ANSI C Optimizing Compiler User Guide

INMOS Limited
The C compiler implementation was developed from the Perihelion Software "C" Compiler and the Codemist Norcroft "C" Compiler.

INMOS Document Number: 72 TDS 348 01
## Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>Host versions</td>
<td>iii</td>
</tr>
<tr>
<td>About this manual</td>
<td>iii</td>
</tr>
<tr>
<td>About the toolset documentation set</td>
<td>iii</td>
</tr>
<tr>
<td>Other documents</td>
<td>iv</td>
</tr>
<tr>
<td>occam and FORTRAN toolsets</td>
<td>iv</td>
</tr>
<tr>
<td>Documentation conventions</td>
<td>v</td>
</tr>
</tbody>
</table>

### User Guide ........................................................................... 1

1 **Introduction** ..................................................................... 3

1.1 Overview ............................................................................. 3

1.1.1 Advantages of using the optimizing ANSI C compiler ........ 3

1.1.2 When optimization should not be used ......................... 4

2 **User Interface** ................................................................. 5

2.1 Command line options ..................................................... 5

2.1.1 Disable optimization O0 ........................................... 6

2.1.2 Enable local optimization O1 ...................................... 6

2.1.3 Enable local and global optimization O2 ...................... 6

2.1.4 Optimize for time QT ............................................... 6

2.1.5 Optimize for space QS ............................................... 6

2.1.6 Enable side effects information messages .................. 7

2.1.7 Disable warning messages ........................................ 7

2.1.8 Compiler map files .................................................. 7

2.2 Language considerations ................................................ 7

2.2.1 const keyword .......................................................... 8

2.2.2 volatile keyword ..................................................... 8

2.2.3 register keyword ...................................................... 8

2.3 Pragmas ............................................................................. 8

2.3.1 No side effects .......................................................... 8

2.4 Messages ............................................................................ 10

2.4.1 Severities ..................................................................... 10

2.4.2 Standard terms .......................................................... 10

2.4.3 abstract declarator ................................................... 10

2.4.4 ANSI trigraphs ............................................................ 12

2.4.4 Information messages ............................................... 12

2.4.5 Warning diagnostics .................................................. 12
## Contents

2.4.6 Recoverable errors .............................................. 20
2.4.7 Serious errors .................................................. 27

### Appendices .......................................................... 39

#### A Local optimization examples .................................... 41
   A.1 Peephole optimization ............................................ 41
      Example: ......................................................... 41
   A.1.1 Summary of effects: ......................................... 41
   A.2 Flowgraph optimizations .......................................... 41
      A.2.1 Branch–chaining optimization ............................... 42
      A.2.2 Dead code elimination ...................................... 42
      A.2.3 Summary of effects of flowgraph optimizations: ........... 42
   A.3 Redundant store elimination .................................... 43
      Example: ......................................................... 43
   A.3.1 Summary of effects: ......................................... 43

#### B Global optimization examples .................................... 45
   B.1 Common subexpression elimination ............................. 45
      Example: ......................................................... 45
   B.1.1 Summary of effects: ......................................... 46
   B.2 Loop–invariant code optimization ............................... 46
      Example: ......................................................... 46
   B.2.1 Summary of effects: ......................................... 47
   B.3 Global optimization example ..................................... 47
   B.4 Tail–call and tail recursion optimization ...................... 47
      B.4.1 Example: (Tail–call optimization) ........................... 48
      B.4.2 Summary of effects: (Tail–call optimization) ............. 48
      B.4.3 Example: (Tail–recursion optimization) ..................... 49
      B.4.4 Summary of effects: (Tail–recursion optimization) ...... 49
   B.5 Workspace allocation by coloring ................................ 49
Host versions

The documentation set which accompanies the ANSI C toolset is designed to cover all host versions of the toolset:

- IMS D7314 – IBM PC compatible running MS–DOS
- IMS D4314 – Sun 4 systems running SunOS.
- IMS D6314 – VAX systems running VMS.

About this manual

This manual is the *ANSI C Optimizing Compiler User Guide* to the ANSI C toolset.

The manual provides reference and user information specific to the ANSI C optimizing compiler. It gives details of:

- command line options for invoking different levels of optimization;
- language considerations when using the optimizing compiler;
- details of pragmas which may be of use when optimizing;
- a list of error messages which may be generated by the optimizing compiler.

Examples of the local and global optimizations available are provided in the appendices.

This manual should be read in conjunction with the reference chapter for the standard ANSI C compiler, provided in the *Tools Reference Manual*. This is because details which are common to both compilers are not repeated in the *Optimizing Compiler User Guide*, including details of command line options and compiler pragmas.

About the toolset documentation set

The documentation set comprises the following volumes:

- *72 TDS 345 01 ANSI C Toolset User Guide*

  Describes the use of the toolset in developing programs for running on the transputer. The manual is divided into two sections; 'Basics' which describes each of the main stages of the development process and includes a 'Getting started' tutorial. The 'Advanced Techniques' section is aimed at more experienced users. The appendices contain a glossary of terms and a bibliography. Several of the chapters are generic to other INMOS toolsets.
• **72 TDS 346 01 ANSI C Toolset Reference Manual**

Provides reference material for each tool in the toolset including command line options, syntax and error messages. Many of the tools in the toolset are generic to other INMOS toolset products i.e. the occam and FORTRAN toolsets and the documentation reflects this. Examples are given in C. The appendices provide details of toolset conventions, transputer types, the assembler, server protocol, ITERM files and bootstrap loaders.

• **72 TDS 347 01 ANSI C Language and Libraries Reference Manual**

Provides a language reference for the toolset and implementation data. A list of the library functions provided is followed by detailed information about each function. Details are also provided about how to modify the runtime startup system, although only the very experienced user should attempt this.

• **72 TDS 348 01 ANSI C Optimizing Compiler User Guide (this manual)**

• **72 TDS 354 00 Performance Improvement with the DX314 ANSI C Toolset**

This document provides advice about how to maximize the performance of the toolset. It brings together information provided in other toolset documents particularly from the *Language and Libraries Reference Manual*. Note: details of how to manipulate the software virtual through-routing mechanism are given in the *User Guide*.

• **72 TDS 355 00 ANSI C Toolset Handbook**

A separately bound reference manual which lists the command line options for each tool and the library functions. It is provided for quick reference and summarizes information provided in more detail in the *Tools Reference Manual* and the *Language and Libraries Reference Manual*.

• **72 TDS 360 00 ANSI C Toolset Master Index**


**Other documents**

Other documents provided with the toolset product include:

• Delivery manual giving installation data, this document is host specific.

• Release notes, common to all host versions of the toolset.

**occam and FORTRAN toolsets**

At the time of writing the occam and FORTRAN toolset products referred to in this document set are still under development and specific details relating to them are
subject to change. Users should consult the documentation provided with the corresponding toolset product for specific information on that product.

Documentation conventions

The following typographical conventions are used in this manual:

**Bold type**

Used to emphasize new or special terminology.

**Teletype**

Used to distinguish command line examples, code fragments, and program listings from normal text.

**Italic type**

In command syntax definitions, used to stand for an argument of a particular type. Used within text for emphasis and for book titles.

**Braces {}**

Used to denote optional items in command syntax.

**Brackets []**

Used in command syntax to denote optional items on the command line.

**Ellipsis ...**

In general terms, used to denote the continuation of a series. For example, in syntax definitions denotes a list of one or more items.

| In command syntax, separates two mutually exclusive alternatives. |
1 Introduction

This chapter introduces the INMOS ANSI C optimizing compiler \texttt{icc} which generates optimized code for all 32-bit transputer targets. The compiler supports most options available on the standard ANSI C compiler provided with the ANSI C toolset, and has additional options to select and control optimization. The \textit{Tools reference} manual should be consulted for details of standard C compiler options. Section 2.1 of this manual identifies options not supported by the optimizing compiler.

1.1 Overview

The ANSI C optimizing compiler evaluates user's source code and converts it into highly efficient object code. The purpose of optimization is to improve the execution time of object code as well as the program's use of memory i.e. workspace or stack or code-size. Compiler options enable the user to control whether the optimization performed is predominantly to improve execution time or memory use. Optimization does not affect the functionality of the program.

