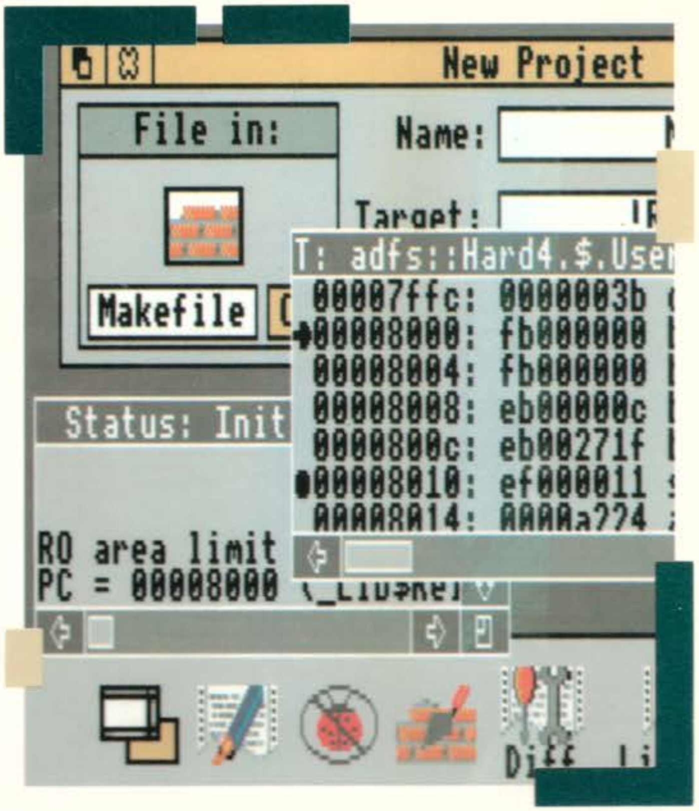
# ACORN DESKTOP DEVELOPMENT ENVIRONMENT

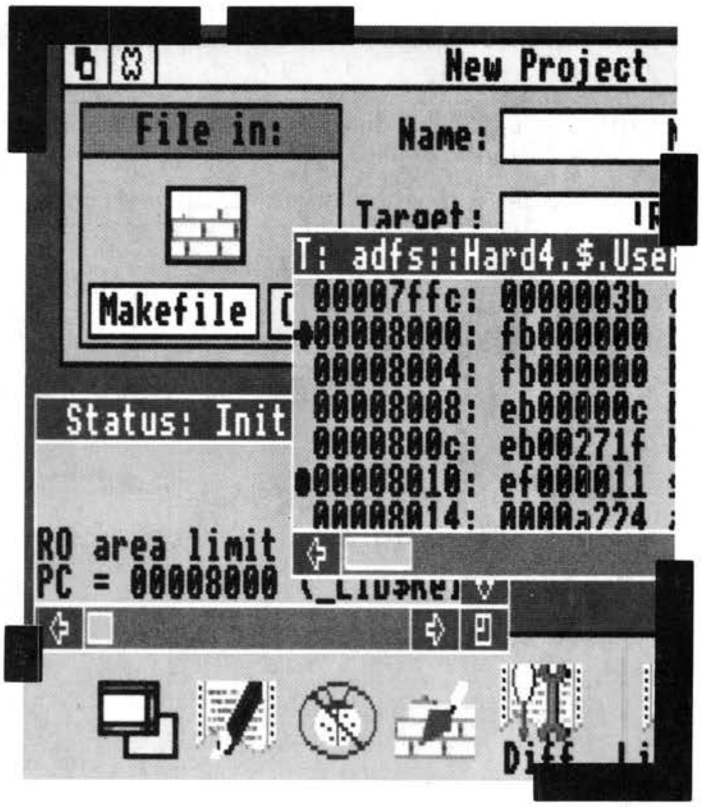


```
T: adfs::Hard4.$.User
 00007ffc: 0000003b
→00008000: fb000000
 00008004: fb000000
 00008008: eb00000c
 0000800c: eb00271f
●00008010: ef000011
 00008014: 0000a224
```

```
RO area limit
PC = 00008000 (_Lib$Rel
```

Acorn

# ACORN DESKTOP DEVELOPMENT ENVIRONMENT



```
T: adfs::Hard4.$.User
 00007ffc: 0000003b
●00008000: fb000000
 00008004: fb000000
 00008008: eb00000c
 0000800c: eb00271f
●00008010: ef000011
 00008014: 0000a224
```

```
RO area limit
PC = 00008000 (_Lib$Re
```

Acorn

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.

Acorn supplies its products through an international dealer network. These outlets are trained in the use and support of Acorn products and are available to help resolve any queries you may have.

Within this publication, the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

Published by Acorn Computers Limited
ISBN 1 85250 094 8
Part number 0470,501
Product number SKD37
Issue 1, May1991

# Contents

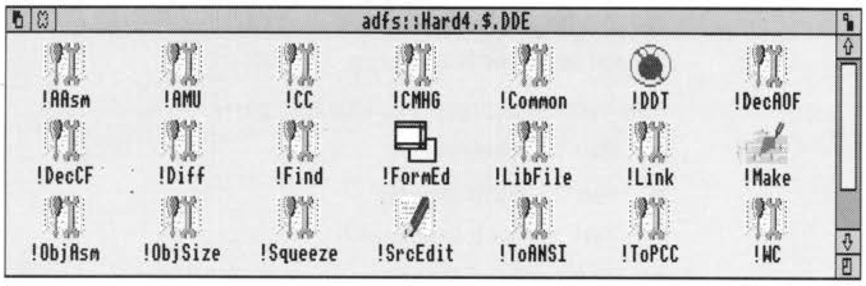
# 1 Introduction

The Desktop Development Environment is an extendable 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.

Since the development environment is designed to support more than one programming language, all the tools not specific to a language are intended to be included in more than one language product. The two products Acorn Desktop C and Acorn Desktop Assembler can therefore be seen as subsets of the total Desktop Development Environment developed by Acorn, each providing all the tools relevant to programming in one language. The purchaser of both products accumulates a complete environment for programming in both languages (potentially mixing them in one application). It is anticipated that third parties may wish to extend the environment with support for additional languages and market other language products including DDE tools.

This user guide describes the non-language specific tools, and may itself be distributed with several language products. It is included with both Acorn Desktop C and Acorn Desktop Assembler.

The Desktop Development Environment consists of a large number of tools. This is illustrated by the directory display showing their icons:



With the exception of the Desktop Debugging Tool (DDT), all these tools are multitasking 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 multitasking. The DDE includes tools to:

- edit program source and other text files

- search and examine text files mechanically
- examine some types of binary file
- compile and link programs
- assemble assembly language programs
- construct relocatable modules
- construct programs efficiently under the control of makefiles, these being set up from a simple desktop interface
- squeeze finished program images to occupy less disk space
- construct linkable libraries
- debug RISC OS desktop applications interactively
- construct template files for RISC OS desktop applications.

## About this guide

This user guide tells you how to use the Desktop Development Environment tools relevant for programming in more than one language, and is included in both the products Acorn Desktop C and Acorn Desktop Assembler. It is accompanied by additional user guide volumes covering the tools specific to each language – *Acorn* ANSI C *Release* 4 in Acorn Desktop C, and *Acorn Assembler Release* 2 in Acorn Desktop Assembler.

The majority of worked examples for the DDE products are described in the accompanying language specific volumes, and those mentioned in this guide are intended to make general points relevant to all the languages, although they cannot be tried out on all different products.

This guide is not intended as an introduction to programming techniques, languages or RISC OS. References are made to the RISC OS *Programmer's Reference manual* available from Acorn.

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 guide describes how to set up the best working environment for your purposes on your equipment using the standard DDE installation program, and covers general methods of operating the DDE.

The chapters are:

- *Installing the* DDE
- *Working in the* DDE

## Part 2 – Interactive tools

This has chapters covering each of the DDE tools which you use with constant interaction as 'foreground' tasks. Each has its own distinctive icon and file type. They are the debugger, template file editor, make and the source text editor.

The chapters are:

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

## Part 3 – Non-interactive tools

This covers the less interactive DDE tools which all have similar interfaces for setting options and running, some performing operations which can be controlled by Make. The first chapter in this part covers the general features common to all the non-interactive tools. The next eleven chapters are ordered alphabetically and each describes an individual tool. The last chapter describes how to extend the DDE by integrating your own tools with it, including how to create your own non-interactive tools.

The chapters are:

- *General features*
- AMU
- *Common*
- *Dec*AOF
- *Dec*CF
- *Diff*
- *Find*
- *LibFile*
- *Link*
- *ObjSize*
- *Squeeze*
- WC

- *Extending the* DDE

### Part 4 – Appendices

As part of the strategy of making the DDE extendable and open, this part of the guide contains specifications of file formats and other interfaces between tools as necessary information for someone adding to the DDE. They also provide useful references for others, for example those interworking assembler with a high level language such as C.

The appendices are:

- *Appendix* A - *Makefile syntax*
- *Appendix* B - Front*End protocols*
- *Appendix* C - DDEU*tils*
- *Appendix* D - SrcEdit *file formats*
- *Appendix* E - *Code file formats*
- *Appendix* F - ARM *procedure call standard*

## Conventions used

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

`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.

Hex integers are given in uppercase, and preceded by `0X`, eg `0XFE1`.
(Not preceded by `&`, as is the case with those of you more familiar with BBC Basic.)

The abbreviation DDE is used in later chapters to mean Desktop Development Environment.

# Part 1 - Getting started

# 2 Installing the DDE

Installing the DDE means setting up a disc directory structure suitable for use for future DDE sessions. You only need to perform this once to set up a suitable structure for a given hardware configuration. Booting the DDE means setting up your machine ready for starting work in the DDE, and is performed at the start of each session (eg after re-booting your machine).

To use the DDE you will need to:

- install the DDE
- boot the DDE.

You will then be able to work within the DDE.

This chapter only describes installation. The chapter entitled *Working in the* DDE explains how to boot and work within the DDE.

## Hardware requirement

The minimum specification of RISC OS system recommended for serious use of the DDE is a 2MB RAM machine with a hard disc drive.

You can use a 2MB or 4MB RAM machine with only floppy disc drives for constructing smaller programs although this will cause you some inconvenience due to disc swapping. This is because the DDE will not fit onto one floppy disc.

A 1MB machine is not recommended, as the number of DDE tools that you can load at the same time is limited.

If you are using a floppy disc machine that has access to an Econet network, you can avoid most of the disc swapping. See the section entitled *Installing the* DDE *on a network* on page 14. On a floppy disc plus Econet system, it is still recommended to store your own program on a work floppy disc (for speed reasons) but this restricts the size of program you can store.

## The Install application

Before installing the DDE on your machine with the Install application, it is wise to take a backup copy of each of the floppy discs supplied with the product.

Acorn Desktop C is supplied with four floppy discs; Acorn Desktop Assembler is supplied with three. These are not intended for use before running Install, but if you later wish to retrieve a file from them, it is useful to know their structure. They are organised with names: Installation Disc, Boot Disc, Work Disc and Reference Disc (not needed for Acorn Desktop Assembler). For diagrams showing their precise directory structure, see the reference cards included with each product.

The discs supplied with the DDE products are arranged so that Install can transform them reasonably easily into a working set of floppy discs during installation on a machine without a hard disc. The Installation discs contain the following:

- Install application
- reference material (which you will only use occasionally)
- programming examples.

The Boot discs contain the following:

- DDE relocatable modules in a !System directory
- desktop tools.

The Work discs contain the following:

- DDE command line tools (called when the desktop tools are run)
- programming examples
- a scrap file directory !DDETmp.

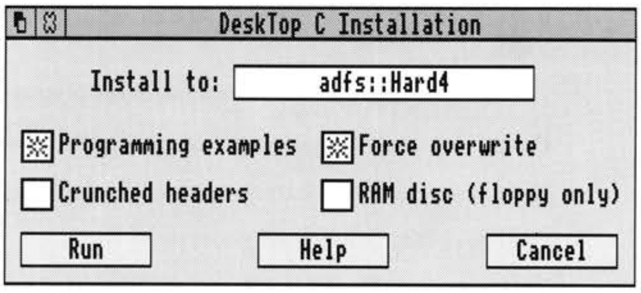
The Reference disc supplied with Acorn Desktop C contains linkable libraries and associated headers.

## Running the Install application

Take the following steps to run the Install application:

1. Take the preparation steps described in the later section covering the type of installation you wish to make.
2. Insert the disc labelled Installation Disc in your drive and click Select on the drive icon to open its root directory.
3. Double click on !Install in the resulting directory display.

The Install application then reads your current filing system and disc name, and displays an options dialogue box:


DeskTop C Installation

Install to: adfs::Hard4

[✓] Programming examples
[✓] Force overwrite
[ ] Crunched headers
[ ] RAM disc (floppy only)

This dialogue box allows you to specify the filing system and disc name to install to, and to set various options for the installation arrangement.

The **Install to:** slot is a writable icon containing the filing system name and disc name forming the destination for Install to copy files to. It is initialised with the current filing system and disc name; in the example above, the adfs filing system and a hard disc called `Hard4`.

When installing on machines with floppy disc drives, there are two ways you can construct work discs:

- containing programming examples to try
- leaving the maximum space for your own code.

The **Programming examples** option allows you to make this choice. This option is enabled by default – programming examples are included. When installing a hard disc or network, this option controls the inclusion of these examples.

Enabling the **Force overwrite** option causes Install to overwrite existing files whose names are duplicated by new ones. This is intended to ensure that files such as tool binaries in your library directory are consistently updated, and is enabled by default.

The **Crunched headers** option only exists in the C Install program, not the Assembler Install. Two versions of the C #include headers for the RISC OS specific library RISC_OSLib exist - the full commented version and the compact crunched version for minimum disc usage and processing time. The **Crunched headers** option specifies which version is installed. The full versions are on the Acorn Desktop C Installation Disc, the crunched versions on the Reference Disc.

When working on a floppy disc only machine, your discs may be set up to load a library of binaries into RAMFS at boot time. This leaves more space on your work floppy disc for your own programs, but less space in RAM for operating DDE tools. Enabling **RAM disc** results in Install setting up your working set of discs to use RAMFS.

The options dialogue box contains three action buttons:

**Run** starts the installation process with the options as set.

**Help** displays text information in a scrollable read-only text window.

**Cancel** cancels the Install process.

If you are installing more than one DDE product, for example both Acorn Desktop C and Acorn Desktop Assembler, merely run the Install process of each product in turn, making sure the options dialogue box settings and so on are the same both times.

After running Install, reset your machine, to make sure there are no problems changing to new versions of relocatable modules supplied with the DDE.

## Installing the DDE on a hard disc machine

To prepare for installation to your hard disc, first check and if necessary adjust the !System application on your hard disc for correct interaction with Desktop C or Desktop Assembler.

It is standard to have a !System application on an Archimedes hard disc. If you have the standard directory `Apps1` on your disc your !System is likely to be located there. !System sets environment variables from its !Boot file executed when a directory display is opened for the directory containing !System. It therefore may be convenient to move your !System application to the root (`$`) directory of your hard disc so that its !Boot file is executed every time you open the root directory of your disc.

