



Easy C++

An Easy to use C Compiler for RISC OS

User Guide

ProAction

3QD
DEVELOPMENTS LTD

APDL

Easy C++

Copyright © ProAction & 3QD Developments Ltd 2026
All rights reserved.

No part of this product may be reproduced in whole or part by any means without written permission of the publisher. Unauthorised hiring, renting, lending, public performance or broadcasting of this product or its parts is prohibited.

While every care is taken, the publisher cannot be held responsible for any errors in this product, or for loss of data or consequential effects from the use of this package.

Easy C++ was written by David Allison.
SIXTH EDITION 2025 (USER GUIDE)

Software and Documentation supplied by
APDL
a part of

3QD Developments Ltd

Single User version downloaded from the free of charge software section of
www.riscos.com

Please note that any free of charge downloads from RISCOS.com are supplied without any technical support package as per the licence agreement.

This document last edited on 29/09/26



If you have found this application and this manual useful please consider making a donation to help cover the costs of cataloguing, documenting, updating and hosting the APDL applications. Scanning the QR code will take you to our donations page.

Contents

Chapter 1 - Introduction	5
What is C	5
The structure of a C system	5
Libraries and linking	6
C standards	6
How to use this guide	6
Chapter 2 - Getting Started	7
Installation of Easy C++	7
Compiling and running a simple program	7
Compiling C++ programs	8
Chapter 3 Using Easy C++	9
Arrangement of files	9
Easy C dialogue box	9
Object window	10
Dialogue box status information	11
Easy C main menu options	12
Setup Dialogue Box	13
Easy C icon bar menu	15
Chapter 4 - The Make facility	17
Dhrystone	17
Creating the 'make file'	17
Using Make	27
Make options	18
Make file format	19
Chapter 5 - Easy C debugger	20
Getting started	20
Source level or object level debugging	20
Running the debugger	21
Running the program	21
Status information	22
Source files pane	22
Source listing window	23
Stack trace	23
Variables	24
Expanding structures and pointers	25
Expression evaluation	25
Variable list menu	25
Debug output	26
Breakpoints	26
The Profiler	29
Main menu options	30
Memory editor	31
Memory editor menu options	32
Icon bar menu	32
Expressions	32
Variable list window	33
Breakpoint Edit dialogue box	33
Examples:	33
Syntax of expressions	34
Regular expressions in Find	34
Limitations	35

Appendix A Using Easy C++ from the command line	37
Getting started	37
Compiling a simple C program	37
The compiler	38
Naming conventions	38
The default filename	38
Compiling multiple files	38
Include files	38
Error messages	39
Compiler qualifiers	39
The linker	43
Linking multiple files	44
Linker qualifiers	44
The library facility	45
Appendix B ARM Assembler	48
Appendix C Easy C language extensions	51

Chapter 1

Introduction

Easy C++ is designed for programmers who wish to learn or develop their skills in either, or both, the C or C++ programming languages. It has a very easy-to-use WIMP front-end to drive the compiler and linker, so there's no need to learn complicated command-line sequences. If you wish to use the command-line Easy C++ will allow you to do so!

Even though this package is aimed at beginners, the C compiler conforms to the international standard for the C programming language, while the C++ compiler conforms to the early specifications for a C++ programming language. As your skills develop, you will find that Easy C++ is quite capable of handling substantial programs, and features an integrated Make facility and ARM assembler for the more adventurous.

What is C

C is a high-level programming language that was developed in the 1970s. The term high-level is used to distinguish algorithmically based languages from those that are specific to machine hardware, namely assembler languages. Basic and Pascal are both examples of high-level languages. C was derived from an older language called BCPL via an inbetween language known as B. It was developed largely by two individuals - Brian Kernighan and Dennis Ritchie.

C++ was an evolution of C, developed to implement principles for object orientated programming. This later became the replacement tool for the compilation of Unix as it allowed the programmer to change code in stages. As it is an evolution of C many of the original structures can still be accessed.

One of the main reasons for developing C was to use it to implement the entire UNIX operating system. This intended use for C can give us some insight into why many people view C as the only programming language worth using, and why it has survived where others have failed. Clearly, for any language to be suitable for implementing an operating system it must be flexible enough to allow operations such as accessing registers in hardware devices. It is also essential that an operating system is as bug-free as possible, as any bugs are likely to have an effect on any applications run under that operating system. To achieve this, the language used must have adequate error checking to pick up potential problems, and should ideally be 'natural' enough that code can be written easily; it is no use if the programmer continually has to allow for idiosyncrasies in the language. Unfortunately these criteria are to an extent mutually exclusive. Any language flexible enough to allow direct access to hardware is wide open to obscure bugs as the programmer is by necessity able to perform functions which might be thought of as 'illegal'. However, C has managed to strike a happy medium and helps explain why it is so popular.

Some may consider C/C++ inferior to assembly language programming. It is, however, best to consider the example of a standard program: 10,000 lines of C++ may become 50,000 lines of assembly code. The other advantage of using an implementation of C is the cross-platform potential: once you gain confidence, you'll be able to implement platform-specific libraries allowing, perhaps; for PC and Acorn versions of a program to be compiled separately!

The structure of a C system

Unlike Basic, which is an interpreted language, C is a compiled language. With an interpreted language, the program is stored largely as it was entered, and when it is executed the interpreter takes each statement in turn and performs the necessary operations to implement the statement.

There are advantages and disadvantages with interpreted languages. On the plus side they are quick and easy to use—you can simply type in a program and run it. This can be very handy for short programs, or when you just want to try something out quickly. The main disadvantage is speed. Each time the program is executed the interpreter has to examine it statement by statement. Typically, the time taken to actually execute the statement makes up only a fraction of the total time taken to decode the statement and decide on what action to take. A second disadvantage is that to execute an interpreted program a copy of the interpreter must be present. This is not a problem with Basic on Acorn systems, as the Basic interpreter is in the RISC OS ROMs. However, on some systems, having to load the interpreter each time an application is run could be a real problem. Another, often overlooked disadvantage, is that each statement in an interpreted program is only examined when it is actually executed.

As we shall see shortly, each statement of a compiled program is looked at as a separate stage before it is run. The advantage of this is that major errors in the syntax of the program can be picked up and cured at an earlier stage. Compiled languages operate differently. The program is entered as it would be for any language, and is

then saved. This is referred to as the *source code*. Next, the *compiler* is invoked. This takes the source code and converts it piece by piece into machine code which performs the necessary operations. This machine code is called the object code. The major advantage of this method is that as the program is converted into machine code it will run much faster than an interpreted equivalent. The main disadvantage is that debugging is harder because if there is an error at compile time or run time you have to go back, edit the source code, and then recompile it all—a lengthy process sometimes.

Libraries and linking

In practice, the situation is more involved. C doesn't in itself contain any built-in functions to perform operations such as arithmetic or printing. Instead, these functions are provided separately in the form of a *library*. The functions in the library may themselves be written in C, or in assembler, or in a totally different language, and they can in turn call other library functions.

Using library functions in a C program is a two stage process. Firstly, a *declaration* of the function must be included in the source code. The purpose of this is to specify the entry and exit parameters of the functions. While it would be possible to type in such a declaration for each function into each program, it is more normal to include one or more standard *headers* in the program using a special compiler instruction. These headers are supplied with the compiler and include declarations for all the library functions.

The second stage is to combine the code for the library functions with the object code from the compiler. This is done by a tool called a linker which takes the object code and merges in the necessary routines from the library code. The output of the linker is a stand-alone program that can be executed without needing an interpreter to be loaded.

C standards

Over the years many C compilers have been produced, and unfortunately, many variations of the language have emerged since the Kernighan and Ritchie (K&R) original. In an attempt to resolve this situation, a standard definition of C has been drawn up by ANSI (the American National Standards Institute).

One of the main areas in which the ANSI standard has helped is the range of library functions available. The original K&R definition made no mention of the library and this meant that programs couldn't be ported from one system to another, because even if a particular function existed in both libraries, you couldn't guarantee that it took the same parameters or returned the same results.

The C++ implementation offered is based upon an early specification, and in some respects may not fully represent current abilities of the language. It is, however, our intention to develop this program further, ultimately offering one of the most capable compilers for RISC OS computers.

The Easy C compiler complies to the ANSI standard, but please note that it has not been validated by any standards authority. The standard ANSI library of functions is provided by the Easy C library STDCLib which calls the relevant functions in the Acorn shared C library (CLib).

How to use this guide

If you are a beginner, you should read the chapter *Getting started* first, which gives step-by-step instructions on how to compile, link and run a simple C program. You should then proceed to the chapter *Tutorial*, which is an introductory course on C programming.

If you have some C programming knowledge, read the chapter *Getting started* for instructions on how to compile, link and run a simple C program. With this information, you should be able to compile and link programs of your own. You may also wish to consult the chapters *Using Easy C++* and *The Make facility* which give a full description of the options available in Easy C. For details of the functions provided, consult the chapters *ANSI functions* and *RISC OS functions*.

This user guide assumes that you are familiar with the RISC OS desktop and filing systems. If you are not, please read the relevant sections of your computer user guides.

On-line help can be obtained using the RISC OS interactive help system with which Easy C is compatible.

Chapter 2 - Getting Started

We took the decision to insist upon hard disc installation for this version of Easy C++ and opted for CD-ROM presentation instead of floppy disc as a CD-ROM allows us to provide a number of support materials in addition to the program and keep them in a single, convenient, location.

Installation of Easy C++

The Easy C++ CD contains an installation application, which will copy to the specified location on your hard disc the required components of the Easy C++ package. It is fully self-sufficient, and installs to your hard disc everything you'll need to run Easy C++ on a day to day basis. Some support materials will remain on the CD-ROM. You can copy them to your hard drive if you prefer.

Compiling and running a simple program

The instructions given below describe how to compile, link and run a simple C program. It is assumed that users with hard discs have installed Easy C according to the previous instructions.

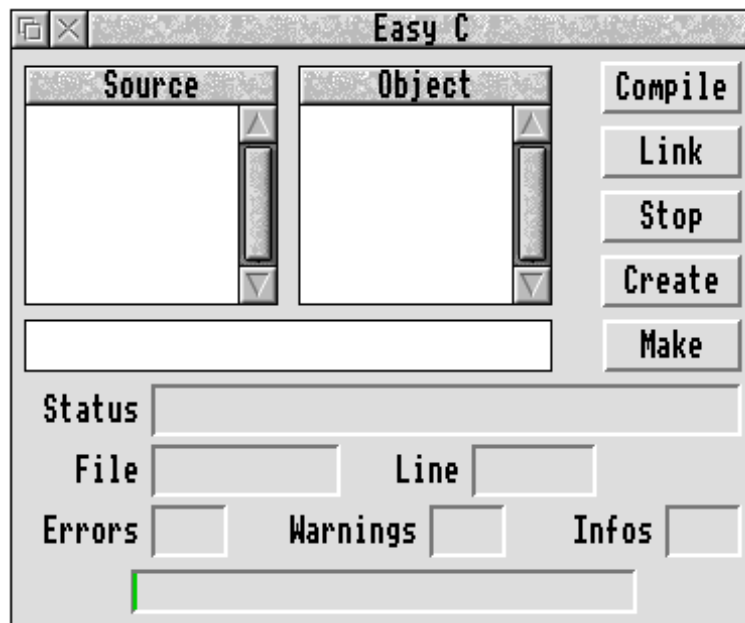
Step 1 - Run Easy C++

Click on the hard disc icon on the icon bar, double-click on the C directory to open it, then double-click on !EasyC. Easy C will now be installed on the icon bar.

If you on occasion see a 'C' logo, rather than 'C++', this is entirely due to historical aspects of this package: Easy C operates identically to Easy C++.

Step 2 - Open the Easy C dialogue box

Click Select on the Easy C icon on the icon bar to open the dialogue box.



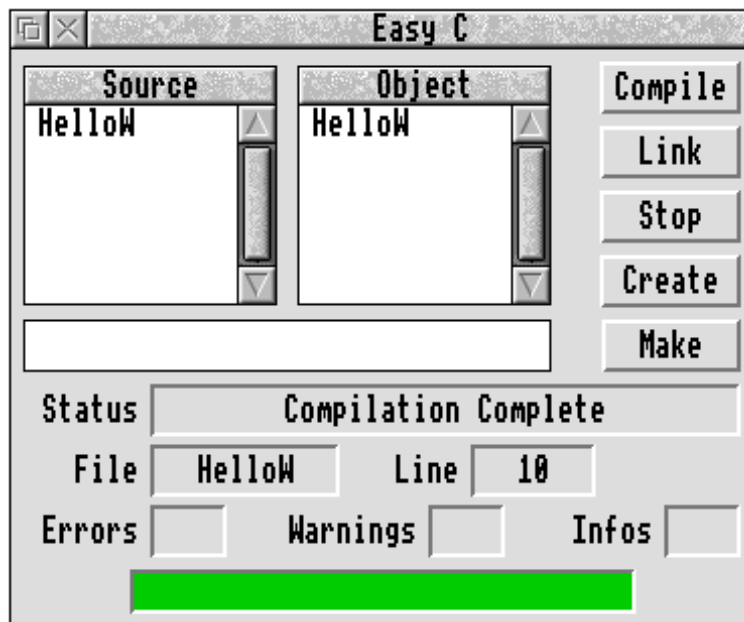
Step 3 - Load the C program to be compiled

For the purpose of this example we will compile an existing C program rather than write a new one. The C program to be compiled is called **HelloW**, and can be found in the **Programs** directory of the CD-ROM. Drag the file **HelloW** into the window labelled **Source** in the dialogue box

Step 4 - Compile the program HelloW

Click on **Compile** to compile HelloW. The Status box at the bottom of the dialogue box will display the message *Compiling*, and the green bar will indicate the progress made. Other icons in the dialogue box will be explained later and may be ignored for the present.

When HelloW has been compiled, the Status box shows the message *Compilation Complete*.



Step 5 - Link HelloW to the standard libraries

The compile action in step 4 generated an object file called *HelloW* which is shown in the **Object window**. Now click on **Link** to link the object file to the standard libraries. This procedure links HelloW with any external functions it uses, and converts the object code into executable machine code. The Status box confirms that linking is taking place and displays the message *Linking Complete* when complete.

Step 6 - Running HelloW

The final machine code program is called *HelloW*, and is stored in the *Programs.General* directory i.e. the directory that contains the *c* and *o* directories. Double-click on it to run it. This simple program prints the message *Hello, world* in a window on the screen.