The compiler implements both local and global levels of optimization:

- The global optimizations include: common subexpression elimination, loop invariant code motion and tail-call optimization. The optimizer examines each function as a single unit, enabling it to obtain as much information as possible about that function, while performing the optimization. Global optimization is more complex than local optimization; generally the more information available to the optimizer the better chance the optimizer has of improving code.

- The local optimizations include: flowgraph, peephole and redundant store elimination. To perform these optimizations efficiently, the optimizer only needs to operate on short sequences of code.

The ANSI C optimizing compiler supports a compiler pragma to enable the user to provide the compiler with more information about individual functions. The pragma may optionally be included in the source code by the user and is described in section 2.3.

1.1.1 Advantages of using the optimizing ANSI C compiler

The advantages of using the optimizing ANSI C compiler are that:

- It saves development time in producing efficient code because optimization can be achieved automatically at compile time.

- There are some optimizations which cannot be performed by the user at a source code level but can only be performed by the optimizing compiler at compile time.

- The compiler is able to analyze the cost/effectiveness of potential optimizations and will only apply an optimization where a saving can be made in either execution time or space.
1.1.2 When optimization should not be used

There are a number of situations when the user is recommended to use the standard ANSI C compiler as opposed to the optimizing compiler:

- When programs have not been fully debugged.
- When developing programs targeted at 16-bit processors.

It should also be noted that compile times are slower when a high level of optimization is being performed.
2 User Interface

This chapter describes the use of compiler options and pragmas which enable and control optimization by the ANSI C optimizing compiler. The chapter also describes features of the C language which influence optimization and provides a list of error messages which may be generated by the compiler.

The compiler supports most of the options and features available on the standard C compiler provided with the ANSI C toolset. The ANSI C Toolset Reference manual should be consulted for details of standard ANSI C compiler. The exceptions are described within the appropriate sections of this manual.

2.1 Command line options

The optimizing ANSI C compiler provides a range of command line options to facilitate a flexible approach to optimization. In addition support is provided (in the form of a compiler pragma) to exploit features of the C language in order to increase optimization.

The options for invoking optimization enable the user to control the level of optimization performed rather than controlling individual optimizations. They are listed in table 2.1 and described in subsequent sections.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSC</td>
<td>Enable side effects information messages.</td>
</tr>
<tr>
<td>O0</td>
<td>Disable optimization.</td>
</tr>
<tr>
<td>O1</td>
<td>Enable local optimization. This is the default.</td>
</tr>
<tr>
<td>O2</td>
<td>Enable both global and local optimization.</td>
</tr>
<tr>
<td>QT</td>
<td>Optimize for time. This is the default.</td>
</tr>
<tr>
<td>QS</td>
<td>Optimize for space.</td>
</tr>
<tr>
<td>WS</td>
<td>Disable warning messages about side effects.</td>
</tr>
</tbody>
</table>

Table 2.1 Optimizing options

The few options which are available on the standard ANSI C compiler but which are not supported by the optimizing compiler are listed in table 2.2.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Performs syntax checks only.</td>
</tr>
<tr>
<td>G</td>
<td>Generate debugging data. The optimizer should only be run on code which has been fully debugged by the standard ANSI C compiler.</td>
</tr>
<tr>
<td>T2, T3, T212, T222, T225</td>
<td>Compile for a 16-bit transputer.</td>
</tr>
</tbody>
</table>

Table 2.2 Options not supported by the optimizing ANSI C compiler

72 TDS 348 01

October 1992
2 User Interface

Note: assembler code inserts are treated the same way by both the standard and optimizing compilers.

2.1.1 Disable optimization 00

This option disables all optimization which can be specifically enabled at the command line using the 01 and 02 options.

2.1.2 Enable local optimization 01

This option enables the following local optimizations:

- Flowgraph optimization, including dead code elimination.
- Peephole optimization.
- Redundant store elimination.

Local optimizations are described, with examples, in appendix A.

In addition, workspace allocation by coloring is enabled. This is in fact a global optimization and is described in section 8.5.

2.1.3 Enable local and global optimization 02

This option, enables the following local and global optimizations:

- All optimizations enabled by option 01.
- Global common subexpression elimination.
- Loop invariant code motion.
- Tail-call optimization.
- Tail recursion optimization.

Global optimizations are described, with examples, in appendix B.

2.1.4 Optimize for time QT

This option controls how optimization is applied once it has been enabled by either the 01 or 02 options. The option instructs the compiler to only perform those optimizations which will not reduce the speed of the program. Where a choice exists between generating faster, but larger code over slower, more compact code, it will generate the faster code. This option is enabled by default.

2.1.5 Optimize for space QS

As above, but does the reverse i.e. only performing those optimizations which will not increase the size of the program. Where a choice exists between generating
faster, but larger code over slower, more compact code, it will generate the more compact code.

There is no definitive list for either the QT or QS options, as to which optimizations will or will not be applied. This will vary depending on the code being optimized.

2.1.6 Enable side effects information messages

The FSC option enables the generation of information messages about the 'side effect' characteristics of functions as the compiler performs optimization. The messages report the actions of the compiler to give the user visibility of how functions are treated with respect to side effects. The messages are purely informational and do not signal any required response from the user.

Information messages are listed in section 2.4. Side effects are discussed in more detail in sections 2.2 and 2.3.

2.1.7 Disable warning messages

The WS option disables messages warning users that functions marked as side effect free may in fact still cause side effects. See section 2.3.

2.1.8 Compiler map files

The map file generated by the optimizing ANSI C compiler differs slightly from that produced by the standard ANSI C compiler, in the following ways:

- The optimizing compiler may remove some local variables altogether. In such cases the variable will not occupy any space in workspace. These variables are listed at the end of the local variable map together with the message "Not Allocated".

- The full list of compiler temporaries is provided.

The format of the compiler map file is described for icc in the Toolset Reference manual.

2.2 Language considerations

Before the compiler can optimize a function call it has to be sure that the optimization is safe i.e. that it will not break the code. Therefore it will treat function calls with caution, assuming that they may modify global non-local variables, unless it can deduce with certainty their true behavior.

This section outlines language features which affect the implementation of optimization by icc.
2.2.1 const keyword

The const keyword states that after it is initialized, a variable cannot subsequently be modified by the program. For const variables, the compiler does not have to make worst case assumptions about its being modified when ambiguous modifications are seen. If a variable is never modified, then declaring it as const will, in general, allow the compiler to do a better job of optimizing.

Note: when pointers to const objects are used e.g.

```c
const char *p
```

the const keyword does not guarantee that the char will not be modified, just that it will not be modified through pointer p.

2.2.2 volatile keyword

The volatile keyword states that a variable may change asynchronously, or have other unknown side effects. The compiler will not move or remove any loads or stores to a volatile variable. volatile should be used for variables shared between parallel threads (or variables modified by interrupt routines), or variables which are mapped onto hardware devices.

2.2.3 register keyword

The register keyword is taken as a hint to the workspace allocator to allocate the variable at a small workspace offset.

2.3 Pragmas

The #pragma directive allows some compiler operations to be activated or deactivated in specific sections of code. Compiler pragmas are optional; when used they are inserted directly into the source code.

2.3.1 No side effects

The optimizing C compiler implements a compiler pragma to help the user maximize the degree of optimization performed.

The pragma enables the user to provide the compiler with more information about functions, whose behavior could otherwise be ambiguous. Its syntax is as follows:

```c
#pragma IMS_nosideeffects ( function )
```

where: function must have been declared but not defined or instanced before the pragma is encountered. A warning is generated and the pragma ignored if:
• the named function has been defined, part defined or instanced before
  the pragma is encountered;
• its parameter is not a function identifier;
• it is given more than one parameter or none at all.

Functions which are marked as side effect free enable the compiler to avoid mak­
ing worst–case assumptions about variables modified when the function is called.

A function is side effect free when, within the body of a function:

• there are no assignments to variables which are defined outside the func­
tion, including writing through a pointer where the pointer points to a vari­
able defined outside the function e.g. I/O or 'passing by reference' is not
  allowed;
• there are no assignments to static variables (although initialization of static
  variables within definitions are allowed);
• there are no reads from or assignments to volatile variables;
• there are no calls to functions which may have side effects.

The user must ensure that a function meets the above criteria before marking it
with the pragma IMS_nosideeffects. If the pragma is used incorrectly, unde­
defined results may occur. Assembler inserts i.e. _asm statements and pointers to
functions, are a particular risk area, as the user might unconsciously introduce a
function which does break the above criteria. For this reason, the compiler will
warn the user, each time assembler code or a pointer to a function is encountered.
Note: these warning messages may be disabled by using the _WS command line
option.