For the Desktop Development Environment to work properly on your hard disc machine, your !System needs to set the environment variables System$Path, Wimp$Scrap and Wimp$ScrapDir. Set up and use your !System before running Install, as Install uses System$Path itself. To set up your !System, open a directory display on the !System directory by double clicking Select with the shift key depressed. Load the obey files !Boot and !Run into a text editor such as Edit (by dragging them to the Edit icon bar icon) and inspect and alter the text if necessary. To set the environment variables, the !Boot file should look like:

```
| !Boot file for !System
IconSprites <Obey$Dir>.!Sprites
if "<System$Path>" = "" then Set System$Path <Obey$Dir>.
if "<Wimp$Scrap>" = "" then Set Wimp$Scrap <Obey$Dir>.ScrapFile
if "<Wimp$ScrapDir>" = "" then Set Wimp$ScrapDir <Obey$Dir>
```

and the !Run file should look like:

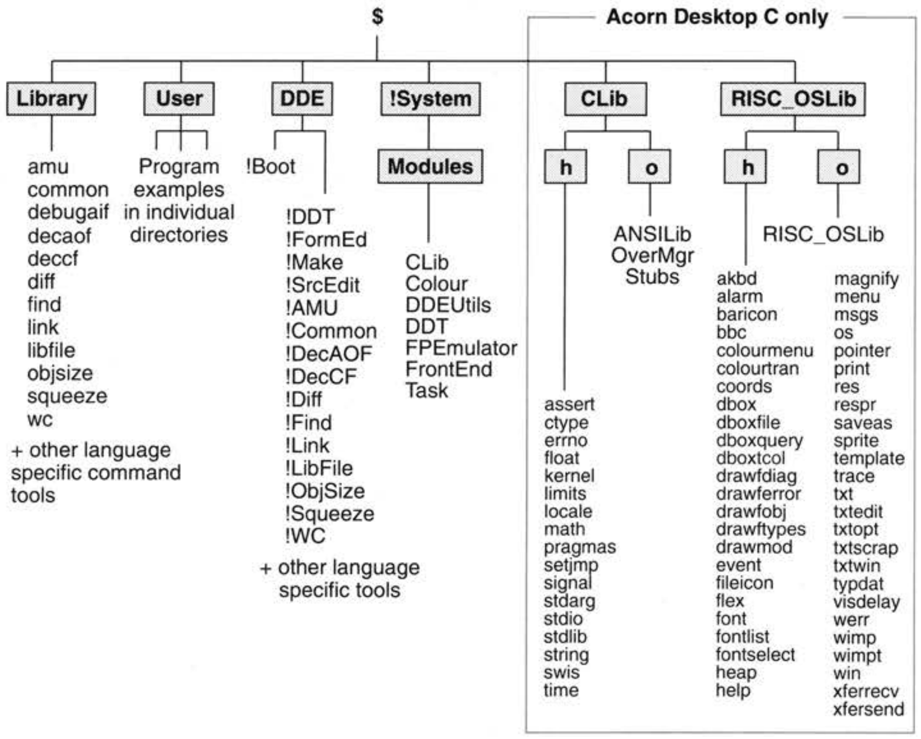
```
| !Run file for !System
IconSprites <Obey$Dir>.!Sprites
Set System$Path <Obey$Dir>.
Set Wimp$Scrap <Obey$Dir>.ScrapFile
Set Wimp$ScrapDir <Obey$Dir>
```

Note that System$Path is set with a terminating '`.`' character, unlike the other two variables above. Note also that the above files are not identical to those supplied on the Acorn Desktop C or Assembler distribution discs, which are intended for floppy disc usage.

Save these files to your hard disc, then double click on !System to use it (execute the !Run file and set the system variables).

If **Install to:** contains the filing system and disc name of your hard disc when you run Install, the entire DDE directory and file structure is set up for use on your hard disc.

The following directory structure is set up for you on your hard disc. It is created if not present, or updated if it is already there:



An application !Boot is left in the directory `$.DDE` to assist you in booting the DDE. If you want to execute this every time you reboot your machine, you can insert a line such as:

```
*$.DDE.!Boot.!Run
```

in your own machine !Boot file. A machine !Boot file is an obey file (created with a text editor such as Edit) placed in your hard disc root directory, executed at machine power-up or reset. It is analogous to the AutoExec.Bat file for DOS machines. To set your machine to execute a machine boot file, type the line:

```
*Opt4,3
```

at the RISC OS command line.

Note that Install places a new version of the shared C library relocatable module in <System$Path> Modules. If you have an older version of this module placed somewhere else and loaded by your own !Boot file (or by one of the applications loaded by your !Boot) delete the old module and alter any references to load the new one.

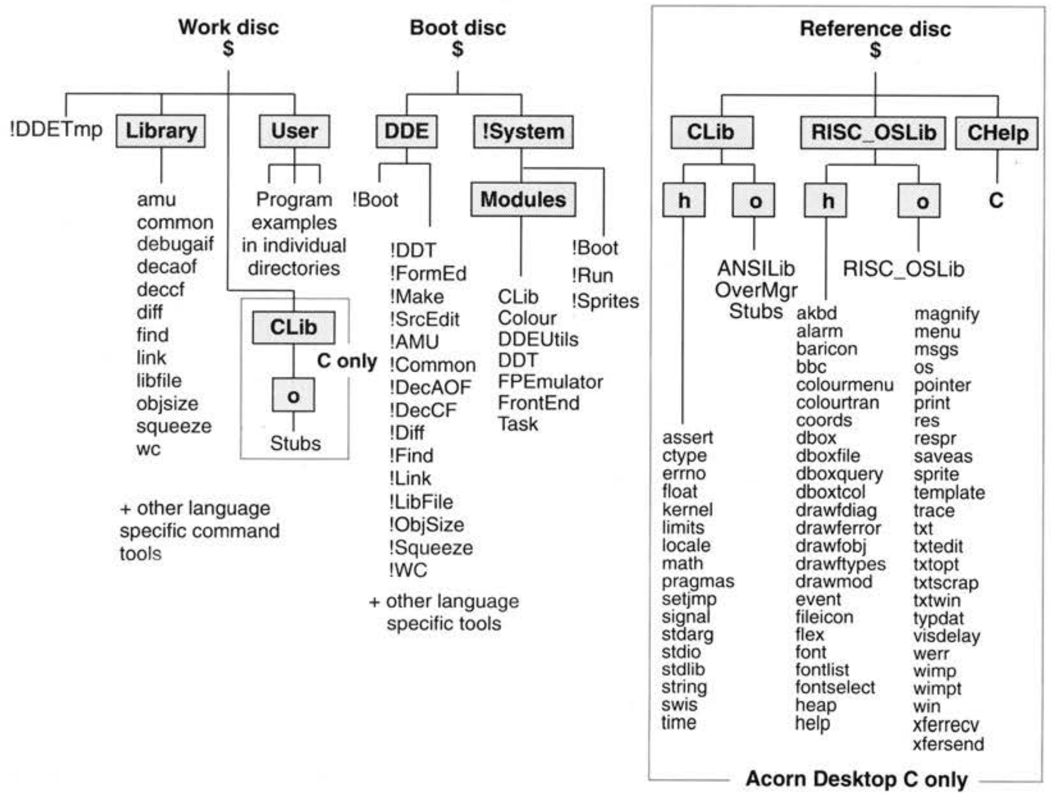
## Installing the DDE on a floppy disc machine

If **Install to:** contains a string such as `adfs::0` or `adfs::UserWorkC` when you run Install, a working set of floppy discs is constructed from the distribution discs.

You will need a number of blank, previously-formatted RISC OS 800K floppy discs. The recommended format to use is RISC OS E format. The number of blank discs needed is the same as the number of distributed discs when making a working set with **RAM disc** enabled, one less without **RAM disc**. Thus, for example, installing Acorn Desktop C with **RAM disc** on requires four blank discs, whereas with **RAM disc** off you only need three.

When you run Install, it prompts you to insert various distribution discs and fresh discs in turn, so that each fresh disc can be named and files copied to it.

The arrangement of files on a working set of floppy discs is similar to that on a hard disc, but split between the floppies. The arrangement of the three main discs without **RAM disc** enabled is as follows:



The reason an extra blank disc is required for installation with **RAM disc** is that an extra Boot disc is created. This contains the `Library` and `CLib` directories of the above Work disc. It loads the command line tools at boot time into RAMFS, allowing them to operate faster and saving space on the Work disc, leaving more disc room for your programs.

To use library binaries in RAMFS, you must make an addition to the Run$Path RISC OS environment variable at boot time. You can place a line in the !Boot file supplied on the RAM floppy disc to set this up, such as:

```
*set Run$Path ,adfs:%.,ram:%.
```

## Installing the DDE on a network

In preparation for installation to a network, you must first log onto the target network with system privilege, then set the current filing system to the network by typing the line:

```
*net
```

at the RISC OS command line. Return to the desktop and run Install.

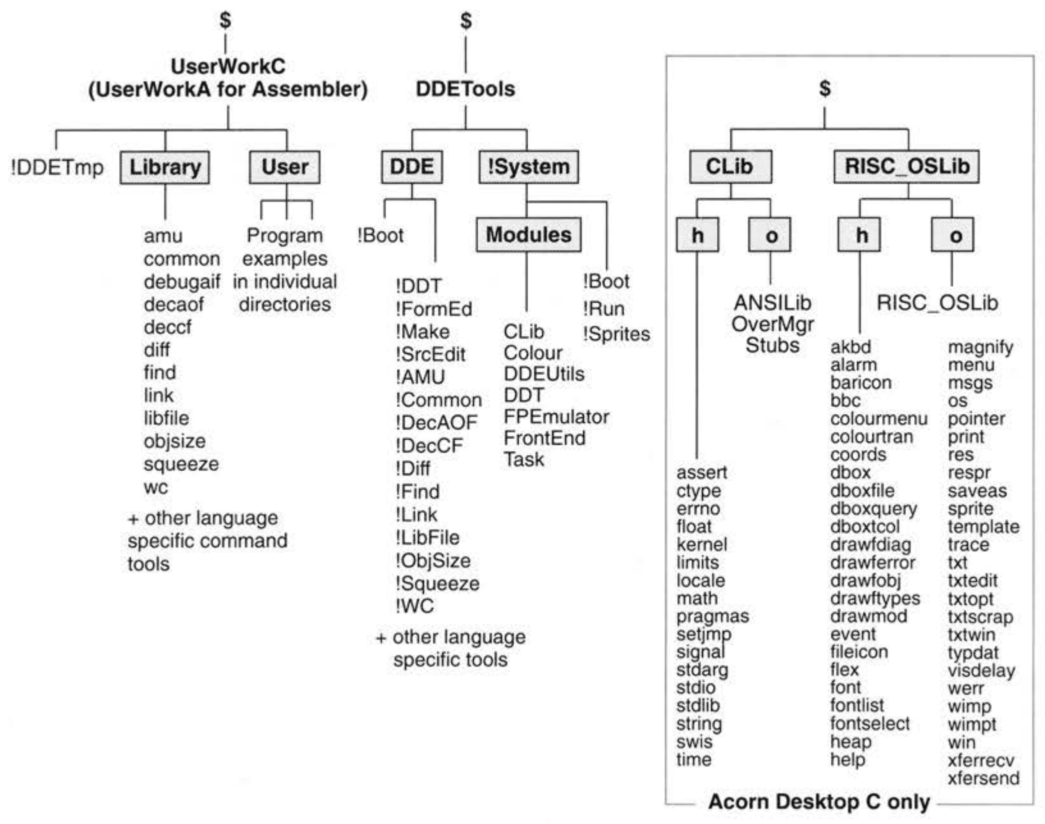
The **Install to:** field appears set to the network fileserver name, and when you run Install, a directory and file structure is set up on the network and one floppy disc from the distribution discs.

You will need one blank floppy disc per user, so format some to RISC OS E format.

Note: If more than one user is to share your network installation, a site license must be purchased through your Acorn Authorised Dealer.

When you run Install, it prompts you to insert various distribution discs and the fresh disc in turn, so that files can be copied.

The arrangement of files on the network is similar to that on a hard disc, but split between different directories. The following is the installed arrangement:



Each network user then requires a copy of the $.DDETools directory within their own directory area, plus a copy of the work floppy disc. Each user requires a copy of the DDETools directory because tools such as Make and SrcEdit write options files into their application directories.

To make a work floppy disc, name a blank formatted disc UserWorkC (C) or UserWorkA (Assembler) and copy the contents of the directory of the same name on the net to the floppy disc.

## Environment variables and the DDE

Various DDE operations depend on the correct settings of environment variables. If you carefully follow the instructions in this user guide for installing and booting the DDE, they should be correctly set and you do not need the following information. These details are summarised here as an aid for tracking down any problems you may have.

Each DDE 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, for example, to store options. 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 entitled *SrcEdit* later in this guide.

### System$Path

| | |
|---|---|
| Set by: | The !Run and !Boot files of the !System application. |
| Purpose: | Used by Install to indicate the directory in which to place relocatable modules. Used by the DDE applications as the place to load relocatable modules from. |
| Problems: | If the !System application has not been run or shown in a directory display System$Path remains unset. Install then stops with an error when run. DDE tools fail to load as they cannot locate required modules. |

### Wimp$Scrap

| | |
|---|---|
| Set by: | The !Run and !Boot files of the !System application. |
| Purpose: | This specifies the filename of a temporary file for passing from one desktop application to another (eg when saving a file from SrcEdit directly to the icon bar icon of WC). |
| Problems: | If the !System application has not been run or shown in a directory display Wimp$Scrap remains unset. Direct saving of files from one DDE tool to another then fails with an error. |

### Wimp$ScrapDir

| | |
|---|---|
| Set by: | On a hard disc the !Run and !Boot files of the !System application. On a floppy disc or network, set by the !Run and !Boot files of the !DDETmp application on the work disc. |

| | |
|---|---|
| Purpose: | This specifies the directory name in which to place temporary files. DDE tools such as the C compiler and assemblers generate temporary output files which you then rename to the file you want when you drag an icon to a directory display. |
| Problems: | If Wimp$ScrapDir is not set, non-interactive DDE tools such as the C compiler and assemblers attempt to create temporary files in `$.tmp` when run. If this fails (for example because this directory doesn't exist) the tools fail to run, generating error messages about being unable to open files in `$.tmp`. |

## Run$Path

| | |
|---|---|
| Set by: | User constructed !boot obey file. |
| Purpose: | This specifies a list of directory names which the system searches to find and execute image files. When the DDE non-interactive tools are run, they execute command line tools from a library directory, which is in the ramfs filing system if the RAMFS floppy disc option is used. |
| Problems: | If incorrectly set, command line tools may not be found and non-interactive tools fail to run. |

## DDE$Path

| | |
|---|---|
| Set by: | The !Run and !Boot files of the DDE !Boot application (set up by Install). |
| Purpose: | This is set to the name of the directory containing the DDE 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 (Acorn Desktop C only)

| | |
|---|---|
| Set by: | The !Run and !Boot files of the DDE !Boot application (set up by C Install). |
| Purpose: | This specifies a list of directory names for the C compiler to search for libraries and their headers. |
| Problems: | The !Boot application is created rather than just copied by Install. If you perform your installation by copying rather than running Install, you will produce a !Boot application which does not set C$Path. If |

unset, the C compiler will not be able to find #include headers such as those of RISC_OSLib, either when used managed by Make or unmanaged.

## C$Libroot (Acorn Desktop C only)

| | |
|---|---|
| Set by: | The !Run and !Boot files of the DDE !Boot application (set up by C Install). |
| Purpose: | This specifies a list of directory names for the C compiler to search for #include headers. See the chapter entitled CC in the accompanying *Acorn* ANSI C *Release* 4 user guide for more details. |

# 3 Working in the DDE

This chapter describes how you boot the DDE to start each programming session, and provides an overview of the most productive way to work with the DDE to produce your programs. The chapter entitled *Installing the* DDE describes how to prepare your working environment.

