` would result in a value of 8, because the alignment of `int` data requires that it be allocated on a 4-byte boundary. Thus, there are 3 extra bytes of padding between the fields '`c`' and '`i`'.

However, if the `__attribute__((packed))` attribute is applied, as in this example:

```
struct packed {
     char c;
     int i;
} __attribute__((packed));
```

then `sizeof` applied to `struct packed` would result in a value of 5, 1 byte for the field '`c`' and 4 bytes for the field '`i`'. The fields in the structure are allocated without regard for their alignment requirements, and are "packed" together as close as possible.

`mode` attribute

`__attribute__((mode(`*value*`))` can apply to any numeric or pointer type, and serves to force a specific size on a type, irrelevant of the underlying type. Supported modes:

Mode	Bits
`byte`	8
`word`	32 or 64 depending on pointer mode
`pointer`	32 or 64 depending on pointer mode
`QI`	8
`HI`	16
`SI`	32
`DI`	64

The IBM-provided headers for z/TPF use modes SI and DI as alternatives to the `__ptr31` and `__ptr64` keywords to specify a pointer size. For example:

```
void *__attribute__((mode(SI))) voidptr32;
void *__attribute__((mode(DI))) voidptr64;
```

`weak` attribute

A symbol may be modified with `__attribute__((weak))` to indicate that it should use weak linking. For a defined symbol, weak linking indicates that multiple definitions of the same symbol are to be silently ignored. For an undefined (`extern`) symbol, weak linking indicates that there should be no linker error message if the symbol has no definition. Function and variable symbols can both be weak. Weak linking is very dependent upon the linker used. On some platforms, a missing weak symbol can be detected by comparing the address of the symbol to `0`. Example:

```
extern int __attribute__((weak)) weakvar;
int is_weak_defined(void) {
   if (&weakvar == (int *)0) {
      return 0; /* not defined */
   } else {
      return 1; /* is defined by another comp unit */
   }
}
```

`deprecated` attribute

The `__attribute__((deprecated))` attribute can appear after a declaration of a function, variable or typedef. Subsequent uses of the declared symbol will cause the compiler to generate warning message #1293, indicating the symbol is deprecated. If possible, the message will also contain the file name and line number of where the symbol is declared so the user can refer to the declaration for more information.

`visibility` attribute

ELF linkage attributes can be controlled with `__attribute__((visibility("`*mode*`")))`. The valid visibility modes are `default`, `hidden`, `protected`, and `internal`. Their meaning is defined by the linker. Note that they only have an effect when *–flinux* or *–fztpf* is in effect, as other platforms do not use **ELF**.

For shared libraries, it may be useful to have symbols default to `hidden` except for a few which are explicitly exported. This can be accomplished by putting *–fvisibility=hidden* on the command line and then marking individual definitions:

```
int this_is_hidden;
int __attribute__((visibility("default"))) not_hidden;
```

__FUNCTION__

DCXX supports a "predefined" identifier named `__FUNCTION__`. `__FUNCTION__` is similar to the C pre-processor identifier `__LINE__` except that it is processed during the compilation-phase instead of the preprocessing-phase.

During compilation, `__FUNCTION__` is replaced by a string constant that contains the name of the current function.

If `__FUNCTION__` identifier occurs outside of function scope, it is replaced with the empty string, `""`, and a warning is issued.

`__FUNCTION__` is different from the ANSI-defined `__func__` identifier. `__func__` is defined to be a single instance within a function of locally declared array of characters which is initialized to the string constant. Thus, every occurrence of `__func__` is guaranteed to address the same array within the function. Since `__FUNCTION__` is simply directly replaced with a string constant, each occurrence could potentially address different versions of the string.

The __rent and __norent qualifiers

`extern` or `static` storage class variables my be qualified with either the `__rent` and `__norent` keyword. This allows for fine control over the location of any specific variables.

By default all extern and static variables will be placed in the Pseudo Register Vector, the **PRV**, and could require a costly run-time initialization. If a variable is `const` and the initialization is appropriate, the variable need not reside in the **PRV** and the initialization can occur at compile time, saving run-time startup costs.

For example, the following declares an array of integers that are never written to, and thus can be initialized at compile-time instead of run-time. Application of the `__norent` keyword will ensure this array is not allocated in the **PRV**:

```
__norent const int array[10] = { 1, 2, 3, 4, 5, 6 };
```

Note that if an element of the array is modified at run-time, the program will no longer be re-entrant. Because of the `const` keyword; the compiler will emit a warning message if it discovers a potential modification of the array.

Although the Systems/C++ runtime is "rent only," the flexibility to define `__norent` data can improve runtime performance.

__bit_sizeof and __bit_offsetof operators

DCXX supports two operators to determine the bit size and offset of class or structure members:

```
__bit_sizeof expr
__bit_offsetof(type, field)
```

They are meant to be used on bit fields, but work on regular fields as well. The *expr* must be a class or structure member reference, either the `.` or `->` operator. The result of `__bit_offsetof` is measured from the beginning of the structure. The result can be used in constant contexts, for example to define `enum` values or array dimensions.

Example usage:

```
int bits = __bit_sizeof ((struct foo *)0)->field;
int offset = __bit_offsetof(struct foo, field);
```

Inline Assembly language support

DCXX supports the ANSI standard inline assembler source feature. This feature may be used within a function, or in external file scope. It specifies assembly source that will be copied, verbatim, to the generated assembly source deck. However, this approach does not take advantage of the more advanced Systems/C++ inline assembly language features.

DCXX also supports the same robust inline assembly language feature as the Systems/C compiler, **DCC**. This feature may be used within a function or in external file scope. It specifies assembly source that will be copied, verbatim, to the generated assembly source deck.

In support of this feature, **DCXX** also provides register-based automatic variables:

A register-based variable is a variable of integral or pointer type, with the `__register()` keyword added to its type declaration. The `__register()` keyword is treated as a storage class qualifier by the compiler.

`__`register(*nn*) — Type specifier.

Specifies that the datum is to be located specifically in register #*nn*.

References to the datum will use the specified register.

If this specifier occurs at file scope, the register is reserved for all functions which follow. This causes the compiler to reserve the register and not use it for the remaining functions. References to the declared datum will use the associated register.

The `extern` specifier may not be used on a `__register` declaration.

In function scope, within the scope of the datum's declaration, the register is not available for use by the compiler. Care must be taken to not use registers normally used by the compiler. These registers include registers 0, 1, 12, 13, 14 and 15, or the registers specified in the *–fframe_base* or *–fcode_base* options. The compiler does not examine the inlined assembly source for uses of these registers. The compiler will issue a warning for declarations that use a register which is reserved across a whole function (such as the code base and the frame base), but will not warn for declarations using registers that are used for function calls.

For example, the following section of code declares a `void *` pointer which is associated with register #5:

```
{
   __register (5) void *r5;
   r5 = 0; /* Put a 0 in register #5 */
}
/* r5 is now available for use again by the compiler. */
```

`__`asm [*n*] ... — Inline assembly source

```
__asm [n]
{
   Any text
}
```

The `__asm` keyword, optionally followed by an integral constant, defines the beginning of assembly language text which will be copied verbatim to the generated assembly language source. This statement may appear within a function, or in file scope. Note that the text must follow the ANSI C++ preprocessor tokenizing rules, otherwise, there are no restrictions on what the text contains. The text may be any number of lines. To use `__asm` statements effectively in *#define* macros and other instances involving the C++ preprocessor, the compiler searches the specified text for escape sequences, and replaces them with certain characters. An escape sequence begins with a single backslash character, "\". The recognized sequences are:

Escape sequence	Replacement
`\c`	continuation
`\n`	new-line
`\p`	pound sign
`\s`	space
`\C`	section name
`\q`	single-quote
`\Q`	double-quote

Any character following the backslash which is not recognized is copied directly. So, to produce the backslash character, one would use \\ in the assembly source.

`\c` and `\C` are special cases, in that the character isn't directly replaced. `\c` causes spaces to be added to the source line up to column 72, where a '`*`' will be placed. That is, `\c` is used to indicate this is an assembly continuation line. `\C` expands into the current code section name for this compilation.

The optional integral constant declares how many bytes the inline assembly source will generate. The compiler uses the value to determine if the code will fit into an existing 4K code region, or if it should be moved to a subsequent region. If the value isn't specified, the compiler counts the number of source lines and multiplies that by 4 to arrive at a reasonable heuristic. The value doesn't need to be exact; but if addressability problems become apparent during assembly of the generated source, this value should be increased appropriately.

Combined with the `__register()` keyword, `__asm` provides a powerful mechanism for generating direct assembly language code and interfacing with C++ variables.

For example, to invoke the **GETMAIN** macro to acquire main memory storage, the following block of C++ code could be used:

```
{
   void *getmain_result;
   unsigned long size;

   size = nnn; /* Size of the desired allocation */
```

```
    {
       __register(1) unsigned long r1;
       __register(2) unsigned long r2;

       /* Need to declare R0 because GETMAIN uses it */
       /* We don't want the compiler to grab it */
       __register(0) unsigned long r0;

       r1 = 0xf0000000;     /* Put X'F0000000' in R1 */
       r2 = size;           /* Store desired size in R2 */
       /* Call GETMAIN - the macro expands to ASM code */
       /*  that is 8 bytes long. */
       __asm 8 {
          GETMAIN RU,LV=(2),LOC=BELOW
       }
       /* Put the result of GETMAIN into the C variable */
       /*   'getmain_result' */
       getmain_result = (void *)r1;
    }
}
```

The following example demonstrates use of the escape sequences within a *#define* macro. The macro defines a fast *strcpy()*-like macro which takes advantage of the string instructions available on some processors. The escape sequences \s and \n are required because the C++ preprocessor considers this one rather long source line. Thus, \n is used to add new-lines where appropriate in the assembly language source. Furthermore, the C++ preprocessor will remove unneeded white space (blanks or tabs) per the C++ syntax rules. Thus, \s is used to ensure that each line begins with a blank. If \s isn't used, the assembler would consider the instruction opcodes to be labels, which is not the intent.

```
#define fast_strcpy(dest, src) { \
     __register(0) r0; \
     __register(2) void *r2 = dest; \
     __register(3) void *r3 = src; \
     __asm 12 { \
         \s   SR    0,0 \n\
         \s   MVST  2,3 \n\
         \s   BO    *-4 \n\
     } }
```

__asm("...":output:input:clobber) — GCC-style inline assembly source

As of version 2.00, **DCXX** supports GCC-style inline assembly. If the __asm keyword is followed by a parenthesis, then **DCXX** recognizes the GCC-style syntax instead.

```
__asm("asm code",
        : output operands
        : input operands
        : clobber list);
```

The comma-separated operand and clobber lists are optional.

In the *asm code*, the same backslash ("\") escape codes are honored as in a regular `__asm { ... }` block. In addition, codes of the form "**%*n***" are substituted with the corresponding operand. *n* is an input or output operand number, starting at "`%0`". To put a "`%`" in your *asm code*, use two of them ("`%%`").

Each input or output operand uses the following syntax:

```
"constraint string" ( expression )
```

The constraint specifies how the "**%*n***" string will be substituted, and what semantic effect that will have on the expression. The expression provides the value that will be given to the assembly code, or an lvalue for where an output operand will be stored.

DCXX supports the following constraint strings:

`r`	General Purpose Register number
`d`	data (general purpose) register number, same as "`r`"
`a`	addressing register number (non-zero GPR)
`dp`	data regpair (even numbered GPR)
`f`	Floating Point Register number
`fp`	float regpair (the number of the first FPR in the pair)
`m`	memory address of the form "*ofs*`(`*index*`,`*base*`)`"
`Q`	memory address with no index reg, of the form "*ofs*`(`*base*`)`"
`I`	unsigned 8-bit integer literal
`J`	unsigned 12-bit integer literal
`K`	signed 16-bit integer literal
`i`	signed 32-bit integer literal
`0 ... 9`	matching constraint — use same register as corresponding operand

The constraint string for an output operand may also have some prefix characters:

`=` write-only output operand

`+` read-write output operand

`&` early clobber output operand

The default is as if "`=`" were specified, in which case the value of the register is copied into the destination after the *asm code* is executed. For a read-write operand, the value is copied into the register before the *asm code* is executed, and then copied from the register to the destination afterwards, so that code can modify the value in a register.

