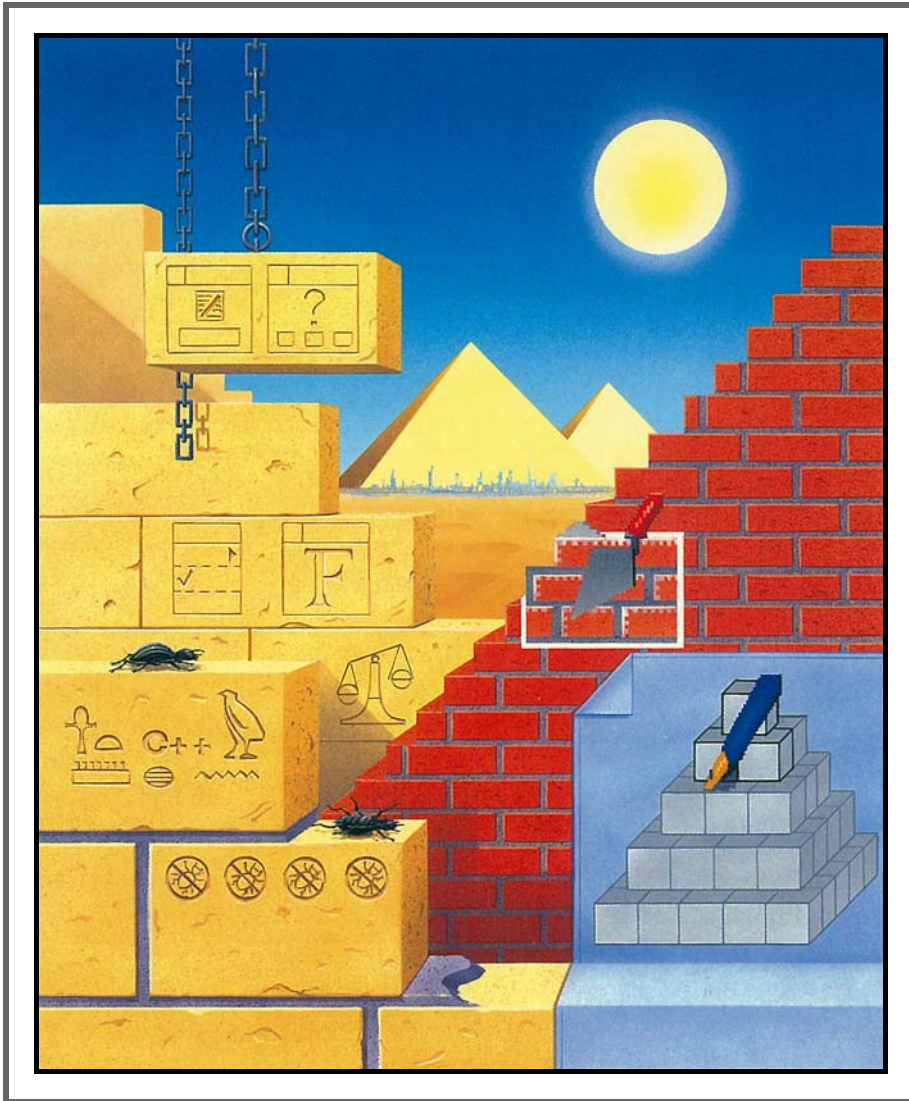


Desktop Tools



Copyright © 1994 Acorn Computers Limited. All rights reserved.

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

Some updates and changes copyright © 2014-2015 RISC OS Open Ltd. All rights reserved.

Issue 1 published by Acorn Computers Technical Publications Department.

Issue 2 published by Castle Technology Ltd.

Issues 3 and 4 published by RISC OS Open Ltd.

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

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

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

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

All trademarks are acknowledged as belonging to their respective owners.

Published by RISC OS Open Ltd.

Issue 1, December 1994 (Acorn part number 0484,230).

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

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

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

Contents

Introduction 1

About this manual 2

Conventions used 3

Part 1 - Getting started 5

Installing Acorn C/C++ 7

Hardware and OS requirements 7

Installation 7

Environment variables and Acorn C/C++ 8

Working with desktop tools 11

Desktop tools 11

Working styles 14

Where to go from here 14

Part 2 - Interactive tools 17

Desktop debugging tool 19

Overview 19

About debuggers 20

Preparing your program 21

Starting a debugging session 23

Specifying program objects 26

Execution control 33

Program examination and modification 41

Options and other commands 46

An example debugging session 50

Make 59

Invoking Make 59

Using Make 60

Makefile format 69

Programmer interface 71

SrcEdit 73

- Starting SrcEdit 73
- SrcEdit menus 74
- Printing a SrcEdit file 88
- Laying out tables – the Tab key 89
- Reading in text from another file 90
- Bracket Matching 90
- Throwback 91
- Saving Options 95
- The SrcEdit icon bar menu 95
- SrcEdit task windows 97
- Keystroke equivalents 100

General features 103

- The Application menu 104
- The SetUp box 105
- Output 107

Part 3 - Non-interactive tools 111

A8Time 113

- Background to pipelines and scheduling 113
- Using A8Time 115
- The SetUp dialogue box 115
- The Application menu 116
- Example output 116
- Command line interface 117

ABC 119

AMU 121

- Starting AMU 121
- The Application menu 123
- Example output 123
- Command line interface 124

DecAOF 127

- The SetUp dialogue box 127
- The Application menu 128
- Example output 129
- Command line interface 129

Diff 131

- The SetUp dialogue box 131
- The Application Menu 132
- Example output 133
- Command line interface 134

Find 135

- The SetUp dialogue box 135
- The Application menu 140
- Example output 140
- Command line interface 141

LibFile 143

- The SetUp dialogue box 143
- Output 145
- Command line interface 147

Link 151

- The SetUp dialogue box 151
- Output 154
- Possible errors during a link stage 155
- Libraries 155
- Generating overlaid programs 156
- Relocatable AIF images 159
- Relocatable modules 160
- Transient utilities 161
- Predefined linker symbols 161
- Command line interface 162

ModSqz 165

- The SetUp dialogue box 165
- The Application menu 166
- Example output 166
- Command line interface 166

ObjSize 169

- The SetUp dialogue box 169
- The Application menu 169
- Example output 170
- Command line interface 170

SID 171

Squeeze 173

- The SetUp dialogue box 173
- The Application menu 174
- Example output 174
- Command line interface 174

UnModSqz 177

- The SetUp dialogue box 177
- The Application menu 178
- Example output 178
- Command line interface 178

Xpand 181

- The SetUp dialogue box 181
- The Application menu 182
- Example output 182
- Command line interface 183

Adding your own desktop tools 185

- The FrontEnd module 186
- Producing a complete Wimp application 187
- The DDEUtils module 202
- SrcEdit 203
- Make 203

Appendices 205

Changes to the tools 207

Makefile syntax 209

- Make and AMU 209
- Makefile basics 210
- Makefile structure 212
- Advanced features 216
- Makefiles constructed by Make 227
- Miscellaneous features 228

FrontEnd protocols 231

- Star Commands 231
- EBNF Grammar of Description Format 231
- WIMP Message returned after a *FrontEnd_SetUp 236

DDEUtils 237

- Filename prefixing SWIs 237
- Filename prefixing *Commands 238
- Long command line SWIs 238
- Throwback SWIs 239
- Throwback WIMP messages 241

SrcEdit file formats 245

- Language File Format 245
- Help File Format 245

Code file formats 247

- Terminology 247
- Byte Sex or Endian-ness 247
- Alignment 248
- Undefined fields 248

AOF 249

- Chunk file format 249
- Object file format 250

ALF 267

- Library file format 267
- Object Code Libraries 270

AIF 271

- Properties of AIF 271
- The Layout of AIF 273
- Zero-Initialisation Code 277

ASD 280

- Order of Debugging Data 280
- Endian-ness and the Encoding of Debugging Data 281
- Representation of Data Types 282
- Representation of Source File Positions 283
- Debugging Data Items in Detail 283

ARM procedure call standard 297

The purpose of APCS 297

The ARM Procedure Call Standard 299

APCS variants 307

C Language calling conventions 309

Some examples 316

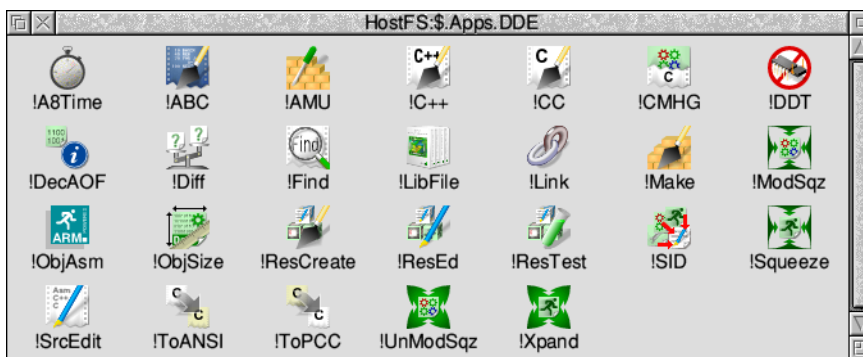
The APCS in non-user ARM modes 318

Index 321

1 Introduction

Acorn C/C++ provides a set of RISC OS desktop applications for programming. These tools interact in ways designed to help your productivity and make the desktop a high quality environment for creating RISC OS applications and relocatable modules from compiled languages or assembler.

The Apps .DDE directory is where the desktop tools reside:



With the exception of the Desktop Debugging Tool (DDT), all these tools are multi tasking RISC OS applications. DDT has to operate outside RISC OS in order to stop it dead at any moment for breakpoints etc., so is windowed but not multi tasking. The desktop tools allow you to:

- edit program source and other text files
- search and examine text files mechanically
- examine some types of binary file
- compile and link programs, relocatable modules and utilities
- assemble and disassemble assembly language programs
- construct programs efficiently under the control of makefiles
- construct linkable libraries
- debug RISC OS desktop applications interactively
- construct resource files for Toolbox applications.

The Acorn C compiler, C++ compiler and Assembler are described in the *Acorn Assembler* and *Acorn C/C++* manuals.

About this manual

This volume is organised into four parts:

- Part 1 – Getting started
- Part 2 – Interactive tools
- Part 3 – Non-interactive tools
- Part 4 – Appendices

Part 1 – Getting started

This part of the manual describes how to install Acorn C/C++ and how to use the desktop tools.

The chapters are:

- *Installing Acorn C/C++*
- *Working with desktop tools*

Part 2 – Interactive tools

This has chapters covering each of the desktop tools which you use with constant interaction as ‘foreground’ tasks. They are the debugger, make and source editor.

The chapters are:

- *Desktop debugging tool*
- *Make*
- *SrcEdit*

Part 3 – Non-interactive tools

This covers the less interactive desktop tools which all have similar interfaces. The first chapter in this part covers the general features common to all the non-interactive tools. The following chapters each describe an individual tool. The last chapter describes how to add your own desktop tools.

The chapters are:

- *General features*
- *A8Time*
- *ABC*
- *AMU*
- *DecAOF*
- *Diff*

- *Find*
- *LibFile*
- *Link*
- *ModSgz*
- *ObjSize*
- *SID*
- *Squeeze*
- *UnModSgz*
- *Xpand*
- *Adding your own desktop tools*

Part 4 – Appendices

This part of the manual gives technical details of the file formats and protocols used in Acorn C/C++.

The appendices are:

- *Changes to the tools*
- *Makefile syntax*
- *FrontEnd protocols*
- *DDEUtils*
- *SrcEdit file formats*
- *Code file formats*
- *ARM procedure call standard*

Conventions used

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

`link options filenames`

This means that you type 'link' exactly as shown, and replace 'options' and 'filenames' by specific examples.

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

Hexadecimal integers are given in uppercase, and preceded by 0x, e.g. 0xFE1. (Not preceded by &, which will be more familiar to users of BBC BASIC.)

Part 1 - Getting started



2

Installing Acorn C/C++

Installing Acorn C/C++ means setting up a suitable disc directory structure. You only need to perform this once to set up a suitable structure.

To use Acorn C/C++ you will need to install it; booting is performed automatically.

This chapter only describes installation. The chapter *Working with desktop tools* explains how to use the desktop tools.

Hardware and OS requirements

The minimum specification of RISC OS system recommended for serious use of Acorn C/C++ is a 4MB RAM machine with a hard disc drive.

A limited subset of features of Acorn C/C++ can be used on a 2MB RAM machine, but its use is not recommended.

The Acorn C/C++ tools and the code they generate are suitable for use on RISC OS 3.10 and later versions and for ARM2 and later processors, including 32-bit versions of RISC OS running on ARM6 and later processors.

Installation

The Desktop Development Environment is supplied as an archive that includes the tools, the example files and PDF versions of the manuals.

Using RISC OS, copy the contents of the archive over the top of the root of your boot disc. This should merge the DDE with your disc image and put everything into the right place. Next, open the directory AcornC/C++.Developer and double-click on !SysMerge. This will update your !System directory to ensure that it contains all the files required to use the tools.

Environment variables and Acorn C/C++

Various Acorn C/C++ operations depend on the correct settings of environment variables. These environment variables are set by the SetPaths application in the AcornC/C++ directory when it is first seen by the filer. You can also run the SetPaths application explicitly at any time to set these variables to their default state. To see all the variables that are set by SetPaths, open the file !SetPaths.SetPaths. Details of the key variables are summarised here, but note that this is not a complete list.

Each desktop tool, when loaded, defines an environment variable of the sort <toolname>\$Dir. The purpose of these variables is to allow each tool access to its application directory. These are not likely to become incorrectly set and cause problems. SrcEdit can be configured with options from its desktop interface, and also from options variables, as described in the chapter *SrcEdit* later in this manual.

DDE\$Path

Purpose: This is set to the name of the directory containing the desktop tools, and is used by Make to start tool interfaces for setting Tool options.

Problems: If DDE\$Path is unset, the Make **Tool options** facility fails with an error mentioning DDE:.

C\$Path

Purpose: This specifies a list of directory names for the C compiler to search for libraries and their headers.

CLib\$Path, TCPIPLibs\$Path, RISCOSLib\$Path

Purpose: These paths specify lists of directory names to be searched for the library object files for CLib, TCPIPLibs and RISC_OSLib respectively.

Hdr\$Path

Purpose: This specifies a list of directory names that can be searched to find standard RISC OS assembly language header files.

Makefiles\$Path

Purpose: This path is used to locate the Shared Makefiles.

LocalRes\$Path

Purpose: This specifies a list of directory names to be searched for resources when using the install target in the Shared Makefiles.

Install\$Dir

Purpose: This is the directory into which files are installed when the install target of the Shared Makefiles is used.

3

Working with desktop tools

This chapter provides an overview of the most productive way to work with the desktop tools to produce your programs. The chapter *Installing Acorn C/C++* describes how to prepare your working environment.

Desktop tools

Acorn C/C++ includes the following tools:

- **DDT** – A windowed debugger for debugging any executable image file, including the !RunImage file of a RISC OS application. DDT presents a windowed interface with RISC OS style controls.
Note that as DDT has to be capable of stopping RISC OS dead at any point in a program, for breakpoints, single stepping, etc, it cannot multitask under the RISC OS desktop.
- **Make** – A desktop application for constructing programs under the management of ‘recipes’ stored in Makefiles. Various types of Makefile can be rapidly constructed using the desktop controls of Make, as well as being executed. This facility for constructing Makefiles is known as ‘project management’ on some programming systems for other types of computer.
- **SrcEdit** – A text editor derived from Edit with many features for constructing program sources and other text files.
- **AMU** – A compact alternative to Make for using, but not constructing, Makefiles.
- **DecAOF** – A utility for examining AOF files output by language compilers or assemblers.
- **Diff** – A text file comparison tool.
- **Find** – A tool for finding text patterns in the names or contents of sets of files.
- **Link** – A tool for constructing usable relocatable modules, program files, etc., from object files produced by language compilers and assemblers.
- **LibFile** – A utility for constructing linkable library files storing general purpose routines for re-use in more than one program.
- **ObjSize** – A utility to measure object file size.
- **Squeeze** – A tool which compacts finished program images so that they occupy much less disc space and load faster.

- **ModSqz** – A tool which reduces the size of relocatable modules.
- **Xpand** and **UnModSqz** – Tools to reverse the effect of **Squeeze** and **ModSqz**.
- **A8Time** – A Cortex-A8 instruction-timing simulator which helps calculate the time required to execute sequences of ARM instructions.
- **SID** – A general-purpose disassembler for ARM binaries.
- **ABC** – A compiler for programs written in BBC BASIC.

Each of the tools listed above is described in more detail in its own chapter later in this volume. The language specific tools are described in the language user guides accompanying this manual.

As well as performing individual tasks, several of the desktop tools cooperate in ways designed to enhance your productivity. An example of this is *throwback*. When a language compiler or assembler detects an error in a program source file, it can cause throwback – opening a SrcEdit window for immediate correction of the offending program line. Another example of cooperation is the ability to drag an output file from one desktop tool to the input of another appropriate desktop tool.

Interactive and non-interactive tools

The desktop tools are divided into two categories – interactive and non-interactive. The non-interactive tools are those that have options set and then are run, without any further interaction with you until the task completes or is halted. The interactive tools are those that operate with constant interaction with you, such as the source editor SrcEdit.

In the list of tools above, the first three (DDT, Make and SrcEdit) are interactive tools, and the rest are all non-interactive. The chapters describing each tool are organised into parts of this manual describing each category of tool. The non-interactive tools all have similar user interfaces, and the features common to all of them are described in the chapter *General features* on page 103.

Entering filenames

Many of the desktop tools require you to specify file or directory names. The interactive tools each have file types that they 'own', which you can double click on in directory displays to start activities. These are:

- **DebugAIF** – execution of one starts a DDT session. Files of this type are displayed in directory displays with the icon:



- **Makefile** – double clicking on one loads it into Make (and may start a Make job). Files of the type Makefile are displayed in directory displays with the icon:

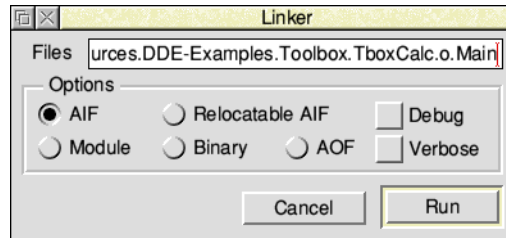


- **Text** – double clicking on one starts a SrcEdit edit.

None of the non-interactive desktop tools own a file type. Input files are specified to these tools by dragging them to their icon bar icons from a directory display or by typing their names into a writable icon in a dialogue box or menu field. When typing filenames into a writable icon, enter absolute filenames such as:

```
adfs::rool.$$.Sources.DDE-Examples.Toolbox.TboxCalc.o.Main
```

To reduce the amount of typing required, any writable icon on a dialogue box that accepts a filename or directory name can be set by dragging a filename from a directory display to it. For example, dragging a filename from a directory display to the **Files** writable icon on the Link SetUp dialogue box adds it to the list of input files already specified:



Many program source files and Makefiles contain filenames, for example in an assembler program line such as:

```
GET ^.h.SWINames
```

RISC OS provides only one current directory, but many tasks (such as assembly processes) can be multitasking, running at the same time. Thus the concept of *work directory* is used in Acorn C/C++. This can be considered rather like a current directory for each task, and file searching is performed relative to this. See the section on each tool to see the way the work directory is set and used by that tool. Most of the simpler tools do not require a work directory.

Working styles

The desktop tools support two main styles of working – *managed* and *unmanaged* development. These differ only in the way you construct your finished programs from sources, not the way you write or debug them, and you can mix and match the two styles as you wish.

Managed development makes use of Makefiles to manage the construction of your finished programs. A Makefile is a ‘recipe’ for processing your sources and linking the object files produced to form the usable program. The tools Make and AMU can both execute the commands in a Makefile running other tools to perform a make job. The tool Make also constructs Makefiles for you, avoiding the need for you to understand their syntax, and making it quick and easy to do this. The main advantages of managed development are: timestamps of files are examined during a make job and no unnecessary reprocessing of unaltered program sources is performed; programs are constructed consistently, following the same recipe each time, even when run by different people. These advantages make managed development the best style for the development of larger programs with source split into several source files.

Unmanaged development makes use of each individual tool directly to process the files as required to construct your programs. This can offer the quickest way of constructing small programs.

When Booting for unmanaged development you have to load each tool that you wish to use, but when Booting for managed development you only need to load Make (or AMU).

When working in either style, it is recommended you place each program project in a separate subdirectory, in the same way that the program examples are arranged. You can place the source, header and object files in suitable subdirectories of the project directory. See the accompanying language specific manuals for more details of subdirectory conventions. Source may be placed elsewhere, but this can make it more difficult to rename or move whole projects to other directories or filing systems.

Where to go from here

If you have studied this chapter in detail you now understand how to construct a simple runnable program from text sources. You may now wish to load various desktop tools and experiment with their use, and there are further chapters that may provide useful general information.

Each desktop tool, such as the text editor SrcEdit and debugger DDT, has a chapter describing it, either in this user guide or in one of the accompanying manuals. If you intend to make much use of any particular tool, its chapter may prove useful reading next.

A large number of the desktop tools are classified as 'non-interactive', and have similar interfaces. The chapter *General features* on page 103 covers the interface features of this class of tool.

Part 2 - Interactive tools



This chapter describes the desktop debugging tool (DDT). DDT is an interactive aid to debugging desktop or non-desktop programs written in compiled languages such as C, Pascal or Fortran. DDT can also be used to debug programs written in ARM assembler using ObjAsm.

DDT can be used on any computer running RISC OS 3.10 or later. It can emulate both 26-bit and 32-bit environments.

Overview

Although DDT can be used to debug desktop programs, and provides a windowed interface, it is not a true multitasking desktop program. This is because DDT has to be able to halt the RISC OS desktop at any point for single stepping, breakpoints etc. This means that its interaction with other RISC OS applications is limited in certain ways:

- When the debugger is active (i.e. when a program is halted under control of the debugger) all other tasks are halted until execution of the program is resumed.

Note: You can always tell when the debugger is active, because the pointer will change to a No Entry sign if you move it outside the debugger's windows:



- Only one application may be running under the debugger at any given time.

The windowed interface of DDT is designed to be easily understood by RISC OS desktop users, and to facilitate this it duplicates many RISC OS features. However, it uses visual details such as unusual colours to act as reminders that it is not operating as a true desktop multitasking program.

Topics covered in this chapter

- the section *About debuggers* introduces the concept of debuggers in general and describes the facilities provided by DDT.
- the section *Preparing your program* describes how to prepare your program for use with DDT.

- the section *Starting a debugging session* describes how to invoke the debugger on your program.
- section *Specifying program objects* describes the way in which various objects in the program you are debugging, such as variable names, procedure names and line numbers are specified.
- section *Execution control* describes how to control execution of a program running under the debugger.
- section *Program examination and modification* describes the debugger's facilities for displaying various objects in the program being debugged and the facilities for changing variable, register and memory contents.
- section *Options and other commands* describes the options in the options dialogue box and other commands which are not covered by any of the previous topics.

About debuggers

This section is aimed mainly at readers who haven't used a program debugger of any sort before. However, others may find it useful reading, as it introduces some of the facilities provided by DDT.

Anyone who has written a program more than about ten lines long has had recourse to debugging techniques: the tracking down and removal of errors. The form this takes depends on many things, not least the language in which the program is written.

Some languages provide primitive debugging facilities of their own. For example ISO C provides the `assert` macro which can be used to ensure a condition is true, as in the following example:

```
assert(i >= 0); /* Ensure following loop is finite */
while (i--) { ... }
```

Some language implementations provide additional debugging facilities. A description of the debugging facilities provided by Acorn's release of C may be found in the accompanying *Acorn C/C++* manual.

Often, however, it is left to the programmer to plant *trace* information in the program itself. For example you might trace the value of the index variable in a while loop as follows:

```
while (i--) { fprintf(tracefile, "i = %d\n"); ... }
```

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

Planting trace information in assembly language programs is more difficult. For example, displaying the contents of all ARM registers is a non-trivial code fragment in ARM assembler.

A debugger enables you to execute your program in a controlled environment where you can stop execution, examine and alter variables, set breakpoints, single step through a program and 'watch' particular variables for changes.

DDT provides the following debugging facilities:

- Start program execution and continue after program execution has been stopped
- Single step program execution, by source statement or ARM instruction
- Stop program execution at a specified program location
- Stop program execution when a specified variable changes its value
- Stop program execution at any time on request
- Trace program execution continuously
- Trace procedure calls
- Trace changes to a specified variable or memory location
- Display source text, symbolic disassembly, variables, registers, memory contents and stack backtrace information
- Alter variable values, register contents or memory contents
- Protect sensitive areas of memory against being accidentally overwritten by your program.

Preparing your program

This section describes how to prepare your program for use with DDT. DDT uses special information in the program being debugged, which provides DDT with information about the source code that generated the program. This information is not automatically included in the output of the compiler. This is mainly for reasons of efficiency; programs which contain debugging information are larger, take longer to compile, and run more slowly than those with no debugging information.

Compiling

You enable the generation of debugging information with the **Debug** option on the compiler SetUp menu. If you are using the compiler from the command line use the `-g` flag to enable debugging information with the Acorn C compiler (other compilers may use different flags, though `-g` is common across a wide range of compilers).

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

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

Note that if you are using an automated program construction tool, such as the Make utility, you may have to delete the object files of the modules you wish to compile with debugging information when you enable the **Debug** option. This is because the modules are not recompiled until the object files are either absent, or out of date with respect to the source files, so you must delete the object files to force recompilation.

Linking

When linking a program to be debugged, you must instruct the linker to include the debugging information generated by the compiler. To do this, enable the **Debug** option on the link menu, or, if you are using the linker from the command line, by using the `-debug` flag.

If you are using Acorn's C compiler to perform the link stage (i.e. without the **Compile only** option enabled on the compiler menu, or without the `-c` flag from the command line) the compiler will automatically instruct the linker to include debugging information if the compiler's debugging option is enabled.

The linker also generates its own debugging information. This debugging information is used by DDT to provide low-level or symbolic debugging facilities. If you do not wish to use source level debugging facilities, you can enable the **Debug** option on the linker without enabling the **Debug** option on the compiler.

Note that !RunImage files compiled or assembled and then linked with Debug enabled are much larger than those produced without debug information. This may require an increase in the WimpSlot size specified in your !Run file, otherwise the following error may be produced at run time:

No writable memory at this address

If you are writing in assembler using ObjAsm you may wish to use the KEEP directive, which instructs the assembler to keep information about local symbols in the symbol table. These will be included in the program when linked with debugging enabled.

You might like to try preparing the following small program for use with the debugger, using the methods described above.

```

1 #include <stdio.h>
2
3 int main(void)
4 {
5     int world;
6
7     for (world = 0; world < 100; world++)
8         printf("Hello, World %d\n", world);
9     return 0;
10 }
```

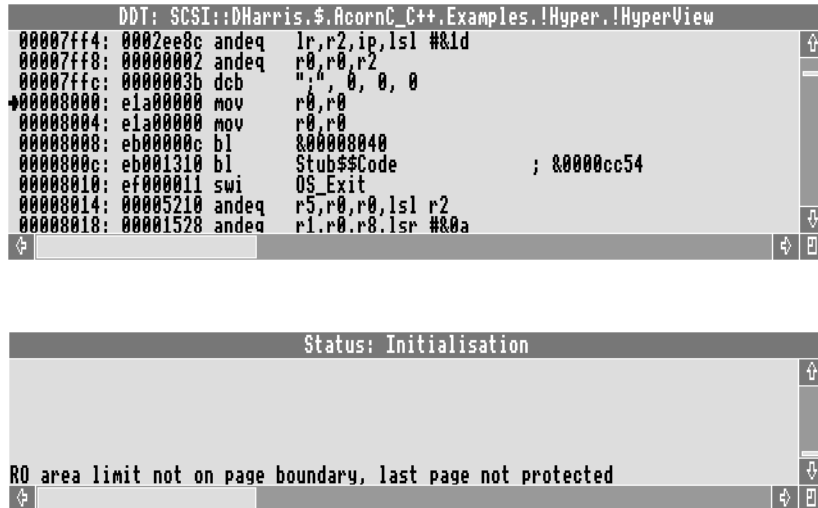
Starting a debugging session

You can start a debugging session in one of the following ways:

- Double click on the !DDT application. This will place the debugger's icon on the icon bar. Then drag the program to be debugged to the debugger's icon. You can drag either a program image or an application directory. If you drag an application directory, the program image within that directory must be called either !Run or !RunImage.
- Choose **Debug** from the debugger application menu. This will produce a dialogue box with two writable icons, one for the name of the application to be debugged, the second for any arguments the application may take. You can specify the program name by dragging an application to the writable icon. When the writable icons have been filled, clicking the **OK** button will invoke the debugger.
- Enter the following *Command:
**DebugAIF program [arguments]*
 where *program* is the name of the program to be debugged, and *arguments* are any command line arguments that program may take. You can enter this command from the supervisor prompt (outside the desktop), from the Shell CLI prompt (obtained by choosing the ***Commands** option on the Task Manager menu) or from a task window CLI prompt.

Try invoking the debugger on the sample program shown at the end of the last section.

Once you have started a debugging session in one of the above ways, two debugger windows will be displayed as follows:



The upper window is the *Context* window. The title bar contains the name of the program being debugged. The Context window displays the source text or symbolic disassembly associated with the current Context or PC location.

When you start a debugging session, the Context window initially displays a symbolic disassembly, like that shown above. This is a disassembly of the run-time system initialisation code. The arrow symbol (→) to the left of the window shows the current PC location. The debugger does not display your source code at this stage because the program has not started executing your code, it still has to execute the initialisation code. Once execution reaches your code (i.e. the first instruction of `main`) your source code will be displayed.

The lower window is the *Status* window. The title bar contains the current status of the program being debugged. The Status window displays error and informational messages, in addition to any data displayed by the debugger's display, trace and watchpoint facilities. The Status display scrolls when any new information is displayed. You can use the scroll bar to examine earlier contents of the status display.

Some messages that may appear in the Status window at this stage are:

No debugging information available

This means that you are debugging a program which has not been linked with debugging information. No source-level or symbolic debugging facilities are available, and debugging is limited to machine-level debugging (i.e. everything must be specified in terms of machine addresses). If you have forgotten to link the program with debugging information you should quit the debugging session, relink the program with debugging enabled and start the debugging session again.

No source level debugging information

This means that you are debugging a program which has been compiled without debugging enabled. No source-level debugging facilities are available, symbolic debugging facilities are available (i.e. objects can be specified in terms of link time symbols). If you have forgotten to compile the program with debugging information, quit the debugging session and recompile the program with debugging enabled.

RO area limit not on page boundary, last page not protected

This message occurs when memory protection is enabled (as it is by default) and the last part of the code or read only area is not page aligned. This means that the last page of the read only area cannot be protected against accidental writes, since writing to data, or a read/write area which immediately follows the code area, would cause an erroneous data abort. You can ignore this message. Future versions of the linker may align the areas on page boundaries when linking with debugging enabled.

Can't set breakpoint on procedure main

When a debugging session is started the debugger automatically tries to set a breakpoint on main if the **Stop at entry** option is enabled (as it is by default). If the address of main cannot be determined, because, for example, the module containing the procedure main has not been compiled with debugging information enabled, or, the program is not written in C, then the above message will be displayed.

Try moving the pointer completely outside the debugger's windows. The pointer will change to a No Entry pointer, indicating that the debugger is active and you cannot select anything outside the debugger's windows. Moving the pointer back inside the debugger's windows changes it back to the usual arrow pointer.

Clicking Menu on either debugger window produces the following menu:

DDT		
Continue	^C	
Single step	^S	
Call		⇄
Return		⇄
Breakpoint	^B	⇄
Watchpoint	^W	⇄
Trace		⇄
Context		
Display	^D	⇄
Change		⇄
Log		⇄
Find		⇄
Options		⇄
*Commands		⇄
Help		
Quit	^Q	

Continue, **Single step**, **Call**, **Return**, **Breakpoint** and **Watchpoint** are explained in the section *Execution control* on page 33.

Trace, **Context**, **Display** and **Change** are explained in the section *Program examination and modification* on page 41.

Log, **Find**, **Options**, **Help**, **Quit** and ***Commands** are explained in the section *Options and other commands* on page 46.

Specifying program objects

Once the debugger is running, the program can be executed, single stepped, have its variables examined or altered and so on. All of these facilities are described in the following sections. However, before you can use these facilities, you must know how to refer to certain program objects. Variable names, line numbers, procedure names and memory addresses all have a syntax which must be used if you are to reference the desired object.

The following notation will be used in describing the syntax:

- An item in square brackets ([]) is an optional item which can be omitted if desired.
- An item in braces ({ }) is an optional item which can be repeated as many times as desired.
- An item in italicised text is a non-terminal item, i.e. an item which must be replaced by a suitable string of characters.

For example, an optional, comma-separated list of numbers would be denoted by:

[*number* { , *number* }]

Procedure names

Procedure names are used, for example, when setting a breakpoint on entry to a procedure. The syntax for a procedure name is:

```
[module:]{procedure:}procedure
```

where *module* is the name of a program module and *procedure* is a procedure name within that module. Each procedure name in the list of procedure names refers to a successive procedure in the textual nesting of procedures. The module name is the leaf filename of the compiled source file. For example, consider the following program fragment stored in file `pas.test`.

```
program raytrace(input, output);
var count : integer; ...
  procedure pixel(x, y : integer);
  var colour : integer; ...
  function reflect(x, y : integer; angle : real) :
integer;
...
  begin (* body of reflect *) end;
  begin (* body of pixel *) end;
begin (* body of raytrace *) end;
```

The full name for function `reflect` would be:

```
test:raytrace:pixel:reflect
```

that is, procedure `reflect` contained in procedure `pixel` contained in procedure `raytrace` (the debugger treats the entire Pascal program as one large procedure) contained in module `test` (module names do not generally make much sense for Pascal, since standard Pascal has no facilities for separate compilation, but many Pascal implementations, including Acorn's ISO Pascal, have extensions to allow separate compilation).

Note: Some Pascal implementations on RISC OS computers do not represent procedure names in the manner described above. Instead, they generate a new procedure name at the outermost level by concatenating enclosing procedure names to the current procedure name separated by a dot. Also, they do not generate a pseudo-procedure for the whole program. Thus, with such an implementation, the full name for function `reflect` would be `test:pixel.reflect`.

You do not need to type the full name every time you wish to refer to a procedure: Since the prefixed module name and procedure names are optional they can be omitted, and the procedure referred to by its name alone (e.g. `reflect` or

`pixel.reflect` in the above example). Sometimes it will be necessary to enter a longer version of the procedure if there are two or more procedures with the same name.

Suppose in the above example there was a procedure:

```
test:raytrace:line:reflect
```

`reflect` on its own would be ambiguous, so you would have to enter `pixel:reflect` or `line:reflect` to specify which one you meant. Note that it is still not necessary to enter the `test:raytrace` prefix, since the `line` or `pixel` prefixes are sufficient to render the procedure name unambiguous.

Similarly, suppose you had two C modules called `quickdraw` and `slowdraw`, each containing a static function `circle`. In this case you would need to enter either `quickdraw:circle` or `slowdraw:circle` to indicate which circle function you were referring to.

Even if two procedures have the same name, it may not be necessary to enter more than the procedure name on its own. When looking at a procedure specification, the debugger searches back along the dynamic call chain (i.e. the chain of procedures called to reach this point in the program) to find a procedure name which matches the first name in the procedure specification. Having found this, it matches the rest of the procedure specification against textually nested procedures contained within the first procedure found.

For instance, in the above example with two `reflect` procedures, if the program was stopped (at a breakpoint, perhaps) at some point in `pixel:reflect`, then `reflect` on its own would refer to `pixel:reflect`, since on looking at the dynamic call chain the debugger would find that it was in a procedure called `reflect`, and would match that against the procedure specification `reflect`.

Variable names

Variable names are used, for example, when setting a watchpoint. The syntax for a variable name is.

```
[procedure-specification:][line number:]variable
```

where *procedure-specification* is a procedure specification as described in the section above, *line number* is a line number in a source file and *variable* is the name of a variable.

As in the case of a procedure specification, the debugger tries its best to match a variable name given to it, by first searching back along the dynamic call chain, and then searching the global variables, so it is usually not necessary to specify more than the variable name on its own.

In the `raytrace` example above, if the program was stopped at some point in the function `reflect` then `x`, `y` and `angle` would refer to the arguments in function `reflect`, `colour` on its own would refer to the local variable `colour` in procedure `pixel` (since the debugger searches back the call chain and finds procedure `pixel` containing a variable `colour`). The variable `count` would refer to the global variable `count` in program `raytrace`.

In some cases, however, it may be necessary to specify more information about the variable; suppose, for example, you wanted to examine the arguments `x` and `y` to the procedure `pixel`. Specifying `x` or `y` on its own would display the `x` or `y` argument in function `reflect` so you must specify `pixel:x` or `pixel:y`.

There may still be some ambiguity in languages other than Pascal. In Pascal you cannot declare local variables within a program block (i.e. between a `begin...end` pair), however C allows declarations in local blocks. Consider for example the following code fragment as it would be displayed in the debugger's source window:

```
DDT: adfs::HardDisc5$.ddt.man.eval
115 int logical(int a, int b, int op)
116 {
117     int tmp; /* tmp used in calculating a op b */
118     if (op == OP_GT || op == OP_GE) { /* > or >= */
119         int tmp;
120     }
121     op = op == OP_GT ? OP_LE : OP_LT; /* Change to <= or < */
122     tmp = a; a = b; b = tmp; /* and swap arguments */
123 }
124
```

There are two declarations of `tmp` in `logical`, so `tmp` or `logical:tmp` may be ambiguous. In this case you must specify a line number before the variable name to remove the ambiguity.

For example, to refer to the `tmp` variable in the outer scope (i.e. at the function level) you could enter:

`117:tmp`

or

`logical:117:tmp`

To refer to the `tmp` variable in the inner block, use:

`120:tmp`

or

`logical:120:tmp`

The line number should be the line number of the declaration of the variable (in this case 117 or 120). The line numbers are displayed in the source window, so it is quite easy to find the line number of the declaration.

The syntax described above is sufficient to refer to all textually nested variables. However, variables in earlier instances of a recursive or mutually recursive procedure cannot be accessed. For example:

```
void hanoi(int src, int dest, int via, int n)
{
    if (n > 1) {
        hanoi(src, via, dest, n - 1);
        hanoi(src, dest, via, 1);
        hanoi(via, dest, src, n - 1);
    } else
        printf("Move disc from peg %d to peg %d\n", src,
            dest);
}
```

Suppose this function is called with $n = 3$ and that it recurses until it hits a breakpoint on the `printf` when $n = 1$. There is no direct way to refer to the variables `src`, `dest` and `via` in an outer call when $n = 2$ or 3 since any reference to these variables will refer to the variables in the call with $n = 1$. What you can do is, use the **Context** option on the debugger's main menu (described in the section *Program examination and modification* on page 41) to change the context to an outer call on the stack. Since the debugger searches from the current context outwards, you can now specify the variable as per normal. The debugger will ignore the variables in inner calls and use the variable in the current context.

C99 variable length arrays are not fully represented in the debugging format. The current C implementation uses a concealed pointer to an allocated block, so to view a local variable-length array, it must be specified as `*array`. This is analogous to the way the debugger currently treats C++ references.

Expressions

Several DDT commands (for example **Display Expression**) may take arbitrary expressions. The syntax for these expressions is based on that found in C.

The following table summarises the operators available along with the precedence of each operator.

1	()	grouping, e.g. <code>a*(b+c)</code>
	[]	subscript, e.g. <code>isprime[n]</code> , <code>matrix[1][2]</code>
	.	record selection, e.g. <code>rec.field</code> , <code>a.b.c</code>
	->	indirect selection, e.g. <code>rec->next</code> is <code>(*rec).next</code>

2	!	logical not, e.g. !finished
	~	bitwise not, e.g. ~mask
	-	negation, e.g. -a
	*	indirection, e.g. *ptr
	&	address, e.g. &var
3	*	multiplication, e.g. a*b
	/	division, e.g. c/d
	%	remainder, e.g. a%b is a-b* (a/b)
4	+	addition, e.g. a+1
	-	subtraction, e.g. b-d
5	>>	right shift, e.g. k>>2
	<<	left shift, e.g. 2<<n
6	<	less than, e.g. a	greater than, e.g. n>10
	<=	less than or equal to, e.g. c<=d
	>=	greater than or equal to, e.g. k>=5
7	!=	not equal to, e.g. count!=limit
8	&	bitwise and, e.g. i & mask
9		bitwise or, e.g. m1 &0100

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

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

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

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

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

Constants may be integers (to the base specified in the Base option), hex integers (preceded by &) character constants, strings or floating point numbers. The following show examples of each:

32768	Integer in the currently selected base
&8000	Hex integer
3.2768e4	Floating point number
'A'	Character constant
"Hello, World"	String

Addresses & low-level expressions

This section describes the syntax for low-level expressions. It is directed mainly at assembly language programmers. You can skip this if you will only be using the high level language debugging facilities.

The syntax for a low-level expression (as used, for example, when setting a breakpoint on a memory address or displaying a disassembly or memory dump) is as follows (an understanding of BNF is assumed):

```
expr ::= value + expr | value | expr  
value ::= '&' hex-number | number | symbol
```

where *hex-number* is a hexadecimal number, *number* is a number in the default base (hexadecimal if no default base specified) which must start with a digit in range 0...9 and *symbol* is a low level symbol in the debugging information produced by the linker.

Examples:

<code>main</code>	Address of function <code>main</code> .
<code>main + &14</code>	Five words into <code>main</code> .
<code>8000</code>	Start of image (assuming the image has not been relocated and the default base is hex.)
<code>Image\$\$RO\$\$Base</code>	Preferred way of specifying base of program.

Execution control

This section describes how you can control the way in which the debugger executes your program.

Continue

Continue starts or restarts execution of the program. Execution continues until one of the following events occurs:

- a watchpoint changes or is cancelled
- the program runs to completion
- an error or abort condition occurs.

You can interrupt execution of the program at any time by pressing Shift-F12. Note that if another task is executing when you press Shift-F12 you may need to generate an event to force execution to return to the program before the Shift-F12 interrupt will be noticed. The simplest way to do this, usually, is to click on the program's icon on the icon bar, or click on one of its windows.

As the debugger sets a breakpoint on procedure `main`, you can usually use **Continue** to start execution of the program and get to the first line of your source text. You cannot do this if

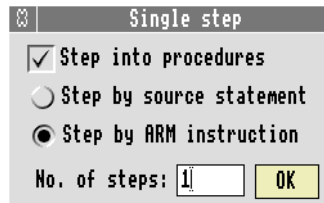
- you have disabled the **Stop at entry** option, or
- the `Can't set breakpoint on main` message appeared when you started the debugging session.

Note that if you have any watchpoints set, the instructions are single stepped instead of executed and the watchpoints are checked after each instruction. If any have changed, the single stepping is stopped at that point. This will be completely transparent, except that the program runs more slowly than normal.

You can use Ctrl-C as a short cut for **Continue**.

Single step

Single step allows you to step execution through one or more source statements or ARM instructions. Choosing **Single step** produces the following dialogue box:



No. of steps allows you to enter the number of statements or instructions to be executed. The **Step by source statement** and **Step by ARM instruction** radio icons allow you to specify whether the contents of **No. of steps** should be treated as a source statement count or an ARM instruction count.

The **Step into procedures** option icon selects whether procedure calls should be treated as a single source statement / ARM instruction or whether single stepping should continue into the procedure call.

Note that the debugger cannot detect certain types of procedure calls, for example, calls via function variables in C. In these cases the debugger will continue stepping into the procedure, regardless of the setting of the **Step into procedures** option.

Note for assembly language programmers: The debugger treats BL instructions as procedure calls, so if some other instruction is used to call a procedure, this will not be detected by the debugger. For instance, consider the following example, which might be produced by the C compiler when calling via a function variable.

```
MOV  lr, pc           ; Set up link. PC = current instruction + 8
LDR  pc, [sp, #o_fn]  ; Load PC from function variable on stack
...                  ; Returns here
```

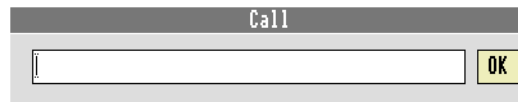
You complete the **Single step** dialogue by clicking on **OK** or pressing Return. The specified number of statements or instructions are then executed.

Note that if you are currently stopped at an ARM instruction for which there is no source information, stepping one source statement will step ARM instructions until an instruction for which source information is available is reached. This can be used when you initially start a debugging session, and wish to step to the first source statement to be executed. This is usually the first instruction of `main` for C programs, but need not necessarily be so, if, for example, the module containing `main` was not compiled with debugging information.

You can use Ctrl-S as a short cut for single stepping 1 instruction or source statement. The **Step into procedures** and **Step by source statement / Step by ARM instruction** are determined by the current settings in the **Single step** dialogue box (i.e. the settings when the dialogue box was last displayed).

Call

Call allows you to call a named procedure. Choosing **Call** produces the following dialogue box:



The writable icon allows you to specify the name of the procedure to be called. You can specify arguments to the procedure in a comma-separated list in round brackets after the procedure name.

The arguments must be word-sized objects (e.g. integers or pointers) or floating-point values. Floating-point arguments occupy the next two adjacent ARM registers or stack words as described in the ARM Procedure Call Standard (i.e. floating-point arguments are not passed in floating-point registers).

Complete the dialogue by clicking on **OK** or pressing Return. The specified procedure is called with the arguments on the program's stack, and in ARM registers R0 - R3.

Note that the program's stack pointer must be initialised before attempting to call a procedure: calling a procedure without a valid stack pointer may result in a Data abort or Address exception. Therefore, if you are debugging a program written in C, you must ensure you have executed the run-time system initialisation code using **Continue** or **Single step** as described above. If you are debugging a program written in assembler, you must ensure that you have executed your own initialisation code, which must initialise the stack pointer.

Return

Return allows you to return from the current procedure. Choosing **Return** produces the following dialogue box:

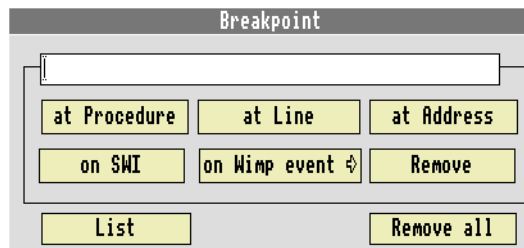


You can enter a value to be returned from the procedure in the value writable icon. This may be either an integer or floating-point value. If you do not specify a value, a default value of 0 (or 0.0 for floating-point values) is used.

Note that the **Return** option returns from the procedure in the current context. If you used the **Context** option to change the current context to an outer context on the stack *n* on the debugger's menu, the **Return** option will return from the procedure in the selected context, rather than the currently executing procedure.

Breakpoint

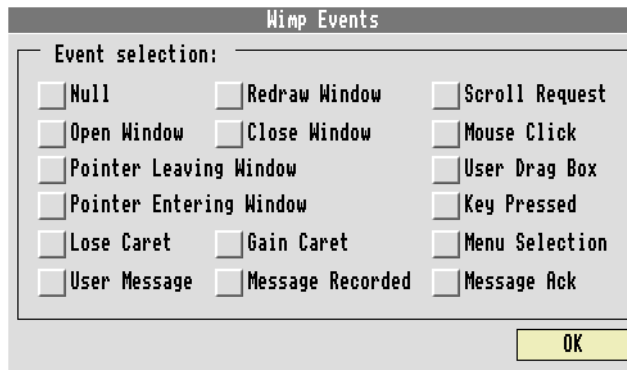
Breakpoint is used to add and remove breakpoints. Choosing **Breakpoint** produces the following dialogue box:



Choosing one of the **at Procedure**, **at Line** or **at Address** buttons sets a breakpoint at the procedure, source line number or memory address entered in the associated writable icon. The syntax for specifying these objects is described in the section *Specifying program objects* on page 26.

Choosing the **on SWI** button causes the debugger to stop when the named SWI is called by the debuggee. SWI names are specified as in the *RISC OS Programmers Reference Manual* except that a leading 'X' is ignored and case is ignored when matching SWI names.

Choosing the **on Wimp event** leads to the following dialogue box:



Select the set of Wimp events you are interested in and click **OK**. The debugger will stop execution of the debuggee when it receives one of the specified events and will display a message describing the event received.

For example:

```
Event = User message, action = 0 (Quit)
```

Choosing **Remove** removes the breakpoint specified in the associated writable icon. The breakpoint may be specified as a breakpoint number, as given in the list breakpoints command, preceded by a hash (#) or it may be specified exactly as specified when setting the breakpoint.

List displays a list of all currently set breakpoints with breakpoint numbers which can be used when removing individual breakpoints.

Remove all removes all current breakpoints.

You can use Ctrl-B as a short cut to produce the Breakpoint dialogue box.

Breakpoints may also be set or cleared by clicking on a line in a source or disassembly display. Clicking on a line sets a breakpoint on the line. The breakpoint is shown by the breakpoint marker (a filled in circle) to the left of the line. Clicking on a line which already has a breakpoint removes the breakpoint.

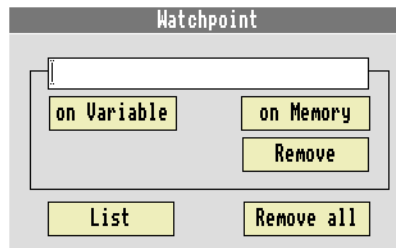
Watchpoint

Choose **Watchpoint** to detect when a variable or memory location changes its value. When a watchpoint is in force, instructions in the program are single stepped instead of being executed and the values of the variables being watched are checked after each instruction or source statement executed. Watchpoints may be set on simple variables such as integers or more complex variables such as

structs and arrays. Setting a watchpoint on a whole array can be very useful if, for example, you are debugging a sort routine; you can track all changes to the array as it is sorted.

Since the debugger is single stepping, execution can be quite slow, typically between 4 and 10 times as slow as normal execution. If this is too slow to be practical, the best approach is to try to isolate the section of code under suspicion, set a breakpoint on entry to this section of code, and only set the watchpoint(s) when the program stops at the breakpoint.

Choosing **Watchpoint** produces the following dialogue box:



Selecting **on Variable** or **on Memory** sets a watchpoint on the variable or memory location specified in the associated writable icon. The syntax for specifying variables or memory addresses is described in the section *Specifying program objects* on page 26.

Remove removes the watchpoint specified in the associated writable icon. As with breakpoints the watchpoint to remove may be specified as a watchpoint number preceded with a hash (#) or exactly as specified when setting the watchpoint.

List displays a list of watchpoints currently in force. **Remove all** removes all watchpoints.

Note that if you are watching a local variable (i.e. a variable stored on the stack) the watchpoint will become invalid on exit from the procedure containing the variable being watched. The debugger detects this and stops execution with the message:

```
Watchpoint watchpoint discarded on exit from procedure
```

where *watchpoint* is the name of the variable being watched.

Also note that when you are watching a variable which is stored in a register, the debugger may erroneously report a change in the variable's value. This is because the C compiler does not allocate registers to variables over the whole range of a procedure. Instead, it allocates the registers over the lifetimes of variables (i.e. the range of the procedure in which the variable is actually used). Outside this range a register may be used for other purposes (such as temporary values in calculations).

It may even be allocated to another variable, if the lifetimes of the variables do not overlap. Thus the debugger may report a change in the variable when it sees the register changing, but of course the register is no longer being used to store the variable.

You can use Ctrl-W as a short cut to produce the Watchpoint dialogue box.

Trace

Trace allows you to select a set of actions about which you wish to be informed. When one of these actions occurs a message to this effect is displayed in the debugger's status window. For certain actions the source / disassembly display is updated to show where the action occurred.

The actions which you can trace are as follows:

Execution

The source / disassembly display is updated for every ARM instruction or source statement executed (ARM instruction if Machine-level debugging is enabled, source statements otherwise). The effect is to produce a continuous execution display in the context window.

Breakpoints

When a breakpoint occurs, instead of stopping execution, a message is displayed in the Status window:

Break at breakpoint

where *breakpoint* is the location of the breakpoint. The source / disassembly display is updated to show where the breakpoint occurred. Execution then continues after the breakpoint.

Watchpoints

When a watchpoint changes, a message of the following form is displayed:

Watchpoint watchpoint changed at location

where *watchpoint* is the name of the variable being watched, and *location* is the program location where the watchpoint was changed. If, for example, you are debugging a sort routine and have a watchpoint on the array being sorted, you can select watchpoint tracing to provide a continuous update of all changes to the array.

Procedures

When procedure tracing is enabled, a message of the following form is displayed:

Entered procedure *procedure name*

This can be useful if you wish to quickly locate the procedure where a fault is occurring.

Event breaks

When a Wimp event break occurs execution is not halted. Instead of stopping at the breakpoint a decoded form of the event data is displayed and execution continues.

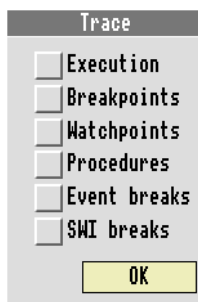
SWI breaks

When a SWI break occurs execution is not halted, a message is displayed:

Break at SWI *SWI Name*

The SWI is then executed and execution continues after the SWI breakpoint.

Choosing **Trace** from the debugger's menu produces the following dialogue box:

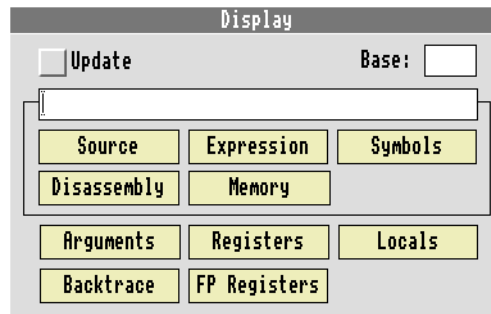


Select the set of actions you are interested in tracing and click on **OK**. A message confirming your selection will be displayed. You won't notice the effects of enabling procedure tracing until execution of the debuggee is resumed.

Program examination and modification

Display

This option allows you to display information about the program being debugged. You can examine source text, instruction disassembly, variable contents, memory contents, stack backtrace information, register contents and low-level symbol values. Choosing **Display** produces the following dialogue box:



You can use Ctrl-D as a short cut to produce this display.

Select the item you want information about. The **Source**, **Expression**, **Symbols**, **Disassembly** and **Memory** icons use the contents of the writable icon to determine what to display. Each icon is described in turn below.

Source

Displays the specified source file in the debugger Context window. You can specify a source line number at which to start the display. The syntax for the filename and line number is:

```
filename[:line]
```

(that is, a valid RISC OS filename optionally followed by a colon (:) and a line number). The line number defaults to 1 if not specified. The filename does not have to be a source file used to generate the program you are debugging: you can display any file you like.

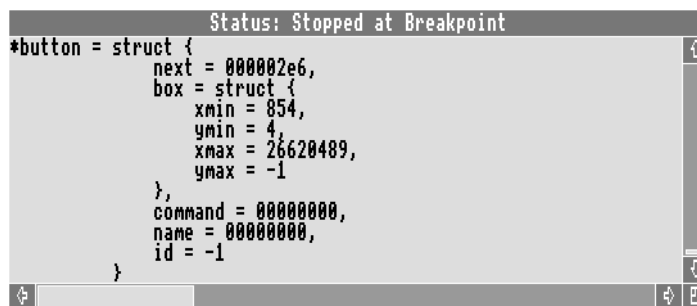
Expression

The writable icon should contain an expression name. The syntax for entering expression names is described in the section *Specifying program objects* on page 26. The expression is displayed in the debugger Status window.