Remember that if the error message *Shared C library not loaded* is displayed, you will have to run another program such as Easy C or Edit to load the modules, or use a proper !Run file. A !Run file prepares the system for the program you are loading: take a look at the Easy C run file for an example.

Step 7 - Compiling further programs

There are some more programs in directory *Programs.General.c* which you may wish to compile and run. Try compiling *Spheres*, *Grep* or *Sieve*.

However, first you must clear the Source and Object windows of any previous files i.e. *HelloW*. To do this, hold down **Ctrl** and drag a new source file, such as *Spheres*, into the Source window. This procedure will load the new file, clearing any previous files first. Alternatively, drag it to the Easy C icon on the icon bar to load it and clear both Source and Object windows at the same time.

Compiling C++ programs

The procedure for compiling C++ programs is identical to that described above. To ensure that it works, use the **Advanced Setup** option from the Easy C++ menu to check C++ is being compiled rather than C.

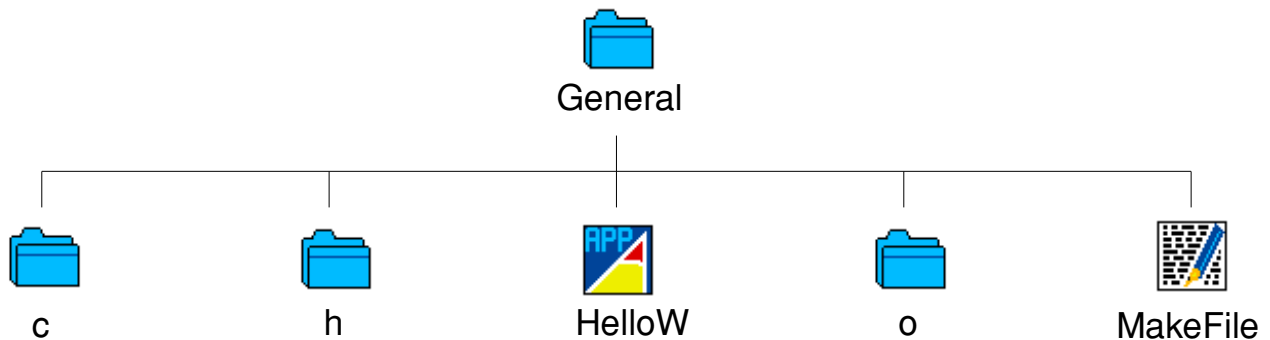
Chapter 3

Using Easy C++

This chapter describes all aspects of the Easy C++ in detail except for the Make facility which is described in the next chapter. It is important, while reading, to bear in mind that Easy C++ works in an identical way to Easy C: hence the references to Easy C. It's the same program, essentially, but offers only compilation of C code.

Arrangement of files

The diagram below shows how to arrange *Source* and *Object* directories for use with Easy C. This is how the example programs supplied with Easy C are arranged.



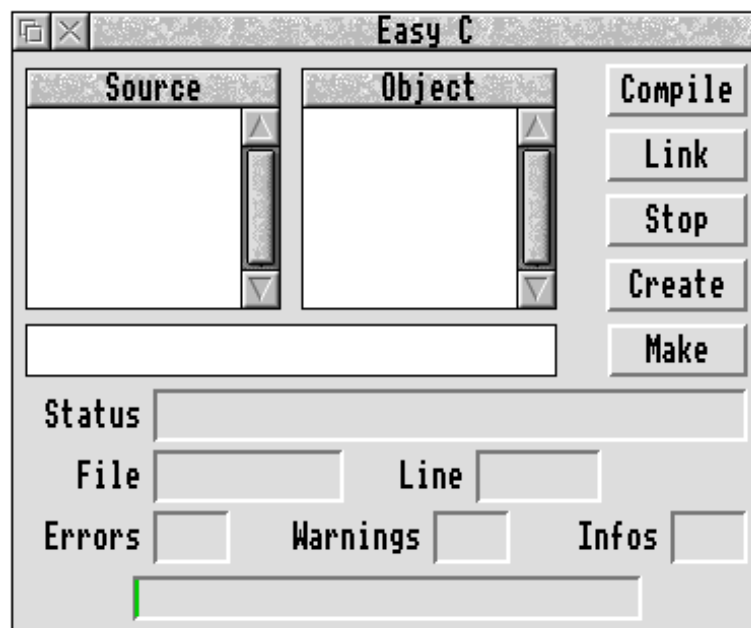
Directory *General* (or a name of your choice) contains directories *c* and *o*, and may be located anywhere in your disc structure. Directory *c* contains the source files, and directory *h*, any local header files.

Directory *o* will contain the object files generated by the compiler. The actual C programs produced, such as *HelloW*, will be put in the same directory as *c* and *o*.

You may optionally have the file *MakeFile* which is generated by the **Create** option. This file is used by the Make facility to compile and link an entire application based on the source and object files in the *c* and *o* directories. Please refer to the chapter *The Make facility* for full details.

Easy C dialogue box

This section describes the facilities and options available on the main Easy C dialogue box.



Source window

The Source window shows the names of the source files to be compiled when you click on **Compile**. Source files

should be text files located in a directory called `c`, and may be dragged into the source window singly or in groups. If **Ctrl** is held when dragging in source files all previous source and object files are deleted. The same thing happens if you drag source files to the Easy C icon bar icon. The compiler generates object code with the same name as the first source file but places it in the `o` directory (which must already exist). So if the first source file is called

```
idefs::IDEDisc4.$.C.Programs.General.c.HelloW
```

The object file generated will be called:

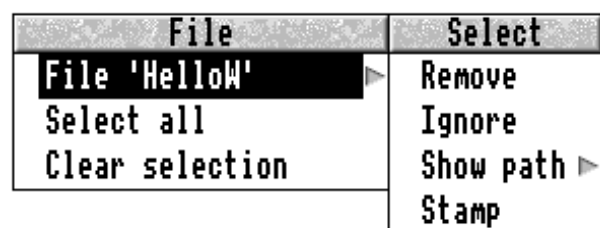
```
idefs::IDEDisc4.$.C.Programs.General.o.HelloW
```

If you haven't got an `o` directory, an error message will occur.

Click **Menu** over the source window to display the menu shown which provides various options for manipulating the source files.

Before you can carry out an operation from this menu, you must select which file or files you wish to apply it to. There are several ways of doing this:

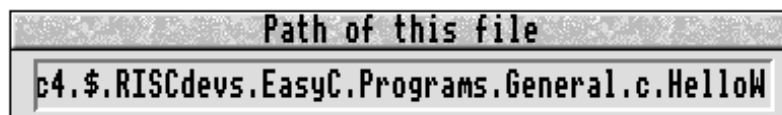
- To select a single file, click **Select** on it or position the pointer over it when you click **Menu**.
- To select a group of files, click **Select** or **Adjust** on the first one, and **Adjust** on each of the others in turn. Use **Adjust** to deselect files.



File

This option opens a sub-menu which provides options which affect the specified file or selection if there is one.

- **Remove** deletes the files or files from the source window. Please note that actual files are not deleted from disc.
- **Ignore** causes the file or files to be shaded and ignored when compilation takes place i.e. they are not compiled.
- **Show path** displays the full pathname (or the last 45 characters) of the selected file. This option is only available if a single file is selected.



- **Stamp** updates the date and time stamps of the file or files. The use of this option is described in the chapter The Make facility.

Select

This will select all the files in the source window.

Clear selection

This will de-select all the files in the source window. You can also do this by clicking Select on the source window background.

Object window

The window entitled **Object** shows the names of the object files which are linked when you click on **Link**. You can link object files automatically placed in this window by the compiler, or drag in other object files of your own from disc. Object files should be located in a directory called `o`.

The linker generates executable code with the same name as the first object file but places it in the parent directory of the `o` directory. So if the object code is called:

```
idefs::IDEDisc4.$.C.Programs.General.c.HelloW
```

The executable file generated will be called:

```
idefs::IDEDisc4.$.C.Programs.General.HelloW
```

Click **Menu** over the object window to display the menu shown which provides various options for manipulating the object files.

Options on this menu are exactly the same as those described previously for source files.

Compile

The **Compile** option compiles the sources files in the source window and generates an object file named after the first source file. Any errors found during compilation are reported in a separate error window. If the error window is closed during compilation, it will be automatically re-opened when compilation is complete.

Link

The Link option links the object files in the Object window with the chosen libraries and generates an executable machine code program named after the first file in the object window. Any linker errors are displayed in the error window.

Stop

This option may be used to stop the compiler or linker before it has completed its task.

Create

This option creates a 'make' file based on the files in the source and object windows. Refer to the chapter The Make facility for full details.

Make

This option 'makes' a program specified by the 'make' file in box to the left of the **Make** button. Please refer to the chapter *The Make facility* for full details.

Dialogue box status information

During compilation and linking the Easy C dialogue box displays useful information about the operation being carried out and its progress. If you wish you may close or iconise the dialogue box during compilation or linking. If you close the dialogue box, it will be automatically re-opened when the operation is complete.

- **Status** gives information about the operation being carried.
- **File** shows the name of the file currently being compiled or linked.
- **Line** shows the number of the current line being compiled.
- **Errors** shows the number of errors found during compilation. By default the error limit is 20, after which the compilation will be stopped. If any errors are reported, the compiler will not generate object code so you will have to fix them before continuing.

Less serious than errors are *Warnings* which notify the user that non-portable or undesirable code has been found during compilation. Warnings do not stop the compilation, and object code is still generated. It is good programming practice to correct any warnings reported.

The least serious errors reported are *Infos* (Informationals). These notify the user of minor errors or idiosyncrasies and can normally be ignored.

The horizontal slider bar shows visually how much of the current file has been compiled as a percentage of the total file. Please note that it does not take the size of 'included' files into account.

Easy C main menu options

Click **Menu** over the Easy C dialogue box (not over the Source or Object windows) to display the main menu. All the settings made from this menu may be saved using Save choices on the icon bar menu described later in this chapter.

Throwback

If this option is selected, Easy C will pass errors directly to a separate text editor which supports the *Throwback* facility. The editor will automatically load the source program, and position the caret on the offending line for so that it can easily be corrected. The exact behaviour depend upon the actual text editor you use.

When Throwback is activated, Easy C does not display errors in its own error window.

A number of editors support Throwback, including **DeskEdit**, **Zap** and **StrongEd**.

If you do not have a suitable editor loaded then this option will be shaded and unavailable. Please note also that you may need to activate throwback using an option in the editor.



Auto link

If Auto link is selected, Easy C will automatically link the source file after it has been compiled. The source file is linked to any object files in the object window and to the chosen libraries. The executable file generated will be named after the source file and placed in the parent directory of the c directory. If Auto link is selected, the Link option in the main dialogue box is shaded and unavailable.

Auto link is a useful option for beginners who wish to compile and link single files with the minimum of fuss.

Auto run

If Auto run is selected, Easy C will automatically link the source file after it has been compiled, and then run the executable file. Operation is identical to the Auto link option described above.

Auto run is ideal for beginners since it allows single C programs to be compiled, linked and run all in one simple operation.

Show errors

Show errors re-opens the error window after it has been closed. It is shaded and unavailable if no errors were reported during the previous compile or link operation.

Make

The options on the Make dialogue box are described fully in the chapter *The Make facility*.

Libraries

The Libraries sub-menu allows you to choose the libraries with which your object code is to be linked during the link operation. At least two libraries are supplied with Easy C, and by default they are both selected.

STDClib (STANdard C Library) contains all the standard ANSI functions such as printf, fopen, strcat. These are described in the chapter ANSI functions. You must always link to this library even if you do not use any ANSI functions.



ROSLib (RISC OS Library) contains a number of functions which allow you to access RISC OS. These functions are RISC OS specific so if your code uses them, it will be non-portable i.e. it will only run on Acorn RISC OS computers. ROSLib functions are described in the chapter RISC OS functions. You need only link to this library if your code uses any ROSLib functions, but there is no significant penalty linking to this library even if your program does not use ROSLib functions.

Please refer to the *Easy C release notes* for details of any other libraries supplied with this package, and to *Appendix A* for details of how to create and add new libraries of your own.

Setup

This opens the Setup Dialogue Box.

Setup Dialogue Box

This allows the Easy C compiler and linker to be configured. Most of the settings in this dialogue box do not normally need to be altered. Any changes made to the settings may be saved using the Save choices option on the icon bar menu described later in this chapter.

Report Warnings

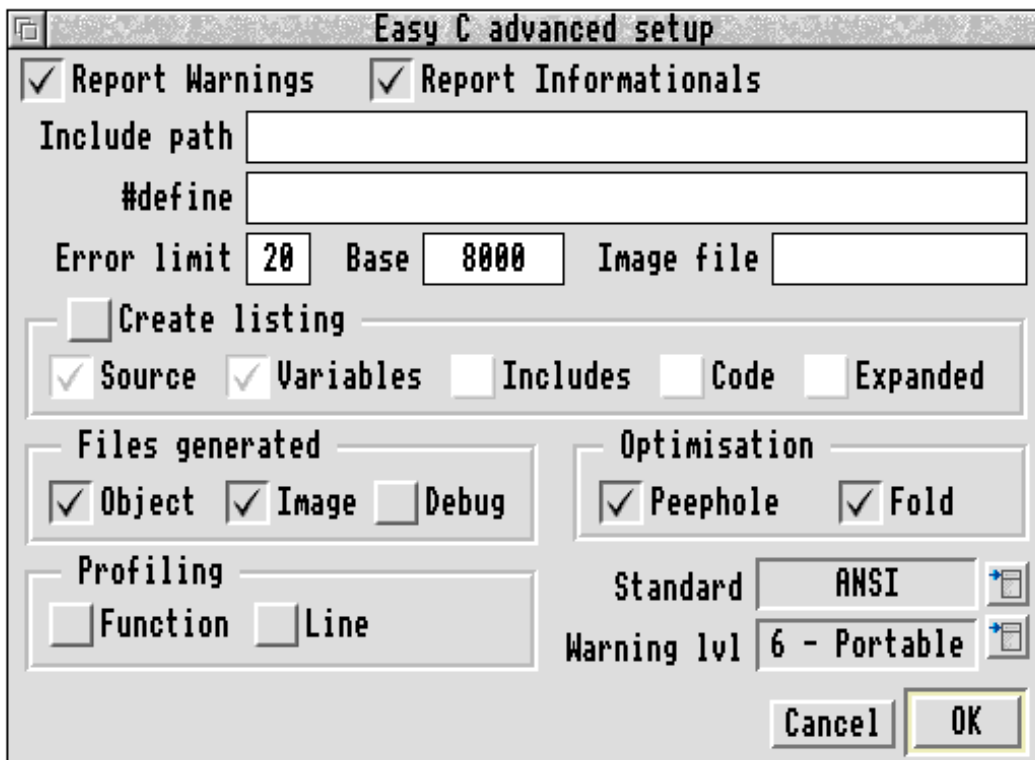
If this option is selected, Warnings are reported to the user in the error window. You may ignore Warnings by de-selected this option.