The compiler may mark a function as side effect free, even though the user has
not applied the IMS_nosideeffects pragma. Use of the FSC command line
option will cause an information message to be generated should this occur.

The compiler will only mark those functions that it detects as unambiguously side
effect free and that are not already marked. It is possible for there to be functions
which are side effect free that the compiler does not detect; for such functions, only
the application by the user of the pragma IMS_nosideeffects will cause these
functions to be marked as side effect free.

If a function that is side effect free is externally visible, then it may be useful to put
the pragma IMS_nosideeffects into the header file before the prototype of the
function. This enables the compiler to generate better code for calls to that function
from other files.
2.4 Messages

This section lists the information, warning and error messages which may be generated by icc when optimization is enabled.

2.4.1 Severities

Diagnostics are tagged with a severity level which indicates their effect on the compilation. Severity levels are the same as those used in the toolset standard but have slightly different meanings, which are described below.

*Information* messages provide the user with information about the functioning or performance of the tool. They do not indicate an error and no user action is required in response.

*Warning* severity diagnostics are generated whenever legal, but unorthodox programming styles are detected. Compilation is unaffected and object code is generated normally.

*Error* severity diagnostics are generated whenever the compiler detects a programming error from which it can recover. Compilation continues, but may abort if more errors are detected subsequently. No object code is generated.

*Serious* severity diagnostics are generated when programming errors are detected from which the compiler cannot recover. Compilation continues but code has been lost. No object code is generated.

*Fatal* errors indicate internal inconsistencies in the software and cause immediate termination of the operation with no output. Fatal errors are unlikely to occur but if they do the fact should be reported to your local INMOS distributor or field applications engineer.

*Error*, *Serious*, and *Fatal* diagnostic messages return error codes for handling by system MAKE programs and batch files.

2.4.2 Standard terms

This section explains some of the standard terms and notation used in compiler error messages.

**abstract declarator**

When using explicit casts or when passing an argument to `sizeof()`, a data type must be specified. This can be done by declaring an object of the correct type without specifying the name of the object. Declarations of this
type are called abstract declarations, because they apply to no known object.
Examples of abstract declarations are:

```c
(int) a = b;  /* 'int' is the abstract declarator */
sizeof(int [3]); /* 'int [3]' is the abstract declarator */
```

**char**

Stands for a single ASCII character.

**context**

Stands for a type, for example, 'character constant', 'integer constant', and
'string constant'.

**deprecated declaration**

This means that a function declaration is incomplete. Declarations should
specify the type of the function and the type of each formal parameter. If
there are no parameters then the function type `void` should be specified.

**expression**

Stands for a C expression.

**filename**

A file name.

**function prototype**

A function declaration which usually precedes the function definition. It
declares the function's type and the types of its parameters.

**identifier**

A C identifier, for example, a variable or function name.

**initializer**

An initial value which is assigned to an object at the time of its declaration.

**message string**

The string which follows a compiler directive.

**op**

An operator. Valid operators include: `"++", "--", "->", "<=",` and the unary
operators `&`, `*`, `+` and `-.`
store class
A C storage class. Valid classes are static or extern.

string
Any string of ASCII characters.

struct/union
A variable of type struct or union.

type
A type identifier.

void context
This can occur at any point in a program where a value is not expected, for example, calling a function without using the returned number.

instruction
A transputer instruction, or a pseudo-instruction as accepted by the __asm construct.

2.4.3 ANSI trigraphs

The ANSI specification includes a number of three character sequences that can be used to represent certain ASCII characters that may not be present on all keyboards. These sequences, known as trigraphs, are used in compiler error messages to stand for these characters.

ANSI standard trigraph sequences consist of a sequence of 2 question marks followed by a third character. A complete list of ANSI trigraphs is given in the chapter titled 'New features in ANSI C ' of the accompanying ANSI C Language and Libraries Manual.

2.4.4 Information messages

These messages are prefixed by the words 'Information -'; they do not signal any required response from the user.

Function identifier has been marked as side effect free

The compiler has checked that the named function is side effect free and has marked it as such, from this point on in the compilation. Marking the function as side effect free may improve the generated code.

2.4.5 Warning diagnostics

These messages are prefixed by the words 'Warning -' and indicate that unexpected results may occur.
#define macro identifier defined but not used

The named macro has been defined, but not referenced in the rest of the program. This message is only generated if specifically enabled by the 'FM' compiler option.

'&' unnecessary for function or array identifier

A pointer to a function or array is implied by use of the name alone; the ' & ' operator is not required.

'int identifier ()' assumed - 'void' intended?

A function was defined without specifying its type. The compiler assumes a function of type int if no type is specified.

identifier already has a descriptor defined, pragma ignored

The pragma IMS_descriptor has already been applied to identifier, more than one application is invalid.

identifier has been called; pragma ignored

The pragma must be applied to identifier before the latter has been called.

identifier has been defined; pragma ignored

The pragma must be applied to identifier before the latter is defined.

identifier has not been declared; pragma ignored

The pragma must be applied to identifier after the latter has been declared.

identifier is not a function; pragma ignored

The argument to the pragma must be a function name.

identifier is not externally visible; pragma ignored

The first argument to the IMS_descriptor pragma must be the name of an externally visible function.

identifier may be used before being set

The compiler has detected a use of a variable which may not have been initialized.

identifier multiply translated, this translation ignored

The IMS_translate pragma has been applied to identifier more than once.

'j' to immediately following label identifier will be removed

In an assembler insert, there is a j (jump) instruction to an immediately following label. This is effectively a no-op, and is removed.

If the user really requires a j 0 instruction, for breakpointing or descheduling purposes, he should write j 0 in the assembler insert.
number treated as number UL in 32-bit implementation

No type was specified for the number. The compiler assumes unsigned long if no type was specified.

op: cast between function pointer and non-function object

The operation is performed upon two arguments, one of which is a function, and the other an object.

A very suspect way of writing through a pointer has been detected in function which is marked as side effect free

The named function is marked side effect free and has some code in it to write through a pointer either in an unportable manner or in a way that is typically considered bad programming practice, e.g. *(int *)23 = ...; it is up to the user to carefully check that the assignment is conformant with the definition of side effect free.

actual type type mismatches format "%char"

The type of an argument to printf or scanf does not match that implied by the control string.

ANSI 'char char char' trigraph for 'char' found – was this intended?

The specified three character sequence was found in the source program. This has been treated as an ANSI trigraph and substituted for the character shown.

argument and old-style parameter mismatch: expression

There is an old (non-prototype) style function definition in scope, and the type of an argument (after default argument promotion has taken place) does not agree with the type of the corresponding formal parameter.

Be sure that assignment through pointer in function is side effect free

The named function is marked side effect free and assigns through a pointer; it is up to the user to carefully check that the assignment is conformant with the definition of side effect free.

Be sure that functions pointed to in function are side effect free

The named function is marked side effect free and calls functions through pointers to them; it is up to the user to carefully check that the functions which may be called in this way are themselves side effect free.

Be sure that the assembler in function is side effect free

The named function is marked side effect free and uses assembly language inserts; it is up to the user to carefully check that assembler is conformant with the definition of side effect free.

Cannot delete temporary file filename

Host file system error.
Cannot generate stack check for function (pragma nolink applied)

A stack check requires a static link, and the function function has been specified not to receive a static link (using IMS_nolink). icc compiles the function with the stack check omitted.

character sequence /* inside comment

The start-of-comment character sequence was detected within a comment. Check that the previous comment was terminated correctly.

Dangling 'else' indicates possible error

Within nested if ... else constructs, there is some ambiguity as to which 'if' relates to which 'else'.

Deprecated declaration identifier () – give arg types

In the prototype declaration of the named function, the argument's names and/or their types were not specified.

division by zero: op

Division, or remainder, by zero, will cause overflow.

Expected ');' perhaps you tried to give too many names – pragma ignored

A ')' was expected but not found in a pragma; it may be that too many parameters have been given.

Expected integer as argument – pragma ignored

An integer argument was expected but not found in a pragma.

Expected string as argument – pragma ignored

The argument to the IMS_linkage pragma must be a string literal.

Expected string as fifth argument; pragma ignored

The fifth argument to the IMS_descriptor pragma must be a string literal.

