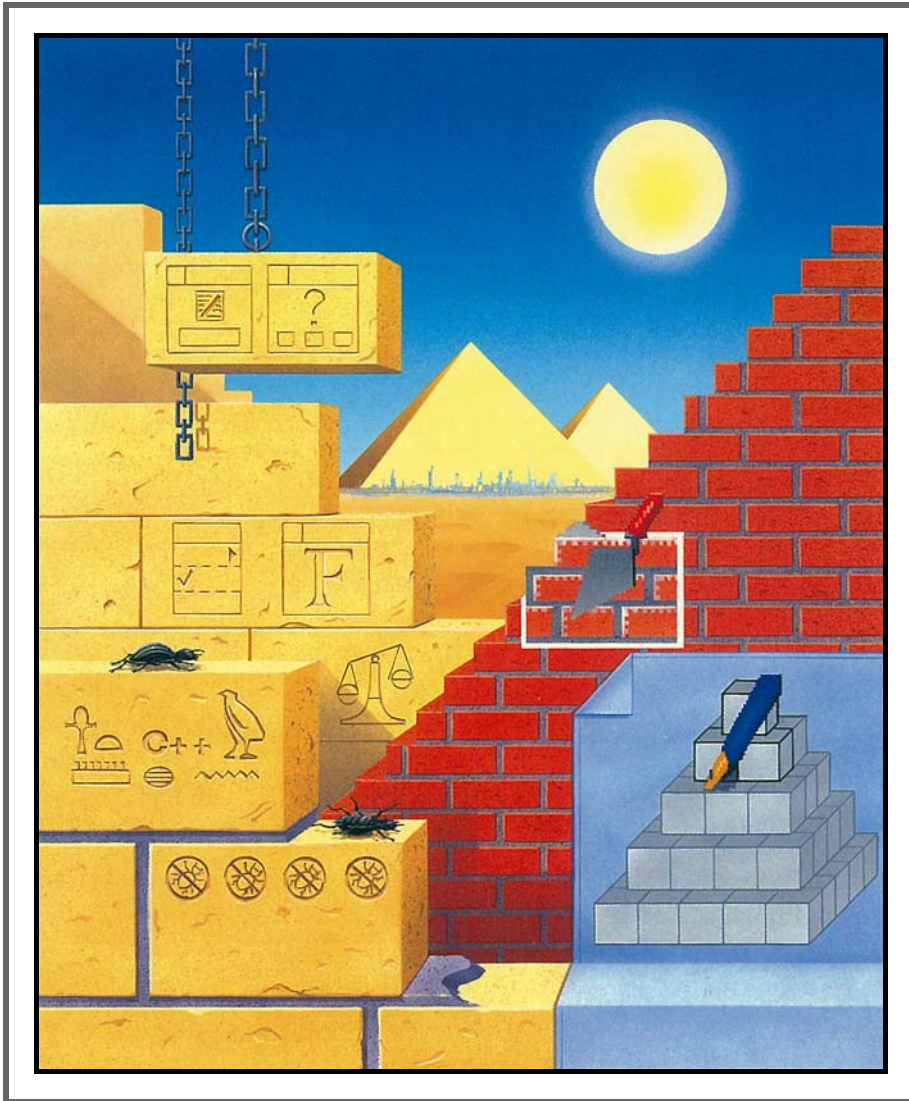


Acorn C/C++



Copyright © 1994 Acorn Computers Limited. All rights reserved.

Some updates and changes copyright © 2002 Castle Technology Ltd. All rights reserved.

Some updates and changes copyright © 2014-2024 RISC OS Open Ltd. All rights reserved.

Issue 1 published by Acorn Computers Technical Publications Department.

Issue 2 published by Castle Technology Ltd.

Issues 3-7 published by RISC OS Open Ltd.

No part of this publication may be reproduced or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, or stored in any retrieval system of any nature, without the written permission of the copyright holder and the publisher, application for which shall be made to the publisher.

The product described in this manual is not intended for use as a critical component in life support devices or any system in which failure could be expected to result in personal injury.

The product described in this manual is subject to continuous development and improvement. All information of a technical nature and particulars of the product and its use (including the information and particulars in this manual) are given by the publisher in good faith. However, the publisher cannot accept any liability for any loss or damage arising from the use of any information or particulars in this manual.

If you have any comments on this manual, please complete the form at the back of the manual and send it to the address given there.

All trademarks are acknowledged as belonging to their respective owners.

Published by RISC OS Open Ltd.

Issue 1, December 1994 (Acorn part number 0484,232).

Issue 2, October 2002 (updates by Castle Technology Ltd).

Issue 3, October 2014 (updates by RISC OS Open Ltd).

Issue 4, April 2015 (updates by RISC OS Open Ltd).

Issue 5, October 2020 (minor corrections).

Issue 6, July 2021 (updates by RISC OS Open Ltd).

Issue 7, September 2024 (updates by RISC OS Open Ltd).

Contents

Introduction 1

- Installation of Acorn C/C++ 1
- The C compiler 2
- The C++ translator 2
- This user guide 2
- Useful references 6

Part 1 – Using the tools 9

CC and C++ 11

- The underlying programs 11
- Getting started with CC and C++ 12
- Libraries 14
- File naming and placing conventions 14
- Include file searching 18
- The SetUp dialogue box 22
- The SetUp menu 24
- Output messages 41
- The icon bar menu 42
- Command lines 43
- Worked examples 52

CMHG 57

- Starting CMHG 58
- The icon bar menu 59
- Example output 59
- Command line interface 60

ToANSI 63

- ToANSI C translation 64
- Starting ToANSI 65
- The icon bar menu 66
- Example output 66
- Command line interface 67

ToPCC 69

- ToPCC C translation 70
- Starting ToPCC 71
- The icon bar menu 72
- Example output 73
- Command line interface 74

Part 2 – C language issues 75

C implementation details 77

Implementation details 78

Identifiers 78

Data elements 78

Structured data types 81

Pointers 84

Arithmetic operations 84

Expression evaluation 85

Implementation limits 85

Standard implementation definition 87

Translation (J.3.1) 87

Environment (J.3.2) 87

Identifiers (J.3.3) 88

Characters (J.3.4) 88

Integers (J.3.5) 90

Floating point (J.3.6) 90

Arrays and pointers (J.3.7) 90

Hints (J.3.8) 91

Structures, unions, enumerations and bitfields (J.3.9) 91

Qualifiers (J.3.10) 92

Preprocessing directives (J.3.11) 92

Library functions (J.3.12) 92

Architecture (J.3.13) 96

C99 features 97

C99 features implemented in the compiler 97

C99 restrictions implemented in the compiler 110

Other C99 features 110

C99 non-compliances 110

Implementation notes 111

Compatibility issues 111

C18 features 112

C18 features implemented in the compiler 112

C18 non-compliances 118

Compatibility issues 118

Extra features 119

#warning directive 119

#pragma directives 119

Special function declaration keywords 125

Special variable declaration keywords 125

Pre-defined macros 126

Intrinsic functions 128

Large file support 129

Inline assembler 130

The C library 135

Compatibility Issues 135

assert.h 136

complex.h 137

ctype.h 138

errno.h 139

fenv.h 141

float.h 142

inttypes.h 143

iso646.h 144

limits.h 145

locale.h 146

math.h 148

setjmp.h 151

signal.h 152

stdalign.h 154

stdarg.h 155

stdatomic.h 157

stdbool.h 158

stddef.h 159

stdint.h 160

stdio.h 162

stdlib.h 181

stdnoreturn.h 193

string.h 194

tgmath.h 201

threads.h 202

time.h 203

uchar.h 208

wchar.h 209

wctype.h 210

The ANSI library 211

Extra functions 212

Part 3 – C++ language issues 213

C++ implementation details 215

- Translation Limits 215
- Identifiers (2.3) 216
- Character Constants (2.5.2) 216
- Floating Constants (2.5.3) 216
- String Literals (2.5.4) 217
- Start and Termination (3.4) 217
- Fundamental Types (3.6.1) 217
- Integral Conversions (4.2) 218
- Expressions (5) 218
- Function Call (5.2.2) 218
- Explicit Type Conversion (5.4) 219
- Multiplicative Operators (5.6) 219
- Shift Operators (5.8) 219
- Relational Operators (5.9) 219
- Storage Class Specifiers (7.1.1) 220
- Type Specifiers (7.1.6) 220
- Asm Declarations (7.3) 220
- Linkage Specifications (7.4) 220
- Class Members (9.2) 221
- Bitfields (9.6) 221
- Multiple Base Classes (10.1) 221
- Argument Matching (13.2) 222
- Exception Handling (experimental) (15) 222
- Predefined Names (16.10) 222

The Streams library 223

- Introduction 224
- filebuf 229
- fstream 233
- ios 237
- istream 248
- manipulators 255
- ostream 259
- stdiobuf 265
- streambuf – protected 266
- streambuf – public 274
- strstream 279
- strstreambuf 282

The Complex Math library 285

- Introduction 286
- cartesian/polar 287
- complex_error 289
- exp, log, pow, sqrt 292
- complex operators 294
- cplxtrig 297

Part 4 – Developing software for RISC OS 299**Portability 301**

- General portability considerations 301
- ISO C vs K&R C 304
- The ToPCC and ToANSI tools 308
- pcc compatibility mode 308
- Environmental aspects 312
- Software Interrupts (SWIs) 314

Assembly language interface 317

- Register names 318
- Register usage 318
- Control arrival 319
- Passing arguments 319
- Return link 320
- Structure results 320
- Storage of variables 321
- Function workspace 321
- Examples 321

RISC OS Compatibility 323

- 32-bit compatibility 323
- The shared C library 324
- Ensuring the necessary components are present 324

How to write relocatable modules in C 327

- Getting started 327
- Constraints on modules written in C 328
- Overview of modules written in C 328
- Functional components of modules written in C 329

Overlays 349

- Paging vs overlays 349
- When to use overlays 350

Part 5 – Appendices 353

Changes to the C compiler 355

C errors and warnings 363

- Interpreting CC errors and warnings 363
- Warnings 364
- Non-serious errors 372
- Serious errors 382
- Fatal errors 397
- System errors 398

C++ errors and warnings 399

- ‘Not implemented’ messages 399

C function index 415

C++ class index 419

Index 423

Introduction

Acorn C/C++ is a development environment for producing RISC OS desktop applications and relocatable modules written in C and/or in C++. It consists of a number of programming tools which are RISC OS desktop applications. These tools interact in ways designed to help your productivity, forming an extendable environment integrated by the RISC OS desktop. Acorn C/C++ may be used with Acorn Assembler (a part of this product) to provide an environment for mixed language development.

This product includes tools to:

- edit program source and other text files
- search and examine text files
- convert C source and header text between ANSI and UNIX dialects
- examine some binary files
- compile and link C programs
- compile and link C++ programs
- construct relocatable modules entirely from C or C++
- compile and construct programs under the control of makefiles, these being set up from a simple desktop interface
- squeeze finished program images to occupy less disc space
- construct linkable libraries
- debug RISC OS desktop applications interactively
- design RISC OS desktop interfaces and test their functionality
- use the Toolbox to interact with those interfaces.

Most of the tools in this product are also of general use for constructing applications in other programming languages, such as Arm Assembler. These non-language-specific tools are described in the accompanying *Desktop Tools* guide.

Installation of Acorn C/C++

Installation of Acorn C/C++ is described in the chapter *Installing Acorn C/C++* on page 7 of the accompanying *Desktop Tools* guide.

The C compiler

The Acorn C compiler for RISC OS (the tool CC supplied as a part of this product) is a strict implementation of the ISO 9899 family of standards, specifically:

- A full implementation of C as defined by the ISO standard ISO/IEC 9899:1990 (this ISO standard is technically identical to the earlier 1989 ANSI C standard).
- Many of the features and library functions from ISO/IEC 9899:1999.
- Many of the features and library functions from ISO/IEC 9899:2018 (this ISO standard is a clarification of the earlier 2011 edition which it replaces).

These versions are commonly referred to as C90, C99 (see *C99 features* on page 97), and C18 (see *C18 features* on page 112) respectively.

To obtain the standard documents, see the section *Useful references* on page 6.

The C++ translator

The C++ translator for RISC OS (the tool C++ supplied as a part of this product) is a port of Release 3.0 of AT&T's CFront product. This is a relatively old implementation of C++ and does not conform to the ISO/IEC 14882:1998 standard.

This user guide

This guide is a reference manual for the C tools CC, C++, CMHG, ToANSI and ToPCC working as part of the development environment of Acorn C/C++. These are the only tools in this product which are not used for programming in other languages, and already described in the accompanying *Desktop Tools* guide. This manual also documents the C and C++ library support provided and other aspects that are particular to this C product:

- special features of this implementation of the C and C++ languages
- operating the Acorn C/C++ tools specific to the C and C++ languages
- developing programs for the RISC OS environment:
 - Portability issues, including the portable C compiler (pcc) facility
 - Desktop applications
 - Relocatable modules
 - Overlays
 - Calling other programs and languages from C.

This guide is not intended as an introduction to C or C++, and does not teach C or C++ programming; nor is it a reference manual for the C standards. Both these needs are addressed by publications listed in the section *Useful references* on page 6.

This guide is organised into parts:

Part 1 – Using the tools

Part 2 – C language issues

Part 3 – C++ language issues

Part 4 – Developing software for RISC OS

Part 5 – Appendices

Part 1 – Using the tools

This part of the guide describes the operation of the programming tools specific to C. The first chapter describes the interaction of the C tools with the rest of the development environment; each of the remaining chapters is devoted to an individual tool. Examples in the text and on disc are used to illustrate several points.

The chapters are:

- *CC and C++*
- *CMHG*
- *ToANSI*
- *ToPCC*

Part 2 – C language issues

This covers issues to do with the C programming language itself, in particular those parts of the ISO standard that are necessarily machine- or operating system-specific.

The chapters are:

- *C implementation details*
How Acorn C implements those aspects of the language which ISO leaves to the discretion of the implementor; and how Acorn C behaves in those areas covered by Annex J of the standard (which lists those aspects which the standard requires each implementation to define).

- *The C library*

This chapter works through the headers of the C library, (`assert.h` to `wctype.h`), outlining the contents of each one:

- function prototypes
- macro, type and structure definitions
- constant declarations.

- *The ANSI library*

This chapter details the ANSI library, a superset of the C library that provides additional features useful in debugging and profiling your software.

Part 3 – C++ language issues

This covers issues to do with the C++ programming language, such as details of its implementation and of the libraries supplied with it.

- *C++ implementation details*

This chapter describes implementation specific behaviour of Acorn C++.

- *The Streams library*

This chapter describes the C++ Streams library, giving a synopsis (including prototypes) and a description of each available interface.

- *The Complex Math library*

This chapter describes the C++ Complex Math library, giving a synopsis (including prototypes) and a description of each available interface.

Part 4 – Developing software for RISC OS

This part of the Guide tells you how to write software in C for the RISC OS environment. Examples in the text and on disc are used to illustrate each type of program development. It also includes a chapter on portability to help with porting applications in C to and from RISC OS.

The chapters are:

- *Portability*

The chapter covers:

- portability considerations in general
- the major differences between ANSI and 'K&R' C
- using the pcc compatibility mode of the Acorn compiler
- standard headers and libraries
- environmental aspects of portability.

-
- *Assembly language interface*
How to handle procedure entry and exit in assembly language, so that you can write programs which interface correctly with the code produced by the C compiler.
 - *RISC OS Compatibility*
Issues in developing software that is compatible with earlier versions of RISC OS running on older Arm processors.
 - *How to write relocatable modules in C*
Relocatable modules – the building blocks of the RISC OS operating system – are needed for device drivers and similar low-level software.
 - *Overlays*
This chapter explains how to write an application using overlays, with a worked example as an illustration.

Part 5 – Appendices

The appendixes are:

- *Changes to the C compiler*
This appendix highlights all the features that have changed since Acorn's final release of the Desktop Development Environment in 1994.
- *C errors and warnings*
Messages produced by the compiler, of varying degrees of severity.
- *C++ errors and warnings*
Messages produced by the translator, of varying degrees of severity.

Conventions used

Throughout this Guide, a fixed-width font is used for text that the user should type, with an italic version representing classes of item that would be replaced in the command by actual objects of the appropriate type. For example:

`cc options filenames`

This means that you type `cc` exactly as shown, and replace *options* and *filenames* by the required parameters.

Where it is necessary to differentiate between text you type, and that output by the computer, your input is shown in **bold**, and the computer's response in a normal weight.

Useful references

C programming

- Harbison, S P and Steele, G L, (1994) *A C Reference Manual*, (fourth edition). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0133262243.
This is a very thorough reference guide to C, including a useful amount of information on the ANSI C standard.
- Kernighan, B W and Ritchie, D M, (1988) *The C Programming Language* (second edition). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362- 8.
This is the original C 'bible', updated to cover the essentials of ANSI C too.
- Koenig, A, (1989) *C Traps and Pitfalls*, Addison-Wesley, Reading, Mass, USA. ISBN 0201179288.
This book explains how to avoid the most common traps and pitfalls that ensnare even the most experienced C programmers. It provides informative reading at all levels.

C++ programming

- Stroustrup, B, (1991) *The C++ Programming Language*, (second edition). Addison-Wesley, Reading, Mass, USA. ISBN 0-201-53992-6.
The standard book describing the C++ language, including a complete copy of the Reference Manual.
- Ellis, A and Stroustrup, B, (1990) *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Mass, USA. ISBN 0-201-51459-1.
The original Reference Manual, used as an ANSI base document, with additional annotations and commentary sections.

RISC OS

- The RISC OS *User Guide* supplied with your computer, which describes how to use the RISC OS operating system and the applications Edit, Paint and Draw.
- The RISC OS *Programmer's Reference Manual*, which contains detailed information describing the features of RISC OS available to software developers.
- The RISC OS *Style Guide*, which contains useful guidelines for software authors.

The C standards

The C standard as defined in *American National Standard for Information Systems – Programming Language C*, ANSI X3.159-1989 was adopted by the BSI as BS EN 29899:1993 and by the International Standards Organisation (ISO) as ISO/IEC 9899:1990. This standard is often referred to as 'C90' and although the section numbers were changed it is technically identical to 'C89'.

These standards were superceded by the ISO/IEC 9899:1999 standard, commonly known as 'C99' and defined in *Programming Languages – C*, ISO/IEC 9899:1999. This standard has been adopted by the BSI as BS ISO/IEC 9899:1999 which is identical.

These standards were themselves superceded by the ISO/IEC 9899:2018 standard, commonly known as 'C18' and defined in *Programming Languages – C*, ISO/IEC 9899:2018. This standard has been adopted by the BSI as BS ISO/IEC 9899:2018 which is identical.

The latest edition of the standard is available to purchase from the standards organisations, furthermore the older C99 standard has been published in book form by Wiley & Sons (*The C Standard: Incorporating Technical Corrigendum 1*, ISBN 978-0-470-84573-8). You should find the coverage of ANSI C and ISO C99 in this and the books listed above adequate for all but the most demanding requirements.

The ANSI C++ standard

The Acorn C++ front end was developed with reference to *The Annotated C++ Reference Manual* referred to above but has not been formally tested against the standard.

Part 1 – Using the tools





CC is a desktop tool which provides an easy interface to the CC and Link programs that Acorn C/C++ installs in your computer's library. It constructs command lines and passes them to these programs. Likewise, **C++** is a desktop tool that constructs command lines for the CC, CFront and Link programs in the library.

Because these two desktop tools are so similar, and share the underlying CC and Link programs, we describe them in the same chapter. Most of the rest of this chapter covers the CC and C++ options, and gives some programming examples.

If you are new to RISC OS and the Acorn C/C++ product, read the whole of this chapter before starting to use Acorn C/C++. If you are an experienced C or C++ programmer, you will find this chapter essential for reference, and may choose to tackle the section *Worked examples* on page 52 first. If you have used earlier versions of the Acorn C/C++ compiler you should refer to section *Changes to the C compiler* on page 355.

The underlying programs

The CC compiler is a full implementation of ISO C90 with many ISO C99 and C18 features as described in the chapter *C99 features* on page 97, and *C18 features* on page 112. It consists of a preprocessor and a code generator; it processes text files containing the source and headers of programs into linkable object files.

The Link program combines these object files to produce executable image files.

CFront is a C++ translator; it is a port of Release 3.0 of AT&T's CFront product. It converts C++ source code to C source code.

The characteristics of CC as a language implementation are defined in *Part 2 – C language issues* on page 75. Similar information for C++ is in *Part 3 – C++ language issues* on page 213.

How the tools use them

The command line that the CC tool produces first calls CC to preprocess and compile the source into object files; it then calls Link to link those object files.

The command line that the C++ tool produces first calls the CC preprocessor in a special C++ compatible mode; it then calls CFront to convert the resultant source files to C; it then calls CC to compile the C source into object files, again using a special C++ compatible mode; it finally calls Link to link those object files.

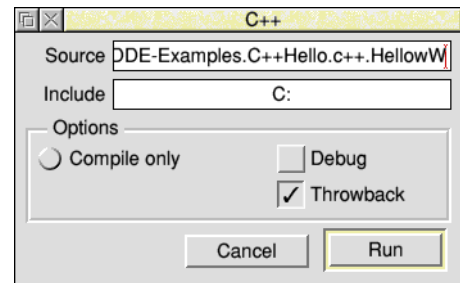
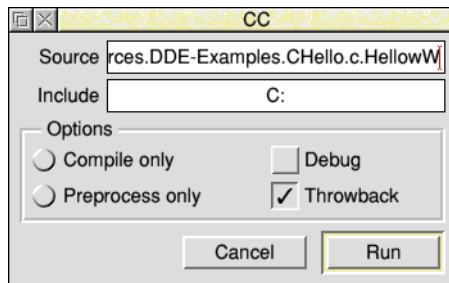
Getting started with CC and C++

To use the CC or C++ tool, first open the Apps . DDE directory display, then double click on !CC or !C++ as required. (You cannot start CC or C++ by double clicking on a file – the tools own no file type unlike, for example, Draw.)

The tool's icon appears on the icon bar:



Clicking Select on this icon, or dragging a source file from a directory display to this icon, brings up the **SetUp** dialogue box. To see this work, open the directory display for Sources . DDE-Examples . C/C++, and then drag either CHello . c . HelloW to the CC icon, or C++Hello . c++ . HelloW to the C++ icon. The **SetUp** dialogue box appears:



As you have dragged a source file to bring up this dialogue box, its name appears in the writable **Source** icon; otherwise this icon would have appeared containing the name of the last filename entered there, or be empty if there were none.

Clicking Menu on the SetUp dialogue box brings up the **SetUp** menu:

CC		C++	
Command line ▶		Command line ▶	
Default path ▶		Default path ▶	
Keep comments		Define ▶	
Define ▶		Undefine ▶	
Undefine ▶		Assembler	
Debug options ▶		Module code	
Profile		✓ Libraries ▶	
Assembler		Features ▶	
Module code		No warnings	
✓ Libraries ▶		✓ Work directory ▶	
Features ▶		Others ▶	
Suppress warnings ▶			
Suppress errors ▶			
UNIX pcc			
ISO C90			
ISO C99			
Errors to file ▶			
Listing			
✓ Work directory ▶			
Others ▶			

The SetUp dialogue box and menu specify the options for the next compilation to be done. You start the next job by clicking Select on the **Run** button on the dialogue box (or on the **Command line** menu dialogue box). Clicking Select on the **Cancel** button removes the **SetUp** dialogue box and clears any changes you have just made to the options settings, leaving them back in the state they were in before you brought up the **SetUp** box. The options last until you adjust them again or reload the tool; or you can save the options for future use with an item from the main icon menu.

Ensure that the option settings are the defaults, as in the above pictures. Click on the **Run** button to compile the HelloW example with an integral link step. Save the executable image file produced in the directory above that holding the source, naming it HelloW, then double click Select on the file's icon to run it. The program runs, putting a Hello world message in the standard RISC OS command line window:

```

Run HostFS:$Sources.DDE-Examples.C/C++.CHello.HelloW
Hello World
Press SPACE or click mouse to continue

```

Libraries

C libraries

There are several libraries provided to support the C compiler:

- The stubs for the shared C library
This provides all the standard facilities of the language, as defined by the ISO standard document. Code using calls to the shared C library will be portable to other environments if a standards-compliant compiler and library are available for that environment. See the chapter *The C library* on page 135.
- The ANSI library
The ANSI library is a stand-alone version of the shared C library that contains a few extra functions useful in debugging and profiling your code. You should use it for development only, using the shared C library in any final product. See the chapter *The ANSI library* on page 211.

C++ libraries

The C++ compiler produces output which uses the ANSI C library (by linking with the stubs). A C++ program also needs to link with the C++ library which is held in `C:c++lib.o.c++lib`. This has support functions such as `new` and `delete`, and includes the streams and complex maths libraries.

File naming and placing conventions

This section explains the concept of a work directory, and describes the naming conventions used to identify the different classes of file you will come across when using Acorn C/C++.

Work directory

Both CC and C++ operate in a *work directory*. The work directory is where the tools place all output files that you don't explicitly place yourself by dragging from a **Save As** dialogue box. This includes object files to be linked by an integral link step, assembly language output and listing output. The work directory is also a place where some input source and header files are looked for – see the next sections for more details.

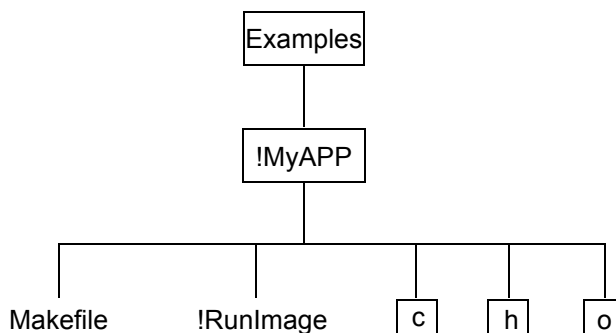
If you're using Make, the work directory is simply the directory containing the *makefile* controlling the job.

If you're using the CC or C++ tools, the work directory is formed from the directory containing the source file, modified by the relative path name specified by the **Work directory** option on the SetUp menu. The default **Work directory** SetUp menu value is ^.

For example, when compiling the example 'Hello world' C program with the default **Work directory** setting:

- The source is in the directory
`Sources.DDE-Examples.C/C++.CHello.c`
- The work directory is therefore
`Sources.DDE-Examples.C/C++.CHello.c.^`, i.e.
`Sources.DDE-Examples.C/C++.CHello.`

A typical directory arrangement is:



The resource files (such as !Run and Res) normally found in an application directory are not shown above. With directories arranged as above and default option settings, the work directories for both the Make and the C/C++ tools are the same, namely `Examples.!MyApp`.

Filename conventions

The Acorn C/C++ system, in common with others, uses naming conventions to identify the classes of file involved in the compilation and linking process. Many systems use conventional suffixes for this. For example, the suffix `.c` denotes C source files on UNIX and Windows systems. This convention clashes with RISC OS's use of the full-stop character in pathnames. It is more natural under RISC OS filing systems to use a prefix convention, e.g. `c.foo`, where `c` is the directory containing C source files, and `foo` is the filename.

However, portability is an increasingly important issue. CC recognises the standard file naming conventions and performs the appropriate transformations to construct valid RISC OS pathnames. The following sections summarise the conventions for referring to source, include, object and program files.

Rooted filenames

A filename is rooted if it is

- a RISC OS filename beginning with a '\$' or an '&' – for example:
`$.library.h.baricon &.h.myheader`
- a UNIX filename beginning with a '/' – for example:
`/library/baricon.h`
- a Windows filename beginning with a '\' – for example:
`\library\baricon.h`

Rooted filenames are used by CC as absolute specifications of filenames, independent of work directories, search paths, etc. Rooted UNIX or Windows filenames are converted into the RISC OS syntax and prefix forms.

Source files

The CC and C++ tools specify the source files to be compiled on the command line they construct and pass to the underlying programs. Dragging a source file to the CC SetUp dialogue box specifies the file as an absolute rooted filename.

Make uses a makefile to specify the source files; their pathnames are normally given relative to the work directory. C source files will be looked for in the subdirectory `c` of the work directory. To aid portability, a file specified as `foo.c` in a makefile will be looked for in `@.c.foo`, where `@` means the work directory. C++ source files are similarly looked for in the subdirectory `c++`.

Include files

The way in which the compiler searches for included files is dealt with in detail in the section *Include file searching* on page 18. Here we describe the issues of naming header files and how to name them in `#include` lines in your C and C++ program source.

Include files are often headers for libraries, and are incorporated by issuing the `#include` directive – dealt with by the preprocessor – at the start of a source file. For instance, in the C `HelloW` example:

```
#include <stdio.h>
```

By convention, header files are placed in subdirectory `h`. This convention is followed here. You can use subdirectory `h` of the work directory for your own header files, which can be incorporated with a source line like:

```
#include "myfile.h"
```

Note that both the example filenames `stdio.h` and `myfile.h` are in suffix form rather than RISC OS prefix form. This is because you can make use of Acorn C/C++'s filename processing to interpret these, leaving program lines which do not need altering to port them to machines expecting suffixes.

To facilitate the porting of code from UNIX and Windows to RISC OS, UNIX-style and Windows-style filenames are translated to equivalent RISC OS-style filenames.

For example:

<code>../include/defs.h</code>	is translated to	<code>^.include.h.defs</code>
<code>..\cls\hash.h</code>	is translated to	<code>^.cls.h.hash</code>
<code>includes.h</code>	is translated to	<code>h.includes</code>

but:

<code>system.defs</code>	is translated to	<code>system.defs</code>
--------------------------	------------------	--------------------------

In the same way, the lists of directory names given as arguments to the compiler's **Include** and **Default path** SetUp options (see below) are translated to RISC OS format before being used, in the rare event that this is necessary.

Object files

If you use the CC or C++ tool to compile a single file with the SetUp dialogue box option **Compile only** enabled, you use a standard **Save As** dialogue box to save the resultant object file. Otherwise the object files created by the compiler are instead stored in the `o` subdirectory of the work directory. Thus the result of compiling `c.sieve` will be found in `o.sieve`.

Program files

If you haven't enabled the **Compile only** option on the CC or C++ tool's SetUp menu, the tool compiles sources to object files, and then links them with the C library stubs to produce an executable program file. You may find it convenient to save this program file in the work directory itself – there is no conventional suffix for these.

Compilation list files

If you enable the **Listing** option on the CC tool's SetUp menu, then for each compiled source file the CC tool creates a compilation listing file in the `l` subdirectory of the work directory. Thus compiling `c.sieve` with **Listing** enabled will by default result in the list file `l.sieve` being created.

The C++ tool does not have a **Listing** option.

Assembly list files

If the CC or C++ tool's SetUp menu option **Assembler** is enabled, no object code is generated. Instead, an assembly listing of the code is created. If only one assembly listing file is produced, you save it from a standard **Save As** dialogue box. If more than one is produced these are placed in the subdirectory `s` of the work directory. Thus compiling `c.sieve` with **Assembler** enabled can result in the assembly language file `s.sieve` being created.

Filename validity

The compiler does not check whether the filenames you give are acceptable – whether they contain only valid characters and are of acceptable length – this is done by the filing system.

Include file searching

The process of converting text C or C++ source to linkable object files of binary code can be seen as a pipeline of several processes. The first stage is preprocessing the source. It is at this stage that the text of header files is brought in at the position of `#include` directives in the source text.

The preprocessor – which is used by both the CC and C++ tools – handles `#include` directives of two forms:

```
#include <filename>
```

or

```
#include "filename"
```

You will normally include four types of header file:

- headers for the standard parts of the C library
- headers for the non-standard parts of the C library
- headers for the other libraries supplied with Acorn C/C++
- headers for your own include files.

A special feature of the Acorn C/C++ system is that the standard headers are built into the compiler, and are used by default. By writing the filename in the angle bracket form, you indicate that the include file is a system file, and thus ensure that the compiler looks first in its built-in filing system. Of the common types of header above, only the headers for the standard parts of the C library should be referred to as system files in angle brackets. Writing the filename in the double quote form indicates that the include file is a user file.

The headers for the non-standard parts of the main C library – `kernel`, `pragmas`, `SWIs` and `varargs` – are not built in to the compiler; nor are the headers for the other libraries supplied with Acorn C/C++. However, by default the CC and C++ tools both set the **Include** icon on their SetUp dialogue box to `C:`. This makes the preprocessor use the value of the `C$Path` system variable to find the headers for the standard libraries supplied in `AcornC/C++.Export.APCS-32.Lib`.

You can include headers for other libraries by adding the parent of the `h` directory holding them to the **Include** writable icon on the tool's **SetUp** dialogue box. The easiest way to do so is to drag the included directory's icon from a directory display to the writable field.

As mentioned before, you can use the subdirectory `h` of the work directory for the last common type of header file – your own header files, which you refer to as user files with directives such as:

```
#include "myfile.h"
```

This is all you need to know for basic use of CC with largely default options. The rest of this section provides a level of detail useful for reference or studying if you wish to use CC in a non-standard way.

Reference section

The way in which the preprocessor looks for included files depends on three factors:

- whether the filename is rooted
- whether the filename in the `#include` directive is between angle brackets `<>` or double quotes `" "`
- use of the **Include** and **Default path** SetUp options (including the special filename `:mem`).

If a filename is not rooted (as defined earlier) the preprocessor looks for it in a sequence of directories called the search path.

Search path

The order of directories in the search path is as follows:

- 1 The compiler's own in-memory filing system.
This is only searched for `#include <filename>` directives when you have not enabled the SetUp menu's **Default path** option.
- 2 The *current place* (see the section *Nested includes* on page 20).
This is only searched for `#include "filename"` directives.

- 3 Arguments to the **SetUp** dialogue box's **Include** option, if used.
As noted above, this is set to `C:` by default, and so the standard library directories in `AcornC/C++.Export.APCS-32.Lib` will be searched.
- 4 The system search path:
 - The path given as an argument to the **Default path** SetUp menu option (see below), if this is enabled; otherwise
 - The value of the system variable `C$Libroot`, if this is set; otherwise
 - `$.Clib`.

Nested includes

The *current place* is the directory containing the source file (C or C++ source, or `#included` header) currently being processed by the compiler. Often, this will be the work directory.

When a file is found relative to an element of the search path, the name of the directory containing that file becomes the new current place. When the compiler has finished processing that file it restores the old current place. So at any given instant, there is a stack of current places corresponding to the stack of nested `#includes`.

For example, suppose the current place is `$.include` and the compiler is seeking the `#included` file `"sys.defs.h"` (or `"sys.h.defs"`, `"sys/defs.h"`, etc). Now suppose this is found as:

```
$.include.sys.h.defs.
```

Then the new current place becomes `$.include.sys`, and files `#included` by `h.defs`, whose names are not rooted, will be sought relative to `$.include.sys`.

This is the search rule used by BSD UNIX systems. If you wish, you can disable the stacking of current places using the SetUp menu option **Features** with the argument `k`, to get the search rule described originally by Kernighan and Ritchie in *The C Programming Language*. Then all non-rooted user includes are sought relative to the directory containing the source file being compiled.

In all this, the penultimate `.c`, `.c++` and `.h` components of the path are omitted. These are logically part of the filename – a filename extension – not logically part of the directory structure. However, directory names other than `c`, `c++`, `h`, `o` and `s` are not so recognised (as filename extensions) and are used 'as is'. For example, the name `sys.new.defs` is exactly that: it is not translated to `sys.defs.new` and, if it is found, the new part of the name does become part of the new current path.

Use of :mem

You can use the SetUp menu option **Default path** to provide your own system search path, as mentioned in step 4 of the section *Search path* above. The preprocessor will then use the argument you give to the **Default path** option as the system search path. You will only require this feature if you use implementations of the C library other than those provided with the Acorn C system.

Use of the **Default path** option also prevents a `#include <filename>` directive being first searched for in the in-memory filing system (see step 1 of the section *Search path* above). It can be reinstated by using the pseudo-filename `:mem` as an argument to the **Default path** or **Include** options. If `:mem` is included in the search path in this way, its position in the path is as specified – not necessarily first – so you can take complete control over where the compiler looks for `#included` files.

Use of C\$Libroot

C\$Libroot is an environment variable that you can use to provide your own system search path, as shown in step 4 of the section *Search path* above. It is not needed for normal use of the compiler.

If C\$Libroot is set, and you have not used the **Default path** option, the preprocessor will use the variable's value as the system search path. By default, C\$Libroot is not set.

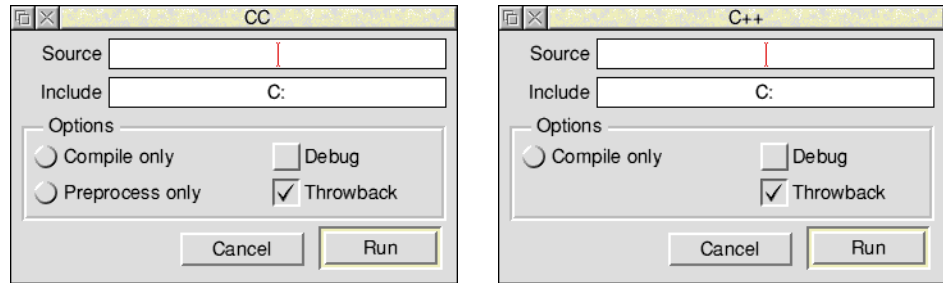
To set the value of C\$Libroot to, for example, `$.MyLib`, at the command line type:

```
*Set C$Libroot $.MyLib
```

This variable is also used by the Acorn C/C++ system as the library search path, if set. With the example given, the compiler will now look for include files in `$.mylib.h`, and for libraries in `$.mylib.o`.

The SetUp dialogue box

Clicking Select on the tool's icon bar icon or dragging a source file from a directory display to this icon brings up the tool's SetUp dialogue box:



Source

This writable icon in the SetUp dialogue box contains the names of the source files to be compiled.

When the SetUp box is obtained by clicking on the tool's main icon, it comes up with this icon containing its previous setting. You can thus repeat your previous compilation by just clicking on the **Run** button.

If the SetUp box appears as a result of dragging a source file to the main icon, the writable **Source** icon appears containing the new source file name.

When the SetUp box appears the **Source** icon has input focus, and can be edited in the normal RISC OS fashion. If you select a further source file in a directory display and drag it to this writable icon, its name is added to a list of those already there.

If you drag pre-compiled or pre-assembled object files to the **Source** icon, they are included in the set of object files linked together in an integral link step after the source files themselves have been compiled to object files.

Include

This SetUp dialogue box icon adds specified directories to the list of places which are searched for `#include` files. The directories in the **Include** icon are searched in the order in which they are given. The path should end with the name of a directory, with no `.h`, which is added automatically.

The default setting of **Include** is to `C:.` This makes the preprocessor search for headers in the directories listed in the RISC OS environment variable `C$Path`, set by `AcornC/C++.!SetPaths`. The directories listed are those that hold all the libraries supplied with the product in `AcornC/C++.Export.APCS-32.Lib`.

For more details of how to use `#include` lines and places searched for headers – both before and after those in this **Include** list – see the section *File naming and placing conventions* on page 14.

Compile only

This option switches off or on the linking of object files. When enabled, the link step is not performed, and the tools output object files. If you're only compiling one source file, you drag the object file produced from a **Save as** dialogue box. Otherwise, multiple files are saved in the `o` subdirectory of the work directory.

If not enabled, both CC and C++ instead perform an integral link step, linking any object files produced by compilation to any additional ones dragged to the **Source** icon, and library files, producing an executable program file. You control the saving of this from a **Save as** dialogue box.

Compile only is not enabled by default.

Preprocess only

This option is not available for the C++ tool.

If this option is enabled, only the preprocessor phase of the compiler is executed. The output from the preprocessor is sent to the standard output window. The standard non-interactive tool output window save facility is useful here to save this output to a file or SrcEdit window. By default, comments are stripped from the output, but see the SetUp menu option **Keep comments** on page 26.

Preprocess only is not enabled by default.

Debug

This option switches on or off production of debugging tables. When enabled, extra information is included in the resultant object files and image files which enables source level debugging of the linked image by the DDT debugger. If this option is disabled, any image file finally produced can only be debugged at machine level.

Enabling debug also disables the instruction scheduler, such that instructions are output in the sequence that the source code implies. There will also be no inlining performed, so calls to functions that would ordinarily be inlined will be left as calls to discrete functions.

If you are only compiling the source to object files, you must remember to enable debugging in the Link tool when you link them. If you don't, you'll lose the debugging information produced by the CC and C++ tools.

Debug is not enabled by default.

Throwback

This option switches editor throwback on or off. When enabled, if the DDEUtils module and SrcEdit are loaded, any compilation errors cause the editor to display an error browser. Double clicking Select on an error line in this browser makes the editor display the source file containing the error, with the offending line highlighted. See the chapter *SrcEdit* on page 73 of the accompanying *Desktop Tools* guide for more details.

Throwback is on by default.

The SetUp menu

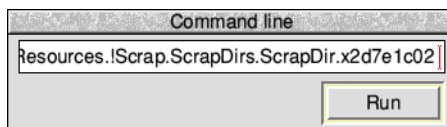
Clicking Menu on the SetUp dialogue box brings up the SetUp menu. The CC menu contains some options not available on the C++ menu, but the two menus are otherwise virtually identical:

CC	C++
Command line ▶	Command line ▶
Default path ▶	Default path ▶
Keep comments ▶	Define ▶
Define ▶	Undefine ▶
Undefine ▶	Assembler
Debug options ▶	Module code
Profile	✓ Libraries ▶
Assembler	Features ▶
Module code	No warnings
✓ Libraries ▶	✓ Work directory ▶
Features ▶	Others ▶
Suppress warnings ▶	
Suppress errors ▶	
UNIX pcc	
ISO C90	
ISO C99	
Errors to file ▶	
Listing	
✓ Work directory ▶	
Others ▶	

The options on this menu are described in the following subsections.

The command line

The **Command line** item at the top of the SetUp menu leads to a small dialogue box in which the command line equivalent of the current SetUp options is displayed:

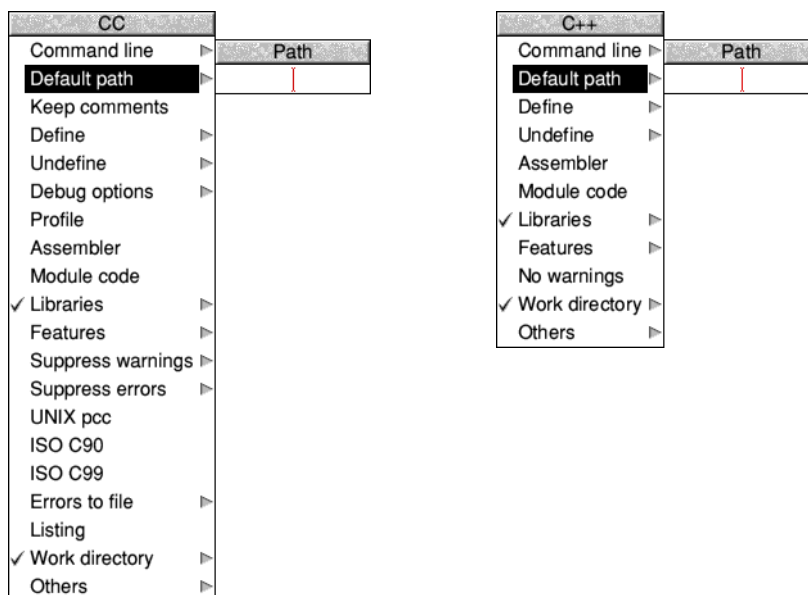


Clicking on the **Run** action button in this dialogue box starts compilation in the same way as that in the main SetUp box. Pressing Return in the writable icon in this box has the same effect. Before starting compilation from the command line box, you can edit the command line textually, although this is not normally useful.

Controlling the preprocessor

Default path

The **Default path** entry on the SetUp menu leads to a writable icon in which you specify a comma-separated list of directories to be searched for included files:



This overrides the system include path with the list of directories. You can specify the memory file system in the list by using the name `:mem` (in any case). An example is:

```
myhdrs, :mem, $.proj.public.hdrs
```

For more details of the system include path and searching for include files in general, see the section *File naming and placing conventions* on page 14.

Default path is not enabled by default.

Keep comments

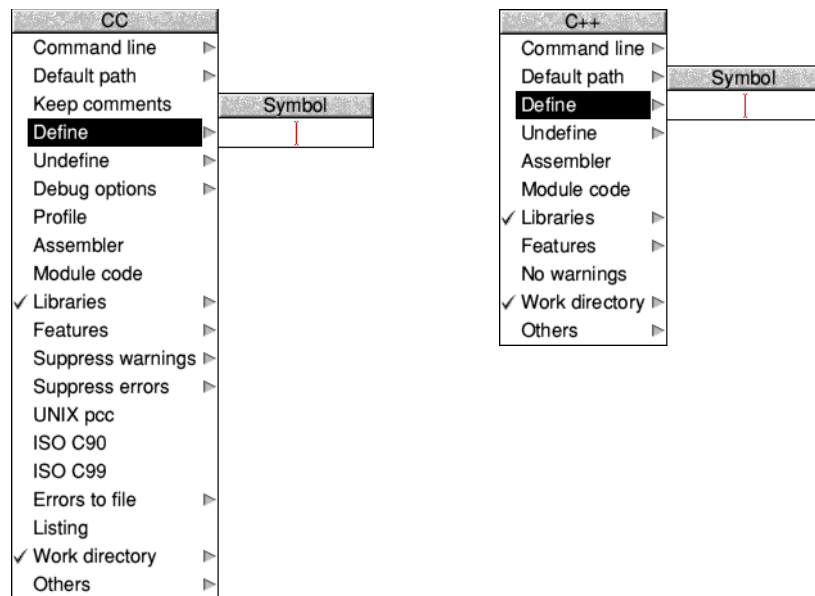
This option is not available for the C++ tool.

When enabled in conjunction with **Preprocess only**, this option retains comments in preprocessor output.

Keep comments is not enabled by default.

Define

The **Define** option on the SetUp menu leads to a writable icon in which you can predefine preprocessor macros:



You can enter two forms of macro predefinition:

```
sym=value
sym
```

These both define *sym* as a preprocessor macro for the compilation. The two forms are equivalent to the lines:

```
#define sym value
#define sym 1
```

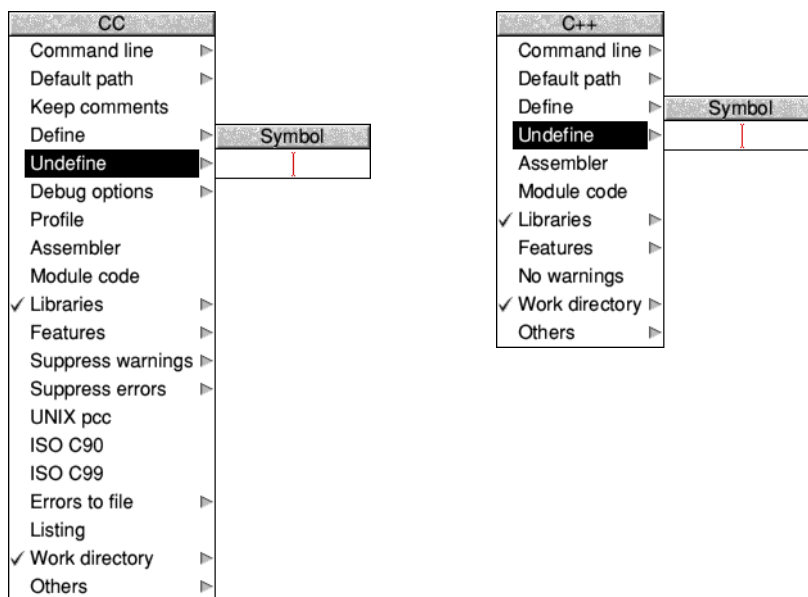
at the head of the source file.

You can enter multiple symbols as a space-separated list.

Define is not enabled by default.

Undefine

The **Undefine** option on the SetUp menu leads to a writable icon in which you can undefine preprocessor macros:



You enter the name of the macro concerned, eg:

```
sym
```

Use of this option is then equivalent to the line:

```
#undef sym
```

at the head of the source file.

You can enter multiple symbols as a space-separated list.

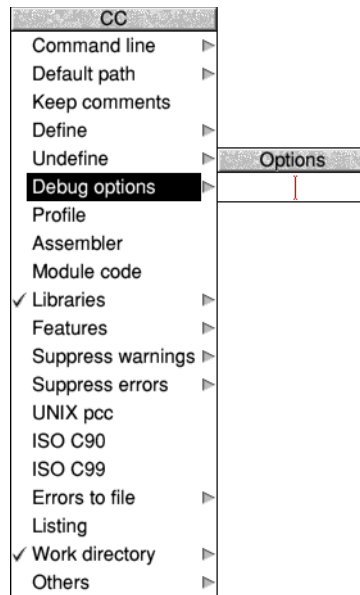
Undefine is not enabled by default.

Controlling code generation

Debug options

This option is not available for the C++ tool.

The **Debug options** option on the SetUp menu leads to a writable item in which you enter a set of modifier letters:



The modifier letters limit the debugging tables generated in response to enabling the **Debug** option on the SetUp dialogue box. The letters recognised are:

- f generate information on functions and top-level variables (outside functions)
- l generate information describing each line in the file
- v generate information describing all variables

You can use these letters in any combination. Specifying all the letters will include all information in the debugging tables, which has the same effect as leaving the options writable item blank.

Debug options is blank by default.

Profile

This option is not available for the C++ tool.

Enabling this SetUp menu option causes the compiler to generate code to count the number of times each function is executed. This is called profiling.

The counts can be printed by calling `_mapstore()` to print them to `stderr` or by calling `_fmapstore("filename")` to print them to a named file of your choice. You should do this just before the final statement of your program.

Profiling is not supported by the shared C library, so you must link programs to be profiled with `ANSILib`. If you wish, you can link with both `Stubs` and `ANSILib`, in which case only the code for `_mapstore()` and `_fmapstore()` will be included from `ANSILib`; your program will continue to use the shared C library, and will be much smaller than if linked with `ANSILib` alone.

The printed counts are lists of *lineno: count* pairs. The *lineno* value is the number of a line in your source code, and the *count* value is the number of times it was executed. Note that *lineno* is ambiguous: it may refer to a line in a `#include` file. However, this is rare and usually causes no confusion.

Provided you didn't compile your program with the **Features** option with `f` as an argument, blocks of counts will be interspersed with function names. In the simple cases, the output reduces to a list of line-pairs like:

function

lineno: count, where count is the number of times function was executed.

If you use the SetUp menu option **Others** to add the text `-px` to the command line, profiling of basic blocks within functions is performed in addition to profiling the functions. If you do this, the *lineno* values within each function relate to the start of each basic block. Sometimes, a statement (such as a `for` statement) may generate more than one basic block, so there can be two different counts for the same line.

Profiled programs run slowly. For example, when compiled with **Profile** enabled, Dhrystone 1.1 runs at about $\frac{5}{8}$ speed; when compiled `-px` it runs at only about $\frac{3}{8}$ speed.

There is no way, in this release of C, to relate execution counts to the proportion of time spent in each section of code. Nor is there any tool for annotating a source listing with profile counts. Future releases of C may address these issues.

Profile is not enabled by default.

Assembler

If this SetUp menu option is enabled, no object code is generated and, naturally, no attempt is made to link it. If only one assembly listing file is produced, you save it from a standard save dialogue box. If more than one is produced these are placed in the subdirectory `s` of the work directory.

Assembler is not enabled by default.

Module code

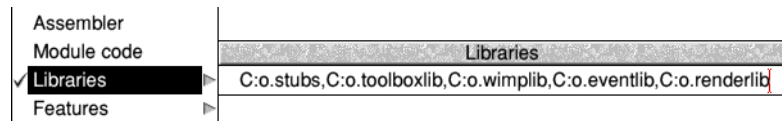
This SetUp menu option must be enabled when compiling code for linking into a RISC OS relocatable module, otherwise it should not be enabled. When enabled, code is produced which allows the module's static data to be separated from its code, hence be multiply instantiated.

Module code is not enabled by default.

Controlling the linker

Libraries

The **Libraries** option on the SetUp menu leads to a writable icon in which you specify a comma-separated list of filenames of libraries to be used in an integral link step:



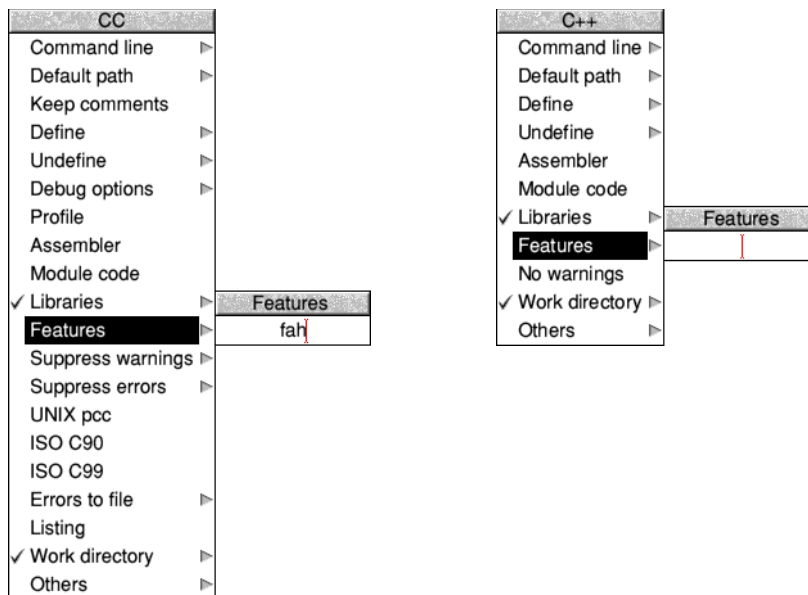
The libraries specified with this option are used instead of the standard one (`C:o.Stubs`), not in addition to it.

Libraries is enabled by default.

Using the Features menu option

Features

The **Features** option on the SetUp menu leads to a small writable icon in which you can specify additional compiler features with single modifier letters:



This entry controls a variety of compiler features, including certain checks on your code more rigorous than usual. At least one of the following modifier letters must be entered if **Features** are enabled:

- Check for certain types of data flow anomalies. The compiler performs data flow analysis as part of code generation. The checks enabled by this option can sometimes indicate when an automatic variable has been used before it has been assigned a value.
- Enable verbose mode. This forces the compiler to always print the summary line when it is finished even if there are no warnings or errors:
`x warnings, y errors, z serious errors`
- Enable the Limited pcc option. This allows characters after `#else` and `#endif` preprocessor directives (treated as comments), and explicit casts of integers to function as pointers (forbidden by C90 and later ISO standards). These features are often required in order to use pcc-style include files in C90, C99 or C18 mode.

- e Check that external names used within the file are still unique when reduced to six case-insensitive characters. Some old linkers only provide six significant characters in their symbol tables. This can cause problems with clashes if a system uses two names such as `getExpr1` and `getExpr2`, which are only unique in the eighth character. The check can only be made within one compilation unit (source file) so cannot catch all such problems. Acorn C and C++ allow external names of up to 256 characters, so this is a portability aid.
- f Do not embed function names in the code area. The compiler does this to make the output produced by the stack backtrace function (which is the default signal handler) and `_mapstore()` more readable. Removing the names from the compiler makes the code slightly smaller (typically 5%) at the expense of less meaningful backtraces and `_mapstore()` outputs.
- h Check that all external objects are declared in some included header file, and that all static objects are used within the compilation unit in which they are defined. These checks support good modular programming practices.
- i In the listing file (see the **Listing** option) include the lines from any files included with directives of the form:
`#include "file"`
- j As above, but for files included by lines of the form:
`#include <file>`
- k Use K&R search rules for nested `#include` directives (the 'current place' is defined by the original source file and is not stacked; see the section *File naming and placing conventions* on page 14 for details).
- l Instruct the compiler not to use the link register, `lr`, as a work register. When this feature is enabled `lr` is only used as a link register for function entry and exit. This may aid debugging in some situations.
- m Give a warning for preprocessor symbols that are defined but not used during the compilation.
- n Embed function names in the code area (see `f` feature). This improves the readability of the output produced by the stack backtrace run time support function and the `_mapstore()` function (see *Profile* on page 29).
However, it does increase the size of the code area slightly (around 5%). In general it is not useful to specify the `f` feature with **Profile** (i.e. `-p`).
- p Report on explicit casts of integers into pointers, eg:
`char *cp = (char *) anInteger;`
Implicit casts are reported anyway, unless suppressed by the **Suppress warnings** option.

- r Let `longjmp` corrupt register variables. This will result in some local variables being corrupted and it should therefore be used with caution. This option is not compliant with the ISO C standard.
- s Annotate the assembly output with addresses and compiled op-codes. This feature is only relevant when generating assembly output as described under `-S` in *Code generation options* on page 49.
- t Allow enums to be different sizes, depending on their values. They can be contained in a signed char, unsigned char, signed short, unsigned short, signed int or unsigned int.
- u By default, the source text as ‘seen’ by the compiler after preprocessing (expansion) is listed. If this feature is specified then the unexpanded source text, as written by the user, is listed. Consider the line

```
p = NULL;
```

By default, this will be listed as `p= (0) ;`. With the `u` feature specified, it will be listed as `p=NULL ;`.
- v Report on all unused declarations, including those from standard headers.
- w Allow string literals to be writable, as expected by some UNIX code, by allocating them in the program’s data area rather than the notionally read-only code area.
- y Only allow enums to be stored as ints. This is the opposite of the `t` feature and is the default setting for backwards compatibility.

When writing high-quality production software, you are encouraged to use at least the `fah` **Features** options in the later stages of program development (the extra diagnostics produced can be annoying in the earlier stages).

Features is not enabled by default.

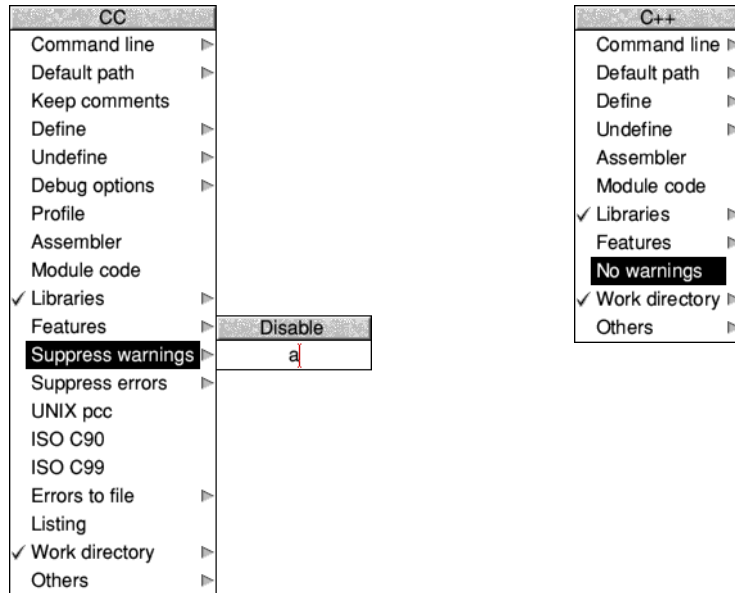
Handling warnings and errors

Suppress warnings

The **Suppress warnings** option on the SetUp menu prevents warnings from appearing.

For the C++ tool, all warnings are suppressed.

For the CC tool, this menu option leads to a writable icon in which you can enter a set of modifier letters:



The modifier letters specify various kinds of warning message to be suppressed by CC. Usually the compiler is very free with its warnings, as this tends to indicate potential portability or other problems. However, too many such messages can be a nuisance in the early stages of porting a program from old-style C, so you can disable them. There are also a couple of warnings that are off by default and can be enabled by a letter prefixed by a +.

The modifier letters for CC are:

- a Suppress Use of = in a condition context warning. This is given when the compiler encounters statements such as `if (a=b) {...` where it is quite possible that `==` was intended.
- b Suppress Unknown pragma warnings.
- c In earlier versions of the compiler this suppressed the Use of reserved C++ keyword warning. This option is now obsolete, instead warnings about C++ keywords are now disabled by default and can be enabled by `+u` (see below).
- d Suppress Old-style function and Deprecated declaration `foo()` - give arg types warnings. Use of old-style function declarations is deprecated in ANSI/ISO C, and in a future version of the standard this feature may be removed. However, it is useful sometimes to suppress this warning when porting old code.
- e Suppress cast between function pointer and non-function object warning.
- f Suppress Inventing "extern int foo()" message. This may be useful when compiling old-style C as if it were standard C.
- h Suppress warnings of format parameters not being declared as `const` in functions which take format parameters (`printf`, `scanf`, `_swix` etc.). This warning is only ever issued in fussy/strict mode and never in pcc mode (see `-fussy` and `-pcc` in *Keyword options* on page 44).
- l Suppress Lower precision in wider context warning. This warning is issued when an expression on the right hand side of an assignment is of a narrower width than the object it is being assigned to with a potential loss of precision (for example multiplying two ints and assigning the result to a long long).
- n Suppress Implicit narrowing cast and Lower precision in wider context warnings. This warning is issued when the compiler detects a cast of an expression to an object of narrower width (eg long to int, float to int). This can cause problems with loss of precision for certain values.
- p Suppress non-ANSI `#include <...>` warning. ANSI require that `#include <...>` should only be used for standard headers, but it can be useful to disable this warning when compiling code which does not conform to this aspect of the standard.
- r Suppress `_swi` format warnings. The `_swix` and `_swix` calls are used to pass parameters as registers to Software Interrupts routines. By default the compiler checks that the number of registers specified matches the number of arguments and warns if not.

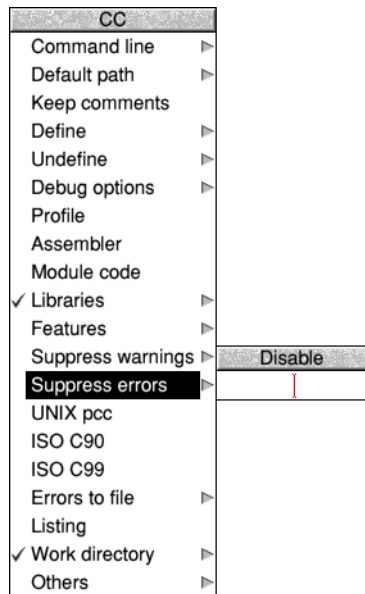
- s Warn about switch/case blocks which switch on an enum type variable but do not provide enough cases to cover all the values in the enumeration.
- v Suppress Implicit return in non-void context warning. This is most often caused by a return from a function which was assumed to return int (because no other type was specified) but is in fact being used as a void function.
- x Disable Declared but not used warnings which are shown when functions or variables are declared but never used within their scope.
- z Suppress Undefined macro in #if - treated as 0 warnings.
- +g Enables warnings about unguarded or wrongly guarded header files.
- +u Enables warnings about C++ keywords. This replaces the old c option, which is now obsolescent.

If you enter a space in the writable icon, then Select or Return, all warning messages from CC are suppressed.

Suppress errors

This option is not available for the C++ tool.

The **Suppress errors** option on the SetUp menu leads to a writable icon in which you can enter a set of modifier letters:



These modifier letters can be used to force CC to accept C source which would normally produce errors. If any of these options are needed, it means that the C source in question does not conform to the C standard (CC normally generates precisely the diagnostics required by the standard).

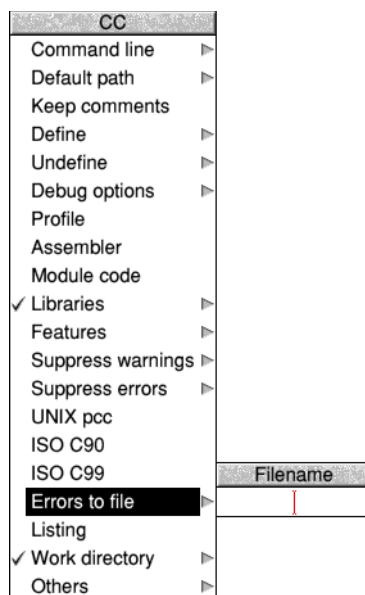
The modifier letters are:

- c Suppresses all implicit cast errors, e.g. 'implicit cast of non-0 int to pointer'.
- f Suppresses errors for unclean casts such as short to pointer. This option is not currently implemented in the compiler, although it was included in previous versions of the documentation.
- i Suppresses syntax checking for #if. This option is not currently implemented in the compiler, although it was in previous versions of the documentation.
- p Suppresses the error which occurs if there are extraneous characters at the end of a preprocessor line.
- z Suppresses the error if a zero-length array is used.

Errors to file

This option is not available for the C++ tool.

Errors to file allows you to specify a file to which error messages are output for later inspection:



Selecting the C dialect

ISO C90

This option is not available for the C++ tool.

Enabling this SetUp menu option switches to using C90 instead of C18 which is the default. This makes the compiler conform to the ISO 9899:1990 standard and the C99 and C18 features described in *C99 features* on page 97 and *C18 features* on page 112 are unavailable. This makes the compiler behave like the Acorn C release 5 compiler, although it shouldn't be necessary to select this mode to compile old programs.

This option changes the syntax that is acceptable to the compiler, but the default header and library files are still used.

ISO C99

This option is not available for the C++ tool.

Enabling this SetUp menu option switches to using C99 instead of C18 which is the default. This makes the compiler conform to the ISO 9899:1999 standard and the C18 features described in *C18 features* on page 112 are unavailable. This should only be necessary to compile old programs.

This option changes the syntax that is acceptable to the compiler, but the default header and library files are still used.

UNIX pcc

This option is not available for the C++ tool.

Enabling this SetUp menu option switches to compiling 'portable C compiler' C rather than ANSI/ISO C. This is based on the original Kernighan and Ritchie (K&R) definition of C, and is the dialect used on UNIX systems such as Acorn's RISC iX product. This option changes the syntax that is acceptable to the compiler, but the default header and library files are still used. See the section on this option in the chapter *Portability* on page 301 for more details.

UNIX pcc is not enabled by default.

Listings

Listing

This option is not available for the C++ tool.

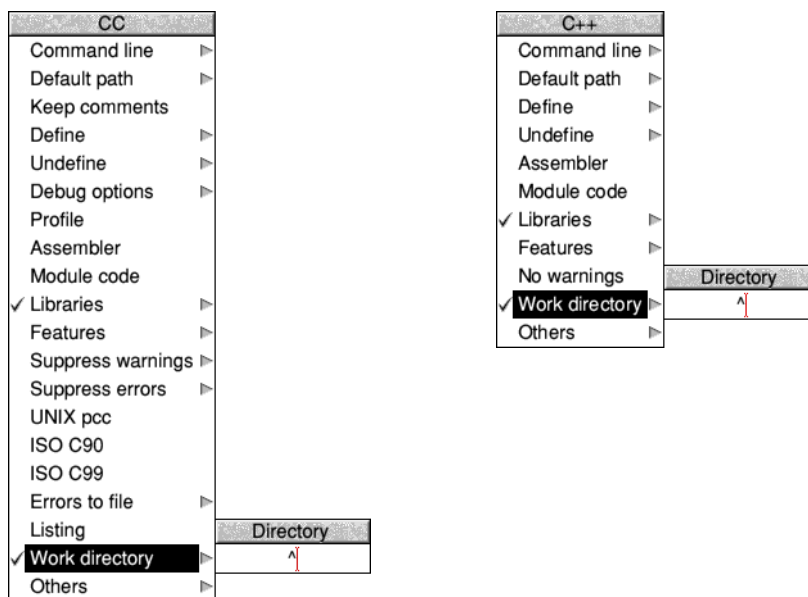
Enabling this SetUp menu option causes a listing file to be created. This consists of lines of source interleaved with error and warning messages. You can get finer control over the contents of this file using the **Features** option (see page 31).

Listing is not enabled by default.

