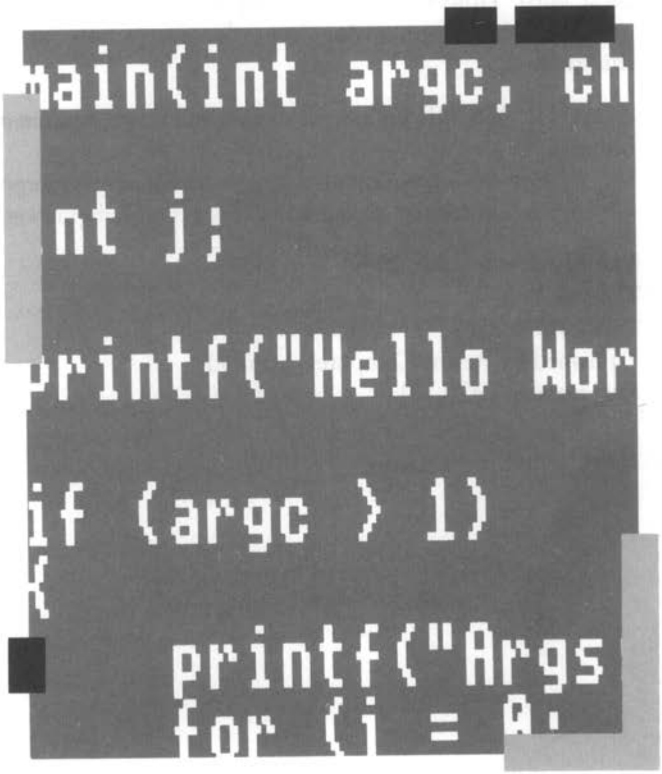
ANSI C RELEASE 3

Acorn
The choice of experience.

ANSI C RELEASE 3



Acorn
The choice of experience.

The product described in this manual and products for use with it are 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 Acorn Computers Limited in good faith. However, Acorn Computers Limited cannot accept any liability for any loss or damage arising from the use of any information or particulars in this manual.

Published September 1989
Release 3
ISBN 1 85250 071 9
Published by Acorn Computers Limited
Part number 0470,101

Contents

Part 2: Language issues

Part 3: Developing software for RISC OS

Indexes

Introduction

The Acorn C compiler for the ARM processor is a full implementation of C as defined by the December 1988 draft ANSI language standard.

About this Guide

This Guide is a reference manual for the Acorn C compiler for RISC OS and covers aspects that are particular to this C product:

- special features of this implementation of the C language
- installing and working with C on your RISC OS computer
- portability issues, including the portable C compiler (pcc) facility
- developing programs for the RISC OS environment:
  - Desktop applications
  - Relocatable modules
  - Overlays
  - Calling other programs from C.

The Guide is not intended as an introduction to C and does not teach C programming, nor is it a reference manual for the ANSI C standard. Both these needs are addressed by publications listed below.

The Guide is organised into four parts:

Part 1: *Using the C compiler and tools*

Part 2: *Language issues*

Part 3: *Developing software for RISC OS*

Part 4: *Appendices*

Part 1: Using the C compiler and tools

This describes how to use the compiler and tools provided for program development. It also helps you with organising the C compiler system for your particular needs and getting the most productive use from your hardware configuration.

The chapters are:

- *How to install and run the compiler*
- *Using the linker*
- *Acorn source-level debugger*
- *Other utilities*

Part 2: Language issues

This covers issues to do with the C programming language itself, in particular those parts of the ANSI standard that are necessarily machine- or operating system-specific. It also includes a chapter on portability to help with porting applications in C to and from RISC OS.

The chapters are:

- *Implementation details*

  How Acorn C implements those aspects of the language which ANSI leaves to the discretion of the implementor.

- *Standard implementation definition*

  How Acorn C behaves in those areas covered by Appendix A.6 of the draft standard (which lists those aspects which the standard requires each implementation to define).

- *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
  - using the conversion utilities, `topcc` and `toansi`
  - standard headers and libraries
  - environmental aspects of portability.

- *ANSI library reference section*

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

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

Part 3: 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.

The chapters are:

- *How to write desktop applications in C*

  This covers the principles of designing an application to be integrated into the RISC OS desktop environment.

- *How to use the template editor*

  The template editor is a utility which takes much of the work out of building components of the Wimp environment, especially windows.

- *RISC OS library reference section*

  A list of fully commented headers for the RISC OS library. This library provides the high-level interface to RISC OS, with all the calls needed to program for the Wimp environment.

- *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.

- *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.

- *Machine-specific features*

  This chapter contains the following sections:

  - The C library kernel
  - Calling other programs from C
  - The shared C library
  - #pragma directives
  - Storage management
  - Handling host errors.

Part 4: Appendices

Appendix A: New features of Release 3

This is the third release of the C compiler product for Acorn computers running the RISC OS operating system. The appendix highlights all those features that are new since the previous release (release 2).

Appendix B: Arthur Operating System library

The Arthur library is for the old Arthur operating system, the precursor to RISC OS. It has been included for backwards compatibility.

Appendix C: Errors and warnings

Messages produced by the compiler, of varying degrees of seriousness.

Appendix D: ARM procedure call standard

This describes the technical details of the procedure call standard that language processors must adhere to in order to integrate into the RISC OS system. You will need this information if you are writing a language processor in C.

Appendix E: kernel.h

Fully-commented headers for the C library kernel. This provides the technical details needed to support the explanatory section on the kernel in the chapter *Machine-specific features*.

Appendix F: The floating point emulator

This covers what you need to know about the floating point emulator in order to use the C compiler system and write applications using it.

Useful references

C programming

- Harbison, S P and Steele, G L, (1984) A C *Reference Manual*, (second edition). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-109802-0.

  This is a very thorough reference guide to C, including a useful amount of information on the draft proposed ANSI C.

  Since the Acorn C compiler is an ANSI compiler, this book is particularly relevant, but you must get the second edition for coverage of the ANSI draft 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 draft ANSI C too.

- Koenig, A, (1989) *C Traps and Pitfalls*, Addison-Wesley, Reading, Mass. ISBN 0-201-17928-8.

  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.

RISC OS

- The *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*.

Reference cards

In a pocket inside the back cover of this Guide there are four reference cards, summarising

- the contents of the three release discs
- an overview of the C compiler directory structure
- options for the compiler, linker and utilities.

The ANSI standard

The Draft Proposed American National Standard (Programming Language C) is available for £65 from

British Standards Institution
Foreign Sales Department
Linford Wood
Milton Keynes
MK14 6LE

Members of the BSI can order copies by telephone; non-members should send a cheque payable to BSI.

However, you should find the coverage of ANSI C in this manual and the books listed above adequate for all but the most demanding requirements.

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 specific examples.

A bold version of the same font is used for text that the computer responds with.

Part 1 – Using the C compiler and tools

How to install and run the compiler

Introduction

The Acorn C compiler system is a powerful and flexible tool for developing software in C. The RISC OS operating system itself provides a rich working environment, with many facilities that you can use to aid software development.

How best to install and run the compiler, and set up your working environment, will depend on the hardware you are using and on your purpose in using the C compiler. This chapter therefore outlines the options available; where specific details are given, these will use the defaults for the procedure in question. The defaults are likely to cater for your needs unless you have special requirements.

This chapter is divided into four sections:

- *Using the compiler* first leads you through compiling, linking and running a simple example program (provided in the package). It then describes how the compiler system works, detailing the command line options and naming conventions.
- *Installation* lists the contents of the three release discs and tells you how to install the system on a hard disc or network, and how to work with a system with a single floppy disc drive.
- *Setting up your working environment* provides guidelines for getting the best use out of your hardware system, outlining ways to exploit the facilities of RISC OS. It suggests some ways to economise on memory and storage space.
- *Examples* works through all the example programs provided.

If you are new to RISC OS and the Acorn C compiler, read the whole of this chapter before starting to use the C compiler system. Experienced C programmers will find this chapter essential for reference, and may choose to tackle the examples section first.

However, even if you are fully conversant with previous releases of the Acorn C compiler and Acorn RISC OS computer systems, you must read the section entitled *The shared C library* at the end of the section *Setting up your working environment*.

Using the compiler

Release 3 of the ANSI C Compiler for Acorn computers running RISC OS comes on three discs:

- Disc 1 – The work disc
- Disc 2 – The library support disc
- Disc 3 – The reference disc.

Each disc is in E format and is write-protected.

Getting started

Before you do anything else, you should make working copies of each disc and keep the originals in a safe place. The *User Guide* supplied with your computer tells you how to format and make backup copies of discs.

Having backed up the three C distribution discs, insert your working copy of Disc 1 in the drive and exit to the Command Line prompt. To do this, press function key F12 or select *** Commands** from the Task Manager menu.

1. Select the Advanced Disc Filing System with

   ```
   *adfs
   ```

2. You will need to ensure that version 3.50, or later, of the shared C library is installed. At the * prompt, type:

   ```
   *RMensure SharedClibrary 3.50 RMload :0.$.!System.modules.clib
   ```

3. To ensure that an up-to-date version of the floating point emulator is installed, at the * prompt type:

   ```
   *RMensure FPEmulator 2.8 RMload :0.$.!System.modules.fpe
   ```

   You are now ready to compile, link, and run your first example program.

4. Select `$.Library` on drive 0 as the library directory with

   ```
   *lib :0.$.Library
   ```

5 Select `$.User` as the current directory with:

```
*DIR :0.$.User
```

6 To compile and link the example program `HelloW`, type

```
*cc c.HelloW
```

7 The compiler will give you a message similar to the following:

`Norcroft RISC OS ARM C 3.00 date`

When this process is completed, the * prompt will return.

8 To run the program, now type:

```
*HelloW
```

and the program will print the message Hello World on the screen.

Pressing Return at the * command will return you to the desktop. (Should this fail, refer to the section entitled *Shared C library* later in this chapter.)

You could incorporate steps 1 to 5 in an Obey file in order to set up your working environment. The file `!CStart` has been included on Disc 1 as an example.

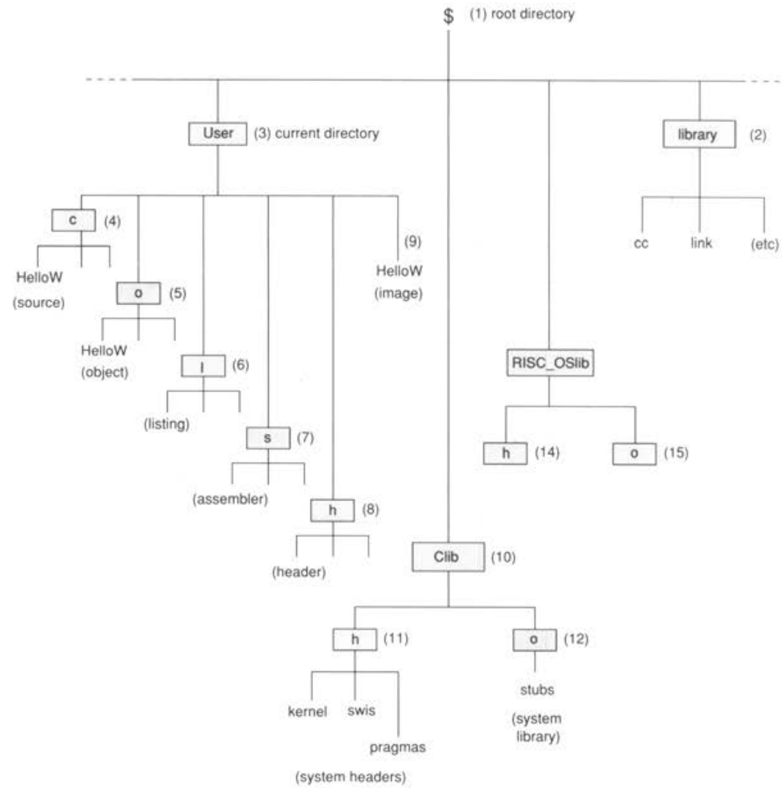
Filing systems

RISC OS supports several filing systems, of which these three are the most commonly found:

- ADFS Advanced Disc Filing System (floppy or hard disc)
- NetFS Network filing system for the Econet network
- RAMFS RAM filing system.

Each uses the same kind of hierarchical directory structure. For full information on filing systems, refer to the *User Guide* supplied with your computer, and the *RISC OS Programmer's Reference Manual*.

The diagram over the page illustrates schematically the organisation of directories for the C compiler system; a copy of it appears on one of the reference cards at the back of this Guide. References to this diagram will be made by parenthesised numbers in the sections that follow.



When you issue a command to the operating system that is not one of the resident commands – such as *cc to invoke the compiler – RISC OS will look in the current filing system.

Three directories are particularly important to the way the search of the filing system works:

- the current directory (3) selected by *Dir

  This is the first place in the currently active filing system where the operating system will search for a program.

- the library directory (2)   set by *Lib until a hard reset

  This is one of the places that RISC OS will search for programs. In the example given in the previous section, once step 4 has been carried out, the operating system looks for the program `cc` (the C compiler) in `$.Library`.

  The library directory is the best place for commonly-used tools such as the C compiler.

  Note that this use of the term *library* is different from its use to refer to extensions to a programming language, such as the ANSI C library.

- the root directory `$` (1)   the top of the directory tree

  A specification of the full pathname of an object (file or directory) will always start with the root directory, eg `$.User.c.HelloW`.

Introducing the C compiler

The general form of the command invoking the C compiler is

`*cc` *`options filenames`*

The options allow you to control the compilation (for example, by overriding default names), and are described in detail later in this chapter. The following description assumes the default settings used by the compiler.

The C compiler looks for source files in the subdirectory `c` of the current directory (4). These are compiled into AOF (Acorn Object Format) and placed in the subdirectory `o` of the current directory (5), using the same filename as the source.

Linking is carried out by default, using `$.Clib.o.Stubs` (12), which interfaces your program to the shared C library. The executable program in Acorn Image Format (AIF) is placed in the current directory, again retaining the source filename (9).

Consider the example given, where `$.User` is the current directory: when the compiler is invoked, the source file `$.User.c.HelloW` is compiled into the object file `$.User.o.HelloW`, and linked with `$.Clib.o.Stubs` to produce the executable image `$.User.HelloW`. This is run in response to step 8 in the sequence given earlier. To complete the picture: the subdirectories `l` and `s` are used for listing (7) and Assembler (8) output respectively, and `h` is used for header files (9); none of these three is used in this example.

A more detailed treatment is given in the section below entitled *Naming conventions*.

Introducing the libraries

There are two types of library provided to support the C compiler:

- the standard ANSI library (also referred to as the C library)

  This provides all the standard facilities of the language, as defined by the ANSI draft standard document. Code using calls to the ANSI library will be portable to other environments if an ANSI compiler is available for that environment.

  The ANSI library used with the Acorn C compiler system is called the *shared C library*. It is a relocatable module, supplied on Disc 1 as `$.!System.`modules.Clib, and must be installed in the relocatable module area (as described in the `HelloW` example). C programs are linked with a small piece of code and data, called `Stubs`, which interfaces with the shared C library. `Stubs` is supplied on Disc 1 as `$.Clib.o.Stubs`.

  The idea behind the shared C library is that a number of applications which are resident in memory at the same time can use it, thus economising on RAM space. It also saves space on disc, benefitting users with single floppy disc drives.

- the operating system library

  This provides you with routines to harness the special facilities of the operating system, in particular the Wimp environment. Code using calls to this library will not port to other environments.

  The operating system library used with the Acorn C compiler system is the RISC OS library, `RISC_OSlib`. It provides all the calls you need to program the Wimp environment and write desktop applications. `RISC_OSlib` is supplied on Disc 2 as `$.RISC_OSlib.o.RISC_OSlib`.

Naming conventions

The Acorn C system, in common with many other C systems, 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 MS-DOS™ systems. This convention clashes with Acorn's use of the full-stop character in pathnames. It is more natural under Acorn filing systems to use a prefix convention, eg `c.foo`, where `c` is the directory containing C source files, and `foo` is the filename.

However, portability is an increasingly important issue in the C world. To this end, the Acorn C system 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.

Source files

Source files will be looked for in subdirectory `c`. To aid portability, a file `foo.c` will be looked for in `@.c.foo`, where `@` means the current directory. In the HelloW example, with `$.User` as the current directory, any of the following commands can be used for step 6:

`*cc c.HelloW`	prefix
`*cc HelloW.c`	suffix
`*cc HelloW`	source in subdirectory `c`
`*cc $.User.c.HelloW`	full pathname

Include files

The way in which the compiler deals with included files is dealt with in detail in the section entitled *Controlling the preprocessor*, later in this chapter. Here we provide an overview, assuming the defaults and covering what you will need for most purposes, namely:

- headers for the ANSI C library
- headers for the RISC OS library
- your own include files.

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

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

in the HelloW example.

By convention, header files are placed in subdirectory `h`. This convention is followed here ((8), (11) and (14)).

A special feature of the Acorn C system is that the standard ANSI headers are built into the C compiler, and are used by default. By placing the filename in angle brackets, you indicate that the include file is a 'system' file, and thus ensure that the compiler looks first in its built-in filing system.

The non-ANSI library headers – `kernel`, `pragmas`, `swis` and `varargs` – are not built in to the compiler. By default, they will be found by the compiler in `$.Clib.h` (10)(11).

Headers for the RISC OS library are located in `$.RISC_OSlib.h` (14). You can incorporate these using the `-I` compiler option. For example, in the `!Balls64` example in `$.DeskEgs` on Disc 1:

in the source	`#include "wimp.h"` etc
on compilation	`cc -c -I$.RISC_OSlib balls64`

This is illustrated in the *Examples* section at the end of this chapter. Placing the filename in double quotes in the `#include` directive indicates a 'user' file. You can use subdirectory `h` of the current directory for your own header files (8), which can be incorporated with a source line like:

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

Object files

The object files created by the compiler are stored in the directory `o` within the current directory. Thus the result of compiling `c.sieve` will be found in `o.sieve`.

Program files

The result of linking the compiler versions of the source files given on the command line with any libraries needed is an executable program file. This is named `@.file1`, where `file1` is the name of the first source file given on the command line. This convention may be overridden using the `-o` flag.

Compilation list files

If the `-list` keyword is specified, a file containing a compilation listing for each compiled source file is created in the directory `@.l` (6). Thus the command

```
*cc -list c.sieve
```

will result in the list file `l.sieve` being created.

Assembly list files

If the `-S` flag is used, no object code is generated. Instead, an assembly listing of the code is created in directory `@.s` (7). Thus

```
*cc -S sieve.c
```

will result in the 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.

Compiler options

You can control many aspects of the compiler's operation by appending options to the command `cc`. All options are prefixed by a minus sign `-`.

Options come in two forms. The first are keywords. These are multiple-character options and control Acorn or RISC OS-specific aspects of the compiler. Keywords are recognised in upper or lower case. The second form of option is the flag. A flag is a single letter. The case of the letter is usually unimportant to the C compiler under discussion. However, UNIX compilers only recognise one form (either the upper- or lower-case one, depending on the flag in question) and this one should be used in preference.

The case of flags is most important when you are considering porting 'make' files to other systems. By using the 'universal' conventions of the `cc` command, you can move your system to different environments with the minimum amount of work.

The keyword options are:

`-help`	Give a description of the compiler's command syntax.
`-arthur`	Add the (obsolescent) Arthur interface library to the list of 'standard' libraries passed to the linker. This is only valid under the Arthur and RISC OS operating systems.
`-pcc`	Compile 'portable C compiler' 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 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 entitled *Portability* for more details.
`-fussy`	Be extra strict about enforcing conformance to the ANSI standard or to pcc conventions. For example, `-fussy` turns off the pre-definition of `ARM` and `arm` by the preprocessor in ANSI mode.
`-list`	Create a listing file. This consists of lines of source interleaved with error and warning messages. Finer control over the contents of this file may be obtained using the `-f` flag (see below).

The flag options are listed below. Some of these are followed by an argument. Whenever this is the case, the 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 the syntax given lists only the form that would be acceptable to a UNIX C compiler. Similarly, only the case of the letter that would be accepted by a UNIX C compiler is shown.

The descriptions are divided into several sections, so that flags controlling related aspects of the compiler's operation are grouped together.

Controlling the linker

`-c`	Do not perform the Link step. This merely compiles the source program(s), leaving the object file(s) in the `o` directory. This is different from the `-C` option described below.

`-l` *`libs`* This specifies a library which will be used in the Link command issued by the compiler. The library to be used follows the flag (with optional white-space between the flag and the list). Multiple `-l` flags may be used to specify more than one library, or a list of library names joined by commas may be given as an argument to one `-l` flag. The libraries given by this option are used instead of the standard one, not in addition to it.

This flag is not compatible with the corresponding UNIX C compiler option, which has no direct equivalent under RISC OS.

Controlling the preprocessor

The compiler's preprocessor handles the include files given by the `#include` directives at the start of your source files. These are of the form

`#include <`*`filename`*`>`

or

`#include "`*`filename`*`"`

The way in which the compiler 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 `-I` and `-j` flags (including the special filename `:mem`).

For maximum portability, only the forms `#include <name.h>` and `#include "name.h"` should be used. Use of `h.name`, though more natural under RISC OS, is not portable.

Rooted filenames

A filename is rooted if it is

- a RISC OS filename beginning with a `$` or an `&`. For example:

```
$.RISC_OSlib.h.baricon
&.h.myheader
```

- a UNIX filename beginning with a `/`. For example:

  `/RISC_OSlib/baricon.h`

- an MS-DOS filename beginning with a `\`

  `\library\baricon.h`

Rooted filenames are used as written (except that UNIX-style and MS-DOS-style filenames are first translated to equivalent RISC OS filenames as described below).

If *`filename`* is not rooted in the sense described above, then the compiler looks for it in a sequence of places (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	(not for "*`filename`*")
2	the 'current place' (see next page)	(not for <*`filename`*>)
3	arguments to the `-I` flag, if used	
4	the system search path:	

- the path given as an argument to the `-j` command line flag (see below), or
- the value of the system variable C$Libroot if this is set and there is no `-j` flag; otherwise
- `$.Clib.`

If `-j` is used, the in-memory filing system is omitted for `#include` <*`filename`*>. It can be reinstated by giving the pseudo-filename `:mem` to a `-I` flag or to the `-j` flag.

Include syntax

Placing the filename in the #include directive between angle brackets indicates that the file is a 'system' file, that is, a header for the C library (eg `<kernel.h>`).

Placing the filename in the #include directive between double quotes indicates that the file is a 'user' file, that is, a header for the RISC OS library, or one of your own include files.

This reflects the search path used by the compiler in each case. As shown in the *Search path* section above:

- for `<filename>` the search path (in order) is 1, 3, 4.
- for `"filename"` the search path is 2, 3, 4.

In both cases, to facilitate the porting of code from UNIX and MS-DOS to RISC OS, UNIX-style and MS-DOS-style filenames are translated to equivalent RISC OS-style filenames. For example:

`../include/defs.h`	is translated to	`^.include.h.defs`
`..\cls\hash.h`	is translated to	`^.cls.h.hash`
`includes.h`	is translated to	`h.includes`

but

`system.defs`	is translated to	`system.defs`

(In the same way, the lists of directory names given as arguments to the compiler's `-I` and `-j` command-line flags (see below) are translated to RISC OS format before being used).

The current place

The current place is the directory containing the source file (C source or #included header) currently being processed by the compiler. Often, this will be the current 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 compiler option -fK, 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 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, 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 with -I and -j

You can use the -j flag to provide your own system search path, as mentioned in item 4 of *Search path*, above. The compiler will then use the argument you give to the -j flag 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 -j option also removes the in-memory filing system from the front of the path searched for #include <*filename*>. It can be reinstated by using the pseudo-filename :mem as an argument to an -I or -j flag. 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$Libroot is an environment variable that you can use to provide your own system search path, as shown at the end of the section entitled *Search path*. The compiler will use the value of C$Libroot, if set, 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 * prompt type:

```
*set C$Libroot $.mylib
```

This variable is also used by the C compiler 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`.

If you do set the value for C$Libroot and you are using AMU makefiles, you will need to alter the LIB argument in the makefile. If you have set C$Libroot to `mylib`, as in the example above, you should use

```
LIB = $.mylib.o. stubs
```

in the makefile.

Preprocessor flags

`-I`*`path`* — This adds the specified directory to the list of places which are searched for include files (after the in-memory or source file directory, according to the type of include file). The directories are searched in the order in which they are given in `-I` options. The path should end with the name of a directory, with no `.h.`, which is added automatically.

`-j` *`dirs`* — This overrides the system include path with the list of directories which follows the flag. The directories are separated by commas. 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`.

`-j` is an Acorn-specific flag, and therefore non-portable.

`-E` — If this flag is specified, only the preprocessor phase of the compiler is executed. The output from the preprocessor is sent to the standard output stream. It can be redirected to a file using the stream redirection notations common to UNIX and MS-DOS (eg `cc -E c.something > rawc`). By default, comments are stripped from the output, but see the next flag.

`-C` When used in conjunction with `-E` above, retains comments in preprocessor output. It is different from the `-c` flag, which is used to suppress the link operation.

`-M` If this flag is specified, only the preprocessor phase of the compiler is executed (as with `cc -E`), and the only output produced is a list of makefile dependency lines suitable for use by the Acorn Make Utility (AMU). For example,

```
o.amu : c.amu
o.amu : $.clib.h.kernel
```

on the standard output stream. This can be redirected to a file or concatenated to a file using standard UNIX/MS-DOS notation. For example:

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

`-zp`*`mod`* This flag can be used to emulate `#pragma` directives. The *`mod`* which follows it is the same sequence of characters that would follow the directive. See the section *`#pragma` directives* in the chapter entitled *Machine-specific features* for details.

`-D`*`sym`*`=value` Define *`sym`* as a preprocessor macro, as if by a line

```
#define sym value
```

at the head of the source file.

`-D`*`sym`* Define *`sym`* as a preprocessor macro, as if by a line

```
#define sym 1
```

at the head of the source file.

`-U`*`sym`* Undefine *`sym`*, as if by a line

```
#undef sym
```

at the head of the source file. This may be used to cancel the effect of otherwise predefined symbols, eg `ARM`. (A macro `ARM` is predefined, and has the value 1 unless `-fussy` is specified).

Controlling code generation

The options described in this section control the production of code by the compiler.

`-g`*`mods`*	This flag is used to specify that debugging tables for use by the Acorn Source-level Debugger should be generated. It is followed by an optional set of letters which specify the level of information required. No modifiers means 'generate all the information possible'. However, the tables can occupy large amounts of memory so it is sometimes useful to limit what is included as follows:
`-gf`	Generate information on functions and top-level variables (outside functions) only.
`-gl`	Generate information describing each line in the file.
`-gv`	Generate information describing all variables.
	The modifiers may be specified in any combination, eg `-gfv`.
`-o` *`file`*	This flag specifies the name of a file in which the output of the linker should be stored. It overrides the default, which is to use the root name of the first source file mentioned on the command line.
`-p[x]`	This flag causes the compiler to generate code to count the number of times each function is executed. If `-px` is specified, the compiler also generates code to count how often each basic block within 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. This should be done 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 (supplied on Disc 2 of this release). 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 an #include file. However, this is rare and usually causes no confusion.

Provided you didn't compile your program with the `-ff` option, blocks of counts will be interspersed with function names. In the simplest case, for example, such as

```
cc -p c.myprog $.clib.o.ansilib
```

the output reduced to a list of line-pairs like

```
function
    lineno: count
```

where *`count`* is the number of times *`function`* was executed.

If you used `cc -px`, 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 `-p`, Dhrystone 1.1 runs at about 5/8 speed; when compiled `-px` it runs at only about 3/8 speed.

There is no way, in this release of C, to relate execution counts to the amount of 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.

`-S` If this flag is specified, no object code is generated and, naturally, no attempt is make to link it. Instead, an assembly listing of the code produced is written to a file called `s.file`, where `file` is the name of the source file (stripped of any directories or suffixes).

Controlling warning messages

The `-w` option controls the suppression of warning messages. 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 they may be disabled.

`-W`*`mod`* If no modifier letters *`mod`* are given, then all warnings are suppressed. If one or more letters follow the flag, then only the class of warnings controlled by those letters are suppressed. The letters are:

`a` Give no `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.

`d` Give no `Deprecated declaration foo() - give arg types` warning. Use of old-style function declarations is deprecated in ANSI 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.

`f` Give no `Inventing 'extern int foo()'` message. This may be useful when compiling old-style C as if it were ANSI C.

`n` Give no `Implicit narrowing cast` warning. This warning is issued when the compiler detects an assignment 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.

`v` Give no `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.

Controlling additional compiler features

The `-f` flag described in this section controls a variety of compiler features, including certain checks more rigorous than usual. Like the previous flag it is followed by modifier letters. At least one letter is required.

`a` 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 been assigned a value.

`c` Enable the 'limited pcc' option. This allows characters after `#else` and `#endif` preprocessor directives (treated as comments), and explicit casts of integers to function pointers (forbidden by ANSI). These 'features' are often required in order to use pcc-style include files in ANSI mode.

`e` Check that external names used within the file are still unique when reduced to six case-insensitive characters. Some 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 allows 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 `-list`) 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 (the 'current place' is defined by the original source file and is not stacked; see the earlier section *Controlling the preprocessor* for details).

`m` Give a warning for preprocessor symbols that are defined but not used during the compilation.

`p` Report on explicit casts of integers into pointers, eg

```
char *cp = (char *) anInteger;
```

Implicit casts are reported anyway, unless suppressed by the `-wc` option.

`u` By default, the source text as 'seen' by the compiler after preprocessing (expansion) is listed. If `-fu` 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 `-fu` specified, as `p=NULL;`.

`v` Report on **all** unused declarations, including those from standard headers.

`w` Allow string literals to be writeable, as expected by some UNIX code, by allocating them in the program's data area rather than the notionally read-only code area.

`x` Turn on additional warnings about:

- use of short integers
  Shorts are slower than longs on the ARM and cause more code to be generated. They should only be used to save space in large arrays of data.
- use of enums
  ANSI defines enum values to be integers so the use of enums is not strictly type-checked. In some dialects of C, enums are more strictly type-checked than this.

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

Compiling and linking

As already shown, to compile and link the simple example program shown above you would type:

```
*cc HelloW
```

This produces the executable program `HelloW`. To produce a program with a different name, you would use the `-o` option, eg:

```
*cc -o greeting HelloW.c
```

This time the linker would produce a program that you could run using the command `*greeting`.

When writing programs that use several source files, you may want to compile them selectively and perform the link as a separate step. For example, if a program consists of the files `e1.c`, `e2.c` and `e3.c`, and you have just edited `e2.c`, you may want to compile this, then link the object file with the two other files:

```
cc -c e2.c
link -o expr o.e1 o.e2 o.e3 $.clib.o.stubs
```

Alternatively,

```
cc -o expr e1.o e2.c e3.o
```

does the trick!

See the linker chapter for more details on linking. To maintain complex, multi-file programs, consider using the Acorn Make Utility, which is supplied with C Release 3, and described in a later chapter of this Guide.

To compile several different source programs and link them all together into one executable file, list all the filenames separated by spaces. The name of the executable program is taken from the first filename given unless `-o progname` is used.

For example, the command:

```
cc mainprog util extra
```

compiles the sources `c.mainprog`, `c.util` and `c.extra` into the object files `o.mainprog`, `o.util` and `o.extra`, and then links all three object files together with the standard library to produce the executable program `mainprog`.

Installation

The release discs

Note first that you might find files called `ReadMe` on one or more of the Release discs. These contain any information that was not available when this Guide was being prepared, or that has since been modified; to inspect their contents, load them into Edit or use *Type from the Command Line * prompt.

Disc 1: Work disc

You may plan to install and run the C compiler on a hard disc or Econet network. However, Disc 1 contains all you need to compile, link and run the examples provided.

Of course, if you are using a single floppy disc drive, you will be using Disc 1 (or a copy of it) as your working disc all the time.

The contents of Disc 1 are:

Directories:

`$.AsdDemo`	Acorn Source-level Debugger demonstration
`$.Clib`	Support for the C library
`$.DeskEgs`	Desktop Application examples

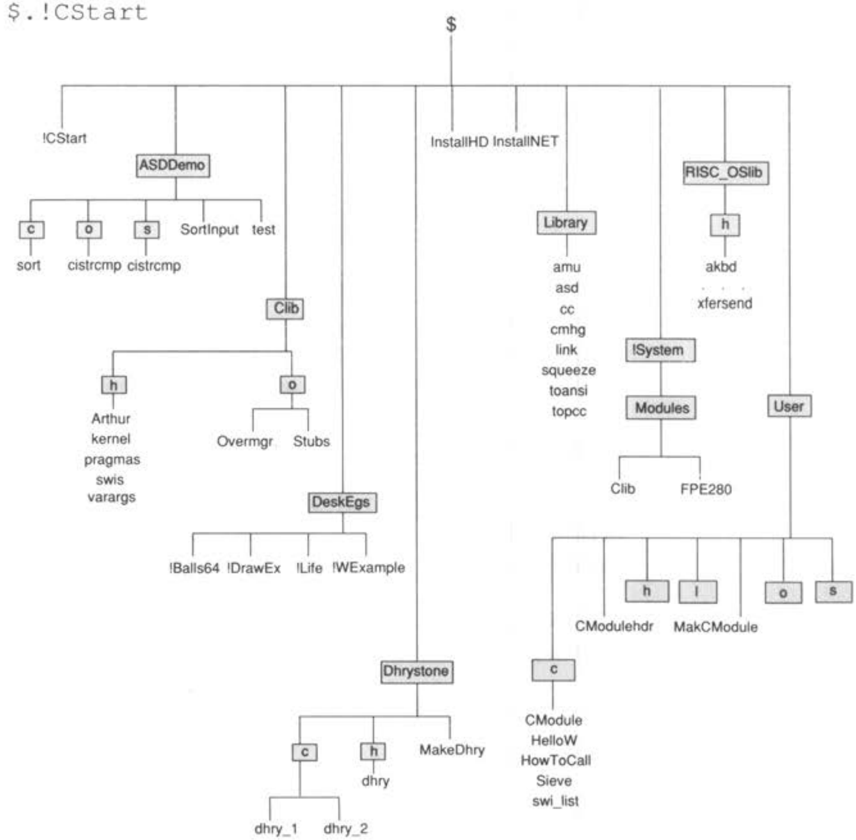
`$.Dhrystone`	Source for the Dhrystone 2.1 benchmark
`$.Library`	Library directory for the compiler and tools
`$.!System`	Relocatable Modules
`$.RISC_OSlib`	Condensed headers for the RISC OS library
`$.User`	Default working directory, containing examples

Installation utilities:

`$.installHD`	for installation on a hard disc
`$.installNET`	for installation on an Econet network

Obey file:

`$.!CStart`



Disc 2: Library support disc

Because of lack of space, the only library on Disc 1 with which you can both compile and link is the shared C library. You can compile with the RISC OS library there, but not link with it. Disc 2 provides full library support for the other libraries.

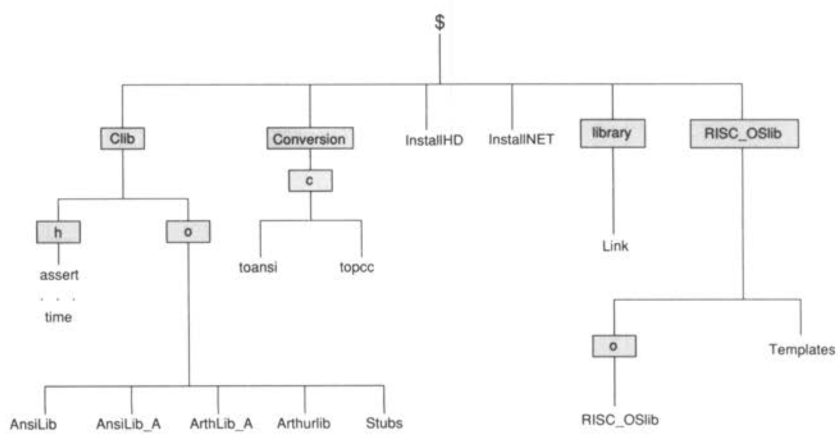
The contents of Disc 2 are:

Directories:

`$.Clib.h`	Condensed headers for the C library
`$.Clib.o`	Binaries of the standalone C library and Arthur library; copy of `Stubs` for linking with RISC_OSlib
`$.Conversion`	Source for the conversion utilities (`toansi` and `topcc`)
`$.Library`	contains a copy of the linker
`$.RISC_OSlib`	Binary of the RISC OS library

Installation utilities:

`$.installHD`	for installation on a hard disc
`$.installNET`	for installation on an Econet network



Disc 3: Reference Disc

To conserve space, the RISC OS headers on Disc 1 and the C library headers on Disc 2 have been condensed, with all the comments stripped out. Disc 3 provides the fully commented headers as on-line reference documentation.

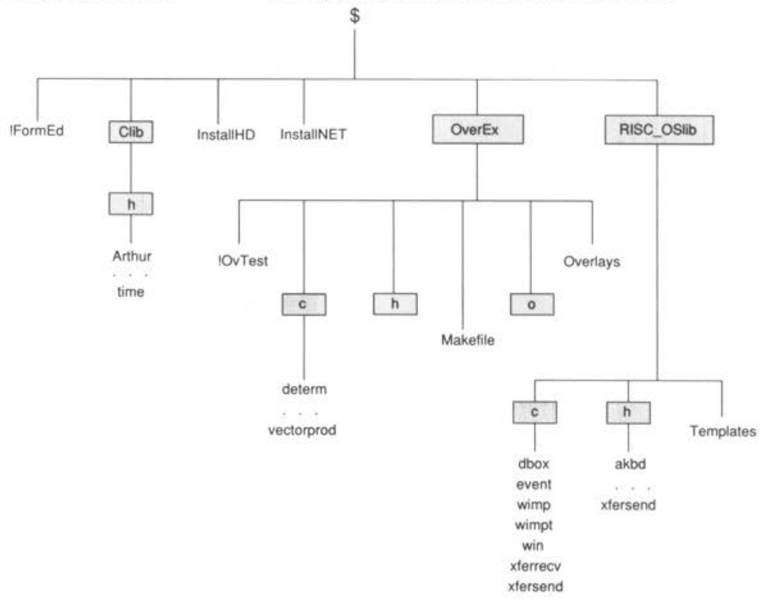
The contents of Disc 3 are:

Directories:

`$.!FormEd`	Template editor utility
`$.Clib`	Fully commented C library headers
`$.OverEx`	Example overlay system
`$.RISC_OSlib`	Fully commented RISC OS library headers, and in subdirectory `c`, sources that you may wish to customise.

Installation utilities:

`$.installHD`	for installation on a hard disc
`$.installNET`	for installation on an Econet network



Using a single floppy drive system

You can compile, link, and run programs on Disc 1, linking with the shared C library.

To use the RISC OS library on a single floppy disc, you should compile on Disc 1, transfer the object file to Disc 2 via the RAM filing system (as described in the *User Guide* supplied with your computer), and then link on Disc 2 which holds the RISC OS library binary.

Similarly, you can use the standalone C library (Ansilib) and Arthur OS library (Arthurlib) by compiling on Disc 1 and linking on Disc 2. Both versions (conforming to the old and the new ARM procedure call standards respectively) are included. The versions conforming to the old standard are identified by having the suffix `_A` added to their filenames.

The headers for the RISC OS library on Disc 1 and for the C library on Disc 2 are condensed – stripped of comments – to economise on space. You only need access to these if using the `-I` or `-j` compiler options: by default, the compiler uses the inbuilt ANSI headers, which is why only machine-specific headers need to be included on Disc 1.

You can create more working space on Disc 1 by getting rid of the examples and unneccessary or infrequently-used material. Make sure you have backed up the original before doing this!

Candidates for removal include:

- The modules Clib 60Kb
  FPE280 24Kb

  Since you need these only once for each session using C (ie before you start using the compiler), you could keep them on a separate disc, as part of a `!Boot` start-up to configure your environment.

- `AsdDemo` 4Kb
- `Clib.h.Arthur` 17Kb

  This could be removed if you are not linking with the Arthur library, which has only been included for backwards compatibility, and will not be supported in the future.

- `Clib.o.overmgr` 1Kb

  This can be removed if you are not using overlays in your program.

- `DeskEgs` 63Kb

- `Dhrystone` 36Kb
- `InstallNET` and `installHD` 16Kb
- `Library.CMhg` 9Kb

  This can be removed if you are not writing a relocatable module.
- `Library.Squeeze` 8Kb

  You will only be squeezing image files once they have been developed, and thus need to use `squeeze` only occasionally.
- `User` all the example material 54Kb

Using a twin floppy drive system

With a twin floppy disc drive system, there will be less need for swapping discs. Keep your work disc in drive 0 and use drive 1 for compilation (using Disc 1) and linking (using Disc 2) in two separate stages. You will need to reset your library to `:1.$.User`.

Using a hard disc or network

You have three options for installing the system on a hard disc or Econet network:

- Run the installation utility provided on each disc.
- Modify the installation utility for your particular needs, and then run it.
- Use the copying facilities of the desktop.

Running the installation utilities

The programs `installHD` and `installNET` are provided on each floppy disc to enable you to install C on a hard disc and Econet network respectively. They can be run by clicking the appropriate icon from the desktop.

They install a sensible default directory structure, and prompt where overwriting would occur to guard against inadvertently writing over an existing object of the same name. If you choose not to overwrite an existing object, the installation will abort at this point.

If you already have a previous release of the C compiler product installed, you should back this up before installing the new system. Then you can overwrite existing components with the new versions, knowing that you have your previous system intact should you need it.

To remove installed directories, at the Command Line type

```
*wipe directory vfr
```

Modifying the installation utilities

You may wish to organise the material differently from the default directory structure installed by the utilities.

You can do this by editing the installation program; instructions for this are available as an option within the program (run the installation program and select the **Further information** option from the main menu).

Using the desktop

The *User Guide* supplied with your computer tells you how to copy files using the desktop.

All the facilities you will need are available, including the provision for altering access if required. You can also set up the copy options to suit your requirements. For example, using the **Confirm** option (which can be set from the Filer menu) will allow you to choose to overwrite existing objects.

Duplications on the release discs:

Linker	The linker is provided on Disc 2 as well as Disc 1.
Library headers	Condensed library headers for the libraries have been provided on Disc 1 to economise on space. These are:

`$.CLib.h.Arthur`	for the Arthur OS library (Arthurlib)
`$.Clib.h.kernel`	machine specific components of the
`$.Clib.h.pragmas`	ANSI library (sharedClib or Ansilib)
`$.Clib.h.swis`	A complete list of RISC OS SWI definitions
`$.RISC_OSlib.h.`	for the RISC OS library

Likewise, the full set of headers for the ANSI library (as well as the Clib.h entries on Disc 1) are provided in condensed form on Disc 2. The standard ANSI library headers are only needed if you compile with the `-I` or `-j` options: by default, the compiler uses its own inbuilt headers for the standard library.

Since the condensed headers have been stripped of all comments, they can be used for compilation, but they do not provide on-line documentation. For this reason, the fully commented headers are provided on Disc 3.

You will need to evaluate the best configuration for the libraries for your particular system and needs. If space is not at a premium, installing the commented headers has the advantage of offering on-line documentation.

The installation utilities provided install everything on each release disc (apart from the utilities themselves). Thus, if you install Disc 1 first, and then install Disc 3, you will be prompted to confirm overwriting the condensed RISC OS library headers from Disc 1 with the fully commented headers from Disc 3.

Setting up your working environment

All the basic tools for developing software in C are provided with the Acorn C compiler product:

•	the compiler		`cc`
•	libraries	shared C library	`clib`
		RISC OS library	`RISC_OSlib`
•	linker		`link`
•	debugger	Acorn Source Level Debugger	`asd`
•	'make' utility	for management of multiple source files	`amu`

These are augmented with the following specialist utilities:

- `CMhg` utility for packaging relocatable modules
- `toansi` `topcc` for converting source files between pcc and ANSI style C
- `!FormEd` template editor for components of the desktop user interface
- `squeeze` image compaction utility

Use the default directories for the compiler libraries, and your library directory for the commonly used tools. This is as provided on the release discs, and installed on hard disc or network by the installation utilities.

FormEd is an application, and as such has all the resources for the template editor in the `!FormEd` directory. If you have a single floppy disc, your storage space will be limited, so keep FormEd on a different disc from your working disc and prepare the desktop components there for incorporation into your desktop applications.

Editors

The options available to you for writing your source code are:

- Edit
- Twin
- editors produced by other software companies.

Edit is a general purpose editor supplied with your RISC OS computer, and works as a multi-tasking application in the desktop. The *User Guide* supplied with your computer describes how to use it.

Edit also provides Task windows, enabling you to run the compiler, for example, as a task in the desktop.

Twin is a specialist editor for developing software, and is available as an Acorn product from your dealer. Twin works from the Command Line and will not function in the desktop; however, it offers many powerful features for developing source code.

Reference manuals

In addition to this manual, you will need to use the *User Guide* supplied with your RISC OS computer. This provides much of the information you need to make use of the facilities of RISC OS to aid in software development.

For serious software development, you will need the *RISC OS Programmer's Reference Manual* which provides you with all the details you need to write code which accesses RISC OS.

Hardware specification

The key limits to your working environment are the memory size and storage media (floppy disc, hard disc or Econet network) available to you.

The minimum specification of an Acorn RISC OS computer system is:

- RAM 1Mb
- storage 1 floppy disc drive

The description that follows applies to this specification, but if you have more memory and storage capacity, you will have more flexibility.

There are three formats available to you for floppy discs:

L	640Kb	compatible with Master Compact
D	800Kb	compatible with Arthur
E	800Kb	

Use E format whenever possible: it has the advantage that it is not necessary to compact the disc to reclaim space left by deleted files, and it can cope with defects on formatting.

The limitation of a single floppy drive system is that you can't compile and link with the RISC OS library on a single disc. Instead, you can compile on one disc, which holds the RISC OS library headers, transfer the object using the RAM filing system, and link on a second disc holding the RISC OS library binary.

This is set up for you on release Disc 1 and Disc 2, and illustrated with the desktop examples in the Examples section.

Ways of organising an economical work disc that gives you the maximum space are explored in the *Installation* section. In addition, you can use the Squeeze utility to reduce the size of executable programs (it will reduce the size of a file by a factor of the order of 2), and this has been done for the tools provided.

If you have two floppy disc drives, you will be able to compile and link with the RISC OS library without needing to swap discs.

Hard disc

For a major software development project, a hard disc provides much more storage capacity, and you can install everything provided on the release discs, including the commented headers for the C and RISC OS libraries to provide on-line reference documentation.

Network

A workstation on an Econet network has access to mass storage which overcomes the capacity limitation of a floppy disc system. However, since increasing demand slows down access speed over a net, you will need to balance this against the advantages of exceeding the storage limits of local disc drives.

A possible strategy is to use the network medium for shared resources, particularly if they are infrequently used, and use the local workstation floppy drive for the individual user's private files (for example, their source, object and image files).

Using C on a RISC OS computer

There are two ways in which you can interact with the system: in the desktop or in Command Line mode.

By working within the desktop, you can use the desktop facilities, but these consume memory, leaving less for the C compiler system and support tools.

Closing down the desktop and working in Command Line mode will provide you with the maximum memory on your system for working with the C compiler.

For software development that runs to a number of source files, use the Acorn Make Utility to drive the compiler, rather than invoking `cc` directly.

Working in the desktop

You can use Edit in the desktop to develop your source code. The compiler and tools are driven from a Command Line, and there are two ways of reaching this from the desktop. You can leave the desktop temporarily – by pressing F12 or selecting *** Command** from the Task Manager menu – or more permanently, by selecting **Exit** from the same menu.

The first of these methods, as illustrated with the HelloW example at the start of this chapter is adequate for demonstration purposes. However, this has the disadvantage that the desktop is still using memory, though you cannot use its facilities.

Edit task windows

Edit provides you with Edit Task windows, which give you a Command Line within a window. Details of Edit Task windows are provided in the *User Guide*.

Memory permitting, you could, for example, have three windows on the screen:

- your source code in an Edit window
- the compiler driven from an Edit task window (using `cc` or `amu`) with all the compiler messages for you to scroll through

- the Acorn Source-level Debugger driven from another Edit task window with all the ASD interactions and output displays.

Further possibilities

You can drive the compiler and tools using your own Obey and Command files, which are easy to invoke from the desktop by clicking on the appropriate icon in a directory display.

Another easy-to-use facility is Tinydirs, which gives you instant access to a directory or application. This is described in the *User Guide* supplied with your computer.

Provided that you have the application memory available, you can use any desktop applications to support development.

Finding more memory

Monitoring and managing memory within the desktop is made easy with the Task manager, described in the *User Guide*.

The desktop environment is already highly economical in its use of RAM resources, so there is little opportunity for saving more. If you are short of space, install only the applications that you need when using the Acorn C system.

Possibilities for increasing the memory available to you in the desktop include:

• sound modules (24Kb)	you could *Unplug these
• system sprite size	trim down to 0 (you don't need this)
• font cache, RAM disc	trim down to 0 if not needed
• for ADFS	juggle the buffering size
• screen size	reduce to the minimum you require (use Mode 0 or 11)

If you still don't have enough space, leave the desktop, and work from the Command Line. Select the **Exit** option from the Task manager menu to enter Command Line mode. From there you can return to the desktop with the *Desktop command.

Any relocatable modules that are installed from disc, and are superflous to your requirements, can be removed from the RMA area. You need only the shared C library; it is possible to use the compiler and tools without the floating point emulator (though you will not be able to run any programs you write in C which do floating point operations).

At the * prompt, typing `modules` will list all ROM-based modules as well as those currently resident in the RMA. The hexadecimal values given for each ROM-based module indicate the workspace requirement, so any which do not display a string of zeros and are superfluous to your requirements are candidates for making economies. You can use the command *RMKill to disable them, and then *RMTidy to make the space available. When you switch on, the modules will be reinstated and their workspace initialised, but you can use the *Unplug command to get rid of them on a more permanent basis (until you reconfigure the CMOS RAM, either using *RMReinit or with a Delete-power on, which initialises all the default settings).

A useful measure of application space available in Command Line mode can be obtained by entering BASIC (type *BASIC at the * prompt). Since BASIC uses about 4Kb of workspace, adding this to the number of bytes free indicated when BASIC is initialised will give you a measure of what is available for working with the C compiler.

A better measure is given by subtracting (32Kb + 4Kb) from the value of BASIC's pseudo-variable HIMEM. At the BASIC prompt, type

```
PRINT (HIMEM - 36*&400)/&400
```

to get the value in Kb.

↓ ↓ ↓ RMA ↓ ↓ ↓	squeezes down on application workspace
	4Kb BASIC workspace
	HIMEM
	application area
	32Kb reserved for the operating system

You can use Twin in Command Line mode for developing your source files, and this offers some special facilities for accommodating large files. Twin does not work like BASIC as regards memory usage, so you cannot measure the space available to you in the same way.

Memory vs disc space

On a 1Mb system with a hard disc, RAM is the limiting resource, while with a 2Mb system and a single floppy disc drive, it is the disc space that is the critical resource.

In the latter case, you can use the extra memory to install the development tools, thereby releasing space on the disc. To do this, you can load tools – such as the compiler itself – into RAMFS and invoke them from there.

Writing desktop applications

If you are developing a desktop application, a convenient way of organising your work is to use the application directory as your current directory.

For example, `!myapplic` will contain the application components `!boot`, `!run`, `!runimage`, `sprites` and `templates`, and in addition, directories `c` for your source files, `h` for your headers (if required) and `o` for your object files.

Starting

You can use Obey or Command files for setting up your working environment at the start of each session when you use the compiler system. The Obey file `!Cstart` is provided on Disc 1 as an example. You can read and edit Obey files using Edit. Double-click on the Obey file in a directory display to run it.

Shared C library

The shared C library provided with Release 3 of the C compiler is version 3.50. This is backwards compatible so that existing software will run with it. However, software compiled with the Release 3 compiler will not work with the old shared C library.

C programs are linked not with the C library but with a small piece of code and data called `stubs`. 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. Use of the shared C library

- saves space on disc
- 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)
- typically costs less than 30Kb of memory (the shared C library plus stubs occupy about 65Kb, whereas most C programs include about 40Kb of Ansilib).

Without the shared C library, it would not be possible to pack so much into this release of C.

The old shared C library

When an application is run which uses the shared C library, the application needs to know where the library module is in memory, so that it can locate the library routines when required. You will encounter a problem if, on first opening your C Release 3 product, you have already installed an application (such as Edit) and then work through the HelloW example. If you now resume using the application, or return to the desktop, it will crash.

This is because the application installed an old version of the shared C library (for example, look at the `!Run` script of the Edit application). When the later version was subsequently installed in the RMA, it will have replaced the old version but the functions in it will not be installed at the same address. As a result, Edit is left pointing at a C library that is not there any more. This eventuality is guarded against in the `!Cstart` example, which will stop with the `!!!Old shared C library!!!` error report.

If this happens, you can proceed by quitting all current applications; then get rid of the old C library by typing (at the Command Line prompt)

```
RMKill SharedCLibrary
```

You will then be able to run `!CStart` successfully.

The long-term solution is simple: replace the old C library in your `!System.modules` directory with the new one (this is where applications look to install modules they need), and ask all users of your software that you have written in C to do the same.

Compiling and running the example programs

The following example programs are to be found in the directory `adfs::C3disc1.$.user`, unless otherwise stated. For each program, we give a 'recipe' for how to compile, link and run the program. If you have a machine with a single floppy disc drive, and 1Mb of RAM, you will need to clean up after running each example. It is assumed that you have read the preceding parts of this chapter.

HelloW

Purpose:	The standard most trivial C program. Try it as an exercise in getting going.
Source:	`c.HelloW`
Compile using:	`cc HelloW`
Run using:	`HelloW`
Clean up using:	`remove HelloW` `remove o.HelloW`

Sieve

Purpose:	The Sieve of Eratosthenes is often presented as a standard benchmark, though it is not very meaningful in this context.
Source:	`c.sieve`
Compile using:	`cc sieve`
Run using:	`sieve`
Clean up using:	`remove sieve` `remove o.sieve`

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 the Acorn Make Utility (AMU).

Location: `adfs::C3disc1.$.dhrystone`

Sources:

```
h.dhry
c.dhry_1
c.dhry2
```

Makefile: `MakeDhry`

Build using: `amu -f MakeDhry`

Run using:

```
dhry2
dhry2reg
```

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 using: `amu -f MakeDhry` again (try altering some of the options in MakeDhry between rebuilds: eg compile in –pcc mode or link with AnsiLib instead of stubs).

Clean up using: `amu -f MakeDhry clean`

SWI_list

Purpose: To illustrate use of the SWI facilities in `<kernel.h>`. You can also try it as an exercise in getting going; later, you can use it to check that `$.CLib.h.swis` contains a complete list of the SWI names and numbers relevant to your machine.

Source: `c.SWI_list`

Compile using: `cc SWI_list`

Run using: `SWI_list > h.myswis`

	Test using:	see instructions embedded in the comment at the head of `c.SWI_list`

	Clean up using:	`remove SWI_list`
		`remove o.SWI_list`
		`remove h.myswis`

HowToCall	Purpose:	To illustrate how to call other programs from C. Read the source, then experiment with the binary. You can also use it as another exercise in getting going. Try making your own makefile for it as an exercise in using AMU.
	Source:	`c.HowToCall`
	Compile using:	`cc HowToCall`

	Run using:	`HowToCall 3`
		`HowToCall HowToCall 2`
		`HowToCall 3 *`
		`HowToCall 3 *` etc

	Clean up using:	`remove HowToCall` `remove o.HowToCall`
CModule	Purpose:	To illustrate how to implement a module in C. You can also use it as another exercise in using AMU.
	Sources:	`c.CModule CModuleHdr`

	Build using:	`cc -zM -c c.CModule`
		`cmhg CModuleHdr o.CModuleHdr`
		`Link -o cmodule -rmf o.CModule o.CModuleHdr $.CLib.o.Stubs`
		or
		`amu -f MakCModule`

	Run using:	`cmodule`

Test 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 using:

```
rmkill TestCModule
remove cmodule
remove o.CModule
remove o.CModuleHdr
```

or

```
amu -f MakCModule clean
```

Example programs for use under the desktop

The example programs which illustrate how to write applications under the desktop, using the RISC OS library, are to be found in the directory `adfs::disc1.$.DeskEgs`. The instructions given assume that you have a configuration of one floppy disc drive and 1Mb of RAM. In such circumstances, it will be necessary to compile the programs on Disc 1, and then use the RAM disc facility to transfer the example directories to Disc 2 for linking with `RISC_OSLib`. Again, you will need to clean up after each example if you have such a configuration. It is also assumed that you have invoked the desktop using *Desktop. If you have a hard disc, omit the RAM transfer steps described below.

To display the contents of an application directory on the desktop, hold Shift while clicking on the directory icon.

When you have explored these examples, refer to the later chapter *How to write desktop applications in* C if you want to go further.

The first example below sets out the exact steps you should take. Then follows a schema for the remaining examples.

WExample

Purpose:	To illustrate installing an icon on the icon bar, and creating/displaying a simple window.
Source:	`c.Wexample`

Build using:	From the command line type the following:
	`*dir adfs::disc1.$.DeskEgs.!Wexample`
	`cc -c Wexample -I$.RISC_OSLib`
	(Press Return to go back to the desktop)
	Allocate at least 48Kb of RAM disc from the Task Manager display
	Click Select on the RAM icon
	Drag the `!Wexample` directory from the Disc 1 directory display onto the RAM directory display
	Remove Disc 1, Insert Disc 2
	Click Select on the :0 icon
	Drag the `!Wexample` directory from the RAM directory display to the Disc 2 directory display
	Press F12 to return to the command line
	`*dir adfs::C3disc2.$.!Wexample`
	`*lib adfs::C3disc2.$.library`
	`link -o !RunImage o.Wexample $.RISC_OSLib.o.RISC_OSLib $.Clib.o.Stubs`
	`squeeze !RunImage`

Run using:	Press Return to go back to the desktop Double-click Select on the `!Wexample` icon from Disc 2 Click Select on the EG icon to get a window

Clean up:	Press F12 to return to the command line
	`*dir adfs::C3disc2.$`
	`*wipe adfs::C3disc2.$.!Wexample ~cfr`
	`*wipe ram:$.* ~cfr`

Schema

Purpose: General schema for compiling, linking and running the remaining desktop examples.

Source: `c.`*`filename`*

Build using: From the command line type the following:

```
*dir adfs::disc1.$.DeskEgs.!filename
cc -c filename -I$.RISC_OSLib
```

(Press Return to go back to the desktop)
Allocate at least 48Kb of RAM disc from the Task Manager display
Click Select on the RAM icon
Drag the *!filename* directory from the Disc 1 directory display onto the RAM directory display
Remove Disc 1, Insert Disc 2
Click Select on the :0 icon
Drag the *!filename* directory from the RAM directory display to the Disc 2 directory display
Press F12 to return to the command line

```
*dir adfs::C3disc2.$.!filename
*lib adfs::C3disc2.$.library
link -o !RunImage o.filename
      $.RISC_OSLib.o.RISC_OSLib
      $.Clib.o.Stubs
squeeze !RunImage
```

Run using: Press Return to go back to the desktop
Double-click Select on the *!filename* icon from Disc 2

Clean up: Press F12 to return to the command line

```
*dir adfs::C3disc2.$
*wipe adfs::C3disc2.$.!filename ~cfr
*wipe ram:$.* ~cfr
```

Life

Purpose: To illustrate use of multiple windows in an application, using the alarm facilities of RISC_OSLib and creating icons in a window.

Source: `c.life`

Build, run and clean up:	see schema.

DrawEx

Purpose:	To illustrate loading files by icon dragging, and rendering draw files in a window.
Source:	`c.DrawEx`
Build, run and clean up:	see schema.

Balls64

Purpose:	To illustrate use of a sprite as a 'virtual display', saving files by icon dragging, and responding to 'help' requests. This application requires at least 320Kb of RAM to run, so you may need to quit some applications to make room for it.
Source:	`c.Balls64`
Build, run and clean up:	see schema.

Recompiling the conversion tools

In the directory `adfs::C3disc2.$.conversion.c`, you will find the sources for the conversion tools `toansi` and `topcc`. If you wish to recompile these as a further exercise in using the Acorn C Compiler, you can follow the instructions presented below.

toansi

Purpose: To help convert programs written in pcc-style C into ANSI C

Source: `c.toansi`

Build using: Copy the conversion directory from Disc 2 into a suitably sized RAM disc area (as in the desktop examples)

```
*dir ram:conversion
*cdir o
cc -c toansi
link -o toansi o.toansi $.Clib.o.Stubs
```

Copy the binary for your `toansi` onto Disc 2

Run using: `toansi infile outfile`

	Clean up:	`wipe ram:$.* ~cfr`
topcc	Purpose:	To help convert programs written in ANSI C into pcc-style C
	Source:	`c.topcc`
	Build, run and clean up:	As `toansi`, substituting `topcc` for `toansi` throughout.

Using the Linker

The purpose of the Linker is to combine the contents of one or more object files (the output of a compiler or Assembler) with selected parts of one or more library files, to produce an executable program.

The compiler incorporates a link step by default, linking with the Stubs interface to the shared C library. Use the `-c` option when invoking the compiler to suppress the link step.

Linker Command Line format

The format of the Link command is:

`Link [options] files`

`files` is a list of one or more object files and libraries; this is described later.

Below is a list of the command line options that the Linker can take. In the descriptions below, the important, frequently-used options are given first, followed by the less common ones. The keywords are case-insensitive. Minimum abbreviations are shown before the brackets.

`-h[elp]`	Print a screen of help text
`-o[utput]`	Name of the linked output file
`-d[ebug]`	Include debugger tables in the output image for use by the Acorn Symbolic Debugger
`-rm[f]`	Generate an `rmf` image
`-ov[erlay]` *`file`*	Generate a RISC OS overlaid image as directed by commands in *`file`*
`-via` *`file`*	Use *`file`* to obtain (further) input file names
`-v[erbose]`	Print messages indicating progress of the link operation

`-map`	Create a map of the base and size of each linker area in the output image (especially useful with `-ov[erlay]`)
`-x[ref]`	List references between linker areas (especially useful with `-ov[erlay]`)
`-ai[f]`	Generate an AIF image (the default)
`-ao[f]`	Generate partially linked AOF output suitable for inclusion in a subsequent link step
`-bi[n]`	Generate a plain binary image
`-m[odule]`	A synonym for `-rmf`
`-w[orkspace]` *n*	Reserve *n* bytes of workspace for a relocatable image
`-en[try]` *n*	Set the image's entry point to *n*
`-c[ase]`	Make matching of symbols case insensitive
`-b[ase]` *n*	Set base address for output file to *n*
`-r[elocatable]`	Generate a relocatable output file
`-db[ug]`	(obsolescent) Generate output for use with the Dbug program. Do not confuse this option with `-d[ebug]`.