By default, Warnings are reported.

Report Informationals

If this option is selected, Informationals are reported to the user in the error window. You may ignore Informationals by de-selected this option.

By default, Informationals are reported.



Include path

Include files are those incorporated using the `#include` directive in one of the following two forms:

```
#include <header.h>
```

and

```
#include "header.h"
```

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

1. The current place (not for the form `<header.h>`). The current place is the directory containing the `c`, `h` and `o` directories. Directory `h` contains local header files for the programs in directory `c`.
2. The path specified by Include path.
3. The system search path. By default this is set to `!EasyC` which contains an `h` directory containing all the header files associated with `STDCLib` and `ROSLib`.

You should specify an Include path to direct the compiler to search an additional path for header files. The standard convention is that included files are stored in a directory called `h`. So the include path should be the

parent directory of the h directory containing the header files. So if your header files are in directory:

```
idefs::IDEDisc4$.C.Headers.h
```

the Include path should be set to:

```
idefs::IDEDisc4$.C.Headers
```

You may either type in the pathname or drag the directory into the icon.

Please refer to *Appendix A* for more advanced information concerning header files and include paths.

```
#define
```

This option allows a comma separated list of macros to be defined before compilation. To supply a value to a macro, enclose the macro name and value (separated by a space) in quotes. Examples:

```
START,END,LENGTH  
"START 0", "END 99", "LENGTH 100"
```

Error limit

This option sets the number of errors after which the compiler will terminate. You may set this to any integer value.

Base

This option sets the base address of the executable code generated by the linker. In normal circumstances, this value should not be changed.

Image file

This option allows you to specify the name of the executable image file generated by the linker. Normally it is named after the object file, but in some circumstances you may wish to give it a different name. For example, if you are writing a WIMP application you may want the image file to be called !RunImage.

Create listing

If this option is selected Easy C will create a listing of the source file as it is being compiled. Easy C saves the listing in sub-directory list using the same name as the source file. If directory list does not exist an error message will be displayed. The contents of the listing is determined by the following options:

- If **Source** is selected, the C source code is included in the listing.
- If **Variables** is selected, the listing includes all variables used in the program and how they are assigned to the processor's registers.
- If **Includes** is selected, the source code for any include files is included in the listing.
- If **Code** is selected, the ARM assembler code is included in the listing.
- If **Expanded** is selected, each line of source code is listed twice, with any macro definitions expanded in the second line.

Files generated

These options determine which files are generated by the compiler and linker. In normal circumstances, these options should not be changed.

- If **Object** is selected, the compiler will generate object code. If not selected programs will be compiled and error checked but no object code generated.
- If **Image** is selected, the linker will generate an image file i.e. executable code. If Image is not selected, the linker will link the object files but not generate executable code.
- If **Debug** is selected the compiler and linker will generate code suitable for use with a separate debugger. The code will be less compact than normal, so Debug should not normally be selected.

Optimisation

These options determine whether the compiler carries out program optimisation to produce faster and more compact code. In normal circumstances, these options should not be changed.

- If **Fold** is selected, the compiler evaluates constant expressions at compile time.
- If **Peephole** is selected, the compiler removes redundant instructions in the executable code.

Profiling

This allows the time spent executing functions or specific lines of code to be reported. If this qualifier is used the program must be run using a separate debugger or profiler or an error will be reported. If **Function** is selected the profiler will give the execution time for functions. If **Line** is selected the profiler will give execution time for lines.

Standard

This is used to ensure that programs conform to a specified standard, and is an aid to producing portable code. By default, Standard is set to ANSI which means that the compiler conforms to the ANSI standard.

If you wish to ensure that your code conforms to the original Kernighan and Ritchie definition of C, set the standard to KR. This changes the syntax acceptable to the compiler and generates errors and warnings for any code that does not comply to the Kernighan and Ritchie standard.

If you set Standard to **Extended** the compiler accepts extensions available in the Easy C compiler. This should not be used if you wish to produce portable code. Refer to *Appendix C* for details of these extensions. To compile C++, set standard to C++.

Warning lvl

This option allows you to specify the level of warnings that you want the compiler to report. There are 6 warning levels:

- 1 **Illegal** reports all major illegal constructs in the code that do not prevent the compiler from generating object code.
- 2 **Typos** reports typographical mistakes like missing closing brackets.
- 3 **Changed** reports changes the compiler has made to the code to make it legal. For example, if a macro contains an undeclared variable, the compiler changes it to 0.
- 4 **Illegal OK** reports less serious illegal constructs that do not prevent the compiler creating object code eg. assigning an int to a pointer
- 5 **User** reports minor user mistakes for example using = instead of == in an if statement.
- 6 **Portable** reports constructs that might make the code non-portable.

Each warning level reports the warnings from all the lower levels. The default warning level is 6, Portable, which is the strictest level and reports all warnings. Warnings can of course be completely disabled by de-selecting the Report Warnings option elsewhere in the dialogue box.

Cancel

Click on Cancel to close the dialogue box ignoring any changes made.

OK

Click Select on OK to close the dialogue box and accept the changes made. Click Adjust on OK to apply the changes without closing the dialogue box.

Easy C icon bar menu

To display the icon bar menu, click Menu on the Easy C icon on the icon bar.

- **Info** displays a dialogue box giving information about the version of Easy C you are using. The version number is in two parts; the first is the version of the Easy C front-end, and the second is the version of the compiler. Please quote both numbers in any correspondence with our office about this product.
- **Save choices** saves the settings made in the Easy C dialogue box. These settings will be loaded automatically next time Easy C is run.
- **Quit** exits Easy C.

Chapter 4

The Make facility

Easy C contains a powerful yet easy-to-use **Make** facility which allows you to make a complete multi-file application all in one simple operation. A make facility is essential for large projects that consist of many source and header files, although it may be used with simple C programs too.

If you are developing a large application which uses a number of source files which in turn are dependant on a number of header files, it is very time consuming to re-compile all files after each change that you make. Clearly you need only recompile those files that have changed or whose dependants have changed. To do this manually is prone to mistakes e.g. you might change a header file but forget to recompile one of the source files that is dependant upon it.

The Make facility resolves all these problems by using a '*make file*' which lists all the source files and their dependencies. You need only click on a button for it to make your program. Only those files that have changed or whose dependencies have changed (such as header files) are recompiled and then everything is linked. The Make facility recognises which files have changed by checking their date and time stamps. Easy C has a facility to create a make file automatically for you. You need only tell it which source files are needed by your application, and it will work out the dependencies and create a make file.

Dhrystone

The best way to illustrate the use of the Make facility is to use a real example. On the Easy C disc in directory *\$.Programs* is a directory called *Dhrystone* containing a popular test program for C compilers. This directory contains all the files associated with the *Dhrystone* project, including directory *c* containing the source files *dhry_1* and *dhry_2*, and directory *h* containing the header file *dhry*. Both source files 'include' *h.dhry*, and are therefore dependent upon it.

Please note that you should create a new directory for each project that you wish to make e.g. *Dhrystone*. It should contain *c*, *o* and *h* directories and the make file for that project. Do not mix projects in a single directory.

Creating the 'make file'

To use Make you must first create a 'make file' which defines all the files in the program and their dependencies. The **Create** option on the main dialogue box builds a make file for the files in the source window.

For the *Dhrystone* example, drag the source files *c.dhry_1* and *c.dhry_2* into the Source window. If there are any files already in the Source or Object windows, remember to clear them by holding down **Ctrl** whilst dragging. Now click on **Create** to create the make file. When complete, the pathname of the make file is placed in the icon beneath the Source and Object windows.

The make file it creates is called *Makefile* and is placed in the same directory as the *c* and *o* directories. If a make file already exists in this directory, you will be asked for confirmation before it is overwritten. Normally the object window should be empty when you create a make file. However, if your application needs to link in a separate object file, you should drag it into the object window before using Create.

Please note that when creating a make file, all source files must be in the same *c* directory, otherwise an error will be reported. Any additional object files you require, must be in the corresponding *o* directory.

Using Make

Before making a program, the icon to the left of the Make button must be set to the pathname of the make file. This is done icon automatically by the Create facility, but at the beginning of a session you must drag the make file into this icon (or onto the Easy C icon on the icon bar). Once the make file pathname has been specified, click on Make to make the program. Easy C will now compile and link those files that are needed to make the program.

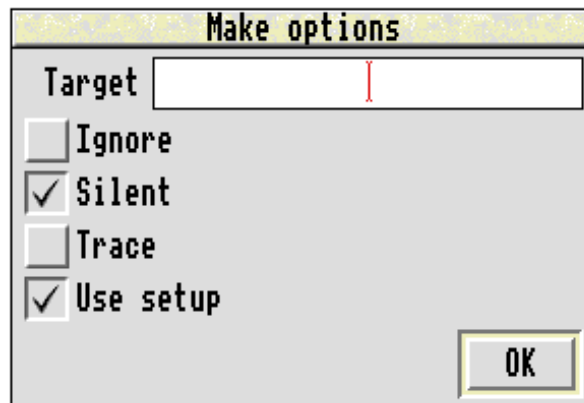
Once *Dhrystone* has been 'made', if you try and make it a second time the message *dhry_1 is up to date* will be displayed. If you modify any of the source or header files, or delete the object or executable files, clicking on Make will intelligently recompile anything that has changed or is missing, and then link the program again. A

quick way of testing this is to 'stamp' one of the source files using the Stamp option on the source window menu, then click on Make.

Make options

The Make option on the main Easy C menu displays a dialogue box from which the make facility may be configured.

- **Target** is used to specify which target you wish to make. In a standard make file there is only one target, so this option can normally be ignored. However, it is possible to edit the make file directly and add other targets, which may be used to make special versions of the program. In this case you should enter the the target name in the icon provided.



- If **Ignore** is selected, the make facility will continue compiling and linking after any errors are reported.
- **Silent** controls whether the make facility echoes in the error window the commands that it generates to compile and link programs. If Silent is selected, the commands are not echoed.
- If **Trace** is selected, the make facility echoes the compile and link commands tin the error window, but does not execute them. This feature is useful for checking that the make facility is generating the correct sequence of commands. The Silent option described above must not be selected for the Trace option to work.
- If **Use setup** is selected the make facility will use the settings from the Setup dialogue box. This means that the settings in this dialogue box will affect the make operation. For example, if List is selected in the Setup dialogue box, a listing will be generated during the compile part of the make operation. This is the default situation, and will be satisfactory for most purposes.

However, there may be situations where you do not wish to change the settings on the Setup dialogue box, but do want the make file to compile or link using a specified set of qualifiers. To do this you will need to de-select Use setup. Normally the setup options are passed to the make facility using the macros CFLAGS and LDFLAGS. If Use setup is not selected, these macros will be blank, so you should set them at the start of the make file with your own settings. For more information, see the sample make file given on the next page.

Make file format

The make file is a text file, so you may load it into a text editor for examination if you wish. Listed overleaf is a commented version of the Dhrystone make file. In normal circumstances you do not need to understand this, but with a little knowledge you will be able to customise it for special applications.

```
# Lines starting with # are comments.
# Makefile generated by Easy C
#
# This makefile builds the following files:
#   c.dhry_1
#   c.dhry_2

# The next line defines a macro OBJJS which lists all the
# objects needed by the program.

OBJJS = o.dhry_1 o.dhry_2

# If c.dhry_1 and c.dhry_2 are modified, they are recompiled
# automatically using default rule cc $(CFLAGS) $* -desk.
# The macro CFLAGS is set by the Setup dialogue box, but may
# be redefined if required.

# The next line defines the target dhry_1 which consists of
# the objects specified by the OBJJS. You may add other targets
# if you wish e.g. to make special versions of the program.
# The action needed to make the target is to link the objects.
# The macro LDFLAGS is set by the Setup dialogue box, but may
# be redefined if required.

dhry_1 : $(OBJJS)
        $(LINK) $(OBJJS) $(LDFLAGS) -desk

# The next two lines state the the files dhry_1 and dhry_2
# need to be recompiled if the header files are modified.
# The recompilation is done automatically by default rules.

o.dhry_1 : h.dhry
o.dhry_2 : h.dhry

#
# End of generated makefile, add other dependencies here
#
```

Note that the filenames in the make file are not specified by full pathnames, but only by the c, o and h directory name plus leafname. This is possible because all the files have the same root pathname. One case where this does not occur is header files from the *Include path*; in this situation, the full pathname must be given.

You may add commands to the make file to carry out special tasks, for example to delete the object files after linking. However, to do this you must tell the command line interpreter where the files are on disc. This is done using the special macro **\$(MPATH)** which expands to the pathname of the directory containing the c and o directories. To delete all object files after linking, add this line after the link line and resave the make file:

```
wipe $(MPATH).o.* ~c
```

Please note that the Create facility does not build a dependency list for system headers (i.e. those in angle brackets <>) since they are not expected to change. If you do intend to change them, you should add these dependencies to the end of the make file.

Chapter 5

Easy C debugger

The Easy C debugger, !EasyDebug provides an easy-to-use yet powerful system for debugging C programs. This latest release is fully StrongARM compatible.

The following features are provided:

- Source and object-level debugging i.e. by line of C source or by ARM instruction.
- Run, Stop, Continue and Single step programs.
- Source listing including powerful Find facility.
Display source with memory address of code.
- Stack trace showing functions called to reach current line.
Options to step up and down the function stack.
- Display local and global variables available at current line.
Edit values of variables.
Expand pointers, structures and unions.
Expression evaluator.
- Set and unset Breakpoints.
Update variable value on breakpoint (watchpoint).
Conditional breakpoints.
Breakpoints can execute simple functions.
Save breakpoints.
- Memory editor with various display modes.
Show register values.
- Profiling facility displays time taken to execute functions.
There are a number of limitations with WIMP programs - see later.

