Systems/C Compiler
Version 2.30

Systems/C
C Compiler
Version 2.30

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the authors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This product includes software developed by International Business Machines Corporation, which contains the following copyright notices:

Copyright (c) 1995-2005 International Business Machines Corporation and others All rights reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

IBM, S/390, zSeries, z/Arch, z/Architecture, OS/390, zOS, MVS, VM, CMS, HLASM, and High Level Assembler are registered trademarks of International Business Machines Corporation.

UNIX is a registered trademark in the United States and/or other countries licensed exclusively through X/Open Company Limited.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States and other countries.

Dignus, Systems/C, Systems/C++ and Systems/ASM are registered trademarks of Dignus, LLC.

Contents

How To Use This Book

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

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 compliant compiler, ANSI C99 preprocessor
- Support for z/Architecture 64-bit data and AMODE using the new z/Architecture ("grande") instructions
- Direct inline assembly source, with the ability to reference assembly constants, such as `EQU` values and `DSECT` member offsets
- IBM C compatibility mode when used with the Systems/ASM, **DASM**, assembler
- C language extensions, including:
  - Support for 128-bit (`__int128`) and 64-bit arithmetic (`long long`)
  - AR-mode support (`__far` and `__based` pointers)
  - Support for remote function pointers and the (`__local` and `__remote` function pointer type qualifiers)
  - Built-in implementations for common C library functions
- Systems/C C library
- GCC compatibility features, including `__attribute__` and `__typeof__`, label addresses, local labels, binary constants and other extensions
- Support for generating programs for MVS, z/OS, CMS, z/VSE, Linux/390, z/Linux, TPF 4.1 or z/TPF.

DCC, the C compiler component of Systems/C, can generate assembly language ready to assemble on the mainframe, for OS/390, z/OS, Linux/390, z/Linux, CMS, TPF 4.1, z/TPF or VSE. When used in conjunction with Systems/ASM, this

presents a powerful programming environment for creating 390 or z/Architecture programs.

Systems/C also supports cross-hosted development, where the compilation of C source occurs on a non-mainframe platform. The resulting assembly language source can be transferred to the mainframe ready for assembly. Or Systems/C can take advantage of a cross-hosted assembler, such as the Systems/ASM assembler. When combined with Systems/ASM, the Systems/C compiler, **DCC**, can generate objects that are then transferred to the mainframe for final linking.

Moreover, in this cross-hosted environment, the **PLINK** utility can perform the final linking for z/OS and MVS programs.

Implementation Definitions

Implementation limits

The Systems/C compiler, **DCC**, exceeds ANSI C requirements for implementation limits. All of the limitations on line length, string constant length, and similar items are fixed only by the available memory at compile time. That is, they can virtually be considered unlimited.

However, while the compiler has an unlimited length for identifiers, the assembler used to create an object may not. HLASM has an effective identifier limit of 1024 bytes, the Systems/ASM, **DASM** assembler has an identifier limit of 4096 bytes. Furthermore, certain compiler options can limit the length of identifiers presented to the assembler. In these cases, the assembler may impose an 8-character limit on external identifiers.

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

In general, as dictated by the C standard, external identifiers are case-specific. However, certain options can be used to cause the compiler to generate assembler source which will not be considered case-specific.

EBCDIC character set

On EBCDIC platforms, the compiler assumes the input source is encoded in a modified `IBM-1047` character set, the same one used by default by IBM's Unix System Services environment. However, the compiler will also recognize the characters from the `IBM-037` character set, allowing either encoding to be used.

If the –fcodepage500 option is specified, the compiler performs character translations to support the `IBM-500` character set. Also, the `#pragma filetag` option can be used to alter the compiler's character set assumptions on an individual source file basis.

See the –fcodepage500 option description for more information about `IBM-500` support.

Basic Data Types and Alignments

The default signedness for `char` is `unsigned`, making `char` equivalent to `unsigned char`. The *–fsigned-char* and *–funsigned-char* options can change this default.

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

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

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

If *–mlp64* is specified, the type `long`, either `signed` or `unsigned`, is 64 bits long and aligned on 64 bit, or doubleword, boundaries. If *–milp32* is specified, `long` variables are 32 bits long and aligned on 32 bit, or fullword, boundaries.

The type `long long`, either `signed` or `unsigned`, is 64 bits long. If *–mlp64* is specifed, `long long` is treated the same as `long` and is aligned on 64 bit, or doubleword, boundaries. If *–milp32* is specified, `long long` is aligned on 32 bit fullword bounaries.

The type `float` is 32 bits long and aligned on 32 bit, or fullword boundaries.

The type `double` is 64 bits long and aligned on 64 bit, or doubleword, boundaries, except in a formal parameter list, where `double` is aligned on 32 bit boundaries for MVS, z/OS, IBM compatibility -mode, CMS and VSE. Linux/390, z/Linux and z/TPF alignments follow the rules of the appropriate Application Binary Interface. If the *–mlp64* option is specified, `double` parameters, as other parameters, will be 64 bit aligned.

The type `long double` is 128 bits long and aligned on 64 bit, or doubleword, boundaries. The *—flong-double-64* option can be specified, in which case the `long double` type is treated the same as the `double` type. The type `__float128` is a `long double` type that is guaranteed to be 128-bits long regardless of the setting of the *–flong-double-64* option.

If the *–fieee* option is enabled, floating point constants and values are in IEEE binary format, otherwise they are in IBM hexadecimal format.

The ISO/IEC TS 18661-3:2015 standard extended and interchange types `_Float32`, `_Float32x`, `_Float64`, `_Float64x`, and `_Float128` types are supported. They are always IEEE binary format and operations on them are performed using IEEE arithmetic, regardless of the setting of the *–fieee* option.

Bitfields are allocated left-to-right within the field, and may be unsigned or signed. By default, a bitfield is considered unsigned unless explicitly declared signed. If the

–flinux or *–fztpf* options are specified, bitfields are considered signed by default. The *–funsigned-bitfields* and *–fsigned-bitfields* option can change this default.

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*, *–fshort-enums* and *–fc370=version* options can alter this behavior to taylor enumeration sizes.

If *–mlp64* is specified, pointers are 64 bits long and aligned on 64 bit, or doubleword, boundaries. If *–milp32* is specified, pointers are 32 bits long and aligned on 32 bit, or fullword, boundaries. Pointers are assumed to be "clean", in the sense that any unused high-order bits are assumed to be zero. `__far` pointers are 64 bits long and aligned on 32 bit, or fullword, boundaries, unless the *–ffar-align* option is enabled. If *–ffar-align* is enabled, `__far` pointers are aligned on 64 bit boundaries. `__based` pointers are 32 bits long and aligned on 32 bit, or fullword, boundaries.

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 not 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.

`__far` pointer values are returned in access register 15 (AR15) in combination with register 15 (`AR15`/`R15`.)

Note that when returning integral values smaller than 32 bits, the values are promoted to the full 32-bit value to completely fill the register. The upper bits of register 15 will be appropriately set based on the signedness of the return type. If *–mlp64* is specified, the value in register 15 will be appropriately expanded to fill the full 64-bit 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. If *–fieee* is not specified, the `float` type is promoted to `double` when returned. If *–fieee* is specified, a `float` return type is returned as a 32-bit IEEE value in floating pointer

register 0 (`FP0`). A `_Float32` value is also returned as a 32-bit IEEE value in floating point register 0 (`FP0`).

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

Compiling, Linking and Running Programs

This section describes how to invoke the Systems/C C compiler, **DCC**, 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, other texts should be consulted.

DCC translates C source code into native IBM 390 or z/Architecture 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. The HLASM format is used for generating programs for traditional mainframe operating systems. GAS format is used for Linux and z/TPF programs.

This chapter explains how to run the **DCC** compiler, what options are available on **DCC**, how to assemble the generated assembly source and how to link the resulting objects.

Running the compiler, DCC

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

In OS/390 or z/OS

In an OS/390 or z/OS environment, the compiler is executed by invoking the `DCC` 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=DCC,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=DCC,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 **dcc** and may be found in the installation directory. The command line is

`dcc [options]` *input-file.c*

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 **DCC** 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 environment

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

`dcc [options]` *input-file.c*

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

In OS/390 and z/OS

In OS/390 and z/OS, the C preprocessor follows typical rules when searching for `#include`d files. First, for names specified with double quotes, unless *–fno-searchlocal* 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. If the file was `#include`d with angle-brackets, the system include location is then examined.

`#include` names are mapped so as to make the typical UNIX-style include names operate in a reasonable fashion on OS/390 and z/OS. 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 a single period. 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)
```

In UNIX and Windows

In UNIX and Windows, the C preprocessor follows typical UNIX rules when searching for `#include`d files. First, for names specified with double quotes, unless *–fno-searchlocal* 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. Then any system include directories are searched.

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.

DCC 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 **DCC** 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 *–fno-user-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 filename 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 behave as though the source had 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.

Description of options

The options available to **DCC** 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.
–nodiginc	Disable "`System Include`" processing.
–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.
–MM*[=filename]*	Generate a source dependence list ignoring system includes.
–MT *target]*	Specify the target name in dependence list.
–MF *filename*	Specify the name of the file for 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,–gdwarf-*N*	Generate DWARF debugging information

–gstabs	Generate STABS debugging information
–gisd	Generate ISD debugging information
–fansi-bitfield-packing –fno-ansi-bitfield-packing	Follow ANSI rules for bitfield allocation in structures.
–fno-nint-bitfield –fno-nonint-bitfield	Allow any integral type in a bitfield declaration.
–fanonstruct	Allow Microsoft's anonymous structure extension.
–fep=*name*	Specify a name that will be placed on the generated **END** card to denote a program entry point.
–fframe-base=*N*	Specify a register to use as the base register for automatic data.
–ffreestanding	A synonym for the *–fno-hosted* option.
–fcode-base=*N*	Specify a register to use as the base register for executable code.
–freserve-reg=*N*	Instruct the compiler that register *N* is reserved.
–fcxx-comments –fno-cxx-comments	Enable or disable recognition of C++/C99 style comments.
–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.
–lnameaddr –fno-lnameaddr	Enable or disable generation of Logical Name Address info.
–fopts=*macro*	Request interesting options noted at top of generated assembly.
–fendmacro=*macro*	Specify text to appear before the END statement.
–rsa=*size*	Specify the amount of space the compiler reserves for the Register Save Area.
–fhlasm	Generated assembly source is to be assembled with HLASM instead of **DASM**.
–fhosted/–fno-hosted	Indicate the target is a hosted or freestanding environment.

–finstrument-functions	Request function beginning/ending instrumentation.
–fc370=*version*	Compile in IBM compatibility mode.
–fxplink	Use eXtra Performance Linkage.
–fdll	In IBM compatibility mode, compile for DLL support
–fexportall	In IBM compatibility mode, export all data/functions
–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.
–ffilencase	Ignore case in all input file names.
–fno-searchlocal	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]*	Generate a listing of the compilation.
–fno-listing	Don't generate a listing of the compilation.
–trigraphs –fno-trigraphs	Enable/disable recognition of trigraph sequences in input
–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.
–fno-rent	Generate non-re-entrant code.
–fmaxerrcount	Limit the number of reported errors.

–fundef	Undefine predefined `#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 –fno-asciiout	String and character constants will be in ASCII instead of EBCDIC.
–fno-alias-stmts	The generated assembler source will not contain ALIAS statements.
–fshort-names	Truncate long names.
–fignore-case –fno-ignore-case	Ignore case differences when generating assembly names.
–fdollar	Allow the dollar-sign character ($) in identifiers.
–fatid	Allow the commercial-at character () in identifiers.
–fwchar=*n*	Specify the size of `wchar_t`.
–fwchar-ucs	Indicate that wide character constants are UCS-2 or UCS-4.
–fmrc –fno-mrc	mainframe-style or UNIX-style return codes from **DCC**.
–fsname=*name*	Explicitly set the section name of the compilation.
–fno-sname	Allow PLINK to choose unique section names for this compilation.
–fsnameprefix=*char*	Explicitly set the section name prefix.
–fllgrande	`long long` (64-bit) data in "grande" (64-bit) registers
–fieee	Floating point values, operations and constants are in binary (IEEE) format.
–fsyntax-only	Only do syntax and other diagnostic checking, do not generate assembly.
–fdfp	Enable support for decimal floating point values.

–ffar=ao –ffar=oa	Specify the order of the two components of `__far` pointers.
–ffar-align	Align __far pointers on 8-byte boundaries.
–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 or z/Linux code generation.
–fvisibility=*setting*	Set ELF object symbol visibility.
–version	Print the compiler version number on STDOUT.
–famode=*val*	Specify the runtime addressing mode.
–fc99	Enable ANSI C99 language features.
–fc11	Enable ANSI C11 language features.
–fc23	Enable ANSI C23 language features.
–march=esa390,–march=esa390z	Enable ESA/390 compilation.
–march=z*N*	Enable use of the edition *N* z/Architecture instructions.
–mlp64	Enable 64-bit compilation, implies -march=z0.
–milp32	Enable 32-bit compilation.
–mafp –mno-afp	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.
–mextended-immediate –mno-extended-immediate	Enable/disable use of extended-immediate facility instructions.
–mdistinct-operands –mno-distinct-operands	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.
–fgnu89-inline –fno-gnu89-inline	Control use of legacy gcc inlining rules.
–finline*[=x[:y:z]]* –fno-inline	Control inlining optimization.
–O*[n]*	Set optimization level
–fasmcomm=*mode*	Control the comments in the assembly output.
–fasmlnno –fno-asmlnno	Include line numbers in C source comments in generated assembly.
–fcodepage500 –fno-codepage500	Primary source is in EBCDIC IBM-500 encoding.
–fsascdigraphs –fno-sascdigraphs	Support alternate digraphs combinations in input source.
–fat –fno-at	Support @-operator in expressions.
–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.
–fpack=*val*	Specify a default maximum structure alignment.
–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
–fuser-sys-hdrmap –fno-user-sys-hdrmap	Use user `$$HDRMAP` for system `#includes`

–fevents=*filename*	Emit an IBM-compatible events listing
–fenum=*val*	Specify a default enumeration size.
–fshort-enums	Specify a smallest enumeration size.
–ftest*[=name]*	Enable a separate test csect.
–fprolkey=*key*	Append a global prologue key.
–fcommon –fno-common	Enable/disable common linkage for uninitialized globals.
–fdfe –fno-dfe	Enable/disable dead function elimination.
–fmapat –fno-mapat	Enable/disable mapping '@' to '_' in external symbol names.
–fctrlz-is-eof –fno-ctrlz-is-eof	Enable/disable treating control-Z as an EOF character.
–fextended-variadic-macros –fno-extended-variadic-macros	Enable or disable support for GCC-style variadic macros
–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.
–fwrapv –fno-wrapv	Control optimizer wrapping assumptions regarding signed integer arithmetic.
–fwrapv-pointer –fno-wrapv-pointer	Control optimizer wrapping assumptions regarding pointer arithmetic.
–fstrict-aliasing –fno-strict-aliasing	Assume pointers to different types are different addresses.
–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
–fsigned-char –funsigned-char	Control if `char` is signed or unsigned by default
–fsave-dsa-over-call –fno-save-dsa-over-call	Control if DSA bytes are saved and restored over alternate linkage call

–flinkageospromote –fno-linkageospromote	Control promotion of integral parameters smaller than `int` for linkage-OS
–fsource-enc=utf8 –fsource-enc=ascii	Select source character encoding
–fdwarf-extern –fno-dwarf-extern	Enable/disable generation of DWARF data for extern variables
–fgcc-version=*ver*	Set a specific GCC version compatibility target
–Wswitch-outside-range –Wno-switch-outside-range	Check the value of a case label is within range of the switch
–Wswitch –Wno-switch	Verify all instances of a switch of an enumerated type appear in case labels
–Wswitch-enum –Wno-switch-enum	Verify all instances of a switch of an enumerated type appear in case labels, ignoring default
–Wlabel-unused –Wno-label-unused	Generate a warning if a label is defined by not used
–Wunused-parameter –Wno-unused-parameter	Generate a warning if a parameter is not used in a function
–Wunused-variable –Wno-unused-variable	Generate a warning if a variable is not used in a scope
–Wunused-function –Wno-unused-function	Generate a warning if a defined static function is not used
–Wincompatible-pointer-types –Wno-incompatible-pointer-types	Generate a warning for conversion of pointers with incompatible types
–Wdiv-by-zero –Wno-div-by-zero	Generate a warning for compile-time division by zero

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:

```
dcc -Dmacro[=text]
```

For example:

```
dcc -DMAXLEN=1024 prog.c
```

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

```
#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 **DCC** 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.

For example, to add the directory `mydir` to the include search path on UNIX systems, the command line would appear as:

```
dcc -Imydir ...
```

Similarly, to add the PDS `MY.PDS` to the search path on OS/390 and z/OS, the JCL would appear as:

```
//COMP EXEC PGM=DCC,PARMS='-I//DSN:MY.PDS'
```

See the section on include file processing on page 11 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.

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.

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 –nodiginc option (Disable "`System Include`" processing)

By default, the compiler examines the license key file searching for a line of the form "`System Include=name`" for specifying the value of the system include directory.

If the *–nodiginc* option is specified, any "`System Include`" line is ignored and the compiler only uses system include files specified with the *–isystem* option (if any.)

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:

```
dcc -ofilename prog.c
```

For example:

```
dcc -omyfile.asm myfile.c
```

will compile the C source file *myfile.c* 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`.

Compilation stops after the completion of the preprocessing step.

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

The *–MM* option is similar to the *–M* option. It causes the generation of the dependency list that *–M* does, except that files found from system include directories (either those specified in the license key file, or from the *–isystem* option) are not included in the output.

The –MT *target]* option (specify the target for the dependence list)

The *–MT* option can be used to specify the target name for the dependence list. This is used in conjunction with either the *–fdep*, *–M* or the *–MM* options.

The value for *target* in the *–MT* option specifies exactly what the target will be.

The –MF *filename* option (specify the name of the file for dependence list)

The *–MF* option names the file for writing dependence information. It is used in conjunction with either the *–M*, *–fdep* or *–MM* options.

It is equivalent to specifying the *=filename* value on either the *–M* or *–fdep* options.

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 compilation as normal. 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 em–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 and –gdwarf-*N* options (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 **DCC**.

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 *–gdwarf-N* version of the option can be used to specify a particular version *N* of the generated DWARF debugging info. Supported versions are 2 and 3.

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 option (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 IBM 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/C++ when the **LANGLVL** is set to **COMMONC**.

When *–fno-ansi-bitifield-packing* is specified (the default), 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 deprecated *–fansi-bitfield* option enables *–fansi-bitfield-packing* and disables *–nonint-bitfield*.

The –nonint-bitfield option (Allow any integral in bitfield declaration

The ANSI standard only allows the types `int`, `signed int`, `unsigned int`, and `bool` to be used in a bitfield declaration. However, because of extensions in other compilers, programmers frequently use other types (e.g. `char`) to declare bitfields.

The *–nonint-bitfield* option (the default) allows any integral type to be used in the declaration of a bitfield member.

The *–no-nonint-bitfield* option restricts bitfield types to the ANSI standard.

The –fanonstruct option (Allow Microsoft's anonymous structure extension)

The *–fanonstruct* option enables support for Microsoft's anonymous structure extension.

Anonymous structures allow for unnamed inner structures or unions within an outer structure or union. The elements of the inner structure are then directly accessible as if they were elements of the outer structure.

Anonymous structures can also be enabled or disabled using the `#pragma anonstruct` pragma.

See the discussion of anonymous structures in the C extensions section for more information.

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 prologue and epilogues, data offsets, alignments and initializers that inter-operate 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.

The value of *version* is used to indicate which version of IBM C is desired. Current supported *version* specifications are `v1r3`, `v2r4`, `v2r6` and `z1r2` thru `z1r11`. When `v1r3` is specified, a prologue/epilogue compatible with IBM C V1R3 will be generated. When `v2r4` is specified, a prologue/epilogue compatible with IBM C V2R4 will be generated. When `v2r6` is specified, a prologue/epilogue compatible with IBM C V2R6 will be generated. When one of `z1r2` thru `z1r11` is specified, the z/OS prologue/epilogues will be generated.

Furthermore, the compiler provides definitions of predefined macros which are used by the IBM header files. These macros convey specific versions numbers which can enable or disable features available to particular revisions of the IBM LE runtime.

Note that the *–fc370* option does not imply the *–frent* option.

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 *–fdll* and *–frent*.

The –fdll option (In IBM compatibility mode, compile for DLL support)

The *–fdll* option causes **DCC** to compile in IBM DLL mode. This allows the generated object to be used in an IBM DLL.

In DLL mode, the compiler will make appropriate DLL references to external data and functions, and will provide the appropriate code to define data and functions in a DLL fashion.

Note that *–fdll* is only meaningful when *–fc370* is also applied.

By default, *–fdll* is equivalent to IBM's `DLL(NOCALLBACKANY)`, which may be explicitly specified with *–fdll=nocba*. If `DLL(CALLBACKANY)` support is desired, *–fdll=cba* can be used instead. In that case, function pointer calls will go through a special runtime call (`@@FXCLBK`) which will automatically detect whether the function pointer is a DLL function pointer (entry point and PRV) or a regular function pointer (just entry point). This is necessary if any function pointers are initialized in modules which were not compiled with *–fdll* and then used in a module which was compiled with *–fdll*.

The –fexportall option (In IBM compatibility mode, export all defined data and functions)

The –fexportall option causes the compiler to provide 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.

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

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

The –fcxx-comments and –fno-cxx-comments options (Enable and disable recognition of C++-style // comments)

The *–fcxx-comments* and *–fno-cxx-comments* options are used to enable and disable recognition of C++/C99-style "`//`" comments. By default, the compiler recognizes C++/C99-style comments.

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 compatibility)

option, or the *–flinux* (Linux/390 and z/Linux mode.) The macro is responsible for function startup, stack management, saving registers, etc. The prologue macro is passed several parameters:

ARCH=ZARCH Added to the prologue parm when the *–mlp64* option is specified on the compiler command line.

BASER=*n* The register number used as the base register for this function. If the value of *n* is 0, then this function does not require a 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.

SP=*n*

Specifies the storage subpool for Systems/C library dynamically allocate storage used for stack and heap memory.

`SP` can only be meaningfully be specified on entry point functions or the `main()` function.

KEY=*val*

Specifies the runtime protection key. *Val* can be either the numeric key value, or the keyword `ENTRY`.

`KEY` can only be meaningfully be specified on entry point functions or the `main()` function. On other functions, it is ignored.

At library start-up, when `KEY` is specified, the Systems/C runtime will switch into the specified key before invoking the entry point or `main()` function. At library termination, the key will be restored to its original value.

ISTK=*val*

The Systems/C runtime allocates storage used for local function variables, the runtime "stack." The size of the initial storage allocation can be specified via the `ISTK` value.

`ISTK` can only be meaningfully be specified on entry point functions, for example `DCALL=YES`, `DCALL=ALLOCATE`) or the `main()` function. On other functions, it is ignored.

ESTK=*val*

As a program runs, the runtime memory allocated for local function variables may need to expand. The size of the memory allocated for this expansion can be specified with the `ESTK` parameter.

`ESTK` can be specified larger to avoid thrashing around small stack allocations, which can improve runtime performance.

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 –fgnu89-inline and –fno-gnu89-inline options (Control use of legacy gcc inlining rules)

By default, **DCC** uses the C99 rules for the `inline` keyword, but if *–fgnu89-inline* is specified then it uses the legacy **gcc**-compatible rules instead. Versions of **gcc** before version 4.3 used the legacy rules, while newer versions of **gcc** default to the C99 rules.

Under the C99 standard rules, the `inline` keyword only has effect if all of the declarations of a function have `inline` and do not have `extern` or `static`. In that case, the function is an "inline function", and its definition is only used if the optimizer decides to inline a call to it. Otherwise, an external reference is generated to satisfy a function call. An external definition is never provided for an inline function. If a declaration with `extern` is seen, then that forces a definition of the function to be emitted. If a function does not meet the rules for an inline function, then the `inline` keyword has no effect (it will be either externally visible or `static`).

Under the gnu legacy rules, the `inline` keyword has no effect unless `extern` is combined with it, in which case the function is considered an "inline function" as above. That is, an external definition is never provided for an `extern inline` function

The C99 and gnu rules are essentially backwards on the meaning of `extern inline` vs. `inline`. Under C99 rules, `extern inline` will always generate an external definition. Under gnu rules, `extern inline` will never generate an external definition.

In C99 mode, the gnu legacy behavior can be forced on a per-function basis by adding `__attribute__((gnu_inline))` to your function's definition.

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

DCC 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 the C99 `inline` keyword) 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 (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 –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 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 *–fno-lnameaddr* 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 causes the compiler to 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 **DCC** command line.

The purpose of this macro is to provide a mechanism to direct any macro-generated source based on **DCC** 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 **DCC** 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* option may not be used if the *–flinux* or *–fc370* option is 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 –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
tt 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 –fhlasm option (Generated assembly source is to be assembled with HLASM instead of DASM)

The Systems/ASM assembler (**DASM**) is the preferred assembler when building non-Linux or non-z/TPF programs. The *–fhlasm* option allows for creating assembly source that can be assembled with IBM's HLASM assembler.