Early clobber ("`&`") means that this output operand may be written within *asm code* before all of the input operands have been read. Without "`&`", **DCXX** may chose to use the same register for one of the input operands as for a write-only output operand, but "`&`" indicates they must use two separate registers.

The clobber list is a comma-separated list of strings indicating resources that are modified by the *asm code*, and which **DCXX** needs to be aware of. The values may be:

`memory` *asm code* writes to memory (this is assumed if one of the output operands has the constraint "`m`" or "`Q`"

`cc` *asm code* modifies the condition code in the PSW register

`r`*n* *asm code* modifies GPR *n*

`f`*n* *asm code* modifies FPR *n*

Clobbered registers may also have the prefix "`&`", which means they are clobbered before all of the input operands are read. Otherwise, **DCXX** may use a clobbered register for an input operand.

For example, the following code modifies a variable using a pointer:

```
int i = 1;
__asm(" ST %1,0(%0)" : : "a"(&i),"r"(123) : "memory");
printf("i is %d", i);   /* prints "i is 123" */
```

Note that the `%0` operand is an input operand, because from **DCXX**'s perspective, it is just providing a value to the *asm code*, that value just happens to be an address that will be written to. Without the "`memory`" clobber string, the compiler might use a cached value for `i` in the `printf` call, instead of reading the value from memory again.

To accomplish the same thing using "`m`" (memory) output operand:

```
int i = 1;
__asm(" ST %1,%0" : "m"(i) : "r"(123));
printf("i is %d", i);   /* prints "i is 123" */
```

You can specify specific registers in your clobber list as an alternative to reserving them with `__register(n)` variables, so that the compiler knows it can't count on the value being the same after *asm code*. For example:

```
__asm("  invocation of macro that uses R3"::"r3");
```

Is roughly the same as:

```
{ __register(3) int r3;   /* reserve R3 */
  __asm {
      invocation of macro that uses R3
  }
}
```

Note that if you clobber a register which is reserved by the compiler (such as the code base or frame base register), the execution will fail because the compiler will still use the reserved register — **DCXX** relies on the reserved registers holding their assigned values.

A matching constraint is typically used on an input operand to match an output operand. The input operand then provides a specific value to be placed in the output operand's register before the *asm code* is executed. For example, this contrived code adds `i` and a constant `11`, then stores the result in `j`:

```
int i,j;
/* ... */
__asm(" LA %0,%2" : "=r"(j) : "0"(i),"J"(11));
```

Note that the input operand for `i` has its own operand number (`%1`), even though it uses the same register as `%0`. That is why the constant literal integer `11` is identified as `%2`.

The `GETMAIN` example above could be expressed more simply using GCC-style inline assembler:

```
void *getmain_result;
unsigned long size;

size = nnn; /* Size of the desired allocation */

__asm(" LR 1,%1\n\
        GETMAIN RU,LV=(%2),LOC=BELOW\n\
        LR %0,1"
      : "=r"(getmain_result)
      : "r"(0xf0000000) /* the value to put in R1 */
      : "r0", "&r1", "cc");
```

The `@` operator

The `@` operator is a C++ language extension that produces the address of its operand expression, similar to the normal C++ language `&` operator.

However, while `&` only operates on lvalue expressions, `@` can operate on any expression.

In the contexts where `&` is valid, `@` is the same as `&`.

If the expression operand to `@` is an rvalue expression, `@` will copy the expression to an automatic-storage temporary and use the address of the temporary.

Furthermore, if the operand to `@` is an array, the result is different than `&`. `&` applied to an array produces the address of the first element of the array. However, `@` applied to an array produces the address of an automatic temporary which contains the address of the array. Note that string constants are arrays, so that `@"STRING"` does not produce the address of the string constant, but the address of a temporary which points to the string constant.

The `@` operator can be used anywhere within the body of a function. Because it creates an automatic temporary in some situations, the `@` operator cannot be used at file scope (e.g. cannot be used in file-scope or `static` initializations.)

The `@` operator can be employed to assist in parameter passing when invoking non-C language functions (e.g. assembly functions) that expect pass-by-reference parameters instead of the typical C pass-by-value parmeters.

__asm__("name") qualifier on function declarations

The GNU extension `__asm__("name")` can be applied to function declarations to alter the name used in the generated object file.

The specification appears after the parameter section of a function declaration. For example:

```
extern int func() __asm__("FUNC");
```

will cause the name `FUNC` to be used when the `func` function is referenced or defined.

This is equivalent to the `#pragma map` facility for mapping function names.

__builtin functions

The compiler supports several builtin functions that are pre-declared and can be referenced directly in the source.

__builtin_alloca

`void __builtin_alloca(size_t)` Used to invoke allocate additional stack space.

__builtin_bswap16

`uint16_t __builtin_bswap16(uint16_t)` Performs byte swapping on a 2-byte value. This will use machine instructions when allowed.

__builtin_bswap32

`uint32_t __builtin_bswap32(uint32_t)` Performs byte swapping on a 4-byte value. This will use machine instructions when allowed.

__builtin_bswap64

`uint64_t __builtin_bswap64(uint64_t)` Performs byte swapping on a 8-byte value. This will use machine instructions when allowed.

__builtin_prefetch

`void __builtin_prefetch(const void *addr, ...)` Indicates the given address will be referenced to reduce cache latency. When the architecture level supports prefetch instructions they will be generated to indicate the data should be made available for a subsequent reference.

ıaddr provides the address of the memory.

`__builtin_prefetch` also accepts two optional arguments, a compile-time constant integer ırw that indicates read or write access, and compile-time constant integer ılocality that indicates temporal locality. ırw can be the value `0` to indicate preparation for read access, `1` for write access. The default is `0`. ılocality can be the value `0`, `1`, `2` or `3`. A value of `3` indicates the memory has a high degree of temporal locality (will be referenced soon) and should be kept in all levels of the cache.

Data prefetching does not cause a fault if the specified ıaddr is invalid; but the expression itself must be valid to be evaluated.

If the target architecture level does not support the prefetch instructions, the ıaddr expression is still evaluated to handle any potential side effects.

__builtin_memcpy

`void *__builtin_memcpy(void *dest, const void *src, size_t len)` Implements the C standard memcpy function.

__builtin_memset

`void *__builtin_memset(void *dest, int val, size_t len)` Implements the C standard memset function.

__builtin_memcmp

`int __builtin_memcmp(const void *src1, const void *src2, size_t len)` Implements the C standard memcmp function.

__builtin_strcpy

`int __builtin_strcpy(char *dest, const void *src, size_t len)` Implements the C standard strcpy function.

__builtin_strlen

`size_t __builtin_strlen(const char *src)` Implements the C standard strlen function.

__builtin_strcmp

`int __builtin_strcmp(const char *src1, const char *src2)` Implements the C standard strcmp function.

__builtin_strcat

`char * __builtin_strcat(char *src1, const char *src2)` Implements the C standard strcat function.

__builtin_strchr

`char * __builtin_strchr(const char *src, int val)` Implements the C standard strchr function.

__builtin_strrchr

`char * __builtin_strrchr(const char *src, int val)` Implements the C standard strrchr function.

__builtin_strncat

`char * __builtin_strncat(char *dest, const char *src, size_t len)` Implements the C standard strncat function.

__builtin_strncmp

`char * __builtin_strncmp(const char *src1, const char *src2, size_t len)` Implements the C standard strncmp function.

__builtin_strncpy

`char * __builtin_strncpy(char *dest, const char *src, size_t len)` Implements the C standard strncmp function.

__builtin_strpbrk

`char * __builtin_strpbrk(const char *str, const char *src)` Implements the C standard strncmp function.

__builtin_fabs

`double __builtin_fabs(double)` Implements the C standard fabs function.

__builtin_fabsf

`float __builtin_fabs(float)` Implements the C standard fabsf function.

__builtin_fabsl

`long double __builtin_fabsl(long double)` Implements the C standard fabsl function.

__builtin_abs

`int __builtin_abs(int)` Implements the C standard abs function.

__builtin_labs

`long __builtin_labs(long)` Implements the C standard labs function.

__builtin_popcount

`int __builtin_popcount(unsigned int)` Returns the number of 1-bits in the parameter.

__builtin_popcountl

`int __builtin_popcountl(unsigned long)` Returns the number of 1-bits in the parameter.

__builtin_popcountll

`int __builtin_popcountll(unsigned long long)` Returns the number of 1-bits in the parameter.

__builtin_frexp

`double __builtin_frexp(double val, int *exp)` Implements the C standard frexp function.

__builtin_frexpf

`float __builtin_frexpf(float val, int *exp)` Implements the C standard frexpf function.

__builtin_frexpl

`long double __builtin_frexpl(long double val, int *exp)` Implements the C standard frexpl function.

__builtin_huge_val

`double __builtin_huge_val(void)` For BFP values, when *–fieee* is specified, this returns a positive IEEE Infinity. Otherwise, this returns the maximum HFP value.

__builtin_huge_valf

`float __builtin_huge_valf(void)` Similarly to `__builtin_huge_valf()` but returns a `float` value.

__builtin_huge_vall

`long double __builtin_huge_vall(void)` Similarly to `__builtin_huge_valf()` but returns a `long double` value.

__builtin_inf

`double __builtin_inf(void)` `__builtin_inf()` returns an IEEE `+Inf` value when the *–fieee* options is enabled. For HFP it returns the largest positive HFP value.

__builtin_inff

`float __builtin_inff(void)` Similarly to `__builtin_inf()` but returns a `float` value.

__builtin_infl

`long double __builtin_infl(void)` Similarly to `__builtin_inf()` but returns a `long double` value.

__builtin_nan

`double __builtin_nan(const char *)` This is an implementation of the ISO C99 function nan.

When the *–fieee* option is enabled, this returns an IEEE quiet `NaN` value. The character string can be used to represent a payload incorporated int the mantissa. In order for this to be a compile-time constant, the character string must be a compile-time constant. The character string is evaluated with the `strtoul` function, and thus the base of the character string can be specified by a leading `0` or leading `0x`. The value is truncated to fit into the IEEE mantissa.

For HFP values, __builtin_nan returns `0.0`.

__builtin_nanf

`float __builtin_nanf(const char *)` Similarly to `__builtin_nan()` but returns a `float` value.

__builtin_nanl

`long double __builtin_nanl(const char *)` Similarly to `__builtin_nan()` but returns a `long double` value.

__builtin_nans

`double __builtin_nans(const char *)` Similar to `__builtin_nan`, except that an IEEE mantissa is made a signaling NaN. The nans function is proposed by WG14 N965.

__builtin_nansf

`float __builtin_nansf(const char *)` Similarly to `__builtin_nans()` but returns a `float` value.

__builtin_nansl

`long double __builtin_nansl(const char *)` Similarly to `__builtin_nans()` but returns a `long double` value.

__atomic functions

The C++ compiler supports the same `__atomic` builtin functions as **gcc** does. These functions provide atomic access to shared memory, so that no intervening operations in other threads or tasks can produce an unpredictable result.

These functions take a `memorder` parameter, which indicates whether there should be a scheduling barrier (and inter-CPU serialization point) before loads and after stores. For read operations (`__atomic_load`) and write operations (`__atomic_store`, `__atomic_clear`), **DCXX** will emit the barriers so long as the `memorder` is not `__ATOMIC_RELAXED`. For read-modify-write operations, the strictest memory ordering (`__ATOMIC_SEQ_CST`) is assumed because they are implemented with the underlying COMPARE SWAP `CS` instruction, which is always serialized.