Getting started

To use the debugger you must first compile your program using the debug option provided in the Easy C setup dialogue box. An example program DebugEx is supplied in directory Programs.General.c. This comprises two source files, DebugEx1 and DebugEx2.

Load Easy C and drag both these source files to the source window, ensuring it is cleared of all previous files. Open the Setup dialogue box by choosing Setup from the main menu or by pressing F5. Now select the option Debug and deselect the option Peephole.

Peephole optimisation is deselected because it removes stack frames and can cause the debugger to give misleading information. You can still debug a program with peephole optimisation, but some function calls might be misleading.

Click on **Compile** to compile the source files. Please note that when the Debug option is selected, Easy C generates debug files in sub directory *debug*. An error will be given if this directory doesn't exist. When compilation is complete, click on **Link** to generate the executable program. The executable program with debug code is given a special filetype and icon.

Source level or object level debugging

Normally you should compile and link your programs with the Debug option enabled. This allows you to debug your programs at source level i.e. you will be able to see the lines of C source code in the debugger and single step through them, set breakpoints at them etc.

If the Debug option is enabled for linking but disabled for compiling, you will only be able to use the debugger at object level i.e. you will only be able to single step, set breakpoints etc. at object code level.

If the Debug option is disabled for linking as well, you will not have access to function names.

When debugging at object level, the operation of the debugger changes as follows:

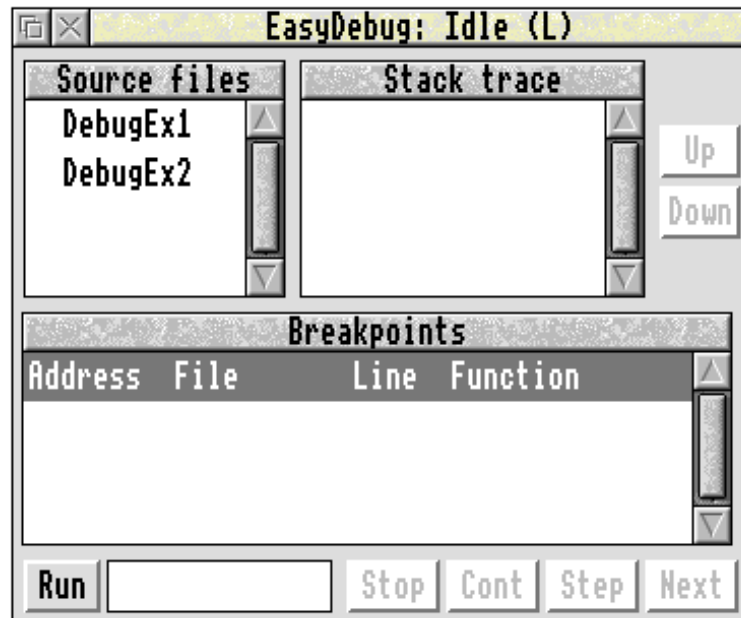
1. The debugger starts automatically in the memory editor.
2. The memory editor is set to **Code** mode and is Active.
3. Single stepping can only be set to Instruction.

Running the debugger

Double-click on the executable file *DebugEx1* to run the debugger, or run the debugger and drag *DebugEx1* to the icon bar icon. Two windows are displayed; one is the main debugger window shown below, and the other is a task window which displays any textual output from the program and accepts any textual input from the user.

Please note that you must have a copy of *Edit* (or any task window provider) loaded. If you have compiled your program using peephole optimisation, a warning will be given at this stage (see previous page). You will also be warned if insufficient memory for your program has been allocated for the wimplot.

The main debugger window contains three separate 'panes'.



- **Source files** - This pane lists the sources files for the current program which were compiled using the Debug option.
- **Stack trace** - This displays a traceback of the program's stack. It will be blank until the program has been run and subsequently stopped.
- **Breakpoints** - This lists the breakpoints that are set. At this stage it will be blank.

Running the program

The debugger is now in the 'idle' state and the options listed below are available. The function key presses shown next to each command below may be used if the main debugger window has the input focus.

Run (F1)

Click on **Run** to run the program. This example program is a simple loop which increments a variable, calls a function and prints the result. The values printed are displayed in the task window. The writable icon next to Run may be used to pass command line arguments to the program.

Stop (F2)

Click on **Stop** to stop the program that is running. When the program stops, the following things occur:

1. The *Source files* pane indicates the current source file with a tick. This is the file in which the program was stopped. The *Source listing window* is opened listing the current source with the current line highlighted. This is the line at which the program was stopped.
2. The *Stack trace pane* is updated with a traceback of the program's stack. The current function is indicated with a tick. This is the function in which the program was stopped.
3. The *Variable list window* is displayed, listing the local and global variables active at the current line. Any variables in the Variable values window are updated.

When a program has stopped, you can do one of the following:

- **Cont (F3)** - When the program has been stopped (either by the Stop button or by a breakpoint or when single stepping), the Cont button continues program execution. Note that the Variable list window automatically closes.
- **Step (F4)** - When the program has been stopped (either by the Stop button or by a breakpoint), the Step button causes the the debugger to single step the program. It steps one line (or one ARM instruction if required—see main menu options) and follows function calls.

After each step the program is stopped and the source, stack trace and variables windows are all updated as described for Stop above.

- **Next (F5)** - Click on Next to step to the next source line. As Step above, but any function calls are skipped over. Any breakpoints within the functions skipped, are not activated. Note that although the debugger skips the function calls, the functions are still executed.

Status information

The title bar displays the current debugger State. The possible states are:

- **Idle** - The program has not yet been run.
- **Running** - The program is running
- **Stopped** - The program has been stopped either by the Stop button or by a breakpoint.
- **Aborted** - The program has stopped in such a way that the user cannot continue. This may be due to a programming error or at the end of the program.
- **Complete** - The program has exited.

The letter in brackets after the state indicates whether the debugger is in Line (L) or Instruction (I) mode when single stepping. In Line mode the debugger steps an entire line of source code; in Instruction mode the debugger steps a single ARM instruction. This mode can be changed from the main debugger menu option Step, or with the keypress Ctrl S.

Source files pane

The pane labelled *Source files* in the top left corner of the main debugger window lists the source files present in the program. Only source files for which there is an associated debug file are displayed so a source file must have been compiled with the **Debug** option enabled in Easy C.

If you click **Select** on a source file a tick will appear next to it and the Source listing window is opened containing a listing of the source file. There can only be one Source listing window at any one time, so choosing another source replaces the previous one.

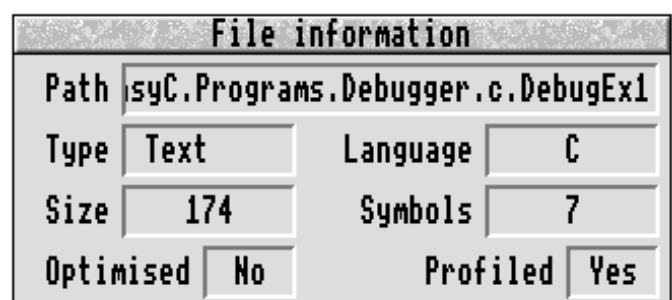
Click **Menu** over a source file to display the source files menu.



File

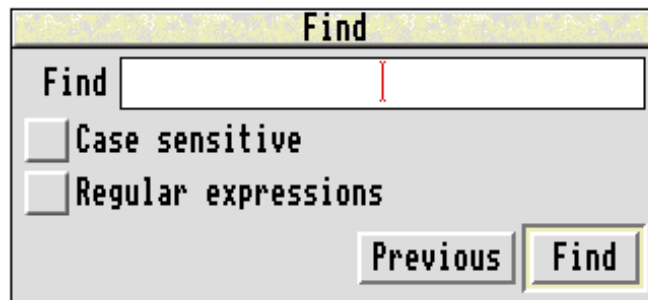
This option displays a dialogue box giving information about the specified source file i.e. the source file over which Menu was clicked.

- **Path** is the full pathname of the source file.
- **Type** is the file type of the source file (usually Text).
- **Language** is the language that the source file was written in. Currently this can only be C.
- **Size** is the size of the source file in bytes.
- **Symbols** is the number of symbols contained in the file.
- **Optimise** indicates whether the program was optimised when compiled.
- **Profile** indicates whether profiling was enabled when the program was compiled.



Find

This options lets you search all the source files in the source file pane for the name of a function.



Type the name of the function you wish to find in the box labelled **Find** or click on **Previous** to display the last function searched for. Now press **Return** or click on **Find** to locate the first occurrence of the function name. If the function name is found, the relevant line is highlighted in the source listing window and the *Found* dialogue box is displayed. If the function name cannot be found, a beep is sounded.

Click on **Next** to find the next occurrence of the function name, or **Previous** to find the last occurrence. A beep is sounded if no more occurrences can be found.

By default the Find facility makes no distinction between upper and lower case characters, but you can specifically request it to match case by clicking on **Case sensitive**.

If **Regular expressions** is selected, the Find facility supports a limited form of regular-expression notation. A *regular expression* specifies a set of character strings to match against - such as *"any string containing digits 5 through 9"* or *"only lines containing uppercase letters."* A member of this set of strings is said to be matched by the regular expression.

Full details of regular expressions are given at the end of this section.

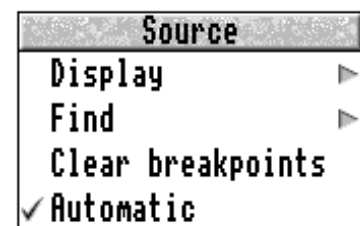
Source listing window

This is opened when you choose a source file from the Source files pane in the main debugger window. It is used for display purposes only—you cannot edit the source file. When the debugger is in the 'stopped' state, the line at which the program was stopped is highlighted in this window.

You may set and unset breakpoints by double-clicking **Select** over the required lines. A bullet character is displayed at the start of a line to confirm a breakpoint is set.

Click **Menu** over the window to display the menu shown.

- **Display** provides a submenu with two options. If Line numbers is selected, each line of the source code is given a line number. If Addresses is selected, each line of the source code is given the address at which the compiled code is located. You can use the memory editor (described later) to examine the code at any address.
- **Find** allows you to search for an object in the source code. Please see opposite for details of how to use Find.
- **Clear breakpoints** clears all the breakpoints in the source file. This operation cannot be undone.
- If **Automatic** is selected, the debugger will automatically open the Source listing window whenever a program is stopped. If it is not selected, and the window is closed using the close icon, it will remain closed unless reopened from the Open window option on the main debugger menu.



Stack trace

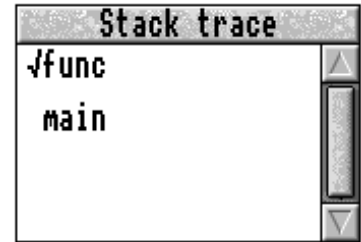
The *Stack trace window* shows a traceback of the program's stack. It is updated each time the program is stopped (either by the Stop button or by a breakpoint or when single stepping).

The first function in the stack trace window is the function at which the program was stopped. Each

subsequent function is the 'calling' function. The final function in the stack trace window is main.

You can click on any function in the list to go to that function. Alternatively you may step up and down the stack using the buttons **Up** and **Down**. Click on **Up** to step up the stack i.e. down the stack trace window towards main. Click on **Down** to move down the stack i.e. up the stack trace window.

Clicking on a function or using Up and Down places a tick next to the chosen function and updates the source and variable windows.

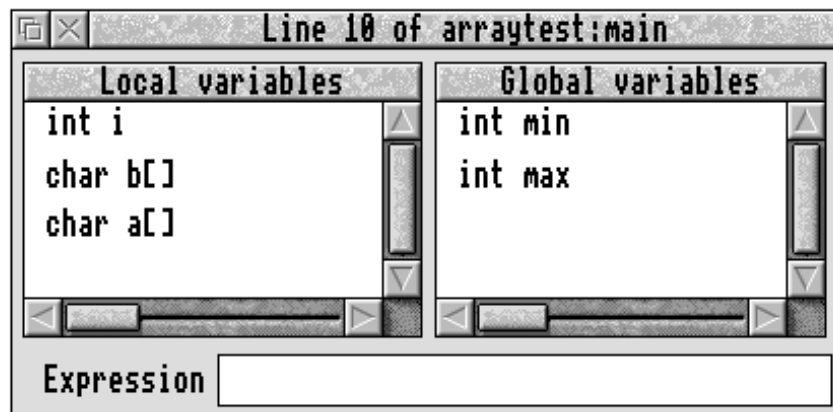


The stack trace facility allows you to easily step from any line in a program to the calling function, and so on until you reach the main() function. At each stage the variables that are declared and the values of those variables can be examined.

Note—When single stepping into a function the debugger stops at the first instruction of the function. This is before the program has built a new stack frame entry for the function so the stack trace cannot be relied upon until the user has actually entered the function. What the stack trace window will display is the name of the current function (the one being stepped into) at the top of the window followed by the function that was second on the stack. By clicking Step again, the stack trace is corrected.

Variables

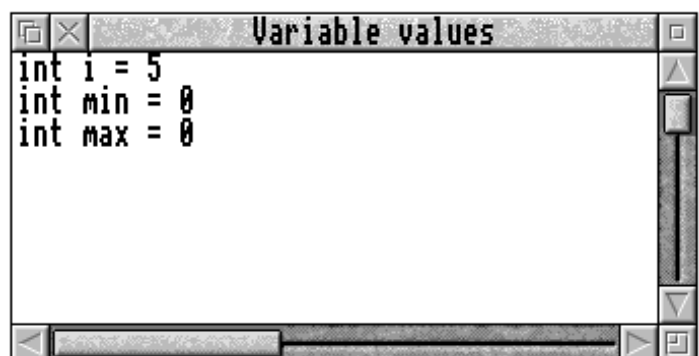
The Variable list window is displayed whenever the program is stopped (either by the Stop button or by a breakpoint or when single stepping). It is also displayed whenever you change function in the Stack trace window. The Variable list window is always automatically closed when the program is running. The Variable window lists the global and local variables available to the user at the point that the program stopped. The global variables include those which are available everywhere in the program (extern) and those which are only available in the current file. The local variables are those which are only available in the current function.



You can display the values of any variables by clicking on them. They become ticked, and the Variable value window opens showing the values of the variables selected.

The Variable value window may be left open at all times and is updated every time the program is stopped. To remove a variable click on its name in the Variable list window to remove the tick.