Choosing your work directory

Work directory

The **Work directory** entry on the SetUp menu leads to a writable icon in which you specify the work directory:



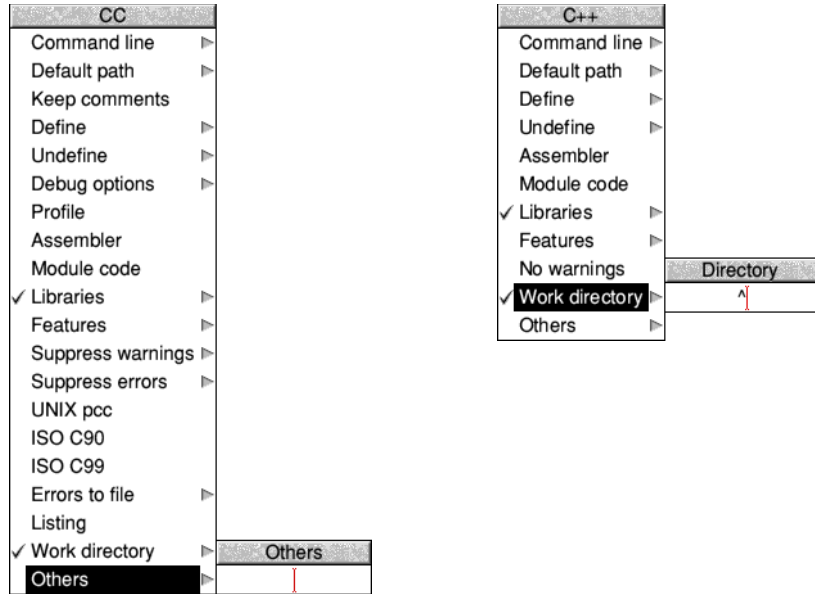
The effect of this option is described in the section *File naming and placing conventions* on page 14.

The default **Work directory** setting is ^.

Specifying other command line options

Others

The **Others** option on the SetUp menu leads to a writable icon in which you can add an arbitrary extra section of text to the command line to be passed to the relevant underlying program:

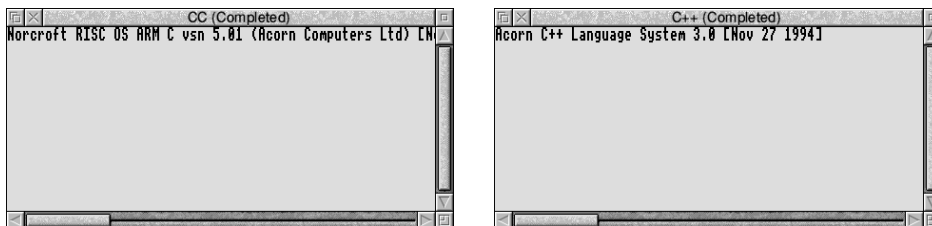


This facility is useful if you wish to use any feature which is not supported by any of the other entries on the SetUp dialogue box and menu. This may be because the feature is used very little, or because it may not be supported in the future.

For a full description of command line options, see *Command lines* on page 43.

Output messages

The CC and C++ tools output text messages as they proceed. These include preprocessed source (see **Preprocess only**), warning and error messages. By default any such text is directed into a scrollable output window:



This window is read-only; you can scroll up and down to view progress, but you cannot edit the text without first saving it. Clicking Select on the scrollable part of this window has no effect, to indicate this.

The contents of the window illustrated above are typical of those you see from a successful compilation – the title line of the compiler with version number, followed by no error messages.

Clicking Adjust on the close icon of the output window switches to the output summary dialogue box. This presents a reminder of the tool running (CC or C++), the status of the task (Running, Paused, Completed or Aborted), the time when the task was started and the number of lines of output that have been generated (ie those that are displayed by the output window):



Clicking Adjust on the close icon of the summary box returns to the output window.

Both the above output displays follow the standard pattern of all the non-interactive Desktop tools. The common features of the non-interactive Desktop tools are covered in more detail in the chapter *General features* on page 103 of the accompanying *Desktop Tools* guide. Both tools' output displays and the menus brought up by clicking Menu on them offer the standard features allowing you to abort, pause, or continue execution (if the execution has not completed); and to save output text to a file, or repeat execution.

Error messages appear in the output viewer, with copies in the editor error browser when throwback is working. The appendixes *C errors and warnings* on page 363 and *C++ errors and warnings* on page 399 contain more details for interpreting error messages.

Preprocessed source appearing in the output window is often very large for compilation of complex source files. The scrolling of the output window is useful to view it, and to investigate it with the full facilities of the source editor, you can save the output text straight into the editor by dragging the output file icon to the SrcEdit main icon on the icon bar (providing Wimp\$Scrap is properly set on your machine).

The icon bar menu

Clicking Menu on either the CC or the C++ icon on the icon bar gives the following menu:



Save options saves all the tool's current options, including those set both from the SetUp dialogue box and from the **Options** item on this menu. When you restart the tool it is initialised with these options rather than the defaults.

The **Options** item on the main menu allows you to enable **Auto run**, **Auto save** or start the output display as either a text window (default) or summary box. When **Auto run** is enabled, dragging a source file to the tool's icon starts a compilation immediately with the current options, rather than displaying the SetUp box first. When **Auto save** is enabled, output object files are saved to suitable places automatically without producing a save dialogue box for you to drag the file from. Both **Auto run** and **Auto save** are off by default.

For a description of each option in the tool's menu see the chapter *General features* on page 103 of the accompanying *Desktop Tools* user guide.

Command lines

For normal use you do not need to understand the syntax of the underlying CC and C++ programs' command lines, as they are generated automatically for you from the **SetUp** dialogue box and menu settings.

The syntax of the command lines is:

```
cc «options» filenames
c++ «options» filenames
```

By default, the C compiler and C++ translator look for source files, and create object, assembler and listing files, beneath the current work directory.

Many aspects of the programs' operation can be controlled via command-line options. All options are prefixed by a minus sign. There are two classes of option: *keywords* and *flags*:

- Keywords are recognised in upper case or lower case.
- A flag is a single letter, sometimes followed by an argument. Whenever this is the case, the C compiler allows white space to be inserted between the flag letter and the argument. However, this is not always true of other C compilers, so in the following subsections we only list the form that would be acceptable to a UNIX C compiler. Similarly, we only use the case of the letter that would be accepted by a UNIX C compiler.

By using the conventions common to many C compilers, you can build portable *makefiles* that you can easily move between different environments.

The options are listed below. Where an option merely gives a page reference to a desktop equivalent, you should see that page for full details. Should you need to use any of the more esoteric options that have no direct desktop equivalent, remember that you can always add them to the SetUp menu's **Others** option (see *Specifying other command line options* on page 40).

Where an option is shaded, we recommend that you don't use it with C++. You may use all options with CC, save for the *Translator options* on page 48, which are used by CFront and hence irrelevant to CC.

Keyword options

Command line option	Description
-help	Outputs a summary of the command line options.
-c18	Allow C99 features as described in <i>C99 features</i> on page 97 and C18 features as described in <i>C18 features</i> on page 112. This is the default option and if -c18 is specified on the command line it overrides -c90, -c99, -ansi and -pcc if any of those options are specified too.
-ansi	Same as -c18
-c90	Equivalent to ISO C90 in the SetUp menu; see page 38. C99 and C18 features are disabled.
-c99	Equivalent to ISO C99 in the SetUp menu; see page 38. C18 features are disabled.
-pcc	Equivalent to UNIX pcc in SetUp menu; see page 38.
-fussy or -strict	Be extra strict about enforcing conformance to the C standard or to pcc conventions (e.g. prohibit the volatile qualifier in -pcc mode).
-list	Equivalent to Errors to file in SetUp menu; see page 37.
-via <i>file</i>	Reads in extra command line arguments from the given <i>filename</i> .
-errors <i>file</i>	Equivalent to Listing in SetUp menu; see page 39.
-litttleend or -li	Compile code suitable for a little-endian Arm.
-bigend or -be	Compile code suitable for a big-endian Arm.
-apcs <3> <i>qualifiers</i>	Specify which variant of the Arm Procedure Call Standard is to be used by the compiler. At least one qualifier must be present, and there must be no space between qualifiers. The following qualifiers are permitted:
/26<bit>	26 bit APCS variant.
/32<bit>	32 bit APCS variant (default).
/reent<rnt>	Reentrant APCS variant.
/nonreent<rnt>	Non reentrant APCS variant.
/swst<ackcheck>	Software stack checking APCS variant.
/noswst<ackcheck>	No software stack checking APCS variant.
/fp	Use a dedicated frame-pointer register.
/nofp	Do not use a frame-pointer.

Command line option	Description
<code>/fpe2</code>	Floating point emulator 2 compatibility.
<code>/fpe3</code>	Floating point emulator 3 compatibility (default).
<code>/vfp</code>	Sets the VFP attribute for any code or data areas, and sets VFP endianness rules for double-precision floating point constants, even if an FPA FPU is selected.
<code>/fpr«egargs»</code>	Floating point arguments passed in floating point registers.
<code>/nofpr«egargs»</code>	Floating point arguments are not passed in floating point registers.
<code>/inter«work»</code>	Support Arm/Thumb interworking.
<code>-arch arch</code>	Selects the processor architecture for which the code will be compiled. Supported values are 2, 2a, 3, 3g, 3m, 4, 4xm, 4t, 4txm, 5, 5xm, 5t, 5txm, 5te, 5tej, 5texp, 5tewmmx, 5tewmmx2, 6, 6k, 6t2, 6z, 7, 7-a, 7-r, 7-a.security, 8-a.32, 8-a.32.crypto, 8.1-a.32, 8.1-a.32.crypto, 8.2-a.32, 8.2-a.32.crypto, 8.3-a.32, 8.3-a.32.crypto, 8.4-a.32, 8.4-a.32.crypto, 8.5-a.32, 8.5-a.32.crypto, 8.6-a.32 and 8.6-a.32.crypto.

Command line option

`-cpu cpu`

Description

Selects the CPU for which the code will be optimised. The parameter may be any of the architecture names listed above or any of the following CPU names: arm2, arm3, arm6, arm600, arm610, arm7, arm7m, arm7dm, arm7tm, arm7tm-s, arm7tdm, arm7tdmi, arm7tdmi-s, arm700, arm704, arm710t, arm720t, arm740t, arm7ej-s, arm8, arm810, strongarm, strongarm1, sa-110, sa-1100, sa-1110, arm9tdmi, arm910t, arm920t, arm922t, arm9e-s, arm9ej-s, arm926ej-s, arm1176jzf-s, xscale, or cortex-a8.

When both `-cpu` and `-arch` are used simultaneously, the compiler will optimise for the processor/architecture given by `-cpu`, but generate code that will run on the architecture given by `-arch`. This allows the user to request optimising for a new processor while not ruling out the code being run on an older one. `-arch` and `-cpu` can be specified in either order; only the last `-cpu` and last `-arch` given on the command line are significant.

The following examples should make this clearer: code optimised for XScale but runs on v3.

default

`-cpu 7TDMI`

code optimised for 7TDMI, runs on v4T.

`-cpu 5`

code optimised for a typical v5 processor, runs on v5.

`-arch 5`

same as `-cpu 5`.

`-arch 3 -cpu`

code optimised for XScale but runs on v3.

XScale

`-cpu 2 -arch 5TE`

code optimised for 2, but requires v5TE (nonsensical, but allowed).

`-fpu fpu`

Selects the floating point unit for which the code will be compiled. Supported values are `fpa` and `none`.

Attempting to use floating point operations when `none` is selected will result in an error.

Command line option	Description
<code>-memaccess <i>types</i></code>	This option allows (or disallows) specific types of memory access; it is used to indicate capabilities of the memory system beyond the Arm core. An access type is enabled by prefixing it with a + and disabled by prefixing it with a -. The default setting is equivalent to specifying <code>-memaccess +L22+S22-L41</code> , and the three supported types are: <ul style="list-style-type: none"> L22 Load of a halfword from a halfword-aligned address S22 Store of a halfword to a halfword-aligned address L41 Load of a (rotated) word from a byte-aligned address
<code>-depend <i>dependfile</i></code>	Saves include file dependency lists, which are suitable for use with 'make' utilities.
<code>-throwback</code>	Equivalent to Throwback option icon in SetUp dialogue box; see page 24.
<code>-desktop <i>directory</i></code>	Equivalent to Work directory in SetUp menu; see page 39.
<code>-C++</code>	Assume C++ code is being processed. This option is only used by the C++ program, when invoking the compiler to pre-process C++ source before translation, and when compiling the generated C. When preprocessing under the -E option, comment handling is changed to correctly deal with C++'s <code>/**</code> comments (which are terminated by the end of the source line), and <code>#pragma</code> lines are passed through to the preprocessor output. During the C compilation stage, use of this flag disables certain warnings (most notably 'no side-effect in void context', and messages about unused variables), otherwise produced by some rather odd code constructs in the generated C. It also arranges that in any warning or error reports, the original (type-qualified) C++ source names are printed rather than the modified names CFront generates in order to implement overloading.

Command line option

`-ccversion version`

Description

Specify the minimum compiler version number that may be used where *version* is the required version number * 100. For example, specifying `-ccversion 542` would generate a `Compiler too old` error if the compiler version was below 5.42 (but note that this feature was only introduced in cc version 5.11).

Preprocessor options

Command line option

`-Idirectory`

Description

Equivalent to **Include** option icon in SetUp dialogue box; see page 22.

`-jdirectories`

Equivalent to **Default path** in SetUp menu; see page 25.

`-E`

Equivalent to **Preprocess only** option icon in SetUp dialogue box; see page 23.

`-C`

Equivalent to **Keep comments** in SetUp menu; see page 26.

`-M`

If this flag is specified, only the preprocessor phase of the compiler is executed (as with `cc -E`) but the only output produced is a list, on the standard output stream, of *makefile* dependency lines suitable for use by a make utility. This can be redirected to a file using standard UNIX notation. For example:

```
cc -M xxx.c >> Makefile.
```

`-Dsymbol«=value»`

Equivalent to **Define** in SetUp menu; see page 26.

`-Usymbol`

Equivalent to **Undefine** in SetUp menu; see page 27.

Translator options

These options affect the operation of CFront.

Command line option

`+v`

Description

Print commands as CFront executes them

`+w`

Equivalent to **Suppress warnings** in C++'s SetUp menu; see page 33. (**Suppress warnings** also uses CC's `-W` option.)

`+p`

Pedantic – compile strict C++

Command line option	Description
<code>+g</code>	Equivalent to Debug option icon in C++'s SetUp dialogue box; see page 23. (Debug also uses CC's <code>-g</code> option.)
<code>-F</code>	Send CFront output to <code>stdout</code> ; do not compile it

Code generation options

If you are using C++, we recommend you only use the following from the code generation options below: `-o`, `-g`, `-Ospace`, `-Otime`, `-S` and `-zM`.

Command line option	Description
<code>-o file</code>	The argument to the <code>-o</code> flag gives the name of the file which will hold the final output of the compilation step. In conjunction with <code>-c</code> , it gives the name of the object file; in conjunction with <code>-S</code> , it gives the name of the assembly language file. Otherwise, it names the final output of the link step.
<code>-g«options»</code>	Equivalent to Debug option icon in SetUp dialogue box and Debug options in SetUp menu; see pages 23 and 28.
<code>-Ospace</code>	Optimise for image size at the expense of execution time
<code>-Otime</code>	Optimise for execution time at the expense of image size
<code>-p«options»</code>	Equivalent to Profile in SetUp menu; see page 29.
<code>-S</code>	Equivalent to Assembler in SetUp menu; see page 30.
<code>-zM</code>	Equivalent to Module code in SetUp menu; see page 30.
<code>-zo</code>	Generate one AOF area for each function in the source file. This option is most useful when creating libraries because it allows the linker to remove unused functions from the final image. This option may increase the code size of individual functions slightly (typically by a few percent) because it reduces the potential for sharing addresses, data, and string literals between functions.

Linker options

Command line option	Description
<code>-c</code>	Equivalent to Compile only option icon in SetUp dialogue box; see page 23.
<code>-llibraries</code>	Equivalent to Libraries in SetUp menu; see page 30.

Warning and error message options

If you are using C++, we recommend you only use the following from the warning and error message options below: `-W`.

Command line option	Description
<code>-Woptions</code>	Equivalent to Suppress warnings in SetUp menu; see page 33.
<code>-eoptions</code>	Equivalent to Suppress errors in SetUp menu; see page 36.

Additional feature options

If you are using C++, we recommend you only use the following from the additional feature options below: `-zr` and `-f`.

Command line option	Description
<code>-zprAlphaNum</code>	This flag can be used to emulate <code>#pragma</code> directives. The letter and digit which follow it are the same characters that would follow the <code>'-'</code> of a <code>#pragma</code> directive. See <i>#pragma directives</i> on page 119 for details.
<code>-zrnumber</code>	This flag allows the size of (most) LDMs and (all) STMs to be controlled between the limits of 3 and 16 registers transferred. This can be used to help control interrupt latency where this is critical.
<code>-zc</code>	This flag makes the <code>char</code> type signed; it is normally unsigned. This option is not recommended for general use and is not required for ANSI-compatible source. If used incorrectly, this option can cause errors in the resulting image.
<code>-ffeatures</code>	Equivalent to Features in SetUp menu; see page 31.

Global data options

These options can be used to tune the compiler's handling of global data.

Command line option	Description
<code>-zzZI threshold size</code>	<p>Sets the value of the zero-initialised data threshold size. The compiler will place uninitialised global variables in the Zero Initialised (ZI) data area if their size is greater than <i>ZI threshold size</i> bytes.</p> <p>For example, you can force uninitialised global variables of any size to be placed in the ZI data area by specifying <code>-zz0</code>, though this may increase your code size. Alternatively you could use <code>-zz100000000</code> to prevent any variables being placed in the ZI data area.</p> <p>Use this option in combination with <code>-zt</code> to avoid increased code size. Option <code>-zzt</code> provides a convenient shorthand. The default threshold size is 8 bytes; this was 100 bytes in older versions of the compiler.</p>
<code>-zt</code>	<p>Disallows tentative declarations. If this option is specified the compiler assumes that any occurrence of a top level construct such as <code>int i;</code> is a definition without initialiser, rather than a tentative definition. Any subsequent definition with initialiser in the same scope will generate an error. This may fault code that conforms to the ISO standard.</p> <p>This option is useful in combination with the <code>-zz</code> option.</p>
<code>-zztZI threshold size</code>	<p>Combines the <code>-zt</code> and <code>-zz</code> options. For example, specify <code>-zzt0</code> to force the compiler to disallow tentative declarations and place all uninitialised global variables in the ZI data area.</p>

Worked examples

Several examples of C and C++ programs included with the Desktop Development Environment are worked through in this guide and in the *Desktop Tools* guide. A collection of examples are listed here illustrating various points and styles of working.

The following examples are in the directory `Sources.DDE-Examples.C/C++`, each in a subdirectory with the name of the example. For each program, we give a 'recipe' for how to compile, link and run the program. Filenames are given relative to the subdirectory containing each example unless otherwise stated. It is assumed that you have read the preceding parts of this chapter. For more details of the tool Make, see the chapter *Make* on page 59 of the accompanying *Desktop Tools* user guide. When you enter any command lines given below, you must first ensure that the currently-selected directory is the subdirectory containing the example being tried.

There are some further less trivial examples that we omit here. These show you how to implement more esoteric features, mainly involving interworking C and/or C++ with assembler. They are described elsewhere in the Acorn C/C++ manual set, together with necessary supporting technical information.

CHello

Purpose:	The standard most trivial C program. Try it as an exercise.
Source:	<code>c.HelloW</code>
Compile using:	default CC SetUp options
Run by:	double clicking on <code>HelloW</code>
Clean up by:	deleting <code>HelloW</code> and <code>o.HelloW</code>

C++Hello

Purpose:	The standard most trivial C++ program. Try it as an exercise.
Source:	<code>c++.HelloW</code>
Compile using:	default C++ SetUp options
Run by:	double clicking on <code>HelloW</code>
Clean up by:	deleting <code>HelloW</code> and <code>o.HelloW</code>

Sieve

Purpose:	The Sieve of Eratosthenes is often presented as a standard benchmark, though it is not very meaningful in this context.
Source:	<code>c.Sieve</code>
Compile using:	default CC SetUp options
Run by:	double clicking on <code>Sieve</code>
Clean up by:	deleting <code>Sieve</code> and <code>o.Sieve</code>

Dhrystone 2.1

Purpose:	Dhrystone 2.1 is the standard integer benchmark. Its results require careful interpretation (it often overstates the real performance of machines). Try as a first exercise in using Makefiles.
Sources:	<code>h.dhry</code> <code>c.dhry_1</code> <code>c.dhry_2</code>
Makefile:	<code>Makefile</code>
Build by:	double clicking on <code>Mk</code> .
Run by:	double clicking on <code>Dhrystone</code> Reply with any number in the range 20000 to 250000 to the prompt for number of iterations. Try a big number such as 200000 and time the execution with a stopwatch or sweep second hand to confirm the claimed performance. Note how performance depends on screen mode.
Rebuild by:	double clicking on <code>Mk</code> again (try altering some of the options in <code>Makefile</code> between rebuilds: eg compile in UNIX pcc mode by altering <code>CFLAGS</code>).
Clean up by:	double clicking on <code>MkClean</code> .

CModule

Purpose:	To illustrate how to implement a module in C. You can also use it as another exercise in using Makefiles. For more details on constructing relocatable modules in C see the chapter <i>How to write relocatable modules in C</i> on page 327.
Sources:	<code>c.CModule</code> <code>CModuleHdr</code>

Build using: CC of `c.CModule` with options **Compile only** and **Module code** enabled, saving output object file as `o.CModule`. CMHG of `cmhg.CModuleHdr` to `o.CModuleHdr`. Link of `o.CModule`, `o.CModuleHdr` and `C:CLib.o.Stubs` with **Module** enabled to the output file `CModule`.

or by: double clicking on `Mk`.

Run from: the command line using `CModule`

Test from: the command line using:

```
help tm1
help tm2
tm1 hello
tm2 1 2 3 4 5
tm1 1 2 3
tm2 hello
```

(try other combinations too)

```
*BASIC
> SYS &88000 : REM should give an error
> SYS &88001 : REM should give divide by 0 error
> SYS &88002 : REM no error, just a message
> SYS &88003 : REM no error, just a message
> SYS &88004 : REM same as &88000...
```

(now repeat some of these after issuing some invalid
* commands...)

```
>*foo
> SYS &88002
```

etc.

```
>QUIT
```

Clean up by: from the command line typing: `RMKill TestCModule` deleting `CModule`, `o.CModule` and `o.CModuleHdr` or by running `MkClean`.

Desktop application examples

The desktop applications !Hyper, !MinApp and !TBoxCalc and the various versions of SaveAs found in the Sources.DDE-Examples.Toolbox directory are all too complex to be described here in great detail.

They are best built by double clicking on the MkInstall file found in each example's source directory, which builds the application and installs it in the AcornC/C++.Install directory. They can be run by double clicking on their application icons.



CMHG (the *C Module Header Generator*) is a desktop tool which provides an easy interface to the CMHG program that Acorn C/C++ installs in your computer's library. The CMHG tool constructs command lines and passes them to the CMHG program. By using CMHG you can write a RISC OS relocatable module entirely in C without having to use Arm assembly language.

Every relocatable module has at its start (ie the part that loads into memory at its lowest address) a header table pointing to various items of data and program. Most of the items pointed to are optional, the pointers being zero if not needed. When writing a relocatable module in assembly language you lay this table out yourself, but when writing in C, you use CMHG to generate this for you. In addition to generating a module header, CMHG also inserts small standard routines to, for example, initialise the C language library support and make service call handling efficient.

To construct a relocatable module you write a number of routines in C with standard prototypes, some of these routines to be called with the processor in supervisor (SVC) mode. These are accompanied by a text description file written in a special syntax which CMHG understands. For details of this language and the specifications of the C routines, see the chapter *How to write relocatable modules in C* on page 327. For more details of relocatable module headers, see the chapter entitled *Modules* in the *RISC OS Programmer's Reference Manual*. For some hints about memory usage from relocatable module code, see the *RISC OS Programmer's Reference Manual*.

The rest of this chapter explains the (simple) controls of the CMHG tool. CMHG is one of the non-interactive desktop tools, its desktop user interface being provided by the FrontEnd module. It shares many common features with the other non-interactive tools. These common features are described in the chapter *General features* on page 103 of the accompanying *Desktop Tools* guide.

A note about Make

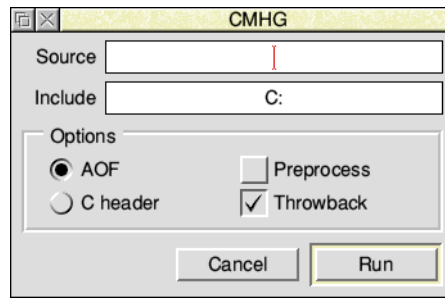
The Make tool (see the chapter *Make* on page 59 of the *Desktop Tools* guide) can also construct command lines for the underlying CMHG program. You'll find it a better tool for managing large projects. However, much of what is in this chapter is relevant, since Make sets options for the CMHG program with the CMHG tool's user interface.

Starting CMHG

To start the CMHG tool, first open the Apps .DDE directory display, then double click on !CMHG. Its icon appears on the icon bar:



Clicking Select on this icon, or dragging a CMHG description file from a directory display to this icon, brings up the **SetUp** dialogue box, from which you control the running of CMHG:



The **Source** writable icon is for the name of the description file to be processed. If you displayed the **SetUp** dialogue box by clicking on the CMHG icon bar icon, you will want to fill this in by dragging a CMHG description file from a directory display to this icon before running CMHG.

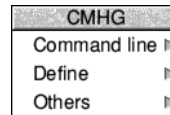
The **Include** icon adds specified directories to the list of places which are searched for `#include` files when the **Preprocess** option is selected. This works in exactly the same way as for the compiler - see *Include* on page 22 for details.

The CMHG tool can produce two types of output file. Selecting the **AOF** option means that it will create an object file. You link this with the object files compiled from your C code to produce your relocatable module. Selecting the **C Header** option causes CMHG to instead produce a C header file that contains `#defines` of constants for the commands declared in the description file, and function prototypes that you will need in your C code to interface with the code created by CMHG.

The **Preprocess** option instructs CMHG to pre-process the CMHG file using the C preprocessor. This is particularly useful for including C header files so that symbolic names can be used in place of numeric values. It also allows the use of conditional directives such as `#ifdef`.

Throwback enables editor throwback to be used when any errors are found in the CMHG file being processed.

Clicking Menu on the SetUp dialogue box brings up the CMHG **SetUp** menu, which contains the following items:



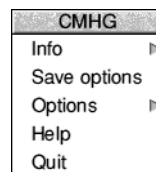
Command line enables you to examine or edit the actual command line. For more information on this option see the section *Command line options* on page 61.

The **Define** option on the SetUp menu leads to a writable icon in which you can predefine preprocessor macros. This works in exactly the same way as for the compiler - see *Define* on page 26 for details.

The **Others** option leads to a writable icon in which you can add an arbitrary extra section of text to the command line to be passed to the underlying CMHG program. This facility is useful if you wish to use any feature which is not supported by any of the other entries on the SetUp dialogue box and menu. For a full description of the available options, see *Command line options* on page 61.

The icon bar menu

Clicking Menu on the CMHG application icon on the icon bar gives access to the following options:



For a description of each option in the application menu see the chapter *General features* on page 103 of the accompanying *Desktop Tools* guide.

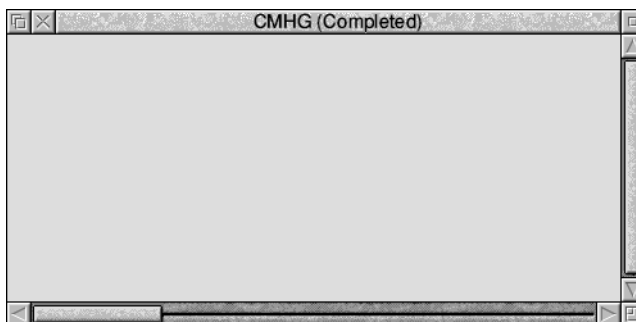
Example output

The following is an example CMHG description file, similar to that used to construct the FrontEnd module, which is itself a relocatable module written in C:

```
; Purpose: module header for the generalised front end module ;
module-is-runnable:                                     ; module start code
initialisation-code:  FrontEnd_init
service-call-handler:  FrontEnd_services  0x11          ; service-memory
```

```
title-string:      FrontEnd
help-string:      FrontEnd  1.00
command-keyword-table: FrontEnd_commands
                    FrontEnd_Start(min-args: 4, max-args: 5,
                                   help-text: "Help text\n"),
                    FrontEnd_Setup(min-args: 8, max-args: 8,
                                   help-text: "Help text\n")
swi-chunk-base-number: 0x081400
```

Running CMHG displays any error messages in the standard text output window for non-interactive tools. If all goes well, as it should do if you try CMHG with the above description file, this window is empty:



The output file produced is an object file. You link this with the object files compiled from your C code to produce your relocatable module.

Command line interface

For normal use you do not need to understand the syntax of the underlying CMHG program's command line, as it is generated automatically for you from the **SetUp** dialogue box and menu settings.

The syntax of the CMHG command line is:

```
cmhg <options> descfile <objfile <defsfile>>
```

descfile Filename of the CMHG description file.

objfile Filename of the output object file to link with your objects to form a relocatable module.

defsfile Filename of the output definitions header file, giving constants for the commands in the description file.

Command line options

Command line option	Description
<code>-help</code>	List command line options with brief explanations.
<code>-depend</code>	Passed on to <code>cc</code> in order to generate dependency information for <code>amu</code> .
<code>-o objfile</code>	The name of the final object file.
<code>-d defsfile</code>	The name of the C header file CMHG will generate.
<code>-p</code>	Instruct CMHG to pre-process the CMHG file using the C preprocessor. This is particularly useful for including C header files so that symbolic names can be used in place of numeric values.
<code>-Idirectory</code>	Specifies which directory or directories are to be searched for <code>#include</code> files. A list of directories may be given, separated by spaces and they are searched in the order given. The default include path (if this option is not used) is C: as for the C compiler. This option behaves like the Include option icon in the C compiler's SetUp dialogue box; see <i>Include</i> on page 22 for more details.
<code>-Dsymbol«=value»</code>	Defines the preprocessor macro <i>symbol</i> . Multiple symbols can be defined using a space-separated list. This behaves like Define in the C compiler's SetUp menu; see <i>Define</i> on page 26 for details.
<code>-Usymbol</code>	Undefines the preprocessor macro <i>symbol</i> . Multiple symbols can be defined using a space-separated list. This behaves like Undefine in the C compiler's SetUp menu; see <i>Undefine</i> on page 27 for details.
<code>-26bit</code>	Generate 26-bit code.
<code>-32bit</code>	Generate 32-bit code (default).
<code>-throwback</code>	Support error processing by Desktop Tools.

For example:

```
cmhg -p -DDEBUG -IC:,\TCPIPLibs: -o o.InetHdr InetHdr
```




ToANSI is a desktop tool which provides an easy interface to the ToANSI program that Acorn C/C++ installs in your computer's library. The ToANSI tool constructs command lines and passes them to the ToANSI program. ToANSI helps convert program source written in the PCC style of C to program source in the ANSI style of C. PCC is the UNIX Portable C Compiler, and closely follows K&R C, as defined by B Kernighan and D Ritchie in their book *The C Programming Language*.

ToANSI enables you to write (with care) programs that can be automatically converted between the PCC and ANSI dialects of C, hence assisting you in constructing easily portable programs. The associated tool ToPCC makes approximately the reverse translations to ToANSI. For more details of portability issues, see the chapter *Portability* on page 301. The changes that ToANSI makes to C source are listed in the section *ToANSI C translation* below.

ToANSI is one of the non-interactive desktop tools, its desktop user interface being provided by the FrontEnd module. It shares many common features with the other non-interactive tools. These common features are described in the chapter *General features* on page 103 of the accompanying *Desktop Tools* guide.

ToANSI C translation

ToANSI makes the following transformations to C source code or header text:

- Function declarations with embedded comments are rewritten without the comment tokens. This reverses the action of ToPCC with regard to function declarations, rewriting

```
type foo( /* args */ );
```

as

```
type foo( args );
```

This transformation is one which requires care in the use of ToANSI, as it can result in invalid C being uncommented.

- Function definitions of the form

```
type foo( a1, a2 )
```

```
type a1;
```

```
type a2;
```

```
{ ... }
```

are rewritten as

```
type foo( type a1, type a2 )
```

- A `va_alist` in the function definition is translated to

```
...
```

- `type foo()` is rewritten as `type foo(void)`.
- `VoidStar` (what ToPCC replaces `void *` with) is left untouched, as if it is correctly typedef'd to something suitable, thereafter its use is correct in both PCC and ANSI C.
- ToPCC rewrites `unsigned` and `unsigned long` constants using the typecasts `(unsigned)` and `(unsigned long)`. ToANSI does not reverse this change, as this is not required for correct ANSI C.

Note that ToANSI performs only simple textual translations and is not able to reliably diagnose C syntax errors, which may produce surprising results, so it is best to use ToANSI only on code you already know compiles.

A note about Make

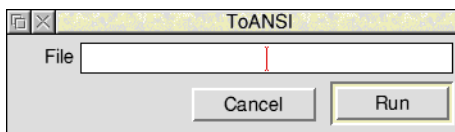
Since porting programs is usually a one-off process involving some experimentation, only direct use of ToANSI makes sense. You cannot use ToANSI from Make.

Starting ToANSI

To start the ToANSI tool, first open the Apps .DDE directory display, then double click on !ToANSI. Its icon appears on the icon bar:



Clicking Select on this icon, or dragging a source file from a directory display to this icon, brings up the **SetUp** dialogue box, from which you control the running of ToANSI:



ToANSI has hardly any options for its use, so its interface is simpler than most of the other Acorn C/C++ tools.

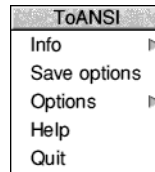
The **File** writable icon is for the name of the description file to be processed. If you displayed the **SetUp** dialogue box by clicking on the ToANSI icon bar icon, you will want to fill this in by dragging a source file from a directory display to this icon before running ToANSI.

Clicking Menu on the SetUp dialogue box brings up the ToANSI **SetUp** menu, which owing to the simplicity of ToANSI only has a single **Command line** item:



The icon bar menu

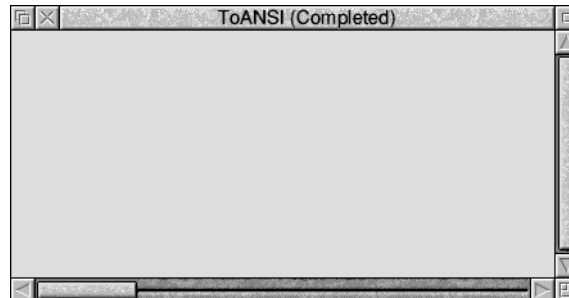
Clicking Menu on the ToANSI application icon on the icon bar gives access to the following options:



For a description of each option in the application menu see the chapter *General features* on page 103 of the accompanying *Desktop Tools* guide.

Example output

Running ToANSI displays any error messages in the standard text output window for non-interactive tools. If all goes well this window is empty:



As an example of using the tool ToANSI, open an empty SrcEdit text window and type the following example C source lines in it:

```
int foo(a, b)
float a;
double b;
{ }
```

Check that your `Wimp$Scrap` environment variable is set to a sensible file name, then save your new text file straight onto the ToANSI icon bar icon. Run ToANSI, then save the output text file straight onto the SrcEdit icon bar icon. The translated file looks like:

```
int foo(float a, double b)
{ }
```

Command line interface

For normal use you do not need to understand the syntax of the underlying ToANSI program's command line, as it is generated automatically for you from the **SetUp** dialogue box and menu settings.

The syntax of the ToANSI command line is:

```
toansi «options» «infile «outfile»»
```

options Options: the `-d` option describes ToANSI, and the `-help` option gives the command line syntax and options.

infile Filename of the input C source or header text file, which defaults to `stdin`.

outfile Filename of the output C source or header text file, which defaults to `stdout`.





ToPCC is a desktop tool which provides an easy interface to the ToPCC program that Acorn C/C++ installs in your computer's library. The ToPCC tool constructs command lines and passes them to the ToPCC program. ToPCC helps convert program source written in the ANSI style of C to program source in the PCC style of C. PCC is the UNIX Portable C Compiler, and closely follows K&R C, as defined by B Kernighan and D Ritchie in their book *The C Programming Language*.

ToPCC enables you to write (with care) programs that can be automatically converted between the ANSI and PCC dialects of C, hence assisting you in constructing easily portable programs. The associated tool ToANSI makes approximately the reverse translations to ToPCC. For more details of portability issues, see the chapter *Portability* on page 301. The changes that ToPCC makes to C source are listed in the section *ToPCC C translation* below.

ToPCC is one of the non-interactive DDE tools, its desktop user interface being provided by the FrontEnd module. It shares many common features with the other non-interactive tools. These common features are described in the chapter *General features* on page 103 of the accompanying *Desktop Tools* guide.

ToPCC C translation

ToPCC makes the following transformations to C source code or header text:

- Function declarations of the form
`type foo(args);`
are rewritten as
`type foo(/* args */);`
Any comment tokens `/*` or `*/` in `args` are removed.
- Function definitions of the form
`type foo(type a1, type a2) {...}`
are rewritten as
`type foo(a1, a2)`
`type a1;`
`type a2;`
- `A...` in the function definition is interpreted as `int va_alist`. Full translation of variadic functions is not performed.
- `type foo(void)`
is rewritten as
`type foo()`
- `Type void *` is converted to `VoidStar` which can be typedef'd to something suitable (eg `char *`).
- Unsigned and unsigned long constants are rewritten using the typecasts `(unsigned)` and `(unsigned long)`.
For example, `300u1` becomes `(unsigned long)300L`.

Note that ToPCC performs only simple textual translations and is not able to reliably diagnose C syntax errors, which may produce surprising results, so it is best to use ToPCC only on code you already know compiles.

A note about Make

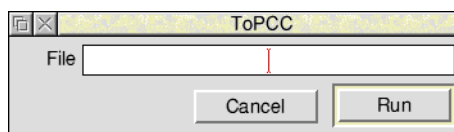
Since porting programs is usually a one-off process involving some experimentation, only direct use of ToPCC makes sense. You cannot use ToPCC from Make.

Starting ToPCC

To start the ToPCC tool, first open the `Apps.DDE` directory display, then double click on !ToPCC. Its icon appears on the icon bar:



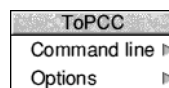
Clicking Select on this icon, or dragging a source file from a directory display to this icon, brings up the **SetUp** dialogue box, from which you control the running of ToPCC:



ToPCC has hardly any options for its use, so its interface is simpler than most of the other Acorn C/C++ tools.

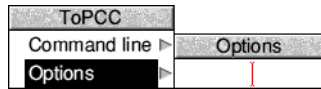
The **File** writable icon is for the name of the description file to be processed. If you displayed the **SetUp** dialogue box by clicking on the ToPCC icon bar icon, you will want to fill this in by dragging a source file from a directory display to this icon before running ToPCC.

Clicking Menu on the SetUp dialogue box brings up the ToPCC **SetUp** menu:



Command line shows you the command line that will be passed to the underlying ToPCC program; you can then alter it if necessary.

Options leads to a writable field in which you can specify one or more single letter options:

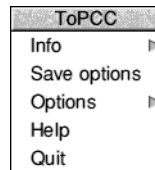


These options are:

- c Don't remove the keyword `const`
- e Don't remove `#error ...` directives
- l Don't include `#line ...` directives
- p Don't remove `#pragma ...` directives
- s Don't remove keyword `signed`
- t Don't remove the second argument to `va_start()`
- v Don't remove the keyword `volatile`

The icon bar menu

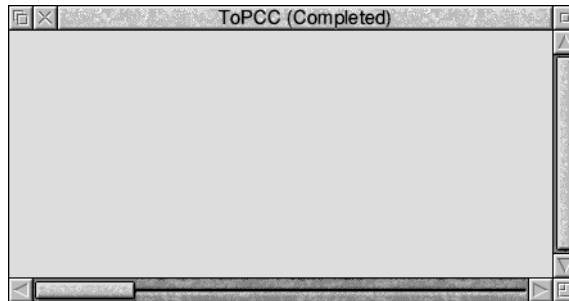
Clicking Menu on the ToPCC application icon on the icon bar gives access to the following options:



For a description of each option in the application menu see the chapter *General features* on page 103 of the accompanying *Desktop Tools* guide.

Example output

Running ToPCC displays any error messages in the standard text output window for non-interactive tools. If all goes well this window is empty:



As an example of using the tool ToPCC, open an empty SrcEdit text window and type the following example C source line in it:

```
int foo(float a);
```

Check that your `Wimp$Scrap` environment variable is set to a sensible file name, then save your new text file straight onto the ToPCC icon bar icon. Run ToPCC, then save the output text file straight onto the SrcEdit icon bar icon. The translated file looks like:

```
int foo(/* float a */);
```

Command line interface

For normal use you do not need to understand the syntax of the ToPCC command line, as it is generated automatically for you from the SetUp dialogue box setting before it is used.

The syntax of the ToPCC command line is:

```
topcc <options> <infile <outfile>>
```

options A minus '-' followed by one or more letters controlling individual features of the conversion; see page 72. As well as the options listed there, the -d option describes ToPCC, and the -help option gives the command line syntax and options.

infile Filename of the input C source or header text file, which defaults to stdin.

outfile Filename of the output C source or header text file, which defaults to stdout.

Part 2 – C language issues

5

C implementation details

Acorn C/C++ is a full implementation of the ANSI C89 language reference, also known as ISO C90. In addition, it provides many of the ISO C99 and C18 features and library functions. For information on the C standards see *The C standards* on page 7.

This chapter is split into parts, each of which details certain aspects of Acorn C's implementation.

- The first part – *Implementation details* on page 78 – gives details of those aspects of the compiler which the ISO/IEC 9899:1999 standard identifies as implementation-defined, and some other points of interest to programmers. They are grouped by subject; the section *Implementation limits* on page 85 lists the points required to be documented as set out in Annex E of the standard.

- The second part – *Standard implementation definition* on page 87 – discusses aspects of the compiler which are not defined by the C standard, but are implementation-defined and must be documented.

Annex J of the ISO/IEC 9899:1999 standard collects together information about portability issues; section J.3 lists those points which are implementation defined, and directs that each implementation shall document its behaviour in each of the areas listed. This part corresponds to appendix J.3, answering the points listed in the appendix, under the same headings and in the same order.

- The third part – *C99 features* on page 97 – describes the sub-set of C99 features that are implemented in the compiler.
- The fourth part – *C18 features* on page 112 – describes the sub-set of C18 features that are implemented in the compiler.
- The fifth part – *Extra features* on page 119 – describes some machine-specific features of the Acorn C compiler: `#pragma` directives, and special declaration keywords for functions and variables.

Implementation details

Identifiers

Identifiers can be of any length. They are truncated by the compiler to 256 characters, all of which are significant (the standard requires a minimum of 31).

The source character set expected by the compiler is 7 bit ASCII, except that within comments, string literals, and character constants, either the full ISO 8859-1 (Latin-1) 8 bit character set or UTF-8 are recognised, depending on the source character set mode (see *Pragmas controlling the source character set* on page 121).

At run time, the C library processes the full ISO 8859-1 8 bit character set, except that the default locale is the C locale (see the section *Standard implementation definition* on page 87). The `ctype` functions therefore all return 0 when applied to codes in the range 160–255. By calling `setlocale(LC_CTYPE, "ISO8859-1")` you can cause the `ctype` functions such as `isupper()` and `islower()` to behave as expected over the full 8 bit Latin alphabet, rather than just over the 7 bit ASCII subset.

Upper and lower case characters are distinct in all identifiers, both internal and external.

In `-pcc` and `-fc` modes an identifier may also contain a dollar character.

Data elements

The sizes of data elements are as follows:

Type	Size in bits	
char	8	
short	16	
int	32	
long	32	
long long	64	
float	32	
double	64	
long double	64	(subject to future change)
all pointers	32	

Integers are represented in two's complement form.

Data items of type `char` are unsigned by default, though they may be explicitly declared as `signed char` or `unsigned char`. (In `-pcc` mode there is no `signed` keyword, so `chars` are signed by default and may be declared unsigned if required.) Single-character constants are thus always positive.

Floating point quantities are stored in the IEEE format. In double and long double quantities, the word containing the sign, the exponent and the most significant part of the mantissa is stored at the lower machine address.

Limits: `limits.h` and `float.h`

The standard defines two header files, `limits.h` and `float.h`, which contain constant declarations describing the ranges of values which can be represented by the arithmetic types. The standard also defines minimum values for many of these constants.

The following table sets out the values in these two headers on the Arm, and a brief description of their significance. See the standard for a full definition of their meanings.

Number of bits in smallest object that is not a bit field (i.e. a byte):

`CHAR_BIT` 8

Maximum number of bytes in a multi-byte character, for any supported locale:

`MB_LEN_MAX` 1

Numeric ranges of integer types:

The middle column gives the numerical value of each range's endpoint, while the right hand column gives the bit patterns (in hexadecimal) that would be interpreted as this value in C. When entering constants you must be careful about the size and signed-ness of the quantity. Furthermore, constants are interpreted differently in decimal and hexadecimal/octal. See the ISO standard or any of the recommended textbooks on the C programming language for more details.

Range	End-point	Hex representation
<code>CHAR_MAX</code>	255	0xff
<code>CHAR_MIN</code>	0	0x00
<code>SCHAR_MAX</code>	127	0x7f
<code>SCHAR_MIN</code>	-128	0x80
<code>UCHAR_MAX</code>	255	0xff
<code>SHRT_MAX</code>	32767	0x7fff
<code>SHRT_MIN</code>	-32768	0x8000
<code>USHRT_MAX</code>	65535	0xffff

Range	End-point	Hex representation
INT_MAX	2147483647	0x7fffffff
INT_MIN	-2147483648	0x80000000
UINT_MAX	4294967295	0xffffffff
LONG_MAX	2147483647	0x7fffffff
LONG_MIN	-2147483648	0x80000000
ULONG_MAX	4294967295	0xffffffff
LLONG_MIN	-9223372036854775807	0x8000000000000000
LLONG_MAX	+9223372036854775807	0x7fffffffffffffff
ULLONG_MAX	18446744073709551615	0xffffffffffffffff

Characteristics of floating point:

FLT_RADIX	2
FLT_ROUNDS	1

The base (radix) of the Arm floating point number representation is 2, and floating point addition rounds to nearest.

Ranges of floating types:

FLT_MAX	3.40282347e+38F
FLT_MIN	1.17549435e-38F
DBL_MAX	1.79769313486231571e+308
DBL_MIN	2.22507385850720138e-308
LDBL_MAX	1.79769313486231571e+308
LDBL_MIN	2.22507385850720138e-308

Ranges of base two exponents:

FLT_MAX_EXP	128
FLT_MIN_EXP	(-125)
DBL_MAX_EXP	1024
DBL_MIN_EXP	(-1021)
LDBL_MAX_EXP	1024
LDBL_MIN_EXP	(-1021)

Ranges of base ten exponents:

FLT_MAX_10_EXP	38
FLT_MIN_10_EXP	(-37)
DBL_MAX_10_EXP	308
DBL_MIN_10_EXP	(-307)
LDBL_MAX_10_EXP	308
LDBL_MIN_10_EXP	(-307)

Decimal digits of precision:

FLT_DIG	6
DBL_DIG	15
LDBL_DIG	15

Digits (base two) in mantissa:

FLT_MANT_DIG	24
DBL_MANT_DIG	53
LDBL_MANT_DIG	53

Smallest positive values such that $(1.0 + x) \neq 1.0$:

FLT_EPSILON	1.19209290e-7F
DBL_EPSILON	2.2204460492503131e-16
LDBL_EPSILON	2.2204460492503131e-16L

Structured data types

The standard leaves details of the layout of the components of structured data types to each implementation. The following points apply to the Acorn C compiler:

- Structures are aligned on word boundaries.
- Structures are arranged with the first-named component at the lowest address.
- A component with a `char` type is packed into the next available byte.
- A component with a `short` type is aligned to the next even-addressed byte.
- All other arithmetic type components are word-aligned, as are pointers and `ints` containing bitfields.
- The only valid type for bitfields are `(signed) int`, `unsigned int` and `_Bool`. (In `-pcc` mode, `char`, `unsigned char`, `short`, `unsigned short`, `long` and `unsigned long` are also accepted.)

- A bitfield of type `int` is treated as unsigned by default (signed by default in `-pcc` mode).
- A bitfield must be wholly contained within the 32 bits of an `int`.
- Bitfields are allocated within words so that the first field specified occupies the lowest addressed bits of the word. (When configured *little-endian*, lowest addressed means least significant; when configured *big-endian*, lowest addressed means most significant.)

Packed Structures

A packed struct is one in which there is neither padding between fields to ensure the natural alignment of each field, nor trailing padding to ensure the natural alignment of a following struct within an array.

Many applications read data from and write data to networks and computer buses in formats defined by international standards and by other programs executing on different processors. The data format is fixed. Data read and data to be written can be precisely mapped in C using packed structs. However, packed structs cannot support reading values of the wrong endianness.

On the Arm, access to unaligned data can be expensive (taking up to 7 instructions and 2 extra work registers). Data accesses via packed structs should be minimised to avoid performance loss. Generally, internal data structures should not be packed.

There is no command-line option to change the default packing of structures. Packed structures must be specified with the type qualifier: `__packed`.

`__packed` behaves as a type qualifier (like `volatile`) and may qualify any non floating point type. Floating types may not be fields of packed structures.

A packed struct or union type must be declared explicitly. It is a different type from the corresponding non packed type and its packedness is an attribute of its struct tag (so `__packed` is more than just a type qualifier). Any variables declared using a packed tag automatically inherit the packed attribute, so `__packed` does not have to be specified:

```
__packed struct P { ... };

struct P pp;    /* pp is a packed struct */

In consequence, the following will be faulted:

struct Foo { ... };

__packed struct Foo PackedFoo;    /* illegal */

or
```

```
struct Foo { ... };
```

```
typedef __packed struct Foo PackedFoo;    /* illegal */
```

This ensures that a packed struct can never be assignment compatible with an unpacked struct. This could happen if `__packed` were merely a type qualifier like `volatile` and `const`.

Each field of a packed struct or packed union inherits the packed qualifier. There are no packed array types. A packed array is simply an array of objects of packed type (there is no inter-element padding).

The effect of casting away `__packed` is undefined. For example:

```
int f(__packed int *px)
{
    return *(int *)px;    /* undefined behaviour */
}
```

A struct (or union) sub-field of a packed struct or union must be declared to have packed struct (or packed union) type.

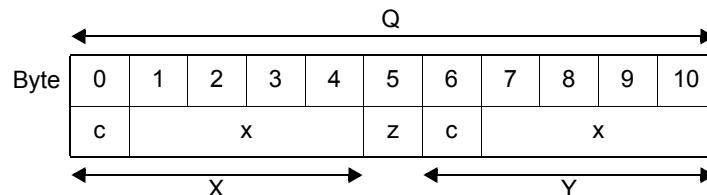
```
struct S {...};
__packed struct P {...};

struct T {
    struct S ss; /* OK */
    struct P pp; /* OK */
};

__packed struct Q {
    struct S ss; /* faulted - sub-structs must be packed */
    struct P pp; /* OK */
};
```

The sub-structs are abutted without any intermediate padding, and they contain no internal padding themselves (because they must be packed).

```
__packed struct P { char c; int x; };
__packed struct {
    struct P X;
    char z;
    struct P Y;
} Q;
```



Pointers

The following remarks apply to pointer types:

- Adjacent bytes have addresses which differ by one.
- The macro `NULL` expands to the value 0.
- Casting between integers and pointers results in no change of representation.
- The compiler warns of casts between pointers to functions and pointers to data (but not in `-pcc` mode).

Pointer subtraction

When two pointers are subtracted, the difference is obtained as if by the expression:

```
((int)a - (int)b) / (int)sizeof(type pointed to)
```

If the pointers point to objects whose size is no greater than four bytes, word alignment of data ensures that the division will be exact in all cases. For longer types, such as doubles and structures, the division may not be exact unless both pointers are to elements of the same array. Moreover the quotient may be rounded up or down at different times, leading to potential inconsistencies.

Arithmetic operations

The compiler performs all of the ‘usual arithmetic conversions’ set out in the standard.

The following points apply to operations on the integral types:

- All signed integer arithmetic uses a two’s complement representation.
- Bitwise operations on signed integral types follow the rules which arise naturally from two’s complement representation.
- Right shifts on signed quantities are arithmetic.
- Any value to be shifted is treated as a 32 bit value, unless of type `long long` in which case it is regarded as a 64 bit value.
- The remainder on integer division has the same sign as the divisor.
- If a value of integral type is truncated to a shorter signed integral type, the result is obtained by masking the original value to the length of the destination, and then sign extending.
- Conversions between integral types never causes an exception to be raised.
- Integer overflow does not cause an exception to be raised.
- Integer division by zero causes an exception to be raised.

The following points apply to operations on floating types:

- When a `double` or `long double` is converted to a `float`, rounding is to the nearest representable value.
- Conversions from floating to integral types cause exceptions to be raised only if the value cannot be represented in a `long int` (or `unsigned long int` in the case of conversion to an `unsigned int`).
- Floating point underflow is not detected; any operation which underflows returns zero.
- Floating point overflow causes an exception to be raised.
- Floating point divide by zero causes an exception to be raised.

Expression evaluation

The compiler performs the ‘usual arithmetic conversions’ (promotions) set out in the standard before evaluating any expression.

- The compiler may re-order expressions involving only associative and commutative operators of equal precedence, even in the presence of parentheses (e.g. $a + (b - c)$ may be evaluated as $(a + b) - c$).
- Between sequence points, the compiler may evaluate expressions in any order, regardless of parentheses. Thus the side effects of expressions between sequence points may occur in any order.
- Similarly, the compiler may evaluate function arguments in any order.
- Any detail of order of evaluation not prescribed by the standard may vary between releases of the Acorn C compiler.

Implementation limits

The C99 standard sets out certain minimum translation limits which a conforming compiler must cope with; you should be aware of these if you are porting applications to other compilers. A summary is given here. The ‘mem’ limit indicates that no limit is imposed other than that of available memory.

Description	Requirement	Acorn C
Nesting levels of blocks	127	mem
Nesting levels of conditional inclusion	63	mem
Pointer, array and function declarators modifying a basic type in a declaration	12	mem
Expressions nested by parentheses	63	mem

Description	Requirement	Acorn C
Significant characters		
in internal identifiers and macro names	63	256
in external identifiers	31	256
External identifiers in one source file	4095	mem
Identifiers with block scope in one block	511	mem
Macro identifiers in one source file	4095	mem
Parameters in one function definition/call	127	mem
Parameters in one macro definition/invoke	127	mem
Characters in one logical source line	4095	no limit
Characters in a string literal	4095	mem
Bytes in a single object	65535	mem
Nesting levels for <code>#include</code> files	15	mem
Case labels in a <code>switch</code> statement	1023	mem
Members in a single <code>struct</code> or <code>union</code> , enumeration constants in a single <code>enum</code>	1023	mem
Nesting of <code>struct/union</code> in a single declaration	63	mem

Standard implementation definition

Translation (J.3.1)

Diagnostic messages produced by the compiler are of the form

"source-file", line #: *severity*: *explanation*

where *severity* is one of

- *warning*: not a diagnostic in the C standard sense, but an attempt by the compiler to be helpful to you.
- *error*: a violation of the C specification from which the compiler was able to recover by guessing your intentions.
- *serious error*: a violation of the C specification from which no recovery was possible because the compiler could not reliably guess what you intended.
- *fatal* (for example, 'not enough memory'): not really a diagnostic, but an indication that the compiler's limits have been exceeded or that the compiler has detected a fault in itself.

Environment (J.3.2)

The mapping of a command line from the Arm-based environment into arguments to `main()` is implementation-specific. The shared C library supports the following:

- The arguments given to `main()` are the words of the *command line* (not including I/O redirections, covered below), delimited by white space, except where the white space is contained in double quotes. A white space character is any character of which `isspace` is true. (Note that the RISC OS Command Line Interpreter filters out some of these).
A double quote or backslash character (`\`) inside double quotes must be preceded by a backslash character. An I/O redirection will not be recognised inside double quotes.

The shared C library supports a pair of *interactive devices*, both called `:tt`, that handle the keyboard and the VDU screen:

- No buffering is done on any stream connected to `:tt` unless I/O redirection has taken place. If I/O redirection other than to `:tt` has taken place, full file buffering is used except where both `stdout` and `stderr` have been redirected to the same file, in which case line buffering is used.

Using the shared C library, the standard input, output and error streams, `stdin`, `stdout`, and `stderr` can be redirected at run-time in the ways shown below. For example, if `mycopy` is a compiled and linked program which simply copies the standard input to the standard output, the following line:

```
*mycopy < infile > outfile 2> errfile
```

runs the program, redirecting `stdin` to the file `infile`, `stdout` to the file `outfile` and `stderr` to the file `errfile`.

The following shows the allowed redirections:

<code>0< filename</code>	read <code>stdin</code> from <i>filename</i>
<code>< filename</code>	read <code>stdin</code> from <i>filename</i>
<code>1> filename</code>	write <code>stdout</code> to <i>filename</i>
<code>> filename</code>	write <code>stdout</code> to <i>filename</i>
<code>2> filename</code>	write <code>stderr</code> to <i>filename</i>
<code>2>&1</code>	write <code>stderr</code> to wherever <code>stdout</code> is currently going
<code>>& filename</code>	write both <code>stdout</code> and <code>stderr</code> to <i>filename</i>
<code>>> filename</code>	append <code>stdout</code> to <i>filename</i>
<code>>>& filename</code>	append both <code>stdout</code> and <code>stderr</code> to <i>filename</i>
<code>1>&2</code>	write <code>stdout</code> to wherever <code>stderr</code> is currently going

Identifiers (J.3.3)

256 characters are significant in identifiers without external linkage. (Allowed characters are letters, digits, and underscores.)

256 characters are significant in identifiers with external linkage. (Allowed characters are letters, digits, and underscores.)

Case distinctions are significant in identifiers with external linkage.

In `-pcc` and `-fc` modes, the character '\$' is also valid in identifiers.

Characters (J.3.4)

The characters in the source character set are either ISO 8859-1 (Latin-1 Alphabet), a superset of the ASCII character set, or UTF-8, depending on the source character set mode (see *Pragmas controlling the source character set* on page 121). In Latin-1 mode the printable characters are those in the range 32 to 126 and 160 to 255. Any printable character may appear in a string or character constant, and in a comment.

The Arm C library supports the ISO 8859-1 (Latin-1) character set, so the following points hold:

- The execution character set is identical to the source character set.
- There are four chars/bytes in an `int`. If the Arm processor is configured to operate with a *little-endian* memory system (as in RISC OS), the bytes are ordered from least significant at the lowest address to most significant at the highest address. If the Arm is configured to operate with a *big-endian* memory system, the bytes are ordered from least significant at the highest address to most significant at the lowest address.
- A character constant containing more than one character has the type `int`. Up to four characters of the constant are represented in the integer value. The first character contained in the constant occupies the lowest-addressed byte of the integer value; up to three following characters are placed at ascending addresses. Unused bytes are filled with the NULL (or `/0`) character.
- There are eight bits in a character in the execution character set.
- There are 32 bits in a wide character in the execution character set.
- All integer character constants that contain a single character or character escape sequence are represented in the source and execution character set.
- Characters of the source character set in string literals and character constants map identically into the execution character set.
- When the source character set is Latin-1 no locale is used to convert multi-byte characters into the corresponding wide characters (codes) for a wide character constant. When the source character set is UTF-8 wide character strings and constants are converted from the UTF-8 in the source to UCS-4.
- A plain `char` is treated as unsigned (but as signed in `-pcc` mode).
- Escape codes are:

Escape sequence	Char value	Description
<code>\a</code>	7	Attention (bell)
<code>\b</code>	8	Backspace
<code>\f</code>	12	Form feed
<code>\n</code>	10	Newline
<code>\r</code>	13	Carriage return
<code>\t</code>	9	Tab
<code>\unnnn</code>	<code>0x0000nnnn</code>	UCS character in hexadecimal
<code>\Unnnnnnnn</code>	<code>0xxxxxxxxx</code>	UCS character in hexadecimal
<code>\v</code>	11	Vertical tab
<code>\xnn</code>	<code>0xnn</code>	ASCII code in hexadecimal
<code>\nnn</code>	<code>0nnn</code>	ASCII code in octal

Integers (J.3.5)

The representations and sets of values of the integral types are set out in the section *Data elements* on page 78. Note also that:

- The result of converting an integer to a shorter signed integer, if the value cannot be represented, is as if the bits in the original value which cannot be represented in the final value are masked out, and the resulting integer sign-extended. The same applies when you convert an unsigned integer to a signed integer of equal length.
- Bitwise operations on signed integers yield the expected result given two's complement representation. No sign extension takes place.
- The sign of the remainder on integer division is the same as defined for the function `div()`.
- Right shift operations on signed integral types are arithmetic.

Floating point (J.3.6)

The representations and ranges of values of the floating point types have been given in the section *Data elements* on page 78. Note also that:

- When a floating point number is converted to a shorter floating point one, it is rounded to the nearest representable number.
- The properties of floating point arithmetic comply with IEC 559 which is equivalent to IEEE 754 and IEC 60559.
- The default state of the `FENV_ACCESS` and `FP_CONTRACT` pragmas is off.

Arrays and pointers (J.3.7)

The ISO standard specifies three areas in which the behaviour of arrays and pointers must be documented. The points to note are:

- The type `size_t` is defined as `unsigned int`.
- Casting pointers to integers and vice versa involves no change of representation. Thus any integer obtained by casting from a pointer will be positive.
- The type `ptrdiff_t` is defined as `(signed) int`.

Hints (J.3.8)

The register storage-class specifier

In the Acorn C compiler, you can declare any number of objects to have the storage class `register`. Depending on which variant of the Arm Procedure Call Standard is in use, there are between five and seven registers available. (There are six available in the default APCS variant, as used by RISC OS.) Declaring more than this number of objects with register storage class must result in at least one of them not being held in a register. It is advisable to declare no more than four. The valid types are:

- any integer type
- any pointer type
- any integer-like structure (any one word struct or union in which all addressable fields have the same address, or any one word structure containing only bitfields).

Note that other variables, not declared with the `register` storage class, may be held in registers for extended periods; and that `register` variables may be held in memory for some periods.

Note also that there is a `#pragma` which assigns a file-scope variable to a specified register everywhere within a compilation unit.

The inline function specifier

Any function that is declared with `inline` at the top-level before (or in) its definition will be inlined when used subsequently in that translation unit. Inlining does not occur if the `-g` or `-Ospace` options have been selected. An `inline` declaration is ineffective if first seen after the function definition. In the event of recursion during inline expansion, inlining stops after the first iteration.

Structures, unions, enumerations and bitfields (J.3.9)

The Acorn C compiler handles structures in the following way:

- When a member of a union is accessed using a member of a different type, the resulting value can be predicted from the representation of the original type. No error is given.
- Structures are aligned on word boundaries. Characters are aligned in bytes, shorts on even numbered byte boundaries and all other types, except bitfields, are aligned on word boundaries. Bitfields are sub-fields of `ints`, themselves aligned on word boundaries.

- A 'plain' bitfield (declared as `int`) is treated as unsigned `int` (signed `int` in `-pcc` mode).
- A bitfield which does not fit into the space remaining in the current `int` is placed in the next `int`.
- The order of allocation of bitfields within `ints` is such that the first field specified occupies the lowest addressed bits of the word.
- Bitfields do not straddle storage unit (`int`) boundaries.
- The integer type chosen to represent the values of an enumeration type is `int` (signed `int`).

Qualifiers (J.3.10)

An object that has `volatile`-qualified type is *accessed* if any word or byte of it is read or written. For `volatile`-qualified objects, reads and writes occur as directly implied by the source code, in the order implied by the source code.

The effect of accessing a `volatile`-qualified `short` is undefined.

Preprocessing directives (J.3.11)

A single-character constant in a preprocessor directive cannot have a negative value.

The standard header files are contained within the compiler itself. The mechanism for translating the standard suffix notation to a RISC OS filename is described in the chapter *CC and C++* on page 11.

Quoted names for includable source files are supported. The rules for directory searching are given in the chapter *CC and C++* on page 11.

The recognized `#pragma` directives and their meaning are described in the section *#pragma directives* on page 119.

The date and time of translation are always available, so `__DATE__` and `__TIME__` always give respectively the date and time.

Library functions (J.3.12)

The C library has or supports the following features:

- The macro `NULL` expands to the integer constant 0.
- If a program redefines a reserved external identifier, then an error may occur when the program is linked with the standard libraries. If it is not linked with standard libraries, then no error will be detected.

- The `assert()` function prints the following message:

```
*** assertion failed: expression, function fn, file filename, line line-no
```

and then calls the function `abort()`. If the program was compiled in C90 compatibility mode the function name is omitted.
- The functions `isalnum()`, `isalpha()`, `isblank()`, `isctrnl()`, `islower()`, `isprint()`, `isupper()` and `ispunct()` usually test only for characters whose values are in the range 0 to 127 (inclusive). Characters with values greater than 127 return a result of 0 for all of these functions, except `isctrnl()` which returns non-zero for 0 to 31, and 128 to 255.

After the call `setlocale(LC_CTYPE, "ISO8859-1")` the following statements also apply to character codes and affect the results returned by the *ctype* functions:

- codes 128 to 159 are control characters
- codes 192 to 223 except 215 are upper case
- codes 224 to 255 except 247 are lower case
- codes 160 to 191, 215 and 247 are punctuation

The mathematical functions return the following values on domain errors:

Function	Condition	Returned value
<code>log(x)</code>	$x \leq 0$	<code>-HUGE_VAL</code>
<code>log10(x)</code>	$x \leq 0$	<code>-HUGE_VAL</code>
<code>sqrt(x)</code>	$x < 0$	<code>-HUGE_VAL</code>
<code>atan2(x, y)</code>	$x = y = 0$	<code>-HUGE_VAL</code>
<code>asin(x)</code>	$\text{abs}(x) > 1$	<code>-HUGE_VAL</code>
<code>acos(x)</code>	$\text{abs}(x) > 1$	<code>-HUGE_VAL</code>

Where `-HUGE_VAL` is written above, a number is returned which is defined in the header `math.h`. Consult the `errno` variable for the error number.

The mathematical functions set `errno` to `ERANGE` on underflow range errors.

A domain error occurs if the second argument of `fmod` is zero, and `-HUGE_VAL` returned.

The set of signals for the generic `signal()` function is as follows:

<code>SIGABRT</code>	Abort
<code>SIGFPE</code>	Arithmetic exception
<code>SIGILL</code>	Illegal instruction
<code>SIGINT</code>	Attention request from user
<code>SIGSEGV</code>	Bad memory access
<code>SIGTERM</code>	Termination request
<code>SIGSTAK</code>	Stack overflow

The default handling of all recognised signals is to print a diagnostic message and call `exit`. This default behaviour applies at program start-up.

When a signal occurs, if `func` points to a function, the equivalent of `signal(sig, SIG_DFL)`; is first executed.

If the `SIGILL` signal is received by a handler specified to the `signal` function, the default handling is reset.

The C library also has the following characteristics relating to I/O:

- The last line of a text stream does not require a terminating newline character.
- Space characters written out to a text stream immediately before a newline character do appear when read back in.
- No null characters are appended to a binary output stream.
- The file position indicator of an append mode stream is initially placed at the end of the file.
- A write to a text stream does not cause the associated file to be truncated beyond that point.
- The characteristics of file buffering are as intended by section 4.9.3 of the standard.
- A zero-length file (on which no characters have been written by an output stream) does exist.
- The validity of filenames is defined by the host computer's filing system.
- The same file can be opened many times for reading, but only once for writing or updating. A file cannot however be open for reading on one stream and for writing or updating on another.

Note also the following points about library functions:

<code>remove()</code>	Cannot remove an open file.
<code>rename()</code>	The effect of calling the <code>rename()</code> function when the new name already exists is dependent on the host filing system. Not all renames are valid: examples of invalid renames include <code>("net:file1", "net:\$.file2")</code> and <code>("net:file1", "adfs:file2")</code> .
<code>fprintf()</code>	Prints <code>%p</code> arguments in hexadecimal format (lower case) as if a precision of 8 had been specified. If the variant form (<code> %#p</code>) is selected, the number is preceded by the character <code>@</code> .
<code>fscanf()</code>	Treats <code>%p</code> arguments identically to <code>%x</code> arguments. Always treats the character <code>-</code> in a <code>%[</code> argument as a literal character.