Notes

- The keywords `-base`, `-workspace`, and `-entry` are followed by numeric arguments. You can use the prefix `&` or `0x` to specify hexadecimal, and the suffixes `k` for 2^{10} and `m` for 2^{20}.
- The default base address for the output file is &8000 (32K). (If the obsolescent `-dbug` is specified, the default base address is &50000, ie 320K).
- The item *`files`* above is a list of one or more filenames, separated by spaces. This part of the command must be given. Each of the files in the list must be in Acorn Object Format (compiled files) or Acorn Library Format (libraries). They may contain references to external objects (procedures and variables) which the Linker will attempt to resolve by matching them against definitions found in other files.
- You can use wildcards in the filename list. Names using wildcards will be expanded into the list of files matching the specification. For example, the name `o.bas*` might yield `o.basmain`, `o.basexpr`, `o.bascmd`.
- Usually, at least one library file will be specified in the list. A library is just a collection of AOF files stored in a single Acorn Library Format file. You can call the procedures in the library for one language from

programs written in another, as long as both languages conform to the ARM Procedure Calling Standard and both run-time libraries use the common run-time kernel. For example, an assembler program could use the C `printf` function, as long as the C run-time system had been initialised.

- Libraries differ from object files in the way the Linker uses them. First, all the object files are linked together. Then for each library in turn, the linker searches for symbol definitions which match currently unsatisfied symbol references. When a library member is loaded, new unsatisfied symbol references may be created, so the library is re-searched until no more members are loaded from it. Note that each library is processed in turn, so references between library members must be ordered. A reference from a member of a later library to a member of an earlier library cannot be satisfied. A *fortiori*, circular dependencies between libraries are forbidden.
- Two common errors occurring during a link step are caused by unresolved and multiple references.
  - In the first case, a symbol has been referenced from a file (whose name is given in the error message), but there is no corresponding definition for the symbol. This is usually caused by the omission of a required object or library file from the list, or the mis-spelling of an external identifier in the original source program.
  - The second error is caused by a clash of names. For example, a procedure might have been defined with the same name as a library procedure, or as a procedure in another object file.

Simple examples

Before we move on to describe the rest of the Link command's options, we give some examples which illustrate the syntax described so far.

```
Link -OUTPUT test.sieve aof.sieve paslib
Link -o %.mybasic o.bas* lib.f77
Link -o null: o.comp*
```

Linker keywords

output

The `-output` keyword is followed by the name of the file to which the final linked program should be written. If the keyword is omitted, the output file defaults to the lower-case name of the output format (eg `aif` for AIF format). If you just want to use Link to check object files for unresolved references, you can specify the device `null:` as the output file; the final object code will be discarded.

debug

The `-debug` (or just `-d`) option instructs the linker to include any input debugging areas in the output image and to append low-level debugging data to it. This allows a program to be debugged using the Acorn Symbolic Debugger (ASD), described in a later chapter of this Guide. Debugging areas include debugger tables generated by high-level language compilers in support of source-language debugging (for example using `cc -g`). Low-level debugging data allows the program to be debugged at the assembly-language level more easily, whether or not its components have been compiled specially.

rmf

The `-rmf` option instructs the linker to generate a RISC OS relocatable module as output. This can only be done if one of the input object files contains a relocatable module header in an AOF area called `!!!Module$$Header`. Such an object can be created using ObjAsm or, more conveniently, using the C Module Header Generator (`cmhg`).

via

Sometimes you may want to link a large number of input files which would be tedious to type on a command line, and whose names can't conveniently be matched by a wildcard specification. Indeed, the length of this list may be longer than the 256 characters allowed for a RISC OS command. To solve this problem, you can store a list of input filenames in another file and then use the `-via` keyword to give the Linker access to them. For example, suppose you created the file `basfiles` with the contents:

```
o.main
o.expr
o.cmd
o.stmnt
```

```
o.lex
o.filing
o.tokens
```

If you then used the command

```
*link -o basic -via basfiles lib
```

the files listed in `basfiles` would be linked, together with the AOF file `lib`.

verbose

If you specify `-verbose` on the command line, the Linker gives a report of its progress. A message is printed as each file is opened and as each module is being relocated. For example:

```
link: opening p.basic
link: opening o.bas1
link: opening o.bas2
link: relocating module o.bas1
link: relocating module o.bas2
link: relocating module ansilib (fpprintf)
...
```

case

If you specify `-case` in the command line, then the Linker will not treat the case of letters as significant in identifiers. By default, the identifiers `main` and `Main` refer to different objects, since they are spelt differently. However, `-case` causes them to be treated as the same identifier.

One occasion when you will want to use this flag is when you are linking a C object file with one from a language such as ISO-Pascal or Fortran-77, both of which are case insensitive. These languages plant symbols in object files in a single case, regardless of how they are spelt in the source file.

base

By default, the base address of the output file of the Linker is &8000. This corresponds to the start of application workspace on all current Acorn computers running RISC OS. Alternatively, if the `-dbug` option is given, the base address is set to &50000. This is so that the (obsolescent) debugger program Dbug can load at &8000 as a normal application, and load the file to be debugged above itself. (There are other changes when `-dbug` is given, as described below.)

Using the `-base` keyword, you can set the base address of the output file to any desired value. For example, you may want a program to have a high load address (as with the `-dbug` option set), but still be directly executable (which a dbug file in AOF format isn't).

The keyword is followed by a number given the base address desired for the output file, eg `-base &80000`, `-base 256k` etc. When this is done, all relocatable objects in the input files are relocated using that base instead of the default.

relocatable

Usually, when an image file is produced, it will execute correctly only at the base address given (or the default). This is because the program will contain references to absolute addresses within itself. However, if you specify the `relocatable` option, the program image can be loaded and executed at any address.

This feat is achieved by adding a relocation table and a small program to perform the relocation of the image. The relocation table is a list of offsets from the start of the program to words which need relocating. These words are adjusted by the difference between the base address of the program and the address where it was loaded. Once the relocation has been performed, the program proper starts executing.

However, although this can be used to make a program statically relocatable, it does not confer true position-independence on the program. That is, the program cannot be moved in memory once it has started and still be expected to work.

workspace

The `-workspace` keyword, when used with the `-relocatable` option, specifies that the output image should execute towards the top of application workspace, leaving n bytes above itself for stack and heap. For example, `link -r -w 64K` will generate an image which, when loaded, will move itself to within `image-size + 64Kb` of the top of application workspace before executing.

Predefined Linker symbols

There are several symbols which the Linker knows about independently of any of its input files. These start with the string `Image$$` and, along with all other external names containing `$$`, are reserved by Acorn.

The symbols are

`Image$$RO$$Base`	Address of the start of the read-only (code) area
`Image$$RO$$Limit`	Address of the byte beyond the end of code area
`Image$$RW$$Base`	Address of the start of the read/write (data) area
`Image$$RW$$Limit`	Address of the byte beyond the end of the data area
`Image$$ZI$$Base`	Address of the start of the zero-initialised area
`Image$$ZI$$Limit`	Address of the byte beyond the end of the zero-initialised area

Although it will often be the case, Acorn do not guarantee that the end of the read-only area corresponds to the start of the read/write area. You should therefore not rely on this being true.

The read/write (data) area may contain code as programs are sometimes self-modifying. Similarly, the read-only (code) area may contain read-only data (eg string literals, floating-point constants; etc).

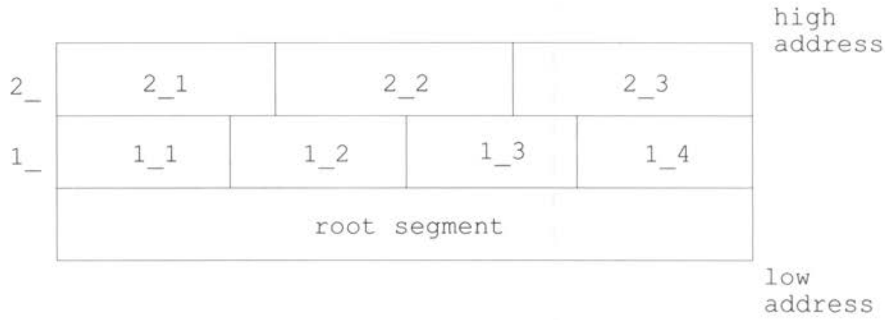
These symbols can be imported as relocatable addresses by assembly language routines that might need them.

Generating overlaid programs

An introduction to overlays and what they are is given in the chapter entitled *Overlays*. The reader new to the concept of overlays should first read that chapter; here we describe only how to use the linker to make an overlaid application.

A simple, 2-dimensional, static overlay scheme is supported. There is one root segment and as many memory partitions as specified by the user (called '1_', '2_', etc). Within each partition, some number of overlay segments (called '1_1', '1_2', ...) share the same area area of memory. The user specifies the contents of each overlay segment and the linker calculates the size of each partition, allowing sufficient space for the largest segment in it. All addresses are calculated at link time: overlaid programs are not relocatable.

A hypothetical example of the memory map for an overlaid program might be:



Segments 1_1, 1_2, 1_3 and 1_4 share the same area of application workspace. Only one of these segments can be in memory at any given instant; the remainder must be on disc.

Similarly, segments 2_1, 2_2 and 2_3 share the 2_ area of memory, but this is entirely separate from the 1_ partition.

The linker assigns AOF AREAs to overlay segments under user control (see below). Usually, a compiler produces one code AREA and one data AREA for each source file (called `C$$code` and `C$$data` when generated by the C compiler). The C compiler option `-zo` allows each separate function to be compiled into a separate code AREA, allowing finer control of the assignment of functions to overlay segments (but at the cost of slightly enlarged code and enlarged object files). The user controls the overlay structure by describing the assignment of certain AREAs to overlay segments. For each remaining AREA in the link list, the linker will act as follows:

- If all references to the AREA are from the same overlay segment, the AREA is included in that segment; otherwise,
- the AREA is included in the root segment.

This strategy can never make an overlaid program use more memory than if the linker put all remaining AREAs in the root, but it can sometimes make it smaller.

By default, only code AREAs are included in overlay segments. Data AREAs can be forcibly included, but it is the user's responsibility to ensure that doing so is meaningful and safe.

On disc, an overlaid program is organised as a RISC OS application. The components of the application must reside in a directory the name of which begins with `!`. This name is specified to the Linker as the argument to its `-o` flag and the linker warns if there is no initial `!`. The linker creates the following components in the application directory:

`!RunImage`	The root segment, an AIF image (which may be squeezed).	
`1_1`	)	
`1_2`	)	Overlay segments, which are plain binary
`...`	)	images, linked at absolute addresses. Overlay
`2_1`	)	segments must not be squeezed.
`...`	)	

If no `!Run` file exists in the application directory, the linker creates a `!Run` file (with 'obey' file type) containing the line

```
Run <obey$dir>.!RunImage.
```

overlay keyword

The overlay file named as argument to the `-ov[erlay]` option, describes the required overlay structure. It is a sequence of 'logical lines':

- A `\` immediately before the end of a physical line continues the logical line on the next physical line.
- Any text from a `;` to end of the logical line inclusive is a comment (for documentation purposes) which is ignored by the linker.

Each logical line has the following structure:

```
<segment-name> <module-name> [ "(" <list-of-AREA-names> ")" ] <module-name> ...
```

For example:

```
1_1     edit1 edit2 editdata(C$$code,C$$data) sort
```

`list-of-AREA-names` is a comma-separated list. If omitted, all AREAs with the CODE attribute are included.

A `module-name` is either the name of an object file (with all leading pathname segments removed) or the name of a library member (again, with all leading pathname segments removed).

In the example above, sort would match the C library module of the same name.

Note that these rules require that, within a link list, modules have unique names. For example, it is not possible to overlay a program made up from test.o.thing and o.thing (two modules called thing). This is a restriction on overlaid programs only.

C library modules

For reference, the C library modules are named as follows:

kernel	Included in all C programs – must be in root
clib	Included in all C programs – must be in root
alloc	Implements malloc family from <stdlib.h>, but used by kernel so must be in root
armsys	Included in all C programs – must be in root
ctype	Implements <ctype.h>
error	Implements <assert.h>
fpprint	Implements floating point part of printf-like things from <stdio.h>
locale	Implements <locale.h>
math	With clib, implements <math.h>
printf	Implements printf-like parts of <stdio.h>
scanf	Implements scanf-like parts of <stdio.h> and the conversion functions from <stdlib.h>
signal	Implements <signal.h>
sort	Implements qsort() and bsearch() from <stdlib.h>
stdio	Implements the remainder of <stdio.h>
stdlib	Implements the remainder of <stdlib.h>
string	Implements <string.h>
time	Implements <time.h>

In general, it would be unusual to be able to place a C library module in an overlay segment. Sort, time and, perhaps, scanf are the most likely candidates.

xref keyword

To help the user partition between overlay segments the linker can generate a list of inter-AREA references. This is requested by using the `-xref` (or `-x`) option. In general, if area A references area B, for example because x in area A calls y in area B, A and B should not share the same area of memory, or every time x calls y or y returns to x, there will be an overlay segment swap.

map keyword

The `-map` option requests the linker to print the base address and size of every AREA in the output program. Although not restricted to use with overlaid programs, `-map` is most useful with them, as it shows how AREAs might be packed more efficiently into overlay segments.

The Acorn Source-level Debugger

Overview

This chapter describes the Acorn Source-level Debugger (ASD). ASD is an interactive aid to debugging programs written in high-level, compiled languages, such as C, ISO-Pascal, and Fortran-77. It can be used on any Acorn computer running the RISC OS operating system. ASD can also be regarded as a *symbolic* debugger.

The first section, *About debuggers*, introduces the concept of debuggers in general, and source-level debuggers in particular. Next, *Using ASD* describes how to invoke ASD. *Specifying source-level objects* describes the way in which various source-level items, such as variable names, line numbers and labels are specified in ASD commands.

The next section, *Program examination commands* describes the ASD commands which are concerned with examination of the program being debugged. You can display and modify the value of program variables or display arbitrary expressions involving variables and constants, and examine the state of execution of the program and the arguments of an active procedure call.

Execution control commands deals with the ASD commands which control the running of the program being debugged. Facilities available include the ability to start (or re-start after a breakpoint) program execution, single-step a statement at a time, set and clear breakpoints, and to initiate the 'watching' of a variable, or the tracing of procedure calls.

Low-level debugging describes ASD facilities which may be used to debug programs, including high-level language programs, at assembly language level.

Miscellaneous commands describes the ASD commands which don't fall into any of the previous categories. These commands include those that display on-line help information, quit from the debugger, and a command allowing you to define your own commands.

An example ASD session gives an example of how ASD might be used to debug a rather bug-ridden sort utility.

Finally, a command summary lists all the ASD commands alphabetically, with a brief description of each. A copy of the list also appears on one of the reference card included with this release of C.

About debuggers

This section is aimed mainly at readers who haven't used a program debugger of any sort before. However, others may find it useful reading, as it introduces some of the terminology used in the rest of this chapter.

Anyone who has written a program more than about ten lines long has had recourse to debugging techniques: the tracking down and removal of errors. The form this takes depends on many things, not least the language in which the program is written. A common technique for tracing bugs in systems which have no explicit debugging support is the planting of 'trace' information in the program itself. For example, in a while loop in C you might print a message:

```
while (i >= 0) {
      printf("while loop: i == %d\n",i);
      ...
}
```

Such additions to the program can be useful, but are tedious to use in compiled languages because every time you want to change the debugging statements, the program has to be recompiled. There is also the possibility that the debugging statements themselves have undesirable side-effects which contribute to the ill-health of the program.

Planting tracing information in assembly language programs is even trickier. In general, the assembly language programmer does not have access to the rich expression evaluation and print formatting routines of high-level languages. For example, displaying the contents of all of the Acorn RISC Machine (ARM) registers in hexadecimal is a non-trivial code fragment in ARM assembler.

To help assembly language programmers debug their programs, a class of utility known as the debugger or monitor has evolved. Such programs allow the user to examine and alter memory locations and machine registers, set breakpoints, single step through a program and 'watch' particular memory locations for changes.

Because much of the terminology used in ASD derives from such debuggers, the next section describes typical facilities that they provide. These are then compared with the equivalent ASD commands.

Starting execution

Machine code monitors usually provide a command of the form `GO addr`, which starts execution from a particular instruction. The ASD equivalent is just `go`, which starts execution at the first statement in the program.

Examining memory

As mentioned above, a machine-level debugger allows you to examine the memory of the machine, and possibly alter its contents. A typical command would be `EXAMINE address range` to display the contents of a range of memory locations in hex, and `REGISTERS` to display the contents of the machine's registers. The ASD equivalent is `print`. This displays the value of an arbitrarily complex expression. Instead of using memory addresses, you can use the names of variables.

Setting breakpoints

It is useful to be able to stop the program at a certain point, so that the state of its variables can be checked against their expected values. This is known as *setting a breakpoint.* When the program reaches the instruction at which the breakpoint has been set, execution is suspended and control returns to the debugger. In machine code terms, breakpoints are set at particular memory addresses. For example:

```
BREAK 1ABFC.
```

Under ASD, you can set breakpoints in terms of source-level addresses. That is, you specify the source-file line number (and possibly the statement within that line) where you want the program to stop. For example:

```
break ray:234
```

You can also set a breakpoint at the entry and exit points of a procedure. A useful extension of breakpoints is being able to break only if some specified condition is true.

Single stepping and tracing

Once a breakpoint has been encountered, you may wish to step slowly through the program, examining changes to variables after each step. At the machine code level, single stepping involves executing one instruction at a time. Under ASD, single stepping works at the level of one statement at a time. You may specify whether a procedure (or function) call counts as one statement or whether each of the statements within that procedure should be stepped individually.

Single stepping can be quite time consuming, especially if you only want to get an idea of the general flow of control within the program. An alternative is to use procedure tracing. When this is enabled, a message is displayed every time the program enters or leaves a procedure.

Setting watchpoints

A common cause of incorrect operation in programs is the corruption of a variable. The reason for this corruption can be very hard to track down, especially if the program contains many global variables which are accessible from a large number of procedures. ASD helps to track down undesired assignments to variables by allowing you to place a *watch* on them. When a variable is being watched, a check is made for any changes in its value. When a change occurs, the program execution is suspended and control returned to the debugger.

This concludes the description of common debugger concepts. Of course, only a few of the commands provided by ASD have been mentioned so far. Detailed descriptions of all of them can be found in the following sections.

Using ASD

This section describes how the debugger is invoked, and how programs to be debugged under it must be compiled. ASD uses special information in the program being debugged, which provides ASD with information about the source code that generated the program. This information is not automatically included in the output of the compiler. (This is mainly for reasons of efficiency: programs which contain debug information are larger, take longer to compile, and run more slowly than those that do not.)

C compiler debugging options

The generation of debug information is enabled by specifying the appropriate option or 'switch' on the command line.

The flag for enabling debugging information is `-g`. This may be followed by one or more letters indicating the level of debugging information to be generated, as follows:

`f`	Produce information on top-level variables and functions
`v`	Produce information on local variables
`l`	Produce information on line numbers
`a`	Produce information on all of the above (ie `-gfvl`)

If no letter follows `-g`, `-ga` is assumed.

The term *top-level variables* is used here to refer to those declared outside any function definition. Their lifetime is the period of execution of the program, and they may be global to the program, or local to the file in which they are declared. Local variables are those declared within the body of a function (including function parameters). These may exist only for the duration of the function call (automatic variables), or for the lifetime of the program (static).

Obviously the most useful option is `-ga`, but this is also the most space-consuming. If you are having problems with the corruption of a global variable on which you want to place a watchpoint, you might compile the program with just the `-gf` option.

Because each module of a program can be compiled with its own debugging level, you need only specify debugging for suspect modules. Well-proven modules in which you have complete faith can be compiled with no debugging information, whereas newer, less reliable code can have maximum debugging specified.

Turning on debugging options inhibits optimisation, and reduces the speed of execution of your program, even when you are not debugging it. This of course does not matter when you are using the debugger, but for maximum speed, programs should be compiled without `-g`.

Linking

When linking a program to be debugged, you must instruct the linker to include the debugging information generated by the compiler. To do this, use the `-debug` option on the `link` command line.

Invoking the debugger

Once you have a successfully linked the program, the debugger may be used to control its execution. To call ASD, ensure that the program of that name is somewhere in your run search path (typically in the directory `$.Library`). Then issue the command:

*asd *image_name* [*arguments*]

from the RISC OS Command Line prompt, where `image_name` is the name of the program you wish to debug, and `arguments` are any Command Line arguments that the program would normally take when run. For example:

```
*asd raytrace
```

As with all Acorn language products, ASD responds to the `-help` option, in this case by printing the version number, command syntax, and some other useful information.

On starting, ASD prints a few lines of the following form:

```
Acorn Source-level Debugger, version number [date]
Object program file raytrace
ASD:
```

If the file specified in the command line could not be found, an error message to that effect is displayed.

To see a list of the commands that ASD provides, type `help`. You may recognise some of these and be tempted to start experimenting with them. However, you are recommended to read the next section before you do.

Remote debugging

Sometimes it may be difficult to use ASD because screen output produced by the program you are trying to use gets confused with diagnostic output from ASD. ASD provides a facility called *remote debugging* which allows you to use a terminal connected to the computer's serial port to enter ASD commands and display ASD output. To start a remote debugging session, use the command `asd -remote` instead of just `asd`. The default baud rate and data format are taken from the system configuration; if these are not correct you can specify them after the `-remote` flag. The syntax for `asd -remote` is

```
asd -remote [baud_rate[,data_format]]
```

`Baud_rate` can be one of 75, 150, 300, 1200, 2400, 4800, 9600 or 19200.

`Data_format` can be one of

7,e,2	7 bits, even parity, 2 stop bits
7,o,2	7 bits, odd parity, 2 stop bits
7,e,1	7 bits, even parity, 1 stop bit
7,o,1	7 bits, odd parity, 1 stop bit
8,n,2	8 bits, no parity, 2 stop bits

8,n,1	8 bits, no parity, 1 stop bit
8,e,1	8 bits, even parity, 1 stop bit
8,o,1	8 bits, odd parity, 1 stop bit.

Specifying source-level objects

Once ASD is running, the object program can be executed, single stepped, have its variables examined and so on. All of these facilities are described in the following sections. However, before you can use these commands, you have to know how to specify certain source-level entities. For example, variable names, line numbers and program labels all have a syntax which must be used correctly if you are to reference the desired object.

Variable names and context

It is clearly important that a source-level debugger allows you to refer to the program's variables by the names they have in the original source code. A variable is simply referenced by its name: if you want to print the value of variable `count`, you would use the command

```
print count
```

When a program written in a block-structure high-level language is executing, there exists a *current context*. This refers to both the textual nesting of procedures, and the dynamic run-time nesting of procedure calls. Taking the first aspect first, and using Pascal as an example, consider the following definitions (an example in C is given a little later):

```
program raytrace(input,output);
   var
      count : integer;
   ...
   procedure pixel(x,y : integer);
      var
         i : integer;
      ...
      function reflect(x,y : integer; angle : real) : integer;
         ...
         function quicksin(angle : real) : real;
         begin
            { body of quicksin ** BREAKPOINT HERE ** }
         end;
      begin
         { body of reflect }
      end;
   begin
      { body of pixel }
   end;
```

```
begin
   { body of raytrace }
end.
```

Assume the program stops (because of a breakpoint, perhaps) in the body of the function `quicksin`. At this point, the variables visible to the program are `angle`, the parameter of the function, `x` and `y`, the parameters of `reflect`, `i`, the local variable of `pixel`, and the global variable `count`.

Variables that are defined in the current context can be accessed from the ASD command prompt simply by giving their names. This would include the real parameter of `quicksin` called `angle` in the current example, but no others.

You can refer to other variables by qualifying their names with the context (procedure name) in which they are defined. For example, the parameters of the function `reflect` would be referred to as `reflect:x`, `reflect:y`, and `reflect:angle` respectively. Notice that in the last example, you can refer to a variable which wouldn't actually be visible to the executing program.

Another example would be a reference to the global integer variable `count`, defined in the main program. Here you use the module name as the qualifier. In Pascal, the name after the `program` keyword is taken as the module name, so the required identifier is `raytrace:count`. In analogous situations in C, you would use the source filename as the qualifier (see the examples below and in the example session at the end of the chapter). In Fortran-77, as with Pascal, the `PROGRAM` name is used to prefix global variables.

To avoid certain ambiguous cases, where more than one procedure of a given name exists, you can 'nest' the qualifiers before the final variable name. For example, the local variable `angle` in the function `reflect` could be referred to as `raytrace:pixel:reflect:angle`, even though the first two components are not strictly required to access the desired object unambiguously in the present example.

The next example, in C, illustrates the further features of specifying variables outside the current context. C does not support the textual nesting of functions, so variables are either defined at the top level (outside any function definitions), or one level down, in a function definition (though see below for further discussion). However, C obviously does support nested function calls, and like Pascal, allows recursive calls.

Consider this source code fragment:

```
/* File >c.expr */
int    val;
int    factor(char **ptr)
{
       int    val;
       /* BREAKPOINT HERE */
       ...
       return val;
}
int    add(char **ptr)
{
       int    v1, val;
       ...
       v1 = factor(ptr);
       return v1+val;
}
int    logical(char **ptr)
{
       int    v1, val;
       ...
       {
              int    v1 = add(ptr);
              return v1 & val;
       }
}
int    expr(char **ptr)
{
       ...
       return logical(ptr);
}
void   main(int argc, char *argv[])
{
       char   *p;
       ...
       val = expr(&p);
       ...
}
```

In this example, there is a global variable `val`, and several local variables. Suppose the current context is in the body of `factor`, which was called by `add`, called by `logical`, called by `expr`, called by `main`. As far as the programmer is concerned, the only variables visible at this stage are local integer `val` and the parameter `ptr`. Because of C's lexical scoping rules, all the other variables which are extant are invisible from `factor`. The global `val` would have been visible, had it not been for the local of the same name hiding it.

Similarly, the only variables accessible directly to ASD are those defined in the current context of `factor`. However, the others can still be examined and altered by giving their defining contexts. Examples are: `expr:ptr`, `main:argc` and `add:v1`.

Consider what the name `expr:val` would refer to. Seemingly there are two candidates: the global variable, which qualifies because `expr` is the name of the module (program) in which it is defined, and the local variable defined in the function which is also called `expr`. In fact, it is the second one which would be referenced, as ASD always accesses more local variables first.

To overcome this ambiguity, there is a special qualifier, the # character. You can regard this as the 'parent' of all the modules in the program, so the element following it in an identifier name is a module. In this example, you would use `#expr:val` to access the global called `val` (and `#expr:expr:val` would have the same meaning as just `expr:val`).

The function `logical` shows another example of possible ambiguity, which can occur in C but not Pascal. In Pascal, you are not allowed to declare local variables within `begin...end` blocks; in C you are. There are two declarations of `v1` in `logical`, so the name `logical:v1` is not precise enough. The way around this is to qualify the name with the line number on which it is declared as well as (or instead of) the function name. These might be `logical:115:v1` and `logical:123:v1` in the present example.

ASD will give an error if a reference to a variable name is ambiguous.

Finally, there is the question of multiple activations of a particular procedure, since both C and Pascal allow recursive function and procedure calls. Consider the standard example:

```
int    factorial(int n)
{
       if (n <= 1)
             return 1;
       else
             return n*factorial(n-1);
}
```

Suppose this function is called with an initial argument of 5. The function would recurse four times, until it was called with `n==1`. At this stage, asking ASD to access variable `n` would yield the most recent version. By prefixing the variable name with a backslash (\) followed by an integer, you can refer to any of the other activations. For example, `\1:n` refers to the earliest invocation of `factorial`, where `n==5`. `\2:n` refers to the next oldest, and `\5:n` would refer to the active one. Negative integers can be used to work backwards from the current activation. So in this example, `\-1:n` and `\4:n` would be the same (`n==2`).

If the function or procedure name is required along with an activation count, the form is `factorial\-1:n`.

This section has been quite involved, but that only reflects the versatility that is required by ASD in order to allow access to the various types of variable instantiations possible in modern, block-structured languages. For the most part, you will find that unqualified variable references are all that is required, and convoluted strings such as

```
#raytrace:fred:jim\-1:count
```

are not needed.

Program locations

Some ASD commands, such as `break`, require arguments which refer to locations in the program. You can refer to a place in the program by procedure entry/exit, line number, statement within a line (in the case of C and Pascal), or label.

Refer to lines simply by giving the line number: `123` refers to the 123rd line of the program. You can qualify line numbers in the same way as variables, so `prog:87` is the 87th line in `prog` and `#ray:3214` is the 3214th line in the module `ray`.

You can use the procedure name alone to set a breakpoint at the entry point of the procedure. Alternatively, the end of a procedure (just before it returns) may be trapped using `proc:$exit`.

To refer to a statement within a line, the notation *`line.stat`* is used. For example, `100.3` refers to the third statement on line 100. Clearly statement `n.1` is the same as statement `n.`.

It is possible that a simple line number is ambiguous. This occurs when an include file is invoked from within a function. For example, suppose you have the file `h.ray` which looks like this:

```
0001  #define     maxx        1000
0002  #define     maxy        1000
0003
0004  typedef     unsigned char     flag, byte;
0005  ...
...
0099  /* End of h.ray */
```

(The line numbers are included for clarity; they wouldn't appear in the file itself.) Suppose further that this file is included in a C source file:

```
0001  main()
0002  {
0003  #include "h.ray"
0004  int   x,y;
0005  real  angle
0006  ...
```

The resulting source as seen by the compiler, with line numbers, is:

```
0001  main()
0002  {
0003  #include "h.ray"
0001  #define     maxx        1000
0002  #define     maxy        1000
0003
0004  typedef     unsigned char     flag, byte;
0005  ...
...
0099  /* End of ray.h */
```

```
0004  int   x,y;
0005  real  angle
0006  ...
```

Line reference `4` might refer to the `typedef` or the `int` declaration. To overcome the ambiguity, it is possible to suffix the line number with the filename: `4(c.ray)`, `4(h.ray)`.

The final type of program location reference is the label. C labels are just identifiers, so these may be used 'as is'.

Expressions

Several ASD commands require arbitrary expressions as arguments. The syntax for these expressions is based on that found in C.

ASD has a rich set of operators and several levels of operator precedence. These are summarised below.

1	`()`	grouping, eg `a*(b+c)`
	`[ ]`	subscript, eg `isprime[n]`, `matrix[1][2]`
	`.`	record selection, eg `rec.field`, `a.b.c`
	`->`	indirect selection, eg `rec->next` is `(*rec).next`
2	`!`	logical not, eg `!finished`
	`~`	bitwise not, eg `~mask`
	`-`	negation, eg `-a`
	`*`	indirection, eg `*ptr`
	`&`	address, eg `&var`
3	`*`	multiplication, eg `a*b`
	`/`	division, eg `c/d`
	`%`	remainder, eg `a%b` is `a-b*(a/b)`
4	`+`	addition, eg `a+1`
	`-`	subtraction, eg `b-d`
5	`>>`	right shift, eg `k>>2`
	`<<`	left shift, eg `2<<n`

6	`<`	less than, eg `a<b`
	`>`	greater than, eg `n>10`
	`<=`	less than or equal to, eg `c<=d`
	`>=`	greater than or equal to, eg `k>=5`
7	`==`	equal to, eg `n==0`
	`!=`	not equal to, eg `count!=limit`
8	`&`	bitwise and, eg `i & mask`
9	`^`	bitwise xor, eg `a ^ b`
10	`|`	bitwise or, eg `m1 | 0x100`
11	`&&`	logical and, eg `a==1 && b!=0`
12	`||`	logical or, eg `a>lim || finished`

The lower the number, the higher the precedence of the operator. Note the syntax for subscripting and record selection. The object to which subscripting is applied must be a pointer or array name. The debugger will check both the number of subscripts and their bounds in languages which support such checking. A warning will be issued for out-of-bound array accesses. As in C, the name of an array may be used without subscripting to yield the address of the first element.

The prefix indirection operator `*` is used to dereference pointer values, in the same way as Pascal's postfix operator `^`. Thus if `ptr` is a pointer type, `*ptr` will yield the object it points to (like `ptr^` in Pascal).

To access the fields of a record through a pointer, you can either use `(*recp).field`, or the C 'shorthand' notation, `recp->field`.

If the lefthand operand of a right shift is a signed variable, then the shift will be an arithmetic one (ie the sign bit is preserved). If the operand is unsigned, the shift is a logical one, and zero is shifted into the most significant bit.

If incompatible types are used during expression evaluation, the debugger will print a warning message, but evaluation will continue.

Constants may be decimal integers, floating point, octal integers or hexadecimal integers. The following examples show each in turn:

```
123
12.3e10
0100 (64 decimal)
0x1ff (511 decimal)
```

Character constants are also allowed, eg `'A'` yields 65 (the ASCII code for A). Note that `1` is an integer, whereas `1.` is a floating point number.

Program examination commands

This section lists and describes those commands which examine the state of the program being debugged. In the syntax descriptions of the commands, various items such as *`context`* are mentioned. These are explained below.

`context` describes an activation state of the program. Possible elements of a context were described in the section *Specifying source-level objects*. Formally, it looks like:

```
[[#]module:][proc:]...proc[\[-]count]
```

In other words, an optional module prefix, optionally prefixed by # to avoid ambiguity with procedures of the same name, followed by a list of procedure names, the last of which which may have optional invocation level following the backslash. Examples are:

`pixel`	procedure or module called `pixel`
`raytrace:pixel`	procedure `pixel` defined in `raytrace`
`raytrace:pixel\-1`	previous invocation of `pixel`
`$ROOT:raytrace:pixel:reflect`	procedure `reflect` defined in `raytrace`

If the program is currently in a 'stopped' state, eg after a breakpoint or watchpoint has been activated, there is an *execution state context*. This refers to the context of the procedure being executed when it was suspended.

`expression` is an arbitrary expression using constants, variables and the operators described in the previous section.

`variable` is a reference to one of the program's variables. If a simple name is used, the variable is looked up within the current context. This may be overridden by prefixing the variable name with a context as described above.

`count` is an unsigned decimal integer.

`format` is a C `printf` function format descriptor, or the word `hex`, `ascii` or `string`. It is beyond the scope of this chapter to describe all of `printf`'s format strings, but the most common ones are:

Type	Format	Description
`int`	`%d`	signed decimal integer (default for integers)
	`%u`	unsigned integer
	`%x`	hexadecimal with lower case letters (same as `hex`)
`char`	`%c`	character (same as `ascii`)
`char *`	`%s`	pointer to character (same as string)
`void *`	`%p`	pointer (same as %.8x, eg 00018abc)
`float`	`%e`	exponent notation, eg 9.999999e+00
	`%f`	fixed point notation, eg 9.999999
	`%g`	general floating point notation, eg 1.1, 1.23e+06

Note that in the `print` command, the first group above (`int` and `char`) should only be used if the expression being printed yields an integer, and the third group should only be used for floating point results. `%p` is safe with any kind of pointer, but `%s` should only be used for expressions which yield a pointer to a zero-terminated string.

Print command

This command can be used to examine the contents of the debugged program's variables. You can also use it to display the result of arbitrary calculations involving variables and constants. The syntax is:

`pr[int][/`*`format`*`]` *`expression`*

The `format` string was described in the previous section. If it is omitted, then for integer expressions, the default set by the `format` command is used. This is `%d` by default. For floating point values, the default format string is `%g`. Pointer results are treated as integers for the purposes of printing and are printed using the format `%.8x` (eg `000100e4`).

Structures, unions, arrays, sets, subranges and strings are printed in formats appropriate to their types. The format string is applied to each individual element.

Note that the / marking the start of the `format` should follow the command name, with no intervening spaces. Examples are:

`print \-1:a+1`	Print `a+1` in the default format
`print isprime[3]`	Print the fourth element of array `isprime`
`print/hex isprime`	Print all elements of array `isprime` in hex
`print/%10s promptstr`	Print the string `promptstr`
`print/%X listp->next`	Print field `next` of structure `listp` (in hex)
`print listp`	Print all fields of structure `listp`
`print/%f angle*180/3.14159`	Convert `angle` from radians to degrees

Format command

This command is used to set the default format string used by the `print` command for integer results. It is set to `%d` when ASD starts up. That may not be suitable (for example, you may want to treat integers as unsigned quantities, or print integers in hex) so `format` allows you to change it. The syntax of the command is:

```
form[at] [format]
```

The *`format`* string is exactly as described previously. Examples are:

```
format hex
format %u
```

There is nothing to stop you from using one of the floating point formats in this command. It wouldn't be very wise, though, as integers would then not be printed correctly at all. (Try `p/%g 123` if you don't believe us.)

Let command

The `let` command enables you to change the value of a variable. It has the syntax:

```
[let] variable=expression {, expression}
```

The *`variable`* and the *`expression`* should be compatible types, though the debugger will perform conversions between integer and floating if necessary (floats are rounded towards zero). Only the real parts are affected by arithmetic on these types.

Note that although you can change the value of an array *element*, using a command such as:

```
let isprime[2]=1
```

you cannot change the address of the array itself, as array names are treated as constants. If the subscript is omitted it defaults to `[0]`.

If multiple expressions are specified on the righthand side, each expression is assigned to `*(&variable + N - 1)`, where `N` is the `N`th expression on the righthand side.

Examples of `let` are:

```
let a=a+1
let rec=rec->next
let isprime[2]=1, 1, 0, 1, 0, 1 ...
```

Symbols command

This command lists the symbols (variables) defined in the given context, or the current context if it is omitted. Its syntax is:

```
sy[mbols] [context]
```

Each variable's name is displayed, along with its type information. An example of the output produced might be:

```
ANGLE Float, local
X     Signed integer, local
Y     Signed integer, local
I     Signed integer, local
```

The format is *name type*, *storage class*. Other types you might see are:

```
Signed half-word (short)
Signed byte (character)
Unsigned integer
Unsigned half-word (short)
Unsigned byte (character)
Float
Double
Pointer to...
Array of...
```

Other storage classes are:

```
register
automatic (local)
static
external
```

To see the global variables, you would quote the module name as the context. For example, to see the external and static variables defined outside of any function definitions in a C program, you might use:

```
sym testp
```

where `testp` is the name of the source file.

Note also the comment about potential problems with register variables in the description of the `watch` command.

Variable command

This provides type and context information about a specified variable. Its syntax is:

v[ariable] *variable*

Examples of its usage and the results displayed are:

```
ASD: var angle
ANGLE Float, local
context: FASTSIN  pascal.raytrace
ASD: var reflect:angle
ANGLE Float, local
context: REFLECT  pascal.raytrace
ASD: var count
COUNT Signed integer, static
context: RAYTRACE  pascal.raytrace
```

Arguments command

This command is used to show the arguments which were passed to the current procedure, or to another active procedure. Its syntax is:

a[rgument]s [*context*]

If the *context* is omitted, the current context is used (usually the procedure that was active when the program was suspended, unless it has been changed by a `context` command). Examples are:

```
args
args \-1
args main
```

For each argument, its name and current value are displayed.

Context command

This is used to set the context in which variable lookups will occur. It also affects the default context used by commands such as `symbols`. When program execution is suspended, the search context is set to the active procedure. The syntax of the command is:

`con[text] [`*`context`*`]`

If you omit the argument, the context will be reset to the active procedure. Examples are:

```
context
con factorial\1
con prog:expr
```

Out command

The next two commands, `out` and `in`, are shorthand ways of changing the current context by one level. `out`, whose syntax is simply:

```
ou[t]
```

sets the context to that of the caller of the current context. For example, if the current context were `pixel:reflect:quicksin` then executing an `out` command would set it to `pixel:reflect`. You will get an error if you issue an `out` command if the current context is the top level of execution.

In command

This command performs the opposite function to `out`. It sets the context to the procedure called from the current level. Continuing with the previous example, if you execute an `in`, the context will be set back to `pixel:reflect:fastsin`.

The syntax of the command is:

```
in
```

You may not issue an `in` command when the current context is that of the executing procedure: an error is given if you try.

Where command

This command prints the current context in terms of a procedure name, line number in the file and filename. The syntax is simply:

```
wh[ere] [context]
```

An example display from the `where` command is:

```
sortfile, line 99 of c.sort
   99     if (! (lbuf = malloc(1 * sizeof (char *))))
```

Backtrace command

This command prints information about all the currently active procedures (most recent first), or for a given number of levels. The syntax is:

```
ba[cktrace] [count]
```

An example of the output from this command is:

```
REFLECT, line  45 of   pascal.raytrace
PIXEL,    line  124 of  pascal.raytrace
RAYTRACE, line 48 of   pascal.raytrace
```

Type command

This command types the contents of a source file (or any text file) between specified locations. The syntax is:

```
t[ype] [expr1][,[[+]expr2][,[file]]]
```

The source lines between *expr1* and *expr2* are specified. *expr1* defaults to the source line associated with the current context or the last line displayed with the `type` command, –3. *expr2* defaults to *expr1*+10. *file* defaults to the filename associated with the current context. If the optional + is given before *expr2*, *expr2* denotes a line count rather than the limit of a line range.

Execution control commands

This section describes the ASD commands which control the execution of the object program. Facilities covered include loading programs to be debugged, the setting of breakpoints and watchpoints, and single stepping.

Load command

This command loads or reloads an image for debugging. It syntax is:

```
load image_file [arguments]
```

where `image_file` is the name of the program you wish to debug and `arguments` are any command line arguments that the program would normally take when run. `image_file` and `arguments` may also be specified on the ASD command line when you invoke ASD.

Cmdline command

This command sets up command line arguments for the debuggee. It syntax is:

```
cm[dline] arguments
```

where `arguments` are any command line arguments that the debuggee would normally take when run.

Go command

This command starts execution of the program. The first time `go` is executed, the program starts from its normal entry point (eg at the start of the `main` function in C). Subsequent `go` commands resume execution from the point where execution was suspended, eg at a breakpoint or a watchpoint.

The syntax is:

```
g[o] [w[hile] expr]
```

If the `while` clause is specified, `expr` is evaluated whenever a breakpoint occurs, and if it evaluates as true (ie non-zero), the breakpoint is not reported and execution is resumed.

Step command

This command steps execution through one or more statements. It can only be issued after the program has been started: you should not use `step` to initiate program execution. The syntax is:

```
s[tep] [in] [count | w[hile] expr]
```

The `in` keyword, if present, denotes that single stepping continues into procedure calls. That is, each statement inside a called procedure is single stepped. If `in` is absent, a procedure call counts as only one statement, and is executed without single stepping. If the optional `count` is omitted, one statement is executed, otherwise `count` statements are executed. If the while clause is specified, `expr` is evaluated after each statement is executed, and execution continues until `expr` evaluates as false (ie zero).

Examples are:

`step 20`	Execute 20 statements
`s in`	Step into a procedure call
`s in 5`	Execute five statements, stepping into any procedure calls
`s w hp < sp`	Step through the current procedure until `hp >= sp`

Break command

This command is used to set a breakpoint. Breakpoints may be specified at procedure entry and exit, lines, statements within a line, or at program labels. The section *Specifying source-level objects*, near the beginning of this chapter, describes how program locations are specified. The syntax of `break` is:

```
[b[reak] [location [count]] [do
      '{'[command]{;command}'}'][if expr]]
```

If you issue `break` with no arguments, a list of the currently set breakpoints is displayed. For example:

```
#1 at FASTSIN
#2 at RAYTRACE:324
#3 at RAYTRACE:$999
```

Breakpoint numbers (`#n`) may be used in the `unbreak` command instead of the location descriptor.

`location` specifies where the breakpoint is to be placed. The section *Specifying source-level object* describes how program locations are specified.

The *`count`* that follows the breakpoint location indicates how many times the statement there must be executed before the program is actually suspended. It defaults to 1, so if the count is omitted, execution will stop the first time the breakpoint is encountered.

Alternatively, the breakpoint may be taken conditionally upon the value of `if` *`expr`* (see the example overleaf).

The `do` clause allows you to specify a list of commands to be executed when the breakpoint occurs. These commands could, for example, print the value of some variable and then continue execution with the `go` command. Normally when a breakpoint occurs the program and source line are displayed. If you specify a `do` clause these are not displayed, though you can display them by placing the `where` command at the start of the command list.

Examples are:

`break fastsin`	Break on entry to procedure `fastsin`
`b raytrace:324 10`	Break at line 324 of module `raytrace`
`b raytrace:$999`	Break at label 999 of module `raytrace`
`b 11 do {pr argv[i];g} if i>2`	Break at line 11 if `i`>2, display `argv[i]` and continue

Note that if you set a breakpoint at a procedure exit, using for example:

```
break proc:$exit
```

then several break points may be set, one for each possible exit. (A C function, for example, may have multiple `return` statements.) You may then delete ones which you do not require using `unbreak` with a breakpoint number, or delete them all using the same location as given in the `break` command.

Unbreak command

This command removes a breakpoint location from current list. It has the form:

unbr[eak] [*location*]

where *location* may either be a source code location, or the breakpoint number, as displayed by the `break` command. If the breakpoint being removed is not the last one, the breakpoint list is *not* renumbered, so once a breakpoint number is assigned, it remains constant.

`unbreak` with no arguments removes a breakpoint provided there is only one breakpoint set.

Examples are:

```
unbrk #1
unbrk raytrace:$999
```

Watch command

This command is used to set a watchpoint on a variable. When the variable is altered, program execution is suspended. The syntax of the command is:

w[atch] [*variable*]

If the argument is omitted, a list of current watchpoints is listed. For example:

```
#1 at K
#2 at ISPRIME[4]
```

As with breakpoints, the watchpoint number may be used in the `unwatch` command to remove a watchpoint.

Examples are:

```
watch k
watch isprime[4]
```

If there are any watchpoints set, execution becomes very slow. This is because the values of the watched variables are checked after every machine instruction that might change them. The best way to deal with this is to set a breakpoint in the area of code under suspicion, and only set the watchpoint(s) when the program stops there.

You should be aware that the C compiler produces code which can use a register to hold more than one variable, if the 'lifetimes' of those variables don't overlap. Thus if you ask for the value of a (register) variable at a point beyond where the compiler 'knows' it will no longer be required, you may actually see the value of a totally different variable. The same goes for changing the variable's value.

Unwatch command

This command clears a watch point. It has the syntax:

unw[atch] [*variable*]

As mentioned above, the *variable* reference may either be an actual variable name, or a watchpoint number preceded by a # sign. `unwatch` with no arguments removes a watchpoint, provided there is only one watchpoint set.

Return command

This command can be used to return to the caller of the current procedure, passing back a result if required. It has the form:

```
return [expression]
```

For example, from a C function you could type something like `return -1` to pass a result back to the caller.

It is not possible to return compound data types (arrays and records) using this command.

Ptrace command

This command enables and disables procedure tracing. When enabled, this causes the name of the current procedure to be printed every time it is entered or left. The syntax of the command is:

```
pt[race] [on|off]
```

If no argument, or `on`, is given, tracing is enabled. If `off` is specified, tracing is disabled. Indentation is used to indicate procedure nesting. For example:

```
Entered main
 Entered init
 Left init
Left main
```

Call command

This command calls a procedure. The syntax is

`call` *`location`* `['('`*`expr`*`]{,`*`expr`*`}')']`

Each *expr* is an argument to the procedure. If the procedure (function) returns a value, this may be examined using the command `print r0`.

Void command

This command is identical to the `call` command above, but does not print a result.

While command

This command is only valid at the end of a multi-statement line. Multi-statement lines are entered by separating the statements with `;` characters. The syntax of the command is

`while` *`expr`*

This causes interpretation of the line to repeat until *expr* evaluates to false (ie zero).

Low-level debugging commands

The section describes the low-level debugging facilities of ASD. These can be used to debug high-level language programs at the machine code level, as well as programs written in assembly language. To get the most out of this section you will need to be familiar with assembly language programming.

Two types of table can be present in a debuggable image: high-level tables produced by the compiler and low-level tables as produced by the linker with the `-debug` flag. Either form of table can be present on its own or both can be present together. However, with the current linker implementation (version 3.00) it is not possible to include high-level tables without including low-level tables.

High-level tables specify detailed information about the source code that generated the image. Low-level tables simply equate symbolic names to memory addresses.

There is no need to compile a program with debugging information if you only wish to use the low-level debugging facilities of ASD. You only need to link or relink it with the `-debug` option. For example:

`cc -c c.sort`	Compile without debugging information
`link -o sort -deb o.sort $.clib.o.ansilib`	
	Link with debugging information

When ASD reads an image and finds high-level debugging tables it sets the default language to one of C, Pascal or Fortran depending on which compiler generated the debugging tables. If ASD does not find any high-level debugging tables it sets the default language to `none`; this enables certain low-level debugging facilities in ASD. If you have a program which contains high-level debugging information and you wish to use the low-level debugging facilities of ASD you should use the `language none` command as soon as you enter ASD. You may also like to specify base 16 so that you can enter numbers and addresses in hexadecimal.

When referring to a low-level symbol you should precede it with an @ character. This tells ASD you are referring to the low-level symbol, not the high-level symbol. For example:

`break main`	Sets a breakpoint at the high-level symbol `main`.
`break @main`	Sets a breakpoint at the low-level symbol `main`.

These are not equivalent. The high-level symbol `main` refers to the address of the code generated by the first statement in the procedure; there may be some stack frame initialisation code before the first statement's code. The low-level symbol refers to the call address of the procedure (ie the first instruction of the stack frame initialisation).

Low-level symbols can be used in the `watch` command to set a watchpoint on a memory word. For example

```
watch @arglist
```

This will stop execution if the word at the location `arglist` changes. However, it is only possible to watch whole words (4 bytes) using low-level symbols since the low-level tables do not give any indication of the size of the object.

You can use memory addresses instead of low-level symbols. For example with the `where` command you can enter

```
where @0x80b0
```

If high-level tables are present and high-level debugging is enabled, this will display the source line that generated the instruction at 0x80b0. Otherwise, it will disassemble the instruction at location 0x80b0 and print the name of the nearest associated low-level symbol and an offset from that symbol to location 0x80b0 as follows:

```
main + 0x18
 +0018 0x0080b0: 0e1a06004 .' a mov     r6,r4
```

Low-level symbols can be used wherever an expression is expected (as in the `print` command). In an expression there is no need to precede the symbol with an @ symbol unless there is a high-level data symbol of the same name. For example:

`pr arglist`	Prints the value of `arglist`
`pr @arglist`	Prints the address of `arglist`
`pr main`	Prints the address of `main`

Note that in the last case there is no need to precede `main` with the `@` symbol even though there is a high-level symbol `main`. This is because high-level code symbols are not permitted in expressions, so `main` is unambiguous.

ASD predefines the following symbols in support of low-level debugging:

- R0, R1, R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15
  These refer to the ARM registers 0 to 15.
- A1, A2, A3, A4
  These refer to arguments 1 to 4 in a procedure call (stored in R0 to R3).

- V1, V2, V3, V4, V5, V6
  These refer to 6 general purpose register variables which the compiler may allocate as it pleases (stored in R4 to R9).
- SL – the Stack Limit register (R10)
- FP – the frame pointer (R11)
- IP – used in procedure entry and exit and as a scratch register (R12)
- SP – the Stack Pointer (R13)
- LR – the Link Register (R14)
- PC – the Program Counter (R15)
- F0, F1, F2, F3, F4, F5, F6, F7
  These refer to the floating point co-processor (or floating point emulator) registers 0 to 7.
- FPPSW – the Floating Point Processor (or emulator) Status Word.
- The lower-case equivalents of each of the above.

You can examine any of these registers with the `print` command and change them with the `let` command. However, when you assign to PC only bits 0..25 are affected; if you wish to change all the bits assign to R15 instead.

These symbols are defined in the root context, so if you have a variable called – say R0 – and you wish to refer to register 0 you can use the # character to specify this as follows:

```
print #r0
```

Registers command

The `registers` command displays the contents of ARM registers 0 to 15 and decodes the flags contained in register 15. The syntax is simply

```
re[gisters]
```

Examine command

This command allows you to examine a range of memory. The syntax is

```
e[xamine] [expr1][,[[+]expr2]]
```

The memory locations between `expr1` and `expr2` are displayed in hex and ASCII. `expr1` defaults to the memory location associated with the current context or the last memory location examined. `expr2` defaults to `expr1` + 128. If the variant form `+expr2` is used, the range between `expr1` and `expr1 + expr2` is examined.

List command

The `list` command displays a range of memory in instruction format. The syntax is

```
l[ist] [expr1][,[[+]expr2]]
```

The memory locations between `expr1` and `expr2` (or `expr1` and `expr1 + expr2`) are symbolically disassembled. `expr1` defaults as in the `examine` command. `expr2` defaults to `expr1` + 80.

Lsym command

The `lsym` command displays low-level symbols and their values. The syntax is

```
ls[ym] [sym]
```

`sym` is a prefix for the symbols to be listed. If `sym` is not specified all symbols are listed. For example `ls ma` might produce the following output.

```
malloc        = 0x0084a4
main          = 0x008098
```

Changing Memory

The syntax of the let command in the section Program Examination Commands was deliberately simplified. The full syntax is

```
[let] expression =|: expression {, expression}
```

If `expression` is an lvalue (ie the name of a variable) the `let` command behaves as before, changing the value of that variable. If the expression is an rvalue (ie a constant or a true expression) it is treated as a word address; memory at that and subsequent word locations is then assigned the values of the expressions on the righthand side of the `let` command.

This allows commands of the following form:

```
0x8008:0xfb000000, 0xeb000053
```

which sets the words at `0x8008` and `0x800c` to the listed values.

PC/pc command

This command sets all the bits of R15 (if given in upper case) or only the pc portion, bits 0 to 25 (if given in lower case). Its syntax is:

`PC|pc` *`expr`*

Miscellaneous commands

This section describes those commands that do not fall within the previous two groupings.

Help command

This command displays a list of available commands, or help on a particular command. Its syntax is:

`help [`*`command`*`]`

If the argument is omitted, a complete list of ASD commands is displayed. If the argument is present, that command's syntax and a brief description is printed. For example, `help print` will display:

`print[/<format>] <expr> Print result of <expr>. If <format>`... (the rest of the explanation is omitted here)

The help information uses angle brackets for items which would be shown in italics in this manual. `help` on its own lists all available commands; `help *` gives help about all available commands.

Base command

This command sets the numeric base to be used for numbers entered by the user. Its syntax is

`bas[e]` *`base`*

`base` is always specified in base 10, regardless of the current base. If *`base`* is 0, the base used to convert an input number will be 8, 10 or 16, depending on whether the number begins with a 0, a non-zero decimal digit, or 0x respectively (this is the same convention as that used in C).

Pcs command

This command selects the procedure call standard to be used. Its syntax is

`pcs [a|r]`

The RISC OS variant of the ARM procedure call standard is used by programs compiled under Release 3 of the C compiler, unless `-zkA` is given as a `cc` command line option. Previous releases use the obsolescent Arthur variant.

If you are debugging a program compiled with an earlier version of the C compiler you will need to use the command `pcs a` before you can use ASD to debug it.

Alias command

This command defines, undefines or lists aliases. Its syntax is

`alias [`*`name`*` [`*`expansion`*`]]`

If no arguments are given, all currently-defined aliases are displayed.

If *`expansion`* is not specified, the specified alias is deleted. Otherwise, *`expansion`* is assigned to the alias *`name`*. *`expansion`* may be enclosed in quotation marks to allow the inclusion of characters which you would otherwise not be able to include in an alias, (the alias expansion character ` and the statement separator `;`).

Aliases are expanded whenever a command line is about to be executed; the command list in a `do` clause is treated as a command line for this purpose.

Aliases are expanded in the following way:

Words consisting of alphanumeric characters enclosed in backquotes are expanded. If there is no corresponding alias the word is replaced with the null string. If the character following the closing backquote is non-alphanumeric, the closing backquote may be omitted. If the word is the first word of a command, the opening backquote may be omitted. To use a backquote in a command line, precede it with another backquote (ie use double backquote for a single backquote).

The alias command allows you to define your own commands. For example, you could define a command called `cstart` which would start a C program; it would be defined as follows:

```
alias cstart "br main; g; unbr main"
```

You can put aliases like these in an Obey file and execute it whenever you run ASD.

Language command

This command is used to tell the debugger what language rules it should obey.

The syntax of the command is:

```
la[nguage] [language]
```

where *language* is one of `f77`, `c`, `pascal` or `none`. The default (which is reverted to if the argument is omitted) is the language that the program's entry module is written in. In the present implementation, `f77`, `c` and `pascal` are equivalent. `language none` is used in conjunction with the low-level debugging facilities of ASD, described in the previous section.

Note that if `language` is set to `none`, loading a C program sets `language` to C.

Obey command

This command executes a set of debugger commands from a file, as if they had been typed at the keyboard. It has the form:

```
o[bey] command_file
```

The commands contained in the specified command file are executed.

Log command

This command causes subsequent typed commands, and their output, to be sent to a file as well as the screen. The format of the command is:

```
lo[g] [filename]
```

To start logging, use the form with the `filename`. For example:

```
log logfile
```

The file will be opened, and a couple of introductory lines sent to it. Thereafter, all user input and command output (excluding `ASD:` prompts) will be sent to the file.

To terminate logging, type `log` without an argument.

The `log` command is useful for capturing the output after – for example – a `ptrace` command, enabling the flow of control to be examined at leisure using an editor such as Twin or Edit.

Quit command

This causes the debugging session to be terminated. It also closes any open `log` and `obey` files. The syntax is:

```
q[uit]
```

* command

Any command whose first non-space character is `*` will be sent to the operating system for execution. This gives access to the RISC OS Command Line interpreter. For example:

```
*cat c
```

An example ASD session

The following example debugging session shows how ASD might be used to fix a rather bug-ridden file-sorting utility. It is not an attempt to demonstrate all the features of ASD.

```
c.sort
----------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "kernel.h"

#define READATTR  5
#define READFILE  16
#define WRITEFILE 0

#define FILEFOUND 1

extern int cistrcmp(char *a, char *b);

#ifndef NOCISTRCMP
/*
 * Rewritten in assembler
 */
int cistrcmp(char *a, char *b)
{
    int ca, cb;
    do {
        if ((ca = *a++ - 'A') < 'Z' - 'A' + 1)
            ca += 'a' - 'A';
        if ((cb = *b++ - 'A') < 'Z' - 'A' + 1)
            cb += 'a' - 'A';
        if (cb = ca - cb)
            return cb;
    } while (cb = ca + 'A');
    return cb;
}
```

```
#endif

void fail(char *msg)
{
    fputs(msg, stderr);
    exit(1);
}

/* See Sedgewick: Algorithms 2nd edition P 108 */
static void sortstrings(char *a[], int n)
{
    int  h, i, j;
    char *v;
    h = 1;
    do
        h = h * 3 + 1;
    while (h <= n);
    do {
        h = h / 3;
        for (i = h + 1; i <= n; i++) {
            v = a[i];
            j = i;
            while (j > h && cistrcmp(a[j-h], v) > 0) {
                a[j] = a[j-h];
                j -= h;
            }
            a[j] = v;
        }
    } while (h > 1);
}

void sortfile(char *f)
{
    _kernel_osfile_block finfo;
    int size;
    char *finbuff, *foutbuff;
    char *cp;
    int  l, linestart;
    char **lbuff;
    int  i;
    if (_kernel_osfile(READATTR, f, &finfo) != FILEFOUND)
        fail("File not found\n");
    size = finfo.start;
    if (!(finbuff = malloc(size + 1)) || !(foutbuff = malloc(size + 1)))
        fail("Out of memory\n");
    finfo.load = (int) finbuff;
    finfo.exec = 0;
    if (_kernel_osfile(READFILE, f, &finfo) < 0)
        fail("Error reading file\n");
    l = 0;
    cp = finbuff;
    linestart = 1;
    for (i = 0; i < size; i++) {
        if (linestart) {
```

```
            l++;
            linestart = 0;
        }
        if (!*cp || *cp == '\n') {
            *cp = 0;
            linestart = 1;
        }
        cp++;
    }
    *(finbuff + size) = 0;
    if (!(lbuff = malloc(l * sizeof(char *))))
        fail("Out of memory\n");
    cp = finbuff;
    for (i = 0; i < l; i++) {
        lbuff[i] = cp;
        cp += strlen(cp);
    }
    sortstrings(lbuff, l);
    cp = foutbuff;
    for (i = 0; i < l; i++) {
        strcpy(cp, lbuff[i]);
        cp += strlen(cp);
        *cp++ = '\n';
    }
    finfo.start = (int) foutbuff;
    finfo.end = (int) foutbuff + size;
    if (_kernel_osfile(WRITEFILE, f, &finfo) < 0)
        fail("Error writing file\n");
    free(finbuff);
    free(foutbuff);
    free(lbuff);
}

int main(int argc, char *argv[])
{
    int i;
    if (argc < 2)
        fail("Usage: sort <filename>\n");
    for (i = 1; i < argc; i++)
        sortfile(argv[i]);
    return 0;
}
----------------
```

The shellsort algorithm used above may be found in *Algorithms* by Robert Sedgewick, second edition p108 (first edition p98). The original algorithm, in Pascal, is shown below.

```
procedure shellsort;

label 0;

var   i,j,h,v:integer;
```

```
    begin
        h:=1;
        repeat
            h:=3*h+1
        until h>N;
        repeat
            h:=h div 3;
            for i:=h+1 to N do
                do begin
                    v:=a[i];
                    j:=i;
                    while a[j-h]>v do
                        do begin
                                a[j]:=a[j-h];
                                j:=j-h;
                                if j<=h
                                    then goto 0
                            end;
0:                  a[j]:=v;
                    end
            until h=1
        end;
```

Compile the C program with debugging information included using the command

```
*cc -g -c c.sort
```

The compiler will give several warnings; you can ignore these.

If you are not used to assembly language programming, skip to the paragraph on the next page that begins 'The next step is to link the program'.

If you are interested in the low-level debugging capabilities of ASD you may like to leave out the C version of `cistrcmp` by using the command

```
*cc -g -c -DNOCISTRCMP c.sort
```

and use the following assembly language routine instead:

```
s.cistrcmp
----------
r0      RN      0
r1      RN      1
r2      RN      2
r3      RN      3
lr      RN      14
pc      RN      15
        AREA    |cistrcmp|, CODE, COMDEF, READONLY
```

```
        MOV     r3,r0
cistrcmp1
        LDRB    r0,[r1],#1
        LDRB    r2,[r3],#1
        SUB     r2,r2,#"A"
        CMP     r2,#"Z"-"A"+1
        ADDCC   r2,r2,#"a"-"A"
        SUB     r0,r0,#"A"
        CMP     r0,#"Z"-"A"+1
        ADDCC   r0,r0,#"a"-"A"
        SUBS    r0,r2,r0
        MOVNES  pc,lr
        ADD     r0,r2,#"A"
        BNE     cistrcmp1
        MOVS    pc,lr

        END
---------
```

To assemble this use the command

```
*objasm s.cistrcmp -to o.cistrcmp -stamp -quit
```

If you do not have a copy of `objasm` there is a pre-assembled object in the file `o.cistrcmp` on Disc 1 of C Release 3.

The next step is to link the program. To do this use the command

```
*link -deb -o sort o.sort $.clib.o.ansilib
```

if you are using the C version of `cistrcmp`, or

```
*link -deb -o sort o.sort o.cistrcmp $.clib.o.ansilib
```

if you are using the assembly language version. You should now have an executable file called `sort` in the current directory.