The `__atomic` functions are type-generic, one function name is used for all types. The variants with the suffix `_n` use or return the value directly, and must operate on regular integer or pointer types. The variants without the suffix work by pointer and can work on any types, including `struct`s. When the underlying data is not an integer or pointer, a call to a run-time function of the same name will be generated. The run-time functions are provided in our C library with the prefix `@@atmc`. They use a global lock, so they are not as efficient as the atomic operations that are supported by the underlying hardware (1/2/4/8 byte operations).

__atomic_load_n

```
type __atomic_load_n(type *src, int memorder)
```

Returns `*src` (read of `*src` is atomic).

__atomic_load

```
void __atomic_load(type *src, type *dst, int memorder)
```

Assigns `*dst = *src` (read of `*src` is atomic).

__atomic_store_n

```
void __atomic_store_n(type *dst, type src, int memorder)
```

Assigns `*dst = src` (write of `*dst` is atomic).

__atomic_store

```
void __atomic_store(type *dst, type *src, int memorder)
```

Assigns `*dst = *src` (write of `*dst` is atomic).

__atomic_exchange_n

```
type __atomic_exchange_n(type *dst, type src, int memorder)
```

Assigns `*dst = src`, and returns the original value of `*dst` (read and write of `*dst` is atomic).

__atomic_exchange

```
void __atomic_exchange(type *dst, type *src, type *ret, int memorder)
```

Assigns `*ret = *dst` then `*dst = *src`, as a single atomic operation (read and write of `*dst` is atomic).

__atomic_compare_exchange_n

```
bool __atomic_compare_exchange_n(type *dst, type *expected, type desired,
     bool weak, int success_memorder, int failure_memorder)
```

Evaluates `if (*dst == *expected) *dst = desired` as a single atomic operation, returning 1 if the assignment was performed (read and write of `*dst` is atomic). `weak` is ignored, but would indicate that the operation is allowed to intermittently fail (return 0 and not perform the assignment) even if the comparison is true.

__atomic_compare_exchange

```
bool __atomic_compare_exchange(type *dst, type *expected, type *desired,
     bool weak, int success_memorder, int failure_memorder)
```

Evaluates `if (*dst == *expected) *dst = *desired` as a single atomic operation, returning 1 if the assignment was performed (read and write of `*dst` is atomic). `weak` is ignored, but would indicate that the operation is allowed to intermittently fail (return 0 and not perform the assignment) even if the comparison is true.

__atomic_OP_fetch

```
type __atomic_add_fetch(type *dst, type val, int memorder)
type __atomic_sub_fetch(type *dst, type val, int memorder)
type __atomic_and_fetch(type *dst, type val, int memorder)
type __atomic_xor_fetch(type *dst, type val, int memorder)
type __atomic_or_fetch(type *dst, type val, int memorder)
type __atomic_nand_fetch(type *dst, type val, int memorder)
```

Evaluates `*dst = *dst OP val`, and then returns the result (read and write of `*dst` is atomic).

__atomic_fetch_OP

```
type __atomic_fetch_add(type *dst, type val, int memorder)
type __atomic_fetch_sub(type *dst, type val, int memorder)
type __atomic_fetch_and(type *dst, type val, int memorder)
type __atomic_fetch_xor(type *dst, type val, int memorder)
type __atomic_fetch_or(type *dst, type val, int memorder)
type __atomic_fetch_nand(type *dst, type val, int memorder)
```

Evaluates `*dst = *dst OP val`, and returns the original value in `*dst` from before the operation (read and write of `*dst` is atomic).

__atomic_test_and_set

```
bool __atomic_test_and_set(void *dst, int memorder)
```

Sets the byte at `*dst` to a non-zero value, and returns 1 if and only if the original value of `*dst` was already non-zero (read and write of `*dst` is atomic). This is less efficient than `__atomic_exchange_n` operating on a 32-bit integer, because the instruction set does not provide an atomic compare-and-swap instruction for 8-bit values.

__atomic_clear

```
void __atomic_clear(bool *dst, int memorder)
```

Assigns the byte at `*dst` to zero (write of `*dst` is atomic). This is less efficient than `__atomic_store_n` operating on a 32-bit integer, because the instruction set does not provide an atomic compare-and-swap instruction for 8-bit values.

__atomic_..._fence

```
void __atomic_thread_fence(int memorder)
void __atomic_signal_fence(int memorder)
```

These functions are identical and provide a barrier and synchronization.

__atomic_..._lock_free

```
bool __atomic_always_lock_free(size_t size, void *ptr)
bool __atomic_is_lock_free(size_t size, void *ptr)
```

These return 1 if atomic operations on types of the given size can be performed efficiently without locks, using hardware instructions. They always return 1 for sizes of 1/2/4/8 bytes. Returns 0 for other sizes, which use a global lock. The `ptr` argument is ignored.

#pragma compiler directives

DCXX supports several #pragma directives:

#pragma options(*opt[,opt]...*)

Specifies compile-time options in the C++ source code. A `#pragma options` must appear before any C++ source.

Options specified in the `#pragma options` are not reflected in the compiler listing. The listing displays the default and command-line option values.

If a `#pragma options` value conflicts with the option value specified on the command line, the compiler uses the command-line specified value.

Currently only the `#pragma options(RENT)` option is supported, all other options are ignored.

#pragma prolkey(*identifier*, "*key*")

Using `#pragma prolkey` allows the user to tailor certain function entry points by adding additional macro arguments.

`#pragma prolkey` specifies that the string "*key*" is to be appended to the keyword list for the prologue macro associated with the entry point named *identifier*. The string "*key*" will be copied verbatim and added to the end of the typical macro arguments for the entry point.

The `#pragma prolkey` directive must appear before the function definition.

#pragma epilkey(*identifier*, "*key*")

`#pragma epilkey` specifies that the string "*key*" is to be appended to the keyword list for the epilogue macro associated with the entry point named *identifier*. The string "*key*" will be copied verbatim and placed on the epilogue macro invocation for the entry point.

Using `#pragma epilkey` allows the user to tailor certain function epilogues by adding additional macro arguments.

The `#pragma epilkey` directive must appear before the function definition.

#pragma map(*identifier*, "*name*")

`#pragma map` specifies that external references to functions or data named *identifier* are to be replaced with the string specified in "*name*." The "*name*" value becomes the value for any **ALIAS** statements emitted in the generated assembly language source.

The `#pragma map` directive may appear anywhere in the compilation.

The *identifier* specified in a `#pragma map` may have C or C++ linkage. If the *identifier* has C++ linkage, then the *identifier* can not specify an overloaded function.

#pragma weakalias(*identifier*, "*name*")

`#pragma weakalias` specifies that a weak definition with of a symbol named *name* should be generated which has the same value as the variable identified by *identifier*.

The `#pragma weakalias` directive may appear anywhere in the compilation.

Note that if the *–fnoalias_stmts* is enabled, `#pragma weakalias` is not supported.

`#pragma weakalias` works on most platforms for both global functions and global variables. However, for re-entrant data based off of the PRV, it is impossible to make a weak alias. This is due to limitations in the object formats' treatment of Q-cons – it is impossible to make two Q-cons with different names but the same address.

#pragma noinline(*identifier*)

Tells the optimizer not to inline the named function even if other heuristics suggest that it could be inlined. This can be useful for certain constructs — such as `__asm` blocks — which are not amenable to being copied.

#pragma error "*text*"

`#pragma error "`*text*`"` causes the compiler to generate an error message. The error message will include the specified *text*.

#pragma warning "*text*"

`#pragma warning "`*text*`"` causes the compiler to generate a warming message. The warning message will include the specified *text*.

#pragma eject

`#pragma eject` causes the listing to move to a new page.

#pragma page(*n*)

`#pragma page(`*n*`)` causes the listing to move forward *n* pages. *n* is optional, and if not provided causes the compiler to move forward one page.

#pragma pagesize(*n*)

`#pragma pagesize(`*n*`)` sets the number of lines on subsequent pages in the listing to *n*. *n* should not be less than 20.

#pragma showinc

`#pragma showinc` causes the compiler to include source lines from `#include` files in the listing. This can be used to selectively add some `#include` source lines in the listing while leaving out others. Use `#pragma noshowinc` to cause source lines from `#include` files to be skipped in the listing.

#pragma noshowinc

`#pragma noshowinc` causes the compiler to not include source lines from `#include` files in the generated listing. This can be used to selectively skip some `#include` source lines. Use `#pragma showinc` to re-enable listing of `#include` source lines.

#pragma pack(*n*)

`#pragma pack` specifies the maximum structure element alignment for structure type *declarations*.

Normally, the C++ compiler aligns elements in a structure based on their natural alignment. `#pragma pack` can be used to impose a maximum alignment, so that no element of a structure will have an alignment greater than the one specified in the `#pragma pack`. Elements which have natural alignments smaller than specified in a `#pragma pack` continue to be aligned on their natural boundary.

`#pragma pack` can specify, 1, 2, 4, 8 and 16 byte maximum alignment values.

The values specified via `#pragma pack` are stacked, a `#pragma pack(reset)` can be used to restore the previous value. When the *–fc370* option is not specified, **DCXX** also recognizes `#pragma pack(pop)` as equivalent to `#pragma pack(reset)`.

There are alternate keywords which can be employed instead of numeric values. `#pragma pack()` selects default packing. `#pragma pack(full)` is equivalent to `#pragma pack(4)`. `#pragma pack(twobyte)` is equivalent to `#pragma pack(2)` and `#pragma pack(packed)` is equivalent to `#pragma pack(1)`.

Specifying no parameter in a `#pragma pack` is equivalent to `#pragma pack(4)`.

If *–fztpf* or *–flinux* was specified on the commandline then the structures produced by **DCXX** are compatible with **g++** for all of the `#pragma pack` settings. If *–fc370* is specified then the structure layout is compatible with IBM's compilers for Language Environment. An additional setting of `#pragma pack(le)` is available which causes structures to be laid out to be compatible with Language Environment, even if compiling for a different platform, such as z/TPF.

#pragma weak(*identifier*)

`#pragma weak` indicates that the identifier is either a weak reference or, when *–flinux* is specified, a weak definition.

For Systems/C++ (*–flinux* not specified) programs, only weak references are supported. Weak references apply to either functions or non-reentrant data. A `#pragma weak` applied to reentrant data has no effect. A weak reference generates a `WXTRN` reference in the resulting assembly source, instead of the default `EXTRN` reference. For example, the following code declares `weak_func()` as being a weak external function. It then tests to see if `weak_func()` is defined before calling it:

```
#pragma weak(weak_func)

void weak_func(void);

main()
{
   /* If weak_func is defined, call it. */
   if(weak_func) {
      weak_func();
   }
}
```

When *–flinux* is specified, a `#pragma weak` may apply to either functions or data, and may be applicable to either references or definitions. The Linux linker will allow multiple weak definitions of the same function or data without complaint.

#pragma ident "*str*"

`#pragma ident "`*str*`"` instructs the compiler to add *str* to the generated object as data. It will not necessarily be loaded into memory at run time, but it will be in the object. This feature is commonly used for versioning and copyright information. It is an alternative to the construct

```
static const char ident[] = "@(#)$Id: prog.c,v 1.42 $";
```

but it is guaranteed to never elide the string as unreferenced.

#pragma comment(user, "*str*")

`#pragma comment(user, "`*str*`")` is equivalent to `#pragma ident "`*str*`"`.

#pragma enum(*enum_size*)

`#pragma enum` defines the amount of storage enumeration values consume in IBM compatibility mode.

The *enum_size* value can be one of `SMALL`, `INT`, `1`, `2`, `4`, `pop` or `reset`.