<code>ftell()</code> and <code>fgetpos()</code> <code>perror()</code>	Set <code>errno</code> to the value of <code>EDOM</code> on failure. Generates the following messages: <table> <tr> <th>Error:</th><th>Message:</th></tr> <tr> <td>0</td><td>No error (<code>errno</code> = 0)</td></tr> <tr> <td>EDOM</td><td>EDOM – function argument out of range</td></tr> <tr> <td>ERANGE</td><td>ERANGE – function result not representable</td></tr> <tr> <td>ESIGNUM</td><td>ESIGNUM – illegal signal number to <code>signal()</code> or <code>raise()</code></td></tr> <tr> <td>EILSEQ</td><td>EILSEQ - character encoding error</td></tr> <tr> <td>EOVERFLOW</td><td>EOVERFLOW - too large for data structure</td></tr> <tr> <td>EFBIG</td><td>EFBIG - data written to file lost due to exceeding file size limit</td></tr> <tr> <td>others</td><td>Error code number has no associated message</td></tr> </table>	Error:	Message:	0	No error (<code>errno</code> = 0)	EDOM	EDOM – function argument out of range	ERANGE	ERANGE – function result not representable	ESIGNUM	ESIGNUM – illegal signal number to <code>signal()</code> or <code>raise()</code>	EILSEQ	EILSEQ - character encoding error	EOVERFLOW	EOVERFLOW - too large for data structure	EFBIG	EFBIG - data written to file lost due to exceeding file size limit	others	Error code number has no associated message
Error:	Message:																		
0	No error (<code>errno</code> = 0)																		
EDOM	EDOM – function argument out of range																		
ERANGE	ERANGE – function result not representable																		
ESIGNUM	ESIGNUM – illegal signal number to <code>signal()</code> or <code>raise()</code>																		
EILSEQ	EILSEQ - character encoding error																		
EOVERFLOW	EOVERFLOW - too large for data structure																		
EFBIG	EFBIG - data written to file lost due to exceeding file size limit																		
others	Error code number has no associated message																		
<code>aligned_alloc()</code> , <code>calloc()</code> , <code>malloc()</code> and <code>realloc()</code>	If the size of the area requested is zero, non-NULL is returned under RISC OS 3.10 and later.																		
<code>abort()</code> <code>exit()</code> , <code>quick_exit()</code>	Closes all open files, and deletes all temporary files. The status returned by <code>exit</code> is the same value that was passed to it. For a definition of <code>EXIT_SUCCESS</code> and <code>EXIT_FAILURE</code> refer to the header file <code>stdlib.h</code> .																		
<code>getenv()</code>	Returns the value of the named RISC OS Environmental variable, or NULL if the variable had no value. For example: <pre>root = getenv ("C\$libroot"); if (root == NULL) root = "\$.arm.clib";</pre>																		
<code>system()</code>	Used either to CHAIN to another application or built-in command or to CALL one as a sub-program. When a program is chained, all trace of the original program is removed from memory and the chained program invoked. If a program is called (which is the default if no CHAIN: or CALL: precedes the program name – a change from Release 2), the calling program and data are moved in memory to somewhere safe and the callee loaded and started up. The return value from the <code>system()</code> call is -2 (indicating a failure to invoke the program) or the value of <code>Sys\$ReturnCode</code> set by the called program (0 indicates success).																		
<code>strerror()</code>	The error messages given by this function are identical to those given by the <code>perror()</code> function.																		

<code>timespec_get()</code>	The only defined epoch is <code>TIME_UTC</code> .
<code>clock()</code>	Returns the time taken by the program since its invocation, as indicated by the host's operating system.

Architecture (J.3.13)

For details of the values assigned to the macros in `<float.h>`, `<limits.h>` and `<stdint.h>` see the respective header file.

The order and encoding of bytes in any object is selected by the choice of little-endian or big-endian as described on page 44. The number of bytes in any object, which also determines the value of the result of the `sizeof` operator, is listed in *Data elements* on page 78, and not repeated here.

Whether any extended alignments are supported, the contexts in which they are supported, and valid alignment values (other than those returned by an `_Alignof` expression for fundamental types) are given in *Variable alignment* on page 114.

C99 features

The current compiler supports the majority of features from the ISO C standard (ISO/IEC 9899:1999) known as C99. This section summarises the C99 features that are implemented – it includes most of the C99 compiler features and some of the C99 library functions.

These C99 features are unavailable when the C90 or PCC compiler options are used.

If any of the C99 library functions, long long variables or variable length arrays are used it is essential that the SharedCLibrary 5.43 or later is loaded as described in *Ensuring the necessary components are present* on page 324. It is also necessary to use the C99 mode and generate code that uses the APCS-32 standard as these functions in the C library will not work reliably in APCS-R mode (C99 and APCS-32 are the default options for the compiler but may be overridden by command line or menu options).

This section is not intended to be a tutorial on the C99 language. For more details refer to a C99 programmer's guide or the C99 standard – see section *The C standards* on page 7.

C99 features implemented in the compiler

Boolean data type

C99 introduces the new `bool` data type, together with the constants `true` and `false` via the `stdbool.h` header file. This header file may only be used when using C99 mode (i.e. it is not available if using C90 or pcc modes).

For example:

```
#include <stdbool.h>
bool myflag = true;
struct flags { bool a:1; bool b:1 };
```

64-bit integers

The new C99 integer type `long long` is implemented using 64-bit integers and `long long` literals are indicated via the `ll` or `LL` suffix (the `LL` suffix is recommended as `ll` may be mis-read as eleven). Unsigned 64-bit integer literals are indicated via the suffix `ull` or `ULL`.

The new format length modifier `ll` may be used with conversion specifiers `d`, `i`, `o`, `u`, `x`, `X` to indicate that the type is `long long`.

For example:

```
#include <stdlib.h>
#include <stdio.h>
```

```
unsigned long long int MSB_set = 1ULL <<63;
printf("MSB_set is %lld in decimal\n", MSB_set);
printf("or %llx in hexadecimal\n", MSB_set);
```

New library functions for long long integers include `llabs`, `lldiv`, `strtoll` and `strtoull` in `stdlib.h`, and `llrint` and `llround` in `math.h`.

C++ style comments

Comments may now use the C++ syntax where the rest of the line following `//` is regarded as a comment.

Interleaved statements and declarations

Statements and declarations can now be interleaved, allowing declarations to be grouped with relevant statements.

```
foo();
int b = 5;
bar();
```

Declarations in for statements

Loop variables may now be declared in a `for` statement. For example:

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

Variable-length arrays

Variable-length arrays, varying in multiple dimensions, may be passed as function parameters. For example:

```
void fred(int w, int h, char d[h][w])
{
    for (int y=0; y<h; y++)
        for (int x=0; x<w; x++)
            d[y][x] ^= 0x80;
}
```

Similarly arrays may be declared using an expression evaluated at run time, to avoid using an explicit `malloc()`. For example:

```
void iconsprites(const char *filename)
{
    char buffer[sizeof "IconSprites "+strlen(filename)];
    sprintf(buffer, "IconSprites %s", filename);
    _kernel_oscli(buffer);
}
```

Flexible array members

The last element of a structure may be an array of unspecified size and this is referred to as a *flexible array member*:

```
struct flex
{
    int len;
    int data[];
};
struct flex *p=malloc(sizeof(struct flex)+10*sizeof(int));
for (int i=0; i<10; i++) p->data[i] = i;
```

The object pointed to by `p` behaves as if it had been declared as:

```
struct flex
{
    int len;
    int data[10];
} *p;
```

Flexible array members have incomplete type, but the structure containing them does not and so the `sizeof` operator may not be applied to the array, but it may be applied to the structure, as in the above example.

Structures with flexible array members cannot themselves be members of other structures or array elements.

The assignment:

```
struct flex *s = *p;
```

will only copy `len` – it will not copy the flexible array member `data`.

Flexible array members may only appear as the last member of a struct that is otherwise non-empty.

The behaviour is undefined if an element of the flexible array member is accessed outside of the structure. ie. in the above example accessing `p->data[10]` would have undefined results.

Designated initialisers

Structure elements can now be initialised by name. For example:

```
enum { ZERO, ONE, TWO, THREE };
char num[] = { [ONE] = 1, [THREE] = 3 };
```

Nested structures can use the same feature, as follows:

```
struct mystruct person =
{
    .sex = MALE,
    .dob = { .day=14, .month = 6, .year=1965},
    .name.christian = "Paul"
};
```

Type qualifiers and 'static' inside function array parameters

Type qualifiers and static declarations are now allowed in function array parameters. For example:

```
void fred(char a[static 4], char b[const]);
```

is equivalent to:

```
void fred(char *a, char * const b);
```

with a guarantee that a points to at least 4 characters.

Non-constant structure initialisers

Automatic structure initialisers can now use expressions. For example:

```
struct mystruct a = { x + y, foo() };
```

This does not apply to static structure initialisers.

Name of current function

`__func__` is pre-defined as the name of the current function (a character string literal). For example:

```
printf("Crash in function %s\n", __func__);
```

Trailing comma allowed in enum declaration

Trailing commas are allowed in enum declarations. For example:

```
enum { ZERO, ONE, TWO, };
```

This may be useful when using macros to generate the strings.

New modifiers for printf, scanf and strftime

In addition to the %ll modifier mentioned above, modifiers are now available for outputting floating point numbers in hexadecimal format and `size_t` and `intmax_t` types. For example:

```
size_t a; intmax_t i; long long l; float f;
printf("a=%zu, i=%jd, l=%lld, f=0x%a\n", a, i, l, f);
```

The `strftime` function has been extended to provide the ISO 8601 week-based year number. This is useful in conjunction with the week number, where the days of each year preceding the first Monday of the year (if any) are regarded as belonging to the previous year for the purpose of the week number. For example, Saturday 2nd January 1999 is in week 53 and week-based year 1998.

%g is replaced by the last 2 digits of the week-based year as a decimal number (00-99). %G is replaced by the week-based year as a decimal number (e.g., 1998).

For example:

```
strftime(buffer, len, "ISO 8601 week number %G/%g", &tm);
```

snprintf and vsnprintf

```
int snprintf(char * restrict s, size_t n, const char*
restrict format, ...);
```

```
int vsnprintf(char * restrict s, size_t n, const char*
restrict format, va_list arg);
```

The new `snprintf` and `vsnprintf` functions in `<stdio.h>` work like `sprintf` and `vsprintf` respectively but `n` specifies the length of the buffer. Up to `n-1` characters are written to the buffer and the string is always terminated with a null. If `n` is zero, nothing is written and `s` may be a null pointer. `snprintf` returns the number of characters that would have been written, excluding the null, had there been no restriction on the length or a negative value if an encoding error occurs.

(See *Restricted pointers* on page 103 for an explanation of the `restrict` keyword).

For example:

```
char buffer[20];
len = snprintf(buffer, sizeof buffer, "Result is %s", s);
if (len >= sizeof buffer) printf("Buffer too small\n");
```

vscanf, vfscanf and vsscanf

```
int vscanf(const char * restrict format, va_list arg);  
int vfscanf(FILE * restrict stream, const char * restrict  
format, va_list arg);  
int vsscanf(const char * restrict s, const char * restrict  
format, va_list arg);
```

The new `vscanf`, `vfscanf` and `vsscanf` functions in `<stdio.h>` work like `scanf`, `fscanf`, and `sscanf` respectively, and the return values are the same, except instead of being called with a variable number of arguments, they are called with an argument list as defined in the `<stdarg.h>` header. The `va_start` macro should have been called to set this up before the function call.

(See *Restricted pointers* on page 103 for an explanation of the `restrict` keyword).

Macros with a variable number of arguments

Macros may be defined with a variable number of arguments in the same way as functions. They are also referred to as variadic macros.

If the argument list ends with `...` then the trailing arguments, including any separating commas, are merged to form a single item – the *variable arguments*.

The `__VA_ARGS__` identifier is used to indicate where the variable arguments should be inserted. For example:

```
#define dprintf(...) fprintf(stderr, __VA_ARGS__)
```

Inline functions

See *The inline function specifier* on page 91.

Compound literals

Compound literals are created using the notation:

```
( typename ) { initialiser-list }
```

For example:

```
typedef struct point { int x, y; } point;  
void draw(point);  
draw( (const point) { 2, 3 } );
```

New and updated header files

The new C99 header files are:

<code><fenv.h></code>	Floating point environment, including macros for floating point exceptions.
<code><stdbool.h></code>	Defines <code>bool</code> type and macros for <code>true</code> and <code>false</code> . See <i>Boolean data type</i> on page 97
<code><stdint.h></code>	Defines macros for integer data types with fixed or minimum number of bits. For example <code>int8_t</code> may be used to declare 8 bit signed integer variables; <code>uint_fast16_t</code> may be used to declare unsigned 16 bit integer variables using the fastest suitable data type available in the compiler.
<code><inttypes.h></code>	Defines format specifiers for use with the integer types defined in <code><stdint.h></code> . It also includes <code><stdint.h></code> .
<code><iso646.h></code>	Defines suitable macros to enable C source code to be written using limited character sets.

The following header files have some new definitions:

<code><math.h></code>	Some new functions.
<code><float.h></code>	Additional floating point characteristics.
<code><stdio.h></code>	The <code>snprintf</code> and <code>vscanf</code> family of functions.

See *The C library* on page 135 or the header files for more details.

Hexadecimal floating-point formats

Hexadecimal floating point constants can now be used. The letter 'p' is used to indicate the exponent as a power of 2. For example:

```
double a = 0x0.8p0; // 0.5 in decimal
```

```
double a = 0x1.8p4; // 24.0 in decimal
```

The conversion specifiers `a` and `A` can be used in `printf` and `scanf` type functions for input and output of hexadecimal floating point numbers. On output the conversion specifier `a` is used to generate lower case hexadecimal digits and `A` is used to generate upper case hexadecimal digits.

Restricted pointers

The `restrict` qualifier may be used with pointers to indicate to the compiler that an object accessed through the pointer has a special association with the pointer. The programmer is guaranteeing that all accesses to the object use the value of

that pointer, either directly or indirectly. This allows the compiler to optimise the generated code as it knows that no other pointer variables can refer to the same object.

For example:

```
// a and b are guaranteed to point to different
// objects, allowing better optimisation:
void fred(char *restrict a, char *restrict b);
```

This qualifier is purely for optimisation – removing the `restrict` qualifier should not change the program's behaviour.

The `restrict` qualifier permits extra optimisation in the Common Subexpression Elimination (CSE) phase, by limiting the number of potential pointer aliases.

The use of `restrict` does not actually guarantee any optimisations, and there are many circumstances in which the `restrict` qualifier will be “lost” during the compilation process, removing any potential benefit (although still generating correct code). For example, only stand-alone restricted pointers are tracked; restricted pointers that are members of a structure are not.

The compiler does not carry out all possible optimisations that `restrict` implies (just as it does not carry out all possible optimisations generally).

The following examples show code snippets that do compile more efficiently because of the addition of the `restrict` keyword.

Restricted pointers to const-qualified types

```
void g(void);

int f(const int data[restrict]) // equivalent to f(const int * restrict data)
{
    int r = data[1];
    g();
    r += data[1];
    return r;
}
```

Without the `restrict` keyword, `data[1]` would have to be reloaded after the call to `g()`, as the compiler does not know whether `data` points to something that is really constant, so `g()` could potentially change it.

In this example, the `restrict` tells the compiler that nothing the function accesses through `data` can change, so the compiler will not reload `data[1]` after `g()`. It is not permitted to modify an object accessed through a restricted pointer to a const-qualified type during the lifetime of that pointer. So in this example, the

`data[1]` must not change throughout the call to `f()`. If the call to `g()` actually did modify `data[1]`, this would invoke undefined behaviour. On the other hand, `g()` can modify `data[0]`, because `f()` doesn't access it.

Restricted pointers cannot alias directly-accessed objects

```
int i;

int f(int * restrict p)
{
    int r = i;
    *p = 4;
    r += i;
    return r;
}
```

Without the `restrict` keyword, `i` would have to be reloaded after writing to `p`, as it is possible that `f` was called as `f(&i)`;

If an object is accessed through a restricted pointer, and is modified (by any means), then all accesses to that object, during the lifetime of block the pointer is declared in, must be through that pointer, else you invoke undefined behaviour. In this example, this assures the compiler that `p` cannot point to `i`: the object `*p` is modified through a restricted pointer, so it knows that the references to `i` (which are not through that pointer) cannot be accessing the same object.

Another example:

```
char * restrict data;

void f(void)
{
    for (int i = 0; i < 100; i++)
        data[i] = 0;
}
```

Without the `restrict` qualifier, the `data` pointer would have to be reloaded every iteration of the loop.

This optimisation is only performed for restricted pointers with file scope or outermost function block, as the compiler cannot cope with cases like:

```
int i;

void f(void)
{
    int r = i;
    {
        int * restrict p = /* something */;
        *p = 4;
    }
    r += i;
    return r;
}
```

The restricted pointer `p` can legitimately point at `i`, as `i` is not accessed in `p`'s block, so `i` must be reloaded after that block.

Restricted pointers with the same scope cannot alias each other

```
int f(int * restrict p, int * restrict q)
{
    int r = *p;
    *q = 5;
    r += *p;
    return r;
}
```

Without `restrict`, `q` could equal `p` on entry, so the write to `*q` could change `*p`, so `*p` would need to be loaded twice.

With the addition of `restrict` to both pointers, the compiler knows that the same location cannot be accessed through both pointers.

At present, this optimisation only handles pointers with file scope, or in the outermost block of a function (which includes parameters), due to full scope information not being available in the CSE phase.

The scope rules are necessary, because assignments to restricted pointers in an inner scope are permitted:

```
int f(int * restrict p)
{
    int r = *p;
    {
        int * restrict p1 = p;
        *p1 = 2;
    }
    r += *p; // *p has changed from original value
    return r;
}
```

However, this is not permitted:

```
int f(int * restrict p)
{
    int r = *p;
    int * restrict p1 = p; // p1 has same scope as p - undefined behaviour
    *p1 = 2;
    r += *p;
    return p;
}
```

so the compiler can legally assume that the write to `*p1` didn't change to `*p`.

Conversely, this is allowed:

```
int f(int * restrict p)
{
    int r = *p;
    int *q = p;
    *q = 5;
    r += *p;
    return p;
}
```

Because it is legal to assign a restricted pointer to a non-restricted one like this, and the compiler does not perform data flow analysis for `restrict`, it is forced to assume that any non-restricted pointer can alias a restricted one. As a result, in the original example, both parameters `p` and `q` must be `restrict`-qualified to enable optimisation. Without restricting both, the programmer could legally change `q` to point to `p` after entry, and the compiler cannot deduce whether this has happened.

Pre-defined macros

The following macros are pre-defined to enable conditional compilation dependent on the compiler version.

The macro `__STDC__` is defined as 1 in C90 and later ISO standard modes. Future versions may define it as 2 or 3 etc.

The macro `__STDC_HOSTED__` is defined as 1 in C99 modes.

The macro `__STDC_VERSION__` is defined as 199901L in C99 mode, but is undefined in C90 mode. Note that this would normally indicate that the compiler is fully compliant with the ISO/IEC 9899:1999 specification. This is not the case, but since many of the C99 features are implemented it was decided that it is better to define this macro than not.

The macros `__STDC_IEC559__` and `__STDC_IEC559_COMPLEX__` are undefined. Although the compiler's floating point arithmetic **is** IEC 559 compliant, it does not meet all of the extra requirements in the IEC559 annex of the C99 standard.

The macro `__STDC_ISO_10646__` is defined as `200401L`. This indicates that `wchar_t` values are viewed as being ISO/IEC 10646 UCS character codes. As such, Universal Character Names in wide character constants or strings always map directly to the `wchar_t` with the same value.

Standard pragmas

The following three C99 standard pragmas are recognised:

```
#pragma STDC FENV_ACCESS OFF/ON/DEFAULT
#pragma STDC FP_CONTRACT OFF/ON/DEFAULT
#pragma STDC CX_LIMITED_RANGE OFF/ON/DEFAULT
```

The default for all three pragmas is `OFF`, but only `FENV_ACCESS` currently has any effect; with it off, the compiler can assume that floating-point operations have no side-effects, and ignores any possibility of exceptions or traps. With it on, care is taken to ensure that all arithmetic occurs (as if) at run-time. For example

```
double fred(void)
{
    return 1.0 / 3.0;
}
```

will compile to

```
fred
    LDFD    f0, =0.3333333333333333
    MOV     pc, lr
```

but

```
#include <fenv.h>

double fred(void)
{
    #pragma STDC FENV_ACCESS ON
    return 1.0 / 3.0;
}
```

will compile to

```
fred
    MVFD    f0, #1
    DVFD    f0, f0, #3
    MOV     pc, lr
```

to ensure that the inexact flag gets set (and with a view to rounding mode altering the result). The compiler will still fold expressions that can be determined to raise no exceptions (e.g. `1.0 / 2.0`). Compile-time evaluation can also be forced thus:


```
double fred(void)
{
    #pragma STDC FENV_ACCESS ON
    static const double one_third = 1.0 / 3.0;
    return one_third;
}
```

Other optimisations, particularly CSE, are effectively inhibited on floating-point arithmetic if `FENV_ACCESS` is on.

In C90 mode, the compiler functions as before, and only considers overflow, invalid operation and divide by zero exceptions as significant. Thus `1.0 / 0.0` would be performed at run-time, but `1.0 / 3.0` would not. For other optimisations, behaviour is somewhere in-between the two C99 states.

Digraphs

These six tokens:

```
<:      :>      <%      %>      %:      %:%:
```

behave, respectively, the same as the six tokens:

```
[      ]      {      }      #      ##
```

This is to enable C source code to be written using restricted character sets where these characters are unavailable. For example:

```
%:include <stdio.h>
char a<:2:> = <% 1, 3 %>;
```

Universal character names

The compiler now accepts C99 universal character names (UCNs) in string and character literals. UCNs are sequences of the form `\uxxxx` or `\Uxxxxxxxx` giving the hexadecimal code of an ISO/IEC 10646 UCS (Unicode) character. They can be used in strings or identifiers, as a source and execution character set-independent way of specifying characters:

```
char *pound = "\u00A3"; // equivalent to Latin-1 char *pound = "£";
int fran\u00E7ais;      // equivalent to Latin-1 int français;
```

The compiler currently only supports them in string and character literals.

C99 restrictions implemented in the compiler

Implicit int type no longer assumed

Previous versions of C assumed that untyped functions and function arguments were of integer type, although the compiler generated a warning. In C99 untyped functions and function arguments are regarded as errors.

Return

Functions declared as void are not allowed to use `return` with an expression, whilst non-void functions must use `return` with an expression.

Other C99 features

Other C99 features implemented in the compiler include:

- Reliable integer division
- Preprocessor arithmetic is done using `intmax_t` or `uintmax_t`
- Partial support for new block scopes for selection and iteration statements
- Integer constant type rules
- Integer promotion rules
- Improved arithmetic is IEC 60559 compliant but it does not meet all of the extra requirements required in the IEC 60559 annex of the C99 standard
- Idempotent type qualifiers (e.g. `const const int x` is equivalent to `const int x`)
- Empty macro arguments
- `va_copy` macro
- Complex number support using the `complex.h` header file
- Type-generic math macros (using the `tgmath.h` header file)

C99 non-compliances

The following C99 features are not implemented in the current version of the compiler:

- Wide character library support (using the `wchar.h` and `wctype.h` header files). The functions in these header files have not been implemented.
- `_Pragma` preprocessing operator

Implementation notes

`_Bool` is an 8-bit type; any value other than 0 or 1 is a trap representation.

Unlike `char`, `short` or `float`, `_Bool` is 'caller-narrowed' in function calls.

Variable-length arrays use the new library functions `__rt_allocauto` and `__rt_freeauto`. Their prototypes are:

```
void *__rt_allocauto(size_t);  
void __rt_freeauto(void *);
```

In the future these could be improved to ensure variable length arrays are freed correctly in `longjmp`-type situations, by associating blocks with stack chunk and stack pointer values.

`long long` uses a host of new run-time support functions – the functions used are compatible with the ones used by the Arm ADS compiler. The compiler inlines many `long long` operations. If you specify `-cpu 4` or similar, it will inline some multiplications (`signed * signed` → `long long`, `unsigned * unsigned` → `long long`, and `unsigned * long long` → `long long`) using the `UMULL` and `SMULL` instructions which are available on StrongARM processors or later.

Compatibility issues

There should be very few problems in compiling existing code using the C99 compiler. The most likely problem is existing code may use implicit int declarations. Another possible problem is that existing code may use keywords that are now reserved. These are `inline`, `restrict`, `_Bool`, `_Complex` and `_Imaginary`.

If problems are encountered the C99 features may be disabled by selecting ISO C90 in the `cc` menu or by specifying `-c90` on the `cc` command line.

C18 features

The current compiler supports the majority of features from the new ISO C standard (ISO/IEC 9899:2018) known as C18. This document summarises the C18 features that are implemented – it includes most of the C18 compiler features and some of the C18 library functions.

These C18 features are unavailable when the C99, C90 or PCC compiler options are used.

If any of the C18 library functions are used it is essential that the SharedCLibrary 6.05 or later is loaded as described in *Ensuring the necessary components are present* on page 324.

This section is not intended to be a tutorial on the C18 language. For more details refer to a C18 programmer's guide or the C18 standard – see section *The C standards* on page 7.

C18 features implemented in the compiler

Static assertions

The compiler, after the preprocessing stage has completed, will evaluate a constant expression for truth. If the result is false the assertion fails and the message printed out as a compilation error. You can use static assertions to check that assumptions your program makes hold true:

```
#include <assert.h>
#include "netinet/in.h"

struct sockaddr_in adr;

static_assert(sizeof(adr.sin_addr) == sizeof(uint32_t), "Bad sock size");
```

They can be used to check all types of compile time generated expressions, including enums:

```
#include <assert.h>

typedef enum
{
    RED,
    GREEN,
    BLUE
} colours_t;

static_assert(RED + 2 == BLUE, "Colours not in order");
```

A static assertion may be placed anywhere where a declaration is permitted.

Non returning functions

Functions that are not intended to return can be marked with a `noreturn` qualifier.

```
#include <stdnoreturn.h>

static noreturn void hangloop(int unused)
{
    while (1) {}
    /* Not reached */
    for (int i = 0; i < 1000; i++)
        unused++;
}
```

Functions marked `noreturn` will be checked for paths that could return, resulting in a warning if a return is determined to be possible. In addition, any unreachable code found in such functions will be eliminated.

Complex type initialisation

Complex number variables can now be initialised using the `CMPLXF`, `CMPLX`, `CMPLXL` macros. For example:

```
#include <complex.h>

static const float complex slope = CMPLXF(1.234, 5.678);
```

Floating point limits

C18 introduces some extra limits in addition to those already found in `<float.h>`.

<code>FLT_HAS_SUBNORM</code>	conveys the implementation's handling of subnormals.
<code>FLT_DECIMAL_DIGITS</code>	declares the digits which can be encoded losslessly.
<code>FLT_TRUE_MIN</code>	declares the absolute minimum value that can be encoded.

For double variables change the `FLT` prefix to `DBL` in each of the above defines, and for long double variables use the prefix `LDBL` instead.

Alternate string literal encodings

Three extra quoted string prefixes are available in C18, in addition to the L prefix used to encode wide strings made up of arrays of `wchar_t`.

The prefix `u8` will encode the quoted string in UTF-8.

```
char *encoded = u8"This car cost £1000"; /* As a UTF-8 string */
```

The prefix `u` makes a string of `char16_t`, `U` makes a string of `char32_t`, which are typically useful for Unicode UCS-2 and UCS-4 manipulation respectively.

```
#include <uchar.h>

char16_t string[] = u"Storage for up to 65536 code points per character";
char32_t widest[] = U"Fully 32 bits per character";
```

Variable alignment

The alignment of a variable type can be queried using the `alignof` operation.

```
#include <stdalign.h>

printf("Structure has %u byte alignment\n", alignof(struct bbox));
```

Because the type being queried can be declared at compile time, after preprocessing, the `alignof` operation is particularly useful when used in conjunction with `static_assert` (see page 112).

In addition automatic variables can be given a type qualifier to cause the compiler to *overalign* them. That is, require the variable to have more strict alignment than they would naturally. The `alignas` qualifier can be applied either when the variable is declared, or when a member is placed inside a structure definition.

```
#include <stdalign.h>

typedef struct
{
    char        c;
    alignas(2) uint8_t x, y; /* At offsets +2 +4 */
} text;

int foo(void)
{
    alignas(4) char a, b;
    return (int)&b - (int)&a; /* Will be a multiple of 4 */
}
```

The `alignas` specifier may only be applied to automatic variables at present due to limitations imposed by other parts of the tool chain, for example when producing an AIF image or relocatable module. The maximum alignment possible using `alignas` for an automatic variable is implicitly determined by the current procedure calling standard, see appendix *ARM procedure call standard* on page 297 of the *Desktop Tools* guide.

The new type `max_align_t`, defined in `<stddef.h>`, is an object whose alignment is the greatest fundamental alignment supported by the compiler.

Generic type selectors

To enable type generic function selection, such as maths functions where a choice of float/double/long double is typically available, the C18 standard has added the `_Generic` keyword.

```
#define mycos(r) _Generic((r), long double: cosl, \
                          default: cos, \
                          float: cosf
                          )(r)

float result = mycos(0.707f); /* Chooses the cosf function */
```

Notice how a default case can be used in addition to specifically mentioned types.

Anonymous structures and unions

Nested intermediate unions and structures can now be left with no name, creating them anonymously, and allowing them to be referenced shorthand. This is most often useful with unions (as in C++) since a structure could just as easily have been written flattened.

```
typedef struct
{
    int a;
    union
    {
        int b;
        float c;
    }; /* No name here makes it anonymous */
    struct
    {
        int x, y, z;
    }; /* Also possible, an anonymous struct */
    struct
    {
        int d;
    } tail;
} mixture;

mixture m;

m.a = 7;
m.c = 1.414f;
m.d = 57; /* Invalid, d's struct has a name, so m.tail.d = 57 */
printf("Float as hex %08X\n", m.b);
```

This feature can be temporarily enabled in C99 and earlier modes by wrapping the structure definition in an `anon_unions` pragma as described on page 122.

Exclusive access mode for fopen

The `fopen()` function now has an exclusive access mode, 'x'. This only applies to 'w' modes (wx, wbx, w+ wb+, w+b) and the 'x' must be the last letter of the mode string. Opens the file in exclusive mode, guaranteeing that no other program has the file open for writing at the same time.

New and updated header files

The new C18 header files are:

<code><stdalign.h></code>	Defines the <code>alignof</code> and <code>alignas</code> keywords as macros.
<code><stdnoreturn.h></code>	Defines the <code>noreturn</code> keyword as a macro.

The following header files have some new definitions:

<code><assert.h></code>	Defines the <code>static_assert</code> keyword as a macro.
<code><complex.h></code>	Adds <code>CMPLXF</code> , <code>CMPLX</code> , <code>CMPLXL</code> macros for float, double, long double initialisation.
<code><float.h></code>	Various extra precision limits.
<code><stdio.h></code>	Removal of <code>gets</code> function.
<code><stdlib.h></code>	The new <code>aligned_alloc()</code> , <code>quick_exit()</code> and <code>at_quick_exit()</code> function prototypes.
<code><time.h></code>	The new <code>timespec_get()</code> and <code>timespec</code> structure definition, along with the UTC time base <code>TIME_UTC</code> .

See *The C library* on page 135 or the header files for more details.

Duplicate type definitions

There can be multiple `typedef` declarations which declare the same type in the same compilation unit, provided the types declared are equivalent. This can be used to help reduce header file ordering issues where two header files may need to have the same definition in.

Pre-defined macros

The following macros are pre-defined to enable conditional compilation dependent on the compiler version.

The macro `__STDC__` is defined as 1 in C90 and later ISO standards. Future versions may define it as 2 or 3 etc.

The macro `__STDC_HOSTED__` is defined as 1 in C99 and later modes.

The macro `__STDC_VERSION__` is defined as 201710L in C18 mode, but is undefined in C90 mode. It is not a mistake that the date encoded is October 2017; this is when the standard was ratified, but it was not published until the following year.

The macros `__STDC_IEC559__` and `__STDC_IEC559_COMPLEX__` are undefined and `__STDC_ISO_10646__` is defined as 200401L, using the rationale given on page 107.

The macros `__STDC_NO_ATOMICS__` and `__STDC_NO_THREADS__` are defined because these two optional extensions, described in Annex B.16 and B.25 of the standard, are not currently supported.

Quick exits

The `quick_exit` function in `<stdlib.h>` performs only minimal environment clean up then causes a normal program termination via the `_Exit` library function. None of the functions registered with `atexit` are called, nor any signal handlers, and `quick_exit` does not itself return to the caller.

Use `at_quick_exit` to register any functions which will be called if `quick_exit` is.

Sub-second resolution time

The `timespec_get` function sets the interval pointed to by `ts` to hold the current calendar time based on the specified time base.

```
#include <time.h>
```

```
int timespec_get(struct timespec *ts, int base);
```

The only base supported by the SharedCLibrary is `TIME_UTC` which fills the `timespec` structure with the system time. Note that on RISC OS the system time is usually incremented at 100Hz, so fields in the `timespec` structure will use that accuracy also.

If the `timespec_get` function is successful it returns the nonzero value `base`; otherwise, it returns zero.

Withdrawal of gets

The `gets` function has been withdrawn from `<stdio.h>` in C18 because it was deemed a serious buffer overrun security risk.

C18 non-compliances

The unicode character library support (using the uchar.h header files) functions have not been implemented.

The normative bounds-checking (annex K) and analysability (annex L) parts of the standard are not supported.

Compatibility issues

There should be very few problems in compiling existing code using the C18 compiler. If problems are encountered the C18 features may be disabled by selecting ISO C90 or C99 in the cc menu or by specifying `-c90` or `-c99` on the cc command line.

ISO/IEC 9899:2018 clarifies the position of some aspects of the earlier standard ISO/IEC 9899:2011 but doesn't add or remove language features. There is no C11 mode available.

Extra features

This section describes the following machine-specific features of the Acorn C compiler:

- `#warning` directive.
- `#pragma` directives.
- special declaration keywords for functions and variables.
- pre-defined macros.
- inline assembler.

#warning directive

Despite this pre-processor directive not being a formal part of any ISO C standard, it has wide support across other compilers in practice. The usage is similar to `#error`, for example:

```
#warning This is a warning
```

will result in cc emitting the line:

```
"myfile.c", line x: Warning: #warning encountered "This is  
a warning"
```

Because this is a non-standard extension, it is not permitted if you specify `-fussy` or `-strict`.

#pragma directives

Pragmas recognised by the compiler come in two forms:

```
#pragma -letter<digit>
```

and

```
#pragma <no_>feature-name
```

A short-form pragma given without a digit resets that pragma to its default state; otherwise to the state specified.

For example:

```
#pragma -s1  
#pragma no_check_stack  
  
#pragma -p2  
#pragma profile_statements
```

The set of pragmas recognised by the compiler, together with their default settings, varies from release to release of the compiler. The current list of recognised pragmas is:

Pragma name	Short form	Short 'No' form	Command line option
warn_implicit_fn_decls	a1 *	a0	-Wf
check_memory_accesses	c1	c0 *	-zpc0 1
warn_deprecated	d1 *	d0	-Wd
continue_after_hash_error (FP register variable)	e1 f1-f4	e0 *	
disable_fpargs_in_regs	g1	g0 *	
force_fpargs_in_regs	g2	g0 *	
include_only_once	i1	i0 *	
optimise_crossjump	j1 *	j0	-zpj0 1
optimise_schedule	l1 *	l0	-zpl0 1
optimise_multiple_loads	m1 *	m0	-zpm0 1
anon_unions	o1	o0 *	-zpo0 1
profile	p1	p0 *	-p
profile_statements (integer register variable)	p2 r1-r7	p0 *	-px
check_stack	s0 *	s1	-zps0 1
force_top_level	t1	t0 *	
utf8_source	u1	u0	-zpu0 1
check_printf_formats	v1	v0 *	
check_scanf_formats (internal use only)	v2 v3	v0 *	
check_swix_formats	v4	v0 *	
side_effects	y0 *	y1	
optimise_cse	z1 *	z0	-zpz0 1

In each case, the default setting is starred (except `utf8_source`, where the default depends on the current system alphabet).

You can also globally set pragmas by options set in the command line passed to the `cc` program (see the section *Command lines* on page 43); the preferred option to use is shown above. Where no option is shown for a pragma, it is because that pragma may only sensibly be used locally, and should be enabled/disabled around the particular program statements it is to affect.

Pragmas controlling the source character set

The pragma `utf8_source` determines the source character set. The compiler's default source character set is Latin-1, unless the current system alphabet is UTF8.

In Latin-1 mode some interpretation is placed on top-bit set characters; UCNs `\u0000` to `\u00FF` are translated to characters `\x00-\xFF`, others are translated to '?', with a warning. String literals are normally passed through without interpretation, to enable the production of code with character sets unknown to the compiler.

In UTF-8 mode, the source and execution multi-byte encoding is UTF-8. Wide character strings and constants are converted from the UTF-8 in the source to UCS-4. UCNs in multi-byte strings and character constants are converted to UTF-8 sequences. UTF-8 sequences are not normally interpreted by the compiler, except when it needs to convert to UCS-4.

Pragmas controlling the preprocessor

The pragma `continue_after_hash_error` in effect implements a `#warning` preprocessor directive. Pragma `include_only_once` asserts that the containing `#include` file is to be included only once, and that if its name recurs in a subsequent `#include` directive then the directive is to be ignored.

The pragma `force_top_level` asserts that the containing `#include` file should only be included at the top level of a file. A syntax error will result if the file is included, say, within the body of a function.

Pragmas controlling printf/scanf/_swi/_swix argument checking

The pragmas `check_printf_formats` is used to make the compiler check that the actual arguments to `printf`, `sprintf`, `fprintf` etc. are type-checked against the format designators in a literal format string. The compiler assumes that the second to last argument in the declaration is the format string and that the variable parameters follow.

The pragma `check_scanf_formats` is used to make the compiler check the actual arguments to `scanf` and `sscanf` are type-checked against the format designators in a literal format string. The compiler assumes that the second to last argument in the declaration is the format string and that the variable parameters follow.

The pragma `check_swix_formats` is used to make the compiler check that the actual number of arguments to `_swi` and `_swix` matches the number of arguments specified.

These pragmas affect all function declarations, including function prototypes, which occur while the pragma is active, regardless of the function name. They are enabled in the relevant libraries before the relevant function prototypes and disabled immediately afterwards to ensure that the parameters to the specified functions are all checked wherever possible.

Programmers may use these pragmas in their own code to make the compiler check the parameters passed to their own variable argument functions where the same parameter format is used.

For example:

```
#pragma check_printf_formats
int debug(char * /* format */, ...);
int debug_colour(int /* colour */, char *, ...);
#pragma no_check_printf_formats
```

Of course, calls using non-literal format strings cannot be checked.

Pragmas controlling anonymous structures and unions

The pragma `anon_unions` allows the use of the feature introduced with C18 to omit the name of unions, where no ambiguity exists, as a shorthand when accessing its members. The pragma name is chosen for compatibility with other compilers, and in fact allows both anonymous structures and unions when the compiler is being run in C99 (or earlier) language mode.

Pragmas controlling optimisation

The pragmas `optimise_crossjump`, `optimise_cse`, `optimise_multiple_loads`, `optimise_schedule`, and `side_effects` give fine control over where these optimisations are applied.

Cross jumping and common sub-expression elimination

For example, it is sometimes advantageous to disable cross-jumping (the *common tail* optimisation) in the critical loop of an interpreter; and it may be helpful in a timing loop to disable common sub-expression elimination. Note that the correct use of the `volatile` qualifier should remove most of the more obvious needs for this degree of control (and `volatile` is also available in the Acorn C compiler's `-pcc` mode unless `-strict` is specified).

Multiple loads

The compiler will try to combine a run of LDR instructions from sequential memory locations into a single LDM instruction when `optimise_multiple_loads` is enabled. There are some processor architectures where LDM will tie up the

memory bus atomically until the load multiple completes, which can affect interrupt and DMA performance, where it may be better to disable the optimisation to give other bus users opportunity to access memory.

Scheduling

The scheduler re-orders instructions to tune performance for the chosen CPU based on a simple model of which opcodes can be run in parallel, which take the longest to complete, and which cause interlocks to occur. Generally, some scheduling is preferable to none for all CPUs after ARM7, even if the model isn't perfect.

The `-cpu` switch selects which scheduling model is adopted. If that would cause unsupported instructions to be output for the target architecture use the `-arch` switch to select a backwards compatible architecture while still following the selected scheduling model.

Impure functions

By default, functions are assumed to be impure, so function invocations are not candidates for common sub-expression elimination. `Pragma noside_effects` asserts that the following function declarations (until the next `#pragma side_effects`) describe pure functions, invocations of which can be common sub-expressions. See also the section `__pure` on page 125.

Pragmas controlling code generation

Stack limit checking

The `pragma nocheck_stack` disables the generation of code at function entry which checks for stack limit violation. In reality there is little advantage to turning off this check: it typically costs only two instructions and two machine cycles per function call. The one circumstance in which `nocheck_stack` must be used is in writing a signal handler for the `SIGSTK` event. When this occurs, stack overflow has already been detected, so checking for it again in the handler would result in a fatal circular recursion.

Memory access checking

The `pragma check_memory_accesses` instructs the compiler to precede each access to memory by a call to the appropriate one of:

`__rt_rdnchk` where *n* is 1, 2, or 4, for byte, short, or long reads (respectively)
`__rt_wrnchk` where *n* is 1, 2, or 4, for byte, short, or long writes (respectively).

Global (program-wide) register variables

The pragmas f0-f4 and r0-r7 have no long form counterparts. Each introduces or terminates a list of `extern`, file-scope variable declarations. Each such declaration declares a name for the **same** register variable. For example:

```
#pragma r1                /* 1st global register */
extern int *sp;
#pragma r2                /* 2nd global register */
extern int *fp, *ap;      /* Synonyms */
#pragma r0                /* End of global declaration */
#pragma f1                /* 1st global FP register */
extern double pi;
#pragma f0                /* End of global declaration */
```

Any type that can be allocated to a register (see the section *Hints* (J.3.8) on page 91) can be allocated to a global register. Similarly, any floating point type can be allocated to a floating point register variable.

Global register r1 is the same as register v1 in the Arm Procedure Call Standard (APCS); similarly, r2 equates to v2, and so on. Depending on the APCS variant, between five and seven integer registers (v1-v7, machine registers R4-R10) and four floating point registers (F4-F7) are available as register variables. (There are six integer registers available in the default APCS variant, as used by RISC OS.) In practice it is probably unwise to use more than three global integer register variables and 2 global floating point register variables.

Provided the same declarations are made in each compilation unit, a global register variable may exist program-wide.

Otherwise, because a global register variable maps to a callee-saved register, its value will be saved and restored across a call to a function in a compilation unit which does not use it as a global register variable, such as a library function.

A corollary of the safety of direct calls out of a global-register-using compilation unit, is that calls back into it are dangerous. In particular, a global-register-using function called from a compilation unit which uses that register as a compiler allocated register, will probably read the wrong values from its supposed global register variables.

Currently, there is no link-time check that direct calls are sensible. And even if there were, indirect calls via function arguments pose a hazard which is harder to detect. This facility must be used with care. Preferably, the declaration of global register variable should be made in each compilation unit of the program. See also the section `__global_reg(n)` on page 125.

Special function declaration keywords

Several special function declaration options are available to tell the Acorn C compiler to treat that function in a special way. None of these are portable to other machines.

`__value_in_regs`

This allows the compiler to return a structure in registers rather than returning a pointer to the structure. For example:

```
typedef struct int64_structt
{
    unsigned int lo;
    unsigned int hi;
} int 64;

__value_in_regs extern int64 mul64(unsigned a, unsigned b);
```

See the appendix ARM *procedure call standard* on page 297 of the *Desktop Tools* guide for details of the default way in which structures are passed and returned.

`__pure`

By default, functions are assumed to be *impure* (i.e. they have side effects), so function invocations are not candidates for common sub-expression elimination. `__pure` has the same effect as `pragma noside_effects`, and asserts that the function declared is a *pure* function, invocations of which can be common subexpressions.

Special variable declaration keywords

`__global_reg(n)`

Allocates the declared variable to a global integer register variable, in the same way as `#pragma rn`. The variable must have an integral or pointer type. See also the section *Global (program-wide) register variables* on page 124.

`__global_freg(n)`

Allocates the declared variable to a global floating point register variable, in the same way as `#pragma fn`. The variable must have type float or double. See also the section *Global (program-wide) register variables* on page 124.

Note that the global register, whether specified by keyword or pragmas, must be declared in all declarations of the same variable. Thus:

```
int x;  
__global_reg(1) x;  
is an error.
```

Pre-defined macros

The following macros are pre-defined in this compiler.

__CC_NORCROFT

This is defined as 1 in this compiler which is based on the original Norcroft cc compiler. This macro may be used for conditional compilation where the source code is likely to be compiled using other compilers.

__CC_NORCROFT_VERSION

This symbol is defined as the compiler version number * 100 in all versions of cc from 5.11 onwards. For example, in cc v5.49 `__CC_NORCROFT_VERSION` has the value 549. See also the command line option `-ccversion version` on page 48.

__riscos

This is defined as 1 when the compiler is targetting RISC OS. This macro may be used for conditional compilation where the source code needs to use concepts that are unique to RISC OS, for example, polling the Window Manager.

__arm

This is defined as 1 when the compiler is targetting the Arm architecture. This is always the case for RISC OS.

Put together, these 4 pre-defined macros reflect the information given in the banner shown when the compiler starts:

```
Norcroft RISC OS Arm C vsn 5.49
```

The remaining macros relate to selections you make at the command line.

__APCS_32

This symbol is pre-defined when the APCS-32 variant of the Arm Procedure Calling Standard is in use. This is the default, but may be disabled by specifying the cc command line option `-apcs /26bit`.

__APCS_FPREGARGS

This symbol is pre-defined when the Floating Point register option of the Arm Procedure Calling Standard is in use. This is not normally used in RISC OS but may be enabled with the cc command line option `-apcs /fpregargs`.

__APCS_INTERWORK

This symbol is pre-defined when the Thumb interworking variant of the Arm Procedure Calling Standard is in use. This is not normally used in RISC OS but may be enabled with the cc command line option `-apcs /interwork` is used.

__APCS_NOFP

This symbol is pre-defined when the no frame pointer variant of the Arm Procedure Calling Standard is in use. This is not normally used in RISC OS but may be enabled with the cc command line option `-apcs /nofp`.

__APCS_NOSWST

This symbol is pre-defined when the no software stack checking variant of the Arm Procedure Calling Standard is in use. This is not normally used in RISC OS but may be enabled with the cc command line option `-apcs /noswst` is used.

__APCS_REENT

This symbol is pre-defined when the re-entrant variant of the Arm Procedure Calling Standard is in use. This applies when the cc command line option `-apcs /reentrant` is used.

The next group of macros follow the Arm C Language Extension, or ACLE, and further reflect selections made at the command line.

__ARM_ARCH

This symbol is set to the currently selected Arm architecture. For example ARMv5 would be 5. From ARMv8.1 onwards the representation changes to be 100× the major version plus the minor version, so ARMv8.1 would be 801.

__ARM_ARCH_ISA_THUMB

This symbol is set to the level of Thumb support on the currently selected CPU. It can take the value 1 or 2, or is unset if Thumb state is not available.

__ARM_ARCH_ISA_A64

Defined if the selected CPU has AArch64 instruction set capability.

__ARM_ARCH_ISA_ARM

Defined if the selected CPU has AArch32 instruction set capability.

__ARM_32BIT_STATE

Defined if the compiler is set to output code which targets the AArch32 instruction set.

Intrinsic functions

Intrinsic functions are a family of Arm architecture specific low level extensions to the C language. They are always available and do not require a header file to be included, but necessarily make the source code non-portable across other systems. The functions available are described below.

```
void __breakpoint(int arg);
```

Creates a hardware BKPT instruction at the point the intrinsic is used with *arg* encoded into it. It is an error to use anything other than a constant integer for *arg*. Note that BKPT will cause an undefined instruction prior to ARMv5, and a prefetch abort if there is no debugger attached.

```
unsigned int __current_pc(void);
```

Returns the value of the program counter at the point the intrinsic is used.

```
unsigned int __current_sp(void);
```

Returns the value of the stack pointer at the point the intrinsic is used.

```
void __nop(void);
```

Emits a single no operation instruction, and flushes the compiler's scheduler. The instruction will not be optimised away (unless it is unreachable). Note that the exact instruction used for a no operation may vary based on the selected architecture.

```
unsigned int __return_address(void);
```

Returns the value of the link register that would be used to return from the current function. Note that inlining and tail call optimisation may mean that the return address is actually that of the outer function.

```
void __schedule_barrier(void);
```

This intrinsic flushes the compiler's scheduler without any instructions being output. Compare this with the no operation intrinsic.

```
int __semihost(int reason, const void *arg);
```

Calls the *semihosting* SWI 0x123456 with *reason* and supplementary *arg* data. Semihosting is used for debugging between a target and monitoring system. For RISC OS this intrinsic should **not** be used as all SWI numbers must be allocated, and none is reserved for semihosting purposes.

Large file support

The shared C library includes extensions for large file support compatible with those defined in POSIX.1-2008 (often referred to as LFS extensions).

Most of the ISO standard C library function definitions are file size agnostic, using abstract data types. The exceptions to this are the functions `fseek` and `ftell`, which are specified in the C standard to use `long int` types, which are defined to be signed 32-bit numbers on RISC OS, as on most other 32-bit systems. These cannot be changed without breaking ISO conformance.

In order to support files larger than 2^{32} bytes long it is necessary to define versions of `fseek` and `ftell` which can use 64-bit offsets. These are named `fseeko` and `ftello`, and they use an extra abstract type, `off_t`, as the relative file pointer type.

These extensions are not enabled by default; they are controlled by the definition of the macros `_LARGEFILE_SOURCE`, `_LARGEFILE64_SOURCE` and `_FILE_OFFSET_BITS`. These macros may be defined either with a `-D` switch on the command line, or by using `#define` in your source file before you `#include <stdio.h>`.

Defining `_LARGEFILE_SOURCE` makes `fseeko`, `ftello` and `off_t` available. The size of `off_t` depends on the value of `_FILE_OFFSET_BITS`. This may be set to either 32 or 64, and this controls whether `off_t` is defined as a 32-bit or 64-bit type. If `_FILE_OFFSET_BITS` is not defined, `off_t` will default to being a 32-bit type.

Defining `_LARGEFILE64_SOURCE` does everything that `_LARGEFILE_SOURCE` does but in addition it defines the types `fpos64_t` and `off64_t`, and the functions `fgetpos64`, `fopen64`, `freopen64`, `fseeko64`, `fsetpos64`, `ftello64` and `tmpfile64`.

The way that these macros are defined means that to enable 64-bit file pointers in your software you have two options.

- 1 Selectable `off_t`. This will leave your source code largely untouched, except for using `fseeko` and `ftello` instead of `fseek` and `ftell`. Your program can be compiled to support either 32-bit or 64-bit file pointers by defining `_FILE_OFFSET_BITS` as required, but note that you have the responsibility to ensure that all the object files and static libraries linked in your program have been compiled using the same settings, or the results will be unpredictable.
- 2 Explicit `off_t`. This involves defining `_LARGEFILE64_SOURCE` and changing your source code to use the types `fpos64_t` and `off64_t`, and the functions `fgetpos64`, `fopen64`, `freopen64`, `fseeko64`, `fsetpos64`, `ftello64` and `tmpfile64`. Any file descriptor opened using the traditional `fopen`, `freopen` or `tmpfile` functions will be limited to 2GB-1 in size and any attempt to read, write or seek outside this range will result in failure, and `errno` will be set to `EOVERFLOW` or `EFBIG`.

Whichever option you choose, you will need to RMEnsure SharedCLibrary 5.64 in order to run the resulting binary.

Inline assembler

The inline assembler is a powerful feature allowing the programmer to easily access the full functionality of the Arm AArch32 instruction set within the C environment. Inline assembler is invoked by an `__asm` statement:

```
__asm
{
    ...
}
```

Assembler instructions are separated by newlines or semicolons. Within the assembler block, C comments and macros work as normal.

The assembler is not a “raw” assembler. The instructions within the `__asm` block are merged into the platform-independent pseudo-instructions generated by the surrounding C code, and are then subject to all the following compiler optimisations (e.g. common sub-expression elimination, dead code analysis, register allocation and peepholing) before being converted back into Arm code. The resulting object code is not guaranteed to be identical to the source assembly – think of it as an optimising assembler.

Within the assembler, physical Arm register names can be used, but these are not automatically bound to any C variables - it is not valid to, say, access `R0` in an attempt to inspect the first argument of your function. Instead, C expressions can be used in the place of registers. Physical registers only have scope within the assembler, and their use will constrain the compiler's register allocation – avoid where possible.

For example:

```
char *my_strcpy(char *dst, const char *src)
{
    char *orig_src;
    int c;
    __asm
    {
        MOV  orig_src, src
loop:
        LDRB c, [src], #1
        STRB c, [dst], #1
        TEQ  c, #0
        BNE  loop
    }
    return orig_src;
}

// returns old I bit
inline int disable_interrupts(void)
{
    int old_i, temp;
    __asm
    {
        MRS temp, CPSR
        AND old_i, temp, #0x80
        ORR temp, temp, #0x80
        MSR CPSR_c, temp
    }
    return old_i;
}
```

Note the powerful conjunction of `inline` and `__asm` in the last example. Also, because of the optimiser, if you do not use the return value of the function when it is inlined, the unneeded AND instruction will be eliminated by the compiler.

If you do use physical registers, you must explicitly transfer them to and from C variables:

```

size_t my_strlen(const char *s)
{
    size_t len;
    __asm
    {
        // the following instruction must be here to logically
        // transfer s into physical register a1. In the final
        // output, no MOV will actually be generated as s is
        // already in a1.
        MOV    a1, s

        MOV    a2, #0

    loop:
        LDRB   a3, [a1], #1
        TEQ    a3, #0
        ADDNE  a2, a2, #1
        BNE    loop

        // a2 must be transferred into a C variable so we can
        // return it.
        MOV    len, a2
    }
    return len;
}

```

Because (virtually) arbitrary expressions can be used, inline assembly can be considerably more expressive than normal assembly. For example:

```

SWP    test, 0, [&semaphore]

MOV    a[x][y][z], #0

MUL    x, x, #31

MOV    y, 1000000/x

MOV    R0, "Hello"

```

The full Arm AArch32 instruction set is supported, with the following notes and exceptions:

- B operates like a C goto - the target is a label. Labels can be placed in assembly the same as in C.
- BL, BLX, and SWI must specify the physical registers they use. This is done by specifying input, output and corrupted register lists as part of the instruction:

```

MOV    R0, #0x124
MOV    R1, sprite_area
MOV    R2, sprite_name
SWI    OS_SpriteOp, {R0-R2}, {R2}, {LR, PSR}
MOV    sprite_address, R2

```


For best practice, BL, BLX, and SWI are the only reason you should use physical registers. Any or all of the lists can be omitted. If they are omitted, such calls are assumed to have no input or output registers, and corrupt R0-R3 and R12, plus LR for BL and BLX.

Note that the inability to manipulate the stack rules out the ability to call APCS functions with more than 4 words of arguments using BL or BLX. To call a normal C function, it is usually better to switch back to C. Alternatively, use an expression:

```
ADD    len, strlen(fred), #1
```

This removes any dependencies on the calling standard.

- No other instructions that change the program counter can be used. BX is not supported.
- The stack cannot be used. Use C variables for storage, and if you do use physical registers, the compiler will preserve them around your code if necessary.
- FPA instructions (and hence registers) are not supported.
- Because the inline assembler does not support the FPA, BL, BLX, and SWI calls must not corrupt FPA registers.

6

The C library

The shared C library is a RISC OS relocatable module which can be used and shared by any applications which are resident in memory. It provides all the standard facilities of the language, as defined by the ISO standard document. Code using calls to the shared C library will be portable to other environments if an ISO compiler and library are available for that environment.

C and C++ programs are linked with a small piece of code and data called `Stubs`, which itself interfaces with the shared C library. The stubs contain your program's copy of the library's data, and an *entry vector* which allows your program to locate library routines in the C library module. `Stubs` is found in the directory `AcornC/C++.Export.APCS-32.Lib.CLib.o`.

Use of the shared C library:

- economises on RAM space when multiple C applications are running
- saves space on disc, benefiting users with single floppy disc drives
- makes programs load faster
- costs practically nothing at run time.

(For example, the Dhrystone benchmark runs just as quickly using the shared C library as when linked stand-alone with `ANSILib`.)

Without the shared C library, it would not be possible to pack so much into Acorn C/C++.

Compatibility Issues

Although the shared C library is normally located in the RISC OS ROM it will be necessary to load a new version if the ROM version is out of date. In particular, the shared C library in RISC OS 4.29 and earlier versions does not support 32-bit code or any of the new features described in *C99 features* on page 97.

See the section *Ensuring the necessary components are present* on page 324 for details on testing and loading the necessary software modules.

assert.h

The `assert` macro puts diagnostics into programs. When it is executed, if its argument expression is false, it writes information about the call that failed (including the text of the argument, the name of the function, the name of the source file, and the source line number, the last three of these being, respectively, the values of the pre-defined string `__func__` and the preprocessing macros `__FILE__` and `__LINE__`) on the standard error stream. It then calls the `abort` function. If its argument expression is true, the `assert` macro returns no value.

In C18 language mode, assertions can be added that are evaluated at compile time rather than when the program is executed. The `static_assert` macro checks that the argument expression is true and abandons compilation if not, printing the second argument for diagnosis.

If `NDEBUG` is `#defined` prior to inclusion of `assert.h`, calls to `assert` expand to null statements. This provides a simple way to turn off the generation of diagnostics selectively.

Note that `assert.h` may be included more than once in a program with different settings of `NDEBUG`.

complex.h

The complex number library provides versions of the trigonometric functions found in `math.h` which instead take complex number arguments.

The C99 language features complex numbers which define a number space with both real and imaginary planes. The real number component is simply a floating point type as in earlier versions of the C language and the imaginary number component is defined in this header file.

The macro `imaginary` expands to the compiler's native type specifier `_Imaginary`, and the macro `complex` expands to the type specifier `_Complex`. The macros `_Imaginary_I` and `_Complex_I` are constant expressions of the imaginary and complex unit definitions, respectively.

Complex variables may be initialised using the `CMPLXF`, `CMPLX`, `CMPLXL` macros for float, double, and long double precisions respectively.

For the full list of complex maths functions consult the `complex.h` header file directly, or consider using the type generic maths functions described in *tgmath.h* on page 201.

ctype.h

`ctype.h` declares several functions useful for testing and mapping characters. In all cases the argument is an `int`, the value of which is representable as an unsigned `char` or equal to the value of the macro `EOF`. If the argument has any other value, the behaviour is undefined.

<code>int isalnum(int c)</code>	Returns true if <code>c</code> is alphabetic or numeric
<code>int isalpha(int c)</code>	Returns true if <code>c</code> is alphabetic
<code>int isblank(int c)</code>	Returns true if <code>c</code> is a blank character used to separate words in the current locale
<code>int iscntrl(int c)</code>	Returns true if <code>c</code> is a control character (in the ASCII locale)
<code>int isdigit(int c)</code>	Returns true if <code>c</code> is a decimal digit
<code>int isgraph(int c)</code>	Returns true if <code>c</code> is any printable character other than space
<code>int islower(int c)</code>	Returns true if <code>c</code> is a lower-case letter
<code>int isprint(int c)</code>	Returns true if <code>c</code> is a printable character (in the ASCII locale this means 0x20 (space) → 0x7E (tilde) inclusive).
<code>int ispunct(int c)</code>	Returns true if <code>c</code> is a printable character other than a space or alphanumeric character
<code>int isspace(int c)</code>	Returns true if <code>c</code> is a white space character viz: space, newline, return, linefeed, tab or vertical tab
<code>int isupper(int c)</code>	Returns true if <code>c</code> is an upper-case letter
<code>int isxdigit(int c)</code>	Returns true if <code>c</code> is a hexadecimal digit, ie in 0...9, a...f, or A...F
<code>int tolower(int c)</code>	Forces <code>c</code> to lower case if it is an upper-case letter, otherwise returns the original value
<code>int toupper(int c)</code>	Forces <code>c</code> to upper case if it is a lower-case letter, otherwise returns the original value

errno.h

This file contains the definition of the macro `errno`, which is of type `volatile int`. It contains six macro constants defining the error conditions listed below.

EDOM

If a domain error occurs (an input argument is outside the domain over which the mathematical function is defined) the integer expression `errno` acquires the value of the macro `EDOM` and `HUGE_VAL` is returned. `EDOM` may be used by non-mathematical functions.

ERANGE

A range error occurs if the result of a function cannot be represented as a double value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the macro `HUGE_VAL`, with the same sign as the correct value of the function; the integer expression `errno` acquires the value of the macro `ERANGE`. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero; the integer expression `errno` acquires the value of the macro `ERANGE`. `ERANGE` may be used by non-mathematical functions.

ESIGNUM

If an unrecognised signal is caught by the default signal handler, `errno` is set to `ESIGNUM`.

EILSEQ

A multibyte character encoding error, as a result of an illegal sequence, results in `errno` being set to `EILSEQ`.

E_OVERFLOW

The value of `errno` is set to `E_OVERFLOW` when an operation is performed where one or more fields of a data structure are not large enough to hold the values required. For example, attempting to use `fopen` on a file whose size is larger than that which can be held in an object of type `off_t`.

EFBIG

When data is lost due the file exceeding the file size limit for the called function the value of `errno` is set to `EFBIG`.

fenv.h

The header `fenv.h` declares two types and several macros and functions to provide access to the floating-point environment. The environment includes any flags and status to describe the underlying implementation, and allows C programmers to query this environment and save/restore it as necessary.

The type `fenv_t` represents the entire floating-point environment. Variables of this type can be initialised to the default environment by assignment to `FE_DFL_ENV`.

The type `fexcept_t` represents the floating-point status collectively, including any status the implementation associates with the flags.

The exception flags can be one of the following:

<code>FE_DIVBYZERO</code>	divide by zero
<code>FE_INEXACT</code>	inexact result
<code>FE_INVALID</code>	invalid result
<code>FE_OVERFLOW</code>	overflow
<code>FE_UNDERFLOW</code>	underflow

The macro `FE_ALL_EXCEPT` is a mask of all exceptions defined by the current implementation. These exception defines are used in conjunction with `feclearexcept`, `feraiseexcept`, and `fesetexceptflag` to change the current floating point exception status, and are also the values returned by `fegetexceptflag`. To determine if individual exceptions are pending `fetestexcept` can be called which returns the current state of those exceptions passed to it, equivalent to a bitwise AND operation.

The rounding behaviour per IEC 60559 can be one of the following:

<code>FE_DOWNWARD</code>	round down
<code>FE_UPWARD</code>	round up
<code>FE_TONEAREST</code>	round to nearest integer
<code>FE_TOWARDZERO</code>	round towards zero

These rounding defines are passed to `fesetround` and returned by `fegetround` to determine the rounding policy of the floating point environment.

The environment as a whole can be read, set, restored, and put on hold using the supporting functions `fegetenv`, `fesetenv`, `feupdateenv`, `feholdexcept`.

float.h

This file contains a set of macro constants which define the limits of computation on floating point numbers. These are discussed in the chapter C *implementation details* on page 77.

inttypes.h

The `inttypes.h` library header file is part of the C99 language extensions and defines a group of `fprintf` and `fscanf` style format specifiers for each of the integer types defined in its partner header file `stdint.h`, which it includes itself.

Each group of macros follows the pattern *function fmt-specifier type-options bits* where *function* is either `PRI` for the `fprintf` family or `SCN` for the `fscanf` family and *type-options* may be omitted or one of `LEAST`, `FAST`, `MAX` or `PTR`. The value of *bits* can be either 8, 16, 32, or 64.

These macros are needed to make `fprintf` and `fscanf` format strings portable across systems that use different widths for their integer types. Consider the following example:

```
int32_t apple_count = 70000;
printf("There are %d apples\n", apple_count);
```

The `%d` specifier prints the value of an `int` type, so this code would work on systems like RISC OS where `int` is 32-bits wide, but would fail when compiled for a processor that used 16-bit integer types. Using the `inttypes.h` macros allows this to be portable across any C99 compliant compiler by changing the `printf` statement to:

```
printf("There are " PRIi32 "apples\n", apple_count);
```

For `fprintf` signed types there are:

<code>PRIdn</code>	<code>PRIdLEASTn</code>	<code>PRIdFASTn</code>	<code>PRIdMAX</code>	<code>PRIdPTR</code>
<code>PRIn</code>	<code>PRiLEASTn</code>	<code>PRiFASTn</code>	<code>PRiMAX</code>	<code>PRiPTR</code>

For `fprintf` unsigned types there are:

<code>PRIon</code>	<code>PRIoLEASTn</code>	<code>PRIoFASTn</code>	<code>PRIoMAX</code>	<code>PRIoPTR</code>
<code>PRiun</code>	<code>PRiULEASTn</code>	<code>PRiUFASTn</code>	<code>PRiUMAX</code>	<code>PRiUPTR</code>
<code>PRIx</code>	<code>PRiXLEASTn</code>	<code>PRiXFASTn</code>	<code>PRiXMAX</code>	<code>PRiXPTR</code>
<code>PRIX</code>	<code>PRIXLEASTn</code>	<code>PRIXFASTn</code>	<code>PRIXMAX</code>	<code>PRIXPTR</code>

For `fscanf` signed types there are:

<code>SCNd</code>	<code>SCNdLEASTn</code>	<code>SCNdFASTn</code>	<code>SCNdMAX</code>	<code>SCNdPTR</code>
<code>SCNi</code>	<code>SCNiLEASTn</code>	<code>SCNiFASTn</code>	<code>SCNiMAX</code>	<code>SCNiPTR</code>

For `fscanf` unsigned types there are:

<code>SCNo</code>	<code>SCNoLEASTn</code>	<code>SCNoFASTn</code>	<code>SCNoMAX</code>	<code>SCNoPTR</code>
<code>SCNu</code>	<code>SCNuLEASTn</code>	<code>SCNuFASTn</code>	<code>SCNuMAX</code>	<code>SCNuPTR</code>
<code>SCNx</code>	<code>SCNxLEASTn</code>	<code>SCNxFASTn</code>	<code>SCNxMAX</code>	<code>SCNxPTR</code>

Eleven macros are defined by this header file which provide alternate spellings of some of the integer operators.

Alternate spelling	Equivalent to
and	&&
and_eq	&=
bitand	&
bitor	
compl	~
not	!
not_eq	!=
or	
or_eq	=
xor	^
xor_eq	^=

For example it is equivalent to write:

```
if (x != y) printf("x and y differ");  
  
and  
  
#include <iso646.h>  
if (x not_eq y) printf("x and y differ");
```

limits.h

This set of macro constants determines the upper and lower value limits for integral objects of various types, as follows:

Object type	Object size
Byte (number of bits)	8
Multibyte character (number of bytes)	1
Object type	Range of values that can be contained
Signed char	-128 to 127
Unsigned char	0 to 255
Char	0 to 255
Short int	-0x8000 to 0x7fff
Unsigned short int	0 to 65535
Int	(~0x7fffffff) to 0x7fffffff
Unsigned int	0 to 0xffffffff
Long int	(~0x7fffffff) to 0x7fffffff
Unsigned long int	0 to 0xffffffff
Long long int	(~0x7fffffffffffffff) to 0x7fffffffffffffff
Unsigned long long int	0 to 0xffffffffffffffff

Note that the long long types are only implemented when C99 language features are enabled in the compiler, they are absent in C90 mode. See also the chapter *C implementation details* on page 77.

locale.h

This file handles national characteristics, such as the different orderings month-day-year (USA) and day-month-year (UK).

```
char *setlocale(int category, const char *locale)
```

Selects the appropriate part of the program's locale as specified by the `category` and `locale` arguments. The `setlocale` function may be used to change or query the program's entire current locale or portions thereof. Locale information is divided into the following types:

<code>LC_COLLATE</code>	string collation
<code>LC_CTYPE</code>	character type
<code>LC_MONETARY</code>	monetary formatting
<code>LC_NUMERIC</code>	numeric string formatting
<code>LC_TIME</code>	time formatting
<code>LC_ALL</code>	entire locale

The locale string specifies which locale set of information is to be used. For example,

setlocale

```
setlocale(LC_MONETARY, "uk")
```

would insert monetary information into the `lconv` structure. To query the current locale information, set the `locale` string to null and read the string returned.

lconv

```
struct lconv *localeconv(void)
```

Sets the components of an object with type `struct lconv` with values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale. The members of the structure with type `char *` are strings, any of which (except `decimal_point`) can point to "", to indicate that the value is not available in the current locale or is of zero length. The members with type `char` are non-negative numbers, any of which can be `CHAR_MAX` to indicate that the value is not available in the current locale. The members included are described above.