If you are using a single floppy disc system you will need to copy `$.clib.o.ansilib` from Disc 2 of the release. You can do this via RAMFS (as described in the section entitled *Compiling and running the example programs* in the chapter *How to install and run the compiler*. Alternatively, you can link with `$.clib.o.stubs`, in which case the only difference will be that the symbolic backtrace you get when you try to run the program will look a little different from that shown below. To do this, use:

```
*link -deb -o sort o.sort $.clib.o.stubs
```

or

```
*link -deb -o sort o.sort o.cistrcmp $.clib.o.stubs
```

The sort program will overwrite its input file, so you might like to retain a copy of the original file so that you can repeat the test and make subsequent tests on the same input data. To copy it use the command

```
*copy sortinput test
```

Now try running the program with the command

```
*sort test
```

This should produce something like the result shown below. This is called a *symbolic backtrace*.

```
Illegal address (e.g. wildly outside array bounds)

Postmortem requested
  Arg1: 0x00000005 5
Function name _real_default_signal_handler
  Arg1: 0x00000005 5
Function name raise
  Arg2: 0x0000000c 12
  Arg1: 0x0001a420 107552
Function name sortstrings
  Arg1: 0x000186a1 100001
Function name sortfile
  Arg2: 0x00018688 99976
  Arg1: 0x00000002 2
Function name main
  Arg2: 0x0000841c 33820 -> [0xe1a0c00d 0xe92dd8f3 0xe24cb004 0xe15d000a]
  Arg1: 0x00000ad8 2776
Function name _main
```

The first line gives a general indication of what might be wrong with your program. In this case it's an illegal address; your program tried to access memory which is outside the addressing range of your computer. Each line starting with `Function name` represents a procedure call `frame` on the stack. The first two, `_real_default_signal_handler` and `raise`, are just internal routines that are called when an exception is raised. The first recognisable line begins `Function name sortstrings`; this is where the illegal address was referenced.

This doesn't look too promising so try running it under ASD to get more clues as to what might be wrong. To run ASD type `*asd` followed by the name of the program you wish to debug, followed by any arguments that program might take. In this case, type the command

```
*asd sort test
```

which should produce the following

```
Acorn source-level debugger, version 3.00
Object program file sort
```

The program crashed in the function `sortstrings`. Since we want the program to stop before making the illegal access, we want to stop at the beginning of `sortstrings`. To do this use the command

```
ASD: br sortstrings
```

`br` stands for break. Since ASD allows minimum abbreviations, you can use any of the commands `b`, `br`, `bre`, `brea` or `break` here. The break command places a special instruction called a *breakpoint* at the specified symbolic location or *context*. A context is just the name of a location within a program; in this case the start of the procedure `sortstrings`. The word `sortstrings` in the break command is an abbreviation for the full name `#sortstrings:$entry` but ASD allows abbreviations provided they do not introduce ambiguity.

As a general rule this is the best way to start a debugging session. By placing a breakpoint just before the section of code we think is wrong (or after the code we know to be correct) we can examine the program state to ensure it is correct and then step through the incorrect code to find exactly where the error is occurring.

Having set the breakpoint we now tell ASD to start executing our program using the command `go`.

```
ASD: go
Stopped at breakpoint #1 in sortstrings, line 43 of c.sort
        43 {
```

The program has now stopped at the beginning of `sortstrings` and control is returned to us at the ASD prompt. Now we want to examine the program state to ensure it is correct before continuing. In this case the most important state information is the function arguments. We can examine them with the command `arguments` (or `arg`).

```
ASD: arg
a 0001a420
n 12
```

There are two arguments to `sortstrings`. `n` is the number of strings to sort, in this case 12. This is correct since there were 12 names in the input file. `a` is an unbounded array of char *s (strings). Since it is unbounded, ASD has no way of knowing its size, so ASD just prints its address instead of printing the contents as it would with a bounded array. However, we can examine the individual elements of `a`, using the `print` command.

```
ASD: pr a[0]
string "Noel"
ASD: pr a[1]
0001a4ac
```

The first element was correct: it contained the string `Noel` which is the first name in the input file. However, the second element just prints a memory address; ASD was expecting to find a string at `a[1]`, but didn't. To find out what it *did* find at `a[1]` use the `examine` command as follows:

```
ASD: ex a[1]
0x0001a4ac: 0x77644500, 0x00647261, 0x64657246, 0x61724600  ".Edward.Fred.Fra"
0x0001a4bc: 0x7369636e, 0x6e614900, 0x65654c00, 0x72614800  "ncis.Ian.Lee.Har"
0x0001a4cc: 0x4a007972, 0x53006d69, 0x6c696568, 0x6f520061  "ry.Jim.Sheila.Ro"
0x0001a4dc: 0x00726567, 0x6e6f694c, 0x4d006c65, 0x69747261  "ger.Lionel.Marti"
0x0001a4ec: 0xe590006e, 0x3e694c3c, 0x00000000, 0xeb0034fe  "n..e<Li>....~4.k"
0x0001a4fc: 0xe59f0070, 0xe5901000, 0xe1a02004, 0xe3a00002  "p..e...e.  a.. c"
0x0001a50c: 0xeb0034f9, 0xe59f0060, 0xe5901000, 0xe1a02004  "y4.k`..e...e.  a"
0x0001a51c: 0xe3a000b5, 0xeb0034f4, 0xe59f0050, 0xe5901000  "5. ct4.kP..e...e"
```

This shows that ASD found a null string at address `0001a4ac`. Here we can easily see that the first byte pointed to is 0. Being suspicious, we wonder what the other elements of `a` point to. We could use `pr a[2]; pr a[3]; ... pr a[11]` to find out. However, it is just as easy to examine the block of memory pointed to by `a`.

```
ASD: ex a
0x0001a420: 0x0001a4a8, 0x0001a4ac, 0x0001a4ac, 0x0001a4ac  "($..,$..,$..,$.."
0x0001a430: 0x0001a4ac, 0x0001a4ac, 0x0001a4ac, 0x0001a4ac  ",$..,$..,$..,$.."
0x0001a440: 0x0001a4ac, 0x0001a4ac, 0x0001a4ac, 0x0001a4ac  ",$..,$..,$..,$.."
0x0001a450: 0x3e694c3c, 0x40000048, 0xe1a02005, 0xeb000807  "<Li>H..@.  a...k"
0x0001a460: 0xe1a0000d, 0xe3540000, 0x059f102c, 0x159f102c  ".. a..Tc,...,..."
0x0001a470: 0xe1a02005, 0xeb000801, 0xe28d1028, 0xe3a00012  ".  a...k(..b.. c"
0x0001a480: 0xeb000198, 0xe1a0100d, 0xe3a00018, 0xeb000195  "...k.. a.. c...k"
0x0001a490: 0xe95ba830, 0x0002a2ac, 0x0002a2d4, 0x0002a284  "0([i,"..T"..."..."
```

Now we can see that all elements (except the first) point to the 0 byte at `0001a4ac`. This means that the arguments to `sortstrings` were wrong; the error therefore occurred earlier. Now try rerunning the program but setting the breakpoint earlier. We could quit ASD and rerun the program, but ASD provides a `load` command which will load an executable image.

```
ASD: load sort test
```

Now set the breakpoint at `sortfile` instead of `sortstrings` and start execution.

```
ASD: br sortfile
ASD: g
Stopped at breakpoint #1 in sortfile, line 66 of c.sort
   66 {
```

Looking at the source we see that `lbuff` is passed as the first argument (`a`) to `sortstrings`. `lbuff` is initialised in the loop just before the call to `sortstrings` so we would like to stop just after the assignment to `lbuff[i]`. We therefore want to set a breakpoint on the following line, but we don't have any line numbers in the listing above. We could rush off and get a listing with line numbers or try counting the lines from the start of the program but ASD can do better than that. Since the variable `lbuff` is initialised just before the loop in which we wish to break, by using the `watch` command we can get ASD to stop the next time `lbuff` is changed.

```
ASD: wa lbuff
ASD: g
Watchpoint #1 at lbuff changed by sortfile, line 99 of c.sort
   99      if (!(lbuff = malloc(l * sizeof(char *))))
```

Now we know the `lbuff` initialisation line is 99 we can either count forward a few lines or use ASD's type command to find the line at which we want to break.

```
ASD: ty 98,105
   99      if (!(lbuff = malloc(l * sizeof(char *))))
  100          fail("Out of memory\n");
  101      cp = finbuff;
  102      for (i = 0; i < l; i++) {
  103          lbuff[i] = cp;
  104          cp += strlen(cp);
  105      }
  106      sortstrings(lbuff, l);
```

That is line 104, so set a breakpoint there.

```
ASD: br 104
ASD: g
Stopped at breakpoint #2 in sortfile, line 104 of c.sort
  104         cp += strlen(cp);
```

Find the value of `lbuff[0] (= cp).`

```
ASD: pr cp
string "Noel"
```

It's the first name in the input file, as we expected, so try stepping over the update of `cp` to see what value it gets next.

```
ASD: s
Stepped to sortfile, line 102 of c.sort
  102     for (i = 0; i < l; i++) {
ASD: pr cp
0001a4ac
```

The update assignment is wrong. After careful study of line 104 we see that we have omitted to count in the 0 byte when updating `cp`. The line should read

```
  104         cp += strlen(cp) + 1;
```

Quit ASD (with the `quit` command), edit the file `c.sort`, fix line 104, recompile `sort.c`, relink and try again.

```
ASD: q
Quitting

*sort test
Illegal address (e.g. wildly outside array bounds)
Postmortem requested
  Arg1: 0x00000005 5
Function name _real_default_signal_handler
  Arg1: 0x00000005 5
Function name raise
  Arg2: 0x0000000c 12
  Arg1: 0x0001a424 107556
Function name sortstrings
  Arg1: 0x000186a5 100005
Function name sortfile
  Arg2: 0x0001868c 99980
  Arg1: 0x00000002 2
Function name main
  Arg2: 0x00008420 33824 -> [0xe1a0c00d 0xe92dd8f3 0xe24cb004 0xe15d000a]
  Arg1: 0x00000ad8 2776
Function name _main
```

The problem is the same one. Start up ASD:

```
*asd sort test
ARM source-level debugger, version 1.00
Object program file sort
```

Set a breakpoint at the start of `sortstrings` and start execution:

```
ASD: br sortstrings; g
Stopped at breakpoint #1 in sortstrings, line 43 of c.sort
   43 {
```

Take a look at the arguments:

```
ASD: arg
a 0001a424
n 12
```

Look at the individual elements of `a`:

```
ASD: pr *a
string "Noel"
ASD: pr *(a+1)
string "Edward"
ASD: pr *(a+11)
string "Martin"
```

They're OK now, so something is wrong with the sort algorithm. Try setting a breakpoint on the inner `while` loop. Use the `type` command to find the line number:

```
ASD: ty 50,60
   50      while (h <= n);
   51      do {
   52          h = h / 3;
   53          for (i = h + 1; i <= n; i++) {
   54              v = a[i];
   55              j = i;
   56              while (j > h && cistrcmp(a[j-h], v) > 0) {
   57                  a[j] = a[j-h];
   58                  j -= h;
   59              }
   60              a[j] = v;
```

The breakpoint must be set on line 56.

```
ASD: br 56; g
Stopped at breakpoint #2 in sortstrings, line 56 of c.sort
   56              while (j > h && cistrcmp(a[j-h], v) > 0) {
```

Examine a few variables:

```
ASD: pr j; pr h
5
4
```

They're both right, so look at the contents of `a[j-h]`:

```
ASD: pr a[j-h]
string "Edward"
```

From our knowledge of the algorithm, it should be comparing against the first string. Looking closely at the Pascal version of the algorithm we see that it was written using 1 origin arrays, and has been rather literally transcribed into C which uses 0 origin arrays. To fix it, we could subtract 1 from each array index. However we just want a quick fix to see if it works, so after line 46 add the following line:

```
47      a--;    /* Quick hack to make array 1 origin - fixme */
```

This may not be portable on some segmented architectures so don't try it on your PC emulator. Fortunately the ARM is non-segmented.

Quit, edit, compile, link and test again:

```
*sort test
```

Well, there was no stack backtrace that time, but did it sort the file?

```
*type test
```

What you see now depends on whether you used the assembly language version of `cistrcmp` or the C version. If you used the C version the file should be sorted correctly, but if you used the assembly language version it will look something like this:

```
Edward
Francis
Fred
Harry
Ian
Jim
Lionel
Lee
Martin
Noel
Roger
Sheila
```

Lionel and Lee are in the wrong order, so back to ASD yet again. But before that we had better restore the original input file.

```
*copy sortinput test
*asd sort test
ARM source-level debugger, version 1.00
Object program file sort
```

We have a fairly good idea of where the problem is since it worked with the C version of `cistrcmp` and didn't work with the assembly language version. However, suppose that we didn't know that. Given that the output is almost but not quite sorted correctly we would naturally have suspicions about the comparison function. We could replace `cistrcmp` with `strcmp` in the source, recompile, relink and compare the output but there is an easier way. We want to substitute `strcmp` for `cistrcmp`, so we set a breakpoint on the first instruction of `cistrcmp`; when that breakpoint occurs we set the `PC = strcmp and` continue. Fortunately this can all be done with one command.

```
ASD: br @cistrcmp do {pc strcmp; g}
```

The `do` clause on the break command is executed whenever the breakpoint occurs.

Note the use of `@cistrcmp` here instead of `cistrcmp`. `@cistrcmp` refers to a low-level symbol, `cistrcmp` refers to a high-level symbol. Both may be present together. In this case there is only one `cistrcmp` since it was generated by an assembly language routine but you still need to use an `@` symbol before it. If both symbols existed, it would be fatal to use `cistrcmp` instead of `@cistrcmp`. High-level procedure symbols point a few words into the procedure (after the frame initialisation); low-level procedure symbols point to the first instruction, which is where we want to break.

```
ASD: g
Program terminated normally
```

Well, the program finished OK, so let's look at the output:

```
ASD: *type test
Edward
Francis
Fred
Harry
Ian
Jim
Lee
Lionel
Martin
Noel
Roger
Sheila
```

It is sorted correctly so the problem is with our assembly language `cistrcmp`. Copy the input file again and reload the image.

```
ASD: *copy sortinput test
ASD: load sort test
```

Now we must tell ASD we want to debug an assembly language procedure. To do this we use the `language none` command. We'll also select hexadecimal.

```
ASD: lang none; base 16
```

Now we'll take a look at that `cistrcmp` routine.

```
ASD: l cistrcmp
cistrcmp$$Base
 0x00013a98: 0xe1a03000 .0 a mov    r3,r0
 0x00013a9c: 0xe4d10001 ..Qd ldrb   r0,[r1],#1
 0x00013aa0: 0xe4d32001 . Sd ldrb   r2,[r3],#1
 0x00013aa4: 0xe2422041 A Bb sub    r2,r2,#&41
 0x00013aa8: 0xe352001a ..Rc cmp    r2,#&1a
 0x00013aac: 0x32822020   .2 addcc  r2,r2,#&20
 0x00013ab0: 0xe2400041 A.@b sub    r0,r0,#&41
 0x00013ab4: 0xe350001a ..Pc cmp    r0,#&1a
 0x00013ab8: 0x32800020  ..2 addcc  r0,r0,#&20
 0x00013abc: 0xe0520000 ..R` subs   r0,r2,r0
 0x00013ac0: 0x11b0f00e .p0. movnes pc,r14
 0x00013ac4: 0xe2820041 A..b add    r0,r2,#&41
 0x00013ac8: 0x1afffff3 s... bne    &00013a9c    (cistrcmp$$Base + 0x4)
 0x00013acc: 0xe1b0f00e .p0a movs   pc,r14
RTSK$$Data
 0x00013ad0: 0x00000028 (... andeq  r0,r0,r8,lsr #32
 0x00013ad4: 0x00008080 .... andeq  r8,r0,r0,lsl #1
 0x00013ad8: 0x000122dc \".. muleq  r1,r12,r2
 0x00013adc: 0x000084a0  ... andeq  r8,r0,r0,lsr #9
 0x00013ae0: 0x000084a8 (... andeq  r8,r0,r8,lsr #9
 0x00013ae4: 0x00000000 .... andeq  r0,r0,r0
```

The problem seemed to be that it only compares the first letter correctly so we'll set a breakpoint immediately after we find that the first letters are equal. That is at location `0x13ac4`.

```
ASD: br @13ac4
ASD: g
Stopped at breakpoint #1 in $ROOT
cistrcmp$$Base + 0x2c
>0x00013ac4: 0xe2820041 A..b add    r0,r2,#&41
```

Take a look at the registers:

```
ASD: r
  R0  = 0x00000000  R1  = 0x0001a4c6  R2  = 0x00000025  R3  = 0x0001a4c1
  R4  = 0x0001a428  R5  = 0x0000000c  R6  = 0x00000001  R7  = 0x00000003
  R8  = 0x00000004  R9  = 0x0001a4c5  R10 = 0x00014734  R11 = 0x00018448
  R12 = 0x00000003  R13 = 0x0001841c  R14 = 0x20008178  R15 = 0x60013ac4
  Flags: N = 0, Z = 1, C = 1, V = 0
```

`R0 = 0` since the first letters were equal. `R2` is some letter – 'A'. So to find out what letter we type

```
ASD: pr/%c r2+'A'
f
```

It's the letter 'F', so it's comparing Fred and Francis. Let's step on:

```
ASD: s
Stepped to $ROOT
cistrcmp$$Base + 0x30
>0x00013ac8: 0x1afffff3 s... bne     &00013a9c    (cistrcmp$$Base + 0x4)
```

and take a look at the registers:

```
ASD: r
  R0  = 0x00000066  R1  = 0x0001a4c6  R2  = 0x00000025  R3  = 0x0001a4c1
  R4  = 0x0001a428  R5  = 0x0000000c  R6  = 0x00000001  R7  = 0x00000003
  R8  = 0x00000004  R9  = 0x0001a4c5  R10 = 0x00014734  R11 = 0x00018448
  R12 = 0x00000003  R13 = 0x0001841c  R14 = 0x20008178  R15 = 0x60013ac8
  Flags: N = 0, Z = 1, C = 1, V = 0
```

The Z flag is set, so it's going to fall through the BNE. This is wrong; the routine should be looping back to compare the rest of the characters. Studying the previous instruction (which was supposed to set the Z flag) we notice that we have omitted the S on the `ADD` instruction which tells the ARM to set the flags based on the result of the instruction. So that line should read:

```
        ADDS    r0,r2,#"A"
```

So exit, edit, reassemble and relink. The sort program should now work. Alternatively you can just use the C version of `cistrcmp`, since it is exactly the same size (when not compiled `-g`) and runs just as fast.

Command summary

This section lists all the ASD commands in alphabetical order giving the minimum abbreviation and a brief description for each.

`al[ias]`	Define, undefine or list aliases
`a[rguments]`	Display arguments of current procedure

`ba[cktrace]`	Display stack-frame history
`bas[e]`	Set the numeric base for integer constants
`b[reak]`	Set a breakpoint or display all breakpoints
`ca[ll]`	Call a procedure or function
`cm[dline]`	Set up arguments for debuggee
`co[ntext]`	Set or reset the current context
`e[xamine]`	Examine memory contents
`fo[rmat]`	Set default print format for integers
`fp[registers]`	Display contents of floating point registers
`g[o]`	Start or resume execution of the program
`h[elp]`	Display general or specific help information
`i[n]`	Set context to current context's caller
`la[nguage]`	Set current language name
`le[t]`	Assign value to a variable
`l[ist]`	Disassemble memory
`loa[d]`	Load an image for debugging
`lo[g]`	Open a log file storing ASD commands and output
`o[bey]`	Execute the command lines stored in a text file
`ou[t]`	Set context to current context's caller
`PC` or `pc`	Set all or pc bits, respectively, of R15
`pcs`	Set procedure call standard
`pr[int]`	Display result of an arbitrary expression
`p[trace]`	Enable or disable procedure tracing

`q[uit]`	Leave the debugger, returning to the OS
`r[egisters]`	Display contents of ARM registers
`ret[urn]`	Return from active procedure, with optional result
`s[tep]`	Single step by one or n statements
`sy[mbols]`	Display variables in current context
`t[ype]`	Type portion of a text file
`unb[reak]`	Clear a breakpoint
`unw[atch]`	Clear a watchpoint
`v[ariable]`	Display information about a variable
`void`	Call a procedure without printing a result
`w[atch]`	Set a watchpoint or display all watchpoints
`wh[ere]`	Display current context
`whi[le]`	Conditionally re-execute current line

Other utilities

This chapter describes two utilities: The Acorn Make Utility (AMU), which assists with the management of programs made from several source and object files, and Squeeze, which compresses runnable programs, typically to about half their original size.

The Acorn Make Utility

AMU assists with the management of programs, documents, applications, and other complex, structured objects made from several components, each of which needs to be translated or processed in some way, and which have some consistency constraints between them. Most often, it is used to help manage programs and the rest of this section is devoted to that application of it.

The input to AMU is a description, prepared by the user, of the system to be managed. The description is written in a stylised way in a text file usually called `makefile`. Of course, you can use any name you like, but AMU doesn't have to be told to look for `makefile` and the use of this name is well established in the programming community so if you use it too, others will immediately understand what you are doing.

In its simplest form, a makefile consists of a sequence of entries which describe:

- what each component of a system depends on
- what commands to execute to make an up-to-date version of that component.

Everything else that you can express in a makefile is conceptually inessential, designed to make the job of description easier for you.

AMU performs two functions for you. Firstly, it expands your description into the simple form just described: a sequence of explicit rules about how to make each component of a system. Then it decides which rules need to be applied to make a completely up-to-date, consistent system. This it does by deciding which components are older than any of the files they depend on. It then executes the commands associated with those entries, in an appropriate order.

An example will make all this clear, so let's look at part of the makefile for AMU itself:

```
amu:          o.amu $.301.clx.o.clxlib
              Link -o amu o.amu $.CLib.o.Stubs
              squeeze amu

o.amu:        c.amu $.301.clx.o.clxlib
              cc -I$.301.clx c.amu

install:
              copy amu %.amu ~cfq
              remove amu
              remove o.amu
```

Each entry consists of a target, followed by a colon character, followed by a list of files on which the target depends, then followed by a list of commands to execute to make the target up to date. Each command line begins with some white space (if you want your makefile to be portable to UNIX systems you should begin these lines with a Tab character). For example, `amu` itself is made from `o.amu`, the compiled AMU program, and a proprietary library called `$.301.clx.o.clxlib` (on the author's computer). If either of these files is newer than `amu`, or if `amu` does not yet exist, then the commands `Link -o amu...` followed by `Squeeze amu`, should be executed.

But what if `o.amu` doesn't yet exist or is not itself up to date? AMU will check this for you and will not use `o.amu` without first making it up to date. To do this it will execute the command(s) associated with the `o.amu` entry.

Thus AMU might well execute for you:

```
cc -I$.301.clx c.amu
Link -o amu o.amu $.CLib.o.Stubs
squeeze amu
```

As you can see, if you do this more than once – for example, because you are developing the program being managed by AMU – it will save you many keystrokes! Now suppose you don't have `$.301.clx.o.clxlib`. What then? Well, the makefile doesn't instruct AMU how to make this so it can do no more than tell you so. Either you must modify the makefile to say how to make it or, more likely, obtain a copy ready-made.

Finally, observe the entry beginning `install:`. This doesn't appear to be connected with any other entry. In fact, it isn't, but if you were to use the command `AMU install`, AMU would try to make the 'install' thing rather than the 'amu' thing (unless you say otherwise, AMU tries to make the first target in the makefile). Now, `install` depends on nothing, so AMU unconditionally executes the commands associated with it, which copy `amu` to the library and remove the binary and the object files from the local directory.

A precise description of a makefile is given below in the section entitled *The makefile*.

The AMU command

The AMU command has the following syntax:

AMU *options target1 target2...*

options are as follows:

-f *makefile*

Read the system description from *makefile* (*makefile* defaults to `makefile` if omitted).

-i

Ignore return codes from commands (equivalent to `.IGNORE`). AMU usually stops if it encounters a bad (non-0) return code.

-k

On encountering a bad (non-0) return code, don't give up, but continue with each branch of the makefile that doesn't depend on the failing command. For example, the C compiler is made from 28 separate object files. After making a major modification which touches many files it would be usual to use `AMU -k`, as each compilation is independent and there is probably little reason to abandon work just because one or two fail. However, if any compilations fail, the link step must be abandoned, as this depends on all compilations succeeding. `AMU -k` does just what is required.

`AMU -k` and `AMU -i` are subtly different. `AMU -k` is appropriate when commands set return codes properly and you want AMU to do as much as possible while you get on with something else. `AMU -i` is strictly for commands that don't or can't set the return code appropriately (for example, textual difference programs traditionally set the return code to 1 to indicate successfully finding differences, and to 2 to indicate failures such as a file not being found).

`-n`

Don't execute any commands; just show on the screen what commands would be executed, giving a reason for wanting to execute each one.

`-o` *`cmdfile`*

Don't execute commands to make the target(s) up to date; write them to *`cmdfile`* for later execution using `*Exec` *`cmdfile`* or `*Obey` *`cmdfile`*. For example, on the author's computer, the makefile for the shared C library contains an 'install' entry which *RMKILLs SharedCLibrary and re-installs the new one. However, it is a bad idea to do this while AMU – which uses the shared C library – is running! It is much safer to write the commands to a file and *Exec them.

`-s`

Don't echo commands to be executed (equivalent to `.SILENT`). Usually, AMU is reassuringly chatty. This will shut it up (but not the commands it executes, the loquacity of which cannot be controlled by AMU).

`-t`

Generate commands to make target(s) up to date by setting source time stamps consistently (only guaranteed to succeed if all sources exist). The *Stamp command is used to set time stamps.

`target1 target2 ...`

A list of targets to be made or macro pre-definitions of the form *name=string*. Targets are made in the order given. If no targets are given, the first target found in `makefile` is used.

Examples:

```
AMU ucc CC=cc160a
AMU Link=Lnk650Exp
AMU install
```

The makefile

A makefile consists of a sequence of logical lines. A logical line may be continued over several physical lines provided each but the last line ends with a \. For example:

```
# This is a comment line \
   continued on the next physical line \
   and on the next, but not thereafter.
```

Comments are ignored by AMU. A comment is introduced by a hash character # and runs to the end of the logical line.

Otherwise there are four kinds of non-empty logical lines in a makefile:

- dependency lines
- command lines
- macro definition lines
- rule and other special lines.

Dependency lines have the form:

```
space-separated-list-of-targets COLON space-separated-list-of-prerequisites.
```

For example:

```
amu :  o.amu $.301.clx.o.clxlib
o.d35 o.d36 o.d37: h.util
```

A dependency line cannot begin with white space. Spaces before the `:` are optional, but some white space must follow to distinguish `:` separating targets and prerequisites from `:` as part of a RISC OS filename.

For example:

```
adfs::4.$.library.amu: o.amu ...
```

(Although a space after the `:` is not required by UNIX's make utility, omission of it is rare in UNIX makefiles).

A line with multiple targets is shorthand for several lines, each with one target and the same righthand side (and the same associated commands, if any). Multiple dependency lines referring to the same target accumulate, though only one such line may have commands associated with it (AMU would not know in what order to execute the commands otherwise). For example:

```
amu:    o.amu
amu:    $.301.clx.o.clxlib
```

is exactly equivalent to the single line form given earlier. In general, the single line form is easier for you to write whereas the multi-line form is more readily generated by a program (for example, `cc -M c.foo` will generate a list of lines of the form `o.foo:` *`h.thing`*, one for each #include *`thing.h`* in `c.foo`). Command lines immediately follow a dependency line and begin with white space.

For maximum compatibility with UNIX makefiles ensure that the first character of every command line is a Tab. Otherwise one or more spaces will do. A semi-colon may be used instead of a new line to introduce commands. This is often used when there are no prerequisites and only a single command associated with a target. For example:

```
clean:; wipe o.* ~cfq
```

Note that, in this case, no white space need follow the `:`.

Macro definition lines are lines of the form:

`macro-name = some text to the end of the logical line`

For example:

```
CC    = ncc
CFLAGS= -fah -c -I$.clib
LD    = Link
LIB   = $.CLib.o.clxlib $.CLib.o.Stubs
CLX   = $.301.clx
```

The = can be surrounded with white space, or not, to taste. Thereafter, wherever `${name}` or `$(name)` is encountered, if *name* is the name of a macro then the whole of `${name}` is replaced by its definition. A reference to an undefined macro simply vanishes. An example which uses the above macro definitions, and which is taken from the makefile for AMU itself, is:

```
amu:            amu.o $(CLX).o.clxlib
                $(LD) -o amu ${LFLAGS} o.amu ${LIB}
```

which expands to

```
amu:            amu.o $.301.clx.o.clxlib
                Link -o amu o.amu $.CLib.o.clxlib $.CLib.o.Stubs
```

Note that `${LFLAGS}` expands to nothing.

Macros can also be defined on AMU's command line. For example:

```
* AMU "LFLAGS=-v -map -xref"
```

would be equivalent to a line

```
LFLAGS=-v -map -xref
```

at the beginning of the makefile (the additional quotes tell the C library's command line processor to treat this whole argument as a single word, even though it contains spaces).

By using macros intelligently, you can minimise the effort needed to move makefiles from computer to computer, dealing with varying locations for prerequisites, for example; or you can just centralise what would otherwise be distributed through many lines of text. It is obviously much easier to add `-g` to a `CFLAGS=` line to make a debuggable version of the compiler than it is to add `-g` to 28 separate `cc` commands! Similarly, using `$(CC)` and `CC=cc`, rather than just `cc`, makes it very easy to use a different version of `cc`; just change the definition of the macro. Whilst this may not seem very useful in a small makefile, it is common practice when describing larger systems such as the C compiler.

Command execution

AMU executes commands by calling the C library function `system`, once for each command to be executed. In turn, `system` issues an OS_CLI SWI to execute the command. Before calling OS_CLI, `system` copies its caller to the top end of application workspace and sets the workspace limit just below the copied program. Any command executed by AMU therefore has less memory to execute in than AMU had initially (the difference being the size of AMU plus the size of AMU's working space).

When the command returns, AMU will be copied back to its original location and will continue, unless, of course, the command set a bad (non-0) value in the environmental variable Sys$ReturnCode (the C library automatically sets Sys$ReturnCode to the value returned by `main()` or passed to `exit()`). If you have limited memory on your computer, or you are trying to run AMU in a limited wimp slot under the desktop, and a program (such as the C compiler) to be run by AMU needs more memory than is left, you can instruct AMU not to execute commands directly, but to write them to a file to be executed later (see the `-o` option described above). Of course, in this case, execution is not terminated or modified (for example, AMU `-i`, described above) by a non-0 return code from a command.

As noted earlier, `AMU -o` is also appropriate when one of the commands would otherwise perturb the running AMU (for example, by installing a new shared C library module in your computer).

Finally, note that there is a RISC OS command length limit of 255 characters. This is imposed by the OS_CLI SWI and is warned of by AMU if you try to exceed it. This limit may be found troublesome when importing makefiles from other environments such as UNIX (where the corresponding limit is often 10Kb!). A common cause of problems here is very big link commands, referring to many object files. To avoid this limitation, many Acorn utilities will accept either an input pattern or an input file containing a list of filenames. The linker, in fact, accepts both (see the chapter entitled *The Linker* for further details).

Advanced features

File naming

To help you move MS-DOS and UNIX makefiles to RISC OS, or to develop makefiles under RISC OS for export to MS-DOS or UNIX, both AMU and the C compiler accept three styles of file naming:

RISC OS native:	`$.301.cfe.c.pp`	`^.include.h.defs`
UNIX-like:	`/301/cfe/pp.c`	`../include/defs.h`
MS-DOS-like:	`\301\cfe\pp.c`	`..\include\defs.h`

(All three of these examples refer to the same two RISC OS files.) The linker offers more limited support – in essence, it recognises `thing.o` and `o.thing` as referring to the same RISC OS file (`o.thing`). In practice, object files almost always live locally (that's the only place the RISC OS and UNIX C compilers will put one) so this support is fairly complete.

AMU will even accept a mixture of naming styles, though good taste demands that this practice be deprecated.

Of course, the mapping between different naming styles cannot be complete (consider the UNIX analogue of `adfs::0.$.Library` or `net#1.251:src.amu`). However, it is usually sufficient to take much of the hard work out of moving reasonably portable makefiles.

VPATH

Usually, AMU looks for files relative to the current directory or in places implicit in the filename. The example given earlier contains the line:

```
amu:            amu.o $.301.clx.o.clxlib
```

which refers to `@.o.amu` (in `@.o`) and `$.clx.o.clxlib` (in `$.clx.o`).

Sometimes, particularly when dealing with multiple versions of large systems, it is convenient to have a complete set of object files locally, a few sources locally, but most sources in a central place shared between versions. For example, we can build different versions of the C compiler this way. If the macro VPATH is defined, then AMU will look in the list of places defined in it for any files it can't find in the places implicit in their names. For example, we might have compiler sources in `somewhere.arm`, `somewhere.mip`, `somewhere.cfe` and put the compiler makefile in `somewhere.ccriscos`. It might contain the following VPATH definition:

```
VPATH=^.arm ^.mip ^.cfe         # note that UNIX VPATHs
                                # separate path elements
                                # with colons, not spaces
```

and then dependency lines like:

```
o.pp:           c.pp            # ^.cfe.c.pp, via VPATH
o.cg:           c.cg            # ^.mip.c.cg, via VPATH
```

Rule patterns, **.SUFFIXES**, **$@**, **$***, **$<** and **$?**

All the examples given so far have been written out longhand, with explicit rules for making targets. In fact, AMU can make inferences if you supply the appropriate rule patterns. These are specified using special target names consisting of the concatenation of two suffixes from the pseudo-dependency `.SUFFIXES`. This sounds very complicated, but is actually quite simple. For example:

```
.SUFFIXES:      .o .c
amu:            o.amu ...
.c.o:;          $(CC) $(CFLAGS) -o $@ c.$*
```

(Note the order here: `.c.o` makes a `.o`-like thing from a `.c`-like thing).

The rule pattern `.c.o` describes how to make `.o`-like things from `.c`-like things. If, as in the above fragment, there is no explicit entry describing how to make a `.o`-like thing (`o.amu`, in the above example) AMU will apply the first rule it has for making .o-like things. Here, order is determined by order in the `.SUFFIXES` pseudo-dependency. For example, suppose `.SUFFIXES` were defined as `.o  .c  .f` and that there were two rules, `.c.o:...` and `.f.o:...` Then AMU would choose the `.c.o` rule because `.c` precedes `.f` in the `.SUFFIXES` dependency. In applying the `.c.o` rule, AMU infers a dependence on the corresponding `.c`-like thing – here `c.amu`. So, in effect, it infers:

```
o.amu:          c.amu
                $(CC) $(CFLAGS) -o o.amu c.amu
```

Note that, in the commands, `$@` is replaced by the name of the target and `$*` by the name of the target with the 'extension' deleted from it. In a similar fashion, `$<` refers to the list of inferred prerequisites. So the above example could be rewritten using the rule:

```
.c.o:;          $(CC) $(CFLAGS) -o $@ $<
```

However, if a `VPATH` were being used, this second form is obligatory. Consider, for example, the fragment:

```
VPATH=^.arm ^.mip ^.cfe
cc:             .... o.pp ....
.c.o:;          $(CC) $(CFLAGS) -o $@ $<
```

There is no explicit rule for making `o.pp`, so AMU will apply the rule pattern `.c.o:`.... This might expand to:

```
o.pp:           ^.cfe.c.pp
                $(CC) $(CFLAGS) -o o.pp ^.cfe.c.pp
```

which has a much more useful effect than:

```
                $(CC) $(CFLAGS) -o o.pp c.pp
```

Finally, `$?` can be used in any command to stand for the list of prerequisites with respect to which the target is out of date (which may be only some of the prerequisites).

Use of `::`

If you use `::` to separate targets from prerequisites, rather than `:`, the righthand sides of dependencies which refer to the same targets are not merged. Furthermore, each such dependency can have separate commands associated with it. Consider, for example:

```
o.t1::   c.t1 h.t1
         cc -g -c c.t1      # executed if o.t1 is out of
                            # date wrt c.t1 or h.t1
o.t1::   c.t1 h.t2
         cc -c c.t1         # executed if o.t1 is out of
                            # date wrt c.t1 or h.t2
```

Miscellaneous features

The special pseudo-target `.SILENT` tells AMU not to echo commands to be executed to your screen. Its effect is as if you used AMU `-s`.

The special pseudo-target `.IGNORE` tells AMU to ignore the return code from the commands it executes. Its effect is as if you used AMU `-i`.

A command line, the first non-white-space character of which is `@` is locally silent; just that command is not echoed. This is only rarely useful.

A command line, the first non-white-space character of which is – has its return code ignored when it is executed. This is extremely useful in makefiles which use commands such as Diff (from the Software Developer's Toolbox) which cannot set the return code conventionally.

The special macro MFLAGS is given the value of the command line arguments passed to AMU. This is most useful when a makefile itself contains AMU commands (for example, when a system consists of a collection of subsystems, each described by its own makefile). MFLAGS allows the same command line arguments to be passed to every invocation of AMU, even the recursive ones. For example, you might invoke AMU like this:

```
* AMU -k LIB=$.experiment.new.lib.grafix
```

and the makefile might contains entries like:

```
subsys_1:    $(COMMON) $(HDRS1) ...
             dir subsys1
             amu $(MFLAGS)
             back
```

Squeeze

The Squeeze utility is a program compactor. It takes an AIF file (such as the product of an execution of the Link program) and compresses it by a factor of about two. The compressed program can be executed directly; it 'expands' automatically when it is run. Squeezed programs can still be debugged using ASD. The advantages of using Squeezed programs is that they occupy less space on a floppy disc, and therefore take less time to load. This is also true of programs loaded from a hard disc as expanding happens at about 1Mb per second, faster than data can be loaded from a hard disc.

The exact saving in space depends on the contents of the image file. If it has many zeros (eg a large area of initialised static data in a C program), a factor of greater than two may be achieved. A hand-coded assembly language program, which contains a greater diversity of instructions than one produced by a compiler, would not achieve such a high compression ratio (3:2 being typical).

Relocatable modules should not be squeezed.

Syntax

The Squeeze command has the format:

```
Squeeze [-v] [-f] srce-file [dest-file]
```

If the -v flag is given, Squeeze will tell you a little about what is going on, including the size of the squeezed image and how long it took to squeeze it.

`-f` instructs Squeeze to go ahead and squeeze things it thinks are already squeezed. This is rarely useful.

The form with only one filename will reduce the given file in situ, overwriting the original with the new compacted form. If you give both filenames, the original is left intact, and the compressed version is stored in the second named file.

Examples

Below are two examples of the use of Squeeze.

```
*squeeze -v mint
-- squeezing 'MINT' to 'MINT'
-- encoding stats (0, 1, 2, 4) 9% 70% 19% 0%
-- compressed size 17519 is 57% of 30388
-- compression took 68csec, 44688 bytes/cpusec
```

```
*squeeze mint lib*.mint
```

Part 2 – Language issues

Implementation details

This chapter gives details of those aspects of the compiler which the draft ANSI standard identifies as implementation-defined, and some other points of interest to programmers. They are grouped here by subject; the final section – *Implementation limits* – lists the points required to be documented as set out in appendix A.6 of the draft standard.

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, the full ISO 8859-1 8-bit character set is recognised. 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 next chapter, *Standard Implementation Definition*). 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.

Data elements

The sizes of data elements are as follows:

Type	**Size in bits**
`char`	8
`short`	16
`int`	32

`long`	32	
`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 `chars` are signed by default). 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 headers, `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 draft standard for a full definition of their meanings.

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

```
CHAR_BIT 8
```

Maximum number of bytes in a multibyte character, for any supported locale:

```
MB_LEN_MAX 1
```

Numeric ranges of integer types: The column on the left gives the numerical values. The column on the right gives the bit patterns (in hexadecimal) that would be interpreted as these values 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 ANSI standard or Harbison and Steele for more details.

CHAR_MAX	255	0xff
CHAR_MIN	0	0x00
SCHAR_MAX	127	0x7f
SCHAR_MIN	-128	0x80
UCHAR_MAX	255	0xff
SHRT_MAX	32767	0x7fff
SHRT_MIN	-32768	0x8000
USHRT_MAX	65535	0xffff
INT_MAX	2147483647	0x7fffffff
INT_MIN	-2147483648	0x80000000
UINT_MAX	4294967295	0xffffffff
LONG_MAX	2147483647	0x7fffffff
LONG_MIN	-2147483648	0x80000000
ULONG_MAX	4294967295	0xffffffff

Characteristics of floating point:

FLT_RADIX	2
FLT_ROUNDS	1

Ranges of floating types:

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

Ranges of base two exponents:

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

Ranges of base ten exponents:

FLT_MAX_10_EXP	38
DBL_MAX_10_EXP	308
LDBL_MAX_10_EXP	308
FLT_MIN_10_EXP	(-37)
DBL_MIN_10_EX	(-307)
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 != 1.0):

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

Structured data types

The draft standard leaves details of the layout of the components of structured data types up 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.
- `char` components are placed in adjacent bytes.
- `short` components are aligned at even-addressed bytes.
- All other arithmetic type components are word-aligned, as are pointers and `int`s containing bitfields.
- The only valid type for bitfields is `int`, either signed or unsigned.

- A bitfield of type `int` is treated as unsigned by default (signed by default in `-pcc` mode).
- Bitfields must be contained within the 32 bits of an `int`.
- Bitfields are allocated within `int`s so that the first field specified occupies the least significant bits of the word.

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 faults 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 draft 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 quantity which specifies the amount of a shift is treated as an unsigned 8-bit value.
- Any value to be shifted is treated as a 32-bit value.
- Left shifts of more than 31 give a result of zero.
- Right shifts of more than 31 give a result of zero from an unsigned or positive signed value, −1 from a negative signed 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 cause exceptions 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:

- The ARM's floating point registers are wider than stored floating point numbers, so that some values may be computed to a slightly higher precision than the stated limits imply.
- 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 draft standard before evaluating any expression.

- The compiler may re-order expressions involving only associative and commutative operators, even in the presence of parentheses.
- 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; moreover, this order may change from release to release.

Implementation limits

The draft 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 compound statements and iteration/selection control structures	15	mem
Nesting levels of conditional compilation	6	mem
Declarators modifying a basic type	12	mem
Expressions nested by parentheses	127	mem
Significant characters		
in internal identifiers and macro names	31	256
in external identifiers	6	256
External identifiers in one source file	511	mem
Identifiers with block scope in one block	127	mem
Macro identifiers in one source file	1024	mem
Parameters in one function definition/call	31	mem
Parameters in one macro definition/invocation	31	mem
Characters in one logical source line	509	no limit
Characters in a string literal	509	mem
Bytes in a single object	32767	mem
Nesting levels for #included files	8	mem
Case labels in a `switch` statement	255	mem
`atexit`-registered functions	32	33

Standard implementation definition

This chapter discusses aspects of the compiler which are not defined by the ANSI draft standard, but are implementation-defined and must be documented.

Appendix A.6 of the December 1988 draft standard collects together information about portability issues; section A.6.3 lists those points which are implementation defined, and directs that each implementation shall document its behaviour in each of the areas listed. This chapter corresponds to appendix A.6.3, answering the points listed in the appendix, under the same headings and in the same order.

Translation (A.6.3.1)

- 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 ANSI sense, but an attempt by the compiler to be helpful to you.
  - error: a violation of the ANSI specification from which the compiler was able to recover by guessing your intentions.
  - serious error: a violation of the ANSI specification from which no recovery was possible because the compiler could not reliably guess what you intended.
  - too many errors/fatal error: (for example, 'not enough memory') these are not really diagnostics but indicates that the compiler limits have been exceeded.

Environment (A.6.3.2)

- The arguments given to `main()` are the words of the Command Line (not including I/O redirections, covered in the next point), delimited by white spaces, except where the white space characters are contained in double quotes. A white space character is any one of: space, form-feed, newline, carriage return, tab or vertical tab (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 term 'interactive device' denotes either the keyboard or the screen (`:tt`). 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 buffering is used except where both `stdout` and `stderr` have been redirected to the same file, in which case line buffering is used.
- The standard input, output and error streams, `stdin`, `stdout`, and `stderr` can be redirected at runtime in the following way. For example, if `copy` is a compiled and linked program which simply copies the standard input to the standard output, the following line:

  ```
  *copy <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 table shows all allowed redirections:

`0<`*`filename`*	read `stdin` from *`filename`*
`<`*`filename`*	read `stdin` from *`filename`*
`1>`*`filename`*	write `stdout` to *`filename`*
`>`*`filename`*	write `stdout` to *`filename`*
`2>`*`filename`*	write `stderr` to *`filename`*
`>&`*`filename`*	write both `stdout` and `stderr` to *`filename`*
`1>&2`	write `stdout` to wherever `stderr` is currently going
`2>&1`	write `stderr` to wherever `stdout` is currently going

Identifiers (A.6.3.3)

- 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.

Characters (A.6.3.4)

- The characters in the source character set are ISO 8859-1 (Latin Alphabet), a superset of the ASCII character set. The printable characters are those in the range 32 to 126 and 160 to 255. All printable characters may appear in string or character constants, and in comments.
- There are no locales implemented for which a multibyte character shift state exists.
- The execution character set is identical to the source character set.
- There are four `chars` in an `int`. The bytes are ordered from least significant at the lowest address to most significant at the highest address.
- There are eight bits in a character in the execution character set.
- All integer character constants that contain a character or 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 characters in the execution character set.
- No locale is used to convert multibyte characters into the corresponding wide characters (codes) for a wide character constant.
- 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. This is not portable.
- A 'plain' `char` is treated as unsigned (signed in `-pcc` mode).
- Escape codes are:

Escape sequence	Char value	Description
`\a`	7	Attention (bell)
`\b`	8	Backspace
`\f`	12	Form feed
`\n`	10	Newline
`\r`	13	Carriage return
`\t`	9	Tab
`\v`	11	Vertical tab
`\xnn`	nn	ASCII code in hexadecimal
`\nnn`	nnn	ASCII code in octal

Integers (A.6.3.5)

The representations and sets of values of the integral types are set out in the previous chapter, in the section *Data elements*. 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 were 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 (A.6.3.6)

The representations and ranges of values of the floating point types have been given above in *Implementation details*, *Data elements*. 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 accord with IEEE 754.

Arrays and pointers (A.6.3.7)

The ANSI draft 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`.

Registers (A.6.3.8)

In the Acorn C compiler, you can declare up to six objects as having the storage class `register`. There are six available registers, so declaring more than six objects with register storage class will 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 structure type which contains only bitfields and which is no more than one word long.

Note that other variables, not declared as `register`, may be held in registers for extended periods, and that `register` variables may be held in memory for some periods.

Structures, unions, enumerations and bitfields (A.6.3.9)

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 parts of `int`s, 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 an `int` is placed in the next `int`.
- The order of allocation of bitfields within `int`s is such that the first field specified occupies the least significant 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 (A.6.3.10)

A read or write constitutes an access to an object that has volatile-qualified type.

Declarators (A.6.3.11)

The number of declarators that may modify an arithmetic, structure or union type is limited only by available memory.

Statements (A.6.3.12)

The number of `case` values in a `switch` statement is limited only by memory.

Preprocessing directives (A.6.3.13)

- 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 an Acorn filename is described in the chapter *How to install and run the compiler*.
- Quoted names for includable source files are supported. The rules for directory searching are given in *How to install and run the compiler*.
- The recognized `#pragma` directives and their meaning are described in the section *`#pragma` directives*, in the chapter entitled *Machine-specific features*.
- The date and time of translation are always available, so __DATE__ and __TIME__ always give respectively the date and time.

Library functions (A.6.3.14)

When using library functions in the Acorn C compiler, note the following points:

- 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, no error will be detected.
- The `assert()` function prints the following message:

  `*** assertion failed:` *`expression`*`, file` *`filename`*`, line,` *`line-number`*

  and then calls the function `abort()`.

- The functions:

  ```
  isalnum()
  isalpha()
  iscntrl()
  islower()
  isprint()
  isupper()
  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 `iscntrl()` 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 for characters:

  0 to 31 are control characters
  128 to 159 are control characters
  192 to 223 except 215 are upper case
  224 to 255 except 247 are lower case
  160 to 191, and 215 and 247 are punctuation

  The results returned by the functions reflect this.

- The mathematical functions return the following values on domain errors:

Function	Condition	Returned value
`log(x)`	`x <= 0`	`-HUGE_VAL`
`log10(x)`	`x <= 0`	`-HUGE_VAL`
`sqrt(x)`	`x < 0`	`-HUGE_VAL`
`atan2(x,y)`	`x = y = 0`	`-HUGE_VAL`
`asin(x)`	`abs(x) > 1`	`-HUGE_VAL`
`acos(x)`	`abs(x) > 1`	`-HUGE_VAL`

  Where `-HUGE_VAL` is written above, a number is returned which is defined in the header `h.math`. 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 `signal()` function is as follows:

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

- The default handling of all the signals recognised is the printing of a suitable message followed by a stack backtrace. 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 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 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 in the draft standard (section 4.9.3).
- 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, and once for writing or updating. A file cannot however be open for reading on one stream and for writing or updating on another.

- Local time zones and Daylight Saving Time are not implemented. The values returned will always indicate that the information is not available.
- Note also the following points about library functions:

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

Error:	**Message:**
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()
others	Error code *number* has no associated message

`calloc()`, `malloc()` and `realloc()`	If size of area requested is zero, `NULL` is returned.
`abort()`	Closes all open files, and deletes all temporary files.
`exit()`	The status returned by `exit` is the same value that was passed to it. For a definition of `EXIT_SUCCESS` and `EXIT_FAILURE` refer to the header file `stdlib.h`.
`getenv()`	Returns the value of the named RISC OS Environmental variable, or `NULL` if the variable had no value.

```
eg root = getenv ("C$libroot");
if (root == NULL) root = "$.arm.clib";
```

`system()` 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 `system()` call is `-2` (indicating a failure to invoke the program) or the value of `Sys$ReturnCode` set by the called program (0 indicates success).

`strerror()` The error messages given by this function are identical to those given by the `perror()` function.

`clock()` Returns the time taken by the program since its invocation, as indicated by the host's operating system.

Portability

Introduction

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 ANSI C and the well-known 'K&R' C as defined in the book *The C Programming Language*, (first edition) by Kernighan and Ritchie
- the `toansi` and `topcc` tools
- 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 targetted. 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 ANSI 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 & 0xfffffff8; /* 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, then 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 ANSI 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 ARRSIZE 10000
typedef struct
        {
          int i;
          short s;
         } ELEM;
ELEM arr[ARRSIZE];
```

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, ANSI 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.

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 fuction 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 targetted 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 ANSI 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.

ANSI C vs K&R C

The ANSI 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 ANSI C environment. In the following sections, we present a list of what we consider to be the major differences between ANSI 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 ANSI 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 ANSI 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

ANSI 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 ANSI 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

- ANSI 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 'stringise' 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 ANSI-conforming compilers.

The `topcc` and `toansi` tools

The programs `topcc` and `toansi` help you to translate C programs and headers between the ANSI and PCC dialects of C. Only limited syntactic translation is performed as described below; other differences must be addressed in the source before or after translation. These programs enable you to write (with care) programs which can be translated directly between the PCC and ANSI dialects.

The command format is:

```
toansi [infile [outfile]]
```

```
topcc [infile [outfile]]
```

`infile` and `outfile` default to `stdin` and `stdout` respectively.

`topcc`

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, `300ul` becomes `(unsigned long)300L`).

`toansi`

Function declarations with embedded comments are rewritten without the comment tokens. This reverses the action of `topcc` with regard to function declarations (see above).

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)`.

pcc compatibility mode

This section discusses the differences apparent when the compiler is used in 'PCC' mode. When given the `-pcc` command line flag, 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 `-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 ANSI somewhat differently from the usual PCC implementation.
- chars are signed by default in `-pcc` mode.
- In `-pcc` mode, the compiler permits the use of the ANSI `'...'` 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 (eg little-endian ARM, VAX, Intel versus big-endian Motorola, IBM370)
  - 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 -pcc` is found undesirable, all warnings can be turned off by using the `-w` command line flag.
- 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 `-E` option is used with the `cc` command.

- There is a different treatment of whitespace sequences (benign).
- <nl> is processed by `cc -E`, but passed by cpp (making lines longer than expected; `cc -E` only).
- Cpp breaks long lines at a token boundary; `cc -E` doesn't (this may break line-size constraints when the source is later consumed by another program `cc -E` only).
- The handling of unrecognised # directives is different (this is mostly benign).

Standard headers and libraries

Use of the compiler in `-pcc` mode precludes neither the use of the standard ANSI headers built in to the compiler nor the use of the run-time library supplied with the C compiler. Of course, the ANSI 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 `-j`, 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 ANSI C Library, and the BSD UNIX library. They are placed under headings corresponding to the ANSI header files:

ctype.h

There are no `isascii()` and `toascii()` functions, since ANSI 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. ANSI 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 ANSI C. ANSI C does not have `asinh()`, `acosh()`, `atanh()`.

`signal.h`

In ANSI 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 ANSI C, since their functionality is provided by `sprintf()`.

`string.h`

On BSD systems, string manipulation functions are found in `strings.h`, whereas ANSI 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 ANSI functions `strchr()` and `strrchr()` respectively.

Functions which refer to string lengths (and other sizes) now use the ANSI 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 ANSI giving details of floating point precision etc.

`limits.h`

A new header added by ANSI to give maximum and minimum limit values for data types.

`locale.h`

A new header added by ANSI 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 and MS-DOS.

The most apparent interface between a C program and its environment is via the arguments to `main()`. The ANSI 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 *Environment* section of the *Standard Implementation Definition* chapter, 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 and MS-DOS, it is common to use a third parameter, normally called `char *environ[]` under UNIX and `char *envp[]` under Microsoft C for MS-DOS, 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 and MS-DOS, 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 '`.`' 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). RISC OS filename components are limited to 10 characters.

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 or MS-DOS. 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`. 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/MS-DOS low-level I/O can be simulated in this way, but the ANSI C run-time library provides sufficient support for most applications to be written in a portable style. If the application is running under the desktop, then limited piping facilities can be achieved by using the calls `wimp_transferblock` and `wimp_sendmessage` to synchronise the data transfer.

RISC OS does not support different memory models as in MS-DOS, so programs which have been written to exploit this will need modification; this should only require the removal of Microsoft C keywords such as `near`, `far` and `huge`, if the program has otherwise been written with portability in mind.

ANSI library reference section

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 source file, and the source line number, the last two of these being, respectively, the values of 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.

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`.

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.

`int isalnum(int c)`	Returns true if `c` is alphabetic or numeric
`int isalph(int c)`	Returns true if `c` is alphabetic
`int iscntrl(int c)`	Returns true if `c` is a control character (in the ASCII locale)
`int isdigit(int c)`	Returns true if `c` is a decimal digit

`int isgraph(int c)`	Returns true if `c` is any printable character other than space
`int islower(int c)`	Returns true if `c` is a lower-case letter
`int isprint(int c)`	Returns true if `c` is a printable character (in the ASCII locale this means 0x20 (space) → 0x7E (tilde) inclusive).
`int ispunct(int c)`	Returns true if `c` is a printable character other than a space or alphanumeric character
`int isspace(int c)`	Returns true if `c` is a white space character viz: space, newline, return, linefeed, tab or vertical tab
`int isupper(int c)`	Returns true if `c` is an upper-case letter
`int isxdigit(int c)`	Returns true if `c` is a hexadecimal digit, ie in 0…9, a…f, or A…F
`int tolower(int c)`	Forces `c` to lower case if it is an upper-case letter, otherwise returns the original value
`int toupper(int c)`	Forces `c` 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 three 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`.

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 entitled *Implementation details*.

limits.h

This set of macro constants determines the upper and lower value limits for integral objects of various types, as follows:

Object type	Minimum value	Maximum value
Byte (number of bits)	`0`	`8`
Signed char	`-128`	`127`
Unsigned char	`0`	`255`
Char	`0`	`255`
Multibyte character (number of bytes)	0	`1`
Short int	`-0x8000`	`0x7fff`
Unsigned short int	`0`	`65535`
Int	`(~0x7fffffff)`	`0x7fffffff`
Unsigned int	`0`	`0xffffffff`
Long int	`(~0x7fffffff)`	`0x7fffffff`
Unsigned long int	`0`	`0xffffffff`

See also the chapter entitled *Implementation details*.

locale.h

This file handles national characteristics such as the different orderings 'month-day-year' (USA) vs '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:

`LC_COLLATE`	string collation
`LC_CTYPE`	character type
`LC-MONETARY`	monetary formatting
`LC_NUMERIC`	numeric string formatting
`LC_TIME`	time formatting
`LC_ALL`	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 nonnegative 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.

math.h

This file contains the prototypes for 22 mathematical functions. All return the type `double`.

Function	Returns
`double acos(double x)`	arc cosine of *x*. A domain error occurs for arguments not in the range –1 to 1
`double asin(double x)`	arc sine of *x*. A domain error occurs for arguments not in the range –1 to 1
`double atan(double x)`	arc tangent of *x*
`double atan2(double x, double y)`	arc tangent of *y/x*
`double cos(double x)`	cosine of *x* (measured in radians)
`double sin(double x)`	sine of *x* (measured in radians)
`double tan(double x)`	tangent of *x* (measured in radians)
`double cosh(double x)`	hyperbolic cosine of *x*
`double sinh(double x)`	hyperbolic sine of *x*
`double tanh(double x)`	hyperbolic tangent of *x*
`double exp(double x)`	exponential function of *x*
`double frexp(double x, int *exp)`	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`
`double ldexp(double x, int exp)`	*x* times 2 raised to the power of `exp`
`double log(double x)`	natural logarithm of *x*
`double log10(double x)`	log to the base 10 of *x*

`double modf(double x, double *iptr)`	signed fractional part of *x*. Stores integer part of *x* in object pointed to by `iptr`.
`double pow(double x, double y)`	*x* raised to the power of *y*
`double sqrt(double x)`	positive square root of *x*
`double ceil(double x)`	smallest integer not less than *x* (ie rounding up)
`double fabs(double x)`	absolute value of *x*
`double floor(double x)`	largest integer not greater than *x* (ie rounding down)
`double fmod(double x, double y)`	floating-point remainder of *x/y*

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 and Arthur 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 startup, the equivalent of `signal(sig, SIG_IGN)` may be executed for some signals selected in an implementation-defined manner (under RISC OS and Arthur 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.

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.

`stdio.h` is required to declare `vfprintf()` without defining `va_list`. Clearly the type `__va_list` there must keep in step.

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 draft ANSI specification and cannot be relied on.

va_start

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

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 `parmN`. 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_end

```
#define va_end(ap) ((void)(*(ap) = (char *)-256))
```

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.

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.

`ptrdiff_t(here int)` the signed integral type of the result of subtracting two pointers

`size_t(here unsigned int)` the unsigned integral type of the result of the `sizeof` operator

`wchar_t(here int)` also in `stdlib.h`. 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.

`size_t offsetof(type, member)` Expands to an integral constant expression that has type `size_t`, the value of which is the offset in bytes from the beginning of a structure designated by `type`, of the member designated by `member` (if the specified member is a bit-field, the behaviour is undefined).

stdio.h

`stdio` declares two types, several macros, and many functions for performing input and output. For a discussion on Streams and Files refer to sections 4.9.2 and 4.9.3 in the ANSI draft, or to one of the other references given in the *Introduction* to this Guide.

`fpos_t` `fpos_t` is an object capable of recording all information needed to specify uniquely every position within a file.

`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 `#ifdef __system_io` clause are for system use only, and cannot be relied on between releases of C.

stdio functions

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 and Arthur the operation fails).

Returns: zero if the operation succeeds, nonzero 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 and Arthur, the operation fails).

Returns: zero if the operation succeeds, nonzero 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 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.

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 and Arthur 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 succesfully 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:

`r`	open text file for reading
`w`	create text file for writing, or truncate to zero length
`a`	append; open text file or create for writing at eof

`rb`	open binary file for reading
`wb`	create binary file for writing, or truncate to zero length
`ab`	append; open binary file or create for writing at eof
`r+`	open text file for update (reading and writing)
`w+`	create text file for update, or truncate to zero length
`a+`	append; open text file or create for update, writing at eof
`r+b` or `rb+`	open binary file for update (reading and writing)
`w+b` or `wb+`	create binary file for update, or truncate to zero length
`a+b` or `ab+`	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.
- 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 or Arthur).
- 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` fuction 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` fuction 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 or Arthur). 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.

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.

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.

Returns: zero on success, or nonzero 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 4.9.6.1 in the ANSI draft 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 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 of 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`
`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.

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 4.9.6.2 in the ANSI draft, or to any of the references listed in the *Introduction* to this Guide. 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 strem. 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.

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 asociated 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 nonnegative value.

getc

`int getc(FILE * stream)`

Equivalent to `fgetc` except that it may be (and is under RISC OS and Arthur) 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.

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 and Arthur) 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 nonnegative 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 `unget` 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 nonzero is returned and the integer expression `errno` is set to an implementation-defined nonzero value (under RISC OS or Arthur `fgetpos` cannot fail).

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.

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 nonzero is returned and the integer expression errno is set to an implementation-defined nonzero value (under RISC OS and Arthur the value is that of EDOM in math.h).

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 nonzero value (under RISC OS or Arthur `ftell` cannot fail).

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: nonzero iff 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: nonzero iff 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 ANSI Draft, section 3.1.3.2), optionally preceeded 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 preceeded 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`.

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 ANSI Draft, section 3.1.3.2), optionally preceeded 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 preceeded 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`.

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 `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.

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, nonzero 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).

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 and Arthur, 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.

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, `quot * denom + 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.

labs

`long int labs(long int j)`

Computes the absolute value of an long integer *j*. If the result cannot be represented, the behaviour is undefined.

Returns: the absolute value.

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, `quot * denom + 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.

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 nonzero 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 nonzero 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 nonzero 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 nonzero 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 caharacter 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.

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 nonzero 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 form *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 and Arthur the strings for the given `errnum`s 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()`
- 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.

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 (as is the case under RISC OS).

```
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\0`.

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 (it is not available under RISC OS).

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
`%d`	the day of the month as a decimal number (01–31)
`%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
`%S`	the second as a decimal number (00–61)
`%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 time zone name or abbreviation, or by no character if no time zone is determinable
`%%`	`%`

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.

Part 3: Developing software for RISC OS

How to write desktop applications in C

In this chapter, you will learn how to construct desktop applications in C, using the facilities provided by the RISC OS library (RISC_OSlib). You will probably find it useful to scan through the contents of the library before reading the chapter. Some familiarity with the *Window Manager* part of the *RISC OS Programmer's Reference Manual* is also assumed. The description of RISC_OSlib here is not exhaustive: it is intended to introduce some common programming techniques used in desktop applications.