To use the DDE you will:

- install the DDE
- boot the DDE.

You will then be able to work within the DDE.

## DDE tools

The DDE is formed from a number of RISC OS desktop programming tools. All DDE language products include the following tools:

- **DDT** – A new 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.
- **FormEd** – An improved version of the FormEd application for producing the Templates resource file of each RISC OS application.
- **Make** – A new 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 new features for constructing program sources and other text files.
- **AMU** – A compact alternative to Make for using, but not constructing, Makefiles.
- **Common** – A utility to find the most common words in a text file.
- **DecAOF** – A utility for examining AOF files output by language compilers or assemblers.

# 3 Working in the DDE

This chapter describes how you boot the DDE to start each programming session, and provides an overview of the most productive way to work with the DDE to produce your programs. The chapter entitled *Installing the* DDE describes how to prepare your working environment.

To use the DDE you will:

- install the DDE
- boot the DDE.

You will then be able to work within the DDE.

## DDE tools

The DDE is formed from a number of RISC OS desktop programming tools. All DDE language products include the following tools:

- **DDT** – A new 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.
- **FormEd** – An improved version of the FormEd application for producing the Templates resource file of each RISC OS application.
- **Make** – A new 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 new features for constructing program sources and other text files.
- **AMU** – A compact alternative to Make for using, but not constructing, Makefiles.
- **Common** – A utility to find the most common words in a text file.
- **DecAOF** – A utility for examining AOF files output by language compilers or assemblers.

- **DecCF** – A utility for examining chunk files.
- **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 efficient 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.
- **WC** – A text file word and character counting utility.

In addition, each product contains language specific tools, such as the C compiler CC forming part of Acorn Desktop C, and the assembler ObjAsm in Acorn Desktop Assembler. 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 DDE 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 DDE tool to the input of another appropriate DDE tool.

## Interactive and non-interactive tools

The DDE 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 four (DDT, FormEd, 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 entitled *General features* on page 117.

## Entering filenames

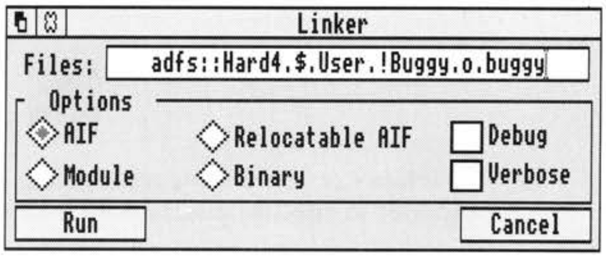
Many of the DDE tools require you to specify file or directory names. The interactive tools each have file types that they 'own', which you can double click on in directory displays to start activities. These are:

- **DebugAIF** – execution of one starts a DDT session. Files of this type are displayed in directory displays with the icon:
- **Template** – double clicking on one starts a FormEd edit. Template files 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 DDE 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::Hard4.$.User.!Buggy.o.Buggy
```

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 while the previous non-multitasking generation of Acorn language products could search for files relative to a suitably set current directory, a new concept of *work directory* is introduced for the DDE. 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.

## Booting the DDE

Booting the DDE at the start of a working session requires three steps:

- If working from floppy disc with a RAM disc, load the library directory from the RAMBoot floppy disc to RAMFS, using the !Boot obey file provided, ensuring that your setting of the RISC OS variable Run$Path is such as:

  ```
  ,adfs:%.,ram:%.
  ```

  so that the library binaries stored in RAMFS can be executed normally.
- Double clicking Select on a DDE !Boot application (in the DDE directory).
- Double clicking Select on the set of DDE tools you wish to use in the directory display of the `DDE` directory.

You are then ready to move to the User directory (on the UserWork disc) and construct your own programs or the example programs provided. If you are working on floppy discs double click on !DDETmp on your work disc to ensure that Wimp$ScrapDir is set correctly.

The DDE !Boot application is set up by the installation process as described in chapter entitled *Installing the* DDE on page 7. Its location depends on the hardware system you installed on. If you have a hard disc drive, the !Boot application is `$.DDE.!Boot`, and you may set your own boot file to execute its !Run file when your machine is started, removing the need for the first step above. If you have installed on a network, the !Boot application is in the subdirectory DDE of each user's copy of the DDETools directory. If you have a floppy disc only system, the !Boot application is in the DDE directory of the Boot disc produced at installation. See the directory diagrams in the chapter entitled *Installing the* DDE for the installed locations of files.

The number of DDE tools that you can usefully load at Boot time ready for use is determined by a number of factors: your machine RAM size; the space occupied by other applications loaded; the size of files you wish to process with the DDE tools; and the space on your icon bar.

## Working styles

The DDE 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 may find it convenient to place your own project in `User`, like the examples. You can place the source, header and object files in suitable subdirectories of the project directory. See the chapters on the language compilers or assemblers in the accompanying language specific user guide 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.

## Two ways of constructing the !Automata example

An illustration of managed and unmanaged development is the way that the program example !Automata (supplied with both Acorn Desktop C and Acorn Desktop Assembler) can be constructed from its sources in both working styles.

All the necessary resource files of !Automata are supplied ready made except for the main program file !RunImage. !RunImage is constructed from a C source file `c.Automata` and an assembler file `s.autoprocs`, linking with two C libraries `Stubs` and `RISC_OSLib`. In Acorn Desktop Assembler there is no C compiler supplied, so the C file is included ready compiled and linked to the libraries to form an object file `o.Automata`. In Acorn Desktop C no assembler is supplied so the assembler file is distributed ready assembled as `o.autoprocs`.

### Constructing !Automata in an unmanaged style

To construct !RunImage in an unmanaged style, compile/assemble and link with the following steps:

**1** At boot time load Link, and either CC (Acorn Desktop C) or ObjAsm (Acorn Desktop Assembler) by double clicking Select on their names in the DDE directory.

**2** Open a directory display on the !Automata application directory by double clicking Select with the shift key pressed, pointing to !Automata in the `User` directory.

**3a** If using Acorn Desktop C, compile the file `c.Automata` to produce the object file `o.Automata`. To do this, open the `c` subdirectory and drag the Automata text file to the CC icon bar icon. The CC setup dialogue box appears with the **Source** writable icon set to the name of the file you dragged. Enable the **Compile only** option by clicking on it, then click on **Run** to start the compiler. The CC Run and then Save dialogue boxes appear. Open the `o` subdirectory and drag the CC output file to it from the save box.

**3b** If using Acorn Desktop Assembler, assemble the file `s.autoprocs` to produce the object file `o.autoprocs`. To do this, open the `s` subdirectory and drag the `autoprocs` text file to the ObjAsm icon bar icon. The ObjAsm Setup dialogue box appears with the **Source** writable icon set to the name of the file you dragged. Click on **Run** to start the assembler. The ObjAsm Run and then Save dialogue boxes appear. Open the o subdirectory and drag the ObjAsm output file to it from the save box.

**4** Link the object files to form the !RunImage AIF file. To do this, open the `o` subdirectory and drag `o.Automata` to the Link icon bar icon. The Link setup dialogue box appears with the **Files** writable icon set to the name of the file you dragged. Drag the `o.autoprocs` file to the **Files** writable icon so that its name is added to the list to be linked. If using Acorn Desktop C you also need to drag the library files `$.RISC_OSLib.o.RISC_OSLib` and `$.CLib.o.Stubs` to the **Files** writable icon to include them in the link. Ensure that the **AIF** radio button is selected and click on **Run** to start the link. The Link Run then Save windows appear. Click on **OK** to save the AIF file produced with the default name, which is !RunImage in the application directory.

### Constructing !Automata in a managed style

To construct !RunImage in a managed style, build and use a Makefile with the following steps:

1. At boot time load Make by double clicking Select on its name in the DDE directory.
2. Open a directory display on the !Automata application directory by double clicking Select with the shift key pressed, pointing to !Automata in the `User` directory.
3. Click Select on the Make icon on the icon bar to show the New Project dialogue box.
4. Fill in the **Name** writable icon with a short name of your choice (10 or fewer alphanumeric characters), the **Target** writable icon with !RunImage (this is the file you wish to produce) and the **Tool** writable icon with Link, as this is the tool which finally outputs the target.
5. Drag the Makefile from the dialogue box to the !Automata directory to create the new Makefile (`project`). A Project dialogue box now appears for your new Makefile.

**6a** If using Acorn Desktop C, open the `c` directory and drag `c.Automata` to the **Insert** writable icon, followed by `o.autoprocs`, `$.CLib.o.Stubs` and `$.RISC_OSLib.o.RISC_OSLib`. Click on **OK** to the right of the **Insert** writable icon. This is the way you specify a set of input files to be processed by the Make job.

**6b** If using Acorn Desktop Assembler, open the `s` directory and drag `s.autoprocs` to the **Insert** writable icon, followed by `o.Automata`. Click on **OK** to the right of the **Insert** writable icon. This is the way you specify a set of input files to be processed by the Make job.

**7** Run a Make job to construct !RunImage from the input files specified by clicking on the **Make** button.

To repeat construction of !RunImage following your instructions stored in the makefile, open a project dialogue box for your project again, and simply repeat step 7 above, or alternatively double click on the Makefile in the directory display.

The file !RunImage is then constructed following your instructions stored in the makefile.

For more details about operating the individual tools, see the chapter about each tool.

For several worked examples illustrating general use of the DDE see the chapter entitled C *tools and the* DDE or the chapter entitled *Assemblers and the* DDE in the accompanying language specific user guide forming part of Acorn Desktop C or Acorn Desktop Assembler.

## Working with the DDE on small machines

As described in the chapter entitled *Installing the* DDE on page 7, the minimum Archimedes system recommended for serious use of the DDE is a 2MB RAM machine with a hard disc drive.

Working without a hard disc drive causes the DDE to be split between a set of floppy discs or between one floppy disc and the network. This means you waste time changing discs, and accessing files from a network can be relatively slow. The size of program you can easily develop is also restricted by the space for your files on an 800K work disc (with some extra effort a project could be split between two work discs). RISC OS provides help to desktop programs using files from more than one disc by including filing system and disc names in absolute filenames as used by the DDE tools. It is therefore possible, for example, to use Link to link object files from more than one disc. The disc containing each file must be inserted both when the filename is dragged from a directory display to the tool dialogue box, and later when the tool is run and wants to read the file. This results in several disc changes.

## New target support

The emphasis of earlier sections of this chapter has been on how to use the DDE tools productively on your host machine, but the DDE products also include improved support for your programs running on their target machines, giving you new options in designing your software.

Acorn Desktop C RISC_OSLib has been slightly extended, with new headers fontlist, help, fontselect, print and txtscrap.

The DDE non-interactive tools all have RISC OS desktop interfaces provided by the FrontEnd relocatable module. You can use this module to implement your own non-interactive tools, for programming or any other purpose, without having to build any handling of the RISC OS desktop into your programs. See the chapter entitled *Extending the* DDE on page 185 for more details.

## Compatibility with previous Acorn language products

There should be few problems in moving from processing your program sources with previous Acorn language products to processing them with DDE products.

Old makefiles, written to be used with the command line tool `amu` supplied with ANSI C Release 3 can still be used by Make or AMU, though Make cannot be used to alter them. You will have to edit an old makefile if it did not operate with the current directory set as the directory containing it.

## 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 DDE tools and experiment with their use, and there are further chapters that may provide useful general information.

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

A large number of the DDE tools are classified as 'non-interactive', and have similar interfaces. Examples are the Link, CC and ObjAsm tools used in the earlier Automata section. The chapter entitled *General features* later in this volume covers the interface features of this class of tool.

Since program examples are inherently specific to one programming language, most included with Acorn Desktop C and Assembler are described in their language specific manuals. Each of these manuals contains an early chapter demonstrating some of the DDE features with worked examples. The C chapter is called *C tools and the DDE*, the assembler equivalent is called *Assemblers and the DDE*. Other C program examples are described in the CC chapter.

Each language specific manual has a section called *Developing software for RISC OS* which contains chapters giving general advice on how to approach typical projects. Two such chapters are *How to write desktop applications in C* and *Writing relocatable modules in assembler*.

# Part 2 - Interactive tools

# 4 Desktop debugging tool

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. It can be used on any of the Archimedes range of computers running RISC OS 2.00 or later.

## 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 (ie 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:

- section entitled *About debuggers* introduces the concept of debuggers in general and describes the facilities provided by DDT.
- section entitled *Preparing your program* describes how to prepare your program for use with DDT.
- section entitled *Starting a debugging session* describes how to invoke the debugger on your program.

- section entitled *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 entitled *Execution control* describes how to control execution of a program running under the debugger.
- section entitled *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 entitled *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 ANSI 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 ANSI C may be found in *Acorn* ANSI C *Release* 4.

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 ANSI C compiler (other compilers may use different flags, though `-g` is common across a wide range of compilers. Refer to your compiler manual for details).

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 provided with Acorn's Desktop Development Environment, 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 ANSI C compiler to perform the link stage (ie 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.

```
#include <stdio.h>

int main(void)
{
    int world;

    for (world = 0; world < 100; world++)
        printf("Hello, World %d\n", world);
    return 0;
}
```

## Starting a debugging session

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

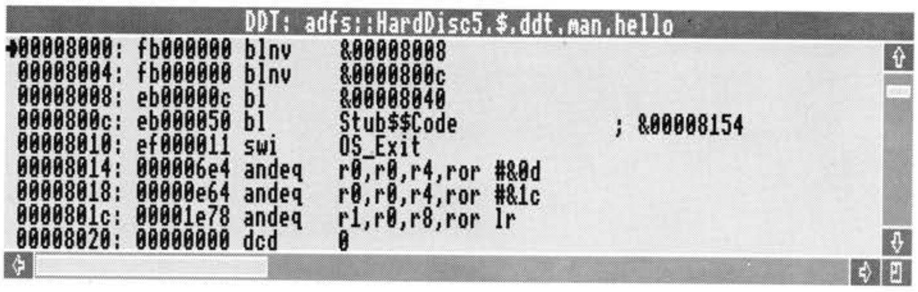
- Double click the `!DDT` application. This will place the debugger's icon on the icon bar. Then drag the program to be debugged to the debugger's icon. You can drag either an program image or an application directory. If you drag an application directory, the program image within that directory must 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]`*

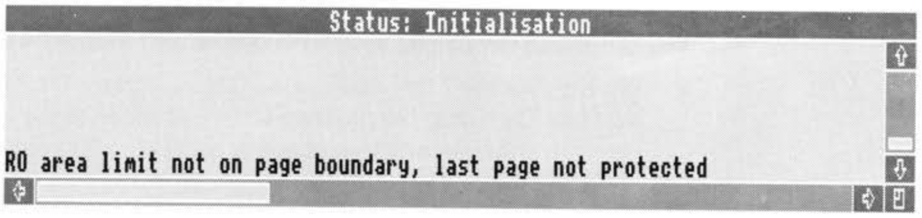
  where *`program`* is the name of the program to be debugged, and *`arguments`* are any command line arguments that program may take. You can enter this command from the supervisor prompt (outside the desktop), from the Shell CLI prompt (obtained by choosing the ***Commands** option on the task manager 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:



```
DDT: adfs::HardDisc5.$.ddt.man.hello
→00008000: fb000000 blnv    &00008008
 00008004: fb000000 blnv    &0000800c
 00008008: eb00000c bl      &00008040
 0000800c: eb000050 bl      Stub$$Code                    ; &00008154
 00008010: ef000011 swi     OS_Exit
 00008014: 000006e4 andeq   r0,r0,r4,ror #&0d
 00008018: 00000e64 andeq   r0,r0,r4,ror #&1c
 0000801c: 00001e78 andeq   r1,r0,r8,ror lr
 00008020: 00000000 dcd     0
```



```
Status: Initialisation

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

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 (ie 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 (ie 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 (ie 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`

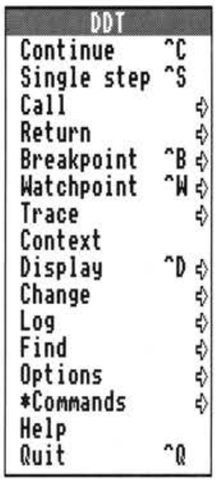
This message occurs when memory protection is enabled (as it is by default) and the last past of the code or read only area is not page aligned. This means that the last page of the read only area cannot be protected against accidental writes, since writing to data, or a read/write area which immediately 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 into 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:



**Continue**, **Single step**, **Call**, **Return**, **Breakpoint** and **Watchpoint** are explained in the section entitled *Execution control* on page 45.