When *–fhlasm* is specified, the compiler will not generate instructions that take advantage of Systems/ASM extensions.

Because of this, some facilities, such as IBM compatibility mode and debugging information embedded in the generated object, are not supported when *–fhlasm* is specified.

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, with *–finstrument-functions*, 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.

```
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 to and exit from an "inlined" function by generating the instrumentation at the proper location.

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 –*hosted* option (Indicate a hosted verses no-hosted environment)

The *–fhosted* option indicates the target environment is "hosted", meaning all standard features are available in the target environment.

The *–fno-hosted* option indiates the target environment is "freestanding" (not hosted), where it is assumed the target environment does not have all the C standard features.

For *–fc99* or *–fc11* options are specified, the *–fno-hosted* options affects the definition of the standard `__STDC_HOSTED__` macro. If *–ffreestanding* option is specified `__STDC_HOSTED__` will have the value 0, otherwise it will have the value 1.

The *–ffreestanding* option is a synonym for *–fno-hosted* option.

The –fcode-base=*N* option (Specify register to use for addressing for executable code)

The *–fcode-base=N* option specifies a different register to use for addressing executable code. The default code base register is **R12**. The code base register points at the beginning of the current 4K block of executable code and can be used for addressing branch targets and literals used within that 4K region. When *–fcode-base* is specified, the `BASER=N` option is provided to the prologue macro (such as `DCCPRLG`) so that it knows which register to set up with the initial code base.

The –freserve-reg=*N* option (Reserve register #*B*)

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 in-line 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 or z/OS 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 –ffilencase option (ignore case in all input file names)

The *–ffilencase* option causes all input file names (`#include` files as well as the primary source file) to be treated as if the lookup was performed on a case-insensitive filesystem. It is only meaningful for files stored in Unix-style case-sensitive filesystems.

The way *–ffilencase* is implemented, the directory is found in the usual (case-sensitive, depending on the OS) fashion. Once a directory is seen, the list of files in it is read. Then that list is used to enable case-insensitive searches. The list is cached, so the overhead is minimal for subsequent searches.

The –fno-searchlocal option (don't look in "local" directories)

The *–fno-searchlocal* 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=filename* option causes the compiler to behave as if you had

```
#include "filename"
```

as 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.

More than one *–fpreinclude* option may be specified. The named files will be processed in the order they appear on the command line.

The –*trigraphs* option (recognize trigraphs)

The *–trigraphs* option enables recognition and replacement of the C trigraph sequences during input processing of the source. This can be disabled using the *–no-trigraphs* option.

ANSI C23 removed support for trigraphs from the C language, when *–fc23* is specified, trigraphs are disabled. They can be enabled again with the *–trigraphs* option, specified after the *–fc23* option.

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` will be included in the listing. Otherwise, only the source from the primary file will be listed.

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

This option can be disabled with the *–fno-listing* option.

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 including 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 `__rent` by default.

`__rent` variables are placed in the Pseudo Register Vector (the **PRV**) and are referenced via Q-CON references in the generated code.

The –fno-rent option (generate non-re-entrant code)

The *–fno-rent* option instructs the compiler to generate non-re-entrant code. When *–fno-rent* is specified, file scope external and static variables will be `__norent` 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 –fundef option (undefined predefined #define values)

DCC predefines the following values as well as the standard ANSI ones (note that each of these begins with two underscores and ends with two underscores):

Macro Name	Replacement Value
`__370__`	`1`
`__BFP__`	Defined if *–fieee* is enabled
`__COUNTER__`	A unique value at each reference, begining with `0`
`__CHAR_UNSIGNED__`	Defined if the default signedness of `char` is `unsigned`
`__DFP__`	Defined if *–fdfp* is enabled
`__I390__`	`1`
`__LONGDOUBLE128`	Defined if *–mlongdouble128* is enabled
`__LONGDOUBLE64`	Defined if *–mlongdouble64* is enabled
`__LP64`	Defined if *–mlp64* enabled
`__LP64__`	Defined if *–mlp64* enabled
`__NO_INLINE__`	Defined if inlining optimizations are disabled
`__OPTIMIZE__`	Defined if *–O* specified and the optimization level is non-zero
`__OPTIMIZE_SIZE__`	Defined if *–O* specified and the optimization level is non-zero
`__SYSC__`	`1`
`__SYSC_ASCIIOUT__`	Defined if *–fasciiout* enabled
`__SYSC_ANSI_BITFIELD_PACKING__`	Defined if *–fansi-bitfield-packing* enabled
`__SYSC_ILP32`	Defined if the *–mlp64* option is not enabled
`__SYSC_LP64`	Defined if *–mlp64* enabled
`__SYSC_VER__`	Compiler version number

If *–fundef* is specified once, these predefined macros are removed. Specifying the *–fundef* option more than once will remove all predefined macros.

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 or 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 *–fno-margins* options 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 does not alter the listing format.

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, two message styles are supported, **microsoft** and **sysc**.

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.

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

Normally, when the *–flinux* is not used, 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 library doesn'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.

If *–fno-asciiout* is present after *–flinux* on the commandline then **DCC** will generate EBCDIC string constants on Linux.

The –fno-alias-stmts option (generated ASM has no `ALIAS` statements)

DCC takes advantage of the assembler `ALIAS` statement to generate labels that are longer than 8 characters or contain lower-case letters. Some older assemblers either don't support this statement, or have problems in the implementation.

When *–fno-alias-stmts* is employed, the generated assembler source will not contain `ALIAS` statements. There are several restrictions imposed when *–fno-alias-stmts* is enabled:

1. IBM compatibility mode is not supported.
2. Only non-reentrant code is support — *–frent* may not be enabled.
3. External names will be truncated by the assembler, also the assembler will map lower-case letters to upper-case.
4. The Systems/C library may not be used because it assumes lower-case names.
5. `#pragma map` and `#pragma weakalias` will not operate as they depends on `ALIAS` statements for their implementation.

Even with these restrictions, *–fno-alias-stmts* can be useful for generating assembler source that is to be linked with an existing program, particularly when used in conjunction with the *–fshort-names* option.

The –fshort-names option (truncate long names)

The *–fshort-names* option causes the compiler to examine each external identifier. If the identifier is too long, it will be truncated in the generated assembler source. This option is typically used in conjunction with the *–fno-alias-stmts* option to generate assembler source which can be easily linked with a previously existing program.

In order to differentiate lower and upper-case letters in the generated assembler source, **DCC** prefixes upper-case letters with a dollar-sign (`$`). Thus, the truncation to 8 characters in the assembler source may not use all of the letters from the C identifier.

When *–fshort-names* is enabled, the compiler will generate a warning when an externally visible long name is encountered. Note that names which are not externally visible are not truncated.

–fshort-names is not value with *–flinux* is specified.

The –fignore-case and –fno-ignore-case options (ignore/don't ignore case differences when generating assembly names)

HLASM assembler source is a case-insentive language. Thus, in order to differentiate lower and upper-case letters in the generated assembler source, **DCC** prefixes upper-case letters with a dollar-sign (`$`). For example, the C function named `MyFunc` would appear in the generated assembly source as `$My$Func`. This symbol would

also have an associated `ALIAS` statement that caused the actual object to contain the characters `MyFunc` without the dollar signs.

The *–fignore-case* option causes the compiler to ignore upper-case letters when generating assembly labels, and not decorate the assembly label with dollar signs.

When *–fignore-case* is specified, the generated assembly labels appear as they do in the C source. Because of this, when *–fignore-case* is specified the compiler may generate invalid assembly source if two C symbols only differ in case. For this reason, the C programmer has to ensure that symbols are unique, regardless of case, when *–fignore-case* is specified.

–fignore-case is not valid with *–flinux* is specified.

–fno-ignore-case can be used to disable *–fignore-case*.

–fno-ignore-case is the default.

The –fdollar option (allow dollar sign character in identifiers)

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

Use *–fno-dollar* to disable this option.

The –fatid option (allow commericial-at character in identifiers)

According to ANSI standard C, the commercial-at character () is not allowed in C identifiers. When the *–fatid* option is specified **DCC** will allow the commercial-at sign character in identifiers, after the first character. The commercial-at is not allowed as the first character in identifiers to avoid conflicts with the '@' operator.

Use *–nofatid* 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.

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 *–fno-wchar-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 `whar_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 data 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. If the –fshort-name option is used, *name* must be 7 characters or less, otherwise *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 –fno-sname option (allow PLINK to choose unique section names)

When the *–fno-sname* 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 *–fno-sname* requires the use of **PLINK** before final linking of the load module to properly map the various section names.

–fno-sname is enabled if the compiler is reading from standard input (I.e., if no source file name was specified on the command line.)

The *–fno-sname* option is ignored for linux and z/TPF compilations.

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 –fllgrande option (`long long` (64-bit) data in "grande" (64-bit) registers)

The *–fllgrande* option is used in 32-bit environments to indicate that operations involving `long long` should be accomplished using the 64-bit "grande" registers instead of two 32-bit registers.

This is only applicable to 32-bit environments. When the *–mlp64* option is enabled, the compiler naturally uses the 64-bit registers for 64-bit operations.

In 32-bit environments, the compiler assumes that the high order word of 64-bit registers is not preserved across a function call. Thus any values in a 64-bit register will be saved and restored across the call.

The *–fllgrande* option requires the z/Architecture hardware instruction set.

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

The *–fieee* option instructs the compiler to use the Binary Floating Point (BFP) format for floating point constants and use the binary floating point 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, **DCC** will convert floating point constant values into their IEEE format for emission in the generated assembler source. Also, **DCC** will use IEEE arithmetic operations for any floating point operations performed by the compiler at compile time. **DCC** will also generate binary floating point instructions for any arithmetic performed at run time.

The `_Hexadec` type modifier can be used to provide for floating point values that are in the IBM hexadecimal floating point format. Such values will be converted to binary for any operations. Similarly, the `_Ieee` type modifier can be used to provide binary floating point values if the *–fieee* option is not enabled.

If *–fieee* is enabled, **DCC** 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.

When *–fieee* is enabled the `IEEEFP=YES` parameter will be specified on the DCC prologue macro to indicate that the runtime default for the compilation is IEEE.

Note that the types `_Float32`, `_Float32x`, `_Float64`, `_Float64x`, `_Float128` are always IEEE format values, regardless of the setting of the *–fieee* option.

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

The –fsyntax-only option (do not generate assembly code)

When the *–fsyntax-only* option is enabled, the compilation process proceeds through the optimization phase and then stops. No assembly code is generated, but all diagnostics up through the optimization phase will be emitted.

This can be useful for "syntax checking" builds for quick development cycles.

The –fdfp option (Enable support for decimal floating point values)

The *–fdfp* option enables support for the decimal floating types as defined in the N1176 draft of ISO/IEC WDTR24732.

The decimal floating point type are `_Decimal32`, `_Decimal64` and `_Decimal128`. Unlike HFP and BFP floating point values, decimal floating point types use a base radix of 10 instead of 16 (HFP) or 2 (BFP).

When *–fdfp* is specified, the `__DFP__` macro will be predefined by the compiler.

For more information, see the section on the decimal floating point types in the *DCC Advanced Features and C Extensions* portion of this manual.

The –fmrc and –fno-mrc options (Mainframe or UNIX-style return codes)

The *–fmrc* and *–fno-mrc* options alter the return code returned by **DCC**.

Normally, on cross-platform (UNIX and Windows) hosts, **DCC** returns a typical UNIX-style return code, 0 for success or warnings, 1 for errors. And, on OS/390

or z/OS, **DCC** 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 *–fno-mrc* option. When *–fmrc* is enabled, **DCC** will return mainframe-style return codes; allowing for the use of mainframe-style return codes on a cross-platform host. When *–fno-mrc* is enabled, **DCC** will return UNIX-style return codes, allowing for the use of UNIX-style return codes on OS/390 or z/OS.

The –ffar=ao and –ffar=oa options (Specify the component order of __far pointers)

`__far` pointers are comprised of two components, the ALET and OFFSET components. The *–ffar=ao* and *–ffar=oa* options provide for altering the order the compiler uses for these components The ability to use either order can be helpful when interfacing to existing programs that assume a different order.

When *–ffar=ao* is enabled, the compiler uses the [ALET,OFFSET] order.

When *–ffar=oa* is enabled, the compiler uses [OFFSET, ALET] order.

The default order is *–ffar=ao*.

The –ffar-align option (align __far pointers on doubleword boundaries)

Normally, `__far` pointers are aligned on fullword, or 32-bit boundaries. The *–ffar-align* option causes the compiler to align `__far` pointers on doubleword, or 64-bit boundaries.

The –fpatch and –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 is area 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 or z/Linux code generation)

The *–flinux* option instructs **DCC** 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 it possible to generate Linux/390 or z/Linux assembler source on any supported platform, including z/OS and OS/390.

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

If the *–mlp64* option is specified, the generated assembler source should be assembled using 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 –fvisibility=setting option (set ELF object symbol visibility)

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

An `ELF` object file contain 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 –version option (print the compiler version number on STDOUT)

The *–version* option causes **DCC** to print the compiler version number on the STDOUT output stream. After this is done, the compiler exits, and 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.

The –fc99 option (enable ANSI C99 language features)

The *–fc99* option enables new language features found in the 1999 ANSI C standard.

By default, Systems/C is compliant with the 1989 version of the ANSI C standard. However, Systems/C does support many of the ANSI C 99 standard language features when the *–fc99* option is enabled. These include support for the `_Bool` data type, recognition of the ANSI C 99 keywords, support for `#pragma STDC FENV_ACCESS`, declarations inter-mixed with statements in a block, support for declaration-clause in `for` statements, extended initializer designators, compound literals, flexible array members, and variable length arrays, variadic preprocessor macros, `_Pragma`, the C99 `inline` keyword, and further ANSI C99 required diagnostics

If *–fc99* is specified, the compiler will also pre-define the `__STDC_VERSION__` macro as outlined by the ANSI C 99 standard.

Systems/C continues to allow implicit `int` declarations even when *–fc99* is specified.

The –fc11 option (enable ANSI C11 language features)

The *–fc11* option enables new language features found in the 2011 ANSI C standard.

When *–fc11* is enabled, the compiler supports static asserts, type-generic expressions, and no-return functions.

The –fc23 option (enable ANSI C23 language features)

The *–fc23* option enables new language features found in the 2023 ANSI C standard.

When *–fc23* is enabled, the compiler will generate warnings when a function is prototyped or declared with "old style" headers, some C23 attribute specifiers will be accepted, '0b' constants are supported, the single-quote separator can be used in integral and floating point constants, 'W' bit-precise constants are supported as well as bit-precise data.

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

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

Values for *N* are 0 through 13.

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* 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-point 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 environments.

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.

For the Systems/C prologue macro, the compiler will add the `ARCH=ZARCH` option to the prologue macro invocation, indicating the generated 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 altered 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.

When *–mlp64* is enabled, the `__SYSC_LP64__` preprocessing symbol will be defined.

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

The –mafp option (enable/disable use of extended FP registers)

The *–mafp* option indicates the availability of the extended floating-point registers. When *–mafp* is used, FP registers numbered 0 to 15 are assumed to be available. When *–mno-afp* is used, only FP registers 0, 2, 4, and 6 will be used. *–mno-afp* is the default, but many of the settings such as *–mlp64* and *–march=z* options will automatically set *–mafp* because the platform can be assumed to support these options. To override this setting, the *–mno-afp* must occur after any other architecture specifications on the commandline.

The *–mfp4* option is a synonym for *–mno-afp*.

The *–mfp16* option is a synonym for *–mafp*.

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.

Note that the type `__float128` is always a 128-bit floating point value, which may be either IEEE or HEXADECIMAL floating point depending on the *–fieee* option or any `_Ieee`/`_Hexadec` type modifiers.

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

The 9th edition of the z/Architecture 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 –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 **DCC** 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 **DCC** 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 **DCC** 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 **DCC** 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-high-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 –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 –fasmlnno 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 –fcodepage500 option (Primary source is in EBCDIC IBM-500 encoding)

On EBCDIC-based platforms, the compiler assumes that the primary input source is encoded in the `IBM-1047` or `IBM-037` code pages. However, some localities prefer to use the `IBM-500` code pages by default. When the –fcodepage500 option is enabled, the compiler assumes the input is encoded in the `IBM-500` code page. Each input file is assumed to be in the `IBM-500` code page unless a `#pragma filetag` in the file specifies otherwise.

The Systems/C include files are provided in the `IBM-1047` codepage, and are protected by `??=pragma filetag("IBM-1047")` statements at the beginning of each file. Thus, source encoded in the `IBM-500` code page can safely use the Systems/C include files, the compiler adjusts appropriately.

When the compiler processes a source file from the `IBM-500` codepage, it simply maps the following bytes to the C characters:

Byte	Character
0x4a	[
0x4f	!
0x5a	]
0xbb	|
0xba	^

This mapping does not apply to bytes in character or string constants.

The –fcodepage500 option is only available on EBCDIC host platforms.

The –fsascdigraphs option (Support alternate digraphs combinations in input source)

The SAS/C™compiler supports an alternate set of digraph character combinations to replace special characters that may not be available in some EBCDIC environments. That is, some of the typical characters used in the C character set may not appear on some EBCDIC terminal keyboards.

The ANSI standard approach to this issue is to employ ANSI standard tri-graph character sequences; which are fully supported by Systems/C.

However, for aiding in the transition from a SAS/C environment to a Systems/C environment, the compiler supports compiling sources that contain SAS/C digraphs.

When the *–fsascdigraphs* option is enabled, the compiler recognizes the following sequences of digraphs as equivalent to the typical ANSI C characters.

C Character	SAS/C digraph
`[` (left bracket)	`(|`
`]` (right bracket)	`|)`
{ (left brace)	`\(` or `(<`
} (right brace)	`\)` or `>)`
`|` (inclusive or)	`\!`
`~` (tilde)	`\^`

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

The `@` operator is an extension provided by **DCC** 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 for more information about the use of `@` in the *DCC Advanced Features and C Extensions* portion of this manual.

The default is *–fno-at*.

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 –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 –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, **DCC** searches for system headers using the rules from both the user and system `$$HDRMAP` files. When *–fno-user-sys-hdrmap* is specified, searches for system headers use only the rules from the system `$$HDRMAP` files. In either case, searches for user headers use just the user `$$HDRMAP` rules.

–fuser-sys-hdrmap is the default.

The –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 144 for more details.

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

The *–fevents=filename* option causes **DCC** 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* `0 0 0 DCC`*D E F G H*

A The number of the file where the error occurred.

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

The *–fenum*=val specifies the default enumeration size.

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 *–fshort-enums* option (Specify smallest enumeration size)

The *–fshort-enums* option indicates the compiler should choose the smallest possible type that will contain the range of the enumeration's values. This is equivalent to the *—fenum=SMALL* option.

The *–fno-short-enums* option specifies enumerations use the type `int` (4 byte integer) which conforms to the C standard and is the usually the default unless the target environment specifies otherwise (e.g. LE environments.)

The *–fno-short-enums* option is equivalent to *–fenum=INT* option.

The enumeration size setting can be altered via the `pragma enum(`*val*`)`pragma in the source.

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 **DCC** 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 –fno-common options (Enable/disable common linkage for uninitialized globals)

In Linux/390, z/Linux, and z/TPF modes, all defined global data is by default placed in `.data`, which is the behavior when *–fno-common* 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 –fdfe and –fno-dfe options (Enable/disable dead function elimination.)

Normally the compiler does not generate code for unreferenced `static` functions. 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 *–fno-dfe* 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 *–fno-def* will be enabled in case the user wishes to reference such functions during a debug sessions. *–fdfe* can be used to re-enable it.

The –fmapat and –fno-mapat 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 –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 –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 –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 **DCC** 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 –fwrapv and –fno-wrapv options (control optimizer wrapping assumptions regarding signed integer arithmetic)

The C standard indicates that signed integer arithmetic which overflows is undefined behavior. The compiler can take advantage of that to optimize expressions involving signed integer arithmetic with the assumption that an overflow cannot occur.

The *–fwrapv* option indicates that the compiler allows for overflow, and does not apply those optimizations. The *–fno-wrapv* option indicates that the compiler assumes any such overflow is invalid and thus can take advantage of such optimizations.

For example, if *–fno-wrapv* is enabled, then the compiler can replace an expression like `(s + 10 > s)` with the expression `(1)` if `s` is a signed integer. In this case, the compiler assumes that `(s+10)` does not "wrap around."

If instead *–fwrapv* is enabled, then `(s+10)` might wrap around to produce a value which would compare to less-then `s` and thus `(s + 10 > s)` might no longer be true.

Many implementations assume *–fno-wrapv* as it is indicated in the langauge standard, however this can cause difficulty with old non-conforming code that uses that assumption to determine if an arithmetic operation had overflowed.

Currently *–fwrapv* is the default.

Enabling *–fno-wrapv* may provide opportunities for improved optimization at the cost of breaking older non-conforming code.

The –fwrapv-pointer and –fno-wrapv-pointer options (control optimizer assumptions regarding pointer arithmetic)

Pointer arithmetic is normally assumed to not "wrap around" the address space as pointer arithmetic overflow is undefined accoring to the standard. With this assumption, the optimizer can perform some optimizations such as replace `(ptr + 10 > ptr)` with the value `1`.

The –fwrapv-pointer option indicates this assumption cannot be made, that pointer arithmetic does meaningfully "wrap around" and is therefore computed modulo the size of the pointer (32-bits or 64-bits.) In this situation, the assumption that allows the above optimization cannot be made.

Some older C code uses the "wrap around" assumption to determine if a pointer arithmetic computation has overflowed the size of a pointer, but the C standard indicates such an assumption is undefined behavior and such C code is invalid.

The default is *–fno-wrapv-pointer*, allowing for the best optimizations.

Setting *–fwrapv-pointer* defeats optimizations that assume pointer arithmetic doesn't "wrap around."

The –fstrict-aliasing option (assume pointers to different types point to different addresses)

The *–fstrict-aliasing* option controls assumptions made when dereferencing pointers. By default, the optimizer and code-generator assumes that any pointer dereference can potentially alter any aliased values in memory.

When *–fstrict-aliasing* is enabled, the optimizer and code-generator can narrow that assumption to only being aliased memory that has, approximately, the same type as the dereferenced pointer.

The aliasing rules assume that types of 'char' (unsigned or signed) can alias anything, and that unsigned and signed variants of integral types can alias each other. Otherwise, if the type isn't the same then it is not aliased. For example, a dereference of a pointer to 'float' does not affect a dereference of a pointer to 'int'.

–fno-strict-aliasing is the default.

The –v option (print version information)

The *–v* option causes **DCC** 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)

DCC 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.

If *–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 **DCC** listing will contain an extra section with cross reference information, indicating where each symbol is read or modified.

The *–fno-xref* option is the default.

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 –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 –flinkageospromote/–fno-linkageospromote options (Control promotion of integral parameters smaller than `int` for linkage-OS)

The *–flinkageospromote* option controls the promotion of integral-typed formal parameters the are smaller than `int` for calls to linkage-OS style functions.

By default, if a prototype declares an integral formal parameter with a size smaller than `sizeof(int)` to a function declared with `OS` linkage, the value will promoted to an `int` and a pointer to the `int` will be passed. This is also the behavior of many other C compilers in the mainframe environemnt.

If *–fno-linkageospromote* is specified, this promotion will not be performed, and the parameter will be a pointer directly to the type specified in the prototype.

Note that for calls that have no prototype in scope, or for parameters involved in variable parameter lists, the default promotions will occur in which case integral values with sizes smaller than `int` will be promoted to `int`.

The –fsource-enc=utf8 and –fsource-enc=ascii options (Select source character encoding)

The *–fsource-enc=utf8* option causes **DCC** 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.

The –fgcc-version=*ver* option (Set a specific GCC version compatibility target)

In Linux (*–flinux*) and z/TPF (*–fztpf*) modes, **DCXX** is mostly compatible with **gcc**, so that it can use the system headers and libraries. Use *–fgcc-version=x.y.z* to specify compatibility with a specific version of **gcc**. The default compatibility level is *–fgcc-version=4.1.2*.

The only version-specific difference supported by **DCC** is bitfield packing of `char` types improved at **gcc** version 4.4.0.

The –Wswitch-outside-range and –Wno-switch-outside-range options (check case label range)

When *–Wswitch-outside-range* is enabled, the compiler will generate a warning if the constant on a case label is outside of the range allowed by the type of the enclosing switch-statement.

The compiler determines the allowed range before default integral promotions are performed on the controlling expression for the switch. A range can be expanded by adding an explicit cast to a larger type, for example, explicitly casting a `(char)` expression to an `(int)` would indicate the range is that of an `(int)`.

The check can be disabled with *–Wno-switch-outside-range*, *–Wswitch-outside-range* is the default.

The optWswitch and Wno-switch options (check enumerations in switch)

When *–Wswitch* is enabled, and the controlling controlling expression of a switch statement is an enumerated type, the compiler will verify that all the enumeration values appear as case labels in the switch block.

If any are missing and there is no `default:` label, the compiler will generate a warning.

The *–Wno-switch* option will disable this check, and will also disable the *–Wswitch-enum* check as if *–Wno-switch-enum* had been specified.

The –Wswitch-enum and –Wno-switch-enum options (check enumerations in switch)

The *–Wswitch-enum* option is similar to the *–Wswitch* option, except that the presence of a `default:` label does not alter the check.

When *–Wswitch-enum* is enabled, and the controlling controlling expression of a switch statement is an enumerated type, the compiler will verify that all the enumeration values appear as case labels in the switch block.

If any are missing the compiler will generate a warning, regardless of the presence of a `default:` label.

The *–Wno-switch-enum* option will disable this check, and will also disable the *–Wswitch* check as if *–Wno-switch* had been specified.

The –Wlabel-unused and –Wno-label-unused options (check for unused statement labels)

When *–Wlabel-unused* is enabled, the compiler generates a warning if a label is defined but not used by the end of the scope.

This option is enabled by default.

The –Wunused-parameter and –Wno-unused-parameter options (check for unused function parameters)

When *–Wunused-paramater* is enabled, the compiler generates a warning if a function defines a parameter, but the parameter is not used within the body of the function.

This option is disabled by default.

The –Wunused-variable and –Wno-unused-variable options (check for unused variables)

When *–Wunused-variable* is enabled, the compiler generates a warning if a declared variable is not used within the body of its scope.

This option is disabled by default.

This option is also enabled by the *–Wall* option.

The –Wunused-function and –Wno-unused-function options (check for unused static functions)

When *–Wunused-function* is enabled, the compiler generates a warning if a static function is defined, but is not used in the compilation.

This option is disabled by default.

This option is also enabled by the *–Wall* option.

The –Wincompatible-pointer-types and –Wno-incompatible-pointer-types options (pointer conversion to incompatible types warning)

The *–Wincompatible-pointer-types* causes the compiler to generate warnings for conversions between pointers to incompatible types. These warnings can be disabled with *–Wno-incompatible-pointer-types*.

This option is enabled by default, and can also be enabled by the *–Wall* option.

The –Wdiv-by-zero and –Wno-div-by-zero options (generate division by zero warning)

The *–Wdiv-by-zero* option enables compile-time warnings for division by zero. A warning is only generated by integral divisions and Hexadecimal floating point divisions by zero. Divisions by zero for IEEE and Decimal floating point are valid approaches to generating a NaN value and do not produce the warning.

The is option is enabled by default.

Assembling the output

For traditional mainframe operating systems (MVS, z/OS, etc...) **DCC** generates HLASM-style assembly code which is assembled with either IBM's HLASM program, or 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".

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

Using HLASM

If the intended assembler is IBM's HLASM assembler, then the *–fhlasm* option should be added to the **DCC** command line. The *–fhlasm* option causes the compiler to generate only statements accepted by the IBM HLASM assembler and use none of the Dignus **DASM** extensions. Some features of **DCC** can not be used if the IBM HLASM assembler is employed to assemble the compiler-generated assembly source.

To use HLASM to assemble the generated source, the XOBJECT option can be added to the HLASM parameters. XOBJECT will cause HLASM to generate a GOFF-format object file that can handle the long names typical of C programs. If your source contains no long names, and no defined file-scoped variables you may omit this option and produce OMF-style objects.

Note, to use HLASM maintenance must be up to PTF #UQ72970. If this maintenance is not applied, HLASM will incorrectly indicate errors in ALIAS statements, or produce incorrect output.

The following is an example of a typical JCL deck for an HLASM jobstep:

```
//ASM   JOB
//ASM EXEC PGM=ASMA90,
//         REGION=2M,PARM='XOBJECT,LIST(133)'
//STEPLIB DD DSN=Systems/ASM load library,DISP=SHR
//SYSLIB DD DSN=CEE.SCEEMAC,DISP=SHR
//       DD DSN=SYS1.MACLIB,DISP=SHR
//SYSUT1 DD DSN=&&SYSUT1,
//         SPACE=(4096,(120,120),,,ROUND),UNIT=O,
//         DCB=BUFNO=1
//SYSPUNCH DD SYSOUT=B
//SYSPRINT DD SYSOUT=*
//SYSIN DD DSN=INPUT.SOURCE(MEMBER),DISP=SHR
```

```
//SYSLIN DD DSN=OUTPUT.OBJECT(MEMBER),DISP=OLD,
//          DCB=(BLKSIZE=3120,LRECL=80,RECFM=FB)
```

Note that if HLASM is invoked with the XOBJECT option enabled, the resulting object will be in IBM's GOFF format and cannot be used with the IBM pre-linker. Unless the Systems/C pre-linker is used, the IBM BINDER will be required to link the resulting object. Furthermore, if the object contains any external identifiers which are inappropriate for a PDS, the BINDER will issue a message and always end with a return code of 8. In this case, the result of the BINDER step should be written to a PDSE or the HFS. Alternatively, the `LET` option can be added to the BINDER, which will allow the object to be written to a PDS, but it will not affect the return code and will require inspection of the BINDER output to ensure all references have been correctly resolved.

Another approach is to use the Systems/C **PLINK** pre-linker's *–p* option to process the generated file, shortening the names. When *–p* is used with **PLINK**, all GOFF and XSD names longer than 8 characters will be reassigned unique shorter names.

Using Systems/ASM

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

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 UNIX as shown below.

```
dasm -o my.obj myfile
```

Systems/ASM can also be used on OS/390 or z/OS to assemble compiler-generated assembler source.

For more information, consult the *Systems/ASM* manual.

Linking Assembled objects on OS/390 or z/OS

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

On cross-platform hosts, where the Systems/ASM assembler is used to produce objects, the Systems/C pre-linker, **PLINK**, can create a TSO TRANSMIT file containing the resulting load module. Or, the objects can be transferred to OS/390 or z/OS via FTP or some other binary-mode transfer mechanism.

To learn more about using **PLINK** to produce load modules on cross-platform hosts, consult the *Systems/C Utilities* manual.

Systems/C contains two versions of the Systems/C library — the RENT version for generating re-entrant programs and the non-rent version for generating non-re-entrant programs.

If the source were compiled with the *–frent* option, the RENT library should be employed to produce a re-entrant load module. This will require using the Systems/C pre-linker **PLINK** during the link step.

If no source was compiled with the *–frent* option, then the non-rent library should be used. In that case, it is not necessary to use 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 or z/OS 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.

When a C source file is compiled with the *–frent* option, 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.

The IBM linker gathers all of the `Q-CON` references together allocating an entry for each in the **PRV**. The **PLINK** utility will also perform this function, which is useful on older platforms using the older IBM linker (e.g. VSE, VM/ESA and MVS 3.8.) The older IBM linker does not process `Q-CON` references correctly, and **PLINK** will be required.

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.

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, **DCC**, produces 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 Systems/C 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 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, producing a single file which can then be processed by the IBM BINDER or older IEWL linker.

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.

The output of **PLINK** can then processed by the IBM LINKER or BINDER to produce the executable load module. On cross-platform hosts, **PLINK** can also perform this step, to produce a TSO TRANSMIT file which can be `RECIEVE`'d on the mainframe host.

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 library names to locate any required library objects.

For example, on a Windows platform the command:

```
plink "-SC:\sysc\lib\objs_rent\&M" prog.obj
```

will read the initial input file, `prog.obj` and examine 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../mylibs -ltwo
```

will read the two primary input objects `t1.obj` and `t2.obj`. It will try and resolve references from the **DAR** archive `libone.a` and then the second **DAR** archive `../mylibs/libtwo.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:SYSMOD` 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.LIBCR.OBJ` as the autocall library:

```
//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.LIBCR.OBJ,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.

For information about running Systems/C programs under the OpenEdition shell, see the *Systems/C C Library* manual.

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.

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 OS/390 or z/OS

Before execution, programs must be prepared, optionally using the Systems/C prelinker, **PLINK**, then the IBM BINDER.

Systems/C provides two versions of the Systems/C C library, one for RENT programs and one for non-RENT programs. If you are using the Systems/C library, it is important to link with the appropriate version. If any source programs reference variables found in the Systems/C library (e.g. `errno`) and that program was compiled with the *–frent* option, then the re-entrant version of the Systems/C library 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 Systems/C library. Normally the Systems/C library is specified as the last library to use for AUTOCALL resolution in the **PLINK** step. Furthermore, **PLINK** must be used for re-entrant programs that use the Systems/C library or to take advantage of **DAR** archive 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.LIBCR.OBJ,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 C reentrant library. 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

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 library contains 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.

DCC Advanced Features and C Extensions

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

Predefined macros

The Systems/C compiler defines the mandatory macros defined by the C standards. These are:

`__DATE__`	The date of the compilation as a string literal in the form "*Mmm dd yyyy*".
`__FILE__`	The name of the current source file as a character string.
`__LINE__`	The current line number within the current source file.
`__STDC__`	The integer constant 1.
`__STDC_HOSTED__`	Defined it the *–fc99* or *–fc11* options are enabled. If *–ffreestanding* is enabled, its value will be the integer constant 0, otherwise it will be the integer constant 1.
`__STDC_HOSTED__`	The integer constant 1 if *–fc99* or *–fc11* options are set.
`__STDC_VERSION__`	The integer constant 199901L if *–fc99* or *–fc11* options are set.
`__TIME__`	The time of the compilation, as a string literal of the form "*hh:mm:ss*".

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.

`__DFP__`	`__DFP__` is defined to `1` if the *–fdfp* option was specified. This indicates that decimal floating point values are supported.
`__CHAR_UNSIGNED__`	`__CHAR_UNSIGNED__` is defined to `1` if the `char` data type is unsigned by default. This is the typical mode of compilation.
`__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.
`__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__`	`__SYSC_ANSI_BITFIELD__` 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 was specified. This indicates that the compilation is targetted for the 64-bit z/Architecture.
`_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.
`__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 *–mlp64* option is not enabled. (The *–milp32* option turns *–mlp64* off.) This indicates that pointers and the types, `int` and `long` are 32-bits wide.
`_ILP32`	`_IPL32` is defined to the value `1` when the *–mlp64* 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.

__int8, __int16, __int32, __int64

DCC 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. The ANSI C standard declares more appropriate types in the `<inttypes.h>` header.

__grande and __regpair long long type modifiers

DCC provides support for operating on `long long` (64-bit) data in either 64-bit "grande" registers or two 32-bit registers.

These modifiers are particularly useful for 32-bit environments where it is desirable to use the 64-bit registers for 64-bit operations instead of the default 32-bit registers.

When *–mlp64* is enabled, `long long` data will use 64-bit registers for all operations and function return values, unless the `__regpair` modifier is applied.

For 32-bit environments, the compiler assumes that the high-order-word of 64-bit registers is not maintained across function calls, thus any value retained in a `long long __grande` variable will be saved before the function call and restored afterwards.

`long long` 64-bit operations are performed in the default mode, which is controlled by the *–fllgrande* option.

If *–fllgrande* is enabled, then any `long long __repair` values will be converted to `long long __grande` values for the operation. Similarly, if *–fllgrande* is not enabled any `long long __grande` values will be converted to `long long __regpair` to perform the operation.

For conditional expressions, if both the 2nd and 3rd operand are the same type, the result is that type, otherwise the operands are converted to the default type.

The `__grande and __regpair` modifiers only apply to `long long` data types.

`long long __grande` data requires z/Architecture hardware support.

ISO/IEC TS 18661-3:2015 floating point interchange and extended types

DCC supports the types described in the ISO/IEC TS 18661-3:2015 document, the floating point interchange types `_Float32`, `_Float64` and `_Float128` as well as the extended `_Float32x` and `_Float64x` types. **DCC** also supports the constant suffixes "`F32`", "`F32X`", "`f32`", "`f32x`", "`F64`", "`F64X`", "`f64`", "`f64x`", "`F128`", "`f128`" to indicate constants of the various types.

These are always IEEE values.

Operations involving those values will be accomplished using IEEE floating point instructions. And, conversions follow the rules described in the ISO/IEC TS 18661-3:2015 document.

DCC does not yet support the `_Decimal` versions of the extended types.

For more information, consult ISO/IEC TS18661-3:2015.

_Ieee and _Hexadec type modifiers

DCC provides support for both IBM hexadecimal and IEEE binary floating point numbers in the same compilation. A specific floating point type can be described using the `_Ieee` or `_Hexadec` type modifiers.

For example:

```
double _Ieee ivar;
```

declares `ivar` to be an 8-byte binary floating point variable.

Similarly,

```
float _Hexadec fvar;
```

declares `fvar` to be an 4-byte hexadecimal floating point variable.

The *–fieee* option controls the default type used for floating point operations and default promotions.

If *–fieee* is enabled then binary floating point arithmetic will be used and any hexadecimal values will be converted to binary values of the same size to accomplish the operation. Similarly, if *–fieee* is not enabled, then hexadecimal floating point

arithmetic will be used, and any binary values will be converted to hexadecimal values of the same size to accomplish the operation.

The *–fieee* also controls the type used for default promotions, which are used when invoking a function where no prototype is in scope, or a function with a variable argument list. If *–fieee* is enabled, then hexadecimal value arguments will be converted to binary values. If *–fieee* is not enabled, then binary value arguments will be converted to hexadecimal values.

For conditional expressions, if the types of the 2nd and 3rd operands are not the same, then the *–fieee* option controls the result type. For example, in this expression:

```
int i;
double _Ieee b1, b2;
double _Hexadec h;
...
h = i ? b1 : b2;
```

`b1` and `b2` are of the same time, so the type of result of the conditional expression is `double _Ieee`. This result would then be converted to a hexadecimal floating point value for assignment to `h1`. But, in this example:

```
int i;
double _Ieee b1;
double _Hexadec h1;
double d;
...
d = i ? h1 : b1;
```

the 2nd and 3rd operand types do not match. If the *–fieee* option is enabled, then `h1` will be converted to a binary `double`. If the *–fieee* option is not enabled, then `b1` will be converted to a hexadecimal `double`.

The `_Ieee` and `_Hexadec` type modifiers only apply to the floating point types, `float`, `double`, `long double` and `__float128`.

`__float128` floating point type

DCC supports the tt __float128 type which is a 128-bit floating point value. It is always 128 bits regardless of the setting *–flong-double-64* compiler option.

The type may be IEEE or Hexadecimal depending on the setting of the *–fieee* option, or the `_Ieee` and `_Hexadecimal` modifiers can be used at the declaration to indicate a preference.

__attribute__

DCC supports the `__attribute__` extension found in the gcc compiler. 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.

`alias` attribute

`__attribute__((alias("name")))` applies to declarations of symbols, and provides the name of another symbol which will provide the actual definition. For example:

```
void foo(void) { }
void __attribute__((alias("foo"))) bar(void);
```

In this example, `foo` would be defined as a regular function, and then the declaration of `bar` would produce a new symbol `bar` which would just refer to the same defined function `foo`.

Note that on many platforms there is no way to make an alias of a reentrant symbol.

`aligned` attribute

The `__attribute__((aligned(n)))` applies to a variable, a structure field member, or a type.

When applied to a variable, it specifies the minimum alignment requested for the variable and similarly for a structure field member.

For example:

```
struct aligned_struct  int x[2] __attribute__((aligned(8)));  ;
```

causes the array `x` to request 8-byte alignment. Furthermore, as that field is 8-byte aligned, the entire structure will be 8-byte aligned.

The aligned attribute can also apply to types, as in:

```
struct S  short f[3];  __attribute__((aligned(8)));
```

In this situation, the array structure member `f` would require 6 bytes, as each `short` requires 3 bytes. But the entire `struct S` structure type would require an 8-byte alignment.

The aligned attribute can only be used to increase the alignment, never reduce it. The `_Packed`, `#pragma pack`, or the `pack` attribute can be used to reduce alignments.

`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");
}
```

`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 a warning message, 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.

An optional constant string message may also be specified, in which case the syntax is `__attribute__((deprecated(`*msg*`)))`. If the *msg* is specified it will be included in any generated warning message.

`unavailable` attribute

The `__attribute__((unavailable))` attribute can appear after a declaration of a function, variable or typedef. Subsequent uses of the declared symbol will cause the compiler to generate a error message, indicating the symbol is not availble. 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.

This can be useful when a previously support, or deprecated interface is no longer supported.

An optional constant string message may also be specified, in which case the syntax is `__attribute__((unavailable(`*msg*`)))`. If the *msg* is specified it will be included in any generated warning message.

`mode` attribute

The `__attribute__((unavailable))` attribute can

`__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;
```

`noinline` attribute

The `noinline` attribute applies to function definitions and indicates the function should not be inlined when compiling with optimization enabled.

`noreturn` attribute

The `noreturn` attribute applies to function declarations and indicates the given function does not return to its caller. For example, several standard C library functions, such as `abort` and `exit` do not return to their caller.

The `noreturn` attribute allows the compiler to assume that code after the function call is unreachable. This can improve optimizations and messages.

The `noreturn` attribute does not affect the exceptional path when it applies, a function marked with `noreturn` may still return from the caller by throwing an exception or calling `longjmp`.

It does not make sense for a `noreturn` function to have a return type other than `void`.

`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.

`used` attribute

`__attribute__((used))` applies to function definitions, and indicates that the given function is used and should not be elided by the compiler, even though it may not appear to be referenced.

This is helpful for a static functions that are referenced from in-line assembly code.

`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 */
   }
}
```

`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__

DCC 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.

_Packed Qualifier

The `_Packed` qualifier may appear on structure or union definitions. It specifies that data elements within a structure/union be aligned on 1 bit boundaries instead of their normal alignment. That is, no inter-element padding will be introduced between data elements, the elements will be packed together. All structure/union elements will be on byte-aligned boundaries. Although `_Packed` may appear in any type, it is only effective on structure or union definitions. `_Packed` applied to a structure declaration has no effect.

This not only alters the inter-element alignment, but affects the size of the entire structure/union. In C, structures have an alignment which is the maximum alignment required of any data element. The `_Packed` keyword causes the maximum alignment to be 1 bit, thus making the entire structure alignment 1 bit.

Because structure data elements in a packed structure do not fall on their usual aligned boundaries, access to these elements via `.` and `->` may be slower.

`_Packed` affects only the first-level data elements of a structure. Structure and unions within a structure are not affected.

For example, in the following structure definition:

```
_Packed struct packed_tag {
    short two_byte_integer;
    double eight_byte_double;
} packed_struct;
```

The field `two_byte_integer` in the variable `packed_struct` will begin at offset 0 in the structure, and the field `eight_byte_double` will begin at byte offset 2, instead of its normal, aligned offset of 8. Furthermore, the size of this structure will be 10 bytes, instead of its normal aligned size of 16.

The `_Packed` qualifier applies to the definition of a structure or union, not the declaration. Thus, the same structure type may be access with or without the `_Packed` qualifier. Packed and non-packed versions of the structure will have different storage layouts.

The `_Packed` qualifier is meaningful for parameter types and structure or union assignments. Parameters must match in terms of the `_Packed` qualifier when a prototype for the function is in scope. Also, `_Packed` and non-packed versions of the same structure may not be assigned to each other.

`_Packed` will only alter alignment when used on structure or union definitions.

Anonymous Structures

Anonymous structures are an extension present in the Microsoft C compiler. Anonymous structures are not part of the ANSI C standard.

Anonymous structures are disabled by default, but can be enabled with the *–fanonstruct* compiler option, or using the `#pragma anonstruct` pragma.

When anonymous structures are enabled, a structure or union variable can be declared within another structure or union without giving it a name. The members of the inner structure or union can be directly accessed as if they were members of the outer structure or union.

For example:

```
/* Example of an anonymous structure */
struct phone
{
  int areacode;
  long number;
}

struct person
{
  char name[30];
  char sex;
  int age;
  int weight;
  struct phone; /* Anonymous structure, no name needed */
```

```
} Jim;

Jim.number = 1234567;
```

type-generic expressions

When the *–fc11* option is enabled, the compiler supports the ANSI C11 type-generic expression facility.

A type-generic expression is not a C expression that involves data, but rather it involves the types of data. The keyword `_Generic` is used to indicate a type-generic expression. In general, these are similar to a "switch-statement" for types. The syntax is:

generic-selection: `_Generic (` *assignment-expression* `,` *generic-assoc-list* `)`

generic-assoc-list: *generic-association*
generic-assoc-list `,` *generic-association*

generic-association: *type-name* `:` *assignment-expression*
`default :` *assignment-expression*

The first *assignment-expression* is called the "controlling expression", it is not evaluated. Rather it's type is compared with the types in each of the *generic-association*s. If the type is compatible, then the given *assignment-expression* is evaluated.

For example, a type-generic `cbrt` macro might be written as:

```
#define cbrt(X) _Generic((X),       \
                long double: cbrtl,   \
                default: cbrt,        \
                float: cbrtf          \
                )(X)
```

If the type of `X` is `long double` then `cbrtl` would be invoked, if it is `float` then `cbrtf` would be invoked, otherwise `cbrt` is invoked. All are passed the argument `X`.

static assertions

When the *–fc11* option is enabled, the compiler supports the ANSI C11 `_Static_assert` declaration. This declaration is used to accomplish a compile-time assertion, that if false, causes an error message.

A `_Static_assert` declaration has the syntax:

`_Static_assert (` *constant-expression* `,` *string-literal* `) ;`

Where *constant-expression* is a compile-time integral constant expression, and *string-literal* is a compile-time string literal.

The compiler will evaluate the *constant-expression*, if it has the value `0` then an error message is produced that includes the text from *string-literal*.

For example:

```
_Static_assert( sizeof(int) == 4, "sizeof(int) must be 4");
```

will produce a compile-time error diagnostic if the size of `int` is not `4`.

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, regardless of the *–frent* or *–fno-rent* option settings.

When *–frent* is enabled, 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.

The __inline keyword

When the `__inline` keyword is specified for a function, it instructs the compiler that this function is a candidate for inlining. The *–finline* option can be used to fine tune function inlining, including instructing the compiler to attempt to inline functions without the `__inline` keyword.

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 C 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.

Statement Expressions

Statement expressions are a gcc extension supported by Systems/C. A statement expression allows for the general semantic power of a compound statement to be used within an expression. This can be especially powerful when combined with **DCC**'s in-line assembly feature.

A statement expression is a compound statement enclosed in parentheses and may appear anywhere an expression may be used. The compound statement within a statement-expression may contain loops, switchs, local variables.

The value of a statement expression is the value of the last expression within the compound statement.

For example, the `max` macro is typically defined

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

This macro evaluates the macro arguments `a` and `b` more than once. If these expressions contain side effects, then unexpected results may occur.

Using statement expressions, an `int` version of this macro could be defined that evaluated its operands only once:

```
#define maxint(a,b) \
    ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

(Note that the variable names `_a` and `_b` were used to avoid conflicts with any potential user-defined identifiers.)

The value of this statement expression would be the value of the last expression, which in this example is the conditional expression comparing `_a` and `_b`.

__typeof__ operator

The syntax of `__typeof__` is similar to `sizeof`. The the result of `__typeof__` is a type that can be used anywhere a `typedef`'d type could be used.

The operand of `__typeof__` can be either a type or an expression.

For example, this declares `y` to be the type that `x` points to:

```
__typeof__ (*x) y;
```

`__typeof__` can be useful in constructing macros that operate regardless of the type of their parameters.

__bit_sizeof and __bit_offsetof operators

DCC supports two operators to determine the bit size and offset of structure fields:

`__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 structure field 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);
```

Binary constants with the '0b' prefix

Integer constants can be expressed in binary form, a sequence of `0` and `1` values when the `0b` or `0B` prefix is used.

For example:

```
i = 0b101010;
```

places the value 42 (decimal) into the variable `i`.

A binary constant is of `(unsigned int)` type unless it's type is explicitly specified by the optional `L`, `LL`, `U`, `UL` and `ULL` suffixes.

Omitted operand in conditional expressions

DCC supports omitting the second operand of a conditional expression. If the second operand (the "true" expression) is omitted, then the first operand (the "test" expression) is used.

For example, the expression:

```
a ? : b
```

has the value of '`a`' if '`a`' is non-zero. The expression has the value of '`b` if '`a` is zero.

This is equivalent to:

```
(t = (a)) ? t : b
```

Note that '`a`' is only evaluated once, and a temporary placeholder is used for the "true" expression. This can be very useful in writting macro definitions where multiple evaluations of a macro operand must be avoided. Or, when side-effects need to be considered. The test operand's value is not recomputed, but saved in a temporary.

Local labels

Local lables provide a mechanism for defining a label with a local lexical scope. The C standard defines labels as having function scope, thus two labels cannot have the same name. Using local labels, the same label name can be used in an inner lexical scope without conflict. The name is only valid until the end of the scope.

This facility can be very useful in writing macros that need to generate labels.

Local labels are declared using the `__label__` statement. A `__label__` statement can only appear at the start of a lexical scope, before any declarations or statements.

A `__label__` statement begins with the `__label__` keyword followed by a comma-separated list of label identifiers and an ending semicolon. For example:

```
__label__ a, b;
```

In the following example, two local labels of the same name are declared in two distinct inner blocks:

```
void func(void)
{
   {
     __label__ x;
     goto x;
     x:;
   }
   {
     __label__ x;
     goto x;
     x:;
   }
}
```

Using local labels can be helpful in macros where a label may be needed but can't conflict with a label in the function; or if the same macro is intended to be invoked many times in a given function.

__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 macros and functions

The C compiler supports several builtin preprocessor macros and builtin functions.