Expected string as second argument – pragma ignored

The second argument to the IMS_translate pragma must be a string literal.

extern 'main' needs to be 'int' function

In a declaration of main( ), the function should always be declared as type int.

extern identifier not declared in header

All objects must be declared before use. This message is only generated if specifically enabled by the 'FM' compiler option.
floating point constant overflow: $op$

Floating point overflow occurred during addition, subtraction, multiplication or division of two constants.

floating to integral conversion failed

Conversion (casting) from a floating point type to an integral type (such as int) failed.

formal parameter $identifier$ not declared — ‘int’ assumed

A formal parameter has been listed in the parameter list of the function definition, but there is no entry for it in the declaration list; it is therefore assumed to be of type int.

Format requires $count$ parameter(s), but $count1$ given

A call to printf or scanf was made with the incorrect number of arguments. The control string indicated that $count$ arguments are needed, but $count1$ were provided. This warning is only generated if pragma IMS_on (pc) is active. The header file stdio.h includes this pragma.

Global optimisation suppressed for $function$ as it contains assembly code:

The user has attempted to globally-optimize a function which contains an assembler insert: the compiler automatically turns the global optimizer off. (This applied only to the named function: the global optimizer will be turned on again for subsequent functions.)

Illegal format conversion ‘%char’

The character sequence ‘%char’ is not a legitimate conversion character for printf or scanf. This warning is only generated if pragma IMS_on (pc) is active. The header file stdio.h includes this pragma.

Illegal language type $string$; replaced by $string$

The language type given for the interface descriptor-string is not a valid one, and has been overridden by a known type.

implicit cast (to $type$) overflow

Overflow occurred when casting an expression.

implicit narrowing cast: $op$

The result of an operation performed at higher precision is immediately, and implicitly, cast to lower precision, thus losing the extra precision: if the extra precision is not required, the operation ought to be performed at the lower precision.

If the narrowing cast is really required, the warning may be suppressed by writing the cast explicitly.

implicit return in non-void $identifier$()

The function does not contain a return statement, even though it is defined to return a value.
2.4 Messages

Incomplete format string

The control string for use with printf or scanf is incomplete. This warning is only generated if pragma IMS_on (pc) is active. The header file stdio.h includes this pragma.

Integer too large to be represented – pragma ignored

An integer parameter to a pragma has been given with a value too large to be able to be dealt with by the compiler.

inventing 'extern int identifier();'

No declaration exists for the function; it will be defined by default as extern int.

label identifier was defined but not used

The named label was set, but not used.

Linkage already set - pragma ignored

The IMS_linkage parameter has been specified more than once.

lower precision in wider context: op

The result of an operation performed at lower precision is immediately cast to a higher precision; it may be that the user was expecting the operation to be performed at the higher precision.

Missing comma in pragma argument list – pragma ignored

Multiple arguments to a pragma must be separated by commas.

Negative value given for vectorspace – pragma ignored

Vectorspace values in the IMS_descriptor pragma must be \( \geq 0 \).

Negative value given for workspace – pragma ignored

Workspace values in the IMS_descriptor pragma must be \( \geq 0 \).

No pragma name given in pragma directive – was this intended?

The compiler has detected a pragma directive which does not have a name. This is not illegal, however, it has no effect.

no side effect in void context: identifier

The value which has been returned by an expression is not being used e.g.

```c
int a;
a;
```

non-portable – not 1 char in ‘...’

The characters enclosed by single quotes represent more than one character. The compiler will read the first character only, for example, ‘AB’ will be read as ‘A’.

72 TDS 348 01

October 1992
Non-positive values for patch size are meaningless – pragma ignored

Patch size values must be > 0.

non-value return in non-void function

A function which should return a value has terminated without using a return statement or with a return statement that has no arguments. The value received from the function by the calling routine is undefined.

odd unsigned comparison with 0 : \( op \)

\( a \geq \) comparison of an unsigned integer with zero, or \( a \leq \) comparison of zero with an unsigned integer, is always true.

omitting trailing ‘\( 0 \)’ for char \( \text{[count]} \)

The char array is fully occupied by characters and there is no room to append the string terminator (\( \text{"0"} \)). \text{count} is the full length of the character array.

repeated definition of \#define macro identifier

The named macro has been defined more than once. The definitions are identical.

Shift of type by \text{count} undefined in ANSI C

A shift of more than the number of bits in type, or less than zero was requested, undefined in ANSI C.

signed constant overflow: \( op \)

Overflow occurred when performing \( op \) upon signed, constant operands.

spurious {} around scalar initialiser

A scalar can take only one initializer, so there is no need to use braces as are required with aggregate types such as arrays.

static identifier declared but not used

The named static object was declared but not used.

struct has no named member

A structure has been declared without any members.

Too many assembler arguments
Too many compiler arguments

There are too many options on the command line. The extra options are ignored.

typedef identifier declared but not used

The named identifier has been declared, but is not used in the program.
2.4 Messages

Undefined macro string in #if – treated as 0

This error occurs when enumeration or undefined constants appear after the preprocessor #if directive. For example, if 'ab' and 'cd' are enumeration constants of the enumerated type 'abcd', the statement #if ab == cd would generate this error.

Union has no named member

A union has been declared without any members.

Unnamed bit field initialised to 0

A static declaration of a structure or union containing an unnamed bit field, the compiler has initialized that field to zero.

Unrecognised #pragma (no '(')

The arguments to a pragma are not correctly enclosed in parentheses.

Unrecognised #pragma (no ')

The arguments to a pragma are not correctly enclosed in parentheses.

Unrecognised #pragma identifier

Identifier is not a pragma recognized by this compiler.

Unsigned constant overflow: op

Overflow occurred when performing op upon unsigned, constant operands.

Unused earlier static declaration of identifier

There is a forward declaration of identifier which is not necessary as the definition of identifier appears before identifier is referenced.

Use of op in condition context

Generated when the invalid operators '=' (assignment) or '~' (bit-not) are used in a condition statement.

This message is given for use of the assignment operator in condition context, e.g.

if (a = b)

as this is often due to mistyping the equality operator, i.e. the desired code is:

if (a == b)

If you really wish to perform the assignment in condition context, the warning message may be suppressed using the form:

if ((a = b) != 0)

Wrong number of parameters to function

A function declared without a prototype was called with the wrong number of arguments. (An error is given if a function declared with a prototype is called with the wrong number of arguments.)
variable `identifier` declared but not used

The variable was declared, but not used anywhere in the program.

(possible error): >= number lines of macro arguments

There are a surprisingly large number of lines of arguments to a macro; this may indicate a syntax error.

2.4.6 Recoverable errors

These messages are prefixed by the words ‘Error –’.

`#ident` is not in ANSI-C

`#ident` is not a recognized preprocessor directive.

`##` first or last token in `#define` body

The `##` preprocessor operator must be preceded by a preprocessor token, and succeeded by a preprocessor token.

`,` (not `;`) separates formal parameters

A semicolon has been used to separate the formal parameters in a function definition (as in Pascal) instead of a comma.

`register` attribute for `identifier` ignored when address taken

An attempt was made to take the address of a variable with `register` storage class. The register attribute will be ignored allowing the address to be taken.

```c
<int> op <pointer> treated as <int> op (int) <pointer>
```

The expression involving a integer and a pointer will result in the pointer being converted (cast) to an integer.

`function` marked as side effect free assigns to a global variable

An assignment to a global variable is a side effect.

`function` marked as side effect free assigns to static

An assignment to a static variable, other than an initialization, is a side effect.

`function` marked as side effect free calls `function` which is not side effect free

The call of a function which is not side effect free is a side effect.

`function` marked as side effect free uses volatile variable

The read of or write to a volatile variable is a side effect.

`instruction` may not have a size specified

An `__asm` pseudo-instruction may not be explicitly sized.
object identifier may not be function — assuming pointer

An attempt was made to use a function where it was not expected, typically when a function is included as a component within a structure.

\textit{op : cast to non-equal type illegal}

A structure or union has been cast into a structure or union of a different type. The cast is illegal and will be ignored.

\textit{op : illegal cast to type}

An illegal cast has been attempted. The cast is illegal and will be ignored.

\textit{op : implicit cast of type to 'int'}

A non-integer object has been used where an int was expected, for example, attempting to use a double as an argument to a switch statement (which requires an integer type).

\textit{op : implicit cast of non-0 int to pointer}

Evaluation of the expression will result in the cast of an integer to a pointer.