`localeconv` returns a pointer to the filled in object. The structure pointed to by the return value will not be modified by the program, but may be overwritten by a subsequent call to the `localeconv` function. In addition, calls to the `setlocale` function with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` may overwrite the contents of the structure.

This file contains the prototypes for commonly required mathematical functions, as well as a number of classification functions.

Function**Returns**

<code>double acos(double x)</code>	arc cosine of x . A domain error occurs for arguments not in the range -1 to 1
<code>double asin(double x)</code>	arc sine of x . A domain error occurs for arguments not in the range -1 to 1
<code>double atan(double x)</code>	arc tangent of x
<code>double atan2(double x, double y)</code>	arc tangent of y/x
<code>double cos(double x)</code>	cosine of x (measured in radians)
<code>double sin(double x)</code>	sine of x (measured in radians)
<code>double tan(double x)</code>	tangent of x (measured in radians)
<code>double acosh(double x)</code>	hyperbolic arc cosine of x . A domain error occurs for arguments not in the range -1 to 1
<code>double asinh(double x)</code>	hyperbolic arc sine of x . A domain error occurs for arguments not in the range -1 to 1
<code>double atanh(double x)</code>	hyperbolic arc tangent of x
<code>double cosh(double x)</code>	hyperbolic cosine of x
<code>double sinh(double x)</code>	hyperbolic sine of x
<code>double tanh(double x)</code>	hyperbolic tangent of x
<code>double exp(double x)</code>	base e exponential function of x (e^x)
<code>double exp2(double x)</code>	base 2 exponential function of x (2^x)
<code>double expm1(double x)</code>	base e exponential function of x , minus 1 ($e^x - 1$)
<code>double frexp(double x, int *exp)</code>	the value x , such that x is a double with magnitude in the interval 0.5 to 1.0 or zero, and value equals x times 2 raised to the power $*exp$
<code>double ldexp(double x, int exp)</code>	x times 2 raised to the power of exp
<code>double log(double x)</code>	natural logarithm of x
<code>double log10(double x)</code>	log to the base 10 of x
<code>double log1p(double x)</code>	natural logarithm of $x+1$
<code>double log2(double x)</code>	log to the base 2 of x

<code>double logb(double x)</code>	exponent of <i>x</i> in the floating point radix
<code>double modf(double x, double *iptr)</code>	signed fractional part of <i>x</i> . Stores integer part of <i>x</i> in object pointed to by <i>iptr</i> .
<code>double scalbn(double x, int n)</code>	efficient <i>x</i> raised to the power of <i>n</i>
<code>double scalbln(double x, long int n)</code>	efficient <i>x</i> raised to the power of <i>n</i>
<code>double cbrt(double x)</code>	cube root of <i>x</i>
<code>double fabs(double x)</code>	absolute value of <i>x</i>
<code>double hypot(double x, double y)</code>	hypotenuse of <i>x</i> and <i>y</i>
<code>double pow(double x, double y)</code>	<i>x</i> raised to the power of <i>y</i>
<code>double sqrt(double x)</code>	positive square root of <i>x</i>
<code>double erf(double x)</code>	error function of <i>x</i>
<code>double erfc(double x)</code>	complementary error function of <i>x</i>
<code>double lgamma(double x)</code>	natural logarithm of the absolute value of the gamma of <i>x</i>
<code>double tgamma(double x)</code>	gamma of <i>x</i>
<code>double ceil(double x)</code>	smallest integer not less than <i>x</i> (ie rounding up)
<code>double floor(double x)</code>	largest integer not greater than <i>x</i> (ie rounding down)
<code>double nearbyint(double x)</code>	the nearest integer to <i>x</i>
<code>double rint(double x)</code>	exception raising form of <code>nearbyint</code>
<code>double round(double x)</code>	round <i>x</i> away from zero
<code>double trunc(double x)</code>	round <i>x</i> to nearest smaller integer
<code>double fmod(double x, double y)</code>	floating-point remainder of <i>x/y</i>
<code>double remainder(double x, double y)</code>	remainder of <i>x/y</i> for IEC 60559
<code>double copysign(double x, double y)</code>	magnitude of <i>x</i> with the sign of <i>y</i>
<code>double nan(const char * tagp)</code>	as for <code>strtod(tagp, NULL)</code>
<code>double nextafter(double x, double y)</code>	next value after <i>x</i> in direction <i>y</i>
<code>double nexttoward(double x, long double y)</code>	next value after <i>x</i> in direction <i>y</i>
<code>double fmax(double x, double y)</code>	larger of <i>x</i> and <i>y</i>
<code>double fmin(double x, double y)</code>	smaller of <i>x</i> and <i>y</i>
<code>double fma(double x, double y, double z)</code>	value of <i>x*y+z</i>

The C99 language extensions implement long double and float versions of each of these mathematics functions, using the pattern *type function[suffix](type, type...)* where *type* determines the *suffix* at compile time: an 'l' for the long double variant; omitted for the conventional double; and an 'f' for the float variant. For example:

```
double fmax(double x, double y)
```

is listed, therefore it can be inferred that

```
long double fmaxl(long double x, long double y) and  
float fmaxf(float x, float y)
```

are also defined. The type generic maths facilities provided by `tgmath.h` described on page 201 can be used to automatically select the appropriately named function based on the types of the variables used, reducing the chances of programming errors.

The macro `FP_FAST_FMA` indicates whether using the `fma` function will be faster than writing out the multiply and add longhand. `FP_FAST_FMAL` and `FP_FAST_FMAF` describe the long double and float variants similarly.

The C99 language extensions include 19 extra utility functions:

- The function `fpclassify` permits classification of floating point values held in real floating point variables, returning one of the classifications `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO`.
- Maths analogues to the `ctype.h` query functions called `isinf`, `isnan`, and `isnormal`.
- Comparison functions called `isgreater`, `isgreaterequal`, `isless`, `islessequal`, `islessgreater` and `isunordered` which perform the equivalent relational operator but without raising floating point exceptions in the case that their arguments are unordered.
- A means to query the sign of a floating point value with `signbit`.
- Rounding functions which return integral results unlike their similarly named floating point returning functions: `rint` has counterpart `lrint` and `llrint`; `round` has counterpart `lround` and `llround`.
- Natural logarithm functions which return integral results unlike their similarly named floating point returning functions: `logb` has counterpart `ilogb`, `ilogbf` and `ilogbl`. The special values `FP_ILOGB0` and `FP_ILOGBNAN` are returned when the exponent is zero or not a number respectively, because these values cannot otherwise be distinctly represented in the more limited integer number space.

setjmp.h

This file declares two functions, and one type, for bypassing the normal function call and return discipline (useful for dealing with unusual conditions encountered in a low-level function of a program). It also defines the `jmp_buf` structure type required by these routines.

setjmp

```
int setjmp(jmp_buf env)
```

The calling environment is saved in `env`, for later use by the `longjmp` function. If the return is from a direct invocation, the `setjmp` function returns the value zero. If the return is from a call to the `longjmp` function, the `setjmp` function returns a non-zero value.

longjmp

```
void longjmp(jmp_buf env, int val)
```

The environment saved in `env` by the most recent call to `setjmp` is restored. If there has been no such call, or if the function containing the call to `setjmp` has terminated execution (eg with a return statement) in the interim, the behaviour is undefined. All accessible objects have values as at the time `longjmp` was called, except that the values of objects of automatic storage duration that do not have volatile type and that have been changed between the `setjmp` and `longjmp` calls are indeterminate.

As it bypasses the usual function call and return mechanism, the `longjmp` function executes correctly in contexts of interrupts, signals and any of their associated functions. However, if the `longjmp` function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behaviour is undefined.

After `longjmp` is completed, program execution continues as if the corresponding call to `setjmp` had just returned the value specified by `val`. The `longjmp` function cannot cause `setjmp` to return the value 0; if `val` is 0, `setjmp` returns the value 1.

signal.h

Signal declares a type (`sig_atomic_t`) and two functions.

It also defines several macros for handling various signals (conditions that may be reported during program execution). These are `SIG_DFL` (default routine), `SIG_IGN` (ignore signal routine) and `SIG_ERR` (dummy routine used to flag error return from `signal`).

```
void (*signal (int sig, void (*func)(int)))(int)
```

Think of this as

```
typedef void Handler(int);  
Handler *signal(int, Handler *);
```

Chooses one of three ways in which receipt of the signal number `sig` is to be subsequently handled. If the value of `func` is `SIG_DFL`, default handling for that signal will occur. If the value of `func` is `SIG_IGN`, the signal will be ignored. Otherwise `func` points to a function to be called when that signal occurs.

When a signal occurs, if `func` points to a function, first the equivalent of `signal(sig, SIG_DFL)` is executed. (If the value of `sig` is `SIGILL`, whether the reset to `SIG_DFL` occurs is implementation-defined (under RISC OS the reset does occur)). Next, the equivalent of `(*func)(sig)`; is executed. The function may terminate by calling the `abort`, `exit` or `longjmp` function. If `func` executes a return statement and the value of `sig` was `SIGFPE` or any other implementation-defined value corresponding to a computational exception, the behaviour is undefined. Otherwise, the program will resume execution at the point it was interrupted.

If the signal occurs other than as a result of calling the `abort` or `raise` function, the behaviour is undefined if the signal handler calls any function in the standard library other than the signal function itself or refers to any object with static storage duration other than by assigning a value to a volatile static variable of type `sig_atomic_t`. At program start-up, the equivalent of `signal(sig, SIG_IGN)` may be executed for some signals selected in an implementation-defined manner (under RISC OS this does not occur); the equivalent of `signal(sig, SIG_DFL)` is executed for all other signals defined by the implementation.

If the request can be honoured, the `signal` function returns the value of `func` for most recent call to `signal` for the specified signal `sig`. Otherwise, a value of `SIG_ERR` is returned and the integer expression `errno` is set to indicate the error.

raise

```
int raise(int /*sig*/)
```

Sends the signal sig to the executing program. Returns zero if successful, non-zero if unsuccessful.

stdalign.h

The `stdalign.h` header file is part of the C18 extensions to the C language, providing support for variable alignment.

Principally, it defines the macros `alignas` and `alignof`.

This header also includes the macros `__alignas_is_defined` and `__alignof_is_defined` which are both set to 1. This may be useful to prevent other conflicting redefinitions of the above two macros, or to allow fallback implementations with compilers that lack these C18 features.

stdarg.h

This file declares a type and defines three macros, for advancing through a list of arguments whose number and types are not known to the called function when it is translated. A function may be called with a variable number of arguments of differing types. Its parameter list contains one or more parameters, the rightmost of which plays a special role in the access mechanism, and will be called `parmN` in this description.

va_list

```
char *va_list[1]
```

An array type suitable for holding information needed by the macro `va_arg` and the function `va_end`. The called function declares a variable (referred to as `ap`) having type `va_list`. The variable `ap` may be passed as an argument to another function. `va_list` is an array type so that when an object of that type is passed as an argument it gets passed by reference, but this is not required by the ISO specification and cannot be relied on.

va_start

```
va_start(ap, parmN)
```

The `va_start` macro will be executed before any access to the unnamed arguments. The parameter `ap` points to an object that has type `va_list`. The `va_start` macro initialises `ap` for subsequent use by `va_arg` and `va_end`. The parameter `parmN` is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `, ...`). If the parameter `parmN` is declared with the register storage class the behaviour is undefined.

Returns: no value.

va_arg

```
va_arg(ap, type)
```

The `va_arg` macro expands to an expression that has the type and value of the next argument in the call. The parameter `ap` is the same as the `va_list` `ap` initialised by `va_start`. Each invocation of `va_arg` modifies `ap` so that successive arguments are returned in turn. The parameter `type` is a type name such that the type of a pointer to an object that has the specified type can be

obtained simply by postfixing a `*` to *type*. If *type* disagrees with the type of the actual next argument (as promoted according to the default argument promotions), the behaviour is undefined.

Returns: The first invocation of the `va_arg` macro after that of the `va_start` macro returns the value of the argument after that specified by `paramN`. Successive invocations return the values of the remaining arguments in succession. Care is taken in `va_arg` so that illegal things like `va_arg(ap, char)` – which may seem natural but are in fact illegal – are caught. `va_arg(ap, float)` is wrong but cannot be patched up at the C macro level.

va_copy

`va_copy(dest, src)`

The `va_copy` macro initialises `dest` to be at the position that `src` is at after the current sequence of `va_arg` invocations have been performed on it. Neither `va_copy` nor `va_start` may be used again on `dest` without an intervening use of `va_end` on `dest`, otherwise the behaviour is undefined.

va_end

`va_end(ap)`

The `va_end` macro facilitates a normal return from the function whose variable argument list was referenced by the expansion of `va_start` that initialised the `va_list ap`. If the `va_end` macro is not invoked before the return, the behaviour is undefined.

stdatomic.h

Atomic data types and functions to manipulate them are an optional part of the C18 language extensions which are syntax checked by the compiler but not otherwise implemented. Furthermore, this header file is not built into the compiler and must be provided as a supplement on disc.

stdbool.h

The `stdbool.h` header file is part of the C99 extensions to the C language, providing support for logical booleans.

It defines the macro `bool` which you can use in your programs to hold boolean state, internally this expands to `_Bool` which is the compiler's native boolean type.

Values for `true` and `false` are also defined as macros.

This header also includes the macro `__bool_true_false_are_defined` which is set to 1. This may be useful to prevent other conflicting redefinitions of the above three macros, or to allow fallback implementations with compilers that lack these C99 features.

stddef.h

This file contains a macro for calculating the offset of fields within a structure. It also defines the pointer constant `NULL` and three types.

<code>ptrdiff_t</code> (here <code>int</code>)	the signed integral type of the result of subtracting two pointers
<code>size_t</code> (here <code>unsigned int</code>)	the unsigned integral type of the result of the <code>sizeof</code> operator
<code>wchar_t</code> (here <code>int</code>)	also in <code>stdlib.h</code> . An integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; the null character has the code value zero and each member of the basic character set has a code value when used as the lone character in an integer character constant.
<code>size_t</code> <code>offsetof</code> (<code>type</code> , <code>member</code>)	Expands to an integral constant expression that has type <code>size_t</code> , the value of which is the offset in bytes from the beginning of a structure designated by <code>type</code> , of the member designated by <code>member</code> (if the specified member is a bit-field, the behaviour is undefined).

stdint.h

The `stdint.h` library header file is part of the C99 language extensions. It declares a group of specific width integer types, and accompanying macros. It should be considered in conjunction with the `inttypes.h` header file described on page 143 which, together, allow highly portable C programs to be written.

Integers are sized in power-of-2 multiples of the smallest memory building block, the byte, so in the following five categories the definition of N can be 8, 16, 32 or 64:

- Specific width types
 Unsigned – `uintN_t`, range 0 to `UINTN_MAX`
 Signed – `intN_t`, range `INTN_MIN` to `INTN_MAX`
- Least width types which possibly have unused bits in some implementations
 Unsigned – `uint_leastN_t`, range 0 to `UINT_LEASTN_MAX`
 Signed – `int_leastN_t`, range `INT_LEASTN_MIN` to `INT_LEASTN_MAX`
- Fastest types which are implemented with the most computationally efficient underlying type for the platform in question. On RISC OS this would most often be a 32 bit integer to match the AArch32 register set
 Unsigned – `uint_fastN_t`, range 0 to `UINT_FASTN_MAX`
 Signed – `int_fastN_t`, range `INT_FASTN_MIN` to `INT_FASTN_MAX`
- Integers to hold pointers where pointer arithmetic is required, `uintptr_t` and `intptr_t`. Again, on RISC OS these will be 32 bit integers to match the AArch32 register set. `UINTPTR_MAX`, `INTPTR_MIN`, and `INTPTR_MAX` describe the range of values as might be expected.
- Integers of greatest width for the implementation, `uintmax_t` and `intmax_t`. `UINTMAX_MAX`, `INTMAX_MIN`, and `INTMAX_MAX` describe the range of values as might be expected.

Several other integer limits of types not declared in `stdint.h` are provided for completeness. For `stddef.h` there are: `PTRDIFF_MIN` and `PTRDIFF_MAX` for `ptrdiff_t`; `SIZE_MAX` for `size_t`; and `WCHAR_MIN` and `WCHAR_MAX` for `wchar_t`. For `signal.h` there are: `SIG_ATOMIC_MIN` and `SIG_ATOMIC_MAX` for `sig_atomic_t`.

Constants held in your program source code can also be expressed definitively using a group of macros that append the qualifier letter required to declare the integer of that type on the platform in question. They are

`INTN_C(value)` and `UINTN_C(value)`
`INTMAX_C(value)` and `UINTMAX_C(value)`

for the signed and unsigned types respectively. Using these macros avoids compiler warnings when declaring constants which are not the compiler's native integer type.

stdio.h

stdio declares a few types, several macros, and many functions for performing input and output. For a discussion on Streams and Files refer to sections 7.19.2 and 7.19.3 in the ISO standard or to one of the other references given in the *Introduction* to this Guide.

FILE	is an object capable of recording all information needed to control a stream, such as its file position indicator, a pointer to its associated buffer, an error indicator that records whether a read/write error has occurred and an end-of-file indicator that records whether the end-of-file has been reached. The objects contained in the <code>#ifdef __system_io</code> clause are for system use only, and cannot be relied on between releases of C.
fpos_t	fpos_t is an object capable of recording all information needed to specify uniquely every position within a file.
off_t	off_t is an object capable of recording the offset from any position within a file to any other position within that file. It is only defined if the macro <code>_LARGEFILE_SOURCE</code> or <code>_LARGEFILE64_SOURCE</code> has been defined. See <i>Large file support</i> on page 129.
fpos64_t	fpos64_t is the equivalent of fpos_t and is used in its place when an application explicitly uses the 64-bit file extension functions <code>fgetpos64</code> and <code>fsetpos64</code> . It is only defined if the macro <code>_LARGEFILE64_SOURCE</code> has been defined. See <i>Large file support</i> on page 129.
off64_t	off64_t is the equivalent of off_t and is used in its place when an application explicitly uses the 64-bit file extension functions <code>ftello64</code> and <code>fseeko64</code> . It is only defined if the macro <code>_LARGEFILE64_SOURCE</code> has been defined. See <i>Large file support</i> on page 129.

remove

```
int remove(const char * filename)
```

Causes the file whose name is the string pointed to by *filename* to be removed. Subsequent attempts to open the file will fail, unless it is created anew. If the file is open, the behaviour of the `remove` function is implementation-defined (under RISC OS the operation fails).

Returns: zero if the operation succeeds, non-zero if it fails.

rename

```
int rename(const char * old, const char * new)
```

Causes the file whose name is the string pointed to by *old* to be henceforth known by the name given by the string pointed to by *new*. The file named *old* is effectively removed. If a file named by the string pointed to by *new* exists prior to the call of the `rename` function, the behaviour is implementation-defined (under RISC OS, the operation fails).

Returns: zero if the operation succeeds, non-zero if it fails, in which case if the file existed previously it is still known by its original name.

tmpfile

```
FILE *tmpfile(void)
```

Creates a temporary binary file that will be automatically removed when it is closed or at program termination. The file is created if possible in `Wimp$ScrapDir`, or failing that, in the directory `$.tmp`; it is then opened for update.

Returns: a pointer to the stream of the file that it created. If the file cannot be created, a null pointer is returned.

tmpfile64

```
FILE *tmpfile64(void)
```

This function is the same as `tmpfile` and is used in its place when an application wishes to use the explicit 64-bit file extension functions `fgetpos64`, `fsetpos64`, `ftello64` and `fseeko64`. It is only defined if the macro `_LARGEFILE64_SOURCE` has been defined. See *Large file support* on page 129.

tmpnam

```
char *tmpnam(char * s)
```

Generates a string that is not the same as the name of an existing file. The `tmpnam` function generates a different string each time it is called, up to `TMP_MAX` times. If it is called more than `TMP_MAX` times, the behaviour is implementation-defined (under RISC OS the algorithm for the name generation works just as well after `tmpnam` has been called more than `TMP_MAX` times as before; a name clash is impossible in any single half year period).

Returns: If the argument is a null pointer, the `tmpnam` function leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to the `tmpnam` function may modify the same object. If the argument is not a null pointer, it is assumed to point to an array of at least `L_tmpnam` characters; the `tmpnam` function writes its result in that array and returns the argument as its value.

fclose

```
int fclose(FILE * stream)
```

Causes the stream pointed to by *stream* to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated.

Returns: zero if the stream was successfully closed, or EOF if any errors were detected or if the stream was already closed.

fflush

```
int fflush(FILE * stream)
```

If the stream points to an output or update stream in which the most recent operation was output, the `fflush` function causes any unwritten data for that stream to be delivered to the host environment to be written to the file. If the stream points to an input or update stream, the `fflush` function undoes the effect of any preceding `ungetc` operation on the stream.

Returns: EOF if a write error occurs.

fopen

```
FILE *fopen(const char * filename, const char * mode)
```

Opens the file whose name is the string pointed to by *filename*, and associates a stream with it. The argument *mode* points to a string beginning with one of the following sequences:

<code>r</code>	open text file for reading
<code>w</code>	create text file for writing, or truncate to zero length
<code>a</code>	append; open text file or create for writing at eof
<code>rb</code>	open binary file for reading
<code>wb</code>	create binary file for writing, or truncate to zero length
<code>ab</code>	append; open binary file or create for writing at eof

<code>r+</code>	open text file for update (reading and writing)
<code>w+</code>	create text file for update, or truncate to zero length
<code>a+</code>	append; open text file or create for update, writing at eof
<code>r+b</code> or <code>rb+</code>	open binary file for update (reading and writing)
<code>w+b</code> or <code>wb+</code>	create binary file for update, or truncate to zero length
<code>a+b</code> or <code>ab+</code>	append; open binary file or create for update, writing at eof

- Opening a file with read mode (`r` as the first character in the *mode* argument) fails if the file does not exist or cannot be read.
- Opening a file with append mode (`a` as the first character in the *mode* argument) causes all subsequent writes to be forced to the current end of file, regardless of intervening calls to the `fseek` function.
- Opening a file with create mode (`w` as the first character in the *mode* argument) may also, starting with C18 supporting versions of the SharedCLibrary, be suffixed with `x` to denote exclusive access. Exclusive access guarantees that no other program has the file open for writing at the same time.
- In some implementations, opening a binary file with append mode (`b` as the second or third character in the *mode* argument) may initially position the file position indicator beyond the last data written, because of null padding (but not under RISC OS).
- When a file is opened with update mode (`+` as the second or third character in the *mode* argument), both input and output may be performed on the associated stream. However, output may not be directly followed by input without an intervening call to the `fflush` function or to a file positioning function (`fseek`, `fsetpos`, or `rewind`), nor may input be directly followed by output without an intervening call to the `fflush` function or to a file positioning function, unless the input operation encounters end-of-file.
- Opening a file with update mode may open or create a binary stream in some implementations (but not under RISC OS). When opened, a stream is fully buffered if and only if it does not refer to an interactive device. The error and end-of-file indicators for the stream are cleared.

Returns: a pointer to the object controlling the stream. If the open operation fails, `fopen` returns a null pointer.

fopen64

```
FILE *fopen64(const char * filename, const char * mode)
```

This function is the same as `fopen` and is used in its place when an application wishes to use the explicit 64-bit file extension functions `fgetpos64`, `fsetpos64`, `ftello64` and `fseeko64`. It is only defined if the macro `_LARGEFILE64_SOURCE` has been defined. See *Large file support* on page 129.

freopen

```
FILE *freopen(const char * filename, const char * mode,
              FILE * stream)
```

Opens the file whose name is the string pointed to by *filename* and associates the stream pointed to by *stream* with it. The *mode* argument is used just as in the `fopen` function. The `freopen` function first attempts to close any file that is associated with the specified stream. Failure to close the file successfully is ignored. The error and end-of-file indicators for the stream are cleared.

Returns: a null pointer if the operation fails. Otherwise, `freopen` returns the value of the stream.

freopen64

```
FILE *freopen64(const char * filename, const char * mode,
                 FILE * stream)
```

This function is the same as `freopen` and is used in its place when an application wishes to use the explicit 64-bit file extension functions `fgetpos64`, `fsetpos64`, `ftello64` and `fseeko64`. It is only defined if the macro `_LARGEFILE64_SOURCE` has been defined. See *Large file support* on page 129.

setbuf

```
void setbuf(FILE * stream, char * buf)
```

Except that it returns no value, the `setbuf` function is equivalent to the `setvbuf` function invoked with the values `_IOFBF` for *mode* and `BUFSIZ` for *size*, or if *buf* is a null pointer, with the value `_IONBF` for *mode*.

Returns: no value.

setvbuf

```
int setvbuf(FILE * stream, char * buf, int mode, size_t
           size)
```

This may be used after the stream pointed to by *stream* has been associated with an open file but before it is read or written. The argument *mode* determines how *stream* will be buffered, as follows:

- `_IOFBF` causes input/output to be fully buffered.
- `_IOLBF` causes output to be line buffered (the buffer will be flushed when a newline character is written, when the buffer is full, or when interactive input is requested).
- `_IONBF` causes input/output to be completely unbuffered.

If *buf* is not the null pointer, the array it points to may be used instead of an automatically allocated buffer (the buffer must have a lifetime at least as great as the open stream, so the stream should be closed before a buffer that has automatic storage duration is deallocated upon block exit). The argument *size* specifies the size of the array. The contents of the array at any time are indeterminate. *buf* must be non-null.

Returns: zero on success, or non-zero if an invalid value is given for mode or size, or if the request cannot be honoured.

fprintf

```
int fprintf(FILE * stream, const char * format, ...)
```

writes output to the stream pointed to by *stream*, under control of the string pointed to by *format* that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behaviour is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The `fprintf` function returns when the end of the format string is reached. The format must be a multibyte character sequence, beginning and ending in its initial shift state (in all locales supported under RISC OS this is the same as a plain character string). The format is composed of zero or more directives: ordinary multibyte characters (not %), which are copied unchanged to the output stream; and conversion specifiers, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character %. For a complete description of the available conversion specifiers refer to section 7.19.6.1 in the ISO standard or to one of the other references in the *Introduction* to this Guide. The minimum value for the maximum number of characters that can be produced by any single conversion is at least 509.

A brief and incomplete description of conversion specifications is:

[flags] [field width] [.precision] specifier-char

flags is most commonly -, indicating left justification of the output item within the field. If omitted, the item will be right justified.

field width is the minimum width of field to use. If the formatted item is longer, a bigger field will be used; otherwise, the item will be right (left) justified in the field.

precision is the minimum number of digits to print for a d, i, o, u, x or X conversion, the number of digits to appear after the decimal digit for e, E, a, A and f conversions, the maximum number of significant digits for g and G conversions, or the maximum number of characters to be written from strings in an s conversion.

Either or both of *field width* and *precision* may be *, indicating that the value is an argument to printf.

The *specifier chars* are:

d, i	int printed as signed decimal
o, u, x, X	unsigned int value printed as unsigned octal, decimal or hexadecimal
f	double value printed in the style [-] ddd.ddd
a, A	hexadecimal double value printed in the style [-] 0xh.hhh
e, E	double value printed in the style [-] d.ddd...e±dd
g, G	double printed in f or e format, whichever is more appropriate
c	int value printed as unsigned char
s	char * value printed as a string of characters
p	void * argument printed as a hexadecimal address
%	write a literal %

Returns: the number of characters transmitted, or a negative value if an output error occurred.

printf

```
int printf(const char * format, ...)
```

Equivalent to fprintf with the argument stdout interposed before the arguments to printf.

Returns: the number of characters transmitted, or a negative value if an output error occurred.

sprintf

```
int sprintf(char * s, const char * format, ...)
```

Equivalent to `fprintf`, except that the argument *s* specifies an array into which the generated output is to be written, rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned sum.

Returns: the number of characters written to the array, not counting the terminating null character.

snprintf

```
int snprintf(char * s, size_t n, const char * format, ...)
```

Equivalent to `sprintf`, except the buffer size *n* is declared. In situations where the amount of output exceeds the size of the buffer, this prevents buffer overruns.

If *n* is zero, nothing is written and *s* may be a null pointer. Otherwise, output characters beyond the *n*-1st are discarded rather than being written to the array, and a null character is written at the end of the characters actually written into the array.

Returns: the number of characters that would have been written had *n* been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is non-negative and less than *n*.

fscanf

```
int fscanf(FILE * stream, const char * format, ...)
```

Reads input from the stream pointed to by *stream*, under control of the string pointed to by *format* that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behaviour is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The format is composed of zero or more directives, one or more white-space characters, an ordinary character (not %), or a conversion specification. Each conversion specification is introduced by the character %. For a description of the available conversion specifiers refer to section 7.19.6.1 in the ISO standard, or to any of the references listed in the chapter *Introduction* on page 1. A brief list is given above, under the entry for `fprintf`.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversions terminate on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including newline characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the `%n` directive.

Returns: the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `fscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early conflict between an input character and the format.

scanf

```
int scanf(const char * format, ...)
```

Equivalent to `fscanf` with the argument `stdin` interposed before the arguments to `scanf`.

Returns: the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `scanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

sscanf

```
int sscanf(const char * s, const char * format, ...)
```

Equivalent to `fscanf` except that the argument *s* specifies a string from which the input is to be obtained, rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the `fscanf` function.

Returns: the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `scanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

vprintf

```
int vprintf(const char * format, va_list arg)
```

Equivalent to `printf`, with the variable argument list replaced by `arg`, which has been initialised by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vprintf` function does not invoke the `va_end` function.

Returns: the number of characters transmitted, or a negative value if an output error occurred.

vfprintf

```
int vfprintf(FILE * stream, const char * format, va_list  
             arg)
```

Equivalent to `fprintf`, with the variable argument list replaced by `arg`, which has been initialised by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfprintf` function does not invoke the `va_end` function.

Returns: the number of characters transmitted, or a negative value if an output error occurred.

vsprintf

```
int vsprintf(char * s, const char * format, va_list arg)
```

Equivalent to `sprintf`, with the variable argument list replaced by `arg`, which has been initialised by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vsprintf` function does not invoke the `va_end` function.

Returns: the number of characters written in the array, not counting the terminating null character.

vsnprintf

```
int vsnprintf(char * s, size_t n, const char * format,  
              va_list arg)
```

Equivalent to `snprintf`, with the variable argument list replaced by `arg`, which has been initialised by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vsnprintf` function does not invoke the `va_end` function.

Returns: the number of characters that would have been written had `n` been sufficiently large, not counting the terminating null character. Thus, the null-terminated output has been completely written if and only if the returned value is non-negative and less than `n`.

vscanf

```
int vscanf(const char * format, va_list arg)
```

Equivalent to `scanf`, with the variable argument list replaced by `arg`, which has been initialised by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vscanf` function does not invoke the `va_end` function.

Returns: the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `vscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

vfscanf

```
int vfscanf(FILE * stream, const char * format,  
            va_list arg)
```

Equivalent to `fscanf`, with the variable argument list replaced by `arg`, which has been initialised by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vfscanf` function does not invoke the `va_end` function.

Returns: the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `vscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

vsscanf

```
int vsscanf(const char * s, const char * format,  
            va_list arg)
```

Equivalent to `sscanf`, with the variable argument list replaced by `arg`, which has been initialised by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vsscanf` function does not invoke the `va_end` function.

Returns: the value of the macro `EOF` if an input failure occurs before any conversion. Otherwise, the `vsscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

fgetc

```
int fgetc(FILE * stream)
```


Obtains the next character (if present) as an unsigned char converted to an int, from the input stream pointed to by *stream*, and advances the associated file position indicator (if defined).

Returns: the next character from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator is set and `fgetc` returns EOF. If a read error occurs, the error indicator is set and `fgetc` returns EOF.

fgets

```
char *fgets(char * s, int n, FILE * stream)
```

Reads at most one less than the number of characters specified by *n* from the stream pointed to by *stream* into the array pointed to by *s*. No additional characters are read after a newline character (which is retained) or after end-of-file. A null character is written immediately after the last character read into the array.

Returns: *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

fputc

```
int fputc(int c, FILE * stream)
```

Writes the character specified by *c* (converted to an unsigned char) to the output stream pointed to by *stream*, at the position indicated by the associated file position indicator (if defined), and advances the indicator appropriately. If the file cannot support positioning requests, or if the stream was opened with append mode, the character is appended to the output stream.

Returns: the character written. If a write error occurs, the error indicator is set and `fputc` returns EOF.

fputs

```
int fputs(const char * s, FILE * stream)
```

Writes the string pointed to by *s* to the stream pointed to by *stream*. The terminating null character is not written.

Returns: EOF if a write error occurs; otherwise it returns a non-negative value.

getc

```
int getc(FILE * stream)
```

Equivalent to `fgetc` except that it may be (and is under RISC OS) implemented as a macro. *stream* may be evaluated more than once, so the argument should never be an expression with side effects.

Returns: the next character from the input stream pointed to by *stream*. If the stream is at end-of-file, the end-of-file indicator is set and `getc` returns EOF. If a read error occurs, the error indicator is set and `getc` returns EOF.

getchar

```
int getchar(void)
```

Equivalent to `getc` with the argument `stdin`.

Returns: the next character from the input stream pointed to by `stdin`. If the stream is at end-of-file, the end-of-file indicator is set and `getchar` returns EOF. If a read error occurs, the error indicator is set and `getchar` returns EOF.

gets

```
char *gets(char * s)
```

Reads characters from the input stream pointed to by `stdin` into the array pointed to by *s*, until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array.

Note: Due to security concerns of overwriting the array *s* whose size is not known, this function should be considered deprecated. In C18 it has been removed entirely.

Returns: *s* if successful. If end-of-file is encountered and no characters have been read into the array, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs during the operation, the array contents are indeterminate and a null pointer is returned.

putc

```
int putc(int c, FILE * stream)
```

Equivalent to `fputc` except that it may be (and is under RISC OS) implemented as a macro. *stream* may be evaluated more than once, so the argument should never be an expression with side effects.

Returns: the character written. If a write error occurs, the error indicator is set and `putc` returns EOF.

putchar

```
int putchar(int c)
```

Equivalent to `putc` with the second argument `stdout`.

Returns: the character written. If a write error occurs, the error indicator is set and `putc` returns EOF.

puts

```
int puts(const char * s)
```

Writes the string pointed to by `s` to the stream pointed to by `stdout`, and appends a newline character to the output. The terminating null character is not written.

Returns: EOF if a write error occurs; otherwise it returns a non-negative value.

ungetc

```
int ungetc(int c, FILE * stream)
```

Pushes the character specified by `c` (converted to an unsigned char) back onto the input stream pointed to by `stream`. The character will be returned by the next read on that stream. An intervening call to the `fflush` function or to a file positioning function (`fseek`, `fsetpos`, `rewind`) discards any pushed-back characters. The external storage corresponding to the stream is unchanged. One character pushback is guaranteed. If the `ungetc` function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail. If the value of `c` equals that of the macro `EOF`, the operation fails and the input stream is unchanged.

A successful call to the `ungetc` function clears the end-of-file indicator. The value of the file position indicator after reading or discarding all pushed-back characters will be the same as it was before the characters were pushed back. For a text stream, the value of the file position indicator after a successful call to the `ungetc` function is unspecified until all pushed-back characters are read or discarded. For a binary stream, the file position indicator is decremented by each successful call to the `ungetc` function; if its value was zero before a call, it is indeterminate after the call.

Returns: the character pushed back after conversion, or EOF if the operation fails.

fread

```
size_t fread(void * ptr, size_t size,
             size_t nmemb, FILE * stream)
```

Reads into the array pointed to by *ptr*, up to *nmemb* members whose size is specified by *size*, from the stream pointed to by *stream*. The file position indicator (if defined) is advanced by the number of characters successfully read. If an error occurs, the resulting value of the file position indicator is indeterminate. If a partial member is read, its value is indeterminate. The `ferror` or `feof` function shall be used to distinguish between a read error and end-of-file.

Returns: the number of members successfully read, which may be less than *nmemb* if a read error or end-of-file is encountered. If *size* or *nmemb* is zero, `fread` returns zero and the contents of the array and the state of the stream remain unchanged.

fwrite

```
size_t fwrite(const void * ptr,
             size_t size, size_t nmemb, FILE * stream)
```

Writes, from the array pointed to by *ptr* up to *nmemb* members whose size is specified by *size*, to the stream pointed to by *stream*. The file position indicator (if defined) is advanced by the number of characters successfully written. If an error occurs, the resulting value of the file position indicator is indeterminate.

Returns: the number of members successfully written, which will be less than *nmemb* only if a write error is encountered.

fgetpos

```
int fgetpos(FILE * stream, fpos_t * pos)
```

Stores the current value of the file position indicator for the stream pointed to by *stream* in the object pointed to by *pos*. The value stored contains unspecified information usable by the `fsetpos` function for repositioning the stream to its position at the time of the call to the `fgetpos` function.

Returns: zero, if successful. Otherwise non-zero is returned and the integer expression `errno` is set to an implementation-defined non-zero value (under RISC OS `fgetpos` cannot fail).

fgetpos64

```
int fgetpos64(FILE * stream, fpos64_t * pos)
```

The `fgetpos64` function is the same as the `fgetpos` function except that the `pos` argument is of type `fpos64_t`. It is only defined if the macro `_LARGEFILE64_SOURCE` has been defined. See *Large file support* on page 129.

fseek

```
int fseek(FILE * stream, long int offset, int whence)
```

Sets the file position indicator for the stream pointed to by *stream*. For a binary stream, the new position is at the signed number of characters specified by *offset* away from the point specified by *whence*. The specified point is the beginning of the file for `SEEK_SET`, the current position in the file for `SEEK_CUR`, or end-of-file for `SEEK_END`. A binary stream need not meaningfully support `fseek` calls with a *whence* value of `SEEK_END`, though the Acorn implementation does. For a text stream, *offset* is either zero or a value returned by an earlier call to the `ftell` function on the same stream; *whence* is then `SEEK_SET`. The Acorn implementation also allows a text stream to be positioned in exactly the same manner as a binary stream, but this is not portable. The `fseek` function clears the end-of-file indicator and undoes any effects of the `ungetc` function on the same stream. After an `fseek` call, the next operation on an update stream may be either input or output.

Returns: non-zero only for a request that cannot be satisfied.

fseeko

```
int fseeko(FILE * stream, off_t offset, int whence)
```

The `fseeko` function is the same as the `fseek` function except that the `offset` argument is of type `off_t`. It is only defined if the macro `_LARGEFILE_SOURCE` or `_LARGEFILE64_SOURCE` has been defined. See *Large file support* on page 129.

fseeko64

```
int fseeko64(FILE * stream, off64_t offset, int whence)
```

The `fseeko64` function is the same as the `fseek` function except that the `offset` argument is of type `off64_t`. It is only defined if the macro `_LARGEFILE64_SOURCE` has been defined. See *Large file support* on page 129.

fsetpos

```
int fsetpos(FILE * stream, const fpos_t * pos)
```

Sets the file position indicator for the stream pointed to by *stream* according to the value of the object pointed to by *pos*, which is a value returned by an earlier call to the `fgetpos` function on the same stream. The `fsetpos` function clears the end-of-file indicator and undoes any effects of the `ungetc` function on the same stream. After an `fsetpos` call, the next operation on an update stream may be either input or output.

Returns: zero, if successful. Otherwise non-zero is returned and the integer expression `errno` is set to an implementation-defined non-zero value (under RISC OS the value is that of `EDOM` in `math.h`).

fsetpos64

```
int fsetpos64(FILE * stream, const fpos64_t * pos)
```

The `fsetpos64` function is the same as the `fsetpos` function except that the *pos* argument is of type `fpos64_t`. It is only defined if the macro `_LARGEFILE64_SOURCE` has been defined. See *Large file support* on page 129.

ftell

```
long int ftell(FILE * stream)
```

Obtains the current value of the file position indicator for the stream pointed to by *stream*. For a binary stream, the value is the number of characters from the beginning of the file. For a text stream, the file position indicator contains unspecified information, usable by the `fseek` function for returning the file position indicator to its position at the time of the `ftell` call; the difference between two such return values is not necessarily a meaningful measure of the number of characters written or read. However, for the Acorn implementation, the value returned is merely the byte offset into the file, whether the stream is text or binary.

Returns: if successful, the current value of the file position indicator. On failure, the `ftell` function returns `-1L` and sets the integer expression `errno` to an implementation-defined non-zero value (under RISC OS `ftell` cannot fail).

ftello

```
off_t ftello(FILE * stream)
```

The `ftello` function is the same as the `ftell` function except that the return value is of type `off_t`. It is only defined if the macro `_LARGEFILE_SOURCE` or `_LARGEFILE64_SOURCE` has been defined. See *Large file support* on page 129.

ftello64

```
off64_t ftello64(FILE * stream)
```

The `ftello64` function is the same as the `ftell` function except that the return value is of type `off64_t`. It is only defined if the macro `_LARGEFILE64_SOURCE` has been defined. See *Large file support* on page 129.

rewind

```
void rewind(FILE * stream)
```

Sets the file position indicator for the stream pointed to by *stream* to the beginning of the file. It is equivalent to `(void)fseek(stream, 0L, SEEK_SET)` except that the error indicator for the stream is also cleared.

Returns: no value.

clearerr

```
void clearerr(FILE * stream)
```

Clears the end-of-file and error indicators for the stream pointed to by *stream*. These indicators are cleared only when the file is opened or by an explicit call to the `clearerr` function or to the `rewind` function.

Returns: no value.

feof

```
int feof(FILE * stream)
```

Tests the end-of-file indicator for the stream pointed to by *stream*.

Returns: non-zero if the end-of-file indicator is set for *stream*.

ferror

```
int ferror(FILE * stream)
```

Tests the error indicator for the stream pointed to by *stream*.

Returns: non-zero if the error indicator is set for *stream*.

perror

```
void perror(const char * s)
```

Maps the error number in the integer expression `errno` to an error message. It writes a sequence of characters to the standard error stream thus: first (if *s* is not a null pointer and the character pointed to by *s* is not the null character), the string pointed to by *s* followed by a colon and a space; then an appropriate error message string followed by a newline character. The contents of the error message strings are the same as those returned by the `strerror` function with argument `errno`, which are implementation-defined.

Returns: no value.

stdlib.h

`stdlib.h` declares four types, several general purpose functions, and defines several macros.

atof

```
double atof(const char * nptr)
```

Converts the initial part of the string pointed to by *nptr* to double * representation.

Returns: the converted value.

atoi

```
int atoi(const char * nptr)
```

Converts the initial part of the string pointed to by *nptr* to int representation.

Returns: the converted value.

atol

```
long int atol(const char * nptr)
```

Converts the initial part of the string pointed to by *nptr* to long int representation.

Returns: the converted value.

strtod

```
double strtod(const char * nptr, char ** endptr)
```

Converts the initial part of the string pointed to by *nptr* to double representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function), a subject sequence resembling a floating point constant, and a final string of one or more unrecognised characters, including the terminating null character of the input string. It then attempts to convert the subject sequence to a floating point number, and returns the result. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Returns: the converted value if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, plus or minus `HUGE_VAL` is returned (according to the sign of the value), and the value of the macro `ERANGE` is stored in `errno`. If the correct value would cause underflow, zero is returned and the value of the macro `ERANGE` is stored in `errno`.

strtol

```
long int strtol(const char * nptr, char **endptr, int
                base)
```

Converts the initial part of the string pointed to by *nptr* to long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white-space characters (as specified by the `isspace` function), a subject sequence resembling an integer represented in some radix determined by the value of *base*, and a final string of one or more unrecognised characters, including the terminating null character of the input string.

It then attempts to convert the subject sequence to an integer, and returns the result. If the value of *base* is 0, the expected form of the subject sequence is that of an integer constant (described precisely in the ISO standard, section 7.20.1.3), optionally preceded by a + or – sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) through z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of the *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits following the sign if present. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Returns: the converted value if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, `LONG_MAX` or `LONG_MIN` is returned (according to the sign of the value), and the value of the macro `ERANGE` is stored in `errno`.

strtoll

```
long long int strtoll(const char * nptr, char **endptr,
                     int base)
```

Converts the initial part of the string pointed to by *nptr* to long long int representation. The behaviour is otherwise as for `strtol`.

strtoul

```
unsigned long int strtoul(const char * nptr, char **
                        endptr, int base)
```

Converts the initial part of the string pointed to by *nptr* to unsigned long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white space characters (as determined by the `isspace` function), a subject sequence resembling an unsigned integer represented in some radix determined by the value of *base*, and a final string of one or more unrecognised characters, including the terminating null character of the input string.

It then attempts to convert the subject sequence to an unsigned integer, and returns the result. If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant (described precisely in the ISO standard, section 7.20.1.3), optionally preceded by a + or – sign, but not including an integer suffix. If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a + or – sign, but not including an integer suffix. The letters from a (or A) through z (or Z) stand for the values 10 to 35; only letters whose ascribed values are less than that of the base are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits following the sign, if present. A pointer to the final string is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Returns: the converted value if any. If no conversion could be performed, zero is returned. If the correct value is outside the range of representable values, ULONG_MAX is returned, and the value of the * macro ERANGE is stored in `errno`.

strtoull

```
unsigned long long int strtoull(const char * nptr,
                               char **endptr, int base)
```

Converts the initial part of the string pointed to by *nptr* to unsigned long long int representation. The behaviour is otherwise as for `strtoul`.

rand

```
int rand(void)
```

Computes a sequence of pseudo-random integers in the range 0 to RAND_MAX, where RAND_MAX = 0x7fffffff.

Returns: a pseudo-random integer.

srand

```
void srand(unsigned int seed)
```

Uses its argument as a seed for a new sequence of pseudo-random numbers to be returned by subsequent calls to `rand`. If `srand` is then called with the same seed value, the sequence of pseudo-random numbers will be repeated. If `rand` is called before any calls to `srand` have been made, the same sequence is generated as when `srand` is first called with a seed value of 1.

calloc

```
void *calloc(size_t nmemb, size_t size)
```

Allocates space for an array of *nmemb* objects, each of whose size is *size*. The space is initialised to all bits zero.

Returns: either a null pointer or a pointer to the allocated space.

free

```
void free(void * ptr)
```

Causes the space pointed to by *ptr* to be deallocated (made available for further allocation). If *ptr* is a null pointer, no action occurs. Otherwise, if *ptr* does not match a pointer earlier returned by `aligned_alloc`, `calloc`, `malloc` or `realloc` or if the space has been deallocated by a call to `free` or `realloc`, the behaviour is undefined.

malloc

```
void *malloc(size_t size)
```

Allocates space for an object whose size is specified by *size* and whose value is indeterminate.

Returns: either a null pointer or a pointer to the allocated space.

realloc

```
void *realloc(void * ptr, size_t size)
```

Changes the size of the object pointed to by *ptr* to the size specified by *size*. The contents of the object is unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate. If *ptr* is a null pointer, the `realloc` function behaves like a call to `malloc` for the specified size. Otherwise, if *ptr* does not match a pointer earlier returned by `calloc`, `malloc` or `realloc`, or if the space has been deallocated by a call to `free` or `realloc`, the behaviour is undefined. If the space cannot be allocated, the object pointed to by *ptr* is unchanged. If *size* is zero and *ptr* is not a null pointer, the object it points to is freed.

Returns: either a null pointer or a pointer to the possibly moved allocated space.

aligned_alloc

```
void *aligned_alloc(size_t alignment, size_t size)
```

Allocates space for an object whose alignment is specified by *alignment*, whose size is specified by *size* and whose value is indeterminate.

Returns: either a null pointer or a pointer to the allocated space.

abort

```
void abort(void)
```

Causes abnormal program termination to occur, unless the signal `SIGABRT` is being caught and the signal handler does not return. Whether open output streams are flushed or open streams are closed or temporary files removed is implementation-defined (under RISC OS all these occur). An implementation-defined form of the status 'unsuccessful termination' (1 under RISC OS) is returned to the host environment by means of a call to `raise(SIGABRT)`.

atexit

```
int atexit(void (* func)(void))
```

Registers the function pointed to by *func*, to be called without its arguments at normal program termination. It is possible to register at least 32 functions.

Returns: zero if the registration succeeds, non-zero if it fails.

exit

```
void exit(int status)
```

Causes normal program termination to occur. If more than one call to the `exit` function is executed by a program (for example, by a function registered with `atexit`), the behaviour is undefined. First, all functions registered by the `atexit` function are called, in the reverse order of their registration. Next, all open output streams are flushed, all open streams are closed, and all files created by the `tmpfile` function are removed. Finally, control is returned to the host environment. If the value of *status* is zero or `EXIT_SUCCESS`, an implementation-defined form of the status 'successful termination' (0 under RISC OS) is returned. If the value of *status* is `EXIT_FAILURE`, an implementation-defined form of the status 'unsuccessful termination' (1 under RISC OS) is returned. Otherwise the status returned is implementation-defined (the value of *status* is returned under RISC OS).

at_quick_exit

```
int at_quick_exit(void (* func)(void))
```

Registers the function pointed to by *func*, to be called without its arguments at quick normal program termination. It is possible to register at least 32 functions.

Returns: zero if the registration succeeds, non-zero if it fails.

quick_exit

```
void quick_exit(int status)
```

Causes normal program termination to occur. No functions registered by the `atexit` function or signal handlers registered by the `signal` function are called. If a signal is raised while the `quick_exit` is executing, the behaviour is undefined. First, all functions registered by the `at_quick_exit` function are called, in the reverse order of their registration. Finally, control is returned to the host environment. If the value of *status* is zero or `EXIT_SUCCESS`, an implementation-defined form of the status 'successful termination' (0 under RISC OS) is returned. If the value of *status* is `EXIT_FAILURE`, an implementation-defined form of the status 'unsuccessful termination' (1 under RISC OS) is returned. Otherwise the status returned is implementation-defined (the value of *status* is returned under RISC OS).

getenv

```
char *getenv(const char * name)
```

Searches the environment list, provided by the host environment, for a string that matches the string pointed to by *name*. The set of environment names and the method for altering the environment list are implementation-defined.

Returns: a pointer to a string associated with the matched list member. The array pointed to is not modified by the program, but may be overwritten by a subsequent call to the `getenv` function. If the specified name cannot be found, a null pointer is returned.

system

```
int system(const char * string)
```

Passes the string pointed to by *string* to the host environment to be executed by a command processor in an implementation-defined manner. A null pointer may be used for *string*, to inquire whether a command processor exists. Under RISC OS, care must be taken, when executing a command, that the command does not overwrite the calling program. To control this, the string `chain:` or `call:` may immediately precede the actual command. The effect of `call:` is the same as if `call:` were not present. When a command is called, the caller is first moved to a safe place in application workspace. When the callee terminates, the caller is restored. This requires enough memory to hold caller and callee simultaneously. When a command is chained, the caller may be overwritten. If the caller is not overwritten, the caller exits when the caller terminates. Thus a transfer of control is effected and memory requirements are minimised.

Returns: If the argument is a null pointer, the `system` function returns non-zero only if a command processor is available. If the argument is not a null pointer, it returns an implementation-defined value (under RISC OS 0 is returned for success and -2 for failure to invoke the command; any other value is the return code from the executed command).

bsearch

```
void *bsearch(const void *key, const void * base,  
              size_t nmemb, size_t size, int (* compar)  
              (const void *, const void *))
```

Searches an array of *nmemb* objects, the initial member of which is pointed to by *base*, for a member that matches the object pointed to by *key*. The size of each member of the array is specified by *size*. The contents of the array must be in ascending sorted order according to a comparison function pointed to by *compar*,

which is called with two arguments that point to the key object and to an array member, in that order. The function returns an integer less than, equal to, or greater than zero if the key object is considered, respectively, to be less than, to match, or to be greater than the array member.

Returns: a pointer to a matching member of the array, or a null pointer if no match is found. If two members compare as equal, which member is matched is unspecified.

qsort

```
void qsort(void * base, size_t nmemb, size_t size,
           int (* compar)(const void *, const void *))
```

Sorts an array of *nmemb* objects, the initial member of which is pointed to by *base*. The size of each object is specified by *size*. The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared. The function returns an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is unspecified.

abs

```
int abs(int j)
```

Computes the absolute value of an integer *j*. If the result cannot be represented, the behaviour is undefined.

Returns: the absolute value.

labs

```
long int labs(long int j)
```

Computes the absolute value of a long integer *j*. If the result cannot be represented, the behaviour is undefined.

Returns: the absolute value.

llabs

```
long long int llabs(long long int j)
```


Computes the absolute value of a long long integer *j*. If the result cannot be represented, the behaviour is undefined.

Returns: the absolute value.

div

```
div_t div(int numer, int denom)
```

Computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the resulting quotient is the integer of lesser magnitude that is the nearest to the algebraic quotient. If the result cannot be represented, the behaviour is undefined; otherwise, $\text{quot} * \text{denom} + \text{rem}$ equals *numer*.

Returns: a structure of type `div_t`, comprising both the quotient and the remainder. The structure contains the following members: `int quot`; `int rem`. You may not rely on their order.

ldiv

```
ldiv_t ldiv(long int numer, long int denom)
```

Computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented, the behaviour is undefined; otherwise, $\text{quot} * \text{denom} + \text{rem}$ equals *numer*.

Returns: a structure of type `ldiv_t`, comprising both the quotient and the remainder. The structure contains the following members: `long int quot`; `long int rem`. You may not rely on their order.

lldiv

```
lldiv_t lldiv(long long int numer, long long int denom)
```

Computes the quotient and remainder of the division of the numerator *numer* by the denominator *denom*. If the division is inexact, the sign of the resulting quotient is that of the algebraic quotient, and the magnitude of the resulting quotient is the largest integer less than the magnitude of the algebraic quotient. If the result cannot be represented, the behaviour is undefined; otherwise, $\text{quot} * \text{denom} + \text{rem}$ equals *numer*.

Returns: a structure of type `lldiv_t`, comprising both the quotient and the remainder. The structure contains the following members: `long long int quot`; `long long int rem`. You may not rely on their order.

Multibyte character functions

The behaviour of the multibyte character functions is affected by the `LC_CTYPE` category of the current locale. For a state-dependent encoding, each function is placed into its initial state by a call for which its character pointer argument, *s*, is a null pointer. Subsequent calls with *s* as other than a null pointer cause the internal state of the function to be altered as necessary. A call with *s* as a null pointer causes these functions to return a non-zero value if encodings have state dependency, and a zero otherwise. After the `LC_CTYPE` category is changed, the shift state of these functions is indeterminate.

mblen

```
int mblen(const char * s, size_t n)
```

If *s* is not a null pointer, the `mblen` function determines the number of bytes comprising the multibyte character pointed to by *s*. Except that the shift state of the `mbtowc` function is not affected, it is equivalent to `mbtowc ((wchar_t *) 0, s, n)`.

Returns: If *s* is a null pointer, the `mblen` function returns a non-zero or zero value, if multibyte character encodings, respectively do or do not have state-dependent encodings. If *s* is not a null pointer, the `mblen` function either returns a 0 (if *s* points to a null character), or returns the number of bytes that comprise the multibyte character (if the next *n* of fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

mbtowc

```
int mbtowc(wchar_t * pwc, const char * s, size_t n)
```

If *s* is not a null pointer, the `mbtowc` function determines the number of bytes that comprise the multibyte character pointed to by *s*. It then determines the code for value of type `wchar_t` that corresponds to that multibyte character. (The value of the code corresponding to the null character is zero). If the multibyte character is valid and *pwc* is not a null pointer, the `mbtowc` function stores the code in the object pointed to by *pwc*. At most *n* bytes of the array pointed to by *s* will be examined.

Returns: If *s* is a null pointer, the `mbtowc` function returns a non-zero or zero value, if multibyte character encodings, respectively do or do not have state-dependent encodings. If *s* is not a null pointer, the `mbtowc` function either

returns a 0 (if *s* points to a null character), or returns the number of bytes that comprise the converted multibyte character (if the next *n* of fewer bytes form a valid multibyte character), or returns -1 (if they do not form a valid multibyte character).

wctomb

```
int wctomb(char * s, wchar_t wchar)
```

Determines the number of bytes need to represent the multibyte character corresponding to the code whose value is *wchar* (including any change in shift state). It stores the multibyte character representation in the array object pointed to by *s* (if *s* is not a null pointer). At most MB_CUR_MAX characters are stored. If the value of *wchar* is zero, the wctomb function is left in the initial shift state).

Returns: If *s* is a null pointer, the wctomb function returns a non-zero or zero value, if multibyte character encodings, respectively do or do not have state-dependent encodings. If *s* is not a null pointer, the wctomb function returns a -1 if the value of *wchar* does not correspond to a valid multibyte character, or returns the number of bytes that comprise the multibyte character corresponding to the value of *wchar*.

Multibyte string functions

The behaviour of the multibyte string functions is affected by the LC_CTYPE category of the current locale.

mbstowcs

```
size_t mbstowcs(wchar_t * pwcs, const char * s, size_t n)
```

Converts a sequence of multibyte characters that begins in the initial shift state from the array pointed to by *s* into a sequence of corresponding codes and stores not more than *n* codes into the array pointed to by *pwcs*. No multibyte character that follow a null character (which is converted into a code with value zero) will be examined or converted. Each multibyte character is converted as if by a call to the mbtowc function. If an invalid multibyte character is found, mbstowcs returns (size_t)-1. Otherwise, the mbstowcs function returns the number of array elements modified, not including a terminating zero code, if any.

wcstombs

```
size_t wcstombs(char * s, const wchar_t * pwcs, size_t n)
```

Converts a sequence of codes that correspond to multibyte characters from the array pointed to by *pwcs* into a sequence of multibyte characters that begins in the initial shift state and stores these multibyte characters into the array pointed to by *s*, stopping if a multibyte character would exceed the limit of *n* total bytes or if a null character is stored. Each code is converted as if by a call to the `wctomb` function, except that the shift state of the `wctomb` function is not affected. If a code is encountered which does not correspond to any valid multibyte character, the `wcstombs` function returns `(size_t) - 1`. Otherwise, the `wcstombs` function returns the number of bytes modified, not including a terminating null character, if any.

stdnoreturn.h

The `stdnoreturn.h` header file is part of the C18 extensions to the C language, providing support for functions that are known not to return.

It defines the macro `noreturn` only.

string.h

`string.h` declares one type and several functions, and defines one macro useful for manipulating character arrays and other objects treated as character arrays. Various methods are used for determining the lengths of the arrays, but in all cases a `char *` or `void *` argument points to the initial (lowest addresses) character of the array. If an array is written beyond the end of an object, the behaviour is undefined.

memcpy

```
void *memcpy(void * s1, const void * s2, size_t n)
```

Copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. If copying takes place between objects that overlap, the behaviour is undefined.

Returns: the value of *s1*.

memmove

```
void *memmove(void * s1, const void * s2, size_t n)
```

Copies *n* characters from the object pointed to by *s2* into the object pointed to by *s1*. Copying takes place as if the *n* characters from the object pointed to by *s2* are first copied into a temporary array of *n* characters that does not overlap the objects pointed to by *s1* and *s2*, and then the *n* characters from the temporary array are copied into the object pointed to by *s1*.

Returns: the value of *s1*.

strcpy

```
char *strcpy(char * s1, const char * s2)
```

Copies the string pointed to by *s2* (including the terminating null character) into the array pointed to by *s1*. If copying takes place between objects that overlap, the behaviour is undefined.

Returns: the value of *s1*.

strncpy

```
char *strncpy(char * s1, const char * s2, size_t n)
```

Copies not more than *n* characters (characters that follow a null character are not copied) from the array pointed to by *s2* into the array pointed to by *s1*. If copying takes place between objects that overlap, the behaviour is undefined. If terminating nul has not been copied in chars, no term nul is placed in *s2*.

Returns: the value of *s1*.

strcat

```
char *strcat(char * s1, const char * s2)
```

Appends a copy of the string pointed to by *s2* (including the terminating null character) to the end of the string pointed to by *s1*. The initial character of *s2* overwrites the null character at the end of *s1*.

Returns: the value of *s1*.

strncat

```
char *strncat(char * s1, const char * s2, size_t n)
```

Appends not more than *n* characters (a null character and characters that follow it are not appended) from the array pointed to by *s2* to the end of the string pointed to by *s1*. The initial character of *s2* overwrites the null character at the end of *s1*. A terminating null character is always appended to the result.

Returns: the value of *s1*.

The sign of a non-zero value returned by the comparison functions is determined by the sign of the difference between the values of the first pair of characters (both interpreted as unsigned char) that differ in the objects being compared.

memcmp

```
int memcmp(const void * s1, const void * s2, size_t n)
```

Compares the first *n* characters of the object pointed to by *s1* to the first *n* characters of the object pointed to by *s2*.

Returns: an integer greater than, equal to, or less than zero, depending on whether the object pointed to by *s1* is greater than, equal to, or less than the object pointed to by *s2*.

strcmp

```
int strcmp(const char * s1, const char * s2)
```

Compares the string pointed to by *s1* to the string pointed to by *s2*.

Returns: an integer greater than, equal to, or less than zero, depending on whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*.

strncmp

```
int strncmp(const char * s1, const char * s2, size_t n)
```

Compares not more than *n* characters (characters that follow a null character are not compared) from the array pointed to by *s1* to the array pointed to by *s2*.

Returns: an integer greater than, equal to, or less than zero, depending on whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2*.

strcoll

```
int strcoll(const char * s1, const char * s2)
```

Compares the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the LC_COLLATE category of the current locale.

Returns: an integer greater than, equal to, or less than zero, depending on whether the string pointed to by *s1* is greater than, equal to, or less than the string pointed to by *s2* when both are interpreted as appropriate to the current locale.

strxfrm

```
size_t strxfrm(char * s1, const char * s2, size_t n)
```

Transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation function is such that if the `strcmp` function is applied to two transformed strings, it returns a value greater than, equal to or less than zero, corresponding to the result of the `strcoll` function applied to the same two original strings. No more than *n* characters are placed into the resulting array pointed to by *s1*, including the terminating null character. If *n* is zero, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behaviour is undefined.

Returns: The length of the transformed string is returned (not including the terminating null character). If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate.

memchr

```
void *memchr(const void * s, int c, size_t n)
```

Locates the first occurrence of *c* (converted to an unsigned char) in the initial *n* characters (each interpreted as unsigned char) of the object pointed to by *s*.

Returns: a pointer to the located character, or a null pointer if the character does not occur in the object.

strchr

```
char *strchr(const char * s, int c)
```

Locates the first occurrence of *c* (converted to a char) in the string pointed to by *s* (including the terminating null character). The BSD UNIX name for this function is `index()`.

Returns: a pointer to the located character, or a null pointer if the character does not occur in the string.

strcspn

```
size_t strcspn(const char * s1, const char * s2)
```

Computes the length of the initial segment of the string pointed to by *s1* which consists entirely of characters not from the string pointed to by *s2*. The terminating null character is not considered part of *s2*.

Returns: the length of the segment.

strpbrk

```
char *strpbrk(const char * s1, const char * s2)
```

Locates the first occurrence in the string pointed to by *s1* of any character from the string pointed to by *s2*.

Returns: returns a pointer to the character, or a null pointer if no character from *s2* occurs in *s1*.

strrchr

```
char *strrchr(const char * s, int c)
```

Locates the last occurrence of *c* (converted to a char) in the string pointed to by *s*. The terminating null character is considered part of the string. The BSD UNIX name for this function is `rindex()`.

Returns: returns a pointer to the character, or a null pointer if *c* does not occur in the string.

strspn

```
size_t strspn(const char * s1, const char * s2)
```

Computes the length of the initial segment of the string pointed to by *s1* which consists entirely of characters from the string pointed to by *s2*.

Returns: the length of the segment.

strstr

```
char *strstr(const char * s1, const char * s2)
```

Locates the first occurrence in the string pointed to by *s1* of the sequence of characters (excluding the terminating null character) in the string pointed to by *s2*.

Returns: a pointer to the located string, or a null pointer if the string is not found.

strtok

```
char *strtok(char * s1, const char * s2)
```

A sequence of calls to the `strtok` function breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a character from the string pointed to by *s2*. The first call in the sequence has *s1* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *s2* may be different from call to call. The first call in the sequence searches for the first character that is not contained in the current separator string *s2*. If no such character is found, then there are no tokens in *s1* and the `strtok` function returns a null pointer. If such a character is found, it is the start of the first token. The `strtok` function then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searches for a token will fail. If such a character is found, it is overwritten by a null character, which terminates the current token. The `strtok` function saves a pointer to the

following character, from which the next search for a token will start. Each subsequent call, with a null pointer as the value for the first argument, starts searching from the saved pointer and behaves as described above.

Returns: pointer to the first character of a token, or a null pointer if there is no token.

memset

```
void *memset(void * s, int c, size_t n)
```

Copies the value of *c* (converted to an unsigned char) into each of the first *n* characters of the object pointed to by *s*.

Returns: the value of *s*.

strerror

```
char *strerror(int errnum)
```

Maps the error number in *errnum* to an error message string.

Returns: a pointer to the string, the contents of which are implementation-defined. Under RISC OS the strings for the given *errnums* are as follows:

- 0 No error (errno = 0)
- EDOM EDOM – function argument out of range
- ERANGE ERANGE – function result not representable
- ESIGNUM ESIGNUM – illegal signal number to `signal()` or `raise()`
- EILSEQ EILSEQ – character encoding error
- EOVERFLOW EOVERFLOW – too large for data structure
- EFBIG EFBIG – data written to file lost due to exceeding file size limit
- others Error code (errno) has no associated message).

The array pointed to may not be modified by the program, but may be overwritten by a subsequent call to the `strerror` function.

strlen

```
size_t strlen(const char * s)
```

Computes the length of the string pointed to by *s*.

Returns: the number of characters that precede the terminating null character.

tgmath.h

The intent of the `tgmath.h` library header is to provide the core set of maths functions which will automatically select the appropriate implementation depending on the type of the variables passed to the function. When using `tgmath.h` header you should not include `math.h` or `complex.h` in your program, this will be done for you.

The rules for determining which implementation to invoke are as follows:

- First, if any argument for generic parameters has type long double, the type determined is long double
- Otherwise, if any argument for generic parameters has type double or is of integer type, the type determined is double
- Otherwise, the type determined is float.

For example:

```
#include <tgmath.h>
coordinate_t value = 42.0; /* Defined as a float */
result = sqrt(value);
```

is equivalent to

```
#include <math.h>
coordinate_t value = 42.0;
result = sqrtf(value);
```

However, if the program subsequently decides to upgrade all its coordinates to use doubles, a simple recompilation will cause `tgmath.h` to swap the function calls over.

The functions implemented in this manner are:

<code>acos</code>	<code>asin</code>	<code>atan</code>	<code>acosh</code>	<code>asinh</code>
<code>atanh</code>	<code>cos</code>	<code>sin</code>	<code>tan</code>	<code>cosh</code>
<code>sinh</code>	<code>tanh</code>	<code>exp</code>	<code>log</code>	<code>pow</code>
<code>sqrt</code>	<code>fabs</code>	<code>atan2</code>	<code>cbrt</code>	<code>ceil</code>
<code>copysign</code>	<code>erf</code>	<code>erfc</code>	<code>exp2</code>	<code>expm1</code>
<code>fdim</code>	<code>floor</code>	<code>fma</code>	<code>fmax</code>	<code>fmin</code>
<code>fmod</code>	<code>frexp</code>	<code>hypot</code>	<code>ilogb</code>	<code>ldexp</code>
<code>lgamma</code>	<code>llrint</code>	<code>llround</code>	<code>log10</code>	<code>log1p</code>
<code>log2</code>	<code>logb</code>	<code>lrint</code>	<code>lround</code>	<code>nearbyint</code>
<code>nextafter</code>	<code>nexttoward</code>	<code>remainder</code>	<code>remquo</code>	<code>rint</code>
<code>round</code>	<code>scalbn</code>	<code>scalbln</code>	<code>tgamma</code>	<code>trunc</code>
<code>carg</code>	<code>cimag</code>	<code>conj</code>	<code>cproj</code>	<code>creal</code>

threads.h

Multiple threads of execution are an optional part of the C18 language extensions which are not currently implemented by the compiler or the C library. Furthermore, this header file is not built into the compiler and must be provided as a supplement on disc.

time.h

`time.h` declares two macros, four types and several functions for manipulating time. Many functions deal with a calendar time that represents the current date (according to the Gregorian calendar) and time. Some functions deal with local time, which is the calendar time expressed for some specific time zone, and with Daylight Saving Time, which is a temporary change in the algorithm for determining local time.

struct tm

`struct tm` holds the components of a calendar time called the broken-down time. The value of `tm_isdst` is positive if Daylight Saving Time is in effect, zero if Daylight Saving Time is not in effect, and negative if the information is not available.

```
struct tm {
    int tm_sec;      /* seconds after the minute, 0 to 60
                      (0-60 allows for the occasional leap
                      second) */
    int tm_min       /* minutes after the hour, 0 to 59 */
    int tm_hour      /* hours since midnight, 0 to 23 */
    int tm_mday      /* day of the month, 0 to 31 */
    int tm_mon       /* months since January, 0 to 11 */
    int tm_year      /* years since 1900 */
    int tm_wday      /* days since Sunday, 0 to 6 */
    int tm_yday      /* days since January 1, 0 to 365 */
    int tm_isdst     /* Daylight Saving Time flag */
};
```

clock

```
clock_t clock(void)
```

Determines the processor time used.