__has_builtin (ıoperand)

`__has_builtin` is a predefined C preprocessor macro that determines if the C compiler recognizes ıoperand as a builtin function or identifier. If it is recognized, it evaluates to a constant one, otherwise it is zero. It is a predefined macro, and can be used in preprocessor expressions, and its availability can be determined with `#if defined(__has_builtin)` .

__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_isdigit

`void __builtin_isdigit(int)` Implements the isdigit() function directly. Note that the C standard requires that the digit characters be consecutive starting at the '0' character, and that they are not affected by the locale setting. Thus, `__builtin_isdigit` can safely be expanded by the compiler.

If the argument value is constant, this produces a constant result.

__builtin_memcpy

`void *__builtin_memcpy(void *dest, const void *src, size_t len)` Implements the C standard `memcpy` function.

__builtin_mempcpy

`void *__builtin_mempcpy(void *dest, const void *src, size_t len)` Implements the POSIX standard `mempcpy` 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_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_frame_address

`__builtin_frame_address` returns the address of the frame for the current function.

`__builtin_frame_address` accepts one integer argument, which specifies the frame to examine. A value of `0` indicates the current frame, a value of `1` indicates the previous frame, etc...

Calling the function with a non-zero argument is not supported, because the compiler can't ensure that the caller has the required frame environment for walking back. Thus, the function is limited to only returning the frame for the current function.

__builtin_return_address

`__builtin_return_address` returns the return address of the current function, or one of its callers. `__builtin_return_address` accepts one integer argument, which indicates the number of frames to scan backward looking for a return address.

In LINUX mode, `__builtin_return_address` can only provide the return address of the current function, and will only accept a constant zero as its argument.

In other modes, `__builtin_return_address` assumes standard system linkage (`R13` addresses either a 31-bit or 64-bit standard save area) when walking back the prescribed number of frames.

Invoking `__builtin_return_address` with anything other than a nonzero argument can have unpredictable results, depending on how the current function was invoked.

__builtin_extract_return_address

`__builtin_extract_return_address` is used to "clean up" the value returned from `__builtin_return_address`. `__builtin_extract_return_address` accepts a `void *` parameter and returns `void *`.

On some architectures and environments, the value from `__builtin_return_address` can contain extra information. For instance, in AMODE 31 on the z/Architecture, the AMODE bit will be set. `__builtin_extract_return_address` will clear that bit to provide an absolute address; only for 31-bit compilations.

In all other situations `__builtin_extract_return_address` simply returns the unaltered pointer parameter.

__builtin_stpcpy

`int __builtin_stpcpy(char *dest, const void *src)` Implements the POSIX standard `stpcpy` function.

__builtin_strcpy

`int __builtin_strcpy(char *dest, const void *src)` 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_stpncpy

`char * __builtin_strncpy(char *dest, const char *src, size_t len)` Implements the POSIX standard `stpncmp` 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 `strpbrk` 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_clz

`int __builtin_clz(unsigned int)` Returns the count of leading zero bits in the parameter. Returns the parameter size (in bits) if all bits are zero.

__builtin_clzl

`int __builtin_clzl(unsigned long)` Returns the count of leading zero bits in the parameter. Returns the parameter size (in bits) if all bits are zero.

__builtin_clzll

`int __builtin_clzll(unsigned long long)` Returns the count of leading zero bits in the parameter. Returns the parameter size (in bits) if all bits are zero.

__builtin_ctz

`int __builtin_ctz(unsigned int)` Returns the count of trailing zero bits in the parameter. Returns the parameter size (in bits) if all bits are zero.

__builtin_ctzl

`int __builtin_ctzl(unsigned long)` Returns the count of trailing zero bits in the parameter. Returns the parameter size (in bits) if all bits are zero.

__builtin_ctzll

`int __builtin_ctzll(unsigned long long)` Returns the count of trailing zero bits in the parameter. Returns the parameter size (in bits) if all bits are zero.

__builtin_ffs

`int __builtin_ffs(int)` Finds the first bit set (beginning with the least significant bit) in the parameter and returns the index of that bit. Returns 0 if no bits are set.

__builtin_ffsl

`int __builtin_ffsl(long)` Finds the first bit set (beginning with the least significant bit) in the parameter and returns the index of that bit. Returns 0 if no bits are set.

__builtin_ffsll

`int __builtin_ffsll(long long)` Finds the first bit set (beginning with the least significant bit) in the parameter and returns the index of that bit. Returns 0 if no bits are set.

__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)` Similar to `__builtin_huge_valf()` but returns a `float` value.

__builtin_huge_vall

`long double __builtin_huge_vall(void)` Similar 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)` Similar to `__builtin_inf()` but returns a `float` value.

__builtin_infl

`long double __builtin_infl(void)` Similar to `__builtin_inf()` but returns a `long double` value.

__builtin_infd32

`_Decimal32 __builtin_infd32(void)` Similar to `__builtin_inf()` but returns a `_Decimal32 +Inf` value.

__builtin_infd64

`_Decimal64 __builtin_infd64(void)` Similar to `__builtin_infd32()` but returns a `_Decima64` value.

__builtin_infd128

`_Decimal128 __builtin_infd128(void)` Similar to `__builtin_infd32()` but returns a `_Decima128` 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 *)` Similar to `__builtin_nan()` but returns a `float` value.

__builtin_nanl

`long double __builtin_nanl(const char *)` Similar to `__builtin_nan()` but returns a `long double` value.

__builtin_nand32

`_Decimal32 __builtin_nand32(const char *)` Similar to `__builtin_nan()` but returns a `_Decimal32` value.

__builtin_nand64

`_Decimal64 __builtin_nand64(const char *)` Similar to `__builtin_nan()` but returns a `_Decimal64` value.

__builtin_nand128

`_Decimal128 __builtin_nand128(const char *)` Similar to `__builtin_nan()` but returns a `_Decimal128` 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 *)` Similar to `__builtin_nans()` but returns a `float` value.

__builtin_nansl

`long double __builtin_nansl(const char *)` Similar to `__builtin_nans()` but returns a `long double` value.

__builtin_abort

_Noreturn void __builtin_abort(void) The __builtin_abort() builtin invokes the abort() function. It is declared with the _Noreturn function specifier to indicate that the abort() function does not return to the caller.

__builtin_unreachable

_Noreturn void __builtin_unreachable(void) The __builtin_unreachable() builtin indicates that the section of code cannot be reached during program execution.

__builtin_trap

_Noreturn void __builtin_trap(void) The __builtin_trap() builtin either generates an instruction that causes a program exception during execution, or it invokes the abort() function.

integer overflow builtins

As well as the specific builtins previously listed, the C compiler supports the GNU integer __builtin overflow functions.

These functions perform integer operations while indicating if the operation overflowed when computing the result.

The basic overflow checking builtin functions are:

```
bool __builtin_sadd_overflow(int, int, int *);
bool __builtin_saddl_overflow(long, long, long *);
bool __builtin_saddll_overflow(long long, long long, long long *);
bool __builtin_uadd_overflow(unsigned int, unsigned int, unsigned int *);
bool __builtin_uaddl_overflow(unsigned long, unsigned long, unsigned long
*);
bool __builtin_uaddll_overflow(unsigned long long, unsigned long long, unsigned
long long *);
bool __builtin_ssub_overflow(int, int, int *);
bool __builtin_ssubl_overflow(long, long, long *);
bool __builtin_ssubll_overflow(long long, long long, long long *);
bool __builtin_usub_overflow(unsigned int, unsigned int, unsigned int *);
bool __builtin_usubl_overflow(unsigned long, unsigned long, unsigned long
*);
bool __builtin_usubll_overflow(unsigned long long, unsigned long long, unsigned
long long *);
bool __builtin_smul_overflow(int, int, int *);
```

```
bool __builtin_smull_overflow(long, long, long *);
bool __builtin_smulll_overflow(long long, long long, long long *);
bool __builtin_umul_overflow(unsigned int, unsigned int, unsigned int *);
bool __builtin_umull_overflow(unsigned long, unsigned long, unsigned long
*);
bool __builtin_umulll_overflow(unsigned long long, unsigned long long, unsigned
long long *);
```

Along with these, the compiler supports the generic functions:
`bool __builtin_add_overflow(`*type1*`, `*type2*`, `*type3*` *);`
`bool __builtin_sub_overflow(`*type1*`, `*type2*`, `*type3*` *);`
`bool __builtin_mul_overflow(`*type1*`, `*type2*`, `*type3*` *);`
where *type1*, *type2* and *type3* can be any signed or unsigned integral types, including `__int128` types and types smaller than `int`.

For determing if an operation would overflow, while discarding the result, the compiler also supports these predicate builtins:
`bool __builtin_add_overflow_p(`*type1* `a, `*type2* `b, `*type3* `c);`
`bool __builtin_sub_overflow_p(`*type1* `a, `*type2* `b, `*type3* `c);`
`bool __builtin_mul_overflow_p(`*type1* `a, `*type2* `b, `*type3* `c);`
where the type of the expression *c*, *type3*, is used to determine if the operation overflows, but the value of *c* is not computed. *c* is simply an expression used to specify the result type.

For the generic functions, the operation is computed as if it were performed with infinite precision and then stored in the result type.

__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`), **DCC** 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.

64-bit integral arithmetic — long long

DCC supports both the `long long` and `unsigned long long` data types. When *–mlp64* is not specified, `long long` and `unsigned long long` are 64 bits (8 bytes), with a 4-byte, or fullword alignment. When *–mlp64* is specified, `long long` and `unsigned long long` are equivalent to `long` and `unsigned long` respecitively, and are 64-bits in size, with 8-byte or doubleword alignment. All of the integral operations are supported on these 64-bit data types. The `long long` and `unsigned long long` data types are not present in the ANSI C89 standard, but were defined in the ANSI C99 standard.

When *–milp32* is specified, the compiler typically uses two registers to implement the various arithmetic operations. 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.

When *–mlp64* is specified, `long long` data is implemented identically to `long` data, typically in one 64-bit register, and functions that return `long long` data do so in register 15.

If the *–fc99* option is not enabled, `long long` and `unsigned long long` are considered an extension to the ANSI C89 standard, and are treated as extended integral types. Arithmetic promotions apply in the fashion dictated by the ANSI C89 standard. For example, if either the left hand side or the right hand side of an arithmetic operation is one of these types, the other side is converted to that type. Note that this applies to the shift operations as well. Per ANSI C89 rules, both sides of the shift operation participate in promotions. So, the value to shift as well as how much to shift, will be promoted to a `long long` or `unsigned long long`.

If the *–fc99* option is enabled, the `long long` data types follow the rules defined in the ANSI C99 standard.

DCC also supports extended `long long` integral constants. These can be specified with the `ULL` and `LL` suffixes. While the `ULL` and `LL` suffixes are defined in the newer ANSI C99 standard, they are considered ANSI C89 extensions if the *–fc99* option

is not enabled. To maintain compatibility with the ANSI C89 standard, when the *–fc99* option is not specified, **DCC** will issue a diagnostic if the value of the constant is too large to be contained in an ANSI C89 defined data types. **DCC** will then use the appropriate `long long` data type for the value. Adding the `ULL` or `LL` suffix will eliminate this warning. For example:

```
0x7fffffffffffff
```

will generate a diagnostic, while

```
0x7fffffffffffffLL
```

will not. In either case, the type of the constant will be `long long`.

If the *–fc99* option is specified, `long long` and `unsigned long long` constant types are supported as defined in the ANSI C99 standard. Furthermore, the promotion rules follow the newer ANSI C99 standard. Thus, if *–fc99* is specified, the code above will not generate a warning with or without the `LL` suffix.

128-bit integral arithmetic — __int128

DCC supports both the `__int128` and `unsigned __int128` data type extension. These are 128-bit integral values, which are supported for conversion, arithmetic, etc... In the 32-bit environment, significant code can be generated when using these types.

There is no support for 128-bit integral constants, so generation of a complete 64-bit constant requires shifting, for example:

```
unsigned __int128 x = (((unsigned __int128)0xffffffffffffffff) << 64) |
                       0xffffffffffffffff;
```

initializes the 128-bit value '*x*' to all `0xff` bytes.

Decimal floating point types

When the *–fdfp* option is specified, `DCC` supports the decimal floating point types as defined in the N1176 draft of ISO/IEC WDTR24732.

The decimal floating types are `_Decimal32`, `_Decimal64` and `_Decimal128`. Unlike hexadecimal or binary floating point values, these values use a radix of 10 instead of 16 (hexadecimal) or 2 (binary).

The compiler supports the arithmetic operations of add, subtract, multiply and divide; the unary arithmetic operators, relational operations and conversions to and from integral and floating point types.

`_Decimal32`, `_Decimal64` and `_Decimal128` values are treated similar to `float`, `double` and `long double` for the purposes of parameter passing and returned values.

To specify a decimal floating point constant, use the suffixes `df` or `DF` for `_Decimal32`, `dd` or `DD` for `_Decimal64`, or `dl` or `DL` for `_Decimal128` values. Note that the case of both letters must be the same, either both lower case or both upper case.

For example:

```
_Decimal32 d32;

d32 = 1.0df;
```

`DCC` doesn't completely support the draft technical report. In particular:

- The translation time data type (TTDT) is not supported.
- When converting a decimal floating point value to an integer type, if the decimal value cannot be represented the result is undefined. The draft and the IBM xlc compiler have different behavior, the GNU GCC compiler and the Dignus compiler have the same behavior.

ANSI C99 features

If the *–fc99* option is enabled, **DCC** supports several new language features defined in the ANSI C99 standard. The ANSI C99 standard describes these language features in more detail.

Currently, Systems/C supports a subset of the C99 standard, including the following features.

__func__ identifier

The `__func__` identifier expands into a reference to a local variable which is initialized with a string containing the current function's name.

Unlike the `__FUNCTION__` extension, each reference to `__func__` is guaranteed to point to the same address.

_Bool data type

The `_Bool` data type is fully supported when *–fc99* is enabled. The Systems/C library also includes the `<stdbool.h>` standard header file.

Mixed statements and declarations

In the 1989 version of the ANSI C standard, data declarations within inner blocks had to appear before statements, this restriction was removed in the 1999 version of the C standard. If *–fc99* is enabled, declarations may appear anywhere a statement may occur.

For example:

```
{
   int i;  // declare i;
   i = 10;
   int j;  // declare j;
   j = i + 10;
}
```

Declaration in `for` statements

In the 1989 version of the C standard, the initialization section of a `for` statement is defined as any normal assignment expression.

The 1999 version of the C standard allows for a declaration clause to appear in the initialization section.

For example, this is valid if the *–fc99* option is enabled:

```
int j;
j = 20;
  /* declare ‘i’ in the for-loop */
for(int i = 10; i<j; i++) {
  printf("i is %d\bs n", i);
}
```

The scope of any declarations in `for` statements continues through the end of the entire `for` body.

#pragma STDC FENV_ACCESS

The `#pragma STDC FENV_ACCESS` pragma is recognized and fully supported.

//-style comments

Systems/C recognizes the `//`-style comment by default.

`long long` data types

Systems/C supports the ANSI C99 `long long` data types and the `ULL` and `LL` constant suffixes.

If *–fc99* is enabled, Systems/C follows the C99 rules for evaluating integral constants, which allow for automatic promotion to the `long long` types.

If *–fc99* is not enabled, Systems/C follows the 1989 standard, and the `long long` data types are considered an extension. In this case, the compiler will produce warnings when a constant does not fit in the 1989 defined data types.

C99 preprocessor

The Systems/C preprocessor is fully comformant with the ANSI C99 definition, regardless of the *–fc99* option.

Some of the new extensions supported by the C preprocessor include:

- Variadic macros

  Systems/C supports the new variadic macro syntax, allowing multiple arguments to `#define`d macros.

- `_Pragma` operator

  Systems/C supports the `_Pragma` C preprocessor operator, allowing for macros that expand into `#pragma` statements.

- ANSI C99 digraphs

  As well as the C89 tri-graph characters, Systems/C fully supports the new C99 digraph characters

- 64-bit constants in #if expressions

  Integral evaluations of expressions in `#if` preprocessor commands are evaluated in terms of `intmax_t`, which is 64-bits.

Inline assembly language support

DCC supports a robust in-line assembly language 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.

In support of this feature, **DCC** 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 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 in-lined assembly source for uses of these registers. The compiler does not flag `__register` declarations using these registers.

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
`\#`	unique decimal value
`\d`	unique decimal value

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`, `\C`, `\#` and `bs d` 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. `\#` expands into a unique decimal value for each `__asm` block. `\#` and `\d` operate identically and can be very helpful in generating unique labels for branch targets. `\d` is provided for those situations where `\#` is cumbersome to use in a C macro environment. `\d` and `\#` are interchangeable.

The optional integral constant declares how many bytes the in-line 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, you could use the following block of C code:

```
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` wasn'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\
      } }
```

In this example, we use the `#` sequence to generate a unique label for every instance of the macro, in combination with the block-expression extension to return the value:

```
#define pound \#

#define here(x)  \
  (({ \
    __register(x) void *reg; \
    __asm {\
&hctr seta pound \n\
         LA x,@l&hctr  \n\
@l&hctr  DS 0h\n\
     } \
     reg; \
   }))
```

The `here()` macro in a code snippet as:

```
void *ptr;
 ...

ptr = here(1); /* get the current address */
               /* using R1 as the temp reg*/
```

where the compiler-generated code would be:

```
* inline ASM source (4 bytes)
&hctr seta 1
 LA 1,@l&hctr
@l&hctr DS 0h
```

each instance of the `here()` macro would place a new value in the `&hctr` assembler counter.

__asm("...":output:input:clobber) — GCC-style inline assembly source

As of version 2.0, **DCC** supports GCC-style inline assembly. If the `__asm` keyword is followed by a parenthesis, then **DCC** recognizes the GCC-style syntax instead of the "classic" syntax.

```
__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.

DCC 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 "`&`", **DCC** 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 **DCC** 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, **DCC** 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 **DCC**'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 — **DCC** 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");
```

Direct references to ASM values

DCC provides a mechanism for directly accessing assembly language values in C code, the `__asmref` macro and the `__asmval` built-in constant. Using these, a program may reference assembly language EQUs or fields within a DSECT.

`__asmref(base,asm-string,type)` — Reference a DSECT field

`__asmref` may be used to reference a field in a DSECT. Its three arguments specify the base address of the storage onto which the DSECT is to be mapped, the assembly-language expression that produces the offset in the DSECT of the field, and the C type that represents the field type.

To use `__asmref` you must `#include` the system header file `<machine/asmref.h>`.

The *base* value is any C value that can be used as a character pointer. Thus, constant expressions or any address or pointer expression is valid.

Typically, the *asm-string* value is an assembly expression subtracting the start of the DSECT from the field name.

The *type* field should be a C type, without surrounding parenthesis. Any C type that can be used in a cast expression is valid.

For example, if you have a DSECT named `MYDSECT` which is defined in the following via an in-line `__asm` directive:

```
__asm {
   MYDSECT DSECT
```

```
   FIELD1  DS    1F
   FIELD2  DS    1F
}
```

and, furthermore, there is a C variable named `mydsect_base` which is a pointer to the base of the storage associated with the DSECT, then the expression

```
__asmref(mydsect_base, "FIELD2-MYDSECT", int)
```

references the FIELD2 field in MYDSECT. This expression is an lvalue so the value in FIELD2 may either be retrieved or stored. That is, the statement:

```
__asmref(mydsect_base, "FIELD2-MYDSECT", int) = 5;
```

stores the integer value 5 into the FIELD2 field. The statement

```
i = __asmref(mydsect_base, "FIELD2-MYDSECT", int);
```

retrieves FIELD2 and places its value in the variable `i`.

Because `__asmref` is a run-time value, the compiler cannot determine at compile time the particulars of the *asm-string*. Thus an `__asmref` value cannot be used to initialize static data.

`__asmval(size,asm-string)` — reference an ASM-defined constant

`__asmval` is used to reference a value defined in assembly language source included in the generated source file, such as EQU values or any assembly language expression which produces an absolute value. An `__asmval` is treated as an unknown constant value by the compiler. It is an rvalue, and thus cannot be assigned to or have its address taken. Furthermore, since the value isn't known by the compiler, an `__asvmal` may not participate in a static initialization. Other than those restrictions, `__asmval` values may appear wherever an integral constant is allowed. `__asmval` values are of type `unsigned long`.

The first parameter is the expected size in bytes of the value, either 1, 2, 3, 4 or 8.

The second parameter is a string which contains the ASM expression defining the value.

The result type of an __asmval expression is `unsigned long` unless the *size* value is 8 and the *–milp32* option is specified. If *size* is 8, and *–mlp64* is not specified, then the type of the `__asmval` is `unsigned long long` (64-bits.)

When *–mlp64* is specified, the `unsigned long` type is 64-bits and can accomodate any sized `__asmval`.

For example, if the following EQU was defined in assembly language source:

```
MYVAL EQU 100
```

then that value may be retrieved by the compiler with the following `__asmval` expression:

```
__asmval(1,"MYVAL")
```

The statement:

```
i = __asmval(1,"MYVAL") + 20;
```

retrieves the value of MYVAL at run-time, adds 20 to it and stores the result in the variable `i`.

#pragma compiler directives

DCC supports several `#pragma` directives:

#pragma anonstruct (*switch*)

`#pragma anonstruct` is used to enable or disable support for the Microsoft anonymous structures extension.

Anonymous structures allow for unnamed inner structures or unions within an outer structure or union. The elements of the inner structure are then directly accessible as if they were elements of the outer structure.

The value of *switch* is one of `on`, `off` or `pop`. `on` enables recognition of anonymous structures, `off` disables it, and `pop` restores the previous setting.

Each use of `#pragma anonstruct` pushes the previous setting and sets the new value. A `#pragma anonstruct pop` can be used to restore the previous value.

A `#pragma anonstruct pop` used when no previous `#pragma anonstruct` was used resets the value to `off`.

For example:

```
  /* Enable anonymous structures */
#pragma anonstruct(on)
struct anon {
           struct {
                int i;
                int k;
           };
       } ;
  /* Restore the previous setting of anonymous structures */
#pragma anonstruct(pop)

struct anon A;

A.i = 10;
```

#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.)

#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.

If the range of enumeration values cannot fit into the number of bytes specified, the compiler will generate an error message.

`#pragma enum` is only useful if the *–fc370* option is specified. It is silently ignored if *–fc370* is not specified.

#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 error "*text*"

`#pragma error "text"` causes the compiler to generate an error message. The error message will include the specified `text`.

#pragma export(*identifier*)

This pragma is only meaningful in IBM C compatibility mode, when *–fc370* is specified.

This option causes the named function or data *identifier* to be exported from a DLL.

When the *–fc370* option is specified, this option is identical to the IBM option of the same name.

#pragma filetag("*codepage*")

The `#pragma filetag` pragma describes the source code character set on EBCDIC platforms. On ASCII platforms, it is silently ignored.

If the value of *codepage* is `IBM-500`, then the codepage 500 translations will be applied. See the –fcodepage500 option for a descrition of those translations.

Any other value for *codepage* will revert the compiler to its normal behavior.

#pragma linkage(*identifier*, *type*)

Specifies that function, or function typedef named *identifier* is to be invoked with the given linkage type, either **OS**, **PLI**, or **ALIGN4**.

`#pragma linkage(`*identifier*`,OS)` and `#pragma linkage(`*identifier*`,PLI)` apply to external function declarations. `#pragma linkage(`*identifier*`,ALIGN4)` may be specified for function definitions as well as external declarations.

Parameters to functions specified with **OS** linkage that are not pointers are passed as addresses to temporary copies of the actual arguments. Pointer-type parameters are passed directly to the function. 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. The last parameter will have the "*VL-bit*" set. This is the basic linkage convention used by the operating system. The compiler assumes that the function's return value will be in register 15. Register 0 will be set to zero before the function call. Before calling a function specified with **OS** linkage, the first 12 bytes of the local save area are saved, and are restored on return.

Parameters to functions specified with **PLI** linkage that are not pointers are passed as addresses to temporary copies of the actual parameter. 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. Pointer-type parameters are passed directly to the function. The last parameter will have the "*VL-bit*" set. An extra parameter is appended to the list which contains a pointer to any returned data. **PLI** designates an entry point as a PL/I linkage entry point.

When setting the "*VL-bit*" for **PLI** and **OS** linkage, the pointer value is logically OR'd with `0x80000000`.

ALIGN4 linkage is useful for programs compiled with with the *–mlp64* option. Normally, when compiling in z/Architecture mode (*–mlp64* enabled), parameters are aligned on 8-byte boundaries. However, the `#pragma linkage (`*identifier*`,ALIGN4)` pragma can be used to indicate the specified function's parameters should be aligned on 4-byte boundaries. Thus, the function can be invoked from a non-z/Architecture program. This is particularly useful for Direct CALL (DCALL) entry points compiled with the *–mlp64* that are intended to be invoked from a non-z/Architecture

environment. Also, as well as defining functions that can be called from non-z/Architecture environments **ALIGN4** linkage can be used to ensure that the outgoing parameter area for external functions is 4-byte aligned when *–mlp64* is enabled. When **DCC** invokes a function with **ALIGN4** linkage, it will ensure the outgoing parameters are 4-byte aligned. Note that in defined functions with **ALIGN4** linkage compiled with the *–mlp64* option enabled, variable argument list support will not operate, as the variable argument list macros defined in `<stdarg.h>` depend on 8-byte alignment when *–mlp64* is enabled. If *–mlp64* is not enabled, **ALIGN4** linkage has no effect.