\textit{op : implicit cast of pointer to 'int'}

Evaluation of the expression will result in the cast of the pointer to an integer.

\textit{op : implicit cast of pointer to non-equal pointer}

Evaluation of the expression will result in the cast of one pointer type to another.

\textit{op may not have whitespace in it}

Two-character operators such as `+=` must not contain spaces.

\textit{<pointer> operator <int> treated as (int) <pointer> operator <int>}

Evaluation of the expression will result in the cast of the pointer to an integer.

\textit{Ancient form of initialisation, use '='}

A }, rather than =, was used to introduce an initializer, this is no longer legal C.

\textit{ANSI C does not support 'long float'}

An object has been declared of type long float, this is illegal in ANSI C, which supports float, double, or long double.

\textit{Array of type illegal — assuming pointer}

An array of functions or void objects has been declared. The compiler treats this as an array of pointers to functions or void objects.
Array [0] found
An empty array has been defined and will be set up instead as an array with one element.

assignment to ‘const’ object identifier
The expression contains an assignment to a constant. The assignment will be carried out.

const typedef identifier has const respecified
A typedef which is already qualified with const, has been qualified with const.

comparison op of pointer and int: literal 0 (for == and !=) is only legal case.
The specified operator was used to compare an object of type int and one of a type pointer. The only legal comparison of this type is between a pointer and 0 using either == or !=.

declaration with no effect
No name has been declared for the object. Specifying only the type of an object generates this error.

differing redefinition of #define macro identifier
The named macro has been defined more than once. The definitions are not identical.

Digit 8 or 9 found in octal number
8 and 9 are meaningless in an octal number.

duplicate macro formal parameter: 'identifier'
The function macro has two formal parameters with the same name.

duplicate member identifier1 of identifier2
Two fields of structure or union identifier2 have the name identifier1.

ellipsis (...) cannot be only parameter
A function declared to take a variable number of parameters must have at least one known parameter.

enumeration constant identifier too large to represent as 'int' – 0 assumed
The value of an enumeration constant has overflowed the range of ints.

extern identifier mismatches top-level declaration
An extern declaration of identifier within a function definition does not match an extern declaration of identifier at the top level.
expected symbol1 or symbol2 – inserted symbol1 before symbol3

symbol1 or symbol2 was expected before symbol3, but neither was found. symbol1 is suggested as the most appropriate choice and the compiler has changed the code accordingly.

formal name missing in function definition

The type of a formal parameter has been omitted in a function definition.

function identifier may not be initialised – assuming function pointer

Initializers cannot be used in function declarations or definitions.

function prototype formal identifier needs type or class – ‘int’ assumed

The type of a formal parameter has been omitted in a function declaration and int has been assumed.

function returning type illegal — assuming pointer

The user has appeared to declare a function which returns a function or an array.

hex number cannot have exponent

A hex number ending in e may not be immediately followed by + or –; separate the number and the additive operator with white space.

illegal bit field type type – ‘int’ assumed

Bit fields cannot be set within non integral variables. The compiler assumes an int instead.

illegal option –D identifier identifier

The compiler D option must be specified for each assignment.

illegal string escape ‘char’ – treated as char

The character following \ does not form part of a valid string escape. The compiler treats the sequence \char as char.

illegal [] member: identifier

An open array may not be a member of a structure or union.

junk at end of #identifier line – ignored

The text following the directive is invalid and will be ignored.

linkage disagreement for identifier – treated as store class

The storage class of a previously defined static or extern object or function disagrees with the current declaration. The object will be treated as though it is in storage class store class.

L’…’ needs exactly 1 wide character

A wide character constant should contain exactly one wide character.
Missing newline before EOF - inserted
   A blank line should have been inserted before the end-of-file character.

Missing type specification - ‘int’ assumed
   A type specification is missing. The object will be assumed to be of type int.

more than 4 chars in character constant
   More than 4 ASCII characters were used to represent a character constant. When using the single quote syntax for character constants a maximum number of 4 characters is permitted in order to accommodate the octal representation of a character. The first 4 characters will be used.

Negative numbers and zero are not allowed in #line
   ANSI C forbids negative numbers or zero in a #line directive.

no chars in character constant ""
   No characters or character codes have been specified for the character constant. A NULL character is assumed.

no initializer list in braced initializer
   There must be at least one entry in the initializer list of a braced initializer.

number illegally followed by letter
   A numerical constant may not be followed immediately by a letter.

number missing in #line
   There is no line number following the preprocessor #line directive.

Numbers greater than 32767 are not allowed in #line
   ANSI C forbids numbers greater than 32767 in a #line directive.

objects that have been cast are not l-values
   An object that has been cast in l-value context; ANSI has made this illegal.

Omitted type before formal declarator – ‘int’ assumed
   No type was specified; type int will be assumed.

operand of # not macro formal parameter: ‘identifier’
   The operand to the # preprocessor operator must be a formal parameter of the function macro containing it.

overlarge escape \number1 treated as \number2
   An octal number in an escape sequence is too large to be represented in the target architecture.
overlarge escape \texttt{\textasciicircum xnumber1} treated as \texttt{\textasciicircum xnumber2} 

A hexadecimal number in an escape sequence is too large to be represented in the target architecture.

parentheses (... ) inserted around expression following text 

Parentheses were expected after the specified text, for example, around a conditional expression such as an \texttt{if} statement.

prototype and old-style parameters mixed 

It is illegal to mix new (prototype) and old-style parameter declarations.

return expression illegal for void function 

A return statement with an expression was found within a \texttt{void} function. The return statement is ignored.

signed constant overflow: \texttt{op} 

Overflow occurred when performing \texttt{op} upon signed, constant operands.

size of 'void' required – treated as 1 

'void' was used as an argument to \texttt{sizeof}. The compiler assumes the size of void to be 1.

size of a [] array required, treated as [1] 

The array is of unspecified size. In these circumstances \texttt{sizeof} return the size of the array type.

size of function required – treated as size of pointer 

A function name was passed to the \texttt{sizeof} function. In these circumstances \texttt{sizeof} returns the size of the pointer to the function.

size\texttt{of bit field} illegal – \texttt{sizeof(int)} assumed 

A bit field was passed to the \texttt{sizeof} function. In these circumstances \texttt{sizeof} casts the bit field to an integer and then returns its size.

Small (single precision) floating value converted to 0.0 

The number is too small to represent in a single word (32 bit) floating point format, and has been rounded to 0.0.

Small floating point value converted to 0.0 

The number is too small to represent in a double word (64 bit) floating point format, and has been rounded to 0.0.

Spurious \texttt{#elif} ignored 

The \texttt{#elif} directive could not be matched with a corresponding \texttt{if} directive and has been ignored.
Spurious #else ignored

The #else directive could not be matched with a corresponding if directive and has been ignored.

Spurious #endif ignored

The #endif directive could not be matched with a corresponding if directive and has been ignored.

static function identifier not defined –treated as extern

A function was defined as static in the function prototype, but the compiler was unable to find the function definition. An extern function is assumed.

string initialiser longer than char [count]

A character array has been initialized with more characters than the array can accommodate. Since the compiler adds a terminating NULL character to strings, string initializers should always contain one less element than the array.

struct has no members

A structure definition must contain at least one member.

struct member identifier may not be function – assuming pointer

A structure member was declared of function type; the compiler treats this as pointer to function type.

struct tag identifier not defined

A structure has been referenced before being defined.

Translation unit contains no external declarations

A translation unit must contain at least one external declaration.

type or class needed (except in function definition) –‘int’ assumed

The type or storage class has been omitted from the function declaration.

Undeclared name, inventing ‘extern int identifier’

An undeclared identifier was encountered and will be given the storage class extern.

union has no members

A union definition must contain at least one member.

union member identifier may not be function – assuming pointer

A union member was declared of function type; the compiler treats this as pointer to function type.
union tag identifier not defined
   A union has been referenced before being defined.

unprintable char number found - ignored
   An unprintable character was found in the source text.

volatile typedef identifier has volatile respecified
   A typedef which is already qualified with volatile, has been qualified
   with volatile.

wrong number of parameters to function
   A function was called with the wrong number of arguments.

2.4.7 Serious errors

These messages are prefixed by the words 'Serious -'.