Enumeration size settings are stacked. The enumeration size can be restored to its previous value using the `pop` or `reset` option.

`SMALL` is the default enumeration packing rules supported in the IBM compiler. That is, enumeration values are packed to the smallest amount of storage that can contain the range of the enumeration values.

`INT` indicates that the size of the enumeration will be 4 bytes.

`1`, `2` and `4` indicate that the size of the enumeration will be the number of bytes specified.

#pragma csect(*section*, "*name*")

Specifies the name to use for a particular section. The types of allowed sections are **CODE**, **STATIC**, and **TEST**.

When compiling in IBM compatibility mode (*–fc370* is enabled), this pragma operates identically to the IBM C `#pragma csect` pragma. Otherwise, this pragma can be used to set the section name value similarly to the *–fname* compiler option. Setting the **CODE** section name to *name* is equivalent to specifying *–fsname=name* on the compiler command line.

This pragma is useful for specifying the section name directly in the source file instead of via JCL or some other mechanism.

Only one `#pragma csect` can be specified for a particular *section*. A `#pragma csect` specification overrides any *–fsname* option specified on the compiler command line.

Note that `#pragma csect(TEST, "`*name*`")` is only meaningful when compiling in IBM compatibility mode (when the *–fc370* is specified.)

extern "ALIGN4"

The `extern "ALIGN4"` linkage can be applied to either function definitions or declarations to alter this default 8-byte alignment when *–mlp64* is specified.

If a `extern "ALIGN4"` applies to a function then calls to the function will assume parameters are aligned on 4-byte (fullword) boundaries. Note that `ALIGN4` linkage

does not affect the size of the paramaters, a `long` or pointer value will continue to be 8 bytes in size. It will simply be aligned on a 4-byte boundary instead of an 8-byte boundary.

For example, in the following declaration:

```
extern "ALIGN4" {
void func(int size, void * __ptr31 starting_address) ;
}
```

the `func` function's arguments will be at offset zero (`size`) and offset 4 (`starting_address`.) Also notice that in this example the `__ptr31` qualifier was specified to indicate that the `void *` pointer is a 4-byte pointer.

extern "OS"

External functions and function pointers can be declared with `extern "OS"` linkage to indicate the function is to be called with the basic linkage convention used by the operating system.

Parameters to these `extern "OS"` functions that are not pointers are passed as addresses to temporary copies of the actual parameters, so that the function receives pointers to the actual parameters. If the parameter is not a pointer, and the type of the parameter is less than 4 bytes in size; it is promoted to an `int` type before making the copy. Also, the last argument's "VL-bit" will be set. That is, for the last pointer argument, the pointer will be ORed with `0x80000000`.

The compiler assumes that any return value from an `extern "OS"` function is in register 15. Register 0 will be set to zero before the function call. Also, before calling the `extern "OS"` function, the first 12 bytes of the local save area are copied into a temporary location and restored on return.

extern "PLI"

Similar to `extern "OS"`, `extern "PLI"` can be used to define functions and functions pointers that represent PLI-style linkage.

Parameters to these `extern "OS"` functions that are not pointers are passed as addresses to temporary copies of the actual parameters, so that the function receives pointers to the actual parameters. If the parameter is not a pointer, and the type of the parameter is less than 4 bytes in size; it is promoted to an `int` type before making the copy. Also, the last argument's "VL-bit" will be set. That is, for the last pointer argument, the value will be ORed with `0x80000000`.

Also, `extern "PLI"` functions don't return any return values in the normal mechanism. For these functions, an extra argument is added which is a pointer to a temporary location to contain the return value.

64-bit arithmetic — long long

DCXX supports both the `long long` and `unsigned long long` data types. `long long` and `unsigned long long` are 64 bits (8 bytes), with a 4-byte, or fullword alignment.

Functions that return a `long long` or `unsigned long long` datum return the value in register 15 and register 0. The most significant bits of the value are in register 15.

DCXX also supports extended `long long` integral constants. These are specified with the `ULL` and `LL` suffixes.

C preprocessor extensions

DCXX supports several common C preprocessor extensions.

#warning

A `#warning` preprocessor control line causes a warning message to be generated. Any text following the `#warning` is provided in the generated message.

For example the following #warning control lines:

```
#warning "This is a warning"

#warning

#warning a string
```

will cause the following diagnostics to be generated:

```
cxx: file line #: Warning #1048: #warning "this is a warning"

cxx: file line #: Warning #1048: #warning

cxx: file line #: Warning #1048: #warning a string
```

#include_next

`#include_next` is intended to "skip" in the *–I* search list when searching for `#include` files. `#include_next` indicates that the search for a `#include` file should begin at the next element in the *–I* search list from wherever the current file was located.

If the current file was specified using an absolute path name, then `#include_next` is treated as `#include`. If the current source is the primary source file, `#include_next` is treated as `#include` and a warning diagnostic is generated.

#ident

`#ident "`*str*`"` is simply a shorter form of `#pragma ident "`*str*`"`. It is used to put a comment in the generated object code, such as a version or copyright message.

Remote function pointers

DCXX provides a remote function pointer facility, that can be used to build programs that invoke functions in other (dynamically loaded) load modules on z/OS.

When the *–ffpremote* option is enabled, a function call that is accomplished through a (remote) function pointer saves the current PRV base in the local frame, loads a new PRV value from the function pointer, then loads the actual function address from the function pointer and branches to the function.

Thus, a remote function address is actually a pointer to a container that contains the new PRV base, and the actual function address. It is not actually the address of the code to branch to.

A remote function pointer container is generated when the address of a function is taken, or when a function pointer value is converted to a `__remote` function pointer.

The compiler generated assembly code employs the `DCCSTPRV` macro to indicate that the PRV value should be saved at the specified location. There is a `DCCSTPRV` macro that is provided for use with the Systems/C runtime, and it can be altered to accomodate any particular runtime environment. The *–fstprv=NAME* option can be used to cause the compiler to invoke a different macro.

When *–ffpremote* is not enabled, the `__remote` keyword can indicate a remote function pointer. Similarly, the `__local` keyword can indicate that the give function pointer is "direct", that it is local to the load module of the caller and does not need a unique PRV. Example usage of `__remote` and `__local` keywords:

```
typedef void __remote (*remote_fp)(void);
typedef void __local (*local_fp)(void);
```

Note that, while a remote function pointer can be converted to a local function pointer, it is not advisable. Invoking that function pointer would not switch the PRVs to the remote load module's PRV, and the invoked function would likely fail mysteriously (or catastrophically) as a result. The compiler generates a warning for this situation.

Special "built-in" implementations for common C library functions.

DCXX provides built-in implementations for some of the more common *C* library functions. These built-in functions are used when the `<string.h>` system header file is included. These include:

memcpy()
memset()
memcmp()
memchr()
strcpy()
strlen()
strcmp()
strcat()
strchr()
strncat()
strncmp()
strncpy()
strrchr()
strpbrk()

`#include <string.h>` to take advantage of the built-in versions of these functions.

Programming for z/Architecture

Systems/C++ supports programming for the new z/Architecture machines, supporting the new z/Architecture instructions and 64-bit addressing mode.

The compiler can take advantage of z/Architecture instructions when either 32-bit or 64-bit code generation is selected, using the *–march=z* option. The specification of *–mlp64* implies *–march=z*.

When z/Architecture mode is enabled, Systems/C will generate z/Architecture instructions.

z/Architecture instructions

When the *–march=z* option is enabled, Systems/C uses the newer z/Architecture instructions. This provides for 64-bit programming when *–mlp64* is specified and offers other improvements for 32-bit programs when *–milp32* is specified.

When *–mlp64* is specified, values retained in registers typically use the complete 64-bit register. This allows for a seamless translation between the `int` and pointer types, supporting existing, although not recommended, C practice.

64-bit z/Architecture programming model

When the *–mlp64* option is enabled, Systems/C++ generates z/Architecture instructions, enabling 64-bit addressing. In this mode, `long` and pointer data is 64-bits wide, and are aligned on a 64-bit boundary, the natural alignment for these types on the z/Architecture.

This size and alignment for `long` and pointer data is also known as the "LP64" programming model. The LP64 programming model is currently used on the most popular UNIX, and Linux 64-bit implementations, maximizing portability with these platforms.

For example, the following structure would be 16 bytes in size, and would be aligned on a 8 byte boundary:

```
struct big_struct {
long long_field;   /* 8 bytes long */
void *ptr_field;   /* 8 bytes long */
}
```

In non-z/Architecture mode, this structure would only be 8 bytes long, and aligned on a 4-byte boundary.

It is important to note that the `long long` data types are simply treated as equivalent to the `long` data types. Thus, in z/Architecture mode, the `long long` data types are also aligned on 8-byte boundaries.

Parameter passing and return values.

When *–mlp64* is specified, and *–flinux* or *–fc370* is not specified, Systems/C continues to use a parameter passing linkage similar to the typical OS/390 linkage. That is, register `R1` points to the parameter block.

In 64-bit mode (*–mlp64* is specified), Systems/C++ aligns parameters on natural register boundaries. That is, parameters are aligned on 8 byte (double word) boundaries. Integral values which are smaller than 8 bytes are right-justified in the 8-byte field.

For example, in calling the function with this prototype:

```
void func(char a, int b, void *c);
```

The value for the first parameter, `a`, would be at offset 7, bytes 0-7 would be cleared. The value for `b`, would be at offset 12, with bytes 8-11 cleared. And, the value for the parameter `c` would be at offset 16, using a full 8 bytes.

The `extern "ALIGN4"` linkage can be applied to either function definitions or declarations to alter this default 8-byte alignment. If a `extern "ALIGN4"` applies to a function then calls to the function will assume parameters are aligned on 4-byte (fullword) boundaries. Note that `ALIGN4` linkage does not affect the size of the paramaters, a `long` or pointer value will continue to be 8 bytes in size. It will simply be aligned on a 4-byte boundary instead of an 8-byte boundary.

Return values from functions are also affected by the *–mlp64* option. When returning values smaller than a 64-bit register, the value will be promoted to completely fill the register. Thus, functions that are undeclared, but return pointer values will continue to work as expected. Although, this is certainly not recommended for portable programs. For example, the following code will operate correctly:

```
 /* note - this function does not define a return type, */
 /*        and thus is assumed by the compiler to return */
 /*        'int' */
undeclared_pointer_return()
{
  static char array[20];
  return array;
}

void call_func(void)
{
   char *ptr_value;

      /* The compiler will generate a warning on this */
      /*  statement, regarding the conversion of the  */
      /*  'int' integral type to a pointer, but the   */
      /*  correct pointer value will be assigned.     */
   ptr_value = undeclared_pointer_return();
}
```

This approach allows older C code to remain compatible with the newer z/Architecture system.

AMODE and address calculations

It is important to recognize that the LP64 model does not require a 64-bit addressing mode. It simply indicates that pointers and `long` data can contain 64-bit values. Systems/C supports these values even when the `AMODE` is not 64-bits. This allows 64-bit addresses/data to be manipulated by 31-bit programs.

Normally, when *–mlp64* is specified, Systems/C++ assumes the `AMODE` is 64. With this assumption, Systems/C++ can generate `LOAD-ADDRESS` instructions for address calculations. However, if the *–famode* option is used to specify an addressing mode other than 64, Systems/C++ will generate z/Architecture code that can be used in any runtime `AMODE`. In this mode of operation, Systems/C++ will not use a LOAD-ADDRESS (`LA`) instruction to perform address calculations, instead using arithmetic instructions to perform these calculation. This allows the code to properly execute, and retain complete 64-bit addresses in any addressing mode.

Also when *–milp32* is specified, pointer arithmetic on `__ptr64` qualifier pointers will not use the LOAD-ADDRESS instruction, instead using other instructions to perform the necessary operation. This allows pointer arithmetic on `__ptr64` qualified pointers to properly operate in any environment.