Complex expressions such as C structs or arrays are displayed in structured format, nested substructures are indented to indicate the level of nesting. Character pointers and arrays are displayed as strings if a terminating 0 is found within the first 80 characters and there are no intervening non-graphic characters apart from newline and carriage return, which are displayed as \n and \r. For example, the following structure:

```
typedef struct _HotSpot
{
    struct _HotSpot *next;
    BBox box;
    char *command;
    char *name;
    ComponentId id;
} HotSpot;
HotSpot *button;
```

would be displayed as:



Arguments

Arguments displays all the arguments to the current procedure. The arguments are displayed as if each individual argument had been displayed using the **Display Expression** facility described above.

If you want to examine the arguments in an outer scope (i.e. in the procedure which called this procedure or the procedure which called that ...) you can use the **Context** item on the main menu to change the current context to that of one of the calling procedures, and then select **Arguments** to display the arguments of that procedure.

Locals

Locals is very similar to **Arguments**. It displays all local variables (including the arguments) in the current procedure.

Backtrace

Backtrace displays a list of procedures in the call chain from the current procedure back to the program entry point.

Procedures which have been compiled with debugging information are displayed in the following form:

procedure, line *line* of *file*

Those which have been compiled or assembled without debugging information look like this:

PC = *address* (procedure + offset)

Procedures in the Shared C Library will appear as:

PC = *address*

A typical backtrace might look something like this:



```

Status: Stopped at Breakpoint
button_click, line 176 of c.handler
click_viewer, line 206 of c.handler
PC = 0000c3a8 (call_wimp_event_handlers + 6c)
PC = 0000c438 (wimp_event_dispatch + 50)
PC = 0000bb0c (event_poll + 68)
main, line 175 of c.main
PC = 039a29c4
PC = 0000ae68 (__main + 54)
PC = 039a2d08

```

Symbols

Symbols displays low-level symbols generated by the linker when linking with debugging enabled. The writable icon gives a comma-separated list of symbols to be displayed. The symbols and their addresses are displayed in the debugger's Status window.

You can use the following wildcard characters in symbol names:

- A star (*) matches 0 or more characters
- A hash (#) matches any single character.

For example:

`_kernel_*` would list all the kernel routines
(e.g. `_kernel_swi`)

`*$*$*$*` would list all the linker generated symbols
(e.g. `Image$$RO$$Base` and `C$$code$$Base`).

Disassembly

This displays a symbolic instruction disassembly in the debugger's Context window. The writable icon should contain a low-level expression which evaluates to a memory address indicating where the disassembly should start. The syntax for low-level expressions is described in the section *Specifying program objects* on page 26.

Memory

This displays a memory dump in the debugger's Context window. The writable icon should contain a low-level expression giving the memory address.

Registers

This displays the contents of ARM user registers 0 - 15 and the flags in R15.

FP Registers

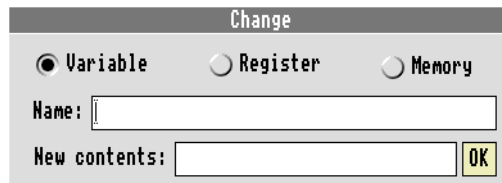
This displays the contents of floating-point registers 0 - 7 and the flags in the floating-point processor status word.

The **Base** writable icon gives the numeric base to be used when displaying Variables, Arguments, Locals, Symbols and ARM registers. If this writable icon is left blank a default of decimal or hexadecimal is used depending on what is being displayed.

The **Update** box applies to Variables, Locals, Arguments, Backtrace, Registers and FP Registers. When **Update** is selected and one of these items is displayed, the item is added to a list of items to be displayed whenever the debugger stops execution (for example, at a breakpoint). There is no way to remove items from this list once they have been added to it.

Change

Change allows you to alter variable, registers or memory contents. Choosing **Change** produces the following dialogue box:



The image shows a 'Change' dialog box with a title bar. Inside, there are three radio buttons: 'Variable' (selected), 'Register', and 'Memory'. Below these are two text input fields: 'Name:' and 'New contents:'. An 'OK' button is located to the right of the 'New contents:' field.

The **Variable**, **Register** and **Memory** radio buttons indicate what is to be changed. The **Name** writable icon indicates which variable, register or set of memory locations is to be changed. The **New contents** writable icon gives the new contents. Clicking **OK** makes the change.

Variable

The **Name** writable icon should contain a variable name as described in the section *Specifying program objects* on page 26. Only simple variables such as integers and pointers or floating-point variables may be changed. The **New Contents** writable icon should contain the new value for the variable, floating-point values are specified in normal C floating-point format.

Register

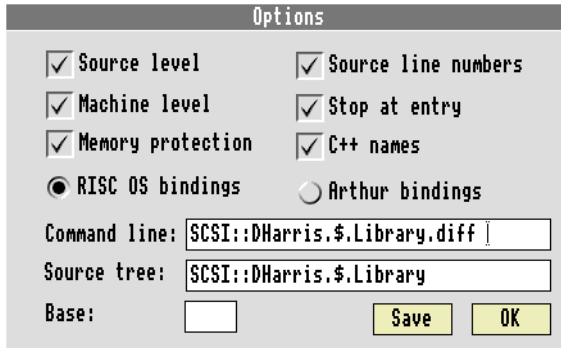
The **Name** writable icon should contain a register name. Valid register names are R0 - R15, SL, IP, SP, LR, PC and F0 - F7. The **New Contents** writable icon should contain a low-level expression or floating-point constant, depending on the type of register being changed. Low-level expressions are described in the section *Specifying program objects* on page 26.

Memory contents

The **Name** writable icon should contain a low-level expression which evaluates to a memory address. The **New Contents** writable icon should contain a comma-separated list of low-level expressions, which are placed in successive memory words starting at the memory word specified in the name writable icon. The syntax for low-level expressions is described in the section *Specifying program objects* on page 26.

Options and other commands

The **Options** item on the debugger main menu produces the following dialogue box:



Source-level debugging

This option enables the display of source information in the debugger Context window. If this option is deselected, a disassembly of the ARM instructions corresponding to the source text will be displayed.

Machine-level debugging

This option enables the tracing of ARM instructions when trace execution is selected.

Memory protection

This option enables or disables protection of sensitive areas of memory. When this option is enabled zero page (0 - &7fff) is protected against being written to by the debuggee and the debuggee's code area is protected against writing.

Source line numbers

This option enables or disables the display of line numbers in source text displays.

Stop at entry

When this option is enabled, the debugger automatically tries to set a breakpoint on procedure `main` when a debugging session is started. This allows you to use **Continue** on the debugger main menu to get rapidly to the start of your source code.

RISC OS bindings / Arthur bindings

This option is provided for backward compatibility.

Command line

This writable icon allows you to change the command line passed to the debuggee. The existing command line is displayed in the icon and may be edited. Note that the first word of the command line should be the program name.

Base

The **Base** writable icon gives the default numeric base when displaying or entering numbers.

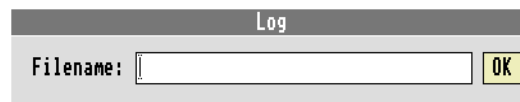
Source tree

Compilers such as Acorn's C may put relative filenames in the debugging information (e.g. `c.display` or `^.mip.c.aetree`). The debugger needs to know where these files can be found. By default it assumes the source files reside in the directory from which the program image was loaded. This writable icon allows you to change this default. It accepts a comma-separated list of directory names, each one ending in a full stop (immediately before the comma).

This could be used when debugging a library whose source is held in a directory different to that of the debuggee program source.

Log

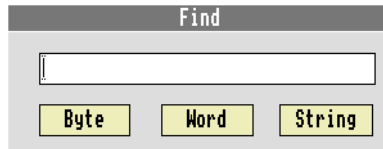
Log allows you to record any information output to the debugger Status window to a text file. Choosing **Log** produces the following dialogue box:



Enter the name of the file into which you wish to log output. The file will be opened as a new log file. Any previous contents of the log file will be overwritten. If a log file was previously open it will be closed when the new log file is opened.

Find

Find allows you to find a sequence of bytes, words or characters in the application workspace. Choosing **Find** produces the following dialogue box:



Word or Byte

The writable icon should contain a comma separated list of low-level expressions giving the word or byte values to be found.

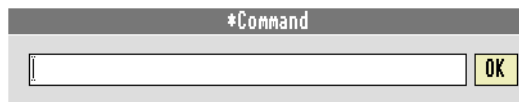
String

The writable icon should contain the sequence of characters to be found, the sequence should be entered without quotation marks of any kind.

All occurrences of the byte, word or character sequence in the application space are reported in the debugger Status window.

*Commands

***Commands** allows you to access the RISC OS CLI from within the debugger. Choosing ***Commands** will lead to the following dialogue box:

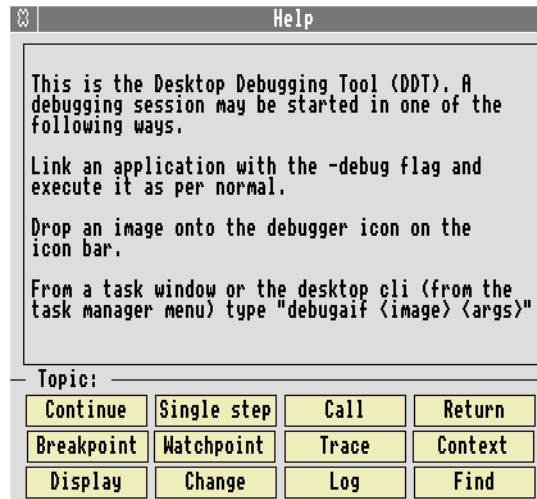


Enter the command you wish to execute in the dialogue box and press Return or click **OK**. If you are debugging a Wimp task (i.e. a task which has called Wimp_Initialise) you should precede the command with the WimpTask command, otherwise the output of any command executed may be displayed in graphics mode.

If you wish to enter several commands you can enter the Gos command or the ShellCLI command in the dialogue box.

Help

Help gives interactive help on the debugger. Choosing **Help** will produce this initial help window:



Choose the icon corresponding to the topic on which you want help. The help will be displayed in the Help box above the topic buttons.

Quit

This quits the debugger and returns to the calling environment (generally the RISC OS desktop).

You can use Ctrl-Q as a short cut for **Quit**.

An example debugging session

The following example debugging session shows how DDT might be used to fix a rather bug-ridden file sorting tool written in C. The source is given here with line numbers for reference later in the chapter. The source, along with the other files to make the application, can be found in `Sort`, which is in the `Sources.DDE-Examples.Debug.Sort` directory.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <stdarg.h>
5
6 #include "kernel.h"
7
8 #define READATTR 5
9 #define READFILE 16
10 #define WRITEFILE 0
11
12 #define FILEFOUND 1
13
14 static void fail(char *errmsg, ...)
15 {
16     va_list ap;
17
18     va_start(ap, errmsg);
19     vfprintf(stderr, errmsg, ap);
20     va_end(ap);
21     exit(1);
22 }
23
24 /* See Sedgewick: Algorithms 2nd edition P 108 */
25 static void sortstrings(char *a[], int n)
26 {
27     int h, i, j;
28     char *v;
29
30     h = 1;
31     do
32         h = h * 3 + 1;
33     while (h <= n);
34     do {
35         h = h / 3;
36         for (i = h + 1; i <= n; i++) {
37             v = a[i];
38             j = i;
39             while (j > h && strcmp(a[j-h], v) > 0) {
```

```
40             a[j] = a[j-h];
41             j -= h;
42         }
43         a[j] = v;
44     }
45 } while (h > 1);
46 }
47
48 void sortfile(char *infile, char *outfile)
49 {
50     _kernel_osfile_block finfo;
51     int size;
52     char *finbuff, *foutbuff;
53     char *cp;
54     int l, linestart;
55     char **lbuff;
56     int i;
57
58     if (_kernel_osfile(READATTR, infile, &finfo) !=
        FILEFOUND)
59         fail("Error opening %s\n", infile);
60     size = finfo.start;
61     finbuff = malloc(size + 1);
62     foutbuff = malloc(size + 1);
63     if ((finbuff == NULL) || (foutbuff == NULL))
64         fail("Out of memory\n");
65     finfo.load = (int) finbuff;
66     finfo.exec = 0;
67     if (_kernel_osfile(READFILE, infile, &finfo) < 0)
68         fail("Error reading %s\n", infile);
69     l = 0;
70     cp = finbuff;
71     linestart = 1;
72     for (i = 0; i < size; i++) {
73         if (linestart) {
74             l++;
75             linestart = 0;
76         }
77         if (!*cp || *cp == '\n') {
78             *cp = 0;
79             linestart = 1;
80         }
81         cp++;
82     }
83     *(finbuff + size) = 0;
84     lbuff = malloc(l * sizeof(char *));
85     if (lbuff == NULL)
```

```
86         fail("Out of memory\n");
87     cp = finbuff;
88     for (i = 0; i < l; i++) {
89         lbuff[i] = cp;
90         cp += strlen(cp);
91     }
92     sortstrings(lbuff, l);
93     cp = foutbuff;
94     for (i = 0; i < l; i++) {
95         strcpy(cp, lbuff[i]);
96         cp += strlen(cp);
97         *cp++ = '\n';
98     }
99     finfo.start = (int) foutbuff;
100    finfo.end = (int) foutbuff + size;
101    if (_kernel_osfile(WRITEFILE, outfile, &finfo) < 0)
102        fail("Error writing %s\n", outfile);
103    free(finbuff);
104    free(foutbuff);
105    free(lbuff);
106 }
107
108 int main(int argc, char *argv[])
109 {
110     if (argc != 3)
111         fail("Usage: Sort <infile> <outfile>");
112     sortfile(argv[1], argv[2]);
113     return 0;
114 }
```

The debugging session

Follow the steps below to debug the example program.

- 1 Compile and link the program by double clicking the `MkInstall` file provided in the `Sort` directory.
Now try running the program:
- 2 Double click on the `!Sort` application directory. The Sort tool icon will appear on the icon bar.

- 3 Drag the example input file `infile` on to the Sort tool icon.

This should sort the input file and display a **Save as** dialogue box, to allow you to save the sorted result. Unfortunately it doesn't, instead it produces a display similar to the following:

```
Internal error: abort on data transfer at &FC138D1C

Postmortem requested
  Arg2: 0x0000000c 12 -> [0xe59ff2b8 0xe59ff2b8 0xe59ff2b8 0xe59ff2b8]
  Arg1: 0x0000cfd8 53208 -> [0x0000d060 0x0000d064 0x0000d064 0x0000d064]
fc138d14 in function sortstrings
  Arg2: 0x0000b371 45937
  Arg1: 0x0000b341 45889
82d8 in function sortfile
  Arg2: 0x0000b300 45824 -> [0x0000b31c 0x0000b341 0x0000b371 00000000]
  Arg1: 0x00000003 3
83e4 in function main
  Arg2: 0x000083b8 33720 -> [0xe1a0c00d 0xe92dd823 0xe24cb004 0xe15d000a]
  Arg1: 0x0000ada0 44448 -> [0x63412e24 0x436e726f 0x2b2b432f 0x736e492e]
fc129b48 in shared library function
8460 in anonymous function
```

This is called a symbolic backtrace.

The first line gives a general indication of what might be wrong with your program. In this case it's an abort on data transfer; the program tried to access memory which is outside the addressing range of your computer.

Each line of the form *address* in function *name* represents a procedure call frame on the stack. The first frame on the stack is function `sortstrings`; this is where the illegal address was referenced.

This doesn't look too promising, so try running it under DDT to get more clues as to what might be wrong:

- 4 Quit the Sort tool.
- 5 Construct a debug version of Sort. To do this, double click the `MkDebugInstall` file provided in the Sort directory.
- 6 Ensure that the currently-selected directory is the `Sources.DDE-Examples.Debug.Sort` directory so that debugger can find the source tree. You can do this by clicking the Menu button over this directory window in the Filer and selecting **Set directory**.
- 7 Start the debugger if you haven't started it already and drag the `!Sort` application directory on to the debugger's icon.
- 8 Drag the sample input file `infile` on to the Sort icon on the icon bar. The debugger's Context and Status windows should now be displayed.

The program actually crashed in the function `sortstrings`. Since you want the program to stop before making the illegal access, you want it to stop at the beginning of function `sortstrings`. So:

- 9 Set a breakpoint on procedure `sortstrings`:

Bring up the breakpoint dialogue box. Enter the name `sortstrings`, and choose **at Procedure**.

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

Tell DDT to start executing your program:

- 10 Choose the **Continue** option from the debugger's menu. The debugger will stop with the following message:

Break at main, line 110 of `c.sort`

The debugger always stops on entry to `main`. However you want it to continue until it reaches `sortstrings`, so:

- 11 Choose **Continue** from the main menu again.

This time the debugger displays the following message:

Break at `sortstrings`, line 30 of `c.sort`

The Source window should contain the source for the start of function `sortstrings`, with the execution location indicator (`=>`) pointing to the first source line of the function `sortstrings`.

Now you want to examine the program state to ensure it is correct before continuing. In this case, the most important state information is the function's arguments. You can examine them as follows:

- 12 Choose **Display** on the debugger's menu (or use the short cut Ctrl-D) and click on the **Arguments** button in the Display dialogue box.

The debugger will display the following in the Status window:

`a = 0000d028`

`n = 12`

The two arguments to `sortstrings` are:

`n` is the number of strings to sort, in this case 12. This is correct, since there were 12 names in the input file.

`a` is a pointer to an array of `char *`s or strings. The debugger displays the value of this pointer, i.e. the address of the array.

Note: You may get a different address when you try running this example depending on the version of the C compiler and library you are using.

Next, examine the individual elements of the array:

- 13** Enter the array element as it would appear on the left hand side of an assignment in C in the Display dialogue box, and click on the **Expression** button.

To examine element 0, enter `a[0]`. To examine element 1, enter `a[1]`. The debugger will display the array elements as follows:

```
a[0] = 0000d0b0 -> "Noel"
a[1] = 0000d0b4
```

The first element was correct: it contained the string `Noel`, which is the first name in the input file. However, the second element is a null string. This is wrong: it should contain the string `Edward`. This means that the arguments to `sortstrings` were wrong. The error therefore occurred earlier, so you want to try re-running the program under the debugger and setting the breakpoint earlier:

- 14** Quit the debugging session and drag the sample input file `infile` to the Sort icon to start a new debugging session.

- 15** Now follow the instructions in step 9 to set the breakpoint at function `sortfile` instead of function `sortstrings`, and continue execution until the program hits the breakpoint at function `sortfile`.

The variable `lbuff` is passed as the first argument (`a`) to `sortstrings`. `lbuff` is initialised in the loop just before the call to `sortstrings`. Therefore you want to set a breakpoint at the start of the initialisation loop:

- 16** Scroll the Source window up until the initialisation loop comes into view. From the line numbers in the Source display you can see that the initialisation loop starts at line 87, with the initialisation of `cp`. So, set a breakpoint on line 87:

- 17** Enter 87 in the Breakpoint dialogue box and click on **at Line**.

- 18** Now choose **Continue** from the main menu.

The program will continue executing until it reaches line 87, where it will stop at the breakpoint. You want to examine each element of the array as it is initialised, since the array is initialised from the pointer `cp`. Set a watchpoint on `cp`:

- 19** Enter `cp` in the Watchpoint dialogue box and click on **on Variable**.

- 20** Choose **Continue** again. The debugger will stop with the message:

```
Watchpoint on cp changed at sortfile, line 88 of c.sort
New contents: 0000d0b0 -> "Noel"
```

This is correct, so:

- 21 Choose **Continue** again. The debugger will respond with:
- ```
Watchpoint on cp changed at sortfile, line 90 of c.sort
New contents: 0000d0b4
```
- This is wrong: it should contain the string Edward. Look at the line which updated the value of cp:
- ```
90 cp += strlen(cp);
```
- This is supposed to update cp to point to the next string in the list of strings to be sorted. It does this by adding the size of the string pointed to by cp into cp. Unfortunately, it miscalculates the size of the string by omitting to take into account the 0 byte at the end of the string. This means that the second and all subsequent strings are treated as null strings, because they are pointing to the 0 byte at the end of the previous string instead of the start of the string.
- To fix this:
- 22 Quit the debugger and the Sort tool.
- 23 Edit the file c.sort and change line 90 to read:
- ```
90 cp += strlen(cp) + 1;
```
- 24 Recompile c.sort using MkDebugInstall.
- Now try re-running the program:
- 25 Double click on the !Sort application directory and drag the file infile to the Sort tool icon, then choose **Continue** twice on the DDT menu to run Sort. The result is the same as when you first tried running it: you get the same exception, although this time trapped by DDT rather than generating a backtrace, so obviously the fix applied to line 90 didn't fix the problem. So, try running it under the debugger again:
- 26 Quit the Sort tool frontend.
- 27 Drag infile to the Sort tool icon.
- 28 Set a breakpoint on function sortstrings and choose **Continue**. The debugger will stop when it reaches main.
- 29 Choose **Continue** again, and the debugger will stop at the start of sortstrings.
- Examine the arguments. All being well they should look something like this:
- ```
a = 0000d02c
n = 12
```


- 30** Display the individual elements of `a` by entering `a[0]` etc., in the Display dialogue box and choosing **Expression**.

Do the same for `a[1]` and `a[11]`. The display should look like this:

```
a[0] = 0000d0b4 -> "Noel"
a[1] = 0000d0b9 -> "Edward"
a[11] = 0000d0f3 -> "Martin"
```

They're correct now, so something must be wrong with the sort algorithm. So, try setting a breakpoint on the inner while loop:

- 31** Scroll the source display to find the line number; it should be line 39. Enter 39 in the Breakpoint dialogue box and click on **at Line** and continue execution. The debugger should display:

```
Break at sortstrings, line 39 of c.sort
```

Examine a few variables:

- 32** Enter `j` in the Display dialogue box and choose **Expression**; then do the same for `h`. The debugger should display:

```
j = 5
h = 4
```

These are both correct, so look at the contents of `a[j-h]`:

- 33** Enter `a[1]` in the Display dialogue box and choose **Expression**. The debugger should display:

```
a[1] = 0000d0b9 -> "Edward"
```

The shellsort algorithm should be comparing against the first string (i.e. `Noel`). It is not, so this is wrong. Looking closely at the algorithm you can see that it has been written assuming array indices start at 1, whereas in C they start at 0.

To fix this, you could subtract 1 from each array index. However you just want a quick fix to see if it works, so:

- 34** Add the following line at the start of the function after line 29:

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

- 35** Compile the program without debugging enabled by using `MkInstall`, and try running the result.

All being well, the program should run to completion and produce a Save as dialogue box for the output. You can just click the **OK** button to save it, or you may like to drag it to the editor icon to load it into the editor to check that it has been sorted correctly.



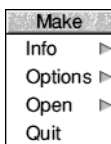
The Make application provides a simple way to construct Makefiles for building multiple-file programs. Although this tool is still provided, its use is no longer recommended. Instead, consider using AMU with the Shared Makefiles. All of the examples in `Sources.DDE-Examples` that have Makefiles have been updated to build in this manner, and provide useful examples of how to do this. See the chapter AMU on page 121 for more information.

Make aids the programmer in the construction and maintenance of multiple-file programs, which can be combined to form any number of final targets (for example, libraries, modules, and application programs). The set of final targets and the files from which they are constructed are known as a *project* (see later for a more detailed description of this term). The facilities provided for a project include

- automatic construction of Makefiles
- automatic maintenance of Makefiles to track changes made to sources and the addition/deletion of source and object files to or from a project
- setting options using dialogue boxes for the tools used to convert source files to object files (e.g. C compiler or ObjAsm options)
- pre-emptive multitasking of the Make process when constructing final targets, including the ability to pause, continue, or abort it at any time
- display of the output of tools used to make a final target, in a scrollable, saveable window.

Invoking Make

Make can be invoked in two ways; by double-clicking on the Make icon from a directory display, or by double-clicking on a file of type `Makefile` (0xFE1). In the latter case this will also run the Acorn Make Utility (AMU) tool to make the first target found in the chosen Makefile.



Clicking Menu on the Make icon gives the menu shown on the left.

Info shows the normal information box about the application.

Options allows the setting of auto-run and display options.

Open is used to open a dialogue box for a given project.

Quit quits Make.

These are described more fully in later sections.

Using Make

To use Make efficiently it is necessary first to understand how to create and maintain a project.

Projects

A project is made up of a collection of source and object files, which combine to form a number of final targets. The life cycle of a project will typically involve the creation and maintenance of the project, the production of final results, and finally, if required, the removal of the project from Make's control. The details of these steps are more fully described in later sections, but here we give an overview of their operation.

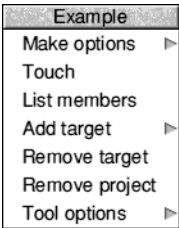
When a new project is created, you give it a unique name, and save its associated Makefile to disc. The persistent state of a project is held in a Makefile, which is automatically maintained by Make, with the option that it can be textually edited for customisation to a particular projects requirements. To achieve this automatic maintenance, the Makefile is divided into sections which are delimited by *active comments* (i.e. lines beginning with a (#), which are otherwise ignored by the AMU program).

The files which make up the project can reside anywhere on disc (or on a network) and can be added to, and removed from, the project by dragging their filer icons onto a dialogue box representing that project.

Final targets for the project are created by clicking on **Make** in the dialogue box relating to that project; the targets will be saved in the same directory as the Makefile for the project.

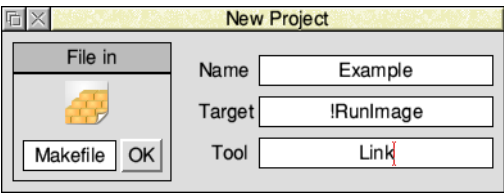
Under the desktop the concept of *current directory* has no sensible meaning, Make therefore uses the work directory in which the Makefile for a project has been saved as a prefix for all filenames used in the project. This prefix is denoted by the at symbol (@).

Clicking Menu on a project dialogue box gives the menu shown below, which is used to further tailor the project. References to this menu are made in a later section on maintaining projects.



Creating new projects

In order to create a new project, you should click Select on the Make icon on the icon bar. This will display the New Project dialogue box as shown below, which allows you to enter information for the new project:



There are three writable icons in the New Project dialogue box which you **must** fill in before a new project can be created. These are:

- Name** you should fill this in with the name of the project. This name will be used to identify the project in the **Open** menu as described later.
- Target** you should fill this in with the name of the main target to be created from this project. For example, if you were creating an application the target name would be !RunImage, if you were creating a module the target name would be the module's name (e.g. FrontEnd).
- Tool** you should fill this in with the name of the tool used to construct the main target. For an application this could be Link, or in the case of a library this could be Libfile.

Note: Make requires this tool to be one which takes intermediate files and creates a final object. Such tools are Link (for a module or application), LibFile (for a library), Squeeze (for a squeezed application) or ModSqz (for a squeezed module).

Having filled in these three boxes, you must then save the Makefile which will be used to hold all information for this project. This is accomplished either by dragging the Makefile icon to a directory viewer (having optionally changed the leafname from the default `Makefile`), or by typing in a full pathname and clicking **OK**. The directory in which the Makefile is saved is important. This directory is where the final targets for the project will be created, since each target will be saved in the `@ work` directory (see the section *Creating a final target for a project* on page 67 for an explanation of this). The sources for the project can be stored anywhere, since they will always be referenced relative to `@`. If any of the Name, Target or Tool icons have not been correctly filled in then an error is reported, and the Makefile is not created.

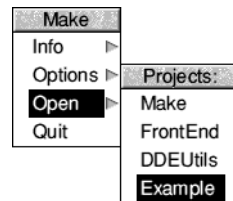
When this process has been completed, the newly created project becomes one of those maintained by Make, until it is explicitly removed (see the section *Removing projects* on page 66 for how this is done). The dialogue box which is used to maintain this project then appears, with the project's name in its title bar. The project can then be maintained as described below.

Maintaining projects

To maintain a project it is necessary to understand how to open and close projects, and how to specify the targets for a project.

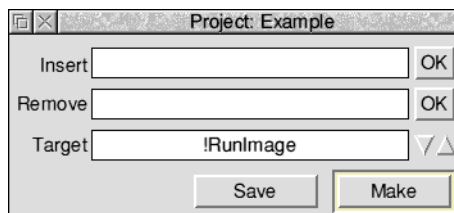
Opening a project

Make keeps a list of all projects which it is maintaining at any one time. This list is shown when you enter the **Open** submenu from Make's application menu. When no projects are known about, this menu item is unselectable.



The list of project names is shown with the most recently registered project at the bottom. Clicking on a project name in this list will open a dialogue box for that project, with the name of the project in its title bar; if the project was already open,

then the dialogue box is brought to the front of the WIMP's window stack. If the project is being opened for the first time, then the directory containing the Makefile for this project is also opened. The dialogue box is shown below:



This dialogue box can be used to add new members to the project, remove members which are no longer required, make final targets, and select the current final target to which these operations refer. These are described in more detail in later sections.

Adding and removing members

When you have written a new source file or created a new object file which you wish to include in a project, you should drag the filer icon for that file to the icon marked **Insert** in the project's dialogue box menu. Typically, the only object files which you will need to insert in a project are external libraries. Any number of files can be dragged in this way to **Insert**, where their full pathnames are displayed, provided that the number of characters displayed does not exceed the buffer for the icon (4096 characters by default, but this can be changed by using a Wimp templates file editor).

Once you are satisfied that this is a list of all the files to be added to the project, click on **OK** to the right of **Insert**. The insertion will then take place. An asterisk appears in the title bar of the project dialogue box to indicate that this project has been modified since its Makefile was last saved.

If you wish to remove members from a project, follow the same procedure as that described for insertion, but drag file icons to the **Remove** icon instead, and click on **OK** to the right of **Remove**. Again an asterisk will appear in the project's title bar, to indicate that a modification has been made.

Note that insertion and removal applies only to the currently selected target when used in conjunction with multiple-target projects (see the section *Multiple targets* on page 64 for more details).

Make uses the following rule for dealing with files dragged to **Insert**: if the filename has, as its last but one component, a string (usually just one character) which corresponds to one of those registered by a translation tool, then it is assumed to

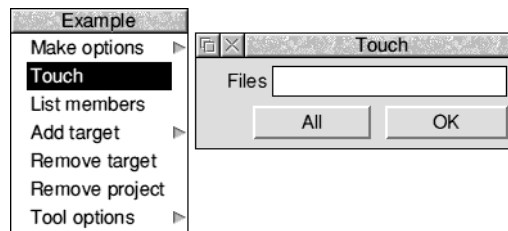
be a program source file and a rule is constructed to make it into an object file; otherwise it is assumed to be an object file (such as a library) and will just be inserted into the list of objects which go to make up the current final target.

Listing members

A list of the members which have been added to a project (and not subsequently removed) can be obtained in a scrolling text window by selecting the **List members** option from that project's dialogue box menu. The filenames in this list are expanded to full pathnames, whereas they will appear relative to @ in the Makefile for the project.

Touching members

You can force a member of the project to be time-stamped using the **Touch** option in a project's dialogue box menu:

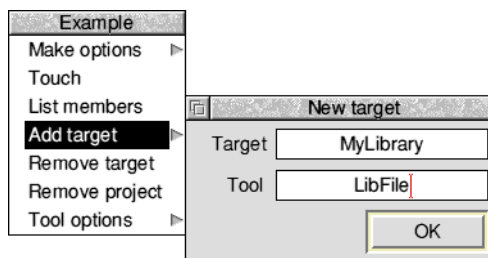


In the Touch dialogue box, you can type (or drag to it) the filename(s) of the file(s) to be touched (either relative to @ as it appears in the Makefile, or as a full pathname), and then click on **OK**. If you wish to touch all source members of the project, then click on **All**; in this case any filename in **Files** is ignored.

Multiple targets

When a project is first created, it has just one final target - the one whose name is entered in the Target icon in the New Project dialogue box. This name will also appear in the Target icon in a project's dialogue box when that project has been opened. This target is referred to as the *current* target, and it is the target which will

be made when you click the Make icon. The current target is also the one to which members are added or removed when you enter filenames in the **Insert** and **Remove** icons from a project's dialogue box.



In order to add a new target, you should use the **Add target** option from a project's dialogue box. In the **Add target** dialogue box you must enter a name for the new target, and the name of the tool which is used to construct that target (e.g. `MyLibrary` and `LibFile`), as shown above.

Targets created in this fashion can be removed by choosing **Remove target** in the project menu. **Remove target** always applies to the current target.

When a project has its dialogue box open, the list of final targets can be traversed using the up and down arrow icons (next to the Target icon). You will notice that any targets which you manually insert in the user-editable section of the Makefile will also appear in the project dialogue box. This is so that you can select them as the target to be made when clicking on the Make icon.

This can be used to create a 'squeezed' image by doing the following:

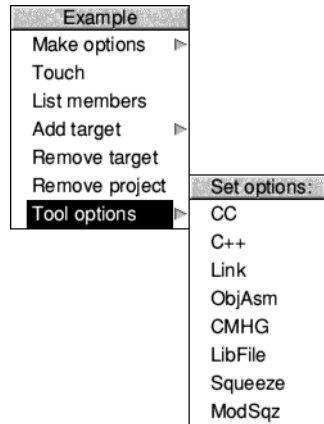
- When you first create the project use a final target name such as `!RunImageU` for the unsqueezed binary. Insert all your sources and library files to this target.
- Then add a target (called, for example, `!RunImage`) with its 'tool' set to Squeeze.
- Insert the `@. !RunImageU` as the only member for this target.

If you used the example names above, and you now make the target `!RunImage`, you will get a squeezed final binary.

Setting tool options

In order to make final targets and object files which will combine to make those final targets, a number of tools such as compilers, assemblers, linkers and library constructors will be used. These tools will typically have a set of options which are normally specified from a dialogue box when using the tools under the control of

the FrontEnd module. It is possible to set the options for a particular tool's use under Make (for a given project) by following the **Tool options** submenu from the project's dialogue box menu.



This will show a list of all the tools which have registered themselves for use with Make (for example, CC, ObjAsm, Link etc.). Clicking Select or Adjust on a tool's name in this list will result in the options dialogue box for that tool being displayed. This dialogue box can then be used to set the options for the tool; these will be translated into command-line options and entered into the `toolflags` section of the Makefile for the project.

Removing projects

A project can be removed from the list of projects maintained by Make by choosing **Remove project** from the project's dialogue box menu. This simply means that it is removed from the list of projects which can be opened from Make's **Open** submenu; the Makefile for the project is still retained.

You will also be asked if you want to remove the files which store the toolflags for the project. If you intend never to reinstate this project as one maintained by Make, then answer **Yes** to this query. If you are just temporarily removing this project from the list, then answer **No**, so that the toolflags state for this project is saved.

If you later wish to reinstate a removed project, this can be done by dragging the Makefile for the project onto the Make icon.

Creating a final target for a project

There are two ways of creating a final target for a project:

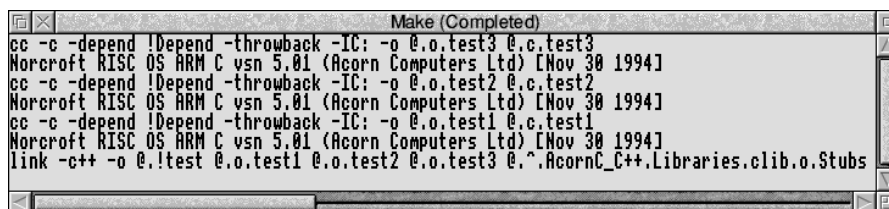
- If you click on **Make** in a project's dialogue box, Make will make the target which is currently showing in the Target icon. An alternative target can be selected by clicking the up and down arrow icons to move through the list of possible final targets.
- If you double click on a filer icon of type Makefile (0xFE1), and you have enabled the **Auto Run** options from Make's **Options** menu, then Make will make the first target that it finds in the Makefile (which will be the target specified when the project was created).

In both of the above cases, the amu program is run pre-emptively using the TaskWindow module to make the chosen target. The space available to load and start up amu is determined by the Wimp **Next** slot. If you get errors such as:

No writable memory at this address

when you run a Make job, try adjusting the **Next** slot.

The output from this process appears by default in a scrollable, saveable text window (or in a summary dialogue box if this option is selected in the **Display** submenu):



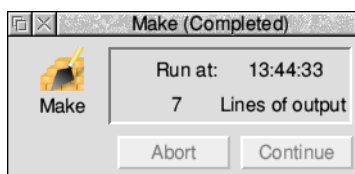
```

Make (Completed)
cc -c -depend !Depend -throwback -IC: -o @.o.test3 @.c.test3
Morcroft RISC OS ARM C vsn 5.01 (Acorn Computers Ltd) [Nov 30 1994]
cc -c -depend !Depend -throwback -IC: -o @.o.test2 @.c.test2
Morcroft RISC OS ARM C vsn 5.01 (Acorn Computers Ltd) [Nov 30 1994]
cc -c -depend !Depend -throwback -IC: -o @.o.test1 @.c.test1
Morcroft RISC OS ARM C vsn 5.01 (Acorn Computers Ltd) [Nov 30 1994]
link -c++ -o @.!test @.o.test1 @.o.test2 @.o.test3 @.~.AcornC_++.Libraries.clib.o.Stubs

```

This window is read-only, you can scroll up and down to view progress, but you cannot edit the text without exporting it to an editor. To indicate this, clicking Select on the scrollable part of this window has no effect.

Clicking Adjust on the close icon of the output window switches to the output summary dialogue box:



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

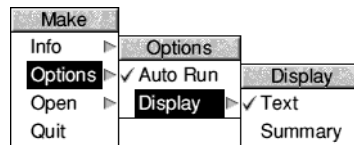
Both the above output displays follow the standard pattern of all the non-interactive desktop tools. The common features of the non-interactive desktop tools are covered in more detail in the chapter *General features* on page 103. Both output displays, and the menus brought up by clicking Menu on them, offer the standard features allowing you to abort, pause, or continue execution, save output text to a file, or repeat execution.

Saving a project without Making it

If you have made changes to a project, and wish these to be written back to the project's Makefile without actually making a target, then click on **Save** in the dialogue box.

Setting Make main options

The **Options** submenu from the Make icon bar menu allows you to set two options: **Auto Run** and **Display**.



Selecting **Auto Run** means that when you double-click on a file of type Makefile (0xFE1) from a directory display, the AMU program is immediately invoked to make the first target found in the Makefile; if you do not select **Auto Run**, then double-clicking on a Makefile merely adds the project to Make's list of maintained projects (if it is not already there), and opens the dialogue box for that project (bringing it to the front of the WIMP's window stack if it is already open).

In the **Display** submenu, you can choose whether the output of all Make processes is displayed in a scrolling text window or in a summary dialogue box.

Text-editing Makefiles

You can use a text editor to customise a project's Makefile. There is a section of the Makefile, following the active comment `User-editable dependencies`, which is left untouched by Make. All other sections of the Makefile will be

over-written and so should not be edited using a text editor (unless you are thoroughly familiar with the operation of Make). The full format of a Makefile is described in *Makefile format* on page 69.

Note that the actual Makefile is only read in if Make is re-loaded and the project then opened, just re-opening the project without re-loading Make is not sufficient.

A good example of how this could be used, is to create a rule which removes an application's binary image and the object files used to create it, so that the next 'make' will remake all objects. This is done by entering in the user-editable section the following lines:

```
clean:; remove !RunImage
      wipe o.* ~cf
```

Using conventional Makefiles

If a file of type `Makefile`, which does not comply to the Makefile format, is double-clicked, or if a file of type `Text` or `Data` is dragged onto the Make icon, it is not registered as a project. Instead Make runs the AMU program with this file as its input Makefile. This allows the use of Makefiles from other systems, and ones which do not fit into the project-oriented way of working required by Make.

Makefile format

The Makefile which is used to maintain a project is a file of type `0xFE1` (`Makefile`), and contains normal ASCII text. This text is arranged into a number of sections which are separated by active comments. For a detailed description of Makefile syntax see appendix *Makefile syntax* on page 209.

Below, we describe each of these sections, beginning with their respective active comments:

<code># Project project_name:</code>	This gives a name to be used for the project in the Open submenu.
<code># Toolflags:</code>	<p>This section has a set of default flags for each of the tools which have registered themselves with Make, for automatic inclusion in a Makefile. The tool will have done this by writing lines (described in the <i>Programmer interface</i> on page 71) into:</p> <pre><Make\$Dir>.choices.tools.</pre> <p>Each macro in the Makefile will be of the type:</p> <pre>toolflags = ...</pre> <p>e.g. <code>ccflags = -c</code></p>
<code># Final targets:</code>	<p>This section contains the rules for making the final targets of the project. For example:</p> <pre>!RunImage:link \$(linkflags)</pre> <p>This information is obtained when the project was created (from the Name and Tool icons in the New Project dialogue box).</p>
<code># User-editable dependencies:</code>	This section is left untouched by Make, and can freely be edited by the user. This allows rules to be added which are specific to a particular project; for example, it may copy sources from a file server to your local disc, before doing a compilation.

- # Static dependencies: This section contains rules for making an object file from corresponding source. It does not refer to `include` files etc. (described in Dynamic dependencies).
- # Dynamic dependencies: This section contains the rules which are created by Make by running the relevant tool on a source file to ascertain its dependencies (e.g. `cc -depend`).

Programmer interface

The following information is given for programmers wishing to add new desktop tools to be used with the Make application.

If you wish to use a tool with Make, which does not come with Acorn C/C++, you can use either of the following two methods:

- Write a description or Setup file (see appendix *FrontEnd protocols* on page 231) for the tool for use by the FrontEnd module and register it with Make as described below in the section *Registering command-line tools with Make*.
- Write a WIMP frontend for the tool which complies with the details given below in the section *Message-passing interface for setting tool options*.

Registering command-line tools with Make

A command-line tool which will be run under the control of the FrontEnd module (for setting its options in a Makefile), will need to append lines of the following format to the file `<Make$Dir>.choices.tools`:

<i>toolname</i>	Name of tool
<i>string</i>	Extension
<i>flags</i>	Default flags for use by Make
<i>rule</i>	Rule for converting sources to objects
<i>pathname</i>	Full pathname of file containing application description
<i>pathname</i>	Full pathname of file containing Frontend setup commands

All the above lines should be terminated by the C newline character `\n`.

Message-passing interface for setting tool options

When the user selects a tool name from the **Tool options** submenu, Make issues a star command to get the frontend module to start up a Wimp frontend for the chosen tool (without an icon appearing on the icon bar). The setup dialogue box for that tool is then displayed, with the Run icon replaced by an **OK** box.

The user can then set options for that tool. A suitable set of command-line options is returned by the generalised frontend, to be used as that tool's *toolflags* entry in the Makefile.

If the star command fails (presumably because the frontend module is not active or because there is no description for the chosen tool), then Make broadcasts a WIMP message (recorded delivery), to see if any application can deal with the request. This is to allow expansion of the system to incorporate other WIMP-based compilers, assemblers, etc., which other parties wish to provide for use under the control of Make.

The WIMP message has the format:

Byte offset	Contents
+16	DDE_CommandLineRequest (reason code) (0x81401)
+20	Make's internal handle
+24 ...	null-terminated application name

If you have written an application which needs to respond to this message, then your application should:

- 1 Acknowledge the WIMP message. You must also store the taskhandle of Make.
- 2 Display a dialogue box to allow the user of your application to set options appropriately.
- 3 When the user has chosen the options, send back a WIMP message to Make, with the following format:

Byte offset	Contents
+16	DDE_CommandLineResponse (reason code) (0x81400)
+20	Application's handle
+24 to +36	Application's name
+36 ...	null-terminated command-line options



SrcEdit is a text editor, based on the RISC OS editor (Edit), with extra features to make it more suitable to create and edit program sources.

You can control SrcEdit from a menu tree, which is described fully in this chapter. However, many menu choices are available directly from the keyboard; once you are familiar with SrcEdit, you may find that you prefer this method. These keystroke equivalents are listed later in this chapter.

Starting SrcEdit

You can load SrcEdit either by double-clicking on the !SrcEdit icon from a directory display, or by double-clicking on a file of type Text (0xFFFF). You will then see an icon similar to that of Edit on the iconbar (a pen and program listing).

Typing in text

When you first open a new SrcEdit window, an I-shaped bar – the *caret* – appears at the top left of the window. This is where text will appear when you start typing. You can open more SrcEdit windows, but only one of them will have a caret in it: this is called the current window. It is also identified by the fact that parts of its border appear in cream rather than grey. You can type only in the current window.

If you type in some text without putting in any carriage returns, and using the system font (the default font) you will find that the window scrolls sideways. This is because the default SrcEdit window is not as wide as the screen. You can break your text into lines by pressing Return. Alternatively, click on the Toggle Size icon to extend the window to the full screen and avoid having to scroll sideways. There is another way of getting all your text into the window, using the **Format** command; this is described later.

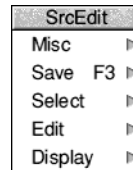
As you type, you will notice that SrcEdit fills the current line and then carries on to the next line, often breaking words in the middle. Ignore this for the moment, as there is a menu option (**Wordwrap**) that will take care of it, and this will be described later.

Inserting and deleting text

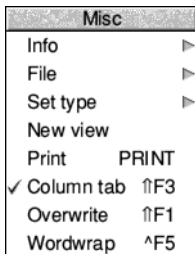
If you need to insert or delete text, position the caret where you want to make the alteration by moving the pointer there and pressing Select. You can insert text simply by typing. If you want to delete the character to the left of the caret, press Backspace; to delete a character to the right of the caret, press Delete. If either key is held down the auto-repeat will come into effect, deleting more characters.

SrcEdit menus

The top level menu for text windows contains the following options:



The Misc menu



This menu offers eight options:

Info tells you about SrcEdit, including the version number of your copy of the program.

File gives information about the file you are working on, in particular:

- whether it has been modified since you last saved it;
- what type of file it is: for example, a Text File or a Command file (its icon, if it has one, is also shown);
- its name, including the full directory pathname;
- its size, in number of characters;
- the time and date it was last saved (or if you have not saved it yet, the time and date when it was first created).

Set type lets you change the file type that will be used when the file is saved.

New view opens a second window on the same text. This allows you to look at two parts of the same document, and makes many actions such as copying from one part of a document to another much easier. Remember that you are looking at one document, not at two separate copies of it: to illustrate this, try looking at the same part of a document in two views (not the way you will normally use **New view!**); enter some changes in the first view and you will see the same changes appearing in the second view. This is particularly useful with large documents.

Print will print the file. See *Printing a SrcEdit file* on page 88 for more information.

Column tabs switches on a different type of tab insertion; for more detail see the section *Laying out tables – the Tab key* on page 89. When this option is on, it is ticked in the Misc menu and ColTab appears in the Title bar.

In SrcEdit the default state is to have **Column tabs** on.

Overwrite, means that each character you type replaces the character at the cursor, instead of pushing the cursor aside and inserting the new character. When this option is on, it is ticked in the Misc menu and Overwrite appears in the Title bar.

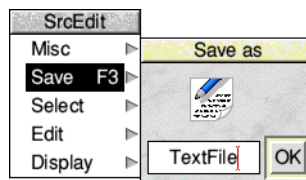
Wordwrap prevents words being split over line-ends as you type. When this option is on, it is ticked in the Misc menu and Wordwrap appears in the Title bar. Do not confuse this option with **Wrap**, selected from the Display submenu. **Wordwrap**, unlike **Wrap**, inserts a newline character (which is there although you cannot see it on the screen) when the cursor moves to a new line.

Saving text – the Save menu

The Save menu allows you to save a complete file; you can also save part of a file using the Select menu.

In order to save a file in the easiest way, you need to have on the screen the directory display for the directory where you want to save the file.

- 1 Click Menu over the SrcEdit window, and move to the **Save** submenu. A dialogue box appears, containing an icon, the current filename, and an **OK** button (as a short-cut you can also display this dialogue box by pressing F3).



- 2 If the file has not been saved before, SrcEdit offers you a default filename of 'TextFile'. If you want a different name, use Backspace or Delete (or press Ctrl-U) to delete TextFile, then type in the name you want.
- 3 Place the pointer on the icon in the box and drag the icon into the directory display where you want to keep the new file. An icon for the file then appears in the directory window.

This action assigns a full pathname to the file, as you will see from the Title bar of the SrcEdit window. When you have made some changes to the text and want to save the file a second time, use the Save option again, but this time, provided you want to use the same filename, you can save the file by clicking the **OK** box. Saving the file with the same name overwrites your old file with the new information.

You can also save **part** of the text, typically for printing or transferring to another application, using the **Select/Save** option, described in the next section.

Manipulating blocks of text – the Select menu

You can *select* blocks of text, then manipulate them.

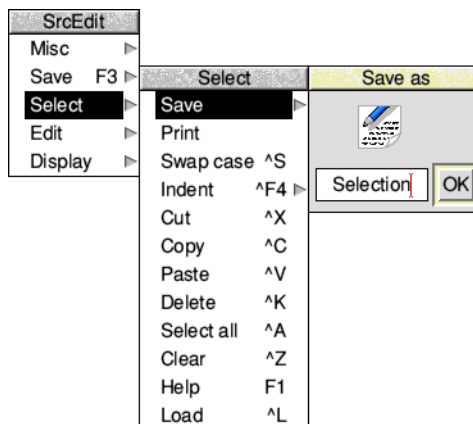
The simplest way to select a block is to position the pointer where you want the block to start, click and hold down the Select button, then drag the pointer to the end of the block and release the button. The selected block of text is highlighted.

If necessary, you can then use Adjust to ‘adjust’ the ends of the block. Position the pointer exactly where you want the block to start or finish, click Adjust and the block lengthens or shrinks accordingly. This is particularly useful when you want to select a block that extends beyond the part of the text you can see in the window. Select a few words or lines at the start of the block, scroll until you can see the point where you want the block to end, place the cursor there and click Adjust.

Here are some other ways of selecting blocks of text:

To	Do this
select a single word	double-click Select
select a single line	triple-click Select
extend block to whole word	double-click Adjust
extend block to include current line	triple-click Adjust

Once selected, text can then be saved, printed, case-swapped, cut, copied, pasted, deleted, de-selected (cleared) or indented by choosing options from the Select menu:



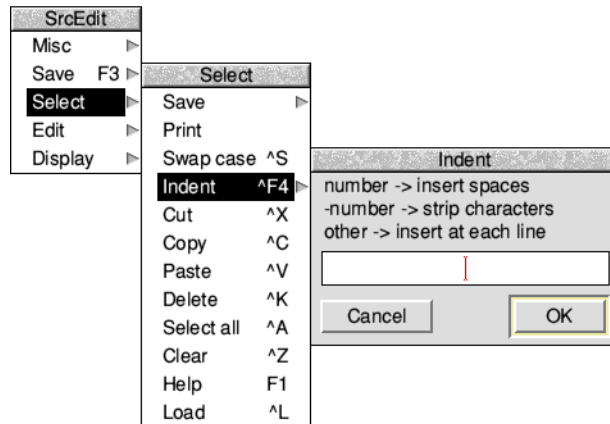
To **Save** a selected block, move to **Save** from the Select menu, and follow the standard saving procedure. Use this option to copy a selection into another SrcEdit window; open a new window and drag the icon into it. The copied block will appear after the current caret position in the destination window. The caret is also moved to the end of the copied text.

To **Print** a selected block of text, use **Print** from the Select menu.

The **Swap case** option will swap all the lower case letters in the selected text for upper case letters and vice-versa. Pressing Ctrl-S has the same effect.

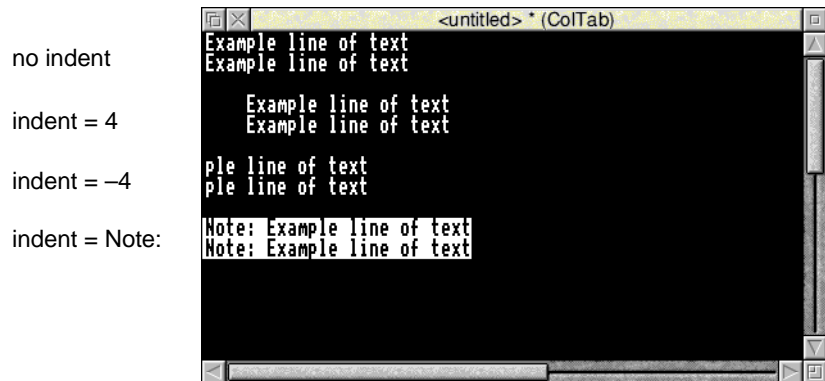
Indent allows you to indent a selected block of text. The indent is defined in character spaces. You can also use **Indent** to add a text prefix to the beginning of each line of a block.

To indent a selected block of text, call up the **Indent** submenu:



You can then type in three different types of indent:

- A positive number gives you an indent of the specified width.
- A negative number, -5, for example, deletes the specified number of spaces or characters from the beginning of the block line; use this to cancel an indent.
- You can also type in text: IGNORE, or Note, for example. This will then appear at the beginning of every line in the selected block. You can remove this text by indenting with a suitable negative number.



To make a **Copy** of a selected block of text, select (highlight) your block of text, then call up the **Select** submenu and click on **Copy** to copy the selected text to the clipboard. The original block remains selected. You can also copy the block by pressing Ctrl-C. Next, position the caret where you want the copy inserted, then

call up the **Select** submenu and click on **Paste**, or press Ctrl-V. Keep clicking on **Paste** to make as many copies as you want. To undo an action, choose **Undo** from the SrcEdit menu.

To move a selected block of text, select your block of text and click on **Cut**, or press Ctrl-X. This moves the selected text to the clipboard, removing it from the screen. You can then place the caret where you want the text moved to and click on **Paste**, again Ctrl-V may be used instead of the menu.

To **Delete** a selected block of text, click on **Delete**, or press Ctrl-K. The marked block then disappears, but note that unlike **Cut** this does not affect the contents of the clipboard. (**Undo** – in the Edit menu – allows you to reverse any changes or deletions made in the Select menu).

To **Select all** the text in the document, click on **Select all** or press Ctrl-A.

To **Clear** or 'deselect' a block of text you have previously selected, click on **Clear** or press Ctrl-Z. The highlighted block reverts to normal and the block is no longer selected.

By selecting some text and choosing the **Help** submenu, some language-specific help can be given on that selection. This help is supplied by a language package, which will have registered a help file containing typically a list of help messages for keywords of a programming language (e.g. the C `printf` function).

The **Load** submenu allows you to load a file into the editor, whose name is given by the current selection. The rule used to determine the name of the file to be loaded (assuming the current selection is in a file whose name has the form *DirectoryPath.LanguageExtension.foo*) is as follows:

- 1 Try to load file *Selection*.
- 2 If (1) fails try to load file:
DirectoryPath.LanguageExtension.Selection
- 3 Try to load file *DirectoryPath.Selection*.
- 4 If (3) fails try the comma-separated list of directories entered by the user from the **Search Path** entry in the **Options** submenu of SrcEdit's icon bar menu, with *Selection* appended as a leafname.
- 5 If (3) and (4) fail, try the comma-separated list of directories which are registered for the current language (see *The SrcEdit icon bar menu* on page 95 for details of how to set the current language).

For example, you may have a C source file with a line `#include "defs.h"`. By selecting `defs.h` and typing Ctrl-L the header file `defs.h` will be loaded into SrcEdit (providing it can be found on one of the search paths).

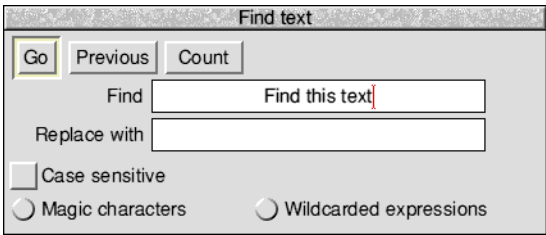
The Edit menu

Edit		
Find	F4	▶
Goto	F5	▶
Undo	F8	
Redo	F9	
CR<->LF	^F8	
Expand tabs	^⇧F1	
Format text	^F6	▶

The first option in the Edit menu is **Find**. At its simplest, this allows you to locate any character(s) in your file. You can also use it to replace text with other text. To make sure that the search is complete, always position the caret at the start of the file before giving the Find command. In the following description, the text being searched for is referred to as a 'string'; it may consist of any sequence of letters, numbers, spaces or other characters.

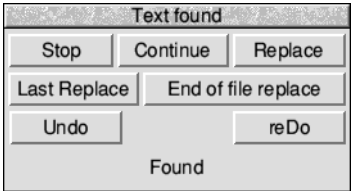
Searching for a string of characters

To use Find without doing anything with the found string, choose **Find** in the Edit submenu: the **Find text** dialogue box appears, with the caret in the **Find** box. Type in the string you want to locate and press Return. The caret then moves to the **Replace with** box.

A screenshot of the 'Find text' dialog box. It has a title bar 'Find text'. Inside, there are three buttons: 'Go' (highlighted with a yellow border), 'Previous', and 'Count'. Below these is a 'Find' label followed by a text input field containing 'Find this text'. Underneath is a 'Replace with' label followed by an empty text input field. At the bottom, there are two radio buttons: 'Case sensitive' (which is selected) and 'Magic characters'. To the right of 'Magic characters' is another radio button labeled 'Wildcarded expressions'.

Since on this occasion you do not want to replace the found strings, either click on **Go**, press Return or press **F1**.

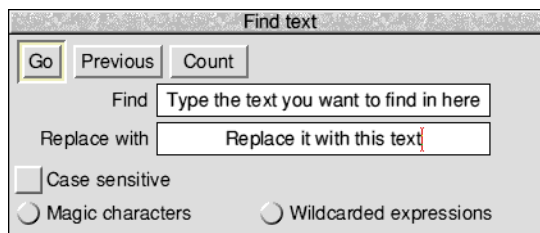
Edit finds the first occurrence after the caret of the word in your file, then displays the **Text found** dialogue box, indicating the operations available.

A screenshot of the 'Text found' dialog box. It has a title bar 'Text found'. Inside, there are several buttons arranged in rows: 'Stop', 'Continue', and 'Replace' in the first row; 'Last Replace' and 'End of file replace' in the second row; 'Undo' and 'reDo' in the third row. At the bottom center, the word 'Found' is displayed.

To look for the next occurrence of your string, click on **Continue**. To abandon the search, click on **Stop** or press Escape.

Replacing a string of characters with a new string

To use Find for replacing a string with a new string, go to the **Find text** dialogue box as before, but this time, insert the new string into the **Replace with** box. Then press Return, and the **Text found** dialogue box appears.



Click on **Replace** to substitute the new string for the old string; if you do not want to change this particular occurrence of the old string, click on **Continue** and SrcEdit moves on to the next one.

If you click **Last Replace**, SrcEdit replaces the currently found instance of the string, but does not search for further occurrences.

If you click on **End of file Replace**, SrcEdit finds and replaces all occurrences of the string from the present one forward to the end of the file, without stopping at each one for instructions.

Clicking on **Undo** takes you back to the last string replaced and returns it to the original version; click **Redo** to change it back again.

The display at the top of the dialogue box keeps you informed of the state of the search; if SrcEdit cannot find the word you have specified, it displays the message **Not Found**.

Using keyboard short-cuts

Besides using the Select button, you can control all these options from the keyboard; the particular keys are indicated by the capital letters in the dialogue box. Press S and the search **Stops**, press C and it **Continues**, D and it will **reDo**, and so on. Pressing Escape or Return also stops the search and removes the **Text found** window.

Other useful facilities

Note that you can use Find to delete strings in a text, by entering nothing in the **Replace with** box, and clicking on **Replace** in the **Text found** dialogue box, thus replacing the found string with nothing: deleting it, in effect.

There are several other useful facilities in the **Find text** dialogue box:

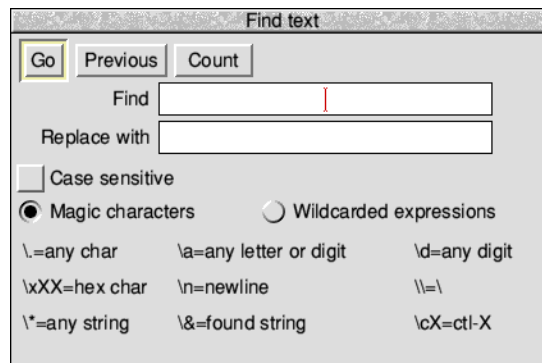
- You can carry out the last Find and Replace operation again, by clicking **Previous** (or by pressing F2).
- You can specify a string and ask SrcEdit to count the number of times it occurs in your file (from the caret position to the end of the file) by clicking on **Count** (or by pressing F3).
- By default, Find makes no distinction between upper and lower case characters – Hello will match to both HELLO and hello, or for that matter, hELLo – you can specifically ask it to match case by clicking next to **Case sensitive** (or by pressing F4). Hello will then match only Hello. Case sensitivity remains selected until you deselect it by clicking again.

Magic characters and their meanings

You can also use the Find facility to search for classes of characters. To activate this feature, click on **Magic characters** (or press F5) in the **Find** dialogue box.

Magic characters are indicated by a \ character, as shown in the lower half of the dialogue box, which shows you the available characters.

Type these characters in directly as shown in the window.



The magic characters operate as follows:

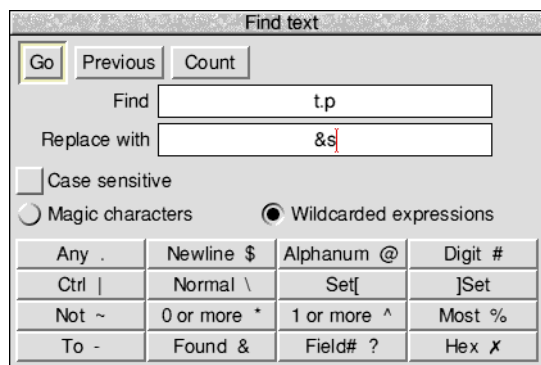
Character	Meaning
\.	matches any character at all, including spaces and non-alphabetic characters.
\a	matches any single alphabetic or digit character. So t\ap matches tip, tap, and top, but not trap.
\d	matches any digit (0 to 9).

Character Meaning

\xXX	matches characters by their ASCII number, expressed in hexadecimal. Thus \x61 matches lower-case a. This is principally useful for finding characters that are not in the normal printable range.
\n	matches the newline character (remember that to the computer, this is a character just like any other).
\\	enables you to search for a string actually containing the backslash character \ while using magic characters. To search for the strings cat\ a or cot\ a, enter c\at\ a.
*	matches any string (including a string consisting of no characters at all). This is really only useful in the middle of a search string. For example, jo*n matches jon, john, and joha an .
\&	is used in the Replace with box to represent the found string : the string matched in the search. This is particularly useful when you have used magic characters in the Find string. For example, if you have searched for t\ap, and you want to add an s to the end of all the strings found, \&s in the Replace with box will replace tip, tap and top by tips, taps and tops.
\cX	matches Ctrl-X, where X is any character.

Wildcarded expressions and their meanings

There is also a facility for specifying wildcarded expressions in search strings. In order to use this facility, click on **Wildcarded Expressions** (or press F6) in the **Find** dialogue box.



Click on the wildcard character you wish to enter and it is copied into the text box.

The wildcard characters operate as follows:

Character	Icon name	Meaning
.	Any	matches any single character.
\$	Newline	matches linefeeds.
@	Alphanum	matches any alphanumeric character. A to Z, a to z, 0 to 9, and _
#	Digit	matches 0 to 9.
	Ctrl	matches any control character. For example, to search for Ctrl-z, type in z
\	Normal	matches any character following it even if it is a special character. # would be searched for as \#.
[]	Set	matches any one of the characters between the brackets. This is always case sensitive.
—	To	[a–z] would match any character (in the ASCII character set) from a to z.
~	Not	does not match character. ~C matches any character apart from C. This can also be applied to sets.
*	0 or more	matches zero or more occurrences of a character or a set of characters. T*O matches T, TO, TOO, TOOO etc.
^	1 or more	matches one or more occurrences of a character or a set of characters. T^O matches TO, TOO, TOOO etc.
%	Most	%c is the same as ^c, except when used as the final element of a search string. In this case the longest sequence of matching characters is found.

Character	Icon name	Meaning
&	Found	refers to the whole of the 'Find' text. It is used in the Replace with box to represent the 'found string': the string matched in the search. This is particularly useful when you have used wildcard characters in the Find string. For example, if you have searched for <code>t.p</code> , and you want to add an <code>s</code> to the end of all the strings found, <code>&s</code> in the Replace with box will replace <code>tip</code> , <code>tap</code> and <code>top</code> by <code>tips</code> , <code>taps</code> and <code>tops</code> .
?	Field	If a string was found that matched the search pattern, then <code>?n</code> refers to the part of the found string which matched the <code>n</code> th ambiguous part of the search pattern, where <code>n</code> is a digit from 0 to 9. Ambiguous parts are those which could not be exactly specified in the search string; e.g. in the search string <code>%#fred*\$</code> there are two ambiguous parts, <code>%#</code> and <code>*\$</code> – which are <code>?0</code> and <code>?1</code> respectively. Ambiguous parts are numbered from left to right. (Only to be used in the Replace with string).
X	Hex	X <code>nn</code> matches the character whose ASCII number is <code>nn</code> , where <code>nn</code> is a two-digit hex number. X <code>61</code> matches lower-case <code>a</code> . This is principally useful for finding characters that are not in the normal printable range.

The full power of the wildcard facility can be illustrated by a few examples.

- To count how many lower case letters appear in a piece of text:

Find: **[a-z]**

and click on **Count**.

- To count how many words are in a piece of text:

Find: **%@**

and Click on **Count**.

- To surround all words in a piece of text by brackets:

Find: **%@**

Replace with: **(&)**

and click on **GO**, then on **End of File Replace** in the Found dialogue box

- To change all occurrences of strings like `#include "h.foo"` into `#include "foo.h"`:

Find: `\#include "h\.%@"`

Replace with: `#include "?0.h"`

and click on **GO**, then on **End of File Replace** in the Found dialogue box

- To remove all ASCII characters, other than those between space and `~`, and the newline character, from a file:

Find: `~[-\~$]`

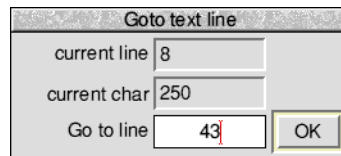
Replace with:

and click on **GO**, then on **End of File Replace** in the Found dialogue box (i.e. find all characters outside the set from the space character to the `~` character, and newline, and replace them with nothing). In fact this could be written without the `\`, since `~` would not make sense in this context if it had its special meaning of **Not**, ie:

Find: `~[~-$]`

Other options on the Edit menu:

To send the caret to a specific line of text, use the **Goto** option. Call up the **Goto** submenu and SrcEdit displays a dialogue box:



Type in the line number you want to move to, then click on **OK**. The dialogue box disappears, and the screen displays the caret, positioned at the beginning of the line you have just specified. Note that this option understands 'line' to mean the string of characters between two presses of Return. If you have not formatted your text, a line in this sense may run over more than one display line.

Undo allows you to step backwards through the most recent changes you have made to the text. The number of changes you can reverse in this way varies according to the operations involved.

Redo allows you to remake the changes you reversed with **Undo**.

CR↔LF allows you to convert the linefeeds in your text to carriage returns (and carriage returns to linefeeds). Carriage returns appear as the characters [0d] in your text.

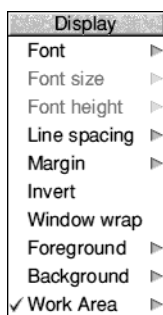
If you convert from linefeeds to carriage returns, the file will be converted to one continuous line, with carriage return characters inserted where linefeeds have been removed. Though it is possible to edit a file in this state, you may find that updating the screen takes a long time. This facility is useful when importing text from other text editors, which may use carriage returns where SrcEdit uses line feeds.

Expand Tabs converts each tab character into eight spaces, since some printers can interpret spaces more easily than the tab character. If you have imported a file that was produced on a word processor, you may find it uses tab characters. These appear in the SrcEdit file as the characters [09] in your text.

Format text allows you to reformat a paragraph of text – from the caret to the next blank line or line starting with a space – so that the lines fill the screen and break correctly at the ends of words. It is useful for tidying up text after editing. Position the caret at the beginning of the paragraph, choose **Format text** in the Edit menu and enter the number of characters per line you want your text to have in the **Format width** dialogue box. Then move the pointer back over the Edit menu and click on **Format text** to format the paragraph.

The setting in the **Format width** dialogue box also controls the length of lines when you are entering text with **Wordwrap** switched on.

The Display menu



Display allows you to change the way your text looks on the screen: you can experiment with fonts, colours, line spacing and margins. However, the features you select do not form part of the text when you save it.

For example, if you choose **New view** in the Misc menu, you will have a second window on your text. If you wish, the Display features in these two windows can be different; this will not affect the text as such.

Font offers you a choice of fonts (typefaces). **System Font** is the default style, and has a fixed character width. For further information on fonts, see the RISC OS *User Guide*.

You can use **Font size** to set the point size (height and width) of the characters displayed on the screen. Either select one of the sizes indicated or position the pointer on the bottom (blank) line of the menu; you can then type in another size.

Font height allows you to set the height of the characters displayed on the screen leaving their width unchanged.

Line spacing increases or decreases the space between lines. Its units are pixels (the smallest unit the screen uses in its current mode). The selected font size assigns a suitable line spacing; this option is therefore used only to increase (or if you type a negative number, to decrease) the given spacing.

Margin sets the left margin, again in pixels.

Invert swaps foreground and background colours, so that black text on white becomes white text on black, and so on.

By default, SrcEdit assumes a text width of 76 characters, but the default window is not as wide as the full screen. You can of course change the number of characters per line (by choosing **Format text** in the Edit menu) or enlarge the window to the full screen by clicking on the Toggle Size icon. Alternatively, clicking **Window wrap** makes your text fit the size of the window. When **Window wrap** is on, you can change the window to any size, and the width of the text will change accordingly. You can revert to the default by selecting **Window wrap** again.

Foreground allows you to set the text to any one of the sixteen colours, by clicking on the selected colour square from the palette displayed.

Background allows you to set the window's background colour, as above.

Work Area allows you to set the extent of your SrcEdit windows so that you can have windows which are wider than the current screen mode. You can specify a wider window in terms of System Font characters in the **Work Area** submenu (the size of System Font characters is used even if the current font used is a fancy font). This is particularly useful if you have sources which, for example, are 80 or 132 characters wide and you are viewing them in a mode with a width of 640 pixels or fewer. The maximum size of window width which can be specified in this manner is 480 System Font characters.

Printing a SrcEdit file

There are two ways of printing a SrcEdit file; however, to use either, you first need to load a printer driver.

If the file you want to print is already loaded into SrcEdit, you can either choose **Print** in the Misc menu, or call up the Save as dialogue box and drag the icon onto the printer driver icon on the icon bar. This will print the current version of the file, whether or not it has been saved.

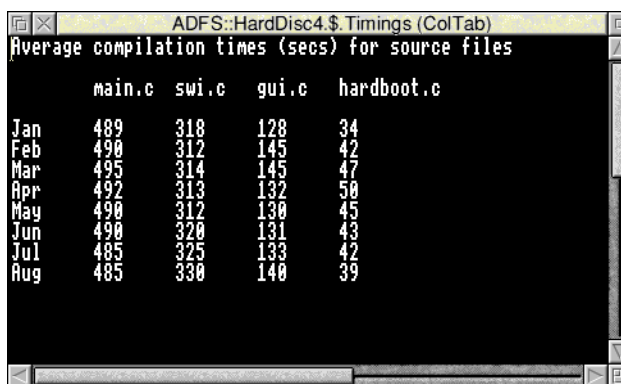
If the file is not loaded into SrcEdit, you can simply drag the file's icon from its directory display onto the printer driver icon. You can also do this if the file is loaded, but if you have made any changes to it since you last saved it, they will not appear in the printed copy; only what has been saved will be printed by this method.

Laying out tables – the Tab key

Tables can be set out in two ways using tabs – as regular columns or irregular columns.

Regular columns

If you want your table to have columns regularly spaced eight characters apart, select **Column tabs** in the Misc submenu. The word ColTab will appear in the window's Title bar to remind you that you have done this. Pressing Tab will then cause the cursor to jump to the next tab position. This is very useful for creating simple tables that will not display much text:



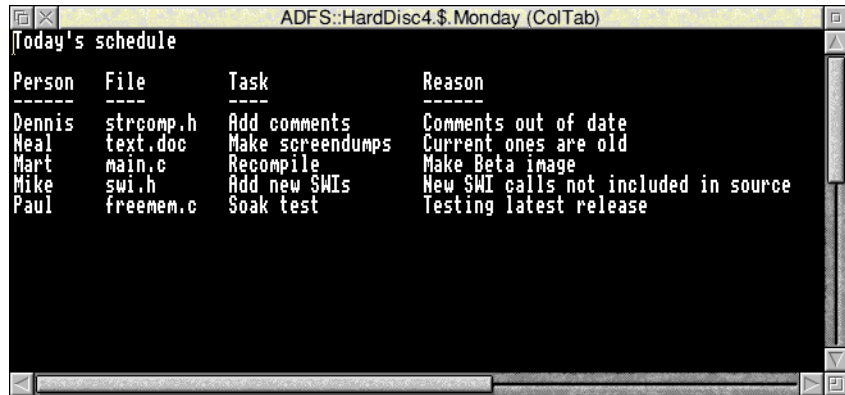
	main.c	swi.c	gui.c	hardboot.c
Jan	489	318	128	34
Feb	490	312	145	42
Mar	495	314	145	47
Apr	492	313	132	50
May	490	312	130	45
Jun	490	320	131	43
Jul	485	325	133	42
Aug	485	330	140	39

Column Tabs is selected by default in SrcEdit.

Irregular columns

To set out a table with irregular columns, make sure that **Column Tabs** in the Misc submenu is **not** selected. Type in the first line – the column headings, for example – as you want it to appear, using spaces to separate the text in the columns. Then press Return. On the next line, pressing Tab will make the cursor jump to the position underneath the start of the next word in the line above.

So, in the following example of a simple diary, the column headings (Person, File, Task and Reason) were typed in using spaces, then the following lines were typed in using tabs (including the dashes used as underlines for the column headings):



ADFS::HardDisc4.\$ Monday (ColTab)

Today's schedule

Person	File	Task	Reason
Dennis	strcomp.h	Add comments	Comments out of date
Neal	text.doc	Make screendumps	Current ones are old
Mart	main.c	Recompile	Make Beta image
Mike	swi.h	Add new SWIs	New SWI calls not included in source
Paul	freemem.c	Soak test	Testing latest release

Note: Both the table layout methods described above will only work with a fixed width font (e.g. the System font). If you create a table and subsequently display the screen in another font, the text in the table will not line up correctly with the column headings.

Reading in text from another file

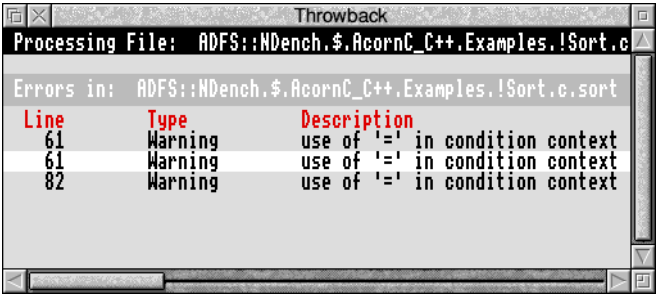
If you want to add all the text from another file into the file you are currently editing, position the caret at the point where the inserted text is to appear. Call up the directory display for the incoming file, and drag its icon into the text window. The entire contents of the source file are then copied into the destination file at the caret position. The caret will appear at the end of the text you have inserted.

Bracket Matching

SrcEdit has a useful bracket-matching facility. If you place the caret to the left of an opening bracket – any of the set (, [, or { – and press F10, the corresponding closing bracket will be highlighted; similarly by placing the caret to the left of a closing bracket – any of the set),], or } – and pressing F10, the corresponding opening bracket will be highlighted. If there is no matching bracket an error message is generated. This is a particularly useful feature in heavily bracketed expressions and blocks of code which extend over a large amount of source code.

Throwback

The purpose of throwback is to allow translators (compilers/assemblers) to signal the editor when they have detected source errors. On receiving such a signal, SrcEdit displays a window which shows the name of the file which was being processed when the error(s) were found, the name of the file in which the error(s) were found, and the relevant line number together with the text of the error message. Also displayed is the severity level of the error(s): Serious Error, Error, or Warning. The complete list of errors is shown in a scrollable window. We shall refer to a single line of this window as an *error line*. You can scroll through these as with any normal text window, using the vertical and horizontal scroll bars.



Double-clicking Select on an error line opens an edit window on the appropriate file (if it is not already open), and highlights the line containing the selected error. The selected error line is also highlighted in the scrollable error window. Clicking Adjust on an error line removes it from the list (presumably you have either corrected the error or have chosen to ignore it). Note that error line numbers refer to the original source when it was processed. You may, in the course of correcting errors, insert or delete lines; the position in the source where errors were detected remains correct despite your edits (provided that the edits are made as a consequence of throwback).

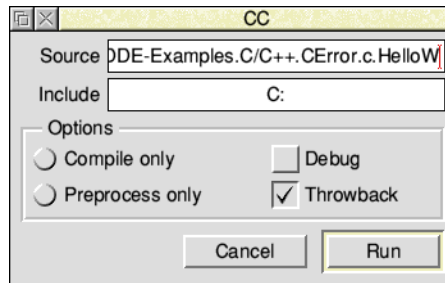
'Informational' throwback is also supported for tools like !Find. The functionality of such a throwback window is the same as for 'error' throwback.

C example throwback session

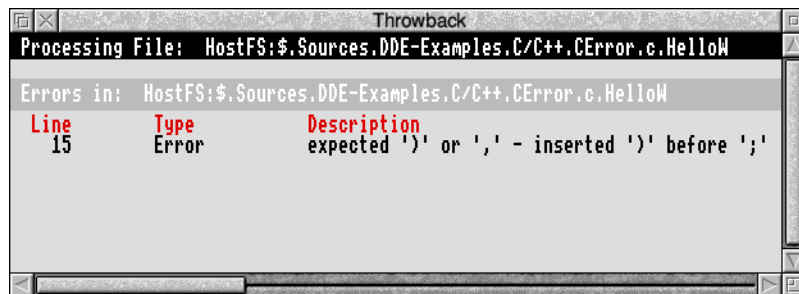
First double click on !SrcEdit and !CC in a directory display to load them as applications with icons on the icon bar. Next open the subdirectory Sources.DDE-Examples.C/C++.CError.c to show the text file Hellow containing the source of the program example of that name.

HelloW is a trivial C program which when run prints Hello World on the screen. It is written to be compiled with an integral link step by CC to form an executable image file. Its source contains a simple error which will be detected by CC when you try to compile it.

Drag the source file HelloW to the CC icon to make the CC SetUp dialogue box appear with the **Source** writable icon initialised to the absolute file name. Ensure that the **Throwback** option is enabled. The correct dialogue box appearance is as follows:



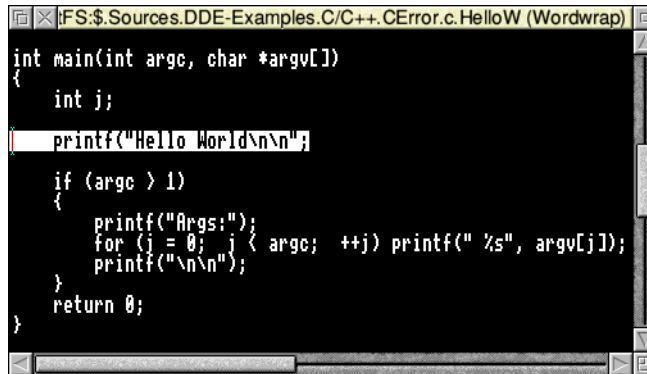
Click Menu on the setup box and ensure that the **Work directory** item on the menu displayed has the default setting of '^'. Click on the **Run** button on the SetUp box to start compilation. This has the normal effect of removing the setup box and putting the CC output display on the screen, but almost immediately afterwards the compiler produces an error and requests SrcEdit to display a Throwback error browser:



Double click Select on the compiler error message:

expected ')' or ',' - inserted ')' before ';'.

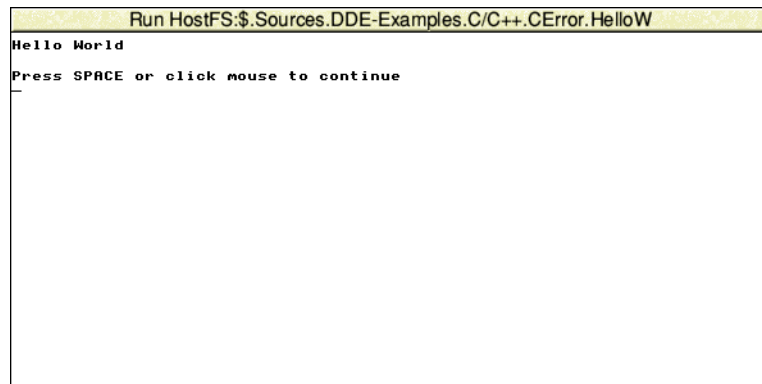
SrcEdit displays the source file with the offending line that caused the error clearly highlighted:



```
int main(int argc, char *argv[])
{
    int j;
    printf("Hello World\n\n");
    if (argc > 1)
    {
        printf("Args:");
        for (j = 0; j < argc; ++j) printf(" %s", argv[j]);
        printf("\n\n");
    }
    return 0;
}
```

Examining this line closely shows that a closing bracket is missing before the ending semicolon. Insert this bracket in SrcEdit and save the file. Click Select on the CC icon bar icon and click on **Run** to repeat the last compilation. If you have changed the HelloW source correctly, the compilation should now complete with no errors, hence without bringing back the SrcEdit browser.

When the CC save dialogue box appears, click on the OK button to save the executable file produced in the directory DDE-Examples.C/C++.CError. Now double click Select on the newly created executable image file in a directory display. The image file should run, printing the Hello World message in a RISC OS run window:



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

Assembler example throwback session

First double click on !SrcEdit, !ObjAsm and !Link in a directory display to load them as applications with icons on the icon bar. Next open a directory display on the subdirectory `Sources.DDE-Examples.ObjAsm.AsmError.s` to show the text file `HelloW` containing the source of the program example of that name.

`HelloW` is a simple assembly language program which when run prints `Hello World` on the screen. It is written to be assembled to an object file by `ObjAsm` then linked to form an executable image file with `Link`. Its source contains a simple error which will be detected by `ObjAsm` when you try to assemble it. The line containing the error is:

```
= "Hello World"13,10,0
```

Examining this line shows that a comma is missing after the close quote. Correct this and you will then be able to assemble the program without error.

C++ example throwback session

First double click on !SrcEdit and !C++ in a directory display to load them as applications with icons on the icon bar. Next open the subdirectory `Sources.DDE-Examples.C/C++.C++Error.cpp` to show the text file `HelloW` containing the source of the program example of that name.

`HelloW` is a trivial C++ program which when run prints `Hello World` on the screen. It is written to be compiled with an integral link step by `CC++` to form an executable image file. Its source contains a simple error which will be detected by C++ when you try to compile it. The line containing the error is:

```
cout << "Hello World\n;
```

Examining this line closely shows that a closing double quote is missing before the ending semicolon. Insert this double quote in `SrcEdit` and save the file. Click **Select** on the C++ icon bar icon and click on **Run** to repeat the last compilation. If you have changed the `HelloW` source correctly, the compilation should now complete with no errors, hence without bringing back the `SrcEdit` browser.

When the C++ save dialogue box appears, click on the OK button to save the executable file produced in the directory `DDE-Examples.C/C++.C++Error`. Now double click **Select** on the newly created executable image file in a directory display. The image file should run, printing the `Hello World` message in a RISC OS run window.

Saving Options

To retain the same set of options whenever you use SrcEdit, set the menu and dialogue box entries to the required configuration and then choose **Save options** from the SrcEdit icon bar menu. The options you have chosen are then saved in two files:

```
<Choices$Dir>.SrcEdit.options
<Choices$Dir>.SrcEdit.liboptions
```

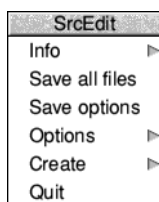
These files are read when SrcEdit starts up. The options saved are:

Foreground Colour	Window work area width
Background Colour	Column tab
Font Width	Overwrite
Font Height	Wordwrap
Left Margin in pixels	Warn multiple edits
Extra spacing between lines	Current language
Window wrap	Search path
Font name	

Setting options in a SrcEdit window

If you set the **Column tab**, **Overwrite** or **Wordwrap** options in the Misc submenu in a SrcEdit window, they will only apply to that session of SrcEdit in that window. To change these three options and retain the new settings whenever you use SrcEdit, you must set them in the Options submenu in the SrcEdit icon bar menu, and then choose **Save options**.

The SrcEdit icon bar menu



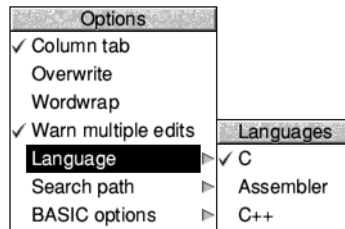
Pressing Menu on the SrcEdit icon on the icon bar produces a menu with the following options:

Info gives you some information about the version of SrcEdit you are using.

Save all files saves all modified buffers, and closes all open windows.

Save options saves the current settings of all SrcEdit options to file, so that there is no need to set the environment variables used to maintain these options.

The **Options** submenu allows you to set the following options:



Column tab, **Overwrite** and **Wordwrap** are similar to the options on the **Misc** submenu in the section entitled *The Misc menu* on page 74. They are used to set the default options for all windows opened by SrcEdit.

Warn multiple edits, if enabled, will warn you when you attempt to load a file which is already loaded in a modified SrcEdit buffer. This reduces the chance of you accidentally editing two copies of the same file, and then saving one over the other. In such a case you will be presented with a dialogue box, giving you the choice of having a read-only copy of the file, a normal editable copy, or to cancel the load of the file. If you choose to have a read-only copy, then the SrcEdit window for the document will have Read-Only in its Title bar and you will be prevented from making any edits to the contents of the document.

The **Language** submenu gives you a list of any language packages which have registered themselves with SrcEdit. You can select which of these languages is current, and this will determine what Help text is available, and also the default search path used when loading from a selection.

Search path – If you load from a selection (i.e. when you have chosen **Load** from the Select submenu), SrcEdit will look in a number of places for the file to be loaded. You may set a comma-separated list of paths to search by typing them into the **Search path** writable icon (described on page 79). Note that each such path should either be a path variable or be explicitly terminated by a dot.

The **BASIC options** submenu provides two options that are used when editing BASIC files. When the **Strip line numbers** option is selected, the line numbers are removed from BASIC files when they are loaded, and when a BASIC file is saved the **Line number increment** option determines the line numbers that are used.

Create leads to a submenu which enables you to open windows for specific types of file: Text, Data, Command, Obey and Make files.

In addition, the **Create** submenu allows you to set up SrcEdit Task windows, these are described in the next section.

Finally, **Quit** stops SrcEdit and removes it from the computer's memory, first presenting you with a dialogue box for confirmation if there are any current files you have not saved.

SrcEdit task windows

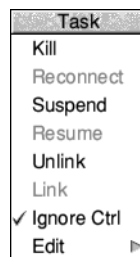
SrcEdit task windows allow you to use Command Line mode in a window. To open a task window, choose **Task window** from the SrcEdit application menu. You can have more than one task window open. When you open a task window, you will see a * prompt. You can now enter commands in the window just as if you were using Command Line mode.

The major advantages in entering commands in a task window instead of at the Command Line prompt are that:

- Other applications continue to run in their own windows while you run the task (this does mean, though, that the task may run more slowly than it would using other methods of reaching the Command Line).
- Commands that you type, plus the output (if any), appear in a conventional SrcEdit window, and may therefore easily be examined by scrolling up and down in the usual way. When you type into the window, or when a command produces output, the window immediately scrolls to the bottom of the text. Anything you type in is passed to the task, and has the same effect as typing whilst in Command Line mode. You can change this by unlinking the window: in this case, anything you type in alters the contents of the window in the same way as any other SrcEdit window, even while a task is running. Any output from the task is appended to the end.

You cannot use graphics in a task window. The output of any commands that use graphics will appear as screen control codes in the task window.

The menu for a task window contains the following options:



Kill stops and destroys the task running in the window.

Reconnect starts a new task in the window, allocating memory to the task from the Task Manager's **Next** slot.

Suspend temporarily halts the task running in the window.

Resume restarts a suspended task.

Unlink prevents the sending of typed-in characters to the task. Instead, they are processed as if the task window were a normal SrcEdit text window.

Link reverses the effect of **Unlink**.

Ignore Ctrl, when selected, prevents any control characters generated by the program from being sent to the window. It also intercepts Ctrl-C and Ctrl-V keystrokes so that instead of being passed on to the task they can be used to copy and paste text (see *Task windows and the clipboard*).

Edit leads to the normal SrcEdit menu. Although this makes available most of SrcEdit's features, you cannot use facilities such as the cursor keys or keys such as Page Up and Home while you are using a Task window.

Task windows and the clipboard

Text in a task window can be copied to the clipboard in the normal way by selecting it and choosing **Copy** from the **Select** submenu, or by pressing Ctrl-C if **Ignore Ctrl** is selected.

You can paste input to a linked task by selecting **Paste** from the **Select** submenu, or by pressing Ctrl-V when **Ignore Ctrl** is in force. The text on the clipboard is sent to the task as if it was being typed.

Some guidelines and suggestions for using task windows

In order to use a task window, you will need to be familiar with Command Line mode. There are some commands which you will find are more useful in a task window than they are directly from the Command Line. In particular:

`*WimpSlot min [max]` can be used to adjust the amount of memory available to the task, which will otherwise start up using the **Next** space allocation. If you want to remove all the memory allocated to a task without closing its window or destroying the task, use the command `*WimpSlot 0 0`.

`*Filer_OpenDir path` opens a new directory display for the directory with the given path. The path must start with a filing system name. For example:

```
adfs::DHarris.$.Research
```

The command `*Spool` should not be used from a task window. Because its effect is to write everything that appears on the screen to the spool file, using `*Spool` from the desktop will produce unusable files full of screen control characters. There is, in any case, no point in using `*Spool`, since the output from the task appears in the window, and can be saved using SrcEdit as normal.

When you run a command in a task window, the computer divides its time between the task window and other activities running in the desktop. You should note that some time-consuming commands, for example, a `*Copy` of a large file, may prevent access to the filing system that they use until the command is complete.

Note that Command Line concepts such as current directory become relevant when you are using Task Windows.

Keystroke equivalents

On occasions, it can be convenient to use the keyboard instead of the mouse, especially once you are familiar with SrcEdit through its menus.

When editing

←, →, ↑, ↓	Move caret one character left, right, up or down.
Shift-←, Shift-→	Move caret one word left or right.
Shift-↑, Shift-↓	Move caret one windowful up or down.
Ctrl-↑	Move caret to start of file.
Ctrl-↓	Move caret to end of file.
Ctrl-←, Ctrl-→	Move caret to start or end of line.
Ctrl-Shift-↑, Ctrl-Shift-↓	Scroll file one line without moving caret.
Backspace	Cut selection, or delete character to left of caret if none.
Delete	Cut selection, or delete character to right of caret if none.
Home	Place caret at top of document.
End	Place caret at end of document.
Shift-End	Delete word at current caret position.
Ctrl-End	Delete line at caret.
Page Up, Page Down	Scroll up or down one windowful.

Keystroke equivalents in the **Select** menu

Ctrl-S	Swap case of text under the selection.
Ctrl-F4	Indent selection.
Ctrl-X	Cut selection to clipboard.
Ctrl-C	Copy selection to clipboard.
Ctrl-V, Insert	Paste from clipboard, replacing selection.
Ctrl-K	Delete selection without affecting the clipboard.
Ctrl-A	Select all.
Ctrl-Z, Escape, Shift-F6	Clear selection.
F1	Request language-specific help.
Ctrl-L	Load file whose leafname is given by selection.

Keystroke equivalents in the **Edit** menu

F4	Display Find dialogue box.
F5	Display Goto dialogue box.
F8	Undo last action.
F9	Redo last action.
Ctrl-F6	Format text block.
Ctrl-F8	Toggle between CR and LF versions of the file.
Ctrl-Shift-F1	Expand tabs.

Keystroke equivalents in the **Misc** menu

Print	Print the file.
Shift-F1	Toggle overwrite mode on or off.
Shift-F3	Toggle column tabs on or off.
Ctrl-F5	Toggle word wrap on or off.

Keystroke equivalents in the Find menu

Note: these keystroke definitions only come into play once the **Find** dialogue box has been displayed (e.g. by typing F4).

↑, ↓	Find / replace text string.
F1	Display Text found dialogue box.
F2	Use previous find and replace strings.
F3	Count occurrences of find string.
F4	Toggle case sensitive switch.
F5	Toggle magic characters switch.
F6	Toggle wildcarded expressions switch.

Keystroke File options

F2	Open a dialogue box enabling you to load an existing SrcEdit file into a new window.
Shift-F2	Open a dialogue box enabling you to insert an existing SrcEdit file at the caret position. If any text is selected, it will be replaced with the loaded file.
Ctrl-F2	Close window.
F3	Save the file in the current window. This is a short-cut to the normal Save as dialogue box.
Shift-F9	Save modified files and close all windows.

This chapter describes those features common to all the Desktop non-interactive tools.

As described in the chapter *Working with desktop tools* on page 11, the Desktop programming tools can be divided into two categories: interactive and non-interactive. The non-interactive tools are those which you set options for and then run, not interacting further until the task completes or is halted. An example of a non-interactive tool is the linker Link, whereas the editor SrcEdit is an interactive tool. The chapters following this each describe an individual non-interactive Desktop tool. Further chapters in the accompanying language user guides describe non-interactive tools specific to programming in particular languages; for example, the language compilers and assemblers themselves.

The non-interactive tools can be further divided into two sub-categories: filters and non-filters. The filter tools are those that take a set of input files and process them to produce output files, examples being Link, Libfile, Squeeze and the language processors. The non-filter tools all perform some immediate action, such as examining text files and presenting you with information as text output. The filter tools are intended to be used both managed and unmanaged by Make (an interactive tool described earlier in this user guide), whereas the non-filter tools are normally just used for unmanaged work.

To start unmanaged use of any of the non-interactive tools, you first double-click Select on a tool application name in a directory display. This loads the tool, putting its application icon on the icon bar (just like any other RISC OS application).

When using the filter type of non-interactive tool managed by Make, there is no need to start each tool and put its icon on the icon bar.

All the non-interactive Desktop tools are implemented as command line programs provided with RISC OS desktop interfaces by the FrontEnd relocatable module, but you do not need to be aware of this when using them, as command lines are automatically generated from your settings of the desktop interface of each tool, making the tools appear to be standard RISC OS applications.

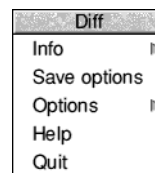
Interface

The interface of each non-interactive tool can be summarised as follows:

- Clicking Menu on the application icon brings up a standard application main menu (for unmanaged use only).
- Clicking Select on the application icon displays the SetUp dialogue box. This allows the user to set options and specify input files etc. A menu is available within the dialogue box enabling other options to be set. Tool SetUp boxes are displayed by Make for managed development.
- Messages generated are output to a Text window or a Summary window. You can toggle between these windows and save the output to a file.
- A processed output file from a filter tool is either saved in a work directory or is saved by you from a standard **Save as** dialogue box which appears when the task has completed without error (unmanaged use only).

The Application menu

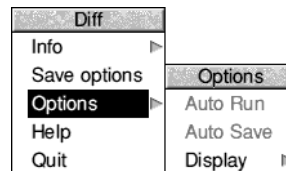
Clicking Menu on the application icon (for example, the Diff tool) gives the following main menu:



Info returns information about the application.

Save options causes the options in the SetUp box, and all submenu options (meta-options) from this main menu, to be saved in a file for later use as defaults when the tool is restarted.

The **Options** submenu allows you to set the following options:

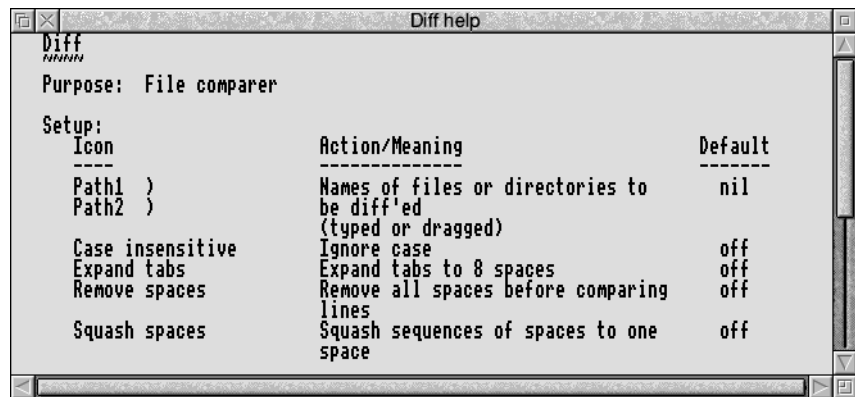


Auto Run will cause the command-line command to be run immediately when a file is dragged onto the icon on the icon bar, without first displaying the SetUp dialogue box. Options remain as they are currently set.

Auto Save suppresses the Save as dialogue box of filter tools if a sensible pathname is available to save the output to. For more details on pathnames see the *METAOPTIONS* section on page 193. Note that 'output' here is used to describe a single file which is produced by running the command-line tool.

The **Display** submenu allows the user to choose whether the tool outputs by default into a text window or a summary window.

Help displays a help file in a scrollable text window, for example:



Quit quits the application.

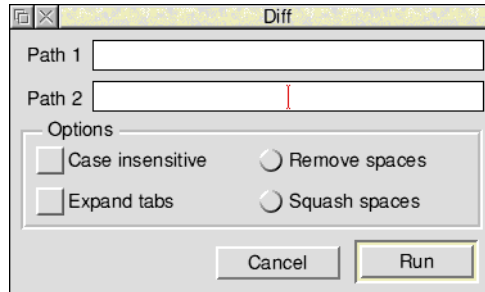
The SetUp box

When working in the unmanaged way, i.e. with the tool application icon on the icon bar, clicking Select on this icon or dragging the name of an input file (if **Auto Run** is off) to this icon displays the SetUp dialogue box. If the SetUp box was displayed by a filename drag, this filename is displayed in the relevant writable icon. Options appear with the previous settings used, making it easy to repeat the last run of a tool.

When working managed by Make, you specify a 'recipe' of tasks to be followed to construct a program from its sources. This recipe is stored as a Makefile, and can be used later. You specify the recipe in terms of what goes in (source files, libraries, etc.), what comes out (e.g. an executable !RunImage file) and the processes followed. The processes followed include specifying the options to be set for the filter tools when they are used. To set these options you follow the **Tool options**

menu item of Make to a list of tools, then Select on the name of the relevant tool. This brings up the SetUp dialogue box of the relevant tool, whether its application icon is on the icon bar or not. The SetUp box appears with options set to helpful default states for managed use.

A typical SetUp dialogue box is that of the application Diff:



The SetUp box for each application is different, but for unmanaged use they all offer the following two action buttons:

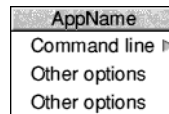
Run runs the tool with the options as set, starting a multitasking task performing the non-interactive job specified. This multitasking depends on the presence of the TaskWindow relocatable module.

Clicking Select on **Run** removes the dialogue box, clicking Adjust on Run leaves the dialogue box on your screen.

Cancel discards any changes made to the options and closes the SetUp box.

The SetUp menu

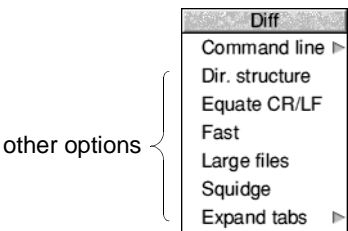
Clicking Menu on the SetUp dialogue box produces a menu with the style of:



Command line leads to a dialogue box showing the command line equivalent of the options set in the SetUp dialogue box. It also shows any extra options set from the **Other options** part of the menu.

Other options are a set of options specific to the particular application.

For example:

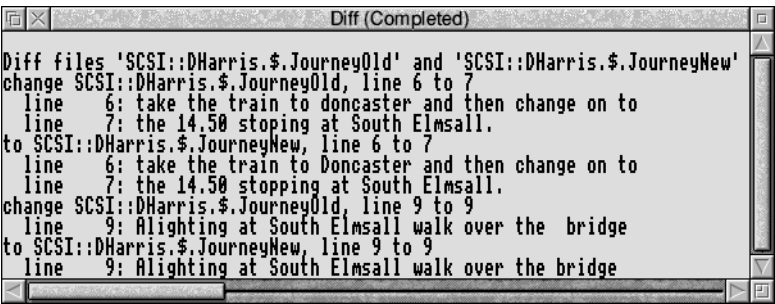


Output

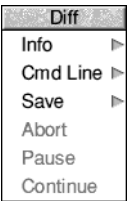
Two types of output window are available for generated messages; Text and Summary.

The Text window

If **Text** has been chosen from the **Display** submenu then a scrollable, saveable text window appears when the tool is running. All textual output sent to the screen by the program appears in the text window. This window can be closed at any time, thus aborting the command-line program. The Title bar of this window shows the name of the tool and the state of the text running, i.e. Running, Completed, Aborted or Paused. An example of a Text window using the application Diff is:



Clicking Menu on a text window displays the following menu:



Info gives information about the program being run.

Cmd Line shows the command line generated and used to run the tool.

Save allows the textual output to be saved in a file.

Abort aborts a running program.

Pause pauses a running program.

Continue continues a paused program.

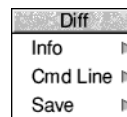
The Summary window

If **Summary** has been chosen from the **Display** submenu then a small summary window, similar to the following, appears when the tool is running:



This summary window displays the sprite of the application and the time at which the command was run. The Title bar is the same as for the text window. There are two action buttons, **Abort** and either **Pause** or **Continue**, which allow the program to be aborted, paused, and continued in an identical fashion to the menu on the Text window.

Clicking Menu on the summary dialogue box displays a menu similar to the following:



Info gives information about the program being run.

Cmd Line shows the command line generated to be used to run the tool.

Save allows the textual output to be saved in a file.

Toggling between the Text and Summary windows

To toggle between the Text and Summary windows click Adjust on the output window's close icon.

Processed file output from filter tools

The numbers and types of files output varies between each filter tool, so for more details see the chapter on the tool in question.

During managed development the saving of processed files is specified by the Makefile, which can be constructed for you by Make.

For unmanaged development, processed files are either saved in positions relative to the work directory, or saved by you from a Save as dialogue box which appears when a job has completed without errors. This box does not appear if you have enabled the **Auto save** option on the application menu.

Part 3 - Non-interactive tools





A8Time is an instruction timing simulator which can help assembly language programmers by calculating the time required to execute sequences of ARM instructions.

It is usually most productive to use a high level language to write software. However, there are two key areas where assembly language is still the language of choice: low-level hardware manipulation (which is confined to the operating system) and high-performance data processing, where even the best compilers can rarely match a human assembly author.

To write the fastest assembly code, you need to be aware of how many clock cycles each instruction takes to execute and how different instructions interact with each other. These details vary considerably from one CPU design to another.

Documentation is sometimes patchy or inaccurate, but a good place to look is often the Technical Reference Manual published by ARM for the CPU that you are targeting.

ARM CPUs have become more complex over time, and with recent designs it can be hard to take account of all the rules that apply to a particular sequence of instructions. A8Time assists you by simulating the instruction scheduler of, as the name suggests, a Cortex-A8 CPU, and enabling you to identify the relative speed of different instruction sequence candidates.

Background to pipelines and scheduling

All ARMs are pipelined. This means that processing of any one instruction is broken down into stages, and adjacent instructions are overlapped such that when one instruction is at one pipeline stage, the previous pipeline stage is processing the next instruction.

The earliest ARMs were relatively simple and used three stages: instruction fetch, instruction decode and instruction execute. (This is the origin of the PC register being 8 bytes ahead by the time an instruction is executed.) Each individual instruction's timing could be predicted in isolation – while the number of cycles could vary depending upon whether its condition code passed or (for multiplies) the data it was working on, the neighbouring instructions had no impact. For multi-cycle instructions, the entire pipeline halted.

Pipelines started getting longer with ARMv4 (StrongARM). This also introduced the concept of delayed results, whereby the following instruction could start progressing through the pipeline sooner if it did not depend upon the output register(s) of earlier ones. Effectively the pipeline forks to enable unrelated instructions to be processed at the same time – a simple form of what is known as superscalar processing.

Where an instruction does depend upon the delayed result of an earlier instruction (known as an interlock) and so cannot proceed, the pipeline is said to experience a stall. The trick to optimising software at the instruction level can be summarised as rearranging the instructions to minimise the number of stalls.

At ARMv5 (ARM9E-S and XScale) interlocks were further refined to take account of the fact that the following instruction does not always need all of its input registers at the same pipeline stage or therefore at the same clock cycle, so the number of stalls could vary depending upon the nature of both the adjacent instructions.

ARMv7 introduced dual-issue pipelining. This means that two instructions can be decoded and start executing during the same clock cycle. However, there are tight restrictions on when this can occur, and these details vary from processor to processor. These restrictions are called hazards, and in addition to the interlocks present in earlier CPUs (which Cortex-A8 calls a “data source hazard”) it features the following:

- Load/store resource hazard: only one load/store can be performed per cycle, though it can be in either pipeline.
- Multiply resource hazard: multiplies cannot be issued to pipeline 1.
- Branch resource hazard: only one branch can be performed per cycle, though it can be in either pipeline.
- Data output hazard: instructions with the same output register(s) cannot be issued in the same cycle.
- Multi-cycle instruction hazard: if an instruction takes more than one cycle, it cannot be issued to pipeline 1.

The additional challenge with Cortex-A8 is to identify when instructions can be rearranged such that they can be dual-issued. As you can imagine, with all the hazards to take account of, it is considerably more complex to calculate the cycle counts of instructions on the Cortex-A8 than with earlier CPUs – hence the need for the A8Time utility.

Many of ARM’s even more recent processors, including the Cortex-A9, A15 and A17, feature out-of-order instruction issue. This effectively takes the burden of instruction scheduling off the shoulders of the assembly author (or compiler).

Using A8Time

A8Time takes as input a text file containing a list of ARM assembler instructions. To make it easier to copy chunks of code from source files, it knows a limited amount of ObjAsm syntax:

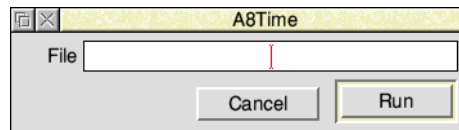
- It recognises ObjAsm's `RN` directive so that you do not need to rename any symbolic register names you may have used.
- Anything else left-aligned is assumed to be a label and is ignored.

It is important to note that A8Time is far from a full CPU simulation. It does not know anything about what any of the instructions actually do, so you need to list them in the order in which they would be encountered by the CPU, including by following any branches yourself. It also makes no allowances for the varied amounts of time it takes to access memory; all the timings it gives assume the data is in the L1 cache.

You should also be aware that not every instruction is recognised by A8Time, although the majority of common ones are.

The SetUp dialogue box

Clicking Select on the application icon or dragging the name of a file (if **Auto Run** is off) from a directory display to the icon brings up the SetUp dialogue box:



The **File** writable icon allows you to specify the name of a file to be processed (typed in or dragged from a directory display).

The SetUp menu

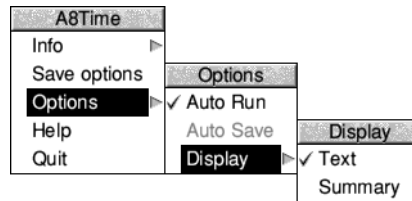
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the A8Time **Command line** options see the section *Command line interface* on page 117.

The Application menu

Clicking Menu on the A8Time application icon gives the following options:



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

Example output

The output of A8Time is a report showing the pipeline usage for the given sequence of instructions. The output appears in one of the standard non-interactive tool output windows. For more details of these see the section *Output* on page 107.

The example below shows the output for the following sequence of instructions:

```
foo      RN      0
bar      RN      1
label
        LDRB    foo, [sp]
        MUL     bar, r2, r3
        ADD     bar, bar, foo, LSL #1
```

A8Time (Completed)		
Cycle	Pipeline 0	Pipeline 1
1	LDRB r0,[sp]	need pipeline 0 for multiply
2	MUL r1,r2,r3	blocked during multi-cycle op
3	MUL (cycle 2)	wait for r0,r1
4	wait for r1	wait for r1
5	wait for r1	wait for r1
6	wait for r1	wait for r1
7	ADD r1,r1,r0,LSL #1	

If the LDRB and MUL instructions are switched, and A8Time is run again, you will see that the code still does exactly the same thing, but does so faster.

Cycle	Pipeline 0	Pipeline 1
1	MUL r1,r2,r3	blocked during multi-cycle op
2	MUL (cycle 2)	LDRB r0,[sp]
3	wait for r0,r1	wait for r0,r1
4	wait for r0,r1	wait for r0,r1
5	wait for r1	wait for r1
6	ADD r1,r1,r0,LSL #1	

Command line interface

For normal use you do not need to understand the syntax of the A8Time command line, as it is automatically generated for you from the SetUp dialogue box settings. The command line syntax for A8Time is:

```
A8Time input [output]
```

Options

input a valid pathname specifying an input assembler file

output a valid pathname specifying an output report file



ABC is a compiler for programs written in BBC BASIC, producing programs which run faster than the original interpreted code, and are often smaller too. ABC works on all versions of RISC OS above 3.00 and with 26 and 32 bit processors.

This application has been supplied as a part of the DDE for many years and, although it is no longer actively developed or supported, it is included for the benefit of those users who need it.

Full documentation can be found in the `Documents.Manuals` directory.





The Acorn Make Utility (AMU) is a tool managing the construction of executable program images, libraries, and so on using operations specified in a Makefile.

AMU uses standard Makefile syntax which is largely compatible with Unix and GNU Make. See appendix *Makefile syntax* on page 209 for details. Some details described in the chapter *Make* on page 59 may also be useful references for AMU, as the command line tool `amu`, which performs the management of program construction, is the same tool used by Make.

Each time that AMU is run, a work directory is set up for that job as the directory containing the Makefile. For the effect of the work directory on each tool, see the chapters on individual tools such as the language processors CC and ObjAsm in this and accompanying user guides.

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

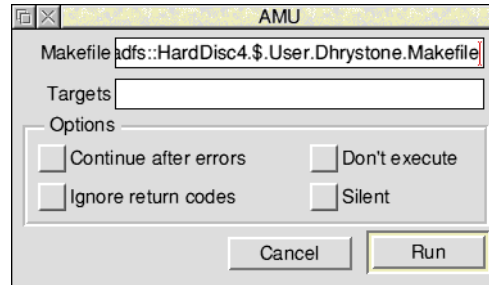
Users of previous versions of AMU should note that the rules for macro priority have changed in this version – see *Macro priority* on page 223 for details.

To simplify the construction of RISC OS programs, a set of Shared Makefiles are provided in the `AcornC/C++.Makefiles` directory. All of the examples in `Sources.DDE-Examples` that have Makefiles have been updated to build in this manner, and provide useful examples of how to do this.

Starting AMU

Since AMU is an alternative tool providing construction management like Make, it is normally used controlled directly from its desktop interface. To start AMU, first double click on !AMU in a directory display to put its icon on the icon bar.

Clicking Select on this icon or dragging the name of a make file (text or Makefile file type) from a directory display to the icon brings up the AMU SetUp dialogue box, from which you control the running of AMU:



Makefile contains the name of the Makefile to be used when AMU is run. If you brought up the SetUp dialogue box by clicking on the AMU icon bar icon, this writable icon contains the previous Makefile used (if any), otherwise it displays the name of the file you dragged to the icon. Dragging another file to this icon replaces its contents with the new name.

Targets contains a space-separated list of the names of the targets in the Makefile to be constructed, and macro predefinitions of the type `name=string`. If this writable icon is empty (default) the first target in the Makefile will be made.

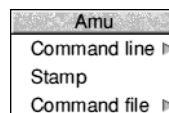
The **Continue after errors** option causes the make job to continue after one of the commands issued by it has returned a bad return code (signalling an error). When the job continues, only those branches of the make job which don't depend on the failed command are executed.

The **Ignore return codes** option causes the make job to continue after one of the commands issued by it has returned a bad return code (signalling an error). When the job continues, all subsequent branches of the make job are executed, as if the return code was good.

The **Don't execute** option stops any commands being executed, instead just printing them to the output window with dependency reasons for each one.

The **Silent** option stops printing of executed commands in the output window.

Clicking Menu on the SetUp dialogue box brings up the AMU SetUp menu, containing a few additional options:



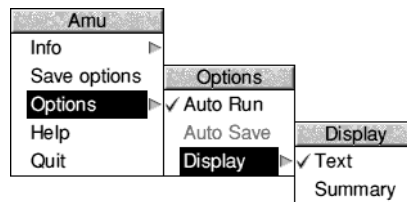
The **Command line** option on the above menu has the standard purpose for non-interactive desktop tools as described in the chapter *General features* on page 103.

The **Stamp** option stops construction of the target, instead causing sources and target to be stamped with current time so that the target appears up to date. This only works if all sources are present.

The **Command file** option leads a writable icon where you specify the name of a file to be written containing commands generated. If you specify a relative filename, this is used relative to the work directory (the location of the Makefile). The commands are written to this file but not executed.

The Application menu

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



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

Example output

Running AMU displays any error messages in the standard text output window for non-interactive tools. If all goes well this window contains no error messages, for example:

```

Amu (Completed)
cc -c -depend !Depend -IC: -throwback -Ddebugging=1 -o @.o.Calc @.c.Calc
Norcroft RISC OS ARM C vsn 5.00d (Acorn Computers Ltd) <alpha test> [Aug 18 1994]
"c.Calc", line 125: Warning: static 'toolbox_events' declared but not used
"c.Calc", line 125: Warning: static 'wimp_messages' declared but not used
"c.Calc", line 125: Warning: static 'operator_event' declared but not used
"c.Calc", line 125: Warning: static 'number_event' declared but not used
c.Calc: 4 warnings, 0 errors, 0 serious errors
cc -c -depend !Depend -IC: -throwback -Ddebugging=1 -o @.o.Main @.c.Main
Norcroft RISC OS ARM C vsn 5.00d (Acorn Computers Ltd) <alpha test> [Aug 18 1994]
Link -o @.!RunImage @.o.Main @.o.Calc C:o.stubs C:o.toolboxlib C:o.eventlib
Squeeze -f @.!RunImage @.!RunImage

```

Command line interface

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

The syntax of the AMU command line is:

```
amu [options] [target1{ target2...}]
```

Options

<code>-e</code>	Enables environment macro override – see <i>Macro priority</i> on page 223 for details.
<code>-f <i>makefile</i></code>	Makefile name (defaults to Makefile if omitted)
<code>-i</code>	Ignore return codes
<code>-k</code>	Continue after errors
<code>-n</code>	Don't execute but do display the commands that would have been executed even if the makefile contains <code>.SILENT</code> and even if the command has a <code>@</code> prefix.
<code>-o <i>commandfile</i></code>	Specify Command file as on SetUp menu
<code>-s</code>	Silent
<code>-t</code>	Equivalent to Stamp on the SetUp menu
<code>-v</code>	Outputs each Makefile command after macro expansion and before it is executed, even if the Makefile contains <code>.SILENT</code> , the command starts with <code>@@</code> or the <code>-s</code> switch was used.
<code>-xn</code>	Enable Makefile debugging. <i>n</i> is a bitfield, so <code>-x1</code> sets level 1, <code>-x2</code> sets level 2, <code>-x4</code> sets level 3, <code>-x6</code> sets levels 2 and 3. The level assignments are: 1: CLI processing, dependency tree creation & command list assignments 2: Macro processing, pattern substitutions etc. 3: Unused 4: Command execution 5: Command expansion of patterns (<code>\$* \$< \$?</code> etc.) 6: Makefile data structure internal debugging 7: Low-level internal debugging

- D Displays the reasons for executing commands **and**
 executes the commands
- E Enables macro definition priority compatibility with
 amu 5.00 (see *Macro priority* on page 223 for details).

target1 {target2} ...

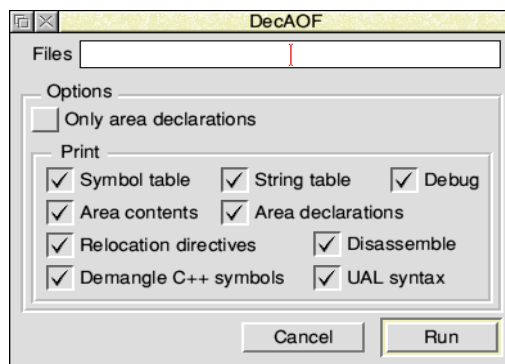
This is a space-separated list of targets to be made or macro pre-definitions of the form `name=string`. Targets are made in the order given. If no targets are given, the first target found in Makefile is used.



DecAOF decodes one or more object files and returns information about each area within the files.

The SetUp dialogue box

Clicking Select on the application icon or dragging the name of a file from a directory display to the icon brings up the SetUp dialogue box:



The **Files** writable icon allows you to specify the name of one or more files to be processed (typed in or dragged from a directory display). These files must be ARM Object Format (AOF) files.

SetUp options

Only area declarations prints a short summary of details about each area in the object file. If this option is selected no other details are printed.

The options offered under the heading of **Print** are all set on by default. Choosing one or more of them will set the remaining options to off.

Symbol table prints the contents of the symbol table.

String table prints the contents of the string table.

Debug prints the debug areas in a readable format.

Area contents prints the area contents in hex.

Area declarations prints the area declarations.

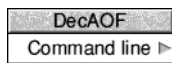
Relocation directives prints linker relocation directives.

Disassemble prints disassembly of code areas. Branches (B and BL) within the same code area are looked up in the symbol table and displayed with the relevant symbol name.

Demangle C++ symbols removes C++ mangling from symbol names.

UAL syntax selects the ARM Unified Assembly Language syntax for disassembly.

The SetUp menu

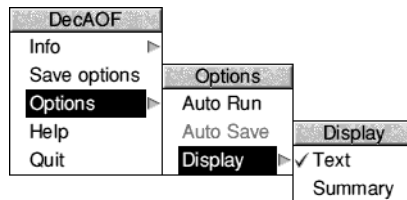


Clicking Menu on the SetUp dialogue box displays the menu shown on the left.

For a description of the DecAOF **Command line** option see the section *Command line interface* on page 129

The Application menu

Clicking Menu on the DecAOF application icon gives the following options:



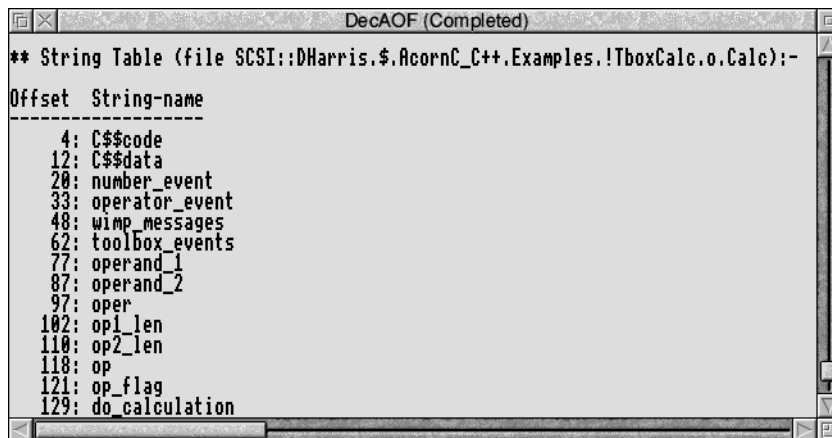
For a description of each option in the application menu see the chapter *General features* on page 103.

Note that **Auto Save** is not available for this application.

Example output

The output of DecAOF appears in one of the standard non-interactive tool output windows. For more details of these see the section *Output* on page 107.

The following window shows an example of the output from DecAOF:



```
DecAOF (Completed)
** String Table (file SCSI::DHarris$.AcornC_++.Examples.!TboxCalc.o.Calc):-
Offset  String-name
-----
  4: C$$code
 12: C$$data
 20: number_event
 33: operator_event
 48: wimp_messages
 62: toolbox_events
 77: operand_1
 87: operand_2
 97: oper
102: op1_len
110: op2_len
118: op
121: op_flag
129: do_calculation
```

Command line interface

For normal use you do not need to understand the syntax of the DecAOF command line, as it is automatically generated for you from the SetUp dialogue box settings. The Command Line syntax for DecAOF is:

```
DecAOF [options] filename [filename...]
```

Options

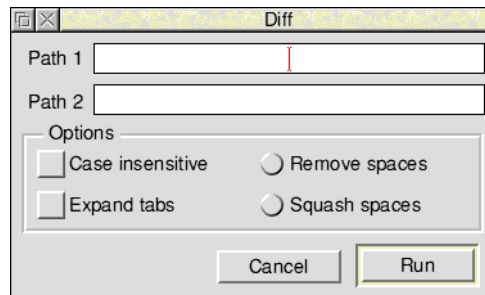
- a print area contents in hex (implies -d)
- b print only the area declarations
- c print disassembly of code area (implies -d)
- c++ demangle C++ symbol names.
- d print area declarations
- g print debug areas
- r print relocation directives (implies -d)
- s print symbol table
- t print string table
- u disassemble to UAL syntax
- filename* a valid pathname specifying an AOF file



Diff displays the textual differences between two files, or between files in two directories, on a line-by-line basis. To compare files more usefully various options allow you to display only those differences of specific interest.

The SetUp dialogue box

Clicking Select on the application icon or dragging the name of a file from a directory display to the icon brings up the SetUp dialogue box:



Path 1 and **Path 2** allow you to specify the names of files or directories to be processed (typed in or dragged from a directory display).

SetUp options

Case insensitive instructs Diff to ignore the case of letters; for example, `Variable` and `variable` would be considered as identical if this was chosen.

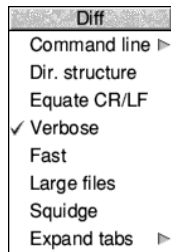
Expand tabs substitutes tabs by multiples of eight spaces.

Remove spaces removes all spaces before comparing lines. This is useful if you wish to examine two files you have been editing but are not interested in any extra spaces you may have introduced.

Squash spaces replaces all instances of two or more spaces by one space.

Note: If you are using Diff to display the differences between two source files where spaces can be critical, e.g. assembler code, and you want to display lines where spaces have been deleted or added, it is essential to ensure that neither **Remove spaces** nor **Squash spaces** have been chosen.

The SetUp menu



Clicking Menu on the SetUp dialogue box displays the menu shown on the left.

Command line enables you to examine or edit the actual command line. For more information on this option see the section *Command line interface* on page 134.

Dir. structure displays only the directory structure of the two files. It does not display any differences between the files.

Equate CR/LF instructs Diff to treat the linefeed and carriage return characters as identical. This is especially helpful when analysing files created by different editors where sometimes linefeeds and sometimes carriage returns are used as end of line terminators.

Verbose tells Diff to print information about all the files being compared, even if they are identical. Turning this option off causes it to only list actual differences.

Fast performs a speedy analysis of two files. It reports only whether there are differences between the two files, not what or where the differences are.

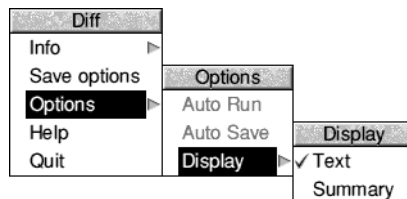
Large files is helpful where very large files are being compared. It sometimes happens that two files differ completely over a large section of text because, for instance, you may have edited in several paragraphs or even several pages of text. Ordinarily Diff would not be able to detect this and would report every line from this point forward as different. However, if **Large files** has been chosen Diff performs a more detailed analysis (thereby taking longer) and can detect this situation. It will then pick up where the two files converge again and display only valid differences from that point onward.

Squidge removes all spaces, except between alphanumerics, where multiple spaces are replaced by one space.

Expand tabs allows you to replace tabs by multiples of any number of spaces you wish.

The Application Menu

Clicking Menu on the Diff application icon gives the following options:



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

Note that **Auto Run** and **Auto Save** are not available for this application.

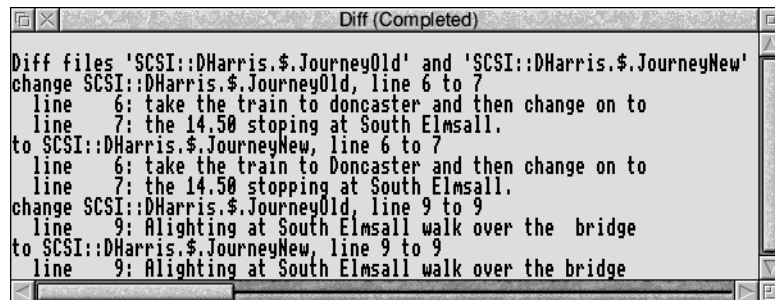
Example output

The output of Diff appears in one of the standard non-interactive tool output windows. For more details of these see the section *Output* on page 107.

The following two examples show the use of options within Diff.

Example 1

In this example two text files have been analysed by Diff with no options being set:



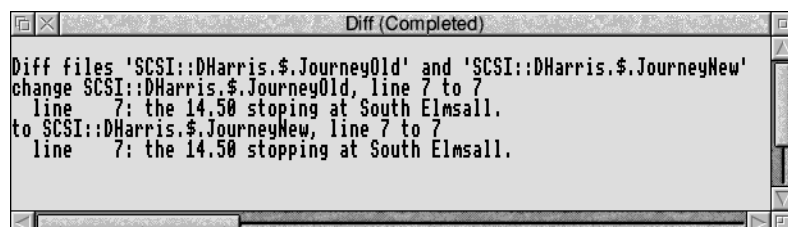
```
Diff files 'SCSI::DHarris$.JourneyOld' and 'SCSI::DHarris$.JourneyNew'
change SCSI::DHarris$.JourneyOld, line 6 to 7
  line 6: take the train to doncaster and then change on to
  line 7: the 14.50 stoping at South Elmsall.
to SCSI::DHarris$.JourneyNew, line 6 to 7
  line 6: take the train to Doncaster and then change on to
  line 7: the 14.50 stopping at South Elmsall.
change SCSI::DHarris$.JourneyOld, line 9 to 9
  line 9: Alighting at South Elmsall walk over the bridge
to SCSI::DHarris$.JourneyNew, line 9 to 9
  line 9: Alighting at South Elmsall walk over the bridge
```

Three differences have been found:

- on line 6 of the first file Doncaster has been spelt with a lowercase d.
- on line 7 of the first file stoping has been spelt with only one p.
- on line 9 of the first file there is an extra space before bridge.

Example 2

In this example the same two files are compared but the **Case insensitive** and **Remove spaces** options have been chosen.



```
Diff files 'SCSI::DHarris$.JourneyOld' and 'SCSI::DHarris$.JourneyNew'
change SCSI::DHarris$.JourneyOld, line 7 to 7
  line 7: the 14.50 stoping at South Elmsall.
to SCSI::DHarris$.JourneyNew, line 7 to 7
  line 7: the 14.50 stopping at South Elmsall.
```

The result is that only the different spelling of the word `stopping` has been displayed.

Command line interface

For normal use you do not need to understand the syntax of the Diff command line, as it is automatically generated for you from the SetUp dialogue box settings. The Command Line syntax for Diff is:

```
Diff [options] filename1 filename2
```

Options

<code>-d</code>	Show only the directory structure, do not display any differences
<code>-e</code>	Equate CR and LF
<code>-f</code>	Perform a fast Diff, all options except <code>-d</code> ignored, do not display any differences
<code>-l</code>	Handle large files more effectively (but more slowly)
<code>-n</code>	Ignore case sensitivity when comparing letters
<code>-r</code>	Remove all spaces before comparing lines
<code>-s</code>	Squash sequences of spaces to one space
<code>-t</code>	As for <code>-r</code> , but <code>-s</code> when between two alphanumeric characters
<code>-v</code>	Be verbose, reporting even identical files.
<code>-x</code>	Expand tabs to spaces (tab stops at multiples of 8)
<code>-Xn</code>	Expand tabs to spaces (tab stops at multiples of <i>n</i>)
<i>filename1</i>	valid pathnames specifying objects to be 'diffed'
<i>filename2</i>	

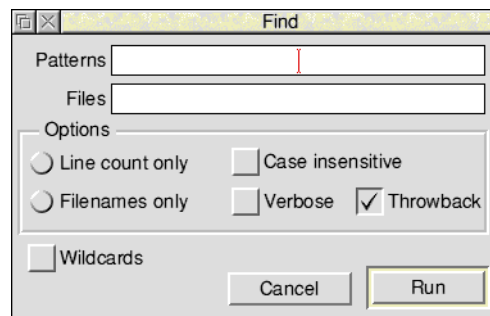


Find searches both the names and the contents of one or more files for text patterns. It includes options allowing you:

- to control whether the case of letters should be considered;
- to use wildcard expressions to specify several filenames;
- to insert wildcard expressions in the pattern string so that digits, control characters, alphanumerics and particular sets of characters can be searched for;
- to start SrcEdit displaying found text using Throwback.

The SetUp dialogue box

Clicking Select on the application icon or dragging the name of a file from a directory display to the icon brings up the SetUp dialogue box:



The **Patterns** writable icon allows you to type in the patterns to be searched for.

If a single pattern includes spaces, the pattern must be enclosed in double quotes, for example:

"the text"

Double quote characters in a search pattern must be preceded by a backslash.

The **Files** writable icon allows you to specify the name of one or more files (typed in or dragged from a directory display) to do the searching in.

SetUp options

Line count only prints only a count of the number of lines matching the pattern from the specified files.

Filenames only lists only the names of files matching the pattern.

Case insensitive will ignore the case of letters; for example, `normal` and `Normal` would be considered as identical if this option was chosen.

Verbose lists the name of each file before searching it for pattern matches.

Throwback enables SrcEdit throwback when text selections are found.

Clicking on **Wildcards** displays a further set of options:

The image shows a 'Find' dialog box with the 'Wildcards' section expanded. The 'Patterns' and 'Files' text boxes are at the top. Below them are radio buttons for 'Line count only' and 'Filenames only', and checkboxes for 'Case insensitive', 'Verbose', and 'Throwback' (which is checked). The 'Wildcards' checkbox is also checked. Below this are 'Cancel' and 'Run' buttons. The expanded 'Wildcards' section contains two sub-sections: 'File wildcards' and 'Pattern wildcards'. 'File wildcards' includes buttons for 'Filename ch. #', '0 or more filename chs. *', 'Sub-directories...', 'Or {', '}' Or', '0 or more (', and ') 0 or more'. 'Pattern wildcards' includes buttons for 'Any .', 'Newline \$', 'AlphaNum @', 'Digit #', 'Ctrl |', 'Normal \', 'Set [', ']' Set', 'Not ~', '0 or more *', and '1 or more ^'.

File wildcards		
Filename ch. #	0 or more filename chs. *	
Sub-directories...	Or {	} Or
0 or more () 0 or more	

Pattern wildcards		
Any .	Newline \$	AlphaNum @
Digit #	Ctrl	Normal \
Set [] Set	Not ~
0 or more *	1 or more ^	

Pattern wildcards

The options listed under **Pattern Wildcards** allow you to specify wildcarded expressions in your search string. Clicking on one of these options will insert a special character into the **Patterns** writable icon immediately before the caret.

Wildcard	Meaning
Any .	Matches any single character. For example: Fr.d will match Fred and Fr1d, but not Fried
Newline \$	Matches the newline character (LineFeed).
Alphanum @	Matches any alphanumeric character a-z, A-z, 0-9 or _.
Digit #	Matches any digit 0-9.
Ctrl l	Matches Ctrl-c, where c is any character between @ and _. For example: x matches Ctrl-x Note: There are two special cases: ? matches the Delete character !c matches Ctrl-'c' where 'c' is the character c with its top bit set
Normal \	Matches the following character even if that character is a special character. For example: \. matches the dot character (not any single character) \c matches lowercase c
Set [Inserts a left square bracket immediately before the caret.
] Set	Inserts a right square bracket immediately before the caret. The preceding two options insert opening and closing square brackets into the Patterns writable icon. You can then manually insert one or more characters between these brackets and Find will match any one of the characters you put inside the brackets. For example: t[aei]n matches tan, ten and tin, but not ton Note that a set is always case-sensitive.
Not ~	Matches any character other than the following character, where the following character is any of the simple character patterns listed above. For example: la~ne matches late, lace and lake, but not lane

Wildcard	Meaning
0 or more *	Matches 0 or more occurrences of the following character, where the following character is any of the simple character patterns listed above. For example: ca*n matches can, cannot and cat
1 or more ^	Matches 1 or more occurrences of the following character, where the following character is any of the simple character patterns listed above. For example: ca^n matches can and cannot, but not cat

File wildcards

The options offered under **File Wildcards** insert special characters into the **Files** writable icon which allow you to specify files in a variety of ways. Several of these options require you to manually insert additional text next to or inside these special characters:

Filename ch. # inserts a hash character immediately before the caret. This character will match any single filename character except ‘.’

For example:

Find adfs::HDisc4.\$.Fred# will search files Fred1 and Freda, but not Fred13, Frederick etc.

Find adfs::HDisc4.\$.Fr#d will search files Fred and Fr2d, but not Fred1d, Freed etc.

0 or more filename chs. * inserts an asterisk immediately before the caret. This character will match any sequence of filename characters except ., {, and }.

For example:

Find adfs::HDisc4.\$.Fred* will search files Fred1 and Freda, and also Fred13, Frederick etc.

Find adfs::HDisc4.\$.Fr*d will search files Fred and Fr2d, and also Frd, Freed, Fr123d etc.

Sub-directories ... inserts three dots immediately before the caret. It must be positioned immediately after a directory name. Find will then search all nominated files in that directory and in any subdirectories in that structure.

For example:

Find adfs::Amy\$.Receipts...monthly

will search all files called monthly in the directory Receipts and also in any subdirectories of Receipts.

Or { inserts a left brace immediately before the caret.

Or } inserts a right brace immediately before the caret.

The preceding two options insert opening and closing braces into the **Files** writable icon. You can then manually insert one or more filename characters between these braces, separating each filename with a comma. Find will then search all filenames inside the braces.

For example:

```
Find adsf::HDisc4.$W.rel.{atype,btype,ctype}
```

would search all three files inside the braces, i.e. *atype*, *btype* and *ctype*.

0 or More (inserts a left bracket immediately before the caret.

) 0 or More inserts a right bracket immediately before the caret.

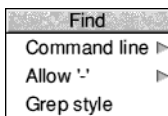
The preceding two options insert opening and closing brackets into the **Files** writable icon. You can then manually insert one or more filename characters between these brackets and Find will search any files with none, one or more occurrences of the characters you put inside the brackets.

For example:

```
Find adsf::HDisc4.$Fr(e)d    will search files Frd, Fred and Freed,
                             but not Frid.
```

```
Find adsf::HDisc4.$Fr(ie)d   will search files Frd, Fried and
                             Frieied, but not Frid, Frieed or
                             Fred.
```

The SetUp menu



Clicking Menu on the SetUp dialogue box displays the menu shown on the left.

Command line option – see *Command line interface* on page 141.

The **Allow ‘-’** option enables you to specify another pattern which will be matched even if it begins with a -. This pattern will be searched for in conjunction with the patterns you have inserted into the **Patterns** writable icon.

If you need to match two or more patterns beginning with a -, then you must precede each additional pattern with -e

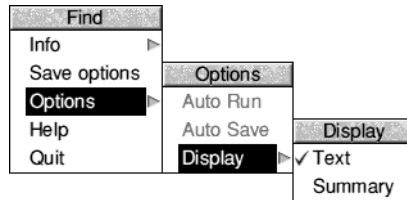
For example:

```
-pattern -e -pattern -e -pattern
```

Grep style enables you to specify patterns using the syntax of the UNIX *grep* tool. This option is provided for users familiar with UNIX.

The Application menu

Clicking Menu on the Find application icon gives the following options:



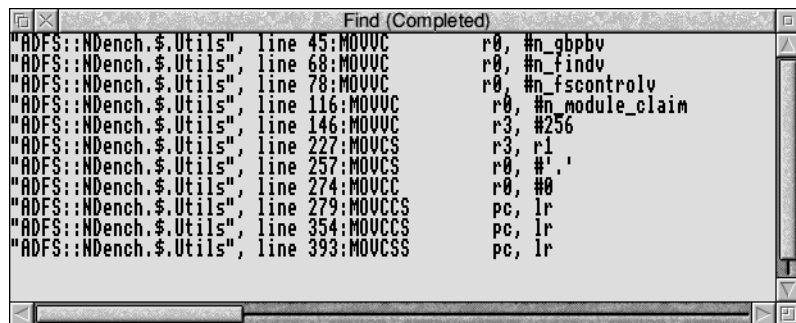
For a description of each option in the application menu see the chapter *General features* on page 103.

Note that **Auto Run** and **Auto Save** are not available for this application.

Example output

The output of Find appears in one of the standard non-interactive tool output windows. For more details of these see the section *Output* on page 107.

The following window shows an example of the output from Find:



In the above example the pattern MOV[CV] was specified in the **Patterns** writable icon in order to list only those instructions beginning with MOVV or MOVV in an assembler source file. Instructions where the fourth letter was not a C or V, such as MOVN, MOVNE and MOVEQS, were, therefore, not listed. The **Throwback** option was not enabled in the above example. With **Throwback** enabled, a SrcEdit Throwback browser would also have appeared allowing the file `Utils` to be edited, starting at the found lines.

Command line interface

For normal use you do not need to understand the syntax of the Find command line, as it is automatically generated for you from the SetUp dialogue box settings. The Command Line syntax for Find is:

```
Find [options] [pattern{ pattern}] -f filepattern{ filepattern}
```

Options

- c list only a count of the number of lines matching from each file.
- n ignore the case of letters when making comparisons.
- l list only the names of files matching patterns.
- v list the name of each file before searching it for matches.
- u accept UNIX grep/egrep-style patterns.
- e allow the following pattern arguments to begin with a -.

Pattern

- .
- \$ matches any single character.
- @ matches the newline character (LineFeed).
- @ matches any alphanumeric character.
- # matches any digit.
- | | c matches Ctrl-c, where c is any character between @ and _.
- \ matches the following character even if that character is a special character.
- [] matches any character inside the square brackets.
- ~ matches any character other than the following character.
- * matches 0 or more occurrences of the following character.
- ^ matches 1 or more occurrences of the following character.
- f marks the end of multiple patterns and the start of filepatterns.

Filepattern

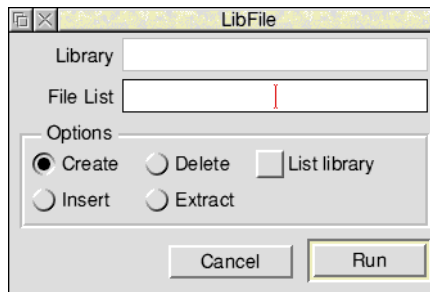
- # matches any filename character except .
- * matches 0 or more filename characters other than .
- . . . searches files in that directory and any subdirectories in that directory.
- { , } searches files contained within braces (filenames separated by commas).
- () search any file with none, one or more occurrences of the characters inside the brackets.



LibFile creates and maintains library archives. It can be used to create archives of files for backup and distribution purposes, for example. A special form of library archive containing AOF files can be created for use with Link. The format of library archive files is described in appendix *Code file formats* on page 247.

The SetUp dialogue box

Click Select on the application icon. This displays the SetUp dialogue box:



The SetUp options

Library is the name of the library to be processed. If a library is being created this will be shaded. A Save as dialogue box will be presented when the library is created.

File List, when used with **Create** or **Insert**, contains the list of files to be placed in the library. When used with **Delete** or **Extract** it contains a list of files in the library which are to be extracted or deleted. You can use wildcard characters in the **File List** (* to match zero or more characters, and # to match a single character).

Create creates a new library containing the files in **File List**. This is the default option.

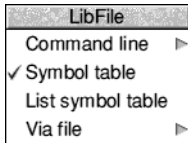
Delete removes the files in **File List** from the specified library.

Insert adds the files in **File List** to the specified library. Files of the same name in the library will be replaced.

Extract copies the files in **File List** from the specified library to disc. The files are not deleted from the library.

List library lists the files contained in the specified library. By default, this option is off.

The SetUp menu



Click Menu on the SetUp dialogue box. This displays the LibFile SetUp menu.

Command line allows you to specify the command line to be presented to the underlying LibFile command line tool. You should take care when modifying the command line. The effect of certain arguments depends on the order in which they appear in the command line. Changing this order may have unanticipated effects. Refer to the section *Command line interface* on page 147.

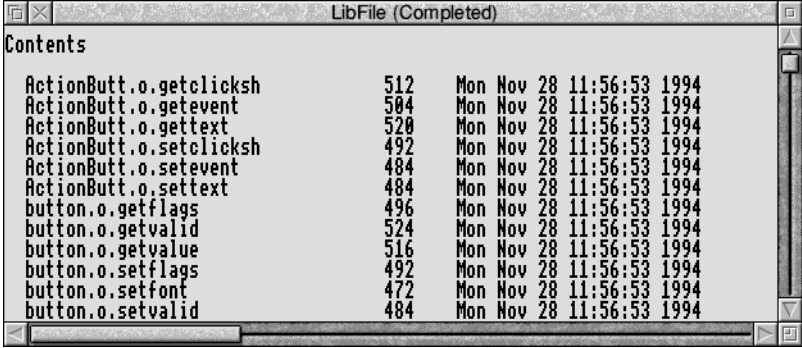
Symbol table adds an external symbol table, as used by Link, to the library. External symbols in any object files in the library are placed in the symbol table. Non object files are ignored. By default, this option is on.

List symbol table lists the symbols in the external symbol table along with the name of the AOF file which generated each symbol. This option is off by default.

Via file allows you to set up a list of files to be used in one file called a Via file. When creating or maintaining libraries with a large number of files it may become tedious having to drag all the files to the **File List** every time, especially if they are in different directories. Enter the name of the Via file in the submenu and press Return.

Output

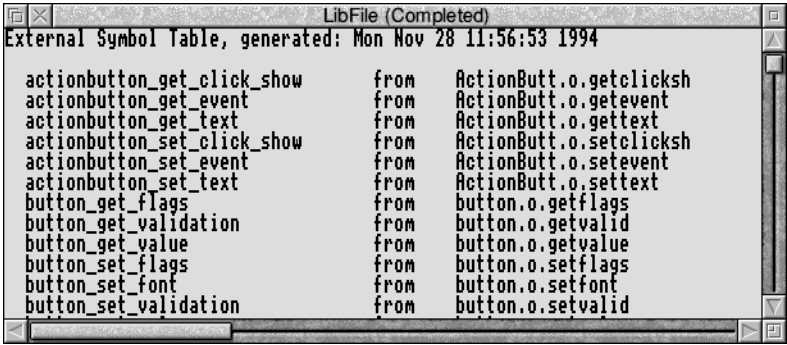
The Output window displays the list of files in the library and/or the list of external symbols when the **List library** or **List symbol table** options are selected. The following windows show examples of each.



LibFile (Completed)

Contents

ActionButt.o.getclicksh	512	Mon Nov 28 11:56:53 1994
ActionButt.o.getevent	504	Mon Nov 28 11:56:53 1994
ActionButt.o.gettext	520	Mon Nov 28 11:56:53 1994
ActionButt.o.setclicksh	492	Mon Nov 28 11:56:53 1994
ActionButt.o.setevent	484	Mon Nov 28 11:56:53 1994
ActionButt.o.settext	484	Mon Nov 28 11:56:53 1994
button.o.getflags	496	Mon Nov 28 11:56:53 1994
button.o.getvalid	524	Mon Nov 28 11:56:53 1994
button.o.getvalue	516	Mon Nov 28 11:56:53 1994
button.o.setflags	492	Mon Nov 28 11:56:53 1994
button.o.setfont	472	Mon Nov 28 11:56:53 1994
button.o.setvalid	484	Mon Nov 28 11:56:53 1994



LibFile (Completed)

External Symbol Table, generated: Mon Nov 28 11:56:53 1994

actionbutton_get_click_show	from	ActionButt.o.getclicksh
actionbutton_get_event	from	ActionButt.o.getevent
actionbutton_get_text	from	ActionButt.o.gettext
actionbutton_set_click_show	from	ActionButt.o.setclicksh
actionbutton_set_event	from	ActionButt.o.setevent
actionbutton_set_text	from	ActionButt.o.settext
button_get_flags	from	button.o.getflags
button_get_validation	from	button.o.getvalid
button_get_value	from	button.o.getvalue
button_set_flags	from	button.o.setflags
button_set_font	from	button.o.setfont
button_set_validation	from	button.o.setvalid

Notes:

- 1 Any directories in the **File List** to be archived will be recursively archived (i.e. all files in the specified directory will be archived and any directories in the specified directory will themselves be recursively archived). This can be useful if, for example, you are backing up an entire source tree on which you are currently working.
- 2 When extracting files, LibFile places absolute filenames from the libraries index in their corresponding absolute filenames on disc. Relative filenames (i.e. those not containing a colon (:) a dollar (\$) or an at sign (@)) are placed in a temporary directory and, when the extraction is finished, a Save as dialogue box is presented. This allows you to drag the extracted files to a suitable place on your disc. The temporary directory is then renamed to the correct place on your disc, or copied and subsequently deleted if you drag to a different device or filing system.

- 3 When creating libraries for distribution purposes, you should not use absolute filenames in the **File List**. If, for example, you created a library with a **File List** of `adfs::Edward.$.PDUtils`, it would not be very useful to someone called Ian or to someone using a network filing system. Instead, set your current directory (from the command line with the `*Dir` command) to `adfs::Edward.$` and use the **File List** `PDUtils`.
- 4 When creating libraries for backup purposes, you can use absolute filenames, since you will always be restoring to your own disc. You should not, however, mix absolute and relative filenames in the same library. LibFile will handle this as described in the note on extracting files above, but the behaviour may be confusing to anyone trying to extract files.
- 5 When creating a library, LibFile builds the library in memory. This means that you cannot create a library bigger than the available memory on your machine. When altering an existing library (using **Insert** or **Delete**) Libfile requires memory space for the new and old libraries. If there is not enough memory for this you can get around the problem by extracting all the files and recreating the library including the files to be inserted, or omitting the files to be deleted.
- 6 When the **Symbol table** option is selected, LibFile always updates the external symbol table regardless of the operation being performed. This is correct for **Create**, **Insert** and **Delete**. For **Extract** this is usually not very useful, so you should generally ensure the **Symbol table** option is deselected when using **Extract**.
- 7 If the **Symbol table** option is not selected, LibFile deletes the external symbol table when used with **Insert** or **Delete**. This prevents a potential problem whereby the external symbol table could become out of date with respect to the object modules in the library.
- 8 Convergence testing is a testing method whereby a binary file (such as an object library) is rebuilt using itself, and the new and old binaries are compared to ensure that they are the same. This can be difficult with tools (such as LibFile) which timestamp files placed in the library, because the new and old libraries will be built at different times, and will always differ. LibFile provides the **Null timestamps** option to circumvent this problem. The **Null timestamps** option uses timestamps of all bits 0, which corresponds to a date of `00:00:00 01-Jan-1900`. Thus, libraries built at different times can be compared using a binary comparison utility, without the timestamps causing extraneous differences to appear.
- 9 Wildcard matching, when applied to library members (when using **Extract** or **Delete**) applies the wildcard across the complete filename. When applied to files (**Create** or **Insert**) wildcards apply to single components of the filename. Thus, the wildcard specification `a#c` would match `a.b` and `abc` when using **Extract** or **Delete**, but would only match `abc` when using **Create** or **Insert**.

Command line interface

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

The format of the LibFile command is:

```
Libfile options library [file_list]
```

Wildcards * and # may be used in *file_list*.

Options

-h	Display a screen of help text
-c	Create a new library containing files in <i>file_list</i>
-i	Insert files in <i>file_list</i> , replace existing members
-d	Delete the members in <i>file_list</i>
-e	Extract members in <i>file_list</i> placing in files of the same name
-o	Add an external symbol table to an object library
-l	List library , may be specified with any other option
-s	List symbol table , may be specified with any other option
-t	Use Null timestamps when creating or updating library
-v <i>file</i>	Take additional arguments from <i>file</i>
-q <i>dir</i>	Place relative filenames in <i>dir</i> when extracting file

Notes:

- 1 Multiple options may be specified in a single options argument. For example, `-c lso` is equivalent to `-c -l -s -o`.
- 2 Most of the above options should be familiar from the description of the desktop interface. One possible exception to this is the `-q` option. This option means 'behave as though the directory specified after the `-q` option were the current working directory (as set by the `dir` command)'.
When extracting files with relative pathnames, LibFile creates this directory if it does not already exist and prefixes the relative pathnames with the specified directory. Note, that you should not add a full stop (.) to the end of the directory specification, LibFile adds this itself.
- 3 The `-q` option is used by the desktop interface (since the desktop has no notion of a current working directory) to tell LibFile where to put files with relative pathname (generally `<Wimp$ScrapDir>Tmp_name` where *Tmp_name* is a name invented by the desktop interface). This directory is then renamed, or copied to a user-specified directory.

- 4 For compatibility with previous versions of LibFile, specifying `-c` with `-o` with a null file list does not create an empty library. Instead, it ignores the `-c` option and adds a symbol table to an existing library.

Examples

```
LibFile -c srcLib *
```

Create a library called `srcLib` in the current directory from all the files in the current directory (including the files contained in any directories in the current directory).

```
LibFile -co adfs::Edward.$.clib.o.AnsiLib o
```

Create the object library `Ansilib` from the object files contained in directory `o` in the current directory.

```
Libfile -e -q :Ian.$.PDUtils :0.PDLib *
```

Extract all the files from `:0.PDLib` and put them in the directory `:Ian.$.PDUtils`.

Assembler example

The programming example `PrintLib`, which you can find in `Sources.DDE-Examples.ObjAsm.PrintLib`, consists of three potentially useful procedures written in assembler which are intended to be assembled to object files using `ObjAsm` and then formed into a library with `LibFile`. They illustrate various programming points as well as how to construct a library.

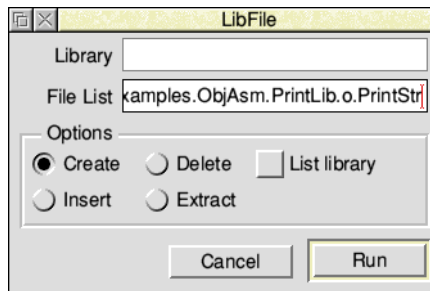
If you examine the assembler source files in `PrintLib.s` you will see that the procedure exported by each file obeys the ARM Procedure Call Standard. This ensures that they, and hence the `PrintLib` library, can be linked with other languages such as C. It is essential that procedures placed in a library have consistent register conventions, so that they can be re-used later without consulting their source text.

The `PrintLib` example is provided with both its assembly language source and the finished library. The facilities provided by this library are used in other programming examples. The procedures it exports are:

<code>print_string</code>	Print a null terminated string pointed to by <code>r0</code> .
<code>print_hex</code>	Print in hexadecimal an integer contained in <code>r0</code> .
<code>print_double</code>	Print in scientific format a double precision floating point number contained in <code>r0,r1</code> .

To reconstruct PrintLib from its sources, first double click on !ObjAsm and !LibFile in a directory display to load them as applications with icons on the icon bar. Then assemble `s.PrintStr`, `s.PrintHex` and `s.PrintDble` to corresponding object files by dragging each source file to the ObjAsm icon and saving the output object files in the default places, i.e. `o.PrintStr`, `o.PrintHex` and `o.PrintDble`.

Next drag `o.PrintStr` to the LibFile icon to make the LibFile SetUp dialogue box appear:

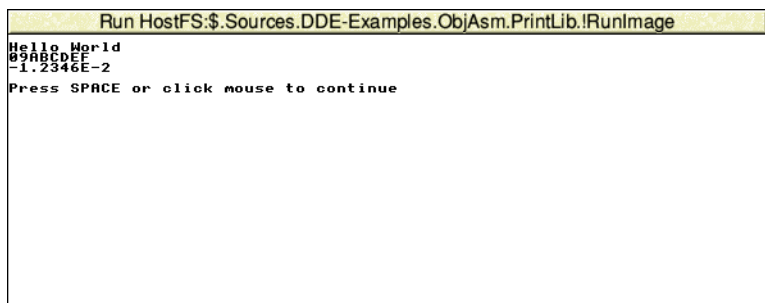


Ensure that the **Create** option is chosen as above. Drag the other two object files to **File List**, then click on **Run**. Finally save the library file produced: it is now ready to use.

The assembly language source file `PrintLib.s.ATestPrLib` is an example program making use of the procedures exported by PrintLib. To use it:

- 1 Double click on the !Link application to load it.
- 2 Assemble `s.ATestPrLib` to `o.ATestPrLib` with ObjAsm.
- 3 Link `o.ATestPrLib` with the finished PrintLib library to produce an executable AIF image file.

Running the test program by double clicking on it should result in text output into a RISC OS output window:





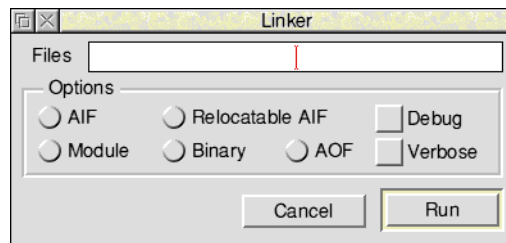


The purpose of Link is to combine the contents of one or more object files (the output of a compiler or Assembler) with selected parts of one or more library files to produce an executable program.

Load the Link application by double-clicking on the !Link icon.

The SetUp dialogue box

Click Select on the application icon. This displays the SetUp dialogue box:



This allows you to set the following options:

The **Files** writable icon allows you to enter the list of object and library files to be linked. You can do this in two ways:

- Type in a space-separated list of the files to be linked. You can use wildcards (* to match zero or more characters, and # to match a single character).
- Drag the icons of the files to be linked onto the **Files** writable icon. Dragging a directory to the icon (e.g. an o directory) links all the files in that directory.

Note: When linking libraries, you must take care to link them in the correct order. See the section *Libraries* on page 155.

AIF generates ARM Image Format (AIF) output. This is the default image used for building an application. You should only choose other image types if AIF is not suitable for some reason. The format of AIF files is described in *Appendix E*.

Module generates Relocatable Module Format (RMF) output. Refer to *Relocatable modules* on page 160 for more details.

Relocatable AIF links an image so that it can be run at any address, usually specified in conjunction with the **Workspace** option on the SetUp menu. See the section *Relocatable AIF images* on page 159 for more details.

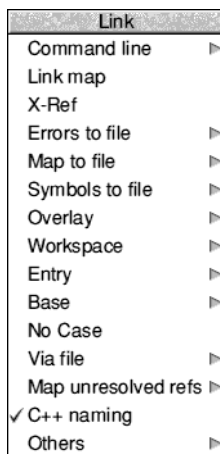
Binary generates a plain binary image (without an image header or any specific image format). The default load address for a binary image is 0. Any other address can be specified using the Base option from Link's SetUp menu. If AIF is also enabled in Link's SetUp dialogue box, then a plain binary image is generated, preceded by an AIF header which describes it.

AOF generates partially linked output in ARM Object Format, suitable for inclusion in a subsequent link step.

Debug allows you to debug a program with the desktop debugger DDT. See the chapter *Desktop debugging tool* on page 19 for more details on preparing a program for use with the debugger. This option is not suitable for use with the module option. This option is switched off by default.

Verbose gives progress reports in the Output window while linking. See the section *Output* on page 154 for an example of this output. This option is switched off by default.

The SetUp menu



Clicking Menu on the SetUp dialogue box displays the menu shown on the left.

Command line allows you to specify the command line to be presented to the underlying Link command line tool. Refer to the section *Command line interface* on page 162 for more details.

Link map displays the base address and size of every code, data and debugging information area, and displays total sizes for the code, data and debugging information in the output window. See the section *Link map option* on page 158 for more information. For details on linker areas, see the section *AOF* on page 249.

X-Ref displays a list of inter-area references. This option is most useful when trying to reduce dependencies between library elements, so that you only need include the minimum set of library elements. It is also useful when using overlays. See the section *X-Ref option* on page 158 for more details.

Errors to file allows you to specify the name of a file to which all errors should be written.

Map to file will write a link map to the given filename (if the **Link map** option is enabled).

Symbols to file will write all symbols found to a file with the given name.

Overlay generates an overlaid image using the specified overlay description file. For details of overlay description files, see the section *Overlay description files* on page 157. This option is not suitable for use when generating Module or Binary output.

Workspace, when used in conjunction with the **Relocatable AIF** option, generates an auto-relocatable image which will relocate itself to the top of its application space. This leaves the specified amount of workspace above the image free for the use of the program being linked. The effect of this option is not currently defined when generating image types other than relocatable AIF.

Entry specifies the entry point of an image if none of the object files themselves specify an entry point. Generally, you should only use it when writing completely in assembler without using the assembler's ENTRY directive.

Base specifies the base address at which the image should be linked. By default this is 0x8000 for AIF images and 0 for binary images. You should always load non-relocatable AIF images at their base address.

No case causes a case insensitive comparison to be used when comparing symbols. You will not generally want to use this option with C (which is case sensitive). However, you may need to use it with other language systems (such as Pascal and Fortran) which are case insensitive, especially if you are trying to interwork with C and one of these languages.

Via file allows you to set up a list of object files to be linked in one file called a **Via file**. Instead of having to drag all the files to the **Files** list on the SetUp dialogue box, just enter the name of the Via file in the submenu.

Map unresolved refs causes all unresolved references to be resolved to a given symbol.

C++ naming will report C++ symbol names using C++ notation. Note that you **must** enable this option when linking C++ compiled code.

Others allows you to specify other options allowed by the underlying command line link tool.

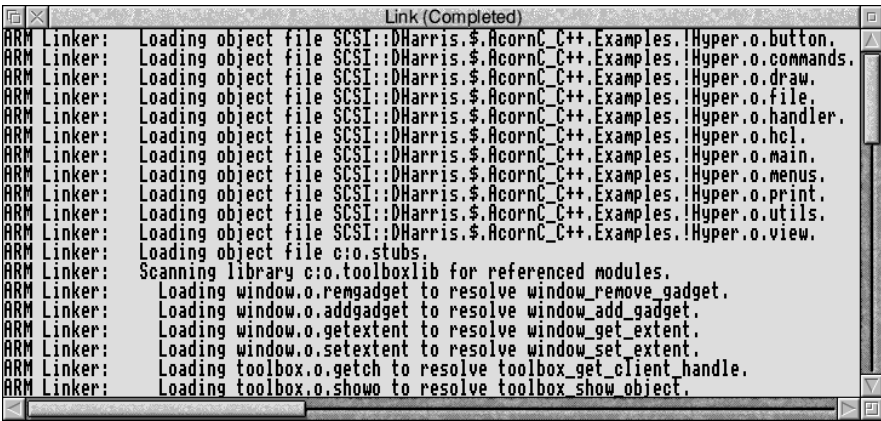
Note: The **Base**, **Workspace** and **Entry** options require a numeric argument to be entered in the associated submenu. You can prefix this argument by **&** or **0x** to specify a hexadecimal value. You can postfix it by **k** for 2^{10} and **m** for 2^{20} .

Output

The Output window displays information printed when you have selected the **Verbose**, **Link map** or **X-Ref** options. It also displays any error messages generated while linking.

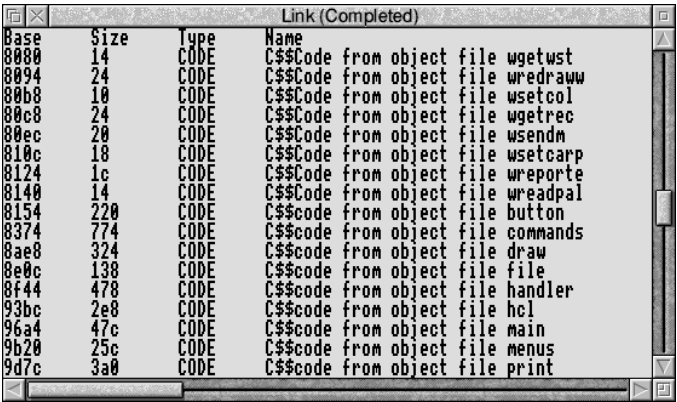
The following windows show examples of the **Verbose** and **Link map** output. You will find an example of the **X-Ref** output in the section *X-Ref option* on page 158.

Verbose output:



```
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.button.  
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.commands.  
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.draw.  
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.file.  
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.handler.  
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.hcl.  
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.main.  
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.menus.  
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.print.  
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.utils.  
ARM Linker: Loading object file SCSI::DHarris.$AcornC_++.Examples.Hyper.o.view.  
ARM Linker: Loading object file c:o.stubs.  
ARM Linker: Scanning library c:o.toolboxlib for referenced modules.  
ARM Linker: Loading window.o.remvgadget to resolve window_remove_gadget.  
ARM Linker: Loading window.o.addgadget to resolve window_add_gadget.  
ARM Linker: Loading window.o.gettextext to resolve window_get_extnt.  
ARM Linker: Loading window.o.settextext to resolve window_set_extnt.  
ARM Linker: Loading toolbox.o.getch to resolve toolbox_get_client_handle.  
ARM Linker: Loading toolbox.o.showo to resolve toolbox_show object.
```

Link map output:



Base	Size	Type	Name
8080	14	CODE	C\$\$Code from object file wgetust
8094	24	CODE	C\$\$Code from object file wredraww
80b8	10	CODE	C\$\$Code from object file wsetcol
80c8	24	CODE	C\$\$Code from object file wgetrec
80ec	20	CODE	C\$\$Code from object file wsendm
810c	18	CODE	C\$\$Code from object file wsetcarp
8124	1c	CODE	C\$\$Code from object file wreporte
8140	14	CODE	C\$\$Code from object file wreadpal
8154	220	CODE	C\$\$code from object file button
8374	774	CODE	C\$\$code from object file commands
8ae8	324	CODE	C\$\$code from object file draw
8e0c	138	CODE	C\$\$code from object file file
8f44	478	CODE	C\$\$code from object file handler
93bc	2e8	CODE	C\$\$code from object file hcl
96a4	47c	CODE	C\$\$code from object file main
9b20	25c	CODE	C\$\$code from object file menus
9d7c	3a0	CODE	C\$\$code from object file print

Possible errors during a link stage

Two common errors which can occur during a link stage are caused by unresolved and multiple references.

In the case of unresolved references, a symbol has been referenced from an object file, but there is no corresponding definition for the symbol. Link will generate an error message giving the name of the undefined symbol. This is usually caused by the omission of a required object or library file from the file list, or the misspelling of an external identifier in the original source program.

Multiple references are caused by a clash of names. For example, a procedure might have been defined with the same name as a library procedure, or as a procedure in another object file.

Libraries

Libraries differ from object files in the way Link uses them. First, all the object files are linked together. Then, for each library in turn, Link searches for symbol definitions which match unsatisfied symbol references. When such a symbol definition is found, the module defining that symbol is loaded.

When a library module is loaded, new unsatisfied symbol references may be created, so the library is re-searched until no more members are loaded from it. Note that each library is processed in turn, so references between libraries must be ordered.

A reference from a member of a library later in the file list to a member earlier in the file list will not be resolved. Therefore you must drag libraries to the file list in the correct order.

Usually, at least one library file will be specified in the list of files to be linked. This will typically be the run-time library for the language you are using. When writing in C, you can use either the shared library (in which case you will need to link with the shared library stubs, `C:o.stubs`) or the unshared library, `C:o.ansilib`. Use the unshared library when linking a program for use with the desktop debugger, or when linking a program which you intend to distribute to people who may not have the shared C library.

You can call the procedures in the library for one language from programs written in another, provided:

- both libraries conform to the ARM Procedure Call Standard (APCS) described in appendix ARM *procedure call standard* on page 297
- the library's initialisation routines have been called.

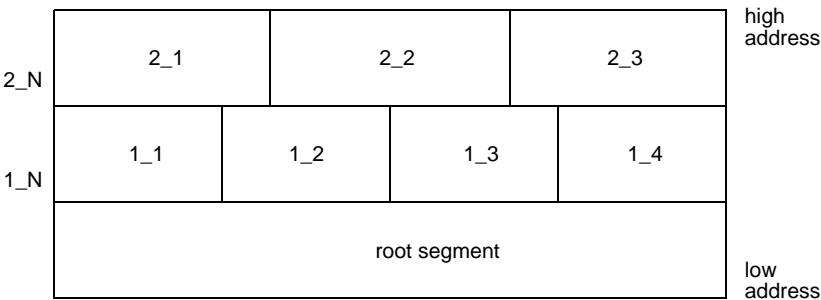
Refer to the chapter *The Shared C library* in the *RISC OS Programmer's Reference Manual* for details on how to initialise the common run-time kernel distributed with the C library.

Generating overlaid programs

An introduction to overlays is given in the *Acorn C/C++ manual*. If you are not familiar with the concept of overlays, you should read the chapter on overlays in that manual first. This section only describes how to use Link to create an overlaid application.

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

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



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

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

Link assigns AOF AREAs to overlay segments under user control. Usually, a compiler produces one code AREA and one data AREA for each source file (called C\$\$code and C\$\$data when generated by the C compiler). The C compiler option -zo (described in the *Acorn C/C++ manual*) allows each separate function to be compiled into a separate code AREA. This gives finer control of the

assignment of functions to overlay segments (but at the cost of slightly enlarged code and enlarged object files). You control the overlay structure by describing the assignment of certain AREAs to overlay segments.

For all remaining code AREAs, Link will act as follows:

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

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

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

On disc, an overlaid program is organised as a RISC OS application. The components of the application (the !RunImage and the various overlay segments) must reside in the application directory. Link creates the following components in the application directory:

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

Overlay description files

The overlay description file, specified in the overlay submenu, describes the required overlay structure. It is a sequence of logical lines:

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

Each logical line has the following structure:

```
segment_name module_name [(list_of_AREA_names)] module_name ...
```

For example:

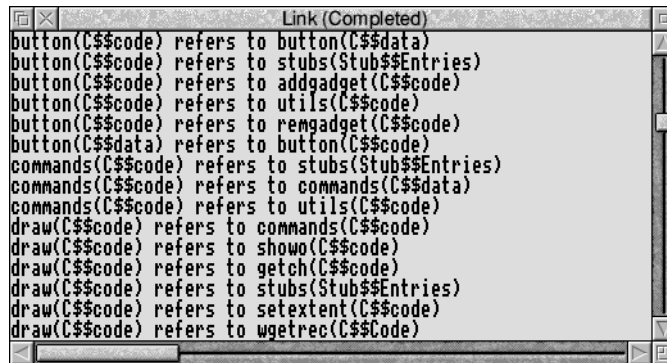
```
1_1 edit1 edit2 editdata(C$$code,C$$data) sort
```

The *list_of_AREA_names* is a comma-separated list of names as they appear when displayed by the DecAOF tool. If omitted, all code AREAs are included.

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

X-Ref option

To help the user-partition between overlay segments, Link can generate a list of inter-AREA references. To do this, choose the **X-Ref** option on the SetUp menu. The following window shows an example of the output from **X-Ref**:



```

button(C$$code) refers to button(C$$data)
button(C$$code) refers to stubs(Stub$$Entries)
button(C$$code) refers to addgadget(C$$code)
button(C$$code) refers to utils(C$$code)
button(C$$code) refers to remgadget(C$$code)
button(C$$data) refers to button(C$$code)
commands(C$$code) refers to stubs(Stub$$Entries)
commands(C$$code) refers to commands(C$$data)
commands(C$$code) refers to utils(C$$code)
draw(C$$code) refers to commands(C$$code)
draw(C$$code) refers to showo(C$$code)
draw(C$$code) refers to getch(C$$code)
draw(C$$code) refers to stubs(Stub$$Entries)
draw(C$$code) refers to setextent(C$$code)
draw(C$$code) refers to wgetrec(C$$code)
  
```

In general, if area A references area B (for example because x in area A calls y in area B) then A and B should not share the same area of memory. Otherwise, every time x calls y or y returns to x, there will be an overlay swap.

Link map option

The **Link map** option displays the base address and size of every area in the output program. It is useful for determining how AREAs might be packed most efficiently into overlay segments.

Linking with the overlay manager

The overlay manager is responsible for loading overlay segments when:

- an inter-segment reference occurs to a segment which is not loaded, or
- a procedure return occurs to a segment which is no longer loaded.

In general, referencing a datum cannot cause an overlay segment to be loaded. One exception to this is an indirect procedure call via a function pointer which will cause an overlay segment to be loaded (Link cannot distinguish this from a normal procedure call, since Link just sees a word relocation to an overlaid procedure). Note that the pointer itself must not be overlaid.

If Link detects a data reference to a non co-resident or potentially non co-resident segment it will issue one of the following messages:

Non co-resident data reference in *module_name(area_name)*

Possible non co-resident data reference in
module_name(area_name)

Certain types of data reference cannot be detected by Link. This happens when read-only data is placed in a code segment. The C compiler places string literals in code areas. This will cause problems if you have external string literals, since Link cannot distinguish between a string literal and a procedure in the code segment. Hence it indirections the string through the Procedure Call Indirection Table (PCIT). So, when your program reads the contents of the string, it will in fact end up reading the contents of the PCIT.

The C compiler option `-fw` (described in the *Acorn C/C++ manual*) causes the compiler to place string literals in data areas. You should use this option on modules which may contain external string literals.

The overlay manager must be included in the link stage. You will find the overlay manager in the object file `C:o.overmgr`. You should drag this object file to the **Files** icon when linking an overlaid program.

Note: The overlay manager is also contained in the non-shared library `ANSILib`, so, if you are using `ANSILib`, you do not need to drag the overlay manager to the **Files** icon. The shared C library does not contain a copy of the overlay manager.

Relocatable AIF images

Usually, when an image file is produced, it will execute correctly only at the specified base address (or the default of `0x8000` if a base is not specified). This is because the program will contain references to absolute addresses within itself. However if you tell Link to generate a relocatable AIF image, you can load and execute the program at any address. Link also inserts a branch in the image header, so that the relocation code is automatically called when you run the program.

This is achieved by adding the following to the end of the image:

- a relocation table
- a small routine to perform the relocation.

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

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

If a **Workspace** value is specified on the SetUp menu, Link inserts the value in the image header. The relocation code examines this value and, if the value is non-zero, relocates the application to the top of application space, leaving the specified amount of workspace between the end of the application and the top of application space for stack and heap usage.

Relocatable modules

When linking a relocatable module, Link performs a similar task as when linking a relocatable AIF image, adding a relocation table and a relocation routine to the end of the module image.

However, the mechanism by which the relocation routine is called is different in a relocatable module: A module must be multiply relocatable, since it may move about in the Relocatable Module Area (RMA) when, for example, the RMA is tidied with the *RMTidy command. The module must call the relocation routine in its initialisation (or re-initialisation) code.

When using the C Module Header Generator (CMHG) tool you need not worry about this, since CMHG automatically generates a module header which includes a call to the relocation routine in its initialisation code.

If you are constructing the module header in assembler, you must make this call yourself. Use the IMPORT directive to import the external symbol `__RelocCode` and place a BL to this symbol in your initialisation code.