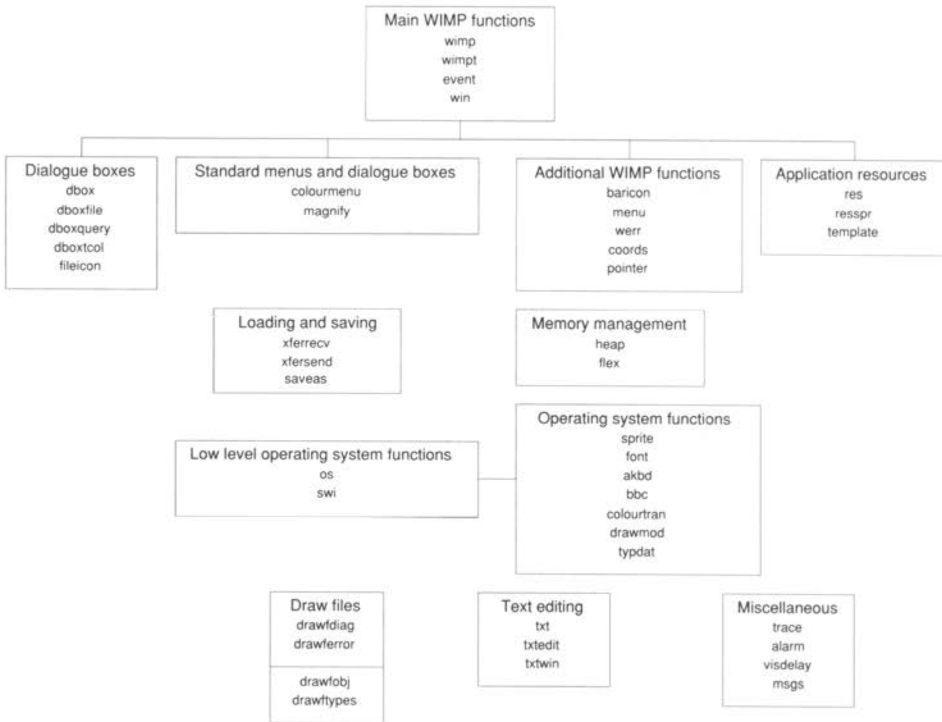
You are also advised to read the *Application notes* section of the *Window Manager* chapter. This describes certain standards to which all desktop applications must conform in order to have an appearance which is consistent with the applications supplied by Acorn. Following these guidelines will make your own applications look and feel like part of the same environment, which makes them easier to learn and use.

The diagram over the page shows approximately how the various parts of the RISC OS library fit together. The diagram is reproduced on one of the reference cards; you may find it useful to refer to it as you work through this chapter.

Some general principles

Event handling

If you have read the *Window Manager* chapter in the *RISC OS Programmer's Reference Manual*, you may be familiar with the idea of Wimp polling, as the means whereby an application finds out what the window manager requires it to do. In this method, an application uses a single polling loop, which must work out which of its windows each request from the Wimp is associated with, and take the appropriate action. RISC_OSlib makes available an alternative means of communicating with the Wimp, using functions called *event handlers*. An application may register event handlers (in the form of C functions) for windows, menus, icon bar icons, etc. It then calls a function in RISC_OSlib which processes events (ie polls the Wimp), and directs each event in turn to the relevant event handler. Event handlers may be added and removed whilst the application is running. This approach makes it easier to keep track of which window, menu, or whatever, is associated with a window manager event.



When a call to register an event handler is made, an extra piece of data may be registered with it. This value (or handle) is then passed as an argument to the event handler when it is called by RISC_OSlib. It is sometimes convenient to use this as a way of allowing an event handler to retain private data. For example, you could use the same event handler for several windows, the handle being a pointer to the data structure associated with the window.

The event handler would then be able to locate the data structure for the window immediately, rather than having to work it out from the window handle passed into the event handler.

Windows and templates

In order to define the windows and dialogue boxes used by your program, you can either set up data structures which correspond to those used by the window manager SWIs, or you can use templates created by the template editor, FormEd. The template editor is an application which allows you to define windows on the screen, and save the definitions in a file ready for loading by your application. This is the approach used in Acorn's own applications, and you will find it makes the process of creating windows for your applications much easier. FormEd is described in the next chapter.

Application resources

Most desktop applications make use of a number of *resource files*. These should be considered as an integral part of your application. You can find a full list of the resource files typically used by an application in the Application Notes section of the *RISC OS Programmer's Reference Manual*; the following are usually present:

- `!Boot` — *Run when the application directory is first displayed
- `!Run` — *Run to start the application
- `!RunImage` — the executable code for the application
- `!Sprites` — used for application and file icon sprites
- `Sprites` — containing other sprites used by the application
- `Templates` — containing window and dialogue box templates.

Developing an application from scratch

This section contains an example of how to develop an application from scratch. You can use the description and the code given as a starting point for writing your own applications. The example application is called `!WExample`. It can be found on Disc 1 as `$.DeskEgs.!WExample` (the source code is there too).

The application is very simple. When started, it places an icon on the icon bar. The icon has a menu consisting of **Info**, which leads to a dialogue box containing information about the program, and **Quit** which closes the program down. Clicking Select on the icon opens a window, which may be resized, moved, closed, and so on, in the normal way. If the window is already open, an error is reported. In itself, this is not a very useful program, but it illustrates the basic principles of writing a program which uses the RISC OS library.

The source code of the program is to be found on Disc 1 as `$.DeskEgs.!WExample.c.WExample`. Fragments of the code are given in this chapter to illustrate the points being described. You may also find it useful to have a listing of the whole program available to see how it all fits together.

The program illustrates the following:

- the general form of a desktop application
- how to initialise a desktop application
- how to create windows, icons and menus
- how to open a window, and respond to Wimp requests for it
- how to respond to user choices from a menu
- how to report errors to the user.

General form of a desktop application

A Wimp application normally consists of initialisation of both the RISC OS library and the application itself, followed by an event processing loop. `main()` in `c.WExample` is an example. You will see from this that the final step of the `main` function is to enter an infinite loop of calls to `event_process()`. The application will be closed down as a result of a call to the ANSI library function `exit()` elsewhere in the program. If you don't like this approach, an alternative is to define a global flag and test it in the event processing loop, for example:

```
/* --- global closedown flag --- */
BOOL all_done = FALSE;
...
/* --- the main event loop --- */
while ( !all_done ) event_process();
```

Note also that `event_process()` can automatically close down the application. To do this, it keeps an *active count*. The active count is initally zero; if it is zero again when `event_process()` is called, the application is closed down. You can change the active count by calling the functions win_activeinc() to increment it and win_activedec() to decrement it. Typically, you would call the first of these on opening a window, and the second on closing it. When you call baricon() to place an icon on the icon bar, win_activeinc() is called for you. If your application does not place an icon on the icon bar, you must make sure you call win_activeinc() yourself before entering the event processing loop. See the description of the win functions in the later chapter, *RISC OS library reference section* for more details.

Initialising a desktop application

The initialisation of WExample occurs in example_initialise(). The first few lines initialise various parts of the RISC OS library:

```
/* RISC_OSlib initialisation */
wimpt_init("RISC_OSlib example");  /* Main Wimp initialisation */
res_init("WExample");             /* Resources */
resspr_init();                    /* Application sprites */
template_init();                  /* Templates */
dbox_init();                      /* Dialogue boxes */
```

Most applications will start with something similar, although there may be more or fewer parts of the library which need initialising. One point to note is the use of the arguments to wimpt_init() and res_init(). wimpt_init() uses its argument in any circumstances where the application is to be referred to by name, for example in the task display, or in error boxes. res_init() uses its argument to locate the application resources; in this case they will be expected to be in a directory whose name can be found from the system variable WExample$Dir. This variable must therefore be set up in the !Run file for the application, for example by:

```
*set WExample$Dir <Obey$Dir>
```

Creating windows, icons and menus

The remainder of example_initialise() creates the window which will be used in the program, sets up a menu to go with the icon, and places the icon on the icon bar. We will consider these one by one.

Creating the window is very straightforward. A pointer to a window definition read from the templates file is passed to wimp_create_wind(). An event handler is then registered for the window. Here is the code to do it, from example_initialise():

```
/* Create the main window, and declare its event handler */
if (!example_create_window("MainWindow", &example_win_handle))
  return FALSE; /* Window creation failed */
win_register_event_handler(example_win_handle, example_event_handler, 0);
```

The code for creating the window is in a separate function, so that we could use it for creating other windows in a more complex program. It is as follows:

```
static BOOL example_create_window(char *name, wimp_w *handle)
{
  wimp_wind *window;     /* Pointer to window definition */

  /* Find template for the window */
  window = template_syshandle(name);
  if (window == 0) return FALSE;

  /* Create the window, dealing with errors */
  return (wimpt_complain(wimp_create_wind(window, handle)) == 0);
}
```

`example_create_window()` illustrates the value of using templates: all the work needed to set up the window definition in the program is avoided. The event handler will not be called unless the window is open; we will come back to this later.

The next step is to create the menu. If possible, you should create all menus during the initialisation. That way, when the user activates a menu, you can be sure that it exists and can be displayed. This may not be possible in some applications, because the menus have to change with circumstances, but you will nearly always be able to create at least part of the menu tree. In addition, clicking with Adjust when menus are created dynamically may fail, for subtle reasons connected with when the window manager calls the menu maker code.

The code to create the menu in the example is:

```
/* Create the menu tree */
if (example_menu = menu_new("Example", ">Info,Quit"), example_menu) == NULL)
  return FALSE; /* Menu create failed */
```

`Example` is the name which appears in the title bar of the menu. `menu.h` explains the syntax of the second argument to `menu_new()` in detail. In this case, `>Info` means that the menu entry for **Info** is to be marked as leading to a submenu consisting of a dialogue box.

If you want to check the menu before it is displayed, perhaps because you want to tick or shade items in it, then instead of calling `event_attachmenu()`, you can call `event_attachmenumaker()` with the name of a function to be called just before the menu is displayed. See the description of the file `event.h` in the *RISC OS library reference section* chapter for details.

Finally, the icon is placed on the icon bar, and event handlers registered for it. There are two event handlers here: `example_iconclick()`, which is called when Select is clicked on the icon, and `example_menuproc()` which is called when a choice is made from the menu. The work of displaying the menu is handled by RISC_OSlib. Here is the code:

```
/* Set up the icon on the icon bar, and register its event handlers */
baricon("!WExample", (int)resspr_area(), example_iconclick);
if (!event_attachmenu(win_ICONBAR, example_menu, example_menuproc, 0))
  return FALSE; /* Unable to attach menu */
```

There are two points to note here. First, the sprite used for the icon will be loaded from the 'Sprites' file in the application directory. The function `resspr_area()` returns a pointer to the sprite area into which this file is loaded. Second, `event_attachmenu()` is used to associate a menu with a window by specifying the window handle. The value `win_ICONBAR` is a special window handle which is used to represent the icon bar.

Opening and maintaining a window

The window is to be opened when the user clicks Select on the icon. As we saw above, clicking Select calls the function `example_iconclick()`, which is as follows:

```
static void example_iconclick(wimp_i icon)
{
  icon = icon; /* We don't need the handle: this stops compiler warning */

  /* Open the window - only one allowed */
  if (example_window_open)
    werr(FALSE, "Only one window may be opened");
  else
  {
    wimp_wstate  state;

    /* Get the state of the window */
    if (wimpt_complain(wimp_get_wind_state(example_win_handle, &state)) == 0)
    {
      state.o.behind = -1;          /* Make sure window is opened in front */
      wimpt_noerr(wimp_open_window(&state.o));
      example_window_open = TRUE;
```

```
    }
  }
}
```

You can ignore the lines of this up to the 'else' part for now: they just report an error if the window is already open. When we open the window, we want to make sure it is in front of any others on the screen. To do this, we read the current state of the window with `wimp_get_wind_state()`, and then ensure that our window is behind the window with `handle -1`, ie in front of all others. The window is actually opened with `wimp_open_wind()`.

Once the window is open, the event handler which we registered earlier will be called by `event_process()` when the window manager generates events for the window. The code for the event handler is:

```
static void example_event_handler(wimp_eventstr *e, void *handle)
{
  handle = handle; /* We don't need the handle: this stops compiler warning */

  /* Deal with event */
  switch (e->e)
  {
    case wimp_EREDRAW:
      example_redraw_window(e->data.o.w);
      break;

    case wimp_EOPEN:
      example_open_window(&e->data.o);
      break;

    case wimp_ECLOSE:  /* Pass on close request */
      wimpt_noerr(wimp_close_wind(e->data.o.w));
      example_window_open = FALSE;
      break;

    default:   /* Ignore any other event */
      break;
  }
}
```

In this case, the event handler is very simple. Redraw and open requests are handled as described in the *Window Manager* chapter of the *RISC OS Programmer's Reference Manual*: see `c.wexample` for the full details of the functions. On a close request (generated by the user clicking the close icon of the window), we simply call `wimp_close_wind()`. After this, the event handler will not be called again, unless the window is re-opened. In an editor, some checks would normally be made before passing on the close request to the window manager: for example, ensuring that the contents of the window had been saved, and either warning the user or rejecting the close window request if they had not.

All events other than redraw, open and close requests are simply ignored. A way of improving the efficiency of the program would be to call `event_setmask()`. This indicates to the window manager that some events are never to be returned to the program. It must be used with care, since it masks the events to all windows. Thus, although the main window of the program has no menu, we could not mask out menu events, since they are used by the icon bar event handler. However, we could safely mask out 'pointer entering window' and 'pointer leaving window' events. Some suitable code for doing this would be:

```
event_setmask(wimp_EPTRENTER | wimp_EPTRLEAVE);
```

Responding to user choices from a menu

The menu is displayed when Menu is pressed over the icon: no special action is needed by the application for this. When the user makes a choice from the menu, the menu event handler we registered earlier is called:

```
/* Menu items */
#define Example_menu_info      1
#define Example_menu_quit      2

...

static void example_menuproc(void *handle, char *hit)
{
  handle = handle; /* We don't need the handle: this stops compiler warning */

  /* Find which menu item was hit and take action as appropriate */
  switch (hit[0])
  {
    case Example_menu_info:
      example_info_about_program();
      break;

    case Example_menu_quit:
      /* Exit from the program. The Wimp gets rid of the window and icon */
      exit(0);
  }
}
```

`handle` is the fourth argument which was given to `event_attachmenu()`: we make no use of it in this example. `hit` is a string in which each entry corresponds to a selection from the menu tree: the first character is the number of the selection from the top level menu, the second from the first submenu chosen, and so on. In this example, only a single, top-level menu was set up, so we are only interested in `hit[0]`.

Handling **Quit** is easy: the program just exits. A hit on **Info** occurs when either the user chooses it by clicking, or when he follows the submenu arrow leading from it. In this case, we call the following function to display a dialogue box containing information about the application:

```
static void example_info_about_program(void)
{
  dbox  d;  /* Dialogue box handle */

  /* Create the dialogue box */
  if (d = dbox_new("ProgInfo"), d != NULL)
  {
    /* Fill in the version number */
    dbox_setfield(d, Example_info_field, example_Version_String);

    /* Show the dialogue box */
    dbox_show(d);

    /* Keep it on the screen as long as needed */
    dbox_fillin(d);

    /* Dispose of the dialogue box */
    dbox_dispose(&d);
  }
}
```

First, the dialogue box is created from the template named `ProgInfo`; (this name is case-sensitive). Most of the fields are also taken from the template, but we want to fill in one field with the current version of the program, from the string `example_Version_String`. When this has been done, using `dbox_setfield()`, the dialogue box is displayed with `dbox_show()`.

The call to `dbox_fillin()` needs some explanation. It will not return until the window manager detects that the dialogue box has been finished with. For example, in this case a click elsewhere on the screen (and which therefore removes the menu tree) would cause `dbox_fillin()` to return. However, in the intervening time, other event handlers for the program may still be called. When the dialogue box is finished with, `dbox_fillin()` removes it from the screen and returns. The event handler then calls `dbox_dispose()` which deletes any internal data that was set up by the call to `dbox_new()`.

Reporting errors

The example application shows three different ways of dealing with errors. In each case, the error is reported in a standard error box, and the application waits until **OK** has been clicked. You will probably have seen this format from the desktop and the applications suite.

First, there are errors generated by the application itself. These are reported with, for example:

```
werr(FALSE, "Only one window may be opened");
```

(in the function `example_iconclick()`). The first parameter indicates whether this error is fatal: in this case it is not. A fatal error causes the application to exit. You can specify a number of parameters to `werr`, using the second one as a format string in the same way as for the ANSI library function `printf`.

When an error is returned by a RISC_OSlib function, we can report it in one of two further ways. The first is illustrated by the following line from `example_redraw_window()`:

```
wimpt_noerr(wimp_redraw_wind(&r, &more));
```

This reports the error in a dialogue box and halts the application.

An alternative is `wimpt_complain()`. This is similar to `wimpt_noerr()`, except that it also returns a pointer to the error, allowing the application to detect the error and take further action, as well as reporting it. A returned value of 0 indicates no error. For example (this is from `example_iconclick()`):

```
if (wimpt_complain(wimp_get_wind_state(example_win_handle, &state)) == 0)
{
... actions if there is no error ...
}
```

With one exception, it is strongly recommended that you report errors using `wimpt_noerr()` or `wimpt_complain()` on *all* calls to RISC_OSlib functions that return an error. This will help you find errors as soon as they occur. If an error does occur and you discard it, the effects of the errors may cause confusion at later stages in the program.

The one exception is reporting errors during redraw, using `wimpt_complain()`. Here you must take some care. If the error box lies over your window, when it is removed a new redraw will be issued, which can lead to the same error again. A possible solution is to keep a flag to avoid this happening, resetting the flag when the contents of the window have been mended.

More RISC_OSlib facilities

In this section, we will examine some more of the facilities provided by the RISC OS library. There is no complete example program to illustrate all of them, but fragments of code are given as illustrations.

The topics covered are:

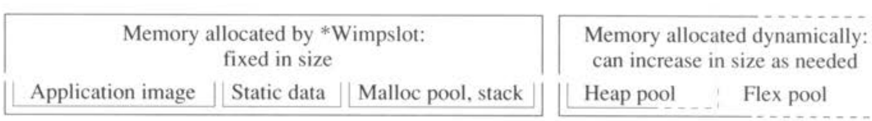
- memory management
- responding to idle and unknown events
- loading and saving.

All of these require some practice to get right.

Memory management

RISC_OSlib includes two sets of functions for memory management: see `flex.h` and `heap.h`. The functions are an alternative to the ANSI C library functions such as `malloc()` and `free()`. The problem with using the standard functions is that they are limited to the memory initially allocated to the application by `*WimpSlot`. If all of this memory becomes used up (or if it becomes so badly fragmented a free area of the right size cannot be found), the application has no way of claiming any more. The `flex` and `heap` functions, on the other hand, are able to request more memory from the operating system as necessary, and they cope well with fragmentation. Flex is a little tricky to use, but its advantages far outweight this. Heap is the simpler of the two modules. We will deal with it first.

The following diagram may help clarify the areas of memory used by an application when it is running:



Heap allocation is similar to `malloc()` in that a pointer to the block allocated is returned to the caller: the routine to do this is called `heap_alloc()`. Memory may be released with `heap_free()`. Before you use heap, you must call `heap_init()`. This must be done after `flex` has been initialised with `flex_init()`, and before any calls to flex functions. The reasons for this are discussed below.

Flex is the more general of the two. The basic difference between flex storage allocation and heap allocation is that blocks allocated with `flex` may be moved in order to make the best use of the available memory, without the application being informed directly. This would cause problems if the application simply kept pointers to blocks of allocated memory: when RISC_OSlib moves blocks around, the pointers would cease to point to the right place. The approach that is used instead is to tell `flex` the address of the pointer to the block, which it will note in its own internal tables. If the block is moved, `flex` can then change the pointer. Thus, instead of writing code such as:

```
char *pointer;
pointer = malloc(size);
```

you would write:

```
char *pointer;
/* Allocate memory, passing in the address of 'pointer' */
flex_alloc((flex_ptr) &pointer, size);
```

The value `&pointer` is called the *anchor* of the flex block.

This may sound a little awkward if you are used to using `malloc` and `free`, but you will soon find that it becomes easy to use.

There is a restriction which you must be aware of if you use `flex`. The anchor of each flex block allocated must not itself move. This means that you cannot have flex pointers within blocks of memory allocated by `flex`. The following program fragments shows why this will not work. (You can skip this explanation if you like – the important point to remember is not to place flex pointers in areas of memory allocated by `flex`.)

```
#define MemSize 100
struct  s_control_block
{
  char  *data;
  ... other fields ...
  int   size;
```

```
} *pointer;

/* Allocate a control block */
flex_alloc((flex_ptr) &pointer, sizeof(struct s_control_block));

/* Allocate the data block itself */
/* The next line is wrong!!! */
flex_alloc((flex_ptr) &(pointer->data), MemSize);
```

Suppose that `flex` moves the control block we allocated at `pointer`, and then tries to move the data block referenced by `pointer->data`. When the memory was allocated, `flex` made a note of the anchors `&pointer` and `&(pointer->data)`. Now suppose it moves the control block, and changes the value of the variable `pointer`. It then moves the data block and attempts to change its anchor. But the anchor it noted was an address within the control block at its original location, and the control block is no longer there. Consequently, `pointer->data`, with `pointer` having its new value, is not changed, rendering the pointer to the data block no longer valid. Not only that, but `flex` will have changed the location which originally contained `pointer->data` and which may by now be part of some other block.

A second restriction is that you must not make a copy of the pointer to a block allocated by `flex`. The reason here is simply that `flex` only knows of one anchor for each block. If the location of the block changes, the original pointer will be changed, but not any copies of it. A place where this can easily cause problems is in passing a pointer to a flex block as an argument to a function. Thus, the following example would not work:

```
void some_function(char *data)
{
  printf("%s\n", data);
}
...

char *pointer;
flex_alloc((flex_ptr) &pointer, 256);
...

/* The next call can go wrong!!! */
some_function(pointer);
```

A safe alternative is to introduce an extra level of indirection:

```
void some_function(char **data)
{
  printf("%s\n", *data);
}
```

```
...

char *pointer;
flex_alloc((flex_ptr) &pointer, 256);
...

/* Pass pointer to reference - this is OK */
some_function(&pointer);
```

You must call `flex_init()` before attempting to use `flex`. There are functions for allocating and freeing memory, and for changing the size of an allocated block of memory both by adding or removing memory from the end of the block, and adding or removing memory from part way through the block.

A flex block will only ever move as the result of other calls to `flex`. You can also rely on the first block that was allocated using `flex` never moving.

We can now conclude the discussion of how `heap` works. Internally, `heap` keeps its allocated memory in a flex block. If you call `heap_init()` immediately after `flex_init()`, then the heap block will be the first flex block allocated, and consequently it will never move. Note that the heap block need not be the first flex block allocated, if you choose otherwise. In this case, if the flex block used for `heap` moves, then `heap_alloc()` will return an error.

Responding to idle and unknown events

There are two special types of event, which are only passed to your application if you specifically claim them.

Idle events

Idle events are generated by the window manager when nothing else is happening: they correspond to the `Null_Reason_Code` generated by the SWI `Wimp_Poll`. Applications should only claim idle events if they want to do some activity which continually needs processor time; an example is dragging the selection box in Draw. When you claim idle events, they are directed to the event handler for the window you specify, as an event of type `wimp_ENULL`. Idle events should be released as soon as they are no longer needed by the application. To claim and release idle events, use the function `win_claim_idle_events()`. See `win.h` for more details.

Unknown events

Unknown events are events which are not associated with a specific window. The following events are considered to be unknown:

- user drag events: `wimp_EUSERDRAG`
- menu events: `wimp_EMENU`
- losing and gaining the caret: `wimp_ELOSECARET` and `wimp_EGAINCARET`
- user message send events, `wimp_ESEND` and `wimp_ESENDWANTACK`, for any **except** the following message types: `wimp_MCLOSEDOWN`, `wimp_MDATASAVE`, `wimp_MDATALOAD`, `wimp_MHELPREQUEST`
- user message acknowledge events: `wimp_EACK`.

To claim unknown events, register one or more unknown event processors, and optionally an unknown event handler. When an unknown event occurs, it is offered to each of the unknown event processors in turn, until either one deals with the event, or they have all been tried. If none of them deals with the unknown event, it is then passed on to the unknown event handler, if any, or discarded if there isn't one.

To register an unknown event processor, call `win_add_unknown_event_processor()`, giving it the name of the unknown event handler, and a handle for any extra data you wish to be passed to the processor (as for window event handlers). The processors are called in the reverse of the order in which you register them, ie the most recently registered one is called first. See the type `win_unknown_event_processor()` in `win.h` for the type used for unknown event processors. Each unknown event processor must return a value indicating whether it has handled the event or not. Unknown event processors may be cancelled by calling `win_remove_unknown_event_processor()`.

To register an unknown event handler, call `win_claim_unknown_events()`, specifying a window handle. Unknown events are then directed to the normal event handler for that window. You can cancel the unknown event handler by calling the same function with an argument of –1.

Loading and saving

There are functions in the RISC OS library for loading and saving data, using the same style as Acorn's own applications. The functions implement the data transfer protocol, as described under `Wimp_SendMessage` in the *Window Manager* chapter of the *RISC OS Programmer's Reference Manual.* They may be used for loading from and saving to files, and for transfers from and to other applications via RAM.

Loading

To load data, use the functions in `xferrecv`. Loading a file is initiated when the user drags a file to either a window opened by the application or its icon bar entry. A message is then sent to the corresponding event handler. In the event handler, you should have something like:

```
... other event cases ...
  case wimp_ESEND:
  case wimp_ESENDWANTACK:

    switch (e->data.msg.hdr.action)
    {
      case wimp_MDATASAVE:   /* import data */
        load_ram();
        break;

      case wimp_MDATALOAD:   /* insert data */
      case wimp_MDATAOPEN:
        load_file();
        break;

      ... other message cases ...
  }
```

For a load via RAM, the code is as follows (in outline):

```
static char *data;
void load_ram(void)
{
  int estsize;   /* Estimate size of file */

  /* Get the type of the file being loaded, and an estimate of its size */
  int filetype = xferrecv_checkimport(&estsize)

  if (filetype != -1)
  {
    int  final_size;

    ... any necessary pre-load checks, e.g. valid filetype ...
    ... allocate a block 'estsize' long, at 'data' ...

    /* Initiate the load */
```

```
    if (final_size = xferrecv_do_import(data, estsize, buffer_processor),
        final_size >= 0)
    {
      ... load was ok ...
    }
    else
    {
      ... error during loading ...
    }
  }
  else /* Filetype of -1 indicates we should try to load via a file */
  {
    load_file();
  }
}
```

Here we check that the load really is via RAM transfer, and if not try to load from a file instead. If we decide to go ahead with the load, a call to `xferrecv_do_import()` is made. If the data being loaded fills up the buffer, then `buffer_processor()` is called. This function is not defined here: what it must do is either to empty the buffer, or to extend it. For a more precise specification, see the definition of `xferrecv_buffer_processor()` in `xferrecv.h`.

The code for loading from a file is:

```
void load_file(void)
{
  char *filename;

  /* Fetch the type and name of the file */
  int filetype = xferrecv_checkinsert(&filename);

  ... any necessary pre-load checks, e.g. valid filetype ...
  ... load file ...

  /* Indicate load is completed */
  xferrecv_insertfileok();
}
```

The work of loading the file here can be done using the standard methods for reading files, such as `os_file()`, or the ANSI C file functions. The file size is usually read first, so that the entire buffer can be allocated before loading starts.

In this function, a pointer to the name of the file being loaded is placed in `filename` by `xferrecv_checkinsert()`. This pointer does not remain valid permanently, and if you want to preserve it (for example, to use in a window title), you should copy it to a buffer of your own. The call `xferrecv_file_is_safe()` may be used to check the validity of the name.

In both cases, it is good practice to turn the hourglass on during the load. Suitable calls for turning it on and off are:

```
/* Turn hourglass on */
visdelay_begin();
...
/* Turn hourglass off */
visdelay_end();
```

Saving

There are two levels to the functions for saving. The bulk of the work is handled by the functions in `xfersend`, which are used for transferring data from the application to the destination of the save operation. The functions in `saveas` are used to display a save dialogue box and respond to dragging the icon from it. It is better to use `saveas`, since this makes the user interface for saving consistent with Acorn's applications. However, even if you are using `saveas`, you will still call some of the functions in `xfersend`, as described in this section.

A save operation is typically initiated by the user choosing something like **Save as** from a menu. In this case, you would start the operation with code such as:

```
int  filetype = ...;    /* Type of file */
char *name    = ...;    /* File name to be placed in dialogue box */
int  estsize  = ...;    /* Estimated size of file */
char *data    = ...;    /* Data to be saved */
saveas(filetype, name, estsize, saver_proc, sender_proc, print_proc,
       (void *)data);
```

The three functions are used for:

- saving the file directly (`xfersend_saveproc`)
- transferring it via RAM a buffer-full at a time (`xfersend_sendproc`)
- printing the file (`xfersend_printproc`).

The last parameter to `saveas()` is an arbitrary handle which is passed to these functions. It is a convenient way of indicating the source of the data to be saved. The functions are called when the user has dragged the file icon to its destination, or specified the name of the file to be saved.

`saveas()` requires the presence of a template called `xfer_send` in the application's resources: it contains the save dialogue box.

Outlines of functions for `saver_proc()` and `sender_proc()` are:

```
/* saver_proc: type is the same as xfersend_saveproc */
BOOL saver_proc(char *filename, void *handle)
{
  ... any checks, eg valid file name ...
  ... save file, using any conventional method ...
}

/* sender_proc: type is the same as xfersend_sendproc */
BOOL sender_proc(void *handle, int *maxbuf)
{
  char *data   = ...; /* Location of the data being sent, initially from handle */
  int  length = ...; /* Size of the block being sent */

  ... here there would be some sort of loop, getting chunks of data up to
        *maxbuf in length, and sending them with code something like: */

  while (...)
  {
    /* The data save itself */
    if (!xfersend_sendbuf(data, length)) return FALSE;
    else
    {
      /* Advance to next block */
      data += length; /* For example */
    }
  }
}
```

As with loading, you may want to turn on the hourglass during the save operation.

Note that you can specify the send and print functions as being NULL.

Using Draw files

You can use the RISC OS library to display files in the the format used by Draw in your own applications. The format of Draw files is described in the *RISC OS Programmer's Reference Manual*; Acorn intends that this should be treated as a standard for graphical data in RISC OS. There are two interfaces to the code for displaying Draw files. You can either draw entire files: the header for this is `drawfdiag.h`. Alternatively, you can draw files object by object; see `drawfobj.h`. The object-level interface also includes functions for adding and deleting objects. In both cases, it is possible to define your own object types, by specifying a function to handle unknown object types thus allowing you to extend the Draw file format.

Draw files use their own coordinate system. When rendering a Draw file, the origin of the file (ie coordinate (0,0)) is mapped to work area coordinate (0,0). The function `draw_shift_diag()` may be used to shift all the coordinates in a Draw file. In addition, the coordinates used in Draw objects are not the same as those used for work area and screen coordinates. There are macros and functions which convert between the two systems. These just multiply the coordinates by the relevant factor: they take no account of where the Draw file origin is. Note also that the Draw file headers refer to the work area and screen coordinates collectively as 'screen units'. This refers purely to their size: you are responsible for applying any further origin shifts to convert them to the coordinate system of the work area or the screen as a whole.

An application which illustrates many of the points described in the preceding sections can be found on Disc 1 as `$.DeskEgs.!DrawEx`. It does not set itself up on the icon bar when it is started; instead, it simply opens a window. Draw files dragged to the window are displayed in it. There is a window menu, with the usual **Info** and **Quit** entries, plus an entry to save the contents of the window: this entry is shaded when there is no file loaded. The application is closed down either by choosing **Quit**, or by closing the window.

The source code is not described here: it is left as an exercise for the reader to see how it works. You may also like to look at the `!Run` file, which shows how to make sure the necessary modules are loaded. Overlooking this is a common source of apparently serious errors in desktop applications.

The programming techniques illustrated by the application include:

- loading and saving files
- using `flex`
- using the active window count to handle closedown
- rendering Draw files.

One thing you may find it useful to look at in detail is the method used to extend flex blocks during a load via RAM. The code to do this is quite simple, but it is easy to get wrong. See the functions `drawex_load_ram()` and `drawex_ram_loader()`.

Common application features

There are a number of functions in the RISC OS library which are intended to help you produce applications with a similar appearance to those written by Acorn. Some of these have already been examined. This section briefly describes some of the others. As usual, for full details, look at the relevant parts of the chapter entitled *RISC OS library reference section*.

Coordinate conversion

You will often need to convert between the work area coordinates and screen. This is not difficult: the *Window Manager* chapter in the *RISC OS Programmer's Reference Manual* describes how to do it. However, you may find it convenient to use the functions in `coord.h` to do the conversion. Using these functions may make it clearer exactly what is happening in the source of your program. There are functions for converting x and y coordinates, points and boxes to either work area or screen coordinates, together with some extra functions used to move boxes, and determine if boxes overlap, and if a line intersects with a box. The conversion functions take a pointer to a `coords_cvtstr` object as a parameter. This consists of a box and two scroll values. You can obtain a suitable value for this parameter from the data structures returned by a number of Wimp functions. For example, the 'box', 'x' and 'y' fields of a `wimp_openstr`, or the 'box', 'scx' and 'scy' fields of a `wimp_redrawstr` are both suitable. Thus a typical fragment which might appear in a redraw loop is:

```
wimp_redrawstr r;
int       screen_x, workarea_x;
...
screen_x = coords_x_toscreen(workarea_x, (coords_cvtstr *)&r.box);
```

You can always obtain the box and scroll values for the current window by finding the window state with `wimp_get_wind_state()`.

Colour translation

Some of the RISC OS graphics primitives such as the draw module and sprite plotting allow colours to be specified as full RGB (red/green/blue) values. RGB colours are usually referred to as 'true' colours. At any instant the desktop will only be able to display approximations to the true colours, specified using 'Gcol' values. The functions in colourtran.h are used to convert between these two ways of referring to colours. You can find further details in the *ColourTrans Module* chapter of the *RISC OS Programmer's Reference Manual*. One point to note about using these functions is that they require the ColourTrans module to be loaded. If you use them, the application's !Run file should include (something like) the following:

```
if "<System$Path>" = "" then Error 0 System resources cannot be found
|
RMEnsure ColourTrans 0.51 RMLoad System:Modules.Colours
RMEnsure ColourTrans 0.51 Error You need ColourTrans 0.51 or later
```

There are separate sets of functions for setting colours to be used in ordinary graphics operations, and for use with anti-aliased fonts.

Colour menus

The function colourmenu_make() constructs a menu of the current desktop colours. You can see an example of this kind of menu in Edit, where it is used for the **Foreground** and **Background** entries of the **Display** submenu. Menus of this form are used when you want to select one of the standard desktop colours, rather than a true colour.

If you do want the user to be able to select a true colour, you can call the function dboxtcol, which allows the red, green and blue levels of a colour to be set using sliders, or by specifying numerical values.

Dialogue boxes

There are a number of functions for handling dialogue boxes. Some of these have already been introduced; here we look at some more of them. You may also find it instructive to look at some dialogue boxes in the templates files of standard applications, using the template editor: this will give you some idea of how they are constructed and what button types you use for the various sorts of field. You can use dialogue boxes both as part of the menu trees, as already described, or on their own. The only difference between these is how you display the dialogue box: for a menu tree, use dbox_show() and for a 'standalone' one, use dbox_showstatic().

As described in the *RISC OS Programmer's Reference Manual*, the fields of a dialogue box consist of icons. You can change the contents of the fields using the routine dbox_setfield(). dbox_setnumeric() can be used to place a number in a field. Values from the fields may be read back with dbox_getfield() and dbox_getnumeric(). Fields may be faded, as for menu items, with dbox_fadefield() and dbox_unfadefield(), to cancel the effect.

To recognise when an action has occurred in a dialogue box, you can either call dbox_fillin(), which enters a Wimp polling loop until a field has been activated, or register your own event handler for the dialogue box with dbox_eventhandler(). The first of these is simpler and usually provides all the flexibility you need. When dbox_fillin() returns, you should call dbox_persist(). This will tell you whether the dialogue box is to be removed from the screen or not. A typical use of these functions is:

```
dbox   dialogue;
BOOL   filling = TRUE;     /* TRUE until the dbox is to be removed */

/* Create dialogue box */
if ((dialogue = dbox_new(<name of the dbox>)) == 0)
  ... error ...

... fill in initial values for fields with dbox_setfield, etc. ...

/* Display the dbox. This is for a dbox in a menu tree */
dbox_show(dialogue);

/* Fill in the dialogue box */
while (filling)
{
  switch (dbox_fillin(dialogue))
  {
    /* Clauses for each field that has an effect */
    case <field number>:
      ... get field contents with dbox_getfield, etc. ...

...
      /*
        Use the following line on (for example) OK and Cancel buttons
      */
      filling = dbox_persist();
      break;

     ... more similar clauses ...

    /* Use the next clause if the dbox has a close icon */
     case dbox_CLOSE:
       filling = FALSE;
       break;

     /* Clauses for uninteresting fields */
     default:        /* Do nothing */
      break;
```

```
  }
}

/* Get rid of the dialogue box */
dbox_dispose(&dialogue);
```

Some special properties of dialogue boxes are worth noting. If there are writeable fields in the dialogue box, the dbox code interprets the up and down arrows to move the caret between them, in field order. Pressing Return advances the caret to the next writeable field. Field 0 may be used in a special way here. If you press Return on the last writeable field, field 0 will be activated, and `dbox_fillin()` will hence return 0 to the caller. If your dialogue box contains an **OK** button, it should normally be field 0, so that repeatedly pressing Return will eventually activate it.

Besides the functions in `dbox.h`, there are also three subsidiary dialogue box functions. `dboxtcol` has already been described. `dboxfile` is a function for handling file dialogue boxes, similar to those used by `xfersend`. `dboxquery.h` is used to handle dialogue boxes that consist simply of a message to the user with **YES** and **NO** buttons, as used by Edit and Draw to ask whether unsaved data is to be discarded or not when a window is closed. See the header files for more details of these functions.

Finally, don't forget to call `template_init()` and `dbox_init()` during the initialisation of your application, in order to load the templates from the application's resources.

Magnifier

The magnifier is used for operations such as **Zoom** in Draw. The function `magnify_select()` can be used to read a magnification factor from a zoom dialogue box. To use this function, you must have a template called `magnifier` in your application's template file.

Displaying and editing text

There are a large number of functions for displaying and editing text in a window, in a similar way to Edit. See `txt.h`, `txtedit.h` and `txtwin.h` for full details. The conceptual model used is as follows.

Text is kept as a linear array of characters, known as a 'txt'. All character codes are allowed. There is a pointer into this called the 'dot', which marks the current editing position, and some other pointers known as markers, which are used (for example) for selecting blocks of text.

The characters are displayed in a window, with a newline for each '\n' character in the buffer. Screen updates happen for each text operation, but the result is only sure to be good when redraws can happen too. When a txt is displayed, the dot is constrained to be visible and the text will be scrolled in order to achieve this.

You can insert and delete characters at the dot, during which the markers will continue to point at the character that they pointed at before. There are functions for moving the dot and querying its position.

You can indicate a part of the buffer as being selected. Characters in the selection are displayed highlighted. No other special meaning is given to the selection. The selection and the dot need not coincide. There are functions to create, delete, move and query markers.

A txt is implemented using a single buffer containing the text, with a gap at the dot. Moving the dot involves a block move of the intervening text, but insertions and deletions are fast. The text buffer is expanded if necessary (it is held in a flex block).

The basic text editing functions are defined in `txt.h`. There are also higher level functions, which are intended for building complete text editors, in `txtedit.h`. `txtwin.h` adds further functions for displaying the same text in multiple windows.

The functions are based on the code in Edit, and you may find it useful to compare them with the way you can see Edit working.

Alarm functions

If your application needs to do some activity after a fixed length of time (for example, periodically updating a window), there are two ways in which it can do this. The first is to claim idle events and repeatedly examine the time. The second, and preferable, way is to use the functions defined in `alarm.h`. These allow you to set one or more alarms, specifying the time when they will occur. When the alarm is triggered an event handler is called. You may have more than one alarm set simultaneously. See `alarm.h` for details of the functions.

Tracing desktop applications

During the development of your program, you will probably want to trace what is happening. One way of doing this is with `werr`, but this is often inconvenient, since it requires acknowledgement by clicking in the OK box, and because it obscures part of the screen, which will cause problems if it is used in a redraw loop.

An alternative is to use the functions defined in `trace.h`. They display their results directly onto the screen using `printf`. This is rather messy, since the trace output does not appear in a window and may thus be overwritten by the output from other application, though it will never interfere with the application. One trick that is sometimes useful is to spool the output to a file, using *Spool so that the trace output can be examined later. In this case, all the other graphics output will also be sent to the file, and you may find it useful to include some sort of distinctive text in your trace output which you can search for using a text editor; for example:

```
tracef0(">>> This is some trace\n");
```

In order to use tracing, you will have to define TRACE, either using a line in your program such as

```
#define TRACE
```

or using the –D command line parameter to the C compiler. When trace is not set, the trace functions are treated as macros which convert into empty statements. Thus, the call to the trace function may be left in your program even when you no longer need the trace. This is often useful for generating debugging and production versions of the program from the same source. Tracing may also be turned on and off dynamically, with `trace_on()` and `trace_off()`, when trace has been compiled in.

There is a general trace function which takes an arbitrary number of parameters (like `printf`), and five functions which take a fixed number of parameters. The general trace function cannot be omitted by leaving `TRACE` undefined, because of the properties of C macro expansion. The functions with a fixed number of parameters are therefore generally preferable.

Where do you go from here?

The next step is to try writing a desktop application of your own. You might like to take one of the example programs and extend it. For example, you could add multiple windows to DrawEx, or allow it to display text and sprite files as well as Draw files, or to display an animated sequence of pictures. Don't try to use all of RISC_OSlib in one go! It is better to become familiar with it gradually, using the functions as you need them. You may also find it useful to glance at the RISC_OSlib header files which have not been mentioned here. They all correspond more or less exactly to sections in the *RISC OS Programmer's Reference Manual.*

Writing desktop applications takes a little getting used to. In particular, the flow of control through the program is driven primarily by events from the window manager. This makes the programming a little harder, but it leads to applications which respond better to user actions. Using RISC_OSlib, you should find that programming in this style soon comes naturally.

Example programs

The following example desktop applications are supplied on Disc 1 in the directory `$.DeskEgs`:

- !Wexample and !DrawEx, as described above.
- !Balls64, which displays coloured balls in a window.
- !Life, which runs Conway's game of life in several windows simultaneously. This is coded as a demonstration of RISC_OSlib, not for speed or as a high-quality animation of Life.

How to use the template editor

The template editor – FormEd – is an application which allows you to define windows on the screen, and save the definitions in a file ready for loading by your application. This is the approach used in Acorn's own applications, and you will find it makes the process of creating windows for your applications much easier.

FormEd is supplied along with Release 3 of the C compiler. To use it, you first need to understand the program interface of the window system, as described in the *RISC OS Programmer's Reference Manual*. Refer, in particular, to the descriptions of the SWIs Wimp_CreateWindow and Wimp_CreateIcon, in the *Window manager* chapter. The account that follows also assumes an understanding of template files; these are described in the same chapter.

Starting FormEd

Start FormEd in the same way as any other RISC OS application, by double-clicking on the FormEd application icon in a directory display, or on a template file. Provided either that FormEd has been 'seen' by the system, or that the run path has been set, the template file will be loaded along with FormEd. If a template file does not appear to load properly, give more memory to FormEd before it starts, using the Task Manager.

If you start FormEd without an existing template file, you can open a new template by clicking on the FormEd icon on the icon bar.

A loaded copy of FormEd can edit only one template file; if you want to edit more than one at once, load a second copy of FormEd. However, you will probably find this very confusing, as a window does not necessarily identify itself as belonging to one application rather than another.

Editing a template file

When you load an application's template file into FormEd, all the windows used by that application are displayed on the screen. Most of the window areas can be regarded as 'pictures' of the real window you will see when running the application; for example, try loading the template file for the Configure application (make a copy before you do this!). The main Configure window will appear, but you will not be able to use it to, for example, set the mouse speed.

While most most parts of the template frame (Title bar, scroll bars, Back icon, etc) have their normal effect, the Close icon is used to delete a template from the file. Be particularly careful, therefore, that you do not delete a template and then save the template file with the same filename (unless, of course, that's what you want to do).

Clicking Menu on a template produces a top-level menu: the upper half relates to icon properties, and the bottom half to window properties. Which of these features are selectable depends on exactly where the pointer was when you clicked Menu: if it was on an icon, you will be able to amend or renumber the icon as well as the window itself. If the pointer was not on an icon, you will still be able to create a new icon.

Each of the window and icon properties in the menu and its submenus maps directly onto bitfields listed in the Wimp_CreateWindow and Wimp_CreateIcon descriptions in the *RISC OS Programmer's Reference Manual*. However, you should also note the following points:

- Each window within a template file must have a name or *identifier* which is unique to that template file. The identifier is used when the window definition is loaded by a call to SWI Wimp_LoadTemplate. To assign an identifier to a window, select **Identifier** from the top-level menu.
- The icons you add to a window are numbered in sequence, starting at 0. If two icons are placed so that they overlap, the window manager uses the numbering to determine which should obscure the other: higher numbers are displayed obscuring lower numbers. You may therefore need to change the number allocated to an icon; this is done by swapping over two icon numbers. Click Menu over the icon you wish to renumber and select **Renumber**. Type in the number of the icon you want to swap with the currently selected icon, and the two will switch numbers.
- To add a new window to a template file, click on the FormEd icon on the icon bar; the new window will appear on the screen.

- Because of the way the icon flag bitfield is organised, you cannot use anti-aliased text within a filled icon. Setting the **Anti-aliased** option in the **Icon flags** menu will make the background and foreground colour unselectable.
- The **V centred** (vertically centred) option applies only to sprites, not to text.

Loading sprites into templates

A template file is often constructed with reference to a specific set of sprites. To display the sprites within the templates, drag the sprite icon from its directory display onto the FormEd icon on the icon bar. You can move sprite icons within templates, and delete them, but to edit a sprite, use the Paint application.

Editing ROM utility templates

It's also possible to update the template files used by ROM utilities. These reside in the `deskfs:` filing system in the ROM. They are accessed via the environment variable Wimp$Path, so by updating this to search a directory of your own first where your updated template files reside, you can replace the window templates used by the utilities in the ROM.

A worked example

This example uses the template file for the Palette utility, which demonstrates some of the points described above.

First, make a copy of the template file from the ROM by typing the following at the Command Line prompt:

```
*adfs
*dir
*cdir templates
*copy deskfs:templates.palette templates.palette
```

Add the following to the !Boot file for your machine:

```
*set Wimp$Path adfs::4.$.,deskfs:
```

This assumes that you have a hard disc. If you don't, amend the line above as appropriate, depending on the location of your templates file.

Now return to the desktop and double-click on your copy of the templates file. Two dialogue boxes will appear: the palette's main tool window and the save box.

The main tool window is covered in cross-hatching: this indicates that the application (in this case, the palette utility code) is involved in redrawing the window.

You can move the window around the screen by dragging on its title bar in the normal way. Move the window to another position, then save the template file using the save box on the menu that appears when you press Menu over the FormEd icon bar icon. Now reset the machine. You will find that the palette utility appears in the new position – where you dragged its window in the template file.

Double-click on the template file again, to re-enter FormEd. Press Menu over the palette template window.

The menu that appears is divided into two parts. The upper half effects whatever icon you were pointing at when you pressed Menu; the lower half affects the window as a whole. By entering the **Window flags**, **Colours**, and **Work area** submenus, you can see which bits within the window description are set and which are clear: compare this with the Wimp_CreateWindow section in the *RISC OS Programmer's Reference Manual*. By clicking or typing on entries within these submenus you can affect such things as the title text and the colours of the window.

Some changes you might make will prevent the code from working properly, as they actually change the behaviour of the window in the program that operates it. Others, such as colour changes, are reasonable ways of setting your own choices for how the palette utility should appear.

Each of the sixteen colour selection buttons is an icon. Point at the black one and press Menu. You can see that it is icon number 16 in this window. By working through the **Amend icon #16** submenu, you can inspect and change every aspect of this icon in exactly the same way as with the whole window.

To move or resize an icon, take the following steps:

1 Ensure that its button type (within the **Amend** submenu) is set to **Click/drag**, so that it responds to dragging events.

2 Drag the icon with Select to move it.

3 Drag the icon with Adjust to change its size.

You can move the icon a pixel at a time using the **Move icon** submenu. Using other top-level submenus, you can make a copy of an icon, or renumber it.

RISC OS library reference section

This chapter presents brief summaries of all the functions in the RISC OS library, grouped alphabetically by header. You should also refer to the *RISC OS Programmer's Reference Manual* for related information.

akbd

These functions provide access to the keyboard under the Wimp.

akbd_pollsh

Checks if Shift is depressed.

Syntax:	`int akbd_pollsh(void)`
Returns:	1 if Shift is depressed, 0 otherwise.

akbd_pollctl

Checks if Control is depressed.

Syntax:	`int akbd_pollctl(void)`
Returns:	1 if Control is depressed, 0 otherwise.

akbd_pollkey

Checks if user has typed ahead.

Syntax:	`int akbd_pollkey(int *keycode)`
Parameters:	`int *keycode` – value of key pressed
Returns:	1 if user has typed ahead. Also passes value of key back through keycode.
Other Information:	Function keys appear as values > 256 (produced by Wimp)

alarm

These functions provide alarm facilities for Wimp programs, using non-busy waiting.

alarm_init

Initialises the alarm system.

Syntax: `void alarm_init(void)`

Parameters: `void`.

Returns: `void`.

Other Information: If this call is made more than once, any pending alarms are cancelled.

alarm_timenow

Reports the current monotonic time.

Syntax: `int alarm_timenow(void)`

Parameters: `void`.

Returns: the current monotonic time.

Other Information: This timer cannot be set by programs, and can therefore be relied on to increment every centisecond. It wraps every few months.

alarm_timedifference

Returns the difference between two times.

Syntax: `int alarm_timedifference(int t1, int t2)`

Parameters: `int t1` – the earlier time
`int t2` – the later time.

Returns: difference between t1 and t2.

Other Information: Times are as in `SWI OS_ReadMonotonicTime`. Deals with wrap-round of timer.

alarm_set

Sets an alarm at the given time.

Syntax: `void alarm_set(int at, alarm_handler proc, void *handle)`

Parameters: `int at` – time at which alarm should occur
`alarm_handler proc` – function to be called at alarm time
`void *handle` – caller-supplied handle to be passed to function.

Returns: `void`.

Other Information: The supplied function is called before passing the event on to any idle event claimer windows. `at` is in terms of the monotonic centisecond timer. The supplied function is passed the time at which it was called. If you have enabled idle events, these are still returned to you; otherwise, RISC_OSlib uses idle events internally to implement alarm calls (using non-busy waiting via `wimp_pollidle()`).

alarm_remove

Removes an alarm which was set for a given time with a given handle.

Syntax: `void alarm_remove(int at, void *handle)`

Parameters: `int at` – the time at which the alarm was to be made

`void *handle` – the given handle.

Returns: `void`.

Other Information: If no such alarm exists, this call has no effect.

alarm_removeall

Removes all alarms which have a given handle.

Syntax: `void alarm_removeall(void *handle)`

Parameters: `void *handle` – the handle to search for.

Returns: `void`.

alarm_anypending

Informs the caller whether an alarm with a given handle is pending.

Syntax: `BOOL alarm_anypending(void *handle)`

Parameters: `void *handle` – the handle.

Returns: True if there are any pending alarms for this handle.

alarm_next

Informs the caller whether an alarm is pending and, if so, for when it is.

Syntax: `BOOL alarm_next(int *result)`

Parameters: `int *result` – time for which alarm is pending

Returns: `True` if an alarm is pending.

Other Information: This should be used by polling loops (if you use the standard `while(TRUE) event_process();` loop, this is done for you). If a polling loop finds that an alarm is set it should use `wimp_pollidle` (with earliest time set to `*result` of `alarm_next()`) rather than `wimp_poll` to do its polling. If you handle idle events yourself, your handler should use `call_next` to call the next alarm function upon receiving an idle event (ie `wimp_ENULL`).

alarm_callnext

Calls the next alarm handler function.

Syntax: `void alarm_callnext(void)`

Parameters: `void`.

Returns: `void`.

Other Information: This is done for you if you use `event_process()` to do your polling (or even if you reach down as far as using `wimpt` for polling).

baricon

Installs the named sprite as an icon on the icon bar and registers a function to be called when Select is clicked.

Syntax: `wimp_i baricon(char *spritename, int spritearea, baricon_clickproc p)`

Parameters: `char *spritename` – name of sprite to be used
`int spritearea` – area where sprite is
`baricon_clickproc p` – pointer to function to be called on click of Select

Returns: the icon number of the installed icon (of type `wimp_i`). This will be passed to function `p` on click of Select.

Other Information: For details of installing a menu handler for this icon see `event_attachmenu()`.

baricon_newsprite

Changes the sprite used on the icon bar.

Syntax: `wimp_i baricon_newsprite(char *newsprite)`

Parameters: `char *newsprite` – name of new sprite to be used

Returns: the icon number of the installed icon sprite.

Other information: `newsprite` must be held in the same area as the sprite used in `baricon()`.

bbc

These functions provide BBC-style graphics and mouse/keyboard control.

bbc: text output functions

The following functions provide BBC-style text output functions. They are retained to allow 'old-style' operations; you are recommended to use SWI calls via `kernel.h` in the C library.

bbc_vdu

Outputs a single character.
Syntax: `os_error *bbc_vdu(int)`

bbc_vduw

Outputs a double character.
Syntax: `os_error *bbc_vduw(int)`

bbc_vduq

Outputs multiple characters. Ctl is a control character. The number of further parameters is appropriate to Ctl (`vduq` knows how many to expect, and assumes the correct parameters have been passed).
Syntax: `os_error *bbc_vduq(int ctl,...)`

bbc_stringprint

Displays a null-terminated string.
Syntax: `os_error *bbc_stringprint(char *)`

bbc_cls

Clears text window.
Syntax: `os_error *bbc_cls(void)`

bbc_colour

Sets text colour.
Syntax: `os_error *bbc_colour(int)`

bbc_pos

Returns X coordinate of text cursor.
Syntax: `os_error *bbc_pos(void)`

bbc_vpos

Returns Y coordinate of text cursor.
Syntax: `os_error *bbc_vpos(void)`

bbc_tab

Positions text cursor at given coordinates.
Syntax: `os_error *bbc_tab(int,int)`

txt: graphics output functions

bbc_plot

Carries out a given plot operation.
Syntax: `os_error *bbc_plot(int plotnumber, int x, int y)`

bbc_mode

Sets the screen mode.
Syntax: `os_error *bbc_mode(int)`

bbc_move

Moves the graphics cursor to the absolute coordinates given.
Syntax: `os_error *bbc_move(int, int)`

bbc_moveby

Moves the graphics cursor to a position relative to its current text cursor position.
Syntax: `os_error *bbc_moveby(int, int)`

bbc_draw

Draws a line to the given absolute coordinates.
Syntax: `os_error *bbc_draw(int, int)`

bbc_drawby

Draws a line to a position relative to the current graphics cursor.
Syntax: `os_error *bbc_drawby(int, int)`

bbc_rectangle

Plots a rectangle, given LeftX, BottomY, Width, and Height.
Syntax: `os_error *bbc_rectangle(int,int,int,int)`

bbc_rectanglefill

Plots a solid rectangle, given LeftX, BottmY, Width, and Height.
Syntax: `os_error *bbc_rectanglefill(int,int,int,int)`

bbc_circle

Draws a circle, given Xcoord, Ycoord, and Radius.
Syntax: `os_error *bbc_circle(int, int, int)`

bbc_circlefill

Draws a solid circle, given Xcoord, Ycoord, and Radius.
Syntax: `os_error *bbc_circlefill(int, int, int)`

bbc_origin

Moves the graphics origin to the given absolute coordinates.
Syntax: `os_error *bbc_origin(int,int)`

bbc_gwindow

Sets up a graphics window, given BottomLeftX, BottomLeftY, TopRightX, and TopRightY.
Syntax: `os_error *bbc_gwindow(int, int, int, int)`

bbc_clg

Clears the graphics window.
Syntax: `os_error *bbc_clg(void)`

bbc_fill

Flood-fills area X,Y, filling from X,Y until either a non-background colour or the edge of the screen is reached.
Syntax: `os_error *bbc_fill(int, int)`

bbc_gcol

Sets a graphics colour to the given value.
Syntax: `os_error *bbc_gcol(int, int)`

bbc_tint

Sets the grey level of a colour: use tint 0-3, as it gets shifted for you.
Syntax: `os_error *bbc_tint(int,int)`

bbc_palette

Physical to logical colour definition: Logical colour, Physical colour, Red level, Green level, Blue level.

Syntax: `os_error *bbc_palette(int,int,int,int,int)`

bbc_point

Finds the logical colour of a pixel at the indicated coordinates x, y.

Syntax: `int bbc_point(int,int)`

bbc_vduvar

Reads a single VDU or mode variable value, for the current screen mode.

Syntax: `int bbc_vduvar(int varno)`

bbc_vduvars

Reads several VDU or mode variable values. `vars` points to a sequence of ints, terminated by –1. Each is a VDU or mode variable number, and the corresponding values will be replaced by the value of that variable.

Syntax: `os_error *bbc_vduvars(int *vars /*in*/, int *values /*out*/)`

bbc_modevar

Reads a single mode variable, for the given screen mode.

Syntax: `int bbc_modevar(int mode, int varno)`

bbc: other calls

bbc_get

Returns a character from the input stream. 0x1xx is returned if an escape condition exists.

Syntax: `int bbc_get(void)`

bbc_cursor

Alters cursor appearance. Argument takes values 0 to 3.

Syntax: `os_error *bbc_cursor(int)`

bbc_adval

Reads data from analogue ports or gives buffer data.

Syntax: `int bbc_adval(int)`

bbc_getbeat

Returns current beat value.

Syntax: `int bbc_getbeat(void)`

bbc_getbeats

Reads beat counter cycle length.

Syntax: `int bbc_getbeats(void)`

bbc_gettempo

Reads rate at which beat counter counts.

Syntax: `int bbc_gettempo(void)`

bbc_inkey

Returns character code from an input stream or the keyboard.

Syntax: `int bbc_inkey(int)`

bbc_rnd

Returns a random number.

Syntax: `unsigned bbc_rnd(unsigned)`

bbc_beats

Sets beat counter cycle length.

Syntax: `os_error *bbc_beats(int)`

bbc_settempo

Sets rate at which beat counter counts.

Syntax: `os_error *bbc_settempo(int)`

bbc_sound

Makes or schedules a sound. Parameters: Channel, Amplitude, Pitch, Duration, and Future Time.

Syntax: `os_error *bbc_sound(int, int, int, int, int)`

bbc_soundoff

Deactivates the sound system.

Syntax: `os_error *bbc_soundoff(void)`

bbc_soundon

Activates the sound system.

Syntax: `os_error *bbc_soundon(void)`

bbc_stereo

Sets the stereo position for the specified channel.

Syntax: `os_error *bbc_stereo(int, int)`

bbc_voices

Sets the number of sound channels.

Syntax: `os_error *bbc_voices(int)`

colourmenu

Creates a Wimp colour setting menu.

colourmenu_make

Creates a menu containing the sixteen Wimp colours, with an optional `None` entry. Text in colour is written in black or white, depending on the background.

Syntax:	`menu colourmenu_make(char *title, BOOL includeNone)`
Parameters:	`char *title` – null-terminated string for menu title `BOOL includeNone` – whether to include 'None' entry
Returns:	On successful completion, pointer to created menu structure, otherwise null.
Other Information:	Hits on this menu start from 1 as for other menus (see `menu` module for details).

colourtran

C interface to the ColourTrans SWIs.

colourtran_select_table

Sets up a translation table in a buffer, given a source mode and palette, and a destination mode and palette.

Syntax:

```
os_error *colourtran_select_table (int source_mode,
wimp_paletteword *source_palette, int dest_mode,
wimp_paletteword *dest_palette, void *buffer)
```

Parameters:

`int source_mode` – source mode

`wimp_paletteword *source_palette` – source palette

`int dest_mode` – destination mode

`wimp_paletteword *dest_palette` – destination palette

`void *buffer` – pointer to store for the table.

Returns: possible error condition.

colourtran_select_GCOLtable

Sets up a list of GCOLs in a buffer, given a source mode and palette, and a destination mode and palette.

Syntax:

```
os_error *colourtran_select_GCOLtable (int source_mode,
wimp_paletteword *source_palette, int dest_mode,
wimp_paletteword *dest_palette, void *buffer)
```

Parameters:

`int source_mode` – source mode

`wimp_paletteword *source_palette` – source palette

`int dest_mode` – destination mode

`wimp_paletteword *dest_palette` – destination palette

`void *buffer` – pointer to store for the list of GCOLs.

Returns: possible error condition.

colourtran_returnGCOL

Informs the caller of the closest GCOL in the current mode to a given palette entry.

Syntax: `os_error *colourtran_returnGCOL (wimp_paletteword entry, int *gcol)`

Parameters:
`wimp_paletteword entry` – the palette entry
`int *gcol` – returned GCOL value.

Returns: possible error condition.

colourtran_setGCOL

Informs the caller of the closest GCOL in the current mode to a given palette entry, and also sets the GCOL.

Syntax: `os_error *colourtran_setGCOL (wimp_paletteword entry, int fore_back, int gcol_in, int *gcol_out)`

Parameters:
`wimp_paletteword entry` – the palette entry
`int fore_back` – set to 0 for foreground, set to 128 for background
`int gcol_in` – GCOL action
`int *gcol_out` – returned closest GCOL.

Returns: possible error condition.

colourtran_return_colournumber

Informs the caller of the closest colour number to a given palette entry, in the current mode and palette.

Syntax: `os_error *colourtran_return_colournumber (wimp_paletteword entry, int *col)`

Parameters:
`wimp_paletteword` – the palette entry
`int *col` – returned colour number.

Returns: possible error condition.

colourtran_return_GCOLformode

Informs the caller of the closest GCOL to a given palette entry, destination mode and destination palette.

Syntax: `os_error *colourtran_return_GCOLformode (wimp_paletteword entry, int dest_mode, wimp_paletteword *dest_palette, int *gcol)`

Parameters:
`wimp_paletteword entry` – the palette entry
`int dest_mode` – destination mode
`wimp_paletteword *dest_palette` – destination palette
`int *gcol` – returned closest GCOL.

Returns: possible error condition.

colourtran_return_colourformode

Informs the caller of the closest colour number to a given palette entry, destination mode and destination palette.

Syntax:
```
os_error *colourtran_return_colourformode
(wimp_paletteword entry, int dest_mode,wimp_paletteword
*dest_palette, int *col)
```

Parameters:
`wimp_paletteword entry` – the palette entry
`int dest_mode` – destination mode
`wimp_paletteword *dest_palette` – destination palette
`int *col` – returned closest colour number.

Returns: possible error condition.

colourtran_return_OppGCOL

Informs the caller of the furthest GCOL in the current mode from a given palette entry.

Syntax:
```
os_error *colourtran_return_OppGCOL (wimp_paletteword
entry, int *gcol)
```

Parameters:
`wimp_paletteword entry` – the palette entry
`int *gcol` – returned GCOL value.

Returns: possible error condition.

colourtran_setOppGCOL

Informs the caller of the furthest GCOL in the current mode from a given palette entry, and also sets the GCOL.

Syntax:
```
os_error *colourtran_setOppGCOL (wimp_paletteword entry,
int fore_back, int gcol_in, int *gcol_out)
```

Parameters:
`wimp_paletteword entry` – the palette entry
`int fore_back` – set to 0 for foreground, set to 128 for background
`int gcol_in` – GCOL action
`int *gcol_out` – returned furthest GCOL.

Returns: possible error condition.

colourtran_return_Oppcolournumber

Informs the caller of the furthest colour number from a given palette entry, in the current mode and palette.