If the variable is an unstructured type, you can change its value by clicking on the value in Variable value window to display the caret. The value may then be edited, and the change committed by pressing **Return**.



Expanding structures and pointers

If the variable in the Variable value window is a structure or pointer it can be dereferenced further by double-clicking on its name. If it is a structure it will be 'opened' and further details displayed. If it is a pointer the value to which it points will be displayed. You can double-click **Select** on an open variable to close it again.

If the variable is an array of unstructured types it is displayed in its entirety. If it is an array of structured types it is displayed one element at a time. You can change the element displayed by clicking on the index value (between [and]) and editing it. Then press **Return** to reopen the array at the correct index. You may also edit the value of array elements.

Expression evaluation

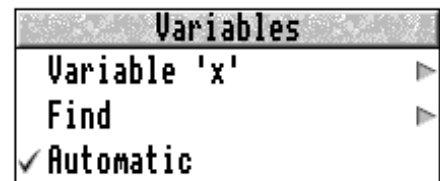
The bottom icon in the Variable list window may be used to evaluate an expression. The result is displayed in the Debug output window. To evaluate an expression type it into the icon and press **Return**. Expressions can use any of the variables in the window and a range of operators. See the section *Expressions* at the end of this chapter for more details.

This is useful if you want to view a complete structure or array on one go, or to look at the contents of a pointer. Expressions are buffered so you can return to earlier expressions without typing them in again. An expression is not buffered if it is identical to a previous expression.

If you click **Menu** over the Expression icon a menu of the last twenty expressions is displayed. Click **Select** on an expression to evaluate it. Alternatively, clicking in the Expression icon so that it has the caret, allows you to step to other expressions in the buffer using the Up and Down arrows. When you have found the one you require, press **Return** to evaluate it (editing it first if you wish).

Variable list menu

Click **Menu** over a variable to open the menu.



Variable

This displays a dialogue box displaying information about the chosen variable i.e. the variable over which Menu was clicked.

- **Name** is the full name of the variable (it may be abbreviated in the menu)
- **Type** is the type of the variable.
- If the variable is a structure, **Tag** is the tag of the structure.
- **Size** is the size of the variable in bytes
- **Storage** indicates where the variable is stored. It may be stack, register or static.
- **Address/Register** is the address of the variable or the name of register in which it is stored.

Variable information			
Name	x		
Type	int		
Tag	Not Applicable		
Size	4		
Storage	register	Address	v1

Find

This lets you search for variables in the list. When found the variable and its value is displayed in the Debug output window. See page 28 for general details of how to use Find.

Automatic

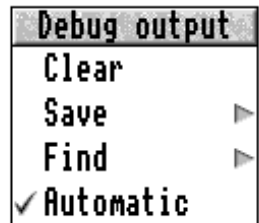
If this is selected the debugger will automatically open the Variable list window whenever a program is stopped. If Automatic is not selected, and the window is closed using the close icon, it will remain closed unless reopened from the **Open window option** on the main debugger menu.

Debug output

The Debug output window is used by the debugger to display information requested by the user. This can either be the value of expressions evaluated in the Variable list window, or the output from the printf function which has been executed at a breakpoint.

Click **Menu** over the window to display the menu shown.

- **Clear** clears the text from the window.
- **Save** saves the contents of the window as a text file using the standard save box.
- **Find** allows you to search for text in the window. Refer to page 23 for details of how to use Find.
- **Automatic.** If this option is selected, the debugger will automatically open the Debug output window whenever something is output into it. If Automatic is not selected, and the window is closed using the close icon, it will remain closed unless reopened from the Open window option on the main debugger menu.



Breakpoints

The Breakpoints pane in the centre of the main debugger dialogue box displays a list of currently active breakpoints. These are set by double clicking in the Source listing window or by using the Set option described below.

By default the program will stop when it reaches one of the set breakpoints, allowing you to examine the variables available. Afterwards you may continue running the program or single step from the breakpoint. A breakpoint is implemented by replacing one of the program instructions by a special instruction which causes the program to stop. A breakpoint is always set at an address in the program space.

Breakpoints			
Address	File	Line	Function
000085A4	DebugEx1	7	main

Each line in the breakpoint pane corresponds to one breakpoint, and gives the following information:

1. The address at which the breakpoint was set.
2. The file in which the breakpoint was set.
3. The line at which the breakpoint was set.
4. The function in which the breakpoint was set.

You may select breakpoints for editing and deletion. To select a single breakpoint, click **Select** over it (this removes any existing selection). To select a group of breakpoints, click **Select** over the first one, and **Adjust** on each of the others in turn. Use **Adjust** to deselect breakpoints.

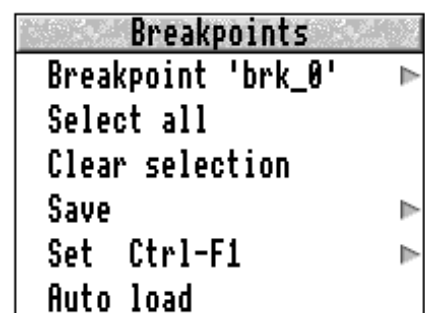
Click **Menu** over the breakpoint pane to display a menu which provides various options for editing the selected breakpoints. If there is no selected breakpoint, the menu options will apply to the breakpoint under the pointer.

Breakpoint 'brk_0'

If there is a single breakpoint selected, this option provides a submenu of options for manipulating the specified breakpoint.

If there is more than one breakpoint selected, this option changes to Selection and provides a submenu of options affecting all the selected breakpoints.

Each time a breakpoint is created it is given a 'tag' which acts like a global integer variable to the debugger, and can be given a value. The tag is of the form *brk_n*, where *n* is a numeric value which is incremented for each subsequent breakpoint created e.g. *brk_0*, *brk_1*, *brk_2* etc.



Edit

This option provides a dialogue box allowing the behaviour of the breakpoint to be changed. This option is not available if there is a selection of breakpoints.

- **Tag** is the tag name for the breakpoint. Each breakpoint is given a default name of the form `brk_n`, but this may be changed to any name you choose.
- **Value** is the value of the tag. By default it is set to 0, but it may be given any integer value.
- If **Stop** is selected the program will stop when the breakpoint is activated, if not selected, the program will not stop. However, the Variable value window will be updated each time the breakpoint is activated. this allows you to monitor the value of any variables in this window without stopping the program. A breakpoint set in this way is often called a *Watchpoint*.
- If **Exec before** is selected the code in the writable icon will be executed when the breakpoint is reached but before any conditional check is made (see overleaf). The executable code allows variables and breakpoint tags to be set and their values printed. A variable is set with the set statement:

```
set count = count + 1
```

A variable is printed using the printf statement following the standard C format:

```
printf("variable x = %d\n", x)
```

The output is displayed in the Debug output window, which will normally open automatically for the output.

Multiple statements can be executed by separating them with semicolons:

```
set brk_0=brk_0 + 1;printf(x = %d\n", x)
```

- If **Conditional** is selected, the breakpoint reached will only be activated if the condition evaluates to True (non zero). In which case the Stop option determines whether the program is stopped. If the condition is False, the program continues running, ignoring the breakpoint. If Conditional is not selected, any breakpoints reached will always be activated.

The conditional expression may contain both variables and breakpoint tags. Examples:

```
brk_0 == 100
```

- If **Exec after** is selected, the code in the writable icon will be executed only after a conditional breakpoint has evaluated to True. The format of the code is the same as for Exec before.
- **Conditional, Exec before** and **Exec after** may be used together to good effect. The following breakpoint settings, will cause the breakpoint whose tag is `brk_0` to become activated the fourth time it is reached. The value of the variable `x` is then printed.
 - **Exec before:** `brk_0 = brk_0 + 1`
 - **Condition%al:** `brk_0 == 4`
 - **Exec after:** `printf("Variable x = %d", x)`



Clear

This option permanently removes the selected breakpoints.

Ignore

This option causes the selected breakpoints to be temporarily suspended. Ignored breakpoints are shaded in the breakpoint pane, and can be resumed by clicking Ignore again.

Select all

This option selects all breakpoints.

Clear selection

This option clears the current selection.

Save

This option saves the current breakpoints to a text file using the standard Save box. The saved file can be reloaded at any time by dragging it into the breakpoints pane.

Reloading a breakpoints file sets the breakpoints in the file being debugged. This is useful to preserve the breakpoint states over a number of runs of the debugger, and saves them having to be reset each time. This is only possible if the source files have not changed. An error is given if you update your source files and then attempt to load an old breakpoints file.

Set

This opens a dialogue box allowing a breakpoint to be set. Normally breakpoints are set by double clicking on the required line the Source listing window. However, this menu allows the user to have a lower level control over setting breakpoints.

- Choose **At function** to set a breakpoint at the first instruction of the function specified in the writable icon.
- Choose **At address** to set a breakpoint at the address specified in the writable icon (in hex).
- Choose **At line** to set a breakpoint in the specified file, at the specified line.

Auto load

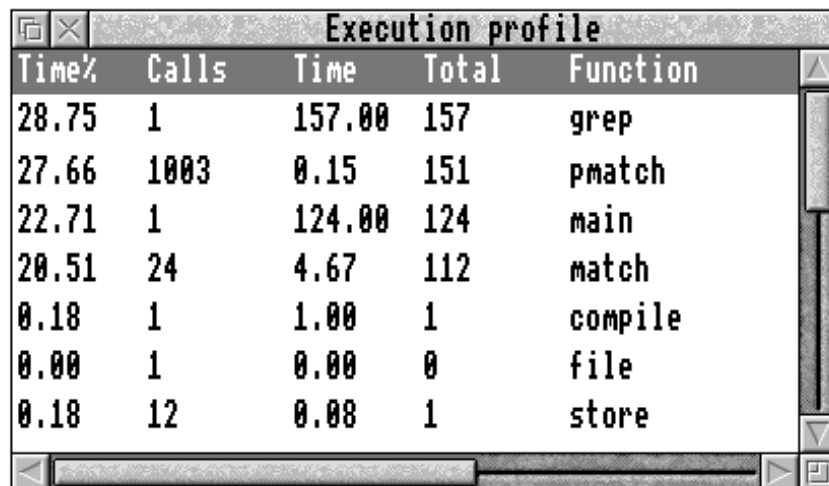
If this option is selected, the debugger will automatically attempt to load a breakpoint file when it is run. This saves having to load a breakpoint file by dragging to the breakpoint pane. The breakpoint file must be called *BrkPoints*, and should reside in the same directory as the executable file being debugged.

The Profiler

This powerful facility allows you to obtain an execution profile of the program while it is running. The profile is a sorted table showing the time taken for functions to execute. Using this information it may be possible to improve the performance of your programs. This profiler is function based only i.e. it reports the time taken for functions to execute.

To use the profiler programs must be compiled using the **Profiling/Function** option on the Easy C Setup dialogue box. If some files are compiled with profiling on and some with it off only those with it on will be profiled.

If the program was compiled with profiling enabled, the window shown below will be displayed when you run the program in the debugger.



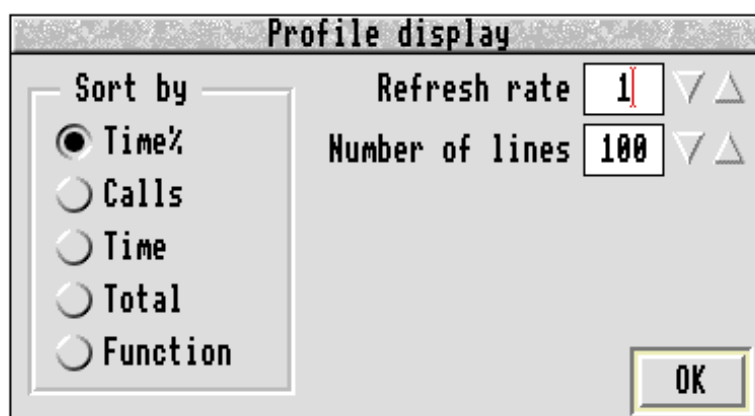
Time%	Calls	Time	Total	Function
28.75	1	157.00	157	grep
27.66	1003	0.15	151	pmatch
22.71	1	124.00	124	main
20.51	24	4.67	112	match
0.18	1	1.00	1	compile
0.00	1	0.00	0	file
0.18	12	0.08	1	store

Each line in the window corresponds to one function, and gives the following information:

- **%time** percentage of the total time spent in this function
- **calls** number of calls to this function
- **time** average time spent in a call to this function in centi-secs
- **total** total time spent in this function in centi-seconds
- **function** function name

Click **Menu** over the profiler window to display the profiler menu.

- **Save** saves the profile information in CSV format suitable for export to other applications.
- **Display** opens a dialogue box (overleaf) which allows the profile display to be modified.
- **Sort by** allows you to choose the sorted order of the functions in the profiler display.



- **Refresh** rate is the rate in seconds at which the profile display is updated.
- **Number** of lines is the number of lines in the profile display. It is used to limit the size of the profile.
- **Find** allows you to search for a function in the profile. When found the profile display is scrolled until the function is visible. Please refer to page 28 for general details of how to use Find.
- **Stop** suspends the profile.
- **Continue** continues profiling after it has been suspended using Stop.

Main menu options

Click **Menu** over the main debugger window (not over the Source files, Stack trace panes or breakpoints), to display the main menu.

Easy Debug	
Open window	▶
Info	▶
Step	Ctrl-S ▶

Open window

This displays a submenu from which all the debugger windows can be opened.

Normally, all the windows are opened automatically when required. The options described below are useful if any of the windows have been closed, and the Automatic option for the window is not enabled.

- The **Memory editor** is described fully later.
- The **Variable list** and **Variable values** windows can only be opened if the program has been run and stopped (either by Stop or by a breakpoint).
- **Profile** opens the Profile window. This window is normally opened when the program is loaded. The Profile window can only be opened if the program has been compiled with the Profiling option.
- **Source** opens the Source listing window for the source file that is ticked in the Source files pane.
- **Debug** output opens the window in which the debugger displays various information.

Windows	
Memory editor	F6
Variable list	F7
Variable values	F8
Profile	F9
Source	F10
Debug output	F11

Info

This option opens a dialogue box giving information about image being debugged.