\space and \tab are invalid string escapes
   Whitespace ('\space' or '\tab') was found within a string. All characters up
   to the first non-whitespace character are ignored; if the first non-white-
   space character is a newline character, this will also be ignored.

{} must have 1 element to initialise scalar or auto
   When initializing a scalar quantity or auto variable only one initializer
   should be specified within the enclosing braces.

#error encountered string
   The #error directive was found.

#include file filename wouldn't open
   The file filename could not be opened.

op : cast to non-equality type illegal
   A structure or union has been cast into a structure or union of a different
   type. The cast is illegal and will be ignored.

op : illegal cast of type to pointer
   A variable has been cast into a pointer type. The cast is illegal and will be
   ignored.

op : illegal cast to type
   An illegal cast has been attempted. The cast is illegal and will be ignored.

context: illegal use in pointer initialiser
   An object of type auto, or its address, cannot be initialized.
(. . .) must have exactly 3 dots
   An ellipsis must consist of three dots.

'break' not in loop or switch – ignored
   A break statement was encountered outside the scope of a loop or switch statement. A break at this point is illegal and will be ignored.

'case' not in switch – ignored
   A case prefix has been encountered outside the body of a switch statement. A case statement at this point is illegal and will be ignored.

'continue' not in loop – ignored
   A continue statement has been encountered outside the body of a loop. A continue statement at this point is illegal and will be ignored.

'default' not in switch – ignored
   A default prefix has been encountered outside the body of a switch statement. A default prefix at this point is illegal and will be ignored.

'goto' not followed by label – ignored
   The text following a goto statement does not represent a label.

'verd' values may not be arguments
   Formal parameters in function definitions or declaration cannot be of type void.

'while' expected after 'do' – found text
   The while statement is missing from a do . . . while construct. text marks the position.

'{' of function body expected – found text
   The opening brace in the body of a function is missing.

'{' or <identifier> expected after type, but found text
   The opening brace following a struct, union or enum is missing. text marks the position.

<asm-directive> expected but found a text
   text indicates where the __asm directive was expected.

<command> expected but found a text
   Statements such as switch or if should be followed by a command. text indicates where the command was expected.

<expression> expected but found text
   text indicates where the expression was expected.
**2.4 Messages**

*<identifier> expected but found text in ‘enum’ definition*

The compiler was expecting to read an enumeration constant when it found symbol. This may be because there is a spurious comma at the end of a list of enumeration constants.

*function has pragma nolink specified, but accesses static data*

The specified function has been specified not to receive a static link (via `IMS_nolink`), but attempts to use static data. It is only possible to use static data when a static link is available.

*identifier is not a label - ldlabeldiff ignored*

The operands to the `ldlabeldiff` pseudo-instruction must be labels.

*identifier1 has pragma nolink specified, but accesses static identifier2*

Function `identifier1` has had the `IMS_nolink` pragma applied to it, which means it cannot access static data.

*identifier1 has pragma nolink specified, but addresses static identifier2*

Function `identifier1` has had the `IMS_nolink` pragma applied to it, which means it cannot address static data.

*instruction not followed by label - ignored*

A load or store _asm instruction must have a constant or label operand.

*store class variables may not be initialised*

Some types of C variables, such as those declared as `extern`, cannot be initialized.

Array size count illegal – 1 assumed

Arrays cannot be larger than 0xffffffff on a 32-bit target, or 65535 on a 16-bit target.

attempt to apply a non-function

A name not declared as a function has been used in a context where a function should be.

attempt to include struct/union identifier object/member within itself

A structure or union declaration may not contain a field of the structure or union type, or a field which references another field.

bit fields do not have addresses

Elements of type bit field in C structures cannot be addressed.

Bit size size illegal – 1 assumed

Bit sizes greater than 32 are set to 1.
Cannot address built-in variable identifier

identifier is a built-in name, such as _lsb or _params, which cannot be addressed.

Cannot call function (it requires a static link)

An attempt has been made to call the specified function which requires a static link, from a function which has been specified not to receive a static link (via IMS_noLink).

Cannot do indirect call (it requires a static link)

An attempt has been made to call a function from a function which has been specified not to receive a static link (via IMS_noLink). All calls through function pointers are assumed to require a static link.

Cannot write to built-in variable identifier

identifier is a built-in name, such as _lsb or _params, which cannot be assigned to.

char and wide (L"...") strings do not concatenate

A char string and a wide char string appear adjacently in the source text. Normally, adjacent strings in the source text are concatenated; however, this is not possible here, as they have different types.

Digit required after exponent marker

Exponents of floating point numbers must be followed by a numeric character. The numeric character may be preceded by '+' or '-'.

duplicate 'default' case ignored

The default prefix has already been specified for the switch construct. The original definition will be used.

duplicate definition of identifier

The named identifier has already been defined.

duplicate definition of struct/union tag identifier

The named structure or union identifier has already been used.

duplicate definition of label identifier – ignored

The specified identifier has already been used. The original definition will be used.

duplicate type specification of formal parameter parameter

The specified parameter has been listed more than once in the function's formal parameter list.

duplicated case constant: constant

The constant has been specified more than once in the same case statement.
2.4 Messages

EOF in comment
The end-of-file was detected inside a comment.

EOF in string
The end-of-file was detected within a string.

EOF in string escape
The end-of-file was detected within a string escape sequence.

EOF not newline after #if . . .
The end-of-file was found after the '#if' directive; a newline character was expected.

expected symbol
symbol was expected.

expected symbol1 − inserted before symbol2
symbol1 was expected before symbol2 and the compiler has changed the code accordingly. For example, in the code "if (TRUE printf();" the compiler would expect to find ')' before 'printf'.

expected symbol1 or symbol2
Either symbol1 or symbol2 was expected.

Expected <identifier> after operator but found text
The specified operator must be followed by an identifier. This error may occur after the structure member operator '.' and the structure pointer operator '->'.

Expecting <declarator> or <type>, but found text
An identifier or type was expected at text. For example, the declaration 'typedef int *[3] test;' generates this error.

Grossly over-long floating point number
There are too many digits in the floating point number. The compiler reads the maximum number of digits allowed and discards the rest.

Grossly over-long hexadecimal constant
There are too many digits in the hexadecimal number. The compiler reads the maximum number of digits allowed and discards the rest.

Grossly over-long number
There are too many digits in the decimal number. The compiler reads the maximum number of digits allowed and discards the rest.

Hex digit needed after 0x or 0X
The hexadecimal specifier 0x must be followed by a valid hexadecimal digit. The compiler assumes a zero digit.
Identifier (name) found in <abstract declarator> – ignored

An identifier should not be used in an abstract declarator. This error is generated, for example, if sizeof(int *test[3]) ; is used instead of the correct form sizeof(int *[3]);.

illegal character (number = 'char') in source
illegal character (hex code number) in source

An unexpected character was found in the source code. The ASCII code of the character (if printable), and the character itself, are given.

illegal in context: error

Illegal expressions such as those involving division by zero generate this error.

illegal in expression: non constant identifier

A constant is required in certain expressions, for example after a case prefix.

Illegal in l-value: 'enum' constant identifier

Enumeration constants cannot be used as l-values in an expression.

Illegal in l-value: function or array identifier

Arrays and function declarators cannot be used as l-values. This error would be generated, for example, by attempting to assign a value to a function declarator.

Illegal in the context of an l-value: op

The operator op cannot appear in l-value context.

Illegal types for operands: operator

The operator has been used with an invalid type. For example, it is illegal to use the structure member operator '. ' with a variable of type int.

Illegal 'void' member/object: identifier

An object or member of a structure or union cannot be declared as being of type void.

incomplete tentative declaration of identifier

The declaration of identifier has gone out of scope before the declaration has been completed.

Invalid command line option (text)

text is not a recognized command line option.

Invalid source file name (filename)

filename is not a valid source file name. (Source file names may not contain hyphens.)
I/O error writing filename
   An error occurred when writing to the named file.

Junk after #if expression
   The #if directive must be terminated by a newline character.

Junk after #include filename
   The #include directive must be terminated by a newline character.

label identifier has not been set
   A label has been referenced but not set. This message will be generated if goto is used with an undefined label.

ldlabelediff not followed by label - ignored
   The operands to the ldlabelediff pseudo-instruction must be labels.

Misplaced ‘else’ ignored
   An else statement was found where it was not expected. It will be ignored.

Misplaced ‘{’ at top level – ignoring block
   An opening brace was found at the top level of a program when it was not expected, for example when not used as part of a function or structure definition.