Syntax:
```
os_error *colourtran_return_Oppcolournumber
(wimp_paletteword entry, int *col)
```

Parameters: `wimp_paletteword` – the palette entry
`int *col` – returned colour number.

Returns: possible error condition.

colourtran_return_ OppGCOLformode

Informs the caller of the furthest GCOL from a given palette entry, destination mode and destination palette.

Syntax:
```
os_error *colourtran_return_OppGCOLformode
(wimp_paletteword entry, int dest_mode, wimp_paletteword
*dest_palette, int *gcol
```

Parameters: `wimp_paletteword entry` – the palette entry
`int dest_mode` – destination mode
`wimp_paletteword *dest_palette` – destination palette
`int *gcol` – returned furthest GCOL.

Returns: possible error condition.

colourtran_return_ Oppcolourformode

Informs the caller of the furthest colour number from a given palette entry, destination mode and destination palette.

Syntax:
```
os_error *colourtran_return_Oppcolourformode
(wimp_paletteword entry int dest_mode, wimp_paletteword
*dest_palette, int *col)
```

Parameters: `wimp_paletteword entry` – the palette entry
`int dest_mode` – destination mode
`wimp_paletteword *dest_palette` – destination palette
`int *col` – returned furthest colour number.

Returns: possible error condition.

colourtran_GCOL_ tocolournumber

Translates a GCOL to a colournumber (assuming 256-colour mode).

Syntax:
```
os_error *colourtran_GCOL_tocolournumber (int gcol, int
*col)
```

Parameters: `int gcol` – the GCOL
`int *col` – returned colour number.

Returns: possible error condition.

colourtran_ colournumbertoGCOL

Translates a colour number to a GCOL (assuming 256-colour mode).

Syntax:
```
os_error *colourtran_colournumbertoGCOL (int col, int
*gcol)
```

Parameters: `int col` – the colour number

`int *gcol` – the returned GCOL.

Returns: possible error condition.

colourtran_ returnfontcolours

Informs the caller of the font colours to match the given colours.

Syntax: `os_error *colourtran_returnfontcolours (font *handle, wimp_paletteword *backgnd,wimp_paletteword *foregnd,int *max_offset)`

Parameters: `font *handle` – the font's handle

`wimp_paletteword *backgnd` – background palette entry

`wimp_paletteword *foregnd` – foreground palette entry

`int *max_offset`

Returns: possible error condition.

Other Information: Closest approximations to fore/background colours will be set, and as many intermediate colours as possible (up to a maximum of `*max_offset`). Values are returned through the parameters.

colourtran_ setfontcolours

Informs the caller of the font colours to match the given colours, and calls `font_setfontcolour()` to set them.

Syntax: `os_error *colourtran_setfontcolours (font *handle,wimp_paletteword *backgnd,wimp_paletteword *foregnd, int *max_offset)`

Parameters: `font *handle` – the font's handle

`wimp_paletteword *backgnd` – background palette entry

`wimp_paletteword *foregnd` – foreground palette entry

`int *max_offset`

Returns: possible error condition.

Other Information: Closest approximations to fore/background colours will be set, and as many intermediate colours as possible (up to a maximum of `*max_offset`). Values are returned through the parameters. `Font_setfontcolours()` is then called with these as parameters.

colourtran_invalidate_cache

To be called when the palette has changed since a call was last made to a function in this module, or a Draw object was rendered.

Syntax: `os_error *colourtran_invalidate_cache (void)`

Parameters: `void`

Returns: possible error condition

coords

This file contains functions for working in the window coordinate system. Functions are provided to convert between screen and work area coordinates, and perform other simple operations on points, lines, or 'boxes'.

It is conventional to think of the point (0,0) as appearing at the top lefthand corner of a document.

coords_x_toscreen/ coords_y_toscreen

Converts x/y work area coordinates into x/y absolute screen coordinates.

Syntax:

```
int coords_x_toscreen(int x, coords_cvtstr *r)
int coords_y_toscreen(int y, coords_cvtstr *r)
```

Parameters: `int x` or `int y` – x or y coordinate in work area coordinates
`coords_cvtstr *r` – conversion box (screen coordinates and scroll offsets).

Returns: x or y screen coordinates.

coords_x_toworkarea/ coords_y_toworkarea

Converts x/y screen coordinates into x/y work area coordinates.

Syntax:

```
int coords_x_toworkarea(int x, coords_cvtstr *r)
int coords_y_toworkarea(int y, coords_cvtstr *r)
```

Parameters: `int x` or `int y` – x or y coordinate in screen coordinates
`coords_cvtstr *r` – conversion box (screen coordinates and scroll offsets).

Returns: x or y work area coordinates.

coords_box_toscreen

Converts a 'box' of workarea coordinates into a 'box' of screen coordinates.

Syntax: `void coords_box_toscreen(wimp_box *b, coords_cvtstr *r)`

Parameters: `wimp_box *b` – workarea box to be converted
`coords_cvtstr *r` – conversion box (screen coordinates and scroll offsets).

Returns: void.

Other Information: b is converted 'in situ' into screen coordinates (ie its contents change).

coords_box_toworkarea

Converts a 'box' of screen coordinates into a 'box' of workarea coordinates.

Syntax: `void coords_box_toworkarea(wimp_box *b, coords_cvtstr *r)`

Parameters: `wimp_box *b` – screen box to be converted
`coords_cvtstr *r` – conversion box (screen coordinates and scroll offsets).

Returns: void.

Other Information: b is converted 'in situ' into workarea coordinates (ie its contents are changed).

coords_point_toscreen

Converts a point (x,y) from workarea coordinates to screen coordinates.

Syntax: `void coords_point_toscreen(coords_pointstr *point, coords_cvtstr *r)`

Parameters: `coords_pointstr *point` – the point in workarea coordinates
`coords_cvtstr *r` – conversion box (screen coordinates and scroll offsets).

Returns: void.

Other Information: point is converted 'in situ' into screen coordinates (ie its contents are changed).

coords_point_toworkarea

Converts a point (x,y) from screen coordinates to workarea coordinates.

Syntax: `void coords_point_toworkarea(coords_pointstr *point, coords_cvtstr *r)`

Parameters: `coords_pointstr *point` – the point in screen coordinates
`coords_cvtstr *r` – conversion box (screen coordinates and scroll offsets).

Returns: void.

Other Information: point is converted 'in situ' into workarea coordinates (ie its contents are changed).

coords_withinbox

Informs the caller if a point (x,y) lies within a 'box'.

Syntax: `BOOL coords_withinbox(coords_pointstr *point, wimp_box *box)`

Parameters:
`coords_pointstr *point` – the point
`wimp_box *box` – the box.

Returns: True if point lies within the box.

coords_offsetbox

Offset a 'box' by a given x and y displacement.

Syntax: `void coords_offsetbox(wimp_box *source, int byx, int byy, wimp_box *result)`

Parameters:
`wimp_box *source` – the box to be moved
`int byx` – x displacement
`int byy` – y displacement
`wimp_box *result` – box when offset.

Returns: `void`.

Other Information: `source` and `result` are permitted to point at the same box.

coords_intersects

Informs the caller whether a line intersects a given 'box'.

Syntax: `BOOL coords_intersects(wimp_box *line, wimp_box *rect, int widen)`

Parameters:
`wimp_box *line` – the line
`wimp_box *rect` – the box
`int widen` – width of line (same units as line and rect).

Returns: True if line intersects box.

coords_boxesoverlap

Informs the caller whether two 'boxes' cover any common area.

Syntax: `BOOL coords_boxesoverlap(wimp_box *box1, wimp_box *box2)`

Parameters:
`wimp_box *box1` – one box
`wimp_box *box2` – the other box.

Returns: True if boxes overlap.

dbox

This file contains functions concerned with the creation, deletion and manipulation of dialogue boxes. It is important to note that the structure of your dialogue templates is an integral part of your program. Always use symbolic names for templates and for fields and action buttons within them. Templates for the dialogue boxes can be loaded using the template module in this library. See the chapter entitled *How to use the Template Editor* for how to use the RISC OS Template Editor in conjunction with this interface. A dbox is an abstract dialogue box handle.

dbox: creation and deletion functions

dbox_new

Builds a dialogue box from a named template. Template editor (FormEd) may have been used to create this template in the `Templates` file for the application.

Syntax: `dbox dbox_new(char *name)`

Parameters: `char *name` – template name (from templates previously read in by `template_init`), from which to construct dialogue box. `name` is as given when using FormEd to create template.

Returns: On successful completion, pointer to a dialogue box structure, otherwise null (eg when not enough space).

Other Information: This only creates a structure; it doesn't display anything! However, it does register the dialogue box as an active window with the window manager.

dbox_dispose

Disposes of a dialogue box structure.

Syntax: `void dbox_dispose(dbox*)`

Parameters: `dbox*` – pointer to pointer to a dialogue box structure

Returns: `void`.

Other Information: This also has the side-effect of hiding the dialogue box, so that it no longer appears on the screen. It also 'un-registers' it as an active window with the window manager and event processor.

dbox_show

Displays the given dialogue box on the screen.

Syntax: `void dbox_show(dbox)`

Parameters: `dbox` – dialogue box to be displayed (typically created by `dbox_new`)

Returns: `void`.

Other Information: Typically used when dialogue box is from a submenu so that it disappears when the menu is closed. If called when this dialogue box is showing, it has no effect. The show will occur near the last menu selection or the last caret setting (whichever is most recent).

dbox_showstatic

Displays the given dialogue box on the screen, and leaves it there, until explicitly closed.

Syntax: `void dbox_showstatic(dbox)`

Parameters: `dbox` – dialogue box to be displayed (typically created by dbox_new)

Returns: `void`.

Other Information: This is typically not used from menu selection, as it will persist longer than the menu (otherwise, it is the same as `dbox_show`).

dbox_hide

Hides a previously displayed dialogue box.

Syntax: `void dbox_hide(dbox)`

Parameters: dbox – dialogue box to be hidden

Returns: `void`.

Other Information: This does not release any storage; it just hides the dialogue box. If called when the dialogue box is already hidden, it has no effect.

dbox fields

A dialogue box has a number of fields, labelled from 0. There are the following distinct field types:

- action fields. Mouse clicks here are communicated to the client. The fields are usually labelled `go`, `quit`, etc. Set/GetField apply to the label on the field, although this is usually set up in the template.
- output fields. These display a message to the user, using SetField. Mouse clicks etc. have no effect.

- input fields. The user can type into these, and simple local editing is provided. Set/GetField can be used on the textual value, or Set/GetNumeric if the user should type in numeric values.
- on/off fields. The user can click on these to display their on/off status. They are always 'off' when the dialogue box is first created. The template editor can set up mutually exclusive sets of these at will. Set/GetField apply to the label on this field, Set/GetNumeric set/get 1 (on) and 0 (off) values.

The function keys can be used instead of the mouse to 'click' action and on/off fields. In addition, if a letter key is pressed, an attempt will be made to match the first capital letter found in any action field, and 'click' on that field. For example, 'y' will match Yes, and 'd' will match reDo.

dbox_field/dbox_fieldtype

type `dbox_field` values are field numbers within a dialogue box. They indicate what sort a field is (ie action, output, input, on/off).

dbox_setfield

Sets the given field, within the given dialogue box, to the given text value.

Syntax: `void dbox_setfield(dbox, dbox_field, char*)`

Parameters: `dbox` – the chosen dialogue box
`dbox_field` – chosen field number
`char*` – text to be displayed in field.

Returns: `void`.

Other Information: If the function is applied to a non-text field, it has no effect. If the field is an indirected text icon, the text length is limited by the size value used when setting up the template in the template editor. Any longer text will be truncated to this length. Otherwise, text is truncated to 12 characters (11 text + 1 null). If the dialogue box is currently showing, the change is immediately visible. This function is really only useful will indirect icons.

dbox_getfield

Puts the current contents of the chosen text field into a buffer, whose size is given as the third parameter.

Syntax: `void dbox_getfield(dbox, dbox_field, char *buffer, int size)`

Parameters: `dbox` – the chosen dialogue box
`dbox_field` – the chosen field number
`char *buffer` – buffer to be used
`int size` – size of buffer.

Returns: `void`.

Other Information: If the function is applied to a non-text field, the null string is put in the buffer. If the length of the chosen field (plus null-terminator) is larger than the buffer, the result will be truncated.

dbox_setnumeric

Sets the given field, in the given dialogue box, to the given integer value.

Syntax: `void dbox_setnumeric(dbox, dbox_field, int)`

Parameters: `dbox` – the chosen dialogue box
`dbox_field` – the chosen field number
`int` – field's contents will be set to this value.

Returns: `void`.

Other Information: If the field is of type `input/output`, the integer value is converted to a string and displayed in the field. If the field is of type `action` or `on/off`, a non-zero integer value selects this field; zero deselects it.

dbox_getnumeric

Gets the integer value held in the chosen field of the chosen dialogue box.

Syntax: `int dbox_getnumeric(dbox, dbox_field)`

Parameters: `dbox` – the chosen dialogue box
`dbox_field` – the chosen field number.

Returns: integer value held in chosen field.

Other Information: If the field is of type `on/off` then return value of 0 means `off`, 1 means `on`. Otherwise, the return value is the integer equivalent of the field contents.

dbox_fadefield

Makes a field unselectable (ie faded by Wimp).

Syntax: `void dbox_fadefield(dbox d, dbox_field f)`

Parameters: `dbox d` – the dialogue box in which field resides
`dbox_field f` – the field to be faded.

Returns: `void`.

Other Information: Fading an already faded field has no effect.

dbox_unfadefield

Makes a field selectable (ie 'unfades' it).

Syntax: `void dbox_unfadefield(dbox d, dbox_field f)`

Parameters: `dbox d` – the dialogue box in which field resides
`dbox_field f` – the field to be unfaded.

Returns: `void`.

Other Information: Unfading an already selectable field has no effect.

dbox: events from dialogue boxes

A dialogue box acts as an input device: a stream of characters comes from it as if it were a keyboard, and an up-call can be arranged when input is waiting. dialogue boxes may have a `close` button that is separate from their action buttons, usually in the header of the window. If this is pressed, `CLOSE` is returned: this should lead to the dialogue box being invisible. If the dialogue box represents a particular pending operation, the operation should be cancelled.

dbox_get

Tells caller which action field has been activated in the chosen dialogue box.

Syntax: `dbox_field dbox_get(dbox d)`

Parameters: `dbox` – the chosen dialogue box.

Returns: field number of activated field.

Other Information: This should only be called from an event handler (since this is the only situation where it makes sense).

dbox_eventhandler

Registers an event handler function for the given dialogue box.

Syntax: `void dbox_eventhandler(dbox, dbox_handler_proc, void* handle)`

Parameters: `dbox` – the chosen dialogue box
`dbox_handler_proc` – name of handler function
`void *handle` – user-defined handle.

Returns: `void`.

Other Information: When a field of the given dialogue box has been activated, the user-supplied handler function is called. The handler should be defined in the form: `void foo (dbox d, void *handle)`. When called, the function `foo` will be passed the relevant dialogue box, and its user-defined handle. A typical action in `foo` would be to call `dbox_get` to determine which field was activated. If handler==0 then no function is installed as a handler (and any existing handler is 'un-registered').

dbox_raweventhandler

Registers a 'raw' event handler for the given dialogue box.

Syntax: `void dbox_raw_eventhandler(dbox, dbox_raw_handler_proc, void *handle)`

Parameters: `dbox` – the given dialogue box
`dbox_raw_handler_proc` – handler function for event
`void *handle` – user-defined handle.

Returns: `void`.

Other Information: This registers a function which will be passed 'unvetted' window events. Under the window manager in RISC OS, the event will be a `wimp_eventstr*` (see Wimp module). The supplied handler function should return True if it processed the event; if it returns False, the event will be passed on to any event handler defined using `dbox_eventhandler()` as above. The form of the handler's function header is: `BOOL func (dbox d, void *event, void *handle)`.

dbox: pending operations

Dialogue boxes are often used to fill in the details of a pending operation. In this case a down-call driven interface to the entire interaction is often convenient. The following facilities aid this form of use.

dbox_fillin

Process events until a field in the given dialogue box has been activated.

Syntax: `dbox_field dbox_fillin(dbox d)`

Parameters: `dbox d` – the given dialogue box

Returns: field number of activated field.

Other Information: Handling of harmful events, like `dbox_popup` (below).

dbox_popup

Build a dialogue box, from a named template, assign message to field 1, do a `dbox_fillin`, destroy the dialogue box, and return the number of the activated field.

Syntax: `dbox_field dbox_popup(char *name, char *message)`

Parameters: `char *name` – template name for dialogue box
`char *message` – message to be displayed in field 1.

Returns: field number of activated field.

Other Information: 'Harmful' events are those which could cause the dialogue to fail (eg keystrokes, mouse clicks). These events will cause the dialogue box to receive a `CLOSE` event.

dbox_persist

When `dbox_fillin` has returned an action event, this function returns True if the user wishes the action to be performed, but the dialogue box to remain.

Syntax: `BOOL dbox_persist(void)`

Parameters: `void`.

Returns: `BOOL` – does the user want the dialogue box to remain on screen?

Other Information: The current implementation returns True when the user has clicked Adjust. The caller should continue round the fill-in loop if the return value is True (ie don't destroy the dialogue box).

dbox_syshandle

Allows the caller to get a handle on the window associated with the given dialogue box.

Syntax: `int dbox_syshandle(dbox)`

Parameters: `dbox` – the given dialogue box

Returns: window handle of dialogue box (this is a `wimp_w` under the RISC OS window manager).

Other Information: This could be used to hang a menu off a dialogue box, or to 'customise' the dialogue box in some way. `dbox_dispose` will also dispose of any such attached menus.

dbox_init

Prepare for use of dialogue boxes from templates.

Syntax: `void dbox_init(void)`

Parameters: `void`

Returns: `void`

Other Information: This function must be called **once** before any dialogue box functions are used. You should call `template_init()` before this function.

dboxfile

Displays dialogue box with message, input field, and OK field and allows input of filename.

Syntax: `void dboxfile(char *message, unsigned filetype, char *a, int bufsize)`

Parameters:
`char *message` – informative message to be displayed in dialogue box
`unsigned filetype` – OS-dependent type of file
`char *a` – default filename (initially) and also used for user-typed filename
`int bufsiz` – size of `a`.

Returns: `void`.

Other Information: The template for the dialogue box must be called `dboxfile_db`. Parameters correspond to the template's icon numbers as follows:

icon #0	OK button
icon #1	message
icon #2	filename

The template should have the following icons:

icon #0	a text icon containing text `OK` with button type `menu icon`
icon #1	an indirected text icon (possibly with a default message) with button type `never`
icon #2	an indirected text icon with button type `writeable`. See the `dboxfile_db` template used by Edit for an example.

The maximum length of `message` is 20. The char array pointed to by `a` will contain the typed-in file name of maximum length `bufsize` (if longer, truncated).

dboxquery

Displays a dialogue box, with YES and NO buttons, and a question, and gets reply.

Syntax: `dboxquery_REPLY dboxquery(char *question)`

Parameters: `char *question` – the question to be asked

Returns: reply by user.

Other Information: Question can be up to 120 chars long, 3 lines of 40 characters. Return will reply `yes`; Escape or `CLOSE` event will reply `cancel`. A call of `dbox_query(0)`, will reserve space for the dialogue box and return with no display. This will mean that space is always available for important things like asking to quit! The template for the dialogue box should have the following attributes:

window flags	moveable, auto-redraw. It is also advisable to have a title icon containing the name of your program (or other suitable text).
icon #1	the message icon: should have indirected text flag set, with button type `never` and validation string L40.
icon #0	the `YES` icon: should be text icon with text string set to `YES`; button type should be `menu icon`..
icon #2	the `NO` icon: should be text icon with text string set to `NO`; button type should be `menu icon`. See the query dialogue box in Edit for an example.

dboxtcol

Displays a dialogue box to allow the editing of a true colour value.

Syntax:

```
BOOL dboxtcol(dboxtcol_colour *colour /*inout*/, BOOL
allow_transparent, char *name, dboxtcol_colourhandler
proc, void *handle)
```

Parameters:

`dboxtcol_colour *colour` – colour to be edited
`BOOL allow_transparent` – enables selection of a 'see-through' colour
`char *name` – title to put in dialogue box.
`dboxtcol_colourhandler proc` – function to act on the colour change
`void *handle` – the handle passed to `proc`.

Returns: True if colour edited, user clicks OK.

Other Information: The dialogue box to be used should be the same as that used by Paint to edit the palette. If the user clicks Select on `OK`, the `proc` is called and the dialogue box is closed. If the user clicks Adjust on `OK`, the `proc` is called and the dialogue box stays on the screen. This allows the client of this function to use `proc` to, say, change a sprite's palette to reflect the edited colour value and then to cause a redraw of the sprite.

drawfdiag

This file contains functions concerned with the processing of Draw format files (diagram level interface). It defines the interface to the simplest version of the DrawFile module. It can read in files to diagrams and render them. There is no checking of whether the end of the diagram has been overrun.

To read in Draw files, it is expected that the caller will do the work of the I/O itself. To dispose of a diagram, the caller can just throw it away: the module does not keep any hidden information about what diagrams it has seen.

Some calls return an offset to the bad data on an error. This is not necessarily the start of an object: it may be bad data part way through it. The offset is relative to the start of the diagram.

The module cannot handle rectangle or ellipse objects: you should use a path instead.

Data types

Diagram: a pointer to the data and a length field. The length must be an exact number of words, and is the amount of space used in the diagram, not the size of the memory allocated to it.

Abstract handle for an object: The object handle is an offset from the start of the diagram to the object data. You may use it to set a pointer directly to an object, when using the object level interface

Error types: Where a routine can produce an error, the actual value returned is a `BOOL`, which is True if the routine succeeded. The error itself is returned in a block passed by the user; if `NULL`, then the details of the error are not passed back.

The error block may contain either an operating system error or an internal error. In the latter case, it consists of a code and possibly a pointer to the location in the file where the error occurred (if `NULL`, the location is not known or not specified). By convention, this should be reported by the caller in the form *`message`* `(location &xx in file)`. For a list of codes and standard errors, see `h.DrawfErrors`. The location is relative to the start of the data block in the diagram.

draw_verify_diag

Verifies a diagram which has been read in from a file.

Syntax: `BOOL draw_verify_diag(draw_diag *diag, draw_error *error)`

Parameters:
`draw_diag *diag` – the diagram to be verified
`draw_error *error` – the first error encountered (if any).

Returns: True if diagram is correct.

Other Information: Each object in the file is checked and the first error encountered causes return (with error set appropriately).

draw_append_diag

Merges two diagrams into one.

Syntax: `BOOL draw_append_diag(draw_diag *diag1, draw_diag *diag2, draw_error *error)`

Parameters:
`draw_diag *diag1` – diagram to which to append `diag2`
`draw_diag *diag2` – diagram to be appended to `diag1`
`draw_error *error` – possible error condition.

Returns: True if merge was successful.

Other Information: Both diagrams should have been processed by `draw_verify_diag()`. `Diag1`'s data block must be at least `diag1.length + diag2.length`. `Diag1.length` will be updated to its new appropriate value. `Diag1`'s bounding box will be set to the union of the bounding boxes of the two diagrams. Offsets of objects in `Diag1` may change due to a change in font table size (if `Diag2` has fonts). Errors referring to specific locations, refer to `Diag2`.

draw_render_diag

Renders a diagram with a given scale factor, in a given Wimp redraw rectangle.

Syntax: `BOOL draw_render_diag(draw_diag *diag, draw_redrawstr *r, double scale, draw_error *error)`

Parameters:
`draw_diag *diag` – the diagram to be rendered
`draw_redrawstr *r` – the Wimp redraw rectangle
`double scale` – scale factor
`draw_error *error` – possible error condition.

Returns: True if render was successful.

Other Information: The diagram must have been processed by `draw_verify_diag()`. `draw_redrawstr` is the same as `wimp_redrawstr`, which may be cast to it. Very small and negative scale factors will result in a run-time error (safe > 0.00009). The caller should do range checking on the scale factor. Following the normal convention for coordinate mapping, the part of the diagram rendered is found by mapping the top left of the diagram, in draw coord space onto a point: `(r->box.x0 - r->scx, r->box.y1 - r->scy)` in screen coordinates.

draw: memory allocation functions

draw_registerMemoryFunctions

Registers three functions to be used to allocate, extend and free memory, when rendering text objects.

Syntax: `void draw_registerMemoryFunctions(draw_allocate alloc, draw_extend extend, draw_free free)`

Parameters: `draw_allocate alloc` – pointer to function to be used for memory allocation
`draw_extend extend` – pointer to function to be used for memory extension
`draw_free free` – pointer to function to be used for memory freeing.

Returns: `void`.

Other Information: These three functions will be used only when rendering text area objects. Any memory allocated during rendering will be freed (using the supplied function) after rendering. If `draw_registerMemoryFunctions()` is never called, or if memory allocation fails, then an attempt to render a text area will produce no effect. The three functions should operate as follows:

- `int alloc(void **anchor, int n)`: allocate `n` bytes of store and set `*anchor` to point to them. Return 0 if store can't be allocated, otherwise non-zero.
- `int extend (void **anchor, int n)`: extend the block of memory which starts at `*anchor` to a total size of `n` bytes. `n` will always be positive, and the new memory should be appended to the existing block (which may be moved by the operation). Return 0 if the memory can't be allocated, otherwise non-zero.

- `void free(void **anchor)`: free the block of memory which starts at `*anchor`, and set `*anchor` to 0.

The specification for these three functions is the same as that for `flex_alloc`, `flex_extend` and `flex_free` (in the flex module), so these can be used as the three required functions.

draw_shift_diag

Shifts a diagram by a given distance.

Syntax: `void draw_shift_diag(draw_diag *diag, int xMove, int yMove)`

Parameters:
`draw_diag *diag` – the diagram to be shifted
`int xMove` – distance to shift in x direction
`int yMove` – distance to shift in y direction.

Returns: `void`.

Other Information: All coordinates in the diagram are moved by the given distance.

draw_querybox

Finds the bounding box of a diagram.

Syntax: `void draw_queryBox(draw_diag *diag, draw_box *box, BOOL screenUnits)`

Parameters:
`draw_diag *diag` – the diagram
`draw_box *box` – the returned bounding box
`BOOL screenUnits` – indication whether the box is to be specified in draw or screen units.

Returns: `void`.

Other Information: The bounding box of `diag` is returned in `box`. If `screenUnits` is true, `box` is in screen units, otherwise, it is in draw units.

draw_convertBox

Converts a box to/from screen coordinates.

Syntax: `void draw_convertBox(draw_box *from, draw_box *to, BOOL toScreen)`

Parameters:
`draw_box *from` – box to be converted
`draw_box *to` – converted box
`BOOL toScreen` – should set to True if conversion is to be from draw coordinates to screen coordinates. False makes conversion from screen coordinates to draw coordinates.

Returns: `void`.

Other Information: `from` and `to` may point to the same box.

draw_rebind_diag

Force the header of a diagram's bounding box to be exactly the union of the objects in it.

Syntax:
```
void draw_rebind_diag(draw_diag *diag)
```

Parameters: `draw_diag *diag` – the diagram.

Returns: `void`.

Other Information: The diagram should have been processed by `draw_verify_diag()` first.

draw: unknown object handling

New types of object can be added by registering an unknown object handler. The handler is called whenever an attempt is made to render an object whose tag is not one of the standard ones known to DrawFile. It is passed a pointer to the object to be rendered (cast to a `void *`), and a pointer to a block into which to write any error status. The object pointer may be cast to one of the standard Draw types (defined in the object level interface), or to a client-defined type. If an error occurs, the handler must return False and set up the error block; otherwise it must return True. Unknown objects must conform to the standard convention for object headers, ie one-word object tag; one-word object size; four-word bounding box. The unknown object handler is only called if the object is visible, ie if there is an overlap between its bounding box and the region of the diagram being rendered. The object size field must be correct, otherwise catastrophes will probably result.

draw_set_unknown_object_handler

Registers a function to be called when an attempt is made to render an object with an object tag which is not known.

Syntax:
```
draw_unknown_object_handler
draw_set_unknown_object_handler
(draw_unknown_object_handler handler, void *handle)
```

Parameters: `draw_unknown_object_handler handler` – the handler function

`void *handle` – arbitrary handle to pass to function.

Returns: The previous handler.

Other Information: The handler can be removed by calling with 0 as a parameter.

drawferror

Definition of error codes and standard messages for the Drawfile rendering functions. For each error, a code and the standard message are listed. See `drawfdiag`, above, for how to use the errors.

BadObject 1	Bad object
BadObjectHandle 2	Bad object handle
TooManyFonts 3	Too many font definitions
BBoxWrong 101	Bounding box coordinates are in the wrong order
BadCharacter 102	Bad character in string
ObjectTooSmall 103	Object size is too small
ObjectTooLarge 104	Object size is too large
ObjectNotMult4 105	Object size is not a multiple of 4
ObjectOverrun 106	Object data is larger than specified size
ManyFontTables 107	There is more than one font table
LateFontTable 108	The font table appears after text object(s)
BadTextStyle 109	Bad text style word
MoveMissing 110	Path must start with a move
BadPathTag 111	Path contains an invalid tag
NoPathElements 112	Path does not contain any line or curve elements
PathExtraData 113	There is extra data present at the end of a path object
BadSpriteSize 114	The sprite definition size is inconsistent with the object size
BadTextColumnEnd 115	Missing end marker in text columns
ColumnsMismatch 116	Actual number of columns in a text area object does not match specified number of columns
NonZeroReserved 117	Non-zero reserved words in a text area object
NotDrawFile 118	This is not a Draw file
VersionTooHigh 119	Version number too high
BadObjectType 120	Unknown object type
CorruptTextArea 121	Corrupted text area (must start with '\!")
TextAreaVersion 121	Text area version number is wrong or missing
MissingNewline 122	Text area must end with a newline character

BadAlign 123	Text area: bad \A code (must be L, R, C or D)
BadTerminator 124	Text area: bad number or missing terminator
ManyDCommands 125	Text area: more than one \D command
BadFontNumber 126	Text area: bad font number
UnexpectedCharacter 127	Text area: unexpected character in \F command
BadFontWidth 128	Text area: bad or missing font width in \F command
BadFontSize 129	Text area: bad or missing font size in \F command
NonDigitV 130	Text area: non-digit in \V command
BadEscape 131	Text area: bad escape sequence
FewColumns 133	Text area must have at least one column
TextColMemory 134	Out of memory when building text area (location field is always 0 for this error).

drawfobj

This file handles the processing of Draw format files (object level interface), and supplements the diagram level interface with routines for dealing with individual objects.

draw_create_diag

Creates an empty diagram (ie just the file header), with a given bounding box.

Syntax: `void draw_create_diag(draw_diag *diag, char *creator, draw_box bbox)`

Parameters: `draw_diag *diag` – pointer to store to hold diagram
`char *creator` – pointer to character string holding creator's name
`draw_box bbox` – the bounding box (in Draw units).

Returns: `void`.

Other Information: diag must point at sufficient memory to hold the diagram. The first 12 chars of `creator` are stored in the file header. `diag.length` is set appropriately by this function.

draw_doObjects

Renders a specified range of objects from a diagram.

Syntax: `BOOL draw_doObjects(draw_diag *diag, draw_object start, draw_object end, draw_redrawstr *r, double scale, draw_error *error)`

Parameters:

`draw_diag *diag` – the diagram

`draw_object start` – start of range of objects to be rendered

`draw_object end` – end of range of objects to be rendered

`draw_redrawstr *r` – Wimp-style redraw rectangle

`double scale` – the scale factor for rendering

`draw_error *error` – possible error condition.

Returns: True if render was successful.

Other Information: Parameters (except range) are used as in `draw_render_diag`, in diagram level module. The diagram must be verified before a call to this function If the range of objects inclues text with anti-aliasing fonts, you **must** call `draw_setFontTable` first. Very small (<0.00009) or negative scale factors will cause run-time errors.

draw_setFontTable

Scans a diagram for a font table object and records it for a subsequent call of `draw_doObjects`.

Syntax: `void draw_setFontTable(draw_diag *diag)`

Parameters: `draw_diag *diag` – the diagram to be scanned.

Returns: `void`.

Other Information: This function must be called for `draw_doObjects` to work on a sequence of objects that includes text objects using anti-aliasing fonts, but no font table object. The font table remains valid until either a different one is encountered during a call to `draw_doObjects`, or until `draw_render_diag` is called, or until a different diagram is rendered.

draw_verifyObject

Verifies the data for an existing object in a diagram.

Syntax: `BOOL draw_verifyObject(draw_diag *diag, draw_object object, int *size, draw_error *error)`

Parameters: `draw_diag *diag` – the diagram
`draw_object object` – the object to be verified
`int *size` – gets set to the amount of memory occupied by the object
`draw_error *error` – possible error condition.

Returns: True if object found and verified.

Other Information: Verifying an object ensures that its bounding box is consistent with the data in it; if not, no error is reported, but the box is made consistent. On an error, the location is relative to the start of the diagram. The object's size is returned only if `size` is a non-null pointer.

draw_createObject

Creates an object after a specified object in a given diagram.

Syntax:
```
BOOL draw_createObject(draw_diag *diag, draw_objectType
newObject, draw_object after, BOOL rebind, draw_object
*object, draw_error *error)
```

Parameters: `draw_diag *diag` – the diagram
`draw_objectType newObject` – the created object
`draw_object after` – the object after which the new object should be created
`BOOL rebind` – if True, the bounding box of the diagram is updated to the union of its existing value and that of the new object
`draw_object *object` – new object's handle
`draw_error *error` – possible error condition.

Returns: True if object was created OK.

Other Information: All data after the insertion point is moved down. `after` may be set to `draw_FirstObject/draw_LastObject` for inserting at the start/end of the diagram. The diagram must be large enough for the new data; its length field is updated. On an error, the location is not meaningful. The handle of the new object is returned in `object`. If this function is used to create a font table, `after` is ignored, and the object merged with the existing one (if such exists) or inserted at the start of the diagram otherwise. This can cause the font reference numbers to change; if a call to this function is followed by a `draw_translateText()`, the font change will be applied (this is only needed when anti-aliased fonts are used in text objects).

draw_deleteObjects

Deletes the specified range of objects from a diagram.

Syntax: `BOOL draw_deleteObjects(draw_diag *diag, draw_object start, draw_object end, BOOL rebind, draw_error *error)`

Parameters:
`draw_diag *diag` – the diagram
`draw_object start` – start of range of objects to be deleted
`draw_object end` – end of range of objects to be deleted
`BOOL rebind` – if set to True, then the diagram's bounding box will be set to the union of those remaining objects
`draw_error *error` – possible error condition.

Returns: True if objects deleted successfully.

Other Information: diagram length is updated appropriately.

draw_extractObject

Extracts an object from a diagram into a supplied buffer.

Syntax: `BOOL draw_extractObject(draw_diag *diag, draw_object object, draw_objectType result, draw_error *error)`

Parameters:
`draw_diag *diag` – the diagram
`draw_object object` – the object to be extracted
`draw_objectType result` – pointer to the buffer
`draw_error *error` – possible error division

Returns: True if the object was extracted successfully.

Other Information: The buffer for the result must be large enough to hold the extracted object (an object's size can be ascertained by calling `draw_verifyObject()`).

draw_translateText

Updates all font reference numbers for text objects following creation of a font table.

Syntax: `void draw_translateText(draw_diag *diag)`

Parameters: `draw_diag *diag` – the diagram.

Returns: `void`.

Other Information: If the font table has not been changed, this function does nothing.

drawftypes

This file contains declarations of all the data types needed for manipulating Draw objects at a low level, enabling you to examine or change their individual properties. For full details, refer to the header file on Disc 3: `$.RISC_OSlib.h.drawftypes`.

drawmod

This file provides a C interface to the Draw module (not to be confused with the Draw application). It defines a number of types used for PostScript-like operations, with enhancements (for full details, refer to the header file on Disc 3: `$.RISC_OSlib.h.drawmod`). The enhancements consist mainly of choice of fill style (fill including/excluding boundary etc), extra winding numbers, differing leading/trailing line caps and triangular line caps. It calls the Draw SWIs.

drawmod_fill

Emulates the Postscript 'fill' operator – ie closes open subpaths, flattens a path, transforms it to standard coordinates and fills the result.

Syntax:

```
os_error *drawmod_fill(drawmod_pathelemptr path_seq,
drawmod_filltype fill_style, drawmod_transmat *matrix,
int flatness)
```

Parameters:

`drawmod_pathelemptr path_seq` – sequence of path elements

`drawmod_filltype fill_style` – style of fill

`drawmod_transmat *matrix` – transformation matrix (0 for the identity matrix)

`int flatness` – flatness in user coordinates (0 means default).

Returns: possible error condition

drawmod_stroke

Emulates PostScript 'stroke' operator.

Syntax:

```
os_error *drawmod_stroke(drawmod_pathelemptr path_seq,
drawmod_filltype fill_style, drawmod_transmat *matrix,
drawmod_line *line_style)
```

Parameters:

`drawmod_pathelemptr path_seq` – sequence of path elements

`drawmod_filltype fill_style` – style of fill

`drawmod_transmat *matrix` – transformation matrix (0 means identity matrix)

`drawmod_line *line_style` – (see typedef in header file for details).

Returns: possible error condition.

drawmod_do_strokepath

Puts a path through all stages of `drawmod_stroke` except the final fill. The resulting path is placed in the buffer.

Syntax:
```
os_error *drawmod_do_strokepath(drawmod_pathelemptr
path_seq, drawmod_transmat *matrix, drawmod_line
*line_style, drawmod_buffer *buffer)
```

Parameters:
`drawmod_pathelemptr path_seq` – sequence of path elements
`drawmod_transmat *matrix` – transformation matrix
`drawmod_line *line_style` – see typedef in header file
`drawmod_buffer *buffer` – buffer to hold stroked path.

Returns: possible error condition.

drawmod_ask_strokepath

Puts a path through all stages of `drawmod_stroke`, except the fill, and returns the size of buffer needed to hold such a path.

Syntax:
```
os_error *drawmod_ask_strokepath(drawmod_pathelemptr
path_seq, drawmod_transmat *matrix, drawmod_line
*line_style, int *buflen
```

Parameters:
`drawmod_pathelemptr path_seq` – sequence of path elements
`drawmod_transmat *matrix` – transformation matrix
`drawmod_line *line_style` – (see typedef in header for details)
`int *buflen` – returned length of required buffer.

Returns: possible error condition.

drawmod_do_flattenpath

Flattens the given path, and puts it into the supplied buffer.

Syntax:
```
os_error *drawmod_do_flattenpath(drawmod_pathelemptr
path_seq, drawmod_buffer *buffer, int flatness)
```

Parameters: `drawmod_pathelemptr path_seq` – sequence of path elements
`drawmod_buffer *buffer` – buffer to hold flattened path
`int flatness` – required flatness.

Returns: possible error condition.

drawmod_ask_flattenpath

Puts the given path through the stages of `drawmod_flattenpath` and returns the size of buffer needed to hold the resulting path.

Syntax:

```
os_error *drawmod_ask_flattenpath(drawmod_pathelemptr
path_seq, int flatness, int *buflen)
```

Parameters: `drawmod_pathelemptr path_seq` – sequence of path elements
`int flatness` – required flatness
`int *buflen` – returned length of required buffer.

Returns: possible error condition.

drawmod_buf_transformpath

Puts a path through a transformation matrix and puts the result in the supplied buffer.

Syntax:

```
os_error *drawmod_buf_transformpath(drawmod_pathelemptr
path_seq, drawmod_buffer *buffer, drawmod_transmat
*matrix)
```

Parameters: `drawmod_pathelemptr path_seq` – sequence of path elements
`drawmod_buffer *buffer` – buffer to hold transformed path
`drawmod_transmat *matrix` – the transformation matrix.

Returns: possible error condition.

drawmod_insitu_transformpath

Puts a path through a transformation matrix by modifying the supplied path itself.

Syntax:

```
os_error
*drawmod_insitu_transformpath(drawmod_pathelemptr
path_seq, drawmod_transmat *matrix)
```

Parameters:	`drawmod_pathelemptr path_seq` – sequence of path elements `drawmod_transmat *matrix` – the transformation matrix.
Returns:	possible error condition.

drawmod_processpath

Puts a path through a set of processes used when doing Stroke and Fill.

Syntax:	`os_error *drawmod_processpath(drawmod_pathelemptr path_seq, drawmod_filltype fill_style, drawmod_transmat *matrix, drawmod_line *line_style, drawmod_options *options, int *buflen)`
Parameters:	`drawmod_pathelemptr path_seq` – sequence of path elements `drawmod_filltype fill_style` – style of fill `drawmod_transmat *matrix` – the transformation matrix `drawmod_line *line_style` – (see typedef in header for details) `drawmod_options *options` – this can have the values detailed below. Note: pass in address of a `draw_options` struct `int *buflen` – returned length of required buffer (only used when `options->tagtype == tag_fill && options->data.opts == option_countsize`).
Returns:	possible error condition.
Other Information:	Possible values for options:

`drawmod_insitu`	output to the input path (only if path size wouldn't change)
`drawmod_fillnormal`	fill path normally
`drawmod_fillsubpath`	fill path, subpath by subpath
OR an address	output bounding box of path to the word-aligned address, and three next words, with word-order lowX, lowY, highX, highY

OR a buffer to hold the processed path.

event

This file handles system-independent central processing for window system events.

event_process

Processes one event.

Syntax: `void event_process(void)`

Parameters: `void`.

Returns: `void`.

Other Information: If the number of current active windows is 0, the program exits. One event is polled and processed (with the exception of some complex menu handling, this really means passing the event on to the `win` module). Unless an application window is claiming idle events, this function waits when the processor is idle. Typically this should be called in a loop in the main function of the application.

event_anywindows

Informs the caller if there are any windows active that can process events.

Syntax: `BOOL event_anywindows(void)`

Parameters: `void`.

Returns: True if there are any active windows.

event_attachmenu

Attaches a menu and its associated handler function to the given window.

Syntax: `BOOL event_attachmenu(event_w, menu, event_menu_proc, void *handle)`

Parameters:
`event_w` – the window to which menu should be attached
`menu` – the menu structure
`event_menu_proc` – the handler for the menu
`void *handle` – caller-defined handle.

Returns: True if able to attach menu.

Other Information: The menu should have been created by a call to `menu_new` or something similar. When the user invokes a menu from the given window, this menu will be activated. The handler function will be called when the user selects a menu entry. The handler's parameter hit is a string containing a character for each level of nesting in a hierarchical menu structure, terminated by a 0 character. A call with `menu == 0` removes the attachment. To catch menu events on the icon bar, attach a menu to `win_ICONBAR` (defined in the `win` module).

event_attachmenumaker

Attaches to the given window a function which makes a menu when the user invokes a menu.

Syntax: `BOOL event_attachmenumaker(event_w, event_menu_maker, event_menu_proc, void *handle)`

Parameters:
`event_w` – the window to which the menu maker should be attached
`event_menu_maker` – the menu maker function
`event_menu_proc` – handler for the menu
`void *handle` – caller-defined handle

Returns: True if able to attach menu maker

Other Information: This works similarly to `event_attachmenu`, except that it allows you to make last minute changes to flags in the menu (such as ticks or fades), before displaying it. A call with `event_menu_maker==0` removes the attachment.

event_clear_current_menu

Clears the current menu tree.

Syntax: `void event_clear_current_menu(void)`

Parameters: `void`.

Returns: `void`.

Other Information: To be used to force all menus to be cleared from the screen.

event_is_menu_being_ recreated

Informs the caller if a menu is being recreated.

Syntax: `BOOL event_is_menu_being_recreated(void)`

Parameters: `void`.

Returns: `void`.

Other Information: Useful for when RISC_OSlib is recreating a menu in response to a click on Adjust (call it in a menu maker).

event: masking off events

event_setmask

Sets the mask used by `wimp_poll` and `wimpt_poll` when polling the Wimp.

Syntax: `void event_setmask(wimp_emask mask)`

Parameters: `wimp_emask mask` – the desired mask.

Returns: `void`.

Other Information: Bits of the mask are set if you want the corresponding events ignored (as in the `wimp_poll` SWI). For example, `event_setmask(wimp_ENULL | wimp_EPTRENTER)` will ignore nulls and pointer entering window events. The default mask is to ignore null events only.

event_getmask

Informs the caller of the current mask being used to poll the Wimp.

Syntax: `wimp_emask event_getmask(void)`

Parameters: `void`.

Returns: The mask currently used.

fileicon

Displays an icon representing a file, in a given window.

Syntax: `void fileicon(wimp_w, wimp_i, int filetype)`

Parameters: `wimp_w` – the given window's handle
`wimp_i` – an existing icon
`int filetype` – RISC OS file type (eg 0x0ffe)

Returns: `void`.

Other Information: If you want a file icon in a dialogue box then pass that dialogue box's window handle through first parameter, eg `fileicon((wimp_w)dbox_syshandle(d),...)`. The second parameter is the icon number of the required icon, within the template set up using FormEd. For an example see the `fileInfo` template for Edit.

flex

These functions provide memory allocation for interactive programs requiring large chunks of store.

flex_alloc

Allocates n bytes of store, obtained from the Wimp.

Syntax: `int flex_alloc(flex_ptr anchor, int n)`

Parameters: `flex_ptr anchor` – to be used to access allocated store
`int n` – number of bytes to be allocated.

Returns: 0 == failure, 1 == success

Other Information: You should pass the `&` of a pointer variable as the first parameter. The allocated store **must** then be accessed indirectly, through this, ie `(*anchor)[0]` ..`(*anchor)[n]`. This is important because the allocated store may later be moved. If there isn't enough store, returns zero leaving anchor unchanged.

flex_free

Frees the previously allocated store.

Syntax: `void flex_free(flex_ptr anchor)`

Parameters: `flex_ptr` anchor – pointer to allocated store.

Returns: `void`.

Other Information: *anchor will be set to 0.

flex_size

Informs the caller of the number of bytes allocated.

Syntax: `int flex_size(flex_ptr)`

Parameters: `flex_ptr` – pointer to allocated store

Returns: number of allocated bytes.

flex_extend

Extend or truncate the store area to have a new size.

Syntax: `int flex_extend(flex_ptr, int newsize)`

Parameters: `flex_ptr` – pointer to allocated store
`int newsize` – new size of store

Returns: 0 == failure, 1 == success.

flex_midextend

Extend or truncate store, at any offset.

Syntax: `int flex_midextend(flex_ptr, int at, int by)`

Parameters: `flex_ptr` – pointer to allocated store
`int at` – offset within the allocated store
`int by` – extent.

Returns: 0 == failure, 1 == success.

Other Information: If `by` is +ve, store is extended, and locations above `at` are copied up by `by`. If `by` is –ve, store is reduced, and any bytes beyond `at` are copied down to `at+by`.

flex_init

Initialise store allocation module.

Syntax: `void flex_init(void)`

Parameters:	`void.`
Returns:	`void.`
Other Information:	Must be called before any other functions in this module.

font

These functions provide access to RISC OS font facilities.

font_cacheaddress

Informs the caller of font cache used and font cache size.

Syntax:	`os_error * font_cacheaddress(int *version, int *cacheused, int *cachesize)`
Parameters:	`int *version` – version number `int *cacheused` – amount of font cache used (in bytes) `int *cachesize` – total size of font cache (in bytes).
Returns:	Possible error condition
Other Information:	Version number is *100, so v.1.07 would be returned as 107.

font_find

Gives the caller a handle to font, given its name.

Syntax:	`os_error * font_find(char* name, int xsize, int ysize, int xres, int yres, font*)`
Parameters:	`char *name` – the font name `int xsize, ysize` – x/y point size (in 16ths of a point) `int xres, yres` – x/y resolution in dots per inch `font*` – the returned font handle
Returns:	Possible error condition.

font_lose

Informs the font manager that a font is no longer needed.

Syntax:	`os_error * font_lose(font f)`
Parameters:	`font f` – the font.
Returns:	possible error condition.

font_readdef

Gets details about a font, given its handle.

Syntax: `os_error * font_readdef(font, font_def*)`

Parameters:
`font` – the font handle
`font_def*` – pointer to buffer to hold returned details.

Returns: possible error condition.

Other Information: This function fills in details about a font into the supplied buffer (a variable of type `font_def`). The fields of this buffer are as follows:

`name`	font name
`xsize, ysize`	x/y point size * 16
`xres, yres`	x/y resolution (dots per inch)
`usage`	number of times `Font_FindFont` has found the font minus number of times `Font_LoseFont` has been used on it
`age`	number of font accesses made since this one was last accessed.

font_readinfo

Informs the caller of the minimal area covering any character in the font bounding box.

Syntax: `os_error * font_readinfo(font, font_info*)`

Parameters:
`font` – the font handle
`font_info*` – pointer to buffer to hold returned details.

Returns: possible error condition.

Other Information: Fills in details of the font in the supplied buffer (variable of type `font_info`). The fields of this buffer are as follows:

`minx`	min x coord in pixels (inclusive)
`maxx`	max x coord in pixels (inclusive)
`miny`	min y coord in pixels (exclusive)
`maxy`	max y coord in pixels (exclusive).

font_strwidth

Determines the width of a string.

Syntax: `os_error * font_strwidth(font_string *fs)`

Parameters: `font_string *fs` – the string, with fields:
`s` – string itself
`x` – max x offset before termination
`y` – max y offset before termination
`split` – string split character
`term` – index of char to terminate by

Returns: possible error condition.

Other Information: On exit fs fields hold:

`s`	unchanged
`x`	x offset after printing string
`y`	y offset after printing string
`split`	number of split characters found; number of printable characters if `split` was –1
`term`	index into string at which terminated.

font_paint

Paints the given string at coordinates x,y.

Syntax: `os_error * font_paint(char*, int options, int x, int y)`

Parameters: `char` – the string
`int options` – set using 'paint options' defined in the header file
`int x, y` – coordinates (either OS or 1/72000 inch)

Returns: possible error condition.

font_caret

Sets the colour, size and position of the caret.

Syntax: `os_error *font_caret(int colour, int height, int flags, int x, int y)`

Parameters: `int colour` – EORed onto screen
`int height` – in OS coordinates
`int flags` – bit 4 set ==> OS coordinates, otherwise 1/72000 inch
`int x,y` – x/y coordinates.

Returns: possible error condition.

font_convertoos

Converts coordinates in 1/72000 inch to OS units.

Syntax: `os_error *font_converttoos(int x_inch, int y_inch, int *x_os, int *y_os)`

Parameters: `int x_inch, y_inch` – x/y coordinates in 1/72000 inch
`int *x_os, *y_os` – x/y coordinates in OS units.

Returns: possible error condition.

font_converttopoints

Converts OS units to 1/72000 inch.

Syntax: `os_error *font_converttopoints(int x_os, int y_os, int *x_inch, int *y_inch)`

Parameters: `int x_os, y_os` – x/y coordinates in OS units
`int *x_inch, *y_inch` – x/y coordinates in 1/72000 inch.

Returns: possible error condition.

font_setfont

Sets up the font used for subsequent painting or size-requests.

Syntax: `os_error * font_setfont(font)`

Parameters: `font` – the font handle

Returns: possible error condition.

font_current

Informs the caller of the current font state.

Syntax: `os_error *font_current(font_state *f)`

Parameters: `font_state *f` – pointer to buffer to hold font state

Returns: possible error condition.

Other Information: returned buffer(into variable of type `font_state`):

`font f`	handle of current font
`int back_colour`	current background colour
`int fore_colour`	current foreground colour
`int offset`	foreground colour offset.

font_future

Informs the caller of font characteristics after a future `font_paint`.

Syntax: `os_error *font_future(font_state *f)`

Parameters: `font_state *f` – pointer to buffer to hold font state.

Returns: possible error condition.

Other Information: buffer contents:

`font f` – handle of font which would be selected

`int back_colour` – future background colour

`int fore_colour` – future foreground colour

`int offset` – foreground colour offset.

font_findcaret

Informs the caller of the nearest point in a string to the caret position.

Syntax: `os_error *font_findcaret(font_string *fs)`

Parameters: `font_string *fs` – the string

fields: `char *s` – the string itself

`int x,y` – x/y coordinates of caret

Returns: possible error condition.

Other Information: returned fields of fs as in `font_strwidth`.

font_charbbox

Informs the caller of the bounding box of a character in a given font.

Syntax: `os_error * font_charbbox(font, char, int options, font_info*)`

Parameters: `font` – the font handle

`char` – the ASCII character

`int options` – only relevant option if `font_OSCOORDS`

`font_info*` – pointer to buffer to hold font information.

Returns: possible error condition.

Other Information: if OS coordinates are used and font has been scaled, box may be surrounded by area of blank pixels.

font_readscalefactor

Informs the caller of the x and y scale factors used by the font. manager for converting between OS coordinates and 1/72000 inch.

Syntax: `os_error *font_readscalefactor(int *x, int *y)`

Parameters: `int *x, *y` – returned scale factors.

Returns: possible error condition.

font_setscalefactor

Sets the scale factors used by the font manager.

Syntax: `os_error *font_setscalefactor(int x, int y)`

Parameters: `int x,y` – the new scale factors

Returns: possible error condition.

Other Information: scale factors may have been changed by another application; well-behaved applications save and restore scale factors.

font_list

Gives the name of an available font.

Syntax: `os_error * font_list(char*, int*)`

Parameters: `char*` – pointer to buffer to hold font name
`int*` – count of fonts found (0 on first call).

Returns: possible error condition.

Other Information: count is –1 if no more names. Typically used in loop until count == –1.

font_setcolour

Sets the current font (optionally), changes foreground and background colours, and offset for that font.

Syntax: `os_error * font_setcolour(font, int background, int foreground, int offset)`

Parameters: `font` – the font handle (0 for current font)
`int background, foreground` – back/foreground colours
`int offset` – foreground offset colour (–14 to +14).

Returns: possible error condition.

font_setpalette

Sets the anti-alias palette.

Syntax: `os_error *font_setpalette(int background, int foreground, int offset, int physical_back, int physical_fore)`

Parameters: `int background` – logical background colour
`int foreground` – logical foreground colour
`int offset` – foreground colour offset
`int physical_back` – physical background colour
`int physical_fore` – physical foreground colour

Returns: possible error condition.

Other Information: `physical_back` and `physical_fore` are of the form `0xBBGGRR00`.

font_readthresholds

Reads the list of threshold values that the font manager uses when painting characters.

Syntax: `os_error *font_readthresholds(font_threshold *th)`

Parameters: `font_threshold *th` – pointer to result buffer.

Returns: possible error condition.

font_setthresholds

Sets up threshold values for painting colours.

Syntax: `os_error *font_setthresholds(font_threshold *th)`

Parameters: `font_threshold *th` – pointer to a threshold table.

Returns: possible error condition.

font_findcaretj

Finds the nearest point where the caret can go (using justification offsets).

Syntax: `os_error *font_findcaretj(font_string *fs, int offset_x, int offset_y)`

Parameters: `font_string *fs` – the string (set up as in `font_findcaret`)

`int offset_x, offset-y` – the justification offsets.

Returns: possible error condition.

Other Information: If the offsets are both zero, the function is the same as `font_findcaret`.

font_stringbbox

Measures the size of a string (without printing it).

Syntax: `os_error *font_stringbbox(char *s, font_info *fi)`

Parameters: `char *s` – the string

`font_info *fi` – pointer to buffer to hold font information.

Returns: possible error condition.

Other Information: fields returned in `fi` are:

`minx, miny`	bounding box min x/y
`maxx, maxy`	bounding box min x/y.

heap

These functions provide `malloc`-style heap allocation in a flex block.

heap_init

Initialises the heap allocation system.

Syntax: `void heap_init(BOOL heap_shrink)`

Parameters: `BOOL heap_shrink` – if True, the flex block will be shrunk (when possible) after `heap_free()`.

Returns: `void`.

Other Information: You must call `flex_init` before calling this routine.

heap_alloc

Allocates a block of storage from the heap.

Syntax: `void *heap_alloc(unsigned int size)`

Parameters: `unsigned int size` – size of block to be allocated.

Returns: pointer to allocated block (or 0 if failed).

Other Information: This uses the `flex` module to allocate Wimp-supplied heap space. If the heap moves as the result of an extension or `flex` can't extend the heap, 0 is returned.

heap_free

Frees a previously allocated block of heap storage.

Syntax: `void heap_free(void *heapptr)`

Parameters: `void *heapptr` – pointer to block to be freed.

Returns: possible error condition.

magnify

This function allows the display and entry of magnification factors.

magnify_select

Displays a dialogue box to set magnification factors.

Syntax: `void magnify_select (int *mul, int *div, int maxmul, int maxdiv, void (*proc)(void *), void *phandle)`

Parameters: `int *mul, *div` – multiplication/division factors
`int maxmul, maxdiv` – maximum mult/div factors
`void(*proc)(void *)` – caller-supplied function
`void *phandle` – handle passed to user function.

Returns: `void`.

Other Information: Displays a template called 'magnifier' (which must be one of your loaded templates). `mul` and `div` are the initial values on the left and right of the `:` in the ratio shown in the dialogue box. They are modified according to user mouse clicks on the arrow icons. `proc` (if non-null) is called each time the magnification factor changes.

The template should have the following attributes:

- window flags – moveable, auto-redraw. It is advisable to have a title icon with the text `magnifier` or similar.
- icon #0 – the multiplication factor icon. This should have an indirected text flag set with text something like `999` and a maximum length of 4. It is also advisable to have a validation string `a0-9` (allowing numeric input). The button type should be 'writeable'.
- icon #1 – the division factor icon (same as icon #0)
- icon #2 – the increase multiplication factor icon should have its text flag set and contain the ⇑ character (like the arrow used in scroll bars). The button type should be 'auto-repeat'.
- icon #3 – the decrease multiplication factor icon (same as icon #2, but using the ⇓ char).
- icon #4 – the increase division factor icon (same as icon #2).
- icon #5 – the decrease division factor icon (same as icon #3).
- icon #6 – (optional but advisable) a text icon placed between icons #0 and #1 as a separator eg `:`

These icons can be arranged in the window however you wish, but a recommended layout is that of the Magnifier dialogue box in Draw or Paint.

menu

These functions deal with the creation, deletion and manipulation of menus.

A menu description string defines a sequence of entries, with the following syntax (curly brackets mean 0 or more, square brackets mean 0 or 1):

`opt ::= !` or `~` or `>` or `space`

`sep ::= ,` or `|`

`l1 ::=` any char but `opt` or `sep`

```
12 ::= any char but sep
name ::= 11 {12}
entry ::= {opt} name
descr ::= entry {sep entry}
```

Each entry defines a single entry in the menu. `|` as a separator means that there should be a gap or line between these menu components.

`opt` ! means 'put a tick by it'
`opt` ~ means 'make it non-selectable'
`opt` > means 'has a dialogue box as 'submenu''
`space` has no effect as an opt.

menu_new

Creates a new menu structure from the given textual description (arranged as above).

Syntax:	`menu menu_new(char *name, char *description)`
Parameters:	`char *name` – name to appear in title of menu `char *description` – textual description of menu
Returns:	pointer to menu structure created
Other Information:	Creates a menu structure, with entries as given in the

textual description. Entries are indexed from 1. For example:

`m=menu_new("Edit", ">Info Create Quit")`

Handler needs to be attached using `event_attachmenu`.

menu_dispose

Disposes of a menu structure.

Syntax:	`void menu_dispose(menu*, int recursive)`
Parameters:	`menu*` – the menu to be disposed of `int recursive` – non-zero ==recursively dispose of submenus.
Returns:	`void`.

menu_extend

Adds entries to the end of a menu.

Syntax:	`void menu_extend(menu, char *description)`
Parameters:	`menu` – the menu to which extension is being made `char *description` – textual description of extension.
Returns:	`void`.

Other Information: extension has the format:

`[sep] entry {sep entry}`

A menu which is already a submenu of another menu cannot be extended.

menu_setflags

Sets or changes flags on an already existing menu entry.

Syntax: `void menu_setflags(menu, int entry, int tick, int fade)`

Parameters: `menu` – the menu

`int entry` – index into menu entries (from 1)

`int tick` – non-zero == tick this entry

`int fade` – non-zero == fade this entry (ie make it unselectable).

Returns: `void`.

menu_submenu

Attaches a menu as a submenu of another at a given entry in the parent menu.

Syntax: `void menu_submenu(menu, int entry, menu submenu)`

Parameters: `menu` – the menu

`int entry` – entry at which to attach submenu

`menu submenu` – pointer to the submenu.

Returns: `void`.

Other Information: This replaces any previous submenu at this entry. Use 0 for submenu to remove an existing entry. Only a strict hierarchy is allowed. When attached as a submenu, a menu can't be extended or explicitly deleted.

menu_make_writeable

Makes a menu entry writeable.

Syntax: `void menu_make_writeable(menu m, int entry, char *buffer, int bufferlength, char *validstring)`

Parameters: `menu m` – the menu

`int entry` – the entry to make writeable

`char *buffer` – pointer to buffer to hold text of entry

`int bufferlength` – size of buffer

`char *validstring` – pointer to validation string

Returns: `void`.

Other Information: The lifetimes of `buffer` and `validstring` must be long enough.

menu_make_sprite

Makes a menu entry into a sprite.

Syntax: `void menu_make_sprite(menu m, int entry, char *spritename)`

Parameters: `menu m` – the menu
`int entry` – entry to be made into sprite
`char *spritename` – name of the sprite.

Returns: `void`.

Other Information: Entry which is initially a non-indirected text entry is changed to an indirected sprite, with sprite area given by `resspr_area()`, and name given by `spritename`.

menu_syshandle

Gives low-level handle to a menu.

Syntax: `void *menu_syshandle(menu)`

Parameters: `menu` – the menu

Returns: pointer to underlying Wimp menu structure.

Other Information: Allows the massaging of a menu by means other than those provided in this module. The returned pointer is in fact a pointer to a `wimp_menustr` (ie `wimp_menuhdr` followed by zero or more `wimp_menuitems`).

msgs

These functions provide support for the messages resource file. Use them to make your applications easily convertible to other natural languages. A messages file for RISC_OSlib error messages is provided; it is not needed if you just want English messages, since these are the defaults.

msgs_init

Reads in the messages file, and initialise message system.

Syntax: `void msgs_init(void)`

Parameters: `void`

Returns: `void`.

Other Information: The messages file is a resource of your application and should be named `messages`. Each line of this file is a message with the following format:

```
<tag><colon><message text><newline>
```

The tag is an alphanumeric identifier for the message, which will be used to search for the message, when using `msgs_lookup()`.

msgs_lookup

Finds the text message associated with a given tag.

Syntax: `char *msgs_lookup(char *tag_and_default)`

Parameters: `char *tag_and_default` – the tag of the message, and an optional default message (to be used if tagged message not found).

Returns: pointer to the message text (if all is well).

Other Information: If the caller just supplies a tag, he will receive a pointer to its associated message (if found). A default message can be given after the tag (separated by a colon). A typical use would be:

```
werr(1, msgs_lookup("error1"))
```

or

```
werr(1, msgs_lookup("error1:Not enough memory").
```

os

This file is provided as an alternative to `kernel.h`. It provides low-level access to RISC OS. `os_error` functions return a pointer to an error if one has occurred, otherwise return NULL (0).

os_swi

Performs the given SWI instruction, with the given registers loaded. An error results in a RISC OS error being raised. A NULL `regset` pointer means that no inout parameters are used.