`#pragma linkage` directives may be applied to function names or `typedef` names where the `typedef` is for a function. Also, the directive must appear before the first use of the function or `typedef` name.

For example, to define a pointer to a `#pragma linkage OS` function, a `typedef` can be employed, as in:

```
typedef osfunc_type();
#pragma linkage(osfunc_type, OS)

osfunc_type *function_ptr;
```

which defines `function_ptr` as a pointer to a function that is to be invoked with `OS` linkage.

#pragma map(*identifier*, "*name*")

`#pragma map` specifies that external references to functions or data named *identifier* are to be replaced with the string *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.

Note that if the *–fno-alias-stmts* is enabled, `#pragma map` is not supported.

#pragma weakalias(*identifier*, "*name*")

`#pragma weakalias` specifies that a weak definition 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 *–fno-alias-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 DXD definitions – it is impossible to make two DXD definitions with different names but the same address.

#pragma noinline(name)

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 options(name[,name]...)

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 `#pragma options(RENT)` and `#pragma options(NORENT)` are supported.

#pragma pack(*n*)

`#pragma pack` specifies the maximum structure element alignment for structure type *declarations*. The `_Packed` qualifier can be used to alter the alignment on particular *definitions* of particular data; while `#pragma pack` applies to the general type declaration of a structure.

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, **DCC** 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 (full)` is equivalent to `#pragma pack(8)` if *–mlp64* or `#pragma pack(4)` if if *–milp32* is in effect. `#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(full)`.

If *–fztpf* or *–flinux* was specified on the commandline then the structures produced by **DCC** are compatible with **gcc** 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.

For further compatibility with IBM's compilers, there is a command line option *–fansi-bitfield-packing*, which causes packing of bitfields within structures to be compatible with IBM's `LANGLVL(COMMONC)` or `LANGLVL(ANSI)` options. This behavior can be controlled with `#pragma pack(ansi)` and `#pragma pack(noansi)`, so that individual source or header files can override the commandline settings.

#pragma prolkey(identifier, "*key*")

Using `#pragma prolkey` allows the user to tailor certain function entry points by adding additional macro arguments. `#program 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 STDC FENV_ACCESS *switch*

When the *–fc99* option is enabled, Systems/C will respect the `#pragma STDC FENV_ACCESS` *`switch`* pragma. This pragma is particularly useful with the *–fieee* option is also enabled. If the *–fc99* option is not enabled, Systems/C will silently ignore `#pragma STDC FENV_ACCESS`.

The *switch* value can be `ON`, `OFF`, or `DEFAULT`. The default mode is `OFF`.

When `#pragma STDC FENV_ACCESS` is `ON`, floating point operations that could raise floating point exceptions (i.e. inexact, or division by zero) are not optimized away and also do not participate in constant folding, unless they are part of a static initialization. The ANSI C99 standard describes this operation in further detail.

The effect of `#pragma STDC FENV_ACCESS ON` is that such operations will be deferred until execution time, allowing the programmer to reset any exception or rounding

mode and taking different action at runtime. Furthermore, any floating point constant folding, common subexpression substitution, or other optimizations, will be negated while `#pragma STDC FENV_ACCESS` is `ON`.

When `#pragma STDC FENV_ACCESS` is specified at file scope, the setting remains as specified until a subsequent `#pragma STDC FENV_ACCESS` is encountered at file scope. When it is encountered within an inner scope, the previous value is restored at the end of that scope.

#pragma warning "*text*"

`#pragma warning "text"` causes the compiler to generate a warning message. The warning message will include the specified `text`.

#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 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 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*`"`.

C preprocessor extensions

DCC 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:

```
cpp: file line #: Warning #1116: #warning "this is a warning"
cpp: file line #: Warning #1116: #warning
cpp: file line #: Warning #1116: #warning a string
```

#error

A `#error` preprocessor control causes an error message to be generated. Any text following the `#error` is provided in the generated message.

For example, the following `#error` control lines:

```
#error "This is an error"
#error
#error a string
```

will cause the following diagnostics to be generated:

```
cpp: file line #:Error #1016: #error "This is an error"
cpp: file line #:Error #1016: #error
cpp: file line #:Error #1016: #error 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.

Extensions for AR-mode support: __far, __based(), __alet and __aletof()

DCC provides extensions to the C language that allow programs to readily access data in *access register* mode. Data that is appropriately declared will be accessed with a base/access register pair. The compiler automatically tracks access registers associated with the access, and automatically enters AR-mode for the access.

Syntax:

type `__far *` *identifier*;
type `__based(`*alet-identifier*`) *` *identifier*;
`__alet` *identifier*
`__aletof(__far` *pointer expression*`)`

Description:

type `__far *` *identifier*

`__far` pointers are 8 bytes large, and have no disassociated ALET value. The first four bytes of a `__far` pointer contain the ALET; the next four bytes contain the pointer.

type `__based(`*alet-identifier*`) *`*identifier*

`__based()` pointers may have either an integral constant or an identifier of type `__alet` for their base. If they use an `__alet` identifier, the identifier must be visible at the time of declaration.

`__based()` pointers are 4 bytes long, containing only the pointer portion of an AR-mode reference. When the value they address is referenced, the appropriate access register is initialized from the `__alet` identifier.

__alet *identifier*

__alet provides a new data type which is a place-holder for ALET values.

__alet declared identifiers may be assigned to and directly compared for equality using the == and != comparison operators. In assignment or comparison they are considered to have the type unsigned int.

__aletof(__far *pointer expression*)

__aletof() is a Systems/C built-in operator used to extract the ALET portion of a __far pointer expression. __aletof() produces an unsigned int value which is the ALET portion derived from the __far pointer expression. __aletof() produces an rvalue expression and thus, cannot be used to alter the ALET of a __far pointer. __based() pointers are the recommended approach for situations which require modification of ALETs.

There are several issues to be aware of in properly using __far and __based() pointers:

- __based() pointers may be freely converted to __far pointers, which simply assigns the ALET value from the __based() pointer's *alet-identifier* to the ALET portion of the __far pointer, and assigns the pointer value of the __based() to the pointer portion of the __far pointer.
- __far and __based() pointers may be converted to non-AR mode pointers. Such a conversion simply drops the ALET specification, assigning the pointer portion to the non-AR mode pointer.
- __far pointers may be converted to __based() pointers but only the pointer portion is converted. Assigning to a __based() pointer from a __far pointer does not alter the __alet identifier associated with the __based() pointer. Further data references via the __based() pointer will appropriately set the access register using the ALET defined by the *alet-identifier*.
- Any value of integral type may also be converted to a __far pointer. The resulting __far pointer will be given an ALET value of zero.
- Passing __based() pointers as a parameter only passes the 4-byte pointer portion. Passing __far pointers passes the 8-byte ALET-offset pair. __alet identifiers may be passed to functions as well.
- Functions may return __far or __based() pointers. If a function returns a __based() pointer, only the pointer portion is returned. A function that returns a __far pointer returns the pointer portion in register 15, and the associated ALET portion in access register 15.
- There are no __far or __based() pointer address constants and thus, no NULL definition specific to __far or __based() pointers. However, __far and __based() pointers may be compared to the NULL constant. Only the pointer portion will be compared.

- Pointer arithmetic and comparison on `__far` and `__based()` pointers is valid. Only the pointer component will be used. If a `__far` or `__based()` pointer exceeds the defined data space size, the value does not "wrap around" to the beginning of the data space. Note that comparison of two `__far` or `__based` pointers only compares the pointer components. For `__based` pointers, the ALETs can be compared by referring to the alet expressions. For `__far` pointers, the `__aletof()` operator can be used to compare the ALET portions of the pointers.
- Conversions of `__far` and `__based` pointers to the `long long` type will only use the pointer component.

Remote function pointers

DCC 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.

Also, the conversion from a local function pointer to a remote function pointer can only be accomplished for the direct reference to the address of a function. An expression that has a local function pointer, or any generic pointer type, cannot be converted to a remote function pointer. The semantics of remote function pointer do not provide a location or scope to allocate the space needed for the container holding the PRV and the function address. If the container were allocated in static scope, then subsequent execution of the conversion would overwrite any previous value, potentially invalidating existing pointers to the container. If the container were allocated in automatic scope, then pointers to that container are invalid when the scope ends. If the container is allocated dynamically then there is no opportunity to deallocate it, and thus the program "leaks" memory.

Thus, the only valid conversion of a `__local` function pointer value to a `__remote` function pointer is the direct address of a function within the same load module, in which case the compiler can safely allocate the container within the static scope of the current compilation unit.

For example, this is valid when compiled without the *–ffpremote* option:

```
extern void function(void);
__remote void (*remote_fp)(void);  /* __remote function ptr */

      ...

remote_fp = &function;  /* compiler can allocate the container */
```

But, this would be invalid:

```
extern void function1(void);
extern void function2(void);
__local void (*local_fp)(void);  /* __local function ptr */
__remote void (*remote_fp)(void);  /* __remote function ptr */
      ...
local_fp = &function2;  /* or any other address */
      ...
remote_fp = local_fp; /* invalid because the source */
                      /* is not a direct function address */
```

Special "built-in" implementations for common C library functions.

DCC provides built-in implementations for some of the more common C library

functions. Built-in functions are used when the `<string.h>` system header file is included. Invoking these functions does not generate a call to an externally linked function, instead the compiler will provide an "in-line" definition of the function semantics. Also, give the right arguments, the compiler can frequently exand many of these functions to one or two assembly instructions, greatly improving run-time performance.

These list of "built-in" functions includes:

memcpy()
memset()
memcmp()
memchr()
stpcpy()
strcpy()
strlen()
strcmp()
strcat()
strchr()
strncat()
strncmp()
strncpy()
strrchr()
strpbrk()

`#include <string.h>` to take advantage of the built-in versions of these functions.

Furthermore, when in IBM compatibility mode (*–fc370* is specified), the compiler supports all of the documented IBM built-in functions. Consult the IBM C compiler documentation for a description of these.

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 `ALIGN4` linkage pragma can be applied to either function definitions or declarations to alter this default 8-byte alignment. If a `#pragma linkage 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=any* option is specified, Systems/C will generate z/Architecture code that can be used in any `AMODE`. When *–famode=any* is specified, Systems/C will not use a `LOAD-ADDRESS` 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 `AMODE`.

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

A pointer may be qualified with the `__ptr64` qualifier, which indicates the pointer contains a 64-bit address. The `__ptr64` qualifier follows the pointer designation (`*`), as this qualifier applies to the pointer, not the value being pointed-to.

This is most useful when *–milp32* is specified, as when *–mlp64* is specified, normal pointers contain 64-bit addresses. `__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*/*–milp32* setting.

When *–mlp64* 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 64-bit 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 appear in any Systems/C function, regardless of the *–mlp64* or *–milp32* 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 64-bit 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);
```

```
func31()
{
   int size;
   char *ptr;

   size = 100;
   ptr = malloc(size); /* allocate 100 bytes */

   /* Invoke the z/Architecture "big_func" */
   /* 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 it 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:

```
#pragma prolkey(ENTRY,"DCALL=YES")
#pragma linkage(ENTRY,ALIGN4)
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*, *–fztpf* nor *–fc370* are specified, the resulting program is intended to be linked with the Systems/C z/Architecture library. The Systems/C z/Architecture library completly supports 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 z/Architecture library, see the *Systems/C Library* manual.

Linking with the Systems/C z/Architecture library is only slightly different from the normal link process. All that needs to be done is specification of the alternate library.

Systems/C now provides 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* manual contains 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* manual contains detailed information about how to produce MVS 3.8 programs.

Programming for CMS

Systems/C offers support for basic CMS programs using the OSRUN facilities of CMS to emulate an OS/390 environment.

The *Systems/C C Library* manual contains detailed information about how to produce CMS programs.

IBM Compatibility Mode

The Systems/C compiler, **DCC**, can produce assembly language source that — when assembled with the Systems/ASM **DASM** assembler — is object compatible with IBM's C product.

This facility allows **DCC** to be used as an IBM C compatible cross-hosted compiler, or natively on OS/390 and z/OS. Thus, you can generate IBM C/C++ compatible objects on any of the supported by Systems/C and the Systems/ASM assembler for eventual linking in an IBM C environment.

The Systems/C pre-linker **PLINK** automatically recognizes input IBM compatible objects and enables its IBM compatibility mode. So IBM compatible objects can be pre-linked with **PLINK**.

Requirements

Using **DCC** to compile in IBM C compatibility mode requires the availability of the IBM C system include files. If you are running version 2.4 or later of OS/390, you may find these in `/usr/include` in your HFS file system. Or, they can be found in the appropriate PDSs (e.g. `CEE.SCEEH.H` and `CEE.SCEEH.SYS.H`.)

Using IBM's NFS server facilities, you can make these available to your cross-platform host for use by **DCC**.

Also, to link the eventual objects into an executable program, you will need the IBM C libraries installed on your mainframe. The IBM C documentation describes the procedures used for linking IBM C programs.

Compiling in IBM compatibility mode under JCL

When running **DCC** via batch JCL in IBM compatibility mode, the compiler will need to reference the IBM header files.

These header files are disributed for use by the IBM C compiler, and assume the IBM C compiler's `#include` look-up rules.

To mimic the function of the IBM compiler, you should add the *–fincstripsuf* and *–fincstripdir* options. This will cause Systems/C to remove any suffixes and directories from the names found in `#include` files.

Also - you should add the appropriate *–I* options to specify the IBM include file PDSs, `-I//DSN:CEE.SCEE.H` and `-I//DSN:CEE.SCEEH.SYS.H`.

For example, the following JCL fragment will appropriately execute Systems/C for compatibility with IBM z/OS 1.2 objects:

```
//CC EXEC PGM=DCC,PARM='-@PARMS'
//STEPLIB DD dignus.load.pds
//PARMS DD *
-fc370=z1r2
-fincstripsuf
-fincstripdir
-I//DSN:CEE.SCEE.H
-I//DSN:CEE.SCEEH.SYS.H
//
  .
  .
```

How Systems/C differs from IBM C

Although, when the *–fc370* option is used, **DCC** is very compatible with IBM C object code, there are some differences that need to be noted.

There are some different requirements for `#pragma` directives, described in the section on `#pragma`s. Normally, these are not an issue.

DCC fully supports IBM's `Decimal` type.

As with IBM C, when in IBM C compatibility mode (the *–fc370* option is enabled), enumerations may have the type `char`, `short` or `int` and be either `signed` or `unsigned`, depending on the range of enumerations values specified in the enumeration type. Normally, the type of an enumeration is `signed int`.

Differences from Systems/C

The objects generated in Systems/C mode can be linked into an IBM C program. However, care must be taken as there are some differences. In Systems/C mode, the

parameter alignment and return conventions are different that IBM C. Also, the Systems/C default prologue and epilogue make assumptions regarding the supporting library that are incorrect when linking with IBM C run-time libraries.

An appropriate approach is to use the *–fprol=* and *–fepil=* options to cause Systems/C to generate correct prologue and epilogue macros, and then treat the Systems/C generated assembly source as you would any assembly function when linking with IBM C.

Alternatively, the Systems/C 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 run-time 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. **DCC** will follow the same algorithm when the *–fansi-bitfield-packing* option is enabled on the **DCC** 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, **DCC** follows this algorithm.

Thus, **DCC** 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.

In versions of **DCC** compiler before 1.91, there was a similar option called *–fansi-bitfield*. The older option had the side effect of also disabling the use of types other than `int` (such as `char`) for bitfields. The newer **DCC** separates this functionality out into an independent *–fno-nonint-bitfield* option. So *–fansi-bitfield* is equivalent to specifying both *–fansi-bitfield-packing* and *–fno-nonint-bitfield*.

Assembling with the Systems/ASM assembler

Assembling the output of **DCC** in IBM C compatibility mode requires the use of the Systems/ASM assembler, **DASM**. used for the The compiler generates **DASM**-supported extensions in the assembly source which are not recognized by the IBM HLASM assembler. **DASM** is also capable of creating an IBM Extended Object Module, which will be properly recognized by the Systems/C pre-linker (**PLINK**), IBM binder, or the IBM pre-linker as IBM C objects. Using the **DASM**, IBM C compatible objects can be generated on any of the supported platforms, including OS/390, z/OS and the cross-platform hosts.

For versions 1.95 and later of the **DCC** compiler, the compiler generates `*PROCESS` lines to provide the correct options to the Systems/ASM assembler. This support is only available in Systems/ASM. For versions of the **DCC** compiler prior to 1.95, the proper options must be specified in the Systems/ASM assembler.

There are three important **DASM** options to consider when assembling the compiler-generated assembler source in IBM C compatibility mode, the *–batch* option, the *–idr* option and the *–fdupalias* option.

The *–batch* option is enabled by default and should not be disabled. Systems/C typically generates several virtual "sources" that should be assembled together when invoking Systems/ASM.

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.

The *–fdupalias* option is required because the compiler generates complicated `ALIAS` statements for IBM compatibility mode which are normally flagged as errors and warnings. The *–fdupalias* option instructs **DASM** to allow these constructs and operate on the appropriately.

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 **DCC** 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 **DCC** and later the *–batch*, *–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

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

To 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.

eXtra Performance Linkage

If XPLINK is used, the building and linking process is altered somewhat. **DASM** must be used with the *–goff* option to create a GOFF object deck. XPLINK also requires *–fdupalias* and *–xthread* options on the **DASM** invocation. The *–fdupalias* option instructs **DASM** to allow sophisticated `ALIAS` definitions to set special XPLINK flags. The *–xthread* option tells **DASM** that each section begins at offset 0, which is the norm for XPLINK (rather than beginning at the end of the previous section).

The actual **DASM** command line will look something like:

```
dasm -goff -fdupalias -xthread -idr "5694A01   0105" -o object ... file.asm
```

XPLINK GOFF objects must be pre-linked and linked with IBM's tools on the mainframe. **PLINK** does not currently support all aspects of these objects.

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:

```
dcc -fc370=v2r6 -IIBM-include-directory  file1.c
dasm -fdupalias -idr "5647A01   2600" -o file1.o file1.s

dcc -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 DCC-generated Assembly Source

The assembly source generated by **DCC** can be customized in several ways to assist in development, particularly within an existing run-time environment.

Note that significant alteration of the generated assembly source will prevent the use of the Systems/C library. Furthermore, re-entrant variables (*–frent*) should be used with caution. Any existing run-time library will need to properly allocate the **PRV** and initialize re-entrant variables.

Specifying alternate Entry/Exit macros

By default, **DCC** 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. **DCC** 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 **DCC** 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. *nnn* denotes the current function's `CINDEX` value.

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 if needed. The value of the `ENTRY` argument to the macro will be `YES` if the function should be externally visible, and an `ENTRY` statement should be generated.

The epilogue macro is responsible for deallocating the local stack space, restoring the register contents to their previous values and returning to the caller.

If the function was compiled with *–mlp64* specified, then `ZARCH=YES` will be added to the prologue parameters. In this case, the compiler assumes that the prologue and epilogue saves and restores the full 64-bit values of the registers. If the `ZARCH=YES` option is not specified, the compiler only assumes that the 32-bit values in registers are saved and restored.

For re-entrant programs, **DCC** 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.

In general, it is not possible to mix functions assembled with an alternate prologue/epilogue with the objects from the Systems/C library.

An alternate prologue macro can be specified by using the option *–fprol=XXX* on the **DCC** 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 facility 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(*name*, "*key-string*")

Directs the compiler to add the string *key-string* to the arguments presented to the prologue macro for the function named *name*. 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(*name*, "*key-string*")

Directs the compiler to add the string *key-string* to the arguments presented to the epilogue macro for the function named *name*. 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

DCC 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 **DCC**-generated assembly source into an existing program structure. The alternate base register can be specified using the *–fcode-base=nn* option.

The specified 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.

In normal operation, the compiler will use 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

DCC 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 better interaction in existing runtime environments, it may be preferable 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.

In normal operation, 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.

Specifying a block tag for automatic variables

At the end of each function, **DCC** generates a DSECT that describes the automatic variables allocated in the function. This DSECT, and the fields in it can be referenced in `__asm` code to gain direct access to the variables local to a function. The names of the fields in the sect follow the template:

FunctionName#VariableName#blocktag

where *blocktag* is normally a counter associated with each C block in the function.

Unfortunately, using the simple counter to locate automatic variables is cumbersome and problematic for the user. The blocks have to be counted by-hand and a change which introduces new blocks will alter this count, requiring a change to any `__asm` code in the program.

DCC provides a mechanism for associating a symbolic name with a block. That name will be used in the DSECT field names for the *blocktag*:

`__dsect_tag("`*block-tag-name*`")`

block-tag-name is any string of characters that constitutes a valid assembly language identifier, '`#`' and '`@`' should be avoided as they may conflict with other compiler-generated names. This string becomes the *blocktag* value in the DSECT field name. `__dsect_tag()` appears immediately following the opening brace of a new C code block. It can appear nowhere else.

Specifying a particular `__dsect_tag()` can be valuable when debugging the application. If the debugged supports references to the symbolic names found in the block tag DSECT, then judicious use of the `__dsect_tag()` specification can improve debugging.

Furthermore, if in-line assembly source requires direct references to automatic variables, specifying a particular `__dsect_tag()` allows for these references.

Note that the compiler may choose to place certain C variables wholly in registers. Thus, the DSECT supplied with the block will not be accurate. If the *–g* option is specified, the compiler will not place variables in registers, and the DSECT will be completely accurate.

Using the Systems/C Direct-CALL Interface

The Systems/C library is implemented using the Systems/C entry and exit macros which assume a Systems/C environment is extent 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, 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, 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.

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.

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.

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.

Note that the values in the `@AUTO` DSECT may not be consistent with the state of automatic variables during expression evaluation and other situations. The value of a variable may not be stored back into memory, or in optimized code, the variable may be completely eliminated.

If you wish to ensure the value of the variable is kept in memory at the location specified in the `@AUTO` DSECT, then the variable must be declared using the `volatile` specification.

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. However, using `__dsect_tag()`, you can associate any name with a particular C function block. That name will be used for the value of *blocktag* for the automatic variables declared in the block.

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.

Note that `TRAP=ON` should not be specified. If signal traps are enabled, then the signal handling code will intercept the normal dump mechanisms and a `SIGILL` signal will be raised. For more information about the `TRAP` runtime setting, see the *Systems/C C Library* documentation.

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 in-line */
/*   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 z/Linux and z/TPF

The Systems/C compiler, **DCC**, 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, z/Linux and z/TPF with little change.

When compiling for 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 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 z/Linux, **DCC** 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 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* and *–march=z* options are enabled by default for z/TPF.

The –flinux option

The *–flinux* option causes the compiler to generate source suitable for assembly by the 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* or *–fztpf* options are 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 z/Linux system #include files

The `#include` files provided with z/Linux 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 **DCC** to support the z/Linux header files. The 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 **DCC** command line, run *gcc* with the *–v* flag, where it produces the options it uses for the GNU compiler. For example:

```
gcc -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* and *–D* options used under normal Linux compiles become clear.

The equivalent **DCC** command line under Linux/390 would be:

```
dcc -flinux \
'-D__WCHAR_TYPE__=long int' -Dlinux -D__s390__ \
-Dunix -D__ELF__ -D__linux__ -D__s390__ -D__unix__ \
-D__ELF__ -D__linux -D__unix -D__CHAR_UNSIGNED__  \
-D__signed=signed \
-I/usr/lib/gcc-lib/s390-ibm-linux/2.95.2/../../../../s390-ibm-linux/include \
-I/usr/lib/gcc-lib/s390-ibm-linux/2.95.2/include \
-I/usr/include \
t.c
```

Placing this command in a UNIX shell script, or other scripting language will make process easier.

Furthermore, note that the z/Linux system include files can be copied to any of the Systems/C++ supported platforms. Doing so enables Systems/C++ to cross-compile for z/Linux on other platforms.

Using z/TPF #include files

For z/TPF builds, IBM has modified the include files to be automatically processed with **DCC**. No changes are required.

Assembling z/Linux or z/TPF assembler source

Systems/C generated assembler source may be assembled directly with the 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 z/Linux as command

When **DCC** is executed with the *–flinux* or *–fztpf* options, the generated assembler source is formatted to be assembled with the z/Linux assembler, *as*. For detailed information regarding the *as* assembler; refer to the manual page on the z/Linux system.

Note that if the *–mlp64* or *–fzpf* 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 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 z/Linux

Once the **DCC**-generated assembler source has been assembled, it can be linked as any other object is linked on z/Linux. This is typically accomplished with the *gcc* compiler driver. The *gcc* compiler driver will invoke the 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 z/Linux on-line manual pages with the commands:

```
man ld

man gcc
```

For example, if the **DCC**-generated assembler source `myprog.s` had been assembled into `myprog.o`, then linking this on 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:

```
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`:

```
dcc -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:

```
dcc -flinux -omytest.s mytest.c
gcc -o mytest mytest.s
```

Using DCC 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, support for the `_Decimal` data type, 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 **DCC** 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. **DCC** 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 **DCC** to build programs for z/TPF.

Using DCC for Linux on other hosts

DCC is supported on many different platforms. On each of these, the compiler can be employed to generate z/Linux or z/TPF assembler source by including the *–flinux* or *–fztpf* options.