Returns: the implementation's best approximation to the processor time used by the program since program invocation. The time in seconds is the value returned, divided by the value of the macro `CLOCKS_PER_SEC`. The value `(clock_t)-1` is returned if the processor time used is not available. In the desktop, `clock()` returns all processor time, not just that of the program.

difftime

```
double difftime(time_t time1, time_t time0)
```

Computes the difference between two calendar times: *time1* - *time0*. Returns: the difference expressed in seconds as a double.

mktime

```
time_t mktime(struct tm * timeptr)
```

Converts the broken-down time, expressed as local time, in the structure pointed to by *timeptr* into a calendar time value with the same encoding as that of the values returned by the *time* function. The original values of the *tm_wday* and *tm_yday* components of the structure are ignored, and the original values of the other components are not restricted to the ranges indicated above. On successful completion, the values of the *tm_wday* and *tm_yday* structure components are set appropriately, and the other components are set to represent the specified calendar time, but with their values forced to the ranges indicated above; the final value of *tm_mday* is not set until *tm_mon* and *tm_year* are determined.

Returns: the specified calendar time encoded as a value of type *time_t*. If the calendar time cannot be represented, the function returns the value *(time_t) -1*.

time

```
time_t time(time_t * timer)
```

Determines the current calendar time. The encoding of the value is unspecified.

Returns: the implementation's best approximation to the current calendar time. The value *(time_t) -1* is returned if the calendar time is not available. If *timer* is not a null pointer, the return value is also assigned to the object it points to.

asctime

```
char *asctime(const struct tm * timeptr)
```

Converts the broken-down time in the structure pointed to by *timeptr* into a string in the style *Sun Sep 16 01:03:52 1973\n\n0*.

Returns: a pointer to the string containing the date and time.

ctime

```
char *ctime(const time_t * timer)
```

Converts the calendar time pointed to by *timer* to local time in the form of a string. It is equivalent to `asctime(localtime(timer))`.

Returns: the pointer returned by the `asctime` function with that broken-down time as argument.

gmtime

```
struct tm *gmtime(const time_t * timer)
```

Converts the calendar time pointed to by *timer* into a broken-down time, expressed as Greenwich Mean Time (GMT).

Returns: a pointer to that object or a null pointer if GMT is not available.

localtime

```
struct tm *localtime(const time_t * timer)
```

Converts the calendar time pointed to by *timer* into a broken-down time, expressed a local time.

Returns: a pointer to that object.

strftime

```
size_t strftime(char * s, size_t maxsize, const char *  
                format, const struct tm * timeptr)
```

Places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. The format string consists of zero or more directives and ordinary characters. A directive consists of a % character followed by a character that determines the directive's behaviour. All ordinary characters (including the terminating null character) are copied unchanged into the array. No more than *maxsize* characters are placed into the array. Each directive is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the `LC_TIME` category of the current locale and by the values contained in the structure pointed to by *timeptr*.

Directive	Replaced by
%a	the locale's abbreviated weekday name
%A	the locale's full weekday name
%b	the locale's abbreviated month name
%B	the locale's full month name
%c	the locale's appropriate date and time representation
%C	the century as a decimal number (00-99)
%d	the day of the month as a decimal number (01-31)
%D	the expansion of %m/%d/%y
%e	as for %d, except the leading zero becomes a leading space
%F	the expansion of %Y-%m-%d which is the ISO 8601 standard form
%g	the last two digits of the ISO 8601 week based year as a decimal number (00-99)
%G	the ISO 8601 week based year as a decimal number (eg 2015)
%h	as for %b
%H	the hour (24-hour clock) as a decimal number (00-23)
%I	the hour (12-hour clock) as a decimal number (01-12)
%j	the day of the year as a decimal number (001-366)
%m	the month as a decimal number (01-12)
%M	the minute as a decimal number (00-61)
%p	the locale's equivalent of either AM or PM designation associated with a 12-hour clock
%r	the locale's 12-hour clock time
%R	the expansion of %H:%M
%S	the second as a decimal number (00-61)
%t	a tab
%T	the expansion of %H:%M:%S (the ISO 8601 time format)
%u	the ISO 8601 weekday as a decimal number (1-7), where Monday is 1
%U	the week number of the year (Sunday as the first day of week 1) as a decimal number (00-53)
%w	the weekday as a decimal number (0(Sunday) -6)
%W	the week number of the year (Monday as the first day of week 1) as a decimal number (00-53)
%x	the locale's appropriate date representation
%X	the locale's appropriate time representation
%y	the year without century as a decimal number (00-99)
%Y	the year with century as a decimal number
%z	the offset from UTC in the ISO 8601 format (eg -0430), or by no characters if no time zone is determinable
%Z	the time zone name or abbreviation, or by no character if no time zone is determinable
%%	a % symbol.

The optional `E` and `O` modifiers to change the above directives to locale alternative representations are ignored. If a directive is not one of the above, the behaviour is undefined.

Returns: If the total number of resulting characters including the terminating null character is not more than *maxsize*, the `strftime` function returns the number of characters placed into the array pointed to by *s* not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

timespec_get

```
int timespec_get(struct timespec * ts, int base)
```

Sets the interval pointed to by *ts* to hold the current calendar time based on the specified time base. If *base* is `TIME_UTC` this is the number of seconds and nanoseconds (rounded to the resolution of the system clock) since an implementation defined epoch. The epoch used by RISC OS is 1st January 1970.

Returns: if successful it returns the non-zero value *base*, otherwise it returns 0.

uchar.h

The extended unicode character handling utilities are part of the C18 language extensions which are not currently implemented as detailed in the section C18 *non-compliances* on page 118. The header file is provided for compatibility with source code ported from other platforms, but will cause a link time error if any of its functions are used.

wchar.h

The extended multibyte wide character handling utilities are part of the C99 language extensions which are not currently implemented as detailed in the section C99 *non-compliances* on page 110. The header file is provided for compatibility with source code ported from other platforms, but will cause a link time error if any of its functions are used.

wctype.h

The extended multibyte wide character classification utilities are part of the C99 language extensions which are not currently implemented as detailed in the section C99 *non-compliances* on page 110. The header file is provided for compatibility with source code ported from other platforms, but will cause a link time error if any of its functions are used.

7

The ANSI library

The ANSI library is a stand-alone version of the shared C library that contains a few extra functions useful in debugging and profiling your code. You should use it for development only, using the shared C library in any final product.

This chapter describes the extra functions provided by the ANSI library. For details of the other functions, see the chapter *The C library* on page 135.

Extra functions

`__heap_checking_on_all_allocates` **`__heap_checking_on_all_deallocates`**

```
void __heap_checking_on_all_allocates (int on);  
void __heap_checking_on_all_deallocates (int on);
```

Calling these functions with a non-zero argument causes `malloc()` and `free()` respectively to check the consistency of the C heap on every call, rather than only when the heap is coalesced. It is especially useful for tracking down exactly where memory corruption is occurring. This feature is disabled by passing an argument of zero.

`_mapstore` **`_fmapstore`**

```
void _mapstore (void);  
void _fmapstore (char *filename);
```

These functions write profiling information for a program to `stderr` or `filename` respectively, if the program has been compiled with profiling enabled.

Part 3 – C++ language issues



8

C++ implementation details

This chapter describes implementation specific behaviour of the C++ Language System. Implementation specific behaviours can be categorised as follows:

- 1 Behaviour that the *Reference Manual* defines as ‘implementation dependent’
- 2 Behaviour that depends on the underlying C compiler or preprocessor used with Release 3.0
- 3 Properties that are defined in the standard header files `stddef.h`, `limits.h`, and `stdlib.h`
- 4 Translation limits
- 5 Language constructs that are not implemented in this release.

This chapter addresses categories 1, 2, 4, and 5. For details about properties defined in the standard header files (category 3), see the headers themselves. Additional information about constructs that are not implemented is provided in the appendix C++ *errors and warnings* on page 399, which contains an alphabetical listing of the ‘not implemented’ error messages.

The ordering and numbering of sections in this chapter corresponds to the order and numbering of the related sections in the *Reference Manual*. The section *Translation Limits* below (which does not have a corresponding section in the *Reference Manual*) precedes the numbered sections.

Translation Limits

Release 3.0 of the Acorn C++ Language System imposes the following translation limits:

- 50 nesting levels of compound statements
- 10 nesting levels of linkage declarations
- 4088 characters in a token
- 22222 virtual functions in a class
- 10000 identifiers generated by the implementation.

Additional translation limits may be inherited from the underlying C compiler and preprocessor.

Identifiers (2.3)

Identifiers reserved by Release 3.0

Release 3.0 reserves identifiers that contain a sequence of two underscores for its own use. In addition, identifiers reserved in the ANSI C standard are also reserved by Release 3.0. Under the +w option, identifiers with double underscores result in a warning in Release 3.0.

Character Constants (2.5.2)

Value of multicharacter constants

The *Reference Manual* states that the value of a multicharacter constant, such as 'abcd', is implementation dependent. Release 3.0 passes these constants to the underlying C compiler, which determines their values. A multicharacter constant containing more characters than `sizeof(int)` is reported as an error by Release 3.0.

Value of (single) character constants

The *Reference Manual* states that the value of a character constant is implementation dependent if it exceeds that of the largest `char`. Release 3.0 accepts octal and hexadecimal character literals that do not fit in a `char`. It uses the low order bits that make up the value of the constant. For example, the octal character constant '\777' is treated as '\377'. The hexadecimal character constant '\x123' is treated as '\x23'.

Wide character constants

Release 3.0 does not implement wide character constants, such as `L'ab'`. A 'not implemented' error message is reported.

Floating Constants (2.5.3)

Long double floating constants

When compiling with the +a0 option, Release 3.0 removes an `l` or `L` suffix from a floating constant before passing the constant to the underlying C compiler. Under the +a1 option such a constant is passed unchanged to the underlying C compiler. In either case, the constant is considered to be of type `long double` for purposes of resolving overloaded function calls.

String Literals (2.5.4)

Distinct string literals

The *Reference Manual* states that it is implementation dependent whether all string literals are distinct. Release 3.0 does not attempt to detect cases where string literals could be represented as overlapping objects. The underlying C compiler may, however, detect such cases and attempt to overlap their storage.

Wide character strings

Release 3.0 does not implement wide character strings, such as `L"abcd"`. A 'not implemented' error message is reported.

Start and Termination (3.4)

Type of `main()`

The *Reference Manual* states that the type of `main()` is implementation dependent. Release 3.0 itself does not impose any restrictions on the type of `main()`, but the underlying C compiler or the target environment may impose such restrictions.

Linkage of `main()`

The Acorn C++ Language System treats `main()` as if its linkage were `extern "C"`.

Fundamental Types (3.6.1)

Signed integral types

Release 3.0 does not implement the type specifier `signed`; it issues a warning and proceeds as though the specifier `signed` had not appeared.

Long double type

When Release 3.0 is invoked with the `+a0` option, the type `long double` is considered to be the same size and precision as the type `double` in the underlying C compiler. Under the `+a1` option, `long double` is passed to the underlying C compiler as `long double`. In either case, type `long double` is considered a distinct type for purposes of resolving overloaded function declarations and invocations.

Alignment requirements

Release 3.0 does not impose any alignment restrictions when allocating objects of a particular type. Such restrictions, if they exist, are enforced by the underlying C compiler.

Integral Conversions (4.2)

Conversion to a signed type

When a value of an integral type is converted to a signed integral type with fewer bits in the representation, Release 3.0 issues a warning message if the `+w` option is specified. The runtime behaviour of such a conversion depends on the treatment of the conversion by the underlying C compiler.

Expressions (5)

Overflow and divide check

The *Reference Manual* states that the handling of overflow and divide check in expression evaluation is implementation dependent. When the second operand of a division or modulus operator is known to be zero at compile time, Release 3.0 reports an error. Overflow and other divide check conditions are handled by the underlying C compiler and execution environment.

Function Call (5.2.2)

Evaluation order

The *Reference Manual* states that the order of evaluation of arguments to a function call is implementation dependent; similarly, the order of evaluation of the postfix expression, which designates the function to be called, and the argument expression list are implementation dependent. In both cases the order depends on the treatment by the underlying C compiler.

Explicit Type Conversion (5.4)

Explicit conversions between pointer and integral types

The *Reference Manual* states that the value obtained by explicitly converting a pointer to an integral type large enough to hold it is implementation dependent. This behaviour is defined by the underlying C compiler. Similarly, the behaviour when explicitly converting an integer to a pointer depends on the underlying C compiler.

Multiplicative Operators (5.6)

Sign of the remainder

The *Reference Manual* states that the sign of the result of the modulus operator is non-negative if both operands are non-negative; otherwise, the sign of the result is implementation dependent. This behaviour depends on the underlying C compiler except when the values of both operands are known at compile time. In this case, the sign of the result is the same as the sign of the numerator.

Shift Operators (5.8)

Result of right shift

The *Reference Manual* states that the result of a right shift when the left operand is a signed type with a negative value is implementation dependent. This behaviour depends on the underlying C compiler.

Relational Operators (5.9)

Pointer comparisons

According to the *Reference Manual*, certain pointer comparisons are implementation dependent. For Release 3.0, the results of these comparisons depend on the underlying C compiler.

Storage Class Specifiers (7.1.1)

Inline functions

The *Reference Manual* states that the `inline` specifier is a hint to the compiler.

When compiling with the `+d` option, Release 3.0 always generates out-of-line calls to inline functions.

Type Specifiers (7.1.6)

Volatile

Release 3.0 does not implement the type specifier `volatile`. If it is applied to a member function, a 'not implemented' error message is issued; otherwise it is ignored and a warning message is issued.

Signed

Release 3.0 does not implement the type specifier `signed`; it is ignored and a warning message is issued.

Asm Declarations (7.3)

Effect of an asm declaration

Release 3.0 passes `asm` declarations to the underlying C compiler without modification. However, the compiler supplied with Acorn C/C++ will fault them.

Linkage Specifications (7.4)

Languages supported

Release 3.0 supports linkage to C and C++.

Linkage to functions

The effect of a "C" linkage specification (`extern "C"`) on a function that is not a member function is that the function name is not encoded with type information, as is otherwise done for C++ functions. Member functions are not affected by linkage specifications.

Linkage to non-functions

The C linkage specification (`extern "C"`), when applied to a non-function declaration, does not affect the C code generated.

Class Members (9.2)

Allocation of non-static data members

The *Reference Manual* states that the order of allocation of non-static data members across *access-specifiers* is implementation dependent. Release 3.0 allocates non-static data members in declaration order.

Bitfields (9.6)

Allocation and alignment of bitfields

The *Reference Manual* states that the allocation and alignment of bitfields within a class object is implementation dependent. Responsibility for the allocation and alignment of bitfields rests with the underlying C compiler.

Sign of 'plain' bitfields

Whether the high-order bit position of a 'plain' `int` bitfield is treated as a sign bit depends on the behaviour of the underlying C compiler.

Multiple Base Classes (10.1)

Allocation of base classes

The *Reference Manual* states that the order in which storage is allocated for base classes is implementation dependent. For non-virtual base classes, Release 3.0 allocates storage in the order that they are mentioned in the derived class declaration.

Argument Matching (13.2)

Integral arguments

The type of the result of an integral promotion (4.1) depends on the execution environment, as does the type of an unsuffixed integer constant (2.5.1). Consequently, the determination of which overloaded function to call may also depend on the execution environment, as illustrated by an example in 13.2 of the *Reference Manual*.

Exception Handling (experimental) (15)

Release 3.0 does not implement exception handling. The keyword `catch` is reserved for future use. A ‘not implemented’ error message is reported if `catch` is seen.

Predefined Names (16.10)

Predefined macros

The following macros are defined by Release 3.0:

`__cplusplus` The decimal constant 1.

Other macros may be predefined by the underlying preprocessor.

9

The Streams library

The Streams library is a part of the C++ library, ported from that supplied with AT&T's CFront product. The only significant change made in porting the library is the handling of file modes, because of the differences between filing systems in RISC OS and UNIX.

Introduction

iostream – buffering, formatting and input/output

Synopsis

```
#include <iostream.h>
class streambuf ;
class ios ;
class istream : virtual public ios ;
class ostream : virtual public ios ;
class iostream : public istream, public ostream ;
class istream_withassign : public istream ;
class ostream_withassign : public ostream ;
class iostream_withassign : public iostream ;

class Iostream_init ;

extern istream_withassign cin ;
extern ostream_withassign cout ;
extern ostream_withassign cerr ;
extern ostream_withassign clog ;

#include <fstream.h>
class filebuf : public streambuf ;
class fstream : public iostream ;
class ifstream : public istream ;
class ofstream : public ostream ;

#include <strstream.h>
class strstreambuf : public streambuf ;
class istrstream : public istream ;
class ostrstream : public ostream ;

#include <stdiostream.h>
class stdiobuf : public streambuf ;
class stdiostream : public ios ;
```

Description

The C++ iostream package declared in `iostream.h` and other header files consists primarily of a collection of classes. Although originally intended only to support input/output, the package now supports related activities such as incore formatting.

In the iostream sections, *character* refers to a value that can be held in either a `char` or `unsigned char`. When functions that return an `int` are said to return a character, they return a positive value. Usually such functions can also return `EOF`

(-1) as an error indication. The piece of memory that can hold a character is referred to as a *byte*. Thus, either a `char*` or an `unsigned char*` can point to an array of bytes.

The `iostream` package consists of several core classes, which provide the basic functionality for I/O conversion and buffering, and several specialised classes derived from the core classes. Both groups of classes are listed below.

Core Classes

The core of the `iostream` package comprises the following classes:

`streambuf`

This is the base class for buffers. It supports insertion (also known as *storing* or *putting*) and extraction (also known as *fetching* or *getting*) of characters. Most members are inlined for efficiency. The public interface of class `streambuf` is described in *streambuf – public* on page 274, and the protected interface (for derived classes) is described in *streambuf – protected* on page 266.

`ios`

This class contains state variables that are common to the various stream classes, for example, error states and formatting states. See *ios* on page 237.

`istream`

This class supports formatted and unformatted conversion from sequences of characters fetched from `streambufs`. See *istream* on page 248.

`ostream`

This class supports formatted and unformatted conversion to sequences of characters stored into `streambufs`. See *ostream* on page 259.

`iostream`

This class combines `istream` and `ostream`. It is intended for situations in which bidirectional operations (inserting into and extracting from a single sequence of characters) are desired. See *ios* on page 237.

istream_withassign
ostream_withassign
iostream_withassign

These classes add assignment operators and a constructor with no operands to the corresponding class **without assignment**. The predefined streams (see below) `cin`, `cout`, `cerr`, and `clog`, are objects of these classes. See *istream* on page 248, *ostream* on page 259, and *ios* on page 237.

Iostream_init

This class is present for technical reasons relating to initialisation. It has no public members. The `Iostream_init` constructor initialises the predefined streams (listed below). Because an object of this class is declared in the `iostream.h` header file, the constructor is called once each time the header is included (although the real initialisation is only done once), and therefore the predefined streams will be initialised before they are used. In some cases, global constructors may need to call the `Iostream_init` constructor explicitly to ensure the standard streams are initialised before they are used.

Predefined streams

The following streams are predefined:

cin

The standard input (file descriptor 0).

cout

The standard output (file descriptor 1).

cerr

Standard error (file descriptor 2). Output through this stream is unit-buffered, which means that characters are flushed after each inserter operation. (See `osfx()` on page 261 in *ostream*, and *unitbuf* on page 242 in *ios*.)

clog

This stream is also directed to file descriptor 2, but unlike `cerr` its output is buffered.

Note: `cin`, `cerr`, and `clog` are tied to `cout` so that any use of these will cause `cout` to be flushed.

In addition to the core classes enumerated above, the `iostream` package contains additional classes derived from them and declared in other headers. Programmers may use these, or may choose to define their own classes derived from the core `iostream` classes.

Classes derived from `streambuf`

Classes derived from `streambuf` define the details of how characters are produced or consumed. Derivation of a class from `streambuf` (the *protected interface*) is discussed in *streambuf – protected* on page 266. The available buffer classes are:

`filebuf`

This buffer class supports I/O through file descriptors. Members support opening, closing, and seeking. Common uses do not require the program to manipulate file descriptors. See *filebuf* on page 229.

`stdiobuf`

This buffer class supports I/O through stdio `FILE` structs. It is intended for use when mixing C and C++ code. New code should prefer to use `filebufs`. See *stdiobuf* on page 265.

`strstreambuf`

This buffer class stores and fetches characters from arrays of bytes in memory (i.e. strings). See *strstreambuf* on page 282.

Classes derived from `istream`, `ostream`, and `iostream`

Classes derived from `istream`, `ostream`, and `iostream` specialise the core classes for use with particular kinds of `streambufs`. These classes are:

`ifstream`

`ofstream`

`fstream`

These classes support formatted I/O to and from files. They use a `filebuf` to do the I/O. Common operations (such as opening and closing) can be done directly on streams without explicit mention of `filebufs`. See *fstream* on page 233.

istream

ostream

These classes support ‘in core’ formatting. They use a `strstreambuf`. See *strstream* on page 279.

stdiostream

This class specialises `iostream` for stdio FILES. See `stdiostream.h`.

See also

ios (page 237), *streambuf – public* (page 274), *streambuf – protected* (page 266), *filebuf* (page 229), *stdiobuf* (page 265), *strstreambuf* (page 282), *istream* (page 248), *ostream* (page 259), *fstream* (page 233), *strstream* (page 279), *manipulators* (page 255)

filebuf

filebuf – buffer for file I/O

Synopsis

```
#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
    enum    seek_dir { beg, cur, end };
    enum    open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
    // and lots of other stuff; see ios on page 237
} ;

#include <fstream.h>

class filebuf : public streambuf {
public:
    static const int openprot ; /* default protection for open */

    filebuf() ;
    ~filebuf() ;
    filebuf(int d);
    filebuf(int d, char* p, int len) ;

    filebuf*    attach(int d) ;
    filebuf*    close();
    int         fd();
    int         is_open();
    filebuf*    open(char *name, int omode, int prot=openprot) ;
    streampos   seekoff(streamoff, seek_dir, int omode) ;
    streampos   seekpos(streampos, int omode) ;
    streambuf*  setbuf(char* p, int len) ;
    int         sync() ;
};
```

Description

filebufs specialise streambufs to use a file as a source or sink of characters. Characters are consumed by doing writes to the file, and are produced by doing reads. When the file is seekable, a filebuf allows seeks. At least 4 characters of putback are guaranteed. When the file permits reading and writing, the filebuf permits both storing and fetching. No special action is required between gets and puts (unlike stdio). A filebuf that is connected to a file descriptor is said to be open.

Under RISC OS openprot is ignored.

The *reserve area* (or *buffer*; see *streambuf – public* on page 274 and *streambuf – protected* on page 266) is allocated automatically if one is not specified explicitly with a constructor or a call to `setbuf()`. `filebuf`s can also be made unbuffered with certain arguments to the constructor or `setbuf()`, in which case a system call is made for each character that is read or written. The `get` and `put` pointers into the reserve area are conceptually tied together; they behave as a single pointer. Therefore, the descriptions below refer to a single `get/put` pointer.

In the descriptions below, assume:

- *f* is a `filebuf`.
- *pfb* is a `filebuf*`.
- *psb* is a `streambuf*`.
- *i*, *d*, *len*, and *prot* are ints.
- *name* and *ptr* are `char*`s.
- *mode* is an int representing an `open_mode`.
- *off* is a `streamoff`.
- *p* and *pos* are `streampos`'s.
- *dir* is a `seek_dir`.

Constructors

filebuf()

Constructs an initially closed `filebuf`.

filebuf(*d*)

Constructs a `filebuf` connected to file descriptor *d*.

filebuf(*d*, *p*, *len*)

Constructs a `filebuf` connected to file descriptor *d* and initialised to use the reserve area starting at *p* and containing *len* bytes. If *p* is null or *len* is zero or less, the `filebuf` will be unbuffered.

Members

pfb=f.attach(d)

Connects *f* to an open file descriptor, *d*. `attach()` normally returns `&f`, but returns 0 if *f* is already open.

pfb=f.close()

Flushes any waiting output, closes the file descriptor, and disconnects *f*. Unless an error occurs, *f*'s error state will be cleared. `close()` returns `&f` unless errors occur, in which case it returns 0. Even if errors occur, `close()` leaves the file descriptor and *f* closed.

i=f.fd()

Returns *i*, the file descriptor *f* is connected to. If *f* is closed, *i* is EOF.

i=f.is_open()

Returns non-zero when *f* is connected to a file descriptor, and zero otherwise.

pfb=f.open(name, mode, prot)

Opens file *name* and connects *f* to it. If the file does not already exist, an attempt is made to create it, unless `ios::nocreate` is specified in *mode*. Under RISC OS, *prot* is ignored. Failure occurs if *f* is already open. `open()` normally returns `&f`, but if an error occurs it returns 0. The members of `open_mode` are bits that may be OR'd together. (Because the OR'ing returns an `int`, `open()` takes an `int` rather than an `open_mode` argument.) The meanings of these bits in *mode* are described in detail in *fstream* on page 233.

p=f.seekoff(off, dir, mode)

Moves the get/put pointer as designated by *off* and *dir*. It may fail if the file that *f* is attached to does not support seeking, or if the attempted motion is otherwise invalid (such as attempting to seek to a position before the beginning of file). *off* is interpreted as a count relative to the place in the file specified by *dir* as described in *streambuf-public* on page 274. *mode* is ignored. `seekoff()` returns *p*, the new position, or EOF if a failure occurs. The position of the file after a failure is undefined.

p=f.seekpos(pos, mode)

Moves the file to a position *pos* as described in *streambuf – public* on page 274. *mode* is ignored. *seekpos()* normally returns *pos*, but on failure it returns EOF.

psb=f.setbuf(ptr, len)

Sets up the reserve area as *len* bytes beginning at *ptr*. If *ptr* is null or *len* is less than or equal to 0, *f* will be unbuffered. *setbuf()* normally returns *&f*. However, if *f* is open and a buffer has been allocated, no changes are made to the reserve area or to the buffering status, and *setbuf()* returns 0.

i=f.sync()

Attempts to force the state of the get/put pointer of *f* to agree (be synchronised) with the state of the file *f*. *fd()*. This means it may write characters to the file if some have been buffered for output or attempt to reposition (seek) the file if characters have been read and buffered for input. Normally, *sync()* returns 0, but it returns EOF if synchronisation is not possible.

Sometimes it is necessary to guarantee that certain characters are written together. To do this, the program should use *setbuf()* (or a constructor) to guarantee that the reserve area is at least as large as the number of characters that must be written together. It can then call *sync()*, then store the characters, then call *sync()* again.

See also

streambuf – public (page 274), *streambuf – protected* (page 266), *fstream* (page 233).

fstream

fstream – iostream and streambuf specialised to files

Synopsis

```
#include <fstream.h>

typedef long streamoff, streampos;
class ios {
public:
    enum    seek_dir { beg, cur, end } ;
    enum    open_mode { in, out, ate, app, trunc, ncreate, noreplace } ;
    enum    io_state { goodbit=0, eofbit, failbit, badbit } ;
    // and lots of other stuff; see ios on page 237
};

class ifstream : istream {
    ifstream() ;
    ~ifstream() ;
    ifstream(const char* name, int =ios::in,
              int prot =filebuf::openprot) ;
    ifstream(int fd) ;
    ifstream(int fd, char* p, int l) ;

    void      attach(int fd) ;
    void      close() ;
    void      open(char* name, int =ios::in,
                  int prot=filebuf::openprot) ;
    filebuf*  rdbuf() ;
    void      setbuf(char* p, int l) ;
};

class ofstream : ostream {
    ofstream() ;
    ~ofstream() ;
    ofstream(const char* name, int =ios::out,
              int prot =filebuf::openprot) ;
    ofstream(int fd) ;
    ofstream(int fd, char* p, int l) ;

    void      attach(int fd) ;
    void      close() ;
    void      open(char* name, int =ios::out,
                  int prot=filebuf::openprot) ;
    filebuf*  rdbuf() ;
    void      setbuf(char* p, int l) ;
};
```

```
class fstream : istream {
    fstream() ;
    ~fstream() ;
    fstream(const char* name, int mode,
            int prot =filebuf::openprot) ;
    fstream(int fd) ;
    fstream(int fd, char* p, int l) ;

    void      attach(int fd) ;
    void      close() ;
    void      open(char* name, int mode,
                  int prot=filebuf::openprot) ;
    filebuf*  rdbuf() ;
    void      setbuf(char* p, int l) ;
};
```

Description

`ifstream`, `ofstream`, and `fstream` specialise `istream`, `ostream`, and `iostream`, respectively, to files. That is, the associated `streambuf` will be a `filebuf`.

In the following descriptions, assume:

- *f* is any of `ifstream`, `ofstream`, or `fstream`.
- *pfb* is a `filebuf*`.
- *psb* is a `streambuf*`.
- *name* and *ptr* are `char*`s.
- *i*, *fd*, *len*, and *prot* are ints.
- *mode* is an int representing an `open_mode`.

Constructors

The constructors for `xstream`, where *x* is either `if`, `of`, or `f`, are:

`xstream()`

Constructs an unopened `xstream`.

`xstream(name, mode, prot)`

Constructs an `xstream` and opens file *name* using *mode* as the open mode. Under RISC OS *prot* is ignored. The error state (`io_state`) of the constructed `xstream` will indicate failure in case the open fails.

xstream(*d*)

Constructs an `xstream` connected to file descriptor `d`, which must be already open.

xstream(*d*,*ptr*,*len*)

Constructs an `xstream` connected to file descriptor `d`, and, in addition, initialises the associated `filebuf` to use the `len` bytes at `ptr` as the reserve area. If `ptr` is null or `len` is 0, the `filebuf` will be unbuffered.

Member functions***f*.attach(*d*)**

Connects `f` to the file descriptor `d`. A failure occurs when `f` is already connected to a file. A failure sets `ios::failbit` in `f`'s error state.

***f*.close()**

Closes any associated `filebuf` and thereby breaks the connection of the `f` to a file.

`f`'s error state is cleared except on failure. A failure occurs when the call to `f.rdbuf() -> close()` fails.

***f*.open(*name*,*mode*,*prot*)**

Opens file `name` and connects `f` to it. If the file does not already exist, an attempt is made to create it unless `ios::nocreate` is set. Under RISC OS `prot` is ignored. Failure occurs if `f` is already open, or the call to `f.rdbuf() -> open()` fails. `ios::failbit` is set in `f`'s error status on failure. The members of `open_mode` are bits that may be OR'd together. (Because the OR'ing returns an `int`, `open()` takes an `int` rather than an `open_mode` argument.) The meanings of these bits in `mode` are:

<code>ios::app</code>	A seek to the end of file is performed. Subsequent data written to the file is always added (appended) at the end of file. <code>ios::app</code> implies <code>ios::out</code> .
<code>ios::ate</code>	A seek to the end of the file is performed during the <code>open()</code> . <code>ios::ate</code> does not imply <code>ios::out</code> .
<code>ios::in</code>	The file is opened for input. <code>ios::in</code> is implied by construction and opens of <code>ifstreams</code> . For <code>fstreams</code> it indicates that input operations should be allowed if possible. Is is legal to include <code>ios::in</code> in the modes of

	an <code>ostream</code> in which case it implies that the original file (if it exists) should not be truncated. If the file being opened for input does not exist, the <code>open</code> will fail.
<code>ios::out</code>	The file is opened for output. <code>ios::out</code> is implied by construction and opens of <code>ofstreams</code> . For <code>fstream</code> it says that output operations are to be allowed. <code>ios::out</code> may be specified.
<code>ios::trunc</code>	If the file already exists, its contents will be truncated (discarded). This mode is implied when <code>ios::out</code> is specified (including implicit specification for <code>ofstream</code>) and neither <code>ios::ate</code> nor <code>ios::app</code> is specified.
<code>ios::nocreate</code>	If the file does not already exist, the <code>open()</code> will fail.
<code>ios::noreplace</code>	If the file already exists, the <code>open()</code> will fail. Only valid with <code>ios::out</code> .

`pfb=f.rdbuf()`

Returns a pointer to the `filebuf` associated with `f`. `fstream::rdbuf()` has the same meaning as `istream::rdbuf()` but is typed differently.

`f.setbuf(p,len)`

Has the usual effect of a `setbuf()` (see `filebuf` on page 229), offering space for a reserve area or requesting unbuffered I/O. Normally the returned `psb` is `f.rdbuf()`, but it is 0 on failure. A failure occurs if `f` is open or the call to `f.rdbuf() -> setbuf` fails.

See also

filebuf (page 229), *istream* (page 248), *ios* (page 237), *ostream* (page 259), *streambuf* – *public* (page 274)

ios

ios – input/output formatting

Synopsis

```

#include <iostream.h>

class ios {
public:
    enum    io_state { goodbit=0, eofbit, failbit, badbit };
    enum    open_mode { in, out, ate, app, trunc, nocreate, noreplace };
    enum    seek_dir { beg, cur, end };
    /* flags for controlling format */
    enum    { skipws=01,
              left=02, right=04, internal=010,
              dec=020, oct=040, hex=0100,
              showbase=0200, showpoint=0400,
              uppercase=01000, showpos=02000,
              scientific=04000, fixed=010000,
              unitbuf=020000, stdio=040000 };
    static const long basefield;
    /* dec|oct|hex */
    static const long adjustfield;
    /* left|right|internal */
    static const long floatfield;
    /* scientific|fixed */

public:
                                ios(streambuf*);
    int                        bad();
    static long                bitalloc();
    void                        clear(int state =0);
    int                        eof();
    int                        fail();
    char                        fill();
    char                        fill(char);
    long                        flags();
    long                        flags(long);
    int                        good();
    long&                       iword(int);
    int                        operator!();
                                operator void*();
    int                        precision();
    int                        precision(int);
    streambuf*                 rdbuf();
    void* &                     pword(int);
    int                        rdstate();
    long                        setf(long setbits, long field);
    long                        setf(long);
    static void                 sync_with_stdio();

```

```
        ostream*      tie();
        ostream*      tie(ostream*);
        long           unsetf(long);
        int            width();
        int            width(int);
        static int      xalloc();

protected:
                                ios();
                                init(streambuf*);

private:
                                ios(ios&);
        void            operator=(ios&);
};

        /* Manipulators */
ios&      dec(ios&) ;
ios&      hex(ios&) ;
ios&      oct(ios&) ;
ostream&  endl(ostream& i) ;
ostream&  ends(ostream& i) ;
ostream&  flush(ostream&) ;
istream&  ws(istream&) ;
```

Description

The stream classes derived from class `ios` provide a high level interface that supports transferring formatted and unformatted information into and out of `streambufs`. This section describes the operations common to both input and output.

Several enumerations are declared in class `ios`, `open_mode`, `io_state`, `seek_dir`, and format flags, to avoid polluting the global name space. The `io_states` are described in *Error states* on page 239. The format fields are described in *Formatting* on page 240. The `open_modes` are described in detail under `pfb=f.open(name, mode, prot)` on page 231, in the section *fstream*. The `seek_dirs` are described under `pos=sb->seekoff(off, dir, mode)` on page 271, in the section *streambuf – public*.

In the following descriptions assume:

- `s` and `s2` are `ioss`.
- `sr` is an `ios&`.
- `sp` is a `ios*`.
- `i`, `oi`, `j`, and `n` are `ints`.
- `l`, `f`, and `b` are `longs`.
- `c` and `oc` are `chars`.
- `osp` and `oosp` are `ostream*s`.

- *sb* is a `streambuf*`.
- *pos* is a `streampos`.
- *off* is a `streamoff`.
- *dir* is a `seek_dir`.
- *mode* is an `int` representing an `open_mode`.
- *fct* is a function with type `ios& (*) (ios&)`.
- *vp* is a `void*&`.

Constructors and assignment

ios(*sb*)

The `streambuf` denoted by *sb* becomes the `streambuf` associated with the constructed `ios`. If *sb* is null, the effect is undefined.

ios(*sr*)

s2=s

Copying of `ios` is not well-defined in general, therefore the constructor and assignment operators are private so that the compiler will complain about attempts to copy `ios` objects. Copying pointers to `istream`s is usually what is desired.

ios()

init(*sb*)

Because class `ios` is now inherited as a virtual base class, a constructor with no arguments must be used. This constructor is declared protected. Therefore `ios::init(streambuf*)` is declared protected and must be used for initialisation of derived classes.

Error states

An `ios` has an internal error state (which is a collection of the bits declared as `io_states`). Members related to the error state are:

***i=s*.rdstate()**

Returns the current error state.

***s*.clear(*i*)**

Stores *i* as the error state. If *i* is zero, this clears all bits. To set a bit without clearing previously set bits requires something like `s.clear(ios::badbit | s.rdstate())`.

`i=s.good()`

Returns non-zero if the error state has no bits set, zero otherwise.

`i=s.eof()`

Returns non-zero if `eofbit` is set in the error state, zero otherwise. Normally this bit is set when an end-of-file has been encountered during an extraction.

`i=s.fail()`

Returns non-zero if either `badbit` or `failbit` is set in the error state, zero otherwise. Normally this indicates that some extraction or conversion has failed, but the stream is still usable. That is, once the `failbit` is cleared, I/O on `s` can usually continue.

`i=s.bad()`

Returns non-zero if `badbit` is set in the error state, zero otherwise. This usually indicates that some operation on `s.rdbuf()` has failed, a severe error, from which recovery is probably impossible. That is, it will probably be impossible to continue I/O operations on `s`.

Operators

Two operators are defined to allow convenient checking of the error state of an `ios`: `operator!()` and `operator void*()`. The latter converts an `ios` to a pointer so that it can be compared to zero. The conversion will return 0 if `failbit` or `badbit` is set in the error state, and will return a pointer value otherwise. This pointer is not meant to be used. This allows one to write expressions such as:

```
if ( cin ) ...  
if ( cin >> x ) ...
```

The `!` operator returns non-zero if `failbit` or `badbit` is set in the error state, which allows expressions like the following to be used:

```
if ( !cout ) ...
```

Formatting

An `ios` has a format state that is used by input and output operations to control the details of formatting operations. For other operations the format state has no particular effect and its components may be set and examined arbitrarily by user code. Most formatting details are controlled by using the `flags()`, `setf()`, and `unsetf()` functions to set the following flags, which are declared in an enumeration in class `ios`. Three other components of the format state are controlled separately with the functions `fill()`, `width()`, and `precision()`.

skipws

If `skipws` is set, whitespace will be skipped on input. This applies to scalar extractions. When `skipws` is not set, whitespace is not skipped before the extractor begins conversion. If `skipws` is not set and a zero length field is encountered, the extractor will signal an error. Additionally, the arithmetic extractors will signal an error if `skipws` is not set and a whitespace is encountered.

left**right****internal**

These flags control the padding of a value. When `left` is set, the value is left-adjusted, that is, the fill character is added after the value. When `right` is set, the value is right-adjusted, that is, the fill character is added before the value. When `internal` is set, the fill character is added after any leading sign or base indication, but before the value. Right-adjustment is the default if none of these flags is set. These fields are collectively identified by the static member, `ios::adjustfield`. The fill character is controlled by the `fill()` function, and the width of padding is controlled by the `width()` function.

dec**oct****hex**

These flags control the conversion base of a value. The conversion base is 10 (decimal) if `dec` is set, but if `oct` or `hex` is set, conversions are done in octal or hexadecimal, respectively. If none of these is set, insertions are in decimal, but extractions are interpreted according to the C++ lexical conventions for integral constants. These fields are collectively identified by the static member, `ios::basefield`. The manipulators `hex`, `dec`, and `oct` can also be used to set the conversion base; see the section *Built-in Manipulators* on page 246.

showbase

If `showbase` is set, insertions will be converted to an external form that can be read according to the C++ lexical conventions for integral constants. `showbase` is unset by default.

showpos

If `showpos` is set, then a '+' will be inserted into a decimal conversion of a positive integral value.

uppercase

If `uppercase` is set, then an uppercase 'X' will be used for hexadecimal conversion when `showbase` is set, or an uppercase 'E' will be used to print floating point numbers in scientific notation.

showpoint

If `showpoint` is set, trailing zeros and decimal points appear in the result of a floating point conversion.

**scientific
fixed**

These flags control the format to which a floating point value is converted for insertion into a stream. If `scientific` is set, the value is converted using scientific notation, where there is one digit before the decimal point and the number of digits after it is equal to the `precision` (see below), which is six by default. An uppercase 'E' will introduce the exponent if `uppercase` is set, a lowercase 'e' will appear otherwise. If `fixed` is set, the value is converted to decimal notation with `precision` digits after the decimal point, or six by default. If neither `scientific` nor `fixed` is set, then the value will be converted using either notation, depending on the value; scientific notation will be used if the exponent resulting from the conversion is less than -4 or greater than or equal to `precision` digits. Otherwise the value will be converted to decimal notation with `precision` digits total. If `showpoint` is not set, trailing zeroes are removed from the result and a decimal point appears only if it is followed by a digit. `scientific` and `fixed` are collectively identified by the static member `ios::floatfield`.

unitbuf

When set, a flush is performed by `ostream::osfx()` after each insertion. Unit buffering provides a compromise between buffered output and unbuffered output. Performance is better under unit buffering than unbuffered output, which makes a system call for each character output. Unit buffering makes a system call for each insertion operation, and doesn't require the user to call `ostream::flush()`.

stdio

When set, `stdout` and `stderr` are flushed by `ostream::osfx()` after each insertion.

The following functions use and set the format flags and variables:

`oc=s.fill(c)`

Sets the *fill character* format state variable to *c* and returns the previous value. *c* will be used as the padding character, if one is necessary (see `width()` below). The default fill or padding character is a space. The positioning of the fill character is determined by the `right`, `left`, and `internal` flags; see above. A parameterised manipulator, `setfill`, is also available for setting the fill character; see *manipulators* on page 255.

`c=s.fill()`

Returns the 'fill character' format state variable.

`l=s.flags()`

Returns the current format flags.

`l=s.flags(f)`

Resets all the format flags to those specified in *f* and returns the previous settings.

`oi=s.precision(i)`

Sets the *precision* format state variable to *i* and returns the previous value. This variable controls the number of significant digits inserted by the floating point inserter. The default is 6. A parameterised manipulator, `setprecision`, is also available for setting the precision; see *manipulators* on page 255.

`i=s.precision()`

Returns the *precision* format state variable.

`l=s.setf(b)`

Turns on in *s* the format flags marked in *b* and returns the previous settings. A parameterised manipulator, `setiosflags`, performs the same function; see *manipulators* on page 255.

`l=s.setf(b,f)`

Resets in *s* only the format flags specified by *f* to the settings marked in *b*, and returns the previous settings. That is, the format flags specified by *f* are cleared in *s*, then reset to be those marked in *b*. For example, to change the conversion base in *s* to be hex, one could write: `s.setf(ios::hex, ios::basefield)`. `ios::basefield` specifies the conversion base bits as candidates for change,

and `ios::hex` specifies the new value. `s.setf(0, f)` will clear all the bits specified by `f`, as will a parameterised manipulator, `resetiosflags`; see *manipulators* on page 255.

`l=s.unsetf(b)`

Unsets in `s` the bits set in `b` and returns the previous settings.

`oi=s.width(i)`

Sets the *field-width* format variable to `i` and returns the previous value. When the field width is zero (the default), inserters will insert only as many characters as necessary to represent the value being inserted. When the *field-width* is non-zero, the inserters will insert at least that many characters, using the fill character to pad the value, if the value being inserted requires fewer than *field-width* characters to be represented. However, the numeric inserters never truncate values, so if the value being inserted will not fit in *field-width* characters, more than *field-width* characters will be output. The *field-width* is always interpreted as a minimum number of characters; there is no direct way to specify a maximum number of characters. The *field-width* format variable is reset to the default (zero) after each insertion or extraction, and in this sense it behaves as a parameter for insertions and extractions. A parameterised manipulator, `setw`, is also available for setting the width; see *manipulators* on page 255.

`i=s.width()`

Returns the *field-width* format variable.

User-defined Format Flags

Class `ios` can be used as a base class for derived classes that require additional format flags or variables. The `iostream` library provides several functions to do this. The two static member functions `ios::xalloc` and `ios::bitalloc`, allow several such classes to be used together without interference.

`b=ios::bitalloc()`

Returns a `long` with a single, previously unallocated, bit set. This allows users who need an additional flag to acquire one, and pass it as an argument to `ios::setf()`, for example.

`i=ios::xalloc()`

Returns a previously unused index into an array of words available for use as format state variables by derived classes.

`l=s.iword(i)`

When *i* is an index allocated by `ios::xalloc`, `iword()` returns a reference to the *i*th user-defined word.

`vp=s.pword(i)`

When *i* is an index allocated by `ios::xalloc`, `pword()` returns a reference to the *i*th user-defined word. `pword()` is the same as `iword` except that it is typed differently.

Other members**`sb=s.rdbuf()`**

Returns a pointer to the `streambuf` associated with *s* when *s* was constructed.

`ios::sync_with_stdio()`

Solves problems that arise when mixing `stdio` and `iostreams`. The first time it is called it will reset the standard `iostreams` (`cin`, `cout`, `cerr`, `clog`) to be streams using `stdiobufs`. After that, input and output using these streams may be mixed with input and output using the corresponding `FILEs` (`stdin`, `stdout`, and `stderr`) and will be properly synchronised. `sync_with_stdio()` makes `cout` and `cerr` unit buffered (see `ios::unitbuf` and `ios::stdio` above). Invoking `sync_with_stdio()` degrades performance a variable amount, depending on the length of the strings being inserted (shorter strings incur a larger performance hit).

`osp=s.tie(osp)`

Sets the `tie` variable to *osp*, and returns its previous value. This variable supports automatic ‘flushing’ of `ioss`. If the `tie` variable is non-null and an `ios` needs more characters or has characters to be consumed, the `ios` pointed at by the `tie` variable is flushed. By default, `cin` is tied initially to `cout` so that attempts to get more characters from standard input result in flushing standard output. Additionally, `cerr` and `clog` are tied to `cout` by default. For other `ioss`, the `tie` variable is set to zero by default.

`osp=s.tie()`

Returns the `tie` variable.

Built-in Manipulators

Some convenient manipulators (functions that take an `ios&`, an `istream&`, or an `ostream&` and return their argument; see *manipulators* on page 255) are:

**`sr<<dec`
`sr>>dec`**

These set the conversion base format flag to 10.

**`sr<<hex`
`sr>>hex`**

These set the conversion base format flag to 16.

**`sr<<oct`
`sr>>oct`**

These set the conversion base format flag to 8.

`sr>>ws`

Extracts whitespace characters. See *istream* on page 248.

`sr<<endl`

Ends a line by inserting a newline character and flushing. See *ostream* on page 259.

`sr<<ends`

Ends a string by inserting a null (0) character. See *ostream* on page 259.

`sr<<flush`

Flushes outs. See *ostream* on page 259.

Several parameterised manipulators that operate on `ios` objects are described in *manipulators* on page 255: `setw`, `setfill`, `setprecision`, `setiosflags`, and `resetiosflags`.

The `streambuf` associated with an `ios` may be manipulated by other methods than through the `ios`. For example, characters may be stored in a queue-like `streambuf` through an `ostream` while they are being fetched through an `istream`. Or for efficiency some part of a program may choose to do `streambuf` operations directly rather than through the `ios`. In most cases the program does not have to worry about this possibility, because an `ios` never saves information about the internal state of a `streambuf`. For example, if the `streambuf` is repositioned between extraction operations the extraction (input) will proceed normally.

See also

Introduction (page 224), *streambuf – protected* (page 266), *streambuf – public* (page 274), *istream* (page 248), *ostream* (page 259), *manipulators* (page 255)

istream – formatted and unformatted input

Synopsis

```
#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
    enum    seek_dir { beg, cur, end };
    enum    open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
    /* flags for controlling format */
    enum    { skipws=01,
              left=02, right=04, internal=010,
              dec=020, oct=040, hex=0100,
              showbase=0200, showpoint=0400,
              uppercase=01000, showpos=02000,
              scientific=04000, fixed=010000,
              unitbuf=020000, stdio=040000 };
    // and lots of other stuff; see ios on page 237
} ;

class istream : public ios {
public:
                                istream(streambuf*);
    int                          gcount();
    istream&                     get(char* ptr, int len, char delim='\n');
    istream&                     get(unsigned char* ptr,int len, char delim='\n');

    istream&                     get(unsigned char&);
    istream&                     get(char&);
    istream&                     get(streambuf& sb, char delim ='\n');
    int                          get();
    istream&                     getline(char* ptr, int len, char delim='\n');
    istream&                     getline(unsigned char* ptr, int len, char delim='\n');
    istream&                     ignore(int len=1,int delim=EOF);
    int                          ipfx(int need=0);
    int                          peek();
    istream&                     putback(char);
    istream&                     read(char* s, int n);
    istream&                     read(unsigned char* s, int n);
    istream&                     seekg(streampos);
    istream&                     seekg(streamoff, seek_dir);
    int                          sync();
    streampos                    tellg();
```

```

        istream&      operator>>(char*);
        istream&      operator>>(char&);
        istream&      operator>>(short&);
        istream&      operator>>(int&);
        istream&      operator>>(long&);
        istream&      operator>>(float&);
        istream&      operator>>(double&);
        istream&      operator>>(unsigned char*);
        istream&      operator>>(unsigned char&);
        istream&      operator>>(unsigned short&);
        istream&      operator>>(unsigned int&);
        istream&      operator>>(unsigned long&);
        istream&      operator>>(streambuf*);
        istream&      operator>>(istream& (*) (istream&));
        istream&      operator>>(ios& (*) (ios&));
};

class istream_withassign : public istream {
        istream_withassign();
        istream&      operator=(istream&);
        istream&      operator=(streambuf*);
};

extern istream_withassign cin;

istream&      ws(istream&);
ios&          dec(ios&) ;
ios&          hex(ios&) ;
ios&          oct(ios&) ;

```

Description

istreams support interpretation of characters fetched from an associated streambuf. These are commonly referred to as input or extraction operations. The istream member functions and related functions are described below.

In the following descriptions assume that

- *ins* is an istream.
- *inswa* is an istream_withassign.
- *insp* is an istream*.
- *c* is a char&
- *delim* is a char.
- *ptr* is a char* or unsigned char*.
- *sb* is a streambuf&.
- *i*, *n*, *len*, *d*, and *need* are ints.
- *pos* is a streampos.
- *off* is a streamoff.

- *dir* is a `seek_dir`.
- *manip* is a function with type `istream& (*)(istream&)`.

Constructors and assignment

istream(sb)

Initialises `ios` state variables and associates buffer *sb* with the `istream`.

istream_withassign()

Does no initialisation.

inswa=sb

Associates *sb* with *inswa* and initialises the entire state of *inswa*.

inswa=ins

Associates `ins->rdbuf()` with *inswa* and initialises the entire state of *inswa*.

Input prefix function

i = ins.ipfx(need)

If *ins*'s error state is non-zero, returns zero immediately. If necessary (and if it is non-null), any `ios` tied to *ins* is flushed (see the description of `ios::tie()` on page 245 onwards of *ios*. Flushing is considered necessary if either *need*==0 or if there are fewer than *need* characters immediately available. If `ios::skipws` is set in *ins.flags()* and *need* is zero, then leading whitespace characters are extracted from *ins*. `ipfx()` returns zero if an error occurs while skipping whitespace; otherwise it returns non-zero.

Formatted input functions call `ipfx(0)`, while unformatted input functions call `ipfx(1)`; see below.

Formatted input functions (extractors)

ins>>x

Calls `ipfx(0)` and if that returns non-zero, extracts characters from *ins* and converts them according to the type of *x*. It stores the converted value in *x*. Errors are indicated by setting the error state of *ins*. `ios::failbit` means that characters in *ins* were not a representation of the required type. `ios::badbit` indicates that attempts to extract characters failed. *ins* is always returned.

The details of conversion depend on the values of *ins*'s format state flags and variables (see *ios* on page 237) and the type of *x*. Except that extractions that use width reset it to 0, the extraction operators do not change the value of *ostream*'s format state. Extractors are defined for the following types, with conversion rules as described below.

<code>char*</code> , <code>unsigned char*</code>	Characters are stored in the array pointed at by <i>x</i> until a whitespace character is found in <i>ins</i> . The terminating whitespace is left in <i>ins</i> . If <i>ins.width()</i> is non-zero it is taken to be the size of the array, and no more than <i>ins.width()</i> - 1 characters are extracted. A terminating null character (0) is always stored (even when nothing else is done because of <i>ins</i> 's error status). <i>ins.width()</i> is reset to 0.
<code>char&</code> , <code>unsigned char&</code>	A character is extracted and stored in <i>x</i> .
<code>short&</code> , <code>unsigned short&</code> , <code>int&</code> , <code>unsigned int&</code> , <code>long&</code> , <code>unsigned long&</code>	Characters are extracted and converted to an integral value according to the conversion specified in <i>ins</i> 's format flags. Converted characters are stored in <i>x</i> . The first character may be a sign (+ or -). After that, if <i>ios::oct</i> , <i>ios::dec</i> , or <i>ios::hex</i> is set in <i>ins.flags()</i> , the conversion is octal, decimal, or hexadecimal, respectively. Conversion is terminated by the first 'non-digit,' which is left in <i>ins</i> . Octal digits are the characters '0' to '7'. Decimal digits are the octal digits plus '8' and '9'. Hexadecimal digits are the decimal digits plus the letters 'a' through 'f' (in either upper or lower case). If none of the conversion base format flags is set, then the number is interpreted according to C++ lexical conventions. That is, if the first characters (after the optional sign) are 0x or 0X a hexadecimal conversion is performed on following hexadecimal digits. Otherwise, if the first character is a 0, an octal conversion is performed, and in all other cases a decimal conversion is performed. <i>ios::failbit</i> is set if there are no digits (not counting the 0 in 0x or 0X) during hex conversion) available.
<code>float&</code> , <code>double&</code>	Converts the characters according to C++ syntax for a float or double, and stores the result in <i>x</i> . <i>ios::failbit</i> is set if there are no digits available in <i>ins</i> or if it does not begin with a well formed floating point number.

The type and name (operator>>) of the extraction operations are chosen to give a convenient syntax for sequences of input operations. The operator overloading of C++ permits extraction functions to be declared for user-defined classes. These operations can then be used with the same syntax as the member functions described here.

ins>>sb

If `ios.ipfx(0)` returns non-zero, extracts characters from `ios` and inserts them into `sb`. Extraction stops when EOF is reached. Always returns `ins`.

Unformatted input functions

These functions call `ipfx(1)` and proceed only if it returns non-zero:

insp=&ins.get(ptr,len,delim)

Extracts characters and stores them in the byte array beginning at `ptr` and extending for `len` bytes. Extraction stops when `delim` is encountered (`delim` is left in `ins` and not stored), when `ins` has no more characters, or when the array has only one byte left. `get` always stores a terminating null, even if it doesn't extract any characters from `ins` because of its error status. `ios::failbit` is set only if `get` encounters an end of file before it stores any characters.

insp=&ins.get(c)

Extracts a single character and stores it in `c`.

insp=&ins.get(sb,delim)

Extracts characters from `ins.rdbuf()` and stores them into `sb`. It stops if it encounters end of file or if a store into `sb` fails or if it encounters `delim` (which it leaves in `ins`). `ios::failbit` is set if it stops because the store into `sb` fails.

i=ins.get().

Extracts a character and returns it. `i` is EOF if extraction encounters end of file. `ios::failbit` is never set.

insp=&ins.getline(ptr,len,delim)

Does the same thing as `ins.get(ptr,len,delim)` with the exception that it extracts a terminating `delim` character from `ins`. In case `delim` occurs when exactly `len` characters have been extracted, termination is treated as being due to the array being filled, and this `delim` is left in `ins`.

insp=&ins.ignore(n,d)

Extracts and throws away up to *n* characters. Extraction stops prematurely if *d* is extracted or end of file is reached. If *d* is EOF it can never cause termination.

insp=&ins.read(ptr,n)

Extracts *n* characters and stores them in the array beginning at *ptr*. If end of file is reached before *n* characters have been extracted, *read* stores whatever it can extract and sets *ios::failbit*. The number of characters extracted can be determined via *ins.gcount()*.

Other members***i=ins.gcount()***

Returns the number of characters extracted by the last unformatted input function. Formatted input functions may call unformatted input functions and thereby reset this number.

i=ins.peek()

Begins by calling *ins.ipfx(1)*. If that call returns zero or if *ins* is at end of file, it returns EOF. Otherwise it returns the next character without extracting it.

insp=&ins.putback(c)

Attempts to back up *ins.rdbuf()*. *c* must be the character before *ins.rdbuf()*'s get pointer. (Unless other activity is modifying *ins.rdbuf()* this is the last character extracted from *ins*.) If it is not, the effect is undefined. *putback* may fail (and set the error state). Although it is a member of *istream*, *putback* never extracts characters, so it does not call *ipfx*. It will, however, return without doing anything if the error state is non-zero.

i=&ins.sync()

Establishes consistency between internal data structures and the external source of characters. Calls *ins.rdbuf()->sync()*, which is a virtual function, so the details depend on the derived class. Returns EOF to indicate errors.

ins>>manip

Equivalent to *manip(ins)*. Syntactically this looks like an extractor operation, but semantically it does an arbitrary operation rather than converting a sequence of characters and storing the result in *manip*. A predefined manipulator, *ws*, is described below.

Member functions related to positioning

insp=&ins.seekg(off,dir)

Repositions *ins.rdbuf()*'s get pointer. See *streambuf – public* on page 274 for a discussion of positioning.

insp=&ins.seekg(pos)

Repositions *ins.rdbuf()*'s get pointer. See *streambuf – public* on page 274 for a discussion of positioning.

pos=ins.tellg()

The current position of *ios.rdbuf()*'s get pointer. See *streambuf – public* on page 274 for a discussion of positioning.

Manipulator

ins>>ws

Extracts whitespace characters.

ins>>dec

Sets the conversion base format flag to 10. See *ios* on page 237.

ins>>hex

Sets the conversion base format flag to 16. See *ios* on page 237.

ins>>oct

Sets the conversion base format flag to 8. See *ios* on page 237.

Caveats

There is no overflow detection on conversion of integers.

See also

ios (page 237), *streambuf – public* (page 274), *manipulators* (page 255)

manipulators

manipulators – iostream out of band manipulations

Synopsis

```
#include <iostream.h>
#include <iomanip.h>

template <class T>

class SMANIP {
    SMANIP( ios& (*) (ios&,T), T);
    friend istream& operator>>(istream&, SMANIP<T>&);
    friend ostream& operator<<(ostream&, SMANIP<T>&);
};

template <class T>
class SAPP {
    SAPP(T) ( ios& (*) (ios&,T));
    SMANIP<T> operator() (T);
};

template <class T>
class IMANIP {
    IMANIP( istream& (*) (istream&,T), T);
    friend istream& operator>>(istream&, IMANIP<T>&);
};

template <class T>
class IAPP {
    IAPP( istream& (*) (istream&,T));
    IMANIP<T> operator() (T);
};

template <class T>
class OMANIP {
    OMANIP( ostream& (*) (ostream&,T), T);
    friend ostream& operator<<(ostream&, OMANIP<T>&);
};

template <class T>
class OAPP {
    OAPP( ostream& (*) (ostream&,T));
    OMANIP<T> operator() (T);
};

template <class T>
class IOMANIP {
    IOMANIP( istream& (*) (istream&,T), T);
    friend istream& operator>>(istream&, IOMANIP<T>&);
    friend ostream& operator<<(ostream&, IOMANIP<T>&);
};

template <class T>
class IOAPP {
    IOAPP( istream& (*) (istream&,T));
    IOMANIP<T> operator() (T);
};
```

```
SMANIP<long>    resetiosflags(long);
SMANIP<int>     setfill(int);
SMANIP<long>    setiosflags(long);
SMANIP<int>     setprecision(int);
SMANIP<int>     setw(int w);
```

Description

Manipulators are values that may be ‘inserted into’ or ‘extracted from’ streams to achieve some effect (other than to insert or extract a value representation), with a convenient syntax. They enable one to embed a function call in an expression containing a series of insertions or extractions. For example, the predefined manipulator for *ostreams*, *flush*, can be used as follows:

```
cout << flush
```

to flush *cout*. Several *iostream* classes supply manipulators: see *ios* on page 237, *istream* on page 248, and *ostream* on page 259. *flush* is a simple manipulator; some manipulators take arguments, such as the predefined *ios* manipulators, *setfill* and *setw* (see below).

In the following descriptions, assume:

- *t* is a *T*, or type name.
- *s* is an *ios*.
- *i* is an *istream*.
- *o* is an *ostream*.
- *io* is an *iostream*.
- *f* is an *ios& (*) (ios&)*.
- *if* is an *istream& (*) (istream&)*.
- *of* is an *ostream& (*) (ostream&)*.
- *iof* is an *iostream& (*) (iostream&)*.
- *n* is an *int*.
- *l* is a *long*.

```

s<<SMANIP<T>(f,t)
s>>SMANIP<T>(f,t)
s<<SAPP<T>(f)(t)
s>>SAPP<T>(f)(t)

```

Returns $f(s, t)$, where s is the left operand of the insertion or extractor operator (i.e. s , i , o , or io).

```

i>>IMANIP<T>(if,t)
i>>IAPP<T>(if)(t)

```

Returns $if(i, t)$.

```

o<<OMANIP<T>(of,t)
o<<OAPP<T>(of)(t)

```

Returns $of(o, t)$.

```

io<<IOMANIP<T>(iof,t)
io>>IOMANIP<T>(iof,t)
io<<IOAPP<T>(iof)(t)
io>>IOAPP<T>(iof)(t)

```

Returns $iof(io, t)$.

`iomanip.h` contains declarations of some manipulators that take an `int` or a `long` argument. These manipulators all have to do with changing the format state of a stream; see `ios` on page 237 for further details.

```

o<<setw(n)
i>>setw(n)

```

Sets the field width of the stream (left-hand operand: o or i) to n .

```

o<<setfill(n)
i>>setfill(n)

```

Sets the fill character of the stream (o or i) to be n .

```

o<<setprecision(n)
i>>setprecision(n)

```

Sets the precision of the stream (o or i) to be n .

`o<<setiosflags(l)`

`i>>setiosflags(l)`

Turns on in the stream (*o* or *i*) the format flags marked in *l*. (Calls `o.setf(l)` or `i.setf(l)`).

`o<<resetiosflags(l)`

`i>>resetiosflags(l)`

Clears in the stream (*o* or *i*) the format bits specified by *l*. (Calls `o.setf(0, l)` or `i.setf(0, l)`).

See also

ios (page 237), *istream* (page 248), *ostream* (page 259)

ostream

ostream – formatted and unformatted output

Synopsis

```

#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
    enum    seek_dir { beg, cur, end };
    enum    open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
    enum    { skipws=01,
              left=02, right=04, internal=010,
              dec=020, oct=040, hex=0100,
              showbase=0200, showpoint=0400,
              uppercase=01000, showpos=02000,
              scientific=04000, fixed=010000,
              unitbuf=020000, stdio=040000 };
    // and lots of other stuff; see ios on page 237
};

class ostream : public ios {
public:
                                ostream(streambuf*);
    ostream&                    flush();
    int                         opfx();
    ostream&                   put(char);
    ostream&                   seekp(streampos);
    ostream&                   seekp(streamoff, seek_dir);
    streampos                  tellp();
    ostream&                   write(const char* ptr, int n);
    ostream&                   write(const unsigned char* ptr, int n);
    ostream&                   operator<<(const char*);
    ostream&                   operator<<(char);
    ostream&                   operator<<(short);
    ostream&                   operator<<(int);
    ostream&                   operator<<(long);
    ostream&                   operator<<(float);
    ostream&                   operator<<(double);
    ostream&                   operator<<(unsigned char);
    ostream&                   operator<<(unsigned short);
    ostream&                   operator<<(unsigned int);
    ostream&                   operator<<(unsigned long);
    ostream&                   operator<<(void*);
    ostream&                   operator<<(streambuf*);
    ostream&                   operator<<(ostream& *) (ostream&);
    ostream&                   operator<<(ios& *) (ios&);
};

```

```
class ostream_withassign {
    ostream_withassign();
    istream&      operator=(istream&);
    ostream&      operator=(streambuf*);
};

extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;

ostream&      endl(ostream&) ;
ostream&      ends(ostream&) ;
ostream&      flush(ostream&) ;
ios&          dec(ios&) ;
ios&          hex(ios&) ;
ios&          oct(ios&) ;
```

Description

ostreams support insertion (storing) into a `streambuf`. These are commonly referred to as output operations. The `ostream` member functions and related functions are described below.

In the following descriptions, assume:

- *outs* is an `ostream`.
- *outswa* is an `ostream_withassign`.
- *outsp* is an `ostream*`.
- *c* is a `char`.
- *ptr* is a `char*` or `unsigned char*`.
- *sb* is a `streambuf*`.
- *i* and *n* are `ints`.
- *pos* is a `streampos`.
- *off* is a `streamoff`.
- *dir* is a `seek_dir`.
- *manip* is a function with type `ostream& (*) (ostream&)`.

Constructors and assignment

`ostream(sb)`

Initialises `ios` state variables and associates buffer `sb` with the `ostream`.

`ostream_withassign()`

Does no initialisation. This allows a file static variable of this type (`cout`, for example) to be used before it is constructed, provided it is assigned to first.

`outswa=sb`

Associates `sb` with `swa` and initialises the entire state of `outswa`.

`inswa=ins`

Associates `ins->rdbuf()` with `swa` and initialises the entire state of `outswa`.

Output prefix function

`i=outs.opfx()`

If `outs`'s error state is non-zero, returns immediately. If `outs.tie()` is non-null, it is flushed. Returns non-zero except when `outs`'s error state is non-zero.

Output suffix function

`osfx()`

Performs 'suffix' actions before returning from inserters. If `ios::unitbuf` is set, `osfx()` flushes the `ostream`. If `ios::stdio` is set, `osfx()` flushes `stdout` and `stderr`.

`osfx()` is called by all predefined inserters, and should be called by user-defined inserters as well, after any direct manipulation of the `streambuf`. It is not called by the binary output functions.

Formatted output functions (inserters)

outs*<<*x

First calls *outs.opfx()* and if that returns 0, does nothing. Otherwise inserts a sequence of characters representing *x* into *outs.rdbuf()*. Errors are indicated by setting the error state of *outs*. *outs* is always returned.

x is converted into a sequence of characters (its representation) according to rules that depend on *x*'s type and *outs*'s format state flags and variables (see *ios* on page 237). Inserters are defined for the following types, with conversion rules as described below:

<code>char*</code>	The representation is the sequence of characters up to (but not including) the terminating null of the string <i>x</i> points at.
any integral type except <code>char</code> and <code>unsigned char</code>	If <i>x</i> is positive the representation contains a sequence of decimal, octal, or hexadecimal digits with no leading zeros according to whether <code>ios::dec</code> , <code>ios::oct</code> , or <code>ios::hex</code> , respectively, is set in <i>ios</i> 's format flags. If none of those flags are set, conversion defaults to decimal. If <i>x</i> is zero, the representation is a single zero character(0). If <i>x</i> is negative, decimal conversion converts it to a minus sign (–) followed by decimal digits. If <i>x</i> is positive and <code>ios::showpos</code> is set, decimal conversion converts it to a plus sign (+) followed by decimal digits. The other conversions treat all values as unsigned. If <code>ios::showbase</code> is set in <i>ios</i> 's format flags, the hexadecimal representation contains 0x before the hexadecimal digits, or 0X if <code>ios::uppercase</code> is set. If <code>ios::showbase</code> is set, the octal representation contains a leading 0.
<code>void*</code>	Pointers are converted to integral values and then converted to hexadecimal numbers as if <code>ios::showbase</code> were set.
<code>float</code> , <code>double</code>	The arguments are converted according to the current values of <i>outs.precision()</i> , <i>outs.width()</i> and <i>outs</i> 's format flags <code>ios::scientific</code> , <code>ios::fixed</code> , and <code>ios::uppercase</code> . (See <i>ios</i> on page 237.) The default value for <i>outs.precision()</i> is 6. If neither <code>ios::scientific</code> nor <code>ios::fixed</code> is set, either fixed or scientific notation is chosen for the representation, depending on the value of <i>x</i> .
<code>char</code> , <code>unsigned char</code>	No special conversion is necessary.