__ptr64 qualifier

Pointers may be qualified with the `__ptr64` qualifier, which indicates the pointer contains 64-bit addresses. This is most useful when *–milp32* is specified, as when *–mlp64* is specified, normal pointers contain 64-bit address. `__ptr64` qualified pointers are 8 bytes long, and aligned on 8-byte boundaries. These pointers can be manipulated and used even when *–milp32* is specified.

z/Architecture instructions will be used for loading, storing and manipulating `__ptr64`-qualified pointers, regardless of the *–mlp64* or *–milp32* setting.

When *–milp32* is specified, pointer arithmetic performed on `__ptr64` qualified pointers is calculated using z/Architecture arithmetic instructions. When *–mlp64* is specified, `__ptr64` qualified pointers are treated as normal pointers.

`__ptr64` qualified pointers can be used in 31-bit code to retain and manipulate values passed to/from z/Architecture routines.

For example, in the following routine, the variable `big_pointer` is acquired from another function (possibly an assembler function) and then incremented. This could could appear in any Systems/C function, regardless of the *–mlp64* or *–ilp32* setting:

```
 /* acquire_ptr() returns a 64-bit address */
extern char * __ptr64 acquire_ptr();

char * __ptr64 big_pointer;

big_pointer = acquire_ptr();
big_pointer += 10;   /* increment pointer by 10 bytes */
```

When *–milp32* is specified, dereferencing a `__ptr64` qualified pointer will cause the compiler to generate a warning, indicating that a potential 64-bit address is being deferenced when the `AMODE` could be something other than 64-bits.

Also, the `__ptr64` qualifier can be used in parameter passing, when invoking a z/Architecture module from 31-bit code.

In the following example, `BIG` is passed a 64-bit `long long` value for the size of a data area, and a 64-bit pointer. When calling from 31-bit code, the compiler will automatically promote the values appropriately:

```
void BIG(long long, char * __ptr64);

void
func31(void)
{
   int size;
   char *ptr;

   size = 100;
   ptr = malloc(size); /* allocate 100 bytes */

   /* Invoke the z/Architecture "BIG" */
   /* function passing the size and a pointer */
   /* to the allocated space. */
   BIG(size, ptr);
}
```

The source for `BIG`, compiled with the *–mlp64* option enabled might look similar to this:

```
#pragma prolkey(BIG,"DCALL=YES")
void BIG(long long size, char * __ptr64 ptr)
{
  long i;
  for(i=0;i<size;i++) {
*ptr = 0;
  }
}
```

Note that is declared to be a Systems/C Direct-CALL (DCALL) function, to be properly invoked from a 31-bit environment.

__ptr31 qualifier

As with the `__ptr64` qualifier, pointers may be qualified with the `__ptr31` qualifier. Such pointers are 4 bytes long and aligned on 4-byte boundaries.

This allows for defining and referencing 31-bit addresses, even when the `AMODE` is 64.

For example, the following structure defines an integer, followed by a 31-bit address:

```
struct example31 {
   int integer_field;
   char * __ptr31 pointer_field;
};
```

This can be quite useful for accessing 31-bit data structures when *–mlp64* is specified.

When *–mlp64* is specified Systems/C++ will automatically convert `__ptr31` qualified addesses into 64-bit addresses when the pointer is dereferenced.

Similarly, any 64-bit addresses will be truncated when stored into `__ptr31` qualified pointers.

The `__ptr31` qualifier can also be useful when invoking z/Architecture code from ESA code, and passing 31-bit pointers. For example, in the following, the function ENTRY is a Systems/C DCALL function, which is compiled with the *–mlp64* option enabled:

```
extern "ALIGN4" {
#pragma prolkey(ENTRY,"DCALL=YES")

void
ENTRY(int size, void * __ptr31 starting_address)
{
  int i;
      /* zero-out 'size' bytes */
  for(i=0;i<size;i++) {
     *starting_address = 0;
  }
}

}
```

The parameter `starting_address` is passed as a 31-bit pointer and can be readily used by the z/Architecture function. The compiler will automatically promote the 31-bit pointer to its complete 64-bit value when it is dereferenced. Note also that `ALIGN4` linkage was applied to `ENTRY` so that it could be invoked from a 31-bit environment.

Systems/C++ z/Architecture library

When neither *–flinux* or *–fc370* are specified, the resulting program is intended to be linked with the Systems/C++ and Systems/C z/Architecture library. These libraries completely support running programs in z/Architecture mode, with all data, including stack, heap and re-entrant data, being loaded above the 4-gigabyte "bar."

For more particular details regarding the Systems/C++ and Systems/C z/Architecture libraries, see the *Systems/C++ Library* and *Systems/C Library* manuals.

Linking with the Systems/C++ and Systems/C z/Architecture libraries is only slightly different from the normal link process. All that needs to be done is specification of the alternate library.

Systems/C provides a reentrant and non-reentrant z/Architecture libraries. On cross-platform hosts, these objects are in the `objs_rent_z` and `objs_norent_z` directories. On OS/390 and z/OS, these are in the `LIBCRZ` and `LIBCNZ` PDSes. To use the Systems/C z/Architecture library, simply specify these directories/PDSs in place of the non-zArchitecture versions.

For example, JCL to execute the `PLINK` pre-linker with the Systems/C z/Architecture reentrant library would be similar to the following:

```
...
//PLINK EXEC PGM=PLINK
//STEPLIB DD DSN=Systems/C load library,DISP=SHR
//STDERR  DD SYSOUT=A
//STDOUT  DD SYSOUT=A
//SYSLIB  DD DSN=DIGNUS.LIBCRZ.OBJ,DISP=SHR
//INDD DD DSN=mypds,DISP=SHR
//SYSIN DD *
 INCLUDE INDD(PROG)
//SYSMOD DD DSN=myoutput.obj,DISP=NEW
```

The same command on a UNIX or Windows platform might be:

```
plink -omyoutput.obj prog.obj "-SC:\sysc\objs_rent_z\&M"
```

assuming Systems/C was installed in the `C:\sysc` directory.

Programming for OpenEdition

Systems/C++ supports creating OpenEdition programs which are executed from the Hiearchical File System (HFS.) This includes 31-bit and 64-bit programs.

The *Systems/C C Library* and *Systems/C++ C++ Library* manuals contain detailed information about how to produce OpenEdition programs and the runtime environment supported under OpenEdition.

Programming for MVS 3.8

Systems/C++ supports creating programs for the MVS 3.8 operating system. Generally, the full support of the Systems/C library is available, with the restrictions inherent in the MVS 3.8 environment.

The *Systems/C++ C++ Library* and *Systems/C C Library* manuals contain detailed information about how to produce MVS 3.8 programs.

IBM C++ Compatibility Mode

The Systems/C++ compiler, **DCXX**, can produce assembly language source that, when assembled with the Systems/ASM assembler, is object compatible with IBM's C++ product.

This facility allows **DCXX** to be used as an IBM C++ compatible cross-hosted compiler. Thus, you can generate IBM C++ compatible objects on any of the cross-platform hosts supported by Systems/C++ and the Systems/ASM assembler for eventual linking in an IBM C++ environment.

Requirements

Using **DCXX** to compile in IBM C++ compatibility mode requires the availability of the IBM C++ system include files. If OS/390 V2.4 or later is available, these can e found in the `/usr/include` directory in the HFS file system.

Using IBM's NFS or SMB server facilities, these can be made available to any cross-platform host for use by **DCXX**.

Also, to link the eventual objects into an executable program, you will need the IBM C and C++ libraries installed on your mainframe. The IBM C++ documentation describes the procedures used for linking IBM C++ programs.

How Systems/C++ differs from IBM C++

When the *–fc370* option is used, **DCXX** is very compatible with IBM C++ object code, although there are some differences that need to be noted.

There are some different requirements for `#pragma` directives, described in the section on `#pragma`s.

The *–fc370* option sets various other options to match the IBM compiler.

Differences from Systems/C++

The objects generated in Systems/C++ mode cannot be directly linked into an IBM C++ program.

Alternatively, the Systems/C Library's Direct-CALL (DCALL) feature can be employed to create a Systems/C environment when the Systems/C++ functions are invoked. Note that the Systems/C functions should not invoke or use any of the IBM C library functions, as the IBM C library functions will be called outside of an IBM C runtime context.

Consult the IBM C documentation for the appropriate information on how to invoke assembler language functions from IBM C.

The –fansi_bitfield_packing option

IBM C bitfield sizes and allocation vary based on the value of the **LANGLVL** option specified on the IBM C compile step. When running under TSO or BATCH, or with the *c89* compiler driver, IBM C will default to **LANGLVL=ANSI**. When run with the *cc* compiler driver, IBM C defaults to **LANGLVL=COMMONC**.

When **LANGLVL=ANSI**, IBM C will allocate bitfields and align structures containing bitfields so that the fewest bytes are used. **DCXX** will follow the same algorithm when the *–fansi_bitfield_packing* option is enabled on the **DCXX** command line.

When **LANGLVL=COMMONC**, IBM C will pad structures that end in bitfields to account for the remaining bits declared in the bitfield type, as many C compilers do. When *–fansi_bitfield_packing* is not specified on the command line, **DCXX** follows this algorithm.

Thus, **DCXX** provides complete structure and bitfield compatibility with IBM C. If the structure sizes or member offsets vary from IBM C++, examine the value of the **LANGLVL** option in the IBM C++ listing and set *–fansi_bitfield_packing* appropriately.

Also the `STRUCTURE MAP` section in the **DCXX** listing can be helpful in determining offsets.

Assembling with Systems/ASM assembler

Assembling the output of **DCXX** in IBM C++ compatibility mode requires the use of the Systems/ASM assembler, **DASM**. Using the Systems/ASM assembler,

IBM C++ compatible objects can be generated on any of the supported platforms, including OS/390, z/OS and the cross-platform hosts.

There are two important Systems/ASM options to consider when assembling the compiler-generated assembler source in IBM C++ compatibility mode, the *–batch* option and the *–idr* option.

The *–batch* option is enabled by default and should not be disabled. When compiling in IBM C compatibility mode, the compiler generates more than one set of assembly source per file, requiring the *–batch* option. *–batch* is enabled by default in the Systems/ASM assembler.

The *–idr* option is used to provide specific information in the IDR section of **END** cards generated by the assembler. The IBM C++ pre-linker and IBM binder examine the IDR information to verify that the object deck was properly generated and that the object format is supported by this particular version of the pre-linker and/or binder. For compatibility with IBM C V1R3, the IDR value should be "`5645001  1300`". For V2R4 compatibility, the IDR should be "`5647A01  2400`". Note that there are three spaces between the two parts of each of these IDR values.

A typical Systems/ASM assembler command line on a UNIX platform, when assembling sources compiled with the *–fc370=v2r4* option would be:

```
dasm -fdupalias -idr "5647A01  2400" -o object ... file.asm
```

For version 1.95 and later, the **DCXX** compiler will place `*PROCESS` lines in the generated assembly source that cause the *–fdupalias* and *–idr* values to be appropriately set. Thus, for version 1.95 of **DCXX** and later the *–fdupalias* and *–idr* options are not required on the Systems/ASM (**DASM**) assembler command line.

Consult the Systems/ASM documentation for more information about these options and how to use the Systems/ASM assembler on your system.

Pre-Linking

On cross-platform hosts, the Systems/C pre-linker (**PLINK**) is capable of performing all of the pre-linking tasks needed for IBM C objects. When an IBM Extended Object Module is discovered in the input objects, **PLINK** switches to "IBM mode," and operates in a fashion compatible with the IBM pre-linker.

Alternatively, the IBM pre-linker (`EDCPRLK`) can be employed to pre-link IBM Extended Object Module objects. Or, on newer systems, the IBM binder can directly process these objects.