To do so, the 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, **DCC** can be employed just as it would be on a native z/Linux host.

Furthermore, it is possible to construct a version of the GNU assembler, *as*, which can assemble the **DCC**-generated assembler source on many UNIX platforms. Or, the GNU assembler can be invoked natively on a z/Linux platform by using network facilities such as `rexec`.

For example, it would be possible to generate z/Linux assembler source on an OS/390 host, then use the OS/390 `REXEC` program to invoke the z/Linux 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 z/Linux documentation, or see http://www.gnu.org.

Systems/C C Library

The Systems/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.

License Information File

DCC 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 and Windows host platforms, the file is named `dignus.inf` and is found in the same directory as the **dcc** 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 **DCC** 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 change, assembled and link-edited to produce the the `DIGNUS` load module. However, changing the expiration date, licensee, options or host platform definitions will invalidate the 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 library header files. If the *–nodiginc* option is specified the "`System Include`" statement is ignored.

On UNIX 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 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, the license texts can be concatenated into one `dignus.asm` or `dignus.inf` file.

Compiler Error and Warning Messages

The following list describes the messages produced by **DCC**. Each entry contains the message number, the type of message and a typical message string, followed by a brief description.

Generally, messages produced by the C preprocessor are in the range 1000-1999. Messages produced by the C parser are in the range 2000-2999 and messages produced by the code generator are in the range 4000-4999.

1010 Warning — ISO C forbids evaluated comma operators in #if expressions

A comma operator was encountered within a `#if` expression, but it is a non-standard construct that should be avoided.

1011 Warning — comment in the middle of a preprocessor directive

A comment was found inside of a preprocessor directive, likely as the result of a typographical error.

1012 Error — too many levels of conditional inclusion (max 63)

There were too many nested `#if`, `#ifdef` or `#ifndef` preprocessor directives.

1013 Error — division by 0

Evaluating a preprocessor directive resulted in division by zero, which has no defined value.

1014 Error — duplicate macro argument

A macro has more than one argument with the same name. Only one of the arguments with this name can be accessed.

1015 Error — empty character constant

The character constant '' was encountered. Perhaps '\'' was intended?

1016 Error — #error XXX

A `#error` directive was encountered.

1017 Warning — file 'XXX' not found

While processing a `#include` or `#include_next` directive an attempt was made to open a file that does not exist or is inaccessible.

1018 Warning — unexpected characters in #assert

A `#assert` directive included unexpected characters which were ignored.

1019 Warning — unexpected characters in preprocessing directive

Unexpected characters in a miscellaneous preprocessing directive were ignored.

1020 Warning — unexpected characters in #ifdef

Unexpected characters in a `#ifdef` directive were ignored.

1021 Warning — unexpected characters in #ifndef

Unexpected characters in a `#ifndef` directive were ignored.

1022 Warning — unexpected characters in #include

Unexpected characters in a `#include` directive were ignored.

1023 Error — unexpected characters in constant integral expression

Unexpected characters in a number were ignored.

1024 Warning — unexpected characters in #line

Unexpected characters in a `#line` directive were ignored.

1025 Warning — unexpected characters in #unassert

Unexpected characters in a `#unassert` directive were ignored.

1026 Warning — unexpected characters in #undef

Unexpected characters in a `#undef` directive were ignored.

1027 Warning — identifier not followed by whitespace in #define

The name of the macro was not followed by a left parenthesis or whitespace, the unexpected characters are ignored.

1030 Error — illegal assertion name for #assert

The `#assert` directive was followed by a token that is not a valid preprocessor name.

1031 Error — illegal character 'X'

The specified character was found in the source code but is not in the character set accepted by Systems/C.

1032 Error — illegal macro name for #ifdef

The `#ifdef` directive was followed by a token that is not a valid preprocessor name.

1033 Error — illegal macro name for #ifndef

The `#ifndef` directive was followed by a token that is not a valid preprocessor name.

1034 Error — illegal assertion name for #unassert

The `#unassert` directive was followed by a token that is not a valid preprocessor name.

1035 Error — illegal macro name for #undef

The `#undef` directive was followed by a token that is not a valid preprocessor name.

1036 Error — not enough arguments to macro

A preprocessor macro was invoked with fewer arguments than it was `#define`d with.

1037 Error — invalid escape sequence '\X'

The specified escape sequence was encountered but could not be evaluated.

1038 Error — macro expansion did not produce a valid filename for #include

The `#include` directive used macros to build up the file name, but the result of evaluating the macros was not a valid filename.

1039 Error — not a valid filename for #line

The `#line` directive was encountered with an invalid filename.

1040 Error — invalid '#include'

A `#include` directive was malformed in some way. For example an opening angle-bracket (`<`) may have been found without a closing angle-bracket (`>`).

1041 Error — invalid integer constant 'XXX'

The specified string could not be converted to an integer.

1042 Error — invalid token in constant integral expression

An unknown token was encountered in an integral expression.

1043 Error — not a valid number for #line

The `#line` directive was encountered without a valid line number.

1044 Error — invalid macro argument

An invalid token was used as an argument in a macro call.

1045 Warning — operator '##' produced the invalid token 'XXX'

The `##` operator, which merges two tokens, produced a combined token which was invalid.

1046 Error — invalid argument to _Pragma

`_Pragma()` was used with an invalid argument.

1047 Warning — input line too large

An input line was encountered that was larger than the preprocessor could handle.

1048 Error — macro XXX already defined

Redefining macros is not allowed.

1049 Warning — malformed identifier with UCN: 'XXX'

A malformed identifier with the indicated Universal Character Name was encountered.

1050 Error — malformed UCN in XXX

The indicated Universal Character Name was encountered but not recognized.

1051 Error — too many arguments to macro 'XXX'

The macro named XXX is invoked with more arguments than specified in is definition. The line number provided in the error message indicates the start of the macro invocation.

1052 Warning — more arguments to macro than the ISO limit (127)

A macro was invoked with more than 127 arguments, which is the limit set by the ISO standard.

1053 Error — too many arguments in macro definition (max 253)

A macro was `#define`d with more than 253 arguments, which is the maximum limit supported by Systems/C.

1054 Warning — macro call with XXX arguments (ISO specifies 127 max)

A macro was invoked with more arguments than is allowed in the ISO standard.

1056 Error — Too many include directories

The `#include` search path holds too many directories.

1057 Error — missing comma in macro argument list

A macro argument list contained extraneous arguments not separated by commas.

1058 Error — missing comma before '...'

To define a variadic macro the argument list must look like `(a, b, ...)` rather than `(a, b ...)`.

1059 Error — missing macro name

An attempt to define an anonymous macro was detected.

1060 Warning — multicharacter constant

A suspicious constant such as `'abc'` was used.

1061 Error — a colon was expected

A question mark was encountered and assumed to be part of a `?:` operator, but no matching colon was found.

1062 Error — '...' must end the macro argument list

A variadic macro must have "`...`" at the end of the argument list rather than in the middle.

1063 Error — a right parenthesis was expected

A left parenthesis was encountered with no matching right parenthesis.

1064 Error — could not flush output (disk full ?)

The preprocessor failed to supply the generated output to the C compiler or *–E* listing.

1065 Warning — non-standard line number in #line

The specified line number includes non-numeric characters.

1066 Error — operator '##' may neither begin nor end a macro

The `##` operator cannot span macros.

1067 Error — '__VA_ARGS__' is forbidden in macros with a fixed number of arguments

The `__VA_ARGS__` symbol is only defined for variadic macros.

1068 Error — output write error (disk full ?)

The preprocessor failed to supply the generated output to the C compiler or *–E* listing.

1069 Warning — null preprocessor directive

A line containing just the pound character (`#`) was encountered.

1070 Error — out-of-bound line number for #line

The line number specified was too large or negative.

1071 Error — operator '#' not followed by a macro argument

The `#` operator was encountered in a macro and should have been applied to an argument variable.

1072 Error — quad sharp

A `##` operator was followed by another `##` operator with no intervening tokens to combine.

1073 Warning — reconstruction of <foo> in #include

The name of the file to be `#include`d was constructed through macro expansion and the result is of the form <*foo*>.

1074 Warning — macro 'XXX' redefined unidentically

More than one definition encountered for the specified macro, and they are not identical.

1075 Error — trying to redefine the special macro XXX

The source code specified a redefinition of a special built-in macro which cannot be changed.

1076 Warning — '__STDC__' redefined

The `__STDC__` macro was redefined, it probably should not be.

1077 Error — rogue #elif

The `#elif` directive was encountered in an improper location (i.e., with no corresponding `#if`).

1078 Warning — rogue #elif in code compiled out

The `#elif` directive was encountered in an improper location (i.e., with no corresponding `#if`), but compilation can continue because the `#elif` is in code that is not compiled.

1079 Error — rogue #else

The `#else` directive was encountered in an improper location (i.e., with no corresponding `#if`).

1080 Warning — rogue #else in code compiled out

The `#else` directive was encountered in an improper location (i.e., with no corresponding `#if`), but compilation can continue because the `#else` is in code that is not compiled.

1081 Error — rogue operator 'XXX' in constant integral expression

An operator which could not be evaluated as an integer was encountered within an integral expression.

1082 Error — rogue '#'

A `#` was encountered in a preprocessor directive and ignored.

1083 Warning — rogue '#' in code compiled out

A `#` was encountered in an unused preprocessor directive and ignored.

1084 Warning — rogue '#' dumped

An unexpected `#` was encountered and passed through to Systems/C.

1085 Warning — right shift of a signed negative value in #if

A right shift (`>>`) was applied to a negative value in an expression being evaluated by the preprocessor. The preprocessor cannot determine if logical shift or arithmetic shift was intended (assuming logical shift).

1086 Error — syntax error in #assert

A malformed `#assert` directive was encountered.

1087 Error — syntax error for assertion in #if

A syntax error was encountered in an assertion within a `#if` directive.

1088 Error — syntax error in #unassert

A malformed `#unassert` directive was encountered.

1089 Warning — trigraph ??X encountered

A trigraph of the form `??`*X* was encountered but not translated.

1090 Error — truncated comment

A `/*` comment was ended at the end of the file rather than by `*/`.

1091 Error — truncated constant integral expression

A constant integral expression was ended prematurely by newline or end of file. For example "`(1`" was encountered without the closing parenthesis.

1092 Error — truncated macro definition

A macro definition was ended prematurely by newline or end of file.

1093 Error — truncated token

A token was ended prematurely by newline or end of file.

1094 Warning — truncated UTF-8 character

A UTF-8 character was in the process of being specified when end of file was encountered.

1095 Error — trying to undef special macro XXX

A special built-in macro was undefined with `#undef`.

1096 Warning — undefining '__STDC__'

The built-in macro `__STDC__` is being undefined with `#undef`.

1097 Error — unfinished #assert

`#assert` directive ended by newline or end of file before completion.

1098 Error — unfinished #ifdef

`#ifdef` directive ended by newline or end of file before completion.

1099 Error — unfinished #ifndef

`#ifndef` directive ended by newline or end of file before completion.

1100 Error — unfinished macro call to macro 'XXX'

An invocation of the macro named XXX ended by a newline or end of file before completion.

1101 Error — unfinished string at end of line

A macro string constant was ended by newline or end of file before the closing quote was found.

1102 Error — unfinished #unassert

`#unassert` directive ended by newline or end of file before completion.

1103 Error — unfinished #undef

`#undef` directive ended by newline or end of file before completion.

1104 Error — unknown preprocessor directive '#XXX'

A preprocessor directive was encountered that was not recognized.

1105 Error — unmatched #endif

A `#endif` directive was encountered with no matching `#if` directive.

1106 Warning — unterminated // comment

A comment beginning with `//` was terminated by end of file rather than a newline character.

1107 Error — unterminated #if construction (depth XXX)

A file has ended before all `#if` directives were matched to `#endif` directives.

1108 Error — void assertion in #assert

`#assert` was given an expression with no value.

1109 Error — void condition (after expansion) for a #if/#elif

`#if` or `#elif` was given an expression involving macro expansion with no value.

1110 Error — void condition for a #if/#elif

`#if` or `#elif` was given a simple expression with no value.

1111 Error — void macro argument

A macro was passed an argument with no value.

1112 Error — void macro name

A macro was defined with no name, just arguments.

1113 Error — void assertion in #unassert

`#unassert` was given an expression with no value.

1114 Warning — wide string for #line

A wide string was used for the filename in a `#line` directive.

1115 Warning — wide string for #include

A wide string was used for the filename in a `#include` directive.

1116 Warning — #warning XXX

A `#warning` directive was encountered.

1117 Warning — a C99-style digraph was translated in non-C99 mode

A C99-style digraph (such as `<:`, `:>`, `<%`, `%>`, `%:`, or `%:%:`) was encountered and translated into the corresponding token. However, *–fc99* was not specified and the compiler is not operating in C99 mode.

1118 Error — overflow on division

When evaluating division (for example in a `#if` conditional), the preprocessor detected an overflow condition.

1119 Error — constant too large for destination type

A typed constant was specified that did not fit within the specified type.

1120 Error — invalid wide character constant: XXX

The preprocessor encountered an invalid wide character constant.

1121 Warning — overflow on XXX

When evaluating an integer operation the preprocessor detected an overflow.

1122 Warning — underflow on XXX

When evaluating an integer operation the preprocessor detected an overflow.

1123 Warning — bitwise XXX yields trap representation

Bitwise math was attempted on a negative number within the preprocessor, which forces the preprocessor to provide an answer dependent upon the underlying representation of negative numbers (for example two's complement).

1124 Warning — shift count greater than or equal to type width

The value was shifted so far that the operation yields zero regardless of the original value.

1125 Warning — shift count negative

A negative shift count was used, rather than a positive shift in the other direction.

1126 Warning — right shift of negative value

The preprocessor evaluated a right shift of a negative value, an operation which is dependent upon whether logical or arithmetic shift is used.

1127 Warning — constant is so large that it is unsigned

The preprocessor encountered a constant that was so large that its default type had to be promoted to unsigned.

1130 Warning — last line of file ends without a newline

The ANSI C standard requires that the last line of a file end in a newline, but this file does not. The C preprocessor has inserted one to allow the compilation to continue.

1131 Error — unfinished character literal at end of line

Character literal ended by newline or end of file before closing single-quote found.

2000 Warning — ANSI C forbids an empty source file

The specified source file has no file-scoped declarations (no functions or data), which is forbidden by ANSI C.

2001 Warning — externally visible name '*XXX*' truncated

When the *–fshort-names* option is specified, this warning will be produced for any externally visible declaration that is too long for the generated assembler source.

2002 Error — character 0x*XXX* not in source character set

The compiler has discovered a character in the input stream which is not part of the C source character set. The character's value is given in the hexadecimal value *XXX*.

This frequently occurs when using FTP to move source from a cross-platform host (ASCII) to a mainframe (EBCDIC) with invalid ASCII < − > EBCDIC translation tables.

2003 Warning — #pragma warning <text>

Produced when a `#pragma warning "<text>"` is encountered in the source.

2004 Error — #pragma error <text>

Produced when a `#pragma error "<text>"` is encountered in the source.

2008 Warning — `#pragma map` not supported when –fno-alias-stmts is enabled.

The `#pragma map` facility uses the HLASM ALIAS feature. If *–fno-alias-stmts* is specified, `#pragma map` will be ignored.

2009 Warning — control reaches the end of '*function*' without a return.

The control flow in the specified function can reach the end of the function without an explicit return statement. This warning is disabled by default.

2010 Warning — expected a return expression for this function

The function ended without explicitly returning a value.

2011 Warning — expression has no side effect

The given expression has no side effect, in that it doesn't alter the state of the machine during program execution.

2012 Warning — unsupported linkage in `#pragma linkage` — ignored

An unrecognized linkage type was discovered in a `#pragma linkage` statement.

2013 Warning — typedef redundant 'typedef'

The symbol was already a `typedef`ed value.

2014 Warning — type already specifies `long long`

Another "`long`" was discovered when the type was already `long long`.

2015 Warning — trailing comma in enumerator list

A comma with no following enumerator value was discovered.

2016 Warning — bit-field size exceeds its type

The size specified on the bit-field declaration is larger than the bit-field's type.

2017 Warning — no declaration.

The statement contains only type information, no datum was declared.

2018 Warning — identifier 'XX' not in formal list

An identifier was discovered in the old-style formal declaration list which did not appear in the function's formal argument list.

2019 Error — function 'XXX' already defined in this compilation.

The specified function *XXX* has already been defined in this compilation, this is a duplicate definition.

2020 Warning — promoted argument #*n* doesn't match prototype.

When an old-style function definition is encountered, and a new-style function prototype for the function is visible, the compiler first checks to see if the argument type from the old-style declaration list matches the type specified in the prototype. If they don't match, but the promoted versions do match, this warning is generated.

2021 Warning — prototype with an ellipse can't match empty parameter list.

When comparing two function types, one which contains a prototype argument list and the other with an empty parameter list, that is not associated with a function definition, the compiler examines the prototype argument list. The ANSI standard specifies that in this case, the prototype argument list can not contain an ellipse.

2022 Warning — promoted prototype parameter #*n* can't match empty parameter list.

When comparing two function types, one which contains a prototype argument list and the other with an empty parameter list, that is not associated with a function definition, the compiler examines the prototype argument list. The ANSI standard specifies, in this case, that the prototype arguments must be compatible with the default promoted type.

2023 Warning — function 'XXX' declared '`static`' but never defined

The compiler encountered a declaration of the function specified as *XXX* with the `static` storage class, but no definition of the function was found in this compilation. Calls to the function will be the same as if the `static` specification was not present on the declaration.

2024 Error — missing type for 'XXX' in new-style function header

The parameter *XXX* was discovered in a new-style function header without an associated type.

2025 Warning — pointer to a function used in arithmetic

Pointer arithmetic was used on a pointer to a function. This is an undefined operation. The compiler will use the size 1 as the pointed-to size.

2026 Warning — comparison of different pointer types lacks a cast

Two pointers of different types were compared.

2027 Warning — increment of a pointer of type '`void *`'

Incrementing a pointer variable adds the size of the pointed-to type to the pointer. As the type (`void`) has no size, the compiler emits this warning and uses a size of 1 byte.

2028 Warning — assignment of incompatible pointers

A pointer of an incompatible type was assigned to another without a cast.

2029 Warning — decrement of a pointer of type '`void *`'

Decrementing a pointer variable subtracts the size of the pointed-to type from the pointer. As the type (`void`) has no size, the compiler emits this warning and uses a size of 1 byte.

2030 Warning — address of register variable 'XXX' requested

ANSI C forbids applying the address-of operator (`&`) to an automatic variable with the `register` specification. The compiler indicates the variable's name in *XXX*.

2031 Warning — pointer of type 'void *' used in arithmetic

Pointer arithmetic determines the size of the pointed-to type for the arithmetic operation. As (`void`) has no size, arithmetic on (`void *`) pointers is invalid. In this instance, the compiler produces this warning and uses a size of 1-byte.

2032 Warning — passing argument *N* converts pointer to integral without a cast

The actual argument is a pointer value while the formal argument of the function expects an integral value. The pointer will be converted to an integer and passed. Note that this message does not appear when the pointer value is the `NULL` constant.

2033 Warning — passing argument *N* converts integral to pointer without a cast

The actual argument is an integral value while the formal argument of the function expects a pointer value. The integral value will be converted to the appropriate pointer type and passed to the function. Note that this message does not appear when the integral value is a constant zero.

2034 Warning — passing argument *N* from incompatible pointer type

The formal parameter type specifies a pointer, and while the actual argument is a pointer, it is not a pointer compatible with the type of the formal argument.

2035 Error — incompatible type for argument *N* of 'XXX'

Argument number *N* of the call to the function specified by *XXX* could not be converted to the type specified by the function's prototype.

2036 Warning — incompatible pointer types in conditional expression

The two results of a conditional expression are incompatible pointers.

2037 Warning — initialization converts integral to pointer without a cast

The initialization value was of an integral type, but the type of the datum to be initialized is a pointer type. The integral value will be converted to the appropriate pointer type and the initialization will be allowed. Typically, such an initialization is in error, but the compiler has allowed it with a warning.

2038 Warning — initialization converts pointer to integral without a cast

The initialization value was of a pointer type, but the type of the datum to be initialized is of an integral type. The pointer value will be converted to the appropriate integral type and the initialization will be allowed. Typically, such an initialization is in error, but the compiler has allowed it with a warning.

2039 Error — `sizeof` applied to incomplete type

The `sizeof` operator has been applied to a structure, union or enumeration type which is not yet defined.

2040 Error — `__alignof` applied to incomplete type

The `__alignof` operator has been applied to a structure, union or enumeration type which is not yet defined.

2041 Warning — `sizeof` applied to a function type

The `sizeof` operator was applied directly to a function type, the value returned is undefined.

2042 Warning — `sizeof` applied to a void type

The `sizeof` operator was applied to the `void` type, the value returned is undefined.

2043 Error — `sizeof` applied to a bit-field

The `sizeof` operator was applied to a bit-field member of a structure.

2044 Warning — `__alignof` applied to a function type

The `__alignof` operator was applied directly to a function type, the value returned is undefined.

2045 Warning — `__alignof` applied to a void type

The `__alignof` operator was applied to the void type, the value returned is undefined.

2046 Error — `__alignof` applied to a bit-field

The `__alignof` operator was applied to a bit-field member of a structure.

2047 Error — expected a structure type in `__offsetof`

The first argument to the `__offsetof` operator must be a structure type.

2048 Error — structure tag 'XXX' not defined in `__offsetof`

The given structure tag name in the first argument of the `__offsetof` operator was not defined.

2049 Error — no identifier specified for initialization

A type specifier was followed by an initialization expression, but no identifier was given for the initializer.

2050 Error — type mismatch in initialization

The type of the datum being initialized is not compatible with the value of the initialization expression.

2051 Warning — assignment from incompatible pointer type

An assignment was made between two pointers that don't point to the same target types.

2052 Warning — assignment truncates pointer without a cast

An assignment from a 64-bit address was made to a 31/32-bit address without casting the pointer type.

2053 Warning — passing argument *N* truncates pointer without a cast

The given argument passes a 64-bit address, but the parameter was expecting a 31/32 bit address.

2054 Warning — dereference truncates pointer

A dereference operator (* or array subscript) was applied to a 64-bit pointer in 32 bit mode. The 64-bit pointer is converted to 32 bits for the dereference.

2055 Warning — ISO C90 forbids mixed declarations and code

The C standards prior to C99 did not allow mixing of declarations and statements in a block. This warning is enabled if -fc90 is specified and can be promoted to an error. The warning can be helpful for compiling code intended to also be compiled with pre-C99 compilers.

2060 Warning — hex escape sequence x*NNN* out of range - truncated

The hex escape sequence appearing within a character or string constant is too large for the character value and has been appropriately truncated for the character type.

2097 Warning — comparison is always true

The comparison expression always results in a true value, for example, an tt unsigned int is always great-than or equal to zero.

2098 Warning — comparison is always false

The comparison expression always results in a false value, for example, an `unsigned int` never less-than zero.

2099 Warning — comparison between pointer and integer

A comparison operation was made between an integral value and a pointer. Typically, such comparisons are invalid, but the compiler has allowed it with this warning.

2100 Error — syntax error: XXX

General syntax error. A more detailed reason will also be given.

2101 Error — pointer subtraction of different types

A subtraction operation of incompatible pointer types was encountered.

2102 Error — incorrect operand types for pointer subtraction

One of the two operand types used in a pointer subtraction operation was invalid.

2103 Error — incorrect operand types for pointer addition

One of the two operand types used in a pointer addition operation was invalid.

2104 Error — invalid operands to binary X

One of the operands to addition, subtraction, division, multiplication or modulus was of the wrong type.

2105 Error — incompatible operand types to binary X

The two operands of addition, subtraction, division, multiplication or modulus are of incompatible types.

2106 Error — invalid operands to ==/!=

At least one of the operands to an equality operation was invalid.

2107 Error — invalid operands to </<=/>/>=

At least one of the operands to an inequality operation was invalid.

2108 Error — invalid operands for <</>>

At least one of the operands to a shift operation was invalid.

2109 Error — undefined label 'X' at end of function

The label 'X' is referenced in a goto statement, but not defined within the body of the function.

2110 Error — invalid type for constant conversion to boolean

The constant may not be converted to a boolean type. Although there is no "boolean" type in C, this can occur if there is an attempt to perform boolean logical operations to constant values of the wrong type.

Note that this does not apply to the ANSI C99 `_Bool` type. This error is produced when attempting to convert a type to a logical operation, e.g. for use as the test expression of an `if`-statement.

2111 Error — invalid conversion to pointer

The type of value cannot be converted to a pointer type.

2112 Error — invalid type for constant conversion to `short int`

The constant may not be converted to a `short` integer.

2113 Error — invalid type for constant conversion to `int`

The constant may not be converted to an integer.

2114 Error — invalid type for constant conversion to `unsigned short int`

The constant may not be converted to an `unsigned short` integer.

2115 Error — invalid type for constant conversion to `unsigned int`

The constant may not be converted to an `unsigned` integer.

2116 Error — invalid type for constant conversion to `unsigned long int`

The constant may not be converted to an `unsigned long` integer.