**Trace**, **Context**, **Display** and **Change** are explained in the section entitled *Program examination and modification* on page 53.

**Log**, **Find**, **Options** and ***Commands** are explained in the section entitled *Options and other commands* on page 58.

## 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, ie 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 the Archimedes 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 (eg `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 of 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 (ie 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
```

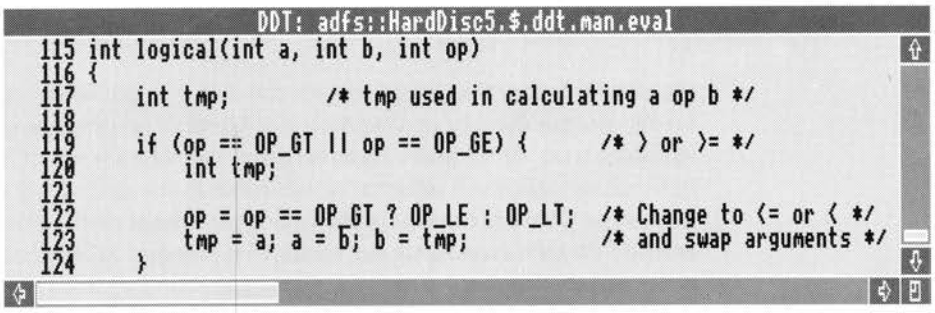
where *`procedure-specification`* is a procedure specification as described in the section above, *`line number`* is a line number in a source file and *`variable`* is the name of a variable.

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 (ie 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
119     if (op == OP_GT || op == OP_GE) {      /* > or >= */
120         int tmp;
121
122         op = op == OP_GT ? OP_LE : OP_LT;  /* Change to <= or < */
123         tmp = a; a = b; b = tmp;           /* and swap arguments */
124     }
```

The 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 (ie 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 entitled *Program examination and modification* on page 53) 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.

## 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, eg `a*(b+c)` |
| | `[ ]` | subscript, eg `isprime[n]`, `matrix[1][2]` |
| | | record selection, eg `rec.field`, `a.b.c` |
| | `->` | indirect selection, eg `rec->next` is `(*rec).next` |
| 2 | `!` | logical not, eg `!finished` |
| | `~` | bitwise not, eg `~mask` |
| | `-` | negation, eg `-a` |
| | `*` | indirection, eg `*ptr` |
| | `&` | address, eg `&var` |
| 3 | `*` | multiplication, eg `a*b` |
| | `/` | division, eg `c/d` |
| | `%` | remainder, eg `a%b` is `a-b*(a/b)` |
| 4 | `+` | addition, eg `a+1` |
| | `-` | subtraction, eg `b-d` |
| 5 | `>>` | right shift, eg `k>>2` |
| | `<<` | left shift, eg `2<<n` |
| 6 | `<` | less than, eg `a<b` |
| | `>` | greater than, eg `n>10` |
| | `<=` | less than or equal to, eg `c<=d` |
| | `>=` | greater than or equal to, eg `k>=5` |
| 7 | `==` | equal to, eg `n==0` |
| | `!=` | not equal to, eg `count!=limit` |
| 8 | `&` | bitwise and, eg `i & mask` |
| 9 | `^` | bitwise xor, eg `a ^ b` |
| 10 | `\|` | bitwise or, eg `m1 \| 0X100` |
| 11 | `&&` | logical and, eg `a==1 && b!=0` |
| 12 | `\|\|` | logical or, eg `a>lim \|\| finished` |

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

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

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

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

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

Constants may be integers (to the base specified in the Base option), hex integers (preceded by `0X` or `&`) character constants, strings or floating point numbers. The following show examples of each.

| | |
|---|---|
| `32768` | Integer in the currently selected base |
| `0X8000` | Hex integer |
| `3.2768e4` | Floating point number |
| `'A'` | Character constant |
| `"Hello, World\n"` | String |

String constants can contain escapes following the standard C syntax:

| | |
|---|---|
| \a | alert |
| \b | backspace |
| \f | form feed |
| \n | new line (cursor to start of next line) |
| \r | carriage return (cursor to start of current line) |
| \t | horizontal tab |
| \v | vertical tab |
| \' | single quote character |
| \" | double quote character |
| \? | question mark character |
| \\ | backslash character |

\x<hexadecimal number>
\<octal number>

## Integer base

The base in which DDT interprets constants entered by you, such as 32768, and in which it displays integer values, is determined by DDT defaults, the setting of the **Base** writable icon on the Options dialogue box (see *Options and other commands* on page 58) and the similar item on the Display box.

If the Display box **Base** writable icon is set, the specified base is used to display integer values. If the Options box **Base** writable icon is set, the specified base is used for input and output, unless overridden for Display output from the Display box. Note that this means that you can change a variable to a value in the Options base, then display it in another base specified on the Display box, for example changing a variable to 153 (base 10) then displaying it as 99 (base 16). If the Options box **Base** writable icon is not set, default bases are used.

### 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:**

| | |
|---|---|
| `main` | Address of function `main`. |
| `main + &14` | Five words into `main`. |
| `8000` | Start of image (assuming the image has not been relocated and the default base is hex.) |
| `Image$$RO$$Base` | 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

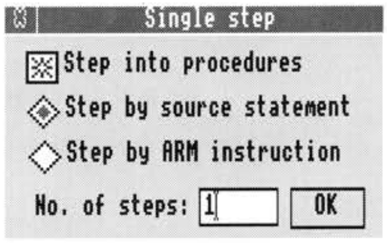
- you have disabled the **Stop at entry** option, or
- the `Can't set breakpoint on main` message appeared when you started the debugging session.

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
```

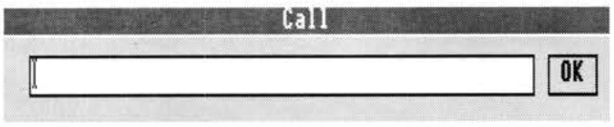
You complete the **Single step** dialogue by clicking on **OK** or pressing Return. The specified number of statements or instructions are then executed.

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 (ie 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.

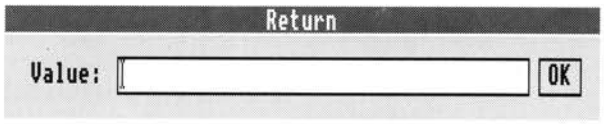
The arguments must be word-sized objects (eg integers or pointers) or floating-point values. Floating-point arguments occupy the next two adjacent ARM registers or stack words as described in the Arm Procedure Call Standard (ie floating-point arguments are not passed in floating-point registers).

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:

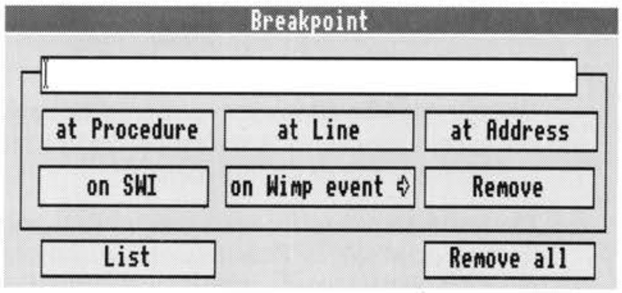


You can enter a value to be returned from the procedure in the value writable icon. This may be either an integer or floating-point value. If you do not specify a value a default value of 0 (or 0.0 for floating-point values) is used.

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

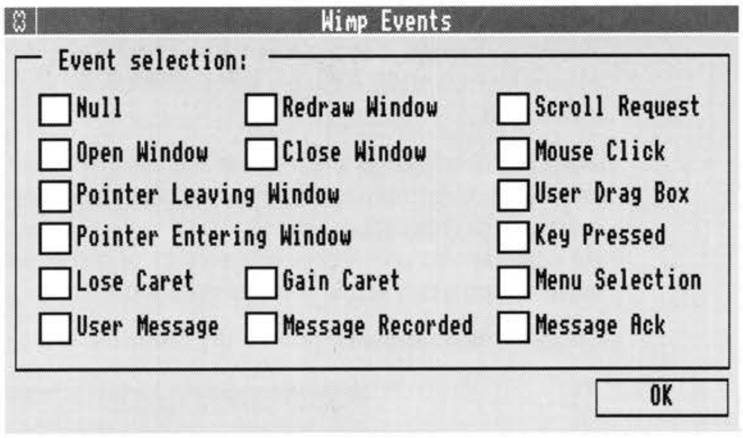
**Breakpoint** is used to add and remove breakpoints. Choosing **Breakpoint** produces the following dialogue box:



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 entitled *Specifying program objects* on page 38.

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.

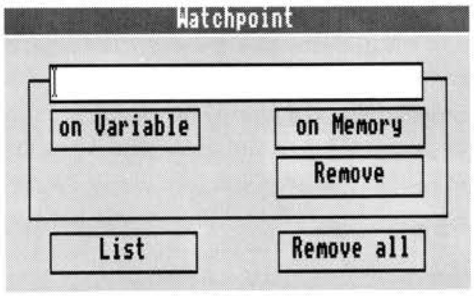
Breakpoints may also be set or cleared by clicking on a line in a source or disassembly display. Clicking on a line sets a breakpoint on the line. The breakpoint is shown by the breakpoint marker (a filled in circle) to the left of the line. Clicking on a line which already has a breakpoint removes the 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 entitled *Specifying program objects* on page 38.

**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 (ie 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 (ie 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.

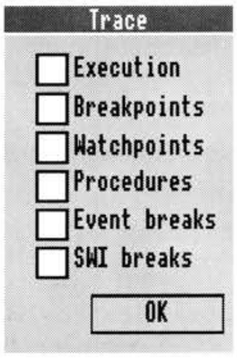
### SWI breaks

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:

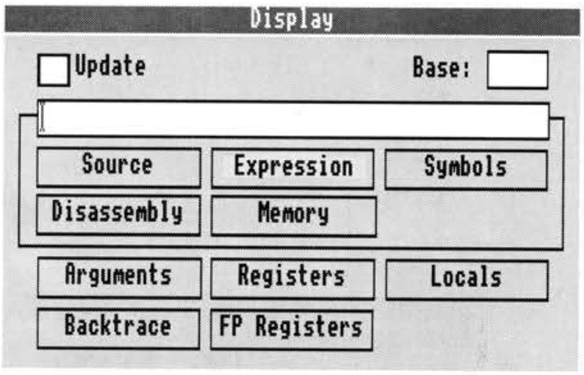


Select the set of actions you are interested in tracing and click on **OK**. A message confirming your selection will be displayed. You won't notice the effects of enabling procedure tracing until execution of the debuggee is resumed.

## 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.

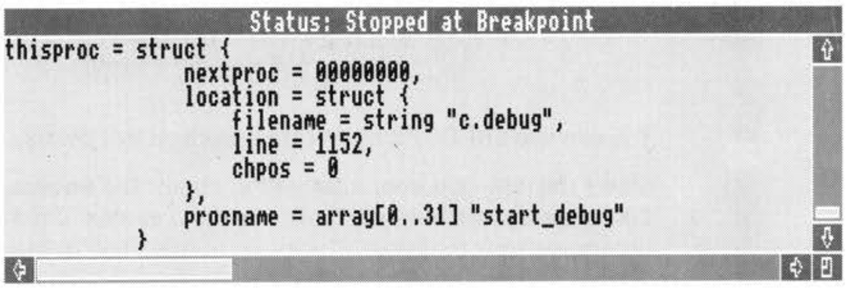
### Expression

The writable icon should contain an expression name. The syntax for entering expression names is described in the section entitled *Specifying program objects* on page 38. 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 indicated 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:

```
struct ProcedureLoc {
   struct ProcedureLoc *nextproc;
      struct SourcePos {
      char *filename;
      int line, chpos;
   } location;
   char procname[32];
}thisproc;
```

would be displayed as:



```
Status: Stopped at Breakpoint
thisproc = struct {
            nextproc = 00000000,
            location = struct {
               filename = string "c.debug",
               line = 1152,
               chpos = 0
            },
            procname = array[0..31] "start_debug"
         }
```

### 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 (ie 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.

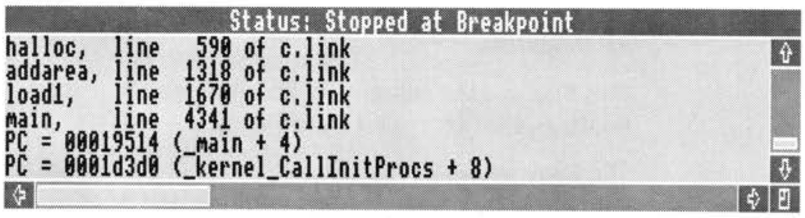
Procedures which have been compiled with debugging information are displayed in the following form:

```
procedure, line line of file
```

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

```
PC = address (procedure + offset)
```

A typical backtrace might look something like this:



```
halloc,  line   590 of c.link
addarea, line  1318 of c.link
loadl,   line  1670 of c.link
main,    line  4341 of c.link
PC = 00019514 (_main + 4)
PC = 0001d3d0 (_kernel_CallInitProcs + 8)
```

The last two entries in this backtrace are procedures in the C library initialisation code, the C library does not contain debugging information. Note that because the program used in the above example has been linked with debugging enabled, the procedure names still show in the C library. If the program had been linked without debugging information, even these would not be available, and the last entry, for example, would just read `PC = 00011358`.

### 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 (eg `_kernel_swi`) and `*$$*$$*` would list all the linker generated symbols (eg `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 entitled *Specifying program objects* on page 38.

**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**

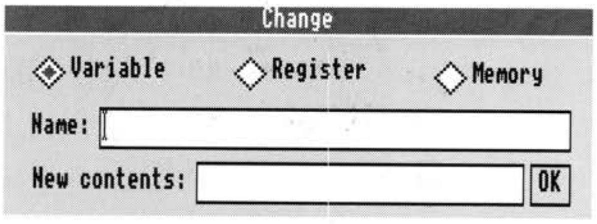
This displays the contents of floating-point registers 0 - 7 and the flags in the floating-point processor status word.

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 **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 entitled *Specifying program objects* on page 38. 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

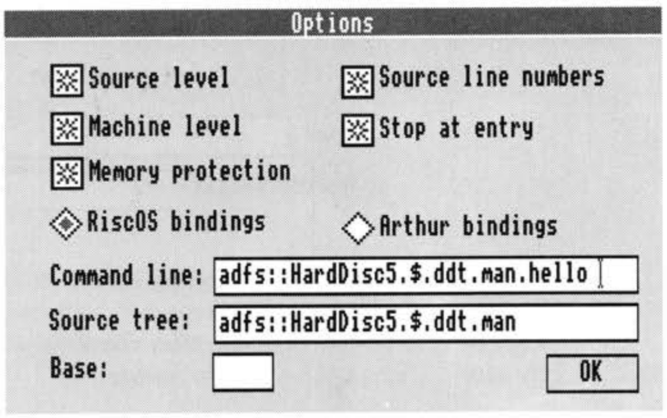
The **Name** writable icon should contain a register name. Valid register names are R0 - R15, SL, FP, IP, SP, LR, PC and F0 - F7. The **New Contents** writable icon should contain a low-level expression or floating-point constant, depending on the type of register being changed. Low-level expressions are described in the section entitled *Specifying program objects* on page 38.

### 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 entitled *Specifying program objects* on page 38.

## Options and other commands

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



[✓ Source level] [✓ Source line numbers]
[✓ Machine level] [✓ Stop at entry]
[✓ Memory protection]
[✓ RiscOS bindings] Arthur bindings
Command line: adfs::HardDisc5.$.ddt.man.hello
Source tree: adfs::HardDisc5.$.ddt.man
Base:

### 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 writing 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 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

The ARM Procedure Call Standard (APCS) has two variants:

- APCS_A, which was used in the Arthur operating systems and earlier operating systems for the ARM processors
- APCS_R, which is used in RISC OS.

Older compilers, such as Acorn's ISO Pascal and Fortran 77, and versions of the C compiler prior to 3.00, generate APCS_A code. APCS_A code can still be used under RISC OS, although machine language veneers may have to be written to interface with libraries such as RISC_OSLib. The variants differ in the bindings of the registers such as the stack pointer and frame pointer. The bindings are as follows:

- RISC OS: SL = R10, FP = R11, IP = R12, SP = R13
- Arthur: FP = R10, IP = R11, SP = R12, SL = R13

The debugger automatically determines which bindings are in force at any instant when displaying stack backtraces or examined stack variables. However, when displaying disassembly or register values, it cannot determine which bindings are being used. This pair of radio icons allow you to tell the debugger which bindings are in force.

It is not essential that you tell the debugger which bindings are being used, the option only determines the way the register names are printed in disassembly and register displays. For example the instruction

```
STMDB R13!, {R10, R11, R12, R13}
```

would appear as:

```
STMDB sp!, {sl, fp, ip, sp}
```

with RISC OS bindings, and as:

```
STMDB sl!, {fp, ip, sp, sl}
```

with Arthur bindings.

### 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

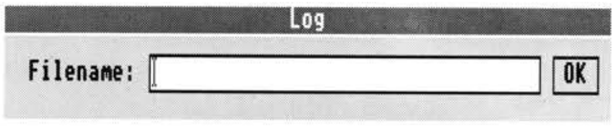
The **Base** writable icon gives the default numeric base when displaying or entering numbers.

### Source tree

Compilers such as Acorn's ANSI C may put relative filenames in the debugging information (eg `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).

## Log

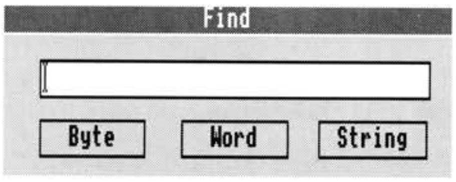
**Log** allows you to record any information output to the debugger Status window to a text file. Choosing **Log** produces the following dialogue box:



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.

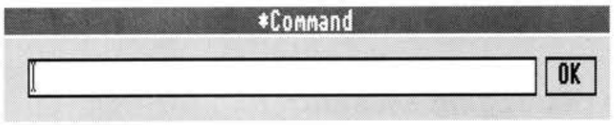
### String

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:

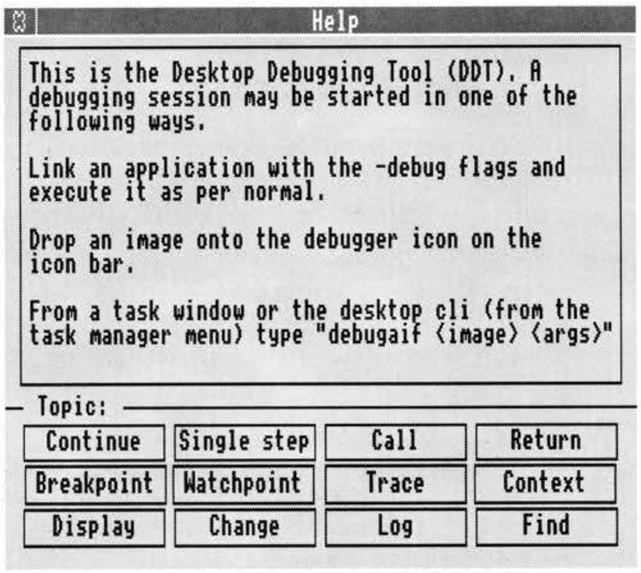


Enter the command you wish to execute in the dialogue box and press Return or click **OK**. If you are debugging a Wimp task (ie a task which has called Wimp_Initialise) you should precede the command with the WimpTask command, otherwise the output of any command executed may be displayed in graphics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



This is the Desktop Debugging Tool (DDT). A debugging session may be started in one of the following ways.

Link an application with the -debug flags and execute it as per normal.

Drop an image onto the debugger icon on the icon bar.

From a task window or the desktop cli (from the task manager menu) type "debugaif <image> <args>"

Choose the icon corresponding to the topic on which you want help. The help will be display 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 the `!Sort` directory, which is in the examples directory `User`.

```
 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
```

```
    h = 1;
    do
        h = h * 3 + 1;
    while (h <= n);
    do {
        h = h / 3;
        for (i = h + 1; i <= n; i++) {
            v = a[i];
            j = i;
            while (j > h && strcmp(a[j-h], v) > 0) {
                a[j] = a[j-h];
                j -= h;
            }
            a[j] = v;
        }
    } while (h > 1);
}

void sortfile(char *infile, char *outfile)
{
    _kernel_osfile_block finfo;
    int size;
    char *finbuff, *foutbuff;
    char *cp;
    int l, linestart;
    char **lbuff;
    int i;

    if (_kernel_osfile(READATTR, infile, &finfo) != FILEFOUND)
        fail("Error opening %s\n", infile);
    size = finfo.start;
    if (!(finbuff = malloc(size + 1)) || !(foutbuff = malloc(size + 1)))
        fail("Out of memory\n");
    finfo.load = (int) finbuff;
    finfo.exec = 0;
    if (_kernel_osfile(READFILE, infile, &finfo) < 0)
        fail("Error reading %s\n", infile);
    l = 0;
    cp = finbuff;
    linestart = 1;
    for (i = 0; i < size; i++) {
        if (linestart) {
            l++;
            linestart = 0;
        }
        if (!*cp || *cp == '\n') {
            *cp = 0;
            linestart = 1;
        }
        cp++;
    }
    *(finbuff + size) = 0;
    if (!(lbuff = malloc(l * sizeof(char *))))
        fail("Out of memory\n");
    cp = finbuff;
    for (i = 0; i < l; i++) {
        lbuff[i] = cp;
        cp += strlen(cp);
    }
    sortstrings(lbuff, l);
    cp = foutbuff;
    for (i = 0; i < l; i++) {
        strcpy(cp, lbuff[i]);
        cp += strlen(cp);
        *cp++ = '\n';
    }
    finfo.start = (int) foutbuff;
    finfo.end = (int) foutbuff + size;
    if (_kernel_osfile(WRITEFILE, outfile, &finfo) < 0)
        fail("Error writing %s\n", outfile);
    free(finbuff);
    free(foutbuff);
    free(lbuff);
}

int main(int argc, char *argv[])
{
    if (argc != 3)
        fail("Usage: Sort <infile> <outfile>");
    sortfile(argv[1], argv[2]);
    return 0;
}
```

## The debugging session

Follow the steps below to debug the example program.

1 Compile and link the program using !Make with the Makefile 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:

```
Illegal address (eg wildly outside array bounds)
Postmortem requested
   Arg2: 0x0000000c 12
   Arg1: 0x000176ac 95916
9dc8 in function sortstrings
   Arg2: 0x00015962 88418
   Arg1: 0x0001594b 88395
83bc in function sortfile
   Arg2: 0x00015914 88340
   Arg1: 0x00000003 3
84bc in function main
   Arg2: 0x00008488 33928 -> [0xe1a0c00d 0xe92dd833
                               0xe24cb004 0xe15d000a]
   Arg1: 0x000154c4 87236 c4c8 in function _main
```

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 illegal address; 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 with Make. To do this, first open the Make project dialogue box for Sort, click Menu on it and Select on the Link item of the **Tool options** submenu. Next, enable the Linker Debug option and click on **OK** to alter the Makefile. Use the Make Touch facility to touch all source members by clicking on **All** in the **Touch** option. Finally, click on the **Make** button to remake Sort.

**6** Start the debugger if you haven't started it already and drag the `!Sort` application directory on to the debugger's icon.

**7** 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:

**8** 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:

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

```
Break at main, line 107 of c.sort
```

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

**10** Choose **Continue** from the main menu again.

This time the debugger displays the following message:

```
Break at sortstrings, line 27 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:

**11** 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 = 000176ac
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, ie 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:

**12** 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] = string "Noel"
a[1] = string ""
```

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:

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

**14** Now follow the instructions in step 8 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:

**15** 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 84, with the initialisation of `cp`. So, set a breakpoint on line 84:

**16** Enter 84 in the Breakpoint dialogue box and click on **at Line**.

**17** Now choose **Continue** from the main menu.

The program will continue executing until it reaches line 84, 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`:

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

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

```
Watchpoint on cp changed at sortfile, line 85 of c.sort
New contents: string "Noel"
```

This is correct, so:

**20** Choose **Continue** again. The debugger will respond with:

```
Watchpoint on cp changed at sortfile, line 87 of c.sort
New contents: string ""
```

This is wrong: it should contain the string `Edward`. Look at the line which updated the value of `cp`:

```
87 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:

**21** Quit the debugger and the Sort tool frontEnd.

**22** Edit the file `c.sort` and change line 87 to read:

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

**23** Recompile `c.sort` using the Make utility.

Now try re-running the program:

**24** 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 87 didn't fix the problem. So, try running it under the debugger again:

**25** Quit the Sort tool frontend.

**26** Drag `infile` to the Sort tool icon.

**27** Set a breakpoint on function `sortstrings` and choose **Continue**.

The debugger will stop when it reaches `main`.

**28** 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 = 000176b0
n = 12
```

**29** 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] = string "Noel"
a[1] = string "Edward"
a[11] = string "Martin"
```

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

**30** 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:

**31** 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]`:

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

```
a[1] = string "Edward"
```

The shellsort algorithm should be comparing against the first string (ie `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:

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

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

**34** Compile the program, this time disabling the **Debug** option of Link using Make (see step 5), 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.

# 5 FormEd

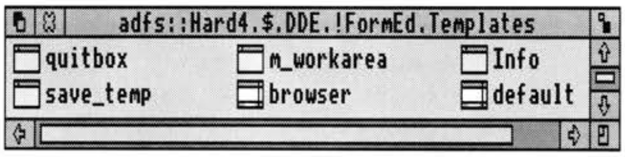
FormEd is the tool used to construct the Templates resource file of a RISC OS desktop application. The template editor FormEd is an application which allows you to define windows on the screen, and save the definitions in a Templates file ready for loading by your application. This is the approach used to construct Acorn's own applications. FormEd is a single document editor, and thus can only edit one template file at a time.

FormEd is supplied with DDE language products. To use it, you first need to understand the program interface of the window system, as described in the RISC OS *Programmer's Reference manual*. Refer, in particular, to the descriptions of the SWIs Wimp_CreateWindow and Wimp_CreateIcon, in the Window manager chapter. The account that follows also assumes an understanding of template files; these are described in the same chapter. For a guide to window styles refer to the *Acorn* RISC OS *Style Guide*.

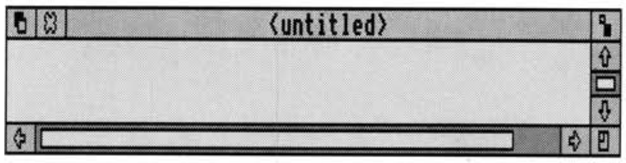
## Starting FormEd

Start FormEd in a similar way to other RISC OS applications, by double-clicking Select or Adjust on !FormEd in a directory display, or on a template file. Provided that FormEd has been 'seen' by the system the template file will be loaded along with FormEd. If a template file does not appear to load properly, give more memory to FormEd before it starts, using the Task Manager. The FormEd icon appears on the icon bar.

If you start FormEd by double clicking on a template file, the FormEd Browser appears, listing the windows defined in the template file. FormEd, as a RISC OS application, has a template file defining its own windows. The appearance of the Browser for `!FormEd.Templates` is:



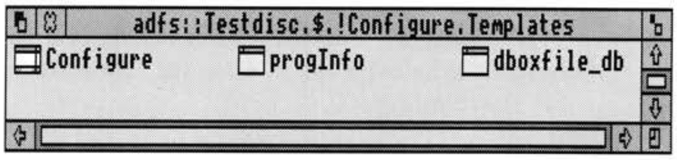
If you start FormEd without an existing template file, you can open a new template by clicking on the FormEd icon on the icon bar. An empty Browser will appear:



## Browser

The FormEd Browser is the central display of a template file edit. It lists all the windows defined by the template file being edited in a way similar to the way a directory display lists the files in a directory. From this list you select which individual windows you want to be displayed for editing. The Browser is a new feature of the version of FormEd distributed as part of the DDE, previous versions merely displaying all defined windows on the screen at once, often resulting in an overcrowded screen.

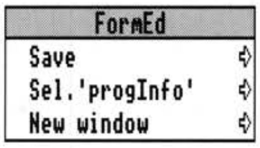
The appearance of a typical FormEd Browser window is shown below. It has a title bar displaying the title of the currently edited template file (or `<untitled>` if none) and a '*' after the title if the template file has been modified (this occurs even if a window has been moved). In this example, the template file has not been altered:



An empty Browser appears when you click on the FormEd icon on the icon bar or select the **New Templates** option on the menu. An empty Browser window has the title `<untitled>`.

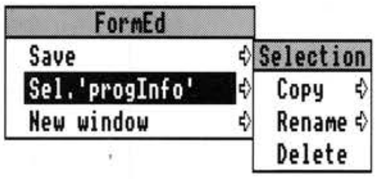
There are two types of windows: scrollable windows and dialogue boxes. A scrollable window is a window which has one or two scroll bars, whereas a dialogue box is a window with no scroll bars. In the work area of the Browser, each window listed is accompanied by one of two icons. These icons indicate whether the window is scrollable or a dialogue box.

Clicking Menu on the Browser brings up the Browser menu, from which you can save the template file being edited, or create or remove windows:



The **Save** option leads to a standard Save as dialogue box from which you control saving the template file being edited in the normal way.

When one or more window names are selected in the Browser in the same way that file names are selected in a directory display, the **Sel.** or **Selection** option leads to a submenu:



From this menu you can copy a single selected window to form a new window with another name, rename a single selected window or delete all selected windows. Since you can only copy or rename a single window, the **Copy** and **Rename** options are shaded out if two or more windows are selected. The **Selection** item is shaded out if no windows are selected.