After the representation is determined, padding occurs. If `outs.width()` is greater than 0 and the representation contains fewer than `outs.width()` characters, then enough `outs.fill()` characters are added to bring the total number of characters to `ios.width()`. If `ios::left` is set in `ios`'s format flags, the sequence is left-adjusted, that is, characters are added after the characters determined above. If `ios::right` is set, the padding is added before the characters determined above. If `ios::internal` is set, the padding is added after any leading sign or base indication and before the characters that represent the value. `ios.width()` is reset to 0, but all other format variables are unchanged. The resulting sequence (padding plus representation) is inserted into `outs.rdbuf()`.

`outs<<sb`

If `outs.opfx()` returns non-zero, the sequence of characters that can be fetched from `sb` are inserted into `outs.rdbuf()`. Insertion stops when no more characters can be fetched from `sb`. No padding is performed. Always returns `outs`.

Unformatted output functions

`outsp=&outs.put(c)`

Inserts `c` into `outs.rdbuf()`. Sets the error state if the insertion fails.

`outsp=&outs.write(s,n)`

Inserts the `n` characters starting at `s` into `outs.rdbuf()`. These characters may include zeros (i.e. `s` need not be a null terminated string).

Other member functions

`outsp=&outs.flush()`

Storing characters into a `streambuf` does not always cause them to be consumed (e.g. written to the external file) immediately. `flush()` causes any characters that may have been stored but not yet consumed to be consumed by calling `outs.rdbuf()->sync`.

`outs<<manip`

Equivalent to `manip(outs)`. Syntactically this looks like an insertion operation, but semantically it does an arbitrary operation rather than converting `manip` to a sequence of characters as do the insertion operators. Predefined manipulators are described below.

Positioning functions

`outsp=&ins.seekp(off,dir)`

Repositions `outs.rdbuf()`'s put pointer. See *streambuf – public* on page 274 for a discussion of positioning.

`outsp=&outs.seekp(pos)`

Repositions `outs.rdbuf()`'s put pointer. See *streambuf – public* on page 274 for a discussion of positioning.

`pos=outs.tellp()`

The current position of `outs.rdbuf()`'s put pointer. See *streambuf – public* on page 274 for a discussion of positioning.

Manipulators

`outs<<endl`

Ends a line by inserting a newline character and flushing.

`outs<<ends`

Ends a string by inserting a null (0) character.

`outs<<flush`

Flushes `outs`.

`outs<<dec`

Sets the conversion base format flag to 10. See *ios* on page 237.

`outs<<hex`

Sets the conversion base format flag to 16. See *ios* on page 237.

`outs<<oct`

Sets the conversion base format flag to 8. See *ios* on page 237.

See also

ios (page 237), *streambuf – public* (page 274), *manipulators* (page 255)

stdiobuf

stdiobuf – iostream specialised to stdio FILE

Synopsis

```
#include <iostream.h>
#include <stdiostream.h>
#include <stdio.h>

class stdiobuf : public streambuf {
    stdiobuf(FILE* f);
    FILE*    stdiofile();
};
```

Description

Operations on a `stdiobuf` are reflected on the associated `FILE`. A `stdiobuf` is constructed in unbuffered mode, which causes all operations to be reflected immediately in the `FILE`. `seekg()`s and `seekp()`s are translated into `fseek()`s. `setbuf()` has its usual meaning; if it supplies a reserve area, buffering will be turned back on.

Caveats

`stdiobuf` is intended to be used when mixing C and C++ code. New C++ code should prefer to use `filebuf`s, which have better performance.

See also

filebuf (page 229), *istream* (page 248), *ostream* (page 259), *streambuf – public* (page 274)

streambuf – protected

streambuf – interface for derived classes

Synopsis

```
#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
    enum    seek_dir { beg, cur, end };
    enum    open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
    // and lots of other stuff; see ios on page 237
} ;

class streambuf {
public:
    streambuf() ;
    streambuf(char* p, int len);
    void      dbp() ;

protected:
    int        allocate();
    char*      base();
    int        blen();
    char*      eback();
    char*      ebuf();
    char*      egptr();
    char*      epptr();
    void       gbump(int n);
    char*      gpptr();
    char*      pbase();
    void       pbump(int n);
    char*      pptr();
    void       setg(char* eb, char* g, char* eg);
    void       setp(char* p, char* ep);
    void       setb(char* b, char* eb, int a=0);
    int        unbuffered();
    void       unbuffered(int);

    virtual int doallocate();
    virtual    ~streambuf() ;
```

```

public:
    virtual int      pbackfail(int c);
    virtual int      overflow(int c=EOF);
    virtual int      underflow();
    virtual          streambuf*
                    setbuf(char* p, int len);
    virtual          streampos
                    seekpos(streampos, int =ios::in|ios::out);
    virtual          streampos
                    seekoff(streamoff, seek_dir, int =ios::in|ios::out);
    virtual int      sync();
};

```

Description

`streambufs` implement the buffer abstraction described in *streambuf – public* on page 274. However, the `streambuf` class itself contains only basic members for manipulating the characters and normally a class derived from `streambuf` will be used. This section describes the interface needed by programmers who are coding a derived class. Broadly speaking there are two kinds of member functions described here. The non-virtual functions are provided for manipulating a `streambuf` in ways that are appropriate in a derived class. Their descriptions reveal details of the implementation that would be inappropriate in the public interface. The virtual functions permit the derived class to specialise the `streambuf` class in ways appropriate to the specific sources and sinks that it is implementing. The descriptions of the virtual functions explain the obligations of the virtuals of the derived class. If the virtuals behave as specified, the `streambuf` will behave as specified in the public interface. However, if the virtuals do not behave as specified, then the `streambuf` may not behave properly, and an `iostream` (or any other code) that relies on proper behaviour of the `streambuf` may not behave properly either.

In the following descriptions assume:

- *sb* is a `streambuf*`.
- *i* and *n* are ints.
- *ptr*, *b*, *eb*, *p*, *ep*, *eb*, *g*, and *eg* are `char*s`.
- *c* is an int character (positive or EOF).
- *pos* is a `streampos`. (See *streambuf – public* on page 274.)
- *off* is a `streamoff`.
- *dir* is a `seekdir`.
- *mode* is an int representing an `open_mode`.

Constructors

streambuf()

Constructs an empty buffer corresponding to an empty sequence.

streambuf(*b*,*len*)

Constructs an empty buffer and then sets up the reserve area to be the *len* bytes starting at *b*.

The Get, Put, and Reserver area

The protected members of `streambuf` present an interface to derived classes organised around three areas (arrays of bytes) managed cooperatively by the base and derived classes. They are the *get area*, the *put area*, and the *reserve area* (or *buffer*). The get and the put areas are normally disjoint, but they may both overlap the reserve area, whose primary purpose is to be a resource in which space for the put and get areas can be allocated. The get and the put areas are changed as characters are put into and got from the buffer, but the reserve area normally remains fixed. The areas are defined by a collection of `char*` values. The buffer abstraction is described in terms of pointers that point between characters, but the `char*` values must point at `chars`. To establish a correspondence the `char*` values should be thought of as pointing just before the byte they really point at.

Functions to examine the pointers

***ptr*=*sb*->base()**

Returns a pointer to the first byte of the reserve area. Space between *sb*->base () and *sb*->ebuf () is the reserve area.

***ptr*=*sb*->eback()**

Returns a pointer to a lower bound on *sb*->gptr (). Space between *sb*->eback () and *sb*->gptr () is available for putback.

***ptr*=*sb*->ebuf()**

Returns a pointer to the byte after the last byte of the reserve area.

***ptr*=*sb*->egptr()**

Returns a pointer to the byte after the last byte of the get area.

***ptr*=*sb*->epptr()**

Returns a pointer to the byte after the last byte of the put area.

ptr=sb->gptr()

Returns a pointer to the first byte of the get area. The available characters are those between *sb->gptr()* and *sb->egptr()*. The next character fetched will be **sb->gptr()* unless *sb->egptr()* is less than or equal to *sb->gptr()*.

ptr=sb->pbase()

Returns a pointer to the put area base. Characters between *sb->pbase()* and *sb->pptr()* have been stored into the buffer and not yet consumed.

ptr=sb->pptr()

Returns a pointer to the first byte of the put area. The space between *sb->pptr()* and *sb->epptr()* is the put area and characters will be stored here.

Functions for setting the pointers

Note that to indicate that a particular area (get, put, or reserve) does not exist, all the associated pointers should be set to zero.

sb->setb(b, eb, i)

Sets *base()* and *ebuf()* to *b* and *eb* respectively. *i* controls whether the area will be subject to automatic deletion. If *i* is non-zero, then *b* will be deleted when *base* is changed by another call of *setb()*, or when the destructor is called for **sb*. If *b* and *eb* are both null then we say that there is no reserve area. If *b* is non-null, there is a reserve area even if *eb* is less than *b* and so the reserve area has zero length.

sb->setp(p, ep)

Sets *pptr()* to *p*, *pbase()* to *p*, and *epptr()* to *ep*.

sb->setg(eb, g, eg)

Sets *eback()* to *eb*, *gptr()* to *g*, and *egptr()* to *eg*.

Other non-virtual members

***i=*sb->allocate()**

Tries to set up a reserve area. If a reserve area already exists or if *sb->unbuffered()* is non-zero, *allocate()* returns 0 without doing anything. If the attempt to allocate space fails, *allocate()* returns EOF, otherwise (i.e. allocation succeeds) *allocate()* returns 1. *allocate()* is not called by any non-virtual member function of *streambuf*.

***i=*sb->blen()**

Returns the size (in chars) of the current reserve area.

dbp()

Writes directly on file descriptor 1 information in ASCII about the state of the buffer. It is intended for debugging and nothing is specified about the form of the output. It is considered part of the protected interface because the information it prints can only be understood in relation to that interface, but it is a public function so that it can be called anywhere during debugging.

sb->gbump(n)

Increments *gptr()* by *n* which may be positive or negative. No checks are made on whether the new value of *gptr()* is in bounds.

sb->pbump(n)

Increments *pptr()* by *n* which may be positive or negative. No checks are made on whether the new value of *pptr()* is in bounds.

sb->unbuffered(i)

***i=*sb->unbuffered()**

There is a private variable known as *sb*'s buffering state. *sb->unbuffered(i)* sets the value of this variable to *i* and *sb->unbuffered()* returns the current value. This state is independent of the actual allocation of a reserve area. Its primary purpose is to control whether a reserve area is allocated automatically by *allocate*.

Virtual member functions

Virtual functions may be redefined in derived classes to specialise the behaviour of `streambufs`. This section describes the behaviour that these virtual functions should have in any derived classes; the next section describes the behaviour that these functions are defined to have in base class `streambuf`.

`i=sb->doallocate()`

Is called when `allocate()` determines that space is needed. `doallocate()` is required to call `setb()` to provide a reserve area or to return EOF if it cannot. It is only called if `sb->unbuffered()` is zero and `sb->base()` is zero.

`i=sb->overflow(c)`

Is called to consume characters. If `c` is not EOF, `overflow()` also must either save `c` or consume it. Usually it is called when the put area is full and an attempt is being made to store a new character, but it can be called at other times. The normal action is to consume the characters between `pbase()` and `pptr()`, call `setp()` to establish a new put area, and if `c != EOF` store it (using `sputc()`). `sb->overflow()` should return EOF to indicate an error; otherwise it should return something else.

`i=sb->pbackfail(c)`

Is called when `eback()` equals `gptr()` and an attempt has been made to putback `c`. If this situation can be dealt with (e.g. by repositioning an external file), `pbackfail()` should return `c`; otherwise it should return EOF.

`pos=sb->seekoff(off, dir, mode)`

Repositions the get and/or put pointers (i.e. the abstract get and put pointers, not `pptr()` and `gptr()`). The meanings of `off` and `dir` are discussed in *streambuf-public* on page 274. `mode` specifies whether the put pointer (`ios::out` bit set) or the get pointer (`ios::in` bit set) is to be modified. Both bits may be set in which case both pointers should be affected. A class derived from `streambuf` is not required to support repositioning. `seekoff()` should return EOF if the class does not support repositioning. If the class does support repositioning, `seekoff()` should return the new position or EOF on error.

`pos=sb->seekpos(pos, mode)`

Repositions the `streambuf` get and/or put pointer to `pos`. `mode` specifies which pointers are affected as for `seekoff()`. Returns `pos` (the argument) or EOF if the class does not support repositioning or an error occurs.

***sb=*sb->setbuf(ptr, len)**

Offers the array at *ptr* with *len* bytes to be used as a reserve area. The normal interpretation is that if *ptr* or *len* are zero then this is a request to make the *sb* unbuffered. The derived class may use this area or not as it chooses. It may accept or ignore the request for unbuffered state as it chooses. `setbuf()` should return *sb* if it honours the request. Otherwise it should return 0.

***i=*sb->sync()**

Is called to give the derived class a chance to look at the state of the areas, and synchronise them with any external representation. Normally `sync()` should consume any characters that have been stored into the put area, and if possible give back to the source any characters in the get area that have not been fetched. When `sync()` returns there should not be any unconsumed characters, and the get area should be empty. `sync()` should return EOF if some kind of failure occurs.

***i=*sb->underflow()**

Is called to supply characters for fetching, i.e. to create a condition in which the get area is not empty. If it is called when there are characters in the get area it should return the first character. If the get area is empty, it should create a non-empty get area and return the next character (which it should also leave in the get area). If there are no more characters available, `underflow()` should return EOF and leave an empty get area.

The default definitions of the virtual functions:

***i=*sb->streambuf::doallocate()**

Attempts to allocate a reserve area using operator `new`.

***i=*sb->streambuf::overflow(c)**

`streambuf::overflow()` should be treated as if it had undefined behaviour. That is, derived classes should always define it.

***i=*sb->streambuf::pbackfail(c)**

Returns EOF.

***pos=*sb->streambuf::seekpos(pos, mode)**

Returns *sb->seekoff(streamoff(pos), ios::beg, mode)*. Thus to define seeking in a derived class, it is frequently only necessary to define `seekoff()` and use the inherited `streambuf::seekpos()`.

`pos=sb->streambuf::seekoff(off, dir, mode)`

Returns EOF.

`sb=sb->streambuf::setbuf(ptr, len)`

Will honour the request when there is no reserve area.

`i=sb->streambuf::sync()`

Returns 0 if the get area is empty and there are no unconsumed characters. Otherwise it returns EOF.

`i=sb->streambuf::underflow()`

Is compatible with the old stream package, but that behaviour is not considered part of the specification of the istream package. Therefore, `streambuf::underflow()` should be treated as if it had undefined behaviour. That is, it should always be defined in derived classes.

See also

streambuf – *public* (page 274), *ios* (page 237), *istream* (page 248), *ostream* (page 259)

streambuf – public

streambuf – public interface of character buffering class

Synopsis

```
#include <iostream.h>

typedef long streamoff, streampos;
class ios {
public:
    enum    seek_dir { beg, cur, end };
    enum    open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
    // and lots of other stuff; see ios on page 237
} ;

class streambuf {
public :

    int      in_avail();
    int      out_waiting();
    int      sbumpc();
    streambuf* setbuf(char* ptr, int len);
    streampos seekpos(streampos, int =ios::in|ios::out);
    streampos seekoff(streamoff, seek_dir, int =ios::in|ios::out);
    int      sgetc();
    int      sgetn(char* ptr, int n);
    int      snextc();
    int      sputbackc(char);
    int      sputc(int c);
    int      sputn(const char* s, int n);
    void      stoss();
    virtual int sync();
};
```

Description

The `streambuf` class supports buffers into which characters can be inserted (put) or from which characters can be fetched (got). Abstractly, such a buffer is a sequence of characters together with one or two pointers (a get and/or a put pointer) that define the location at which characters are to be inserted or fetched. The pointers should be thought of as pointing between characters rather than at them. This makes it easier to understand the boundary conditions (a pointer before the first character or after the last). Some of the effects of getting and putting are defined by this class but most of the details are left to specialised classes derived from `streambuf`. (See *filebuf* on page 229, *strstreambuf* on page 282, and *stdiobuf* on page 265.)

Classes derived from `streambuf` vary in their treatments of the get and put pointers. The simplest are unidirectional buffers which permit only gets or only puts. Such classes serve as pure sources (producers) or sinks (consumers) of characters. Queue-like buffers (e.g. see *strstream* on page 279 and *strstreambuf* on page 282) have a put and a get pointer which move independently of each other. In such buffers characters that are stored are held (i.e. queued) until they are later fetched. File-like buffers (e.g. *filebuf*, see *filebuf* on page 229) permit both gets and puts but have only a single pointer. (An alternative description is that the get and put pointers are tied together so that when one moves so does the other.)

Most `streambuf` member functions are organised into two phases. As far as possible, operations are performed inline by storing into or fetching from arrays (the *get area* and the *put area*, which together form the *reserve area*, or *buffer*). From time to time, virtual functions are called to deal with collections of characters in the get and put areas. That is, the virtual functions are called to fetch more characters from the ultimate producer or to flush a collection of characters to the ultimate consumer. Generally the user of a `streambuf` does not have to know anything about these details, but some of the public members pass back information about the state of the areas. Further detail about these areas is provided in *streambuf – protected* on page 266, which describes the protected interface.

The public member functions of the `streambuf` class are described below. In the following descriptions assume:

- *i*, *n*, and *len* are ints.
- *c* is an int. It always holds a ‘character’ value or EOF. A ‘character’ value is always positive even when `char` is normally sign extended.
- *sb* and *sb1* are `streambuf*s`.
- *ptr* is a `char*`.
- *off* is a `streamoff`.
- *pos* is a `streampos`.
- *dir* is a `seek_dir`.
- *mode* is an int representing an `open_mode`.

Public member functions:

***i=*sb->in_avail()**

Returns the number of characters that are immediately available in the get area for fetching. *i* characters may be fetched with a guarantee that no errors will be reported.

***i=*sb->out_waiting()**

Returns the number of characters in the put area that have not been consumed (by the ultimate consumer).

***c=*sb->sgetc()**

Moves the get pointer forward one character and returns the character it moved past. Returns EOF if the get pointer is currently at the end of the sequence.

***pos=*sb->seekoff(*off, dir, mode*)**

Repositions the get and/or put pointers. *mode* specifies whether the put pointer (`ios::out` bit set) or the get pointer (`ios::in` bit set) is to be modified. Both bits may be set in which case both pointers should be affected. *off* is interpreted as a byte offset. (Notice that it is a signed quantity.) The meanings of possible values of *dir* are

<code>ios::beg</code>	The beginning of the stream.
<code>ios::cur</code>	The current position.
<code>ios::end</code>	The end of the stream (end of file.)

Not all classes derived from `streambuf` support repositioning. `seekoff()` will return EOF if the class does not support repositioning. If the class does support repositioning, `seekoff()` will return the new position or EOF on error.

***pos=*sb->seekpos(*pos, mode*)**

Repositions the `streambuf` get and/or put pointer to *pos*. *mode* specifies which pointers are affected as for `seekoff()`. Returns *pos* (the argument) or EOF if the class does not support repositioning or an error occurs. In general a `streampos` should be treated as a 'magic cookie' and no arithmetic should be performed on it. Two particular values have special meaning:

<code>streampos(0)</code>	The beginning of the file.
<code>streampos(EOF)</code>	Used as an error indication.

`c=sb->sgetc()`

Returns the character after the get pointer. Contrary to what most people expect from the name **it does not move the get pointer**. Returns EOF if there is no character available.

`sb1=sb->setbuf(ptr, len, i)`

Offers the *len* bytes starting at *ptr* as the reserve area. If *ptr* is null or *len* is zero or less, then an unbuffered state is requested. Whether the offered area is used, or a request for unbuffered state is honoured depends on details of the derived class. `setbuf ()` normally returns *sb*, but if it does not accept the offer or honour the request, it returns 0.

`i=sb->sgetn(ptr, n)`

Fetches the *n* characters following the get pointer and copies them to the area starting at *ptr*. When there are fewer than *n* characters left before the end of the sequence `sgetn ()` fetches whatever characters remain. `sgetn ()` repositions the get pointer following the fetched characters and returns the number of characters fetched.

`c=sb->snextc()`

Moves the get pointer forward one character and returns the character following the new position. It returns EOF if the pointer is currently at the end of the sequence or is at the end of the sequence after moving forward.

`i=sb->sputbackc(c)`

Moves the get pointer back one character. *c* must be the current content of the sequence just before the get pointer. The underlying mechanism may simply back up the get pointer or may rearrange its internal data structures so the *c* is saved. Thus the effect of `sputbackc ()` is undefined if *c* is not the character before the get pointer. `sputbackc ()` returns EOF when it fails. The conditions under which it can fail depend on the details of the derived class.

`i=sb->sputc(c)`

Stores *c* after the put pointer, and moves the put pointer past the stored character; usually this extends the sequence. It returns EOF when an error occurs. The conditions that can cause errors depend on the derived class.

`i=sb->sputn(ptr, n)`

Stores the *n* characters starting at *ptr* after the put pointer and moves the put pointer past them. `sputn ()` returns *i*, the number of characters stored successfully. Normally *i* is *n*, but it may be less when errors occur.

sb->stossc()

Moves the get pointer forward one character. If the pointer started at the end of the sequence this function has no effect.

i=sb->sync()

Establishes consistency between the internal data structures and the external source or sink. The details of this function depend on the derived class. Usually this ‘flushes’ any characters that have been stored but not yet consumed, and ‘gives back’ any characters that may have been produced but not yet fetched. `sync()` returns `EOF` to indicate errors.

See also

ios (page 237), *istream* (page 248), *ostream* (page 259), *streambuf – protected* (page 266)

strstream

strstream – iostream specialised to arrays

Synopsis

```
#include <strstream.h>

class ios {
public:
    enum    open_mode { in, out, ate, app, trunc, nocreate, noreplace } ;
    // and lots of other stuff; see ios on page 237
} ;

class istrstream : public istream {
public:
    istrstream(char*) ;
    istrstream(char*, int) ;
    strstreambuf*    rdbuf() ;
} ;

class ostrstream : public ostream {
public:
    ostrstream();
    ostrstream(char*, int, int=ios::out) ;
    int        pcount() ;
    strstreambuf*    rdbuf() ;
    char*        str();
};

class strstream : public strstreambase, public iostream {
public:
    strstream();
    strstream(char*, int, int mode);
    strstreambuf*    rdbuf() ;
    char*        str();
};
```

Description

strstream specialises iostream for ‘incore’ operations, that is, storing and fetching from arrays of bytes. The streambuf associated with a strstream is a strstreambuf (see *strstreambuf* on page 282).

In the following descriptions assume:

- *ss* is a strstream.
- *iss* is an istrstream.
- *oss* is an ostrstream.

- *cp* is a `char*`.
- *mode* is an `int` representing an `open_mode`.
- *i* and *len* are `ints`.
- *ssb* is a `strstreambuf*`.

Constructors

istrstream(*cp*)

Characters will be fetched from the (null-terminated) string *cp*. The terminating null character will not be part of the sequence. Seeks (`istream::seekg()`) are allowed within that space.

istrstream(*cp*, *len*)

Characters will be fetched from the array beginning at *cp* and extending for *len* bytes. Seeks (`istream::seekg()`) are allowed anywhere within that array.

ostrstream()

Space will be dynamically allocated to hold stored characters.

ostrstream(*cp*, *n*, *mode*)

Characters will be stored into the array starting at *cp* and continuing for *n* bytes. If `ios::ate` or `ios::app` is set in *mode*, *cp* is assumed to be a null-terminated string and storing will begin at the null character. Otherwise storing will begin at *cp*. Seeks are allowed anywhere in the array.

strstream()

Space will be dynamically allocated to hold stored characters.

strstream(*cp*, *n*, *mode*)

Characters will be stored into the array starting at *cp* and continuing for *n* bytes. If `ios::ate` or `ios::app` is set in *mode*, *cp* is assumed to be a null-terminated string and storing will begin at the null character. Otherwise storing will begin at *cp*. Seeks are allowed anywhere in the array.

istrstream members

***ssb* = *iss*.rddbuf()**

Returns the `strstreambuf` associated with *iss*.

ostrstream members

***ssb* = *oss*.rdbuf()**

Returns the *strstreambuf* associated with *oss*.

***cp*=*oss*.str()**

Returns a pointer to the array being used and ‘freezes’ the array. Once *str* has been called the effect of storing more characters into *oss* is undefined. If *oss* was constructed with an explicit array, *cp* is just a pointer to the array. Otherwise, *cp* points to a dynamically allocated area. Until *str* is called, deleting the dynamically allocated area is the responsibility of *oss*. After *str* returns, the array becomes the responsibility of the user program.

***i*=*oss*.pcount()**

Returns the number of bytes that have been stored into the buffer. This is mainly of use when binary data has been stored and *oss*.str() does not point to a null terminated string.

strstream members

***ssb* = *ss*.rdbuf()**

Returns the *strstreambuf* associated with *ss*.

***cp*=*ss*.str()**

Returns a pointer to the array being used and ‘freezes’ the array. Once *str* has been called the effect of storing more characters into *ss* is undefined. If *ss* was constructed with an explicit array, *cp* is just a pointer to the array. Otherwise, *cp* points to a dynamically allocated area. Until *str* is called, deleting the dynamically allocated area is the responsibility of *ss*. After *str* returns, the array becomes the responsibility of the user program.

See also

strstreambuf (page 282), *ios* (page 237), *istream* (page 248), *ostream* (page 259)

strstreambuf

strstreambuf – streambuf specialised to arrays

Synopsis

```
#include <iostream.h>
#include <strstream.h>

class strstreambuf : public streambuf {
public:
    strstreambuf() ;
    strstreambuf(char*, int, char*);
    strstreambuf(int);
    strstreambuf(unsigned char*, int, unsigned char*);
    strstreambuf(void* (*a)(long), void(*f)(void*));

    void          freeze(int n=1) ;
    char*         str();
    virtual streambuf*      setbuf(char*, int)
};
```

Description

A `strstreambuf` is a `streambuf` that uses an array of bytes (a string) to hold the sequence of characters. Given the convention that a `char*` should be interpreted as pointing just before the char it really points at, the mapping between the abstract get/put pointers (see *streambuf – public* on page 274) and `char*` pointers is direct. Moving the pointers corresponds exactly to incrementing and decrementing the `char*` values.

To accommodate the need for arbitrary length strings `strstreambuf` supports a dynamic mode. When a `strstreambuf` is in dynamic mode, space for the character sequence is allocated as needed. When the sequence is extended too far, it will be copied to a new array.

In the following descriptions assume:

- `ssb` is a `strstreambuf*`.
- `n` is an `int`.
- `ptr` and `pstart` are `char*`s or `unsigned char*`s.
- `a` is a `void* (*)(long)`.
- `f` is a `void* (*)(void*)`.

Constructors

strstreambuf()

Constructs an empty `strstreambuf` in dynamic mode. This means that space will be automatically allocated to accommodate the characters that are put into the `strstreambuf` (using operators `new` and `delete`). Because this may require copying the original characters, it is recommended that when many characters will be inserted, the program should use `setbuf()` (described below) to inform the `strstreambuf`.

strstreambuf(*a*, *f*)

Constructs an empty `strstreambuf` in dynamic mode. *a* is used as the allocator function in dynamic mode. The argument passed to *a* will be a `long` denoting the number of bytes to be allocated. If *a* is null, operator `new` will be used. *f* is used to free (or delete) areas returned by *a*. The argument to *f* will be a pointer to the array allocated by *a*. If *f* is null, operator `delete` is used.

strstreambuf(*n*)

Constructs an empty `strstreambuf` in dynamic mode. The initial allocation of space will be at least *n* bytes.

strstreambuf(*ptr*, *n*, *pstart*)

Constructs a `strstreambuf` to use the bytes starting at *ptr*. The `strstreambuf` will be in static mode; it will not grow dynamically. If *n* is positive, then the *n* bytes starting at *ptr* are used as the `strstreambuf`. If *n* is zero, *ptr* is assumed to point to the beginning of a null terminated string and the bytes of that string (not including the terminating null character) will constitute the `strstreambuf`. If *n* is negative, the `strstreambuf` is assumed to continue indefinitely. The get pointer is initialised to *ptr*. The put pointer is initialised to *pstart*. If *pstart* is null, then stores will be treated as errors. If *pstart* is non-null, then the initial sequence for fetching (the get area) consists of the bytes between *ptr* and *pstart*. If *pstart* is null, then the initial get area consists of the entire array.

Member functions

ssb->freeze(*n*)

Inhibits (when *n* is non-zero) or permits (when *n* is zero) automatic deletion of the current array. Deletion normally occurs when more space is needed or when *ssb* is being destroyed. Only space obtained via dynamic allocation is ever freed. It is an error (and the effect is undefined) to store characters into a *strstreambuf* that was in dynamic allocation mode and is now frozen. It is possible, however, to thaw (unfreeze) such a *strstreambuf* and resume storing characters.

ptr=ssb->str()

Returns a pointer to the first `char` of the current array and freezes *ssb*. If *ssb* was constructed with an explicit array, *ptr* will point to that array. If *ssb* is in dynamic allocation mode, but nothing has yet been stored, *ptr* may be null.

ssb->setbuf(0,*n*)

ssb remembers *n* and the next time it does a dynamic mode allocation, it makes sure that at least *n* bytes are allocated.

See also

streambuf – *public* (page 274), *strstream* (page 279)

10

The Complex Math library

The Complex Math library is a part of the C++ library, ported from that supplied with AT&T's CFront product.

Introduction

complex – introduction to C++ complex mathematics library

Synopsis

```
#include <complex.h>
class    complex;
```

Description

This section describes complex mathematics functions and operators found in the C++ Library.

The Complex Mathematics library implements the data type of complex numbers as a class, `complex`. It overloads the standard input, output, arithmetic, assignment, and comparison operators, discussed in *complex operators* on page 294. It also overloads the standard exponential, logarithm, power, and square root functions, discussed in *exp*, *log*, *pow*, *sqrt* on page 292, and the trigonometric functions of sine, cosine, hyperbolic sine, and hyperbolic cosine, discussed in *cplxtrig* on page 297, for the class `complex`. Routines for converting between Cartesian and polar coordinate systems are discussed in *cartesian/polar* on page 287. Error handling is described in *complex_error* on page 289.

Diagnostics

Functions in the Complex Mathematics Library may return the conventional values $(0, 0)$, $(0, \pm\text{HUGE})$, $(\pm\text{HUGE}, 0)$, or $(\pm\text{HUGE}, \pm\text{HUGE})$, when the function is undefined for the given arguments or when the value is not representable. (`HUGE` is the largest-magnitude single-precision floating-point number and is defined in the file `<math.h>`. The header file `<math.h>` is included in the file `<complex.h>`.) In these cases, the external variable `errno` is set to the value `EDOM` or `ERANGE`.

See also

cartesian/polar (page 287), *complex_error* (page 289), *complex operators* (page 294), *exp*, *log*, *pow*, *sqrt* (page 292), *cplxtrig* (page 297).

cartesian/polar

cartesian/polar – functions for the C++ Complex Math Library

Synopsis

```
#include <complex.h>

class complex {
public:
    friend double    abs(complex);
    friend double    arg(complex);
    friend complex   conj(complex);
    friend double    imag(complex);
    friend double    norm(complex);
    friend complex   polar(double, double = 0);
    friend double    real(complex);
};
```

Description

The following functions are defined for `complex`, where:

- d , m , and a are of type `int`
- x and y are of type `complex`.

$d = \text{abs}(x)$

Returns the absolute value or magnitude of x .

$d = \text{norm}(x)$

Returns the square of the magnitude of x . It is faster than `abs`, but more likely to cause an overflow error. It is intended for comparison of magnitudes.

$d = \text{arg}(x)$

Returns the angle of x , measured in radians in the range $-\pi$ to π .

$y = \text{conj}(x)$

Returns the complex conjugate of x . That is, if x is `(real, imag)`, then `conj(x)` is `(real, -imag)`.

$y = \text{polar}(m, a)$

Creates a complex given a pair of polar coordinates, magnitude m , and angle a , measured in radians.

$d = \text{real}(x)$

Returns the real part of x .

$d = \text{imag}(x)$

Returns the imaginary part of x .

See also

Introduction (page 286), *complex_error* (page 289), *complex operators* (page 294), *exp*, *log*, *pow*, *sqrt* (page 292), *cplxtrig* (page 297)

complex_error

complex_error – error-handling function for the C++ Complex Math Library

Synopsis

```
#include <complex.h>

class c_exception
{
    int            type;
    char           *name;
    complex        arg1;
    complex        arg2;
    complex        retval;

public:
    c_exception( char *n, const complex& a1,
                const complex& a2 = complex_zero );

    friend int      complex_error( c_exception& );

    friend complex  exp( complex );
    friend complex  sinh( complex );
    friend complex  cosh( complex );
    friend complex  log( complex );

};
```

Description

In the following description of the complex error handling routine:

- *i* is of type `int`
- *x* is of type `c_exception`.

***i* = complex_error(*x*)**

Invoked by functions in the C++ Complex Mathematics Library when errors are detected.

Users may define their own procedures for handling errors, by defining a function named `complex_error` in their programs. `complex_error` must be of the form described above.

The element type is an integer describing the type of error that has occurred, from the following list of constants (defined in the header file):

SING	argument singularity
OVERFLOW	overflow range error
UNDERFLOW	underflow range error

The element name points to a string containing the name of the function that incurred the error. The variables `arg1` and `arg2` are the arguments with which the function was invoked. `retval` is set to the default value that will be returned by the function unless the user's `complex_error` sets it to a different value.

If the user's `complex_error` function returns non-zero, no error message will be printed, and `errno` will not be set.

If `complex_error` is not supplied by the user, the default error-handling procedures, described with the complex math functions involved, will be invoked upon error. These procedures are also summarised in the table below. In every case, `errno` is set to `EDOM` or `ERANGE` and the program continues.

Note that complex math functions call functions included in the math library which has its own error handling routine, `matherr`. Users may also override this routine by supplying their own version.

Default error handling procedures				
		Types of Errors		
type		SING	OVERFLOW	UNDERFLOW
errno		EDOM	ERANGE	ERANGE
EXP	real too large/small	—	($\pm H$, $\pm H$)	(0, 0)
	imag too large	—	(0, 0)	—
LOG	arg = (0, 0)	M, (H, 0)	—	—
SINH	real too large	—	($\pm H$, $\pm H$)	—
	imag too large	—	(0, 0)	—
COSH	real too large	—	($\pm H$, $\pm H$)	—
	imag too large	—	(0, 0)	—

Key:	M	Message is printed (EDOM error)
	(H, 0)	(HUGE, 0) is returned
	($\pm H$, $\pm H$)	(\pm HUGE, \pm HUGE) is returned
	(0, 0)	(0, 0) is returned

See also

Introduction (page 286), *cartesian/polar* (page 287), *complex operators* (page 294), *exp*, *log*, *pow*, *sqrt* (page 292), *cplxtrig* (page 297)

exp, log, pow, sqrt

exp, log, pow, sqrt – exponential, logarithm, power, square root functions for the C++ complex library

Synopsis

```
#include <complex.h>

class complex {
public:
    friend complex  exp(complex);
    friend complex  log(complex);
    friend complex  pow(double, complex);
    friend complex  pow(complex, int);
    friend complex  pow(complex, double);
    friend complex  pow(complex, complex);
    friend complex  sqrt(complex);
};
```

Description

The following math functions are overloaded by the complex library, where:

- x , y , and z are of type `complex`.

$z = \text{exp}(x)$

Returns e^x .

$z = \text{log}(x)$

Returns the natural logarithm of x .

$z = \text{pow}(x, y)$

Returns x^y .

$z = \text{sqrt}(x)$

Returns the square root of x , contained in the first or fourth quadrants of the complex plane.

Diagnostics

`exp` returns $(0, 0)$ when the real part of x is so small, or the imaginary part is so large, as to cause overflow. When the real part is large enough to cause overflow, `exp` returns $(\text{HUGE}, \text{HUGE})$ if the cosine and sine of the imaginary part of x are positive, $(\text{HUGE}, -\text{HUGE})$ if the cosine is positive and the sine is not, $(-\text{HUGE}, \text{HUGE})$ if the sine is positive and the cosine is not, and $(-\text{HUGE}, -\text{HUGE})$ if neither sine nor cosine is positive. In all these cases, `errno` is set to `ERANGE`.

`log` returns $(-\text{HUGE}, 0)$ and sets `errno` to `EDOM` when x is $(0, 0)$. A message indicating `SING` error is printed on the standard error output.

These error-handling procedures may be changed with the function `complex_error` (see page 289).

See also

Introduction (page 286), *cartesian/polar* (page 287), *complex_error* (page 289), *complex operators* (page 294), *cplxtrig* (page 297)

complex operators

complex_operators: operators for the C++ complex math library

Synopsis

```
#include <complex.h>

class complex {
public:
    friend complex  operator+(complex, complex);
    friend complex  operator-(complex);
    friend complex  operator-(complex, complex);
    friend complex  operator*(complex, complex);
    friend complex  operator/(complex, complex);

    friend int      operator==(complex, complex);
    friend int      operator!=(complex, complex);

    void            operator+=(complex);
    void            operator-=(complex);
    void            operator*=(complex);
    void            operator/=(complex);
};
```

Description

The basic arithmetic operators, comparison operators, and assignment operators are overloaded for complex numbers. The operators have their conventional precedences. In the following descriptions for complex operators:

- x , y , and z are of type `complex`.

Arithmetic operators:

$z = x + y$

Returns a `complex` which is the arithmetic sum of complex numbers x and y .

$z = -x$

Returns a `complex` which is the arithmetic negation of complex number x .

$z = x - y$

Returns a `complex` which is the arithmetic difference of complex numbers x and y .

$z = x * y$

Returns a `complex` which is the arithmetic product of complex numbers x and y .

 $z = x / y$

Returns a `complex` which is the arithmetic quotient of complex numbers x and y .

Comparison operators

 $x == y$

Returns non-zero if complex number x is equal to complex number y ; returns 0 otherwise.

 $x != y$

Returns non-zero if complex number x is not equal to complex number y ; returns 0 otherwise.

Assignment operators

 $x += y$

Complex number x is assigned the value of the arithmetic sum of itself and complex number y .

 $x -= y$

Complex number x is assigned the value of the arithmetic difference of itself and complex number y .

 $x *= y$

Complex number x is assigned the value of the arithmetic product of itself and complex number y .

 $x /= y$

Complex number x is assigned the value of the arithmetic quotient of itself and complex number y .

Warning

The assignment operators do not produce a value that can be used in an expression. That is, the following construction is syntactically invalid:

```
complex    x, y, z;  
x = ( y += z );
```

whereas:

```
x = ( y + z );  
x = ( y == z );
```

are valid.

See also

Introduction (page 286), *cartesian/polar* (page 287), *complex_error* (page 289), *exp*, *log*, *pow*, *sqrt* (page 292), *cplxtrig* (page 297)

cplxtrig

cplxtrig – trigonometric and hyperbolic functions for the C++ complex library

Synopsis

```
#include <complex.h>

class complex {
public:
    friend complexsin(complex);
    friend complexcos(complex);

    friend complexsinh(complex);
    friend complexcosh(complex);

};
```

Description

The following trigonometric functions are defined for `complex`, where:

- x and y are of type `complex`.

$y = \sin(x)$

Returns the sine of x .

$y = \cos(x)$

Returns the cosine of x .

$y = \sinh(x)$

Returns the hyperbolic sine of x .

$y = \cosh(x)$

Returns the hyperbolic cosine of x .

Diagnostics

If the imaginary part of x would cause overflow `sinh` and `cosh` return $(0, 0)$. When the real part is large enough to cause overflow, `sinh` and `cosh` return $(\text{HUGE}, \text{HUGE})$ if the cosine and sine of the imaginary part of x are non-negative, $(\text{HUGE}, -\text{HUGE})$ if the cosine is non-negative and the sine is less than 0,

(-HUGE, HUGE) if the sine is non-negative and the cosine is less than 0, and (-HUGE, -HUGE) if both sine and cosine are less than 0. In all these cases, *errno* is set to ERANGE.

These error-handling procedures may be changed with the function `complex_error` (see page 289).

See also

Introduction (page 286), *cartesian/polar* (page 287), *complex_error* (page 289), *complex operators* (page 294), *exp, log, pow, sqrt* (page 292)

Part 4 – Developing software for RISC OS



11 Portability

The C programming language has gained a reputation for being portable across machines, while still providing capabilities at a machine-specific level. The fact that a program is written in C by no means indicates the effort required to port software from one machine to another, or indeed from one compiler to another. Obviously the most time-consuming task is porting between two entirely different hardware environments, running different operating systems with different compilers. Since many users of the Acorn C compiler will find themselves in this situation, this chapter deals with a number of issues you should be aware of when porting software to or from our environment. The chapter covers the following:

- general portability considerations
- major differences between ISO C and the well-known 'K&R' C as defined in the book *The C Programming Language*, (first edition) by Kernighan and Ritchie
- using the Acorn C compiler in 'pcc' compatibility mode
- environmental aspects of portability.

General portability considerations

If you intend your code to be used on a variety of different systems, there are certain aspects which you should bear in mind in order to make porting an easy and relatively error-free process. It is essential to single out items which may make software system-specific, and to employ techniques to avoid non-portable use of such items. In this section, we describe general portability issues for C programs.

Fundamental data types

The size of fundamental data types such as `char`, `int`, `long int`, `short int` and `float` will depend mainly on the underlying architecture of the machine on which the C program is to run. Compiler writers usually implement these types in a manner which best fits the architectures of machines for which their compilers are targeted. For example, Release 5 of the Microsoft C Compiler has `int`, `short int` and `long int` occupying 2, 2 and 4 bytes respectively, where the Acorn C Compiler uses 4, 2 and 4 bytes. Certain relations are guaranteed by the ISO C Standard (such as the fact that the size of `long int` is at least that of `short int`), but code which makes any assumptions regarding implementation-defined issues such as whether `int` and `long int` are the same size will not be maximally portable.

A common non-portable assumption is embedded in the use of hexadecimal constant values. For example:

```
int i;
i = i & 0xffffffff8; /* set bottom 3 bits to zero, assuming 32-bit int */
```

Such non-portability can be avoided by using:

```
int i;
i = i & ~0x07; /* set bottom 3 bits to zero, whatever sizeof(int) */
```

If you find that some size assumptions are inevitable, it is best to use the specific width integer types defined in `stdint.h` (see page 160). If you need compatibility with versions of C prior to C99 (where `stdint.h` is not available) at least use a series of `assert` calls when the program starts up, to indicate any conditions under which successful operation is not guaranteed. Alternatively, write macros for frequently-used operations so that size assumptions are localised and can be altered locally.

Byte ordering

A highly non-portable feature of many C programs is the implicit or explicit exploitation of byte ordering within a word of store. Such assumptions tend to arise when copying objects word by word (rather than byte by byte), when inputting and outputting binary values, and when extracting bytes from or inserting bytes into words using a mix of shift-and-mask and byte addressing. A contrived example is the following code which copies individual bytes from an `int` variable `w` into an `int` variable pointed to by `p`, until a null byte is encountered. The code assumes that `w` does contain a null byte.

```
int a;
char *p = (char *)&a;
int w = AN_ARBITRARY_VALUE;

for (;;)
{
    if ((*p++ = w) == 0) break;
    w >>= 8;
}
```

This code will only work on a machine with even (or little-endian) byte-sex, and so is not portable. The best solution to such problems is either to write code which does not rely on byte-sex, or to have different code to deal appropriately with different byte-sex and to compile the correct variant conditionally, depending on your target machine architecture.

Store alignment

The only guarantee given in the ISO C Standard regarding alignment of members of a `struct`, is that a 'hole' (caused by padding) cannot exist at the beginning of the `struct`. The values of 'holes' created by alignment restrictions are undefined, and you should not make assumptions about these values. In particular, two structures with identical members, each having identical values, will only be considered equal if field-by-field comparison is used; a byte-by-byte, or word-by-word comparison may not indicate equality.

This may also have implications on the size requirements of large arrays of `structs`. Given the following declarations:

```
#define ARR_SIZE 10000
typedef struct
{
    int i;
    short s;
} ELEM;
ELEM arr[ARR_SIZE];
```

this may require significantly different amounts of store under, say, a compiler which aligns `ints` on even boundaries, as opposed to one which aligns them on word boundaries.

Pointers and pointer arithmetic

A deficiency of the original definition of C, and of its subsequent use, has been the relatively unrestrained interchanging between pointers to different data types and integers or longs. Much existing code makes the assumption that a pointer can safely be held in either a `long int` or `int` variable. While such an assumption may indeed be true in many implementations on many machines, it is a highly non-portable feature on which to rely.

This problem is further compounded when taking the difference of two pointers by performing a subtraction. When the difference is large, this approach is full of possible errors. For this purpose, ISO C defines a type `ptrdiff_t`, which is capable of reliably storing the result of subtracting two pointer values of the same type; a typical use of this mechanism would be to apply it to pointers into the same array. Furthermore, ISO C99 and later ISO standards define `uintptr_t` and `intptr_t` in `stdint.h`, which are integers of the correct size to hold pointers for pointer arithmetic.

Function argument evaluation

Whilst the evaluation of operands to such operators as `&&` and `||` is defined to be strictly left-to-right (including all side-effects), the same does not apply to function argument evaluation. For example, in the function call `f(i, i++)`, the issue of whether the post-increment of `i` is performed after the first use of `i` is implementation-dependent. In any case, this is an unwise form of statement, since it may be decided later to implement `f` as a macro, instead of a function.

System-specific code

The direct use of operating system calls is, as you would expect, non-portable. If you use code which is obviously targeted for a particular environment, then it should be clearly documented as such, and should preferably be isolated into a system-specific module, which needs to be modified when porting to a new machine or operating system. Pathnames of system files should be `#defined` and not hard-coded into the program, and, as far as possible, all processing of filenames should be made easy to modify. Many file operations can be written in terms of the ISO input/output library functions, which will make an application more portable. Obviously, binary data files are inherently non-portable, and the only solution to this problem may be the use of some portable external representation.

ISO C vs K&R C

The ISO C Standard has succeeded in tightening up many of the vague areas of K&R C. This results in a much clearer definition of a correct C program. However, if programs have been written to exploit particular vague features of K&R C, then their authors may find surprises when porting to an ISO C environment. In the following sections, we present a list of what we consider to be the major differences between ISO and K&R C. These differences are at the language level, and we defer discussion of library differences until a later section. The order in which this list is presented follows approximately relevant parts of the ISO C Standard Document.

Lexical elements

The ordering of phases of translation is well-defined. Of special note is the preprocessor which is conceptually token-based (which does not yield the same results as might naively be expected from pure text manipulation).

A number of new keywords have been introduced with the following meanings:

- The type qualifier `volatile` which means that the object may be modified in ways unknown to the implementation, or have other unknown side effects. Examples of objects correctly described as `volatile` include device registers, semaphores and flags shared with asynchronous signal handlers. In general, expressions involving `volatile` objects cannot be optimised by the compiler.
- The type qualifier `const` which indicates that a variable's value should not be changed.
- The type specifier `void` to indicate a non-existent value for an expression.
- The type specifier `void *`, which is a generic pointer to or from which pointer variables can be assigned, without loss of information.
- The signed type qualifier, to sign any integral types explicitly.
- `structs` and `unions` have their own distinct name spaces.
- There is a new floating-point type `long double`.
- The K&R C practice of using `long float` to denote `double` is now outlawed in ISO C.
- Suffixes `U` and `L` (or `u` and `l`), can be used to explicitly denote unsigned and long constants (eg. `32L`, `64U`, `1024UL` etc).
- The use of 'octal' constants `8` and `9` (previously defined to be octal `10` and `11` respectively) is no longer supported.
- Literal strings are to be considered as read-only, and identical strings may be stored as one shared version (as indeed they are, in the Acorn C Compiler). For example, given:

```
char *p1 = "hello";
char *p2 = "hello";
```

`p1` and `p2` will point at the same store location, where the string `hello` is held. Programs should not therefore modify literal strings.

- Variadic functions (ie those which take a variable number of arguments) are declared explicitly using an ellipsis (...). For example, `int printf(const char *fmt, ...);`
- Empty comments `/**/` are replaced by a single space (use the preprocessor directive `##` to do token-pasting if you previously used `/**/` to do this).

Conversions

ISO C uses value-preserving rules for arithmetic conversions (whereas K&R C implementations tend to use unsigned-preserving rules). Thus, for example:

```
int f(int x, unsigned char y)
{
    return (x+y)/2;
}
```

does signed division, where unsigned-preserving implementations would do unsigned division.

Aside from value-preserving rules, arithmetic conversions follow those of K&R C, with additional rules for `long double` and `unsigned long int`. It is now also possible to perform `float` arithmetic without widening to `double`. Floating-point values truncate towards zero when they are converted to integral types.

It is illegal to attempt to assign function pointers to data pointers and vice versa (even using explicit casts). The only exception to this is the value 0, as in:

```
int (*pfi)();
pfi = 0;
```

Assignment compatibility between `structs` and `unions` is now stricter. For example, consider the following:

```
struct {char a; int b;} v1;
struct {char a; int b;} v2;
v1 = v2; /* illegal because v1 and v2
          strictly have different types*/
```

Expressions

- `structs` and `unions` may be passed by value as arguments to functions.
- Given a pointer to function declared as, say, `int (*pfi)();`, then the function to which it points can be called either by `pfi();` or `(*pfi)();`.
- Due to the use of distinct name spaces for `struct` and `union` members absolute machine addresses must be explicitly cast before being used as `struct` and `union` pointers. For example:

```
((struct io_space *)0x00ff)->io_buf;
```

Declarations

Perhaps the greatest impact on C of the ISO Standard has been the adoption of function prototypes. A function prototype declares the return type and argument types of a function. For example, `int f(int, float);` declares a function returning `int` with one `int` and one `float` argument. This means that a function's argument types are part of the type of that function, thus giving the advantage of stricter argument type-checking, especially across source files. A function definition (which is also a prototype) is similar except that identifiers must be given for the arguments. For example, `int f(int i, float f);`. It is still possible to use 'old style' function declarations and definitions, but you are advised to convert to the 'new style'. It is also possible to mix old and new styles of function declaration. If the function declaration which is in scope is an old style one, normal integral promotions are performed for integral arguments, and `floats` are converted to `double`. If the function declaration which is in scope is a new style one, arguments are converted as in normal assignment statements.

Empty declarations are now illegal.

Arrays cannot be defined to have zero or negative size.

Statements

- ISO has defined the minimum attributes of control statements (eg the minimum number of case limbs which must be supported by a compiler). These values are almost invariably greater than those supported by PCCs, and so should not present a problem.
- A value returned from `main()` is guaranteed to be used as the program's exit code.
- Values used in the controlling statement and labels of a `switch` can be of any integral type.

Preprocessor

- Preprocessor directives cannot be redefined.
- There is a new `##` directive for token-pasting.
- There is a directive `#` which produces a string literal from its following characters. This is useful for cases where you want replacement of macro arguments in strings.

- The order of phases of translation is well defined and is as follows for the preprocessing phases:
 - 1 Map source file characters to the source character set (this includes replacing trigraphs).
 - 2 Delete all newline characters which are immediately preceded by \.
 - 3 Divide the source file into preprocessing tokens and sequences of white space characters (comments are replaced by a single space).
 - 4 Execute preprocessing directives and expand macros.

Any `#include` files are passed through steps 1-4 recursively.

The macro `__STDC__` is `#defined` to 1 in ISO-conforming compilers.

The ToPCC and ToANSI tools

The desktop tools ToPCC and ToANSI help you to translate C programs and headers between the ANSI/ISO and PCC dialects of C. For more details of their use and capabilities see the earlier chapters ToANSI and ToPCC.

pcc compatibility mode

This section discusses the differences apparent when the compiler is used in 'PCC' mode. When the **UNIX pcc** setup option is enabled, the C compiler will accept (Berkeley) UNIX-compatible C, as defined by the implementation of the Portable C Compiler and subject to the restrictions which are noted below.

In essence, PCC-style C is K&R C, as defined by B Kernighan and D Ritchie in their book *The C Programming Language*, with a small number of extensions and clarifications of language features that the book leaves undefined.

Language and preprocessor compatibility

In UNIX `pcc` mode, the Acorn C compiler accepts K&R C, but it does not accept many of the old-style compatibility features, the use of which has been deprecated and warned against for many years. Differences are listed briefly below:

- Compound assignment operators where the `=` sign comes first are accepted (with a warning) by some PCCs. An example is `+=` instead of `+=`. Acorn C does not allow this ordering of the characters in the token.
- The `=` sign before a `static` initialiser was not required by some very old C compilers. Acorn C does not support this syntax.

- The following very peculiar usage is found in some UNIX tools pre-dating UNIX Version 7:

```
struct {int a, b;};
double d;

d.a = 0;
d.b = 0x....;
```

This is accepted by some UNIX PCCs and may cause problems when porting old (and badly written) code.

- enums are less strongly typed than is usual under PCCs. `enum` is a non-K&R extension to C which has been standardised by ISO somewhat differently from the usual PCC implementation.
- chars are signed by default in UNIX `pcc` mode.
- In UNIX `pcc` mode, the compiler permits the use of the ISO ``...'` notation which signifies that a variable number of formal arguments follow.
- In order to cater for PCC-style use of variadic functions, a version of the PCC header file `varargs.h` is supplied with the release.
- With the exception of enums, the compiler's type checking is generally stricter than PCC's – much more akin to lint's, in fact. In writing the Acorn C compiler, we have attempted to strike a balance between generating too many warnings when compiling known, working code, and warning of poor or non-portable programming practices. Many PCCs silently compile code which has no chance of executing in just a slightly different environment. We have tried to be helpful to those who need to port C among machines in which the following varies:
 - the order of bytes within a word (little-endian versus big-endian)
 - the default size of `int` (four bytes versus two bytes in many PC implementations)
 - the default size of pointers (not always the same as `int`)
 - whether values of type `char` default to signed or unsigned `char`
 - the default handling of undefined and implementation-defined aspects of the C language.

If the verbosity of CC in UNIX `pcc` mode is found undesirable, all warnings and/or errors can be turned off using the **Suppress warnings** and/or **Suppress errors** setup options.

- The compiler's preprocessor is believed to be equivalent to UNIX's `cpp`, except for the points listed below. Unfortunately, `cpp` is only defined by its implementation, and although equivalence has been tested over a large body of UNIX source code, completely identical behaviour cannot be guaranteed. Some of the points listed below only apply when the **Preprocess only** option is used with the CC tool.
- There is a different treatment of whitespace sequences (benign).
- `n1` is processed by CC with **Preprocess only** enabled, but passed by `cpp` (making lines longer than expected).
- `Cpp` breaks long lines at a token boundary; CC with **Preprocess only** enabled doesn't (this may break line-size constraints when the source is later consumed by another program).
- The handling of unrecognised `#` directives is different (this is mostly benign).

Standard headers and libraries

Use of the compiler in UNIX `pcc` mode precludes neither the use of the standard ISO headers built in to the compiler nor the use of the run-time library supplied with the C compiler. Of course, the ISO library does not contain the whole of the UNIX C library, but it does contain almost all the commonly used functions. However, look out for functions with different names, or a slightly different definition, or those in different 'standard' places. Unless the user directs otherwise using **Default path**, the C compiler will attempt to satisfy references to, say, `<stdio.h>` from its in-store filing system.

Listed below are a number of differences between the ISO C Library, and the BSD UNIX library. They are placed under headings corresponding to the ISO header files:

ctype.h

There are no `isascii()` and `toascii()` functions, since ISO C is not character-set specific.

errno.h

On BSD systems there are `sys_nerr` and `sys_errlist()` defined to give error messages corresponding to error numbers. ISO C does not have these, but provides similar functionality via `perror(const char *s)`, which displays the string pointed to by `s` followed by a system error message corresponding to the current value of `errno`.

There is also `char *strerror(int errnum)` which, when given a purported value of `errno`, returns its textual equivalent.

math.h

The #defined value `HUGE`, found in BSD libraries, is called `HUGE_VAL` in ISO C. ISO C does not have `asinh()`, `acosh()`, `atanh()`.

signal.h

In ISO C the `signal()` function's prototype is:

```
extern void (*signal(int, void(*func)(int)))(int);
```

`signal()` therefore expects its second argument to be a pointer to a function returning `void` with one `int` argument. In BSD-style programs it is common to use a function returning `int` as a signal handler. The PCC-style function definitions shown below will therefore produce a compiler warning about an implicit cast between different function pointers (since `f()` defaults to `int f()`). This is just a warning, and correct code will be generated anyway.

```
f(signo)
int signo;
{
    .....
}

main()
{
    extern f();
    signal(SIGINT, f);
}
```

stdio.h

`sprintf()` now returns the number of characters 'printed' (following UNIX System V), whereas the BSD `sprintf()` returns a pointer to the start of the character buffer.

The BSD functions `ecvt()`, `fcvt()` and `gcvt()` are not included in ISO C, since their functionality is provided by `sprintf()`.

string.h

On BSD systems, string manipulation functions are found in `strings.h`, whereas ISO C places them in `<string.h>`. The Acorn C Compiler also has `strings.h` for PCC-compatibility.

The BSD functions `index()` and `rindex()` are replaced by the ISO functions `strchr()` and `strrchr()` respectively.

Functions which refer to string lengths (and other sizes) now use the ISO type `size_t`, which in our implementation is `unsigned int`.

stdlib.h

`malloc()` returns `void *`, rather than the `char *` of the BSD `malloc()`.

float.h

A new header added by ISO giving details of floating point precision etc.

limits.h

A new header added by ISO to give maximum and minimum limit values for data types.

locale.h

A new header added by ISO to provide local environment-specific features.

Environmental aspects

When porting an application, the most extensive changes will probably need to be made at the operating system interface level. The following is a brief description of aspects of RISC OS and Acorn C which differ from systems such as UNIX.

The most apparent interface between a C program and its environment is via the arguments to `main()`. The ISO Standard declares that `main()` is a function defined as the program entry point with either no arguments or two arguments

(one giving a count of command line arguments, commonly called `int argc`, the other an array of pointers to the text of the arguments themselves, after removal of input/output redirection, commonly called `char *argv[]`). As discussed in the section *Environment* (J.3.2) on page 87, Acorn C supports the style of input/output redirection used by UNIX BSD4.3, but does not support filename wildcarding. Further parameters to `main()` are not supported.

Under UNIX, it is common to use a third parameter, normally called `char *environ[]`, to give access to environment variables. The same effect can be achieved in our system by using `getenv()` to request system variable values explicitly; the names of these variables are as they appear from a RISC OS `*Show` command. The string pointed at by `argv[0]` is the program name (similar to UNIX, except the name is exactly that typed on invocation, so if a full pathname is used to invoke the program, this is what appears in `argv[0]`).

File naming is one of the least portable aspects in any programming environment. RISC OS uses a full stop (.) as a separator in pathnames and does not support filename extensions (nor does UNIX, but existing UNIX tools make assumptions about file naming conventions). The best way to simulate extensions is to create a directory whose name corresponds to the required extension (in a manner similar to the use of `c` and `h` directories for C source and header files).

In RISC OS 2 and RISC OS 3 filename components are normally limited to 10 characters. In later versions of RISC OS this limit has been removed. Filenames are not case-sensitive under RISC OS as they are in UNIX, so `Test` and `test` are regarded as referring to the same file and files with these names cannot co-exist in the same directory.

The RISC OS desktop uses co-operative multi-tasking where each application executes until it relinquishes control by calling the `Wimp_Poll` Software Interrupt (SWI). It is essential that applications relinquish control quite frequently to achieve a good degree of responsiveness in the desktop, and in normal use each application should not take more than 10ms to complete its poll cycle. Where a complex operation is performed, such as sorting or re-formatting it is reasonable to take much longer, but it is always preferable to perform complex tasks over several polling cycles, to give other applications a chance to respond. See the *User Interface Toolbox* guide for more details on writing RISC OS applications.

Multi-threading is not supported in current versions of RISC OS, but similar functionality can often be obtained by using callbacks. See `OS_AddCallback` in the *RISC OS Programmer's Reference Manual* for details.

Software Interrupts (SWIs)

The Acorn C compiler has support for making Software Interrupt (SWI) calls to RISC OS routines, which can be used to replace any system calls which you make under UNIX. File information, for example, can be obtained in a way similar to `stat()` under UNIX, by making an `OS_GBPB` SWI with `R0` set to the reason code `11` (full file information). Most of the UNIX low-level I/O can be simulated in this way, but the ISO C run-time library provides sufficient support for most applications to be written in a portable style.

Refer to the RISC OS *Programmer's Reference Manual* for full details of RISC OS SWIs.

SWI Functions in `kernel.h`

The include file `kernel.h` has function prototypes and appropriate typedefs for issuing SWIs. Briefly, the type `_kernel_swi_regs` allows values to be placed in registers `R0-R9`, and `_kernel_swi()` can then be used to issue the SWI; a list of SWI numbers can be found in the include file `swis.h`. For example, to set up a callback from within a module, we call the `OS_AddCallBack` SWI with the address to call in `R0` and the module's private word pointer in `R1`. This can be achieved using `_kernel_swi` as follows:

```
#include "swis.h"
#include "kernel.h"

_kernel_oserror *err;
_kernel_swi_regs regs;
regs.r[0] = (int) module_callentry;
regs.r[1] = (int) pw;
err = _kernel_swi(OS_AddCallBack, &regs, &regs);
```

For more information see the comments in the `kernel.h` header file.

SWI Functions in `swis.h`

The `swis.h` header file also provides two functions `_swi` and `_swix` for calling SWIs, which accept parameters as arguments in the function call, thus avoiding the need to set up the `_kernel_swi_regs` data structure. This works by passing details of the registers as a bit-field parameter in the function call. The macros `_IN`, `_INR`, `_OUT` and `_OUTR` are used to define the registers that are being passed in or returned. `_IN(reg)` and `_OUT(reg)` are used to define individual registers, while `_INR(reg1, reg2)` and `_OUTR(reg1, reg2)` are used to define a range of registers.

The above example can be written more succinctly using the `_swix` function as follows:

```
#include "swis.h"

_kernel_oserror *err;
err = _swix(OS_AddCallBack, _INR(0,1),
           (int) module_cbkentry, (int) pw);
```

For more information see the comments in the `swis.h` header file.

In-line SWI Functions

Finally, the compiler supports in-line SWIs via the `__swi` macro which avoids using one of the above SWI veneer functions. This is slightly faster, but has several limitations which mean it is of limited use.

The SWI function is defined as follows:

```
extern int __swi(swi-number) swi_func(params...);
```

For example, we can define a function `os_write0` which writes a null-terminated string by invoking the `OS_Write0` SWI:

```
#include "swis.h"
extern int __swi(OS_Write0) os_write0(char *);
os_write0("Hello World\n");
```

In-line SWI functions have the following features and limitations:

- A maximum of four parameters may be passed to the SWI, corresponding to registers `r0` to `r3`.
- The value returned in `r0` by the SWI is returned by the function, but values returned in other registers cannot be accessed.
- When running in 26-bit mode (ie on RISC OS version 2, 3 or 4) only SWIs which preserve flags may be called and only the non-X form of SWIs (as the X form might return an error, thus corrupting the flags).
- When running in 32-bit mode the non-X and the X form of SWIs may be used but there is no way of checking for an error being returned so it is only of use where errors are to be ignored.

Since there are so many restrictions, the use of in-line SWIs is normally restricted to special cases in libraries where they are used to optimise certain SWI calls where speed is likely to be important.

Interworking assembly language and C – writing programs with both assembly language and C parts – requires use both of ObjAsm and of CC and/or C++. Further explanation of examples is provided in the chapter *Interworking assembler with C* on page 111 of the *Acorn Assembler* guide.

Interworking assembly language and C can be very useful for construction of top quality RISC OS applications. Using this technique you can take advantage of many of the strong points of both languages. Writing most of the bulk of your application in C allows you to take advantage of the portability of C, the maintainability of a high level language and the power of the C libraries and language. Writing critical portions of code in assembler allows you to take advantage of all the speed of the Arm processor and all the features of the machine (eg use the complete floating-point instruction set).

The key to interworking C and assembler is writing assembly language procedures that obey the Arm Procedure Call Standard (APCS). This is a contract between two procedures, one calling the other. The called procedure needs to know which Arm and floating-point registers it can freely change without restoring them before returning, and the caller needs to know which registers it can rely on not being corrupted over a procedure call.

Additionally, both procedures need to know which registers contain input arguments and return arguments, and the arrangement of the stack has to follow a pattern that debuggers and so on can understand. For the specification of the APCS, see the appendix *ARM procedure call standard* on page 297 of the *Desktop Tools* guide.

This chapter explains how C uses the APCS, in terms of the appearance of assembly language optionally output by CC and the way the stack set up by the C run-time library works.

Register names

The following names are used in referring to Arm AArch32 registers:

a1	R0	Argument 1, also integer result, temporary
a2	R1	Argument 2, temporary
a3	R2	Argument 3, temporary
a4	R3	Argument 4, temporary
v1	R4	Register variable
v2	R5	Register variable
v3	R6	Register variable
v4	R7	Register variable
v5	R8	Register variable
v6	R9	Register variable
s1	R10	Stack limit
fp	R11	Frame pointer
ip	R12	Temporary work register
sp	R13	Lower end of current stack frame
lr	R14	Link address on calls, or workspace
pc	R15	Program counter and processor status
f0	F0	Floating point result
f1	F1	Floating-point work register
f2	F2	Floating-point work register
f3	F3	Floating-point work register
f4	F4	Floating-point register variable (must be preserved)
f5	F5	Floating-point register variable (must be preserved)
f6	F6	Floating-point register variable (must be preserved)
f7	F7	Floating-point register variable (must be preserved)

In this section, ‘at [r]’ means at the location pointed to by the value in register r; ‘at [r, #n]’ refers to the location pointed to by r+n. This accords with ObjAsm’s syntax.

Register usage

The following points should be noted about the contents of registers across function calls.

- Calling a function (potentially) corrupts the argument registers a1 to a4, ip, lr, and f0–f3. The calling function should save the contents of any of these registers it may need.
- Register lr is used at the time of a function call to pass the return link to the called function; it is not necessarily preserved during or by the function call.

- The stack pointer `sp` is not altered across the function call itself, though it may be adjusted in the course of pushing arguments inside a function. The limit register `s1` may change at any time, but should always represent a valid limit to the downward growth of `sp`. User code will not normally alter this register.
- Registers `v1` to `v6`, and the frame pointer `fp`, are expected to be preserved across function calls. The called procedure is responsible for saving and restoring the contents of any of these registers which it may need to use.

Control arrival

At a procedure call, the convention is that the registers are used as follows:

- `a1` to `a4` contain the first four arguments. If there are fewer than four arguments, just as many of `a1` to `a4` as are needed are used.
- If there are more than four arguments, `sp` points to the fifth argument; any further arguments will be located in succeeding words above `[sp]`.
- `fp` points to a backtrace structure.
- `sp` and `s1` define a temporary workspace of at least 256 bytes available to the procedure.
- `s1` contains a stack chunk handle, which is used by stack handling code to extend the stack in a non-contiguous manner.
- `lr` contains the value which should be restored into `pc` on exit from the called procedure.
- `pc` contains the entry address of the called procedure.

Passing arguments

All integral and pointer arguments are passed as 32-bit words. Floating point 'float' arguments are 32-bit values, 'double'-argument 64-bit values. These follow the memory representation of the IEEE single and double precision formats.