2118 Error — invalid type for constant conversion to `long int`

The constant may not be converted to a `long` integer.

2119 Error — invalid type for constant conversion to `double`

The constant may not be converted to a `double`.

2120 Error — invalid type for constant conversion to `float`

The constant may not be converted to a `float`.

2121 Error — invalid type for constant conversion to `unsigned char`

The constant may not be converted to an `unsigned char`.

2122 Error — invalid type for constant conversion to `signed char`

The constant may not be converted to a `signed char`.

2123 Error — invalid type for constant conversion to `long long`

The constant may not be converted to a `long long`.

2124 Error — invalid type for constant conversion to `unsigned long long`

The constant may not be converted to an `unsigned long long`.

2125 Error — invalid conversion to double

The value may not be converted to a double.

2126 Error — conversion to a non-scalar type requested

The conversion specifies a structure as the destination type.

2127 Error — conversion specifies array type

The conversion specifies an array as the destination type.

2128 Error — invalid type specifier

The type specifier does not follow the ANSI rules for valid type specifies.

2129 Warning — declaration of 'X' masks formal parameter

The variable specified as *X* was declared in automatic scope — but it also is the name of a parameter to the current function. References to the parameter will not be possible within the scope of this declaration.

2130 Error — redeclaration of extern 'X' with different types

The external variable specified as *X* was redeclared with a different type.

2131 Error — redeclaration of 'X'

The variable specified as *X* was already declared in the current scope.

2132 Error — redeclaration of extern 'X' as a static

A variable specified as *X*, previously declared with external scope was declared as `static`.

2133 Error — redeclaration of static 'X' as an extern

A variable specified as *X*, previously declared with `static` linkage was declared as external.

2134 Error — redefinition of 'X'

The `typedef`, `struct` or `enum` specified as *X* was redefined.

2135 Error — use of incomplete tag 'X' in declaration of 'Y'

The structure or enumeration tag '*X*' was not defined by the type '*Y*' was declared.

2136 Warning — implicit declaration of function 'XXX'

There was no function declaration or prototype in scope when a call to function '*XXX*' was encountered. The function will be implicitly declared to return the type *int*.

2137 Error — redeclaration of enumeration tag 'XXX'

The enumeration tag specified as *XXX* was already declared in the current scope.

2138 Error — function definition declared 'typedef'

The "`typedef`" keyword appeared on the function header for a defined function.

2139 Error — field 'XXX' already defined in this structure

The field specified by *XXX* was previously defined in the structure.

2140 Error — field reference to a non-structure

The field reference (`.` or `->`) operation was applied to a datum which isn't a structure or pointer to a structure.

2141 Error — no field 'X' in structure 'Y'

The field '*X*' doesn't appear in structure '*Y*'.

2142 Error — storage size of 'X' isn't known

The compiler cannot determine how much space should be reserved for the symbol *X*. This typically occurs when an incomplete structure or union type is used in the declaration.

2143 Warning — redefinition of typedef 'X'

The named typedef symbol X was already defined as a typedef. Older K&R compilers silently allow such a redefinition if the type was the same. Some other compilers only produce an error if the type is different; while others always make this an warning. This message may be promoted or demoted to effect the desired behavior.

2145 Error — field 'XXX' declared as a function

The named structure field was declared as a function, which is not allowed in structures.

2146 Warning — static function 'XXX' declared in block scope

ANSI C does not allow function declarations in inner blocks with the `static` specifier.

2147 Warning — no function prototype given for 'XXX'

A function call was detected when no prototype for the function was available. This warning is disabled by default.

2148 Warning — struct/union has no members

No members were found in processing a structure declaration. ANSI C requires a member list for structures.

2150 Error — label 'X' already defined

A label statement was encountered for a label that was already defined elsewhere in the function.

2151 Error — case label is not an integral constant

Case values must be integral constants.

2152 Error — duplicate case value

A previous case label is already present for this value.

2153 Error — duplicate '`default`' label for switch

A default label is already present for this switch statement.

2154 Error — switch value must be of integral type

The value specified in the `switch()` statement must be of integral type.

2155 Error — no enclosing `for`/`while`/`do` for `continue`

A continue was encountered outside of any `for`, `while` or `do` scope.

2156 Error — no enclosing `for`/`while`/`do` for `break`

A break was encountered outside of any `for`, `while` or `do` scope.

2157 Error — invalid expression type in return

The type of the expression used in the return statement is not convertible to the type specified in the function declaration.

2158 Error — `__asm` size is not an integral constant

The option size argument to inline assembly source must be an integral constant.

2159 Warning — function returns `void` — return value ignored

The function was specified as returning the `void` type, but the return statement contains an expression value.

2160 Warning — integer constant out of range

The constant was too large to fit in an integer, long or unsigned long.

2161 Warning — integer constant is so large that it is unsigned

The constant was larger than the maximum `signed long` value. Thus, per the ANSI standard, it is considered an `unsigned` value.

If the *–fc99* option is specified, the value will be considered an `unsigned long long` value.

2162 Warning — `__asm` line is too long for \c continuation

A was specified in a line in an `__asm` block, but the existing line is already longer than 72 columns. No continuation character will be placed in column 72, essentially ignoring the \c.

2163 Warning — explicit type is missing, (int) assumed

A type did not provide an explicit type specifier and the specifier of "int" was given. This was the defined behavior in the C89 standard, later standards require a diagnostic.

2164 Warning — multi-character character constant

A character constant containing more than one character was discovered in the input source file. Although the ANSI standard allows this, primarily for multibyte character support, it is frequently a programming error.

2165 Error — character constant too large

A multibyte character constant contained too many characters. The length of multibyte character constants is limited to the size of a `wchar_t`. The size of `wchar_t` can be altered with the *–fwchar=n* option.

2166 Error — numeric constant contains digits beyond the radix

An octal or hexadecimal numeric constant uses an inappropriate digit.

2167 Error — invalid conversion in cast expression

A cast expression specified a target conversion type which isn't possible.

2168 Warning — cast to pointer from integer of different size

A cast expression was applied to an integral type, casting that value to a pointer type. In this situation, the size of the integral value was either larger or smaller than the size of the target pointer type.

2169 Warning — cast to integer from pointer of different size

A cast expression was applied to a pointer type, casting that value to a integral type. In this situation, the size of the pointer value was either larger or smaller than the size of the target integral type.

2172 Warning — unrecognized –q option

The value specified for the *–q* option is unknown.

2178 Error — invalid –fmargins values 'XXX' ignored.

The *–fmargins=m,n* option specifies invalid values for *m* and, if specified, *n*. *m* must be greater than `0` and less then `32761`, and *n* must be greater than *m* and less than `32761`.

2173 Warning — unrecognized –f option

The value specified for the *–f* option is unknown.

2174 Error — too many input files

The compiler can only compile one file at a time.

2175 Warning — unknown option: 'XX' — ignored.

The text specified by *XX* isn't an option recognized by the compiler and has been ignored.

2179 Warning — bad value in –fwchar option 'XX' — ignored.

The text specified by XX was not a correct value for the *–fwchar* option. *–fwchar* can be specified as either 2 or 4.

2180 Error — License validation failed: XXX

The license information isn't valid. A reason is given and the compilation is halted.

2181 Warning — License warning

There are issues with the license information which should be noted; but processing will continue.

2185 Error — can't open input file 'X'

The specified input file *X* cannot be located and/or opened.

2186 Error — can't open output file 'X'

The specified output file *X* cannot be located and/or opened.

2187 Warning — unrecognized –W option

The value specified for the *–W* option is unknown and ignored.

2189 Error — all dimensions except the first must be specified for a multi-dimensional array

For the compiler to properly determine the size of an array, only the first dimension may be omitted.

2190 Error — invalid array initializer

An initializer was specified for an array datum without the appropriate left brace.

2191 Error — incorrect character array initializer

A string constant was specified for a character array initialization after some of the previous indices were initialized.

2192 Error — invalid structure initializer

An initializer was specified for a structure datum without the appropriate left brace.

2193 Error — too many initializers for structure

All of the fields in the structure are initialized, with some initialization values remaining.

2194 Error — invalid initialization to static data

Static data can only be initialized with constants.

2195 Error — can't initialize a function

A function cannot be initialized.

2196 Error — can't initialize a typedef

A typedef cannot be initialized.

2197 Warning — initializer string is too long, truncated

The string constant is larger than the character array's specified size. The string constant will be truncated at the number of bytes specified by the array declaration.

2198 Warning — braces around scalar initializer for 'XXX'

Extra braces appear around an initializer for the variable *XXX*.

2199 Warning — bit-field initializer value too large, truncated

The value specified for a bit-field initialization is larger then the bit-field can accommodate. The value is truncated on the left to fit in the field.

2200 Error — invalid initializer

The type of the initialization expression was not compatible with the datum to initialize.

2201 Error — character array initialized from wide string

A character array cannot be initialized with a wide-string constant.

2202 Warning — initialization from incompatible pointer type

An initialization expression was encountered where the type of the initializing value was not compatible with the type of the target datum.

2203 Warning — file-scoped declaration of 'XXX' globally reserves register #*R*

A file-scoped datum declared with the `__register` keyword reserves the register for the remaining functions in this compilation. The named register will be unavailable for use by the compiler.

2204 Error — `__register` variable 'XXX' declared extern

Because `__register` is used to associated a particular machine register with a datum, the class of the datum must not be `extern`.

2205 Warning — ANSI C restricts enumerator values to range of '`int`'

Enumerator values must be in the range supported by the `int` data type.

2206 Error — overflow in enumeration values

When the compiler computed an enumerator value by adding one to the previous value, the expression overflows the range supported by the `int` data type.

2207 Error — bit-field 'XXX' must be of type `signed int`, `unsigned int` or `int`

The named bit-field is not of a valid bit-field type. This error can only occur if one of the *–fansi-bitfield* or *–fno-nonint-bitfield* options is enabled.

2208 Warning — bit-field 'XXX' type invalid. Type '`unsigned int`' assumed.

This warning can only occur if *–fc370* is enabled. The type for the specified bit-field is integral, but it is not allowed according to the ANSI standard. The compiler has substitute the type `unsigned int` as the bit-field's type.

2209 Warning — bit-field 'XXX' type invalid in ANSI C

The named bit-field specifies an integral type, but this type is not signed int, unsigned int or int. The compiler allows these types of bit-fields as extension to the ANSI standard. This warning can only occur if one of the *–fansi-bitfield* or *–fno-nonint-bitfield* options is enabled.

2210 Error — invalid type specifier

A type specification was expected.

2211 Error — both short & long in type specifier

A type specifier contains both the short and long keyword.

2212 Error — both signed and unsigned in type specifier

A type specifier contains both the signed and unsigned keyword.

2213 Error — enumerator value for 'X' not an integral constant

Values assigned to enumeration constants must themselves be integral constants. *X* provides the name of the enumeration constant.

2214 Error — structure or union tag used in enumeration specifier

The `enum` keyword was applied to a structure or union tag.

2215 Warning — use of incomplete enumeration tag 'XXX'

The enumeration tag *XXX* was used before it was defined.

2216 Error — bit-field width not an integer constant

The size specification of a bit-field must be an integer constant.

2217 Error — bit-field size of 0 for 'X'

Bit-field sizes must be larger than zero. *X* specifies the name of the erroneous field.

2218 Error — invalid type for bit-field

Systems/C bit-fields must be an integral type, i.e. `long`, `int`, `short`, `char` or their `unsigned` variants.

2219 Error — enumeration tag used in struct/union specifier

The struct keyword was applied to an enumeration tag.

2220 Error — redefinition of struct/union 'X'

The structure "*X*" is already defined in this scope.

2221 Error — use of incomplete structure tag 'X'

The structure tag *X* was used before it was completely declared.

2222 Error — __register specification is not an integral constant

The value which specifies a particular register number must be an integral constant.

2223 Error — parameter name missing

A parameter of no names was declared in an old-style function argument declaration list.

2224 Error — incorrect type for `__based` identifier

The type for a `__based` identifier must be `__alet`.

2225 Error — undefined identifier 'X' for `__based`

The specified identifier *X* for a `__based` pointer's alet was undefined.

2226 Error — `__based` constants must be of integral type

If a constant is used for a `__based` pointer, it must be of integral type.

2227 Error — duplicate identifiers in function declaration

This name was already used in an old-style function parameter identifier list.

2228 Error — array size for 'XXX' not an integral constant

The size specified for an array must be an integral constant. *XXX* specifies the name of the array.

2229 Error — redeclaration of 'XXX' in parameter declaration list

The given name *XXX* was already declared in an old-style formal parameter list.

2230 Error — lvalue expected

A modifiable lvalue was expected as the destination of an assignment.

2231 Error — assignment to a void typed lvalue

A `void`-type may not be assigned to.

2232 Error — can't assign to a function

A function may not be assigned to.

2233 Error — invalid pointer assignment

The type of the expression on the source an assignment could not be converted to the pointer type specified by the destination.

2234 Error — assigning to 'XXX' from incompatible type 'XXX'

The type of the source of an assignment could not be converted to the type of the destination.

2235 Warning — assigning to a const datum

The destination of an assignment was specified with a *const* storage class.

Because this is a warning, the compiler will allow the assignment. However, `const` data should not be written to.

2236 Warning — assignment converts pointer to integral without a cast

The destination of an assignment is of integral type, while the source is a pointer type. The compiler will convert the pointer to the integral type.

2237 Warning — assignment converts integral to pointer without a cast

The destination of an assignment is of pointer type, while the source is a integral type. The compiler will convert the pointer to the appropriate pointer type.

2240 Error — undefined identifier 'X'

The named identifier *X* wasn't declared in any visible scope.

2241 Error — too many arguments for call to function 'X'

The call to function *X* doesn't match the function prototype in number of arguments.

2242 Error — too few arguments for call to function X

The call to function *X* doesn't match the function prototype in number of arguments.

2243 Error — invalid use of void expression as a parameter

`void` typed expressions may not be used as actual parameters in function calls.

2244 Error — dangling comma in argument list

A spurious comma was encountered in a function call's actual parameter list.

2245 Error — invalid or missing parameter

A parameter was expected.

2246 Error — array subscript not of integral type

The value specified as the subscript of an array must be of integral type.

2247 Error — subscripted value is neither array nor pointer

The subscript operation must be applied to either arrays or pointers.

2248 Error — call is not to a function or via a function pointer

The call operation must be applied to function or the value of function pointers.

2249 Error — invalid argument type for `->`

The indirect operation must be applied to a pointer to a structure.

2250 Error — expected identifier after `'->'`

A field name was expected after an indirection operation.

2251 Error — postfix `++/--` not allowed in constant expressions

Assignments, which the postfix operators `++` and `--` imply, are not allowed in constant expressions.

2252 Error — lvalue required for postfix `'++/--'`

Assignments, which the prefix operators `++` and `--` imply, are not allowed in constant expressions.

2253 Error — expected a value after a cast expression

Cast operations must be applied to values.

2254 Error — prefix ++/-- not allowed in constant expressions

Assignments, which the prefix operators `++` and `--` imply, are not allowed in constant expressions.

2255 Error — lvalue required for prefix '++/--'

The operand to a prefix `++` or `--` operation must be an lvalue.

2256 Error — operands to '&' must have integral type

Bitwise-AND can only be applied to values of integral type.

2257 Error — operands to '^' must have integral type

Bitwise-exclusive OR can only be applied to values of integral type.

2258 Error — operands to '|' must have integral type

Bitwise-OR can only be applied to values of integral type.

2259 Error — operands to '&&' must be scalar

Logical-AND con only be applied to value of scalar type.

2260 Error — operands to '||' must be scalar

Logical-OR can only be applied to value of scalar type.

2261 Error — test value for conditional expression is not scalar

The test value for a conditional expression must be of scalar type.

2262 Error — type mismatch in conditional expression

The two types for the true and false branches of a conditional expression are not compatible.

2263 Error — incorrect operand to unary '&'

The address-of operand must be applied to an lvalue or a structure or array.

2264 Error — missing operand to unary '*'

The pointer operation expected something to point to.

2265 Error — operand to unary '*' must have pointer type

The operand to the pointer operation must be a pointer.

2266 Error — operand of unary '+' must have arithmetic type

Unary plus can only be applied to operands of arithmetic type.

2267 Error — operand of unary '+' must have arithmetic type

Unary minus can only be applied to operands of arithmetic type.

2268 Error — operand of unary ' ' must have scalar type

Unary complement can only be applied to operands of scalar type.

2269 Error — operand of unary '!' must have scalar type

Unary negation can only be applied to operands of scalar type.

2270 Error — lvalue needed for assignment with binary operator

The binary operator specified expected an lvalue to contain the result of the operation.

2271 Error — missing left parenthesis after __dsect_tag

An left parentheses is expected after the `__dsect_tag` keyword.

2272 Error — missing string in __dsect_tag()

A string, specifying the tag value to use for emitting DSECT information for variables within this scope is missing.

2273 Error — missing right parenthesis in __dsect_tag()

The `__dsect_tag()` specification requires a closing right parenthesis.

2274 Error — attempt to take address of bitfield structure member

The address-of operation returns a byte address. As such, the address of a bitfield is not allowed.

2275 Error — expected expression before multiplicative '*'

The compiler expected an rvalue-expression before the "*" token.

2276 Error — expected expression after multiplicative '*'

The compiler expected an rvalue-expression after the "*" token.

2277 Error — expected expression before division operator '/'

The compiler expected an rvalue-expression before the "/" token. This frequently occurs when a C++ or ANSI C99 //-style comment is encountered and the *–fno-cxx-comments* option is enabled.

2278 Error — expected expression after division operator '/'

The compiler expected an rvalue-expression after the "/" token. This frequently occurs when a C++ or ANSI C99 //-style comment is encountered and the *–fno-cxx-comments* option is enabled.

2279 Error — expected expression before modulus operator '%'

The compiler expected an rvalue expression before the "%" token.

2280 Error — expected expression after modulus operator '%'

The compiler expected an rvalue expression after the "%" token.

2281 Error — request for member 'XXX' in something that is not a structure or union

The structure member selection operator (`->` or `.`) specified a source which is not a structure or union.

2282 Warning — assignment discards '`const`' from pointer target type

The source pointer has the `const` qualifier on its pointed-to type, while the target pointer does not. Thus, indirect references through the destination pointer have the potential to alter `const`-qualified data.

2283 Warning — assignment discards '`volatile`' from pointer target type.

The source pointer has the `volatile` qualifier on its pointer-to type, while the destination pointer does not. Thus, indirect references through the destination pointer will not honor the semantics of `volatile`-qualified data.

2284 Warning — passing of argument *N* discards '`const`' from pointer type

Argument #*N* of the argument list is a pointer which points to `const`-qualified data, while the function prototype for function call specifies a pointer which is not `const`-qualified. Thus, indirection references within the function have the potential to alter `const`-qualified data.

2285 Warning — passing of argument *N* discards '`volatile`' from pointer type

Argument #*N* of the argument list is a pointer which points to `volatile`-qualified data, while the function prototype for the function call specifies a pointer which is not `volatile`-qualified. Thus, indirect references within the function will not honor the semantics of `volatile`-qualified data.

2286 Warning — division by zero

The compiler has detected a division where the divisor is a constant 0.

This warning is not generated for floating point division because that can be a way to generate NaN for IEEE and DFP floating point.

2287 Warning — initialization discards 'const' from pointer target type

The source pointer has the `const` qualifier on its pointed-to type, while the target pointer does not. Thus, indirect references through the destination pointer have the potential to alter `const`-qualified data.

2288 Warning — initialization discards 'volatile' from pointer target type.

The source pointer has the `volatile` qualifier on its pointer-to type, while the destination pointer does not. Thus, indirect references through the destination pointer will not honor the semantics of `volatile`-qualified data.

2290 Error — size specifier in __asmval must be an integral constant

The size field of an `__asmval` constant must be a integral constant.

2291 Error — size specifier in __asmval must be between 1 and 4, or 8

The size field of an `__asmval` constant must be a constant, integral expression with the value 1, 2, 3, 4 or 8.

2295 Error — redeclaration of formal parameter 'XXX'

The named parameter has already been declared in the function's parameter declarations.

2296 Warning — unary negation applied to an unsigned type

The ANSI C standard indicates that the operands of unary negation undergo integral promotions, and the result of the negation is that type. This may convert an unsigned operand type into a signed type.

2300 Error — size specification in _Decimal specifier must be of integral type

The size field of a `_Decimal` value must be an integral constant.

2301 Error — size specification in _Decimal must be constant

The size field of a `_Decimal` value must be a compile-time constant.

2302 Error — size value in _Decimal must be in the range 1 to 31

`_Decimal` values may have between 1 and 31 digits.

2303 Error — precision specification in _Decimal specifier must be of integral type

The precision field of a `_Decimal` value must be an integer constant.

2304 Error — precision specification in _Decimal specifier must be constant

The precision field of a `_Decimal` value must be a compile-time constant.

2305 Error — precision value in _Decimal must be in the range 0 to 31

_Decimal values may have a precision between 0 and 31 digits.

2306 Error — precision value must be less than or equal to size in _Decimal

The precision field of a _Decimal specifier must be less then or equal to the size field.

2307 Warning — digits may have been lost in the whole-number part

In a conversion from _Decimal to _Decimal, the number of non-fractional (whole number) digits in the target is smaller than the source. This could result in a loss of values at runtime.

2310 Error — digitsof() must be applied to a Decimal type

__digitsof() cannot be applied to the type specified.

2311 Error — precisionof() must be applied to a Decimal type

__precisionof() cannot be applied to the type specified.

2315 Warning — non-zero digits lost in Decimal constant

During constant evaluation, either evaluating constant _Decimal arithmetic, or in converting a constant to a particular size and precision, a left-shift operation shifted out a non-zero digit.

2318 Warning — `#pragma options` must be specified before the first C statement

The `#pragma options` statement must be located in the C source before any other C statements or declarations.

2316 Warning — `_Decimal` multiplication truncates digits

In a multiplication operation involving `_Decimal` data types, an intermediate value is potentially too large to calculate (more than 31 decimal digits.) In this case, the generated decimal multiplication code will truncate lower-precision digits. The result of such a multiplication may or may not be accurate, depending on the values of the `_Decimal` types at run time.

2319 Warning — unrecognized option "XXX" in `#pragma options`

A `#pragma options` statement contained an option statement which the compiler does not support.

2320 Error — only one `#pragma csect` KIND allowed per program

Only one `#pragma csect CODE`, `#pragma csect DATA`, or `#pragma csect TEST` is allowed per program source file.

2321 Warning — `#pragma prolkey` for 'XXX' replaced

A `#pragma prolkey` for the specified function was discovered after one already had been processed. The new specification replaces the previous one.

2322 Warning — extraneous text after `#pragma` ignored

Extra text followed a `#pragma` statement. This text, up to the end of the source line, is ignored.

2324 Warning — `#pragma map` for symbol 'XXX' already specified, this one ignored

2325 Warning — unrecognized `#pragma` XXX ignored

The compiler did not recognize the specified `#pragma`. This warning is disabled by default and can be enabled via the –Wunknown-pragmas option.

A `#pragma map` statement for the symbol with a different map target was previously specified in the program. This `#pragma map` is ignored.

2324 Warning — redundant `#pragma map` for symbol 'XXX' ignored

A `#pragma map` statement mapped the same symbol name to the same text target. The redundant version is ignored.

2330 Error — operands to '<<'/'>>' must have integral type

One of the operands of a bitwise shift expression was not of integral type. ANSI C requires these operands to be of integral type.

2331 Warning — 'XXX' initialized and declared '`extern`'

The specified identifier is declared as "`extern`" and also has an initialization expression. `extern` variables may be initialized, but it is undefined if the initialization will actually occur at run time. This message occurs for variables declared at file scope.

2332 Error — 'XXX' is both '`extern`' and initialized

The specified identifier is declared as "`extern`" and also has an initialization expression, and is declared within an inner block in a function. ANSI C forbids such declarations.

2333 Error — 'XXX' already initialized

The specified identifier has previously be declared in this compilation, and that declaration already has an initialization expression.

2334 Warning — left shift count >= width of type

A left-shift operation (<<) was applied where the shift amount was greater than or equal to the number of bits in the type of the value to be shifted. The result will be 0.

2335 Warning — right shift count >= width of type.

A right-shift (>>) operation was applied where the shift amount was greater than or equal to the number of bits in the type of the value to be shifted. The result will be 0.

2336 Warning — left shift count negative

A negative shift amount was discovered in the left shift operation (<<). The result will be 0.

2337 Warning — right shift count negative

A negative shift amount was discovered in the right shift operation (>>). The result will be 0.

2338 Error — flexible array member not at end of struct

ANSI C requires that a flexible array member be the last member of a structure definition.

2339 Error — array size missing in 'XXX'

The automatic variable *XXX* was declared as an array, but no array size was specified.

2340 Error — array size missing in field 'XXX'

An array structure field was specified in a structure tag declaration without the necessary size specification. Note that Systems/C allows as an extension structure member arrays of size 0.

2341 Warning — ANSI C forbids zero-sized array field 'XXX'

An array structure field was specified with a size of 0. Although ANSI prohibits this, Systems/C allows it as an extension.

2342 Error — use of incomplete structure in field 'XXX'

A structure field member, which itself is a structure, used a structure tag which is not defined.