Syntax: `void os_swi(int swicode, os_regset *regs)`

os_swix

Performs the given SWI instruction, with the given registers loaded. Calls returning `os_error*` use the X form of the relevant SWI. If an error is returned then the `os_error` should be copied before further system calls are made. If no error occurs then NULL is returned.

Syntax: `os_error *os_swix(int swicode, os_regset *regs)`

If `swicode` does not have the X bit set, `os_swi` is called and these functions return NULL (regardless of whether an error was raised). You should therefore use X bit set `swicodes` to save confusion.

SWIs with varying numbers of arguments and results:
NULL result pointers mean that the result from that register is not required. The swi codes can be of the X form if required, as specified by `swicode`.

```
os_error *os_swi0(int swicode); /* zero arguments and results */
os_error *os_swi1(int swicode, int r0)
os_error *os_swi2(int swicode, int r0, int r1)
os_error *os_swi3(int swicode, int r0, int r1, int r2)
os_error *os_swi4(int swicode, int r0, int r1, int r2, int r3)
os_error *os_swi6(int swicode, int r0, int r1, int r2, int r3, int r4, int r5)
os_error *os_swi1r(int swicode, int r0in, int *r0out)
os_error *os_swi2r(int swicode, int r0in, int r1in, int *r0out, int *r1out)
os_error *os_swi3r(int swicode, int, int, int, int*, int*, int*)
os_error *os_swi4r(int swicode, int, int, int, int, int*, int*, int*, int*)
os_error *os_swi6r(int swicode,
int r0, int r1, int r2, int r3, int r4, int r5,
int *r0out, int *r1out, int *r2out, int *r3out, int *r4out, int *r5out)
```

os_byte

Performs an `OS_Byte` SWIx, with x and y passed in register `r1` and `r2` respectively.

Syntax: `os_error *os_byte(int a, int *x /*inout*/, int *y /*inout*/)`

os_word

Performs an `OS_Word` SWIx, with operation number given in `wordcode` and `p` pointing at necessary parameters to be passed in `r1`.

Syntax: `os_error *os_word(int wordcode, void *p)`

os_gbpb

Performs an `OS_GBPB` SWI. `os_gbpbstr` should be used like an `os_regset`.

Syntax: `os_error *os_gbpb(os_gbpbstr*)`

os_file

Performs an `OS_FILE` SWI.

Syntax: `os_error *os_file(os_filestr*)`

os_args

Performs an `OS_Args` SWI.

Syntax: `os_error *os_args(os_regset*)`

os_find

Performs an `OS_Find` SWI.

Syntax: `os_error *os_find(os_regset*)`

os_cli

Performs an `OS_CLI` SWI.

Syntax: `os_error *os_cli(char *cmd)`

os_read_var_val

Reads a named environment variable into a given buffer (of size `bufsize`). If the variable doesn't exist, `buf` points at a null string.

```
os_read_var_val(char *name, char *buf /*out*/, int bufsize)
```

pointer

These functions deal with setting the pointer shape.

pointer_set_shape

Sets pointer shape 2, to sprite, from sprite area.

Syntax: `os_error *pointer_set_shape(sprite_area *, sprite_id *, int, int)`

Parameters: `sprite_area*` – area where sprite is to be found
`sprite_id*` – identity of the sprite
`int, int` – active point for pointer.

Returns: possible error condition.

Other Information: A typical use is to change pointer shape on entering or leaving application window (appropriate events are returned from `wimp_poll`).

pointer_reset_shape

Resets pointer shape to shape 1.

Syntax: `void pointer_reset_shape(void)`

Parameters: `void`.

Returns: `void`.

Other Information: Typically should be called when leaving an application window.

res

These functions provide access to resources.

res_init

Initialises, ready for calling other `res` functions.

Syntax: `void res_init(const char *progname)`

Parameters: `const char *a` – your program name.

Returns: `void`.

Other Information: Call this before using any `res` or `resspr` functions.

res_findname

Creates a full pathname for a `resname` file.

Syntax: `int res_findname(const char *resname, char *buf /*out*/)`

Parameters: `const char *resname` – name of one of your resource files
`char *buf` – buffer to put full pathname in.

Returns: True (always).

Other Information: the full pathname is constructed as: `<ProgramName$Dir>.resname` where `ProgramName` has been set using `res_init`.

res_openfile

Opens a named resource file, in a given ANSI-style mode.

Syntax: `FILE *res_openfile(const char *resname, const char *mode)`

Parameters: `const char *resname` – name of the resource file
`const char *mode` – usual ANSI open mode (`r`, `w`, etc)

Returns: ANSI FILE pointer for opened file.

Other Information: `resname` should be a 'leafname' (a call to `res_findname` is made for you).

resspr

These functions provide access to sprite resources.

resspr_init

Initialises, ready for calls to `resspr` functions.

Syntax: `void resspr_init(void)`

Parameters: `void`

Returns: `void`.

Other Information: call before using any `resspr` functions and before using `template_init()`, if your templates have sprites. This function reads in your sprites.

resspr_area

Returns a pointer to the sprite area being used.

Syntax: `sprite_area *resspr_area(void)`

Parameters: `void`

Returns: pointer to sprite area being used.

Other Information: Useful for passing parameters to functions like `baricon` which expect to be told sprite area to use.

saveas

These functions handle the export of data by dragging the icon from the dialogue box.

saveas

Displays a dialogue box to enable the user to export application data.

Syntax:

```
BOOL saveas(int filetype, char *name, int estsize,
xfersend_saveproc, xfersend_sendproc, xfersend_printproc,
void *handle)
```

Parameters:

`int filetype` – type of file to save to

`char *name` – suggested file name

`int estsize` – estimated size of the file

`xfersend_saveproc` – caller-supplied function for saving application data to a file

`xfersend_sendproc` – caller-supplied function for RAM data transfer (if application is able to do this)

`xfersend_printproc` – caller-supplied function for printing application data, if Save icon is dragged onto printer icon

`void *handle` – handle to be passed to handler functions.

Returns: True if data exported successfully.

Other Information: This function displays a dialogue box with the following fields:

- a sprite icon appropriate to the given file type
- the suggested filename
- an OK button.

A template called `xfer_send` must be in the application's templates file to use this function, set up as in the Edit, Draw and Paint applications). `xfer_send` deals with the complexities of message-passing protocols to achieve the data transfer. Refer to the typedefs in `xfersend.h` for an explanation of what the three caller-supplied functions should do. If you pass 0 as the `xfersend_sendproc`, no in-core data transfer will be attempted. If you pass 0 as the xfersend_printproc, the file format for printing is assumed to be the same as for saving. The estimated file size is not essential, but may improve performance.

saveas_read_leafname_during_send

Gets the 'leaf' of the filename in the filename field of the xfer-send dialogue box.

Syntax: `void saveas_read_leafname_during_send(char *name, int length)`

Parameters: `char *name` – buffer to put filename in
`int length` – size in bytes of supplied buffer.

Returns: void.

sprite

These functions provide access to RISC OS sprite facilities. Only a brief description is given for each call. More details can be found in the *RISC OS Programmer's Reference Manual*, in the chapter entitled *Sprites*.

sprite: simple operations

sprite_screensave

Saves the current graphics window as a sprite file, with optional palette (equivalent to *ScreenSave).

Syntax: `os_error *sprite_screensave(const char *filename, sprite_palflag)`

sprite_screenload

Load a sprite file onto the screen (equivalent to *ScreenLoad).

Syntax: `os_error *sprite_screenload(const char *filename)`

sprite: operations on system/user area

sprite_area_initialise

Initialises an area of memory as a sprite area.

Syntax: `void sprite_area_initialise(sprite_area *, int size)`

sprite_area_readinfo

Reads information from a sprite area control block.

Syntax: `os_error *sprite_area_readinfo(sprite_area *, sprite_area *resultarea)`

sprite_area_reinit

Reinitialises a sprite area. If the sprite area is a system area, the function is equivalent to *SNew.

Syntax:

```
os_error *sprite_area_reinit(sprite_area *)
```

sprite_area_load

Loads a sprite file into a sprite area. If the file is a system area, the function is equivalent to *SLoad.

Syntax:

```
os_error *sprite_area_load(sprite_area *, const char
*filename)
```

sprite_area_merge

Merges a sprite file with a sprite area. If the file is a system area, the function is equivalent to *SMerge.

Syntax:

```
os_error *sprite_area_merge(sprite_area *, const char
*filename)
```

sprite_area_save

Saves a sprite area as a sprite file. If the sprite area is a system area, the function is equivalent to *SSave.

Syntax:

```
os_error *sprite_area_save(sprite_area *, const char
*filename)
```

sprite_getname

Returns the name and length of the nth sprite in a sprite area into a buffer.

Syntax:

```
os_error *sprite_getname(sprite_area *, void *buffer, int
*length, int index)
```

sprite_get

Copies a rectangle of screen delimited by the last pair of graphics cursor positions as a named sprite in a sprite area, optionally storing the palette with the sprite.

Syntax:

```
os_error *sprite_get(sprite_area *, char *name,
sprite_palflag)
```

sprite_get_rp

Copies a rectangle of screen delimited by the last pair of graphics cursor positions as a named sprite in a sprite area, optionally storing the palette with the sprite. The address of the sprite is returned in `resultaddress`.

Syntax:

```
os_error *sprite_get_rp(sprite_area *, char *name,
sprite_palflag, sprite_ptr *resultaddress)
```

sprite_get_given

Copies a rectangle of screen delimited by the given pair of graphics coordinates as a named sprite in a sprite area, optionally storing the palette with the sprite.

Syntax:

```
os_error *sprite_get_given(sprite_area *, char *name,
sprite_palflag, int x0, int y0, int x1, int y1)
```

sprite_get_given_rp

Copies a rectangle of screen delimited by the given pair of graphics coordinates as a named sprite in a sprite area, optionally storing the palette with the sprite. The ddress of the sprite is returned in `resultaddress`.

Syntax:

```
os_error *sprite_get_given_rp(sprite_area *, char *name,
sprite_palflag, int x0, int y0, int x1, int y1,
sprite_ptr *resultaddress)
```

sprite_create

Creates a named sprite in a sprite area of specified size and screen mode, optionally reserving space for palette data with the sprite.

Syntax:

```
os_error *sprite_create(sprite_area *, char *name,
sprite_palflag, int width, int height, int mode)
```

sprite_create_rp

Creates a named sprite in a sprite area of specified size and screen mode, optionally reserving space for palette data with the sprite. The address of the sprite is returned in `resultaddress`.

Syntax:

```
os_error *sprite_create_rp(sprite_area *, char *name,
sprite_palflag, int width, int height, int mode,
sprite_ptr *resultaddress)
```

sprite: operations on system/user area, name/sprite pointer

sprite_select

Selects the specified sprite for plotting using `plot(0xed,x,y)`.

Syntax:

```
os_error *sprite_select(sprite_area *, sprite_id *)
```

sprite_select_rp

Selects the specified sprite for plotting using `plot(0xed,x,y)`. The address of the sprite is returned in `resultaddress`.

Syntax:

```
os_error *sprite_select_rp(sprite_area *, sprite_id *,
sprite_ptr *resultaddress)
```

sprite_delete

Deletes the specified sprite.

Syntax:

```
os_error *sprite_delete(sprite_area *, sprite_id *)
```

sprite_rename

Renames the specified sprite within the same sprite area.

Syntax:
```
os_error *sprite_rename(sprite_area *, sprite_id *, char
*newname)
```

sprite_copy

Copies the specified sprite as another named sprite in the same sprite area.

Syntax:
```
os_error *sprite_copy(sprite_area *, sprite_id *, char
*copyname)
```

sprite_put

Plots the specified sprite using the given GCOL action.

Syntax:
```
os_error *sprite_put(sprite_area *, sprite_id *, int gcol)
```

sprite_put_given

Plots the specified sprite at (x,y) using the given GCOL action.

Syntax:
```
os_error *sprite_put_given(sprite_area *, sprite_id *,
int gcol, int x, int y)
```

sprite_put_scaled

Plots the specified sprite at (x,y) using the given GCOL action, and scaled using the given scale factors.

Syntax:
```
os_error *sprite_put_scaled(sprite_area *, sprite_id *,
int gcol, int x, int y, sprite_factors *factors,
sprite_pixtrans pixtrans[])
```

sprite_put_greyscaled

Plots the specified sprite at (x,y) using the given GCOL action, greyscaled using the given scale factors.

Syntax:
```
os_error *sprite_put_greyscaled(sprite_area *,
sprite_id *, int x, int y, sprite_factors *factors,
sprite_pixtrans pixtrans[])
```

sprite_put_mask

Plots the specified sprite mask in the background colour.

Syntax:
```
os_error *sprite_put_mask(sprite_area *, sprite_id *)
```

sprite_put_mask_given

Plots the specified sprite mask at (x,y) in the background colour.

Syntax:
```
os_error *sprite_put_mask_given(sprite_area *,
sprite_id *, int x, int y)
```

sprite_put_mask_scaled

Plots the sprite mask at (x,y) scaled, using the background colour/action.

Syntax:
```
os_error *sprite_put_mask_scaled(sprite_area *,
sprite_id *, int x, int y, sprite_factors *factors)
```

sprite_put_char_scaled

Paints char scaled at (x,y).

Syntax: `os_error *sprite_put_char_scaled(char ch, int x, int y, sprite_factors *factors)`

sprite_create_mask

Creates a mask definition for the specified sprite.

Syntax: `os_error *sprite_create_mask(sprite_area *, sprite_id *)`

sprite_remove_mask

Removes the mask definition from the specified sprite.

Syntax: `os_error *sprite_remove_mask(sprite_area *, sprite_id *)`

sprite_insert_row

Inserts a row into the specified sprite at the given row.

Syntax: `os_error *sprite_insert_row(sprite_area *, sprite_id *, int row)`

sprite_delete_row

Deletes the given row from the specified sprite.

Syntax: `os_error *sprite_delete_row(sprite_area *, sprite_id *, int row)`

sprite_insert_column

Inserts a column into the specified sprite at the given column.

Syntax: `os_error *sprite_insert_column(sprite_area *, sprite_id *, int column)`

sprite_delete_column

Deletes the given column from the specified sprite.

Syntax: `os_error *sprite_delete_column(sprite_area *, sprite_id *, int column)`

sprite_flip_x

Flips the specified sprite about the x axis.

Syntax: `os_error *sprite_flip_x(sprite_area *, sprite_id *)`

sprite_flip_y

Flips the specified sprite about the y axis.

Syntax: `os_error *sprite_flip_y(sprite_area *, sprite_id *)`

sprite_readsize

Reads the size information for the specified `sprite_id`.

Syntax: `os_error *sprite_readsize(sprite_area *, sprite_id *, sprite_info *resultinfo)`

sprite_readpixel

Reads the colour of a given pixel in the specified `sprite_id`.

Syntax:
```
os_error *sprite_readpixel(sprite_area *, sprite_id *,
int x, int y, sprite_colour *resultcolour)
```

sprite_writepixel

Writes the colour of a given pixel in the specified `sprite_id`.

Syntax:
```
os_error *sprite_writepixel(sprite_area *, sprite_id *,
int x, int y, sprite_colour *colour)
```

sprite_readmask

Reads the state of a given pixel in the specified sprite mask.

Syntax:
```
os_error *sprite_readmask(sprite_area *, sprite_id *,
int x, int y, sprite_maskstate *resultmaskstate)
```

sprite_writemask

Writes the state of a given pixel in the specified sprite mask.

Syntax:
```
os_error *sprite_writemask(sprite_area *, sprite_id *,
int x, int y, sprite_maskstate *maskstate)
```

sprite_restorestate

Restores the old state after one of the sprite redirection calls.

Syntax:
```
os_error *sprite_restorestate(sprite_state state)
```

sprite_outputtosprite

Redirects VDU output to a sprite, saving the old state.

Syntax:
```
os_error *sprite_outputtosprite(sprite_area *area,
sprite_id *id, int *save_area, sprite_state *state)
```

sprite_outputtomask

Redirects output to a sprite's transparency mask, saving the old state.

Syntax:
```
os_error *sprite_outputtomask(sprite_area *area,
sprite_id *id, int *save_area, sprite_state *state)
```

sprite_outputtoscreen

Redirects output back to screen, saving the old state.

Syntax:
```
os_error *sprite_outputtoscreen(int *save_area,
sprite_state *state)
```

sprite_sizeof_spritecontext

Gets the size of the save area needed to save the sprite context.

Syntax:
```
os_error *sprite_sizeof_spritecontext(sprite_area *area,
sprite_id *id, int *size)
```

sprite_sizeof_screencontext

Gets the size of the save area needed to save the screen context.

Syntax:
```
os_error *sprite_sizeof_screencontext(int *size)
```

sprite_removewastage

Removes the lefthand wastage from a sprite.

Syntax: `os_error *sprite_removewastage(sprite_area *area, sprite_id *id)`

template

This file contains functions used for loading and manipulating templates (typically set up using the template editor, FormEd). The templates are assumed to be held in a file `Templates` in the application's directory. The dialogue box module of the RISC OS library uses these templates when creating dialogue boxes.

template_copy

Creates a copy of a template.

Syntax: `template *template_copy (template *from)`

Parameters: `template *from` – the original template

Returns: a pointer to a copy of `from`.

Other Information: Copying includes fixing up pointers into workspace for indirected icons/title, and the allocation of this space.

template_readfile

Reads the template file into a linked list of templates.

Syntax: `BOOL template_readfile (char *name)`

Parameters: `char *name` – name of template file

Returns: Non-zero if sprites are used in the template file.

Other Information: Note that a call is made to `resspr_area()`, in order to fix up a window's sprite pointers, so you must have already called `resspr_init`.

template_find

Finds a named template in the template list.

Syntax: `template *template_find(char *name)`

Parameters: `char *name` – the name of the template (as given in FormEd)

Returns: a pointer to the found template.

template_loaded

Sees if there is anything in the template list.

Syntax: `BOOL template_loaded(void)`

Parameters: `void`

Returns: Non-zero if there is something in the template list.

template_init

Initialises ready for the use of templates.

Syntax: `void template_init(void)`

Parameters: `void`

Returns: `void`.

Other Information: Should be called before any operations which use templates (such as dialogue box creation).

template_syshandle

Gets a pointer to the underlying window used to create a template.

Syntax: `wimp_wind *template_syshandle(char *name)`

Parameters: `char *templatename`.

Returns: Pointer to template's underlying window (0 if template not found).

Other Information: Any changes made to the `wimp_wind` structure will affect future windows generated using this template.

trace

These functions provide centralised control for trace/debug output.

tracef

Outputs tracing information.

Syntax:

```
void tracef(char*,...)
void tracef0(char*)
void tracef1(char*, int)
void tracef2(char*, int,int)
void tracef3(char*, int,int,int)
void tracef4(char*, int,int,int,int)
```

Parameters: `char*` – `printf`-style format string
... – variable argument list.

Returns: `void`.

Other Information: called by `tracef0`, `tracef1` etc. Fixed-format ones will compile to nothing if `trace` is not set at compile time.

trace_is_on

`int trace_is_on(void)` returns True if tracing is turned on

trace_on

`void trace_on(void)` turns tracing on

trace_off

`void trace_off(void)` turns tracing off

txt

A `txt` is an array of characters, displayed in a window on the screen. It behaves in many ways similarly to a single buffer from Edit (see the *User Guide* for details of this application). It uses the system variable Edit$Options to set up colours, fonts and other features. You must call `flex_init` before calling txt.

txt: interface functions

txt_new

Creates a new txt object, containing no characters with a given title (to appear in its window).

Syntax: `txt txt_new(char *title)`

Parameters: `char *title` – the text title to appear in its window.

Returns: pointer to the newly created text.

Other Information: This function does not result in the text being displayed on the screen; it simply creates a new text object. 0 is returned if there is not enough space to create the object.

txt_show

Displays a given text object in a free-standing window of its own.

Syntax: `void txt_show(txt t)`

Parameters: `txt t` – the text to be displayed.

Returns: `void`.

Other Information: `t` should have been created using `txt_new`.

txt_hide

Hides a text which has been displayed.

Syntax: `void txt_hide(txt t)`

Parameters: `txt t` – the text to be hidden.

Returns: `void`.

txt_settitle

Changes the title of the window used to display a text object.

Syntax: `void txt_settitle(txt t, char *title)`

Parameters: `txt t` – the text object
`char *title` – new title of window.

Returns: `void`.

Other Information: Long titles may be truncated when displayed.

txt_dispose

Destroys a text and the window associated with it.

Syntax: `void txt_dispose(txt *t)`

Parameters: `txt *t` – pointer to the text.

Returns: `void`.

txt: general control operations

A text object's main data content is an array of characters. This resides in a buffer of known size. The characters of the array are not laid out precisely in the buffer; a gap is used in order to make insertion and deletion fast. When initially created, a text has `bufsize=0`.

txt_bufsize

Tells caller how many characters can be stored in the buffer before more memory needs to be requested from the operating system.

Syntax: `*int txt_bufsize(txt)`

Parameters: `txt t` – the text.

Returns: size of buffer.

txt_setbufsize

Allocates more space for the text buffer.

Syntax: `BOOL txt_setbufsize(txt, int)`

Parameters: `txt t` – the text
`int b` – new buffer size.

Returns: True if space could be allocated successfully.

Other Information: This call increases the buffer size, so that at least `b` characters can be stored before requiring more from the operating system.

The character array is displayed on the screen in a window. The characters travel horizontally from left to right. If a `\n` is encountered, this signifies the end of the current text line, and the start of a new one. All lines have the same height, although characters may be of differing widths. There is no limit on the number of characters allowed in a line. There is no restriction on the characters allowed in the array: any number from 0 to 255 is acceptable.

txt_charoptions

Informs the caller of the currently set charoptions.

Syntax: `txt_charoption txt_charoptions(txt)`

Parameters: `txt t` – text object.

Returns: Currently set charoptions.

Clearing the DISPLAY flag can be used during a long and complex sequence of edits, to reduce the overall amount of display activity. The UPDATED flag is set by the insertion or deletion of any characters in the array.

txt_setcharoptions

Sets the flags which are used to control the display of text in a screen window.

Syntax: `void txt_setcharoptions(txt, txt_charoption affect, txt_charoption values)`

Parameters: `txt t` – text object
`txt_charoption affect` – flags to affect
`txt_charoption values` – values to give to affected flags.

Returns: `void`.

Other Information: Only the flags named in `affect` are affected – they are set to the value `values`. This therefore has the meaning:

```
(previousState & ~affect) | (affect & values)
```

txt_setdisplayok

Sets the display flag in charoptions for a given text.

Syntax: `void txt_setdisplayok(txt)`

Parameters: `txt t` – text object

Returns: `void`.

Other Information: This asserts to the system that the display is up to date, preventing a redraw. It is useful only in very specialised circumstances.

txt: operations on the array of characters

`dot` is an index into the character array. If there are `n` characters in the array, with indices in `0...n-1`, then `dot` is in `0...n`. It is thought of as pointing just before the character with the same index, but it can also point just after the last one. When the text is displayed, the character after the `dot` is always visible. The caret is a visible indication of the position of the `dot` within the array. It can be made visible using `SetCharOptions` above.

txt_dot

Informs the caller of where the `dot` (current position) is in the array of characters.

Syntax: `txt_index txt_dot(txt t)`

Parameters: `txt t` – text object.

Returns: An index into the array of characters.

txt_size

Informs the caller as to the maximum value `dot` can take.

Syntax:	`txt_index txt_size(txt t)`
Parameters:	`txt t` – text object.
Returns:	Maximum permissible value of `dot`.

txt_setdot

Sets the `dot` at a given index in the array of characters.

Syntax:	`void txt_setdot(txt t, txt_index i)`
Parameters:	`txt t` – text object. `txt_index i` – index at which to set `dot`.
Returns:	`void`.
Other Information:	If `i` is outside the bounds of the array it is set to the beginning or end of the array, as appropriate.

txt_movedot

Moves the `dot` by a given distance in the array.

Syntax:	`void txt_movedot(txt, int by)`
Parameters:	`txt t` – text object `int by` – distance to move by
Returns:	`void`
Other Information:	If the resulting `dot` is outside the bounds of the array it is set to the beginning or end of the array, as appropriate.

txt_insertchar

Inserts a character into the text just after the `dot`.

Syntax:	`void txt_insertchar(txt t, char c)`
Parameters:	`txt t` – text object `char c` – the character to be inserted.
Returns:	`void`.
Other Information:	If the `DISPLAY` option flag is set, the window is redisplayed after insertion.

txt_insertstring

Inserts a given character string into a text.

Syntax:	`void txt_insertstring(txt t, char *s)`
Parameters:	`txt t` – text object `char *s` – the character string.
Returns:	`void`.

Other Information: If the DISPLAY option flag is set, the window is redisplayed after insertion.

txt_delete

Deletes n characters from the dot onwards.

Syntax: `void txt_delete(txt t, int n)`

Parameters: txt t – text object
int n – number of characters to delete.

Returns: void.

Other Information: If dot+n is beyond the end of the array, deletion is to the end of the array.

txt_replacechars

Deletes ntodelete characters from dot, and inserts n characters in their place, where the characters are pointed at by a.

Syntax: `void txt_replacechars(txt t, int ntodelete, char *a, int n)`

Parameters: txt t – text object
int ntodelete – number of characters to delete
char *a – pointer to characters to insert
int n – number of characters to insert.

Returns: void.

txt_charatdot

Informs the caller of the character held at dot in the array.

Syntax: `char txt_charatdot(txt t)`

Parameters: txt t – text object.

Returns: Character at dot.

Other Information: Returns 0 if dot is at or beyond end of array.

txt_charat

Informs the caller of the character at a given index in the array.

Syntax: `char txt_charat(txt t, txt_index i)`

Parameters: txt t – text object
txt_index i – the index into the array.

Returns: Character at given index in array.

Other Information: Returns 0 if index is at or beyond end of array.

txt_charsatdot

Copies at most n characters from `dot` in the array into a supplied buffer.

Syntax: `void txt_charsatdot(txt, char/*out*/ *buffer, int /*inout*/ *n)`

Parameters: `txt t` – text object

`char *buffer` – the buffer

`int *n` – maximum characters to copy.

Returns: `void`.

Other Information: If you are close to the end of the array, `n` characters may not be available. In this case, characters up to the end of the array are copied, and `*n` is updated to report how many were copied.

txt_replaceatend

Deletes a specified number of characters from the end of the array and then inserts specified characters.

Syntax: `void txt_replaceatend(txt, int ntodelete, char*, int)`

Parameters: `txt t` – text object

`int ntodelete` – number of characters to delete

`char *s` – pointer to characters to insert

`int n` – number of characters to insert.

Returns: `void`.

txt: layout-dependent operations

These operations are provided specifically for the support of cursor-key-driven editing.

txt_movevertical

Moves the `dot` by a specified number of textual lines, with the caret staying in the same horizontal position on the screen.

Syntax: `void txt_movevertical(txt t, int by, int caretstill)`

Parameters: `txt t` – text object

`int by` – number of lines to move by

`int caretstill` – set to non-zero, if you want the text to move rather than the caret.

Returns: `void`.

txt_movehorizontal

Moves the caret (and `dot`) horizontally.

Syntax: `void txt_movehorizontal(txt, int by)`

Parameters: `txt t` – text object
`int by` – distance to move by.

Returns: `void`.

Other Information: This behaves like `txt_movedot()`, except that if `by` is positive and the end of the current text line is encountered, the caret will continue to move to the right on the screen.

txt_visiblelinecount

Gives the number of lines visible or partially visible on the display.

Syntax: `int txt_visiblelinecount(txt t)`

Parameters: `txt t` – text object.

Returns: Number of visible lines

Other Information: Takes into account current window size, font etc.

txt_visiblecolcount

Gives the number of columns currently visible.

Syntax: `int txt_visiblecolcount(txt t)`

Parameters: `txt t` – text object.

Returns: Visible column count.

Other Information: If a fixed pitch font is currently in use, this gives the number of display columns; otherwise, it makes a guess for average characters.

txt: operations on markers

Markers are indices into the array. Once set, a marker will point to the same character in the array regardless of insertions or deletions within the array. If the character pointed at by the marker is deleted, the marker will point to the next character. Markers never fall off the end of the array, but stay at the top or bottom of it, if that's where they end up.

txt_newmarker

Creates a new marker in the text.

Syntax: `void txt_newmarker(txt, txt_marker *mark)`

Parameters: `txt t` – text object
`txt_marker *mark` – pointer to your text marker.

Returns: `void`.

Other Information: The marker itself is kept by the client of this function, but the text object retains a pointer to it. The client's marker is updated by the text object whenever necessary. Its initial value is the same as `dot`. If the character at which a marker points is deleted, then the marker gets moved to the value of `dot` when the deletion occurred. If characters are inserted when the marker is at `dot`, the marker stays with `dot`.

txt_movemarker

Resets an existing marker.

Syntax: `void txt_movemarker(txt t, txt_marker *mark, txt_index to)`

Parameters: `txt t` – text object

`txt_marker *mark` – the marker

`txt_index to` – place to move the marker to.

Returns: `void`.

Other Information: The marker must already point into this text object.

txt_movedottomarker

Moves the `dot` to a given marker.

Syntax: `void txt_movedottomarker(txt t, txt_marker *mark)`

Parameters: `txt t` – text object

`txt_marker *mark` – pointer to the marker.

Returns: `void`.

txt_indexofmarker

Gives the current index into the array of a given marker.

Syntax: `txt_index txt_indexofmarker(txt t, txt_marker *mark)`

Parameters: `txt t` – text object

`txt_marker *mark` – pointer to the marker.

Returns: Index of marker.

txt_disposemarker

Delete a marker from a text object.

Syntax: `void txt_disposemarker(txt, txt_marker*)`

Parameters: `txt t` – text object

`txt_marker *mark` – the marker to be deleted.

Returns: `void`.

Other Information: You should remember to dispose of a marker which logically ceases to exist, otherwise the text object will continue to update the location where it was.

txt: operations on a selection

The selection is a contiguous portion of the array which is displayed highlighted.

txt_selectset

Informs the caller whether there is a selection made in a text.

Syntax:	`BOOL txt_selectset(txt t)`
Parameters:	`txt t` – text object.
Returns:	`True` if there is a selection in this text.

txt_selectstart

Gives the index into the array of the start of the current selection.

Syntax:	`txt_index txt_selectstart(txt t)`
Parameters:	`txt t` – text object.
Returns:	Index of selection start.

txt_selectend

Gives the index into the array of the end of the current selection.

Syntax:	`txt_index txt_selectend(txt t)`
Parameters:	`txt t` – text object.
Returns:	Index of selection end.

txt_setselect

Sets a selection in a given text, from start to end.

Syntax:	`void txt_setselect(txt, txt_index start, txt_index end)`
Parameters:	`txt t` – text object `txt_index start` – array index of start of selection `txt_index end` – array index of end of selection.
Returns:	`void`.
Other Information:	If `start >= end` then the selection will be unset.

txt: input from the user

Characters entered into the keyboard, and various mouse events, are buffered up by the text object for use by the client.

A call to the event handler registered with a text object will give an event code to the event handler, to say what sort of event has occurred. The following event codes are defined; any that are not understood should be ignored.

- Codes 0 – 255: key codes from the keyboard

- Codes 256 – 511: various function keys, etc; refer to `h.akbd` for the rules.
- Mouse events:

  A mouse event occurs when the mouse is pointing in the text object and a button is pressed or released, or the mouse moves while any button is depressed. A mouse event will result in Get producing an EventCode with bit 31 set, bits 24..28 as a `mouseeventflags` value, and the rest of the word containing an index value.

  The index shows where in the visible representation of the array the mouse event happened. If all three index bytes are 255, the event happened outside the window. The `mouseeventflags` show what button transitions occurred:

MSELECT	Select's new value
MEXTEND	Adjust's new value
MSELOLD	Select's old value
MEXTOLD	Adjust's old value
MEXACT	the event is in exactly the same place as the last one.

  The byte gives the values of the select and extend buttons: 1 for depressed and 0 for not depressed. It gives their previous values, allowing transitions to be detected. It reports whether the position of the mouse is exactly the same as for the last event, so that multiple clicks may be detected. No assumptions should be made concerning the relationship of these bits to the last mouse event sent to the programmer, as polling delays etc. could cause any combinations to happen.

  If `txt_EXTRACODE` is set, the identity of the event is not defined by this interface. This is used for any expansion. Clients of this interface which receive such events that they do not recognise, should ignore them without reporting an error.

  The Menu button on the mouse is not transmitted through this interface, but caught elsewhere. Use `event_attach_menu` to attach a menu handler to the `txt_syshandle` of a txt object.

- Keyboard events:

  `txt_EXTRACODE + akbd_Fn + 1`: – help request

  `txt_EXTRACODE + akbd_Fn + akbd_Sh + 2`: insert drag file

  `txt_EXTRACODE + akbd_Fn + 127`: – close icon

  `txt_EXTRACODE + akbd_Sh + akbd_Ctl + akbd_Upk`: scroll up one line

  `txt_EXTRACODE + akbd_Sh + akbd_Ctl + akbd_DownK`: scroll down one line

  `txt_EXTRACODE + akbd_Sh + akbd_UpK`: scroll up one page

  `txt_EXTRACODE + akbd_Sh + akbd_DownK`: scroll down one page

  In the current implementation of txt, `txt_queue` never returns more than 1, so `wimpt_last_event()` can be accessed to get more information.

txt_get

Gives the next user event code to the caller.

Syntax: `txt_eventcode txt_get(txt t)`

Parameters: `txt t` – text object

Returns: The event code

Other Information: The returned code can be ASCII, or various other (system-specific) values for function keys etc. This function can only be called within an event handler.

txt_queue

Informs the caller of how many event codes are currently buffered for a given text.

Syntax: `int txt_queue(txt t)`

Parameters: `txt t` – text object

Returns: Number of buffered event codes.

Other information: This function can only be called within an event handler.

txt_unget

Puts an event code back on the front of the event queue for a given text.

Syntax: `void txt_unget(txt t, txt_eventcode code)`

Parameters: `txt t` – text object

`txt_eventcode code` – the event code.

Returns: `void`.

Other information: This function can only be called within an event handler.

txt_eventhandler

Registers an `eventhandler` function for a given text, which will be called whenever there is a value ready which can be picked up by `txt_get()`.

Syntax: `void txt_eventhandler(txt, txt_event_proc, void *handle)`

Parameters: `txt t` – text object

`txt_event_proc func` – event handler function

`void *handle` – caller-defined handle to be passed to func.

Returns: `void`.

Other Information: If `func==0`, no function is registered.

txt_readeventhandler

Informs the caller of the currently registered `eventhandler` function associated with a given text, and the handle which is passed to it.

Syntax: `void txt_readeventhandler(txt t, txt_event_proc *func, void **handle)`

Parameters: `txt t` – text object

`txt_event_proc *func` – returned pointer to handler func

`void **handle` – returned pointer to handle.

Returns: `void`.

txt: direct access to the array of characters

txt_arrayseg

Gives a direct pointer into the memory used to hold the characters in a text.

Syntax: `void txt_arrayseg(txt t, txt_index at, char **a, int *n)`

Parameters: `txt t` – text object

`txt_index at` – index into the text

`char **a` – `*a` will point at the character whose index in the text is `at`

`int *n` – number of contiguous bytes after `at`.

Returns: `void`.

Other Information: It is permissible for the caller of this function to change the characters pointed at by `*a`, provided that a redisplay is prompted (using `setcharoptions`).

txt: system hook

txt_syshandle

Obtains a `wimp_w` value for the window underlying a text.

Syntax: `int txt_syshandle(txt t)`

Parameters: `txt t` – text object.

Returns: System-dependent handle for the given text.

txtedit

These functions provide text editing facilities.

txtedit_install

Installs an event handler for the `txt t`, thus making it an editable text.

Syntax: `txtedit_state *txtedit_install(txt t)`

Parameters: `txt t` – the text object (created via `txt_new`)

Returns: A pointer to the resulting `txtedit_state`.

txtedit_new

Creates a new text object and loads the given file into it. The text can then be edited.

Syntax: `txtedit_state *txtedit_new(char *filename)`

Parameters: `char *filename` – the file to be loaded.

Returns: a pointer to the `txtedit_state` for this text.

Other Information: If the file cannot be found, then 0 is returned as a result, and no text is created. If `filename` is a null pointer, then an editor window with no given file name will be constructed. If the file is already being edited, then a pointer to the existing `txtedit_state` is returned.

txtedit_dispose

Destroys the given text being edited.

Syntax: `void txtedit_dispose(txtedit_state *s)`

Parameters: `txtedit_state *s` – the text to be destroyed.

Returns: `void`.

Other Information: This will ask no questions of the user before destroying the text.

txtedit_mayquit

Check if we may safely quit editing.

Syntax: `BOOL txtedit_mayquit(void)`

Parameters: `void`.

Returns: True if we may safely quit, otherwise False.

Other Information: If a text is being edited, then a dialogue box is displayed asking the user if he really wants to quit. This calls `dboxquery()`, and therefore requires the template `query` as described in `dboxquery.h`.

txtedit_prequit

Deals with a `PREQUIT` message from the Task Manager.

Syntax: `void txtedit_prequit(void)`

Parameters: `void`.

Returns: `void`.

Other Information: Calls `txtedit_mayquit()`, to see if we may quit, if text is being edited. If user replies that we may quit, then all texts are disposed of, and this function sends an acknowledgement to the Task Manager.

txtedit_menu

Sets up a menu structure for the text being edited, tailored to its current state.

Syntax: `menu txtedit_menu(txtedit_state *s)`

Parameters: `txtedit_state *s` – the text's current state.

Returns: a pointer to an appropriately formed menu structure.

Other Information: The menu created will have the same form as that displayed when Menu is clicked on an Edit window. (For Edit version 1.00). Entries in the menu are set according to the supplied `txtedit_state`.

txtedit_menuevent

Applies a given menu hit to a given text.

Syntax: `void txtedit_menuevent(txtedit_state *s, char *hit)`

Parameters: `txtedit_state *s` – the text to which hit should be applied
`char *hit` – a menu hit string.

Returns: `void`.

Other Information: This can be called from a menu event handler.

txtedit_doimport

Import data into the specified `txtedit` object, from a file of a given type.

Syntax: `BOOL txtedit_doimport(txtedit_state *s, int filetype, int estsize)`

Parameters: `txtedit_state *s` – the text object
`int filetype` – type of the file
`int estsize` – the file's estimated size.

Returns: `True` if the import is completed successfully.

txtedit_doinsertfile

Inserts a named file in a given text object.

Syntax: `void txtedit_doinsertfile(txtedit_state *s, char *filename, BOOL replaceifwasnull)`

Parameters: `txtedit_state *s` – the text object

`char *filename` – the given file

`BOOL replaceifwasnull` – if set to `True` then the text object will be considered to have come from `filename`, ie the window title is updated.

Returns: `void`.

txtwin

These functions give control of multiple windows on text objects. When the Text is updated, all the windows are updated in step. All the windows have the same title information.

txtwin_new

Creates an extra window on a given text object.

Syntax: `void txtwin_new(txt t)`

Parameters: `txt t` – the text to have a window added to it.

Returns: `void`

Other Information: The created window will be in the same style as for `txt_new()`, with the same title information. The window will be made visible.

txtwin_number

Informs the caller of the number of windows currently on a given text.

Syntax: `int txtwin_number(txt t)`

Parameters: `txt t` – the text.

Returns: The number of windows currently on `t`.

txtwin_dispose

Removes a window, previously on `t`.

Syntax: `void txtwin_dispose(txt t)`

Parameters: `txt t` – the text

Returns: `void`

Other Information: This call will have no effect if there is only one window on `t`.

txtwin_setcurrentwindow

Ensures that the last window to which the last event was delivered is the current window on a given text.

Syntax: `void txtwin_setcurrentwindow(txt t)`

Parameters: `txt t` – the text.

Returns: `void`.

Other Information: Call this when constructing menus, since the same menu structure is attached to each window on the same text object.

visdelay

These functions enable a visual indication of some delay.

visdelay_begin

Changes pointer to show user there will be some delay (currently the RISC OS hourglass).

Syntax: `void visdelay_begin(void)`

Parameters: `void`.

Returns: `void`.

Other Information: Under RISC OS, the hourglass will only appear if the delay is longer than 1/3 sec.

visdelay_percent

Indicates to the user that a delay is `p` percent complete.

Syntax: `void visdelay_percent(int p)`

Parameters: `int p` – percentage complete.

Returns: `void`.

visdelay_end

Removes the indication of delay.

Syntax: `void visdelay_end(void)`

Parameters: `void`.

Returns: `void`.

visdelay_init

Initialises ready for `visdelay` functions.

Syntax: `void visdelay_init(void)`

Parameters: `void`.

Returns: void.

werr

This function provides error reporting in Wimp programs, causing a (possibly fatal) error message to appear in a pop-up dialogue box.

Syntax: `void werr(int fatal, char* format, ...)`

Parameters: `int fatal` – non-zero indicates fatal error
`char *format` – printf-style format string
... – variable arg list of message to be printed.

Returns: void.

Other Information: The program exits if `fatal` is non-zero. The pointer is restricted to the displayed dialogue box to stop the user continuing until he has clicked on the `OK` button. The message should be divided into at most three lines, each of 40 characters or less.

wimp

This file provides a C interface to RISC OS Wimp SWIs, and the following useful type definitions.

wimp_flags

`typedef enum{`

`wimp_WMOVEABLE = 0x00000002,`	is moveable
`wimp_REDRAW_OK = 0x00000010,`	can be redrawn entirely by Wimp ie no user graphics
`wimp_WPANE = 0x00000020,`	window is stuck over tool window
`wimp_WTRESPASS = 0x00000040,`	window is allowed to go outside main area
`wimp_WSCROLL_R1= 0x00000100,`	scroll request returned when scroll button clicked – auto-repeat
`wimp_SCROLL_R2 = 0x00000200,`	as SCROLL_R1, debounced, no auto
`wimp_REAL_COLOURS = 0x000000400,`	use real window colours.
`wimp_BACK_WINDOW = 0x000000800,`	this window is a background window.
`wimp_HOT_KEYS = 0x000001000,`	generate events for 'hot keys'
`wimp_WOPEN = 0x00010000,`	window is open

`wimp_WTOP = 0x00020000,`	window is on top (not covered)
`wimp_WFULL = 0x00040000,`	window is full size
`wimp_WCLICK_TOGGLE = 0x00080000,`	`open_window_request` was due to click on Toggle size icon
`wimp_WFOCUS = 0x00100000,`	window has input focus
`wimp_WBACK = 0x01000000,`	window has Back icon
`wimp_WQUIT = 0x02000000,`	has a Close icon
`wimp_WTITLE = 0x04000000,`	has a title bar
`wimp_WTOGGLE= 0x08000000,`	has a Toggle size icon
`wimp_WVSCR = 0x10000000,`	has vertical scroll bar
`wimp_WSIZE = 0x20000000,`	has Adjust size icon
`wimp_WHSCR = 0x40000000,`	has horizontal scroll bar
`wimp_WNEW = 0x80000000`	use these new flags

```
}wimp_flags;
```

Note: Always set the WNEW flag.

wimp_wcolours

If the work area background is 255, it isn't painted. If the title foreground is 255, you get no borders, title etc. at all.

```
typedef enum{
wimp_WCTITLEFORE,
wimp_WCTITLEBACK,
wimp_WCWKAREAFORE,
wimp_WCWKAREABACK,
wimp_WCSCROLLOUTER,
wimp_WCSCROLLINNER,
wimp_WCTITLEHI,
wimp_WCRESERVED
}wimp_wcolours;
```

wimp_iconflags

If the icon contains anti-aliased text, the colour fields give the font handle

```
typedef enum{
```

`wimp_ITEXT = 0x00000001,`	icon contains text
`wimp_ISPRITE = 0x00000002,`	icon is a sprite
`wimp_IBORDER = 0x00000004,`	icon has a border

`wimp_IHCENTRE = 0x00000008,`	text is horizontally centred
`wimp_IVCENTRE = 0x00000010,`	text is vertically centred
`wimp_IFILLED = 0x00000020,`	icon has a filled background
`wimp_IFONT = 0x00000040,`	text is in an anti-aliased font
`wimp_IREDRAW = 0x00000080,`	redraw needs application's help
`wimp_INDIRECT = 0x00000100,`	icon data is 'indirected'
`wimp_IRJUST = 0x00000200,`	text right-justified in box
`wimp_IESG_NOC = 0x00000400,`	if selected by Adjust, don't cancel other icons in same ESG
`wimp_IHALVESPRITE=0x00000800,`	plot sprites half-size
`wimp_IBTYPE = 0x00001000,`	4-bit field: button type
`wimp_ISELECTED = 0x00200000,`	icon selected by user (inverted)
`wimp_INOSELECT = 0x00400000,`	icon cannot be selected (shaded)
`wimp_IDELETED = 0x00800000,`	icon has been deleted
`wimp_IFORECOL = 0x01000000,`	4-bit field: foreground colour
`wimp_IBACKCOL = 0x10000000`	4-bit field: background colour
`}wimp_iconflags;`	

wimp_ibtype

Button types:

`typedef enum{`	
`wimp_BIGNORE,`	ignore all mouse ops
`wimp_BNOTIFY,`	
`wimp_BCLICKAUTO,`	
`wimp_BCLICKDEBOUNCE,`	
`wimp_BSELREL,`	
`wimp_BSELDOUBLE,`	
`wimp_BDEBOUNCEDRAG,`	
`wimp_BRELEASEDRAG,`	
`wimp_BDOUBLEDRAG,`	
`wimp_BSELNOTIFY,`	
`wimp_BCLICKDRAGDOUBLE,`	
`wimp_BCLICKSEL,`	useful for on/off and radio buttons
`wimp_Bwritable = 15`	

```
}wimp_ibtype;
```

wimp_bbits

Button state bits

```
typedef enum{
wimp_BRIGHT = 0x001,
wimp_BMID = 0x002,
wimp_BLEFT = 0x004,
wimp_BDRAGRIGHT = 0x010,
wimp_BDRAGLEFT = 0x040,
wimp_BCLICKRIGHT = 0x100,
wimp_BCLICKLEFT = 0x400
}wimp_bbits;
```

wimp_dragtype

```
typedef enum{
wimp_MOVE_WIND = 1,      change position of window
wimp_SIZE_WIND = 2,      change size of window
wimp_DRAG_HBAR = 3,      drag horizontal scroll bar
wimp_DRAG_VBAR = 4,      drag vertical scroll bar
wimp_USER_FIXED = 5,     user drag box – fixed size
wimp_USER_RUBBER = 6,    user drag box – rubber box
wimp_USER_HIDDEN = 7     user drag box – invisible box
}wimp_dragtype;
```

wimp_w

```
typedef int wimp_w;
```

Abstract window handle.

wimp_i

```
typedef int wimp_i;
```

Abstract icon handle.

wimp_t

```
typedef int wimp_t;
```

Abstract task handle.

wimp_icondata

The data field in an icon.

```
typedef union{
```

```
  char text[12];                   up to 12 bytes of text
  char sprite_name[12];            up to 12 bytes of sprite name
  struct {
    char *name;
    void *spritearea;              0 → use the common sprite area
                                   1 → use the Wimp sprite area
    BOOL nameisname;               if False, name is in fact a sprite
                                   pointer.
  } indirectsprite;
  struct {                         if indirect
    char *buffer;                  pointer to text buffer
    char *validstring;             pointer to validation string
    int bufflen;                   length of text buffer
  } indirecttext;
} wimp_icondata;
```

wimp_box

```
typedef struct{
int x0, y0, x1, y1
} wimp_box;
```

wimp_wind

If there are any icon definitions, they should follow this structure immediately in memory.

```
typedef struct{
wimp_box box;                      screen coordinates of work area
int scx, scy;                      scroll bar positions
wimp_w behind;                     handle to open window behind,
                                   or -1 if top
wimp_wflags flags;                 word of flag bits defined above
char colours[8];                   colours: index using
                                   wimp_wcolours.
wimp_box ex;                       maximum extent of work area
wimp_iconflags titleflags;         icon flags for title bar
wimp_iconflags workflags;          just button type relevant
```

`void *spritearea;`	0 → use the common sprite area
	1 → use the Wimp sprite area
`int minsize;`	two 16-bit OS-unit fields, (width/height) giving minimum size of window
	0 → use title
`wimp_icondata title;`	title icon data
`int nicons;`	number of icons in window
`} wimp_wind;`	

wimp_winfo

Result of `get_info` call. Space for icons must follow.

```
typedef struct {
wimp_w w;
wimp_wind info;
} wimp_winfo;
```

wimp_icon

Icon description structure.

`typedef struct {`	
`wimp_box box;`	bounding box – relative to window origin (work area top left)
`wimp_iconflags flags;`	word of flag bits defined above
`wimp_icondata data;`	union of bits & bobs as above
`} wimp_icon;`	

wimp_icreate

Structure for creating icons.

```
typedef struct {
wimp_w w;
wimp_icon i;
} wimp_icreate;
```

wimp_openstr

`typedef struct {`	
`wimp_w w;`	window handle

```
wimp_box box;              position on screen of visible
                           work area
int x, y;                  'real' coordinates of visible
                           work area
wimp_w behind;             handle of window to go behind
                           (–1 = top, –2 = bottom)
} wimp_openstr;
```

wimp_wstate

Result for window state enquiry.

```
typedef struct {
wimp_openstr o;
wimp_wflags flags;
} wimp_wstate;
```

wimp_etypes

Event types.

```
typedef enum {
wimp_ENULL,                null event
wimp_EREDRAW,              redraw event
wimp_EOPEN,
wimp_ECLOSE,
wimp_EPTRLEAVE,
wimp_EPTRENTER,
wimp_EBUT,                 mouse button change
wimp_EUSERDRAG,
wimp_EKEY,
wimp_EMENU,
wimp_ESCROLL,
wimp_ELOSECARET,
wimp_EGAINCARET,
wimp_ESEND = 17,           send message, don't worry if it
                           doesn't arrive
wimp_ESENDWANTACK = 18,    send message, return ack if not
                           acknowledged
wimp_EACK = 19             acknowledge receipt of message
} wimp_etype;
```

wimp_emask

Event type masks.

```
typedef enum {
wimp_EMNULL = 1 << wimp_ENULL,
wimp_EMREDRAW = 1 << wimp_EREDRAW,
wimp_EMOPEN = 1 << wimp_EOPEN,
wimp_EMCLOSE = 1 << wimp_ECLOSE,
wimp_EMPTRLEAVE = 1 << wimp_EPTRLEAVE,
wimp_EMPTRENTER = 1 << wimp_EPTRENTER,
wimp_EMBUT = 1 << wimp_EBUT,
wimp_EMUSERDRAG = 1 << wimp_EUSERDRAG,
wimp_EMKEY = 1 << wimp_EKEY,
wimp_EMMENU = 1 << wimp_EMENU,
wimp_EMSCROLL = 1 << wimp_ESCROLL
} wimp_emask;
```

wimp_redrawstr

```
typedef struct {
wimp_w w;
wimp_box box;               work area coordinates
int scx, scy;               scroll bar positions
wimp_box g;                 current graphics window
} wimp_redrawstr;
```

wimp_mousestr

```
typedef struct {
int x, y;                   mouse x and y
wimp_bbits bbits;           button state
wimp_w w;                   window handle, or -1 if none
wimp_i i;                   icon handle, or -1 if none
} wimp_mousestr;
```

wimp_caretstr

```
typedef struct {
wimp_w w;
wimp_i i;
int x, y;                   offset relative to window origin
```

`int height;`	–1 if calc within icon bit 24 → VDU-5 type caret bit 25 → caret invisible bit 26 → bits 16…23 contain colour bit 27 → colour is 'real' colour
`int index;`	position within icon
`} wimp_caretstr;`	

wimp_msgaction

Message action codes are allocated just like SWI codes.

`typedef enum {`	
`wimp_MCLOSEDOWN = 0,`	reply if any dialogue with the user is required, and the closedown sequence will be aborted.
`wimp_MDATASAVE = 1,`	request to identify directory
`wimp_MDATASAVEOK = 2,`	reply to message type 1
`wimp_MDATALOAD = 3,`	request to load/insert dragged icon
`wimp_MDATALOADOK = 4,`	reply that file has been loaded
`wimp_MDATAOPEN = 5,`	warning that an object is to be opened
`wimp_MRAMFETCH = 6,`	transfer data to buffer in my workspace
`wimp_MRAMTRANSMIT = 7,`	I have transferred some data to a buffer in your workspace
`wimp_MPREQUIT = 8,`	
`wimp_PALETTECHANGE = 9,`	
`wimp_FilerOpenDir = 0x0400,`	
`wimp_FilerCloseDir = 0x0401,`	
`wimp_Notify = 0x40040`	net filer notify broadcast
`wimp_MMENUWARN = 0x400c0,`	menu warning. Sent if `wimp_MSUBLINKMSG` set. Data sent is:

	submenu field of relevant `wimp_menuitem`.
	screen x-coord
	screen y-coord
	list of menu selection indices (`0..n-1` for each menu)
	terminating –1 word.
	Typical response is to call `wimp_create_submenu`.
`wimp_MMODECHANGE = 0x400c1,`	
`wimp_MINITTASK = 0x400c2,`	
`wimp_MCLOSETASK = 0x400c3,`	
`wimp_MSLOTCHANGE = 0x400c4,`	Slot size has altered
`wimp_MSETSLOT = 0x400c5,`	Task Manager requests application to change its slot size
`wimp_MTASKNAMERQ = 0x400c6,`	Request task name
`wimp_MTASKNAMEIS = 0X400c7,`	Reply to task name request
`wimp_MHELPREQUEST = 0x502,`	interactive help request
`wimp_MHELPREPLY = 0x503,`	interactive help message

Messages for dialogue with printer applications

`wimp_MPrintFile = 0x80140,`	Printer application's first response to a DATASAVE
`wimp_MWillPrint = 0x80141,`	Acknowledgement of PrintFile
`wimp_MPrintTypeOdd = 0x80145,`	Broadcast when strange files dropped on the printer
`wimp_MPrintTypeKnown = 0x80146,`	Acknowledgement to above
`wimp_MPrinterChange = 0x80147`	New printer application installed

```
} wimp_msgaction;
```

wimp_msghdr

Message block header. `size` is the size of the whole `msgstr`, see below.

`typedef struct {`	
`int size;`	20<=size<=256, multiple of 4

`wimp_t task;`	task handle of sender (filled in by Wimp)
`int my_ref;`	unique ref number (filled in by Wimp)
`int your_ref;`	(0==>none) if non-zero, acknowledge
`wimp_msgaction action;`	message action code
`} wimp_msghdr;`	

wimp_msgdatasave

`typedef struct {`	
`wimp_w w;`	window in which save occurs.
`wimp_i i;`	icon there
`int x; int y;`	position within that window of destination of save
`int estsize;`	estimated size of data, in bytes
`int type;`	file type of data to save
`char leaf[12];`	proposed leaf-name of file, 0-terminated
`} wimp_msgdatasave;`	

wimp_msgdatasaveok

`w, i, x, y, type, estsize` copied unaltered from DataSave message.

`typedef struct {`	
`wimp_w w;`	window in which save occurs.
`wimp_i i;`	icon there
`int x;int y;`	position within that window of destination of save.
`int estsize;`	estimated size of data, in bytes
`int type;`	file type of data to save
`char name[212];`	the name of the file to save
`} wimp_msgdatasaveok;`	

wimp_msgdataload

For a data load reply, no arguments are required.