Arguments are passed **as if** by the following sequence of operations:

- Push each argument onto the stack, last argument first.
- Pop the first four words (or as many as were pushed, if fewer) of the arguments into registers `a1` to `a4`.
- Call the function, for example by the `branch with link` instruction:

`BL functionname`

In many cases it is possible to use a simplified sequence with the same effect (eg load three argument words into `a1-a3`).

If more than four words of arguments are passed, the calling procedure should adjust the stack pointer after the call, incrementing it by four for each argument word which was pushed and not popped.

Return link

On return from a procedure, the registers are set up as follows:

- `fp`, `sp`, `s1`, `v1` to `v6` and `f4` to `f7` have the same values that they contained at the procedure call.
- Any result other than a floating point or a multi-word structure value is placed in register `a1`.
- A floating point result should be placed in register `f0`.

Structure values returned as function results are discussed below.

Structure results

A C function which returns a multi-word structure result is treated in a slightly different manner from other functions by the compiler. A pointer to the location which should receive the result is added to the argument list as the first argument, so that a declaration such as the following:

```
s_type afunction(int a, int b, int c)
{
    s_type d;
    /* ... */
    return d;
}
```

is in effect converted to this form:

```
void afunction(s_type *p, int a, int b, int c)
{
    s_type d;
    /* ... */
    *p = d;
    return;
}
```

Any assembler-coded functions returning structure results, or calling such functions, must conform to this convention in order to interface successfully with object code from the C compiler.

Storage of variables

The code produced by the C compiler uses argument values from registers where possible; otherwise they are addressed relative to `fp`, as illustrated in *Examples* below.

Local variables, by contrast, are always addressed with positive offsets relative to `sp`. In code which alters `sp`, this means that the offset for the same variable will differ from place to place. The reason for this approach is that it permits the stack overflow procedure to recover by changing `sp` and `s1` to point to a new stack segment as necessary.

Function workspace

The values of `sp` and `s1` passed to a called function define an area of readable, writable memory available to the called function as workspace. All words below `[sp]` and at or above `[s1, #-512]` are guaranteed to be available for reading and writing, and the minimum allowed value of `sp` is `s1-256`. Thus the minimum workspace available is 256 bytes.

The C run-time system, in particular the stack extension code, requires up to 256 bytes of additional workspace to be left free. Accordingly, all called functions which require no more than 256 bytes of workspace should test that `sp` does not point to a location below `s1`, in other words that at least 512 bytes remain. If the value in `sp` is less than that in `s1`, the function should call the stack extension function `__rt_stkovf_split_small`. Functions which need more than 256 bytes of workspace should amend the test accordingly, and call `__rt_stkovf_split_big`, as described below. The following examples illustrate a method of performing this test.

Note that these are the C-specific aliases for the kernel functions `_kernel_stkovf_split_0frame` and `_kernel_stkovf_split_frame` respectively, described in the chapter *The shared C library* in the *RISC OS Programmer's Reference Manual*. For backwards compatibility the alternative aliases `x$stack_overflow` and `x$stack_overflow1` are also defined.

Examples

The following fragments of assembler code illustrate the main points to consider in interfacing with the C compiler. If you want to examine the code produced by the compiler in more detail for particular cases, you can request an assembler listing by enabling the **Assembler** option on the CC SetUp menu.

This is a function `gggg` which expects two integer arguments and uses only one register variable, `v1`. It calls another function `ffff`.

```

        AREA    |C$$code|, CODE, READONLY
        IMPORT  |ffff|
        IMPORT  |__rt_stkovf_split_small|
        EXPORT  |gggg|
gggx    DCB     "gggg", 0           ;name of function, 0 terminated
        ALIGN   ;padded to word boundary
gggy    DCD     &fff000000 + gggy - ggxx
        ;dist. to start of name
;Function entry: save necessary regs. and args. on stack
gggg    MOV     ip, sp
        STMFD   sp!, {a1, a2, v1, fp, ip, lr, pc}
        SUB     fp, ip, #4         ;points to saved pc
;Test workspace size
        CMP     sp, sl
        BLMI    |__rt_stkovf_split_small|
;Main activity of function
; ....
        ADD     v1, v1, #1         ;use a register variable
        BL      |ffff|            ;call another function
        CMP     v1, #99            ;rely on reg. var. after call
; ....
;Return: place result in a1, and restore saved registers
        MOV     a1, result
        LDMEA   fp, {v1, fp, sp, pc}

```

If a function will need more than 256 bytes of workspace, it should replace the two-instruction workspace test shown above with the following:

```

        SUB     ip, sp, #n
        CMP     ip, sl
        BLMI    |__rt_stkovf_split_big|

```

where `n` is the number of bytes needed. Note that `__rt_stkovf_split_big` must be called if more than 256 bytes of frame are needed. `ip` must contain `sp_needed`, as shown in the example above.

A function which expects a variable number of arguments should store its arguments in the following manner, so that the whole list of arguments is addressable as a contiguous array of values:

```

        MOV     ip, sp             ;copy value of sp
        STMFD   sp!, {a1, a2, a3, a4} ;save 4 words of args.
        STMFD   sp!, {v1, v2, fp, ip, lr, pc}
        ;save v1-v6 needed
        SUB     fp, ip, #20        ;fp points to saved pc
        CMP     sp, sl            ;test workspace
        BLMI    |__rt_stkovf_split_small|

```

Some complete program examples are described in the chapter *Interworking assembler with C* on page 111 of the *Acorn Assembler* guide.

RISC OS is a fast, efficient multi-tasking operating system for Arm processors. It was originally designed by Acorn Computers Ltd for its range of desktop computers, including the Archimedes, A4, A5000, A7000 and Risc PC. It has since been used in a wide range of other products including Set Top Boxes and Internet Appliances.

This chapter discusses some of the important considerations when developing software for RISC OS and in particular issues in developing software that is compatible with earlier versions of RISC OS running on older Arm processors.

32-bit compatibility

RISC OS was originally designed for the ARM2 and ARM3 processors, which had a 26-bit address bus and a combined 24-bit program counter and 8-bit status flags in register R15.

The ARM6 expanded the address bus and program counter to 32 bits and moved the status flags to a separate register, but included backwards-compatible 26-bit modes to allow ARM2 and ARM3 user programs to run unmodified. The latest Arm processors have removed these 26-bit compatibility modes in favour of the new 16-bit *Thumb* architecture. As a result, versions of RISC OS for newer processors such as the ARM9, ARM10 and XScale will be 32-bit only, using only 32-bit modes.

In the past RISC OS has used only 26-bit modes for backwards compatibility (with the exception of FIQ handlers and, in RISC OS 4, the FPEmulator). Indeed, RISC OS 4 and earlier versions do not support operation in 32-bit modes for general applications or modules.

The components included here will allow programs to be built so they run on any system from an ARM2 Archimedes to a modern ARMv7 architecture platform. This is achieved by creating 26/32-bit neutral code that performs the same when running in either a 26-bit or 32-bit mode.

Converting existing C applications to 32-bit form is very simple – just re-compile them using this compiler and the associated libraries and ensure that the latest modules are distributed and installed with the application, as explained below.

A few APIs will change slightly in 32-bit versions of RISC OS to support the full addressing range. Some existing SWIs such as OS_ReadLine use a single register to hold a pointer and flags. This limits the address range that can be used and could cause the flags to be corrupted if high pointer addresses were used. The new 32-bit APIs will avoid these problems while retaining backwards compatibility as far as possible and you should refer to the relevant documentation for details.

If you compile with `-arch 4` or later, also specify `-memaccess -L22-S22` if you want your software to be Risc PC compatible. This is to stop the compiler from using the ARMv4 halfword memory access instructions, which although available on the StrongARM, do not work in the Risc PC.

The shared C library

The SharedCLibrary module supplied with this version of the Acorn C/C++ compiler is suitable for RISC OS 3.10 or later. It requires FPEmulator 4.03 or later (to support the LFM and SFM instructions) and it also requires the CallASWI module on RISC OS 3.60 or earlier.

The C library supports the new APCS-32 calling standard, together with the existing APCS-R. It no longer supports APCS-A.

On a 32-bit RISC OS APCS-R will no longer be supported, unless via some form of 26-bit emulation software.

Ensuring the necessary components are present

It is important that RISC OS applications check the installed versions of the software components that they need and load new versions, as appropriate.

C programs compiled to run in 32-bit mode will need SharedCLibrary 5.17 or later (this was the first 32-bit compatible shared C library).

C programs which use the new C99 library functions, 64 bit integers or variable length arrays will need SharedCLibrary 5.43 or later.

C programs which use 64 bit file pointers (see *Large file support* on page 121) will need SharedCLibrary 5.64 or later.

C programs which use the new C18 library functions `aligned_alloc()`, `at_quick_exit()`, `quick_exit()`, or `timespec_get()` will need SharedCLibrary 6.05 or later.

It is recommended that the necessary components are tested and loaded as needed by the application's !Run file, as follows:


```
RMEnsure UtilityModule 3.10 Error This application requires RISC OS 3.10 or
later
RMEnsure UtilityModule 3.70 RMEnsure CallASWI 0.02 RMLoad
System:Modules.CallASWI
RMEnsure UtilityModule 3.70 RMEnsure CallASWI 0.02 Error This application
requires CallASWI 0.02 or later
RMEnsure FPEmulator 4.03 RMLoad System:Modules.FPEmulator
RMEnsure FPEmulator 4.03 Error This application requires
FPEmulator 4.03 or later
RMEnsure SharedCLibrary 5.17 RMLoad System:Modules.CLib
RMEnsure SharedCLibrary 5.34 Error This application requires SharedCLibrary
5.34 or later
```

Note that you should always specify 5.17 (the first 32-bit version) as the version of SharedCLibrary required in the first RMEnsure - this will avoid the potentially fatal possibility of killing a RAM version that is currently in use (killing the ROM version is safe). The second RMEnsure asks for 5.34, as that is the first non-beta version. If you do require a newer version, only change the second RMEnsure.

14 How to write relocatable modules in C

Relocatable modules are the basic building blocks of RISC OS and the means by which RISC OS can be extended by a user. The archetypal use for RISC OS extensions is the provision of device drivers for devices attached to hardware.

Relocatable modules also provide mechanisms which can be exploited to:

- extend RISC OS's repertoire of built-in commands (* commands)
- provide services to applications (for example, as does the shared C library module)
- implement applications.

A complete discussion of these topics is beyond the scope of this chapter.

For modules which provide services, the principal mechanism for accessing those services from user code is the SoftWare Interrupt (SWI). For example, the shared C library implements a handler for a single SWI which, when called from the library stubs linked with the application, returns the address of the C library module which in turn allows the library stubs to be initialised to point to the correct addresses within the library module. Thereafter, library services are accessed directly by procedure call, rather than by SWI call. All this illustrates is the rich variety of mechanisms available to be exploited.

Getting started

To write a module in C you will need:

- the CC and CMHG tools supplied with Acorn C/C++
- the C Library stubs supplied with Acorn C/C++
- a thorough understanding of RISC OS modules (read the *Modules* chapter of the *RISC OS Programmer's Reference Manual*).

Constraints on modules written in C

A module written in C will nearly always use the shared C library module via the library stubs. In the exceptional circumstances where you need to use the stand-alone C library – typically to enforce the use of a specific version – you **must** link with the module variant (`C:o.ansilibm` rather than `C:o.ansilib`).

Modules that use any of the C99 library functions, long long variables or variable length arrays require the SharedCLibrary 5.43 or later to be loaded as explained in *C99 features* on page 97.

Normally all components of a module written in C **must** be compiled with the compiler SetUp menu option **Module code** enabled (equivalent to the `-zM` command line option). This allows the module's static data to be separated from its code and multiply instantiated. The only exception to this rule is for special cases where the `module-is-not-re-entrant` keyword has been specified.

Prior to RISC OS 4.00, floating point instructions could not be used in modules, except in the module's runnable application code.

Floating point instructions can be used in supervisor mode with RISC OS 4.00 and later versions, but if used in the SWI or other handler functions it is necessary to preserve the floating point registers, as the CMHG veneers do not preserve them. This can be achieved by writing some assembler veneers or by using the new `FPEmulator_ChangeContext` SWI which is in FPEmulator 4.14 and later. Floating point exceptions within modules should be avoided and are best disabled.

Floating point in interrupts and callbacks should also save the floating point context in the same way.

Overview of modules written in C

A module written in C includes the following:

- a Module Header (described in the *Modules* chapter of the *RISC OS Programmer's Reference Manual*), constructed using CMHG;
- a set of entry and exit 'veneers', interfacing the module header to the C run-time environment (also constructed using CMHG);
- the stubs of the shared C library;
- code written by you to implement the module's functionality – for example:
 - *command handlers, SWI handlers and service call handlers.

These parts must be linked together using the Link tool with the SetUp box **Module** option enabled.

The next section describes:

- how to write a CMHG input file to make a module header and any necessary entry veneers
- the interface definitions to which each component of your module must conform
- how to write a CMHG input file to generate entry veneers for vector and event handlers written in C.

Functional components of modules written in C

The following components may be present in a module written in C (all are optional except for the title string and the help string which are obligatory):

- Runnable application code (called start code in the module header description). This will be present if you tell CMHG that the module is runnable and include a `main()` function amongst your module code. This code is executed by an `RMRUN` command but not by an `RMLoad` command. Note that the runnable code does not have some of the restrictions that normally apply to module code, for example, it is possible to allocate large amounts of memory on the stack.
- Initialisation code. 'System' initialisation code is always present, as the shared library must be initialised. Your initialisation function will be called after the system has been initialised if you declare its name to CMHG.
- Finalisation code. The C library has to be closed down properly on module termination. Your own finalisation code will be called before the system has been closed down if you declare its name to CMHG.
- Service call handler. This will be present if you declare the name of a handler function to CMHG. In addition, you can give a list of service call numbers which you wish to deal with and CMHG will generate fast code to ignore other calls without calling your handler.
- A title string in the format described in the *RISC OS Programmer's Reference Manual*. CMHG will insist that you give it a valid title string.
- A help string in the format described in the *RISC OS Programmer's Reference Manual*. Again, CMHG will insist that you give a valid help string.
- An optional date string which is added to the help string. If this is not specified the date is taken from the timestamp of the CMHG input file.
- Help and command keyword table. This section is optional and will be present only if you describe it to CMHG and declare the names of the command handlers to CMHG. Obviously, their implementations must be included in the linked module.

- International help file. Specified when messages in the help and command keyword table are internationalised using a messages file.
- SWI chunk base number. Present only if declared to CMHG.
- SWI handler code. Present if you declare the name of a handler function to CMHG.
- SWI decoding table. Present only if described to CMHG.
- SWI decoding code. Present only if you declare the name of your decoding function to CMHG.
- Vector handlers. CMHG will generate entry veneers for vector handlers. You can register these veneers with RISC OS using SWI OS_Claim, etc; you have to provide implementations of the handlers themselves. The names of the handler functions and of the entry veneers have to be given to CMHG.
- An event handler. CMHG can generate an entry veneer for an event handler. You can register this veneer with RISC OS using SWI OS_Claim; you have to provide the implementation of the handler itself. The name of the handler function and of the entry veneer has to be given to CMHG.
- Generic veneers. CMHG will generate generic veneers for callbacks, etc. You can register these veneers with RISC OS using SWI OS_AddCallBack, OS_CallEvery, etc; you have to provide implementations of the handlers themselves. The names of the handler functions and of the entry veneers have to be given to CMHG.
- Library initialisation code. This can be used to intercept the initialisation of the C library when the module is first loaded.
- Library enter code. If declared, the initialisation of the C library when the module is started as an application can be intercepted.
- Library finalisation code. This component can be declared to intercept the finalisation of the C library when the module is killed.

Each component that you wish to use must be described in your input to CMHG. Use of most components also requires that you write some C code which must conform to the interface descriptions given in the sections below.

The C module header generator

The C Module Header Generator (CMHG) is a special-purpose assembler of module headers. It accepts as input a text file describing which module facilities you wish to use and generates as output a linkable object module (in Arm Object Format). For details of how to run the CMHG tool, see the chapter entitled CMHG earlier in this manual.

The format of input to CMHG

Input to CMHG is in free format and consists of a sequence of 'logical lines'. Each logical line starts with a keyword which is followed by some number of parameters and (sometimes) keywords. The precise form of each kind of logical input line is described in the following sections.

A logical line can be continued on the next line of input immediately after a comma (that is, if the next non-white-space character after a comma is a newline then the line is considered to be continued).

Lists of parameters can be separated by commas or spaces, but use of comma is required if the line is to be continued.

A comment begins with a `;` and continues to the end of the current line. A comment is valid anywhere that trailing white space is valid (and, in particular, after a comma).

A keyword consists of a sequence of alphabetic characters and minus signs. Often, a keyword is the same as the description of the corresponding field of the module header (as described in the *RISC OS Programmer's Reference Manual*) but with spaces replaced by minus signs. For example: `initialisation-code`; `title-string`; `service-call-handler`.

Keywords are always written entirely in lower case and are always immediately followed by a `:`. Character case is significant in all contexts: in keywords, in identifiers, and in strings.

Numbers used as parameters are unsigned. Three formats are recognised:

- unsigned decimal
- `0xhhh...` (up to 8 hex digits)
- `&hhh...` (up to 8 hex digits).

In the following sections, the parts headed *CMHG description* tell you what you have to describe to CMHG in order to use the facility described in that section; the parts headed *C interface* introduce a description of the interface to which the handler function you write must conform.

Using the preprocessor

CMHG will preprocess the CMHG file using the C preprocessor if the `-p` command line option is used. This is particularly useful for using symbolic variable names defined in header files, rather than entering numeric values which are then defined in multiple files.

For example, this CMHG text:

```
swi-chunk-base-number:    0x82880
service-call-handler:    Window_services 0x46, 0x44ec1, 0x44ec2
```

can be replaced by the more readable and maintainable:

```
#include "window.h"
#include "services.h"
swi-chunk-base-number:    Window_SWIChunkBase
service-call-handler:    Window_services Service_ModeChange,
                        Service_ToolboxTaskBorn, Service_ToolboxTaskDied
```

It also allows conditional inclusion of CMHG lines:

```
#ifdef DEBUG
command-keyword-table:    Window_commands Window_Memory(), Window_Tasks()
#endif
```

The command-line options `-I` and `-D` may be used to specify the include path and to define macros, as for the C compiler. See *Command line options* on page 61.

Runnable application code

CMHG description:

```
module-is-runnable:          ; No parameters.
```

C interface:

```
/*
 * Start code
 * =====
 *
 * Entered in user-mode with argc and argv
 * set up as for any other application. Malloc
 * obtains storage from application workspace.
 */
int main(int argc, char *argv[]);
```

To be useful (ie re-runnable), a runnable application must implement at least one * command handler (see below) for its command line, which, when invoked, enters the module (calls SWI OS_Module with the Enter reason code).

Note that the runnable application code is only executed when the module is RMRUN. It is not executed when RMLoad is used).

atexit functions set up in the runnable application code will be called when the application code exits. This behaviour is different to atexit functions set up in initialisation code.

Initialisation code

CMHG description:

```
initialisation-code: user_init ; The name of your initialisation function.
                           ; Any valid C function name will do.
```

C interface:

```
/*
 * Initialisation code
 * =====
 *
 * Return NULL if your initialisation succeeds; Return the special value
 * initialise_NO_STATIC_DATA if your initialisation succeeds but you do not require
 * static data for this instantiation (including shared library static
 * data - you should assume that any shared library call might need to use
 * static data). Otherwise return a pointer to an error block.
 * cmd_tail points to the string of arguments with which the module is
 * invoked (may be "", and is control-terminated, not zero terminated).
 * podule_base is the 'R11' value established by module initialisation
 * which is the podule base address if the code has been invoked from a
 * podule, otherwise it is the number of other instantiations of the module.
 * pw is the 'R12' value established by module initialisation. You may
 * assume nothing about its value (in fact it points to some RMA space
 * claimed and used by the module veneers). All you may do is pass it back
 * for your module veneers via an intermediary such as SWI OS_CallEvery
 * (use _swix() to issue the SWI call).
 */
_kernel_oserror *user_init(const char *cmd_tail, int podule_base, void *pw);
```

Note that you can choose any valid C function name as the name of your initialisation code (CMHG insists on no more than 31 characters).

atexit functions set up in the module initialisation code or in other module routines, other than main, will be called on module finalisation.

Finalisation code

CMHG description:

```
finalisation-code: user_final ; The name of your finalisation function.
                           ; Any valid C function name will do.
```

C interface:

```
/*
 * Finalisation code
 * =====
 *
 * Return NULL if your finalisation succeeds. Otherwise return a pointer to
 * an error block if your finalisation handler does not wish to die (e.g.
 * toolbox modules return a 'Task(s) active' error).
 * fatal, podule and pw are the values of R10, R11 and R12 (respectively)
 * on entry to the finalisation code.
 */
_kernel_oserror *user_final(int fatal, int podule, void *pw);
```

A call to library finalisation code is inserted automatically by CMHG; the C library finalisation code will call your finalisation handler immediately before closing down the library (on module finalisation).

Service call handler

CMHG description:

```
service-call-handler:  sc_handler <number> <number> ...
```

C interface:

```
/*
 * Service call handler
 * =====
 *
 * Return values should be poked directly into r->r[n]; the right
 * value/register to use depends on the service number (see the relevant
 * RISC OS Programmer's Reference Manual section for details).
 * pw is the private word (the 'R12' value).
 */
void sc_handler(int service_number, _kernel_swi_regs *r, void *pw);
```

Service calls provide a generic mechanism. Some need to be handled quickly; others are not time critical. Because of this, you may give a list of service numbers in which you are interested and CMHG will generate code to ignore the rest quickly. The fast recognition code looks like:

```
TEQ      r1, #FirstInterestingServiceNumber
TEQNE    r1, #SecondInterestingServiceNumber
...
TEQNE    r1, #NthInterestingServiceNumber
MOVNE    pc, lr                ; drop into service call entry veneer.
```

If you give no list of interesting service numbers then all service calls will be passed to your handler.

In order to construct a relocatable module which implements a RISC OS application (by using the `module-is-runnable` keyword) you must claim and deal with the `Service_Memory` service call. See the relevant section in the *RISC OS Programmer's Reference Manual* for details of this service call.

The following is a suitable handler written in C for this service call:

```
#define Service_Memory 0x11
void FrontEnd_services(int service_number, _kernel_swi_regs *r, void *pw)
{
    IGNORE(pw);
    /* keep application workspace (r2 holds CAO pointer) */
    if (service_number == Service_Memory && r->r[2] ==
        (int)Image__RO_Base)
    {
        r->r[1] = 0; /* refuse to relinquish app. workspace */
    }
}
```

The above handler needs to compare the contents of `r[2]` with the address of the base of your module containing it. This is not a value directly available in C, so the following assembly language fragment can be used to gain access to the symbol `Image$$RO$$Base`, which is defined by Link when your module is linked together:

```
IMPORT |Image$$RO$$Base|
EXPORT Image__RO_Base

AREA Code_Description, DATA, REL
Image__RO_Base
DCD |Image$$RO$$Base|

END
```

Title string

CMHG description:

`title-string: title`

`title` must consist entirely of printable, non-space ASCII characters.

Any underscores in the title are replaced by spaces. CMHG will fault any title longer than 31 characters and warn if the length of the title string is more than 16.

Help string

CMHG description:

`help-string: help d.dd comment ; help string and version number`

The help string is restricted to 15 or fewer alphanumeric, ASCII characters and underscores. Longer strings are truncated (with a warning) to 15 characters then padded with a single space. Shorter titles are padded with one or two TAB characters so they will appear exactly 16 characters long.

The version number must consist of a digit, a dot, then 2 consecutive digits. Conventionally, the first digit denotes major releases; the second digit minor releases; and the third digit bug-fix or technical changes. If the version number is omitted, 0.00 is used.

CMHG automatically inserts the date into the version string, as required by RISC OS convention. The date may be specified with the `date-string` keyword or if none is specified it defaults to the timestamp of the CMHG input file. Note that earlier versions of CMHG used the current date, but that behaviour has been removed as it caused the string date to change when a module was rebuilt without any modifications.

A 'comment' of up to 34 characters can also be included after the version number. It will appear in the tail of the module's help string, after the date. A typical use is for annotating the help string in the following style:

```
SomeModule      0.92 (14 Oct 2002) Experimental version
```

CMHG refuses to generate a help string longer than 79 characters and warns if it has to truncate your input.

Date string

CMHG description:

```
date-string: date ; date module was released
```

This date string is free format, but should normally be the day of the month, the three letter month name and the four digit year, separated by spaces. For example:

```
date-string: 14 Jun 2003
```

The date is used in the help string as explained above.

Help and command keyword table

CMHG description:

```
command-keyword-table: cmd_handler command-description+
```

(Here *command-description*+ denotes one or more command descriptions).

A command-description has the format:

```
star-command-name " ("
    min-args:      unsigned-int      ; default 0
    max-args:      unsigned-int      ; default 0
    gstrans-map:    unsigned-int      ; default 0
    fs-command:     ; set bit 31 in the flags byte
    status:         ; set bit 30 in the flags byte
    configure:      ; set bit 30 in the flags byte
    help:           ; set bit 29 in the flags byte
    international:  ; set bit 28 in the flags byte
    invalid-syntax: text
    help-text:      text
    add-syntax:     ; append invalid-syntax string to
                   ; help-text string
    ") "
```

Each sub-argument is optional. A comma after any item allows continuation on the next line.

A *text* item follows the conventions of ISO C string constants: it is a sequence of implicitly concatenated string segments enclosed in " and " .

Segments may be separated by white space or newlines (no continuation comma is needed following a string segment).

Within a string segment \ introduces an escape character. All the single character ASCII escapes are implemented, but hexadecimal and octal escape codes are not implemented. A \ immediately preceding a newline allows the string segment to be continued on the following line (but does **not** include a newline in the string; if a newline is required, it must be explicitly included as \n).

min-args and *max-args* record the minimum and maximum number of arguments the command may accept; *gstrans-map* records, in the least significant 8 bits, which of the first 8 arguments should be subject to expansion by OS_GSTrans before calling the command handler.

The keywords *fs-command*, *status*, *configure*, *help* and *international* set bits in the command's information word which mark the command as being of one of those classes. The *status* and *configure* keywords are synonyms; both set bit 30 of the flags word.

The `invalid-syntax` and `help-text` messages should be self-explanatory. If `add-syntax` is also specified then the `invalid-syntax` string will be appended to the `help-text` string when it is printed. Note that modules designed for RISC OS 3.60 or later should use the internationalised help flag (`international`). This option is then superfluous.

The `international` flag indicates that the `invalid-syntax` and `help-text` messages are tokens to be looked up in the messages file specified by the `international-help-file` keyword (see below).

Example CMHG description:

```
command-keyword-table: cmd_handler
tm0(    min-args: 0, max-args: 255,
        help-text: "Syntax:\ttm0 <filenames>\n"),
tm1(    min-args:1, max-args:1,
        add-syntax:,
        help-text: "tm1 does something.\n",
        invalid-syntax: "Syntax:\ttm1" " <integer>"
        "\n"),
tm2(    min-args:1, max-args:2,
        international:,
        help-text: "HTM2",
        invalid-syntax: "STM2")
```

This describes three `*cmd` commands, `*tm0`, `*tm1` and `*tm2`, which are to be handled by the C function `cmd_handler`. The handler function will be called with 0 as its third argument if it is being called to handle the first command (`tm0`, above), 1 as its third argument if it is being called to handle the second command (`tm1`, above), etc. The programmer must keep the CMHG description in step with the implementation of `cmd_handler`.

The `*tm2` command illustrates the use of the `international` flag. The `help-text` and `invalid-syntax` parameters specify the names of tokens that will be looked up in a messages file. The path to the messages file must be specified by using the `international-help-file` keyword (see below).

C interface:

```
/*
 * Command handler
 * =====
 *
 * If cmd_no identifies a command, then arg_string gives the command tail
 * (which you may not overwrite), and argc is the number of parameters.
 * NB. arg_string is control terminated so it may not be a C string.
 * Return NULL if the command has been successfully handled; otherwise
 * return a pointer to an error block describing the failure (in this
 * case, the veneer code will set the 'V' bit).
 *
 * If cmd_no identifies a *Help entry, then arg_string denotes a buffer
 * that you can assemble your output into, and argc is the length of the
 * buffer, in bytes. cmd_handler must return NULL, an error pointer or
```

```

* help_PRINT_BUFFER (if help_PRINT_BUFFER is returned, the zero-
* terminated buffer will be printed).
*
* If cmd_no identifies a *Configure option, then arg_string may contain
* one of the two special values arg_CONFIGURE_SYNTAX or arg_STATUS;
* otherwise it points at the command tail, with leading spaces skipped.
* If arg_string is set to arg_CONFIGURE_SYNTAX, the user has typed
* *Configure with no parameter; simply print your syntax string. If
* arg_string is set to arg_STATUS, print your current configured status.
* Otherwise, the user has typed *Configure with one or more parameters
* as described in the command tail. The parameter argc contains an
* undefined value in all three cases. Return NULL, an error pointer, or
* one of the four special values defined below.
*
* pw is the private word pointer ('R12') value passed into the entry
* veneer
*/
#define help_PRINT_BUFFER      ((_kernel_oserror *) arg_string)
#define arg_CONFIGURE_SYNTAX  ((char *) 0)
#define arg_STATUS            ((char *) 1)
#define configure_BAD_OPTION  ((_kernel_oserror *) -1)
#define configure_NUMBER_NEEDED ((_kernel_oserror *) 1)
#define configure_TOO_LARGE   ((_kernel_oserror *) 2)
#define configure_TOO_MANY_PARAMS ((_kernel_oserror *) 3)

#define CMD_tm0                0
#define CMD_tm1                1
#define CMD_tm2                2

_kernel_oserror *cmd_handler(const char *arg_string, int argc, int cmd_no, void
*pw);

```

International help file

CMHG description:

international-help-file: *path* ; message file pathname

This gives the pathname of the message file to be used when outputting text from any entries in the help and command keyword table that have the international flag set.

SWI chunk base number

CMHG description:

swi-chunk-base-number: *number*

You should use this entry if your module provides any SWI handlers. It denotes the base of a range of 64 values which may be passed to your SWI handler. SWI chunks are allocated by RISC OS Open and if your software is to be distributed to other RISC OS users you **must** contact RISC OS Open to get a chunk allocated uniquely

to your product. See the chapter *An introduction to SWIs* in the *RISC OS Programmer's Reference Manual* for more details. Information on how to request an allocation can be found at <https://www.riscosopen.org/content/allocate/>.

SWI handler code

CMHG description:

swi-handler-code: swi_handler ; any valid C function name will do

C interface:

```
/*
 * SWI handler code
 * =====
 *
 * swi_offset contains the offset of the SWI into your SWI chunk.
 * r points to the registers passed to the SWI.
 *
 * Return NULL if the SWI is handled successfully; otherwise return
 * a pointer to an error block which describes the error.
 * The veneer code sets the 'V' bit if the returned value is non-NULL.
 * The special value error_BAD_SWI may be returned if you do not
 * implement a SWI; the veneer will arrange for the appropriate
 * standard internationalised error 'SWI value out of range for
 * module Example' to be returned.
 * The handler may update any of its input registers (R0-R9).
 * pw is the private word pointer ('R12') value passed into the
 * SWI handler entry veneer.
 */
#define error_BAD_SWI ((_kernel_oserror *) -1)

_kernel_oserror *swi_handler(int swi_offset, _kernel_swi_regs *r, void *pw);
```

If your module is to handle SWIs then it must include both `swi-handler-code` and `swi-chunk-base`.

Example CMHG description:

```
swi-chunk-base-number: 0x88000
swi-handler-code:      widget_swi
```


SWI decoding table

CMHG description:

```
swi-decoding-table: swi-base-name swi-name*
```

This table, if present, is used by OS_SWINumberTo/FromString.

Example CMHG description:

```
swi-chunk-base-number: 0x88000
swi-handler-code:      widget_swi
swi-decoding-table:    Widget,
                      Init  Read  Write  Close
```

This would be appropriate for the following name/number pairs:

```
Widget_Init      0x88000
Widget_Read      0x88001
Widget_Write     0x88002
Widget_Close     0x88003
```

SWI decoding code

CMHG description:

```
swi-decoding-code: name_decode/number_decode; any valid C function names
```

C interface:

```
/*
 * SWI decoding code
 * =====
 *
 * For the text->number converter, string points to the name to convert
 * (terminated by a control character). If the SWI is recognised, return
 * the offset into the chunk (0-63), otherwise return <0.
 *
 * For the number->text converter:
 *   swi_no is the offset (0..63) of the SWI within the SWI chunk.
 *   buffer is a pointer to a buffer;
 *   offset is the offset within the buffer at which to place the text;
 *   size is the length of the buffer.
 * You should write the SWI name into the buffer at the position given
 * by offset, then return offset updated by the length of the text
 * written (excluding any terminating NUL, if you add one). If you don't
 * have a name for the SWI, just return offset, unaltered.
 *
 * pw is the private word pointer ('R12') passed into the entry veneer.
 */
int name_decode(const char *string, void *pw);
int number_decode(int swi_no, char *buffer, int offset, int size, void *pw);
```

CMHG description (single function version for backwards compatibility):

```
swi-decoding-code: swi_decode; any valid C function name will do
```

C interface (single function version for backwards compatibility):

```
/*
 * SWI decoding code
 * =====
 *
 * On entry, r[0] < 0 means a request to convert from text to a number.
 * In this case r[1] points to the string to convert (terminated by a
 * control character, NOT necessarily by NUL).
 * Set r[0] to the offset (0..63) of the SWI within the SWI chunk if
 * you recognise its name; set r[0] < 0 if you don't recognise the name.
 *
 * On entry, r[0] >= 0 means a request to convert from a SWI number to
 * a SWI string:
 *   r[0] is the offset (0..63) of the SWI within the SWI chunk.
 *   r[1] is a pointer to a buffer;
 *   r[2] is the offset within the buffer at which to place the text;
 *   r[3] is the length of the buffer.
 * You should write the SWI name into the buffer at the position given
 * by r[2] then update r[2] by the length of the text written (excluding
 * any terminating NUL, if you add one).
 *
 * pw is the private word pointer ('R12') passed into the entry veneer.
 *
 * For a more elegant interface, supply a pair of '/'-separated function
 * names to CMHG's swi-decoding-code command.
 */
void swi_decode(int r[4], void *pw);
```

If you omit a SWI decoding table then your SWI decoding code will be called instead. Of course, you don't have to provide either.

Vector handlers

CMHG description:

vector-handlers: entry_name/handler_name ...

Any number of entry_name/handler_name pairs may be given. If you omit the / and the handler name, CMHG constructs a handler name by appending `_handler` to the entry name.

C interface:

```
/*
 * Vector handlers
 * =====
 *
 * This is the name of the vector handler entry veneer
 * compiled by CMHG. Use this name as an argument to SWI
 * OS_Claim. (EventV claimants should use a CMHG event handler).
 */
```

```

* Note that vector handlers were previously called IRQ handlers
* and were documented as being for attaching to IrqV. IrqV has
* long been deprecated; you should use OS_ClaimDeviceVector
* instead.
*/
extern void entry_name(void);

/*
* This is the handler function you must write to handle the
* vector for which entry_name is the veneer function.
*
* If a handler function is installed onto a vector, then:
*   Return VECTOR_CLAIM to intercept the call.
*   Return VECTOR_PASSON to pass on the call.
* If you use a vector handler veneer for any other purpose, always
* return non-0, and consider the use of a generic veneer instead.
* It is not currently possible to return an error from a vector
* handler.
*
* 'r' points to a vector of words containing the values of R0-R9 on
* entry to the veneer. If r is updated, the updated values will be
* loaded into R0-R9 on return from the handler.
*
* pw is the private word pointer ('R12') value with which the
* vector entry veneer is called.
*/
int handler_name(_kernel_swi_regs *r, void *pw);

```

Handlers must be installed from some part of the module which runs in SVC mode (eg initialisation code, a SWI handler, etc). The name to use at installation time is the `entry_name` (**not** the name of the handler function). This is because C functions have to be entered and exited via a veneer which switches the static base context and changes to SVC mode if necessary.

Refer to the *RISC OS Programmer's Reference Manual* for information on how to install and remove handlers.

Event handler

CMHG description:

event-handler: entry_name/handler_name event_no event_no ...

Only one entry_name/handler_name pair may be given.

C interface:

```
/*
 * Event handler
 * =====
 *
 * This is the name of the event handler entry veneer compiled by CMHG.
 * Use this name as an argument to, for example, SWI OS_Claim, in
 * order to attach your handler to EventV.
 */
extern void entry_name(void);

/*
 * This is the handler function you must write to handle the event for
 * which entry_name is the veneer function.
 *
 * Return 0 if you wish to claim the event.
 * Return 1 if you do not wish to claim the event.
 *
 * 'r' points to a vector of words containing the values of R0-R9 on
 * entry to the veneer. If r is updated, the updated values will be
 * loaded into R0-R9 on return from the handler.
 *
 * pw is the private word pointer ('R12') value with which the event
 * entry veneer is called.
 */
int handler_name(_kernel_swi_regs *r, void *pw);
```

The name to use at installation time is the `entry_name` (**not** the name of the handler function). Refer to the *RISC OS Programmer's Reference Manual* for details and for information on how to install and remove event handlers. As an example, this is the skeleton of an event handler for key presses and mouse clicks:

```
/* the claim/free functions... */
#define EventV 16
#define EnableEvent 14
#define DisableEvent 13
#define MouseClick 10
#define Keypress 11

static void claim_release(int claim, void *pw)
{
    _swix(claim ? OS_Claim : OS_Release, _INR(0,2),
        EventV, register_event, pw);
}
```

```

static void add_remove(int add)
{
    _swix(OS_Byte, _INR(0,1),
        add ? EnableEvent : DisableEvent, MouseClick);
    _swix(OS_Byte, _INR(0,1),
        add ? EnableEvent : DisableEvent, Keypress);
}

static void claim_free_events(int claim,void *pw)
{
    if (claim) {
        claim_release(1,pw);
        add_remove(1);
    } else {
        add_remove(0);
        claim_release(0,pw);
    }
}

/* init... */
extern _kernel_oserror *events_init(char *cmd_tail, int podule_base, void *pw)
{
    IGNORE(cmd_tail);
    IGNORE(podule_base);
    claim_free_events(1,pw);
    return NULL;
}

/* finalise... */
extern _kernel_oserror *events_final (int fatal, int podule, void *pw)
{
    IGNORE(fatal);
    IGNORE(podule);
    /* handle low level events */
    claim_free_events(0,pw);
    return NULL;
}

/* the handler itself... */
extern int event_handler(_kernel_swi_regs *r,void *pw)
{
    IGNORE(pw);
    /* switch on the event code */
    switch (r->r[0]) {
        case MouseClick:
        case Keypress:
            break;
        default:
            break;
    }
    return 1;
}

```

Generic veneers

CMHG description:

generic-veneers: entry_name/handler_name ...

Any number of entry_name/handler_name pairs may be given. If you omit the / and the handler name, CMHG constructs a handler name by appending _handler to the entry name.

C interface:

```
/*
 * Generic veneers
 * =====
 *
 * This is the name of the generic entry veneer compiled by CMHG.
 * Use this name as an argument to, for example, SWI OS_CallEvery
 * or OS_AddCallBack.
 *
 * These veneers ensure that your handlers preserve R0-R11
 * and the processor flags (unless you return an error pointer.
 * The veneer can be entered in either IRQ or SVC mode. R12 and
 * R14 are corrupted.
 */
extern void entry_name(void);

/*
 * This is the handler function that the veneer declared above
 * calls.
 *
 * For a standard exit, return NULL. For handlers that can return an
 * error, return an error block pointer, and the veneer will set the
 * 'V' bit, and set R0 to the error pointer.
 *
 * 'r' points to a vector of words containing the values of R0-R9 on
 * entry to the veneer. If r is updated, the updated values will be
 * loaded into R0-R9 on return from the handler.
 *
 * pw is the private word pointer ('R12') value with which the
 * entry veneer is called.
 */
_kernel_oserror *handler_name(_kernel_swi_regs *r, void *pw);
```

Handlers must be installed from some part of the module which runs in SVC mode (eg initialisation code, a SWI handler, etc). The name to use at installation time is the entry_name (**not** the name of the handler function). This is because C functions have to be entered and exited via a veneer which switches the static base context and changes to SVC mode if necessary.

Generic veneers are general-purpose entry points that can be used for things like callback handlers registered with `OS_AddCallBack` and filing system entry points. They work in the same way as vector handlers, except they either return preserving processor flags, or set `V` to return an error.

Library initialisation code

CMHG description:

```
library-initialisation-code: xxxx
```

The code `xxxx` is called instead of `_clib_initialisemodule`. Because the C library has not been initialised at this point the first thing it must do is call `_clib_initialisemodule`.

Library enter code

CMHG description:

```
library-enter-code: xxxx
```

The code `xxxx` is called instead of `_clib_entermodule` when the module is started as an application. The C library has only been partially initialised at this point, so there is no USR mode stack or heap. Because this entry point is called in USR mode, this requires it to be written in assembler. You may choose to perform some actions and then branch to `_clib_entermodule` (this will set up the heap and stack in the application slot, parse the command line and branch to `main()`, and will not return to you) or you may choose to replace this functionality in its entirety.

Library finalisation code

CMHG description:

```
library-finalisation-code: xxxx
```

The code `xxxx` is called instead of `_clib_finalisemodule`. The last thing it should do is call `_clib_finalisemodule`.

Non-reentrant modules

CMHG description:

```
module-is-not-re-entrant:           ; No parameters.
```

Indicates to the C library stubs that it need not copy the module data into a separate RMA workspace area. This could save memory, but as it will not be possible to re-initialise or re-instantiate the module this option should not be used in normal circumstances.

Turning interrupts on and off

The following (<kernel.h>) library functions support the control of the interrupt enable state:

```
int _kernel_irqs_disabled(void);
/*
 * Returns non-0 if IRQs are currently disabled.
 */

void _kernel_irqs_off(void);
/*
 * Disable IRQs.
 */

void _kernel_irqs_on(void);
/*
 * Enable IRQs.
 */
```

These functions allow you to disable interrupts around a piece of code that is not interrupt safe, then restore the original state afterwards:

```
int irqs_were_on = !_kernel_irqs_disabled();
if (irqs_were_on)
    _kernel_irqs_off();
/* do atomic stuff... */
if (irqs_were_on)
    _kernel_irqs_on();
```

15 Overlays

Overlays are a very old technique for squeezing quart-sized programs into pint-sized memories: a kind of poor man's paging.

In common with paged programs, an overlaid program is stored on some backing store medium such as a hard disc and its components (called overlay segments) are loaded into memory only as required. In theory, this reduces the amount of memory required to run a program at the expense of increasing the time taken to load it and repeatedly re-load parts of it. It is a classic space-time trade-off. In practice, except in rather special circumstances, the saving in memory accruing from the use of overlays is rather modest and less than you might expect. Indeed, as discussed below, overlays have rather restricted applicability under RISC OS. Nonetheless, they can occasionally be useful when dealing with systems that have limited memory.

Paging vs overlays

In a paged system, a program and its workspace is broken up into fixed size chunks called *pages*. A combination of special hardware and operating system support ensures that pages are loaded only when needed and that un-needed pages are soon discarded. In principle, the author of a paged program need not be aware that it will be paged (but this is often not true in practice if the author wishes the program to run at maximum speed). Both code and data are paged, automatically. In general, for single programs which re-use their workspace whenever possible, one sees a ratio of program size plus workspace size to occupied memory size in the region 1.5 to 3. One can always increase the ratio arbitrarily by integrating several sequentially used programs into a single image and by never re-using workspace. But, fundamentally, paging rarely squeezes more than a quart-sized program into a pint-sized memory. Of course, there are other benefits of paging, but these are beyond the scope of this section.

In contrast, an overlaid program is broken up into variable sized chunks (called overlay segments) by the user, who also determines which of these chunks may share the same area of memory. As the overlay system permits two code fragments which share the same area of memory to call one another and return successfully to the caller, this is merely a matter of performance. However, if data is included in an overlaid segment the situation becomes more complicated and the user has more work to do. For example, it must be ensured that all code which uses the data resides in the same segment as the data. Furthermore, it must be acceptable that

the data is re-initialised every time the segment is re-loaded. Thus, in general, it is possible to overlay two work areas each of which is private to two distinct sets of functions which are not simultaneously resident in memory. Overall, it would be unusual to overlay more than a quart-sized program into a pint-sized memory, much as with paging (you may achieve a factor as high as four for code, but non-overlaid data will usually dilute the overall factor substantially; it all depends on the details of your application).

A more detailed description of the low-level aspects of overlays is given in the section *Generating overlaid programs* on page 156 of the *Desktop Tools* guide. If you are especially interested in using overlays you may prefer to read that section next. Otherwise, if you are more interested in when to use overlays, please read on.

When to use overlays

Overlays work best when a program has several semi-independent parts. A good model for purposes of understanding is to think of a special-purpose command interpreter (the root segment) which can invoke separate commands (overlay segments) in response to user input. Consider, for example, a word processor which consists of a text editor and a collection of printer drivers. It is clear that each of the printer drivers can be overlaid (you are unlikely to have more than one printer); it may even be plausible to overlay each with the editor itself (you may not be able to edit while printing – depending on how fast the printer goes and on how much CPU time is required to drive it). Furthermore, if the time taken to load an overlay segment can be tacked on to an interaction with the user, it is probable that the program will feel little slower than if it were memory-resident. In summary: overlays work best if your program has many independent sub-functions.

On the other hand, if your program has many semi-independent parts, it may be better to structure it as several independent programs, each called from a control program. By using the shared C library, each program can be relatively small, and the Squeeze utility can be used to reduce the space taken by it on backing store by nearly a factor of 2. (See the chapter *Squeeze* on page 173 of the *Desktop Tools* guide for details). In contrast, overlay segments cannot be squeezed (though the root program can be). So, if you can structure your application as independent, squeezed programs it may take up less precious disc space and load faster, especially from a floppy disc, than if you structure it using overlays.

If adopted, this strategy will force the independent programs to communicate via files. Provided the data to be communicated has a simple structure this causes no problems for the application; provided it is not too voluminous, use of the RAM filing system (RamFS) is suggested as this is fast and requires no special application code in order to use it.

So, overlays are most appropriate for applications which manipulate very large amounts of highly structured data – Computer Aided Design applications are archetypal here – whereas multiple independent programs are most appropriate for applications which manipulate relatively small amounts of simply structured data and are otherwise dominated by large amounts of code.

Part 5 – Appendices



Appendix A: Changes to the C compiler

Acorn Computers produced five releases of the C/C++ compiler product for RISC OS, the final one being released as part of the Desktop Development Environment in 1994. The next release was made by Castle Technology Ltd. in 2002, adding support for 32-bit versions of RISC OS. Since 2009 the compiler and tools have been maintained by RISC OS Open Ltd. This appendix details the changes that have been made since the final release from Acorn.

Changes up to version 5.54

- The compiler now defaults to generating APCS-32 code and the libraries have been updated for use in a 32-bit environment. The resulting executable is suitable for modern processors such as the ARM9 or Intel XScale processors which do not have a 26-bit mode, while still being suitable for older processors such as the ARM2, ARM3, ARM6 and ARM7 families.
- The compiler and other tools may now be used in a 32-bit environment.
- Many C99 features are now available in the compiler, but note that the standard C libraries are not C99 compliant. 64 bit integers (long long) are supported.
- Improved compiler error messages and new warnings, The compiler now reports which function parameter it is complaining about when one causes an error. The compiler also generates a warning if number of parameters passed in `_swi` and `_swix` calls does not match the specified format. Warnings may be disabled using options described in *Handling warnings and errors* on page 34.
- C compiler has command line options, including `-ccversion version` on page 48
- CMHG has several new command line options and keyword. See *Command line options* on page 61 and *The format of input to CMHG* on page 341.

Changes up to version 5.56

- Added support for `__packed` structures and pointers. See *Packed Structures* on page 82.
- Added inline assembler. See *Inline assembler* on page 122.
- Added Thumb interworking (`-APCS /inter`).

-
- `int f() { }` is now correctly treated as an old-style function. Use `int f(void) { }` instead.
 - Improved error reporting for function arguments.
 - C99 pragmas added. See *Standard pragmas* on page 108.
 - C99 concatenation of wide and normal string literals supported.
 - Universal character name and UTF-8 support. See *Implementation details* on page 78 and *Universal character names* on page 109.
 - Support for LDRH etc added.
 - `-Otime` and `-Ospace` now have more effect.
 - CSE optimises long long expressions.
 - Optimisation of static integer and floating const variables.
 - long long shifts by constant much improved.
 - All long long multiplies will now be inlined (if `-cpu 3M` or later and `-Otime`). Can also generate SMLAL and UMLAL.
 - ARMv5 BLX instruction used for faster function pointer calls.
 - Other more complex function pointer forms (such as `MOV lr, pc` and `LDR pc, xxx`) reinstated; these had been disabled at some point between 5.02 and 5.54 as a bug workaround.
 - Very slow CSE analysis of complex functions (eg various cyphers) fixed.
 - Header file guard `#defines` interpreted and handled internally (a speed optimisation which saves having to reload header files).
 - Improved compile-time evaluation of floating arithmetic.
 - Able to inline more `<math.h>` functions. Now knows the CNF instruction.
 - FPA float to double conversions no longer eliminated (for correct behaviour with signalling NaNs and underflow traps).
 - Fixes to inline functions with difficult return types.
 - Trailing padding removed from structs with flexible array members.
 - Constant data placed in read-only area.
 - Mapping symbols (`$d/$a`) now placed in object file to aid DecAOF and DDT.
 - long long variables now output correctly in debugging information, (requires latest version of DDT).
 - Changes to warning control options:
 - `-fo` (to enable `Old-style function`) removed - it is now enabled by default for C90 and C99 modes.

- `-Wd` now disables `Old-style` function as well as deprecated declaration.
- Warnings that are normally off can now be enabled with, `-W+`.
- `-W+g` enables warnings about unguarded or wrongly guarded header files.
- The old `-Wg` option (non-constant `printf` format) is now `-Wh`.
- `-Wl` controls lower precision in wider context separately from `-Wn`, which still controls both that and implicit narrowing cast.
- `-Ws` (the static initialiser in static data warning) has been removed.
- `-Wx` disables declared but not used warnings.
- `-W+u` enables warnings about C++ keywords. This replaces the old `-Wc` option, which is now obsolescent.
- Added `-zz` and `-zt` options to control data allocation. See *Global data options* on page 52.
- APCS-A and APCS-M support removed.

Changes up to version 5.59

- Added support for switch on a long long expression.
- Added complex and imaginary number support (`<complex.h>`, including C99 Annex G).
- `_Bool` (and `_Complex/_Imaginary`) now available in pcc and C90 modes.
- MLA instruction used more often, and handled more intelligently.
- CSE aliasing optimisations take account of `restrict` qualifier. See *Restricted pointers* on page 103.
- Optimised handling of narrow (<32 bit) data and computations.
- Will use ARMv5E's `SMULxy` and `SMLAxy` instructions where appropriate.
- Improved checking of `printf/scanf` format strings.
- Can now inline `signbit()`.
- Improved compilation of `(int) longlong`, `(int) longlong >> 32`, and `longlong >> 1` or `<< 1`. Also, long long multiplication and division by powers of two transformed into shifts.
- Can transform integer division by constant into a 32x32->64 multiplication (if available on CPU, and `-Otime` selected).
- Pointer subtraction optimised to use only multiply and/or shift.
- Added inter-statement compile-time evaluation of long long arithmetic.

-
- Improved CSE handling, especially for FP constants and comparisons.
 - IEEE 754 conformance improved; generally edging the compiler closer to C99 Annex F.
 - `asm` keyword recognised in C++ mode.
 - `-arch` command-line parameter added. See *Keyword options* on page 45.
 - Some improvements to treatment of volatile objects.
 - Changes to handling of floating arguments for `-apcs /fpregargs`.
 - Banner and help now sent to `stderr` instead of `stdout`.

Changes up to version 5.64

- Added instruction scheduling.
- Various other performance enhancements in low-level code generation.
- Various CSE enhancements.
- Copy-propagation optimisation added to register allocator.
- Now defaults to tuning performance for XScale, while remaining ARMv2 compatible (ie `-cpu XScale -arch 2`).
- BLX opcode added to inline assembler.
- Enums can now use smaller containers rather than always being int-sized. See *t* on page 34
- Some hard-coded function size limits increased.

Changes up to version 5.69

- Change to unaligned load behaviour. The default setting is now not to use unaligned loads as a shortcut to rotate the aligned 32-bit value, because the behaviour of unaligned loads has changed (optional in ARMv6, mandatory in ARMv7). The same thing can be achieved on earlier versions using `-memaccess [+/-] L22 [+/-] S22-L41`.
- Added the following code generation and optimisation options: `-arch 6k`, `-arch 6t2`, `-arch 7` and `-cpu cortex-a8`. When `-arch 7` or `-cpu cortex-a8` is specified, then a new set of scheduling rules are selected which target an approximation of the Cortex-A8.

Changes up to version 5.71

- Added support for `#warning` to the pre-processor. See *#warning directive* on page 112.
- You can now use `-wz` to suppress Undefined macro in `#if - treated as 0`.

- Code tuning for ARM11 now available using `-cpu ARM1176JZF-S`.
- The built-in standard C headers now incorporate the changes from the shared C library, up to version 5.64. This includes a correction to `FOPEN_MAX` as well as support for 64-bit file pointers (see *Large file support* on page 121). `wchar.h` and `wctype.h` are also included for the first time, but note that the shared C library does not currently implement the functions defined therein.
- Static structures can now be initialised using compound literals.

Changes up to version 5.74

- Module relocation offsets are now shown correctly if a disassembly is requested using `-zM -S` on the command line.
- Will use the MLS instruction for ARMv6T or later where an expression of the form `a = b - (c * d)` is encountered.
- The `-cpu` switch now recognises architectures `2a/3g/5tewmmx/5tewmmx2/6z` for consistency with ObjAsm.

Changes up to version 5.76

- The inline assembler now permits compare operations on the PC. This allows the common idiom `TEQ pc, pc` to be used to check if the processor is in 32 bit mode. Other ALU operations involving the PC are still faulted because the result would be undefined.
- The inline assembler now accepts the BKPT instruction. Note that BKPT is only available from ARMv5 and later. If no debugger is attached the processor will cause a prefetch abort.

Changes up to version 5.78

- A new command line option, `-fpu`, is introduced to select the Floating Point Unit to target.
- The `-cpu` switch now recognises architectures `7-a/7-r/7-a.security/8-a` and `Cortex-A5/Cortex-A7/Cortex-A9/Cortex-A15` or consistency with ObjAsm.
- Using `SWP[B]` in the inline assembler for a `-cpu` or `-arch` that doesn't support that instruction will now emit a warning.
- When the option `-apcs/interwork` is in force `__APCS_INTERWORK` is now predefined.
- The maximum size limit for a statically declared array or structure has been increased to 16MB.

Changes up to version 5.83

- When `-fpu None` is specified any attempt to use floating point will report an error.
- The table of known CPUs has been further extended (to closely match `ObjAsm`) up to and including the Cortex-A57.
- The code generator now knows about the ARMv6/v7/v8 instruction sets. Where appropriate and permitted by the `-cpu` or `-arch` selection it is now possible to infer from C source code: `BFC`, `BFI`, `MOVW/MOVT`, `NOP`, `REV`, `PKHBT/PKHTB`, `SXTB/SXTH`, `UXTAB/SXTAB`, `UXTAH/SXTAH`, `UBFX/SBFX`.
- The inline assembler now accepts all of the ARMv6/v7/v8 mnemonics, up to and including ARMv8.4. Note that the same restrictions apply as before with regards to mnemonics that change the stack pointer or program counter directly, therefore use of `ERET`, `RFE`, `SRS` is not permitted.
- A number of Arm intrinsic functions have been made available to allow C-like functions to produce specialist instructions without needing to resort to inline assembler.

Changes up to version 5.88

- Added support for the extra C18 features and keywords.
- In support of C18 the built in headers `<stdalign.h>`, `<stdnoreturn.h>` and `<uchar.h>` have been added, along with other appropriate additions to the existing library headers.
- The table of known CPUs has been further extended (to closely match `ObjAsm`) up to and including the Cortex-X1. The table entries for v8 now take into account those processors which don't support the CRC instruction and will warn on its use in the inline assembler.

Changes up to version 5.91

- The value of `CHAR_MIN` and `CHAR_MAX` in `<limits.h>` now adaptively change to reflect whether a `char` is signed or unsigned, controlled by command line option `-zc`.
- Calls to the `_swix` library function may now be inlined directly as a SWI instruction, skipping the library call entirely. This optimisation applies to SWIs which the compiler knows follow APCS-like calling conventions taking up to 4 register arguments and returning with `V` set if an error occurred.
- The ANSI header files `<stdatomic.h>` and `<threads.h>` are recognised and will not trigger a warning if included. Note that these headers are not built-in to the compiler and must be provided as an implementation on disc.
- Predefines from the Arm C Language Extensions 2021Q2 are now set.

Changes up to version 5.93

- The `_Atomic` keyword is now recognised and syntax checked by the compiler. Note that the qualifier doesn't current affect code generation, but it allows a disc based implementation to support both qualifier and specifier forms by judicious use of the preprocessor.
- The compiler will now warn about suspect switch/case blocks specifically when a case can never be reached because either the width of the control variable makes it impossible (for example, case 500 for an unsigned char width value), or switching on an enum where the case is not a member of that enum.
- Enabling warning `-W+s` will now diagnose missed cases in a switch/case block where the control variable is an enumerated value. All of the enum values must have coverage, or a `default` case to catch those missing.
- Expressions which compare a pointer with a string literal are now warned as likely programming mistakes, most likely `strcmp` was intended.
- APCS qualifier `/vfp` is permitted and has the effect of reversing the word order for any double precision constants embedded in the object file. These options are also passed to ObjAsm if the compiler is asked to process an assembler source file, so that the resulting object files will have matching calling convention attributes set.
- Attempts to redefine any of the built-in preprocessor names, such as `__TIME__`, now produce a distinct error message.



Appendix B: C errors and warnings

This appendix gives a brief description of the intended purposes of error and warning messages from the CC tool, along with some hints for interpreting them. It then lists most of the common errors in alphabetical order. It is not a complete list. Since the messages are designed as far as possible to be self-explanatory, some of the more simple common ones are not listed here.

Interpreting CC errors and warnings

The compiler can produce error and warning messages of several degrees of severity. They are as follows:

- **Warnings** indicating curious, but legal, program constructs, or constructs that are indicative of potential error;
- **Non-serious errors** that still allow code to be produced;
- **Serious errors** that may cause loss of code;
- **Fatal errors** that may stop the compiler from compiling;
- **System errors** that signal faults in the system itself.

Warnings from CC are intended to provide a helpful level of checking, in addition to the level required by the ISO standard. On some other systems, such as UNIX, separate facilities (like lint) are provided to perform this checking. Warnings flag program constructs that may indicate potential errors, or those not recommended because they may function differently on other machines, and hence hinder the portability of code.

Some warnings point out the use of facilities provided in this ISO C implementation which are above the minimum required by ISO – for example, use of external identifiers that are identical in the first six characters, which may not be differentiated by other systems which conform to the ISO standard.

Programs ported from other machines may cause large numbers of warning messages from CC, which you can disable with the **Suppress warnings** option (see page 33 for more information).

You can also enable additional checks with the CC and C++ **Features** option. This is best done in the final stages of a project, and will help you to produce high-quality software.

Errors and serious errors collectively respond to ISO 'diagnostics'; whether an error is serious or not reflects the compiler's view, not yours, or that of the ISO committee.

After issuing a warning, non-serious, or serious error, CC continues compiling, sometimes producing more such messages in the process. Compilation of C by CC can be thought of as a pipeline process, starting with preprocessing, syntax analysis, then semantic analysis (when the structure of a portion of code is analysed). When syntax errors in C are encountered by CC, the compiler can often guess what the error was, correct it, and continue. When semantic errors are found, portions of your code are often ignored before continuing, and serious error messages are reported.

Unfortunately, the compact and powerful nature of C leads to a high proportion of semantic errors. Ignoring portions of your code is likely to make subsequent portions incorrect, so one serious error can often start a cascade of error messages. Often, therefore, it is sensible to ignore a set of error messages following a serious error message.

If the compiler produces any message more serious than a warning, it will set a bad return code, usually terminating any 'make' of which it is a part in the process. Any serious error will cause the output object file to be deleted; fatal and system errors cause immediate termination of compilation, with loss of the object file and bad return code set.

Future releases of the compiler may distinguish further forms of error, or produce slightly different forms of wording.

In pcc mode, constructs that are erroneous in ISO mode are reported, even though legal in pcc mode.

Warnings

Warning messages indicate legal but curious C programs, or possibly unintended constructs (unless warnings are suppressed). On detection of such a condition, the compiler issues a warning message, then continues compilation.

Warning messages

'&' unnecessary for function or array *xx*

This is a reminder that if *xx* is defined as `char xx[10]` then *xx* already has a pointer type. There is a similar reminder for function names too. Example:

```
static char mesg[] = "hello\n";
int main ()
{
    char *p = &mesg; /* mesg is already compatible with char * */
    ...
}
```

actual type '*xx*' mismatches format '*%x*'

A type error in a `printf` or `scanf` format string. Example:

```
{
    int i;
    printf("%s\n", i); /* %s need char* not int */
    ...
}
```

ANSI '*xx*' trigraph for '*x*' found - was this intended?

This helps to avoid inadvertent use of ANSI/ISO trigraphs. Example:

```
printf("Type ??/!/: "); /* "??" is trigraph for "\" */
```

argument and old-style parameter mismatch : *xx*

A function with a non-ISO declaration has been called using a parameter of a wrong data type. Example:

```
int fnl(a , b)
int a;
int b;
{
    return a * b;
}
...
int main()
{
    int l; float m;
    fnl(l , m);          /* m should be 'int' */
    ...
}
```

character sequence /* inside comment

You cannot nest comments in C. Example:

```
/* comment out func() for now...
/* func() returns a random number */
int func(void)
{
    ...
    return i;
}
*/
```

Dangling 'else' indicates possible error

This hints that you may have mismatched your ifs and elses. Remember an else always refers to the most recent unmatched if. Use braces to avoid ambiguity. Example:

```
if (a)
    if (b)
        return 1;
    else if (c)
        return 2;
else /* this belongs to the if (a). Or does it?*/
    return 3;
```

Deprecated declaration of xx() - give arg types

A feature of the ISO standard is that argument types should be given in function declarations (prototypes). 'No arguments' is indicated by void. Example:

```
extern int func(); /* should have 'void' in the parentheses */
```

extern clash xx , xx clash (ANSI 6 char monospace)

Using compiler **Feature** option e, it was found that two external names were not distinct in the first six characters. Some linkers provide only six significant characters in their symbol table. Example:

```
extern double function1 (int i);
extern char * function2 (long l);
```

extern 'main' needs to be 'int' function

This is a reminder that `main()` is expected to return an integer. Example:

```
void main()
{
    ...
}
```

extern *xx* not declared in header

Compiling with **Feature** h, an external object was discovered which was not declared in any included header file.

floating point constant overflow

This is typically caused by a division by zero in a floating point constant expression evaluated at compile time. Example:

```
#define lim 1
#define eps 0.01
static float a = eps/(lim-1); /* lim-1 yields 0 */
```

floating to integral conversion failed

A cast (possibly implicit) of a floating point constant to an integer failed at compile time. Example:

```
static int i = (int) 1.0e20; /* INT_MAX is about 2e10 */
```

formal parameter '*xx*' not declared - 'int' assumed

The declaration of a function parameter is missing. Example:

```
int func(a)
/*a should be declared here or within the parentheses*/
{
    ...
}
```

Format requires *nn* parameters, but *mm* given

Mismatch between a `printf` or `scanf` format string and its other arguments. Example:

```
printf("%d, %d\n",1); /* should be two ints */
```

function *xx* declared but not used

When compiling with **Feature** v, the function *xx* was declared but not used within the source file.

Illegal format conversion '%x'

Indicates an illegal conversion implied by a `printf` or `scanf` format string.

Example:

```
printf("%w\n",10); /* no such thing as %w */
```

implicit narrowing cast : xx

An arithmetic operation or bit manipulation is attempted involving assignment from one data type to another, where the size of the latter is naturally smaller than that of the assigned value. Example:

```
double d = 1.0; long l = 2L; int i = 3;
i = d * i;
i = 1 | 3;
i = 1 & ~1;
```

implicit return in non-void function

A non-void function may exit without using a return statement, but won't return a meaningful result. Example:

```
int func(int a)
{
    int b=a*10;
    .../* no return <expr> statement */
}
```

incomplete format string

A mistake in a `printf` or `scanf` format string. Example:

```
printf("Score was %d%",score); /* 2nd % should be %% */
```

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

If the definition of a function omits its return type – it defaults to `int`. You should be explicit about the type, using `void` if the function doesn't return a result.

Example:

```
main()
{
    ...
}
```

inventing 'extern int xx();'

The declaration of a function is missing. Example:

```
printf("Type your name: ");  
/* forgot to #include <stdio.h> */
```

lower precision in wider context: xx

An arithmetic operation or bit manipulation is attempted involving assignment from int, short or char to long. Example:

```
long l = 1L; int i = 2; short j = 3;  
l = i & j;  
l = i | 5;  
l = i * j;
```

One circumstance in which this causes problems is when code like

```
long f(int x){return 1<<x;}
```

is moved to machines where int has 16 bits.

No side effect in void context: 'op'

An expression which does not yield any side effect was evaluated; it will have no effect at run-time. Example:

```
a+b;
```

no type checking of enum in this compiler

Compiling with **Feature x**, an enum declaration was found, and this message refers to the ISO stipulation that enum values be integers, less strictly typed than in some earlier dialects of C.

Non-ANSI #include <xx>

A header file has been #included which is not defined in the ANSI/ISO standard.

< > should be replaced by " ".

non-portable - not 1 char in 'xx'

Assigning character constants containing more than one character to an int will produce non-portable results. Example:

```
static int exitCode = 'ABEX';
```

non-value return in a non-void function

The expression was omitted from a `return` statement in a function which was defined with a non-void return type. Example:

```
int func(int a)
{
    int b=a*10;
    ...
    return; /* no <expr> */
}
```

odd unsigned comparison with 0 : `xx`

An attempt has been made to determine whether an unsigned variable is negative. Example:

```
unsigned u , v;
if (u < 0) u = u * v;
if (u >= 0) u = u / v;
```

Old-style function: `xx`

Compiling with **Feature o**, it was noted that the code contains a non-ISO function declaration. Example:

```
void fn2(a , b)
int a;
int b;
{ b = a; }
```

omitting trailing `'\0'` for `char[nn]`

The character array being equated to a string is one character too short for the whole string, so the trailing zero is being omitted. Example:

```
static char msg[14] = "(C)1988 Acorn\n"; /* needs 15 */
```

repeated definition of `#define` macro `xx`

When compiling with **Feature h**, a macro has been repeatedly `#defined` to take the same value.

shift by nn illegal in ANSI C

This is given for negative constant shifts or shifts greater than 31. On the Arm, the bottom byte of the number given is used, ie it is treated as (unsigned char) nn. NB: negative shifts are not treated as positive shifts in the other direction. Example:

```
printf("%d\n", 1<<-2);
```

'short' slower than 'int' on this machine (see manual)

For speed you are advised to use ints rather than shorts where possible. This is because of the overhead of performing implicit casts from short to int in expression evaluation. However, shorts are half the size of ints, so arrays of shorts can be useful. Example:

```
{
    short i,j; /* quicker to use ints */
    ...
}
```

spurious {} around scalar initialiser

Braces are only required around structure and array initialises. Example:

```
static int i = {INIT_I}; /* don't need braces */
```

static xx declared but not used

A static variable was declared in a file but never used in it. It is therefore redundant.

Unrecognised #pragma (no '-' or unknown word)

#pragma directives are of the form

```
#pragma -xd
or
#pragma long_spelling
```

where *x* is a letter and *d* is an optional digit. These messages warn against unknown letters and missing minus signs.

use of 'op' in condition context

Warns of such possible errors as = and not == in an if or looping statement. Example:

```
if (a=b) {
    ...
}
```

variable `xx` declared but not used

This refers to an automatic variable which was declared at the start of a block but never used within that block. It is therefore redundant. Example:

```
int func(int p)
{
    int a; /* this is never used */
    return p*100;
}
```

`xx` may be used before being set

Compiling with **Feature a**, an automatic variable is found to have been used before any value has been assigned to it.