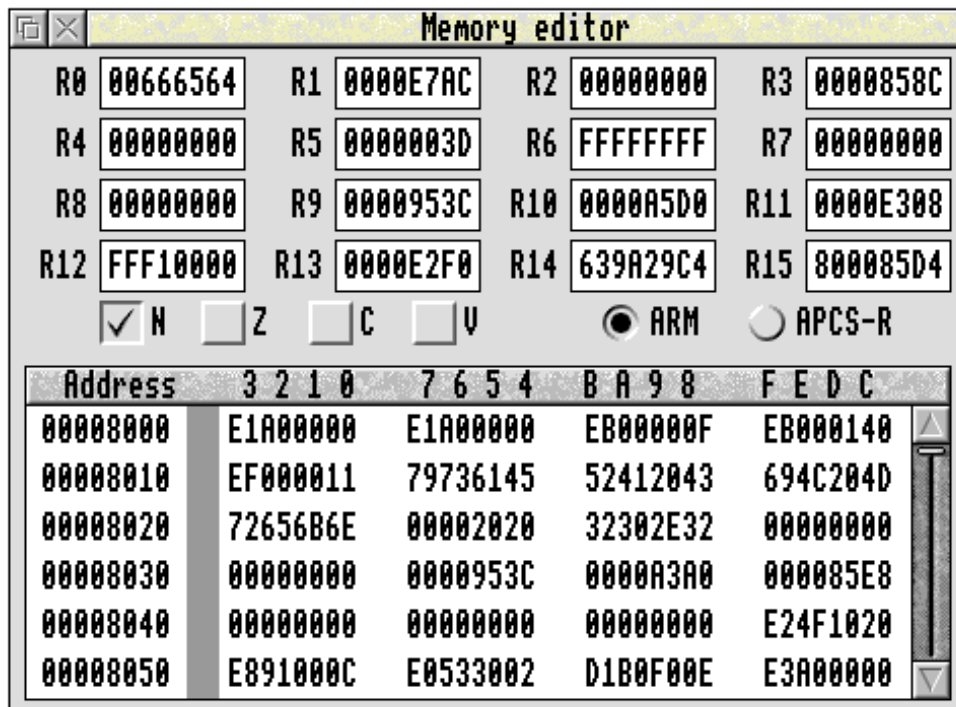
Step

This option opens a submenu allowing you to choose the step size when single stepping.

- **Line** causes the debugger to single step one C source line at a time.
- **Instruction** causes the debugger to single step one ARM instruction at a time.

Memory editor

The memory editor is an extremely powerful window allowing the user to examine and modify memory and registers of the program being debugged. It should only be used by experienced users as it has the potential to corrupt the program environment.



The window consists of a register list and a scrolling pane. The register values are displayed in writable icons and are in hex. You can modify these by clicking on them, getting a caret, editing the value and pressing return. Any register may be modified including the program counter, frame pointer and stack pointer.

Warning - This is dangerous.

Below the register list is a display of the four processor status flags. These are writable and take immediate effect.

Warning - This is dangerous

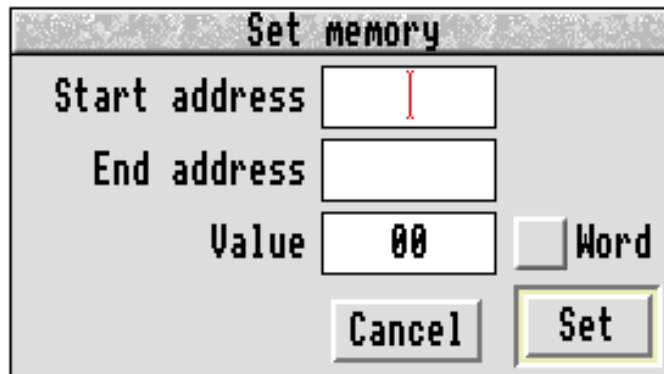
The options **ARM** and **APCS-R** control how the register display and disassembly are shown. In ARM mode the registers are displayed as R0-R15. In APCS (ARM Procedure Calling Standard - RISCOS) mode the names of the registers are a1-a4, v1-v6, sl, fp, ip, sp, lr, pc. This applies to the disassembly listing as well. The scrolling pane contains the memory dump. This can be scrolled to display all of the memory allocated to the program being debugged. The display mode can be modified from the menu option **Mode**:

- **Word** displays memory in 4 byte words (16 bytes per line)
- **Text** displays memory in text format (32 bytes per line)
- **Byte** displays memory in byte format (16 bytes per line)
- **Code** displays memory in disassembled instruction format

If **Word** or **Byte** mode is set the memory may be modified by clicking on the word or byte to modify, editing it and pressing **Return**. If **Code** mode is set a breakpoint may be set by double clicking on the address field.

Memory editor menu options

- **Follow.** If the pointer is placed over an address in **Word** mode clicking on will display memory at the address under the pointer. In **Code** mode if the pointer is placed on a branch instruction clicking on **Follow** will display code at the branch address.
- **Push, Pop.** These operate on a stack basis. **Push** works like **Follow** except it pushes the address currently displayed on a stack. The address may be redisplayed by clicking on **Pop**. This is useful for following a pointer or a BL instruction.
- **Goto** allows the dump to be moved to a particular address specified in hex.
- **Set memory** opens the dialogue box shown and allows a range of memory to be set to a value.



The 'Set memory' dialog box contains the following fields and controls:

- Start address:** A text input field with a red cursor.
- End address:** A text input field.
- Value:** A text input field containing '00'.
- Word:** A checkbox that is currently unchecked.
- Buttons:** 'Cancel' and 'Set' (highlighted with a yellow border).

Memory	
Follow	
Push	
Pop	

Mode	▶
Goto	F5 ▶
Set memory	F6 ▶

Automatic	
Active	F7
Refresh	F8

- **Start address** and **End address** specify the range of memory to be set to **Value**. If **Word** is not selected each byte in the range is set to the value specified. If **Word** is selected, each word in the range is set to the value specified. The value should be in decimal unless preceded by & when it should be in hex.
- **Automatic** makes the debugger automatically open this window when the program has stopped.
- **Active** makes the memory editor active. This means that it will keep up to date with the activities in the program being debugged. What this entails depends on the mode of the dump window. If the mode is **Code**, then the display will show the last two instructions to be executed by the program. This is useful for single stepping instructions. The current instruction is displayed in black and the last executed instruction in grey. If the mode is not **Code**, then the dump will be refreshed every time the program stops. This is useful for watching an area of memory for changes.
- **Refresh** refreshes the dump window and makes it up to date.

Icon bar menu

Click **Menu** over the EasyDebug icon bar icon to open the icon bar menu.

- **Info** displays a dialogue box giving information about the version of EasyDebug you are using. Please quote the version number given in any correspondence with the publisher about Easy C++.
- **WimpSlot** allows you to allocate memory from the WimpSlot for the program you are debugging. If there is insufficient memory, a warning will be given.
- **Save choices** saves various settings in the debugger so that they are used automatically next time it is run. Settings saved include the automatic status of windows and the single step mode.
- **Quit**, exits EasyDebug application, but not the debugger module.

Easy Debug	
Info	▶
WimpSlot	▶
Save choices	
Quit	

Expressions

Expressions may be used in the Variables list window and the breakpoint Edit dialogue box.

Variable list window

The *Expression icon* allows you to type an expression to be evaluated and displayed in the Debug output window. If the expression is an aggregate type (struct/union or array) the whole value is printed. If the expression is a 'pointer to char' or an 'array of char', the value is printed as a character string. Only variables in the current scope may be printed, including local variables in the current function, static variables in the current file, and global variables in other files. Examples:

- `count` an integer
- `count * 2` multiplication
- `my_struct` a structure
- `my_struct.start` a struct member

Breakpoint Edit dialogue box

In the *breakpoint Edit window* expressions may be used in the **Condition** icon and in the **Exec before** and **Exec after** icons.

Condition expressions evaluate to an integral value. If this value is non-zero then the condition is said to be *True*. Condition expressions will usually involve relational operators such as `==`, `>`, `<` etc.

Exec before and **Exec after** expressions can contain the commands `set` and `printf` to change the value of a variable and print out values respectively. There may be multiple expressions in the code separated by semicolons.

The syntax of the `set` command is:

```
set <expression> = <expression>
```

The left expression must be an lvalue (must have an address) and the right expression must match the type.

The `printf` command is similar to the C run time library call.

```
printf ( <format-string> { , <expression> } )
```

The format string contains the normal conversion characters (`%d`, `%x`, `%e`, etc) but only the simple form is supported. This means that you cannot do things like `%-12.4f`.

One extension to the normal C `printf` is the `O%t` format which says to print the struct or array in the expanded form (like the expression in the variables list window).

The output from the `printf` is to the debug output window.

In these expressions, the 'tag' of a breakpoint may be used as a local integer variable.

Examples:

Conditions:

```
fred == 4
count < 10
brk_0 > 5
```

Exec before and Exec after:

```
set brk_0 = brk_0 + 1
printf("The value is %d\n", ptr->count)
printf("%d\n",brk_1) ; set brk_1 = brk_1 * 2
```

Syntax of expressions

The syntax of supported expressions in BNF is:

```
procedure ::= statement
           | procedure ";" statement
statement = "set" expression "=" expression
           | "printf" "(" string { " , " expression } ")"
expression = logical_or
logical_or = logical_and
           | logical_or "||" logical_and
logical_and = bitwise_or
           | logical_and "&&" bitwise_or
bitwise_or = exclusive_or
           | bitwise_or "|" exclusive_or
exclusive_or = bitwise_and
           | exclusive_or "^" bitwise_and
bitwise_and = equals
           | bitwise_and "&" equals
equals = relational
       | equals "==" relational
       | equals "!=" relational
relational = shift
           | relational "<" shift
           | relational ">" shift
           | relational "<=" shift
           | relational ">=" shift
shift = add_sub
       | shift ">>" add_sub
       | shift "<<" add_sub
add_sub = mult_div
       | add_sub "+" mult_div
       | add-sub "-" mult_div
mult_div = unary
       | mult_div "*" unary
       | mult_div "/" unary
       | mult_div "%" unary
unary = "*" unary
       | "&" unary
       | "!" unary
       | "-" unary
       | "~" unary
       | array_struct
array_struct = primary
           | primary "[" expression "]"
           | primary "." identifier
           | primary "->" identifier
primary = "sizeof" "(" expression ")"
        | "sizeof" primary
        | identifier
        | integer
        | floating
        | "(" expression ")"
```

Regular expressions in Find

Regular expressions (RE's) are built from the following single-character RE's:

- c Any ordinary character not listed below matches itself.
- \ Backslash. When followed by a special character, the RE matches the "quoted" character. A backslash followed by one of <, >, (,), {, }, or |, represents an operator in a regular expression, as described below.
- . Dot. Matches any single character except NEWLINE.
- ^ As the left most character, a caret (or circumflex) constrains the RE to match the left most portion of a line. A match of this type is called an "anchored match" because it is "anchored" to a specific place in the line. It loses its special meaning if it appears in any position other than the start of the RE.
- \$ As the right most character, a dollar sign constrains the RE to match the right most portion of a line. The \$ character loses its special meaning if it appears in any position other than at the end of

	the RE.
\<	This sequence constrains the one-character RE immediately following it only to match something at the beginning of a "word"; that is, either at the beginning of a line, or just before a letter, digit, or underline and after a character not one of these.
\>	This sequence constrains the one-character RE immediately following it only to match something at the end of a "word."
[c...]	A non-empty string of characters enclosed in square brackets matches any single character in the string. eg. [abcxyz] matches any single character from the set abcxyz. When the first character is a caret (^), then the RE matches any character except NEWLINE and those in the remainder of the string. For example, [^45678] matches any character except 45678. A caret in any other position is interpreted as an ordinary character.
[[c...]	The right square bracket does not terminate the string if it is the first character (after an initial ^, if any) in the bracketed string. In this position it is treated as an ordinary character.
[l-r]	The minus sign, between two characters, indicates a range of consecutive ASCII characters to match. For example, the range [0-9] is equivalent to the string [0123456789]. Such a bracketed string of characters is known as a character class. The - is treated as an ordinary character if it occurs first (or first after an initial) or last in the string.
d	Delimiter character. The character used to delimit an RE within a command is special for that command (for example, see how / is used in the g command, below).

The following rules and special characters allow for constructing RE's from single-character RE's:

A concatenation of RE's matches a concatenation of text strings, each of which is a match for a successive RE in the search pattern.

*	A single-character RE, followed by an asterisk (*) matches zero or more occurrences of the single-character RE. Such a pattern is called a closure. For example, [a-z][a-z]* matches any string of one or more lower case letters.
\{m\}	A one-character RE followed by \{m\}, \{m ,\}, or
\{m ,\}	\{m ,n\} is an RE that matches a range of occurrences
\{m ,n\}	of the one-character RE. The values of m and n must be non-negative integers less than 256;
\{m\}	matches exactly m occurrences;
\{m ,\}	matches at least m occurrences;
\{m ,n\}	matches any number of occurrences between m and n, inclusively. Whenever a choice exists, the RE matches as many occurrences as possible.
\(...\)	An RE enclosed between the character sequences \(and \) matches whatever the unadorned RE matches, but saves the string matched by the enclosed RE in a numbered substring register. There can be up to nine such sub-strings in an RE, and parenthesis operators can be nested.
\n	Match the contents of the nth substring register from the current RE. This provides a mechanism for extracting matched substrings. For example, the expression \(.*\)\1\$ matches a line consisting entirely of two adjacent non-null appearances of the same string. When nested parentheses substrings are present, n is determined by counting occurrences of \(starting from the left.
//	The null RE (//) is equivalent to the last RE encountered.

Limitations

EasyDebug is a fully functional debugger. When debugging normal (non WIMP) programs it operates exactly as expected. However, when debugging WIMP programs it must interact with the program and the Desktop. The RISC OS WIMP has not been designed to cope with stepping through a running program. This is evident by observing the restrictions on calling Wimp_Poll in certain circumstances. For example, when output is directed to a sprite, or in the middle of a message sequence requiring acknowledgement.

The debugger works by controlling the program being debugged by a series of commands. The program being debugged is running in a TaskWindow and therefore a simple non WIMP program may multitask. As with all WIMP based programs, the debugger gains control of the processor when other programs call Wimp_Poll. This is the only way that this can happen. So, when the program being debugged is in a stopped state (say at a breakpoint, or during single stepping), the taskwindow is continually calling Wimp_Poll, allowing other tasks to run, including the debugger which may send further commands to it.

When debugging a WIMP program, this action of calling Wimp_Poll when the program has stopped can affect the WIMP itself because it may occur in the middle of a sequence of lines where Wimp_Poll is prohibited. This is the reason for some of the restrictions presented here.