Consult the *Systems/C Utilities* manual for more information about using **PLINK** to pre-link IBM Extended Object Modules.

Linking

The perform the final link of IBM Extended Object Modules, the IBM linker can be employed. For cross-platform hosts, the pre-linked object can be transferred to the mainframe host for use by the mainframe linker.

Alternatively, for cross-platform hosts, **PLINK** can be employed to create a TSO TRANSMIT module, which can then be `RECEIVE`'d on the mainframe platform.

To learn more about how to use **PLINK** to produce load modules on cross-platform hosts, consult the *Systems/C Utilities* manual.

Debugging

Debugging of **DCXX**-generated objects is fully supported.

The IBM `dbx` or `C/370 Debug` debuggers can debug **-DCC**-generated files in IBM compatibility mode. In IBM compatibility mode, the compiler can generate "ISD" debugging information or the newer DWARF debugging information, compatible with the information generated by the IBM compilers.

When using the Dignus runtime library or other modes, the Systems/DBG debugger `DDBG` can be used to debug programs. To learn more about the `DDBG` debugger, consult the *Systems/DBG* manual.

To request that the compiler generate debugging information, add the *–g* option.

Example

In the following example, we are compiling the two sources, `file1.c` and `file2.c` in IBM compatibility mode, targetting OS/390 2.6. Then, we perform the pre-linking operation on the cross-platform host, resulting in an object suitable for final linking on the mainframe host.

It is assumed that the IBM system include files have been made available in the *IBM-include-directory*, via some network or other mechanism (e.g. NFS.)

First, compile and assemble each of the files:

```
dcxx -fc370=v2r6 -IIBM-include-directory  file1.c
dasm -fdupalias -idr "5647A01   2600" -o file1.o file1.s

dcxx -fc370=v2r6 -IIBM-include-directory  file2.c
dasm -fdupalias -idr "5647A01   2600" -o file2.o file2.s
```

Then, we use **PLINK** on the cross-platform host to pre-link the two files. Also, in this step, we assume the IBM object files are available in a **DAR** archive, prepared from the appropriate PDS on the mainframe. Again, this could be via a network mechanism from the mainframe. In this example, the **DAR** archive is named `libsceeobj.a` and resides in the directory `ibmlibs`. The resulting output file is written to `prog.obj`

```
plink  -oprog.obj file1.o file2.o -Libmlibs -lsceeobj
```

At this point, `prog.obj` is the pre-linker output file and is ready to transmit to the mainframe for final linking.

Customizing DCXX-generated Assembly Source

The assembly source generated by **DCXX** can be customized in several ways to assist in development.

Altering the generated assembly source will prevent the use of the Systems/C C and C++ libraries. Furthermore, re-entrant variables (*–frent*) should be used with caution. Your own run-time library will need to properly allocate the **PRV** and initialize re-entrant variables.

Specifying alternate Entry/Exit macros

By default, **DCXX** generates invocations of the Systems/C prologue and epilogue macros, **DCCPRLG** and **DCCEPIL**. These macros will suffice in many situations. However, when producing assembly code that will become part of an existing program, it may not be appropriate to include all of the function provided by the Systems/C environment. Typically, in an existing program, there are existing prologue and epilogue macros already in use. **DCXX** can be instructed to use those macros instead of the Systems/C macros, generating assembly source that can be assembled and linked into your existing program.

The assembly code generated by **DCXX** makes several assumptions that you must ensure are preserved by your own prologue and epilogue macros:

The prologue macro is responsible for saving the previous values of the registers in the caller's register save area.

The prologue and epilogue are responsible for maintaining the run-time stack. The size of local stack space required for a function will be named as the **FRAME** argument to the macro invocation. The generated code assumes that the frame register is completely updated at completion of the epilogue code. By default, the frame register is R13, but it can be changed via the *–fframe_base=* option. If the size of the local data is greater than 4096, then a literal, **@FRAMESIZE_*nnn***, will also be

allocated which is guaranteed to be addressable in the first 4K region to contain the frame size.

The base register is set up correctly to point to the entry point of the function. The entry point has another label named **REGION_*nnn*_1** which the compiler can reference. The prologue macro is responsible for establishing this label. The value of *nnn* is the function's index number, which is provided as the **CINDEX** argument to the prologue macro. This number is unique for all functions in a compilation. Also, the base register is named by the compiler in the value of the **BASER** argument to the prologue macro.

The prologue macro is responsible for declaring the function as externally visible (i.e. `ENTRY`) if needed. The value of the **ENTRY** argument to the macro will be *YES* if the function should be externally visible.

If the `ZARCH=YES` parameter is specified on the prologue macro, then this is a z/Architecture function, and the prologue is responsible for saving 64-bit registers. As well, the epilogue should restore the 64-bit registers.

If the `ENB=`***nn*** parameter is specified, this function contains exception handling information. The value *nn* defines the offset in the function's automatic variables where the start of the exception handling information can be found. It is not recommended that C++ functions which require exception handling be used in existing routines, because of the complicated runtime requirements.

The prologue macro is responsible for deallocating the local stack space, restoring the register contents to their previous values and returning to the caller.

For re-entrant programs, **DCXX** also generates an invocation of the **DCCPRV** macro to acquire the address of the Pseudo-Register Vector (`PRV`). **DCCPRV** accepts one argument, **REG**=*nn*, which specifies which register should contain the address of the PRV when the macro has been expanded. **DCC** will invoke the macro at the start of each function that needs to address data in the PRV and will save the resulting value at a location in the local stack frame. The supplied **DCCPRV** macro works in conjunction with the Systems/C stack and the supplied **DCCPRLG** macro. If an alternate prologue is used, **DCCPRV** must be adjusted appropriately to build re-entrant programs. If the `ZARCH=YES` was specified on the prologue macro, then the generated assembler code expects a complete 64-bit pointer for the value of the `PRV`.

In general, it is not possible to mix functions assembled with an alternate prologue/epilogue with the objects from the Systems/C++ or Systems/C libraries.

An alternate prologue macro can be specified by using the option *–fprol=XXX* on the **DCXX** command line. An alternate epilogue may be specify using *–fepil=XXX*. An alternate PRV address macro may be specified using *–fprv=XXX*.

Adding keywords to prologue/epilogue macros

In some instances, with slight modification, an existing prologue or epilogue can function in a new manner. For example, any existing prologue/epilogue may be adequate for all situations except program start-up, where a slight change is needed. To facilitate this, the Systems/C++ compiler can add extra arguments to the prologue and epilogue macros on a per-function basis, via the **#pragma prolkey** and **#pragma epilkey** directives.

#pragma prolkey(*identifier*, *"key-string"*)

Directs the compiler to add the string "*key-string*" to the arguments presented to the prologue macro for the identified function. The *key-string* may be any C string constant, and thus can comprise several arguments separated by commas. A leading comma will be provided by the compiler if needed.

#pragma epilkey(*identifier*, *"key-string"*)

Directs the compiler to add the string "*key-string*" to the arguments presented to the epilogue macro for the identified function. The *key-string* may be any C string constant, and thus can comprise several arguments separated by commas. A leading comma will be provided by the compiler if needed.

Specifying an alternate base register

DCXX assumes that register 12 is the code base register for functions. However, you can specify an alternate register for this purpose, to improve integration of **DCXX**-generated assembly source into an existing program structure. The alternate base register can be specified using the *–fcode_base=nn* option.

The specified code base register is also passed to the prologue macro in the value of the **BASER** argument.

The code base register can be any register except the frame base register. However, the compiler will registers 0, 1, 14 and 15 for function calls. Use of registers 0, 1, 14 or 15 as the code base should be carefully employed.

Specifying an alternate frame register

DCXX assumes that register 13 will be used for addressing automatic data, local to the function. That is, register 13 is the frame base register.

However, for improved interaction in some existing programs, it may be better to choose another register as the frame register.

The *–fframe_base=nn* option may be used to specify a different frame register for addressing automatic data. The default Systems/C prologue and epilogue macros do not support using an alternate frame register. Thus, proper use of the *–fframe_base=nn* option requires that prologue and epilogue macro implementations which support the named frame base register be provided

The compiler will use registers 0, 1, 14 and 15 for function calls. Use of these registers as the frame base register should be avoided.

Using the Systems/C Library Direct-CALL interface

The Systems/C and Systems/C++ libraries are implemented using the Systems/C++ entry and exit macros which assume a Systems/C++ environment is extant at run-time.

The Systems/C++ environment includes items such as the local stack frame used for automatic variables in your C++ code, the Systems/C++ run-time heap, Systems/C++ global/initialized data, I/O data blocks, etc.

Thus, in order to call a Systems/C++ function which uses the Systems/C++ entry and exit linkage macros, this environment must be established and accessible.

For typical Systems/C++ programs, where your initial function is a C++ `main()` function, the Systems/C++ library handles creation of this environment.

However, there are circumstances where there is no Systems/C++ `main()` function. For example, when calling Systems/C++ routines from COBOL or directly from assembler source in a system exit.

For this situation, Systems/C++ provides the Direct-CALL (DCALL) interface, where a Systems/C++ function can be directly called from any environment. This interface can be employed to either automatically create and destroy a Systems/C++ environment, or to create and re-use, then destroy, a Systems/C++ environment.

This functionality is identical to the Systems/C DCALL interface. The one caveat is that you may want to use `extern "C"` functions for the DCALL entry points because C++ symbols usually have special names in order to deal with overloading.

For more detailed information on the Systems/C++ Direct-CALL run-time environment, consult the *Systems/C C Library* manual.

Debugging Systems/C++ Programs

Because the output of the Systems/C++ compiler is formatted assembly source, the debugging approaches you are familiar with for debugging assembly programs are applicable.

Accessing symbols in a debugging session

For most mainframe debuggers, external symbols are usually readily accessible as they have associated ESD information in the object deck and load module, although no C++ type information is provided.

For automatic variables, the compiler on a per-function basis generates an @AUTO DSECT which describes the variables. The @AUTO DSECT is provided at the end of each function, and contains a description of the automatic variables allocated in the function. By USING the frame base register, typically register 13, you can reference this DSECT to examine or change automatic variables in your debugging session.

The format of the @AUTO DSECT is:

```
@AUTO#funname   DSECT
funname#varname#blocktag  DS   variable-description
funname#varname#blocktag  DS   variable-description
                  .
                  .
                  .
```

Each automatic variable has one entry in the DSECT. The entries in the DSECT are made unique from any other @AUTO DSECTs by prefixing the function's name, followed by a pound character (#).

Furthermore, each entry is made unique from other entries in the same @AUTO DSECT by appending a pound character (#) and a *blocktag*. Typically, the *blocktag* is a counter value associated with the block within the C function.

The *variable-description* following the DS includes the size of the automatic variable, along with some basic type information. When the C++ type can be represented by an assembly-language DS-specification, that will be used. For those C++ types that aren't representable, the *X'nn'* DS-specification will be used. The basic types in the C++ language have equivalent DS specifications, and will be represented. More complex types, such as structures, don't have equivalent DS specifications and will appear as *X'nn'*.

Forcing a dump

The ready support of direct, inline assembly makes forcing a dump a nice, quick approach to program debugging. Simply place specific values in a register (using `__register()` declarations) and force an 0C1 dump. The register trace back will contain the value you are interested in.

In the following example, the macro `OhC1` is defined to generate assembly code that will force the dump. Then, after loading the value of `errno` into register 2, the macro is invoked.

```
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>

/*  Define a macro that generates inline */
/*   assembly code to force an 0C1.  */
#define OhC1(label,ax) __asm 2 { label dc x'00',ax }


func()
{
   if (open("MYDD",O_RDONLY,0) < 0) {
      __register(2) int r2;

      /* Place the value of 'errno' in */
      /*  register #2. */
      r2=errno;

      /* Force an 0C1 - the value of errno */
      /*  will be in R2 in the register dump. */
      OhC1(labeli,x'00');
   }
}
```

Compiling for Linux/390, z/Linux and z/TPF

The Systems/C++ compiler, **DCXX**, supports compiling source for assembly and execution under Linux running on 390 hardware, Linux/390, or for 64-bit z/Series machines, z/Linux and z/TPF. This allows the programmer to use the same compiler for both OS/390, z/OS, TPF 4.1, Linux/390, z/Linux and z/TPF with little change.