`xx` treated as `xxul` in 32-bit implementation

This message warns of two's complement arithmetic's dependence on assigning negative constants to unsigned ints, and it explains that ints and long ints are both 32 bits.

Non-serious errors

These are errors which will allow 'working' code to be produced – they will not produce loss of code. On detection of such an error the compiler issues an error message, if enabled, then continues compilation.

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

Incorrect punctuation between function parameters. Example:

```
extern int func(int a;int b);
```

ANSI C does not support `'long float'`

This used to be a synonym for `double`, but is not allowed in ANSI/ISO C.

ancient form of initialisation, use `'='`

An obsolete syntax for initialisation was used, or incorrectly nested brackets have been found. Example:

```
int i{1};          /* use int i=1; */
```


array [0] found

The minimum subscript count allowed is 1. (Remember that the subscripts go from 0 - n-1.) Example:

```
static int a[0];
```

array of xx illegal - assuming pointer

Illegal objects have been declared to occupy an array. Examples:

```
int fn2[5]();           /* array of functions */
void v[10];            /* array of voids */
```

assignment to 'const' object 'xx'

You can't assign to objects declared as const. Example:

```
{
    const int ic = 42; /* initialisation ok */
    ic = 69; /* can't change it now */
    ...
}
```

comparison 'op' of pointer and int:

literal 0 (for == and !=) is the only legal case

You cannot use the comparison operators between an integer and a pointer type. As the message implies, you can only check for a pointer being (not) equal to NULL (int 0). Example:

```
{
    int i,j,*ip;
    j = i>ip; /* can't compare an int and an int * */
    ...
}
```

declaration with no effect

The compiler detected what appeared to be a declaration statement, but which resulted in no store being allocated. This may imply that a data type name was omitted.

differing pointer types: 'xx'

An illegal implicit type cast was detected in a comparison operation between two pointers of different types. Example:

```
{
    int      *ip;
    char     *cp;
    printf("%d\n", ip==cp); /* can't compare these */
    ...
}
```

differing redefinition of #define macro xx

#define gives a definition contradicting that already assigned to the named macro.

ellipsis (...) cannot be only parameter

Although C allows variable length argument lists, the '...' parameter cannot stand alone in this function declaration. Example:

```
void fnl(...) { }
```

expected 'xx' or 'x' - inserted 'x' before 'yy'

Often caused by omitting a terminating symbol in a statement when the compiler is able to insert this symbol for you, and then to recover. Example:

```
int f(int j)
{
    return j;
}
int main()
{
    int i=f(10;      /* ')' omitted here */
    return i;
}
```

formal name missing in function definition

This error occurs when a comma in a function definition led the compiler to suspect a further formal parameter was going to follow, but none did. Example:

```
int a(int b,) /* missing parameter */
{
    ...
}
```

function prototype formal 'xx' needs type or class - 'int' assumed

A formal parameter in a function prototype was not given a type or class. It needs at least one of these (register being the only allowed class). Example:

```
void func(a); /* I mean int a or perhaps register a */
```

function returning xx illegal - assuming pointer

A function apparently intends to return an illegal object. Example:

```
int fn3() []           /* hoping to return an array */
{
    int list[3] = {1,2,3};
    return list;
}
```

function xx may not be initialised - assuming function pointer

A function is not a variable, so cannot be initialised. As an attempt to initialise xx has been made, xx is treated as of type function *. Example:

```
extern int func(void);
static int fn() = func; /* the compiler will use
    static int (*fn)() = func; instead */
```

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

Warns of an illegal implicit cast within an expression. Typically *op* is an operator which has no business being used on pointers anyway, such as *|* or dyadic ***. Example:

```
{
    int i, *ip;
    i = i | ip; /* bitwise-or on a pointer?! */
    ...
}
```

junk at end of #xx line - ignored

The xx is either *else* or *endif*. These directives should not have anything following them on the line. Example:

```
/* text after the #else should be a comment */
#else if it isn't defined
...
```

L'...' needs exactly 1 wide character

The `wchar_t` declaration of a wide character names an identifier comprising other than one character. Example:

```
wchar_t wc = L'abc';
```

linkage disagreement for 'xx' - inconsistent '__global_reg' usage

You have attempted to declare a variable with both `__global_reg` and either `static` or `extern` linkage.

linkage disagreement for 'xx' - treated as 'xx'

There was a linkage type disagreement for declarations, eg a function was declared as `extern` then defined later in the file as `static`. Example:

```
int func(int a); /* compiler assumes extern here */
...
static func(int a) /* but told static here */
{
    ...
}
```

more than 4 chars in character constant

A character constant of more than four characters cannot be assigned to a 32 bit `int`. Example:

```
{
    int i = '12345'; /* more than four chars */
    ...
}
```

no chars in character constant ''

At least one character should appear in a character constant. The empty constant is taken as zero. Example:

```
{
    int i = ''; /* less than one char == '\0' */
    ...
}
```

objects that have been cast are not l-values

The programmer tried to use a cast expression as an l-value. Example:

```
char *p;
*((int *)p)=10; /* (int *)p is NOT an l-value */
```

omitted <type> before formal declarator - 'int' assumed

This is given in a formal parameter declaration where a type modifier is given but no base type. Example:

```
int func(*a); /* a is a pointer, but to what? */
```

'op': cast between function pointer and non-function object

Casts between function and object pointers can be very dangerous! One possibly valid (but still very suspect) use is in casting an array of `int` into which machine code has been loaded into a function pointer. Example:

```
static int mcArray[100];
/*pointer to function returning void*/
typedef void (*pfv)(void);
...
((pfv)mcArray)(); /* convert to fn type and apply */
```

'op': implicit cast of non-0 int to pointer

Zero, equal to a `NULL` pointer, is the only `int` which can be legally implicitly cast to a pointer type. Example:

```
{
    int i, *ip;
    ip = i;          /* only the constant int 0 can be implicitly
                       cast to a pointer type */
    ...
}
```

'op': implicit cast of pointer to non-equal pointer

An illegal implicit cast has been detected between two different pointer types. The type casting must be made explicit to escape this error. Example:

```
{
    int      *ip;
    char     *cp;
    ip = cp; /* differing pointer types */
    ...
}
```

'op': implicit cast of pointer to 'int'

An illegal implicit cast has been detected between an integer and a pointer. Such casts must be made explicitly. Example:

```
{
    int i, *ip;
    i = ip; /* pointer must be cast explicitly */
    ...
}
```

overlarge escape '\\xxxx' treated as '\\xxx'

A hexadecimal escape sequence is too large. Example:

```
int novalue()
{
    if (seize) return '\\xffff';          /* \xffff' too large */
    else return '\\xff';
}
```

overlarge escape '\\x' treated as '\\x'

An octal escape sequence is too large. Example:

```
int novalue()
{
    if (huit) return '\\777';             /* \777 too large */
    else return '\\77';
}
```

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

The only legal operators allowed in this context are + and -.

prototype and old-style parameters mixed

Use has been made of both the ISO style function/definition (including a type name for formal parameters in a function's heading) and pcc style parameters lists.

Example:

```
void fn4(a, int b)
int a;
{
    a = b;
}
```

references to non-const static data are not supported when using -zM with APCS /noswst

Code generated with the -zM switch uses the stack limit register to locate its static data, but you have specified a calling standard that does not use a stack limit register.

'register' attribute for 'xx' ignored when address taken

Addresses of register variables cannot be calculated, so an address being taken of a variable with a `register` storage class causes that attribute to be dropped.

Example:

```
{
    register int i, *ip;
    ip = &i; /* & forces i to lose its register attribute */
    ...
}
```

return <expr> illegal for void function

A function declared as `void` must not return with an expression. Example:

```
void a(void)
{
    ...
    return 0;
}
```

size of 'void' required - treated as 1

This indicates an attempt to do pointer arithmetic on a `void *`, probably indicating an error. Example:

```
{
    void      *vp;
    vp++; /* how many bytes to increment by ? */
    ...
}
```

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

If an array is declared as having an empty first subscript size, the compiler cannot calculate the array's size. It therefore assumes the first subscript limit to be 1 if necessary. This is unlikely to be helpful.

```
extern int array[][10];
static int s = sizeof(array); /*can't determine this*/
```

sizeof <bit field> illegal - sizeof(int) assumed

Bitfields do not necessarily occupy an integral number of bytes but they are always parts of an int, so an attempt to take the size of a bitfield will return sizeof(int). Example:

```
struct s {
    int exp : 8;
    int mant : 23;
    int s : 1;
};
int main(void)
{
    struct s st;
    int i = sizeof(st.exp); /* can't obtain this in bytes */
    ...
}
```

**Small (single precision) floating value converted to 0.0
Small floating point value converted to 0.0**

A floating point constant was so small that it had to be converted to 0.0. Example:

```
static float f = 1.0001e-38 - 1.0e-38; /* 1e-42 too small for float */
```

**Spurious #elif ignored
Spurious #else ignored
Spurious #endif ignored**

One of these three directives was encountered outside any #if or #ifdef scope. Example:

```
#if defined sym
...
#endif
#else /* this one is spurious */
...
#endif
```

static function xxx not defined - treated as extern

A prototype declares the function to be static, but the function itself is absent from this compilation unit.

string initialiser longer than char [nn]

An attempt was made to initialise a character array with a string longer than the array. Example:

```
static char str[10] = "1234567891234";
```


struct component `xx` may not be function - assuming function pointer

A variable such as a structure component cannot be declared to have type function, only function *. Example:

```
struct s {
    int fn(); /* compiler will use int (*fn)(); */
    char c;
};
```

type or class needed (except in function definition) - int assumed

You can't declare a function or variable with neither a return type nor a storage class. One of these must be present. Examples:

```
func(void); /* need, eg, int or static */
x;
```

Undeclared name, inventing 'extern int `xx`'

The name `xx` was undeclared, so the default type `extern int` was used. This may produce later spurious errors, but compilation continues. Example:

```
int main(void) {
    int i = j; /*j has not been previously declared*/
    ...
}
```

unprintable character `xx` found - ignored

An unrecognised character was found embedded in your source – this could be file corruption, so back up your sources! Note that 'unprintable character' means any non-whitespace, non-printable character.

variable `xx` may not be function - assuming function pointer

A variable cannot be declared to have type function, only function *. Example:

```
int main(void)
{
    auto void fn(void); /* treated as void (*fn)(void); */
    ...
}
```

xx may not have whitespace in it

Tokens such as the compound assignment operators (+= etc) may not have embedded whitespace characters in them. Example:

```
{
    int i;
    ...
    i + = 4; /* space not allowed between + and = */
    ...
}
```

Serious errors

These are errors which will cause loss of generated code. On detection of such an error, the compiler will attempt to continue and produce further diagnostic messages, which are sometimes useful, but will delete the partly produced object file.

'...' must have exactly 3 dots

This is caused by a mistake in a function prototype where a variable number of arguments is specified. Example:

```
extern int printf(const char *format,...); /*one . too many*/
```

'{' of function body expected - found 'xx'

This is produced when the first character after the formal parameter declarations of a function is not the { of the function body. Example:

```
int func(a)
int a;
    if (a) ... /* omitted the { */
```

'{' or <identifier> expected after 'xx', but found 'yy'

xx is typically struct or union, which must be followed either by the tag identifier or the open brace of the field list. Example:

```
struct *fred; /* Missed out the tag id */
```

'xx' variables may not be initialised

A variable is of an inappropriate class for initialisation. Example:

```
int main()
{
    extern int n=1;
    return 1;
}
```

'op': cast to non-equal 'xx' illegal**'op': illegal cast of 'xx' to pointer****'op': illegal cast to 'xx'**

These errors report various illegal casting operations. Examples:

```
struct s {
    int a,b;
};
struct t {
    float ab;
};
int main(void)
{
    int i;
    struct s s1;
    struct t s2;
    /* '=': illegal cast to 'int' */
    i = s1;
    /* '=': illegal cast to non-equal 'struct' */
    s1 = s2;
    /* <cast>: illegal cast of 'struct' to pointer */
    i = *(int *) s1;
    /* <cast>: illegal cast to 'int' */
    i = (int) s2;
    ...
}
```

'op': illegal use in pointer initialiser

(Static) pointer initialisers must evaluate to a pointer or a pointer constant plus or minus an integer constant. This error is often accompanied by others. Example:

```
extern int count;
static int *ip = &count*2;
```

{ } must have 1 element to initialise scalar

When a scalar (integer or floating type) is initialised, the expression does not have to be enclosed in braces, but if they are present, only one expression may be put between them. Example:

```
static int i = {1,2}; /* which one to use? */
```

Array size *nn* illegal - 1 assumed

Arrays have a maximum dimension of 0xffffffff. Example:

```
static char dict[0x10000000]; /* Too big */
```

attempt to apply a non-function

The function call operator () was used after an expression which did not yield a pointer to function type. Example:

```
{
    int i;
    i();
    ...
}
```

Bit fields do not have addresses

Bitfields do not necessarily lie on addressable byte boundaries, so the & operator cannot be used with them. Example:

```
struct s {
    int h1,h2 : 13;
};
int main(void)
{
    struct s s1;
    short *sp = &s1.h2; /* can't take & of bit field */
    ...
}
```

Bit size *nn* illegal - 1 assumed

Bitfields have a maximum permitted width of 32 bits as they must fit in a single integer. Example:

```
struct s {
    int f1 : 40; /* This one is too big */
    int f2 : 8;
};
```

'break' not in loop or switch - ignored

A break statement was found which was not inside a for, while or do loop or switch. This might be caused by an extra }, closing the statement prematurely. Example:

```
int main(int argc)
{
    if (argc == 1)
        break;
    ...
}
```

'case' not in switch - ignored

A case label was found which was not inside a switch statement. This might be caused by an extra }, closing the switch statement prematurely. Example:

```
void fn(void)
{
    case '*': return;
    ...
}
```

<command> expected but found a 'op'

This error occurs when a (binary) operator is found where a statement or side-effect expression would be expected. Example:

```
if (a) /10; /* mis-placed ) perhaps? */
...
```

'continue' not in loop - ignored

A continue statement was found which was not inside a for, while or do loop. This might be caused by an extra }, closing the loop statement prematurely. Example:

```
while (cc) {
    if (dd)                /* intended a { here */
        error();
    }                      /*this closes the while */
    if (ee)
        continue;
}
```

'default' not in switch - ignored

A default label was found which was not inside a switch statement. This might be caused by an extra }, closing the switch statement prematurely. Example:

```
switch (n) {
    case 0:
        return fn(n);
    case 1: if (cc)
        return -1;
    else
        break;
} /* spurious } closes the switch */
default:
    error();
}
```

duplicate case constant: *nn*

The case label whose value is *nn* was found more than once in a switch statement. Note that *nn* is printed as a decimal integer regardless of the form the expression took in the source. Example:

```
switch (n) {
    case ' ':
    ...
    case ' ':
    ...
}
```

duplicate 'default' case ignored

Two cases in a single switch statement were labelled default. Example:

```
switch (n) {
    default:
    ...
    default:
    ...
}
```

duplicate definition of 'struct' tag '*xx*'

There are duplicate definitions of the type struct *xx* {...} ;. Example:

```
struct s { int i,j;};
struct s {float a,b;};
```

duplicate definition of 'union' tag '*xx*'

There are duplicate definitions of the type union *xx* {...} ;. Example:

```
union u {int i; char c[4];};
union u {double d; char c[8];};
```

duplicate type specification of formal parameter '*xx*'

A formal function parameter had its type declared twice, once in the argument list and once after it. Example:

```
void fn(int i)
int i;          /* this one is redundant */
{
    ...
}
```

EOF in comment
EOF in string
EOF in string escape

These all refer to unexpected occurrences of the end of the source file.

Expected <identifier> after 'xx' but found 'xx'
expected 'xx' - inserted before 'yy'

This typically occurs when a terminating semi-colon has been omitted before a }. (Common amongst Pascal programmers) Another case is the omission of a closing bracket of a parenthesised expression. Examples:

```
int fn(int a, int b, int c)
{
    int d = a*(b+c;           /* missing ) */
    return d                  /* missing ; */
}
```

Expecting <declarator> or <type>, but found 'xx'

xx is typically a punctuation character found where a variable or function declaration or definition would be expected (at the top level). Example:

```
static int i = MAX;+1;          /* spurious ; ends expression */
```

<expression> expected but found 'op'

Similar to above. An operator was found where an operand might reasonably be expected. Example:

```
func(>>10); /* missing left hand side of >> */
```

grossly over-long floating point number

Only a certain number of decimal digits are needed to specify a floating point number to the accuracy that it can be stored to. This number of digits was exceeded by an unreasonable amount.

grossly over-long number

A constant has an excessive number of leading zeros, not affecting its value.

hex digit needed after 0x or 0X

Hexadecimal constants must have at least one digit from the set 0-9, a-f, A-F following the 0x. Example:

```
int i = 0xg; /* illegal hex char */
```

<identifier> expected but found 'xx' in 'enum' definition

An unexpected token was found in the list of identifiers within the braces of an enum definition. Example:

```
enum colour {red, green, blue,;}; /* spurious ; */
```

identifier (xx) found in <abstract declarator> - ignored

The `sizeof()` function and cast expressions require abstract declarators, ie types without an identifier name. This error is given when an identifier is found in such a situation. Examples:

```
i = (int j) ip; /* trying to cast to integer */
l = sizeof(char str[10]); /* probably just mean sizeof(str) */
```

illegal bit field type 'xx' - 'int' assumed

Int (signed or unsigned) is the only valid bitfield type in ISO-conforming implementations. Example:

```
struct s { char a : 4; char b : 4;};
```

illegal in case expression (ignored): xx

illegal in constant expression: xx

illegal in floating type initialiser: xx

All of these errors occur when a constant is needed at compile time but a variable expression was found.

illegal in l-value: 'enum' constant 'xx'

An incorrect attempt was made to assign to an enum constant. This could be caused by misspelling an enum or variable identifier. Example:

```
enum col {red, green, blue};
int fn()
{
    int read;
    red = 10;
    ...
}
```


illegal in the context of an l-value: 'xx'**illegal in lvalue: function or array 'xx'**

An incorrect attempt was made to assign to `xx`, where the object in question is not assignable (an l-value). You can't, for example, assign to an array name or a function name. Examples:

```
{
    int a,b,c;
    a ? b : c = 10;          /* ?: can't yield l-values. */
    if (a)                   /* use this instead */
        b = 10;
    else
        c = 10;
    ...
}
```

or, in the same context,

```
*(a ? &b: &c) = 10;
```

illegal in static integral type initialiser: xx

A constant was needed at compile time but a suitable expression wasn't found.

illegal types for operands : 'op'

An operation was attempted using operands which are unsuitable for the operator in question. Examples:

```
{
    struct {int a,b;} s;
    int i;
    i = *s;          /* can't indirect through a struct */
    s = s+s;         /* can't add structs */
    ...
}
```

incomplete type at tentative declaration of 'xx'

An incomplete non-static tentative definition has not been completed by the end of the compilation unit. Example:

```
int incomplete[];
...
/* should be completed with a declaration like: */
/* int incomplete[SOMESIZE];                      */
```

junk after #if <expression>
junk after #include "xx"
junk after #include <xx>

None of these directives should have any other non-whitespace characters following the expression/filename. Example:

```
#include <stdio.h> this isn't allowed
```

label 'xx' has not been set

An attempt has been made to use a label that has not been declared in the current scope, after having been referenced in a goto statement. Example:

```
int main(void)
{
    goto end;
}
```

misplaced '{' at top level - ignoring block

{ } blocks can only occur within function definitions. Example:

```
/* need a function name here */
{
    int i;
    ...
}
```

misplaced 'else' ignored

An else with no matching if was found. Example:

```
if (cc)          /* should have used { } */
    i = 1;
    j = 2;
else
    k = 3;
...
```

misplaced preprocessor character 'xx'

Usually a typing error; one of the characters used by the preprocessor was detected out of context. Example:

```
char #str[] = "string";          /* should be char *str[] */
```

missing #endif at EOF

A #if or #ifdef was still active at end of the source file. These directives must always be matched with a #endif.

missing '"' in pre-processor command line

A line such as `#include "name` has the second `"` missing.

missing ')' after `xx(...` on line `nn`

The closing bracket (or comma separating the arguments) of a macro call was omitted. Example:

```
#define rdch(p) {ch=*p++;}
...
{
    rdch(p
    /* missing ) */
    ...
}
```

missing ',', ' or ')' after `#define xx(...`

One of the above characters was omitted after an identifier in the macro parameter list. Example:

```
#define rdch(p {ch = *p++;}
```

missing '<' or '"' after `#include`

A `#include` filename should be within either double quotes or angled brackets.

missing hex digit(s) after `\x`

The string escape `\x` is intended to be used to insert characters in a string using their hexadecimal values, but was incorrectly used here. It should be followed by between one and three hexadecimal digits. Example:

```
printf("\xxx/"); /* probably meant "\\xxx/" */
```

missing identifier after `#define`**missing identifier after `#ifdef`****missing identifier after `#undef`**

Each of these directives should be followed by a valid C identifier. Example:

```
#define @ at
```

missing parameter name in `#define xx(...`

No identifier was found after a `,` in a macro parameter list. Example:

```
#define rdch(p,) {ch=*p++;}
```

no ')' after #if defined (...

The defined operator expects an identifier, optionally enclosed within brackets. Example:

```
#if defined(debug
```

no identifier after #if defined

See above.

non static address 'xx' in pointer initialiser

An attempt was made to take the address of an automatic variable in an expression used to initialise a static pointer. Such addresses are not known at compile-time. Example:

```
{
    int i;
    static int *ip = &i; /*&i not known to compiler*/
    ...
}
```

non-formal 'xx' in parameter-type-specifier

A parameter name used to declare the parameter types did not actually occur in the parameter list of the function. Example:

```
void fn(a)
int a,b;
{
    ...
}
```

number nn too large for 32-bit implementation

An integer constant was found which was too large to fit in a 32 bit int. Example:

```
static int mask = 0x800000000; /*0x800000000 intended?*/
```

objects or arrays of type void are illegal

void is not a valid data type.

overlarge floating point value found**overlarge (single precision) floating point value found**

A floating point constant has been found which is so large that it will not fit in a floating point variable. Examples:

```
float f = 1e40; /* largest is approx 1e38 for float */
double d = 1e310; /* and 1e308 for double */
```

quote (" or ') inserted before newline

Strings and character constants are not allowed to contain unescaped newline characters. Use `<nl>` to allow strings to span lines. Example:

```
printf("Total =
```

re-using 'struct' tag 'xx' as 'union' tag

There are conflicting definitions of the type `struct xx {...}` ; and `union xx {...}` ;. Structure and union tags currently share the same name-space in C.

Example:

```
struct s {int a,b;};
...
union s {int a; double d;};
```

re-using 'union' tag 'xx' as 'struct' tag

As above.

size of struct 'xx' needed but not yet defined

An operation requires knowledge of the size of the struct, but this was not defined. This error is likely to accompany others. Example:

```
{
    struct s;                /* forward declaration */
    struct s *sp;            /* pointer to s */
    sp++;                    /* need size for inc operation */
    ...
}
```

size of union 'xx' needed but not yet defined

See above.

storage class 'xx' incompatible with 'xx' - ignored

An attempt was made to declare a variable with conflicting storage classes.

Example:

```
{
    static auto int i; /* contradiction in terms */
    ...
}
```

storage class 'xx' not permitted in context xx - ignored

An attempt was made to declare a variable whose storage class conflicted with its position in the program. Examples:

```
register int i;          /* can't have top-level regs */
void fn(a)
static int a;           /* or static parameters */
{
    ...
}
```

struct 'xx' must be defined for (static) variable declaration

Before you can declare a static structure variable, that structure type must have been defined. This is so the compiler knows how much storage to reserve for it.

Examples:

```
static struct s s1;      /* s not defined */
struct t;
static struct t t1;      /* t not defined */
```

struct/union 'xx' not yet defined - cannot be selected from

The structure or union type used as the left operand of a . or → operator has not yet been defined so the field names are not known. Example:

```
{
    struct s s1;          /* forward reference      */
    s1.a = 12;            /* don't know field names yet */
    ...
}
```

**too few arguments to macro `xx`(... on line `nn`
too many arguments to macro `xx`(... on line `nn`**

The number of arguments used in the invocation of a macro must match exactly the number used when it was defined. Example:

```
#define rdch(ch,p) while((ch = *p++)==' ');
...
    rdch(ptr);/* need ptr and ch */
...
```

too many initialisers in {} for aggregate

The list of constants in a static array or structure initialiser exceeded the number of elements/fields for the type involved. Example:

```
static int powers[8] = {0,1,2,4,8,16,32,64,128};
```

**type '`xx`' inconsistent with '`xx`'
type disagreement for '`xx`'**

Conflicting types were encountered in function declaration (prototype) and its definition. Example:

```
void fn(int);
...
int fn(int a)
{
    ...
}
```

A pernicious error of this type is caused by mixing ISO and old-style function declarations. Example:

```
int f(char x);
int f(x)char x;
{
    ...
}
```

typedef name '`xx`' used in expression context

A typedef name was used as a variable name. Example:

```
typedef char flag;
...
{
    int i = flag;
```

undefined struct/union 'xx' cannot be member

A struct/union not already defined cannot be a member of another struct/union. In particular this means that a struct/union cannot be a member of itself: use pointers for this. Example:

```
struct s1 {
    struct s2 type; /* s2 not defined yet */
    int count;
};
```

unknown preprocessor directive : #xx

The identifier following a # did not correspond to any of the recognised pre-processor directives. Example:

```
#asm                /* not an ISO directive */
```

uninitialised static [] arrays illegal

Static [] arrays must be initialised to allow the compiler to determine their size. Example:

```
static char str[];                /* needs {} initialiser */
```

union 'xx' must be defined for (static) variable declaration

Before you can declare a static union variable, that union type must have been defined. Example:

```
static union u u1; /* compiler can't ascertain size */
```

'while' expected after 'do' - found 'xx'

The syntax of the do statement is do statement while (expression). Example:

```
do                /* should put these statements in {} */
    l = inputLine();
    err = processLine(l); /* finds err, not while */
while (!err);
```


Fatal errors

These are causes for the compiler to give up compilation. Error messages are issued and the compiler stops.

compiler too old: this is version x.xx, code requires version x.xx or later

The command line option `-ccversion version` on page 48 was used to specify that a later version of the compiler is required to compile the code.

couldn't create object file 'file'

The compiler was unable to open or write to the specified output code file, perhaps because it was locked or the `o` directory does not exist.

macro args too long

Grossly over-long macro arguments, possibly as a result of some other error.

macro expansion buffer overflow

Grossly over-complicated macros were used, possibly as a result of some other error.

out of store (in cc_alloc)

The compiler has run out of memory – either shorten your source programs, or free some RAM by, for example, quitting some other applications.

If running under the desktop, you can use the Task Manager to increase your `wimpslot` size.

too many errors

More than 100 serious errors were detected.

System errors

There are some additional error messages that can be generated by the compiler if it detects errors in the compiler itself. It is very unusual to encounter this type of error. If you do, note the circumstances under which the error was caused and contact RISC OS Open.

These error messages all look like this:

```
Internal inconsistency: either resource shortage or compiler fault. If you
cannot alter your program to avoid this failure, please contact your supplier.
```

Appendix C: C++ errors and warnings

This appendix contains the text and explanation for all ‘not implemented’ messages produced by the C++ Language System Release 3.0. They are listed here in alphabetical order.

Each message is preceded by a file name, a line number, and the text ‘not implemented’. A complete error has this syntax:

```
"file", line n: not implemented: message
```

where the *message* is as used in the headings below. The line number is usually the line on which a problem has been diagnosed.

A ‘not implemented’ message is issued when Release 3.0 encounters a **legal** construct for which it cannot generate code. Because code is not generated, ‘not implemented’ messages cause the CC command to fail, and the program is not linked. Release 3.0 does, however, attempt to examine the rest of your program for other errors.

‘Not implemented’ messages

actual parameter expression of type string literal

A template is instantiated with a string literal actual argument:

```
template <char* s> struct S { /*...*/};
```

```
S<"hello world"> svar;
```

```
"file", line 3: not implemented: actual parameter expression of type string  
literal
```

address of bound member as actual template argument

A template is instantiated with the address of a class member bound to an actual class object:

```
template <int *pi> class x {};  
class y { public: int i; } b;
```

```
x< &b.i > xi;
```

```
"file", line 4: not implemented: address of bound member (& ::b . y::i) as  
actual template argument
```

& of op

This message should not be produced.

1st operand of .* too complicated

The first operand of a function call expression involves a pointer to a member function and is an expression that may have side effects or may require a temporary.

```
struct S { virtual int f(); };
int (S::*pmf)() = &S::f;
S *f();
int i = (f()->*pmf)();

"file", line 5: not implemented: 1st operand of .* too complicated
```

2nd operand of .* too complicated

The second operand of a pointer to member operator is an expression that has side effects.

```
struct S { int f(); };
int (S::*pmf)() = &S::f;
S *sp = new S;
int i = 5;
int j = (sp->*(i+=5, pmf))();

"file", line 5: not implemented: 2nd operand of .* too complicated
```

call of virtual function *function* before class has been completely declared

```
class x {
public:
    virtual x& f();
    int foo(x t = pt->f());
private:
    static x* pt;
    int i;
};

"file", line 6: not implemented: call of virtual function x::f() before class
x has been completely declared - try moving call from argument list into
function body or make function non-virtual
```

cannot expand inline function *function* with for statement in inline

A for statement appears in the definition of an inline function.

```
struct S {
    int s[100];
    S() { for (int i = 0; i < 100; i++) s[i] = i; }
};

"file", line 1: not implemented: cannot expand inline function S::S() with for
statement in inline
```

cannot expand inline function *function* with statement after "return"

A value-returning inline function contains a statement following a return statement.

```
inline int f(int i) {
    if (i) return i;
    return 0;
}
```

"file", line 4: not implemented: cannot expand inline function f() with statement after "return"

cannot expand inline function *function* with two local variables with the same name (*name*)

Two variables with the same name and different types are declared within the body of a value-returning inline function.

```
inline int f(int i) {
    { int x = i; }
    { double x = i; }
    return 0;
}
```

"file", line 5: not implemented: cannot expand inline function f() with two local variables with the same name (x)

cannot expand inline function needing temporary variable of array type

An inline function that contains a local declaration of an array object is called.

```
inline int f(int i) {
    int a[1];
    a[0] = i;
    return i;
}
int v = f(0);
```

"file", line 6: not implemented: cannot expand inline function needing temporary variable of array type

cannot expand inline function with return in if statement

This message should not be produced.

cannot expand inline function with static *name*

An inline function contains the declaration of a static object.

```
inline void f() {
    static int i = 5;
}
```

"file", line 2: not implemented: cannot expand inline function with static i

cast of non-integer constant

A cast of a non-integer constant as an actual parameter to a template class.

```
template <int i> class x;  
int yy;
```

```
x< (int)&yy > xi;
```

"file", line 4: not implemented: cast of non-integer constant

cannot expand inline void *function* called in comma expression

A call of an inline void function that cannot be translated into an expression (that is, one that includes a loop, a goto, or a switch statement) appears as the first operand of a comma operator.

```
int i;  
inline void f() { for (;;) ; }  
void g() { for (f(), i = 0; i < 10; i++) ; }
```

"file", line 3: not implemented: cannot expand inline void f() called in comma expression

cannot expand inline void *function* called in for expression

A call of an inline void function that cannot be translated into an expression (that is, one that includes a loop, a goto, or a switch statement) appears in the second expression of a for statement.

```
void inline f() { for (;;) ; } void g() { for (;;) f() ; }
```

"file", line 2: not implemented: cannot expand inline void f() called in for expression

cannot expand value-returning inline *function* with call of ...

A value-returning inline function is defined, and it contains a call to another inline function that is not value-returning.

```
inline void f() { for(;;) ; }  
inline int g() { f(); return 0; }
```

"file", line 2: not implemented: cannot expand value-returning inline g() with call of non-value-returning inline f()

cannot merge lists of conversion functions

A derived class with multiple bases is declared and there are conversion operators declared in more than one of the base classes.

```
struct B1 {
    operator int();
};
struct B2 {
    operator float();
};
struct D : public B1, public B2 { };

"file", line 7: not implemented: cannot merge lists of conversion functions
```

catch

The keyword `catch` appears; `catch` is reserved for future use.

```
int catch;

"file", line 1: not implemented: catch
"file", line 1: warning: name expected in declaration list
```

class defined within sizeof

A class or union definition appears as the type name in a `sizeof` expression.

```
int i = sizeof (struct S { int i; });

"file", line 1: not implemented: class defined within sizeof
"file", line 1: error: S undefined, size not known
```

class hierarchy too complicated

This message should not be produced.

conditional expression with *type*

The second and third operands of a conditional expression are member functions or pointers to members.

```
struct S { int i, j; };
void f(int i) {
    int S::*pmi = i ? &S::i : &S::j;
}

"file", line 3: not implemented: conditional expression with int S::*
```

constructor needed for argument initializer

The default value for an argument is a constructor or is an expression that invokes a constructor.

```
struct S { S(int); };
int f(S = S(1));
int g(S = 5);
```

"file", line 2: not implemented: constructor as default argument

"file", line 3: not implemented: constructor needed for argument initializer

copy of *member[]*, no memberwise copy for *class*

An implementation-generated copy operation for a class *X* is required, but the operation cannot be generated because *X* has an array member whose type is a class with either a virtual base class or its own defined copy operation. The workaround is to add a memberwise copy operator to *X*.

```
struct S1 {};
struct S2 : S1 { S2& operator=(const S2&); };
struct X { S2 m[1]; };
X var1;
X var2 = var1;
```

"file", line 5: not implemented: copy of S2[], no memberwise copy for S2

default argument too complicated

A default argument in a declaration not at file scope requires the generation of a temporary.

```
struct S {
    S();
    int f(const int &r = 1);
};
```

"file", line 3: not implemented: default argument too complicated

"file", line 3: not implemented: needs temporary variable to evaluate argument initializer

ellipsis (...) in argument list of template function *name*

An ellipsis is used in a template function declaration:

```
template <class T> f(T, ...);
```

"file", line 1: not implemented: ellipsis (...) in argument list of template function f()

explicit template parameter list for destructor of specialized template class *name*

Explicit template parameters are included in declaration of a specialised class' destructor:

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

struct S<int> {
    ~S<int>();
};

"file", line 4: not implemented: explicit template parameter list for
destructor of specialized template class S <> -- please drop the parameter
list
```

Instead, declare the specialised destructor as follows:

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

struct S<int> {
    ~S();
};
```

formal type parameter *name* used as base class of template

The formal type parameter is used as the base class of a template class:

```
template <class T> struct S : public T { /*...*/ };

"file", line 1: not implemented: formal type parameter T used as base class of
template
```

forward declaration of a specialized version of template *name*

A forward declaration of a specialised, rather than generalised template:

```
template <class T> struct S; struct S<int>;

"file", line 2: not implemented: forward declaration of a specialized version
of template S <int >
```

general initializer in initializer list

The initialiser list in a declaration contains an expression that cannot easily be evaluated at compile time or that requires runtime evaluation.

```
int f();
int i[1] = { f() };

"file", line 2: not implemented: general initializer in initializer list
```

initialization of *name* (automatic aggregate)

An aggregate at local scope is initialised. This message is not issued if the `+a1` option (produces declarations acceptable to an ANSI C compiler) is specified.

```
void f() {  
    int i[1] = {1};  
}
```

"file", line 2: not implemented: initialization of i (automatic aggregate)

initialization of union with initializer list

An object of union type is initialised with an initialiser list. This message is not issued if the `+a1` option (produces declarations acceptable to an ANSI C compiler) is specified.

```
union U { int i; float f; };  
U u = {1};
```

"file", line 2: not implemented: initialization of union with initializer list

initializer for class member array with constructor

This message should always be accompanied by an error message. The 'not implemented' message is inappropriate and should not be reported.

initializer for local static too complicated

This message should not be produced.

initializer for multi-dimensional array of objects of class *class* with constructor *name*

A multi-dimensional array of a class with a constructor has an explicit initialiser.

```
struct S { S(int); };  
S s[2][2] = {1,2,3,4};
```

"file", line 2: not implemented: initializer for multi-dimensional array of objects of class S with constructor ::s

implicit static initializer for multi-dimensional array of objects of class with constructor

```

class x {
public:
    x() ;
};

main() {
    static x xx[10][20];
}

```

"file", line 7: not implemented: implicit static initializer for multi-dimensional array of objects of class x with constructor

initializer list for local variable *name*

This message should not be produced.

label in block with destructors

A labelled statement appears in a block in which an object with a destructor exists.

```

struct S { S(int); ~S(); };
void f() {
    S s(5);
xyz:    ;
}

```

"file", line 5: not implemented: label in block with destructors

local class *name* within template function

A local class is defined inside a template function. A similar message is issued for local enums and local typedefs defined inside a template function:

```

template <class T> f() {
    class l { /*...*/ };
    enum E { /*...*/ };
    typedef int* ip;
};

```

"file", line 2: not implemented: local class l (local to f()) within template function

"file", line 3: not implemented: local enum E(local to f()) within template function

"file", line 4: not implemented: local typedef ip within template function

local static class *name* (*type*)

A static array of objects of a class with a constructor is declared at local scope.

```
class S {
public:
    S();
};
void f() {
    static S s[9];
}

"file", line 2: not implemented: local static class s ( S [9])
```

local static *name* has *class::~~class()* but no constructor (add *class:: class()*)

A static class object with a destructor, but no constructor, appears at local scope.

```
struct S { ~S(); };
void f() { static S s; }

"file", line 1: warning: S has S::~~S() but no constructor
"file", line 2: not implemented: local static s has S::~~S() but no constructor
(add S:: S())
```

lvalue *op* too complicated

This message should not be produced.

needs temporary variable to evaluate argument initializer

A default argument requires a temporary variable.

```
void f() {
    int g(const int& = 5);
}

"file", line 2: not implemented: needs temporary variable to evaluate argument
initializer
```

nested class *type* as parameter type to template class *name*

A nested class is used as the actual parameter for a template class instantiation:

```
template <class T> struct S;

struct outer {
    struct inner {};
};

S<outer::inner> svar;

"file", line 7: not implemented: nested class outer::inner as parameter type
to template class S
```

nested class *name* within nested class *name* within template class *name*

Classes may only be nested directly within template classes, classes within nested classes within template classes are not implemented:

```
template <class T> class S {
class nest1 {
    class nest2 { /*...*/ };
};

"file", line 3: not implemented: nested class S::nest1::nest2 within nested
class S::nest1 within template class S
```

nested depth class beyond 9 unsupported

Classes are nested more than nine levels deep.

```
struct S1 {
    struct S2 {
        struct S3 {
            struct S4 {
                struct S5 {
                    struct S6 {
                        struct S7 {
                            struct S8 {
                                struct S9 {
                                    struct S10 { enum { e }; };
                                };
                            };
                        };
                    };
                };
            };
        };
    };
};

"file", line 20: not implemented: nested depth class beyond 9 unsupported
```

non-trivial declaration in switch statement

A 'non-trivial' declaration appears within a switch statement. Such a declaration might declare an object of reference type, a static object, a `const` object, an object of a class type with constructor or destructor, an object with an initialiser list, or an object initialised with a string literal.

```
void f(int i) {
    switch (i) {
    default:
        int& j = i;
    }
}

"file", line 2: not implemented: non-trivial declaration in switch statement
(try enclosing it in a block)
```

Note that since it is illegal to jump past a declaration with an explicit or implicit initialiser unless the declaration is in an inner block that is not entered, most declarations in switch statements and not contained in inner blocks will be errors.

out-of-line definition of member function of class nested within template class

The member functions of a class nested within a template function must be defined within the definition of the nested class.

```
template <class t> struct x {
    struct y { void foo(); };
    // ...
};

template <class t>
    void x<t>::y::foo(){}

"file", line 7: not implemented: out-of-line definition of member function of
class nested within template class ( x::y:: foo())
```

overly complex *op* of *op*

This message should not be produced.

parameter expression of type float, double or long double

A template taking a non-type argument is declared taking a float, double or long double argument:

```
template <double d> struct S { /*...*/};

"file", line 1: not implemented: parameter expression of type float, double,
or long double
```

postfix template function operator ++(): please make a class member function

The postfix implementation of a template increment or decrement operator must be a member function.

```
template <class t> struct x {
    int operator++(int); // ok
};

template <class t>
    int operator++(x<t>&,int); // sorry

x<int> xi;

"file", "", line 6: not implemented: postfix template function operator ++():
please make a class member function
```

pointer to member function *type* too complicated

This message should not be produced.

public specification of overloaded function

The base class member in an access declaration refers to an overloaded function. A similar message is issued for `private` and `protected` access declarations.

```
struct B { int f(); int f(int); };
class D : private B {
public:
    B::f;
};

"file", line 2: not implemented: public specification of overloaded B::f()
```

reuse of formal template parameter name

A template formal parameter name is reused within the template declaration:

```
template <class T> struct S {
    int T;
};

"file", line 2: not implemented: reuse of formal template parameter T
```

specialized template name not at global scope

A specialised template is declared at other than global scope:

```
template <class T> struct S {
    T var;
};

void f() {
    struct S <int > {
        int var;
    };
};

"file", line 6: not implemented: specialized template S not at global scope
```

static member anonymous union

A static class member is declared as an anonymous union.

```
class C {
    static union {
        int i;
        double d;
    };
};

"file", line 5: not implemented: static member anonymous union
```

struct name member name

This message should not be produced.

template function actuals too complicated (please simplify)

```
#include <iostream.h>

template <class i> struct x { x(); };

template <class t>
ostream& operator<<(ostream &os, x<t>&) { return os; }

x<int> z;

main() {
/*
 * ok: simplified invocation of actual template function:
 *     cout << "hello"; cout << z << endl;
 */

// generates sorry message: actuals too complicated
cout << "hello" << z << endl;
}

"file", line 17: not implemented: template function operator <<(): actuals too
complicated (please simplify)
```

template function instantiated with local class *name*

```
template <class T> int f(T);

f2() {
    struct local { /*...*/};
    local lvar;
    f(lvar);
}

"file", line 6: not implemented: template function f() instantiated with local
class local
```

temporary of class *name* with destructor needed in *expr* expression

An expression containing a `?:`, `| |`, or `&&` operator requires a temporary object of a class that has a destructor.

```
struct S { S(int); ~S(); };
S f(int i) {
    return i ? S(1) : S(2) ;
}

"file", line 3: not implemented: temporary of class S with destructor needed
in ?: expression
```


too few initializers for *name*

The initialiser list for an array of class objects has fewer initialisers than the number of elements in the array.

```
struct S { S(int); S(); };
S a[2] = {1};
```

"file", line 2: not implemented: too few initializers for ::a

***type1* assigned to *type2* (too complicated)**

A pointer is initialised or assigned with an expression whose type is too complicated.

```
struct S1 {};
struct S2 { int i; };
struct S3 : S1, S2 {};
int S3::*pmi = &S2::i;
```

"file", line 4: not implemented: int S2::* assigned to int S3::* (too complicated)

use of *member* with formal template parameter

An attempt to use a member of a formal parameter type, such as `T::type`, is not currently supported. For example,

```
template <class T> class U {
    typedef T TU;
    // ...
};

template <class Type> class V {
    Type::TU t;
    // ...
};
```

"file", line 9: not implemented: use of Type::TU with formal template type parameter

"file", line 9: cannot recover from earlier errors

visibility declaration for conversion operator

An access declaration is specified for a conversion operator.

```
struct B { operator int(); };
class D : private B {
public:
    B::operator int;
};
```

"file", line 1: not implemented: visibility declaration for conversion operator

volatile functions

A member function is specified as `volatile`.

```
struct S {  
    int f() volatile;  
};
```

"file", line 2: not implemented: volatile functions

wide character constant

wide character string

A wide character constant or a wide character string is used.

```
int wc = L'ab';  
char *ws = L"abcd";
```

"file", line 1: not implemented: wide character constant

"file", line 2: not implemented: wide character string

C function index

Main entries are printed in bold type.

Symbols

`__heap_checking_on_all_allocates` **212**
`__heap_checking_on_all_deallocates` **212**
`__rt_stkovf_split_big` 321
`__rt_stkovf_split_small` 321
`_fmapstore` 29, **212**
`_kernel_stkovf_split_0frame` 321
`_kernel_stkovf_split_frame` 321
`_kernel_swi` 314
`_mapstore` 29, 32, **212**

A

`abort` 93, 95, **185**
`abs` **188**
`acos` 93, **148**
`acosh` **148**
`aligned_alloc` 95, **185**
`asctime` **204**
`asin` 93, **148**
`asinh` **148**
`assert` 93, **136**
`at_quick_exit` **186**
`atan` **148**
`atan2` 93, **148**
`atanh` **148**
`atexit` **185**
`atof` **181**
`atoi` **181**
`atol` **181**

B

`bsearch` **187**

C

`calloc` 95, **184**
`cbrt` **149**
`ceil` **149**
`clearerr` **179**
`clock` 96, **203**
`copysign` **149**
`cos` **148**
`cosh` **148**
`ctime` **205**

D

`difftime` **204**
`div` **189**

E

`erf` **149**
`erfc` **149**
`exit` 95, **185**
`exp` **148**
`exp2` **148**
`expm1` **148**

F

`fabs` **149**

`fclose` **164**
`feof` **179**
`ferror` **180**
`fflush` **164**
`fgetc` **172**
`fgetpos` 95, **176**
`fgetpos64` 129, **177**
`fgets` **173**
`floor` **149**
`fma` **149**
`fmax` **149**
`fmin` **149**
`fmod` 93, **149**
`fopen` **164**
`fopen64` 129, **165**
`fprintf` 94, **167**
`fputc` **173**
`fputs` **173**
`fread` **176**
`free` **184**
`freopen` **166**
`freopen64` 129, **166**
`frexp` **148**
`fscanf` 94, **169**
`fseek` **177**
`fseeko` 129, **177**
`fseeko64` 129, **177**
`fsetpos` **178**
`fsetpos64` 129, **178**
`ftell` 95, **178**
`ftello` 129, **179**
`ftello64` 129, **179**
`fwrite` **176**

G

`getc` **174**
`getchar` **174**
`getenv` 95, **187**
`gets` **174**
`gmtime` **205**

H

`hypot` **149**

I

`isalnum` 93, **138**
`isalpha` 93, **138**
`isblank` 93, **138**
`iscntrl` 93, **138**
`isdigit` **138**
`isgraph` **138**
`islower` 93, **138**
`isprint` 93, **138**
`ispunct` 93, **138**
`isspace` **138**
`isupper` 93, **138**
`isxdigit` **138**

L

`labs` **188**
`lconv` **146**
`ldexp` **148**
`ldiv` **189**
`lgamma` **149**
`llabs` **188**
`lldiv` **189**
`localtime` **205**
`log` 93, **148**
`log10` 93, **148**
`log1p` **148**
`log2` **148**
`logb` **149**
`longjmp` 33, **151**

M

`main` 87, 307, 312
`malloc` 95, **184**, 312
`mblen` **190**

mbstowcs **191**
mbtowc **190**
memchr **197**
memcmp **195**
memcpy **194**
memmove **194**
memset **199**
mktime **204**
modf **149**

N

nan **149**
nearbyint **149**
nextafter **149**
nexttoward **149**

O

order **150**

P

perror 95, **180**
pow **149**
printf 101, 121, **168**
putc **174**
putchar **175**
puts **175**

Q

qsort **188**
quick_exit 95, **186**

R

raise **153**
rand **183**

realloc 95, **184**
remainder **149**
remove 94, **162**
rename 94, **163**
rewind **179**
rint **149**
round **149**

S

scalbln **149**
scalbn **149**
scanf 101, 121, **170**
setbuf **166**
setjmp **151**
setlocale 93, **146**
setvbuf **167**
signal 93, 94, 311
sin **148**
sinh **148**
snprintf 101, 103, **169**
sprintf **169**, 312
sqrt 93, **149**
srand **184**
sscanf **170**
strcat **195**
strchr **197**, 312
strcmp **196**
strcoll **196**
strcpy **194**
strcspn **197**
strerror 95, **199**
strftime 101, **205**
strlen **199**
strncat **195**
strncmp **196**
strncpy **195**
strpbrk **197**
strrchr **198**, 312
strspn **198**
strstr **198**
strtod **181**

strtok **198**
strtol **182**
strtoll **182**
strtoul **183**
strtoull **183**
struct tm **203**
strxfrm **196**
system 95, **187**

T

tan **148**
tanh **148**
tgamma **149**
time **204**
timespec_get 96, **207**
tmpfile **163**
tmpfile64 129, **163**
tmpnam **163**
tolower **138**
toupper **138**
trunc **149**

U

ungetc **175**

V

va_arg **155**
va_copy **156**
va_end **156**
va_list **155**
va_start **155**
vfprintf **171**
vfprintf 102, **172**
vprintf **170**
vscanf 102, 103, **172**
vsprintf 101, **171**
vsprintf **171**

vsscanf 102, **172**

W

wcstombs **191**
wctomb **191**

X

x\$stack_overflow 321
x\$stack_overflow1 321

C++ class index

Main entries are printed in bold type.

C

c_exception
 complex_error **289**
cerr **226**
cin **226**
clog **226**
complex **286**
 – **294**
 * **295**
 *= **295**
 / **295**
 /= **295**
 + **294**
 += **295**
 -= **295**
 != **295**
 == **295**
abs **287**
arg **287**
conj **287**
cos **297**
cosh **297**
exp **292**
imag **288**
log **292**
norm **287**
polar **288**
pow **292**
real **288**
sin **297**
sinh **297**
sqrt **292**
cout **226**

F

filebuf 227, **229**
 attach **231**
 close **231**
 fd **231**
 filebuf **230**
 is_open **231**
 open **231**
 seekoff **231**
 seekpos **232**
 setbuf **232**
 sync **232**
fstream 227, **233**
 attach **235**
 close **235**
 fstream **234**
 open **235**
 rdbuf **236**
 setbuf **236**

I

IAPP **255**
ifstream 227, **233**
 attach **235**
 close **235**
 ifstream **234**
 open **235**
 rdbuf **236**
 setbuf **236**
IMANIP **255**
IOAPP **255**
IOMANIP **255**
ios 225, **237**
 ! **240**

***** 240
<< 246
>> 246
bad 240
bitalloc 244
clear 239
dec 241
eof 240
fail 240
fill 243
fixed 242
flags 243
good 240
hex 241
init 239
internal 241
ios 239
isword 245
left 241
oct 241
precision 243
pword 245
rdbuf 245
rdstate 239
right 241
scientific 242
setf 243
showbase 241
showpoint 242
showpos 241
skipws 241
stdio 242
sync_with_stdio 245
tie 245
unitbuf 242
unsetf 244
uppercase 242
width 244
xalloc 244
iostream 225
iostream_init 226
iostream_withassign 226
istream 225, 248

>> 250, 254
gcount 253
get 252
getline 252
ignore 253
ipfx 250
istream 250
istream_withassign 250
manip 253
peek 253
putback 253
read 253
seekg 254
sync 253
tellg 254
istream_withassign 226
istrstream 228, 279
istrstream 280
rdbuf 280

M

main 217
matherr 290

O

OAPP 255
ofstream 227, 233
attach 235
close 235
ofstream 234
open 235
rdbuf 236
setbuf 236
OMANIP 255
ostream 225, 259
<< 262
dec 264
endl 264
ends 264

flush **263, 264**
hex **264**
manip **263**
oct **264**
opfx **261**
osfx **261**
ostream **261**
ostream_withassign **261**
put **263**
seekp **264**
tellp **264**
write **263**
ostream_withassign 226
ostrstream 228, **279**
 ostrstream **280**
pcount **281**
rdbuf **281**
str **281**

S

SAPP **255**
SMANIP **255**
stdiobuf 227, **265**
stdiostream 228
streambuf 225, **266, 274**
 allocate **270**
 base **268**
 blen **270**
 dbp **270**
 doallocate **271, 272**
 eback **268**
 ebuf **268**
 egptr **268**
 epptr **268**
 gbump **270**
 gptr **269**
 in_avail **276**
 out_waiting **276**
 overflow **271, 272**
 pbackfail **271, 272**
 pbase **269**

pbump **270**
pptr **269**
sbumpc **276**
seekoff **271, 273, 276**
seekpos **271, 272, 276**
setb **269**
setbuf **272, 273, 277**
setg **269**
setp **269**
sgetc **277**
sgetn **277**
snextc **277**
sputbackc **277**
putc **277**
putn **277**
stossc **278**
streambuf **268**
sync **272, 273, 278**
unbuffered **270**
underflow **272, 273**
strstream **279**
 rdbuf **281**
 str **281**
 strstream **280**
strstreambuf 227, **282**
 freeze **284**
 setbuf **284**
 str **284**
 strstreambuf **283**



Index

Symbols

`__func__` 100
`__STDC__` 107, 116
`__STDC_HOSTED__` 107, 116
`__STDC_VERSION__` 107, 117
`__VA_ARGS__` 102
`__alignas_is_defined` 154
`__alignof_is_defined` 154
`__APCS_32` 126
`__APCS_FPREGARGS` 127
`__APCS_INTERWORK` 127
`__APCS_NOFP` 127
`__APCS_NOSWST` 127
`__APCS_REENT` 127
`__arm` 126
`__ARM_32BIT_STATE` 128
`__ARM_ARCH` 127
`__ARM_ARCH_ISA_A64` 128
`__ARM_ARCH_ISA_ARM` 128
`__ARM_ARCH_ISA_THUMB` 127
`__asm` statement 130
`__bool_true_false_are_defined` 158
`__CC_NORCROFT` 126
`__CC_NORCROFT_VERSION` 126
`__global_freg(n)` 125
`__global_reg(n)` 125
`__pure` 125
`__riscos` 126
`__value_in_regs` 125
`__Bool` 111, 158
`__FILE_OFFSET_BITS` 129
`__IN` 314
`__kernel_swi_regs` 314
`__LARGEFILE_SOURCE` 129
`__LARGEFILE64_SOURCE` 129
`__OUT` 314

`__Pragma` 110
`__swi` 314
`__swix` 314, 315
`:mem` 19, 21
`:tt` 87
`//` 98
`#include` 16, 18–21, 22, 58, 61

Numerics

64-bit integers 97

A

absolute machine addresses 306
alignment 114, 303
an 318, 319
anonymous structures and unions 122
ANSI library 14, 29, 211–212
ANSI standard 7
APCS 44, 317
 APCS-32 97
 APCS-R 97
arguments 222
 passing to assembler 319–320
arithmetic
 IEC 60559 110
 IEEE 110
arithmetic operations 84–85
arrays 90, 282–284, 307
 flexible members 99
 variable-length 98, 111
asm declarations 220
assembly language 317–322
 inline 130

assembly output 33
assert.h 116, 136

B

bibliography 6–7
bitfields 91, 221
BL 319
bool datatype 97, 158
BS ISO/IEC 9899:1999 7
BS ISO/IEC 9899:2018 7
buffers 225, 227
 characters 274–278
 file I/O 229–232
buttons *see application (button name)*
byte ordering 302

C

C Module Header Generator *see* CMHG
C++ 2, 11–55
 Assembler 18, 30
 Auto run 42
 Auto save 42
 Cancel 13
 command line 40, 43–50
 Command line (menu option) 13, 25
 Compile only 17, 23
 Debug 23
 Default path 17, 19–21, 25
 Define 26
 Features 20, 31–33
 icon bar menu 42
 Include 17, 19–21, 22
 Module code 30
 Options 42
 Others 40
 Run 13, 22, 25
 Save options 42
 SetUp dialogue box 12–13, 22–24
 SetUp menu 13, 24–40

 Source 12, 22
 Suppress warnings 32, 33
 Throwback 24
 Undefine 27
 Work directory 15, 39
C++ library 14, 223–298
C++Hello example 52
 see also HelloW example
C\$Libroot 20, 21
C\$Path 19, 22
C18 2, 7, 112
 non-compliances 118
C90 2, 7, 38, 44
C99 2, 7, 38, 44, 97, 110, 111
 non-compliances 110
 restrictions 110
callbacks 313
cartesian coordinates 287–288
case sensitivity 43
CC 2, 11–55, 327
 Assembler 18, 30, 321
 Auto run 42
 Auto save 42
 Cancel 13
 command line 40, 43–50
 Command line (menu option) 13, 25
 Compile only 17, 23
 Debug 23
 Debug options 28
 Default path 17, 19–21, 25
 Define 26
 Errors to file 37
 Features 20, 29, 31–33, 39
 icon bar menu 42
 Include 17, 19–21, 22
 ISO C90 38
 ISO C99 38
 Keep comments 26
 Libraries 30
 Listing 17, 32, 39
 Module code 30
 Options 42
 Others 29, 40

- Preprocess only 23, 41, 310
- Profile 29
- Run 13, 22, 25
- Save options 42
- SetUp dialogue box 12–13, 22–24
- SetUp menu 13, 24–40
- Source 12, 22
- Suppress errors 36
- Suppress warnings 32, 33–36
- Throwback 24
- Undefine 27
- UNIX pcc 38
- Work directory 15, 39
- CFront 2, 11, 12, 48
- characters 88–89
 - testing and mapping 138
- chars 78
- CHello example 52
 - see also* HelloW example
- classes
 - members 221
 - multiple base 221
- CMHG 57–60, 327–348
 - AOF 58
 - C Header 58
 - command line 60
 - Command line (menu option) 59
 - command line options 61
 - date in module version string 336
 - Define (menu option) 59
 - description files 59, 331
 - icon bar menu 59
 - Include 58
 - include files 61
 - module version string 336
 - Others (menu option) 59
 - pre-defining macros from the command line 61
 - Preprocess 58
 - SetUp dialogue box 58
 - SetUp menu 59
 - Source 58
 - Throwback 58

- using the C preprocessor 331
- CModule example 53
- comments 98, 305
- common subexpression elimination 123
- compatibility issues 111, 118
- compiler *see* CC and C++
- Complex Math library 285–298
 - operators 294–296
- complex numbers 110, 286
- complex.h 116, 137
- compound literals 102
- const qualifier 305
- constants
 - character 114, 216
 - floating 216
 - hexadecimal 302
 - octal 305
- control statements 307
- conventions 5
- conversion specifiers 97, 103
- conversions 218, 219, 225, 306
- cpp 310
- cross-jumping 122
- ctype.h 138, 310
- current place 19–20

D

- data elements 78–81
 - limits 79–81, 145
- date in module version string 336
- debugging
 - machine level 23
 - source level 23
 - tables 23, 28
- declarations 307
 - in for statements 98
- device drivers 327
- Dhrystone 2.1 example 53
- diagnostics 136
- dialogue boxes *see application (dialogue box name)*
- digraphs 109

doubles 78, 85

E

enum 100
enumeration types 92
errno.h 139–140, 311
errors 24, 36, 37, 41, 87, 239–240, 363–414
 browser 24
 Complex Math library 289–291
 domain 139
 EDOM 139, 290, 293
 EFBIG 140
 EILSEQ 139
 EOVERFLOW 139
 ERANGE 139, 290
 ESIGNUM 139
 file too big 140
 illegal sequence 139
 overflow 139
 range 139
examples 52–55
exception handling 222
exponential functions 292–293
expressions 218
 evaluation 85

F

false 97, 158
fenv.h 103, 141
FILE 162
filenames 14–18
 case-sensitivity 313
 extensions 15, 313
 length 313
 rooted 16, 19
files
 buffering 94
 closing 164
 creating 231, 235

 deleting 162
 flushing 164
 formatted I/O 227
 large 129
 naming 163
 opening 164–166, 231, 234, 235
 position indicators 176–180
 reading 229
 renaming 163
 seeking 231, 235
 syncing 232
 temporary 163
 writing 229
 zero-length 94
flags 43–50
flexible array members 99
float.h 103, 116, 142, 312
floating point 90, 103, 142
 environment 141
 hexadecimal format 101, 103
floats 78, 85, 306
fn 318, 320
for
 declarations in 98
fp 319, 320
fpos_t 162
fpos64_t 129, 162
functions
 arguments 304
 array parameter 100
 calls 218
 declaration keywords 125
 declarations 307
 definitions 307
 name of 100
 non returning 113
 one per area 49
 prototypes 307
 untyped 110
 workspace 321

G

get area 268

H

halfword instructions 324
header files 11, 14, 18, 103
 from CMHG 58
 standard 18
heap checking 212
HelloW example 12–13, 16
HUGE and HUGE_VAL 311
Hyper example 55
hyperbolic functions 297–298

I

I/O
 buffering 166–167
 redirection 88
I/O functions 162–180
icons *see application (icon name)*
Idempotent type qualifiers 110
identifiers 78, 88, 216
IEEE double precision 319
IEEE single precision 319
implementation limits 85
include files 16, 22, 25, 58, 61, 121
 nesting 20
 searching for 18–21
inline assembler 130
inline functions 91
inline SWI functions 315
input functions 169–170, 172–173, 174, 175–176
installation 1
integers 90
interactive devices 87
intmax_t 101
Intrinsic functions 128
 __breakpoint 128

 __current_pc 128
 __current_sp 128
 __nop 128
 __return_address 128
 __schedule_barrier 128
 __semihost 129

ints 78
inttypes.h 103, 143
ip 318
ISO 8601 101
ISO 8859-1 88
ISO standard 11, 87–96
 vs K&R 304–308
ISO/IEC 9899:1990 2
ISO/IEC 9899:1999 2, 7, 107
ISO/IEC 9899:2018 2, 7
iso646.h 103, 144

K

kernel.h 19, 314
Kernighan and Ritchie 6

L

large file support 129
Latin-1 character set 88
LDM 50
libraries 4, 14, 19, 22, 30, 135–212, 223–298
 ISO vs BSD UNIX 310–312
limits.h 145, 312
Link 11, 12, 23
 Debug 23
link register 32
linkage specifications 220
listings 17, 30, 32, 39, 321
locale.h 146–147, 312
logarithmic functions 292–293
long doubles 78, 85, 305, 306
long floats 305
long ints 85

long longs 78, 97, 111
longs 78, 305
lr 128, 318, 319

M

macros 102, 110, 222
 pre-defined 107, 116, 126
Make 14, 16, 47, 57, 65, 70
manipulators 255–258
math.h 103, 148–150, 311
mathematical functions 93, 148–150, 188–190
memory allocation functions 184–185
menus *see application (menu name)*
MinApp example 55
modules 30, 57, 327–348
 application code 329, 332
 components 329–330
 constraints 328
 date string 329, 336
 event handler 330, 344–345
 finalisation code 329, 333–334
 generic veneers 330, 346–347
 header 57
 help and command keyword table 329, 337–339
 help string 329, 335–336
 initialisation code 329, 333
 international help file 330, 339
 library enter code 330, 347
 library finalisation code 330, 347
 library initialisation code 330, 347
 non-reentrant 347
 service call handler 329, 334–335
 SWI chunk base number 330, 339–340
 SWI decoding code 330, 341–342
 SWI decoding table 330, 341
 SWI handler code 330, 340
 title string 329, 335
 turning interrupts on and off 348
 vector handlers 330, 342–343
 version number 336

multibyte character functions 190–191
multibyte string functions 191–192
multi-tasking 313
multi-threading 313

O

object files 11, 14, 17, 22, 23, 42, 58, 60
off_t 129, 162
off64_t 129, 162
offsetof 159
op-codes 33
operating system interface 187, 304, 312–313
operators
 alternative spellings 144
 multiplicative 219
 relational 219
 shifts 219
optimisation 122–123
output 41–42, 60, 66, 73
output functions 167–169, 171, 173, 174–175, 176
overlays 349–351
 alternatives to 350

P

paging 349
pathname separator 313
pc 128, 319
pcc 31, 38, 44, 63–74, 122, 308–310
pointers 78, 84, 90, 303, 306
 restricted 103
 subtraction 84
polar coordinates 287–288
portability 301–315
portable C compiler *see* pcc
power functions 292–293
pragmas 50, 72, 119–124
 header file 19
preprocessor 11, 12, 18, 23, 25–28, 32, 48, 121,

- 307, 310
- directives 92
- translation ordering 308
- using with CMHG 331
- see also* CC and C++
- profiling 29, 212
- program termination functions 185–186
- ptrdiff_t 159
- put area 268

R

- RAM filing system 350
- random numbers 183–184
- register storage class 91
- register variables 124, 125
- registers
 - names 318
 - usage 318–319
- reserve area 268
- resource files 15
- restrict 103
- Risc PC 324
- rooted filenames *see* filenames (rooted)

S

- scheduler 23, 123, 129
- search functions 187
- setjmp.h 151
- SetPaths 22
- shared C library 14, 29, 87, 92–96, 135–210, 324, 359
 - modules 327
- SharedCLibrary 97, 112, 324, 328
- shorts 78
- Sieve example 53
- signal.h 152–153, 311
- signals 139, 152–153
- signed qualifier 305
- size_t 101, 159

- sl 319, 320, 321
- SMULL 111
- Software Interrupt *see* SWI
- sort functions 188
- source files 11, 14, 16
- sp 128, 319, 320, 321
- specifiers
 - storage class 220
 - type 220
- square root functions 292–293
- SrcEdit 23, 24
- stack checking 44, 123
- stack extension 321
- stdalign.h 116, 154
- stdarg.h 155–156
- stdatomic.h 157
- stdbool.h 97, 103, 158
- stddef.h 159
- stderr 88
- stdin 88
- stdint.h 103, 160–161
- stdio.h 103, 116, 162–180, 312
- stdlib.h 116, 181–192, 312
- stdnoreturn.h 116, 193
- stdout 88
- STM 50
- streams 226–227, 237–247
 - formatting 240–245, 250–252, 262–263
- Streams library 223–284
- string functions
 - appending 195
 - comparison 195–196
 - conversion 181–183
 - copying 194–195
 - error message mapping 199
 - length 197, 198, 199
 - locating 197–198
 - time 205–210
 - tokenising 198–199
 - transformation 196–197
- string literals 33, 217, 305, 307
- string.h 194–200, 312
- StrongARM 111, 324

- structures 81–83, 91, 125, 303, 305
 - anonymous 122
 - initialising 100
 - results 320
- stubs 14, 29, 135, 327, 328
 - entry vectors 135
- summary 41
- SWI 314, 327
- SWI functions 314–315
 - in-line 315
 - kernel.h 314
 - swis.h 314
- SWI numbers 314
- swis.h 19, 314
- switch statement 307

T

- TBoxCalc example 55
- text streams 94
- tgmath.h 110, 201
- threads.h 202
- throwback 24, 47
- time.h 116, 203–210
- ToANSI 63–67, 308
 - command line 67
 - Command line (menu option) 65
 - File 65
 - icon bar menu 66
 - SetUp dialogue box 65
 - SetUp menu 65
- token-pasting 307
- tools 9–74
 - common features 42, 57, 63, 69
- ToPCC 69–74, 308
 - command line 74
 - Command line (menu option) 71
 - File 71
 - icon bar menu 72
 - Options 72
 - SetUp dialogue box 71
 - SetUp menu 71–72

- translation limits 215
- trigonometric functions 297–298
- true 97, 158
- types 217
 - checking 309
 - generic 115
- typographic conventions *see* conventions

U

- uchar.h 208
- UMULL 111
- unions 91, 305
 - anonymous 122
- Universal character names 109
- UNIX 16, 17, 20, 312, 313
- unsigned long ints 306
- unsigned qualifier 85, 305

V

- varargs.h 19
- variable-length arrays 98, 111
- variables
 - declaration keywords 125
 - storing 321
- variadic functions 305
- variadic macros 102
- verbose mode 31
- vn* 319, 320
- void 305
- void * 305
- volatile qualifier 92, 122, 305

W

- warnings 33–36, 87, 363–414
- wchar_t 159
- wchar.h 209
- wctype.h 210

week number 101

Windows 16, 17

work directory 14, 39, 47



Reader's Comment Form

Acorn C/C++, Issue 7

We would greatly appreciate your comments about this Manual, which will be taken into account for the next issue:

Did you find the information you wanted?

Do you like the way the information is presented?

General comments:

If there is not enough room for your comments, please continue overleaf

How would you classify your experience with computers?

☐

Used computers before

☐

Experienced User

☐

Programmer

☐

Experienced Programmer

Please send an e-mail with your
comments to:
manuals@riscosopen.org

Your name and address:

This information will only be used to get in touch with you in case we wish to explore your comments further