The debugger does not interfere with the messages of the program being debugged, it communicates using its

own message passing routines, not through `Wimp_SendMessage`. However, if you attempt to single step through a sequence of lines which are `Wimp_Poll` restricted, then the program will not behave as it would outside the debugger.

The restrictions while using the debugger can be summarised as:

- 1 Do not single step through a series of lines between receiving a message with event type 18 (`User_Message_Recorded`) and the sending of any acknowledgement. This is because the WIMP expects the program to acknowledge the message before calling `Wimp_Poll`.
- 2 In a redraw loop, the program will behave differently under the debugger than outside. If the redraw loop contains calls to `Wimp_RedrawWindow` when running under the debugger, the debugger will do the rest of the redraw loop for you and store the results. When you then call `Wimp_GetRectangle`, the debugger will return the stored values. The upshot of this is that the screen will be cleared to background completely on the first call to `Wimp_RedrawWindow` as opposed to each rectangle being done on each iteration of the loop.
- 3 Although the debugger will deal correctly with stepping through code which redirects output to a sprite, it will not deal with a program which claims the output vector itself. If you try to step through code which does this, the computer will appear to crash because the vector will be claimed through calls to `Wimp_Poll`.
- 4 The code being debugged must conform to the APCS-R standard (ARM Procedure Call Standard - RISC OS) in order to be debugged. The debugger expects the registers to be used as specified in the standard. If they are not (for example, the frame pointer register, `fp`, not being `R11`, etc) the debugger may enter a loop preventing the WIMP from being polled and effectively hanging the machine.

Appendix A

Using Easy C++ from the command line

Easy C++ may be used from the command line instead of, or in conjunction with, the desktop front-end system described throughout this user guide. There are three main reasons why you might wish to run C from the command line instead of the desktop:

1. To access the Easy C Library facility, which cannot be used from the desktop front-end.
2. To use batch files for carrying out complex sequences of compile and link commands.
3. To free extra memory for long programs. The front-end application uses about 200K of memory which can be saved by quitting it, and then compiling and linking from the command line.

Getting started

To use Easy C from the command line, run it in the usual way, then press **F12** to go to the command line. You may wish to save memory by quitting the Easy C application from the icon bar first. Loading Easy C also loads the C module called *EasyC*, which contains the compiler, linker and library facility. You can get help about this module by entering:

```
*help easyc
```

Help on the Compiler, Linker and Library facility can be found using the following commands:

```
*help cc
*help link
*help library
```

These show the general command syntax, optional qualifiers, and the default parameters for the commands.

Compiling a simple C program

The following instructions tell you how to compile, link and run a simple C program from the command line. It assumes that you have loaded the system in the manner described above, and are in the directory of your hard disc to which you have installed the program.

1. Change the current directory to the examples directory, by typing:

```
*dir $.Programs.General
```

or the equivalent pathname if you are using a hard disc system.

2. To compile the example program *HelloW*, type:

```
*cc HelloW
```

The file *HelloW* is a C program stored in sub directory *c* of the current directory. Notice that we do not specify the directory name since the compiler will automatically look in directory *c* for it. If all is well, a message will be displayed indicating that the compilation is complete and there were no errors.

3. The compiler produces object code for the program *HelloW* in sub directory *o* of the current directory. The next step is to link this to the standard library of functions.

```
*link HelloW
```

Note again that we do not need to specify directory *o*. During linking, C will access the standard library and header files, and generate an executable program called *HelloW* in the current directory. The * prompt will re-appear when linking is complete.

4. To run *HelloW*, type:

```
*HelloW
```

and the program will print *hello, world* on the screen.

5. Steps 2 to 4 above could be incorporated in one command, thus:

```
*cc HelloW -run
```

A few other examples are provided in directory *c* for you to try.

The compiler

The general form of the compiler command is:

```
*cc [filename],... [qualifiers]
```

Qualifiers are keywords that modify the operation of the compiler, and are described in detail later in this chapter.

By default C compiles source files in sub directory *c* of the current directory and produces object code (in Acorn Object Format) which is placed in sub directory *o* of the current directory using the same filename as the source. During compilation, any errors in the source code are reported and only when all errors have been corrected will the object code be generated.

Naming conventions

In the example above where *\$.Programs.General* is the current directory, Easy C compiles the source program *\$.Programs.General.c.HelloW* to produce the object file *\$.Programs.General.o.HelloW*. This naming convention is similar to that provided in many other C systems, and should be suitable for most users. However, it may be altered by modifying the system variables *C\$Source* and *C\$Object* using the **set* command. *C\$Source* specifies which sub directory of the current directory contains the C source code, and *C\$Object* specifies which sub directory of the current directory the object code will be saved in.

The default filename

If the compiler is invoked without specifying a source name, C uses the name specified by the system variable *C\$CompileFile*, which is set by default to *HelloW*. So typing:

```
*cc
```

will compile *HelloW* in the usual way. You can change the default source filename for example:

```
*set C$CompileFile program
```

Compiling multiple files

A number of files may be compiled in one operation by specifying their names separated by commas after the *cc* command. For example:

```
*cc file1,file2,file3
```

This command will generate object files *o.file1*, *o.file2* and *o.file3*.

Include files

Include files are those incorporated into a C program using the *#include* directive at the start of a source file. Include files are generally headers for libraries, but may also be users own include files. By convention, header files are placed in sub directory *h*; the '*h*' may be specified before or after the filename in the *#include* directive. Both of the following examples include the header file *stdio*, but for maximum portability, the first form should normally be used.

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

The C preprocessor handles two forms of *#include* directives:

System headers

If the filename is placed between angle brackets (<>), it indicates that it is a system header file. The order of directories in the search path is:

1. At the path specified by the *-include* qualifier (if there is one). This is stored in the system variable *C\$LocalHeaders*.
2. At the path specified by the system variable *C\$HeaderFiles*. By default, *C\$HeaderFiles* is set to *!EasyC*. Within directory *!EasyC* is an *h* directory containing all the standard system headers.

Local headers

If the filename is placed between double quotes (" "), it indicates that it is a local header file. The order of directories in the search path is:

1. In the local sub directory `h` of the current directory.
2. Paths 1 and 2 specified for system headers above.

If the header filename begins with `$` or `&` then it is said to be rooted. Easy C uses rooted filenames exactly as they are entered.

Error messages

During compilation C generates *Errors*, *Warnings* and *Informationals*. If the compiler cannot understand the source code an error message is displayed and object code is not produced. Compilation stops at the end of the source code, or after a specified number of errors are found (20 by default). Note that once an error has been detected other errors may be reported later as a consequence of it. Warnings and Informationals are for information only and do not prevent object code being produced.

Compiler qualifiers

The compiler is controlled by appending *qualifiers* to the `cc` command. Qualifiers are prefixed by a minus sign `-`, and may be in upper or lower case. A full list of compiler qualifiers may be displayed by entering:

```
*help cc
```

In the listing produced, optional parameters are shown in square brackets [], and the default setting for each qualifier is shown in round brackets (). The vertical bar | is used to separate further modifiers, which may be abbreviated to their initial letters. Notice that many qualifiers may be preceded by the word *no* to turn the qualifier off.

Qualifier	-list
Abbreviation	-l
Default	-nolist

This qualifier creates a listing of the C source file as it is being compiled. The listing contains lines of source code with line numbers, together with any error or warning messages. Fuller control over the contents of the file can be controlled using the `-show` qualifier. To compile *HelloW* and produce a listing, enter:

```
*cc HelloW -list
```

By default, C puts listing files in sub directory `list` of the current directory using the same name as the source filename. If sub directory `list` does not exist, an error message will be displayed.

The name of the default directory for listings is set by the system variable `C$List` which may be modified if required. Alternatively a listing may be directed to any specified file by giving a full pathname, for example:

```
*cc HelloW -list=$.listings.HelloW_l
```

Qualifier	-optimise
Abbreviation	-o
Default	-optimise=all

C carries out *program optimisation* at compile time to produce faster and more compact code. In some situations it may be preferable to switch off all, or individual types of optimisation. This is done with the `-optimise` qualifier. To turn off all optimisation use:

```
*cc HelloW -nooptimise
```

If you wish to compile using just **Fold** optimisation, enter:

```
*cc HelloW -optimise=fold
```

The types of optimisation currently supported are:

- **Fold** Evaluates constant expressions at compile time.
- **PeepHole** Removes redundant instructions in the executable code.

Qualifier	-standard
Abbreviation	-s
Default	-standard=ANSI

This qualifier is used to ensure that programs conform to a specified standard, and is an aid to producing portable code. For example, if you wish to ensure that your program conforms to the original Kernighan and Ritchie (KR) definition of C, use the following:

```
*cc HelloW -standard=KR
```

This changes the syntax acceptable to the compiler, and generates warnings for any code that does not conform. The standards currently supported are:

- **KR** The original Kernighan and Ritchie definition of C.
- **ANSI** The implementation of C as defined by the ANSI standard.
- **Extended** The extensions added in Easy C.
- **C++** The Easy C++ implementation of the language.

Qualifier	-object
Abbreviation	-ob
Default	-object

The object code generated by the compiler is normally saved in sub directory o of the current directory, using the same name as the source filename. You may specify a full object pathname using this qualifier, for example:

```
*cc HelloW -object=$.newname
```

Qualifier	-warnings
Abbreviation	-w
Default	-warnings

Warnings notify the user that the program contains non-portable or undesirable code. In such circumstances, the compiler displays the offending line number followed by the appropriate warning message. Warnings do not stop the compilation, and object code is still generated. Normally warnings are a useful aid to programming, and are particularly useful for producing portable code. However, if too many warnings are given they may become a nuisance, and can be disabled using:

```
*cc HelloW -nowarnings
```

This qualifier can also be followed by a warning level (see page 18):

```
*cc HelloW -warnings=3
```

Qualifier	-informationals
Abbreviation	-i
Default	-informationals

Informationals notify the user of minor errors or idiosyncrasies in the code. Informationals are the least serious errors that C reports and do not stop the compilation. Informationals may be disabled using:

```
*cc HelloW -noinformationals
```

Qualifier	-define
Abbreviation	-d

The -define qualifier allows macros to be defined at compile time. The following example defines macro START:

```
*cc HelloW -define=START
```

A number of macros may be defined as follows:

```
*cc HelloW -define=START,END,LENGTH
```

To assign values to macros use:

```
*cc HelloW -define="START 0","END 99","LENGTH 100"
```

Note the macros include a space and are therefore enclosed in quotes

Qualifier	-debug
Abbreviation	-deb
Default	-nodebug

This qualifier is used to specify that debugging information should be generated for use with a separate source level debugger.

Qualifier	-errorlimit
Abbreviation	-e
Default	-errorlimit=20

The qualifier -errorlimit sets the number of errors after which the compiler will terminate. The error limit may be changed by specifying an integer after the qualifier -errorlimit, for example:

```
*cc HelloW -errorlimit=5
```

If you want the compiler to compile to the end of the source file regardless of the number of errors found, switch off the error limit using:

```
*cc HelloW -noerrorlimit
```

Qualifier	-show
Abbreviation	-sh
Default	-show=source

The -show qualifier controls the listing produced by the -list qualifier. It has seven modifiers which determine what information is included in the listing. For example, if you wish to list assembler code and variables, use:

```
*cc HelloW -list -show=code,variables
```

The modifiers are:

- **Source** The C source code is included in the listing.
- **Expanded** Each line is listed twice, with macros expanded in the second line.
- **Code** ARM assembler code is included in the listing.
- **Variables** Lists the variables used, and how they are assigned to the registers.
- **Includes** The source code for any include files is included in the listing.
- **All** All of the above modifiers are used.
- **None** None of the above; the listing contains only error or compiler messages.

Qualifier	-silent
Abbreviation	-si
Default	-nosilent

The -silent qualifier switches off all output during compilation, except for any error messages.

Qualifier	-includes
Abbreviation	-inc
Default	-includes=" "

The `-include` qualifier allows you to specify the path that C will search for local headers i.e. if you include a local header file using the form:

```
#include "MyHeader.h"
```

To include `$.Programs.General.LocalHead.h.MyHeader` in a program `NewProg`, use the following command:

```
*cc NewProg -include=$.Programs.LocalHead
```

The `-includes` qualifier sets the system variable `C$LocalHeaders`.

Qualifier	-link
Abbreviation	-lin
Default	-nolink

If the `-link` qualifier is specified, C links the program automatically after compilation. So the command `*cc HelloW -link` is equivalent to the following two commands:

```
*cc HelloW
*link HelloW
```

If the `-link` qualifier is used, Easy C does not generate an object file, saving time especially on floppy discs and networks (this is not true when compiling from the Wimp front-end—object files are always generated). The `-link` qualifier may be used when compiling a number of source files, for example:

```
*cc file1,file2,file3 -link
```

Object code produced is named after the first filename specified i.e. `file1`.

Qualifier	-run
Abbreviation	-r
Default	-norun

If the `-run` qualifier is specified, C links and runs the program automatically after compilation. The executable file produced will have the same name as the source file. So the command `*cc HelloW -run` is equivalent to the following three commands:

```
*cc HelloW
*link HelloW
*HelloW
```

Qualifier	-image
Abbreviation	-im
Default	-image

The `-image` qualifier allows you to specify the full pathname of the file to which the final executable program should be written. The `-image` qualifier is relevant only if there is a `-link` or `-run` qualifier on the command line. For example:

```
*cc HelloW -link -image=$.hello
*cc HelloW -run -image=prog
```

If `-image` is omitted, the source filename is used, and is written to the current directory.

Qualifier	-library
Abbreviation	-lib
Default	-library

By default object files are linked with the library specified by `Lib$Library` (set to `STDCLib`). The `-library` qualifier allows you to specify a range of libraries to be linked if the `-link` or `-run` qualifiers are specified. For example:

```
*cc HelloW -link -library=STDCLib,ROSLib
```

Qualifier	-base
Abbreviation	-b
Default	-base=&8000

The -base qualifier sets the base address of the executable code, eg.

```
*cc HelloW -link -base=&A000
```

It is relevant only if the -link qualifier is specified.

Qualifier	-map
Abbreviation	-m
Default	-nomap

The -map qualifier is used to produce a map of the functions linked by the -link qualifier. Map files are normally saved in sub directory map of the current directory with the same filename as the source. If the directory does not exist an error will be displayed.

```
*cc HelloW -link -map
*cc HelloW -link -map=$.mapfile
```

Further details of the -map qualifier are given below in the section on the linker.