Misplaced preprocessor character char
   A preprocessor directive character (# or \) was found where it was not expected.

Missing #endif at EOF
   An 'endif' directive is missing. This error will not be generated until the last of the currently open files is about to be closed (ANSI standard does not require #if and #else statements to match in included files).

Missing char in preprocessor command line
   A ‘quote’ character is missing from a preprocessor command line. The missing character could be ’, <, >, or ”.

Missing ‘)’ after identifier ( . . . on line number
   A closing parenthesis is missing from the macro which will be substituted at line number.

Missing ‘,’ or ‘)’ after #define identifier ( . .
   The list of parameters in a macro definition is either incomplete or has not been correctly terminated by a closing parenthesis.

Missing < or ” after #include
   The opening ‘quote’ character which introduces the filename is missing.
Missing hex digit(s) after `\x`
The hexadecimal introducer sequence `\x` was found, but no hexadecimal digit was specified. The compiler assumes that the letter \x was intended.

Missing identifier after `#define`
The definition is empty. `#define` must be followed by an identifier.

Missing identifier after `#ifdef`
`#ifdef` must be followed by an identifier.

Missing identifier after `#ifndef`
`#ifndef` must be followed by an identifier.

Missing identifier after `#undef`
`#undef` must be followed by an identifier.

Missing include directory name
The `J` command line option must be followed by a directory name.

Missing map file name
The `P` command line option must be followed by a map file name.

Missing object file name
The `O` command line option must be followed by an object file name.

Missing parameter name in `#define identifier (...)`
A parameter is missing from the specified macro definition. This error would be generated by a definition of the form `#define test(arg,)`.

Newline or end of file within string
A newline or end-of-file character was encountered within a string.

No `)’` after `#if defined(...)`
The closing parenthesis is missing from the directive.

No file name given
No source file was specified on the command line.

No identifier after `#if defined`
`#if defined` must be followed by an identifier.

Non-formal `identifier` in parameter-type-specifier
The parameter `identifier` was included in the declarator list of a function, but not in the parameter list. For example, a definition such as `int foo() int x; { }` would generate this error.
non-static address \texttt{identifier} in pointer initialiser

Pointers cannot be initialized with the address of an object of type \texttt{auto}.

Number \texttt{number} too large for 32-bit implementation

The specified number is too large to be represented in 32 bits.

objects of type 'void' can not be initialised

Initializing objects of type \texttt{void} is illegal.

only const and volatile can qualify a pointer: found \texttt{type}

The only type qualifiers of a pointer are \texttt{const} and \texttt{volatile} but \texttt{type} was found instead.

Operand \texttt{number} to \texttt{instruction} is larger than a word

The arguments to an \texttt{__asm load or store pseudo-instruction} must fit in a machine word.

Operand \texttt{number} to \texttt{instruction} is not word-sized

The arguments to an \texttt{__asm store pseudo-instruction} must fit exactly in a machine word.

Operand to \texttt{instruction} must be a constant or local variable

An illegal operand has been given to an \texttt{__asm ld1 or stl} instruction.

Operand to \texttt{instruction} is larger than a word

The operand to a primary instruction inside \texttt{__asm} must fit in a machine word.

Out of memory
Out of store (for error buffer)
Out of store (in \texttt{cc\_alloc})

The compiler ran out of available memory.

Overlarge (single precision) floating point value found

The number is too large to represent in single word (32 bit) floating point format.

Overlarge floating point value found

The number is too large to represent in double-word (64 bit) floating point format.

quote (char) inserted before newline

The specified quote character was found before a newline character. This may indicate a spurious character or a missing closing quote.

re-using \texttt{struct/union tag} \texttt{identifier as union/struct tag}

The named identifier has been used to identify two different types of object.
size of struct identifier needed but not yet defined
size of union identifier needed but not yet defined

The size of the structure/union has not yet been defined. This error can occur when an undefined structure/union is used as an argument to the sizeof function and when an undefined structure/union is used in the declaration of a variable. In the second case the error occurs because the compiler attempts to determine the size of the structure/union for memory allocation purposes.

storage class store class incompatible with store class – ignored

Two incompatible storage classes have been used in a declaration. For example, extern static foo; generates this error because extern and static are incompatible types.

storage class store class not permitted in context context – ignored

The specified storage class is not permitted in the context in which it has been used. This error would be generated, for example, if storage class auto were to be used at the top level.

struct identifier has no identifier field

The structure contains no field of that name.

struct identifier must be defined for (static) variable declaration

An undefined structure has been used in a variable declaration.

struct identifier not yet defined –cannot be selected from

A reference was made to an undefined structure.

Too few operands for instruction

A load or store __asm pseudo-instruction has too few arguments.

Too few arguments to macro identifier( . . . on line number

There are too few arguments to the macro which will be substituted at line number.

Too many operands for instruction

A load or store __asm pseudo-instruction has too many arguments.

Too many arguments to macro identifier( . . . on line number

There are too many arguments to the macro which will be substituted at line number.

Too many errors

After 100 Serious errors, the compilation aborts.

too many initialisers in {} for aggregate

An aggregate type, for example an array, has been initialized with more values than can be accommodated.
2.4 Messages

type type1 inconsistent with type2

Two incompatible type identifiers are being used in the declaration of a single object. For example, the declaration double int x; would generate this error.

type disagreement for identifier

The specified identifier has already been assigned a different type.

typedef name type used in expression context

A type definition has been used in an expression.

type qualifier type qualifier not allowed to qualify type qualifier type

'const' may not be repeated in the qualifying list of a type, and similarly for 'volatile'.

undefined struct/union identifier1 member/object: identifier2

The structure or union is, at present, undefined.

Uninitialised static [] arrays illegal

Static arrays of unspecified size must be initialized.

union identifier has no identifier field

The union contains no field of that name.

union identifier must be defined for (static) variable declaration

An undefined union has been used in a variable declaration.

union identifier not yet defined – cannot be selected from

A reference was made to an undefined union.

Unknown directive: #identifier

identifier is not a valid preprocessor directive. Check spelling and/or syntax.

unknown instruction instruction

instruction is not a defined transputer instruction.

zero width named bit field – 1 assumed

Named bit fields must be at least one bit wide.
Appendices
A Local optimization examples

This appendix briefly describes each of the local optimizations available.

A.1 Peephole optimization

This optimization is performed by the compiler at assembly code level. The compiler scans the assembly code for sequences of instructions which may be reduced to a single instruction.

Example:

If a source code instruction generated the following assembly code instructions:

\[ \text{ldc } x \]
\[ \text{ldc } y \]
\[ \text{and} \]

then they could be reduced to the following single instruction:

\[ \text{ldc } x \text{ & } y \]

where: the expression \( x \text{ & } y \) is evaluated by the compiler (since \( x \) and \( y \) are constants).

In a similar manner, the sequence:

\[ \text{ldl } n \]
\[ \text{stl } n \]

could be removed altogether.

A.1.1 Summary of effects:

- Slight improvement to execution time: some instructions are no longer executed.
- Slight improvement to code size: some instructions are no longer coded.

A.2 Flowgraph optimizations

Flowgraph optimizations cover a wide range of local optimizations which are performed on short sequences of code.
The following examples describe typical optimizations of this type.

A.2.1 Branch-chaining optimization

When the destination of one jump is to another jump, then the first jump is replaced with a jump to the destination of the second jump.

This optimization cannot be performed at source code level and is best demonstrated in assembly code:

```
  j  L1
  L1: j  L2
```

becomes:

```
  j  L2
  L1: j  L2
```

A.2.2 Dead code elimination

Dead code elimination is the removal of statements which cannot be reached and is another type of flowgraph optimization. For example:

```c
void p(void)
{
  while (1)
    ...loop body – contains no break statements
    s;  /* This statement cannot be reached*/
}
```

With dead code elimination, this code segment would be transformed to:

```c
void p(void)
{
  while (1)
    ...loop body – contains no break statements
}
```

The statement 's' is deleted.

A.2.3 Summary of effects of flowgraph optimizations:

The effect on both execution time and code size varies on the particular optimization performed. For the examples shown above the results are:

- Branch-chaining – Improved execution time and a slight reduction in code size.
• Dead code elimination – code size is improved.

A.3 Redundant store elimination

Assignments to variables which are not subsequently used, are deleted by an optimization called redundant store elimination.