`typedef struct {`	
`wimp_w w;`	target window

```
wimp_i i;                          target icon
int x; int y;                      target coordinates in target
                                   window work area
int size;                          must be 0
int type;                          type of file
char name[212];                    the filename follows.
} wimp_msgdataload;
```

wimp_msgdataopen

`wimp_msgdataopen` derives its typedef from `wimp_msgdataload`, since the data provided when opening a file is exactly the same. The window, x and y refer to the bottom lefthand corner of the icon that represents the file being opened, or w=–1 if there is no such icon.

wimp_msgramfetch

Transfer data in memory.

```
typedef struct {
char *addr;                        address of data to transfer
int nbytes;                        number of bytes to transfer
} wimp_msgramfetch;
```

wimp_msgramtransmit

'I have transferred some data to a buffer in your workspace'.

```
typedef struct {
char *addr;                        copy of value sent in RAMfetch
int nbyteswritten;                 number of bytes written
} wimp_msgramtransmit;
```

wimp_msghelprequest

```
typedef struct {
wimp_mousestr m;                   where the help is required
} wimp_msghelprequest;
```

wimp_msghelpreply

```
typedef struct {
char text[200];                    the helpful string
} wimp_msghelpreply;
```

wimp_msgprint

Structure used in all print messages.

```
typedef struct {
int filler[5] ;
int type ;                         filetype
char name[256-44] ;                filename
} wimp_msgprint;
```

wimp_msgstr

Message block.

```
typedef struct {
  wimp_msghdr hdr;
  union {
    char chars[236];
    int words[59];                 maximum data size.
    wimp_msgdatasave datasave;
    wimp_msgdatasaveok datasaveok;
    wimp_msgdataload dataload;
    wimp_msgdataopen dataopen;
    wimp_msgramfetch ramfetch;
    wimp_msgramtransmit ramtransmit;
    wimp_msghelprequest helprequest;
    wimp_msghelpreply helpreply;
    wimp_msgprint print;
  } data;
  } wimp_msgstr;
```

wimp_eventdata

```
typedef union {
wimp_openstr o;                    for redraw, close, enter, leave
                                   events
struct {
   wimp_mousestr m;
   wimp_bbits b;} but;             for button change event
wimp_box dragbox;                  for user drag box event
struct {wimp_caretstr c; int chcode;} key;  for key events
int menu[10];                      for menu event: terminated by -1
```

```
struct {wimp_openstr o; int x, y;} scroll;   for scroll request
                                                x=-1 for left, +1 for right
                                                y=-1 for down, +1 for up
                                              scroll by +/-2 -> page scroll
                                              request
wimp_caretstr c;                              for caret gain/lose
wimp_msgstr msg;                              for messages
} wimp_eventdata;
```

wimp_eventstr

Wimp event description.

```
typedef struct {
wimp_etype e;                                 event type
wimp_eventdata data;
} wimp_eventstr;
```

wimp_menuhdr

```
typedef struct {
char title[12];                               menu title (optional)
char tit_fcol, tit_bcol, work_fcol, work_bcol;   colours
int width, height;                            size of following menu items
int gap;                                      vertical gap between items
} wimp_menuhdr;
```

wimp_menuflags

Use `wimp_INOSELECT` to shade the item as unselectable, and the button type to mark it as writeable.

```
typedef enum {
wimp_MTICK = 1,
wimp_MSEPARATE = 2,
wimp_Mwriteable = 4,
wimp_MSUBLINKMSG = 8,                         show a => flag, and inform
                                              program when it is activated
wimp_MLAST = 0x80                             signal last item in the menu
} wimp_menuflags;
```

wimp_menuptr

Only for the circular reference in `menuitem/str`.

```
typedef struct wimp_menustr *wimp_menuptr;
```

wimp_menuitem

Submenu can also be a `wimp_w`, in which case the window is opened as a dialogue box within the menu tree.

```
typedef struct {
wimp_menuflags flags;          menu entry flags
wimp_menuptr submenu;          wimp_menustr* pointer to sub
                               menu, or wimp_w dialogue box,
                               or –1 if no submenu
wimp_iconflags iconflags;      icon flags for the entry
wimp_icondata data;            icon data for the entry
} wimp_menuitem;
```

wimp_menustr

```
typedef struct {
wimp_menuhdr hdr;              zero or more menu items follow
                               in memory
} wimp_menustr;
```

wimp_dragstr

```
typedef struct {
wimp_w window;
wimp_dragtype type;
wimp_box box;                  initial position for drag box
wimp_box parent;               parent box for drag box
} wimp_dragstr;
```

wimp_which_block

```
typedef struct {
wimp_w window;                 handle
int bit_mask;                  bit set => consider this bit
int bit_set;                   desired bit setting
} wimp_which_block;
```

wimp_pshapestr

```
typedef struct {
int shape_num;            pointer shape number (0 turn
                          off pointer)
char *shape_data;         shape data, NULL pointer
                          implies existing shape
int width, height;        Width and height in pixels
                          Width = 4n, where n is an
                          integer.
int activex, activey;     active point (pixels from top
                          left)
} wimp_pshapestr;
```

wimp_font_array

```
typedef struct {
char f[256];              initialise all to zero before
                          using for first
                          load_template, then just use
                          repeatedly without altering
} wimp_font_array;
```

wimp_template

Template reading structure

```
typedef struct {
int reserved;             ignore – implementation detail
wimp_wind *buf;           pointer to space for putting
                          template in
char *work_free;          pointer to start of free Wimp
                          workspace – you have to provide
                          the Wimp system with
                          workspace to store its redirected
                          icons in end of workspace you
                          are offering to the Wimp
char *work_end;
wimp_font_array *font;    points to font reference count
                          array; 0 pointer implies fonts
                          not allowed
char *name;               name to match with (can be
                          wildcarded)
```

```
int index;                                  position in index to search from
                                            (0 = start)
} wimp_template;
```

wimp_paletteword

The gcol char (least significant) is a gcol colour except in 8-bpp modes, when bits 0..2 are the tint and bits 3..7 are the gcol colour.

```
typedef union {
struct {char gcol; char red; char green; char blue;}
  bytes;
int word;
} wimp_paletteword;
```

wimp_palettestr

```
typedef struct {
wimp_paletteword c[16];                     Wimp colours 0..15
wimp_paletteword screenborder, mouse1, mouse2, mouse3;
} wimp_palettestr;
```

Function prototypes

wimp_initialise

```
os_error *wimp_initialise(int *v)
```

Closes and deletes all windows, returning Wimp version number.

wimp_taskinit

```
os_error *wimp_taskinit(char *name, wimp_t *t)
```

name is the name of the program. Used instead of wimp_initialise. Returns your task handle.

wimp_create_wind

```
os_error *wimp_create_wind(wimp_wind *, wimp_w *)
```

Defines (but does not display) window, returning window handle.

wimp_create_icon

```
os_error *wimp_create_icon(wimp_icreate *, wimp_i
*result)
```

Adds icon definition to that of window, returning icon handle.

wimp_delete_wind

`os_error *wimp_delete_wind(wimp_w)`

wimp_delete_icon

`os_error *wimp_delete_icon(wimp_w, wimp_i)`

wimp_open_wind

`os_error *wimp_open_wind(wimp_openstr *)`

Makes a window appear on the screen.

wimp_close_wind

`os_error *wimp_close_wind(wimp_w)`

Removes from the active list the window with its handle in the integer argument.

wimp_poll

`os_error *wimp_poll(wimp_emask mask, wimp_eventstr *result)`

Polls the next event from the Wimp.

wimp_save_fp_state_on_poll (void)

`os_error *wimp_save_fp_state_on_poll(void)`

Activates the saving of the floating point state on calls to `wimp_poll` and `wimp_pollidle`; this is needed if you do any floating point at all, as other programs may corrupt the FP status word, which is effectively a global in your program.

wimp_corrupt_fp_state_on_poll (void)

`void *wimp_corrupt_fp_state_on_poll(void)`

Disables the saving of the floating point state on calls to `wimp_poll` and `wimp_pollidle`; use only if you never use FP at all.

wimp_redraw_wind

`os_error *wimp_redraw_wind(wimp_redrawstr*, BOOL*)`

Draws a window outline and icons. Return False if there's nothing to draw.

wimp_update_wind

`os_error *wimp_update_wind(wimp_redrawstr*, BOOL*)`

Returns the visible portion of a window. Returns False if there's nothing to redraw.

wimp_get_rectangle

`os_error *wimp_get_rectangle(wimp_redrawstr*, BOOL*)`

Returns the next rectangle in the list, or False if done.

wimp_get_wind_state

```
os_error *wimp_get_wind_state(wimp_w, wimp_wstate
  *result)
```

Reads the current window state.

wimp_get_wind_info

```
os_error *wimp_get_wind_info(wimp_winfo *result)
```

On entry `result->w` gives the window in question. Space for any icons must follow `*result`.

wimp_set_icon_state

```
os_error *wimp_set_icon_state(wimp_w, wimp_i,
wimp_iconflags value, wimp_iconflags mask)
```

Sets an icon's flags as `(old_state & ~mask) ^ value`.

wimp_get_icon_info

```
os_error *wimp_get_icon_info(wimp_w, wimp_i, wimp_icon
*result)
```

Gets the current state of an icon.

wimp_get_point_info

```
os_error *wimp_get_point_info(wimp_mousestr *result)
```

Gives information regarding the state of the mouse.

wimp_drag_box

```
os_error *wimp_drag_box(wimp_dragstr *)
```

Starts the Wimp dragging a box.

wimp_force_redraw

```
os_error *wimp_force_redraw(wimp_redrawstr *r)
```

Marks an area of the screen as invalid. If `r->wimp_w == -1`, use screen coordinates. Only the first five fields of `r` are valid.

wimp_set_caret_pos

```
os_error *wimp_set_caret_pos(wimp_caretstr *)
```

Sets the position and size of the text caret.

wimp_get_caret_pos

```
os_error *wimp_get_caret_pos(wimp_caretstr *)
```

Gets the position and size of the text caret.

wimp_create_menu

```
os_error *wimp_create_menu(wimp_menustr *m, int x, int y)
```

'Pops up' a menu structure. Set `m==(wimp_menustr*)-1` to clear the menu tree.

wimp_decode_menu

`os_error *wimp_decode_menu(wimp_menustr *, void *, void *)`

wimp_which_icon

`os_error *wimp_which_icon(wimp_which_block *, wimp_i *results)`

The `results` appear in an array, terminated by a `(wimp_i) -1`.

wimp_set_extent

`os_error *wimp_set_extent(wimp_redrawstr *)`

Alters the extent of a window's work area – only the handle and the first set of four coordinates are looked at.

wimp_set_point_shape

`os_error *wimp_set_point_shape(wimp_pshapestr *)`

Sets the pointer shape on screen.

wimp_open_template

`os_error *wimp_open_template(char *name)`

Opens the named file to allow `load_template` to read a template from the file.

wimp_close_template

`os_error *wimp_close_template(void)`

Closes the currently open template file.

wimp_load_template

`os_error *wimp_load_template(wimp_template *)`

Loads a window template from an open file into buffer.

wimp_processkey

`os_error *wimp_processkey(int chcode)`

Hands back to the Wimp a key that you do not understand.

wimp_closedown

`os_error *wimp_closedown(void)`

wimp_taskclose

`os_error *wimp_taskclose(wimp_t)`

Calls closedown in the multi-tasking form.

wimp_starttask

`os_error *wimp_starttask(char *clicmd)`

Starts a new Wimp task, with the given CLI command.

wimp_getwindowout

`os_error *wimp_getwindowoutline(wimp_redrawstr *r)`

Sets `r→w` on entry. On exit, `r→box` will be the screen coordinates of the window, including border, title, scroll bars.

wimp_pollidle

`os_error *wimp_pollidle(wimp_emask mask, wimp_eventstr *result, int earliest)`

Like `wimp_poll`, but does not return before the earliest return time. This is a value produced by `OS_ReadMonotonicTime`.

wimp_ploticon

`os_error *wimp_ploticon(wimp_icon*)`

Called only within an update or redraw loop, and just does the plotting. This need not be a real icon attached to a window.

wimp_setmode

`os_error *wimp_setmode(int mode)`

Sets the screen mode. Palette colours are maintained, if possible.

wimp_readpalette

`os_error *wimp_readpalette(wimp_palettestr*)`

wimp_setpalette

`os_error *wimp_setpalette(wimp_palettestr*)`

The `bytes.gcol` values of each field of the `palettestr` are ignored; only the absolute colours are taken into account.

wimp_setcolour

`os_error *wimp_setcolour(int colour)`

bits 0…3 = Wimp colour (translate for current mode)
4…6 = gcol action
7 = foreground/background.

wimp_spriteop

`os_error *wimp_spriteop(int reason_code, char *name)`

Calls SWI `Wimp_SpriteOp`.

wimp_spriteop_full

`os_error *wimp_spriteop_full(os_regset *)`

Calls SWI `Wimp_SpriteOp` allowing full information to be passed.

wimp_baseofsprites

```
void *wimp_baseofsprites(void)
```

Returns a `sprite_area*`, which may be moved about by `mergespritefile`.

wimp_blockcopy

```
os_error *wimp_blockcopy(wimp_w, wimp_box *source, int
x, int y)
```

Copies the source box (defined in window coordinates) to the given destination (in window coordinates). Invalidates any portions of the destination that cannot be updated using on-screen copy.

wimp_errflags

```
typedef enum {

wimp_EOK = 1,               put in OK box
wimp_ECANCEL = 2,           put in CANCEL box
wimp_EHICANCEL = 4          highlight CANCEL rather than OK
} wimp_errflags;
```

If OK and CANCEL are both 0 you get an OK.

wimp_reporterror

```
os_error *wimp_reporterror(os_error*, wimp_errflags,
char *name)
```

Produces an error window. Uses sprite called `error` in the Wimp sprite pool. `name` should be the program name, appearing after `error in` at the head of the dialogue box.

wimp_sendmessage

```
os_error *wimp_sendmessage(wimp_etype code, wimp_msgstr*
msg, wimp_t dest)
```

`dest` can also be 0, in which case the message is sent to every task in turn, including the sender. `msg` can also be any other `wimp_eventdata*` value.

wimp_sendwmessage

```
os_error *wimp_sendwmessage(wimp_etype code, wimp_msgstr
*msg, wimp_w w, wimp_i i)
```

Sends a message to the owner of a specific window or icon. `msg` can also be any other `wimp_eventdata*` value.

wimp_create_submenu

```
os_error *wimp_create_submenu(wimp_menustr *sub, int x,
int y)
```

`sub` can also be a `wimp_w`, in which case it is opened by the Wimp as a dialogue box.

wimp_slotsize

```
os_error *wimp_slotsize (int *currentslot,
                         int *nextslot,
                         int *freepool)
```

`currentslot/nextslot==0 ->` just read setting.

wimp_transferblock

```
os_error *wimp_transferblock(
      wimp_t sourcetask,
      char *sourcebuf,
      wimp_t desttask,
      char *destbuf,
      int buflen)
```

Transfers memory between domains.

wimp_setfontcolours

```
os_error *wimp_setfontcolours(int foreground, int
background)
```

Sets font manager colours. The Wimp handles how many shades etc. to use.

wimp_readpixtrans

```
os_error *wimp_readpixtrans(sprite_area *area, sprite_id
*id, sprite_factors *factors, sprite_pixtrans *pixtrans)
```

Tells you how the Wimp will plot a sprite when asked to `PutSpriteScaled`.

wimp_command_tag

```
typedef enum {
wimp_command_TITLE = 0,
wimp_command_ACTIVE = 1,
wimp_command_CLOSE_PROMPT = 2,
wimp_command_CLOSE_NOPROMPT = 3
} wimp_command_tag;
```

wimp_commandwind

```
typedef struct {
wimp_command_tag tag;
char *title
} wimp_commandwind;
```

wimp_commandwindow

`os_error *wimp_commandwindow(wimp_commandwind commandwindow)`

Opens a text window for normal VDU 4-type output. The tag types correspond to the four kinds of call to SWI wimp_CommandWindow described in the *RISC OS Programmer's Reference Manual.* `title` is only required if `tag == wimp_command_TITLE`. It is the application's responsibility to set the tag correctly.

wimpt

These functions provide low-level Wimp functionality.

wimpt_poll

Polls for an event from the Wimp (with extras to buffer one event).

Syntax: `os_error *wimpt_poll(wimp_emask mask, wimp_eventstr *result)`

Parameters: `wimp_emask mask` – ignore events in the mask
`wimp_eventstr *result` – the event returned from Wimp

Returns: possible error condition.

Other Information: If you want to poll at this low level (ie avoiding `event_process()`), use this function rather than `wimp_poll`. Using `wimpt_poll` allows you to use the routines shown below.

wimpt_fake_event

Posts an event to be collected by `wimpt_poll`.

Syntax: `void wimpt_fake_event(wimp_eventstr *)`

Parameters: `wimp_eventstr` – the posted event

Returns: `void`

Other Information: use with care!

wimpt_last_event

Informs the caller of the last event returned by `wimpt_poll`.

Syntax: `wimp_eventstr *wimpt_last_event(void)`

Parameters: `void`

Returns: pointer to last event returned by `wimpt_poll`.

wimpt_last_event_was_a_key

Informs the caller if the last event returned by `wimpt_poll` was a key stroke.

Syntax: `int wimpt_last_event_was_a_key(void)`

Parameters: `void`

Returns: non-zero if last event was a keystroke.

Other Information: retained for backwards compatibility. Use `wimpt_last_event` for preference, and test if `e` field of returned struct == `wimp_EKEY`.

wimpt_noerr

Halts the program and reports an error in a dialogue box (if e!=0).

Syntax: `void wimpt_noerr(os_error *e)`

Parameters: `os_error *e` – error return from system call

Returns: `void`.

Other Information: Useful for 'wrapping up' system calls which are not expected to fail; if failure occurs, your program probably has a logical error. Call when an error would mean disaster: for example:

```
wimpt_noerr(some_system_call(.......));
```

The error message is :

```
ProgName has suffered a fatal internal error
(errormessage) and must exit immediately.
```

wimpt_complain

Reports an error in a dialogue box (if e!=0).

Syntax: `os_error *wimpt_complain(os_error *e)`

Parameters: `os_error *e` – error return from system call

Returns: the error returned from the system call (ie. `e`).

Other Information: Useful for 'wrapping up' system calls which may fail. Call when your program can still limp on regardless (taking some appropriate action).

wimptt: control of graphics environment

wimpt_checkmode

Registers the current screen mode with the `wimpt` module.

Syntax: `BOOL wimpt_checkmode(void)`

Parameters: `void`

Returns: True if screen mode has changed.

wimpt_mode

Reads the screen mode.

Syntax: `int wimpt_mode(void)`

Parameters: `void`

Returns: screen mode.

Other Information: faster than a normal OS call. Value is only valid if `wimpt_checkmode` is called at redraw events.

wimpt_dx/wimpt_dy

Informs the caller of OS x/y units per screen pixel.

Syntax: `int wimpt_dx(void)`
`int wimpt_dy(void)`

Parameters: `void`

Returns: OS x/y units per screen pixel.

Other Information: faster than a normal OS call. Value is only valid if `wimpt_checkmode` is called at redraw events.

wimpt_bpp

Informs the caller of bits per screen pixel.

Syntax: `int wimpt_bpp(void)`

Parameters: `void`

Returns: bits per screen pixel (in current mode)

Other Information: faster than a normal OS call. Value is only valid if `wimpt_checkmode` is called at redraw events.

wimpt_init

Set program up as a Wimp task.

Syntax: `void wimpt_init(char *programname)`

Parameters: `char *programname` – name of your program

Returns: `void`.

Other Information: Remembers screen mode, and sets up signal handlers so that task exits cleanly, even after fatal errors. Response to signals SIGABRT, SIGFPE, SIGILL, SIGSEGV and SIGTERM is to display error box with message:

progname has suffered an internal error (type = *signal*) and must exit immediately

SIGINT (Escape) is ignored. *progname* will appear in the Task manager display and in error messages. Calls `wimp_taskinit` and stores `task_id` returned. Also installs exit-handler to close down task when program calls `exit()` function.

wimpt_programname

Informs the caller of the name passed to `wimpt_init`.

Syntax: `char *wimpt_programname(void)`

Parameters: `void`.

Returns: pointer to the program's name.

wimpt_reporterror

Reports an OS error in a dialogue box (including program name).

Syntax: `void wimpt_reporterror(os_error*, wimp_errflags)`

Parameters: `os_error*` – OS error block
`wimp_errflags` – flag whether to include OK and/or CANCEL (highlighted or not) button in dialogue box

Returns: `void`.

Other Information: similar to `wimp_reporterror()`, but includes the program name automatically (eg the one passed to `wimpt_init`).

wimpt_task

Informs the caller of its task handle.

Syntax: `wimp_t wimpt_task(void)`

Parameters: `void`

Returns: task handle.

wimpt_forceredraw

Causes the whole screen to be invalidated (running applications will be requested to redraw all windows).

Syntax: `void wimpt_forceredraw(void)`

Parameters: `void`.

Returns: `void.`

win

This file offers central management of RISC OS windows, constructing a very simple idea of 'window class' within RISC OS. RISC OS window class implementations register the existence of each window with this module.

This structure allows event-processing loops to be constructed that have no knowledge of what other modules are present in the program. For instance, the dialogue box module can contain an event-processing loop without reference to what other window types are present in the program.

Claiming Events

win_register_event_handler

Installs an event handler function for a given window.

Syntax: `void win_register_event_handler(wimp_w, win_event_handler, void *handle)`

Parameters: `wimp_w` – the window's handle
`win_event_handler` – the event handler function
`void *handle` – caller-defined handle.

Returns: `void.`

Other Information: This call has no effect on the window itself – it just informs the `win` module that the supplied function should be called when events are delivered to the window. To remove a handler, call with a null function pointer:

```
win_register_event_handler(w,(win_event_handler)0,0)
```

To catch key events for an icon on the icon bar, register a handler for `win_ICONBAR`:

```
win_event_handler(win_ICONBAR, handler_func, handle)
```

To catch load events for an icon on the icon bar, register a handler for `win_ICONBARLOAD`:

```
win_event_handler(win_ICONBARLOAD, load_func, handle)
```

win_claim_idle_events

Causes 'idle' events to be delivered to a given window.

Syntax: `void win_claim_idle_events(wimp_w)`

Parameters: `wimp_w` – the window's handle.

Returns: `void`.

Other Information: To cancel this, call with window handle `(wimp_w)-1`.

win_add_unknown_event_processor

Adds a handler for unknown events onto the front of the queue of such handlers.

Syntax:

```
void win_add_unknown_event_processor
(win_unknown_event_processor, void *handle)
```

Parameters: `win_unknown_event_processor` – handler function

`void *handle` – passed to handler on call.

Returns: `void`.

Other Information: The `win` module maintains a list of unknown event handlers. An unknown event results in the 'head of the list' function being called; if this function doesn't deal with the event it is passed on to the next in the list, and so on. Handler functions should return a Boolean result to show if they dealt with the event, or if it should be passed on. 'Known' events are as follows:

ENULL, EREDRAW, ECLOSE, EOPEN, EPTRLEAVE, EPTRENTER, EKEY, ESCROLL, EBUT and ESEND/ESENDWANTACK for the following msg types: MCLOSEDOWN, MDATASAVE, MDATALOAD, MHELPREQUEST

All other events are considered 'unknown'. If none of the unknown event handlers deals with the event, then it is passed on to the unknown event claiming window (registered by `win_claim_unknown_events()`). If there is no such claimer, then the unknown event is ignored.

win_remove_unknown_event_processor

Removes the given unknown event handler with the given handle from the stack of handlers.

Syntax:

```
void win_remove_unknown_event_processor
(win_unknown_event_processor, void *handle)
```

Parameters:	`win_unknown_event_processor` – the handler to be removed `void *handle` – its handle.
Returns:	`void`.
Other Information:	The handler to be removed can be anywhere in the stack (not necessarily at the top).

win_idle_event_claimer

Informs the caller of which window is claiming idle events.

Syntax:	`wimp_w win_idle_event_claimer(void)`
Parameters:	`void`
Returns:	Handle of window claiming idle events.
Other Information:	Returns `(wimp_w)-1`, if no window is claiming idle events.

win_claim_unknown_events

Cause any unknown, or non-window-specific events to be delivered to a given window.

Syntax:	`void win_claim_unknown_events(wimp_w)`
Parameters:	`wimp_w` – handle of window to which unknown events should be delivered.
Returns:	`void`.
Other Information:	Calling with `(wimp_w)-1` cancels this. See `win_add_unknown_event_processor()` for details of which events are 'known'.

win_unknown_event_claimer

Informs the caller of which window is claiming unknown events.

Syntax:	`wimp_w win_unknown_event_claimer(void)`
Parameters:	`void`
Returns:	Handle of window claiming unknown events.
Other Information:	Return of `(wimp_w)-1` means no claimer registered.

win: menus

win_setmenuh

Attaches the given menu structure to the given window.

Syntax:	`void win_setmenuh(wimp_w, void *handle)`

Parameters: `wimp_w` – handle of window
`void *handle` – pointer to menu structure.

Returns: `void`.

Other Information: Mainly used by higher level RISC_OSlib routines to attach menus to windows (eg `event_attachmenu()`).

win_getmenuh

Returns a pointer to the menu structure attached to the given window.

Syntax: `void *win_getmenuh(wimp_w)`

Parameters: `wimp_w` – handle of window

Returns: pointer to the attached menu (0 if no menu attached).

Other Information: As for `win_setmenuh()`, this is used mainly by higher level RISC OS routines (eg `event_attachmenu()`).

win: event processing

win_processevent

Delivers an event to its relevant window, if such a window has been registered with this module (via `win_register_event_handler()`).

Syntax: `BOOL win_processevent(wimp_eventstr*)`

Parameters: `wimp_eventstr*` – pointer to the event which has occurred

Returns: True if an event handler (registered with this module) has dealt with the event, False otherwise.

Other Information: the main client for this routine is `event_process()`, which uses it to deliver an event to its appropriate window. Keyboard events are delivered to the current owner of the caret.

win: termination

win_activeinc

Increment by one the `win` module's idea of the number of active windows owned by a program.

Syntax: `void win_activeinc(void)`

Parameters: `void`

Returns: `void`.

Other Information: `event_process()` calls `exit()` on behalf of the program when the number of active windows reaches zero. Programs which wish to remain running even when they have no active windows should ensure that `win_activeinc()` is called once before creating any windows, so that the number of active windows is always >= 1. This is done for you if you use `baricon()` to install your program's icon on the icon bar.

win_activedec

Decrements by one the `win` module's idea of the number of active windows owned by a program.

Syntax: `void win_activedec(void)`

Parameters: `void`.

Returns: `void`.

Other Information: See the note in `win_activeinc()` regarding program termination.

win_activeno

Informs the caller of the number of active windows owned by your program.

Syntax: `int win_activeno(void)`

Parameters: `void`.

Returns: number of active windows owned by the program.

Other Information: This is given by (number of calls to `win_activeinc()`) minus (number of calls to `win_activedec()`). Note that modules in the RISC OS library itself may have made calls to `win_activeinc()` and `win_activedec()`.

win_give_away_caret

Gives the caret away to the open window at the top of the Wimp's window stack (if that window is owned by your program).

Syntax: `void win_give_away_caret(void)`

Parameters: `void`.

Returns: `void`.

Other Information: If the top window is interested it will take the caret. If not then nothing happens. This only works if polling is done using the `wimpt` module, which is the case if your main inner loop goes something like: `while (TRUE) event_process().`

win_settitle

Changes the title displayed in a given window.

Syntax: `void win_settitle(wimp_w w, char *newtitle);`

Parameters: `wimp_w w` – given window's handle

`char *newtitle` – null-terminated string giving new title for window.

Returns: `void`.

Other information: The title icon of the given window must be indirected text. This will change the title used by all windows created from the given window's template if you have used the template module (since the Window Manager uses your address space to hold indirected text icons). To avoid this, the window can be created from a copy of the template, ie

```
template *t = template_copy(template_find("name"));
wimp_create_wind(t->window, &w);
```

xferrecv

This file covers the general purpose importing of data by dragging icons.

xferrecv_checkinsert

Sets up the acknowledge message for a MDATAOPEN or MDATALOAD and gets the filename to load from.

Syntax: `int xferrecv_checkinsert(char **filename)`

Parameters: `char **filename` – returned pointer to filename.

Returns: the file's type (eg. 0x0fff for Edit).

Other Information: This function checks to see if the last Wimp event was a request to import a file. If it was, the function returns file type and a pointer to file's name is put into `*filename`. Otherwise, it returns –1.

xferrecv_insertfileok

Deletes the scrap file (if used for transfer), and sends acknowledgement of MDATALOAD message.

Syntax: `void xferrecv_insertfileok(void)`

Parameters: `void`

Returns: `void`.

xferrecv_checkprint

Sets up an acknowledge message to a MPrintTypeOdd message and gets the filename to print.

Syntax: `int xferrecv_checkprint(char **filename)`

Parameters: `char **filename` – returned pointer to filename.

Returns: The file's type (eg. `0x0fff` for Edit).

Other Information: The application can either print the file directly or convert it to Printer$`Temp` for printing by the printer application.

xferrecv_printfileok

Sends an acknowledgement back to the printer application. If a file is sent to Printer$Temp, this also fills in the file type in the message.

Syntax: `void xferrecv_printfileok(int type)`

Parameters: `int type` – type of file sent to Printer$Temp (eg `0x0fff` for Edit).

Returns: `void`.

xferrecv_checkimport

Sets up an acknowledgement message to a MDATASAVE message.

Syntax: `int xferrecv_checkimport(int *estsize)`

Parameters: `int *estsize` – sender's estimate of file size

Returns: File type.

xferrecv_buffer_processor

This is a typedef for the caller-supplied function to empty a full buffer during data transfer.

Syntax: `typedef BOOL (*xferrecv_buffer_processor)(char **buffer, int *size)`

Parameters: `char **buffer` – new buffer to be used

`int *size` – updated size.

Returns: False if unable to empty buffer or create new one.

Other Information: This is the function, supplied by the application, which will be called when the buffer is full. It should empty the current buffer, or create more space and modify size accordingly, or return False. `*buffer` and `*size` are the current buffer and its size on function entry.

xferrecv_doimport

Loads data into a buffer, and calls the caller-supplied function to empty the buffer when full.

Syntax: `int xferrecv_doimport(char *buf, int size, xferrecv_buffer_processor)`

Parameters: `char *buf` – the buffer

`int size` – buffer's size

`xferrecv_buffer_processor` – caller-supplied function to be called when the buffer is full.

Returns: Number of bytes transferred on successful completion; –1 otherwise.

xferrecv_file_is_safe

Informs the caller if the file was received from a 'safe' source (see below for what this means).

Syntax: `BOOL xferrecv_file_is_safe(void)`

Parameters: `void`

Returns: True if file is safe.

Other Information: 'Safe' in this context means that the supplied filename will not change in the foreseeable future.

xfersend

This file covers the general purpose export of data by dragging icons.

xfersend: caller-supplied function types

xfersend_saveproc

A function of this type should save to the given file and return True if successful. Handle is passed to the function by `xfersend()`.

Syntax: `typedef BOOL (*xfersend_saveproc)(char *filename, void *handle)`

Parameters: `char *filename` – file to be saved

`void *handle` – the handle you passed to `xfersend()`.

Returns: True if the save was successful.

xfersend_sendproc

A function of this type should call `xfersend_sendbuf()` to send one buffer-full of data no bigger than `*maxbuf`.

Syntax: `typedef BOOL (*xfersend_sendproc)(void *handle, int *maxbuf)`

Parameters: `void *handle` – handle which was passed to `xfersend()`

`int *maxbuf` – size of receiver's buffer.

Returns: True if the data was successfully transmitted.

Other Information: Your `sendproc` will be called by functions in the `xfersend` module to do an in-core data transfer, on receipt of MRAMFetch messages from the receiving application. If `xfersend_sendbuf()` returns False, then return False **immediately**.

xfersend_printproc

A function of this type should either print the file directly, or save it into the given filename, from where it will be printed by the printer application.

Syntax:
```
typedef int (*xfersend_printproc)(char *filename, void
*handle)
```

Parameters: `char *filename` – file to save into, for printing
`void *handle` – handle that was passed to `xfersend()`

Returns: Either the file type of the file it saved, or one of the reason codes #defined below.

Other Information: This is called if the file icon has been dragged onto a printer application.

Reason codes:

`#define xfersend_printPrinted -1`	file dealt with internally
`#define xfersend_printFailed -2`	had an error along the way

The `saveproc` should report any errors it encounters itself. If saving to a file, it should convert the data into a type that can be printed by the printer application (ie text).

xfersend: library functions

xfersend

Allows the user to export application data, by icon drag.

Syntax:
```
BOOL xfersend(int filetype, char *name, int estsize,
xfersend_saveproc, xfersend_sendproc, xfersend_printproc,
wimp_eventstr *e, void *handle)
```

Parameters: `int filetype` – type of file to save to
`char *name` – suggested file name
`int estsize` – estimated size of the file
`xfersend_saveproc` – caller-supplied function for saving application data to a file

`xfersend_sendproc` – caller-supplied function for in-core data transfer (if application is able to do this)
`xfersend_printproc` – caller-supplied function for printing application data, if icon is dragged onto printer application
`wimp_eventstr *e` – the event which started the export (usually mouse drag)
`void *handle` – handle to be passed to handler functions.

Returns: True if data exported successfully.

Other Information: You should typically call this function in a window's event handler, when you get a mouse drag event. See the `saveas.c` code for an example of this. `xfersend` deals with the complexities of message-passing protocols to achieve the data transfer. Refer to the above type definitions for an explanation of what the three caller-supplied functions should do.

If `name` is 0 then a default name of `Selection` is supplied.

If you pass 0 as the `xfersend_sendproc`, no in-core data transfer will be attempted.

If you pass 0 as the `xfersend_printproc`, the file format for printing is assumed to be the same as for saving. The estimated file size is not essential, but may improve performance.

xfersend_sendbuf

Sends the given buffer to a receiver.

Syntax: `BOOL xfersend_sendbuf(char *buffer, int size)`

Parameters: `char *buffer` – the buffer to be sent
`int size` – the number of characters placed in the buffer.

Returns: True if send was successful.

Other Information: This function should be called by the caller-supplied `xfersend_sendproc` (if such exists) to do in-core data transfer (see notes on `xfersend_sendproc` above).

xfersend_file_is_safe

Informs the caller if the file's name can be reliably assumed not to change (during data transfer!).

Syntax: `BOOL xfersend_file_is_safe(void)`

Parameters: `void`.

Returns: True if file is 'safe'.

Other Information: See also the `xferrecv` module.

Returns: True if file recipient will not modify it; changing the window title of the file can be done conditionally on this result. This can be called within your `xfersend_saveproc`, `sendproc`, or `printproc`, or immediately after the main `xfersend`.

xfersend_set_fileissafe

Allows the caller to set an indication of whether a file's name will remain unchanged during data transfer.

Syntax: `void xfersend_set_fileissafe(BOOL value)`

Parameters: `BOOL value` – True means the file is safe.

Returns: `void`.

Assembly language interface

Object code modules from the Acorn C compiler can be linked with those produced by ObjAsm, provided they observe the conventions of the ARM Procedure Call Standard.

This chapter gives a brief description of how to handle procedure entry and exit in assembly language in order to interface to C. For details of ObjAsm syntax and AOF files, you should consult *Appendix D: ARM Procedure Call Standard* and the Archimedes *Assembler* Guide.

Register names

The following names are used in referring to ARM 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
`sl`	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 `sl` 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 `sl` define a temporary workspace of at least 256 bytes available to the procedure.
- `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.
- `sl` contains a stack chunk handle, which is used by stack handling code to extend the stack in a non-contiguous manner.

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`, `sl`, `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 `sp`, 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 sl to point to a new stack segment as necessary.

Function workspace

The values of sp and sl passed to a called function define an area of readable, writeable memory available to the called function as workspace. All words below [sp] and at or above [sl,#-512] are guaranteed to be available for reading and writing, and the minimum allowed value of sp is sl-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 sl, in other words that at least 512 bytes remain. If the value in sp is less than that in sl, the function should call the stack extension function x$stack_overflow. Functions which need more than 256 bytes of workspace should amend the test accordingly, and call x$stack_overflow1, 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 *How to use the C library kernel*.

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 with the compiler option -S.

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  |x$stack_overflow|
        EXPORT  |gggg|
gggx    DCB     "gggg", 0          ;name of function, 0 terminated
```

```
        ALIGN                           ;padded to word boundary
gggy    DCD     &ff000000 + gggy - gggx
                                        ;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
        CMPS    sp, sl
        BLLT    |x$stack_overflow|
;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
        BLLT    |x$stack_overflow1|
```

where `n` is the number of bytes needed. Note that `x$stack_overflow1` 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
        CMPS  sp, sl                    ;test workspace
        BLLT  |x$stack_overflow|
```

How to write relocatable modules in C

Introduction

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 Archimedes hardware.

Relocatable modules also provide mechanisms which can be exploited to:

- extend RISC OS's repertoire of built-in commands (* commands) (analogous to plugging additional ROMs into a BBC microcomputer of pre-Archimedes vintages)
- provide services to applications (for example, as does the shared C library module)
- implement 'terminate and stay resident' (TSR) applications.

The idea of TSR applications will be most familiar to PC users, whereas extending the * command set (via 'software ROM modules') will seem most familiar to those with a background in the BBC computer. 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 mechanism available to be exploited.

Getting started

To write a module in C you will need:

- a copy of the C compiler Release 3, C Shared Library module Release 3.5, and Shared C library stubs (Release 3);
- a copy of the C Module Header Generation tool, `cmhg`;
- a thorough understanding of RISC OS modules (read the chapter of the *RISC OS Programmer's Reference Manual* entitled *Modules*).

If you also intend to interface assembly code to your module, you should note that the procedure call standard used by the Release 3 C system is different from that used by pre-Release 3 C systems as follows:

APCS register name:	`sl`	`fp`	`ip`	`sp`
Release 3 register number:	10	11	12	13
pre-Release 3 register number:	13	10	11	12

All other register numberings are invariant between the releases.

Constraints on modules written in C

A module written in C **must** use the shared C library module via the library stubs. Use of the stand-alone C library (AnsiLib) is **not** a supported option.

All components of a module written in C **must** be compiled using the compiler option `-zM`. This allows the module's static data to be separated from its code and multiply instantiated.

Modules written in C should **not** be compiled with stack limit checking disabled. The stack limit check is cheap and can save your machine from crashing.

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 command with the `-m[odule]` option.

In the next section we describe:

- how to drive `cmhg` to make a module header and any necessary entry veneers
- the interface definitions to which each component of your module must conform
- how to use `cmhg` to generate entry veneers for IRQ 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.
- 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 on `exit()` if you register it with the C library by using the `atexit()` library function.
- 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.

- 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.
- 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.
- IRQ handlers. Though not associated with the module header, cmhg will generate entry veneers for IRQ 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.

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 Acorn Object Format). The command format is:

```
cmhg <input-file-name> <output-file-name>
```

or

```
cmhg <input-file-name>
```

if you merely wish to check the correctness of your input.

Example:

```
cmhg MyModHdr o.modhdr
```

cmhg will not create or overwrite the output object file if it detects any error in its input.

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 it, 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. You may omit any trailing arguments that you don't need from your handler implementations.

Runnable application code

cmhg description:

```
module-is-runnable:                ; No parameters.
```

C interface:

```
int main(int argc, char *argv[]);
   /*
    * Entered in user-mode with argc and argv
    * set up as for any other application. Malloc
    * obtains storage from application workspace.
    */
```

To be useful (ie re-runnable) as a 'terminate and stay resident' application, 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).

Initialisation code

cmhg description:

```
    initialisation-code: user_init  ; The name of your initialisation function.
                                    ; Any valid C function name will do.
```

C interface:

```
    _kernel_oserror *user_init(char *cmd_fail, int podule_base, void *pw);
   /*
    * Return NULL if your initialisation succeeds; otherwise return a pointer to an
    * error block. cmd_tail points to the string of arguments with which the
    * module is invoked (may be "").
    * podule_base is 0 unless the code has been invoked from a podule.
    * 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_Call Every (use _kernel_swi() to
    * issue the SWI call).
    */
```

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).

Finalisation code

User finalisations are handled by using atexit() to register finalisation functions. A call to library finalisation code is inserted automatically by cmhg; the C library finalisation code will call these registered functions immediately before closing down the library (on module finalisation).

Service call handler

cmhg description:

```
service-call-handler:  sc_handler <number> <number> ...
```

C interface:

```
void sc_handler(int service_number, _kernel_swi_regs *r, void *pw);
/*
 * 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.
 */
```

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:

```
CMPS    r1, #FirstInterestingServiceNumber
CMPNES  r1, #SecondInterestingServiceNumber
...
CMPNES  r1, #NthInterestingServiceNumber
MOVNES  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.

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 current date into the version string, as required by RISC OS convention.

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      0.91 (27 JUN 1989) Experimental version
```

`cmhg` refuses to generate a help string longer than 79 characters and warns if it has to truncate your input.

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:                     ;>flag bits in
   status:                         ;>the flag byte
   configure:                      ;>of the cmd table
   help:                           ;>info word.
   invalid-syntax: <text>
   help-text:      <text>
   ")"
```

Each sub-argument is optional. A comma after any item allows continuation on the next line.

A `<text>` item follows the conventions of ANSI 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** inlude a newline in the string, which must be represented by \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` and `help` set bits in the command's information word which mark the command as being of one of those classes.

`invalid-syntax` and `help-text` messages are (should be) self-explanatory.

Example `cmhg` description:

```
command-keyword-table: cmd_handler
  tm0(min-args: 0, max-args: 255,
     help-text: "Syntax\ttm1 <filenames>\n"),
  tm1(min-args:1, max-args:1,
     help-text: "Syntax\ttm2" " <integer>"
     "\n")
```

This describes two * commands, *tm0 and *tm1, 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.

C interface:

```
_kernel_oserror *cmd_handler(char *arg_string, int argc, int cmd_no, void *pw);
/*
 * If cmd_no identifies a *HELP entry, then cmd_handler must return
 * arg_string or NULL (if arg_string is returned, the NUL-terminated
 * buffer will be printed).
 * Return NULL if 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).
 * *STATUS and *CONFIGURE handlers will need to cast 'arg_string' to
 * (possibly unsigned) long and ignore argc. See the RISC OS Programmer's
 * Reference Manual for details.
 * pw is the private word pointer ('r12') value passed into the entry veneer
 */
```

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 Acorn: read the documentation carefully to discover which chunks you may use safely. In some cases you may need to write to Acorn to get a chunk allocated uniquely to your product (though this should not be undertaken lightly and should only be done when all alternatives have been exhausted). See the chapter entitled *An introduction to SWIs* in the *RISC OS Programmer's Reference Manual* for more details.

SWI handler code

`cmhg` description:

```
swi-handler-code:  swi_handler  ; any valid C function name will do
```

C interface:

```
_kernel_oserror *swi_handler(int swi_no, _kernel_swi_regs *r, void *pw);
/*
 * 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 handler may update any of its input registers (r0-r9).
 * ps is the private word pointer ('r12') value passed into the
 * swi_handler entry veneer.
 */
```

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:  swi_decoder ; any valid C
                                  function name will do
```

C interface:

```
void swi_decode(int r[4], void *pw);
/*
 * 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 th 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] points to the byte beyond the end of the buffer.
```

```
 * You should write th SWI name into the buffer at th 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 swi_decode
 * entry veneer.
 */
```

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.

IRQ handlers

`cmhg` description:

```
irq-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:

```
extern int entry_name(_kernel_swi_regs *r, void *pw);
/*
 * This is name of the IRQ 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 IrqV.
 */
int handler_name(_kernel_swi_regs *r, void *pw);
/*
 * This is the handler function you must write to handle the IRQ for
 * which entry_name is the veneer function.
 *
 * Return 0 if you handled the interrupt.
 * Return non-0 if you did NOT handle the interrupt (because,
 * for example, it wasn't for your handler, but for some other
 * handler further down the stack of handlers).
 *
 * 'r' points to a vector of words containing the values of r0-r9 on
 * entry to the veneer. Pure IRQ handlers do not require these, though
 * event handlers and filing system entry points do. 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 IRQ entry veneer is called.
 */
```

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 cannot be entered directly from IRQ mode, but have to be entered and exited via a veneer which switches to SVC mode. Running in SVC mode gives your handler maximum flexibility.

IRQ handlers can also be used as event handlers and filing system entry points. A full discussion of these topics is beyond the scope of this Guide; refer to the *RISC OS Programmer's Reference Manual* for details and for information on how to install and remove handlers.

Turning interrupts on and off

The following (`<kernel.h>`) library functions support the control of the interrupt enable state:

```
int _irqs_disabled(void);
/*
 * Returns non-0 if IRQs are currently disabled.
 */
void _irqs_off(void);
/*
 * Disable IRQs.
 */
void _irqs_on(void);
/*
 * Enable IRQs.
 */
```

These functions suffice to allow saving, restoring and setting of the IRQ state. Ground rules for using these functions are beyond the scope of this document. However, general advice is to leave the IRQ state alone in SWI handlers which terminate quickly, but to enable it in long-running SWI handlers.

What a SWI handler does to the IRQ state is part of its interface contract with its clients: you, the implementor, control that interface contract.

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 floppy disc or 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 tradeoff. 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, their use can occasionally be a 'life saver'.

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 abitrarily 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.

RISC OS is not a paged system, but Acorn's sister product, the Unix-based R140/RISC iX, is.

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 entitled *Generating overlaid programs* in the *Linker* chapter. 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 section on Squeeze in the chapter *Other utilities* for details). In contrast, overlay segments cannot be squeezed (though the root program can be). Consider, for example, the following programs from this release of C:

Program	**Squeezed Size**	**Unsqueezed Size**
`amu`	13Kb	23Kb
`cmhg`	9Kb	16Kb
`link`	22Kb	41Kb
`squeeze`	8Kb	14Kb
`SharedCLibrary`		61Kb

So, if you can structure your application as independent, squeezed programs it may take up less precious floppy 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.

Naturally, if you are porting an existing application to RISC OS, your view will be coloured by whether or not it is already structured to use overlays. If it is, it will probably be best to stick to using overlays, rather than attempting to split the application up into semi-independent sub-applications.

On the other hand, if you are writing an application from scratch, you probably want to ponder this question in more depth. For example, to what other systems will the application be targetted? Using multiple semi-independent applications may work very nicely under Unix or OS/2 where the output of one process can be piped into another, but less well under MS-DOS where use of overlays is much more the norm.

Machine-specific features

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

- the C library kernel
- calling other programs from C
- the shared C library
- `#pragma` directives
- storage management
- handling host errors.

How to use the C library kernel

C library structure

The C library is organised into layers, like the skins of an onion. At the centre is the language-independent library kernel. This is implemented in assembly language and provides basic support services, described below, to language run-time systems and, directly, to client applications.

One level out from the library kernel is a thin, C-specific layer, also implemented in assembly language. This provides compiler support functions such as structure copy, interfaces to stack-limit checking and stack extension, `setjmp` and `longjmp` support, etc. Everything above this level is written in C.

Finally, there is the C library proper. This is implemented in C and, with the exception of one module which interfaces to the library kernel and the C-specific veneer, is highly portable.

The library kernel

The library kernel is designed to allow run-time libraries for different languages to co-reside harmoniously, so that inter-language calling can be smooth. At the present time, the Fortran-77 library uses the run-time kernel, but the Pascal library does not. Currently, code compiled by the F77 compiler does not adhere to the new procedure-call standard, so inter-working with C is not possible in this release.

The library kernel provides the following facilities:

- a generic, status-returning, procedural interface to SWIs
- a procedural interface to the following commonly used SWIs:

  OS_Byte
  OS_Rdch
  OS_Wrch
  OS_BGet
  OS_BPut
  OS_GBPB
  OS_Word
  OS_Find
  OS_File
  OS_Args
  OS_CLI /* use is not advised – use `_kernel_system()` */

- a procedural interface to the following arithmetic functions:

  unsigned integer division
  unsigned integer remainder
  unsigned divide by 10 (much faster than general division)
  signed integer division
  signed integer remainder
  signed divide by 10 (much faster than general division).

- a procedural interface to the following miscellaneous functions:

  finding the identity of the host system (RISC OS, Arthur, etc)
  determining whether the floating point instruction set is available
  getting the command string with which the program was invoked
  returning the identity of the last OS error
  reading an environmental variable
  setting an environmental variable
  invoking a sub-application
  claiming memory to be managed by a heap manager

unwinding the stack
finding the name of a function containing a given address
finding the source language associated with code at a given address.

- support for manipulating the IRQ state from a relocatable module:

  getting the processor mode
  determining if IRQs are enabled
  enabling IRQs
  disabling IRQs.

- support for allocating and freeing memory in the RMA area:

  allocating a block of memory in the RMA
  extending a block of memory in the RMA
  freeing a block of memory in the RMA.

- support for stack-limit checking and stack extension:

  finding the current stack chunk
  four kinds of stack extension – small-frame and large-frame extension, number of actual arguments known (eg Pascal), or unknown (eg C) by the callee.

- trap handling, error handling, event handling and escape handling.

Most of these functions are described in the C library header file `<kernel.h>`. This header also declares the data structures you will need to use in order to call these functions or to interpret their results. See *Appendix D* for a detailed description.

Interfacing a language run-time system to the Acorn library kernel

In order to use the kernel, a language run-time system must provide an area named RTSK$$DATA, with attributes READONLY. The contents of this area must be a `_kernel_languagedescription` as follows:

```
typedef enum { NotHandled, Handled } _kernel_HandledOrNot

typedef struct {
   int regs [16];
} _kernel_registerset;

typedef struct {
   int regs [10];
} _kernel_eventregisters;

typedef void (*PROC) (void);
typedef _kernel_HandledOrNot
```

```
    (*_kernel_trapproc) (int code, _kernel_registerset *regs);
typedef _kernel_HandledOrNot
    (*_kernel_eventproc) (int code, _kernel_registerset *regs);

typedef struct {
    int size;
    int codestart, codeend;
    char *name;
    PROC (*InitProc)(void);  /* that is, InitProc returns a PROC */
    PROC FinaliseProc;
    _kernel_trapproc TrapProc;
    _kernel_trapproc UncaughtTrapProc;
    _kernel_eventproc EventProc;
    _kernel_eventproc UnhandledEventProc;
    void (*FastEventProc) (_kernel_eventregisters *);
    int (*UnwindProc) (_kernel_unwindblock *inout, char **language);
    char * (*NameProc) (int pc);
} _kernel_languagedescription;
```

Any of the procedure values may be zero, indicating that an appropriate default action is to be taken. Procedures whose addresses lie outside of `[codestart...codeend]` also cause the default action to be taken.

codestart, codeend

These values describe the range of program counter (PC) values which may be taken while executing code compiled from the language. The linker ensures that this is describable with just a single base and limit pair if all code is compiled into areas with the same unique name and same attributes (conventionally, *Language*$$code, CODE, READONLY. The values required are then accessible through the symbols *Language*$$code$$Base and *Language*$$code$$Limit).

InitProc

The kernel contains the entrypoint for images containing it. After initialising itself, the kernel calls (in a random order) the InitProc for each language RTS present in the image. They may perform any required (language-library-specific) initialisation: their return value is a procedure to be called in order to run the main program in the image. If there is no main program in its language, an RTS should return 0. (An InitProc may not itself enter the main program, otherwise other language RTSs might not be initialised. In some cases, the returned procedure may be the main program itself, but mostly it will be a piece of language RTS which sets up arguments first.)

It is an error for all InitProcs in a module to return 0. What this means depends on the host operating system; if RISC OS, SWI OS_GenerateError is called (having first taken care to restore all OS handlers). If the default error handlers are in place, the difference is marginal.

FinaliseProc

On return from the entry call, or on call of the kernel's Exit procedure, the FinaliseProc of each language RTS is called (again in a random order). The kernel then removes its OS handlers and exits setting any return code which has been specified by call of `_kernel_setreturncode`.

TrapProc, UncaughtTrapProc

If an image is not being run under a debugger, the kernel installs OS trap and error handlers. On occurrence of a trap, or of a fatal error, all registers are saved in an area of store belonging to the kernel. These are the registers at the time of the instruction causing the trap, except that the PC is wound back to address that instruction rather than pointing a variable amount past it.

The PC at the time of the trap together with the call stack are used to find the TrapHandler procedure of an appropriate language. If one is found, it is invoked in user mode. It may return a value (Handled or NotHandled), or may not return at all. If it returns Handled, execution is resumed using the dumped register set (which should have been modified, otherwise resumption is likely just to repeat the trap). If it returns NotHandled, then that handler is marked as failed, and a search for an appropriate handler continues from the current stack frame.

If the search for a trap handler fails, then the same procedure is gone through to find a 'uncaught trap' handler.

If this too fails, it is an error. It is also an error if a further trap occurs while handling a trap. The procedure `_kernel_exittraphandler` is provided for use in the case the handler takes care of resumption itself (eg via `longjmp`).

(A language handler is appropriate for a PC value if LanguageCodeBase <= PC and PC < LanguageCodeLimit, and it is not marked as failed. Marking as 'failed' is local to a particular kernel trap handler invocation. The search for an appropriate handler examines the current PC, then R14, then the link field of successive stack frames. If the stack is found to be corrupt at any time, the search fails).

EventProc, UnhandledEventProc

The kernel always installs a handler for OS events and for Escape flag change. On occurrence of one, all registers are saved and an appropriate EventProc, or failing that an appropriate UnhandledEventProc is found and called. Escape pseudo-events are processed exactly like Traps. However, for 'real' events, the search for a handler terminates as soon as a handler is found, rather than when a willing handler is found (this is done to limit the time taken to respond to an event). If no handler is willing to claim the event, it is handed to the event handler which was in force when the program started. (The call happens in CallBack, and if it is the result of an Escape, the Escape has already been acknowledged.)

In the case of escape events, all side effects (such as termination of a keyboard read) have already happened by the time a language escape handler is called.

FastEventProc

The treatment of events by EventProc isn't too good if what the user level handler wants to do is to buffer events (eg conceivably for the key up/down event), because there may be many to one event handler call. The FastEventProc allows a call at the time of the event, but this is constrained to obey the rules for writing interrupt code (called in IRQ mode; must be quick; may not call SWIs or enable interrupts; mustn't check for stack overflow). The rules for which handler gets called in this case are rather different from those of (uncaught) trap and (unhandled) event handlers, partly because the user PC is not available, and partly because it is not necessarily quick enough. So the FastEventProc of each language in the image is called in turn (in some random order).

UnwindProc

UnwindProc unwinds one stack frame (see description of `_kernel_unwindproc` for details). If no procedure is provided, the default unwind procedure assumes that the ARM Procedure Call Standard has been used; languages should provide a procedure if some internal calls do not follow the standard.

NameProc

NameProc returns a pointer to the string naming the procedure in whose body the argument PC lies, if a name can be found; otherwise, 0.

How the run-time stack is managed and extended

The run-time stack consists of a doubly-linked list of stack chunks. The initial stack chunk is created when the run-time kernel is initialised. Currently, the size of the initial chunk is 16Kb. Subsequent requests to extend the stack are rounded up to at least this size, so the granularity of chunking of the stack is fairly coarse. However, clients may not rely on this.

Each chunk implements a portion of a descending stack. Stack frames are singly linked via their frame pointer fields within (and between) chunks. See *Appendix C: ARM Procedure Call Standard* for more details.

In general, stack chunks are allocated by the storage manager of the master language (the language in which the root procedure – that containing the language entry point – is written). Whatever procedures were last registered with `_kernel_register_allocs()` will be used (each chunk 'remembers' the identity of the procedure to be called to free it). Thus, in a C program, stack chunks are allocated and freed using `malloc()` and `free()`.

In effect, the stack is allocated on the heap, which grows monotonically in increasing address order.

The use of stack chunks allows multiple threading and supports languages which have co-routine constructs (such as Modula-2). These constructs can be added to C fairly easily (provided you can manufacture a stack chunk and modify the `fp`, `sp` and `sl` fields of a `jmp_buf`, you can use `setjmp` and `longjmp` to do this).

Stack chunk format

A stack chunk is desribed by a `_kernel_stack_chunk` data structure located at its low-address end. It has the following format:

```
typedef struct stack_chunk {
    unsigned long sc_mark;       /* == 0xf60690ff */
    struct stack_chunk *sc_next, *sc_prev;
    unsigned long sc_size;
    int (*sc_deallocate)();
} _kernel_stack_chunk;
```

`sc_mark` is a magic number; `sc_next` and `sc_prev` are forward and backward pointers respectively, in the doubly linked list of chunks; `sc_size` is the size of the chunk in bytes and includes the size of the stack chunk data structure; `sc_deallocate` is a pointer to the procedure to call to free this stack chunk – often `free()` from the C library. Note that the chunk lists are terminated by NULL pointers – the lists are not circular.

The seven words above the stack chunk structure are reserved to Acorn. The stack-limit register points 512 bytes above this (ie 560 bytes above the base of the stack chunk).

Stack extension

Support for stack extension is provided in two forms:

- `fp`, arguments and `sp` get moved to the new chunk (Pascal/Modula-2-style)
- `fp` is left pointing at arguments in the old chunk, and `sp` is moved to the new chunk (C-style).

Each form has two variants depending on whether more than 4 arguments are passed (Pascal/Modula-2-style) or on whether the required new frame is bigger than 256 bytes or not (C-style). See *Appendix D: ARM Procedure Call Standard* for further details.

_kernel_stkovf_copyargs

Pascal/Modula-2-style stack extension, with some arguments on the stack (ie stack overflow in a procedure with more than four arguments). On entry, `ip` must contain the number of argument words on the stack.

_kernel_stkovf_copy0args

Pascal/Modula-2-style stack extension, without arguments on the stack (ie stack overflow in a procedure with four arguments or fewer).

_kernel_stkovf_split_frame

C-style stack extension, where the procedure detecting the overflow needs more than 256 bytes of stack frame. On entry, `ip` must contain the value of `sp` – the required frame size (ie the desired new `sp` which would be below the current stack limit).

_kernel_stkovf_split_0frame

C-style stack extension, where the procedure detecting the overflow needs 256 or fewer bytes of stack frame.

Stack chunks are deallocated on returning from procedures which caused stack extension, but with one chunk of latency. That is, one extra stack chunk is kept in hand beyond the current one, to reduce the expense of repeated call and return when the stack is near the end of a chunk; others are freed on return from the procedure which caused the extension.

Calling other programs from C

The C library procedure `system()` provides the means whereby a program can pass a command to the host system's command line interpreter. The semantics of this are undefined by the draft ANSI standard.

RISC OS distinguishes two kinds of commands, which we term *built-in commands* and *applications*. These have different effects. The former always return to their callers, and usually make no use of application workspace; the latter return to the previously set-up 'exit handler', and may use the currently-available application workspace. Because of these differences, `system()` exhibits three kinds of behaviour. This is explained below.

Applications in RISC OS are loaded at a fixed address specified by the application image. Normally, this is the base of application workspace, 0x8000. While executing, applications are free to use store between the base and end of application workspace. The end is the value returned by SWI OS_GetEnv. They terminate with a call of SWI OS_Exit, which transfers control to the current exit handle.

When a C program makes the call `system("command")` several things are done:

- The calling program and its data are copied to the top end of application workspace and all its handlers are removed.
- The current end of application workspace is set to just below the copied program and an exit handler is installed in case `"command"` is another application.
- `"command"` is invoked using SWI OS_Cli.

When `"command"` returns, either directly (if it is a built-in command) or via the exit handler (if it is an application), the caller is copied back to its original location, its handlers are re-installed and it continues, oblivious of the interruption.

The value returned by `system()` indicates

- whether the command or application was successfully invoked
- if the command is an application which obeys certain conventions, whether or not it ran successfully.

The value returned by `system` (with a non-NULL command string) is as follows:

< 0 – couldn't invoke the command or application (eg command not found);

>=0 – invoked OK and set Sys$ReturnCode to the returned value.

By convention, applications set the environmental variable Sys$ReturnCode to 0 to indicate success and to something non-0 to indicate some degree of failure. Applications written in C do this for you, using the value passed as an argument to the `exit()` function or returned from the `main()` function.

If it is necessary to replace the current application by another, use:

```
system("CHAIN:command");
```

If the first characters of the string passed to `system()` are `"CHAIN:"` or `"chain:"`, the caller is not copied to the top end of application workspace, no exit handler is installed, and there can be no return (return from a built-in command is caught by the C library and turned into a SWI OS_Exit).

Typically, `CHAIN:` is used to give more memory to the called application when no return from it is required. The C compiler invokes the linker this way if a link step is required. On the other hand, the Acorn Make Utility (AMU) calls each command to be executed. Such commands include the C compiler (as both use the shared C library, the additional use of memory is minimised). Of course, a called application can call other applications using `system()`. A callee can even `CHAIN:` to another application and still, eventually, return to the caller. For example, AMU might execute:

```
system("cc hello.c");
```

to call the C compiler. In turn, `cc` executes:

```
system("CHAIN:link -o hello o.hello $.CLib.o.Stubs");
```

to transfer control to the linker, giving link all the memory cc had.

However, when Link terminates (calls `exit()`, returns from `main()` or aborts) it returns to AMU, which continues (providing Sys$ReturnCode is good).

The shared C library

Release 3 of C makes extensive use of the shared C library module, first introduced with Release 2 of C and subsequently used by the RISC OS applications Edit, Paint, Draw and Configure.

The shared C library is a RISC OS relocatable module (called SharedCLibrary) which contains the whole of the ANSI C library. Once installed in your computer it can be used by every program written in C. Consequently, it save both RAM space and disc space.

In fact, this is as much as you really need to know about the shared C library and probably as far as you should delve at first reading. So, if you are eager to try your first practical work with this release of C, skip the rest of this section. However, if you are curious and would like to know more about what it really costs to use it, its benefits, and a little of how it works, then read on.

Costs involved in using the shared C library

The SharedCLibrary modules occupies about 61Kb. Each program that uses it must be linked with the *library stubs*, a small object module containing space for a copy of the shared C library's data and an entry vector via which functions in the shared library can be called. The stubs occupy just 5Kb. Thus a single program linked with the shared C library consumes about 65Kb of RAM for C library. However, two programs in memory at the same time use only 70Kb for library and three programs, only 75Kb.

In contrast, a program linked with Release 3 of AnsiLib will include a minimum of 40Kb. So, as soon as you have two or more C programs in memory at the same time, it is cheaper to use the SharedCLibrary. Usually, you will have Edit resident (which uses the shared C library anyway) and then you may want to run `cc` under AMU. In this situation, use of the shared C library saves 45Kb of RAM.

Efficient use of RAM is not the only consideration. The C compiler includes 48Kb of AnsiLib and when squeezed occupies 172Kb on disc. However, when linked with Stubs and squeezed it occupies only 140Kb. There are similar savings from Link, AMU, ASD, and Squeeze, as well as for the programs you compile (the 'hello world' program is reduced in size from over 40Kb to just 5.5Kb).

Without using the shared C library it would not be possible to use C Release 3 on a system with only a single floppy disc drive (imagine the loss of 150Kb of work space, together with a minimum image size of 40Kb). And, of course, smaller programs load much faster from a floppy disc.

If you have a larger Acorn system, use of the shared C library still brings benefits:

- Small programs load noticeably faster, even from a hard disc.
- No hard disc is ever big enough; saving 25-40Kb per program is not to be sneezed at if you have 40 or 50 programs (1-2Mb saved).
- Much more can be packed into the RAMFS – perhaps all the tools you ever use, giving almost instantaneous loading of them.

Execution time costs

It costs only 4 cycles (0.5µs) per function call and a very small penalty on access to the library's static data by the library (the user program's access to the same data is unpenalised). In general, the difference in performance between using the shared C library and linking a program stand-alone with AnsiLib is less than 1%. For the important Dhrystone-2.1 benchmark the performance difference cannot be measured (you can try this experiment for yourself using the sources provided in `$.User.c`).

How it works

The shared C library module implements a single SWI which is called by code in the library stubs when your program linked with the stubs starts running. That SWI call tells the stubs where the library is in the machine. This allows the vector of library entry points contained in the stubs to be patched up in order to point at the relevant entry points in the library module.

The stubs also contain your private copy of the library's static data. When code in the library executes on your behalf, it does so using your stack and relocates its accesses to its static data by a value stored in your stack-chunk structure by the stubs initialisation code and addressed via the stack-limit register (this is why you must preserve the stack-limit register everywhere if you use the shared C library and call your own assembly language sub-routines). The compiler's register allocation strategy ensures that the real dynamic cost of the relocation is almost always low: for example, by doing it once outside a loop that uses it many times.

If you go on to write your own relocatable modules in C, you'll use the `-zM` feature of the compiler which causes similar code to be generated.

`#pragma` directives

Pragmas recognised by the compiler come in two forms:

```
#pragma -<letter><optional-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 nocheck_stack