Qualifier	-throwback
Abbreviation	-th
Default	-nothrowback

This qualifier causes errors generated by the compiler to be transmitted to a suitable editor using the throwback facility. Editors currently supporting throwback include *StrongED*, *Zap*, *SourceEdit* and *DeskEdit*. This facility is only intended for use from the Easy C front-end in the desktop.

Qualifier	-desktop
Abbreviation	-des
Default	-nodesktop

This qualifier causes the compiler to ignore the usual directory naming conventions, and accept full pathnames for source and object files. This facility is only intended for use from the Easy C front-end in the desktop.

Qualifier	-profile
Abbreviation	-p
Default	-profile=none

This qualifier allows the time spent executing functions or specific lines of code to be reported. If this qualifier is used, the program must be run using a separate debugger or profiler otherwise an error will be reported. There are four modifiers which determine how this qualifier operates:

- **Function** The profiler will give the execution time for functions.
- **Line** The profiler will give execution time for lines.
- **Both** Both of the above will be selected.
- **None** No profile information will be inserted in the code.

The linker

The linker is used to link together one or more object files together with functions from one or more libraries, to produce executable code. It is compatible with object files generated by the C Compiler as well as other compilers and assemblers. The C linker is contained in the same module as the compiler, and is loaded in the same way.

The general form of the link command is:

```
*link filename,... [qualifiers]
```

Qualifiers are keywords that modify the operation of the linker, and are described in detail below.

By default, the linker links the object code in sub directory o of the current directory, and produces executable code which is placed in the current directory using the same name as the object file. It also links to the library

specified by the system variable *Lib\$Library*. By default this is STDCLib; the shared C library. For example:

```
*link HelloW
```

links o.HelloW to the library STDCLib, and produces the executable file HelloW. This naming convention is similar to that provided in many other C systems, but may be altered by modifying the system variables *C\$Object* and *Link\$Image*, which are set by default to o. and "" respectively.

Linking multiple files

To link a number of files together, use:

```
*link part1,part2,part3
```

The commas separating the filenames must be included. The executable code produced by this command is called *part1* (the name of the first object file), and is put in the current directory.

Linker qualifiers

The operation of the linker can be controlled by appending qualifiers to the link command. A full list of qualifiers can be displayed by typing:

```
*help link
```

The format of the listing produced is similar to that described earlier for the compiler.

Qualifier	-image
Abbreviation	-i
Default	-image

Normally the image (executable) file produced by the linker is given the same name as the first object file, and is put in the current directory. The -image qualifier allows you to specify a different path and filename for the executable file, for example:

```
*link HelloW -image=$.helloprog
```

It is useful if you wish to re-link a file without overwriting the previous executable file. It can also be used to prevent executable code from being generated, thus:

```
*link HelloW -noimage
```

In this case the linking will still take place, and any errors reported, but no file will be produced.

Qualifier	-library
Abbreviation	-l
Default	-library=STDCLib

The -library qualifier specifies the library or range of libraries to be linked to the specified object files. Easy C is supplied with two libraries: STDCLib which is the Shared C Library containing all the standard ANSI functions, and ROSLib containing operating system, graphics and sound functions. The Shared C Library is contained in the relocatable module Clib, which must be present for programs to run.

If the -library qualifier is not present, the linker links to the default library STDCLib, for example:

```
*link HelloW
```

To link to a number of libraries, specify their names after the -library qualifier, for example:

```
*link HelloW -library=STDCLib,ROSLib
```

C searches for the libraries according to the path specified by the system variable *Link\$Library* which by default is set to !EasyC.Libraries.

The next section of this user guide describes the commands which enable new libraries to be created and maintained.

Qualifier	-base
Abbreviation	-b
Default	-base=&8000

The -base qualifier sets the base address of the executable code. For example:

```
*link HelloW -base=&A000
```

The default base is &8000. To execute code with a base address other than &8000, you must load it at the new address and use *go to run it.

Qualifier	-debug
Abbreviation	-d
Default	-nodebug

This qualifier is used to specify that debugging information should be generated for use with a separate source level debugger.

Qualifier	-map
Abbreviation	-m
Default	-nomap

The -map qualifier is used to produce a map of the functions linked by the -link qualifier. Map files are normally saved in sub directory map of the current directory using the same name as the object file. The map sub directory name is specified by the system variable *Link\$Map*.

```
*link HelloW -map
*link HelloW -map=$.mapfile
```

Qualifier	-throwback
Abbreviation	-th
Default	-nothrowback

This qualifier causes errors generated by the compiler to be transmitted to a suitable editor using the throwback facility. Editors currently supporting throwback include *StrongEd*, *Zap SourceEdit* and *DeskEdit*. This is only intended for use from the Easy C front-end in the desktop.

Qualifier	-desktop
Abbreviation	-des
Default	-nodesktop

This qualifier causes the compiler to ignore the usual directory naming conventions, and accept full pathnames for source and object files. This facility is only intended for use from the Easy C front-end in the desktop.

The library facility

The library facility in C is used for the creation and maintenance of libraries. It allows you to build your own libraries of functions for special applications. These may be used instead of, or in addition to, the libraries provided with the package. The Library facility allows you to create new libraries, and insert your own object files into them. Object files inserted into libraries are called modules, and may contain one or more functions. You may insert, delete or extract modules from libraries, but you cannot do the same for the individual functions within the modules.

The general form of the library command is:

```
*library [filename] [qualifiers]
```

If the filename is omitted the default library specified by the system variable *Lib\$Library* is used. By default it is the shared library *STDCLib*. The library pathname is specified by the system variable *Link\$Library* and is set to *!EasyC.Libraries*. A list of qualifiers can be displayed by typing:

```
*help library
```

Qualifier	-create
Abbreviation	-c

This qualifier allows a new library to be created.

```
*library NewLib -create
```

This command creates a new library containing no functions.

Qualifier	-verify
Abbreviation	-v

This qualifier verifies the integrity of a library. In addition it displays information about the library, including its date of creation, copyright and identification messages.

```
*library NewLib -verify
```

or for the default library

```
*library -verify
```

Qualifier	-list
Abbreviation	-l

The -list qualifier lists the contents of a library. For example, to list the contents of the default library, type:

```
*library -list
```

Or, to list the contents of the ROSLib library, type:

```
*library ROSLib -list
```

Each function in the library is listed, together with the object module in which it is contained.

Qualifier	-history
Abbreviation	-h

This qualifier controls the library history mechanism. To turn this on, use:

```
*library ROSLib -history=on
```

To turn the history mechanism off, use:

```
*library ROSLib -history=off
```

To display the history of a library, use:

```
*library ROSLib -history
```

Qualifier	-insert
Abbreviation	-i

The -insert qualifier lets you add an object module to a library. This should be in AOF format e.g. generated by the C compiler, another compiler or an assembler. eg. to add the object file *o.utils* to the NewLib library ensure you are in the parent directory of the o directory and type:

```
*library NewLib -insert=utils
```

A number of modules may be added as follows:

```
*library NewLib -insert=utils,utils1,utils2
```

Qualifier	-delete
Abbreviation	-d

The `-delete` qualifier allows you to delete an object module from a library. For example, to delete `utils` module from `NewLib` type:

```
*library NewLib -delete=utils
```

Please note that you cannot delete individual functions, only complete modules.

Qualifier	<code>-extract</code>
Abbreviation	<code>-e</code>

The `-extract` qualifier extracts an object module from a library and places it on disc. The module is not deleted from the library. Use `-extract` to copy object modules from one library to another. For example to extract the `utils` module from the `NewLib` library type:

```
*library NewLib -extract=utils
```

The module will be placed in the current directory.

Qualifier	<code>-copyright</code>
Abbreviation	<code>-co</code>

This qualifier allows a copyright message to be specified for a library, and is displayed whenever the library is listed or verified. It can only be used when a library is created, and cannot be changed afterwards. For example:

```
*library NewLib -create -copyright="Copyright (c) 2026"
```

Please note that the copyright string must be enclosed in quotes if it contains spaces.

Qualifier	<code>-identification</code>
Abbreviation	<code>-id</code>

The `-identification` qualifier allows an identification string to be specified for a library. This string is displayed when the library is listed or verified. The identification qualifier can only be used at the time the library is created, for example:

```
*library NewLib -create -identification="Version 1.00"
```

Appendix B

ARM Assembler

Files compiled using Easy C may contain assembly language in the form of ARM mnemonics and floating point instructions. A piece of assembly language may appear at any point that a function definition may appear in a C program. It is enclosed in the pre-processor lines:

```
#pragma asm
...
#pragma endasm
```

The instruction format is identical to that normally recognised for the ARM. The code is parsed on a line-by-line basis. If more than one instruction is placed on a line, they must be separated by colon characters. The assembler is pre-processed just as a normal C program, so the assembly language programmer may take advantage of the powerful conditional and macro processing capabilities of the C pre-processor. The code produced is in AOF (Acorn Object File) format.

Registers

Register names may take one of a number of formats:

- Normal register names may use the format: R0 or a1 etc.
- The second form is that used in the ARM procedure calling standard.
- The floating point register names may use: F1 or f1, etc.
- The stack pointer may be: R13, sp or SP.
- The link register may be: R14, lk or link.
- The program counter may be: R15, pc or PC.

The procedure call standard names and normal names may be used at will. It is suggested that the procedure call standard name format be used when an external procedure call is made.

Expressions

Where a constant is required in assembly language, a constant expression may be used. The operators which may be included in the expression are normal C constant expressions. Hex numbers are represented by the characters 0x followed by the number, octal numbers by 0 followed by the number. Character constants are enclosed in single quotes. If an immediate value is required and is not loadable in one instruction, an error is given.

Labels

A label is declared by prefixing its name by a full stop. A label declared in this way is not visible outside of the file being compiled. To make it external it must be exported by the EXPORT keyword. This may appear either before or after the definition of the label.

```
.label1
.label2
EXPORT label2
```

In the above example *label2* is exported and may be referenced outside of the file. Any label may be used in a C program as a function. If the label is exported, it may be used in a C program which is linked to the file. A local label may be declared as follows:

```
.123$
```

This label is only in scope in assembly language, and may not be used in C. All the normal instructions may use labels (ADR, LDR etc.), and labels may be targets for branch instructions.

Comments

A comment can be included at any point by prefixing it with a semicolon. Comments always terminate at the end of lines. In addition, C style comments may also be used i.e. the comment is enclosed in /* and */ pairs.

Pseudo instructions

- **IMPORT <label>** import a symbol from another file
- **EXPORT <label>** make a symbol visible outside this file
- **ALIGN** align to a 4 byte boundary
- **DCB <expr[,expr...]>** declare bytes (may be in quotes)
- **DCD <expr[,expr...]>** declare words
- **ADR <reg>,<label>** address of
- **LDR <reg>,<label>** load at address
- **STR <reg>,<label>** store at address
- **LDF <reg>,<label>** load floating at address
- **STF <reg>,<label>** store floating at address

The following instructions are pseudonyms for DCB and DCD:

DEFB, EQUB, DCB, DEFW, EQUW, DCD

Note: all code and data written using the assembler is placed in the `C$$Code` area of the object file and is therefore notionally read-only. If it is desired to declare writable data, it must be done outside the assembler by use of normal C variable definitions.

Software interrupts

The ARM SWI instruction allows the programmer to enter ARM supervisor mode and perform some complex operation under a "software interrupt". This assembler recognises all the SWIs that are present in the operating system (including extension modules). There is no need to define symbols to represent SWIs, they can be invoked by simply quoting their name in the SWI instruction.

For example:

```
SWI OS_WriteS
SWI Wimp_GetRectangle
SWI OS_WriteS+ 'A'
SWI 0x12344
```

Macros

The assembly language is pre-processed by the C pre-processor. This allows macros to be defined using the `#define` pre-processor control line. This is of the form:

```
#define <macro-name>[(parameters)] <replacement text>
```

A macro may take parameters. If it does, these are textually substituted into the line before any lexical analysis is performed. Instructions may be separated on one line by colons. This feature may be taken advantage of, to produce multi-line macros. Conditional assembly may also be performed by using the C pre-processors `#if`, `#ifdef` etc. control lines.

Errors

Syntactic and semantic errors are reported as the normal errors given by the C compiler.

Example program

The following program *Asm_ex* demonstrates the use of assembler in a C program. It is supplied within the programs directory of the CD-ROM.

```
#pragma asm
/* this is the assembly language equivalent of the ubiquitous
   "hello world" program. It prints "Hello World" a number of
   times followed by a number. */

        IMPORT printf                ; imported from run time library

.string                                ; declare a label string
        DCB "Hello World %d\n",0    ; declare some bytes
        ALIGN                        ; align to word

.limit                                ; declare a label (not visible          ; outside)
        DCD 10                      ; declare a word 10

        EXPORT example              ; label example is to be exported
.example                               ; declare the label
        STMFD SP!,{r0-r12 ,link}    ; store regs
        MOV R4,#0                   ; set counter to 0
        LDR R5,limit                ; load limit
.l$                                         ; declare a local label l$
        ADR a1,string               ; get address of string into first arg
        MOV a2,R4                   ; get counter into second arg
        BL printf                   ; call printf
        ADD R4,R4,#1                ; add 1 to counter
        CMP R4,R5                   ; at limit?
        BNE l$                      ; no, do another
        LDMFD SP!,{r0-r12 ,PC}^     ; return

#pragma endasm

main()
{example() ;
```

Appendix C

Easy C language extensions

The Easy C compiler supports 3 simple extensions to the standard ANSI specification. These extensions may only be used if the compiler is switched to Extended mode using the Standard option on the Setup dialogue box, or using the -standard qualifier from the command line.

1. The symbol \$ is allowed in variable names.
2. Binary numbers can be specified by prefixing them with 0b.
3. Easy C supports automatic argument counting for use when writing functions in ARM assembler where you need to know how many arguments are being passed to the function. This facility is used by inserting the keyword auto in the function declaration as follows:

```
function(auto int data, ...);
```

The number of arguments is stored in the top 6 bits of register a1, with the first argument in register a2.

Here are just a few of the other products you will be able to
download from RISCOS.com free of charge
(You can make a donation to help support future product releases but this is entirely voluntary)



This Documentation is © 3QD Developments Ltd 2026
Some of the products shown above might not be available immediately.
Not all products will run on all versions of RISC OS, check the product information page for details.

Other Products list last updated 29/2/26