2343 Error — use of incomplete union in field 'XXX'

A structure field member, which itself is a union, used a union tag which is not defined.

2344 Warning — initialization of flexible array member

According to the ANSI C standard, flexible array member fields may not be initialized. The compiler supports this as an extension, but generates the warning to provide the required diagnostic.

2345 Error — declaration of 'XXX' as array of voids

The symbol *XXX* was declared to be an array of the (`void`) data type.

2346 Error — declaration of field 'XXX' as array of voids

The field within a structure declaration is declared as an array of the (`void`) data type.

2347 Error — structure tag 'XXX' used in union specifier

The specified structure tag name was used in combination with the `union` keyword as part of a type specification.

2348 Error — union tag 'XXX' used in structure specifier

The specified union tag name was used in combination with the `struct` keyword as part of a type specification.

2350 Error — controlling expression of an if-statement must have scalar type

The value specified in the test portion of the `if` statement must be of scalar type.

2351 Error — controlling expression of a while-statement must have scalar type

The value specified in the test portion of the `while` statement must be of scalar type.

2352 Error — controlling expression of a do-statement must have scalar type

The value specified in the test portion of the `do` statement must be of scalar type.

2353 Error — controlling expression of a for-statement must have scalar type

If a value is specified in the test portion of a `for` statement, it must be of scalar type.

2354 Error — nested initialization of flexible length array

Flexible length arrays at the end of a struct cannot be nested.

2356 Warning — condition is always false

A controlling conditional expression of an `if`, `while`, `do` or `for` statement evaluates to a constant which is always false.

2357 Warning — condition is always true

A controlling conditional expression of an `if`, `while`, `do` or `for` statement evaluates to a constant which is always true.

2358 Warning —- enumeration values not handled in switch...

When a controlling expression of a switch statement is an enumerated type, the compiler verifies that all the values of the enumeration appear as case labels in the switch. If any are missing this warning is generated. The –Wswitch and –Wswitch-enum options control generation of this message.

2359 Warning — case value not in enumerated type 'XXX'

When a controlling expression of a switch statement is an enumerated type, the compiler checks that the values used in the case labels are one of the values in the enumerated type. This message includes the name of the enumerated type for reference.

2360 Warning — dereferencing '`void *`' pointer

A dereference (`*`) was made through a pointer which points to a (`void`) data type.

2361 Warning — index operator applied to '`void *`' pointer

The array index operator (`[]`) was applied to a pointer which points to a (`void`) data type.

2362 Warning — case label value is less/greater than minimum/maximum value for type

The constant specified in the case for a switch is outside of the range of values of the controlling expression for the switch.

The –Wswitch-outside-range option controls generation of this message.

2363 Warning — case label not in enumerated type 'XXX'

If the –Wswitch option is enable, and the controlling expression for a switch is an enumeration, the compiler checks that a case label value appears in the enumeration's list. If it doesn't, this message will be generated.

2365 Error — array 'XXX' is too large to fit in the address spa ce

The declaration of an array, or array field member, produces a data item that has a size larger than the entire address space of the target machine.

2366 Warning — ANSI C forbids zero-sized array

The ANSI standard explicitly forbids an array that has a zero-constant size declaration. Several compilers accept this as an extension, so this warning can be suppressed.

2367 Warning — subscript out of range

The value in an array indexing operation was constant, and was larger than the size of the array.

2368 Error — variable length array may not be initialized

A declaration of a variable length array (one in which the size is determined at runtime) may not contain an initilizer expression.

2369 Error — array size expression for 'XXX' not an integral type

The expression denoting the size of an array must be of integral type.

2370 Error — size of array 'XXX' is negative

Array sizes are not allowed to be less than zero.

2371 Warning — return type of 'main' is not 'int'

The ANSI C standard requires that the return type of the `main()` function be `int`.

2375 Warning — return converts integral to pointer without a cast

The expression specified in a `return` statement is of integral type, while the function's return type is a pointer.

2376 Warning — return converts pointer to integral without a cast

The expression specified in a `return` statement is of pointer type, while the function's return type is integral.

2377 Warning — return discards 'const' from pointer target type

The expression specified in a `return` statement is not a `const`-qualified pointer, while the function's return type is `const`-qualified.

2378 Warning — return discards 'volatile' from pointer target type

The expression specified in a return statement is not a `volatile`-qualified pointer, while the function's return type is `volatile`-qualified.

2379 Warning — incompatible pointer type in return

The type of the pointer expression on a `return` statement was incompatible with the declared return (pointer) type of the function.

2380 Error — increment of a pointer to an unknown structure

Either the prefix or postfix version of the increment operator (`++`) was applied to a pointer to an undefined structure.

2381 Error — decrement of a pointer to an unknown structure

Either the prefix or postfix version of the decrement operator (`--`) was applied to a pointer to an undefined structure.

2382 Error — arithmetic on pointer to an incomplete type

A pointer arithmetic operation was attempted where the target type of the pointer was not defined.

2383 Warning — unnamed struct/union that defines no data

An unnamed structure or union tag was defined that had no instances of data.

2384 Warning — floating constant out of range

The given floating point constant is too large to represent in the target floating point format.

2385 Warning — assignment converts a floating point type to one with less precision

The assignment statement converts a floating point type of one precision to one of a smaller precision. For example, converting a `(double)` typed value to a `(float)`, or a `(long double)` to a `(double)`.

2386 Warning — passing argument N converts a floating point type to one with less precision

The parameter expression in a function call statement describes a floating point type that is larger than the data being initialized. The value will be converted to the smaller value, possibly loosing precision.

2387 Warning — return converts a floating point type to one with less precision

The expression in a `return` statement describes a floating point type that is larger than the data being initialized. The value will be converted to the smaller value, possibly loosing precision.

2388 Warning — initialization converts a floating point type to one with less precision

The initialization expression describes a floating point type that is larger than the data being initialized. The value will be converted to the smaller value, possibly loosing precision.

2389 Warning — floating point operation result is out of range

A floating point constant operation result is out of range for the floating point type. The compiler will not fold the operation, and instead will generate code to calculate it at runtime.

2390 Warning — assignment converts `__far` pointer to local pointer without a cast

The assignment statement converts a `__far` pointer to a local pointer, which only includes the pointer portion of the `__far` pointer. The ALET component of the `__far` pointer is discarded.

2391 Warning — passing argument *N* converts `__far` pointer to local pointer without a cast

Argument #*N* is a `__far` pointer, while the function prototype specifies a local pointer. Only the pointer portion of the `__far` pointer will be passed to the function, the ALET portion of the `__far` pointer is discarded.

2392 Warning — return converts `__far` pointer to local pointer without a cast

The type of the expression in a return statement is a `__far` pointer, while the function returns a local pointer. Only the pointer portion of the `__far` pointer will be returned, the ALET portion of the `__far` pointer is discarded.

2393 Warning — initialization converts `__far` pointer to local pointer without a cast

The type of the expression in an initialization is a `__far` pointer, while the datum being initialized is a local pointer. Only the pointer portion of the `__far` pointer will be stored in the datum, the ALET portion of the `__far` pointer is discarded.

2395 Error — argument to `__aletof()` is not a `__far` pointer

The expression argument in an `__aletof` invocation was not a `__far` pointer.

2399 Warning — non-constant member-designator in offsetof

The ANSI standard requires that the offsetof expression be the same as an address-constant, which implies that the member-designator portion must be a constant expression. Many compilers accept non-constant member designators in the offsetof() expression, as does **DCC**. This warning is disabled by default, and can be re-enabled with the *–fenable-warning=2399* option. For compatibility with the IBM C compiler, this warning is re-enabled with when *–fc370* is specified.

2400 Warning — use of bit-field member in `offsetof()` is undefined

The ANSI standard indicates that using a bitfield structure member in an `offsetof()` operation results in undefined behavior. Many compilers will not compile this code and there is no guarantee regarding the result of the operation.

2401 Error — initializer element is not computable at load time

The value used to initialize a datum is not constant, and thus could not be used to initialize static data.

2402 Error — array index in initialization designator exceeds bounds

An array index designator was found in the initializer for an array that exceeds the bounds specified for the array.

2403 Error — array index value not constant in initializer

An array index designator was found that uses a non-constant expression, which is not permitted by the standard.

2404 Warning — extra elements in initializer

There are more elements in the initializer than there are in the object being initialized. Extra elements will be ignored.

2405 Warning — ANSI C forbids an empty initializer list

An empty initializer list, consisting only of an open and close brace, was discovered. The ANSI standard syntax requires an initializer expression in this situation, however, the compiler proceeds as if no initializer was specified.

2406 Warning — anonymous structure/union members are a C11 language extension

Anonymous structure or unions as members of a structure or union are a language extension supported by Systems/C and defined in the ANSI C11 standard. This message can be eliminated by adding the *–fanonstruct* option, or by specifying the C11 or later standard using the *–fc11* (or later) option.

2412 Error — invalid enumeration size

The value specified on either the *–fenum* option or `#pragma enum` pragma is invalid or not supported.

2413 Error — the enum cannot be packed to the requested size

An enumeration constant value produces a range for an enumeration that doesn't fit into the size specified in either the *–fenum* or `#pragma enum` settings.

2415 Warning — unrecognized `#pragma STDC`

A `#pragma STDC` pragma was discovered where the token following the `STDC` symbol was not recognized. Currently, the only recognized symbol following `STDC` is `FENV_ACCESS`.

2416 Warning — invalid switch to `#pragma STDC FENV_ACCESS` ignored

A token following a `#pragma STDC FENV_ACCESS` pragma was not `ON`, `OFF`, or `DEFAULT`. The `#pragma STDC FENV_ACCESS` is ignored.

2429 Error — invalid size for `__register` variable 'x'

`__register` variables must be declared with types that completely fit in a register. In *–milp32* (32-bit compilations), these are the 4-byte sized types. When *–mlp64*, both 8-byte and 4-byte sizes are supported.

2430 Error — address of `__register` variable

Addresses of `__register` variables are not allowed, as hardware registers have no address.

2431 Error — type of `__register` variable 'x' is not integral or pointer

The `__register` type keyword can only be applied in combination with integral or pointer types.

2441 Error — compound expression only allowed within a function

A compound expression, which is a compound statement enclosed in parentheses that returns a value, can not appear at file scope. Because the compound expression contains C statements, it can only be used within the body of a function.

2450 Warning — ANSI C forbids conditional expression with only one void side

The ANSI C standard requires that if either the second or third operand of a conditional expression is `void`, then both operands must be `void`. **DCC** allows only one operand to be the `void` type, with this warning. The other, non-`void` expression will be converted to `void`.

2451 Warning — ANSI C requires second operand in conditional expression, assuming test value

The ANSI C standard requires an expression for the second operand of a conditional expression, the "true" value expression. **DCC** allows this expression to be absent, and will assume the value to use for the "true" expression is the same as the value of the test expression.

2461 Warning — declaration of long double 'XXX' treated as double

This warning is generated when the *–fc370* option is enabled, as DCC treats `long double` the same as `double`, differently from the approach used in the IBM compiler. The declared datum will be have the same size and properties as a `double` value.

2470 Warning — use of __FUNCTION__ outside of function scope

The special pre-defined identifier, `__FUNCTION__` was encountered at file scope, outside of the scope of any function. The compiler uses the empty string (`""`), for the value of `__FUNCTION__` and proceeds.

2473 Error — 'XXX' is unavailable

The symbol specified has been marked with the `unavailable __attribute__`. If possible, the line where the symbol was declared is presented.

2474 Warning — 'XXX' is deprecated

The symbol specified has been marked with the `deprecated __attribute__`. If possible, the line where the symbol was declared is presented.

2475 Warning — 'XXX' attribute directive ignored

The given name was present in an `__attribute__` directive but is unrecognized. The directive is ignored.

2476 Error — unable to emulate mode 'XX'

An `__attribute__((__mode__(...)))` declaration modifier was found which **DCC** does not support.

2477 Error — invalid pointer mode 'XX'

An `__attribute__((__mode__(...)))` declaration modifier was encountered which is incompatible with pointer types.

2480 Warning — unused label 'XX'

The specified label was defined in the scope, but was not directly used in a goto statement nor addressed.

Can be disabled with the *–Wno-label-unused* option.

2481 Warning — unused variable 'XX'

The specified variable was not used in the scope.

This warning is disabled by default, and is enabled as part of the *–Wall* option, or specifically by the *–Wunused-variable* option.

2482 Warning — unused parameter 'XX'

The specified parameter was not used in the function.

The warning is disabled by default, and can be enabled with the *–Wunused-parameter* option.

2483 Warning — unused function 'XX'

The specified static function is defined in the compilation, but not invoked.

The warning is disabled by default, and can be enabled with the *–Wunused-function* and *–Wall* options.

2500 Warning — #pragma linkage(...,fetchable) must appear only once

Only one function amy be modified with the `fetchable` linkage attribute per compilation unit.

2504 Error — z/Architecture is required when –fllgrande is specified

The *–fllgrande* option indicates that `long long` (64-bit) data should be kept in a single 64-bit register instead of two 32-bit register pairs. Thus, the z/Architecture hardware must be used.

2508 Warning — 'XXX' declared in parameter list; its scope may not be what you expect

When a structure name is seen for the first time in a parameter list, it is treated as a predeclaration inside of the function's scope. Then if a structure declaration is seen at file scope, the two declarations are still treated separately. Providing a predeclaration of the structure at file scope before the function declaration will generate the desired behavior:

```
struct foo;              /* file scope predeclaration */
void func(struct foo x);
```

2509 Warning — #pragma for 'xxx' ignored

A `#pragma` provided an attribute for the specified symbol. However, at the end of compilation this symbol was still undefined or defined in a way that was incompatible with the attribute.

2510 Error — The decimal-floating-point-facility (*–march=z6* or -mdecimal-floating-point-facility) is required when *–fdfp* is specified

The compiler uses the instructions available in the decimal-floating-point-facility to generate the code required for the `_Decimal32, _Decimal64 and _Decimal128` data types.

A *–march=z6* or greater option enables the use of those instructions.

2514 Warning — static and non-static on same symbol

A `static` definition or declaration was found for a symbol that was previously declared without `static`. The ambiguous source code may result in a static symbol with one compiler and a global one with another.

2515 Error — cannot initialize non-reentrant data with the address of reentrant data

A file-scope or static initialization of a non-reentrant variable referenced the address of a reentrant variable. There is no way for a non-reentrant initializer to take know the address of the PRV, so this form of initialization is impossible.

2525 Warning — signed bit field of length 1

When a bit field is declared with a signed type, and has a length one, then if the bit is set, the value retrieved will be `-1` instead of `1` due to sign extension rules.

2601 Error — can't mix decimal floating point operands and other float types

The decimal floating point types (`_Decimal32`, `_Decimal64` and `_Decimal128`) cannot be used in an expression along with the floating point types (`float`, `double` and `long double`) as the compiler doesn't know if the operation is to be performed using decimal floating arithmetic or floating point arithmetic.

Adding an explicit cast to either a decimal floating point or a floating point type will address the issue.

2602 Warning — decimal floating point constant out of range

The constant value was too large for the target decimal floating point type.

2603 Warning — assignment converts a floating point type to one with less precision

The target of the assignment expression was a floating point type with less precision than the type of the source. The value will be converted to the target size with possible loss of precision.

2604 Warning — passing argument N converts a floating point type to one with less precision

The parameter in a function call is a floating point value with a precision larger than the called function's prototype specified. The value will be converted to the smaller precision which may result in a loss of precision.

2605 Warning — return converts a floating point type to one with less precision

The function's return type is a floating point type with a smaller precision than the expression found in the `return` statement. The value will be converted to one with smaller precision, possibly resulting in a loss of precision.

2606 Warning — initialization converts a floating point type to one with less precision

A variable's initialization expression is of a floating point type that has a larger precision than the variable being initialized. The expression's value will be converted to the smaller precision, which may result in a loss of precision.

2607 Warning — floating point operation result is out of range

The result of a decimal floating point operation is out of range for the given decimal floating point type.

2610 Warning — ANSI C forbids conversion between function pointers and object pointers

According to the ANSI C standard, function pointer types may only be converted to other function pointer types. They cannot be converted to object types (such as "`void *`"). However, this is only a warning as **DCC** does allow the conversion.

2615 Error — invalid argument in __builtin stdarg evaluation

The value passed to a __builtin stdarg evalation was either not a proper lvalue or was the wrong type.

2616 Warning — use of a type that undergoes default argument promotions in 'va_start/va_arg' is undefined

In variadic functions, arguments under default promotions. Thus, in `va_arg` and `va_start`, using types that would be promoted is undefined behavior, as the actual parameter may not match the type.

These include the type `float` (which would be promoted to `double`) and any integral type with a conversion rank lower than `int`.

2617 Error — 'va_start' used in function with non-variable arguments

`va_start` can only be used in variadic functions, not functions with fixed argument lists.

2620 Warning — function declared 'noreturn' has a 'return' statement

A return statement was encountered in a function declared with `__attribute__((noreturn))`.

2621 Error — type qualifiers or the 'static' keyword are invalid unless they are in the outermost array index of a parameter

For C99 array declarations, a type qualifier or the keyword 'static' can only appear in a declaration of a function parameter with an array type, and then only in the outermost array type derivation.

2625 Warning — assignment expression used as condition

An assignment statement was encountered in an expression being evaluated for 'true' or 'false' (non-zero or zero.) For example, as the control value of an if-statement. It is possible that '`==`' was intended instead of '`=`'.

If the assignment is desired, the warning can be defeated by enclosing the assignment expression in parenthesis.

2630 Warning — bit field declaration

A bit field member of a structure was declared. This warning is disabled by default.

2631 Warning — function returns (long long) without a __grande or __regpair modifier, defaults to xxxx

A function was declared with a `long long` return type without a `__grande` or `__regpair` attribute. This warning is disabled by default.

In some environments the mechanism for returning values in grande registers is different than returning values in register pairs, so the distinction is important. Enabling this warning can help locate function declarations that need to be adjusted.

2640 Error — invalid constant in __builtin_fp_classify()

The constant presented to the __builtin_fp_classify() expression was not a floating-point value.

2641 Error — expression in __builtin_fp_classify is not a floating point value

The expression argument for __builtin_fp_classify() is not a floating point value.

2650 Error — type-name in _Atomic specifier must not contain array, function, atomic or qualified type

The C11 standard requires the type-name in an _Atomic (...) type specifier not specify an array, function, atomic-qualified or other qualified type.

2651 Error — _Atomic qualifier cannot be applied to an array or function type

The C11 standard indicates that the _Atomic qualifier may not be applied to either an array or function type.

2662 Error — An identifier may not begin with a universal character representing a digit

The C11 standard requires that universal character identifiers begin with a non-digit (character) value.

2663 Error — XXX is not a valid universal character for an identifier

The C11 standard requires that a universal character names shall not specify a character whose short identifier is less than 00A0 other than 0024 (`$`), 0040 (`@`) or 0060 (
tt '), nor one in the range D800 through DFFF inclusive.

The disallowed characters are in the basic character set and the code positions reserved for ISO/IEC 10646 control characters, the DELETE character and the S-zone (reserved for use by UTF-16).

2667 Error — invalid type for argument to __builtin_isdigit

The argument to `__builtin_isdigit` must be an arithmetic type.

2668 Error — invalid type for conversion to _Decimal

Only scalars can be converted to _Decimal values.

2670 Error — invalid call to __atomic builtin

A call to an `__atomic` builtin was malformed, probably because a valid type for the type-generic parameters could not be determined.

2671 Error — 'void' must be the first and only parameter if specified

If a parameter declaration of type (void) is used, it must be the only parameter type and cannot be used in combination with other types.

2672 Error — 'XXX' cannot be declared as type (void)

The variable XXX has been invalidly declared with a `(void)` type.

2673 Error — field 'XXX' cannot be declared as type (void)

A member of a structure or union has been invalidly declared with a `(void)` type.

The variable XXX has been invalidly declared with a (void) type.

2675 Error — value in _Alignas must be a type or an integer constant expression

The argument to the alignment specifier (_Alignas()) must be either a type, or a constant integer expression.

2676 Error — invalid value in alignment specifier

The value provided to the alignment specifier (_Alignas()) was either negative, not a power of 2, or otherwise unsupported.

2677 Error — alignment specifier not allowed in typedef

An alignment specifier (_Alignas()) may not be specified in a typedef.

2678 Error — alignment specifier not allowed for bitfields

An alignment specifier (_Alignas()) may not be specified in bitfield structure members.

2679 Error — alignment specifier not allowed in parameter types

An alignment specifier (_Alignas()) may not be specified in types for function parameters.

2680 Error — attribute sequence not allowed in this context

The ANSI attribute sequence [[...]] may not be specified in a structure or union type reference. It is only allowed on a forward tag declaration, or the structure/union definition.

2681 Error — size value in _BitInt must be in the range 1 to XXX

The number of bits for a _BitInt type specifier must be in the range supported for the target. This is typically 64 for most targets.

2700 Error — static assert failed

The expression in a static assertion evaluated to a zero, causing a compile-time message.

2701 Error — static_assert expression is not an integral constant

`static_assert()` argument must evaluate to a numeric constant.

2703 Error — label 'x' referenced outside of any function

Label addresses can only be referenced within function scope.

2710 Error — variable length array 'x' at file scope

Variable length arrays may only be locally-scoped (or parameter) variables.

2711 Error — field 'x' declared as a variable length array

Structures and unions may not contain variable length arrays.

2712 Error — 'x' declared as function returning a function

Functions may not be used as a return type. Use a function pointer instead.

2750 Error — bad option(s)

Either a compiler option is invalid, or the options create an invalid combination. The message will have more details.

2998 Error — maximum error count exceeded — compilation halted.

If the *–fmaxerrcount=N* option is specified, and *N* errors have been discovered, the compiler halts compilation and emits this error message.

2999 Error — compilation halted due to previous errors

Previous errors have placed the compiler in a state at which continuing makes no sense. In this case, the compiler halts with no more output.

4010 Warning — CSECT name 'XXX' is too long, truncated to 'YYY'

The CSECT name specified, or determined by the compiler is longer than the allowed 7 characters. It will be truncated to 7 characters.

4011 Note — CSECT mapped to XXX avoid conflicts

Certain section names can cause conflicts with the IBM linker. To avoi d those, the CSECT name has been mapped to the given value.

4012 Error — CSECT name must have at least one alphabetic character.

The Systems/C compiler generates two sections, one for CODE and one for DATA, using the upper-cased CSECT name for the CODE section and the lower-cased CSECT name for upper case. Thus, to distinguish between these two, there must be at least one alphabetic character.

4013 Error — invalid code base register

The register specified in the *–fcode-base=X* option was invalid.

4014 Error — invalid frame base register

The register specified in the *–fframe-base=X* option was invalid.

4015 Error — –fc370=*ver* is required when –fxplink is specified

XPLINK compatibility mode is only available if LE370 compatibility mode is also enabled (specifying a version of LE to be compatible with). Add *–fc370=ver* to your command line.

4016 Error — –fhlasm is not allowed in combination with other options

The *–fhlasm* option cannot be used in combination with the other options that require the use of the **DASM** assembler.

Some of these options include *–fc370*, *–xplink*, and debuggable code that requests debugging information embedded in the generated object file.

4017 Error — –fno-alias-stmts is not allowed in combination with other options

The *–fno-alias-stmts* option cannot be used in combination with one or more of the other options because the generated assembly source requires the use of `ALIAS` statements.

4018 Error — bad option(s)

Either a compiler option is invalid, or the options create an invalid combination. The message will have more details.

4020 Error — invalid call to built-in 'XXX'

The call to the given built-in function *XXX* is invalid, either the number of arguments were wrong, or the arguments were of the wrong type.

4030 Error — can't open output ASM code file "XXX"

The compiler cannot open the specified file *XXX* for writing the assembly source.

4031 Error — can't write output assembly source.

The compiler has encountered an error when generating the output assembly source. The message is followed by a message from the operating system indicating the error.

4050 Warning — __register(XXX) variable conflicts with reserved register

A `__register` variable conflicts with a register which is reserved for function linkage. This means generated code will still use the specified GPR, possibly overwriting the variable or experiencing undefined behavior if the GPR is overwritten.

4060 Error — invalid __asm operand

A `__asm` expression specified an input or output operand that did not match its constraint, or there was an invalid constraint or clobber constraint.

4061 Error — invalid alias cycle in symbol 'XXX'

A combination of `__attribute__((alias(...)))` and `__asm__(...)` has produced an alias symbol that maps to itself. The assembly output will be `.set foo,foo` or `FOO EQU FOO`.

5000 Warning — parameter mismatch when attempting to inline call to 'XX' from 'XX'

The inliner attempted to inline a call and failed because the arguments mismatched. This could indicate a problem with prototypes or missing parameters and often indicates a problem present even without inlining. You can disable the inliner with *–fno-inline* if necessary.

5010 Warning — possible use of uninitialized variable 'variable'

Depending on the program flow, it is possible that the identified use of the variable occurs before the value has been assigned a value. This indicates that the value of the variable may be indeterminate at this point.

To avoid the message, ensure that the variable is assigned a value before its use.

9999 Error — internal error XXXXX

An internal consistency check or other error was encountered. Implementation and problem specific information is provided in the value *XXXXX*. Contact Dignus, LLC for assistance.

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 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