Example:

```c
void p(void)
{
    int x;
    ... some code
    x = 27;
    ... some more code which does not read x
}
```

With redundant store elimination, this segment of code would be transformed to:

```c
void p(void)
{
    int x;
    ... some code
    ... some more code which does not read x
}
```

The assignment of a value to x is removed.

A.3.1 Summary of effects:

• Slight improvement to execution time.
• Slight improvement in code size.
B Global optimization examples

This appendix briefly describes each of the global optimizations available.

B.1 Common subexpression elimination

The purpose of common subexpression elimination is to remove from the program any redundant computations. An expression is redundant where it is identical to and computes the same value as another expression whose value is still available for use.

Such commonality is not restricted to explicit computations in the source code but may include implicit computations such as array element address calculation. Subscripted expressions often repeat in blocks of code. Where this happens it is often more efficient to extract expressions which occur more than once so that they are evaluated once only.

Example:

Code segment before common subexpression elimination is applied:

```c
{ int a[10][10], i, j;
  for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
      a[i][j] = a[i][j] + t;
}
```

Code segment after common subexpression elimination is applied:

```c
{ int a[10][10], i, j;
  for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
      { int *temp3 = &a[i][j];
        *temp3 = *temp3 + t;
      }
}
```

Notice that the subscripted variable in the summation has been replaced by a single variable *temp3.

Common subexpression elimination is achieved by saving the result of a computation in a temporary location rather than recomputing the expression.
B.1.1 Summary of effects:

- Improvement to execution time: expressions which were evaluated several times are now only evaluated once.
- Improvement to code size: expressions which were coded several times are now only coded once.
- Increase in workspace size: expressions which were evaluated several times now have their value stored in a temporary variable in workspace.

The compiler evaluates each potential case and only applies the optimization if it is worthwhile.

B.2 Loop-invariant code optimization

This optimization removes expressions which remain constant during the execution of a loop, to outside the loop so that they are executed once only. Invariant expressions often include subscripting calculations as well as computations in the source code.

Example:

Code segment before loop-invariant code optimization is applied:

```c
int a[10][10], i, j;
for (i = 0; i < 10; i++)
    for (j = 0; j < 10; j++)
        a[i][j] = a[i][j] + t;
```

Code segment after loop-invariant code optimization is applied:

```c
int a[10][10], i, j;
for (i = 0; i < 10; i++)
    { int *temp1 = &a[i];
      int *temp2 = &a[i];
      for (j = 0 ; j < 10; j++)
        temp1[j] = temp2[j] + t
    }
```

In this example the value of \( i \) remains constant during iterations of the inner loop which increments \( j \). The calculation of \( &a[i] \) can therefore be moved outside the inner loop.
B.2.1 Summary of effects:

- Improvement to execution time: expressions which were evaluated on every loop iteration are now only evaluated once.
- Slight increase in code size: extra code has to be inserted to store the result of an expression in a temporary.
- Increase in workspace size: expressions which were evaluated on every loop iteration are now evaluated into a temporary variable outside of the loop.

B.3 Global optimization example

This example is based on the source code used in the previous two sections and shows what happens when both global subexpression elimination and loop-invariant code optimization are applied:

```c
int a[10][10], i, j;
for (i = 0; i < 10; i++)
{
    int *temp1 = &a[i];
    for (j = 0; j < 10; j++)
    {
        int *temp3 = temp1[j];
        *temp3 = *temp3 + t;
    }
}
```

B.4 Tail-call and tail recursion optimization

The purpose of these optimizations is to make function calls more efficient. When the last operation performed by a function is to call another function, tail-call optimization may be applied. In C programs a function may in fact, call itself, in which case the optimization is called tail recursion optimization.

The optimization is achieved by substituting a jump instruction instead of the call instruction. This optimization cannot be performed at source code level.

When a jump instruction is used, the return from the other function will return the caller to the caller of the current function, thereby saving one return sequence. The called function's workspace is also laid on top of the current function's workspace, thus saving stack size.
B.4.1 Example: (Tail–call optimization)

Take the following code segment:

void p(int x)
{
  ...body of p
  q(x+1);
}

Without optimization the code generated for routine 'p' would be:

```
p:
  ajw -3
  ...body of p
  ldl 2  —x
  adc 1
  ldl 1  —<static_link>
  call $q
  ajw 3
  ret
```

After tail–call optimization, the code generated is:

```
p:
  ajw -3
  ...body of p
  ldl 2  —x
  adc 1
  stl 2  —x
  ajw 3
  j  $q
```

Note: that the workspace for routine 'q' is overlaid on the workspace for routine 'p'.

B.4.2 Summary of effects: (Tail–call optimization)

- Little effect on execution time.
- Workspace requirements are reduced as the called function's workspace is overlaid on the calling function's workspace.
B.4.3 Example: (Tail-recursion optimization)

```c
void p(int x)
{
    ...body of p
    p(x+1);
}
```

This code segment when compiled without optimization would cause the following code to be generated for routine p.

```
P:
   ajw   -3
   ... body of p
   ldl   2   —x
   adc   1
   ldl   1   —<static_link>
   call  $p
   ajw   3
   ret
```

After tail-recursion optimization, the code generated is:

```
P:
   ajw   -3
   ..3:
   ... body of p
   ldl   2   —x
   adc   1
   stl   2   —x
   j     ..3
```

**Note:** that the workspace for the second invocation of 'p' is laid on top of the workspace for the first invocation of 'p'. Also note that the second invocation of 'p' does not re-execute the routine entry code (in this example, an 'ajw -3' instruction).

B.4.4 Summary of effects: (Tail-recursion optimization)

- Execution time is improved as the called function's entry sequence is already evaluated. In addition, it may not be necessary to assign the actual parameters to the formal parameters of the function called.
- Workspace requirements are reduced as the called routine's workspace is overlaid on the calling routine's workspace.

B.5 Workspace allocation by coloring

This method of workspace allocation can be performed when the lifetimes of two variables, 'a' and 'b' do not overlap. When this is the case 'a' and 'b' may be allocated in the same workspace slot.
For example, in the following segment of code, variables 'a' and 'b' can be placed in the same workspace slot because their values are never required at the same time:

```c
{  
    int a, b;  
    a = func1(27);  
    proc1(a);  
    proc1(a);  
    /* 'a' is not used after this point, 'b' is not used before this point */  
    b = func2(34);  
    proc2(b);  
}
```

When optimization is enabled, the compiler will use this method of workspace allocation, provided the code is suitable.

If workspace is allocated by coloring, then the compiler calculates a usage count for each variable, and places the most frequently used variables at lower workspace positions.

When the command line option '00' is used i.e. when optimization is disabled, all variables are allocated their own unique workspace slot.
Index

Symbols
#pragma, 8
IMS_nosideeffects, 8
__asm, use when optimizing, 9

A
ANSI C
compiler, optimizing, 3
language, use when optimizing, 7
trigraphs, 12

B
Branch–chaining optimization, 42

C
Command line options, optimizing compiler, 5
Common subexpression elimination, 45
Compiler
diagnostics, terminology, 10
optimizing, 3
command line options, 5
global optimizations, 45
information messages, 7
language considerations, 7
local optimizations, 41
messages, 10
running, 5
const, 8

D
Dead code elimination, 42

E
Error messages, optimizing compiler, 10

F
Flowgraph optimization, 41

G
Global compiler optimizations, 45

H
Host, versions, iii

I
i cc, optimizing compiler, 3
command line options, 5
global optimizations, 45
information messages, 7
language considerations, 7
local optimizations, 41
messages, 10
running, 5

L
Local compiler optimizations, 41
Loop–invariant code, optimization, 46

M
Messages. See Error messages

O
Object code, optimizing, 3

72 TDS 348 01

October 1992
Optimizing object code
for space, 6, 45
for time, 6, 45
global optimizations, 45
language considerations, 7
local optimizations, 41
using icc, 3

P
Peephole optimization, 41
Performance improvement, using optimizing compiler, 3
Pragmas, optimizing compiler, 8

R
Redundant store elimination, 43
register, 8

S
Space, optimizing compilation, 6

T
Tail recursion optimization, 47
Tail-call optimization, 47
Time, optimizing compilation, 6
Toolset, documentation, iii
conventions, v
Trigraphs, 12

V
volatile, 8

W
Warnings. See Error messages
Workspace, allocation, optimizing, 49