When compiling for Linux/390, z/Linux or z/TPF, Systems/C++ produces assembler source suitable for assembling with the GNU GAS assembler, *as*. Because it produces assembler source, many of the same features available when generating HLASM source are available, e.g. `__asm`, `__register`, etc. As the generated assembler source is targeted at the Linux/390, z/Linux or z/TPF assembler, any inline assembler source inside of `__asm` blocks similarly needs to take this into account.

However, the prologue and epilogue for functions, as well as the calling linkage convention are different from those used with OS/390 and z/OS. Therefore the Systems/C extensions related to prologue/epilogue function do not apply when compiling in this environment.

In general, to generate a program for Linux/390 or z/Linux, **DCXX** is executed with the *–flinux* option, enabling generation of *as*-style assembler source. For z/TPF, the *–fztpf* options is used. This source is then assembled, producing an object file in ELF format. That file can then be linked with any other Linux/390 or z/Linux objects to produce the program.

If the *–mlp64* option is specified, the resulting assembly language is targeted as 64-bit z/Linux, and should be assembled with the z/Linux version of the `as` assembler. The *–mlp64* option is enabled by default for z/TPF.

The –flinux option

The *–flinux* option causes the compiler to generate source suitable for assembly by the Linux/390 or z/Linux GNU assembler, *as*. This assembler source is very similar to HLASM source, except that *as* does not support some of the more advanced

features of HLASM. For example, there is no USING statement, no macro preprocessor, etc. Thus, the generated assembler source is a more direct representation. For more information about the input accepted by *as*, see the GNU info file for *as*.

The *–flinux* option also disables those features which are not supported because of this different assembler syntax. Using any disabled features will typically produce a warning message and the feature will be ignored.

If the *–mlp64* option is specified, the generated assembler source should be assembled with the z/Linux version of *as*, creating a 64-bit ELF object. Otherwise, it should be assembled with the Linux/390 version of *as*, creating a 32-bit ELF object.

The *–flinux* and *–fztpf* options enable the *–fieee* option, causing IEEE constants and IEEE floating point instructions to be used for floating point arithmetic.

Using Linux/390 and z/Linux system #include files

The `#include` files provided with Linux/390 take advantage of many GNU extensions, and assume the presence of several pre-defined macros. Furthermore, the system header files are tailored to each release of the GNU C compiler, *gcc*.

Many of these extensions have been added to **DCXX** to support the Linux/390 and z/Linux header files. The Linux/390 and z/Linux system include files expect pre-defined macros, which Systems/C++ provides automatically when *–flinux* is specified. The *–I* search list should include the GNU C compiler headers in the proper order.

To determine what should be added to the **DCXX** command line, run *g++* with the *–v* flag, where it produces the options it uses for the GNU compiler. For example:

```
g++ -v t.c
```

produces:

```
Reading specs from
     /usr/lib/gcc-lib/s390-ibm-linux/2.95.2/specs
gcc version 2.95.2 19991024 (release)
 /usr/lib/gcc-lib/s390-ibm-linux/2.95.2/cpp -lang-c -v
-D__GNUC__=2 -D__GNUC_MINOR__=95 -Dlinux -D__s390__ -Dunix
-D__ELF__ -D__linux__ -D__s390__ -D__unix__ -D__ELF__
-D__linux -D__unix -Asystem(linux) -Acpu(s390)
-Amachine(s390) -Asystem(unix) -D__CHAR_UNSIGNED__ t.c
/tmp/ccy98GUC.i
GNU CPP version 2.95.2 19991024 (release) (Linux for
S/390)
```

```
#include "..." search starts here:
#include <...> search starts here:
 /usr/include
 /usr/lib/gcc-lib/s390-ibm-linux/2.95.2/../../../../s3
 90-ibm-linux/include
 /usr/lib/gcc-lib/s390-ibm-linux/2.95.2/include
 /usr/include
End of search list.
```

The *–I* options used under normal Linux/390 compiles become clear.

The equivalent **DCXX** command line under Linux/390 would be:

```
dcxx -flinux \
-I/usr/lib/gcc-lib/s390-ibm-linux/2.95.2/../../../../s390-ibm-li
nux/include \
-I/usr/lib/gcc-lib/s390-ibm-linux/2.95.2/include \
-I/usr/include \
t.c
```

The path to the Systems/C++ C++ library `#include` files should also be specified. For example, if the Systems/C++ C++ library files are in

`/usr/local/dignus/include/libcxx`

then you would add

`-I/usr/local/dignus/include/libcxx`

as the first `#include` search path entry.

It is suggested that the command above be placed in a shell script at your installation, to make compiling Linux/390 and z/Linux programs easier.

Furthermore, note that the Linux/390 and z/Linux system include files can be copied to any of the Systems/C++ supported platforms. Doing so enables Systems/C++ to cross-compile Linux/390 source on other platforms.

Using z/TPF #include files

For z/TPF builds, IBM has modified the include files to be automatically processed with **DCXX**. No changes are required.

Assembling Linux/390, z/Linux or z/TPF assembler source

Systems/C++ generated assembler source may be assembled directly with the Linux/390 or z/Linux versions of *as* as appropriate. The source can also be passed to the *gcc* compiler driver for assembly. The *gcc* compiler driver will invoke *as* to accomplish the assembly.

Using the Linux/390 or z/Linux as command

When **DCXX** is executed with the *–flinux* or *–fztpf* options, the generated assembler source is formatted to be assembled with the Linux/390 and z/Linux assembler, *as*. For detailed information regarding the *as* assembler; refer to the manual page on the Linux/390 or z/Linux system.

Note that if the *–mlp64* or *–fztpf* option is specified, the 64-bit z/Linux version of *as* should be used.

Some helpful options are:

–a Turn on assembly listings. Adding *l* enables an assembly listing, adding *s* enables a symbol listing. Adding *=filename* will direct the listing to a particular *filename*.

–o file Direct the assembler to write the object to *file*.

–v Announce the assembler version.

For example, if the generated output from **DCC** was in the file `myprog.s`, then the following command on Linux/390 and z/Linux will assemble the file, place a listing in `myprog.lst` and produce the object file `myprog.o`:

```
as -als=myprog.lst -o myprog.o myprog.s
```

Using the gcc driver to assemble

As an alternative to directly invoking the assembler, the GNU compiler driver, *gcc*, can be used to assemble **DCC**-generated assembler source. If the generated assembler source file ends in "`.s`", *gcc* will invoke the assembler for this file to create a "`.o`" object file. For example, the `myprog.s` assembler source could be assembled with:

```
gcc -c myprog.s
```

The *–c* option indicates that linking should not be performed. This will execute the assembler and produce the file `myprog.o`.

Linking on Linux/390 and z/Linux

Once the **DCXX**-generated assembler source has been assembled, it can be linked as any other object is linked on Linux/390 or z/Linux. This is typically accomplished with the *gcc* compiler driver. The *gcc* compiler driver will invoke the Linux/390 or z/Linux linker, *ld*, passing the name of the object file, along with any library files which may be needed.

For more information regarding the *ld* linker or the *gcc* compiler driver, consult the Linux/390 or z/Linux on-line manual pages with the commands:

```
man ld

man gcc
```

For example, if the **DCXX**-generated assembler source `myprog.s` had been assembled into `myprog.o`, then linking this on Linux/390 or z/Linux to produce `myprog` is simply:

```
gcc -o myprog myprog.o
```

At this point, `myprog` is ready to run.

Example Linux/390 compile and link

By way of example, consider the following simple C++ source. For this example, we do not include any Linux/390 system headers, which simplifies the DCC command line:

```
extern "C" int printf(const char *,...);

int main()
{
   printf("Hi from Linux/390!\n");
}
```

If this C++ source is in the file `./mytest.c` on a Linux/390 host, then the following commands will compile, assemble and link the program, producing the executable `mytest`:

```
dcxx -flinux -omytest.s mytest.c
as -al=mytest.lst -o mytest.o mytest.s
gcc -o mytest mytest.o
```

Notice also that on the *as* step, a listing file was specified — `mytest.lst`. If no assembler listing is needed, then the *as* step can be incorporated into the linking step, and the commands simply become:

```
dcxx -flinux -omytest.s mytest.c
gcc -o mytest mytest.s
```

Using DCXX for z/TPF

The Systems/C++ compiler can be used to write programs for z/TPF, by specifying the *–fztpf* option.

When *–fztpf* is specified, the compiler generates *as*-style assembly source and should be assembled with the GNU GAS assembler for 64-bit Linux.

The resulting object file is an ELF object file that can be linked as normal in a z/TPF environment.

The normal extensions available in Systems/C++ are also available in a z/TPF environment; including in-line assembly, various `#pragmas` and other language features that offer improved compatibility with TPF 4.1 compiles.

Systems/C++ can also produce a compiler listing similar to the one used in a TPF 4.1 environment.

As of PUT 07, the IBM `maketpf` utility supports the use of **DCXX** for z/TPF, no changes to `maketpf` should be required. Furthermore, the `maketpf` utility invokes the `tpf-dcxx` script to accomplish the compile and link, so no direct invocation of *as* is needed. **DCXX** is fully integrated into `maketpf` and its use is supported by IBM.

Consult the IBM z/TPF documentation for more information on `maketpf` and on using **DCXX** to build programs for z/TPF.

Using DCXX for Linux on other hosts

DCXX is supported on many different platforms. On each of these, the compiler can be employed to generate Linux/390 or z/Linux assembler source by including the *–flinux* or *–fztpf* option.

To do so, the Linux/390,z/Linux or z/TPF system include files need to be available to the host platform for reference there, either via network access or a copy. Once the system include files are available, **DCXX** can be employed just as it would be on a native Linux/390 or z/Linux host.

Furthermore, it is possible to construct a version of the GNU assembler, *as*, which can assemble the **DCXX**-generated assembler source on many UNIX platforms. Or, the GNU assembler can be invoked natively on a Linux/390 or z/Linux platform by using network facilities such as `rexec`.

For example, it would be possible to generate Linux/390 assembler source on an OS/390 host, then use the OS/390 `REXEC` program to invoke the Linux/390 assembler to assemble the source.

Similarly, it is possible to construct a version of the GNU linker, *ld*, which will execute on many UNIX platforms to link the objects to produce an executable.

For more information regarding the GNU *as* and *ld* tools, and how to configure and build them on alternative hosts, refer to your Linux/390 or z/Linux documentation, or see http://www.gnu.org.

Systems/C C Library

The Systems/C C library provides the ANSI standard functions, as well as several extensions which aide in the porting of other programs to the mainframe.

For detailed information on the run-time environment, consult the *Systems/C C Library* manual.

Systems/C++ C++ Library

The Systems/C++ C++ library provides the ANSI standard functions and the Standard Template Library, as well as several extensions which aide in the porting of other programs to the mainframe.

For detailed information on the run-time environment, consult the *Systems/C++ C++ Library* manual.

Note that both **libcxx** and **libstdcxx** changed extensively at version 2.10. If linking against either of these libraries, it is important that all objects were compiled with a version of **DCXX** newer than 2.10.

License Information File

DCXX consults the license information file each time it is executed. Information in the file includes the licensee name, expiration, license key, and other pertinent information.

This file must be accessible or the compiler will not execute.