#pragma -p2
#pragma profile_statements
```

The current list of recognised pragmas is:

pragma name	short form	'no' form
`warn_implicit_fn_decls`	`a1`	`a0`
`warn_implicit_casts`	`b1`	`b0`
`check_memory_accesses`	`c1`	`c0`
`warn_deprecated`	`d1`	`d0`
`continue_after_hash_error`	`e1`	`e0`
`optimise_crossjump`	`j1`	`j0`
`optimise_multiple_loads`	`m1`	`m0`
`profile`	`p1`	`p0`
`profile_statements`	`p2`	`p0`
`check_stack`	`s0`	`s1`
`check_printf_formats`	`v1`	`v0`
`check_scanf_formats`	`v2`	`v0`
`check_formats`	`v3`	`v0`
`side_effects`	`y0`	`y1`
`optimise_cse`	`z1`	`z0`

The set of pragmas recognised by the compiler, together with their default settings, varies from release to release of the compiler. In general, the only pragmas you should need to use are `check_stack` and `nocheck_stack`. These enable and disable, respectively, the generation of code to check the stack limit on function entry and exit. In reality there is little advantage to turning stack checks off: they cost at most two instructions and two machine cycles (about 0.25µs) per function call. The one occasion when `nocheck_stack` would be used is in writing a signal handler for the `SIGSTAK` 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.

Storage management (malloc, calloc, free)

The aim of the storage manager is to manage the heap in as 'efficient' a manner as possible. However, 'efficient' does not mean the same to all programs and since most programs differ in their storage requirements, certain compromises have to be made. The main two issues to be considered are *speed* and *heap fragmentation*.

You should also try to keep the peak amount of heap used to a minimum so that, for example, a C program may invoke another C program leaving it the maximum amount of memory. This implementation has been tuned to hold the overhead due to fragmentation to less than 50%, with a fast turnover of small blocks.

The heap can be used in many different ways. For example it may be used to hold data with a long life (persistent data structures) or as temporary work space; it may be used to hold many small blocks of data or a few large ones or even a combination of all of these allocated in a disorderly manner. The storage manager attempts to address all of these problems but like any storage manager, it cannot succeed with all storage allocation/deallocation patterns. If your program is unexpectedly running out of storage, using the following information on the storage manager's stategy for managing the heap may help you to remedy it.

Note the following:

- The word *heap* refers to the section of memory currently under the control of the storage manager.
- All block sizes are in bytes and are rounded up to a multiple of four bytes.
- All blocks returned to the user are word-aligned.
- All blocks have an overhead of eight bytes (two words). One word is used to hold the block's length and status, the other contains a guard constant which is used to detect heap corruptions. The guard word may not be present in future releases of the ANSI C library.

Allocation of blocks

When an allocation request is received by the storage manager, it is categorised into one of three sizes of blocks; small, medium or large (0 < small <= 64 < medium <= 512 < large < 16777216).

The storage manager keeps track of the free sections of the heap in two ways. The medium and large sized blocks are chained together into a linked list (overflow list) and small blocks of the same size are chained together into linked lists (bins). The overflow list is ordered by ascending block address, while the bins have the most recently freed block at the start of the list. When a small block is requested, the bin which contains the blocks of the required size is checked, and if the bin is not empty, the first block in the list is returned to the user. If there was no block of the exact size available, the bin containing blocks of the next size up is checked, and so on, until a block is found. If a block is not found in the bins, the last block (highest address) on the overflow list is taken. If the block is large enough to be split into two blocks, and the remainder a usable size (> 12 including the overhead), the block is split, the top section returned to the user and the remainder, depending on its size, is either put in the relevant bin at the front of the list or left in the overflow list.

The allocation of medium blocks is the same as for small blocks, except that the search for a block ignores the bins and starts with the overflow list which is searched in reverse order for a block of usable size.

When a large block is requested, the overflow list is searched in increasing address order and the first block in the list which is large enough is taken. If the block is large enough to be split into two blocks, and the size of the remainder is larger than a small block (> 64) then the block is split, the top section is returned to the overflow list and bottom section given to the user.

Failure to allocate a block immediately

If there is no block of the right size available, the storage manager has two options:

- Take all the free blocks on the heap and join adjacent free blocks together (coalescing), in the hope that a block of the right size will be created which can then be used.
- Ask the operating system for more heap. The block returned is put on the overflow list and allocation of the user block continues as above.

The heap will only be coalesced if there is enough free memory in it to make it worthwhile, or if the request for more heap was denied. Coalescing causes the bins and overflow list to be emptied, the heap to be scanned, adjacent free blocks coalesced, and the free blocks scattered into bins and overflow list in increasing address order.

Deallocation of blocks

When a block is freed, if it will fit in a bin, it is put at the start of the relevant bin list. Otherwise, it is just marked as being free and effectively taken out of the heap until the next coalesce phase, when it will be put in the overflow list. This is done because the overflow list is in ascending block address order and it would have to be scanned so that the freed block could be inserted at the correct position. Surprisingly, fragmentation is also reduced if the block is not reusable until after the next coalesce phase.

Handling host errors

Calls to RISC OS can be made via one of the functions in the C header file `kernel.h`, (such as `_kernel_osfind(64, ".....")`). If the call causes an operating system error, the function will return the value `_kernel_ERROR`. To find out what the error was, a call to `_kernel_last_oserror` should be made. This will return a pointer to a `_kernel_oserror` block containing the error number and any associated error string. If there has been no error since `_kernel_last_oserror` was last called, the function returns the NULL pointer. Some functions in the ANSI C library call `_kernel` functions, so if an ANSI C library function (such as `fopen(".....", "r")`) fails, try calling `_kernel_last_oserror` to find out what the error was.

For more details about operating system calls, refer to the `kernel.h` header (reproduced as Appendix E in this Guide), and for more information about RISC OS error handling, refer to the chapter entitled *Generating and handling errors* in the *RISC OS Programmer's Reference Manual.*

Appendix A: New features of Release 3

Release 3 of the C compiler product is a powerful and effective vehicle for developing software for the RISC OS operating system and incorporates many more features than the previous release, Release 2. The scope of the Guide has accordingly been extended and much use made of worked examples provided on disc as well as in the text. Particular attention is given to machine- and operating system-specific features.

The key additional features of Release 3 are:

- conformity with the latest ANSI draft (December 1988)
- RISC OS library extensions
- support for developing the following types of program for RISC OS:
    - Desktop applications
    - Relocatable modules
    - Overlaid applications
- improved portability to and from RISC OS
- enhanced and new software tools, previously part of the Software Developer's Toolbox.

Further details are given below in the sections *Additional Software* and *New Features of the Guide*.

All known bugs in the compiler system have been fixed, and the performance of the compiler in terms of speed and size of compiled code has been improved, typically by a few percent, though some operations such as integer divide have been speeded up sufficiently to make a 40% overall difference to an arithmetic encoding program.

Further details are given in the Release Note supplied in the release package.

The Procedure Call Standard has been revised since Release 2 of the Compiler, and this is covered in the section *New Procedure Call Standard* and in Appendix D.

Support for the Brazil operating system, which was developed for prototype ARM-based machines, has been dropped, so the Brazil library, Superlib, and its header are not included.

Additional software

Additional software for Release 3 consists of:

- existing utilities incorporated in Release 3
- new utilities
- examples.

Existing utilities incorporated in Release 3

The following utility programs are part of the Software Developer's Toolbox, and were not part of Release 2 of the C product. Upgraded versions of these utilities are included in C Release 3:

- AMU – the Acorn 'make' utility

  Details of the enhanced features of AMU are given in the chapter entitled *Other Utilities*.

- ASD – the Acorn Source-level Debugger

  The functionality of the debugger has been extended to include support for debugging at the assembly language level, including the facility for inspecting register contents and blocks of memory. ASD can therefore now be used to debug high-level language programs at the source code level or at the machine code level, as well as to debug programs written in assembler.

  A complete worked example to illustrate use of the debugger is supplied in the the chapter on the debugger and on disc (directory `AsdDemo` on Disc 1).

- Squeeze – an image file compaction utility.

New utilities

The following utility programs have not previously been released as part of an Acorn product, and are included in C Release 3:

- FormEd

  This takes much of the hard work out of preparing templates (icons, dialogue boxes, menus etc) for the Window Manager environment. It can be found in `$.!FormEd` on Disc 3.

- Conversion utilities – `toansi` and `topcc`

  `toansi` converts pcc-style source to ANSI-style source. `topcc` converts ANSI-style source to pcc-style source. The executable images for these utilities are in `$.Library` on Disc 1, and their source files are in `$.Conversion` on Disc 2.

- `cmhg` – the C Relocatable Module Header Generator

  `cmhg` can be found in `$.Library` on Disc 1. It is a special-purpose module header assembler for modules written in C. It is described in the chapter entitled *How to write relocatable modules in C*.

Examples

There were four example programs provided in Release 2:

`HelloW`	simple 'Hello World' example
`Sieve`	the sieve of Eratosthenes
`Balls64`	colourful graphics demonstration
`HowToCall`	illustrating calling other programs from C.

`Balls64` has been used as the starting point for one of the Desktop Application illustrations, and `HowToCall` has been modified in line with changes in the way other programs are called from C. Many more example programs have been included with Release 3 of the compiler, the complete list being:

File	Description	Directory	Disc
`HelloW`	Simple 'Hello World' example.	`$.User`	1
`Sieve`	The sieve of Eratosthenes.	`$.User`	1

`CModule`
`CmoduleHdr`
`MakCModule`
	Example Relocatable Module
	in C with header file
	and 'make' file.
		`$.User`	1

`HowToCall`	Illustrates calling other programs from C.	`$.User`	1
`swi_list`	Generates a list of SWI names.	`$.User`	1
`AsdDemo`	ASD debugger demonstration.	`$.AsdDemo`	1
`Dhrystone`	Source for the Dhrystone 2.1 benchmark.	`$.Dhrystone`	1
`OverEx`	Example of use of overlays.	`$.OverEx`	3
	Examples to illustrate features of Desktop Applications:	`$.DeskEgs`	1
`!Balls64`	colourful graphic display;		
`!DrawEx`	example which includes rendering a Draw file;		
`!Life`	runs Conway's game of life;		
`!WExample`	example developed in thismanual.		

Upgrades

The following software elements were provided in Release 2, and have been upgraded for Release 3:

- `fpe280` Floating point emulator
  For details, see *Appendix F: the Floating Point Emulator*.
- `link` Linker
  Link has been re-implemented; it is now smaller and faster and provides support for overlays.

New features of the Guide

Part 1: Using the C compiler and tools

The additional material in the Guide reflects the extra software functionality supplied in the product:

- *How to install and run the compiler*

This section has been extensively revised, and has new sections to cover setting up your working environment and an overview of the C compiler system.

- *Using the linker*

This has been revised to accommodate the changes to the linker command line interface, and the new functionality supporting overlays.

- *Acorn Source-level Debugger*

The content of the *ASD Guide* from the Software Developer's Toolbox has been revised to incorporate ASD's new features and to cover the extended example ASD session.

- *Other utilities*

The coverage of AMU and Squeeze has been revised to incorporate their enhancements.

Part 2: Language issues

The following chapters have been revised to address the changes to the ANSI standard.

- *Implementation details*
- *Standard implementation definition.*

The following chapters are new:

- *Portability*
- *ANSI library reference section.*

Part 3: Developing software for RISC OS

A new section of the manual, with chapters to cover:

- *How to write desktop applications in C*
- *How to use the template editor*
- *RISC OS library reference section*
- *Assembly language interface*
- *How to write relocatable modules in C*
- *Overlays*
- *Machine-specific features.*

Part 4: Appendices

The following appendices are new to the manual:

- *New features of Release 3*
- *ARM procedure call standard*
- `kernel.h` (the low-level interface to RISC OS)

- *The floating point emulator*.

The appendix covering the Arthur Operating System library (ArthurLib) no longer lists the functions: for details of these, refer to the header files on Disc 3.

The appendix covering Errors and Warnings has been revised and updated.

Changes to the compiler

These are listed in the Release Note supplied with the release package.

New Procedure Call Standard

C Release 3 conforms to the new ARM Procedure Call Standard. Full technical details are given in *Appendix D: ARM Procedure Call Standard*, but essentially the changes have been made to support the writing of modules in C for better commonality with other Acorn products.

The changes made have maintained the backwards compatibility of the shared C library. Thus old software (compiled before Release 3, using the old Procedure Call Standard) will run with the new shared C library but software compiled with Release 3 will not run with the old shared library.

To maintain standalone compatibility, two binaries of the stand-alone C library (AnsiLib) and the Arthur operating system library (ArthurLib) have been provided, the files with the suffix `_A` conforming to the old standard. However, this is an interim measure. ArthurLib is now obsolete, being replaced by RISC_OSlib, and support for the old Procedure Call Standard will be dropped from the next release of C. You are therefore advised to recompile all existing application sources with the new compiler, and revise any software that needs to observe the Procedure Call Standard so that it is in line with the new standard.

Appendix B: Arthur Operating System library

Using the Arthur libraries

Under RISC OS, the Arthur Operating System library is obsolescent, and will not be supported in the future. It is included in this release only for compatibility with Release 1 and Release 2 C. A C program should interact with RISC OS via the RISC OS library (RISC_OSlib), as used by Edit, Paint, Draw and other applications included in the RISC OS Applications Suite. The chapter entitled *RISC OS library reference section* gives full details.

Low-level access to RISC OS is now provided via the language-independent C library kernel. The *RISC OS library reference section* also gives information on this; alternatively, you can refer to the header file `kernel.h` listed in Appendix E and provided on Disc 3 of this release. As the library kernel is part of the shared C library and part of every C program linked with AnsiLib, it is the preferred interface for occasional low-level access to the operating system.

Two variants of the Arthur library are provided (in `$.clib.o` on Disc 2):

- `Arthurlib` conforms to the new ARM procedure call standard
- `ArthLib_A` conforms to the old ARM procedure call standard.

The standard is covered in Appendix D of this Guide.

To use Arthurlib functions, their declarations must be inserted in your code by means of a `#include` line. As an example, here is the 'hello world' program again, here using the `mode()` function to change screen mode:

```
#include <stdio.h>
#include <Arthur.h>
int main()
{
      art_mode(7);
```

```
        printf("Hello world!\n");
        return 0;
}
```

When the above program is compiled and linked, the -arthur option has to be used:

```
cc -arthur hello
```

This causes the linker to use the arthurlib library in addition to the usual ansilib one.

Because it is quite possible for the names of the constants, functions and variables declared in h.arthur to clash with other identifiers used in a program, names are long and are all prefixed by art_. Constants relating to Wimp and sound functions are prefixed by Wimp_ and Sound_ respectively. An example is art_mode(), illustrated above. Often, a function name is the same as the equivalent BBC BASIC keyword, written in lower case and prefixed by art_, as in art_mode(), art_vdu(), and art_clg(), etc.

If the macro symbol ARTHUR_OLD_NAMES is defined before the Arthur header file is included, then all the names documented in this chapter can be used without their art_ prefixes. Here is yet another version of the hello program using this method:

```
#define ARTHUR_OLD_NAMES
#include <stdio.h>
#include <Arthur.h>
int main()
{
        mode(7);
        printf("Hello world!\n");
        return 0;
}
```

An alternative way of achieving the same effect as the #define line above would be to use the compiler command line option -DARTHUR_OLD_NAMES.

General ArthurLib functions

These functions deal with general I/O features of Arthur, including graphics, sound and keyboard. In general their functionality emulates that of similarly named BASIC keywords. Brief descriptions are given below, but you are recommended to refer to the *BBC BASIC Guide* for comprehensive descriptions.

Functions such as `art_osfile()` are essentially those described in detail in the *RISC OS Programmer's Reference Manual*. Any C structures referred to are defined in the Arthur header file `<Arthur.h>` (ie `$.clib.h.Arthur`). In the function declarations, the ANSI prototype facility to give names as well as types to arguments is used. This makes the arguments' use a little more self-explanatory.

On the working disc (Disc 1 of the release), headers are given in compressed form to save space. Comments and argument names are omitted for brevity. On the documentation disc (Disc 3), headers are given in full. You should therefore consult the file `$.clib.h.Arthur` on Disc 3.

Appendix C: Errors and warnings

Levels of errors and warnings

The compiler can produce error or warning messages of several degrees of severity. They are:

- Warnings indicating curious, but legal, program constructs, or constructs that are indicative of potential error.
- Non-serious errors which still allow code to be produced.
- Serious errors which may produce loss of code.
- Fatal errors which stop the compiler from compiling.
- System errors which signal faults in the compiler itself.

Errors and serious errors collectively correspond to ANSI 'diagnostics'; whether an error is serious or not reflects the compiler's view, not that of the user or the ANSI committee.

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 part. 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 a bad return code set.

Future releases of the compiler may distinguish further errors or produce slightly different forms of wording.

In pcc mode, constructs that are erroneous in ANSI mode are warned of, even though legal in pcc mode.

The messages are listed alphabetically in each section.

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

#define macro '*xx*' defined but not used

'&' 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 trigraphs. Example:

```
printf("Type ??/!!: "); /* "??/" is trigraph for "\" */
```

argument and old-style parameter mismatch : *xx*

A function with a non-ANSI 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 mis-matched your `if`s and `else`s. Remember an `else` always refers to the most recent un-matched `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 ANSI draft 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 monocase)

Using compiler option -fe, 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 -fh, an external object was discovered which was not declared in any included header file.

floating point overflow when folding

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 `-fv`, 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 = l | 3;
i = l & ~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> */
```

label *xx* was defined but not used

Example:

```
errlab: exit(-1); /* no corresponding goto errlab */
```

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;}
```

(which fails if `int` has 16 bits) is moved to machines such as the IBM PC.

`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 `-fx`, an `enum` declaration was found, and this message refers to the ANSI 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 draft 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 `-fo`, it was noted that the code contains a non-ANSI 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 mesg[14] = "(C)1988 Acorn\n";/* needs 15 */
```

repeated definition of #define macro xx

When compiling with `fh`, 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

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.

undefined macro '*xx*' in #if - treated as 0

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 option `-fa`, an automatic variable is found to have been used before any value has been assigned to it.

***xx* treated as *xx*ul 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 C.

ancient form of initialisation, use '='

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.

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";
```

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.

digit 8 or 9 found in octal number

Octal (base 8) numbers may only have digits up to 7. Example:

```
static int i = 0178; /* probably meant 0177, ie 0xff */
```

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 */
```

illegal string escape *'\x'* - treated as x

Unrecognised string escape (\ followed by a character) found. The \ is ignored. Example:

```
      printf("\w"); /* no such escape */
```

<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*' - 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 */
{
	...
```

missing newline before EOF - inserted

The last line of the source file did not have its terminating end of line character.

more than 4 chars in '...'

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 '\xfff';  /* \xfff' 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 ANSI 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;
}
```

'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*/
```

size of function required - treated as size of pointer

The compiler cannot know the size of a function at compile time, so instead it uses the size of a `(*)()`. Example:

```
extern int func(void);
int main(void)
{
	int i = sizeof(func);
	...
```

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 of any `#if` or `#ifdef` scope. Example:

```
#if defined sym
...
#endif
#else /* this one is spurious */
...
```

static function *xx* not defined - treated as extern

A prototype declares the function to be static, but the function itself is absent from this compilation unit.

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);*/
   ...
```

wrong number of parameters to '*xx*'

The function *xx* was called with the wrong number of parameters, as declared by its protype. Example:

```
size_t strlen(const char *s);
...
{
      int i = strlen(str,j); /* only str needed */
```

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.

#error encountered "*xx*"

Source intentionally producing an error with a `#error` directive. The compiler stops immediately, unless `#pragma -e` is set. Example:

```
#if CHAR_BIT != 8
#error This program needs eight-bit characters
#endif
```

#include file "*xx*" wouldn't open
#include file <*xx*> wouldn't open

Probably caused by a spelling mistake in the file name. Example:

```
#include <stdef.h> /* missed out a 'd' */
```

'...' 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;
```

\\<space> and \\<tab> are invalid string escapes

Use `<space>` and `\t` respectively for these characters in strings and character constants. Example:

```
      printf("\ Next?"); /* No need for \ */
```

{} 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 `0xffffff`. Example:

```
static char dict[0x1000000]; /* 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();
	}
```

Digit required after exponent marker

A syntax error in a floating point constant was found. Example:

```
	a = b*1.1e; /* need [+/-]digits here */
```

duplicated 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 definition of '*xx*'
duplicate definition of label *xx* -ignored

These both refer to various types of duplicated definition. Examples:

```
static int i;
void fn(void)
{
        lab:
        ...
        lab:  /* redefinition of lab */
}
char i;       /* redefinition of i */
int fn()      /* redefinition of fn() */
{
        ...
}
```

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 >> */
```

'goto' not followed by label - ignored

Self explanatory.

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

Hexdecimal 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 ANSI-conforming implementations. Example:

```
struct s { char a : 4; char b : 4;};
```

illegal character (0x%lx = 'xx') in source
illegal character (hex code 0x%x) in source

(as for above but applies to unprintable characters). Example:

```
char @str = "string";          /* should be char *str */
```

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 mis-spelling 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 fuction 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++;}
```

`newline or end of file within string`

`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; /*0x80000000 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 namespace 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*' has no xx field

The field name used with a `.` or → operator is not a valid one for the union or structure type '*xx*' being referenced. Example:

```
struct s {int a,b;};
...
{
      struct s s1;
      s1.c = 3;/* no c field */
      ...
```

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 ANSI 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 preprocessor directives. Example:

```
#asm         /* not an ANSI 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.

`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.

`no store left`
`out of store (in cc_alloc)`

The compiler has run out of memory – either shorten your source programs or free some RAM using the `*UNPLUG` command. To do this, first check which modules are present in your machine by typing `*MODULES`. If there is a module that you do not currently need, you can release its space by typing

```
*UNPLUG modulename
*RMTidy
```

It can later be restored using the `*RMREINIT` command. For further details, refer to the chapter entitled *Modules* in the *Programmer's Reference Manual*, (second edition).

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.

`too many file names`

An attempt was made to compile too many files at once. 25 is the maximum that will be accepted.

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 your Acorn supplier.

These error messages all look like this:

```
************************************************************************
* The compiler has detected an internal inconsistency. This can occur  *
* because it has run out of a vital resource such as memory or disk    *
* space or because there is a fault in it. If you cannot easily alter  *
* your program to avoid causing this rare failure, please contact your *
* Acorn dealer. The dealer may be able to help you immediately and will *
* be able to report a suspected compiler fault to Acorn Computers.     *
************************************************************************
```

Appendix D: ARM Procedure Call Standard

Introduction

This Appendix relates to the implementation of compiler code-generators and language run-time library kernels for the Acorn RISC Machine (ARM).

The reader should be familiar with the ARM's instruction set, floating point instruction set and assembler syntax before attempting to use this information to implement a code-generator. In order to write a run-time kernel for a language implementation, additional information specific to the relevant ARM operating system will be needed (some information is given in the sections describing the standard register bindings for this procedure-call standard).

The main topics covered in this Appendix are the procedure call and stack disciplines. These disciplines are observed by Acorn's C language implementation for the ARM and, eventually, will be observed by the Fortran and Pascal compilers too. Because C is the first-choice implementation language for RISC OS applications and the implementation language of Acorn's UNIX product RISC iX, the utility of a new language implementation for the ARM will be related to its compatibility with Acorn's implementation of C.

At the end of this document are several examples of the usage of this standard, together with suggestions for generating effective code for the ARM.

The purpose of APCS

The ARM Procedure Call Standard is a set of rules, designed:

- to facilitate calls between program fragments compiled from different source languages (eg to make subroutine libraries accessible to all compiled languages)
- to give compilers a chance to optimise procedure call, procedure entry and procedure exit (following the reduced instruction set philosophy of the ARM). This standard defines the use of registers, the passing of

arguments at an external procedure call, and the format of a data structure that can be used by stack backtracing programs to reconstruct a sequence of outstanding calls. It does so in terms of *abstract register names*. The binding of some register names to register numbers and the precise meaning of some aspects of the standard are somewhat dependent on the host operating system and are described in separate sections.

Formally, this standard only defines what happens when an external procedure call occurs. Language implementors may choose to use other mechanisms for internal calls and are not required to follow the register conventions described in this document except at the instant of an external call or return. However, other system-specific invariants may have to be maintained if it is required, for example, to deliver reliably an asynchronous interrupt (eg a `SIGINT`) or give a stack backtrace upon an abort (eg when dereferencing an invalid pointer). More is said on this subject in later sections.

Design criteria

This procedure call standard was defined after a great deal of experimentation, measurement, and study of other architectures. It is believed to be the best compromise between the following important requirements:

- Procedure call must be extremely fast.
- The call sequence must be as compact as possible. (In typical compiled code, calls outnumber entries by a factor in the range 2:1 to 5:1.)
- Extensible stacks and multiple stacks must be accommodated. (The standard permits a stack to be extended in a non-contiguous manner, in stack chunks. The size of the stack does not have to be fixed when it is created, avoiding a fixed partition of the available data space between stack and heap. The same mechanism supports multiple stacks for multiple threads of control.)
- The standard should encourage the production of re-entrant programs, with writeable data separated from code.
- The standard must support variation of the procedure call sequence, other than by conventional return from procedure (eg in support of C's `longjmp`, Pascal's `goto-out-of-block`, Modula-2+'s exceptions, UNIX's signals, etc) and tracing of the stack by debuggers and run-time error handlers. Enough is defined about the stack's structure to ensure that implementations of these are possible (within limits discussed later).

The Procedure Call Standard

Register names

The ARM has 16 visible general registers and 8 floating-point registers. In interrupt modes some general registers are shadowed and not all floating-point operations are available, depending on how the floating-point operations are implemented.

This standard is written in terms of the register names defined in this section. The binding of certain register names (the 'call frame registers') to register numbers is discussed separately. We do this so that:

- Diverse needs can be more easily accommodated, as can conflicting historical usage of register numbers, yet the underlying structure of the procedure call standard – on which compilers depend critically – remains fixed.
- Run-time support code written in assembly language can be made portable between different register bindings, if it obeys the rules given in the section entitled *Defined bindings of the procedure call standard.*

The register names and fixed bindings are given immediately below.

General Registers

First, the four argument registers:

```
a1   RN   0   ; argument 1/integer result
a2   RN   1   ; argument 2
a3   RN   2   ; argument 3
a4   RN   3   ; argument 4
```

Then the six 'variable' registers:

```
v1   RN   4   ; register variable
v2   RN   5   ; register variable
v3   RN   6   ; register variable
v4   RN   7   ; register variable
v5   RN   8   ; register variable
v6   RN   9   ; register variable
```

Then the call-frame registers, the bindings of which vary (see the section on register bindings for details):

```
sl            ; stack limit / stack chunk handle
fp            ; frame pointer
ip            ; temporary workspace, used in
                procedure entry
sp   RN   13 ; lower end of current stack frame
```

Finally, `lr` and `pc`, which are determined by the ARM's hardware:

```
lr   RN   14 ; link address on calls/temporary workspace
pc   RN   15 ; program counter and processor status
```

In the obsolete APCS-A register bindings described below, `sp` is bound to `r12`; in all other APCS bindings, `sp` is bound to `r13`.

Notes

Literal register names are given in lower case, eg `v1`, `sp`, `lr`. In the text that follows, symbolic values denoting 'some register' or 'some offset' are given in upper case, eg `R`, `R+N`.

References to 'the stack' denoted by `sp` assume a stack that grows from high memory to low memory, with `sp` pointing at the top or front (ie lowest addressed word) of the stack.

At the instant of an external procedure call there must be nothing of value to the caller stored below the current stack pointer, between `sp` and the (possibly implicit, possibly explicit) stack (chunk) limit. Whether there is a single stack chunk or multiple chunks, an explicit stack limit (in `sl`) or an implicit stack limit, is determined by the register bindings and conventions of the target operating system.

Here and in the text that follows, for any register `R`, the phrase 'in `R`' refers to the contents of `R`; the phrase 'at `[R]`' or 'at `[R, #N]`' refers to the word pointed at by `R` or `R+N`, in line with ARM assembly language notation.

Floating Point Registers

The floating point registers are divided into two sets, analogous to the subsets `a1-a4` and `v1-v6` of the general registers. Registers `f0-f3` need not be preserved by a called procedure; `f0` is used as the floating-point result register. In certain restricted circumstances (noted below), f0-f3 may be used to hold the first four floating-point arguments. Registers f4-f7, the so called 'variable' registers, must be preserved by callees.

The floating-point registers are:

```
f0      FN      0       ; floating point result (or 1st FP argument)
f1      FN      1       ; floating point scratch register (or 2nd FP arg)
f2      FN      2       ; floating point scratch register (or 3rd FP arg)
f3      FN      3       ; floating point scratch register (or 4th FP arg)
f4      FN      4       ; floating point preserved register
f5      FN      5       ; floating point preserved register
f6      FN      6       ; floating point preserved register
f7      FN      7       ; floating point preserved register
```

Data representation and argument passing

The ARM Procedure Call Standard is defined in terms of N (>= 0) word-sized arguments being passed from the caller to the callee, and a single word or floating point result passed back by the callee. The standard does not describe the layout in store of records, arrays and so forth, used by ARM-targeted compilers for C, Pascal, Fortran-77, and so on. In other words, the mapping from language-level objects to APCS words is defined by each language's implementation, not by APCS, and, indeed, there is no formal reason why two implementations of, say, Pascal for the ARM should not use different mappings and, hence, not be cross-callable.

Obviously, it would be very unhelpful for a language implementor to stand by this formal position and implementors are strongly encouraged to adopt not just the letter of APCS but also the obviously natural mappings of source language objects into argument words. Strong hints are given about this in later sections which discuss (some) language specifics.

Register usage and argument passing to external procedures

Control Arrival

We consider the passing of N (>= 0) actual argument words to a procedure which expects to receive either exactly N argument words or a variable number V (>= 1) of argument words (it is assumed that there is at least one argument word which indicates in a language-implementation-dependent manner how many actual argument words there are: for example, by using a format string argument, a count argument, or an argument-list terminator).

At the instant when control arrives at the target procedure, the following shall be true (for any M, if a statement is made about argM, and M > N, the statement can be ignored):

```
arg1 is in a1
arg2 is in a2
arg3 is in a3
arg4 is in a4
for all I >= 5, argI is at [sp, #4*(I-5)]
```

`fp` contains 0 or points to a stack backtrace structure (as described in the next section).

The values in `sp`, `sl`, `fp` are all multiples of four.

`lr` contains the `pc+psw` value that should be restored into `r15` on exit from the procedure. This is known as the *return link value* for this procedure call.

`pc` contains the entry address of the target procedure.

Now, let us call the lower limit to which sp may point *in this stack chunk* `SP_LWM` (Stack-Pointer Low Water Mark). Remember, it is unspecified whether there is one stack chunk or many, and whether `SP_LWM` is implicit, or explicitly derived from `sl`; these are binding-specific details. Then:

> Space between `sp` and `SP_LWM` shall be (or shall be on demand) readable, writeable memory which can be used by the called procedure as temporary workspace and overwritten with any values before the procedure returns.

`sp >= SP_LWM + 256.`

This condition guarantees that a stack extension procedure, if used, will have a reasonable amount – 256 bytes – of work space available to it, probably sufficient to call two or three procedure invocations further.

Control Return

At the instant when the return link value for a procedure call is placed in the `pc+psw`, the following statements shall be true:

`fp`, `sp`, `sl`, `v1-v6`, and `f4-f7` shall contain the same values as they did at the instant of the call. If the procedure returns a word-sized result, `R`, which is not a floating point value, then `R` shall be in `a1`. If the procedure returns a floating point result, `FPR`, then `FPR` shall be in `f0`.

Notes

The definition of control return means that this is a 'callee saves' standard.

The requirement to pass a variable number of arguments to a procedure (as in old-style C) precludes the passing of floating point arguments in floating point registers (as the ARM's fixed point registers are disjoint from its floating point registers). However, if a callee is defined to accept a fixed number K of arguments and its interface description declares it to accept exactly K arguments of matching types, then it is permissible to pass the first four floating point arguments in floating point registers `f0-f3`. However, Acorn's C compiler for the ARM does not yet exploit this latitude.

The values of `a2-a4`, `ip`, `lr` and `f1-f3` are not defined at the instant of return.

The `Z`, `N`, `C` and `V` flags are set from the corresponding bits in the return link value on procedure return. For procedures called using a `BL` instruction, these flag values will be preserved across the call.

The flag values from `lr` at the instant of entry must be instated; it is not sufficient merely to preserve the flag values across the call. (Consider a procedure `ProcA` which has been 'tail-call optimised' and does: `CMPS a1, #0; MOVLT a2, #255; MOVGE a2, #0; B ProcB`. If `ProcB` merely preserves the flags it sees on entry, rather than restoring those from `lr`, the wrong flags may be set when `ProcB` returns direct to `ProcA`'s caller).

This standard does not define the values of `fp`, `sp` and `sl` at arbitrary moments during a procedure's execution, but only at the instants of (external) call and return. Further standards and restrictions may apply under particular operating systems, to aid event handling or debugging. In general, you are strongly encouraged to preserve `fp`, `sp` and `sl`, at all times.

The minimum amount of stack defined to be available is not particularly large, and as a general rule a language implementation should not expect much more, unless the conventions of the target operating system indicate otherwise. For example, code generated by the Arthur/RISC OS C compiler is able, if there is inadequate local workspace, to allocate more stack space from the C heap before continuing. Any language unable to do this may have its interaction with C impaired. That `sl` contains a stack chunk handle is important in achieving this. (See the later discussion of RISC OS register bindings for further details).

The statements about sp and SP_LWM are designed to optimise the testing of the one against the other. For example, in the RISC OS user-mode binding of APCS, sl contains SL_LWM+512, allowing a procedure's entry sequence to include something like

```
CMP  sp, sl
BLLT |x$stack_overflow|
```

where x$stack_overflow is a part of the run-time system for the relevant language. If this test fails, and x$stack_overflow is not called, there are at least 512 bytes free on the stack.

This procedure should only call other procedures when sp has been dropped by 256 bytes or less, guaranteeing that there is enough space for the called procedure's entry sequence (and, if needed, the stack extender) to work in.

If 256 bytes are not enough, the entry sequence has to drop sp before comparing it with sl in order to force stack extension (see later sections on implementation specifics for details of how the RISC OS C compiler handles this problem).

The stack backtrace data structure

At the instant of an external procedure call, the value in fp is zero or it points to a data structure that gives information about the sequence of outstanding procedure calls. This structure is in the format shown below:

```
fp points to here:      | save mask pointer |     [fp]
                        | return link value |     [fp, #-4]
                        | return sp value   |     [fp, #-8]
                        | fp value          |     [fp, #-12]
                       [|  saved v6 value   |]
                       [|  saved v5 value   |]
                       [|  saved v4 value   |]
                       [|  saved v3 value   |]
                       [|  saved v2 value   |]
                       [|  saved v1 value   |]
                       [|  saved a4 value   |]
                       [|  saved a3 value   |]
                       [|  saved a2 value   |]
                       [|  saved a1 value   |]
                       [|  saved f7 value   |]three words
```

```
[|  saved f6 value   |]three words
[|  saved f5 value   |]three words
[|  saved f4 value   |]three words
```

This picture shows between four and 26 words of store, with those words higher on the page being at higher addresses in memory. The values shown in square brackets are optional, and the presence of any does not imply the presence of any other. The floating point values are in extended format and occupy three words each.

At the instant of procedure call, all of the following statements about this structure shall be true:

- The *return fp value* is either 0 or contains a pointer to another stack backtrace data structure of the same form. Each of these corresponds to an active, outstanding procedure invocation. The statements listed here are also true of this next stack backtrace data structure and, indeed, hold true for each structure in the chain.
- The *save mask pointer* value, when bits 0, 1, 26, 27, 28, 29, 30, 31 have been cleared, points twelve bytes beyond a word known as the *return data save instruction*.
- The return data save instruction is a word that corresponds to an ARM instruction of the following form:

  ```
  STMDB   sp!, {[a1], [a2], [a3], [a4],
                [v1], [v2], [v3], [v4], [v5], [v6],
                fp, ip, lr, pc}
  ```

  Note the square brackets in the above denote optional parts: thus, there are 12 x 1024 possible values for the return data save instruction, corresponding to the following bit patterns:

  ```
      1110 1001 0010 1101 1101 10xx xxxx xxxx    APCS-R, APCS-U

  or                       !    !  !

      1110 1001 0010 1100 1100 11xx xxxx xxxx    APCS-A (obsolete)
  ```

  The least significant 10 bits represent argument and variable registers: if bit N is set, then register N will be transferred.

  The optional parts `[a1]`, `[a2]`, `[a3]`, `[a4]`, `[v1]`, `[v2]`, `[v3]`, `[v4]`, `[v5]` and `[v6]` in this instruction correspond to those optional parts of the stack backtrace data structure that are present such that: for all M, if

[vM] or [aM] is present then so is [| saved vM value |] or [| saved aM value |], and if [vM] or [aM] is absent then so is [| saved vM value |] and [| saved aM value |]. This is as if the stack backtrace data structure were formed by the execution of this instruction, following the loading of ip from sp (as is very probably the case).

- The sequence of up to four instructions following the return data save instruction determines whether saved floating point registers are present in the backtrace structure. The four optional instructions allowed in this sequence are:

```
STFE f7, [sp, #-12]! ; 1110 1101 0110 1101 0111 0001 0000 0011
STFE f6, [sp, #-12]! ; 1110 1101 0110 1101 0110 0001 0000 0011
STFE f5, [sp, #-12]! ; 1110 1101 0110 1101 0101 0001 0000 0011
STFE f4, [sp, #-12]! ; 1110 1101 0110 1101 0100 0001 0000 0011
                                           !
```

  Any or all of these instructions may be missing, and any deviation from this order or any other instruction terminates the sequence.

  (A historical bug in the C compiler (now fixed) inserted a single arithmetic instruction between the return data save instruction and the first STFE. Some Acorn software allows for this.)

  The bit patterns given are for APCS-R/APCS-U register bindings. In the obsolete APCS-A bindings, the bit indicated by '!' is 0.

  The optional instructions saving f4, f5, f6 and f7 correspond to those optional parts of the stack backtrace data structure that are present such that: for all M, if STFE fM is present then so is [| saved fM value |]; if STFE fM is absent then so is [| saved fM value |].

- At the instant when procedure A calls procedure B, the stack backtrace data structure pointed at by fp contains exactly those elements [v1], [v2], [v3], [v4], [v5], [v6], [f4], [f5], [f6], [f7], fp, sp and pc which must be restored into the corresponding ARM registers in order to cause a correct exit from procedure A, albeit with an incorrect result.

Notes

The following example suggests what the entry and exit sequences for a procedure are likely to look like (though entry and exit are not defined in terms of these instruction sequences because that would be too restrictive; a good compiler can often do better than is suggested here):

```
entry   MOV   ip, sp
        STMDB   sp!, {argRegs, workRegs, fp, ip, lr, pc}
        SUB     fp, ip, #4
exit    LDMDB   fp, {workRegs, fp, sp, pc}^
```

Many apparent idiosyncrasies in the standard may be explained by efforts to make the entry sequence work smoothly. The example above is neither complete (no stack limit checking) nor mandatory (making arguments contiguous for C, for instance, requires a slightly different entry sequence; and storing `argRegs` on the stack may be unnecessary).

The `workRegs` registers mentioned above correspond to as many of `v1` to `v6` as this procedure needs in order to work smoothly. At the instant when procedure A calls any other, those workspace registers not mentioned in A's return data save instruction will contain the values they contained at the instant A was entered. Additionally, the registers `f4-f7` not mentioned in the floating point save sequence following the return data save instruction will also contain the values they contained at the instant A was entered.

This standard does not require anything of the values found in the optional parts `[a1]`, `[a2]`, `[a3]`, `[a4]` of a stack backtrace data structure. They are likely, if present, to contain the saved arguments to this procedure call; but this is not required and should not be relied upon.

Defined bindings of the procedure call standard

APCS-R and APCS-U: The RISC OS and RISC iX PCSs

These bindings of the ARM Procedure Call Standard are used by:

- RISC OS applications running in ARM user-mode
- compiled code for RISC OS modules and handlers running in ARM SVC-mode
- RISC iX applications (which make no use of `sl`) running in ARM user mode
- RISC iX kernels running in ARM SVC mode.

The call-frame register bindings are:

```
sl    RN    10   ; stack limit / stack chunk handle
                 ;   unused by RISC iX applications
fp    RN    11   ; frame pointer
ip    RN    12   ; used as temporary workspace
sp    RN    13   ; lower end of current stack frame
```

Although not formally required by this standard, it is considered good taste for compiled code to preserve the value of `sl` everywhere.

The invariants `sp > ip > fp` have been preserved, in common with the obsolete APCS-A (described below), allowing symbolic assembly code (and compiler code-generators) written in terms of register names to be ported between APCS-R, APCS-U and APCS-A merely by relabelling the call-frame registers provided:

- When call-frame registers appear in `LDM`, `LDR`, `STM` and `STR` instructions they are named symbolically, never by register numbers or register ranges.
- No use is made of the ordering of the four call-frame registers (eg in order to load/save `fp` or `sp` from a full register save).

APCS-R: Constraints on `sl` (For RISC OS applications and modules)

In SVC and IRQ modes (collectively called module mode) `SL_LWM` is implicit in `sp`: it is the next megabyte boundary below `sp`. Even though the SVC-mode and IRQ-mode stacks are not extensible, `sl` still points 512 bytes above a skeleton stack-chunk descriptor (stored just above the megabyte boundary). This is done for compatibility with use by applications running in ARM user-mode and to facilitate module-mode stack-overflow detection. In other words:

```
sl = SL_LWM + 512.
```

When used in user-mode, the stack is segmented and is extended on demand. Acorn's language-independent run-time kernel allows language run-time systems to implement stack extension in a manner which is compatible with other Acorn languages. `sl` points 512 bytes above a full stack-chunk structure and, again:

```
sl = SL_LWM + 512.
```

Mode-dependent stack-overflow handling code in the language-independent run-time kernel faults an overflow in module mode and extends the stack in application mode. This allows library code, including the run-time kernel, to be shared between all applications and modules written in C.

In both modes, the value of `sl` must be valid immediately before each external call *and each return from an external call.*

Deallocation of a stack chunk may be performed by intercepting returns from the procedure that caused it to be allocated. Tail-call optimisation complicates the relationship, so, in general, `sl` is required to be valid immediately before every return from external call.

APCS-U: Constraints on `sl` (For RISC iX applications and RISC iX kernels)

In this binding of the APCS the user-mode stack auto-extends on demand so `sl` is unused and there is no stack-limit checking.

In kernel mode, `sl` is reserved to Acorn.

APCS-A: The obsolete Arthur application PCS

This obsolete binding of the procedure-call standard is used by Arthur applications running in ARM user-mode. The applicable call-frame register bindings are as follows:

```
sl    RN    13    ; stack limit/stack chunk handle
fp    RN    10    ; frame pointer
ip    RN    11    ; used as temporary workspace
sp    RN    12    ; lower end of current stack frame
```

(Use of `r12` as `sp`, rather than the architecturally more natural `r13`, is historical and predates both Arthur and RISC OS.)

In this binding of the APCS, the stack is segmented and is extended on demand. Acorn's language-independent run-time kernel allows language run-time systems to implement stack extension in a manner which is compatible with other Acorn languages.

The stack limit register, `sl`, points 512 bytes above a stack-chunk descriptor, itself located at the low-address end of a stack chunk. In other words:

```
sl = SL_LWM + 512.
```

The value of `sl` must be valid immediately before each external call and each return from an external call.

Although not formally required by this standard, it is considered good taste for compiled code to preserve the value of `sl` everywhere.

Notes on APCS bindings

Invariants and APCS-M

In all future supported bindings of APCS `sp` shall be bound to `r13`. In all supported bindings of APCS the invariant `sp > ip > fp` shall hold. This means that the only other possible binding of APCS is APCS-M:

```
sl    RN    12   ; stack limit/stack chunk handle
fp    RN    10   ; frame pointer
ip    RN    11   ; used as temporary workspace
sp    RN    13   ; lower end of current stack frame
```

This binding of APCS is unlikely to be used (by Acorn).

Further Restrictions in SVC Mode and IRQ Mode

There are some consequences of the ARM's architecture which, while not formally acknowleged by the ARM Procedure Call Standard, need to be understood by implementors of code intended to run in the ARM's SVC and IRQ modes.

An IRQ corrupts `r14_irq`, so IRQ-mode code must run with IRQs off until `r14_irq` has been saved. Acorn's preferred solution to this problem is to enter and exit IRQ handlers written in high-level languages via hand-crafted 'wrappers' which on entry save `r14_irq`, change mode to SVC, and enable IRQs and on exit restore the saved `r14_irq` (which restores IRQ mode and the IRQ-enable state). Thus the handlers themselves run in SVC mode, avoiding this problem in compiled code.

Both SWIs and aborts corrupt `r14_svc`. This means that care has to be taken when calling SWIs or causing aborts in SVC mode.

In high-level languages, SWIs are usually called out of line so it suffices to save and restore `r14` in the calling veneer around the SWI. If a compiler can generate in-line SWIs, then it should, of course, also generate code to save and restore `r14` in-line, around the SWI, unless it is known that the code will not be executed in SVC mode.

An abort in SVC mode may be symptomatic of a fatal error or it may be caused by page faulting in SVC mode. Acorn expects SVC-mode code to be 'correct', so these are the only options. Page faulting can occur because an instruction needs to be fetched from a missing page (causing a prefetch abort) or because of an attempted data access to a missing page (causing a data abort). The latter may occur even if the SVC-mode code is not itself paged (consider an unpaged kernel accessing a paged user-space).

A data abort is completely recoverable provided `r14` contains nothing of value at the instant of the abort. This can be ensured by:

- saving `r14` on entry to every procedure and restoring it on exit
- not using `r14` as a temporary register in any procedure
- avoiding page faults (stack faults) in procedure entry sequences.

A prefetch abort is harder to recover from and an aborting BL instruction cannot be recovered, so special action has to be taken to protect page faulting procedure calls.

For Acorn C, `r14` is saved in the second or third instruction of an entry sequence. Aligning all procedures at addresses which are 0 or 4 modulo 16 ensures that the critical part of the entry sequence cannot prefetch-abort. A compiler can do this by padding all code sections to a multiple of 16 bytes in length and being careful about the alignment of procedures within code sections.

Data-aborts early in procedure entry sequences can be avoided by using a software stack-limit check like that used in APCS-R.

Finally, the recommended way to protect BL instructions from prefetch-abort corruption is to precede each BL by a `MOV ip, pc` instruction. If the BL faults, the prefetch abort handler can safely overwrite `r14` with `ip` before resuming execution at the target of the BL. If the prefetch abort is not caused by a BL then this action is harmless, as `r14` has been corrupted anyway (and, by design, contained nothing of value at any instant a prefetch abort could occur).

Examples from Acorn language implementations

Example procedure calls in C

Here is some sample assembly code as it might be produced by the C compiler:

```
; gggg is a function of 2 args that needs one register variable (v1)
gggg    MOV     ip, sp
        STMFD   sp!, {a1, a2, v1, fp, ip, lr, pc}
        SUB     fp, ip, #4              ; points at saved PC
        CMPS    sp, sl
        BLLT    |x$stack_overflow|      ; handler procedure
        ...
        MOV     v1, ...                 ; use a register variable
        ...
        BL      ffff
        ...
        MOV    ..., v1                  ; rely on its value after ffff()
```

Within the body of the procedure, arguments are used from registers, if possible; otherwise they must be addressed relative to `fp`. In the two argument case shown above, `arg1` is at `[fp,#-24]` and `arg2` is at `[fp,#-20]`. But as discussed below, arguments are sometimes stacked with positive offsets relative to `fp`.

Local variables are never addressed offset from `fp`; they always have positive offsets relative to `sp`. In code that changes `sp` this means that the offsets used may vary from place to place in the code. The reason for this is that it permits the procedure `x$stack_overflow` to recover by setting `sp` (and `sl`) to some new stack segment. As part of this mechanism, `x$stack_overflow` may alter memory offset from `fp` by negative amounts, eg `[fp, #-64]` and downwards, provided that it adjusts `sp` to provide workspace for the called routine.

If the function is going to use more than 256 bytes of stack it must do:

```
SUB     ip, sp, #<my stack size>
CMPS    ip, sl
BLLT    |x$stack_overflow_1|
```

instead of the two-instruction test shown above.

If a function expects no more than four arguments it can push all of them onto the stack at the same time as saving its old `fp` and its return address (see the example above); arguments are then saved contiguously in memory with `arg1` having the lowest address. A function that expects more than four arguments has code at its head as follows:

```
MOV     ip, sp
STMFD   sp!, {a1, a2, a3, a4}         ; put arg1-4 below stacked args
STMFD   sp!, {v1, v2, fp, ip, lr, pc}   ; v1-v6 saved as necessary
SUB     fp, ip, #20                   ; point at newly created call-frame
CMPS    sp, sl
BLLT    |x$stack_overflow|
...
...
LDMEA   fp, {v1, v2, fp, sp, pc}^     ; restore register vars & return
```

The store of the argument registers shown here is not mandated by APCS and can often be omitted. It is useful in support of debuggers and run-time trace-back code and required if the address of an argument is taken.

The entry sequence arranges that arguments (however many there are) lie in consecutive words of memory and that on return `sp` is always the lowest address on the stack that still contains useful data.

The time taken for a call, enter and return, with no arguments and no registers saved, is about 22 S-cycles.

Although not required by this standard, the values in `fp`, `sp` and `sl` are maintained while executing code produced by the C compiler. This makes it much easier to debug compiled code.

Multi-word results other than double precision reals in C programs are represented as an implicit first argument to the call, which points to where the caller would like the result placed. It is the first, rather than the last, so that it works with a C function that is not given enough arguments.

Procedure calls in other language implementations

Assembler

The procedure call standard is reasonably easy and natural for assembler programmers to use. The following rules should be followed:

- Call-frame registers should always be referred to explicitly by symbolic name, never by register number or implicitly as part of a register range.

- The offsets of the call-frame registers within a register dump should not be wired into code. Always use a symbolic offset so that you can easily change the register bindings.

Fortran

The Acorn/TopExpress Arthur/RISC OS Fortran-77 compiler violates the APCS in a number of ways that preclude inter-working with C, except via assembler veneers. This may be changed in future releases of the Fortran-77 product.

Pascal

The Acorn/3L Arthur/RISC OS ISO-Pascal compiler violates the APCS in a number of ways that preclude inter-working with C, except via assembler veneers. This may be changed in future releases of the ISO-Pascal product.

Lisp, BCPL and BASIC

These languages have their own special requirements which make it inappropriate to use a procedure call of the form described here. Naturally, all are capable of making external calls of the given form, through a small amount of assembler 'glue' code.

General

Note that there is no requirement specified by the standard concerning the production of re-entrant code, as this would place an intolerable strain on the conventional programming practices used in C and Fortran. The behaviour of a procedure in the face of multiple overlapping invocations is part of the specification of that procedure.

Various lessons

This document is not intended as a general guide to the writing of code-generators, but it is worth highlighting various optimisations that appear particularly relevant to the ARM and to this standard.

The use of a callee-saving standard, instead of a caller-saving one, reduces the size of large code images by about 10% (with compilers that do little or no interprocedural optimisation).

In order to make effective use of the APCS, compilers must compile code a procedure at a time. Line-at-a-time compilation is insufficient.

The preservation of condition codes over a procedure call is often useful because any short sequence of instructions (including calls) that forms the body of a short `IF` statement can be executed without a branch instruction. For example:

```
if (a < 0) b = foo();
```

can compile into:

```
        CMP     a, #0
        BLLT    foo
        MOVLT   b, a1
```

In the case of a 'leaf' or 'fast' procedure – one that calls no other procedures – much of the standard entry sequence can be omitted. In very small procedures, such as are frequently used in data abstraction modules, the cost of the procedure can be very small indeed. For instance, consider:

```
        typedef struct {...; int a; ...} foo;
        int get_a(foo* f) {return(f->a);}
```

The procedure `gta` can compile to just:

```
        LDR     a1, [a1, #aOffset]
        MOVS    pc, lr
```

This is also useful in procedures with a conditional as the top level statement, where one or other arm of the conditional is 'fast' (ie calls no procedures). In this case there is no need to form a stack frame there. For example, using this, the C program:

```
        int sum(int i)
        {
           if (i <= 1)
              return(i);
           else
              return(i + sum(i-1));
        }
```

could be compiled into:

```
sum     CMP     a1, #1 ; try fast case
        MOVSLE  pc, lr ; and if appropriate, handle quickly!
        ; else, form a stack frame and handle the rest as normal code.
        MOV     ip, sp
        STMDB   sp!, {v1, fp, ip, lr, pc}
        CMP     sp, sl
        BLLT    overflow
        MOV     v1, a1                  ; register to hold i
        SUB     a1, a1, #1              ; set up argument for call
        BL      sum                     ; do the call
        ADD     a1, a1, v1              ; perform the addition
        LDMEA   fp, {v1, fp, sp, pc}^   ; and return
```

This is only worthwhile if the test can be compiled using only `ip`, and any spare of `a1-a4`, as scratch registers. This technique can significantly speed up certain speed-critical routines, such as read and write character. At the present time, this optimisation is not performed by the C compiler.

Finally, it is often worth applying the 'tail call' optimisation, especially to procedures which need to save no registers. For example, the code fragment:

```
extern void *malloc(size_t n)
{
    return primitive_alloc(NOTGCABLEBIT, BYTESTOWORDS(n));
}
```

is compiled by the C compiler into:

```
malloc  ADD     a1, a1, #3              ; 1S
        MOV     a2, a1, LSR #2          ; 1S
        MOV     a1, #1073741824         ; 1S
        B       primitive_alloc         ; 1N+2S = 4S
```

This avoids saving and restoring the call-frame registers and minimises the cost of interface 'sugaring' procedures. This saves five instructions and, on a 4/8MHz ARM, reduces the cost of the malloc sugar from 24S to 7S.

Appendix E: kernel.h

```
/*
 *      Interface to host OS.
 *      Copyright (C) Acorn Computers Ltd., 1988
 */

#ifndef __size_t
#  define __size_t 1
   typedef unsigned int size_t;   /* from <stddef.h> */
#endif

typedef struct {
   int r[10];              /* only r0 - r9 matter for swi's */
} _kernel_swi_regs;

typedef struct {
   int load, exec;         /* load, exec addresses */
   int start, end;         /* start address/length, end address/attributes */
} _kernel_osfile_block;

typedef struct {
   void * dataptr;         /* memory address of data */
   int nbytes, fileptr;
   int buf_len;            /* these fields for Arthur gpbp extensions */
   char * wild_fld;        /* points to wildcarded filename to match */
} _kernel_osgbpb_block;

typedef struct {
   int errnum;             /* error number */
   char errmess[252];      /* error message (zero terminated) */
} _kernel_oserror;

typedef struct stack_chunk {
   unsigned long sc_mark;        /* == 0xf60690ff */
   struct stack_chunk *sc_next, *sc_prev;
   unsigned long sc_size;
   int (*sc_deallocate)();
} _kernel_stack_chunk;

extern _kernel_stack_chunk *_kernel_current_stack_chunk(void);

extern void _kernel_setreturncode(unsigned code);
```

```
extern void _kernel_exit(int);

extern void _kernel_exittraphandler(void);

#define _kernel_HOST_UNDEFINED    -1
#define _kernel_BBC_MOS1_0         0
#define _kernel_BBC_MOS1_2         1
#define _kernel_BBC_ACW            2
#define _kernel_BBC_MASTER         3
#define _kernel_BBC_MASTER_ET      4
#define _kernel_BBC_MASTER_COMPACT 5
#define _kernel_ARTHUR             6
#define _kernel_SPRINGBOARD        7
#define _kernel_A_UNIX             8

extern int _kernel_hostos(void);
/*
 *  Returns the identity of the host OS
 */

extern int _kernel_fpavailable(void);
/*
 * Returns 0 if floating point is not available (no emulator nor hardware)
 */

extern _kernel_oserror *_kernel_swi(int no, _kernel_swi_regs *in, _kernel_swi_regs
*out);
/*
 *  Generic SWI interface.  Returns NULL if there was no error.
 *  The SWI number may have the X bit set (bit 17) or not; it makes no
 *  difference
 */

extern char *_kernel_command_string(void);
/*
 * Returns the address of (maybe a copy of) the string used to run the program
 */

/*
 *  The int value returned by the following functions may have value:
 *    >= 0  if the call succeeds (significance then depends on the function)
 *     -1   if the call fails but causes no os error (eg escape for rdch).  Not
 *          all functions are capable of generating this result.  This return
 *          value corresponds to the C flag being set by the SWI.
 *     -2   if the call causes an os error (in which case, _kernel_oserror must
 *          be used to find which error it was)
 */

#define _kernel_ERROR (-2)

extern int _kernel_osbyte(int op, int x, int y);
/*
 *  Performs an OSByte operation.
 *  If there is no error, the result contains
```

```
 *     the return value of R1 (X) in its bottom byte
 *     the return value of R2 (Y) in its second byte
 *     1 in the third byte if carry is set on return, otherwise 0
 *     0 in its top byte
 *  (Not all of these values will be significant, depending on the
 *   particular OSByte operation).
 */

extern int _kernel_osrdch(void);
/*
 * Returns a character read from the currently selected OS input stream
 */

extern int _kernel_oswrch(int ch);
/*
 * Writes a byte to all currently selected OS output streams
 * The return value just indicates success or failure.
 */

extern int _kernel_osbget(unsigned handle);
/*
 * Returns the next byte from the file identified by 'handle'.
 * (-1 => EOF).
 */

extern int _kernel_osbput(int ch, unsigned handle);
/*
 * Writes a byte to the file identified by 'handle'.
 * The return value just indicates success or failure.
 */

extern int _kernel_osgbpb(int op, unsigned handle, _kernel_osgbpb_block *inout);/*
 * Reads or writes a number of bytes from a filing system.
 * The return value just indicates success or failure.
 * Note that for some operations, the return value of C is significant,
 * and for others it isn't.  In all cases, therefore, a return value of -1
 * is possible, but for some operations it should be ignored.
 * (To confuse matters further, some Brazil filing systems don't set C when
 *  they should, so the best course of action may be to ignore the result
 *  unless it indicates an error).
 */

extern int _kernel_osword(int op, int *data);
/*
 * Performs an OSWord operation.
 * The size and format of the block *data depends on the particular OSWord
 * being used; it may be updated.
 */

extern int _kernel_osfind(int op, char *name);
/*
 * Opens or closes a file.
 *   Open returns a file handle (0 => open failed without error)
 *   Close the return value just indicates success or failure
```

```
 */

extern int _kernel_osfile(int op, const char *name, _kernel_osfile_block *inout);
/*  Performs an OSFile operation, with values of r2 - r5 taken from the osfile
 *  block.  The block is updated with the return values of these registers,
 *  and the result is the return value of r0 (or an error indication)
 */

extern int _kernel_osargs(int op, unsigned handle, int arg);
/*
 *  Performs an OSArgs operation.
 *  The result is an error indication, or
 *    the current filing system number (if op = handle = 0)
 *    the value returned in R2 by the OSArgs operation otherwise
 */

extern int _kernel_oscli(const char *s);
/*
 *  Hands the argument string to the OS command line interpreter to execute
 *  as a command.  This should not be used to invoke other applications:
 *  _kernel_system exists for that.  Even using it to run a replacement
 *  application is somewhat dubious (abort handlers are left as those of the
 *  current application).
 *  The return value just indicates error or no error
 */

extern _kernel_oserror *_kernel_last_oserror(void);
/*
 *  Returns a pointer to an error block describing the last os error.
 *  (Not cleared before a SWI call, so it need have no connection with the
 *   last SWI called unless it is known that that produced an error).
 *  If _kernel_swi caused the last os error, the error already returned by that
 *  call gets returned by this too.
 *  Never returns NULL: if there has been no error, returns a pointer to
 *  errnum 0 and null errmess
 */

extern _kernel_oserror *_kernel_getenv(const char *name, char *buffer, unsigned
size);
/*
 *  Reads the value of a system variable, placing the value string in the
 *  buffer (of size size).
 *  Under Arthur, this just gives access to  OS_ReadVarVal.
 *  Under Brazil, it accesses the file  $.environ
 *  (lines of which have the format  varname space value newline).
 */

extern _kernel_oserror *_kernel_setenv(const char *name, const char *value);
/*
 *  Updates the value of a system variable to be string valued, with the
 *  given value (value = NULL deletes the variable)
 *  Under Brazil, this returns the error "Not implemented"
 */
```

```
extern int _kernel_system(const char *string, int chain);
/*
 *  Hands the argument string to the OS command line interpreter to execute.
 *  If the string causes an application to be invoked, it will execute as a
 *  sub-program of the caller if chain is 0 (so that when it terminates
 *  control returns to the caller); as a replacement if chain is non-zero.
 *  Note that running sub-programs requires care: the OS provides no means
 *  of protection against program load overwriting the current application
 *  (in which case, when it exits the result is unlikely to be pretty).
 *  And of course, since the sub-program executes in the same address space,
 *  there is no protection against errant writes by it to the code or data
 *  of the caller.
 *  The return value just indicates error or no error
 */


extern unsigned _kernel_alloc(unsigned minwords, void **block);
/*
 *  Tries to allocate a block of sensible size >= minwords.  Failing that,
 *  it allocates the largest possible block (may be size zero).
 *  Sensible size means at least 2K words.
 *  *block is returned a pointer to the start of the allocated block
 *  (NULL if 'a block of size zero' has been allocated).
 */

typedef void freeproc(void *);
typedef void * allocproc(unsigned);

extern void _kernel_register_allocs(allocproc *malloc, freeproc *free);
/*
 *  Registers procedures to be used by the kernel when it requires to
 *  free or allocate storage.  The allocproc may be called during stack
 *  extension, so may not check for stack overflow (nor may any procedure
 *  called from it), and must guarantee to require no more than 41 words
 *  of stack.
 */

extern int _kernel_escape_seen(void);
/*
 * Returns 1 if there has been an escape since the previous call of
 * _kernel_escape_seen (or since program start, if there has been no
 * previous call).  Escapes are never ignored with this mechanism,
 * whereas they may be with the language EventProc mechanism since there
 * may be no stack to call it on.
 */

typedef union {
    struct {int s:1, u:16, x: 15; unsigned mhi, mlo; } i;
    int w[3]; } _extended_fp_number;

typedef struct {
   int r4, r5, r6, r7, r8, r9;
   int fp, sp, pc, sl;
   _extended_fp_number f4, f5, f6, f7; } _kernel_unwindblock;
```

```
extern int _kernel_unwind(_kernel_unwindblock *inout, char **language);
/*
 *  Unwinds the call stack one level.
 *  Returns >0 if it succeeds
 *           0 if it fails because it has reached the stack end
 *          <0 if it fails for any other reason (eg stack corrupt)
 *  Input values for fp, sl and pc  must be correct.
 *  r4-r9 and f4-f7 are updated if the frame addressed by the input value
 *  of fp contains saved values for the corresponding registers.
 *  fp, sp, sl and pc are always updated
 *  *language is returned a pointer to a string naming the language
 *  corresponding to the returned value of pc.
 */

extern char *_kernel_procname(int pc);
/*
 *  Returns a string naming the procedure containing the address pc.
 *  (or 0 if no name for it can be found).
 */

extern char *_kernel_language(int pc);
/*
 *  Returns a string naming the language in whose code the address pc lies.
 *  (or 0 if it is in no known language).
 */

/*  divide and remainder functions.
 *  The signed functions round towards zero.
 *
 *  The div functions actually also return the remainder in a2, and use of
 *  this by a code-generator will be more efficient than a call to the rem
 *  function.
 *
 *  Language RTSs are encouraged to use these functions rather than providing
 *  their own, since considerable effort has been expended to make these fast.
 */

extern unsigned _kernel_udiv(unsigned divisor, unsigned dividend);
extern unsigned _kernel_urem(unsigned divisor, unsigned dividend);
extern unsigned _kernel_udiv10(unsigned dividend);

extern int _kernel_sdiv(int divisor, int dividend);
extern int _kernel_srem(int divisor, int dividend);
extern int _kernel_sdiv10(int dividend);

/*
 * Description of a 'Language description block'
 */

typedef enum { NotHandled, Handled } _kernel_HandledOrNot;

typedef struct {
   int regs [16];
```

```
} _kernel_registerset;

typedef struct {
   int regs [10];
} _kernel_eventregisters;

typedef void (*PROC) (void);
typedef _kernel_HandledOrNot (*_kernel_trapproc) (int code, _kernel_registerset
*regs);
typedef _kernel_HandledOrNot (*_kernel_eventproc) (int code, _kernel_registerset
*regs);

typedef struct {
   int size;
   int codestart, codeend;
   char *name;
   PROC (*InitProc)(void);
   PROC FinaliseProc;
   _kernel_trapproc TrapProc;
   _kernel_trapproc UncaughtTrapProc;
   _kernel_eventproc EventProc;
   _kernel_eventproc UnhandledEventProc;
   void (*FastEventProc) (_kernel_eventregisters *);
   int (*UnwindProc) (_kernel_unwindblock *inout, char **language);
   char * (*NameProc) (int pc);
} _kernel_languagedescription;

typedef int _kernel_ccproc(int, int, int);

extern int _kernel_call_client(int a1, int a2, int a3, _kernel_ccproc callee);
/* This is for shared library use only, and is not exported to shared library
 * clients. It is provided to allow library functions which call arbitrary
 * client code (C library signal, exit, _main) to do so correctly if the
 * client uses the old calling standard.
 */

extern int _kernel_client_is_module(void);
/* For shared library use only, not exported to clients.  Returns a
 * non-zero value if the library's client is a module executing in user
 * mode (ie module run code).
 */

extern int _kernel_processor_mode(void);

extern void _kernel_irqs_on(void);

extern void _kernel_irqs_off(void);

extern int _kernel_irqs_disabled(void);
/* returns 0 if interrupts are enabled; some non-zero value if disabled. */

extern void *_kernel_RMAalloc(size_t size);

extern void *_kernel_RMAextend(void *p, size_t size);
```

```
extern void _kernel_RMAfree(void *p);
```

Appendix F: The floating point emulator

The floating point emulator is a relocatable module which provides support for floating point instructions. It must be resident in memory to run programs which perform operations on real numbers.

Its primary function is to emulate floating point instructions in software on machines which do not have the optional hardware floating point co-processor attached.

However, even with the co-processor attatched, the floating point emulator is still required

- to interface with the co-processor
- to perform range reduction on trigonometric function arguments
- for a few floating point instructions that the co-processor does not directly support.

There are two variants of the floating point emulator:

FPE280	software-only floating point support – v 2.80 and earlier
FPEmulator	hardware-assisted AND software-only support – v 3.10 and later.

Both have the same module name, namely `FPEmulator`. You can find out the version number of the module currently resident in your computer by typing the following at the * prompt:

```
*help modules
```

FPE280

On initialisation, this module disables the floating point co-processor if it finds one present. It occupies 25Kb.

FPEmulator

This behaves just like FPE280 if no co-processor is attached (ie it emulates all floating point instructions in software), but it makes use of the co-processor if it is present. It occupies 37Kb.

Using the compiler

Without the floating point maths co-processor

If your machine does not have the floating point co-processor attached, the floating point emulator is required to run any C program which performs operations on real numbers.

The floating point emulator supplied with Release 3 of the C compiler is FPE280, and is located on Disc 1 as the file `FPE280` in the `$.Modules` directory.

Before loading the emulator, it is a good idea to issue a command that will check that no more recent version of the module is already present, by typing

```
*RMEnsure FPEmulator 2.80
```

Then load the emulator:

```
*RMLoad $.Modules.fpe280
```

Once you have set up your working environment, you will find it convenient to place the module in your `!System` directory (as `!System.modules.FPEmulator`) and arrange for it to be loaded automatically on power up.

Observe the change of file name to `FPE280`, since existing applications will incorporate the earlier name in their start-up sequence.

With the floating point maths co-processor

In order to make use of the speed increase given by the floating point co-processor, you will need to use the FPEmulator module.

This is supplied with the co-processor, and you will find it convenient to copy the module into your `!System` directory and arrange for it to be loaded automatically on power up.

Floating point requirements of applications

Applications should cater for both floating point environments: with and without the co-processor. In general, programs do not need to know whether a co-processor is fitted; the only effective difference is in the speed of execution. However, the combined hardware and software variant, FPEmulator, is larger than the software-only variant, FPE280, since it includes the code for interfacing with the co-processor. Therefore, to cater for both environments, an application must be able to accommodate the extra 12Kb RAM taken up by FPEmulator.

Software products do not have to supply either version of the floating point emulator. FPE280 is supplied with new machines, version 2.7 of the emulator is bundled with the RISC OS upgrade kit, and FPEmulator is supplied with the co-processor itself. It is then up to you to have the appropriate emulator in your `!System` directory; this should be covered in the manual accompanying the application.

Function Index

The main entry for each function is printed in bold type.

T

U

V

W

X

Subject Index

Symbols

A

E

F

G

H

I

N

O

P

Q

R

U

V

W

X

Reader's Comment Form

C Release 3 Guide

We would greatly appreciate your comments about this Guide, 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 programming experience?

☐ **New to programming**

☐ **New to C programming**

☐ **Experienced with other C compilers**

☐ **Used Acorn C Release 1 or 2**

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.

Cut out (or photocopy) and post to:

Dept RC, Technical Publications
Acorn Computers Limited
645 Newmarket Road
Cambridge CB5 8PB.