The **New window** option allows you to add a simple scrollable window to the set defined by the template file being edited. To create a dialogue box you first create a scrollable window and then remove the scroll bars – this causes the icon in the Browser to change to a dialogue box icon.

Single clicking Select and Adjust on window names selects one or more windows, like selection of files in a directory display. Double clicking Select on a window name causes that window to be displayed for editing, or brings it to the front if it is already displayed.

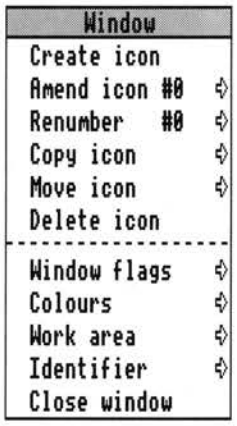
## Editing a window

When you load an application's template file into FormEd, all the windows used by that application are listed in the Browser window. Double clicking on a window name in the Browser displays that window for editing.

When FormEd displays a window defined by a template file, most of the window areas can be regarded as pictures of the real window you will see when running the application. For example, try loading the template file for the Configure application (make a copy before you do this!). The main Configure window will appear in the Browser as a scrollable window with the name of `Configure`. Double clicking on that window name makes the configure window appear, but you will not be able to use it to, for example, set the mouse speed.

While most parts of the border of a displayed window (title bar, scroll bars, back icon, etc) have their normal actions, the **Close** icon is used to close the display of that window. This can be reversed by double clicking on the window name in the Browser.

Clicking Menu on a displayed window produces a top-level menu:



This is the menu from which to change most window and icon properties, eg add or remove scroll bars, change icon wording. The upper half of this menu relates to icon properties, and the bottom half to window properties. Which of these features is selectable (not shaded out) depends on exactly where the pointer was when you clicked Menu: if it was on an icon, you will be able to amend or renumber the icon as well as the window itself. If the pointer was not on an icon, you will still be able to create a new icon.

Each of the window and icon properties in the menu and its submenus maps directly onto bitfields listed in the Wimp_CreateWindow and Wimp_CreateIcon descriptions in the RISC OS *Programmer's Reference manual*. However, you should also note the following points:

- Each window within a template file has a name or identifier which is unique to that template file. The identifier is used when the window definition is loaded by a call to SWI Wimp_LoadTemplate.

- The icons you add to a window are numbered in sequence, starting at 0. If two icons are placed so that they overlap, the window manager uses the numbering to determine which should obscure the other: higher numbers are displayed obscuring lower numbers. You may therefore need to change the number allocated to an icon; do this by swapping over two icon numbers. Click Menu over the icon you wish to renumber and select **Renumber**. Type in the number of the icon you want to swap with the currently selected icon, and the two will switch numbers.
- To add a new window to a template file, use the Browser menu.
- Because of the way the icon flag bitfield is organised, you cannot use anti-aliased text within a filled icon. Setting the **Anti-aliased** option in the Icon flags menu will make the background and foreground colour unselectable.
- The **V centred** (vertically centred) option applies only to sprites, not to text.

## Merging Templates files

To merge the window definitions of two Templates files into one file, load one Templates file into FormEd, then drag the other from a directory display to the displayed FormEd browser. The browser then shows that its file has been modified, and adds to the window list any new windows added by the merge.

Any window defined by the second Templates file with an identifier not used in the first file is added to the merged file. If both original Templates files define a window with the same identifier, the 'duplicate' window from the second Templates file is ignored.

The RISC OS desktop limits the number of windows that can be defined in a Templates file, so combinations of large Templates files which together would define too many windows cannot be merged.

## Displaying sprites in template windows

Windows defined by template files often have icons in which sprites are displayed. Such common items as radio buttons and option boxes are examples.

To display a sprite, you first specify its name in the writable submenu of the **Sprite** option in the **Icon flags** submenu (reached by following **Amend icon** on the top menu). If a sprite of the name entered is defined in the FormEd default sprite file (as is the case for standard icons such as radio buttons) the sprite then appears. If a sprite of the specified name is not in the FormEd file `default`, to display it you have to drag a sprite file containing it to the FormEd icon bar icon. You can move sprite icons within templates, and delete them. To edit a sprite, use the Paint application.

When you run a finished program, standard icons such as radio buttons are found in the wimp sprite area shared between all applications. When you display the Templates file of your application using FormEd, as described above such icons are instead found in the sprites file called `default` in the FormEd application directory, or a sprites file dragged to the FormEd icon. The default file is a copy of the wimp sprite pool forming part of RISC OS 2.00. To view the sprites stored in the default file, open the FormEd application directory by double clicking Select on !FormEd in the DDE directory while pressing the Shift key, then double click on the file `default` to load it into Paint. To dump the wimp sprite pool of your machine to a file on disc called `WSprites` (which will probably create a file identical to `default` if you have RISC OS 2.00) type in and run the following 2 line Basic program:

```
SYS "Wimp_BaseOfSprites" TO rom
SYS "OS_SpriteOp", &10C, rom, "WSprites"
```

## Editing ROM utility templates

It is possible to update the template files used by ROM utilities. These reside in the `deskfs:` filing system in the ROM. You access them via the environment variable `Wimp$Path`, so by updating this to search a directory of your own first where your updated template files reside, you can replace the window templates used by the utilities in the ROM.

## Example FormEd session

This example uses the template file for the Palette utility, which demonstrates some of the points described above.

1 Make a copy of the template file from the ROM by typing the following at the Command line prompt:

```
*adfs
*dir
*cdir templates
*copy deskfs:templates.palette templates.palette
```

2 Add the following to the !Boot file for your machine:

```
*set Wimp$Path adfs::4.$.,deskfs:
```

This assumes that you have a hard disc. If you don't, amend the line above as appropriate, depending on the location of your templates file.

3 Now return to the desktop and double-click on your copy of the templates file.

The FormEd Browser will appear, showing that two dialogue boxes are defined: the palette's main tool window and the Save box.

4 Double click Select in the Browser on each window name in turn to bring them up for editing.

The main tool window appears covered in cross-hatching: this indicates that the application (in this case, the palette utility code) is involved in redrawing the window.

You can move the windows around the screen by dragging on its title bar in the normal way. Move the main window to another position, noticing the star (*) appearing in the Browser title to show that you have modified the template file.

5 Save the modified file using the save box on the menu that appears when you press Menu over the FormEd Browser.

6 Now reset the machine.

You will find that the palette utility appears in the new position – where you dragged its window in the template file.

7 Double-click on the template file again, then the main window name in the Browser, to re-enter FormEd and display the main window.

8 Press Menu over the palette template window.

The menu that appears is divided into two parts. The upper half affects whatever icon you were pointing at when you pressed Menu; the lower half affects the window as a whole.

By entering the **Window flags**, **Colours**, and **Work area** submenus, you can see which bits within the window description are set and which are clear: compare this with the Wimp_CreateWindow section in the RISC OS *Programmer's Reference manual*. By clicking or typing on entries within these submenus you can affect such things as the title text and the colours of the window.

Some changes you might make will prevent the code from working properly, as they actually change the behaviour of the window in the program that operates it. Others, such as colour changes, are reasonable ways of setting your own choices for how the palette utility should appear.

9 Point at the black colour selection button and press Menu.

Each of the sixteen colour selection buttons is an icon. You can see that it is icon number 16 in this window.

By working through the **Amend icon #16** submenu, you can inspect and change every aspect of this icon in exactly the same way as with the whole window.

To move or resize an icon, take the following steps:

1 Ensure that its button type (within the **Amend** submenu) is set to Click/drag, so that it responds to dragging events.

2 Drag the icon with Select to move it.

3 Drag the icon with Adjust to change its size.

You can move the icon a pixel at a time or to specific coordinates using the **Move icon** submenu. Using other top-level submenus, you can make a copy of an icon, or renumber it.

# 6 Make

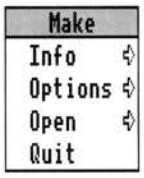
The Make application aids the programmer in the construction and maintenance of multiple-file programs, which can be combined to form any number of final targets (for example, libraries, modules, and application programs). The set of final targets and the files from which they are constructed are known 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 (eg 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 as shown below:



**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.

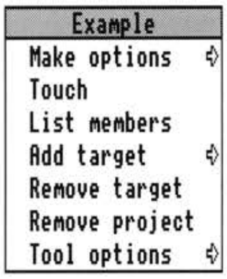
When a new project is created, you give it a unique name, and save its associated Makefile to disc. The persistent state of a project is held in a Makefile, which is automatically maintained by Make, with the option that it can be textually edited for customisation to a particular projects' requirements.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

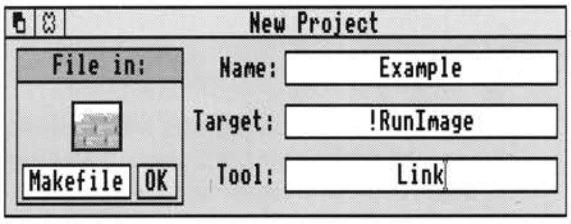
Under the desktop the concept of *current directory* has no sensible meaning, Make therefore uses the work directory in which the Makefile for a project has been saved as a prefix for all filenames used in the project. This prefix is denoted by the at symbol (@).

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:



Name: Example
Target: !RunImage
Tool: Link

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 (eg 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. |

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 entitled *Creating a final target for a project* on page 87 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.

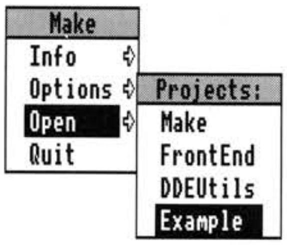
When this process has been completed, the newly created project becomes one of those maintained by Make, until it is explicitly removed (see the section entitled *Removing projects* on page 86 for how this is done). The dialogue box which is used to maintain this project then appears, with the project's name in its title bar. The project can then be maintained as described below.

## Maintaining projects

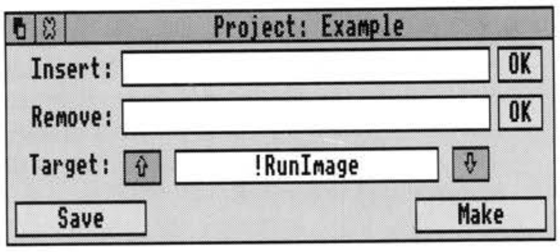
To maintain a project it is necessary to understand how to open and close projects, and how to specify the targets for a project.

### 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 editing the templates file using !FormEd).

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 entitled *Multiple targets* on page 84 for more details).

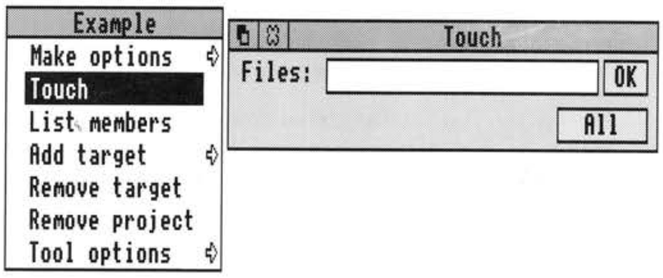
Make uses the following rule for dealing with files dragged to **Insert**: if the filename has, as its last but one component, a string (usually just one character) which corresponds to one of those registered by a translation tool, then it is assumed to be a program source file and a rule is constructed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

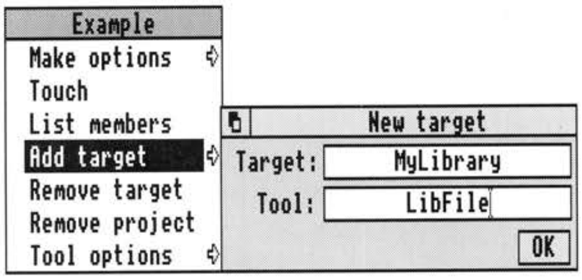
You can force a member of the project to be time-stamped using the **Touch** option in a project's dialogue box menu:



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.



Target: MyLibrary
Tool: LibFile

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 (eg `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.

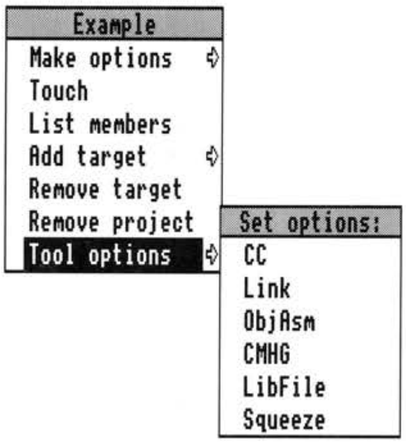
This can be used to create a 'squeezed' image by doing the following:

- 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, Aasm, 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:

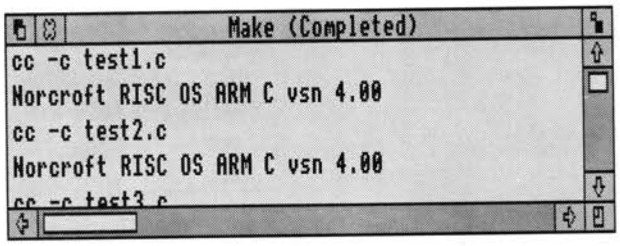
- If you click on **Make** in a project's dialogue box, Make will make the target which is currently showing in the Target icon. An alternative target can be selected by clicking the up and down arrow icons to move through the list of possible final targets.
- If you double click on a filer icon of type 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.

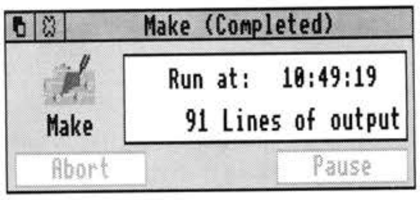
The output from this process appears by default in a scrollable, saveable text window (or in a summary dialogue box if this option is selected in the **Display** submenu):



```
cc -c test1.c
Norcroft RISC OS ARM C vsn 4.00
cc -c test2.c
Norcroft RISC OS ARM C vsn 4.00
cc -c test3.c
```

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:



Run at: 10:49:19
91 Lines of output

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 (ie those that are displayed by the output window). Clicking Adjust on the close icon of the summary box returns to the output window.

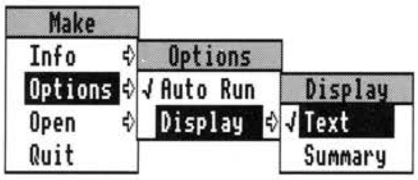
Both the above output displays follow the standard pattern of all the non-interactive DDE tools. The common features of the non-interactive DDE tools are covered in more detail in the chapter entitled *General features*. Both output displays, and the menus brought up by clicking Menu on them, offer the 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 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 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 later in the section entitled *Makefile format*.

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* A - *Makefile syntax*.

Below, we describe each of these sections, beginning with their respective 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. The tool will have done this by writing lines (described in the section entitled *Programmer interface* on page 91) into `<Make$Dir>.choices.tools.`

Each macro in the Makefile will be of the type:

```
toolflags =...
```

eg

```
ccflags = -c
```

| | |
|---|---|
| `# Final targets` | This section contains the rules for making the final targets of the project (eg `!RunImage:link $(linkflags)`). This information is obtained when the project was created (from the **Name** and **Tool** icons in the New Project dialogue box). |
| `# User-editable dependencies` | 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 Winchester, before doing a compilation. |
| `# Static dependencies` | This section contains rules for making an object file from its 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 (eg `cc -depend`). |

## An Example

In order to demonstrate using Make, you can manage the desktop example program `Automata` (which is formed from more than one source module). This example can be found in the `User` directory in the subdirectory `Automata`.