On UNIX, Linux and Windows host platforms, the file is named `dignus.inf` and is found in the same directory as the **dcxx** executable. The `dignus.inf` file is a text file which can be edited by any text editor. However, changing the expiration date, licensee, options or host platform definitions will invalidate your license.

On OS/390 and z/OS, the license information file is named `DIGNUS` and is found in the same load module PDS as the **DCXX** executable module. `DIGNUS` is in load module format, and is generated from assembly language source. To make changes in the license information, the assembly language source must be changed, assembled and link-edited to produce the the `DIGNUS` load module. However, changing the expiration date, licensee, options or host platform definitions will invalidate your license.

As well as license information, the file can also specify the location of the System/C++ system include files. These are the files which are specified in angle brackets in C++ preprocessor `#include` directive, e.g:

```
#include <stdio.h>
```

The "System Include" statement is used to specify the location of the System/C and Systems/C++ library header files.

On UNIX, Linux and Windows host platforms, this is typically the include subdirectory of the Systems/C++ installation, e.g.:

```
System Include=sysc_directory/include
```

On OS/390 and z/OS, this is base name of the PDSs which constitute the Systems/C and Systems/C++ library header files. This can be specified in the `dignus.asm` file as:

```
DC   C'System Include=//DSN:sysc_prefix.INCLUDE'
DC   X'15'
DC   X'0'
```

The special keyword "`LICENSE`" at the beginning of the path is expanded to the path in which DCC found the license file itself. For example, if the license file is in `C:\DIGNUS\BIN`, and you had the following line in your license file:

```
System Include=LICENSE\ ..\ include
```

then DCC would look in `C:\DIGNUS\INCLUDE` for the headers.

Your `dignus.inf`, or `dignus.asm` assembly language source to create `DIGNUS`, is provided separately from the installation materials. Editing this file is part of the installation process, and is described further there.

If you have more than one licensed product from Dignus, LLC, you can simply concatenate the license text together into one `dignus.asm` or `dignus.inf` file.

Run-time support for exceptions

DCXX has several distinct modes for implementing exceptions. When *–flinux* or *–fztpf* is specified, it generates exception calls and data structures which are compatible with those generated by *g++* (version 4.1.2), using GNU *libstdc++*. When *–fc370* is specified, it generates exception calls and data structures which are compatible with those used by the IBM C++ compiler, using the Language Environment runtime libraries. When using our own Systems/C++ runtime library, we handle exceptions as described in the following sections.

Systems/C++-style exceptions

Exception handling in **DCXX** has two main components: per-function tables detailing the current exception cleanup state and some helper functions in the runtime library (such as the provided **libcxx**). The exception ABI was re-designed in the 2.10 release in order to provide for zero execution overhead if no exceptions are thrown. As a result, any programs linked with **libcxx** version 2.10 or above must have been compiled with **DCXX** version 2.10 or above.

When **DCXX** encounters a function which needs exception handling (either for destruction of local data or for a `catch` statement), it adds a parameter to the `DCCPRLG` macro, `EHB=`*label*. The provided label indicates the beginning of a constant exception handling table.

Exception Handling Table

In 24/31-bit mode, the exception handling table consists of a series of entries of the following format:

Offset	Bytes	Description
0	4	offset to start of call region
4	4	offset to end of call region
8	4	offset to catch handler

For 64-bit mode, the exception handling table format is:

Offset	Bytes	Description
0	8	offset to start of call region
8	8	offset to end of call region
16	8	offset to catch handler

In both tables, the addresses are encoded as an offset from the start of the function. The table is terminated by a 0 value in the first field.

During stack unwinding, if it is unwinding a function that does not have an exception handling table, then it unwinds the stack without incident. If there is an exception handling table and the call (i.e., `BALR`) within that function was within a call region, then the corresponding catch handler is branched to if it is non-zero. If the catch handler offset is zero then it simply unwinds the stack. However, if there is an exception handling table but none of the entries match the call region, then stack unwinding calls `std::terminate()`.

The catch handler block itself will typically either attempt to catch the exception (`__eh_begin_catch()`), or call a destructor then re-throw (`__eh_resume()`).

Runtime support

DCXX generates calls to functions in the C++ language support library (`libcxx` to implement runtime support for exceptions:

`void *__cxa_allocate_exception(unsigned long size)`
: When a `throw` statement is processed, the C++ compiler generates a call to `__cxa_allocate_exception()` to allocate the memory needed to store the thrown object. A special exception-aware alternative to `malloc()` or `new` is necessary because it would be inappropriate to use an interface which may possibly throw an exception in the process of throwing an exception. If `malloc()` cannot throw an exception then `__cxa_allocate_exception()` can use `malloc()`. If `malloc()` fails then `__cxa_allocate_exception()` must call `std::terminate()` rather than returning `NULL`. Also, in the Dignus C++ runtime library, `__cxa_allocate_exception()` allocates an extra header before the returned pointer which is used internally for tracking uncaught exceptions.

  Any pointer returned from `__cxa_allocate_exception()` must be passed to either `__cxa_throw()` or `__cxa_free_exception()`.

`void __cxa_free_exception(void *exc)`
: Releases the memory allocated by `__cxa_allocate_exception()`.

`void __cxa_throw(void *exception, void *exc_type, void (*dtor)(void*))`
: `__cxa_throw()` unwinds the stack when an exception is thrown until the top of stack is reached (in which case `std::terminate()` is called) or until a

catch block consumes the exception. It is provided with a pointer to the exception that is thrown, a pointer with a unique value specific to the type of the thrown exception (a type info record), and a function pointer for the destructor to clean up the exception when it's handled. As the stack is unwound, the exception handling tables are processed in order. `__cxa_throw()` must save the thrown object and the thrown exception record so that the other exception functions can access them.

`void *__eh_begin_catch(void *type)`
: At the beginning of a `catch` block, **DCXX** generates a call to `__eh_begin_catch()` to determine if the thrown object type matches the `catch` type. For `catch (...)` (which matches everything), `type` holds `NULL` and `__eh_begin_catch()` immediately returns the pointer to the thrown object. Otherwise, `__eh_begin_catch()` must examine the type info to determine if the types match (possibly taking base class casts into account) and then compute the object address and return it to the caller. If the type doesn't match, the value `NULL` is returned.

  If `__eh_begin_catch()` returns non-`NULL` then the catch handler must ensure that `__cxa_end_catch()` is ultimately called.

`void __cxa_end_catch(void)`
: Once a catch block completes, either successfully or with a re-throw, a call to `__cxa_end_catch()` is generated to finish the process. If there are no other references to the destructor (from re-throw or through `std::exception_ptr`, then it calls the destructor provided to the `__cxa_throw()` call and `__cxa_free_exception()`. Then it returns to the caller.

`void __eh_rethrow_explicit(void)`
: If the user explicitly coded a "`throw;`" to re-throw the exception then DCXX generates a call to `__eh_rethrow_explicit()`, which resumes the stack unwinding. Note that `__cxa_end_catch()` must still be called, usually while the stack is being unwound for the rethrown exception.

`void __eh_resume(void)`
: **DCXX** generates a call to `__eh_resume()` to resume stack unwinding after a destructor call.

In addition, the C++ standard library (**libstdcxx**) calls the following run-time support functions to implement `std::exception_ptr`:

`void __cxa_increment_exception_refcount(void *exc)`
: Increment the reference count to allow the run-time support functions to know about external references to an exception. The exception's destructor will not be called until the reference count reaches zero.

`void __cxa_decrement_exception_refcount(void *exc)`
: Decrement the count of external references to the provided exception.

`void *__cxa_current_primary_exception(void)`
: Return the exception that is currently being caught, and increment its reference count.

`void __cxa_rethrow_primary_exception(void *exc)`
: Re-throw an exception which was returned from `__cxa_current_primary_exception()`. Note that the original `__eh_begin_catch()` call still needs to be completed with a `__cxa_end_catch()` call during unwinding.

`int __cxa_uncaught_exception(void)`
: Returns non-zero if there are any uncaught exceptions.

ASCII/EBCDIC Translation Table

The Systems/C compiler and utilities use the following tables to translate characters between ASCII and EBCDIC. These tables represent the mapping of the IBM Code Page 1047 (IBM1047) to ISO LATIN-1.

However, this is not the official IBM1047 mapping. The official mapping maps EBCDIC X'15' to LINEFEED X'85' and maps EBCDIC X'25' to NEWLINE X'0A'. This is reversed from their traditional mappings. Some vendors use the traditional mapping and some use the official mapping.

The Dignus compilers and utilities use the tradtitional mappings.

ASCII to EBCDIC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	37	2D	2E	2F	16	05	15	0B	0C	0D	0E	0F
1	10	11	12	13	3C	3D	32	26	18	19	3F	27	1C	1D	1E	1F
2	40	5A	7F	7B	5B	6C	50	7D	4D	5D	5C	4E	6B	60	4B	61
3	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	7A	5E	4C	7E	6E	6F
4	7C	C1	C2	C3	C4	C5	C6	C7	C8	C9	D1	D2	D3	D4	D5	D6
5	D7	D8	D9	E2	E3	E4	E5	E6	E7	E8	E9	AD	E0	BD	5F	6D
6	79	81	82	83	84	85	86	87	88	89	91	92	93	94	95	96
7	97	98	99	A2	A3	A4	A5	A6	A7	A8	A9	C0	4F	D0	A1	07
8	20	21	22	23	24	25	06	17	28	29	2A	2B	2C	09	0A	1B
9	30	31	1A	33	34	35	36	08	38	39	3A	3B	04	14	3E	FF
A	41	AA	4A	B1	9F	B2	6A	B5	BB	B4	9A	8A	B0	CA	AF	BC
B	90	8F	EA	FA	BE	A0	B6	B3	9D	DA	9B	8B	B7	B8	B9	AB
C	64	65	62	66	63	67	9E	68	74	71	72	73	78	75	76	77
D	AC	69	ED	EE	EB	EF	EC	BF	80	FD	FE	FB	FC	BA	AE	59
E	44	45	42	46	43	47	9C	48	54	51	52	53	58	55	56	57
F	8C	49	CD	CE	CB	CF	CC	E1	70	DD	DE	DB	DC	8D	8E	DF

EBCDIC to ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	9C	09	86	7F	97	8D	8E	0B	0C	0D	0E	0F
1	10	11	12	13	9D	0A	08	87	18	19	92	8F	1C	1D	1E	1F
2	80	81	82	83	84	85	17	1B	88	89	8A	8B	8C	05	06	07
3	90	91	16	93	94	95	96	04	98	99	9A	9B	14	15	9E	1A
4	20	A0	E2	E4	E0	E1	E3	E5	E7	F1	A2	2E	3C	28	2B	7C
5	26	E9	EA	EB	E8	ED	EE	EF	EC	DF	21	24	2A	29	3B	5E
6	2D	2F	C2	C4	C0	C1	C3	C5	C7	D1	A6	2C	25	5F	3E	3F
7	F8	C9	CA	CB	C8	CD	CE	CF	CC	60	3A	23	40	27	3D	22
8	D8	61	62	63	64	65	66	67	68	69	AB	BB	F0	FD	FE	B1
9	B0	6A	6B	6C	6D	6E	6F	70	71	72	AA	BA	E6	B8	C6	A4
A	B5	7E	73	74	75	76	77	78	79	7A	A1	BF	D0	5B	DE	AE
B	AC	A3	A5	B7	A9	A7	B6	BC	BD	BE	DD	A8	AF	5D	B4	D7
C	7B	41	42	43	44	45	46	47	48	49	AD	F4	F6	F2	F3	F5
D	7D	4A	4B	4C	4D	4E	4F	50	51	52	B9	FB	FC	F9	FA	FF
E	5C	F7	53	54	55	56	57	58	59	5A	B2	D4	D6	D2	D3	D5
F	30	31	32	33	34	35	36	37	38	39	B3	DB	DC	D9	DA	9F