```
        IMPORT |__RelocCode|
init
...
        BL      |__RelocCode|
...
```

`__RelocCode` only goes in the executable if it is a module and it explicitly imports it (so assembler modules can be linked with `link -rmf` and not get this function appended unless they actually need it).

Note: any code executed before the call to the relocation routine must be position-independent.

When creating a module header in assembler, the AREA containing the header should have the attributes CODE and READONLY. The AREA name should be chosen so that the AREA will be the first AREA in the module. Link sorts AREAs

first by attribute, then by AREA name, so you should choose an AREA name which is lexicographically less than all other AREA names in your module. The CMHG tool uses an AREA name of `!!!Module$$Header`, but this is not obligatory.

Unlike earlier versions, the linker now supports initialised static data in modules. Consequently, modules written in C++ will now have their global constructors called, but note that it is necessary to call `_____main()` if the module does not have a `main()` function.

Transient utilities

Utility or transient programs (filetype FFC) can be linked with `link -util`. The binary output features a header, with similarities to an AIF header, as required by some versions of RISC OS. It also inserts relocation code if the symbol `__RelocCode` is referenced (required to support any C source files that uses code or static data pointers), in the same way as described above for relocatable modules. After the relocation code, a small footer used to deactivate 26-bit emulation (if Aemulor is installed) is inserted, provided all object files were compiled/assembled using APCS32. The read-only and read-write fields in the header are initialised, with the header itself assigned to the former and the relocation code and footer (if present) assigned to the latter.

The `-nozeroinit` flag is implicit if `-util` is used, since the RMA block into which a utility is loaded will not be large enough for runtime initialisation of a zero-init area, and there is no space in the header to specify this size anyway.

Notes: The C library cannot be used when linking a utility. Utility programs must not be squeezed. For more details on utilities, refer to the *RISC OS Programmer's Reference Manual*.

Predefined linker symbols

All symbols containing the substring `$$` are reserved by Acorn for use by Link.

For each AREA in the output file formed by coalescing one or more areas of the same name (e.g. `C$$code`) Link generates two symbols:

<code>area_name\$\$Base</code>	Address of the start of the area.
<code>area_name\$\$Limit</code>	Address of the byte beyond the end of the area.
<code>area_name</code>	The name of the area in the output file. You can use these symbols in your programs to refer to the Base and Limit of areas in your programs.

In addition, Link creates four conceptual areas in the output, and defines Base and Limit symbols for them.

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

Although it will often be the case, there is no guarantee that the end of the read-only area corresponds to the start of the read/write area. You should not therefore rely on this being true.

The read/write (data) area may contain code, as programs are sometimes self-modifying. Similarly, the read-only (code) area may contain read-only data (e.g. strings, floating-point constants etc.).

Command line interface

The format of the Link command is:

`Link options file_list`

Options

Abbreviations are shown capitalised.

General options

<code>-Output file</code>	Put final output in <i>file</i>
<code>-Debug</code>	Include debugging information in image
<code>-ERRORS file</code>	Put diagnostics to <i>file</i> , not stderr
<code>-LinkVersion vers</code>	Aborts the link if link isn't at least that specified as a parameter. This can be used to ensure you have a new enough linker to support the things you are needing it to do. For example, to check that the linker supports the initialisation of static data in modules, use: <code>-LinkVersion 5.25</code>
<code>-LIST file</code>	Put Map and Xref listing to <i>file</i> , not stdout
<code>-VIA file</code>	Take more object file names from <i>file</i>

-Verbose	Give informational message while linking
-MAP	Print an area map to the standard output
-Xref	Print an area cross-reference list
-Symbols <i>file</i>	List symbol definitions to <i>file</i>
-SymDefs <i>file</i>	Create or update ARM SYMDEFS definition in <i>file</i> . These files are compatible with ADS, the ARM Development System. The file will contain a textual symbol table that the linker can also input. This can be useful in certain cases and it is used by RISC OS ROM modules which use the Shared C library SYMDEFS.

Output options

-AIF	Absolute AIF (the default)
-AIF -Relocatable	Relocatable AIF
-AIF -R -Workspace <i>n</i>	Self-moving AIF
-AOF	Partially linked AOF
-BIN	Plain binary
-BIN -AIF	Plain binary described by a prepended AIF header
-IHF	Intellec Hex Format (readable text)
-SPLIT	Output RO and RW sections to separate files (-BIN, -IHF)
-SHL <i>file</i>	Shared-library + stub, as described in <i>file</i>
-SHL <i>file</i> -REENTrant	Shared-library + reentrant stub
-RMF	RISC OS Module
-Overlay <i>file</i>	Overlaid image as described in <i>file</i>
-ELF	Executable and Linking Format
-ELF -Partial	Partially linked ELF output
-UTIL	RISC OS transient utility

Special options

-RO-base <i>n</i>	
-Base <i>n</i>	Specify base of image
-RW-base <i>n</i>	
-DATA <i>n</i>	Specify separate base for image's data

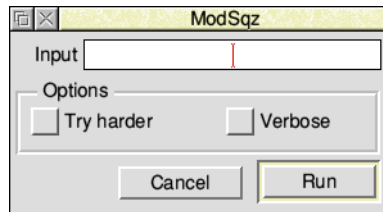
-Entry <i>n</i>	Specify entry address
-Entry <i>n+obj (area)</i>	Specify entry as offset <i>n</i> within <i>area</i> found in object file <i>obj</i> (prefix <i>n</i> with & or 0x for hex; postfix with K for *2 ¹⁰ , M for *2 ²⁰)
-Case	Ignore case when symbol matching
-MATCH <i>n</i>	Set last-gasp symbol matching options
-FIRST <i>obj (area)</i>	Place <i>area</i> from object <i>obj</i> first in the output image
-LAST <i>obj (area)</i>	Place <i>area</i> from object <i>obj</i> last...
-REMOVE	Remove unused areas
-NOZEROinit	Explicitly initialise all zero-init areas. This means the linker will place explicit zero bytes in the image and there will not be any zero-init areas that are zeroed at runtime by the executable's initialisation code
-NOUNUSEDareas	Don't eliminate AREAs unreachable from the AREA containing the entry point (AIF images only)
-Unresolved <i>sym</i>	Make all unresolved references refer to <i>sym</i>
-rescan	Instructs the linker to rescan the libraries to satisfy unresolved references imported from objects in later libraries. This option should not be used routinely – only when co-dependent libraries are being used.
-C++	Support C++ external naming conventions



ModSqz compresses a relocatable module, saving disc space and potentially making the module load faster.

The SetUp dialogue box

Clicking Select on the application icon or dragging the name of a file (if **Auto Run** is off) from a directory display to the icon brings up the SetUp dialogue box:



The **Input** writable icon allows you to specify the name of a file to be processed (typed in or dragged from a directory display). This file must be a relocatable module.

Try harder will force ModSqz to compress the file even if the file is considered by ModSqz to be too small to warrant compression.

Verbose outputs messages and compression statistics.

The SetUp menu

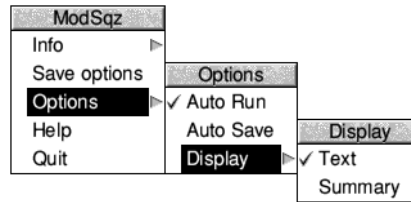
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the ModSqz **Command line** options see the section *Command line interface* on page 166.

The Application menu

Clicking Menu on the ModSgz application icon gives the following options:



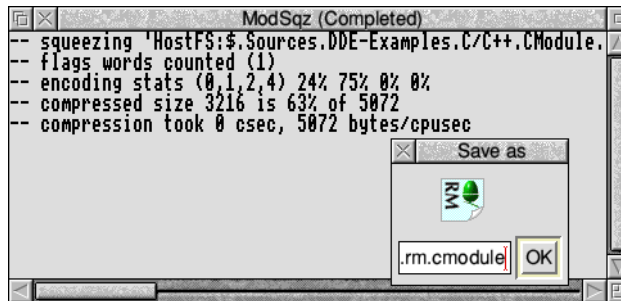
When **Auto save** is enabled, squeezing overwrites the input file with the squeezed version automatically without producing a save dialogue box for you to drag the file from. **Auto save** is off by default, whereas **Auto Run** is on by default.

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

Example output

The output of ModSgz appears in one of the standard non-interactive tool output windows. For more details of these see the section *Output* on page 107.

The following window shows an example of the output from ModSgz, together with a standard save dialogue box (which appears if **Auto Save** is not enabled):



Command line interface

For normal use you do not need to understand the syntax of the ModSgz command line, as it is automatically generated for you from the SetUp dialogue box settings. The command line syntax for ModSgz is:

```
ModSgz [options] unsqueezed-file [squeezed-file]
```

Options

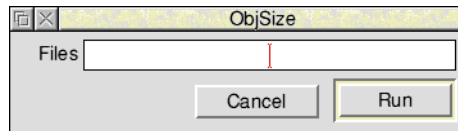
- f compress file regardless of size
- v output messages and compression statistics
- unsqueezed-file* a valid pathname specifying an input module file
- squeezed-file* a valid pathname specifying an output module file



ObjSize analyses one or more object or library files and returns the code-size, data-size and debug-size of each file.

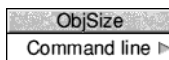
The SetUp dialogue box

Clicking Select on the application icon or dragging the name of a file (if **Auto Run** is off) from a directory display to the icon brings up the SetUp dialogue box:



The **Files** field allows you to specify the name of one or more files to be processed (typed in or dragged from a directory display). These files must be ALF or AOF files.

The SetUp menu

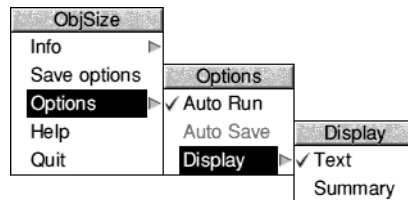


Clicking Menu on the SetUp dialogue box displays the menu shown on the left.

For a description of the ObjSize **Command line** option see the section *Command line interface* on page 170.

The Application menu

Clicking Menu on the ObjSize application icon gives the following options:



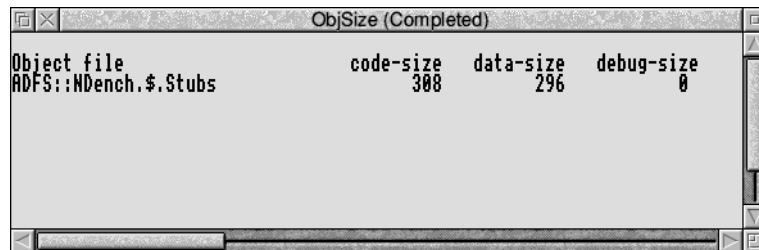
For a description of each option in the application menu see the chapter *General features* on page 103.

Note that **Auto Save** is not available for this application, and that **Auto Run** is enabled by default.

Example output

The output of ObjSize appears in one of the standard non-interactive tool output windows. For more details of these see the section *Output* on page 107.

The following window shows an example of the output from ObjSize:



The three object sizes displayed by ObjSize are:

- `code-size` The size of the object code.
- `data-size` The total size of all areas in the AOF file which have the attribute `data` or `zero-Init`.
- `debug-size` The total size of all areas in the AOF file (compiled with the debug option set) which have the attribute `debug`.

If a library file is being analysed ObjSize displays the above three object sizes for each individual member of the library file and then displays the overall totals of these to provide a set of totals for the entire library.

Command line interface

For normal use you do not need to understand the syntax of the ObjSize command line, as it is automatically generated for you from the SetUp dialogue box settings. The Command Line syntax for ObjSize is:

```
ObjSize filename [filename...]
```

filename a valid pathname specifying an ALF or AOF file.



SID is a general-purpose disassembler for ARM binaries. It does not currently disassemble Thumb binaries.

As the binary is processed, SID can generate various warnings about potentially problematic constructs. For example, code which is not 32-bit safe, instructions with Undefined or Unpredictable behaviour or branches to locations outside the binary.

SID makes an attempt at intelligently deciphering common structures, like C functions, branch tables, module SWI tables, etc. To do all this, SID must employ various heuristics because, even though assembling (or even compiling) source code into binary is a systematic process, conversion in the opposite direction is not.

Full documentation can be found in the file `Documents.DDE.SID`

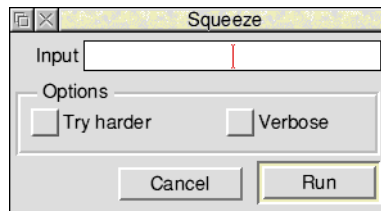


Squeeze compresses an executable ARM-code program, saving disc space and often making the program load faster.

It is also possible to squeeze relocatable modules, but it is better to use the ModSgz tool for this purpose (see page 165). If a module is squeezed it is converted into a program which installs the module in the RMA when run. This program contains a binary image of the module within itself. Squeeze compresses this program. In contrast, modules compressed with ModSgz are still modules.

The SetUp dialogue box

Clicking Select on the application icon or dragging the name of a file (if **Auto Run** is off) from a directory display to the icon brings up the SetUp dialogue box:



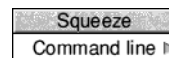
The **Input** writable icon allows you to specify the name of a file to be processed (typed in or dragged from a directory display). This file must be an AIF file.

Try harder will force Squeeze to compress the file even if the file is considered by Squeeze to be too small to warrant compression.

Verbose outputs messages and compression statistics.

The SetUp menu

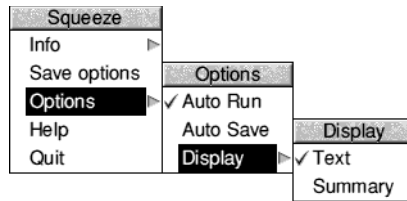
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the Squeeze **Command line** option see the section *Command line interface* on page 174.

The Application menu

Clicking Menu on the Squeeze application icon gives the following options:



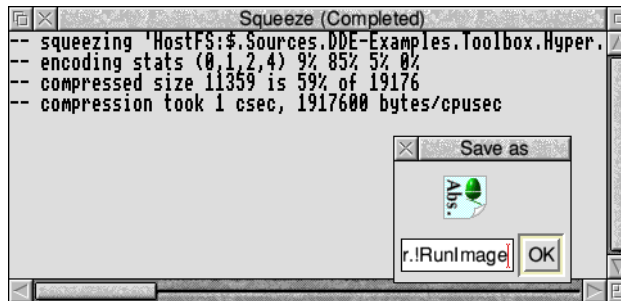
When **Auto save** is enabled, squeezing overwrites the input file with the squeezed version automatically without producing a save dialogue box for you to drag the file from. **Auto save** is off by default, whereas **Auto Run** is on by default.

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

Example output

The output of Squeeze appears in one of the standard non-interactive tool output windows. For more details of these see the section *Output* on page 107.

The following window shows an example of the output from Squeeze, together with a standard save dialogue box (which appears if **Auto Save** is not enabled):



Command line interface

For normal use you do not need to understand the syntax of the Squeeze command line, as it is automatically generated for you from the SetUp dialogue box settings. The command line syntax for Squeeze is:

```
Squeeze [options] unsqueezed-file [squeezed-file]
```

Options

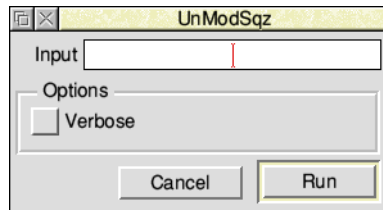
- f compress file regardless of size
- v output messages and compression statistics
- unsqueezed-file* a valid pathname specifying an input AIF file
- squeezed-file* a valid pathname specifying an output AIF file



UnModSqz reverses the effect of the ModSqz tool, expanding a compressed module back to its original state.

The SetUp dialogue box

Clicking Select on the application icon or dragging the name of a file (if **Auto Run** is off) from a directory display to the icon brings up the SetUp dialogue box:



The **Input** writable icon allows you to specify the name of a file to be processed (typed in or dragged from a directory display). This file must be a relocatable module.

Verbose outputs messages and compression statistics.

The SetUp menu

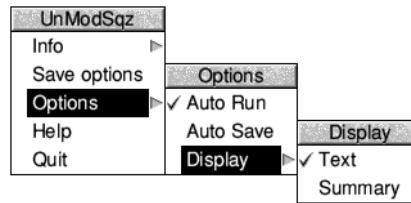
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the UnModSqz **Command line** options see the section *Command line interface* on page 178.

The Application menu

Clicking Menu on the UnModSgz application icon gives the following options:



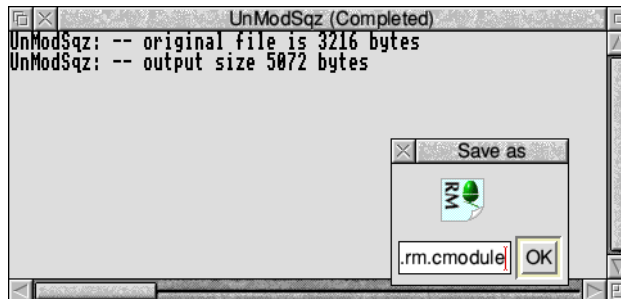
When **Auto save** is enabled, unsqueezing overwrites the input file with the expanded version automatically without producing a save dialogue box for you to drag the file from. **Auto save** is off by default, whereas **Auto Run** is on by default.

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

Example output

The output of UnModSgz appears in one of the standard non-interactive tool output windows. For more details of these see the section *Output* on page 107.

The following window shows an example of the output from UnModSgz, together with a standard save dialogue box (which appears if **Auto Save** is not enabled):



Command line interface

For normal use you do not need to understand the syntax of the UnModSgz command line, as it is automatically generated for you from the SetUp dialogue box settings. The command line syntax for UnModSgz is:

```
UnModSgz [options] squeezed-file [unsqueezed-file]
```

Options

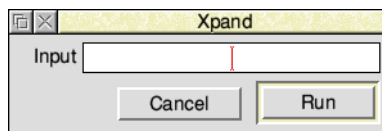
- `-v` output messages and compression statistics
- `squeezed-file` a valid pathname specifying an input module file
- `unsqueezed-file` a valid pathname specifying an output module file



Xpand reverses the effect of the Squeeze tool, expanding a compressed ARM-code program back to its original size.

The SetUp dialogue box

Clicking Select on the application icon or dragging the name of a file (if **Auto Run** is off) from a directory display to the icon brings up the SetUp dialogue box:



The **Input** writable icon allows you to specify the name of a file to be processed (typed in or dragged from a directory display). This file must be a squeezed AIF file.

The SetUp menu

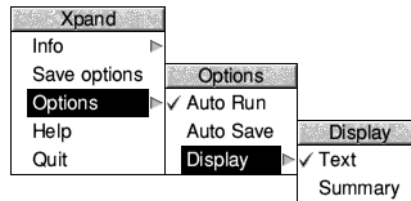
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the Xpand **Command line** options see the section *Command line interface* on page 183.

The Application menu

Clicking Menu on the Xpand application icon gives the following options:



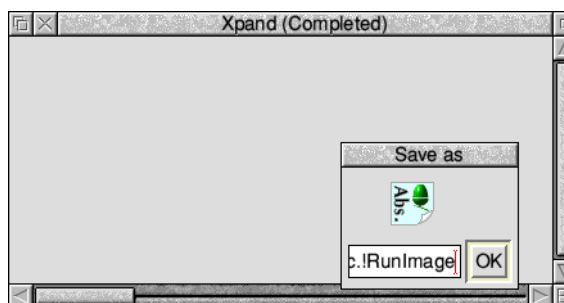
When **Auto save** is enabled, expanding overwrites the input file with the expanded version automatically without producing a save dialogue box for you to drag the file from. **Auto save** is off by default, whereas **Auto Run** is on by default.

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

Example output

The output of Xpand appears in one of the standard non-interactive tool output windows. For more details of these see the section *Output* on page 107.

When Xpand runs successfully the output window will be empty. The following example shows this, together with the standard save dialogue box which appears if **Auto Save** is not enabled:



Command line interface

For normal use you do not need to understand the syntax of the Xpand command line, as it is automatically generated for you from the SetUp dialogue box settings. The command line syntax for Xpand is:

```
Xpand squeezed-file [unsqueezed-file]
```

Options

<i>squeezed-file</i>	a valid pathname specifying an input AIF file
<i>unsqueezed-file</i>	a valid pathname specifying an output AIF file

The underlying technologies used in Acorn C/C++ have been designed in a way which allows third parties to add tools and applications, provided that they follow a number of rules and conventions which are given in this section. Unless you are a software developer, intending to use these technologies in your products, or intending to add further desktop tools, then you can skip this section. (Of course you may just be interested in how it all works, in which case read on!).

The FrontEnd module will act as a generic application, as described in the chapter *General features* on page 103. It is assumed here that you are familiar with this chapter, and that you have a feel for how the non-interactive tools operate.

The extensions you can make fall roughly into the following categories:

- Adding a compiler for another language – this will require all of the information given below.
- Adding a utility that you wish to run under the desktop, with the same look and feel as the other desktop non-interactive tools. For instance you may like to port the UNIX sed stream editor to RISC OS, with a Wimp front end – this only requires knowledge of how to describe an application to the FrontEnd module.
- Creating your own project management tool, similar to Make – this will require knowledge of the message-passing protocols used with the FrontEnd module, and also the format of a makefile used to maintain a project.

In this chapter you will find further technical information on the following:

- the FrontEnd module
- the DDEUtils module
- the SrcEdit editor
- the Make project management tool.

The FrontEnd module

Overview

The purpose of the FrontEnd module is to ease the job of putting consistent Wimp frontends onto a number of simple tools which are normally driven from the command line (e.g. Link, CC, ObjAsm etc.). A Wimp application can then be made by supplying a formal description of the mapping between the Wimp interface and command line options, a templates file, !Run, !Sprites and !Boot files, a message file, and a !Help file (also a !SetUp file if this is to be used by Make – see *Make* on page 203 for more details).

To give you a feel for how the FrontEnd module interacts with your command line tool, here is a brief description of how it works. The FrontEnd module understands two star commands:

`*FrontEnd_Start`

`*FrontEnd_SetUp`

The former of these is used to invoke a Wimp front end for a tool, with an icon on the icon bar; the latter is used to allow Make options for the tool to be set using a Wimp interface.

***FrontEnd_Start**

When the FrontEnd module gets a `*FrontEnd_Start` command it creates a new instantiation of itself called `FrontEnd%toolname` where *toolname* is the name of the tool invoked; it then enters that instantiation as the current application, and does a SWI WimpInitialise to become a Wimp task. Because this task stops the Wimp from mapping out its application workspace, by responding to service call 0x11, the task appears in the applications task list of the Task Manager display. From this point on, the behaviour of the Wimp task is governed by the formal description file which was initially passed to the `*FrontEnd_Start` command.

***FrontEnd_SetUp**

The `*FrontEnd_SetUp` command is similar, except it calls its new instantiation `FrontEnd%Mtoolname`, and does not produce an icon on the icon bar. The templates for windows used by the application must be provided by you, and they must follow the conventions laid down later in the section *Template files* on page 190.

When the user causes the command line tool to be run (for example by clicking on the **Run** icon in the application's dialogue box), the FrontEnd module starts up a task called *toolname_task* running under the control of the task window module; thus the tool is pre-emptively multitasked, and any output the tool

produces is stored and will be displayed in a window, if this is what the user wishes. When the user quits the application, the FrontEnd module ensures that the relevant instantiation is also removed from the RISC OS module list.

Example

To be suitable, your command line program has to be non-interactive. This means it should start with a command line, then run to error or completion without any further user interaction, outputting reports as screen text. A compiler such as CC fits this description, but an editor such as SrcEdit does not.

The tool ToANSI is a simple example of the non-interactive desktop tools. You may find it instructive to examine the file Desc in Apps.DDE. !ToANSI.

Changes since previous versions

This section describes FrontEnd v1.27 which has been significantly extended since the version released with the fifth release of Acorn C/C++.

The major changes are:

- New `ctrl_chars` and `tab_width` keywords for controlling text output
- Text output is now displayed immediately, rather than waiting for a newline (this allows progress indicators to output a series of dots for example)
- Text output lines may be terminated with CR, LF, CR+LF or LF+CR
- `quoted_string` keyword includes text inside quotes
- Icons can now be made to generate text when they are de-selected

Producing a complete Wimp application

In order to produce a complete Wimp application you will need to provide the following:

- !Run, !Boot and (possibly) !SetUp files
- a !Sprites file
- a Templates file
- a Description file
- a Messages file (optional)
- a !Help file (optional).

These are described in more detail below.

!Run, !Boot and !SetUp files

Your !Boot file will be the same as for normal applications, including doing things like setting file types, and performing *IconSprites commands on your sprites.

A typical !Run file will look like any of those supplied with the desktop non-interactive tools, such as !Link, !Find, or !Diff. The size of WimpSlot does not depend in any way on the size of the command-line tool which is running under the FrontEnd module, but instead refers to the application workspace used by the module, when starting up as a Wimp task (currently a minimum of 16k). You should ensure that you have a command of the following form:

```
*Set toolname$Dir <Obey$Dir>
```

so that your resource files can be found. Having made sure that the FrontEnd and Task Window modules are loaded (by using *RMEnsure) you then issue the *FrontEnd_Start command with application name and full pathname of the description file as parameters. You may need the facilities provided by the DDEUtils module, in which case you should *RMEnsure it in your !Run file

For example for !Diff, the !Run file is:

```
*If "<System$Path>" = "" Then Error 0 System resources cannot be found
*WimpSlot -Min 128k -Max 128k
*IconSprites <Obey$Dir>.!Sprites
*Set Diff$Dir <Obey$Dir>
*RMEnsure UtilityModule 3.10 Error This application only runs on RISC OS 3 (version
3.10) or later
*RMEnsure SharedCLibrary 5.34 Error This application requires the Shared C Library
module (is it unplugged?)
*RMEnsure FPEmulator 2.87 Error This application requires the FP Emulator module
(is it unplugged?)
*RMEnsure TaskWindow 0.47 Error This application requires the Task Window module
(is it unplugged?)
*RMEnsure FrontEnd 0 System:modules.frontend
*RMEnsure Frontend 1.21 Error You need version 1.21 of the FrontEnd module to run
!Diff
*RMEnsure DDEUtils 0 System:modules.ddeutils
*RMEnsure DDEUtils 1.52 Error You need version 1.52 of the DDEUtils module to run
!Diff
*WimpSlot -Min 32k -Max 32k
*FrontEnd_Start -app Diff -desc <Diff$Dir>.desc
```

A typical !SetUp file is very similar to a !Run file, but will be used when the FrontEnd module gets a request from Make to start up the Wimp front end for a tool, to allow the user to set options from a dialogue box. This file should only need to do the following:

```
*Set toolname$Dir <Obey$Dir>
*RMEnsure FrontEnd 0 System:modules.frontend
*RMEnsure Frontend 1.31 Error You need version 1.31 of the FrontEnd module to set
the toolname options from Make
*WimpSlot -min 16k -max 16k
*FrontEnd_SetUp -app %0 -desc %1 -task %2 -handle %3 -toolflags %4
```

Again, examples of a !SetUp file can be found in the set of non-interactive desktop tools.

!Sprites file

The !Sprites file will contain the sprite for the application icon on the icon bar, and also optionally a small sprite, both of which should comply with RISC OS style. The name of the large sprite should be the same as the application (e.g. !Link, !Find etc).

Template files

The set of window templates which you should supply in a file called `Templates` is as follows:

Window name	Status	Details
progInfo	Mandatory	<p>Should be as standard Acorn applications information boxes.</p> <p>Icon #1 must be indirected text, with a buffer size large enough to accept the application name.</p> <p>Icon #4 must be indirected text, with a buffer size large enough to accept the version string.</p>
SetUp	Mandatory	<p>This dialogue box is used to set the most common options for the command line tool. Rarer options can be set from a menu by the user pressing the Menu button on this dialogue box. The title bar must be indirected text, and have a buffer size large enough to accept the application name.</p> <p>Icon #0 must be indirected text (buffer size 12 bytes), and have a button type of <code>Click</code>, and should contain the text <code>Run</code>. It is used to invoke the command line tool with the chosen options.</p> <p>Icon #1 must be text, and have a button type of <code>Click</code>, and should contain the text <code>Cancel</code>. It is used to close the Options dialogue box, and revert to the options settings as they were when the dialogue box was last opened.</p> <p>Other icons are of your choice, and can be used to map to command line options. You must, however, follow the conventions described in the section <i>Writing an application description</i> on page 192.</p> <p>All icons must be de-selected in the template file – the default values should be set in the Description file.</p>

Window name	Status	Details
CmdLine	Mandatory	<p>This dialogue box is used to show the command line equivalent of the options which the user has chosen. The title bar should contain some explanatory text like <code>Command Line</code>.</p> <p>Icon #0 must be indirected text with buffer size 12 bytes, with button type <code>Click</code>, and containing the text <code>Run</code>. It is used to invoke the command line tool with the shown command line.</p> <p>Icon #1 must be indirected text with buffer size typically at least 256 bytes, and with a button type of <code>Writeable</code>.</p>
Help	Optional	<p>Used to display help text when the user selects Help from the application's main menu. The title bar should contain some appropriate text. The window should not have its <code>Auto-redraw</code> flag set.</p>
query	Mandatory	<p>Used to ask the user if they really want to kill off a task which is running.</p> <p>Icon #0 must be text, button type <code>Click</code>, and is used to reply <code>Yes</code>.</p> <p>Icon #1 must be indirected text, buffer size 256 bytes.</p> <p>Icon #2 must be text, button type <code>Click</code>, and is used to reply <code>No</code>.</p>
Output	Optional	<p>Used to display in a scrolling window, the textual output of the command line tool. The window's <code>Auto-redraw</code> flag must not be set.</p> <p>The title bar must be indirected text, and have a buffer size large enough to accept the application name, plus a space and the string <code>(Completed)</code>.</p>

Window name	Status	Details
Summary	Optional	Used to give a summary of the textual output produced by the command line tool. Icon #2 must be text, with button type <code>Click</code> , containing the text <code>Abort</code> . It is used to abort the task. Icon #3 must be indirected text, with a buffer size large enough to hold strings <code>Pause</code> and <code>Continue</code> , button type <code>Click</code> . It is used to pause and continue the task.
xfer_send	Mandatory if the Tool produces output that the user is able to save	Used as a save dialogue box for the textual output of a tool. Icon #0 must be text, with button type <code>Click</code> , containing the text <code>OK</code> . Icon #2 must be indirected text, with a buffer size of 256, and button type <code>writeable</code> . Icon #3 must be indirected text.
save	Mandatory if user is able to save anything	As for xfer_send, but is used to save the result file generated by running the tool. It should also have a <code>close</code> icon.

Writing an application description

As previously mentioned, your application running under the `FrontEnd` module is driven by a formal description written in a language whose EBNF (Extended Backus Naur Form) grammar is given in appendix *FrontEnd protocols* on page 231. This section gives an explanation of the semantics of the language, and hence explains how to write your own description.

As can be seen from the EBNF rule in appendix *FrontEnd protocols* for an application, the description file consists of 10 sections, with only the first section being mandatory (`TOOLDETAILS`). Each of these sections is described separately below, and the sections that are used must appear in the order shown below.

TOOLDETAILS section

The tool details section is the only section which you **must** have in the description. The section starts with the name of the tool, which must be the same as the string passed as the `-app` parameter to `*FrontEnd_Start`. This name will be used in window and menu title bars to identify the application.

Normally the tool will reside in your current library directory, and hence the command will be invoked using only the tool name. If you wish to change this you can specify a `command_is` entry, which gives a pathname for the tool. For example if you have an application called `example`, but the executable image for this application is held in `!RunImage` in the application directory, then you should have a line in the description file saying:

```
command_is "<example$Dir>.!RunImage";
```

The `version` entry will typically be a version number and optional date for the tool. These will be used in the Program Information dialogue box (`progInfo`).

If your tool understands a particular file-type, then this can be entered using the keyword `filetype`. This is used when the user double-clicks on a file of this type in a directory display. The effect is as if the user has dragged the file icon to your icon on the icon bar.

By default the tool is run in a Wimp slot of 640k, under the Task Window module. If you want this value to be different, then use the `Wimp slot` command in the description.

Since the limit on RISC OS command lines is 256 characters for RISC OS 3.00 (1,024 characters for RISC OS 4.00 and later versions), you may find this to be an unnecessarily strict limit when passing a potentially large list of full pathnames to a tool on its command line. If you use the `has_extended_cmdline` keyword in the description, then the `FrontEnd` module will request space from the `DDEUtils` module to place the command line arguments in. If the tool is written in C (or runs under any other run-time environment which cooperates with `DDEUtils`) the tool will pick up the arguments from `DDEUtils`. Using this option, your command line is limited only by the size of the writable icons in your dialogue boxes. If written in C, the tool must have been linked with the stubs or `ANSILib` to use this feature.

METAOPTIONS section

The `METAOPTIONS` section refers to non-application-specific options.

If the `has_auto_run` keyword is used, the application's main menu option **Auto Run** will not be greyed out. In addition, if you include the keyword `on`, then this option will be enabled by default. **Auto Run** means that if a file is dragged to the application icon, then the tool will immediately be run, rather than first displaying the Options dialogue box.

The `has_auto_save` keyword refers to the **Auto Save** option in the application's main menu, and the keyword on turns this option on by default. If this option is on, then rather than producing a Save as dialogue box to save the file output of the tool, the tool is run to directly write to the desired output place. The location where output should be sent is given following the `has_auto_save` keyword; in order to specify this location, you must first give an icon number in the Options dialogue box, whose first entry will be used to determine the directory where the output will go (using the `from icn <integer>` keywords).

For example, if you have the line:

```
has_auto_save ^."!RunImage" from icn 3;
```

and icon 3 of the options dialogue box contains the text:

```
adfs::4.$.objects.file1 adfs::4.$.objects.file2
```

then the filename `adfs::4.$.objects.file1` will be used to form the output filename. First the leafname `file1` is stripped off to leave the directory name `adfs::4.$.objects` which will form the stub of the output filename. This stub is then manipulated by the string which is specified between the keyword `has_auto_save` and the keyword `from`. You can indicate parent directories using any (reasonable) number of `^`'s and can refer to the original leafname using the keyword `leafname` (in this example `leafname` would map to `file1`). This leafname can have literal strings prepended or appended to it.

If the application is to have textual output, then you can specify that you want text and/or summary window(s) by using the keywords `has_text_window` and `has_summary_window`. Beware that if you don't have any output windows at all, then the user has no way of pausing/aborting/examining the running task. The default display mode is text, but this can be explicitly stated as text or summary using the keyword `display_dft_is`.

When writing to a text window, control characters (other than tab and newline) are normally filtered out and replaced with question marks and tab characters normally generate a single space. This default behaviour can be changed by using the following keywords:

`ctrl_chars escape` causes control characters to be output as lower case two-digit hexadecimal numbers in square brackets, for example `[1f]`.

`ctrl_chars hide` causes control characters to be suppressed and not output.

`ctrl_chars text` causes control characters to be output as a question mark.

`tab_width n` sets the tab width to `n` where `n` is 0 to hide tabs, 1 to show tab characters as a single space or 2-32 for true tab points of the specified width.

The saved output is not affected by the settings of `ctrl_chars` and `tab_width` as it is always the received data that is saved, not the data as it appears in the window.

FILEOUTPUT section

The FILEOUTPUT section deals with the production and saving of a single output object. To enable the user to then save this output, it is sent to a temporary file, which is then copied to a permanent file when the corresponding icon is dragged to a directory display – the icon can also be dragged to another application.

By default it is assumed that the output filename for a tool is that which appears last on the command line with no special preceding flag. If your command line tool requires a flag such as `-o` to go before the output filename, then this is specified using the `output_option_is` keyword.

Also by default, the name which appears in the Save as dialogue box is the string `Output`, assuming that no **Auto Save** string has been specified. This can be changed using the `output_dft_string` keyword.

Certain tools produce an output file, or not, depending on the combination of options on their command line. By using the `output_dft_is` keyword, you can specify whether the default mode of operation is to produce output or not. This state will then be changed as the user chooses options from the options dialogue box and menu which either turn output production on or off (see the DBOX section and the MENU section).

DBOX section

The DBOX section describes the properties of the main dialogue box used to set options for the command line tool.

The purpose of the icon definitions is to show how icon clicks and drags etc. map onto command line option strings, and how these affect the state of other icons and menu entries. Essentially, icon numbers correspond to those numbers used in the template for the dialogue box.

There are four types of icon definition:

- 1 those that map directly onto command line strings
- 2 those that increase or decrease the numeric value of another icon
- 3 those that cause a string to be inserted in a writable icon
- 4 those that extend and contract the dialogue box.

The most complex of these is the icon which maps to a command line string. Such an icon can be of two Wimp types:

- a writable indirected text icon

- a click icon.

The former of these contributes to the command line, if it contains any text, and is generally used for specifying filenames to the command line tool. The latter is generally used to turn flags on and off, and contribute to the command line. The mapping onto the command line is given after the keyword `maps_to`; this may begin with an optional string literal (e.g. `-f`), optionally followed by keywords `string`, `quoted_string` or `number`. These latter keywords are used for writable indirected text icons, and refer to their contents.

The `quoted_string` variant puts the string inside double quotes. For example, given a writable field at icon 20 holding text 'hello "world"':

```
icn 20 maps_to "-display " quoted_string;
icn 20 maps_to "-display " quoted_string prefix_by "-I";
```

would, respectively, result in:

```
-display "hello \"world\" "
-display "-Ihello -I\"world\" "
```

If you want each item in the writable text icon to be preceded by a particular string, this can be specified using the `prefix_by` keyword.

The keywords `on` and `off` may be used after the icon to indicate whether the mapping applies when the click icon is selected or de-selected. If both are absent `on` is assumed.

You can also specify that selecting this icon causes the values of other icons to be used in the command line, by using the `followed_by` keyword. These items will be separated by the entry given after the `separator_is` keyword. As discussed in the FILEOUTPUT section, it is possible to specify whether a tool produces output by default; each icon can be made to toggle this state using the keywords `produces_no_output` and `produces_output`. The `not_saved` keyword should be used if the value of the particular icon should not be saved when the user picks the **Save options** entry from the application's main menu.

Some examples should make this clearer:

```
icn 3 maps_to "-c";
```

This would be used for a click icon, which when selected will result in `-c` being inserted into the command line.

```
icn 3 on maps_to "-c";
icn 3 off maps_to "-x";
```

The first line has behaves exactly as in the previous example. The second line causes `-x` to be inserted into the command line if icon 3 is de-selected.

```
icn 6 maps_to "-f " string not_saved;
```

This would be used for a writable indirected text icon, whose string contents should follow the literal `-f` on the command line. It would typically be used for specifying input filename(s). The contents of icon 6 would not be saved when the user chose the **Save options** menu entry.

Using the `increases` or `decreases` keyword is typically used for arrow icons, used to increase and decrease the numeric value of another icon. The default amount by which the increase or decrease is made is 1, but this can be changed using the keyword `by`. Minimum and maximum values can also be specified. The button type of such an arrow icon should be `click` or `auto-repeat`.

If an icon should just be used to insert a useful string in another writable indirected text icon, then this is specified using the keyword `inserts`. Whenever such an icon is clicked, the given string literal is inserted into the keyboard buffer, if the options dialogue box currently has the input focus. Its button type should be `Click`.

The `extends` keyword is used for an icon which is used to toggle the options dialogue box, from large to small and vice versa. The `from` icon number is the icon which is used to mark the bottom of the dialogue box when small; the `to` icon number is the icon which is used to mark the bottom of the dialogue box when large.

The list of icon definitions can optionally be followed by a list of icon default values, using the keyword `defaults`. Each icon can be listed with the keywords `on` and `off` for click icons, or a string or numeric literal value for writable indirected text icons. These defaults refer to those used when the tool is invoked via `*FrontEnd_Start`; if the tool has different options by default when invoked from `Make`, these are listed using the `make_defaults` keyword.

Following this in the description is an optional specification of what happens when drags occur, from the filer or from other applications. After the keyword `imports_start`, which begins this part of the description, you can optionally specify a `wild_card_is` string, which is used whenever a directory is dragged to your application. Typically this wildcard will be `*`. Hence a directory `adfs: :4.$.foo` dragged onto the application will expand to `adfs: :4.$.foo. *`. There then follows a list of `drag_to` specifications, each of which gives either a specific icon number in the dialogue box, or the keywords `any` or `iconbar`; the icon list following the word `inserts` is where the filenames of the dragged files will be inserted, with an optional separator string. If no separator string is given then a drag will overwrite the previous contents of the writable indirected text icon. Here are some examples:

```
drag_to icn 3 inserts icn 3;
```

This means that a drag onto icon 3, will insert the filename into icon 3, and subsequent drags to this icon will overwrite it.

```
drag_to icn 6 inserts icn 6 separator_is " ";
```

```
drag_to any inserts icn 6 separator_is " ";
```

```
drag_to iconbar inserts icn 6;
```

These means that a drag to icon 6, or anywhere else on the dialogue box, or to the icon bar will insert the filename of the dragged icon in icon number 6. In the case of the iconbar, the contents of icon 6 will be overwritten.

It is not possible to use `followed_by` and `prefix_by` at the same time. For example, if icon 20 is an option button, 21 is a label and 22 a writable, the following is **not** possible:

```
icn 20 on maps_to "-e " followed_by icn 22 prefix_by "-I";
```

```
icn 22 maps_to " ";
```

However the desired effect can be achieved as follows:

```
icn 20 maps_to " ";
```

```
icn 22 maps_to "-e " string prefix_by "-I";
```

MENU section

The MENU section is similar to the DBOX section, except that it is used to specify the way that menu entries on the menu attached to the options dialogue box map to command line option strings. This menu is typically for less commonly used options.

Each entry in the menu entry list begins with a literal string, which is used to give the text that will appear in that menu entry. This may be followed by the keywords `on` and `off` to indicate whether the mapping applies when the menu item is selected or de-selected. If both are absent `on` is assumed.

This is followed, after the keyword `maps_to`, by string literal (which may be null) to which that menu entry maps in the command line. This is optionally followed by the keyword `sub_menu`, in which case this menu entry will be given a writable submenu with the given string literal as its title, and with a buffer size given by the supplied integer value. If you want each item in the submenu buffer to be preceded by a particular string, this can be specified using the `prefix_by` keyword. The `produces_output`, `produces_no_output` and `not_saved` keywords are as described above for the DBOX section.

Menu default values can be set in a similar manner to those for the dialogue box icons. This is done using the `defaults` keyword, and then following each menu entry with the keyword `on` or `off` depending on the desired default state of that entry. If the entry has a writable submenu, this can also be given a default string or integer value. Also a separate set of option defaults can be set for when the `FrontEnd` module is invoked from `Make`. Menu entries are numbered from 1 (ignoring the command line equivalent entry).

For example:

```
menu_start
  "First option" maps_to "-a";
  "Second option" maps_to "-b " sub_menu "Value: " 8;
defaults
  menu 1 off,
  menu 2 on sub_menu "42";
menu_end
```

will result in a menu with two entries (other than the command line equivalent, which is always the first entry). By default **First option** will not be ticked, but **Second option** will be ticked and its writable submenu will contain the value 42.

SELECTIONS section

This section allows you to state which options when enabled should enable other options. This can be done for both icons in the main options dialogue box and for entries in its attached menu. For example:

```
selections_start
  icn 3 selects icn 4, icn 5, menu 3;
selections_end
```

means that when icon 3 is selected, then icons 4 and 5 and menu entry 3 will be selected.

`on` and `off` keywords may be used to indicate that deselecting an icon should cause the action respectively (the `on` keyword is redundant as it is assumed if absent, but it may be used for clarity). For example:

```
icn 3 off selects icn 4, icn 5, menu 3;
```

means that when icon 3 is selected, then icons 4 and 5 and menu entry 3 will be selected too.

DESELECTIONS section

This works just like the selections section described above:

```
deselections_start
    icn 3 on deselects icn 4, icn 5, menu 3;
deselections_end
```

means that when icon 3 is selected, then icons 4 and 5 and menu entry 3 will be deselected.

INCLUSIONS section

The INCLUSIONS section is similar to the SELECTIONS section, except that the listed icons and menu entries are made selectable rather than selected. When the icon or menu which caused this inclusion is deselected, then the included items become selectable again.

Inclusions can be useful where, for example, turning on an option box makes a writable field valid (e.g. an output filename for some logging text).

MAKE_EXCLUSIONS section

Certain tools require that some options are made unselectable when the FrontEnd module is invoked from Make. The MAKE_EXCLUSIONS section allows these icons and menu entries to be listed.

EXCLUSIONS section

The EXCLUSIONS section is similar to the INCLUSIONS section, except that the listed icons and menu entries are made unselectable (greyed out). When the icon or menu which caused this exclusion is deselected, then the excluded items become selectable again.

RULES section

To simplify the rules and make them more readable it is possible to include all of the selections, deselections, inclusions and exclusions in a single rules section. This may be used to replace those sections or it may follow them to provide additional rules.

The RULES section is formatted in the following way:

```
rules_start
    <list [on|off]> <selects|deselects|includes|excludes> <list>;
rules_end
```


For example:

```
rules_start
    icn 3 off selects icn 10;
    menu 3 on deselects icn 4;
    menu 3 on excludes icn 5, icn 6, menu 4;
rules_end
```

Rules may appear in any order within the rules section.

Rules are executed recursively when an icon state changes. For example, if the user were to switch on icon 4 given the following:

```
icn 4 selects icn 5;
icn 5 selects icn 6;
icn 6 selects icn 7, icn 8;
icn 8 deselects icn 5;
```

then icons 5, 6, 7 and 8 would be selected, but the selection of icon 8 would deselect icon 5, so that would be deselected again. The calculation of rules is done internally and the front-end updated only with the final outcome, so for most valid rules minimal flicker would be observed for these kind of complex rules. Circular references should be avoided but should not lead to problems as FrontEnd should spot them.

ORDER section

By default the command line for the tool is constructed in the following order:

- 1** the dialogue box icons in the order given in the DBOX section
- 2** the menu entries in the order given in the MENU section
- 3** the output option if appropriate.

If this ordering is not satisfactory, you can give another ordering by using the `order_is` keyword followed by a list of icon numbers, menu entries and string literals. This mechanism can be used to insert string literals which always appear on the command line.

MAKE_ORDER section

The `MAKE_ORDER` section is similar to the `ORDER` section, except that it gives a way of specifying an alternative command line ordering, when invoked from Make.

Messages files

There are a number of textual messages (warnings and errors and the like), which the FrontEnd module issues. The purpose of the messages file for an application is to allow internationalisation of the messages. A messages file is supplied with

each of the non-interactive tools, which you can use for your application; it should be in a file called `<toolname$Dir>.Messages`. If no such file is present, then FrontEnd's internal default English messages are used.

Providing interactive help

Responses to interactive help requests are handled by the FrontEnd module. In each of the desktop non-interactive tools directories you will find a Messages file for the tool. In this file are help messages for the various dialogue boxes of the tools. In general a message whose tag field is the name of the dialogue box, is used when the pointer is not over an icon; when the pointer is over an icon, the icon number is used to distinguish the help message.

For example, an entry in the messages file of:

```
SETUP3:This is where you specify the input filenames  
will result in the message
```

```
This is where you specify the input filenames
```

appearing in !Help's interactive help window, when the pointer is over icon number 3 of the SetUp dialogue box.

!Choices file

When the user selects **Save choices** from the application's main menu, the current setting of options is saved in a file `<toolname$Dir>.!choices`.

The DDEUtils module

The DDEUtils module is intended for three purposes:

- to relax the command line length limit
- to solve the problem of 'current directory' under the desktop
- to provide throwback to the editor on finding source errors.

Further details are given in appendix DDEUtils on page 237.

SrcEdit

Resource files

A language compiler needs to supply three lines of information about itself to SrcEdit when it is installed. It does this by appending these three lines to the file `<SrcEdit$Dir>.choices.languages` of the form shown in appendix *SrcEdit file formats* on page 245.

The language help file is used when the user selects a portion of his text and requests language help on this. The format of entries in the help file is shown in appendix *SrcEdit file formats*.

Make

You will have noticed that when the user selects Menu on a project in Make, it is possible to select options for a tool, by picking the name of that tool from the **Tool options** menu. This is done by Make issuing the star command `*FrontEnd_SetUp`; the FrontEnd module then replies with a Wimp message (details of which are given in appendix *FrontEnd protocols* on page 231) containing the desired command line.

In order to achieve this, a tool which is being added must append six lines to the file `<Make$Dir>.choices.tools_def` of the form:

<i>tool_name</i>	
<i>extension</i>	the string used to identify a source written in this language; e.g. c for the C language
<i>make_defaults</i>	the default options for this tool when in a makefile
<i>conversion_rule</i>	i.e. how to convert source files to object files
<i>description_file</i>	full pathname of file containing application description
<i>setup_file</i>	full pathname of file containing SetUp actions for when tool is invoked via Make

Appendices



Appendix A: Changes to the tools

The major changes since the final release of the Acorn C/C++ compiler and the associated tools by Acorn Computers are:

- Tools are now 32-bit compatible and suitable for use on 32-bit versions of RISC OS.
- The Makefile syntax has been greatly extended and is largely compatible with the GNU Makefile syntax. New features include conditional directives; inclusion files; *simply expanded* variables and string functions – see appendix *Makefile syntax* on page 209 for details.
- The FrontEnd module has been extended with additional keywords and improved rule handling for selecting, deselecting and shading groups of icons. See *The FrontEnd module* on page 186 for details.
- DecAOB disassemblers know about all 32-bit ARM instructions up to ARMv8.
- DDT emulation is now an order of magnitude faster. It also supports the new C99 features and emulates ARM instructions up to ARMv5.
- Diff, Find and the other tools can now use long filenames, making them suitable for use with RISC OS 4 and filing systems that allow more than 10 characters.
- The APCS-32 ARM Procedure Calling Standard is now supported.
- ELF and SYMDEFS files can be processed as library members by LibFile.
- The linker now provides initialised static data in modules, new options including `-linkversion` and `-rescan`, support for ARM SYMDEFS and support for creating transient utilities.
- Various faults have been rectified.
- ResGen is now included. This is a tool for creating resource file data in a form suitable for use by ResourceFS from a set of input files and corresponding resource file names. It was originally supplied with Acorn's Application Note 280 and now has `-26bit` and `-32bit` options to select the APCS variant.
- The A8Time, ABC, SID, ModSqz, UnModSqz and Xpand tools have been added.
- SrcEdit has been updated to support the global clipboard.



Appendix B: Makefile syntax

This appendix covers the syntax of Makefiles understood by AMU, and the way they are arranged by Make. If all you need to do is construct and use simple Makefiles with Make, you do not need to study this information. It is included for those wishing to study, modify or construct Makefiles manually.

The Makefile syntax used here is very similar to that used by GNU Make and there should be few difficulties in using GNU-style Makefiles with AMU.

There have also been several enhancements since AMU 5.05 which was supplied with the original Acorn C/C++ development environment. In particular, functions, directives and enhanced macros are now available. The ability to include other files is particularly useful for very large projects as it allows certain rules to be abstracted into a separate file for including in each Makefile.

Make and AMU

Makefiles may be constructed by hand, using a text editor such as SrcEdit, or semi-automatically using Make. For more details of operating Make, see the chapter *Make* on page 59. Makefiles may be used to run a make job using either Make or AMU. In both cases, make jobs operate by the command line tool `amu` interpreting the Makefile text and issuing command lines to other tools. The command line tool `amu` is installed in your library directory.

Command execution

AMU executes commands by calling the C library function `system`, once for each command to be executed. In turn, `system` issues an OS_CLI SWI to execute the command. Before calling OS_CLI, `system` copies its caller to the top end of application workspace and sets the workspace limit just below the copied program. Any command executed by AMU therefore has less memory to execute in than AMU had initially (the difference being the size of AMU plus the size of AMU's working space).

When the command returns, AMU will be copied back to its original location and will continue, unless, of course, the command set a bad (non-0) value in the environment variable `Sys$ReturnCode` (the C library automatically sets `Sys$ReturnCode` to the value returned by `main()` or passed to `exit()`). If you have limited memory on your computer, or you are trying to run AMU in a limited

Wimpslot under the desktop, and a program (such as the C compiler) to be run by AMU needs more memory than is left, you can instruct AMU not to execute commands directly, but to write them to an output window to be saved and executed later (see the **Don't execute** option of Make and AMU). Of course, in this case, execution is not terminated or modified by a non-0 return code from a command.

Finally, note that there is a RISC OS command length limit of 1,024 characters (256 characters in versions older than RISC OS 4.00). The desktop tools such as the linker and C compiler cooperate with the DDEUtils module to allow much longer command lines, but care must be taken to avoid generating long command lines for other operations, such as wipe, etc.

Makefile basics

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

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

Everything else that you can express in a Makefile is designed to make the job of description easier for you.

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

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

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

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

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

Each entry consists of

- a target, followed by a colon character, followed by

- a list of files on which the target depends, followed by
- a list of commands to execute to make the target up to date.

Each command line begins with some white space (if you want your Makefile to be portable to UNIX systems you should begin these lines with a Tab character). For example, `amu` itself is made from `o.amu`, the compiled AMU program, and a proprietary library called `$.301.clx.o.clxlib`. If either of these files is newer than `amu`, or if `amu` does not yet exist, then the commands `link -o amu ...` followed by `squeeze amu`, should be executed.

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

Thus AMU might well execute for you:

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

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

File name truncation

Machines that have file name truncation configured off can result in error messages being displayed where a Makefile contains a rule where a (non-file) target name has more than 10 characters.

For example, in the following Makefile extract:

```
install_rom: ${TARGET}
    ${CP} ${TARGET} ${DESTINATION}.${TARGET} ${CPFLAGS}
    @echo install_rom complete
```

typing in:

```
*amu install_rom
```

would result in the following error message:

```
AMU: failed to read time stamp for 'install_rom'
```

If you are going to use long target names you must ensure that file name truncation is configured on.

Macros as targets

The first target in a Makefile cannot be a macro. If you need to use a macro in this way then you should insert an 'extra' target.

For example:

```
all: ${PROG}

${PROG}: myprog.o
    @echo ${PROG} rebuilt
```

Makefile structure

Makefiles contain normal ASCII text, and are of type 0xFE1 (Makefile). For backwards compatibility they may also be used with text (0xFFF) file type, though these cannot be adjusted automatically by Make.

A Makefile consists of a sequence of logical lines. A logical line may be continued over several physical lines provided each but the last line ends with a \. For example:

```
# This is a comment line \
  continued on the next physical line \
  and on the next, but not thereafter.
```

A comment is introduced by a hash character # and runs to the end of the logical line. The active comment line:

```
# Dynamic dependencies:
```

is interpreted by AMU as a marker for the start of dependencies to be kept up to date during a make job (see *Makefiles constructed by Make* on page 227). All other comment lines are ignored by AMU.

Otherwise there are four kinds of non-empty logical lines in a Makefile:

- dependency lines
- command lines
- macro definition lines
- rule and other special lines.

Dependency lines have the form:

```
space-separated-list-of-targets COLON space-separated-list-of-prerequisites
```

For example:

```
amu : o.amu $.301.clx.o.clxlib
o.d35 o.d36 o.d37: h.util
```

A dependency line cannot begin with white space. Spaces before the colon are optional, but some white space must follow to distinguish a colon separating targets and prerequisites from a colon as part of a RISC OS filename.

For example:

```
adfs::4.$$.library.amu: o.amu ...
```

(Although a space after the colon is not required by UNIX's make utility, omission of it is rare in UNIX Makefiles).

A line with multiple targets is shorthand for several lines, each with one target and the same right-hand side (and the same associated commands, if any). Multiple dependency lines referring to the same target accumulate, though only one such line may have commands associated with it (AMU would not know in what order to execute the commands otherwise). For example:

```
amu: o.amu
amu: $.301.clx.o.clxlib
```

is exactly equivalent to the single line form given earlier. In general, the single line form is easier for you to write whereas the multi-line form is more readily generated by a program (for example, Make will generate a list of lines of the form `o.foo: h.thing`, one for each `#include thing.h` in `c.foo`). Command lines immediately follow a dependency line and begin with white space.

For maximum compatibility with UNIX Makefiles ensure that the first character of every command line is a Tab. Otherwise one or more spaces will do. A semi-colon may be used instead of a new line to introduce commands. This is often used when there are no prerequisites and only a single command associated with a target. For example:

```
clean:; wipe o.* ~cfq
```

Note that, in this case, no white space need follow the colon.

Macro definitions

Macros may be defined in one of the following ways:

```
macro-name = value
macro-name := value
macro-name ?= value
macro-name += value
```

The macro name should comprise letters, numbers and the underscore symbol. Macro names are case sensitive, so `cc` and `CC` refer to different macros. The macro is defined to be *value* which is all of the text on the line following the equals sign. The `=` can be surrounded with white space, or not, to taste.

See the section *Macro priority* on page 223 for more details on macros. The different forms of definition are explained below.

Defining recursively expanded macros

Recursively expanded macros are defined as follows

```
macro-name = value
```

For example:

```
CC = ncc
CFLAGS = -fah -c -I$.clib
LD = link
LIB = $.CLib.o.clxlib $.CLib.o.Stubs
CLX = $.301.clx
```

Thereafter, wherever `${name}` or `$(name)` is encountered, if *name* is the name of a macro then the whole of `${name}` is replaced by its definition. A reference to an undefined macro simply vanishes. If the macro contains references to other macros those references are expanded whenever this macro is substituted (i.e. the current value of those macros is substituted when this macro is used).

An example which uses the above macro definitions, and which is taken from the Makefile for AMU itself, is:

```
amu:      amu.o $(CLX).o.clxlib
          $(LD) -o amu ${LFLAGS} o.amu ${LIB}
```

which expands to

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

Note that `${LFLAGS}` expands to nothing.

Simply extended macros

macro-name := *value*

This defines a *simply extended macro* where the value is macro expanded immediately. The value of a simply expanded macro is scanned once and for all, expanding any references to other macros, when the macro is defined. It does not contain any references to other macros; it contains their values as of the time this macro was defined. Therefore,

```
x := foo
y := $(x) bar
x := later
```

is equivalent to

```
y := foo bar
x := later
```

When a simply expanded variable is referenced, its value is substituted verbatim, so unlike normal macros, forward references are not allowed.

Conditional macro assignment

macro-name ?= *value*

This does conditional assignment – it defines a recursively expanded macro if the macro is not yet defined. Note that a macro may be defined to be empty in which case this conditional assignment will have no effect.

Adding to a macro definition

macro-name += *value*

This adds *value* to the end of the current macro definition. Whether the value is expanded before being appended depends on whether the macro being altered is a simply expanded or recursively expanded macro. If the macro was previously undefined it is defined as a recursively expanded macro.

By using macros intelligently, you can minimise the effort needed to move Makefiles from computer to computer; for example, dealing with varying locations for prerequisites, or centralising what would otherwise be distributed through many lines of text. It is obviously much easier to add -g to a CFLAGS= line to make a debuggable version of the compiler than it is to add -g to 28 separate cc commands. Similarly, using \$(CC) and CC=cc, rather than just cc, makes it very easy to use a different version of cc; just change the definition of the macro. Whilst this may not seem very useful in a small Makefile, it is common practice when describing larger systems such as the C compiler. Macros are used extensively in the Shared Makefiles provided in the AcornC/C++.Makefiles directory.

Advanced features

There are several advanced features in AMU 5.27 which were not in AMU 5.05 released with the original Acorn C/C++ development environment. These include substitutions, functions, directives and new rules for macro priority, as explained below. The new macro priority rules are the most likely source of problems when using existing Makefiles, and backwards compatibility options have been provided, as explained in *Macro priority* on page 223.

Substitutions

Basic substitutions are supported:

```
$(VAR:search=replace)
```

search and *replace* are simple strings, and the construct represents the value of the VAR macro with all occurrences of 'search' replaced by 'replace'. The more advanced GNU % substitutions are also supported.

Functions

Several functions are supported within Makefiles, using the same syntax as GNU make. These functions are particularly useful for processing filenames and lists of filenames. Where functions operate on lists each element in the list should be separated by one or more spaces. Multiple spaces are always treated the same as a single space. Where functions return a list, the elements are separated by a single space.

The functions supported are listed below:

\$(addsuffix *suffix*, *list*...)

This function adds the supplied *suffix* to each element in *list* to form a new list with the same number of elements. For example:

```
$(addsuffix .c,main display)
```

produces the result `main.c display.c'.

\$(addprefix *prefix*, *list*...)

This function prepends the supplied *prefix* to each element in *list* to form a new list with the same number of elements. For example:

```
$(addprefix c.,foo bar)
```

produces the result `c.foo c.bar'.

`$(dir names...)`

This function returns a list containing the directory-part of each filename in *names*. The directory-part of the file name is everything up to and including the last dot in it (note that RISC OS uses a dot as the directory separator). For example,

```
$(dir ADFS:.$.src.main/c)
```

produces the result 'ADFS:.\$.src.'.

`$(error text...)`

Generates a fatal error where the message is *text*. Note that the error is generated when this function is evaluated. So, if you put it inside a command script or assign it to a recursive variable then it won't be evaluated until later. The text is expanded before the error is generated. For example,

```
ifdef ERRMSG
$(error error is $(ERRMSG))
endif
```

`$(filter pattern...,list)`

Returns all of the items in *text* that match any of the patterns in the *pattern* list. The patterns are specified as in the *patsubst* function below. The filter function can be used to separate out different types of strings (such as file names) in a variable. For example:

```
sources := html.c editor.c render.s ucslib.h
target: $(sources)
        cc $(filter %.c,$(sources)) -o target
```

This means that *target* depends on all of the files in *sources* but only the c files should be compiled by *cc*.

`$(filter-out pattern...,list)`

This is the opposite of the filter function. . It returns all the items in *list* that do not match any of the patterns in the *pattern* list.

For example:

```
$(filter-out .h, main.c adfs.h usb.h)
```

will return 'main.c'.

`$(findstring find,in)`

Searches in for an occurrence of *find*. If it occurs, the value is *find*; otherwise, the value is empty. You can use this function in a conditional to test for the presence of a specific substring in a given string. Thus, the two examples,

```
$(findstring a,a b c)
```

```
$(findstring a,b c)
```

produce the values 'a' and '' (the empty string), respectively. See section Conditionals that Test Flags, for a practical application of findstring.

`$(firstword list...)`

Returns the first word in the list. For example,

```
$(firstword Now is the time)
```

produces the result now. See the word function to extract subsequent word.

`$(if condition,then-part[,else-part])`

The if function returns *then-part* if *condition* is *true*, or *else-part* otherwise. The *condition* is regarded as true if it expands to a non-empty string (leading and trailing spaces are stripped **before** it is evaluated). Note that if condition expands to one or more spaces it will be regarded as being true – use the *strip* function if this is likely to cause a problem.

The *else-part* may be omitted in which case the function returns an empty string if *condition* is false.

`$(join list1, list2)`

Concatenates the two arguments by concatenating the corresponding word from each list to form the result, i.e. the two first words (one from each argument) form the first word of the result, the two second words form the second word of the result. If one list has more words than the other, the extra words are copied into the result.

For example:

```
$(join a b,.c .o)
```

returns a.c b.o. The join function can be used to merge the results of the *dir* and *notdir* functions, to produce the original list of files which was given to those two functions.

`$(notdir names...)`

Extracts the leafname of each filename in *names*, that is everything after the last dot in each filename. If the file name contains no directory name, then the leafname is returned unmodified. A file name that ends with a dot becomes an empty string which can cause problems as the resulting list may not have the same number of elements as *names*.

For example:

```
$(notdir $(dir ADFS:.$.src.main/c))
```

produces the result `'main/c'`.

`$(origin variable)`

The origin function returns a string describing the origin of the variable – it does not look at the value of the variable. The variable should not normally have a `$` prefix or parentheses as the function is examining the status of the variable itself, not its value.

The result is a string, as follows:

undefined	variable has not been defined
default	variable has a default definition
environment	variable was defined as an environment variable and <code>-e</code> option has not been used
environment	variable was defined as an environment variable and the <code>-e</code> option is turned on (see section Summary of Options)
file	variable was defined in a makefile
command line	variable was defined on the command line
override	variable was defined with an override directive in a makefile (see section <i>Override</i> on page 221)
automatic	variable is an automatic variable defined for the execution of the commands for each rule (<code>\$@</code> , <code>\$*</code> , <code>\$<</code> or <code>\$?</code>).

`$(patsubst pattern, replacement, text)`

Returns the *text* string after replacing any words of *pattern* with *replacement*. Words in *text* are separated by one or more spaces.

Wildcards are allowed in *pattern* as follows:

- `%` matches any non-space characters within a word. If `%` appears in the replacement string it is replaced by the text that matched `%` in *pattern*.
- `\%` Matches a percent character: `%`
- `\\` Matches a backslash character if the next character is `%`

Multiple spaces between words in *text* are replaced with a single space and leading and trailing space is discarded.

For example,

```
$(patsubst %.c, %.o, foo.c bar.c)
```

produces the value 'foo.o bar.o'.

A short-cut *substitution operator* may be used to achieve the same effect as follows:

```
$(var:pattern=replacement)
```

This is equivalent to:

```
$(patsubst pattern, replacement, $(var))
```

For example, to obtain the list of source files corresponding to a list of object files:

```
objects = foo.o bar.o baz.o
$(objects:.o=.c)
```

\$(sort *list*)

Sorts the words of *list* in lexical order, removing duplicate words. The output is a list of words separated by single spaces. For example:

```
$(sort one two three)
```

will expand to 'one three two'.

\$(subst *from*, *to*, *text*)

Returns the string *text* with each occurrence of *from* replaced with *to*. For example,

```
$(subst /, ./, /pskirrow/src/main)
```

returns 'pskirrow.src.main'.

\$(strip *string*)

Replaces multiple spaces in string with a single space and removes all leading and trailing spaces. This can be useful when using conditionals which will not equate spaces as being the same as a null string.

\$(warning *text*...)

This function generates a warning but allows AMU to continue processing the Makefile. The warning *text* is treated in the same way as for the `error` function above. This function returns an empty string.

\$(wildcard *filename*)

Returns filename if the file exists or nothing otherwise, but note that wildcards are not allowed. Note that this differs from GNU make where this function may be used to expand a wildcard to a list of matching files.

`$(word n, text)`

Returns the *n*th word that occurs in *text*, with *n*=1 representing the first word. If *n* is bigger than the number of words in *text*, the value is empty. For example,

```
$(word 3, RISC OS Open Limited)
```

returns 'Open'.

`$(wordlist s, e, text)`

This function returns the sub-list of words in *text* starting with word *s* and ending with word *e* (inclusive) counting the first word as word 1. If the range exceeds the words in the list, then the missing words are regarded as empty. If *s* is larger than *e* or greater than the number of words in the list then nothing is returned.

For example,

```
$(wordlist 2, 3, one two three)
```

returns 'two three'.

`$(words text)`

Returns the number of words in *text*. For example, to extract the last word from *text*:

```
$(word $(words text), text)
```

The implementation of wildcard does not support wildcards but it can be used to test for existence of a specific file or list of files though.

Directives

Directives are special keywords which are placed at the start of the line. They may optionally be preceded by a '*.*'. The support directives are listed below.

Override

Macro assignment override is supported with the `override` directive. This allows macro definitions made on the command-line, which are otherwise immutable, to be overridden (also see Macros and Macro Priority sections below).

For example:

```
override macro = value
```

defines *macro* to be a recursively expanded macro set to *value*.

```
override macro := value
```

defines *macro* to be a simply expanded macro set to *value*.

```
override variable += text
```

Appends text to the end of the current definition of macro.

Include

If a line of the Makefile starts with the word `include` (or `.include`), then the parameters following it are taken to be filenames whose contents should be logically inserted in the Makefile at that point (i.e.. just like the C preprocessor's `#include`). This means that common makefile fragments can be abstracted out of individual component makefiles, thus simplifying and standardising the build structure.

Preceding `include/.include` with a hyphen suppresses any file not found errors from arguments to the include directive.

For example:

```
include makebase
```

Conditional Directives

There are six conditional directives to control which parts of the makefile are parsed:

```
ifeq  ifneq  ifdef  ifndef  
else  endif
```

The last two must appear on lines on their own. The first four introduce the conditional section of the makefile. Conditionals are evaluated as the makefile is parsed, not when the rules are interpreted, so you can use it to alter what AMU sees in the makefile. If the condition evaluates to a non-empty value then the text after the conditional is used, otherwise the text after the matching `else` directive is used (if it exists). Beware of values containing just spaces which are not regarded as being empty.

Conditionals may be nested to 8 levels in AMU (other versions of make may allow more or fewer levels).

The `ifeq` and `ifneq` directives take two parameters. They make take several styles of parameter, but they are all equivalent:

```
ifeq (param1,param2)  
ifeq "param1" "param2"  
ifeq 'param1' 'param2'  
ifeq 'param1' "param2"
```

For example:

```
ifeq (${MAKECMDGOALS},clean)
    this text is inserted if the macro expands to "clean"
else
    this text is inserted if the macro doesn't match
endif
```

The else clause is optional.

`ifeq` compares the two parameters for equality. `ifneq` compares the two parameters for inequality. `ifdef` and `ifndef` take a single parameters which is the name of a macro. `ifdef` returns true if the specified name names a macro with a non-empty value, and false otherwise. `ifndef` returns the opposite.

The fact than a macro defined with an empty value is treated as undefined by this directive can be the cause of confusion. If you want to distinguish between undefined macros and those with an empty value, you can instead use:

```
ifeq ($(origin macroname),undefined)
    then the macro was undefined
endif
```

Macro priority

Macro definitions do not always take effect in this version of AMU. The macro priority system mirrors that used by GNU make. Each macro has an origin (which is returned by the `origin` function) as does each attempted definition. If the existing definition is higher priority, the definition does **not** succeed. This can lead to odd-looking, but correct, behaviour. By default, the rank (from highest to lowest priority) is:

- override
- command line
- environment override
- file
- environment
- automatic
- default
- undefined

An important side-effect of this is macros defined on the command-line now **outrank** those defined in the makefile. This can cause unexpected behavioural changes to the unwary. If the Makefile really wants to override the command-line definition, it must use the `override` directive. Note that `+=` is affected by this protection too - you cannot add to a command line macro unless you use the `override` directive.

To aid compatibility with makefiles written for earlier versions of AMU, a new command-line option `-E` is supported. This changes the ranking system to more closely mirror that employed by AMU 5.06. However, using this option does not prevent the environment being searched for undefined macros. This alternative rank is:

- override
- file
- command line
- automatic
- default
- environment override
- environment
- undefined.

Aside: There is a peculiar behaviour of GNU make which is mirrored by AMU with regard to environment override macros. If `-e` is in effect (environment overrides Makefile), then the origin of a macro that has come from the environment will be just “environment” until another definition is attempted in the makefile. At that point, the macro's origin is boosted up to “environment override”, but not before. This is not a bug.

File naming

To help you move Windows and UNIX Makefiles to RISC OS, or to develop Makefiles under RISC OS for export to Windows or UNIX, both AMU and the C compiler accept three styles of file naming:

RISC OS native:	<code>\$.301.cfe.c.pp</code>	<code>^.include.h.defs</code>
UNIX-like:	<code>/301/cfe/pp.c</code>	<code>../include/defs.h</code>
Windows-like:	<code>\301\cfe\pp.c</code>	<code>..\include\defs.h</code>

(All three of these examples refer to the same two RISC OS files.)

The linker offers more limited support; in essence, it recognises `thing.o` and `o.thing` as referring to the same RISC OS file (`o.thing`). In practice, object files almost always live locally (that's the only place the RISC OS and UNIX C compilers will put one) so this support is fairly complete.

AMU will even accept a mixture of naming styles, though this practice should be discouraged.

The mapping between different naming styles cannot be complete (consider the UNIX analogue of `ads:0$.Library` or `net#1.251:src.amu`). However, it is usually sufficient to take much of the hard work out of moving reasonably portable Makefiles.

VPATH

Usually, AMU looks for files relative to the work directory or in places implicit in the filename. The example given earlier contains the line:

```
amu: amu.o $.301.clx.o.clxlib
```

which refers to:

```
@.o.amu (in @.o) and $.301.clx.o.clxlib (in $.301.clx.o)
```

Sometimes, particularly when dealing with multiple versions of large systems, it is convenient to have a complete set of object files locally, a few sources locally, but most sources in a central place shared between versions. For example, we can build different versions of the C compiler this way. If the macro VPATH is defined, then AMU will look in the list of places defined in it for any files it can't find in the places implicit in their names. For example, we might have compiler sources in `somewhere.arm`, `somewhere.mip`, `somewhere.cfe` and put the compiler Makefile in `somewhere.ccriscos`. It might contain the following VPATH definition:

```
VPATH=^.arm ^.mip ^.cfe# note that UNIX VPATHs
# separate path elements
# with colons, not spaces
```

and then dependency lines like:

```
o.pp: c.pp # ^.cfe.c.pp, via VPATH
cc $(ccflags) -o o.pp $?

o.cg: c.cg # ^.mip.c.cg, via VPATH
cc $(ccflags) -o o.cg $?
```

Rule patterns, .SUFFIXES, \$@, \$*, \$< and \$?

All the examples given so far have been written out longhand, with explicit rules for making targets. In fact, AMU can make inferences if you supply the appropriate rule patterns. These are specified using special target names consisting of the concatenation of two suffixes from the pseudo-dependency `.SUFFIXES`. This sounds very complicated, but is actually quite simple. For example:

```
.SUFFIXES: .o .c
amu:      o.amu ...
.c.o:;    $(CC) $(CFLAGS) -o $@ c.$*
```

(Note the order here: `.c.o` makes a `.o`-like thing from a `.c`-like thing).

The rule pattern `.c.o` describes how to make `.o`-like things from `.c`-like things. If, as in the above fragment, there is no explicit entry describing how to make a `.o`-like thing (`o.amu`, in the above example) AMU will apply the first rule it has for making `.o`-like things. Here, order is determined by order in the `.SUFFIXES` pseudo-dependency. For example, suppose `.SUFFIXES` were defined as `.o .c .f` and that there were two rules, `.c.o:...` and `.f.o:...`. Then AMU would choose the `.c.o` rule because `.c` precedes `.f` in the `.SUFFIXES` dependency. In applying the `.c.o` rule, AMU infers a dependence on the corresponding `.c`-like thing - here `c.amu`. So, in effect, it infers:

```
o.amu:    c.amu
          $(CC) $(CFLAGS) -o o.amu c.amu
```

Note that, in the commands, `$@` is replaced by the name of the target and `$*` by the name of the target with the 'extension' deleted from it. In a similar fashion, `$<` refers to the list of inferred prerequisites. So the above example could be rewritten using the rule:

```
.c.o:;    $(CC) $(CFLAGS) -o $@ $<
```

However, if a `VPATH` were being used, this second form is obligatory. Consider, for example, the fragment:

```
VPATH=^.arm ^.mip ^.cfe
cc:      .... o.pp ....
.c.o:;    $(CC) $(CFLAGS) -o $@ $<
```

There is no explicit rule for making `o.pp`, so AMU will apply the rule pattern `.c.o:?`. This might expand to:

```
o.pp:     ^.cfe.c.pp
          $(CC) $(CFLAGS) -o o.pp ^.cfe.c.pp
```

which has a much more useful effect than:

```
$(CC) $(CFLAGS) -o o.pp c.pp
```

Finally, `$?` can be used in any command to stand for the list of prerequisites with respect to which the target is out of date (which may be only some of the prerequisites).

Use of ::

If you use `::` to separate targets from prerequisites, rather than `:`, the right-hand sides of dependencies which refer to the same targets are not merged. Furthermore, each such dependency can have separate commands associated with it. Consider, for example:

```
o.t1:: c.t1 h.t1
      cc -g -c c.t1 # executed if o.t1 is out of
      # date wrt c.t1 or h.t1
o.t1:: c.t1 h.t2
      cc -c c.t1 # executed if o.t1 is out of
      # date wrt c.t1 or h.t2
```

These features are used extensively by Make in the construction of Makefiles.

Prefix\$Dir

The DDEUtils module provides an environment variable `Prefix$Dir` set to the work directory. This is provided to allow you to execute binaries placed in the work directory.

Makefiles constructed by Make

A Makefile constructed by Make, i.e. used to maintain a project, is a file of type `0xFE1` (Makefile). This text is arranged into a number of sections which are separated by *active comments*.

When maintaining a project the meta-symbol `@` is used to stand for the pathname of the work directory. This overcomes the problem of a current directory not being appropriate under the RISC OS desktop. If the absolute filename of a Makefile is:

```
adfs::4.$$.any.thing.makefile
```

then all filenames for the project can use `@` to replace `adfs::4.$$.any.thing`.

For example:

```
adfs::4.$$.any.thing.c.foo
```

becomes denoted by

```
@.c.foo
```

AMU is invoked with the `-desktop` flag to indicate that `@` should be expanded.

Tools like `cc` and `objasm` which must produce dependency information are invoked with a flag `-depend !Depend`. AMU's parser for `!Depend` files generated by tools it invokes is much stricter than that for makefiles in general. Only one target is permitted per line (though multiple dependencies may be listed) and no whitespace is allowed between the target name and the following `'`.

Below, we describe each of the Makefile sections, beginning with their corresponding active comments:

# Project: project_name	This gives a name to be used for the project in the Open submenu.
# Toolflags:	This section has a set of default flags for each of the tools which have registered themselves with !Make, for automatic inclusion in a Makefile. Each rule would be of the type: <code>toolFLAGS =</code>
# Final targets:	This section contains the rules for making the final targets of the project. For example: <code>!RunImage: link \$(linkflags) -o !RunImage -via objects</code>
# User-editable dependencies:	This section is left untouched by !Make, and can freely be edited by the user using a text editor.
# Static dependencies:	This section contains rules for making an object file from its corresponding source. It does not refer to include files and the like (described below in the section <code>Dynamic dependencies</code>).
# Dynamic dependencies:	This section contains the rules which are created by !Make by running the relevant tool on a source file to ascertain its dependencies (e.g. <code>cc -depend</code>).

Miscellaneous features

The special pseudo-target `.SILENT` tells AMU not to echo commands to be executed to your screen. Its effect is as if you used the Make or AMU option **Silent**.

The special pseudo-target `.IGNORE` tells AMU to ignore the return code from the commands it executes. Its effect is as if you used the Make or AMU option **Ignore return codes**.

A command line in a Makefile, the first non-white-space character of which is `@`, is locally silent; just that command is not echoed. This is only rarely useful.

A command line, the first non-white-space character of which is `-` has its return code ignored when it is executed. This is extremely useful in Makefiles which use commands such as `diff` which cannot set the return code conventionally.

The special macro `MFLAGS` is given the value of the command line arguments passed to AMU. This is most useful when a Makefile itself contains AMU commands (for example, when a system consists of a collection of subsystems,

each described by its own Makefile). `MFLAGS` allows the same command line arguments to be passed to every invocation of AMU, even the recursive ones. For example, you might invoke AMU like this:

```
* amu -k LIB=$.experiment.new.lib.grafix
```

and the Makefile might contain entries like:

```
subsys_1:$(COMMON) $(HDRS1) ...
    dir subsys1
    amu $(MFLAGS)
    back
```

Appendix C: FrontEnd protocols

Star Commands

Two star commands are supported:

```
*FrontEnd_Start    -app <application name>
                   -desc <description_filename>

*FrontEnd_SetUp    -app <application_name>
                   -desc <description_filename>
                   -task <task-id_of_caller>
                   -handle <app-specific_handle>
                   -toolflags <filename>
```

The application specific handle can be used by the caller to identify return messages, if many *FrontEnd_SetUp commands have been made.

EBNF Grammar of Description Format

The following is an EBNF grammar for an application description:

Note: Blank lines and characters following # (up to newline) are ignored.

```
APPLICATION ::= TOOLDETAILS
              [METAOPTIONS]
              [FILEOUTPUT]
              [DBOX]
              [MENU]
              [DESELECTIONS]
              [EXCLUSIONS]
              [MAKE_EXCLUSIONS]
              [ORDER]
              [MAKE_ORDER]
              <EOF>

TOOLDETAILS ::= tool_details_start
              name <string> ";"
              [command_is <string>;]
              version <number_and_optional_date>
              ";"
              [filetype &<3digit_hexnumber> ";"]
```

```

        [wimpslot <integer>k ";" ]
        [has_extended_cmdline ";" ]
    tool_details_end

    METAOPTIONS ::= metaoptions_start
        [has_auto_run [on] ";" ]
        [has_auto_save [on]
            {"^."}[<string>][leafname]
        [<string>] from icn <integer> ";" ]
        [has_text_window ";" ]
        [has_summary_window ";" ]
        [display_dft_is text|summary ";" ]
    metaoptions_end

    FILEOUTPUT ::= fileoutput_start
        [output_option_is <string> ";" ]
        [output_dft_string <string> ";" ]
        [output_dft_is (produces_output|
            produces_no_output) ";" ]
    fileoutput_end

    DBOX ::= dbox_start
        ICONS
        [ICONDEFAULTS]
        [IMPORTS]
    dbox_end

    MENU ::= menu_start
        MENULIST
        [MENUDEFAULTS]
    menu_end

#-----

    MENULIST ::= { MENUENTRY }

    MENUENTRY ::= <string> maps_to <string>
        [sub_menu <string> <integer>
            [prefix_by <string>]]
        [produces_no_output|
            produces_output]
        [not_saved] ";"

    MENUDEFAULTS ::= defaults
        menu <integer> on | off [sub_menu
            <string>
            | <integer>

```



```

{ "," menu <integer> on | off [sub_menu
  <string>
  | <integer>
}
";"
[make_defaults
menu <integer> on | off [sub_menu
  <string>
  | <integer>
{
  ","
  menu <integer> on | off [sub_menu
  <string>
  | <integer>
}
";"
]

#-----
ICONLIST      ::= icn <integer> { "," icn <integer> }
ENTRYLIST     ::= menu <integer> { "," menu <integer> }
ICON_ENTRYLIST ::= menu | icn <integer> { "," menu | icn
  <integer> }

#-----
ICONS          ::= icons_start
                  ICONDEFLIST
                  icons_end
ICONDEFLIST    ::= { ICONDEF }
ICONDEF        ::= icn <integer> ( maps_to ([<string>]
                                      [CONVERSION])
                                [prefix_by <string>]
                                [followed_by [spaces] OPTLIST]
                                [separator_is <string>]
                                [produces_no_output
                                | produces_output]
                                [not_saved] )
                                | (increases | decreases icn
                                   <integer>
                                [by] <integer> [max <integer>]
                                   [min <integer>] ) )

```

```

| inserts <string> ";"
| extends from icn <integer>
  to icn <integer> ";"

OPTLIST      ::= OPTENTRY { ", " OPTENTRY }
OPTENTRY     ::= icn <integer>
CONVERSION   ::= string|number
ICONDEFAULTS ::= defaults
               icn <integer> on | off | <string>
               | <integer>
               { ", " icn <integer> on | off
               <string> | <integer>
               }
               ";"
               [make_defaults
               icn <integer> on | off | <string>
               | <integer>
               { ", " icn <integer> on | off
               <string> | <integer> }
               ";"
               ]

#-----

DESELECTIONS ::= deselections_start
               DESELECTIONLIST
               deselections_end

DESELECTIONLIST::={ DESELECT }

DESELECT     ::= icn <integer> deselects
               ICON_ENTRYLIST ";"
               | menu <integer> deselects
               ICON_ENTRYLIST ";"

#-----

EXCLUSIONS ::= exclusions_start
               EXCLUSIONLIST
               exclusions_end

EXCLUSIONLIST::= { EXCLUDE }

```

```

EXCLUDE      ::= icn <integer> excludes
                ICON_ENTRYLIST ";"
                | menu <integer> excludes
                ICON_ENTRYLIST ";"

#-----

MAKE_EXCLUSIONS ::= make_excludes ICON_ENTRYLIST ";"

ORDER          ::= order_is
                (menu|icn <integer>) | <string> |
                output
                { ",", (menu|icn <integer>) |
                <string> | output}
                ";"

MAKE_ORDER     ::= make_order_is
                (menu|icn <integer>) | <string> |
                output
                { ",", (menu|icn <integer>) |
                <string> | output}
                ";"

#-----

IMPORTS        ::= imports_start
                [wild_card_is <string> ";"]
                IMPORTLIST
                imports_end

IMPORTLIST     ::= { IMPORT }

IMPORT         ::= drag_to
                (icn <integer>|any|iconbar)
                inserts
                ICONLIST
                [separator_is <string>] ";"

```

WIMP Message returned after a *FrontEnd_SetUp

When an application like Make does a *FrontEnd_SetUp command, the FrontEnd module replies to that application when the user has chosen his options with a WIMP message of the format:

Byte offset	Contents
+16	reason code 0x00081400
+20	handle which was passed to *FrontEnd_SetUp
+24 to +36	application name
+36 ...	null-terminated command-line options

Appendix D: DDEUtils

The DDEUtils module performs three functions. These functions have been combined in one module for convenience:

- **Filename prefixing.** This allows a unique current working directory to be set for each task running under RISC OS.
- **Long command lines.** A mechanism for passing long command lines (> 255 characters) between programs (e.g. between AMU and Link).
- **Throwback.** Throwback allows a language processor (e.g. CC or ObjAsm) to inform an editor that an error has occurred while processing a source file. The editor can then display the source file at the location of the error.

These functions are described individually in the rest of the chapter.

Filename prefixing SWIs

DDEUtils_Prefix (0x42580)

Entry: R0 = Pointer to 0 terminated directory name, or R0 = 0
Exit: All registers preserved
Error: None
Use: This sets a directory name to be prefixed to all relative filenames used by this task. If R0 = 0 this removes any previously set prefix. If you use this SWI within a program to set a directory prefix you should call it again with R0 = 0 immediately before exiting your program.

DDEUtils_ReadPrefix (0x4258A)

Entry: R0 = Task handle, or R0 = 0 for current task
Exit: R0 = Pointer to control terminated directory name, or 0 if none.
Error: None
Use: This returns the directory name that will be prefixed to all relative filenames used by the task whose handle is given in R0, or by the current task if R0 = 0. If no prefix is set for the task 0 is returned.

Filename prefixing *Commands

`*Prefix [directory]`

This sets the specified directory name to be prefixed to all relative filenames used by this task. `*Prefix` with no arguments removes any previously set prefix.

The system variable `<Prefix$Dir>` is set to the prefix used for the currently executing task. This can be set by you, and this will have the same effect as `*Prefix`.

Long command line SWIs

These SWIs are used to pass long command lines between programs. Typically they will be called by library veneers. For example, the C run-time library initialisation calls `DDEUtils_GetCLSize` and `DDEUtils_GetCL` to fetch any long command lines set up by a calling program and calls `DDEUtils_SetCLSize` and `DDEUtils_SetCL` in the system library call.

`DDEUtils_SetCLSize (0x42581)`

Entry: R0 = Length of command line buffer required

Exit: R0 destroyed

Error: None

Use: This SWI should be called by a program when it has a long command line which it wishes to pass to another program. The SWI should be called with the length of the command line in R0. A buffer of suitable size is allocated in the RMA.

`DDEUtils_SetCL (0x42582)`

Entry: R0 = Pointer to zero terminated command line tail

Exit: All registers preserved

Error: Possible errors are

`CLI buffer not set`

This error is generated if the program has not previously called `DDEUtils_SetCLSize` to establish the size of the command line.

Use: This should be called after calling `DDEUtils_SetCLSize` to set the size of the command line buffer. R0 contains a pointer to the command tail (i.e. the command line without the name of the program to be run).

DDEUtils_GetCLSize (0x42583)

Entry: –
Exit: R0 = Size of command line
Error: None
Use: This is called by a program which may have been run with a long command line. The size of the command line is returned in R0. 0 is returned if no command line has been set.

DDEUtils_GetCl (0x42584)

Entry: R0 = Pointer to buffer to receive command line
Exit: All registers preserved
Error: None
Use: This SWI is called to fetch the command line. The command line is copied into the buffer pointed to by R0.

DDEUtils_FlushCL (0x4258B)

Entry: –
Exit: All registers preserved
Error: None
Use: This is called to clear the command line buffer and free the allocated memory.

Throwback SWIs

DDEUtils_ThrowbackRegister (0x42585)

Entry: R0 = task handle of caller
Exit: All registers preserved
Error: Possible errors are:

 Another task is registered for throwback
 Throwback not available outside the desktop
Use: This registers a task which is capable of dealing with throwback messages, with the throwback module. The task handle will be used in passing Wimp messages to the caller, when they are generated by an application.

DDEUtils_ThrowbackUnRegister (0x42586)

Entry: R0 = task handle of caller

Exit: All registers preserved

Error: Possible errors are:

Task not registered for throwback
Throwback not available outside the desktop

Use: This call should be made when the Wimp task which registered itself for throwback is about to exit.

DDEUtils_ThrowbackStart (0x42587)

Entry: -

Exit: All registers preserved

Error: Possible errors are:

No task registered for throwback
Throwback not available outside the desktop

Use: When a non-desktop tool detects errors in the source(s) it is processing, and throwback is enabled, the tool should make this SWI to start a throwback session.

DDEUtils_ThrowbackSend (0x42588)

Entry: R0 = reason code
R2-R5 = depends on reason code (see below)

If R0 = 0 (Throwback_ReasonProcessing)
R2 = pointer to nul-terminated full pathname of file being processed

If R0 = 1 (Throwback_ReasonErrorDetails)
R2 = pointer to nul-terminated full pathname of file being processed
R3 = line number of error
R4 = severity of error
= 0 for warning
= 1 for error
= 2 for serious error
R5 = pointer to nul-terminated description of error

If R0 = 2 (Throwback_ReasonInfoDetails)
R2 = pointer to nul-terminated full pathname of file being processed

R3 = line number to which 'informational' message refers
 R4 = must be 0
 R5 = pointer to nul-terminated 'informational' message

Exit: R0-R4 preserved

Error: Possible errors are:

No task registered for throwback
 Throwback not available outside the desktop

Use: This SWI should be called with reason

Throwback_ReasonProcessing

once, when the first error in processing a file was found. Then it should be called once for each error found, with the reason

Throwback_ReasonErrorDetails

or for each informational line that needs displaying with the reason:

Throwback_ReasonInfoDetails

DDEUtils_ThrowbackEnd (0x42589)

Exit: All registers preserved

Error: Possible errors are:

No task registered for throwback
 Throwback not available outside the desktop

Throwback WIMP messages

These messages are sent by the DDEUtils module to an editor that has registered itself for throwback using the SWI DDEUtils_ThrowbackRegister. You only need to know about them if you want to write your own editor.

Byte Offset	Contents
+16	DDEUtils_ThrowbackStart (0x42580)

The translator then passes messages giving full information on each error, or each 'informational' message, to the editor.

A complete series of messages sent by the translator to the editor is described by the grammar below. Items in <..> are individual Wimp messages, identified by their reason code.

```
ErrorDialogue ::=          <DDEUtils_ThrowbackStart>
                          ErrorsWhileProcessing
                          {ErrorsWhileProcessing}
                          <DDEUtils_ThrowbackEnd>

ErrorsWhileProcessing ::=  <DDEUtils_ProcessingFile>
                          Error Found In {Error Found
                          In}

ErrorFoundIn ::=          <DDEUtils_ErrorIn>
                          <DDEUtils_ErrorDetails>

InfoDialogue ::=          <DDEUtils_ThrowbackStart>
                          InfoDetails{InfoDetails}
                          <DDEUtils_ThrowbackEnd>

InfoDetails ::=           <DDEUtils_InfoforFile>
                          <DDEUtils_InfoDetails>
```

The format of such Wimp messages is as follows:

Byte Offset	Contents
+16	DDEUtils_ProcessingFile (0x42581)
+20	Nul-terminated filename
Byte Offset	Contents
+16	DDEUtils_ErrorsIn (0x42582)
+20	Nul-terminated filename
Byte Offset	Contents
+16	DDEUtils_ErrorDetails (0x42583)
+20	Line number
+28	Severity = 0 for warning = 1 for error = 2 for serious error
+32	Nul-terminated description
Byte Offset	Contents
+16	DDEUtils_ThrowbackEnd (0x42584)
Byte Offset	Contents
+16	DDEUtils_InfoforFile (0x42585)
+20	Nul-terminated filename
Byte Offset	Contents
+16	DDEUtils_InfoDetails (0x42586)
+20	Line number
+28	must be 0
+32	Nul-terminated 'informational' message

Appendix E: SrcEdit file formats

Language File Format

language_name

searchpath is a comma-separated list of full pathnames for default search path when loading from a selection. Note that each item in this list should either be a path variable (e.g. C:), or be terminated by a dot (this line can be left blank, though putting @. on the line would be preferable)

helppath is the full pathname of language help file (this line can be left blank, though putting @. on the line would be preferable)

Help File Format

%<keyword>

<line 1 of help text>

<line 2 of help text>

<line 3 of help text>

<line 4 of help text>

etc

There is no limit on the number of help lines for a given keyword.



Appendix F: Code file formats

This appendix defines three file formats used by the Desktop tools to store processed code and the format of debugging data used by DDT:

- AOF – ARM Object Format
- ALF – Acorn Library Format
- AIF – ARM Image Format
- ASD – ARM Symbolic Debugging Format.

Desktop tools language processors such as CC and ObjAsm generate processed code output as AOF files. An ALF file is a collection of AOF files constructed from a set of AOF files by the LibFile tool. The Link tool accepts a set of AOF and ALF files as input, and by default produces an executable program file as output in AIF.

Terminology

Throughout this appendix the terms *byte*, *half word*, *word*, and *string* are used to mean the following:

Byte: 8 bits, considered unsigned unless otherwise stated, usually used to store flag bits or characters.

Half word: 16 bits, or 2 bytes, usually unsigned. The address of a half word (i.e. of its least significant byte) must be divisible by 2.

Word: 32 bits, or 4 bytes, usually used to store a non-negative value. The address of a word (i.e. of its least significant byte) must be divisible by 4.

String: A sequence of bytes terminated by a NUL (0x00) byte. The NUL is part of the string but is not counted in the string's length. Strings may be aligned on any byte boundary.

Note: a word consists of 32 bits, 4-byte aligned.

Byte Sex or Endian-ness

There are two sorts of AOF or ALF: little-endian and big-endian.

In little-endian AOF or ALF, the least significant byte of a word or half-word has the lowest address of any byte in the (half-)word.

In big-endian AOF or ALF, the most significant byte of a (half-)word has the lowest address.

For data in a file, *address* means 'offset from the start of the file'.

There is no guarantee that the endian-ness of an AOF or ALF file will be the same as the endian-ness of the system used to process it (the endian-ness of the file is always the same as the endian-ness of the target ARM system).

The two sorts of AOF or ALF cannot, be mixed (the target system cannot have mixed endian-ness: it must have one or the other). Thus the ARM linker will accept inputs of either sex and produce an output of the same sex, but will reject inputs of mixed endian-ness.

Alignment

Strings and bytes may be aligned on any byte boundary.

AOF and ALF fields defined in this appendix make no use of half-words and align words on 4-byte boundaries.

Within the contents of an AOF or ALF file the alignment of words and half-words is defined by the use to which AOF or ALF is being put.

For all current ARM-based systems, words are aligned on 4-byte boundaries and half-words on 2-byte boundaries.

Undefined fields

Fields not explicitly defined by this appendix are implicitly reserved. It is required that all such fields be zeroed. Meaning may be ascribed to such fields at any time, but this will usually be done in a manner which gives no new meaning to zeroes.

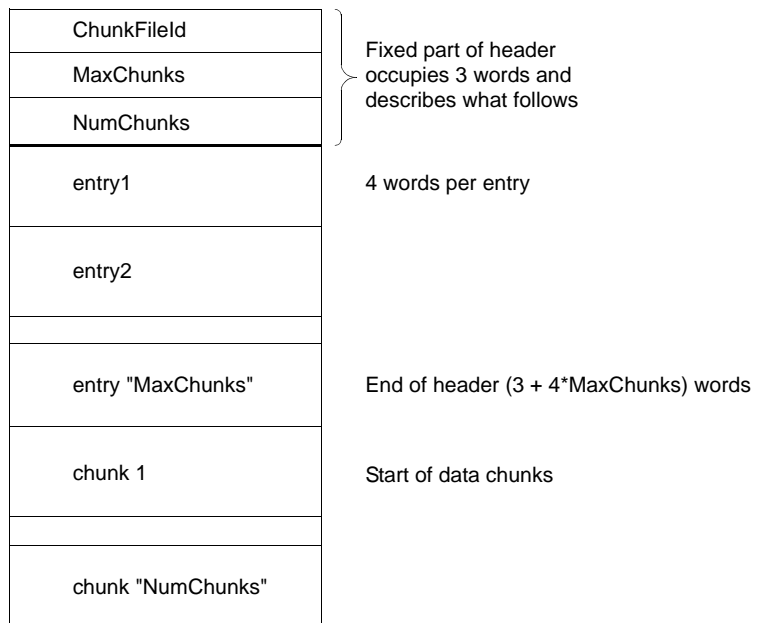
AOF

ARM object format files are output by language processors such as CC and ObjAsm.

Chunk file format

A chunk is accessed via a header at the start of the file. The header contains the number, size, location and identity of each chunk in the file. The size of the header may vary between different chunk files but is fixed for each file. Not all entries in a header need be used, thus limited expansion of the number of chunks is permitted without a wholesale copy. A chunk file can be copied without knowledge of the contents of the individual chunks.

Graphically, the layout of a chunk file is as follows:



ChunkFileId marks the file as a chunk file. Its value is 0xC3CBC6C5. The endianness of the chunk file can be deduced from this value (if, when read as a word, it appears to be 0xC5C6CBC3 then each word value must be byte-reversed before use).

The `MaxChunks` field defines the number of the entries in the header, fixed when the file is created. The `NumChunks` field defines how many chunks are currently used in the file, which can vary from 0 to `MaxChunks`. The value of `NumChunks` is redundant as it can be found by scanning the entries.

Each entry in the header comprises four words in the following order:

<code>chunkId</code>	is an 8-byte field identifying what data the chunk contains (note that this is an 8-byte field, not a 2-word field, so it has the same byte order independent of endian-ness).
<code>fileOffset</code>	is a one word field defining the byte offset within the file of the start of the chunk. All chunks are word-aligned, so it must be divisible by four. A value of zero indicates that the chunk entry is unused.
<code>size</code>	a one word field defining the exact byte size of the chunk (which need not be a multiple of four).

The `chunkId` field provides a conventional way of identifying what type of data a chunk contains. It is split into two parts. The first four characters contain a unique name allocated by a central authority (ARM). The remaining four characters can be used to identify component chunks within this domain. The 8 characters are stored in ascending address order, as if they formed part of a NUL-terminated string (which they do not), independently of endian-ness.

For AOF files, the first part of each chunk's name is `OBJ_`; the second components are defined later in this section.

Object file format

Each piece of an object file is stored in a separate, identifiable, chunk. AOF defines five chunks as follows:

Chunk	Chunk Name
Header	<code>OBJ_HEAD</code>
Areas	<code>OBJ_AREA</code>
Identification	<code>OBJ_IDFN</code>
Symbol Table	<code>OBJ_SYMT</code>
String Table	<code>OBJ_STRT</code>

Only the header and areas chunks must be present, but a typical object file will contain all five of the above chunks.

Each name in an object file is encoded as an offset into the string table, stored in the `OBJ_STRT` chunk (see *String table chunk* (`OBJ_STRT`) on page 266). This allows the variable-length nature of names to be factored out from primary data formats.

A feature of chunk file format is that chunks may appear in any order in the file. However, language processors which must also generate other object formats – such as Unix's `a.out` format – should use this flexibility cautiously.

A language translator or other system utility may add additional chunks to an object file, for example a language-specific symbol table or language-specific debugging data, so it is conventional to allow space in the chunk header for additional chunks; space for eight chunks is conventional when the AOF file is produced by a language processor which generates all five chunks described here.

The header chunk should not be confused with the chunk file's header.

Format of the AOF header chunk

The AOF header is logically in two parts, though these appear contiguously in the header chunk. The first part is of fixed size and describes the contents and nature of the object file. The second part is variable in length (specified in the fixed part) and is a sequence of area declarations defining the code and data areas within the OBJ_AREA chunk.

The AOF header chunk (OBJ_HEAD) has the following format:

Object file type	} 6 words in the fixed part
Version ID	
Number of areas	
Number of symbols	
Entry Area Index	
Entry Offset	
1st Area Header	} 5 words per area header
2nd Area Header	
nth Area Header	(6 + (5*Number of Areas)) words in the AOF header

Object file type

0xC5E2D080 marks the file as being in relocatable object format (the usual output of compilers and assemblers and the usual input to the linker).

The endian-ness of the object code can be deduced from this value and shall be identical to the endian-ness of the containing chunk file.

Version ID

Encodes the version of AOF to which the object file complies: version 1.50 is denoted by decimal 150; version 2.00 by 200; version 3.10 by 310; and this version 3.11 by decimal 311 (0x137).

Number of areas

The code and data of the object file is presented as a number of separate areas, in the OBJ_AREA chunk, each with a name and some attributes (see below). Each area is declared in the (variable-length) part of the header which immediately follows the fixed part. The value of the `Number of Areas` field defines the number of areas in the file and consequently the number of area declarations which follow the fixed part of the header.

Number of symbols

If the object file contains a symbol table chunk OBJ_SYMT, then this field defines the number of symbols in the symbol table.

Entry address area / entry address offset

One of the areas in an object file may be designated as containing the start address of any program which is linked to include the file. If this is the case, the entry address is specified as an `Entry Area Index`, `Entry Offset` pair. `Entry Area Index`, in the range 1 to `Number of Areas`, gives the 1-origin index in the following array of area headers of the area containing the entry point. The entry address is defined to be the base address of this area plus `Entry Offset`.

A value of 0 for `area-index` signifies that no program entry address is defined by this AOF file.

Format of area headers

The area headers follow the fixed part of the AOF header. Each area header has the following form:

Area name	(offset into string variable)
Attributes + Alignment	
Area size	
Number of relocations	
Base address or zero	5 words in total

Area name

Each area within an object file must be given a name which is unique amongst all the areas in the file. `Area Name` gives the offset of that name in the string table (stored in the OBJ_STRT chunk – see *String table chunk* (OBJ_STRT) on page 266).

Area size

This field gives the size of the area in bytes, which must be a multiple of 4. Unless the `Uninitialised` bit (bit 4) is set in the area attributes (see *Attributes and Alignment* on page 253), there must be this number of bytes for this area in the OBJ_AREA chunk. If the `Uninitialised` bit is set, then there shall be no initialising bytes for this area in the OBJ_AREA chunk.

Number of relocations

This word specifies the number of relocation directives which apply to this area, (equivalently: the number of relocation records following the area's contents in the OBJ_AREA chunk – see *Format of the areas chunk* on page 259).

Attributes and Alignment

Each area has a set of attributes encoded in the most-significant 24 bits of the `Attributes + Alignment` word. The least-significant 8 bits of this word encode the alignment of the start of the area as a power of 2 and shall have a value between 2 and 32 (this value denotes that the area should start at an address divisible by $2^{\text{alignment}}$).

The linker orders areas in a generated image first by attributes, then by the (case-significant) lexicographic order of area names, then by position of the containing object module in the link list. The position in the link list of an object module loaded from a library is not predictable.

The precise significance to the linker of area attributes depends on the output being generated.

Bit 8

Bit 8 encodes the `absolute` attribute and denotes that the area must be placed at its Base Address. This bit is not usually set by language processors.

Bit 9

Bit 9 encodes the `code` attribute: if set the area contains code; otherwise it contains data.

Bits 10 and 11

Bits 10, 11 encode the `common block definition` and `common block reference` attributes, respectively.

Bit 10 specifies that the area is a `common block definition`.

Bit 11 defines the area to be a reference to a common block, and precludes the area having initialising data (see Bit 12, below). In effect, bit 11 implies bit 12.

If both bits 10 and 11 are set, bit 11 is ignored.

Common areas with the same name are overlaid on each other by the linker. The `Area Size` field of a common definition area defines the size of a common block. All other references to this common block must specify a size which is smaller or equal to the definition size. If, in a link step, there is more than one definition of an area with the common definition attribute (area of the given name with bit 10 set), then each of these areas must have exactly the same contents. If there is no definition of a common area, its size will be the size of the largest common reference to it.

Although common areas conventionally hold data, it is quite legal to use bit 10 in conjunction with bit 9 to define a common block containing code. This is most useful for defining a code area which must be generated in several compilation units but which should be included in the final image only once.

Bit 12

Bit 12 encodes the `zero-initialised` attribute, specifying that the area has no initialising data in this object file, and that the area contents are missing from the `OBJ_AREA` chunk. Typically, this attribute is given to large uninitialised data areas. When an uninitialised area is included in an image, the linker either includes a read-write area of binary zeroes of appropriate size, or maps a read-write area of appropriate size that will be zeroed at image start-up time. This attribute is incompatible with the read-only attribute (see Bit 13, below).

Whether or not a zero-initialised area is re-zeroed if the image is re-entered is a property of the relevant image format and/or the system on which it will be executed. The definition of AOF neither requires nor precludes re-zeroing.

To summarise, bits 10, 11 and 12 interact as follows:

12	11	10	Interaction
0	0	1	Initialised common definition
0	1	1	Initialised common definition
0	1	0	Uninitialised reference to common block
1	0	1	Uninitialised reference to common block
1	1	0	Uninitialised reference to common block
1	1	1	Uninitialised reference to common block
1	0	0	Zero-initialised (bss = unnamed common reference)

So, an initialised common definition is inferred if bit 10 is set and bit 11 is not, a Zero-initialised area is inferred if bit 12 is set and both bits 10 and 11 are unset, all other bit combinations infer an uninitialised reference to common block.

Bit 13

Bit 13 encodes the `read_only` attribute and denotes that the area will not be modified following relocation by the linker. The linker groups read-only areas together so that they may be write protected at run-time, hardware permitting. Code areas and debugging tables should have this bit set. The setting of this bit is incompatible with the setting of bit 12.

Bit 14

Bit 14 encodes the `position_independent` (PI) attribute, usually only of significance for code areas. Any reference to a memory address from a PI area must be in the form of a link-time-fixed offset from a base register (e.g. a PC-relative branch offset).

Bit 15

Bit 15 encodes the `debugging_table` attribute and denotes that the area contains symbolic debugging tables. The linker groups these areas together so they can be accessed as a single continuous chunk at or before run-time (usually, a debugger will extract its debugging tables from the image file prior to starting the debuggee).

Usually, debugging tables are read-only and, therefore, have bit 13 set also. In debugging table areas, bit 9 (the code attribute) is ignored.

Bits 16-19 encode additional attributes of code areas and shall be non-0 only if the area has the code attribute (bit 9 set).

Bit 16

Bit 16 encodes the 32-bit PC attribute, and denotes that code in this area complies with a 32-bit variant of the ARM Procedure Call Standard (APCS). Such code may be incompatible with code which complies with a 26-bit variant of the APCS. For details, refer to *32-bit PC vs 26-bit PC* on page 307.

Bit 17

Bit 17 encodes the `reentrant` attribute, and denotes that code in this area complies with a reentrant variant of the ARM Procedure Call Standard.

Bit 18

Bit 18, when set, denotes that code in this area uses the ARM's `extended floating-point instruction set`. Specifically, function entry and exit use the LFM and SFM floating-point save and restore instructions rather than multiple LDFEs and STFES. Code with this attribute may not execute on older ARM-based systems.

Bit 19

Bit 19 encodes the `No Software Stack Check` attribute, denoting that code in this area complies with a variant of the ARM Procedure Call Standard without software stack-limit checking. Such code may be incompatible with code which complies with a limit-checked variant of the APCS.

Bits 20-27 encode additional area attributes and have different meanings for code areas and data areas.

Bit 20

In an area where the code attribute (bit 9) is set, bit 20 indicates that this is a Thumb code area.

For a data area, where bit 9 is unset, bit 20 encodes the `based` attribute, denoting that the area is addressed via link-time-fixed offsets from a base register (encoded in bits 24-27). Based areas have a special role in the construction of shared libraries and ROM-able code, and are treated specially by the linker.

Bit 21

In a code area, bit 21 indicates that this area may contain half word load or store instructions. This bit is set by ObjAsm if you specify the `HALFWORD` attribute when declaring the area, or if you use any instructions from the group `LDRH`, `LDRSH`, `STRH`, `LDREXH`, `STREXH`, `LDAH`, `STLH`, `LDAEXH` or `STLEXH`.

In a data area, bit 21 encodes the `Shared Library Stub Data` attribute. In a link step involving layered shared libraries, there may be several copies of the stub data for any library not at the top level. In other respects, areas with this attribute are treated like data areas with the common definition (bit 10) attribute. Areas which also have the zero initialised attribute (bit 12) are treated much the same as areas with the common reference (bit 11) attribute.

This attribute is not usually set by language processors, but is set only by the linker.

Bit 22

For code areas, bit 22 indicates that the area has been compiled to be suitable for ARM/Thumb interworking. Specifically, any exported functions in this area, whether it be ARM or Thumb code, must be written such that they can return control to a calling function of either instruction set. In data areas this bit shall be set to 0.

Bit 23

Bit 23 is reserved and shall be set to 0.

Bits 24-27

In data areas, bits 24-27 encode the `base register` used to address a based area. If the area does not have the based attribute, or is a code area, then these bits shall be set to 0.

Bits 28-30

Bits 28-30 are reserved and shall be set to 0.

Bit 31

Bit 31 is valid for both code and data areas, and when set, indicates that double-precision floating point data in the area is stored using VFP rules (endianness matches the rest of the file) rather than FPA rules (most-significant word always stored first). For code areas in little-endian mode, it also indicates that when functions in the area receive double-precision arguments and/or return a double-precision value in integer registers or on the stack, the lower register

number or lower stack address holds the least significant word (little-endian VFP compatibility mode). Otherwise (big-endian mode or FPA compatibility mode), the lower register or lower stack address holds the most-significant word.

An area with this attribute cannot be linked with an area which lacks the attribute.

Area Attributes Summary

Bit	Mask	Attribute Description
8	0x00000100	Absolute attribute
9	0x00000200	Code attribute
10	0x00000400	Common block definition
11	0x00000800	Common block reference
12	0x00001000	Uninitialised (0-initialised)
13	0x00002000	Read only
14	0x00004000	Position independent
15	0x00008000	Debugging tables
Code areas only		
16	0x00010000	Complies with the 32-bit APCS
17	0x00020000	Reentrant code
18	0x00040000	Uses extended FP inst set
19	0x00080000	No software stack checking
20	0x00100000	Thumb code (else ARM)
21	0x00200000	Uses half word instructions
22	0x00400000	Has ARM/Thumb interworking
Data areas only		
20	0x00100000	Based area
21	0x00200000	Shared library stub data
24-27	0x0F000000	Base register for based area
Code and data areas		
31	0x80000000	Floating point data uses VFP rules

Format of the areas chunk

The areas chunk (ChunkId of OBJ_AREA) contains the actual areas (code, data, zero- initialised data, debugging data, etc.) plus any associated relocation information. Graphically, an area's layout is:

Area 1
Area 1 relocation
Area n
Area n relocation

An area is simply a sequence of byte values. The endian-ness of the words and half-words within it shall agree with that of the containing AOF file.

An area is followed by its associated table of relocation directives (if any). An area is either completely initialised by the values from the file or is initialised to zero, as specified by bit 12 of its area attributes.

Both the area contents and the table of relocation directives are aligned to 4-byte boundaries.

Relocation directives

A relocation directive describes a value which is computed at link time or load time, but which cannot be fixed when the object module is created.

In the absence of applicable relocation directives, the value of a byte, halfword, word or instruction from the preceding area is exactly the value that will appear in the final image.

A field may be subject to more than one relocation.

Pictorially, a relocation directive looks like:

Offset						
1	II	B	A	R	FT	24-bit SID

Offset

Offset is the byte offset in the preceding area of the subject field to be relocated by a value calculated as described below.

SID (Subject Identification)

The interpretation of the 24-bit SID field depends on the A bit.

If A (bit 27) is 1, the subject field is relocated (as further described below) by the value of the symbol of which SID is the 0-origin index in the symbol table chunk.

If A (bit 27) is 0, the subject field is relocated (as further described below) by the base of the area of which SID is the 0-origin index in the array of areas, (or, equivalently, in the array of area headers).

FT (Field Type)

The 2-bit field type FT (bits 25, 24) describes the subject field:

- 00 the field to be relocated is a byte
- 01 the field to be relocated is a half-word (2 bytes)
- 10 the field to be relocated is a word (4 bytes)
- 11 the field to be relocated is an instruction or instruction sequence

Bytes, halfwords and instructions may only be relocated by values of suitably small size. Overflow is faulted by the linker.

An ARM branch, or branch-with-link instruction is always a suitable subject for a relocation directive of field type instruction.

II (Instruction Instruction)

If the subject field is an instruction sequence (FT = 11), then Offset addresses the first instruction of the sequence and the II field (bits 29 and 30) constrains how many instructions may be modified by this directive:

- 00 no constraint (the linker may modify as many contiguous instructions as it needs to)
- 01 the linker will modify at most 1 instruction
- 10 the linker will modify at most 2 instructions
- 11 the linker will modify at most 3 instructions

R (relocation type)

The way the relocation value is used to modify the subject field is determined by the R (PC-relative) bit, modified by the B (based) bit.

R (bit 26) = 1 and B (bit 28) = 0 specifies PC-relative relocation: to the subject field is added the difference between the relocation value and the base of the area containing the subject field. In pseudo C:

```
subject_field = subject_field + (relocation_value -
                                base_of_area_containing(subject_field))
```

As a special case, if A is 0, and the relocation value is specified as the base of the area containing the subject field, then it is not added and:

```
subject_field = subject_field -
                base_of_area_containing(subject_field)
```

This caters for relocatable PC-relative branches to fixed target addresses.

If R is 1, B is usually 0. If B is 1 this is used to denote that the inter-link-unit value of a branch destination is to be used, rather than the more usual intra-link-unit value (this allows compilers to perform the tail-call optimisation on reentrant code).

R (bit 26) = 0 and B (bit 28) = 0, specifies plain additive relocation: the relocation value is added to the subject field. In pseudo C:

```
subject_field = subject_field + relocation_value
```

R (bit 26) = 0 and B (bit 28) = 1, specifies based area relocation. The relocation value must be an address within a based data area. The subject field is incremented by the difference between this value and the base address of the consolidated based area group (the linker consolidates all areas based on the same base register into a single, contiguous region of the output image). In pseudo C:

```
subject_field = subject_field + (relocation_value -
                                base_of_area_group_containing(relocation_value))
```

For example, when generating reentrant code, the C compiler will place address constants in an adcon area based on register *sb*, and load them using *sb* relative LDRs. At link time, separate adcon areas will be merged and *sb* will no longer point where presumed at compile time. B type relocation of the LDR instructions corrects for this.

Bits 29-31

Bit 31 of the relocation flags word shall be 1, and (unless FT bits are 11) bits 29 and 30 shall be 0.

Format of the symbol table chunk

The `Number of Symbols` field in the fixed part of the AOF header (`OBJ_STRT`) defines how many entries there are in the symbol table. Each symbol table entry has the following format:

Name	} 4 words per entry
Attributes	
Value	
Area name	

Name

This value is an index into the string table (in chunk `OBJ_STRT`) and thus locates the character string representing the symbol.

Value

This is only meaningful if the symbol is a defining occurrence (bit 0 of Attributes set), or a common symbol (bit 6 of Attributes set):

- if the symbol is absolute (bits 0,2 of Attributes set), this field contains the value of the symbol
- if the symbol is a common symbol (bit 6 of Attributes set), this field contains the byte-length of the referenced common area
- otherwise, Value is interpreted as an offset from the base address of the area named by Area Name, which must be an area defined in this object file.

Area Name

is meaningful only if the symbol is a non-absolute defining occurrence (bit 0 of Attributes set, bit 2 unset). In this case it gives the index into the string table for the name of the area in which the symbol is defined (which must be an area in this object file).

Symbol Attributes

The **Symbol** Attributes word is interpreted as follows:

- Bit 0 denotes that the symbol is defined in this object file.
- Bit 1 denotes that the symbol has global scope and can be matched by the linker to a similarly named symbol from another object file.

Specifically:

Bits 1 and 0

- | | |
|-----------|---|
| 01 | (bit 1 unset, bit 0 set)
denotes that the symbol is defined in this object file and has scope limited to this object file (when resolving symbol references, the linker will only match this symbol to references from within the same object file). |
| 10 | (bit 1 set, bit 0 unset)
denotes that the symbol is a reference to a symbol defined in another object file. If no defining instance of the symbol is found the linker attempts to match the name of the symbol to the names of common blocks. If a match is found it is as if there were defined an identically-named symbol of global scope, having as its value the base address of the common area. |
| 11 | denotes that the symbol is defined in this object file with global scope (when attempting to resolve unresolved references, the linker will match this definition to a reference from another object file). |
| 00 | Reserved by Acorn. |

Bit 2

Bit 2 encodes the **absolute** attribute which is meaningful only if the symbol is a defining occurrence (bit 0 set). If set, it denotes that the symbol has an absolute value, for example, a constant. If unset, the symbol's value is relative to the base address of the area defined by the `Area Name` field of the symbol.

Bit 3

Bit 3 encodes the **case insensitive reference** attribute which is meaningful only if bit 0 is unset (that is, if the symbol is an external reference). If set, the linker will ignore the case of the symbol names it tries to match when attempting to resolve this reference.

Bit 4

Bit 4 encodes the **weak** attribute which is meaningful only if the symbol is an external reference, (bits 1,0 = 10). It denotes that it is acceptable for the reference to remain unsatisfied and for any fields relocated via it to remain unrelocated. The linker ignores weak references when deciding which members to load from an object library.

Bit 5

Bit 5 encodes the **strong** attribute which is meaningful only if the symbol is an external defining occurrence (if bits 1,0 = 11). In turn, this attribute only has meaning if there is a non-strong, external definition of the same symbol in another object file. In this case, references to the symbol from outside of the file containing the strong definition, resolve to the strong definition, while those within the file containing the strong definition resolve to the non-strong definition.

This attribute allows a kind of link-time indirection to be enforced. Usually, a strong definition will be absolute, and will be used to implement an operating system's entry vector having the **forever binary** property.

Bit 6

Bit 6 encodes the **common** attribute, which is meaningful only if the symbol is an external reference (bits 1,0 = 10). If set, the symbol is a reference to a common area with the symbol's name. The length of the common area is given by the symbol's `Value field` (see above). The linker treats common symbols much as it treats areas having the **Common Reference** attribute – all symbols with the same name are assigned the same base address, and the length allocated is the maximum of all specified lengths.

If the name of a common symbol matches the name of a common area, then these are merged and the symbol identifies the base of the area.

All common symbols for which there is no matching common area (reference or definition) are collected into an anonymous, linker-created, pseudo-area.

Bit 7

Bit 7 is reserved and shall be set to 0.

Bit 8-13

Bits 8-13 encode additional attributes of symbols defined in code areas.

Bit 8 encodes the **code datum** attribute which is meaningful only if this symbol defines a location within an area having the **Code** attribute. It denotes that the symbol identifies a (usually read-only) datum, rather than an executable instruction.

Bit 9 encodes the **floating-point arguments in floating-point registers** attribute. This is meaningful only if the symbol identifies a function entry point. A symbolic reference with this attribute cannot be matched by the linker to a symbol definition which lacks the attribute.

Bit 10 is reserved and shall be set to 0.

Bit 11 is the **simple leaf function** attribute which is meaningful only if this symbol defines the entry point of a sufficiently simple leaf function (a leaf function is one which calls no other function). For a reentrant leaf function it denotes that the function's inter-link-unit entry point is the same as its intra-link-unit entry point.

Bit 12 is the **Thumb** attribute. This is meaningful only if this symbol defines a location within an area having the **code** attribute.

Bit 13 encodes the **soft floating-point** attribute. This is meaningful only if the symbol identifies a function entry point. It declares that the function not only receives its floating point arguments in integer registers, but also returns any floating point return value in integer registers. A symbolic reference with this attribute cannot be matched by the linker to a symbol definition which lacks the attribute.

Bit 14-31

Bits 14-31 are reserved and shall be set to 0.

Symbol Attribute Summary

Bit	Mask	Attribute Description
0	0x00000001	Symbol is defined in this file
1	0x00000002	Symbol has global scope
2	0x00000004	Absolute attribute
3	0x00000008	Case-insensitive attribute
4	0x00000010	Weak attribute
5	0x00000020	Strong attribute
6	0x00000040	Common attribute
Code symbols only		
8	0x00000100	Code area datum attribute
9	0x00000200	FP args in FP regs attribute
10	0x00000400	Reserved and currently set to 0
11	0x00000800	Simple leaf function attribute
12	0x00001000	Thumb attribute
13	0x00002000	Soft floating point attribute
14-31		Reserved and currently set to 0

String table chunk (OBJ_STRT)

The string table chunk contains all the print names referred to from the header and symbol table chunks. This separation is made to factor out the variable length characteristic of print names from the key data structures.

A print name is stored in the string table as a sequence of non-control characters (codes 32-126 and 160-255) terminated by a NUL (0) byte, and is identified by an offset from the start of the table. The first 4 bytes of the string table contain its length (including the length of its length word), so no valid offset into the table is less than 4, and no table has length less than 4.

The endian-ness of the length word shall be identical to the endian-ness of the AOF and chunk files containing it.

Identification chunk (OBJ_IDFN)

This chunk should contain a string of printable characters (codes 10-13 and 32-126) terminated by a NUL (0) byte, which gives information about the name and version of the tool which generated the object file. Use of codes in the range 128-255 is discouraged, as the interpretation of these values is host dependent.

ALF

ALF is the format of linkable libraries (such as the C RISC OS Toolbox library `toolboxlib`).

Library file format

For library files, the first part of each chunk's name is 'LIB_'; for object libraries, the names of the additional two chunks begin with 'OFL_'.

Each piece of a library file is stored in a separate, identifiable chunk, named as follows:

Chunk	Chunk Name	
Directory	LIB_DIRY	
Time-stamp	LIB_TIME	
Version	LIB_VSRN	
Data	LIB_DATA	
Symbol table	OFL_SYMT	– object code libraries only
Time-stamp	OFL_TIME	– object code libraries only

There may be many LIB_DATA chunks in a library, one for each library member. In all chunks, word values are stored with the same byte order as the target system; strings are stored in ascending address order, which is independent of target byte order.

LIB_DIRY

The LIB_DIRY chunk contains a directory of the modules in the library, each of which is stored in a LIB_DATA chunk. The directory size is fixed when the library is created. The directory consists of a sequence of variable length entries, each an integral number of words long. The number of directory entries is determined by the size of the LIB_DIRY chunk.

This is shown pictorially in the following diagram:

ChunkIndex	
EntryLength	the size of this LIB_DIRY chunk (an integral number of words)
DataLength	the size of the Data (an integral number of words)
Data	

ChunkIndex

ChunkIndex is a word containing the 0-origin index within the chunk file header of the corresponding LIB_DATA chunk. Conventionally, the first 3 chunks of an OFL file are LIB_DIRY, LIB_TIME and LIB_VSRN, so ChunkIndex is at least 3. A ChunkIndex of 0 means the directory entry is unused.

The corresponding LIB_DATA chunk entry gives the offset and size of the library module in the library file.

EntryLength

EntryLength is a word containing the number of bytes in this LIB_DIRY entry, always a multiple of 4.

DataLength

DataLength is a word containing the number of bytes used in the data section of this LIB_DIRY entry, also a multiple of 4.

Data

The Data section consists of, in order:

- a 0-terminated string (the name of the library member)
- any other information relevant to the library module (often empty)
- a 2-word, word-aligned time stamp.

Strings should contain only ISO-8859 non-control characters (codes [0-31], 127 and 128+[0-31] are excluded).

The string field is the name used to identify this library module. Typically it is the name of the file from which the library member was created.

The format of the time stamp is described in *Time Stamps* on page 269. Its value is an encoded version of the last-modified time of the file from which the library member was created.

To ensure maximum robustness with respect to earlier, now obsolete, versions of the ARM object library format:

- Applications which create libraries or library members should ensure that the LIB_DIRY entries they create contain valid time stamps.

- Applications which read LIB_DIRY entries should not rely on any data beyond the end of the name string being present, unless the difference between the DataLength field and the name-string length allows for it. Even then, the contents of a time stamp should be treated cautiously and not assumed to be sensible.

Applications which write LIB_DIRY or OFL_SYMT entries should ensure that padding is done with NUL (0) bytes; applications which read LIB_DIRY or OFL_SYMT entries should make no assumptions about the values of padding bytes beyond the first, string-terminating NUL byte.

Time Stamps

A library time stamp is a pair of words encoding the following:

- a 6-byte count of centi-seconds since the start of the 20th century
- a 2-byte count of microseconds since the last centi-second (usually 0).

centiseconds since 00:00:00 1st January 1900	first (most significant) word
u-seconds	second (least significant) word

The first word stores the most significant 4 bytes of the 6-byte count; the least significant 2 bytes of the count are in the most significant half of the second word.

The least significant half of the second word contains the microsecond count and is usually 0.

Time stamp words are stored in target system byte order: they must have the same endian-ness as the containing chunk file.

LIB_TIME

The LIB_TIME chunk contains a 2-word time stamp recording when the library was last modified. It is, hence, 8 bytes long.

LIB_VSRN

The version chunk contains a single word whose value is 1.

LIB_DATA

A LIB_DATA chunk contains one of the library members indexed by the LIB_DIRY chunk. The endian-ness or byte order of this data is, by assumption, the same as the byte order of the containing library/chunk file.

No other interpretation is placed on the contents of a member by the library management tools. A member could itself be a file in chunk file format or even another library.

Object Code Libraries

An object code library is a library file whose members are files in ARM Object Format (see section AOF on page 249 for details).

An object code library contains two additional chunks: an external symbol table chunk named OFL_SYMT; and a time stamp chunk named OFL_TIME.

OFL_SYMT

The external symbol table contains an entry for each external symbol defined by members of the library, together with the index of the chunk containing the member defining that symbol.

The OFL_SYMT chunk has exactly the same format as the LIB_DIRY chunk except that the Data section of each entry contains only a string, the name of an external symbol, and between 1 and 4 bytes of NUL padding, as follows:

ChunkIndex	the size of this OFL_SYMT chunk (an integral number of words)
EntryLength	
DataLength	the size of the External Symbol Name and Padding (an integral number of words)
External Symbol Name	
Padding	

OFL_SYMT entries do not contain time stamps.

OFL_TIME

The OFL_TIME chunk records when the OFL_SYMT chunk was last modified and has the same format as the LIB_TIME chunk (see *Time Stamps* on page 269).

AIF

ARM Image Format (AIF) is a simple format for ARM executable images, which consists of a 128 byte header followed by the image's code, followed by the image's initialised static data.

Properties of AIF

Two variants of AIF exist:

- **Executable AIF** (in which the header is part of the image itself) can be executed by entering the header at its first word. Code in the header ensures the image is properly prepared for execution before being entered at its entry address.
- **Non-executable AIF** (in which the header is not part of the image, but merely describes it) is intended to be loaded by a program which interprets the header, and prepares the following image for execution.

The two flavours of AIF are distinguished as follows:

- The fourth word of an executable AIF header is BL `entrypoint`. The most significant byte of this word (in the target byte order) is 0xEB.
- The fourth word of a non-executable AIF image is the offset of its entry point from its base address. The most significant nibble of this word (in the target byte order) is 0x0.

The base address of an executable AIF image is the address at which its header should be loaded; its code starts at base + 0x80. The base address of a non-executable AIF image is the address at which its code should be loaded.

Executable AIF

The following remarks about executable AIF apply also to non-executable AIF, except that loader code must interpret the AIF header and perform any required decompression, relocation, and creation of zero-initialised data. Compression and relocation are, of course, optional: AIF is often used to describe very simple absolute images.

It is assumed that on entry to a program in ARM Image Format (AIF), the general registers contain nothing of value to the program (the program is expected to communicate with its operating environment using SWI instructions or by calling functions at known, fixed addresses).

A program image in ARM Image Format is loaded into memory at its load address, and entered at its first word. The load address may be:

- an implicit property of the type of the file containing the image (as is usual with UNIX executable file types, Acorn Absolute file types, etc.)
- read by the program loader from offset 0x28 in the file containing the AIF image
- given by some other means, e.g. by instructing an operating system or debugger to load the image at a specified address in memory.

An AIF image may be compressed and can be self-decompressing (to support faster loading from slow peripherals, and better use of space in ROMs and delivery media such as floppy discs). An AIF image is compressed by a separate utility which adds self-decompression code and data tables to it.

If created with appropriate linker options, an AIF image may relocate itself at load time. Two kinds of self-relocation are supported:

- relocate to load address (the image can be loaded anywhere and will execute where loaded)
- self-move up memory, leaving a fixed amount of workspace above, and relocate to this address (the image is loaded at a low address and will move to the highest address which leaves the required workspace free before executing there).

The second kind of self-relocation can only be used if the target system supports an operating system or monitor call which returns the address of the top of available memory. The ARM linker provides a simple mechanism for using a modified version of the self-move code illustrated in *Self-Move and Self-Relocation Code* on page 277, allowing AIF to be easily tailored to new environments.

AIF images support being debugged by the Desktop debugging tool (DDT). Low-level and source-level support are orthogonal, and both, either, or neither kind of debugging support need be present in an AIF image.

For details of the format of the debugging tables see ASD on page 280.

References from debugging tables to code and data are in the form of relocatable addresses. After loading an image at its load address these values are effectively absolute. References between debugger table entries are in the form of offsets from the beginning of the debugging data area. Thus, following relocation of a whole image, the debugging data area itself is position independent and may be copied or moved by the debugger.

The Layout of AIF

The layout of a compressed AIF image is as follows:

Header	
Compressed image	
Decompression data	This data is position-independent
Decompression code	This code is position-independent

The header is small, fixed in size, and described below. In a compressed AIF image, the header is **not** compressed.

An uncompressed image has the following layout:

Header	
Read-only area	
Read-write area	
Debugging data	(optional)
Self-relocation code	Position-independent
Relocation list	List of words to relocate, terminated by -1

Debugging data is absent unless the image has been linked using the linker's `-d` option and, in the case of source-level debugging, unless the components of the image have been compiled using the compiler's `-g` option.

The relocation list is a list of byte offsets from the beginning of the AIF header, of words to be relocated, followed by a word containing `-1`. The relocation of non-word values is not supported.

After the execution of the self-relocation code – or if the image is not self-relocating – the image has the following layout:

Header	
Read-only area	
Read-write area	
Debugging data	(optional)

At this stage a debugger is expected to copy any debugging data to somewhere safe, otherwise it will be overwritten by the zero-initialised data and/or the heap/stack data of the program. A debugger can seize control at the appropriate moment by copying, then modifying, the third word of the AIF header (see *AIF Header Layout* on page 275).

AIF Header Layout

00	BL DecompressCode	NOP 0 if the image is not compressed
04	BL SelfRelocCode	NOP 0 if the image is not self-relocating
08	BL DBGInit/Zerolnit	NOP 0 if the image has none
0C	BL ImageEntryPoint or EntryPoint offset	BL to make header addressable via R14 but the application shall not return ... Non-executable AIF uses an offset, not BL
10	<Program Exit Instr>	...last ditch in case of return
14	Image ReadOnly size	Includes header size if executable AIF; excludes header size if non-executable AIF
18	Image ReadWrite size	Exact size - a multiple of 4 bytes
1C	Image Debug size	Exact size - a multiple of 4 bytes
20	Image zero-init size	Exact size - a multiple of 4 bytes
24	Image debug type	0,1,2 or 3 (see below)
28	Image base	Address the image (code) was linked at
2C	Work space	Min work space - in bytes - to be reserved a self-moving relocatable image
30	Address mode: 26/32 +3 flag bytes	LS byte contains 26 or 32 bit 8 set when using a separate data base
34	Data base	Address the image data was linked at
38	Two reserved words ... initially 0 ...	
40	<Debug Init Instr>	NOP if unused
44	Zero-init code (14 words as below)	Header is 32 words long

Notes

NOP is encoded as MOV r0, r0.

BL is used to make the header addressable via r14 in a position-independent manner, and to ensure that the header will be position-independent. Care is taken to ensure that the instruction sequences which compute addresses from these r14 values work in both 26-bit and 32-bit ARM modes.

Program Exit Instruction will usually be a SWI causing program termination. On systems which lack this, a branch-to-self is recommended. Applications are expected to exit directly and **not** to return to the AIF header, so this instruction should never be executed. The ARM linker sets this field to SWI 0x11 by default, but it may be set to any desired value by providing a template for the AIF header in an area called AIF_HDR in the **first** object file in the input list to **Link**.

Image ReadOnly Size includes the size of the AIF header only if the AIF type is executable (that is, if the header itself is part of the image).

An AIF image is re-startable if, and only if, the program it contains is re-startable (note: an AIF image is **not** reentrant). If an AIF image is to be re-started then, following its decompression, the first word of the header must be set to NOP. Similarly, following self-relocation, the second word of the header must be reset to NOP. This causes no additional problems with the read-only nature of the code segment: both decompression and relocation code must write to it. On systems with memory protection, both the decompression code and the self-relocation code must be bracketed by system calls to change the access status of the read-only section (first to writable, then back to read-only).

The **image debug type** has the following meaning:

- 0: No debugging data are present.
- 1: Low-level debugging data are present.
- 2: Source level (ASD) debugging data are present.
- 3: 1 and 2 are present together.

All other values of image debug type are reserved to ARM Ltd.

Debug Initialisation Instruction (if used) is expected to be a SWI instruction which alerts a resident debugger that a debuggable image is commencing execution. Of course, there are other possibilities within the AIF framework. The linker sets this field to NOP by default, but it can be customised by providing your own template for the AIF header in an area called AIF_HDR in the **first** object file in the input list to **Link**.

The **Address mode** word (at offset 0x30) is 0, or contains in its least significant byte (using the byte order appropriate to the target):

- the value 26, indicating the image was linked for a 26-bit ARM mode, and may not execute correctly in a 32-bit mode
- the value 32, indicating the image was linked for a 32-bit ARM mode, and may not execute correctly in a 26-bit mode.

A value of 0 indicates an old-style 26-bit AIF header.

If the Address mode word has bit 8 set ((address_mode & 0x100) != 0), then the image was linked with separate code and data bases (usually the data is placed immediately after the code). In this case, the word at offset 0x34 contains the base address of the image's data.

Zero-Initialisation Code

The Zero-initialisation code is as follows:

```
ZeroInit
    NOP                ; or <Debug Init Instruction>
    SUB    ip, lr, pc   ; base+12+[PSR]-(ZeroInit+12+PSR)
                        ; = base-ZeroInit
    ADD    ip, pc, ip   ; base-ZeroInit+ZeroInit+16
                        ; = base+16
    LDMIB  ip, {r0,r1,r2,r3} ; various sizes
    SUB    ip, ip, #16   ; image base
    LDR    r2, [ip, #48] ; flags
    TST    r2, #256      ; separate data area?
    LDRNE  ip, [ip, #52] ; Yes, so get it...
    ADDEQ  ip, ip, r0     ; No, so add + R0 size
    ADD    ip, ip, r1     ; + RW size = base of 0-init area
    MOV    r0, #0
    CMP    r3, #0
00  MOVLE  pc, lr        ; nothing left to do
    STR    r0, [ip], #4
    SUBS   r3, r3, #4
    B      %B00
```

Self-Move and Self-Relocation Code

This code is added to the end of an AIF image by the linker, immediately before the list of relocations (which is terminated by -1). Note that the code is entered via a BL from the second word of the AIF header so, on entry, r14 points to AIFHeader + 8. In 26-bit ARM modes, r14 also contains a copy of the PSR flags.

On entry, the relocation code calculates the address of the AIF header (in a CPU-independent fashion) and decides whether the image needs to be moved. If the image doesn't need to be moved, the code branches to R(elocateOnly).

```

RelocCode
NOP                                ; required by ensure_byte_order()
                                   ; and used below.
SUB    ip, lr, pc                  ; base+8+[PSR]-(RelocCode+12+[PSR])
                                   ; = base-4-RelocCode
ADD    ip, pc, ip                  ; base-4-RelocCode+RelocCode+16 = base+12
SUB    ip, ip, #12                 ; -> header address
LDR    r0, RelocCode              ; NOP
STR    r0, [ip, #4]               ; won't be called again on image re-entry
LDR    r9, [ip, #&2C]             ; min free space requirement
CMP    r9, #0                     ; 0 => no move, just relocate
BEQ    RelocateOnly

```

If the image needs to be moved up memory, then the top of memory has to be found. Here, a system service (SWI 0x10) is called to return the address of the top of memory in r1. This is, of course, system specific and should be replaced by whatever code sequence is appropriate to the environment.

```

LDR    r0, [ip, #&20]             ; image zero-init size
ADD    r9, r9, r0                 ; space to leave = min free + zero init
SWI    #&10                       ; return top of memory in r1.

```

The following code calculates the length of the image inclusive of its relocation data, and decides whether a move up store is possible.

```

ADR    r2, End                    ; -> End
01 LDR    r0, [r2], #4             ; load relocation offset, increment r2
CMNS   r0, #1                     ; terminator?
BNE    %B01                      ; No, so loop again
SUB    r3, r1, r9                 ; MemLimit - freeSpace
SUBS   r0, r3, r2                 ; amount to move by
BLE    RelocateOnly              ; not enough space to move...
BIC    r0, r0, #15                ; a multiple of 16...
ADD    r3, r2, r0                 ; End + shift
ADR    r8, %F02                  ; intermediate limit for copy-up

```

Finally, the image copies itself four words at a time, being careful about the direction of copy, and jumping to the copied copy code as soon as it has copied itself.

```

02 LDMDB r2!, {r4-r7}
STMDB r3!, {r4-r7}
CMP    r2, r8                    ; copied the copy loop?
BGT    %B02                      ; not yet
ADD    r4, pc, r0
MOV    pc, r4                    ; jump to copied copy code
03 LDMDB r2!, {r4-r7}
STMDB r3!, {r4-r7}
CMP    r2, ip                    ; copied everything?
BGT    %B03                      ; not yet
ADD    ip, ip, r0                 ; load address of code
ADD    lr, lr, r0                 ; relocated return address

```

Whether the image has moved itself or not, control eventually arrives here, where the list of locations to be relocated is processed. Each location is word sized and is relocated by the difference between the address the image was loaded at (the address of the AIF header) and the address the image was linked at (stored at offset 0x28 in the AIF header).

```

RelocateOnly
    LDR    r1, [ip, #&28]    ; header + 0x28 = code base set by Link
    SUBS   r1, ip, r1        ; relocation offset
    MOVEQ  pc, lr           ; relocate by 0 so nothing to do
    STR    ip, [ip, #&28]    ; new image base = actual load address
    ADR    r2, End          ; start of reloc list
04  LDR    r0, [r2], #4      ; offset of word to relocate
    CMNS   r0, #1           ; terminator?
    MOVEQ  pc, lr           ; yes => return
    LDR    r3, [ip, r0]      ; word to relocate
    ADD    r3, r3, r1        ; relocate it
    STR    r3, [ip, r0]      ; store it back
    B      %B04             ; and do the next one
End                                     ; The list of offsets of locations to
                                     ; relocate starts here, terminated by -1

```

You can customise the self-relocation and self-moving code generated by **Link** by providing your version of it in an area called AIF_RELOC in the **first** object file in Link's input list.

ASD

Acknowledgement: This design is based on work originally done for Acorn Computers by Topexpress Ltd.

This section specifies the format of symbolic debugging data generated by ARM compilers, which is used by the Desktop debugging tool (DDT) to support high level language oriented, interactive debugging.

For each separate compilation unit (called a *section*) the compiler produces debugging data, and a special area in the object code (see section AOF on page 249 for an explanation of ARM Object Format, including areas and their attributes). Debugging data are position independent, containing only relative references to other debugging data within the same section, and relocatable references to other compiler-generated areas.

Debugging data areas are combined by the linker into a single contiguous section of a program image. For a description of the linker's principal output format see section AIF on page 271.

Since the debugging section is position-independent, the debugger can move it to a safe location before the image starts executing. If the image is not executed under debugger control, the debugging data are simply overwritten.

The format of debugging data allows for a variable amount of detail. This potentially allows the user to trade off among memory used, disc space used, execution time, and debugging detail.

Assembly-language level debugging is also supported, though in this case the debugging tables are generated by the linker. If required, the assembler can generate debugging table entries relating code addresses to source lines. Low-level debugging tables appear in an extra section item, as if generated by an independent compilation (see *Debugging Data Items in Detail* on page 283). Low-level and high-level debugging are orthogonal facilities, though DDT allows the user to move smoothly between levels if both sets of debugging data are present in an image.

Order of Debugging Data

A debug data area consists of a series of *items*. The arrangement of these items mimics the structure of the high-level language program itself.

For each debug area, the first item is a section item, giving global information about the compilation, including a code identifying the language, and flags indicating the amount of detail included in the debugging tables.

Each datum, function, procedure, etc., definition in the source program has a corresponding debug data item; these items appear in an order corresponding to the order of definitions in the source. This means that any nested structure in the source program is preserved in the debugging data, and the debugger can use this structure to make deductions about the scope of various source-level objects. Of course, for procedure definitions, two debug items are needed: a **procedure** item to mark the definition itself, and an **endproc** item to mark the end of the procedure's body and the end of any nested definitions. If procedure definitions are nested then the procedure-endproc brackets are nested too. Variable and type definitions made at the outermost level, of course, appear outside of all procedure/endproc items.

Information about the relationship between the executable code and source files is collected together and appears as a **fileinfo** item, which is always the final item in a debugging area. Because of the C language's `#include` facility, the executable code produced from an outer-level source file may be separated into disjoint pieces interspersed with that produced from the included files. Therefore, source files are considered to be collections of 'fragments', each corresponding to a contiguous area of executable code, and the fileinfo item is a list with an entry for each file, each in turn containing a list with an entry for each fragment. The fileinfo field in the section item addresses the fileinfo item itself. In each procedure item there is a 'fileentry' field, which refers to the file-list entry for the source file containing the procedure's start; there is a separate one in the endproc item because it may possibly not be in the same source file.

Endian-ness and the Encoding of Debugging Data

The ARM can be configured to use either a little-endian memory system (the least significant byte of each 4-byte word has the lowest address), or a big-endian memory system (the most significant byte of each 4-byte word has the lowest address).

In general, the code to be generated varies according to the endian-ness (or byte-sex) of the target. The linker has insufficient information to change an object file's byte sex, so object files are encoded using the byte order of the intended target, independently of the byte order of the host system on which the compiler or assembler runs. The linker accepts inputs having either byte order, but rejects mixed sex inputs, and generates its output using the same byte order.

This means that producers of debugging tables must be prepared to generate them in either byte order, as required. In turn, this requires definitions to be very clear about when a 4-byte word is being used (which will require reversal on output or

input when cross-sex compiling or debugging), and when a sequence of bytes is being used (which requires no special treatment provided it is written and read as a sequence of bytes in address order).

Representation of Data Types

Several of the debugging data items (e.g. procedure and variable) have a **type** word field to identify their data type. This field contains, in the most significant 24 bits, a code to identify a base type, and in the least significant 8 bits, a pointer count:

- 0 to denote the type itself
- 1 to denote a pointer to the type
- 2 to denote a pointer to a pointer to...
- etc.

For simple types the code is a positive integer as follows, (all codes are decimal):

void	0
signed integers	
single byte	10
half-word	11
word	12
unsigned integers	
single byte	20
half-word	21
word	22
floating point	
float	30
double	31
long double	32
complex	
single complex	41
double complex	42
functions	
function	100

For compound types (arrays, structures, etc.) there is a special kind of debug data item (array, struct, etc.) to give details such as array bounds and field types. The type code for compound types is negative, the negation of the (byte) offset of the debug item from the start of the debugging area.

If a type has been given a name in a source program, it will give rise to a **type** debugging data item which contains the name and a type word as defined above. If necessary, there will also be a debugging data item, such as an **array** or **struct** item, to define the type itself. In that case, the type word will refer to this item.

Set types in Pascal are not treated in detail: the only information recorded for them is the total size occupied by the object in bytes. Neither are Pascal file variables supported by the debugger, since their behaviour under debugger control is unlikely to be helpful to the user.

FORTTRAN character types are supported by special kinds of debugging data item, the format of which is specific to each FORTRAN compiler.

Representation of Source File Positions

Several of the debugging data items have a **sourcepos** field to identify a position in the source file. This field contains a line number and character position within the line packed into a single word. The most significant 10 bits encode the character offset (0-based) from the start of the line and the least-significant 22 bits give the line number.

Debugging Data Items in Detail

The Code and Length Field

The first word of each debugging data item contains the byte length of the item (encoded in the most significant 16 bits), and a code identifying the kind of item (in the least significant 16 bits). The defined codes are:

1	section
2	procedure/function definition
3	endproc
4	variable
5	type
6	struct
7	array
8	subrange
9	set
10	fileinfo
11	contiguous enumeration
12	discontiguous enumeration
13	procedure/function declaration
14	begin naming scope
15	end naming scope
16	bitfield

The meaning of the second and subsequent words of each item is defined below.

If a debugger encounters a code it does not recognise, it should use the length field to skip the item entirely. This discipline allows the debugging tables to be extended without invalidating existing debuggers.

Text Names in Items

Where items include a string field, the string is packed into successive bytes beginning with a length byte, and padded at the end to a word boundary with 0 bytes. The length of a string is in the range [0..255] bytes.

Offsets in File and Addresses in Memory

Where an item contains a field giving an offset in the debugging data area (usually to address another item), this means a byte offset from the start of the debugging data for the whole section (in other words, from the start of the section item).

When the same structure is used to map debugging data in memory, an offset field may be used to hold a pointer to another debug item in memory, rather than the offset of it in the debug area.

Section Items

A section item is the first item of each section of the debugging data. After its code and length word it contains the fields listed below. First there are 4 flag bytes:

lang	a byte identifying the source language
flags	a byte describing the level of detail
unused	
asdversion	a byte version number of the debugging data

The following language byte codes are defined:

LANG_NONE	0	Low-level debugging data only
LANG_C	1	C source level debugging data
LANG_PASCAL	2	Pascal source level debugging data
LANG_FORTRAN	3	FORTRAN-77 source level debugging data
LANG_ASM	4	ARM Assembler line number data

All other codes are reserved to ARM.

The flags byte uses the following mask values:

- | | |
|---|---|
| 1 | debugging data contains line-number information |
| 2 | debugging data contains information about top-level variables |
| 3 | both of the above |

The asdversion byte should be set to 3, the version of this definition.

The flag bytes are followed by the following word-sized fields:

codestart	address of first instruction in this section
datastart	address of start of static data for this section
codesize	byte size of executable code in this section
datasize	byte size of the static data in this section
fileinfo	offset in the debugging area of the fileinfo item for this section (0 if no fileinfo item present)
debugsize	total byte length of debug data for this section
name or nsyms	string or integer (the first byte of string is the string's length, followed by a non-NULL-terminated string of characters with NULL padding up to the next word boundary)

codestart and datastart are addresses, relocated by the linker. The fileinfo field, nominally an offset, is also used as a pointer when this structure is mapped in memory. The fileinfo field is 0 if no source file information is present.

The name field contains the program name for Pascal and FORTRAN programs. For C programs it contains a name derived by the compiler from the root file name (notionally a module name). In each case, the name is similar to a variable name in the source language. For a low-level debugging section (language = 0), the field is treated as a 4 byte integer giving the number of symbols following.

For linker-generated low-level debugging data, the fields have the following values:

language	0
codestart	Image\$\$RO\$\$Base
datastart	Image\$\$RW\$\$Base
codesize	Image\$\$RO\$\$Limit - Image\$\$RO\$\$Base
datasize	Image\$\$RW\$\$Limit - Image\$\$RW\$\$Base
fileinfo	0
nsyms	number of symbols in the following debugging data
debugsize	total size of the low-level debugging data including the size of this section item

For linker-generated low-level debugging data, the section item is followed by nsyms symbol items, each consisting of 2 words:

sym	flags + byte offset in string table of symbol name
value	the value of the symbol

sym encodes an index into the string table in the 24 least significant bits, and the following flag values in the 8 most significant bits:

ASD_GLOBSYM	0	if the symbol is absolute
ASD_ABSSYM	0x01000000L	if the symbol is global
ASD_TEXTSYM	0x02000000L	if the symbol names code
ASD_DATASYM	0x04000000L	if the symbol names data
ASD_ZINITSYM	0x06000000L	if the symbol names 0-initialised data

Note that the linker reduces all symbol values to absolute values, so that the flag values record the history, or origin, of the symbol in the image.

Immediately following the symbol table is the string table, in standard AOF format. It consists of:

- a length word
- the strings themselves, each terminated by a NUL (0).

The length word includes the size of the length word, so no offset into the string table is less than 4. The end of the string table is padded with NULs to the next word boundary (so the length is a multiple of 4).

Procedure Items

A procedure item appears once for each procedure or function definition in the source program. Any definitions within the procedure have their related debugging data items between the procedure item and its matching endproc item. After its code and length field, a procedure item contains the following word-sized fields:

type	the return type if this is a function, else 0 (see <i>Representation of Data Types</i> on page 282)
args	the number of arguments
sourcepos	the source position of the procedure's start (see <i>Representation of Data Types</i> on page 282)
startaddr	address of 1st instruction of procedure prologue
entry	address of 1st instruction of the procedure body (see note below)
endproc	offset of the related endproc item (in file) or pointer to related endproc item (in memory)

type	the return type if this is a function, else 0 (see <i>Representation of Data Types</i> on page 282)
fileentry	offset of the file list entry for the source file (in file) or a pointer to it (in memory)
name	string (the first byte of string is the string's length, followed by a non-NULL-terminated string of characters with NULL padding up to the next word boundary)

The entry field addresses the first instruction following the procedure prologue. That is, the first address at which a high-level breakpoint could sensibly be set. The startaddr field addresses the start of the prologue. That is, the instruction at which control arrives when the procedure is called.

Label Items

A label in a source program is represented by a special procedure item with no matching endproc, (the endproc field is 0 to denote this). Pascal and FORTRAN numerical labels are converted by their respective compilers into strings prefixed by \$n.

For FORTRAN77, multiple entry points to the same procedure each give rise to a separate procedure item, all of which have the same endproc offset referring to the unique, matching endproc item.

Endproc Items

An endproc item marks the end of the debugging data items belonging to a particular procedure. It also contains information relating to the procedure's return. After its code and length field, an endproc item contains the following word-sized fields:

sourcepos	position in the source file of the procedure's end (see <i>Representation of Source File Positions</i> on page 283)
endpoint	address of the code byte after the compiled code for the procedure
fileentry	offset of the file-list entry for the procedure's end (in file) or a pointer to it (in memory)
nreturns	number of procedure return points (may be 0)
retadds	array of addresses of procedure return code

If the procedure body is an infinite loop, there will be no return point, so `nreturns` will be 0. Otherwise each member of `retaddrs` should point to a suitable location at which a breakpoint may be set 'at the exit of the procedure'. When execution reaches this point, the current stack frame should still be for this procedure.

Variable Items

A variable item contains debugging data relating to a source program variable, or a formal argument to a procedure (the first variable items in a procedure always describe its arguments). After its code and length field, a variable item contains the following word-sized fields:

type	type of this variable (see <i>Representation of Data Types</i> on page 282)
sourcepos	the source position of the variable (see <i>Representation of Source File Positions</i> on page 283)
storageclass	a word encoding the variable's storage class
location	see explanation below
name	string (the first byte of string is the string's length, followed by a non-NULL-terminated string of characters with NULL padding up to the next word boundary)

The following codes define the storage classes of variables:

1	external variables (or FORTRAN common)
2	static variables private to one section
3	automatic variables
4	register variables
5	Pascal 'var' arguments
6	FORTRAN arguments
7	FORTRAN character arguments

The meaning of the location field of a variable item depends on the storage class; it contains:

- an absolute address for static and external variables (relocated by the linker)
- a stack offset (an offset from the frame pointer) for automatic and var-type arguments
- an offset into the argument list for FORTRAN arguments
- a register number for register variables, (the 8 floating point registers are numbered 16..23).

No account is taken of variables which ought to be addressed by +ve offsets from the stack-pointer rather than -ve offsets from the frame-pointer.

The sourcepos field is used by the debugger to distinguish between different definitions having the same name (e.g. identically named variables in disjoint source-level naming scopes such as nested blocks in C).

Type Items

A type item is used to describe a named type in the source language (e.g. a typedef in C). After its code and length field, a type item contains two word-sized fields:

type	a type word (see <i>Representation of Data Types</i> on page 282)
name	string (the first byte of string is the string's length, followed by a non-NULL-terminated string of characters with NULL padding up to the next word boundary)

Struct Items

A struct item is used to describe a structured data type (e.g. a struct in C or a record in Pascal). After its code and length field, a struct item contains the following word-sized fields:

fields	the number of fields in the structure
size	total byte size of the structure
fieldtable...	an array of fields struct field items

Each struct field item has the following word-sized fields:

offset	byte offset of this field within the structure
type	a type word (see <i>Representation of Data Types</i> on page 282)
name	string (the first byte of string is the string's length, followed by a non-NULL-terminated string of characters with NULL padding up to the next word boundary)

Union types are described by struct items in which all fields have 0 offsets.

C bit fields are not treated in full detail: a bit field is simply represented by an integer starting on the appropriate word boundary (so that the word contains the whole field).

Array Items

An array item is used to describe a one-dimensional array. Multi-dimensional arrays are described as 'arrays of arrays'. Which dimension comes first is dependent on the source language (which is different for C and FORTRAN). After its code and length field, an array item contains the following word-sized fields:

size	total byte size of the array
flags	see below
basetype	a type word (see <i>Representation of Data Types</i> on page 282)
lowerbound	constant value or location of variable
upperbound	constant value or location of variable

If the size field is zero, debugger operations affecting the whole array, rather than individual elements of it, are forbidden.

The following mask values are defined for the flags field:

ARRAY_UNDEF_LBOUND	1	lower bound is undefined
ARRAY_CONST_LBOUND	2	lower bound is a constant
ARRAY_UNDEF_UBOUND	4	upper bound is undefined
ARRAY_CONST_UBOUND	8	upper bound is a constant
ARRAY_VAR_LBOUND	16	lower bound is a variable
ARRAY_VAR_UBOUND	32	upper bound is a variable

A bound is described as undefined when no information about it is available.

A bound is described as constant when its value is known at compile time. In this case, the corresponding bound field gives its value.

If a bound is described as variable, the offset field identifies a variable debug item describing the location containing the bound. In a debug area in an object file, the offset field contains the offset from the start of the debug area to the variable item; in memory it contains a pointer to the corresponding variable item. Note that a variable item may be used to describe a location known to the compiler, which need not correspond to a source language variable.

Subrange Items

A subrange item is used to describe a subrange typed in Pascal. It also serves to describe enumerated types in C, and scalars in Pascal (in which case the base type is understood to be an unsigned integer of appropriate size). After its code and length field, a subrange item contains the following word-sized fields:

sizeandtype	see below
lb	low bound of subrange
hb	high bound of subrange

The sizeandtype field encodes the byte size of container for the subrange (1, 2 or 4) in its least significant 16 bits, and a simple type code (see *Representation of Data Types* on page 282) in its most significant 16 bits. The type code refers to the base type of the subrange.

For example, a subrange 256..511 of unsigned short might be held in 1 byte.

Set Items

A set item is used to describe a Pascal set type. Currently, the description is only partial. After its code and length field, a set item consists of a single word:

size	byte size of the object
------	-------------------------

Enumeration Items

An enumeration item describes a Pascal or C enumerated type. After its code and length word, the description of a 'contiguous enumeration' contains the following word-sized fields

type	a type word describing the type of the container for the enumeration (see <i>Representation of Data Types</i> on page 282)
count	the cardinality of the enumeration
base	the first (lowest) value (may be -ve)
nametable	a character array containing 'count' names (see <i>Text Names in Items</i> on page 284) (the first byte of name is the name's length, followed by a non-NULL-terminated string of characters with NULL padding up to the next word boundary)

The description of a discontinuous enumeration (such as the C enumeration enum bits {bit0=1, bit1=2, bit2=4, bit3=8, bit4=16}) contains the following fields after its code and length word:

type	as above
count	as above
nametable	a table of count (value, name) pairs

Each nametable entry has the following format (which is variable in length):

val	a word describing the enumerated value (1/2/4/8/16 in the example)
name	the name of the enumerated element (may be several words long) (the first byte of name is the name's length, followed by a non-NULL-terminated string of characters with NULL padding up to the next word boundary)

Function Declaration Items

After its code and length word, a function declaration item contains the following fields:

type	a type word (see <i>Representation of Data Types</i> on page 282) describing the return type of the function or procedure
argcount	the number of arguments to the function
args	a sequence of argcount argument description items

Each argument description item contains the following:

type	a type word (see <i>Representation of Data Types</i> on page 282) describing the type of the argument
name	the name of the argument (may be several words) (the first byte of name is the name's length, followed by a non-NULL-terminated string of characters with NULL padding up to the next word boundary)

An argument descriptor need not be named; in this case the length of the name is zero, and the name field is a single zero word.

Begin and End Naming Scope Items

These debug items are used to mark the beginning and end of a naming scope. They must be properly nested in the debug area.

In each case, after the code and length word, there is one word-sized field:

codeaddress	address of the start/end of scope (determined by the code word)
-------------	---

Bitfield Items

A bitfield item describes an individual bitfield member of a C structure. After its code and length word, a bitfield item contains the following fields:

type	a type word describing the type of the bitfield (see <i>Representation of Data Types</i> on page 282)
container	a type word describing the type of the container for the bitfield
size	a byte giving the size of the bitfield, in bits
offset	a byte giving the offset of the bitfield within the container
zero	2 zero bytes

The offset is the offset of the least-significant bit of the bitfield from the least significant bit of its container.

Fileinfo Items

A fileinfo item appears once per section, after all other debugging data items. If the fileinfo item is too large for its length to be encoded in 16 bits, its length field must be written as 0 (since this is the last item in a section and the section header contains the length of the whole section, the length field is strictly redundant).

Each source file is described by a sequence of fragments. Each fragment describes a contiguous region of the file, within which the addresses of compiled code increase monotonically with source file position. The order in which fragments appear in the sequence is not necessarily related to the source file positions to which they refer.

Note that for compilations which make no use of the `#include` facility, the list of fragments may have only one entry, and all line-number information can be contiguous.

After its code and length word, the fileinfo item is a sequence of file entry items with the following format:

len	length of this entry in bytes (including the length of the following fragments)
date	date and time when the file was last modified may be 0, indicating not available, or unused)
filename	string (or "" if the name is not known) (the first byte of string is the string's length, followed by a non-NULL-terminated string of characters with NULL padding up to the next word boundary)
fragment data	see below

If present, the date field contains the number of seconds since the beginning of 1970 (the Unix date origin).

Following the final file entry item, is a single 0 word marking the end of the sequence.

The fragment data is a word giving the number of following fragments followed by a sequence of fragment items:

n	number of fragments following
fragments...	n fragment items

Each fragment item consists of 5 words, followed by a sequence of byte pairs and half word pairs, formatted as follows:

size	length of this fragment in bytes (including length of following lineinfo items)
firstline	linenumber
lastline	linenumber
codestart	pointer to the start of the fragment's executable code
codesize	byte size of the code in the fragment
lineinfo...	a variable number of bytes matching line numbers to code addresses

Each lineinfo item describes a source statement and consists of a pair of (unsigned) bytes, possibly followed by a two or three (unsigned) half words, (each half word has the byte ordering appropriate to the target memory system's endian-ness or byte sex).

The short form (pair of bytes) lineinfo item is as follows:

codeinc	# bytes of code generated by this statement
lineinc	# source space occupied by this statement

lineinc describes how to calculate the source position (line, column) of the next statement from the source position of this one:

If lineinc is in the range $0 \leq$ and < 64 , the new position is (line+lineinc,1).

If lineinc ≥ 64 , the new position is (line, column+lineinc -64).

The number of bytes of code generated for a statement may be zero, provided the line increment is non-zero (such an item may describe a block end or block start, for example).

It is not possible to describe a statement which generates no code and no line number increment, as that encoding is used as an escape to the long form lineinfo items described below.

If codeinc is greater than 255, or lineinc is required to describe a line number change greater than 63 or a column change greater than 191, then both bytes are written to describe 0 increments, and the real values are given in the following two or three (unsigned) half words. (Note that there are two ways to describe 0 increments: 0 lines and 0 columns, which serves to discriminate between the two half word and three half word forms). If the starting column for the next statement is 1, the two half word form is used, which in effect is a triple of half words as follows:

zero	2 zero bytes
lineinc	# source lines occupied by this statement
codeinc	# bytes of code generated by this statement

Note that the order of the lineinc and codeinc half words is the reverse of the corresponding bytes.

If the starting column for the next statement is not 1, the three half word form is used, which in effect is a quadruple of half words, as follows:

	codeinc = 0, lineinc = 64
lineinc	# source lines occupied by this statement
codeinc	# bytes of code generated by this statement
newcol	starting column for the next statement

Note as above that the order of the lineinc and codeinc half words is the reverse of the corresponding bytes. Note also that the column item here is the absolute column number for the next statement, and not an increment as in the two byte form.

(This encoding of lineinfo items is an incompatible change from the previous format (version 2): in that format, lineinc in a two byte lineinfo item always describes a line increment, and accordingly, there is no four half word form. Programs interpreting asd tables should interpret lineinfo items differently according to the table format in the section item.)

Appendix G: ARM procedure call standard

This Appendix relates to the implementation of compiler code-generators and language run-time library kernels for the Advanced RISC Machine (ARM) but is also a useful reference when interworking assembly language with high level language code.

The reader should be familiar with the ARM's instruction set, floating-point instruction set and assembler syntax before attempting to use this information to implement a code-generator. In order to write a run-time kernel for a language implementation, additional information specific to the relevant ARM operating system will be needed (some information is given in the sections describing the standard register bindings for this procedure-call standard).

The main topics covered in this Appendix are the procedure call and stack disciplines. These disciplines are observed by Acorn's C language implementation for the ARM and, eventually, will be observed by other high level language compilers too. Because C is the first-choice implementation language for RISC OS applications, the utility of a new language implementation for the ARM will be related to its compatibility with Acorn's implementation of C.

At the end of this document are several examples of the usage of this standard, together with suggestions for generating effective code for the ARM.

The version of APCS described in this appendix is the version supported by the C compiler and other tools. ARM Limited have defined a newer version called AAPCS, details of which are available from the ARM web site, but this is not currently supported by the RISC OS tools.

The purpose of APCS

The ARM Procedure Call Standard (APCS) is a set of rules which regulate and facilitate calls between separately compiled or assembled program fragments.

The APCS defines:

- constraints on the use of registers
- stack conventions

- the format of a stack-based data structure, used by stack tracing programs to reconstruct the sequence of outstanding calls (i.e. nested function calls awaiting completion)
- the passing of machine-level arguments, and the return of machine-level results at externally visible function/procedure calls
- support for the ARM shared library mechanism; a standard way for shared (reentrant) code to address the static data of its clients.

Since the ARM CPU is used in a wide variety of systems, the APCS is not a single standard, but a consistent family of standards. See APCS *variants* on page 307 for details of the variants in the family. Implementors of run-time systems, operating systems, embedded control monitors, etc., must choose the variant(s) most appropriate to their requirements.

Naturally, there can be no binary compatibility between program fragments which conform to different members of the APCS family. Those concerned with long-term binary compatibility must choose their options carefully.

Note: 'function' is used to mean function, procedure or subroutine.

Design criteria

Throughout its history, the APCS has compromised between fastest, smallest and easiest to use.

The criteria considered to be important are:

- Function call should be fast and it should be easy for compilers to optimise function entry sequences.
- The function call sequence should be as compact as possible.
- Extensible stacks and multiple stacks should be accommodated.
- The standard should encourage the production of reentrant code, with writable data separated from code.
- The standard should be simple enough to be used by assembly language programmers, and should support simple approaches to link editing, debugging and run-time error diagnosis.

Overall, compact code and a clear definition have been ranked most highly, with simplicity and ease of use ahead of performance in matters of fine detail where the impact on performance is small.

The ARM Procedure Call Standard

This section defines the ARM Procedure Call Standard.

A program fragment which conforms to the APCS while making a call to an external function (one which is visible between compilation units) is said to be *conforming*. A program which conforms to the APCS at all instants of execution is said to be ‘strictly conforming’ or to ‘conform strictly’.

Note: In general, compiled code is expected to be strictly conforming; hand-written code merely conforming.

Whether or not (and when) program fragments for a particular ARM-based environment are required to conform strictly to the APCS is part of the definition of that environment.

In the following sections, clauses following ‘shall’ and ‘shall not’ are obligations which must be met in order to conform to the APCS.

Register names

The ARM has 15 visible general registers, a program counter register and 8 floating-point registers.

In non-user machine modes, some general registers are shadowed. In all modes, the availability of the floating-point instruction set depends on the processor model, hardware and operating system.

General registers

Name	Number	APCS Role
a1	0	argument 1 / integer result / scratch register
a2	1	argument 2 / scratch register
a3	2	argument 3 / scratch register
a4	3	argument 4 / scratch register
v1	4	register variable
v2	5	register variable
v3	6	register variable
v4	7	register variable
v5	8	register variable
sb/v6	9	static base / register variable
sl/v7	10	stack limit / stack chunk handle / reg. variable
fp	11	frame pointer
ip	12	scratch register / new-sb in inter-link-unit calls
sp	13	lower end of current stack frame
lr	14	link address / scratch register
pc	15	program counter

The 16 integer registers are divided into 3 sets:

- 4 argument registers which can also be used as scratch registers or as caller-saved register variables;
- 5 callee-saved registers, conventionally used as register variables;
- 7 registers which have a dedicated role, at least some of the time, in at least one variant of APCS-3 (see APCS *variants* on page 307).

The 5 frame registers fp, ip, sp, lr and pc have dedicated roles in all variants of the APCS.

The ip register has a dedicated role only during function call; at other times it may be used as a scratch register.

Note: Conventionally, ip is used by compiler code generators as the/a local code generator temporary register.

There are dedicated roles for sb and sl in some variants of the APCS; in other variants they may be used as callee-saved registers.

The APCS permits lr to be used as a register variable when not in use during a function call. It further permits an ARM system specification to forbid such use in some, or all, non-user ARM processor modes.

Floating point registers

Each ARM floating-point (FP) register holds one FP value of single, double, extended or internal precision. A single-precision value occupies 1 machine word; a double-precision value 2 words; an extended precision value occupies 3 words, as does an internal precision value.

Name	Number	APCS Role
f0	0	FP argument 1 / FP result / FP scratch register
f1	1	FP argument 2 / FP scratch register
f2	2	FP argument 3 / FP scratch register
f3	3	FP argument 4 / FP scratch register
f4	4	floating point register variable
f5	5	floating point register variable
f6	6	floating point register variable
f7	7	floating point register variable

The floating-point (FP) registers are divided into two sets, analogous to the subsets a1-a4 and v1-v5/v7 of the general registers:

- registers f0-f3 need not be preserved by called functions; f0 is the FP result register and f0-f3 may hold the first four FP arguments (see *Data representation and argument passing* on page 305 and *APCS variants* on page 307)
- registers f4-f7, the so called ‘variable’ registers, preserved by callees.

The Stack

The stack is a singly-linked list of ‘activation records’, linked through a ‘stack backtrace data structure’ (see below), stored at the high-address end of each activation record.

The stack shall be readable and writable by the executing program.

Each contiguous chunk of the stack shall be allocated to activation records in descending address order. At all instants of execution, sp shall point to the lowest used address of the most recently allocated activation record.

There may be multiple stack chunks, and there are no constraints on the ordering of these chunks in the address space.

Associated with sp is a possibly-implicit stack chunk limit, below which sp shall not be decremented (see *APCS variants* on page 307).

At all instants of execution, the memory between sp and the stack chunk limit shall contain nothing of value to the executing program: it may be modified unpredictably by the execution environment.

The stack chunk limit is said to be implicit if chunk overflow is detected and handled by the execution environment. Otherwise it is explicit.

If the stack chunk limit is implicit, `sl` may be used as `v7`, an additional callee-saved variable register.

If the conditions of the remainder of this subsection hold at all instants of execution, then the program conforms strictly to the APCS; otherwise, if they hold at and during external (inter-compilation-unit-visible) function calls, the program merely conforms to the APCS.

If the stack chunk limit is explicit, then:

- `sl` shall point at least 256 bytes above it
- `sl` shall identify the current stack chunk in a system-defined manner
- at all times, `sl` shall identify the same chunk as `sp` points into.

Note: $sl \geq \text{stack chunk limit} + 256$ allows the most common limit checks to be made very cheaply during function entry.

This final requirement implies that on changing stack chunks, `sl` and `sp` must be loaded simultaneously by means of an:

```
LDM ..., {..., sl, sp}.
```

In general, this means that return from a function executing on an extension chunk, to one executing on an earlier-allocated chunk, should be via an intermediate function invocation, specially fabricated when the stack was extended.

The values of `sl`, `fp` and `sp` shall be multiples of 4.

The stack backtrace data structure

The value in `fp` shall be zero or shall point to a list of stack backtrace data structures which partially describe the sequence of outstanding function calls.

If this constraint holds when external functions are called, the program is conforming; if it holds at all instants of execution, the program is strictly conforming).

The stack backtrace data structure has the format shown below:

fp points to here:	save code pointer	[fp]
	return link value	[fp, #-4]
	return sp value	[fp, #-8]
	return fp value	[fp, #-12]
Optional values	saved v6 value	
	saved v5 value	
	saved v4 value	
	saved v3 value	
	saved v2 value	
	saved v1 value	
	saved a4 value	
	saved a3 value	
	saved a2 value	
	saved a1 value	
	saved f7 value	three words
	saved f6 value	three words
	saved f5 value	three words
	saved f4 value	three words

The above picture shows between four and twenty-six words, with those words higher on the page being at higher addresses in memory. The values shown inside the large brackets are optional, and their presence need not imply the presence of any other. The floating point values are stored in an internal format, and occupy three words each.

Function invocations and backtrace structures

If function invocation A calls function B, then A is called a *direct ancestor* of the invocation of B. If invocation A[i] calls invocation A[2] calls... calls B, then each of the A[i] is an ancestor of B and invocation A[i] is 'more recent' than invocation A[j] if $i > j$.

The **return fp value** shall be 0, or shall be a pointer to a stack backtrace data structure created by an ancestor of the function invocation which created the backtrace structure pointed to by fp. No more recent ancestor shall have created a backtrace structure.

Note: There may be any number of tail-called invocations between invocations which create backtrace structures.

The **return link value**, **return sp value** and **return fp value** are, respectively, the values to restore to pc, sp and fp at function exit.

In the 32-bit PC variant of the APCS, the **save code pointer** shall point twelve bytes beyond the start of the sequence of instructions that created the stack backtrace data structure.

In the 26-bit PC variant of the APCS, the **save code pointer**, when cleared of PSR and mode bits, shall point twelve bytes beyond the start of the sequence of instructions that created the stack backtrace data structure.

Control arrival

At the instant when control arrives at the target function:

- pc contains the address of an entry point to the target function (reentrant functions may have two entry points).
- lr shall contain the value to restore to pc on exit from the function (the **return link value** – see *The stack backtrace data structure* on page 302)
Note: In 26-bit variants of the APCS, lr contains the PC + PSR value to restore to pc on exit from the function (see *APCS variants* on page 307)
- sp shall point at or above the current stack chunk limit; if the limit is explicit, it shall point at least 256 bytes above it (see *The Stack* on page 301)
- fp shall contain 0 or shall point to the most recently created stack backtrace structure (see *The stack backtrace data structure* on page 302)
- the space between sp and the stack chunk limit shall be readable, writable memory which can be used by the called function as temporary workspace, and overwritten with any values before the function returns (see *The Stack* on page 301)
- arguments shall have been marshalled as described below.

If the target function is reentrant (see *The Stack* on page 301) then it has two entry points and control arrives:

- at the ‘intra-link-unit entry point’ if the caller has been directly linked with the callee
- at the ‘inter-link-unit entry point’ if the caller has been separately linked with a ‘stub’ of the callee.

Note: Sometimes the two entry points are at the same address; usually they will be separated by a single instruction.

On arrival at the intra-link-unit entry point, sb shall identify the static data of the link unit which contains both the caller and the callee.

On arrival at the inter-link-unit entry point, ip shall identify the static data of the link unit containing the target function, or the target function shall make neither direct nor indirect use of static data.

In practice this usually means the callee must be a leaf function making no direct use of static data.

The way in which sb ‘identifies’ the static data of a link unit is not specified by the APCS.

If the call is by tail continuation, ‘calling function’ means that which would be returned to, were the tail continuation converted to a return).

If code is not required to be reentrant or sharable then sb may be used as v6, an additional variable register.

Data representation and argument passing

Argument passing in the APCS is defined in terms of an ordered list of machine-level values passed from the caller to the callee, and a single word or floating point result passed back from the callee to the caller. Each value in the argument list shall be:

- a word-sized, integer value
- a floating point value (of size 1, 2 or 3 words).

A callee may corrupt any of its arguments, howsoever passed.

Note: The APCS does not define the layout in store of records, arrays and so forth, used by ARM-targeted compilers for C, Pascal, Fortran-77, etc.; nor does it prescribe the order in which language-level arguments are mapped into their machine-level representations. In other words, the mapping from language-level data types, and arguments to APCS words is defined by each language implementation, not by the APCS. Indeed, there is no formal reason why two ARM-targeted implementations of the same language should not use different mappings and, hence, not support cross-calling. Obviously, it would be very unhelpful to stand by this formal position so implementors are encouraged to adopt not just the letter of the APCS but also the natural mappings of source language objects into argument words. Guidance about this is given in *C Language calling conventions* on page 309.

At the instant control arrives at the target function, the argument list shall be allocated as follows:

- In APCS variants which support the passing of floating-point arguments in floating-point registers (see *APCS variants* on page 307), the first four floating-point arguments (or fewer if the number of floating-point arguments is less than four) shall be in machine registers $\text{f}0\text{-f}3$.
- The first four remaining argument words (or fewer if there are fewer than four argument words remaining in the argument list) shall be in machine registers $\text{a}1\text{-a}4$.

- The remainder of the argument list (if any) shall be in memory, at the location addressed by `sp` and higher-addressed words thereafter.

A floating-point value not passed in a floating-point register is treated as 1, 2 or 3 integer values, as appropriate to its precision.

Control return

When the return link value for a function call is placed in the `pc`:

- `sp`, `fp`, `s1/v7`, `sb/v6`, `v1-v5`, and `f4-f7` shall contain the same values as they did at the instant of control arrival
- if the function returns a simple value of size one word or less, then that value shall be in `a1`
Note: a language implementation is not obliged to consider **all** single-word values simple. See *C Language calling conventions* on page 309)
- if the function returns a simple floating point value then that value shall be in `f0`.

The values of `ip`, `lr`, `a2-a4`, `f1-f3` and any stacked arguments are undefined.

The definition of control return means that this is a ‘callee saves’ standard.

Note: In 32-bit ARM modes, the caller’s PSR flags are not preserved across a function call. In 26-bit ARM modes, the caller’s PSR flags are naturally reinstated when the return link pointer is placed in `pc`. Note that the `N`, `Z`, `C` and `V` flags from `lr` at the instant of entry must be reinstated; it is not sufficient merely to preserve the PSR across the call. Consider, a function `ProcA` which tail continues to `ProcB` as follows:

```
CMP      a1, #0
MOVLT    a2, #255
MOVGE    a2, #0
B         ProcB
```

If `ProcB` merely preserves the flags it sees on entry, rather than restoring those from `lr`, the wrong flags may be set when `ProcB` returns direct to `ProcA`’s caller. See *APCS variants* on page 307).

APCS variants

There are, currently, $2 \times 2 \times 2 \times 2 = 16$ APCS variants, derived from four independent choices.

The first choice – 32-bit PC vs 26-bit PC – is determined by whether it is necessary to support 26-bit CPU modes.

The second choice – implicit vs explicit stack-limit checking – is fixed by a combination of memory-management hardware and operating system software: if your ARM-based environment supports implicit stack-limit checking then use it; otherwise use explicit stack-limit checking.

The third choice – of how to pass floating-point arguments – supports efficient argument passing in both of the following circumstances:

- the floating point instruction set is emulated by software and floating point operations are dynamically very rare
- the floating point instruction set is supported by hardware or floating point operations are dynamically common.

In each case, code conforming to one variant is not compatible with code conforming to the other.

Only the choice between reentrant and non-reentrant variants is a true user level choice. Further, as the alternatives are compatible, each may be used where appropriate.

32-bit PC vs 26-bit PC

Older ARM CPUs and the 26-bit compatibility mode of newer CPUs use a 24-bit, word-address program counter, and pack the 4 status flags (NZCV) and 2 interrupt-enable flags (IF) into the top 6 bits of r15 , and the 2 mode bits (m0 , m1) into the least-significant bits of r15 . Thus r15 implements a combined PC + PSR.

Newer ARM CPUs use a 32-bit program counter (in r15) and a separate PSR.

In 26-bit CPU modes, the PC + PSR is written to r14 by an ARM branch with link instruction, so it is natural for the APCS to require the reinstatement of the caller's PSR at function exit (a caller's PSR is preserved across a function call).

In 32-bit CPU modes this reinstatement would be unacceptably expensive in comparison to the gain from it, so the APCS does not require it and a caller's PSR flags may be corrupted by a function call.

Implicit vs explicit stack-limit checking

ARM-based systems vary widely in the sophistication of their memory management hardware. Some can easily support multiple, auto-extending stacks, while others have no memory management hardware at all.

Safe programming practices demand that stack overflow be detected.

The APCS defines conventions for software stack-limit checking sufficient to support efficiently most requirements (including those of multiple threads and chunked stacks).

The majority of ARM-based systems are expected to require software stack-limit checking.

Floating-point arguments in floating-point registers

Historically, many ARM-based systems have made no use of the floating point instruction set, or they used a software emulation of it.

On systems using a slow software emulation and making little use of floating-point, there is a small disadvantage to passing floating-point arguments in floating-point registers: all variadic functions (such as `printf`) become slower, while only function calls which actually take floating-point arguments become faster.

If your system has no floating-point hardware and is expected to make little use of floating point, then it is better not to pass floating-point arguments in floating-point registers. Otherwise, the opposite choice is best.

Reentrant vs non-reentrant code

The reentrant variant of the APCS supports the generation of code free of relocation directives (position independent and addressing all data (indirectly) via a static base register). Such code is ideal for placement in ROM and can be multiply threaded (shared between several client processes).

In general, code to be placed in ROM or loaded into a shared library is expected to be reentrant, while applications are expected not to be.

See also *C Language calling conventions* on page 309.

APCS-2 compatibility

APCS-2 – the second definition of The ARM Procedure Call Standard – is described in the *RISC OS Programmer's Reference Manual*.

APCS-R (APCS-2 for Acorn's RISC OS) is the following variant of APCS-3:

- 26-bit PC

- explicit stack-limit checking
- no passing of floating-point arguments in floating-point registers
- non-reentrant code

with the Acorn-specific constraints on the use of `s1` noted in APCS-2.

APCS-U (APCS-2 for Acorn's RISCiX) is the following variant of APCS-3:

- 26-bit PC
- implicit stack-limit checking (with `s1` reserved to Acorn)
- no passing of floating-point arguments in floating-point registers
- non-reentrant code.

The (in APCS-2) obsolescent APCS-A has no equivalent in APCS-3.

C Language calling conventions

Argument representation

A floating point value occupies 1, 2, or 3 words, as appropriate to its type. Floating point values are encoded in IEEE 754 format, with the most significant word of a double having the lowest address.

The C compiler widens arguments of type `float` to type `double` to support inter-working between ANSI C and classic C.

`Char`, `short`, `pointer` and other integral values occupy 1 word in an argument list. `Char` and `short` values are widened by the C compiler during argument marshalling.

On the ARM, characters are naturally unsigned. In `-pcc` mode, the C compiler treats a plain `char` as signed, widening its value appropriately when used as an argument, (classic C lacks the signed `char` type, so plain `chars` are considered signed; ANSI C has signed, unsigned and plain `chars`, the third, conventionally reflecting the natural signedness of characters).

A structured value occupies an integral number of integer words (even if it contains only floating point values).

Argument list marshalling

Argument values are marshalled in the order written in the source program.

If passing floating-point (FP) arguments in FP registers, the first 4 FP arguments are loaded into FP registers.

The first 4 of the remaining argument words are loaded into a1-a4, and the remainder are pushed on to the stack in reverse order (so that arguments later in the argument list have higher addresses than those earlier in the argument list). As a consequence, a FP value can be passed in integer registers, or even split between an integer register and the stack.

This follows from the need to support variadic functions, (functions having a variable number of arguments, such as printf, scanf, etc.). Alternatives which avoid the passing of FP values in integer registers require that a caller know that a variadic function is being called, and use different argument marshalling conventions for variadic and non-variadic functions.

Non-simple value return

A non-simple type is any non-floating-point type of size greater than 1 word (including structures containing only floating-point fields), and certain 1 word structured types.

A structure is called integer-like if its size is less than or equal to one word, and the offset of each of its addressable sub-fields is zero. An integer-like structured result is considered simple and is returned in a1.

```
struct {int a:8, b:8, c:8, d:8;} and  
union {int i; char *p;} are both integer-like;
```

```
struct {char a; char b; char c; char d;} is not.
```

A multi-word or non-integer-like result is returned to an address passed as an additional first argument to the function call. At the machine level:

```
TT tt = f(x, ...);
```

is implemented as:

```
TT tt; f(&tt, x, ...);
```

Function entry

A complete discussion of function entry is complex; a few of the most important issues and special cases are discussed here.

The important issues for function entry are:

- establishing the static base (if the function is to be reentrant)
- creating the stack backtrace data structure (if needed)
- saving the floating point variable registers (if required)
- checking for stack overflow (if the stack chunk limit is explicit).

A function is called **leaf** if its body contains no function calls.

If function F calls function G immediately before an exit from F, the call- exit sequence can often be replaced instead by a **return to G**. After this transformation, the return to G is called a *tail call* or *tail continuation*.

There are many subtle difficulties with tail continuations. Suppose stacked arguments are unstacked by callers (almost mandatory for variadic callees), then G cannot be directly tail-called if G itself takes stacked arguments. This is because there is no return to F to unstack them. Of course, if this call to G takes fewer arguments than the current call to F, then some of F's stacked arguments can be replaced by G's stacked arguments. However, this can be hard to assert if F is variadic. More straightforwardly, there may be no tail-call of G if the address of any of F's arguments or local variables has 'leaked out' of F. This is because on return to G, the address may be invalidated by adjustment of the stack pointer. In general, this precludes tail calls if any local variable or argument has its address taken.

If a function is a leaf function, or all function calls from its body are tail calls and, in both cases, the function uses no v-registers (v1–v7) then the function need create no stack backtrace structure (such functions will also be termed 'frameless').

A leaf function which makes no use of static data need not establish a static base.

Function entry - establishing the static base

The ARM shared library mechanism supports both the direct linking together of functions into a **link unit**, and the indirect linking of functions with the stubs of other link units. Thus a reentrant function can be entered directly via a call from the same link unit (an intra-link-unit call), or indirectly via a function pointer or direct call from another link unit (an inter-link-unit call).

The general scheme for establishing the static base in reentrant code is:

```
intra MOV ip, sb ; intra link unit (LU) calls target here
inter          ; inter-LU calls target here, having loaded
               ; ip via an inter-LU or fn-pointer veneer.

<create backtrace structure, saving sb>

MOV sb, ip ; establish sb for this LU

<rest of entry>
```

Code which is not required to be reentrant need not use a static base. Code which is reentrant is marked as such, which allows the linker to create the inter-LU veneers needed between independent reentrant link units, and between reentrant and non-reentrant code.

Function entry - creating the stack backtrace structure

For non-reentrant, non-variadic functions the stack backtrace structure can be created in just 3 instructions, as follows:

```
MOV    ip, sp      ; save current sp, ready to save as old sp
STMFD  sp!, {a1-a4, v1-v5, sb, fp, ip, lr, pc} ; as needed
SUB    fp, ip, #4
```

Each argument register a1-a4 need only be saved if a memory location is needed for the corresponding parameter (because it has been spilled by the register allocator or because its address has been taken).

Each of the registers v1-v7 need only be saved if it used by the called function. The minimum set of registers to be saved is {fp, old-sp, lr, pc}.

A reentrant function must avoid using ip in its entry sequence:

```
STMFD  sp!, {sp, lr, pc}
STMFD  sp!, {a1-a4, v1-v5, sb, fp}          ; as needed
ADD    fp, sp, #8+4*|{a1-a4, v1-v5, sb, fp}| ; as used above
```

sb (aka v6) must be saved by a reentrant function if it calls any function from another link unit (which would alter the value in sb). This means that, in general, sb must be saved on entry to all non-leaf, reentrant functions.

For variadic functions the entry sequence is more complicated again. Usually, it will be desired or required to make a contiguous argument list on the stack. For non-reentrant variadic functions this can be done by:

```
MOV    ip, sp      ; save current sp, ready to save as old sp
STMFD  sp!, {a1-a4} ; push arguments on stack
SFMFD  f0, 4, [sp]   ; push FP arguments on stack...
STMFD  sp!, {v1-v6, fp, ip, lr, pc}          ; as needed
SUB    fp, ip, #20   ; if all of a1-a4 pushed...
```

It is not necessary to push arguments corresponding to fixed parameters (though saving a1-a4 is little more expensive than just saving, say, a3-a4).

If floating point arguments are not being passed in floating point registers then there is no need for the SFMFD. SFM is not supported by the issue-1 floating-point instruction set and must be simulated by 4 STFES. See *Function entry - saving and restoring floating point registers* below.

For reentrant variadic functions, the requirements are yet more complicated and the sequence becomes less elegant.

Function entry - saving and restoring floating point registers

The issue-2 floating-point instruction set defines two new instructions, **Store Floating Multiple** (SFM) and **Load Floating Multiple** (LFM), for saving and restoring the floating-point registers, as follows:

- SFM and LFM are exact inverses;

- a SFM will never trap, whatever the IEEE trap mode and the value transferred (unlike a STFE which can trap on storing a signalling NaN);
- SFM and LFM transfer 3-word internal representations of floating point values which vary from implementation to implementation, and which, in general, are unrelated to any of the supported IEEE representations;
- any 1-4, cyclically contiguous floating-point registers can be transferred by SFM/LFM (e.g. {f4-f7}, {f6, f7, f0}, {f7, f0}, {f1}).

On function entry, a typical use of SFM might be as follows:

```
SFMFD  f4, 4, [sp]!      ; save f4-f7 on a Full Descending stack,
                        ; adjusting sp as values are pushed.
```

On function exit, the corresponding sequence might be as follows:

```
LFMEA  f4, 4, [fp, #-N]  ; restore f4-f7; fp-N points just
                        ; above the floating point save area.
```

On function exit, sp-relative addressing may be unavailable if the stack has been discontiguously extended.

In issue-1 instruction set compatibility modes, SFM and LFM have to be simulated using sequences of STFEs and LDFEs.

Function entry - checking for stack limit violations

In some environments, stack overflow detection will be implicit: an off stack reference will cause an address error or memory fault which may, in turn, cause stack extension or program termination.

In other environments, the validity of the stack must be checked on function entry and, perhaps at other times. There are three cases:

- the function uses 256 bytes or less of stack space
- the function uses more than 256 bytes of stack space, but the amount is known and bounded at compile time
- the function uses an amount of stack space unknown until run time.

The third case does not arise in C, save with stack-based implementations of the non-standard, BSD-Unix `alloca()` function. The APCS does not support `alloca()` in a straightforward manner.

In Modula-2, Pascal and other languages there may be arrays created on block entry or passed as **open array arguments**, the size of which is unknown until run time. These are located in the callee's stack frame, so impact stack limit checking. In practice, this adds little complication, as discussed in *Stack limit checking - vari-sized frames* on page 314.

The check for stack limit violation is made at the end of the function entry sequence, by which time `ip` is available as a work register. If the check fails, a standard run-time support function (`'__rt_stkovf_split_small'` or `'__rt_stkovf_split_big'`) is called. Each environment which supports explicit stack limit checking must provide these functions, which can do one of the following:

- terminate execution
- extend the existing stack chunk, decrementing `sl`
- allocate a new stack chunk, resetting `sp` and `sl` to point into it, and guaranteeing that an immediate repeat of the limit check will succeed.

Stack limit checking - small, fixed frames

For frames of 256 bytes or less the limit check is as follows:

```
<create stack backtrace structure>
CMP     sp, sl
BLMI    |__rt_stkovf_split_small|
SUB     sp, sp, #<size of locals>    ; <= 256, by hypothesis
```

This adds 2 instructions and, in general, only 2 cycles to function entry.

After a call to `__rt_stkovf_split_small`, `fp` and `sp` do not, necessarily, point into the same stack chunk. Arguments passed on the stack must be addressed by offsets from `fp`, not by offsets from `sp`.

Stack limit checking - large, fixed frames

For frames bigger than 256 bytes, the limit check proceeds as follows:

```
SUB     ip, sp, #FrameSizeBound      ; can be done in 1 instr
CMP     ip, sl
BLMI    |__rt_stkovf_split_big|
SUB     sp, sp, #InitFrameSize       ; may take more than 1 instr
```

`FrameSizeBound` can be any convenient constant at least as big as the largest frame the function will use. Note that functions containing nested blocks may use different amounts of stack at different instants during their execution.

`InitFrameSize` is the initial stack frame size: subsequent adjustments within the called function require no limit check.

After a call to `__rt_stkovf_split_big`, `fp` and `sp` do not, necessarily, point into the same stack chunk. Arguments passed on the stack must be addressed by offsets from `fp`, not by offsets from `sp`.

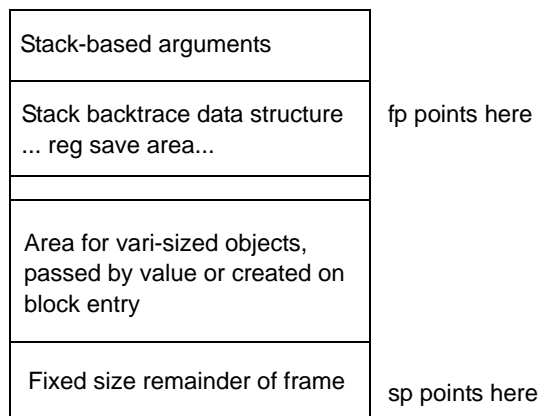
Stack limit checking - vari-sized frames

(For Pascal-like languages).

The handling of frames the size of which is unknown at compile time, is identical to the handling of large frames, save that:

- the computation of the proposed new stack pointer is more complicated, involving arguments to the function itself
- the addressing of the vari-sized objects is more complicated than the addressing of fixed size objects need be
- the vari-sized objects have to be initialised by the called function.

The general scheme for stack layout in this case is as follows:



Objects notionally passed by value are actually passed by reference and copied by the callee.

The callee addresses the copied objects via pointers located in the fixed size part of the stack frame, immediately above sp. These can be addressed relative to sp. The original arguments are all addressable relative to fp.

After a call to `__rt_stkovf_split_big`, fp and sp do not, necessarily, point into the same stack chunk. Arguments passed on the stack must be addressed by offsets from fp, not by offsets from sp.

If a nested block extends the stack by an amount which can't be known until run time then the block entry must include a stack limit check.

Function exit

A great deal of design effort has been devoted to ensuring that function exit can usually be implemented in a single instruction (this is not the case if floating-point registers have to be restored). Typically, there are at least as many function exits as entries, so it is always advantageous to move an instruction from an exit sequence

to an entry sequence, (Fortran may violate this rule by virtue of multiple entries, but on average the rule still holds true). If exit is a single instruction then, in multi-exit functions, further instructions can be saved by replacing branches to a single exit by the exit instructions themselves.

Exit from functions which use no stack and save no floating point registers is particularly simple:

```
MOV    pc, lr
```

(26-bit compatibility demands MOVS pc, lr to reinstate the caller's PSR flags, but this must not be used in 32-bit modes).

Exit from other functions which save no floating-point registers is by:

```
LDMEA  fp, {v1-v5, sb, fp, sp, pc}          ; as saved
```

Here, it is crucial that fp points just below the **save code pointer**, as this value is not restored, (LDMEA is a pre-decrement multiple load). (26-bit compatibility demands LDMEA fp, {regs}^, to reinstate the caller's PSR flags, but this must not be used in 32-bit modes).

The saving and restoring of floating-point registers is discussed above.

Some examples

This section is not intended to be a general guide to the writing of code generators, but it seems worthwhile to highlight some of the optimisations that appear particularly relevant to the ARM and to this standard.

In order to make effective use of the APCS, compilers must compile code a procedure at a time. Line at a time compilation is insufficient.

In the case of leaf functions, much of the standard entry sequence can be omitted. In very small functions, such as those that frequently occur implementing data abstractions, the function-call overhead can be tiny.

Consider:

```
typedef struct {...; int a; ...} foo;
int foo_get_a(foo* f) {return(f-a);}
```

The function foo_get_a can compile to just:

```
LDR    a1, [a1, #aOffset]
MOV    pc, lr                ; MOVS in 26-bit modes
```

In functions with a conditional as the top level statement, in which one or other arm of the conditional is leaf (calls no functions), the formation of a stack frame can be delayed.

For example, the C function:

```
int get(Stream *s)
{
    if (s->cnt > 0)
    { --s;
      return *(s-p++);
    }
    else
    {
        ...
    }
}
```

... could be compiled (non-reentrantly) into:

```
get MOV    a3, a1
; if (s->cnt > 0)
    LDR    a2, [a3, #cntOffset]
    CMP    a2, #0
; try the fast case, frameless and heavily conditionalized
    SUBGT  a2, a2, #1
    STRGT  a2, [a3, #cntOffset]
    LDRGT  a2, [a3, #pOffset]
    LDRBGT a1, [a2], #1
    STRGT  a2, [a3, #pOffset]
    MOVGT  pc, lr
; else, form a stack frame and handle the rest as normal code
    MOV    ip, sp
    STMDB  sp!, {v1-v3, fp, ip, lr, pc}
    CMP    sp, sl
    BLMI   |__rt_stkovf_split_small|
    ...
    LDMEA  fp, {v1-v3, fp, sp, pc}
```

This is only worthwhile if the test can be compiled using any spare of a1-a4 and ip, as scratch registers. This technique can significantly accelerate certain speed-critical functions, such as read and write character.

Finally, it is often worth applying the tail call optimisation, especially to procedures which need to save no registers.

For example:

```
extern void *malloc(size_t n)
{
    return primitive_alloc(NOTGCABLEBIT, BYTESTOWORDS(n));
}
```

... is compiled (non-reentrantly) by the C compiler into:

```
malloc
    ADD    a1, a1, #3           ; 1S
    MOV    a2, a1, LSR #2      ; 1S - BYTESTOWORDS(n)
    MOV    a1, #1073741824     ; 1S - NOTGCABLEBIT
    B      primitive_alloc    ; 1N+2S = 4S
```

In this case, the optimisation avoids saving and restoring the call-frame registers and saves 5 instructions (and many cycles-17 S cycles on an uncached ARM with $N=2S$).

The APCS in non-user ARM modes

There are some consequences of the ARM's architecture which, while not explicit in the ARM Procedure Call Standard, need to be understood by implementors of code intended to run in the ARM's SVC and IRQ modes.

An IRQ corrupts r14_irq, so IRQ-mode code must run with IRQs off until r14_irq has been saved.

A general solution to this problem is to enter and exit IRQ handlers written in high-level languages via hand-crafted wrappers, which on entry save r14_irq, change mode to SVC, and enable IRQs; and on exit restore the saved r14_irq, IRQ mode and the IRQ-enable state. Thus the handlers themselves run in SVC mode, avoiding the problem in compiled code.

SWIs corrupt r14_svc, so care has to be taken when calling SWIs in SVC mode.

In high-level languages, SWIs are usually called out of line, so it suffices to save and restore r14 in the calling veneer around the SWI. If a compiler can generate in-line SWIs, then it should, of course, also generate code to save and restore r14 in-line around the SWI, unless it is known that the code will not be executed in SVC mode.

Aborts and pre-ARM6-based ARMs

With pre-ARM6-based ARMs (ARM2, ARM3), aborts corrupt r14_svc. This means that care has to be taken when causing aborts in SVC mode.

An abort in SVC mode may be symptomatic of a fatal error, or it may be caused by page faulting in SVC mode. Page faulting can occur because an instruction needs to be fetched from a missing page (causing a prefetch abort), or because of an attempted data access to a missing page. The latter may occur even if the SVC-mode code is not itself paged, (consider an unpaged kernel accessing a paged user-space).

A data abort is recoverable provided r14 contains nothing of value at the instant of the abort. This can be ensured by:

- saving R14 on entry to every function and restoring it on exit;
- not using R14 as a temporary register in any function;
- avoiding page faults (stack faults) in function entry sequences.

A prefetch abort is harder to recover from, and an aborting BL instruction cannot be recovered, so special action has to be taken to protect page faulting function calls.

In code compiled from C, r14 is saved in the 2nd or 3rd instruction of an entry sequence. Aligning all functions at addresses which are 0 or 4 modulo 16, ensures the critical part of the entry sequence cannot prefetch-abort. A compiler can do this by padding code sections to a multiple of 16 bytes, and being careful about the alignment of functions within code sections.

Data-aborts early in function entry sequences can be avoided by using a software stack-limit check.

A possible way to protect BL instructions from prefetch-aborts, is to precede each BL by a

```
MOV    ip, pc
```

instruction. If the BL faults, the prefetch abort handler can safely overwrite r14 with ip before resuming execution at the target of the BL. If the prefetch abort is not caused by a BL then this action is harmless, as r14 has been corrupted anyway, (and, by design, contained nothing of value at any instant a prefetch abort could occur).

Index

Symbols

- *DebugIAF 23
- *Filer_OpenDir 99
- *FrontEnd_SetUp 186, 203
- *FrontEnd_Start 186, 188, 197
 - invoking using `command_is` 193
- *IconSprites 188
- *Prefix 238
- *RMEnsure 188
- *RMTidy 160
- *Spool 99
- *WimpSlot 98

A

- a.out format 251
- A8Time
 - Application menu 116
 - command line interface 117
 - menu options
 - Command line 115
 - Output window 116
 - SetUp
 - dialogue box 115
 - menu 115
 - SetUp options
 - File 115
- Acorn Library Format *see* ALF
- Acorn Make Utility *see* AMU
- AIF 151, 247, 271-279
 - debugging 272
 - executable 271
 - layout of an image 273
 - layout of an uncompressed image 273
 - layout of the header 275

- non-executable 271
- relocation 272
- self-move 277
- self-relocation 272, 277
- zero-initialisation 277
- ALF 247, 267-270
 - alignment 248
 - ChunkIndex 268
 - Data 268
 - DataLength 268
 - EntryLength 268
 - LIB_DATA 269
 - LIB_DIRY 267
 - LIB_TIME 269
 - LIB_VSRN 269
 - library file chunks 267
 - object code libraries 270
 - OFL_SYMT 270
 - OFL_TIME 270
 - Time stamps 269
- AMU 121-125
 - Application menu 123
 - command line 124
 - controlling operation 122
 - SetUp dialogue box 122
 - SetUp menu 122
 - specifying makefile to be used 122
 - specifying targets 122
- amu command line tool 121
- AOF 247, 249-266
 - alignment 248
 - area attributes 258
 - area name 253
 - area size 253
 - AREAs 156
 - attributes 160
 - packing 158

-
- attributes and alignment 253
 - chunk file format 249
 - entry address area / entry address offset 252
 - files 143
 - format of area headers 253
 - format of the areas chunk 259
 - format of the symbol table chunk 262
 - header chunk format 251
 - identification chunk (OBJ_IDFN) 266
 - number of areas 252
 - number of relocations 253
 - relocation directives 259
 - string table chunk (OBJ_STRT) 266
 - symbol attributes 263
 - symbol table 252
 - version ID 252
 - APCS 155, 297-319
 - APCS-2 compatibility 308
 - argument passing 305
 - C language calling conventions 309
 - argument list marshalling 309
 - argument representation 309
 - non-simple value return 310
 - data representation 305
 - design criteria 298
 - examples 316
 - function entry 310
 - non-user ARM modes 318
 - purpose 297
 - registers 299
 - floating point 301
 - general 300
 - stack 301
 - stack backtrace 302
 - variants of APCS 307
 - application description
 - arrow icons 197
 - DBOX section 195
 - DESELECTIONS section 199
 - EXCLUSIONS section 200
 - FILEOUTPUT section 195
 - icon default values 197
 - icon types 195
 - INCLUSIONS section 200
 - MAKE_EXCLUSIONS section 200
 - MAKE_ORDER section 201
 - MENU section 198
 - METAOPTIONS section 193
 - ORDER section 201
 - RULES section 200
 - SELECTIONS section 199
 - toggling dialogue box size 197
 - TOOLDETAILS section 193
 - applications
 - adding new ones 185
 - porting to RISC OS 185
 - ARM Object Format *see* AOF
 - ARM Procedure Call Standard *see* APCS
 - arrow icons 197
 - ASD 247, 280-296
 - compilation units (sections) 280
 - data encoding 281
 - data items 283
 - Array item 290
 - code and length field 283
 - Endproc item 287
 - Enumeration item 291
 - Fileinfo item 293
 - Function Declaration item 292
 - Label item 287
 - offsets 284
 - Procedure item 286
 - Scope items 292
 - Section item 284
 - Set item 291
 - Struct item 289
 - Subrange item 291
 - text names 284
 - Type item 289
 - Variable item 288
 - data types 282
 - debug data areas (items) 280
 - endian memory systems 281
 - order of 280
 - Source file position 283
 - sourcepos field 283

- Auto Run option
 - enabling 193
- Auto Save option
 - enabling 194

B

- breakpoints
 - setting 27
 - on addresses and low-level expressions 32
 - on procedure names 27
- byte
 - definition 247
 - sex 247

C

- C module header generator (CMHG) 160
- chunk file
 - chunkId 250
 - format 249
 - header entries 250
 - layout 249
 - offset 250
- command line interface 106
 - A8Time 117
 - DecAOF 129
 - Diff 134
 - Find 141
 - LibFile 147
 - Link 162
 - ModSqz 166
 - ObjSize 170
 - Squeeze 174
 - UnModSqz 178
 - Xpand 183
- command lines
 - passing long command lines *see* DDEUtils module
- compiler

- adding a new one 185
- compiling a program
 - with debugging information 22
- conditional macro assignment 215
- Context window 24
- controlling DDT execution 33

D

- DDEUtils module 185, 202, 237
- DDT 19-57
 - accessing nested variables 30
 - breakpoints
 - on addresses and low-level expressions 32
 - on procedure names 27
 - Context window 24
 - enabling debugging 22
 - error messages 24
 - example session 50
 - execution control 33
 - limitations 19
 - linking a program 22
 - main menu 26
 - menu options
 - *Commands 48
 - Breakpoint 36
 - Call 35
 - Change 44
 - Continue 33
 - Debug 23
 - Display 41
 - Find 48
 - Help 49
 - Log 47
 - Options 46
 - Quit 49
 - Single step 34
 - Trace 39
 - Watchpoint 37
 - menu shortcuts
 - Breakpoint 37

- Continue 33, 49
- Display 41
- Single step 35
- Watchpoint 39
- preparing a program 21
- program examination and modification 41
- specifying program objects 26
- starting a debugging session 23
- Status window 24
- variable length arrays 30
- watchpoints
 - on variable names 28
- debugging
 - source-level 22
- debugging *see also* DDT (desktop debugging tool)
- DecAOF
 - Application menu 128
 - command line interface 129
 - menu options
 - Command line 128
 - Output window 129
 - SetUp
 - dialogue box 127
 - menu 128
 - SetUp options
 - Area contents 127
 - Area declarations 128
 - Debug 127
 - Files 127
 - Only area declarations 127
 - Relocation directives 128
 - String table 127
 - Symbol table 127
- desktop utility
 - adding a new one 185
- Diff
 - Application menu 132
 - command line interface 134
 - menu options
 - Command line 132
 - Dir. structure 132
 - Equate CR/LF 132
 - Expand tabs 132

- Fast 132
- Large files 132
- Squidge 132
- Output window 133
- SetUp
 - dialogue box 131
 - menu 132
- SetUp options
 - Case insensitive 131
 - Expand tabs 131
 - Remove spaces 131
 - Squash spaces 131

E

- EBNF rule, for application 192
- Entry points *see* Link menu options
- environment variables 8
 - C\$Path 8
 - CLib\$Path 8
 - DDE\$Path 8
 - Hdr\$Path 8
 - Install\$Dir 9
 - LocalRes\$Path 9
 - Makefiles\$Path 8
 - RISCOSLib\$Path 8
 - TCPIPLibs\$Path 8
- error messages
 - DDT 24
- error throwback 239
- Errors
 - linking a program 155
- EXCLUSIONS 200
- extracting files
 - LibFile 145

F

- file formats
 - AIF 271-279
 - ALF 267-270

- AOF 249-266
- SrcEdit 245
- undefined fields 248
- file type
 - Text 75
- filename prefixing *see* DDEUtils module
- Find
 - Application menu 140
 - command line interface 141
 - menu options
 - Allow 139
 - Command line 139
 - Grep style 139
 - Output window 140
 - SetUp
 - dialogue box 135
 - menu 139
 - SetUp options
 - Case insensitive 136
 - Filenames only 136
 - Files 135
 - Line count only 136
 - Patterns 135
 - Throwback 136
 - Verbose 136
 - Wildcards 136
 - SetUp wildcard filenames
 - 0 or More 139
 - 0 or more filename chs. 138
 - Filename ch. 138
 - Or 139
 - Sub-directories 138
 - SetUp wildcard patterns
 - 0 or more 138
 - 1 or more 138
 - Alphanum 137
 - Any 137
 - Ctrl 137
 - Digit 137
 - Newline 137
 - Normal 137
 - Not 137
 - Set 137

- finding
 - text in a file 80
- fonts *see* SrcEdit (fonts)
- format of AOF area headers 253
- FrontEnd
 - !Run, !Boot and !SetUp files 188
 - producing new RISC OS applications 186
- FrontEnd module 185, 186-202
 - operation when command line tool is run 186

H

- half word
 - definition 247
- hardware and OS requirements for Acorn C/C++ 7

I

- icon types 195
- IMPORT directive 160
- installing Acorn C/C++ 7
 - hardware requirement 7
- invoking a WIMP frontend for a tool 186

K

- KEEP directive 23

L

- language processors – output format 247
- LIB_DATA 269
- LIB_DIRY 267
- LIB_TIME 269
- LIB_VSRN 269
- LibFile 143-148
 - command line interface 147

-
- extracting files 145
 - limitations when creating libraries 146
 - menu options
 - Command line 144
 - List symbol table 144
 - Null timestamps 146
 - Via file 144
 - Output window 145
 - SetUp
 - dialogue box 143
 - menu 144
 - SetUp options
 - Create 143
 - Delete 143
 - Extract 144
 - File list 143
 - Insert 143
 - Library 143
 - List library 144
 - Object library 144
 - libraries
 - linking 155
 - symbol references 155
 - library archives
 - AOF files 143
 - Link 151-164
 - AIF 151
 - command line interface 152, 162
 - errors 155
 - IMPORT directive 160
 - inter-area references 158
 - libraries 155
 - linking with the overlay manager 158
 - loading 151
 - menu options
 - Base 153
 - Command line 152
 - Debug 152
 - Entry 153
 - Link map 152, 154, 158
 - No case 153
 - Overlay 153, 157
 - Relocatable AIF 153
 - Verbose 154
 - Via file 153
 - Workspace 152, 153, 160
 - X-Ref 152, 158
 - Output window 154
 - overlying programs 156
 - predefined symbols 161
 - relocatable AIF images 159
 - relocatable module format (RMF) 151
 - relocatable modules 160
 - SetUp
 - dialogue box 151
 - menu 152
 - SetUp options
 - AIF 151
 - Binary 152
 - Files 151
 - Module 151
 - Relocatable AIF 152
 - specifying files to be linked 151
 - utility programs 161
 - linking
 - preparing to debug a program 22, 152
 - little endian 247
- ## M
- macros
 - conditional assignment 215
 - definitions 213
 - priority 223
 - recursively defined 214
 - simply extended 215
 - Make 14, 203
 - command execution 209-210
 - command line tools 71
 - invoking 59
 - Makefiles
 - conventional Makefiles 69
 - editing 68
 - file naming 224
 - format 69

- specifying 122
- structure 212
- menu options
 - Info 59
 - Open 59
 - Options 59
- MFLAGS macro 228
- Output window 67
- programmer interface 71
- projects 60
 - adding a member 63
 - adding a target 65
 - creating a final target 67
 - creating a new project 61
 - final targets 60
 - listing members 64
 - opening a project 62
 - removing a member 63
 - removing a project 66
 - setting tool options 65
 - touching members 64
- rule patterns 225-227
- tool options, message passing 72
- VPATH macro 225
- WIMP message format 72

Make project management tool 185

makefiles

- directives 221
- functions 216
- macros 213
- substitutions 216

ModSgz

- Application menu 166
- command line interface 166
- menu options
 - Command line 165
- Output window 166
- SetUp
 - dialogue box 165
 - menu 165
- SetUp options
 - Input 165
 - Try harder 165

- Verbose 165

module headers

- creating in assembler 160

multi-tasking

- pre-emptive multi-tasking 186

N

nested variables

- accessing in DDT 30

O

OBJ_

- name of AOF files 250

OBJ_AREA

- areas chunk 259

OBJ_IDFN 266

OBJ_STRT 266

ObjAsm

- KEEP directive 23

object file

- format 250
- chunk names 250
- type 252

ObjSize

- Application menu 169
- command line interface 170
- menu options
 - Command line 169
- Output window 170
- SetUp
 - dialogue box 169
 - menu 169
- SetUp options
 - Files 169

OFL_SYMT 270

OFL_TIME 270

output formats in Link 154

- AIF 151
- binary 152

- RMF 151
- Output window
 - A8Time 116
 - DecAOF 129
 - Diff 133
 - Find 140
 - LibFile 145
 - Link 154
 - ModSqz 166
 - ObjSize 170
 - Squeeze 174
 - UnModSqz 178
 - Xpand 182
- overlay description files 157
- overlay manager
 - linking 158
- overlying programs 156

P

- packing
 - AREAs 158
- parent directories
 - indicating with ^, 194
- porting applications to RISC OS 185
- predefined linker symbols 161
- Prefix\$Dir 227
- procedure names
 - setting breakpoints in DDT 27
- program objects
 - specifying in DDT 26
- project management tool
 - creating 185
- projects *see* MAKE

R

- recursively expanded macros 214
- relocatable AIF images 159
- relocatable module area (RMA) 160
- relocatable module format (RMF) 151

- relocatable modules 160
- relocating applications on the stack
 - the Workspace option 160
- resource files in SrcEdit 203

S

- saving single output object 195
- SetPaths 8
- simply extended macros 215
- source-level debugging 22
- Squeeze
 - Application menu 174
 - command line interface 174
 - menu options
 - Command line 173
 - Output window 174
- SetUp
 - dialogue box 173
 - menu 173
- SetUp options
 - Input 173
 - Try harder 173
 - Verbose 173
- SrcEdit 203
 - Application menu options
 - Create 96
 - Options 96
 - Save All 95
 - Save Options 95
 - Backspace 75
 - block operations 76
 - bracket-matching 90
 - carriage return 87
 - case sensitivity in Find 82
 - colours 88
 - ColTab 89
 - copy a selection 77
 - copying
 - Ctrl-C shortcut 78
 - copying block 78
 - counting occurrences 82

- Ctrl-U 75
- cutting
 - Ctrl-X shortcut 79
- Delete 75
- deleting
 - Ctrl-K shortcut 79
- deleting block 79
- entering text 73
- file formats 245
- find a specific line 86
- finding text 80-85
- fonts 87
- format width 87
- formatting text 87
- Goto
 - line 86
 - option 86
- indenting 77
- inserting/deleting text 74
- keyboard shortcuts 81
- keystroke equivalents 100
- line spacing 88
- linefeed 87
- magic characters 82
- margin 88
- moving block 79
- pasting
 - Ctrl-V shortcut 79
- printing a file 88
- reading text from another file 90
- redoing changes 86
- replacing text 81
- resource files 203
- searching for text 80
- select a block 76
- selected block - saving a 77
- signalling errors via throwback 91
- starting 73
- tabs 87, 89
- task windows 97
- text found dialogue box 80
- text wrap 88
- throwback 203

- undoing changes 79, 81, 86
- wildcarded expressions 83
- window 73

string

- definition 247

SWI

- DDEUtils_FlushCL 239
- DDEUtils_GetCL 239
- DDEUtils_GetCLSize 239
- DDEUtils_Prefix 237
- DDEUtils_ReadPrefix 237
- DDEUtils_SetCL 238
- DDEUtils_SetCLSize 238
- DDEUtils_ThrowbackEnd 241
- DDEUtils_ThrowbackRegister 239
- DDEUtils_ThrowbackSend 240
 - Throwback_ReasonErrorDetails 241
 - Throwback_ReasonInfoDetails 241
 - Throwback_ReasonProcessing 241
- DDEUtils_ThrowbackStart 240
- DDEUtils_ThrowbackUnRegister 240
- WimpInitialise 186

symbol references

- to libraries 155

symbols

- predefined linker symbols 161

T

- targets
 - specifying to AMU 122
- task windows 97
 - using the clipboard 98
- Templates file
 - CmdLine 191
 - Output 191
 - progInfo 190
 - query 191
 - save 192
 - SetUp 190
 - Summary 192
 - Window name 190

- xfer_send 192
- TextFile 75
- Throwback
 - example session 91-93, 94
 - SWIs 239
- throwback 12
 - protocol 239
 - SrcEdit 91
- throwback *see also* DDEUtils module
- Thumb 258, 265, 266
- tool output, specifying default 195
- tools
 - defaults when invoking from Make 197
- tools, interactive 12, 103
 - DDT 19
 - entering filenames 12
 - Make 59
 - SrcEdit 73
- tools, non-interactive 12, 103
 - A8Time 113
 - AMU 121
 - Application menu 104
 - DecAOF 127
 - Diff 131
 - entering filenames 12
 - file output 109
 - Find 135
 - LibFile 143
 - Link 151
 - ModSqz 165
 - ObjSize 169
 - Output windows 107
 - Summary 108
 - Text 107
 - toggleing between 108
 - SetUp dialogue box 105
 - SetUp menu 106
 - SID 119, 171
 - Squeeze 173
 - starting 103
 - UnModSqz 177
 - Xpand 181
- transient utilities 161

U

- UnModSqz
 - Application menu 178
 - command line interface 178
 - menu options
 - Command line 177
 - Output window 178
- SetUp
 - dialogue box 177
 - menu 177
- SetUp options
 - Input 177
 - Verbose 177
- utility programs 161

V

- variable length arrays 30
- variable names
 - setting watchpoints in DDT 28
- via file
 - use in LibFile 144
 - use in Link 153

W

- watchpoints
 - setting 28
- WIMP
 - description file 186
 - frontend, adding to tools 186
 - invoking frontend for a tool 186
 - producing complete WIMP application 187
 - setting MAKE options 186
- wimpslot
 - default 193
 - size 188
- word
 - definition 247
- work directory 13

writing an application description 192

X

Xpand

- Application menu 182
- command line interface 183
- menu options
 - Command line 181
- Output window 182
- SetUp
 - dialogue box 181
 - menu 181
- SetUp options
 - Input 181



Reader's Comment Form

Desktop Tools, Issue 4

We would greatly appreciate your comments about this Manual, which will be taken into account for the next issue:

Did you find the information you wanted?

Do you like the way the information is presented?

General comments:

If there is not enough room for your comments, please continue overleaf

How would you classify your experience with computers?

☐

Used computers before

☐

Experienced User

☐

Programmer

☐

Experienced Programmer

*Please send an e-mail with your
comments to:*

ctools@riscosopen.org

Your name and address:

This information will only be used to get in touch with you in case we wish to explore your
comments further