On the release discs there is initially no project for `Automata`, so that you can use this as an example of creating a project from scratch. If you follow the instructions below, you should be able to create and manipulate a project for `Automata`.

1 Double-click Select on the !Make icon from a directory display to install the Make application on the icon bar. By clicking Select on the Make icon (the one with the brick wall and trowel) you will get the dialogue box used to create a new project.

2 In the New Project dialogue box you will need to fill in a name for the project (for example Automata) the name of the final target (which you should type as `!RunImage` since this is an application) and the name of the tool used to create the final target (in this case you should enter `link`).

3 When you have filled in the New Project dialogue box you can then save the project to a directory display by dragging out the Makefile icon. It is best if you drag this icon to the directory viewer for `User.!Automata`. This now becomes the Automata project, and you will see the dialogue box for the project pop up to replace the New Project box. In its title bar this dialogue box will have the name of the project. Also, the project name will appear in the **Open** list from Make's main menu.

4 You will now need to add the members of the project. Do this by dragging the source and object files from the `User.!Automata` directory onto the **Insert** icon in the project's dialogue box, and then clicking **OK**. For Acorn Desktop C you will also need to insert the stubs object file (or ANSILib) and RISC_OSLib into the project. In order to see the members which you have added, you can click Menu on the project's dialogue box, and select **List members**. Note that the exact mixture of source and object files depends on which DDE product you are using (eg Acorn Desktop C or Acorn Desktop Assembler).

5 Set the options for the tools used to construct the example program. For instance, if you have the Acorn Desktop C product, then you must add the RISC_OS Lib headers to the C compiler's include path, using the **Include** icon in the C compiler's dialogue box. Such options can be set for a particular tool by clicking Menu on the project's dialogue box, then selecting the tool's name in the **Tool options** menu entry.

6 You can make the final binary for !Automata by clicking on **Make** in the project's dialogue box, or by double-clicking the project's Makefile from a directory display.

## Programmer interface

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

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

- Write a description file for the tool for use by the FrontEnd module and register it with Make as described below in the section entitled *Registering command-line tools with Make.*
- Write a WIMP frontend for the tool which complies with the details given below in the section entitled *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`:

| | |
|---|---|
| *`toolname`* | Name of tool. |
| *`string`* | Extension. |
| *`flags`* | Default flags for use by Make. |
| *`rule`* | Rule for converting sources to objects. |
| *`pathname`* | Full pathname of file containing application description. |

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

For typical examples see the entries in `<Make$Dir>.choices.tools` after installing the DDE.

## 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) (&81401) |
| +20 | Make's internal handle |
| +24 … | nul-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) (&81400) |
| +20 | Application's handle |
| +24 to +36 | Application's name |
| +36 … | nul-terminated command-line options |

# 7 SrcEdit

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` (`&0fff`). 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.

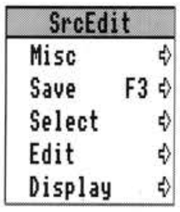
As you type, you will notice that SrcEdit fills the current line and then carries on to the next line, often breaking words in the middle. Ignore this for the moment, as there is a menu option (**Wordwrap**) that will take care of it, and this will be described later.

### 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 a character, position the caret immediately after it and press either Backspace or Delete; hold the key down and the auto-repeat will come into effect, deleting more characters.

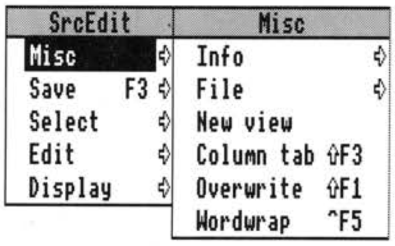
## SrcEdit menus

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



### The Misc menu

This menu offers six 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).

**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.

**Column tabs** switches on a different type of tab insertion; for more detail see the section entitled *Laying out tables: the Tab key* on page 106. When this option is on, it is ticked in the Misc menu and `ColTab` appears in the Title bar.

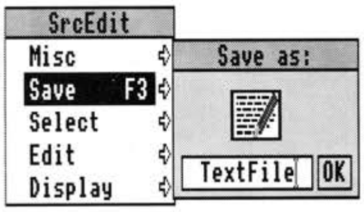
In SrcEdit the default state is to have **Column tabs** on.

**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, described below.



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. Move to **Save**, and a box appears, containing an icon, the current filename, and an **OK** box. 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. Place the pointer on the icon in the menu 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.

## Manipulating blocks of text – the Select menu

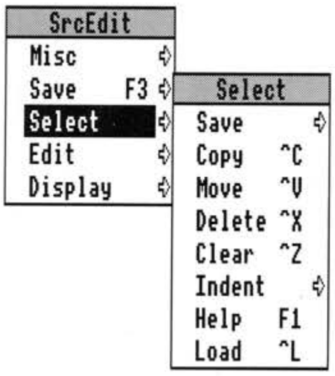
You can *select* blocks of text, then manipulate them.

The simplest way to select a block is to position the pointer where you want the block to start and, using the Select button, 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.

To select a single word, position the pointer anywhere within the word and double-click Select; select a single line by triple-clicking. Double-clicking with Adjust will extend the block to include the whole of the current word at the pointer, triple-clicking with Adjust extends it to the current line.

Once selected, text can be saved, copied, moved, deleted, deselected (cleared), indented, searched for programming help in an information library or treated as a filename to load by choosing **Load** 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 at the end of any text that is already in the destination window.

To make a copy of a selected block of text, first position the caret where you want the copy inserted, then call up the **Select** submenu and choose **Copy**; the original block remains selected. Keep clicking on **Copy** to make as many copies as you want.

If the caret is already at the position where you want the copied block to appear, press and hold Ctrl while making the selection in the usual way. Still holding Ctrl, press C, and the block will be copied to the caret position.

If you copy to a position inside a selected block, both the original and the new copy remain selected. If you then make multiple copies you will get double the number you indicate. This may happen accidentally if you position the caret immediately to the right of a selected block ending in a carriage return: because the carriage return does not appear on the screen it is not highlighted, but is still part of the selected block. To undo an action, choose **Undo** from the Edit menu (see later).

To move a selected block of text, place the caret where you want the text moved to, then click on **Move**.

If the caret is already where you want the block to end up, press and hold Ctrl while making the selection in the usual way. Still holding Ctrl, press V, and the block will be moved to the caret position.

To delete a selected block of text, click on **Delete**. The marked block then disappears.

**Undo** – in the Edit menu – allows you to reverse any changes or deletions made in the Select menu.

To clear (or deselect) a block of text you have previously marked, click on **Clear**. The marked block reverts to the normal display and the block is no longer selected.

**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 a block.

To indent a selected block of text, call up the **Indent** submenu, then type in a number.

- A positive number gives you an indent of the specified width.
- A negative number, eg –5, removes 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 NB, 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 negative number.

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 (eg 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:

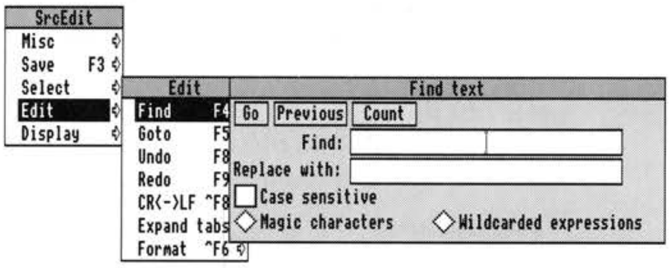
   *`DirectoryPath.LanguageExtension.Selection`*
3. Try to load file *`DirectoryPath.Selection`*.
4. If (3) fails try the comma-separated list of directories entered by the user from the **Search Path** entry in the **Options** submenu of SrcEdit's icon bar menu, with *`Selection`* appended as a leafname.
5. If (3) and (4) fail, try the comma-separated list of directories which are registered for the current language (see the section entitled *Application menu* on page 109 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

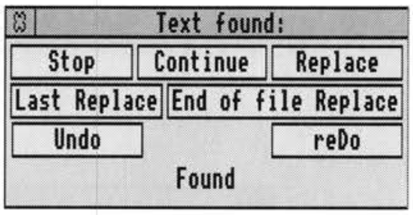
The first option in the Edit menu is **Find**. At its simplest, this allows you to locate any character(s) in your file. You can also use it to replace text with other text. To make sure that the search is complete, always position the caret at the start of the file before giving the **Find** command. 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.

To use **Find** without doing anything with the found strings, 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.



To start a search, 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:



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

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 Edit moves on to the next one.

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

If you click **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 bottom of the menu keeps you informed of the state of the search; if Edit cannot find the word you have specified it displays the message `Not Found`.

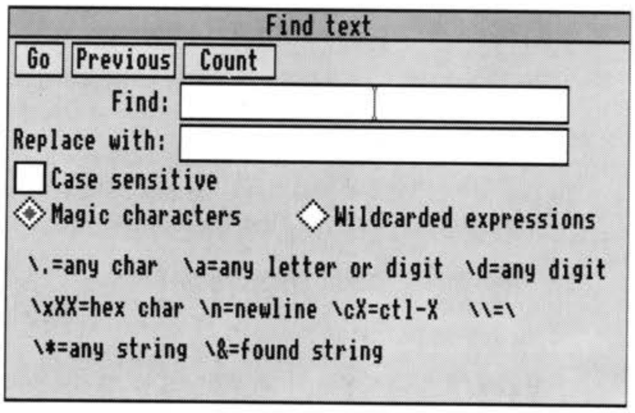
Besides using the Select button, you can control all these options by pressing keys; the particular keys are indicated by the capital letters in the dialogue box. Press S and the search stops, press C and it continues, D and it will redo, and so on. Pressing Escape or Return will stop the search and remove the **Text found** window.

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, accessed in the **Find text** dialogue box:

- You can carry out the last Find and Replace operation again, by clicking **Previous** or pressing F2.
- You can specify a string and ask Edit 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 pressing F3.

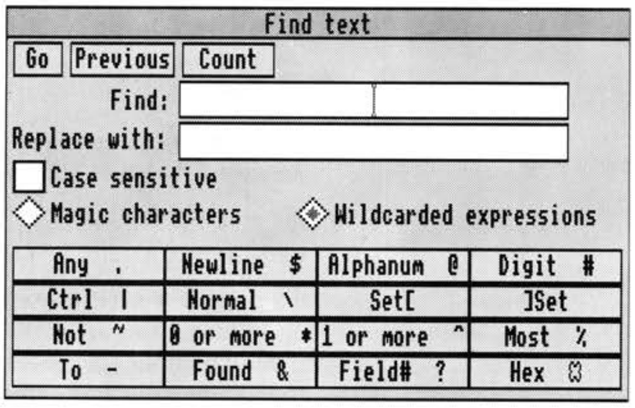
You can match case by selecting **Case sensitive** or pressing F4. By default, Find makes no distinction between upper and lower case characters – `Hello` will match to both `HELLO` and `hello`, or for that matter, `hElLo`. If you ask SrcEdit to match case, `Hello` will only match `Hello`. Case sensitivity remains selected until you deselect it by clicking again.



```
\.=any char  \a=any letter or digit  \d=any digit
\xXX=hex char \n=newline \cX=ctl-X  \\=\
\*=any string \&=found string
```

- In order to remain backwards compatible with versions of the RISC OS Edit, SrcEdit supports the **Magic Characters** facilities, which can be accessed by clicking on the **Magic Characters** in the **Find** dialogue box. You will notice that the dialogue box expands to show the meaning of characters which have a special use. They operate as follows:

| | |
|---|---|
| `\*` | 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`, `johnson`, and `jonathan`. |
| `\a` | matches any single alphabetic or digit character. So `t\ap` matches `tip`, `tap`, and `top`, but not `trap`. |
| `\d` | matches any digit. |
| `\.` | matches any character at all, including spaces and non-alphabetic characters. |
| `\n` | matches the newline character (remember that to the computer, this is a character just like any other). |
| `\cX` | matches Ctrl-X, where X is any character. |
| `\&` | is used in the **Replace with** box to represent the *found string*: the string matched in the search. This is 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`. |
| `\\` | 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`. |
| `\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. |



In SrcEdit there is also a facility for specifying wildcarded expressions in search strings, providing the power of an editor like Twin. In order to use this facility, click on **Wildcarded Expressions** in the **Find** dialogue box. A number of action icons show the features which are available. These are:

| **Action icon** | **Expression** | **Action** |
|---|---|---|
| Any | `.` | matches any single character. |
| Newline | `$` | matches the newline character (LineFeed). |
| Alphanum | `@` | matches any alphanumeric character `a-z`, `A-z`, `0-9` or `_`. |
| Digit | `#` | matches any digit `0-9`. |
| Ctrl | `\|` | `c` or `\|C` will match the character Ctrl-C. |
| Normal | `\` | `\c` will match the character `c` even if `c` is a special character. eg `\.` means the dot character not any single character. |
| Set\|/Set\| | `[]` | `[abc]` matches any one of the characters `a`, `b`, `c`. Note that a set is always case-sensitive. |
| Not | `~` | `~c` matches any other character than `c`, where `c` is any of the simple character patterns listed above. |
| 0 or more | `*` | `*c` matches 0 or more occurrences of `c`, where `c` is any of the simple character patterns listed above. |
| 1 or more | `^` | ^c matches 1 or more occurrences of c, where c is any of the simple character patterns listed above. |
| Most | `%` | `%c` matches the most contiguous characters matching c, where `c` is any of the simple character patterns (except Any) listed above. |
| To | `-` | `[c1-c2]` matches any character in the ASCII character set between `c1` and `c2` inclusive. |
| Found | `&` | refers to the whole of the found string. (only to be used in the **Replace with** string) |
| Field# | `?` | if a pattern was found which matched the search string, then `?n` refers to the part of the found string which matched the `n`'th ambiguous part of the search string, where `n` is a digit `0` to 9. Ambiguous parts are those which could not be exactly specified in the search string; eg in the search string `%#fred*$` there are two ambiguous parts, `%#` and `*$`, which |

| | | |
|---|---|---|
| | | are ?0 and ?1 respectively. Ambiguous parts are numbered from left to right. (only to be used in the **Replace with** string) |
| Hex | ¤ | ¤nn matches the character whose ASCII number is nn, where nn is a two-digit hex number. |

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

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

  ```
  Find: [a-z]
  ```

  and click on **Count**.
- To count how many words are in a text:

  ```
  Find: %@
  ```

  and Click on **Count**.
- To surround all words in a 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 non-printing ASCII characters (other than newline) from a file:

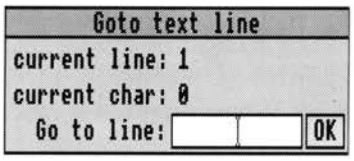
  ```
  Find: ~[ -\~$]
  Replace with:
  ```

  and click on **GO**, then on **End of File Replace** in the Found dialogue box (ie 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:  ~[ --$]
  ```

**Returning to the Edit menu**

To send the caret to a specific line of text, use the **Goto** option. Call up the **Goto** submenu and Edit 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 line feeds in your text to carriage returns and back again.

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 return where SrcEdit uses the line feed character.

**Expand Tabs** converts tab characters to the equivalent number of spaces, since some printers can interpret spaces more easily than the tab character.

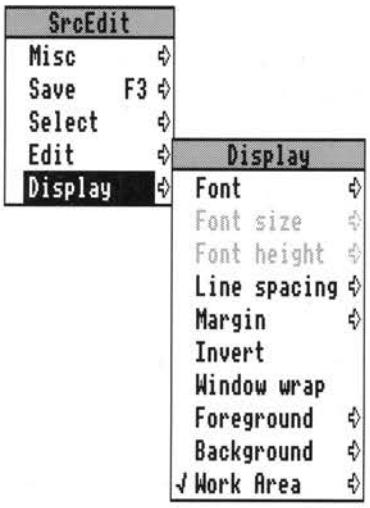
**Format text** allows you to reformat a paragraph of text – from the caret to the next blank line or line starting with a space – so that the lines fill the screen and break correctly at the ends of words. It is useful for tidying up text after editing. Position the caret at the beginning of the paragraph, 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 **Word wrap** 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. Normally SrcEdit restricts the maximum horizontal extent of a window to the size of the screen in the current mode, but 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 mode 12. The maximum size of window width which can be specified in this manner is 192 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, 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 files'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

To set out a table, 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. If **Column tabs** is turned off then pressing Tab on the next line will make the cursor jump to the position underneath the start of the next word in the line above.

If you want your table to have columns regularly spaced eight characters apart, click on **Column tabs** in the Misc menu. 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.

## Reading in text from another file

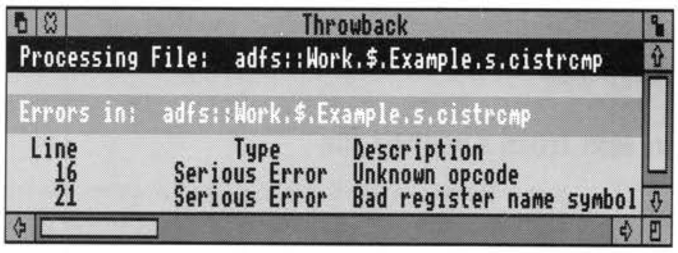
If you want to add all the text from another file into the file you are currently editing, position the caret at the point where the inserted text is to appear. Call up the directory display for the incoming file, and drag its icon into the text window. The entire contents of the source file are then 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 Ctrl-), the corresponding closing bracket will become the current selection; similarly by placing the caret to the left of a closing bracket (any of the set `)`, `]`, or `}`) and pressing Ctrl-(, the corresponding opening bracket will be selected. 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, and is useful in conjunction with the Ctrl-F7 feature (toggle caret and selection), thus moving the selection between matching brackets.

## 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.


Throwback
Processing File: adfs::Work.$.Example.s.cistrcmp

Errors in: adfs::Work.$.Example.s.cistrcmp

```
 Line         Type         Description
  16      Serious Error    Unknown opcode
  21      Serious Error    Bad register name symbol
```

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.

## 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:

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

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

| **Feature** | **Default** |
|---|---|
| Foreground Colour | black |
| Background Colour | white |
| Font Width | 10 |
| Font Height | 10 |
| Left Margin in pixels | 0 |
| Extra spacing between lines | 0 |
| Window wrap | off |
| Font name | System font |

| | |
|---|---|
| Window work area width | Screen width |
| Column tab | on |
| Overwrite | off |
| Wordwrap | off |
| Warn multiple edits | on |
| Current language | none |
| Search path | none |

## Application menu

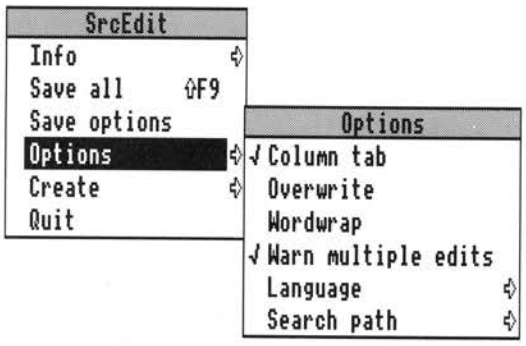
Pressing Menu on the SrcEdit icon on the icon bar produces a menu with the following options:

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

**Save All** 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 94. 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 (ie 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 in (2) in the **Load** submenu in the section entitled *SrcEdit menus* on page 94). Note that each such path should either be a path variable or be explicitly terminated by a dot.

**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.

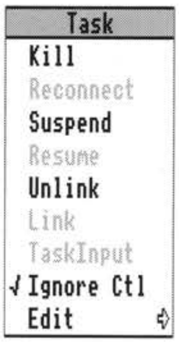
The major advantages in entering commands in a task window instead of at the Command Line prompt are that:

- 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 Edit 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 Edit window, even while a task is running. Any output from the task is appended to the end.

You can also supply input to a task window by selecting some text from another text file and choosing **TaskInput** from the task window menu. The selection may be in any Edit window.

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.

**Resum**e 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 Edit text window.

**Link** reverses the effect of **Unlink**.

**TaskInput** reads task input from the currently selected block.

**Ignore Ctl**, when selected, prevents any control characters generated by the program from being sent to the screen.

**Edit** leads to the normal Edit menu. Although this makes available most of Edit'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.

## 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, but need not be a full pathname. For example, `adfs::@` will open a display for the current directory.

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 without moving caret. |

| | |
|---|---|
| Ctrl-Shift-← | Scroll all documents up by one line. |
| Ctrl-Shift-→ | Scroll all documents down by one line. |
| Copy | Delete character to right of caret. |
| Shift-Copy | Delete word at current caret position. |
| Ctrl-Copy | Delete line at caret. |
| Home | Place caret at top of document. |
| Insert | Insert space to right of caret. |
| Page Up/Page Down | Scroll up or down one windowful. |
| Shift-Page Up/Page Down | Move caret up or down one line without scrolling. |
| Ctrl-Page Up/Page Down | Move caret and scroll up or down one line. |
| Shift-F3 | Toggle column tabs on or off. |
| Shift-F1 | Toggle overwrite mode on or off. |
| Ctrl-F5 | Toggle word wrap on or off. |
| Ctrl-F7 | Make where the caret is the current selection, and move the caret to where the selection was (ie toggle caret and selection). |

**Keystroke equivalents in the Select menu**

| | |
|---|---|
| Ctrl-Z | Clear selection. |
| Ctrl-X | Delete selection. |
| Ctrl-C | Copy selection to caret. |
| Ctrl-V | Move selection to caret. |
| 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. |
| Ctrl-F4 | Indent text block. |
| F5 | Display **GoTo** dialogue box. |
| F6 | If no block is selected, select the single character after the caret. If a block is selected, and the caret is outside it, extend the selection up to the caret. If a |

| | |
|---|---|
| | block is selected and the caret is inside it, cut the block from the caret position to the nearest end of the block. |
| Shift-F6 | Clear the current selection. |
| F7 | Copy the selected block at the current caret position. |
| Shift-F7 | Move the current selection to the caret position. |
| 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 Find menu

Note: these keystroke definitions only come into play once the **Find** dialogue box has been displayed (eg 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 Edit file into a new window. |
| Shift-F2 | Open a dialogue box enabling you to insert an existing SrcEdit file at the caret position. |
| F3 | Save the file in the current window. This is a shortcut to the normal Save as dialogue box. |

# Part 3 - Non-interactive tools

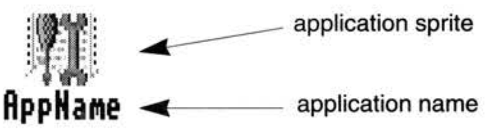
# 8 General features

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

As described in the chapter entitled *Working in the* DDE on page 19, the large number of programming tools forming the Desktop Development Environment 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 DDE 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). The interactive DDE tools all have different icons, but the application icons of the non-interactive tools are all similar:



The icon shows a spanner and screwdriver (representing an application tool), with the name of the application beneath.

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.

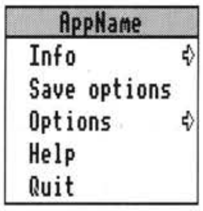
All the non-interactive DDE tools are implemented as command line programs provided with RISC OS desktop interfaces by the FrontEnd relocatable module, but you do not need to be aware of this when using them, as command lines are automatically generated from your settings of the desktop interface of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

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

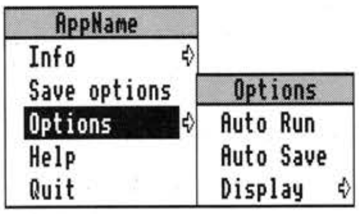
Clicking Menu on the application icon gives the following main menu:



**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:

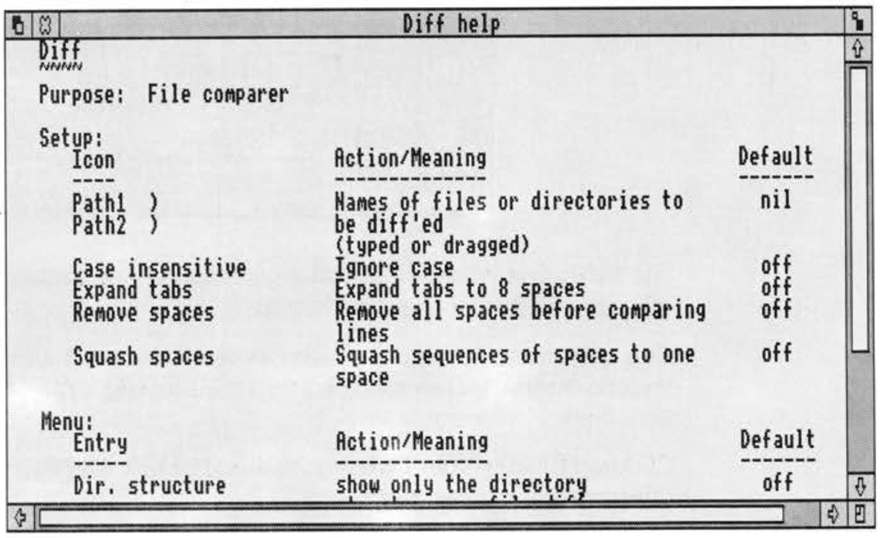


**Auto Run** will cause the command-line command to be run immediately when a file is dragged onto the icon on the icon bar, without first displaying the SetUp dialogue box. Options remain as they are currently set.

**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 191. 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:



```
                              Diff help
Diff
Purpose:  File comparer

Setup:
   Icon                Action/Meaning                   Default
   ----                --------------                   -------
   Path1  )            Names of files or directories to   nil
   Path2  )            be diff'ed
                       (typed or dragged)
   Case insensitive    Ignore case                        off
   Expand tabs         Expand tabs to 8 spaces            off
   Remove spaces       Remove all spaces before comparing off
                       lines
   Squash spaces       Squash sequences of spaces to one  off
                       space

Menu:
   Entry               Action/Meaning                   Default
   -----               --------------                   -------
   Dir. structure      show only the directory            off
```

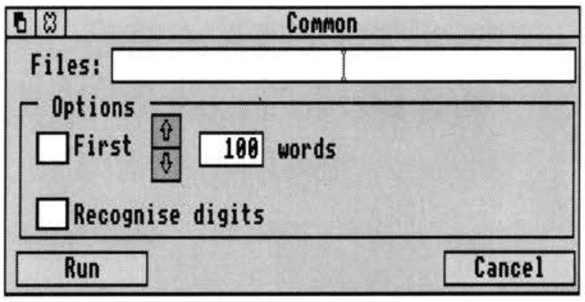
**Quit** quits the application.

## The Setup box

When working in the unmanaged way, ie with the tool application icon on the icon bar, clicking Select on this icon or dragging the name of an input file 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 (eg 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 Common:



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

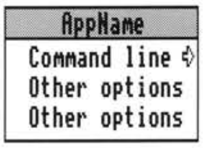
**Run** runs the tool with the options as set, starting a multitasking task performing the non-interactive job specified. This multitasking depends on the presence of the TaskWindow relocatable module.

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, and any extra options set from the **Other options** part of the above menu.

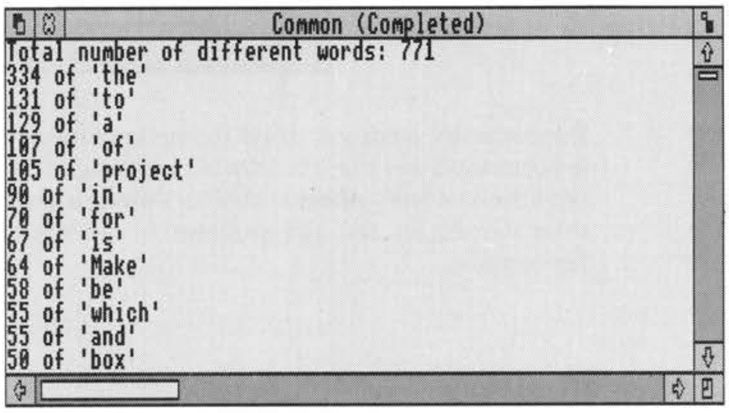
**Other options** are a set of options specific to the particular application.

## Output

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

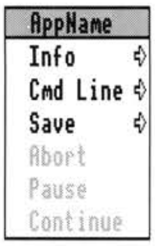
### The Text window

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, ie Running, Completed, or Paused. An example of a Text window using the application Common is:



```
Total number of different words: 771
334 of 'the'
131 of 'to'
129 of 'a'
107 of 'of'
105 of 'project'
90 of 'in'
70 of 'for'
67 of 'is'
64 of 'Make'
58 of 'be'
55 of 'which'
55 of 'and'
50 of 'box'
```

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.

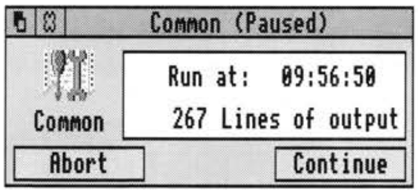
**Abort** aborts a running program.

**Pause** pauses a running program.

**Continue** continues a paused program.

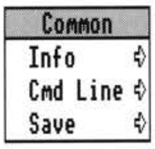
## The Summary window

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.

# 9 AMU

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. All the facilities provided by AMU are also provided by Make, which in addition assists you in constructing your makefiles. It is therefore recommended that you use Make rather than AMU, except where extreme memory shortage makes the larger size of Make a problem and the extra facilities are not needed.

Since use of AMU is deprecated, the description in this chapter is brief. For details of makefile syntax, see *Appendix A - Makefile syntax*. Some details described in the chapter entitled *Make* on page 79 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.

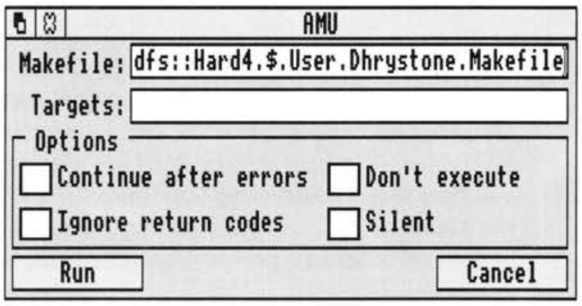
Each time that AMU is run, a work directory is set up for that job as the directory containing the makefile. For the effect of the work directory on each tool, see the chapters on individual tools such as the language processors CC and ObjAsm in this and accompanying user guides.

AMU is one of the non-interactive DDE tools, its desktop user interface being provided by the FrontEnd module. It shares many common features with the other non-interactive tools. These common features are described in the chapter entitled *General features* on page 117.

## 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:


AMU

Makefile: dfs::Hard4.$.User.Dhrystone.Makefile

Targets:

Options

☐ Continue after errors ☐ Don't execute

☐ Ignore return codes ☐ Silent

**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 writable 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.

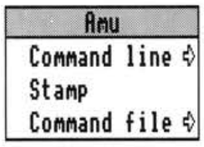
The **Continue after errors** option causes the make job to continue after one of the commands issued by it has returned a bad return code (signalling an error). When the job continues, only those branches of the make job which don't depend on the failed command are executed.

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:



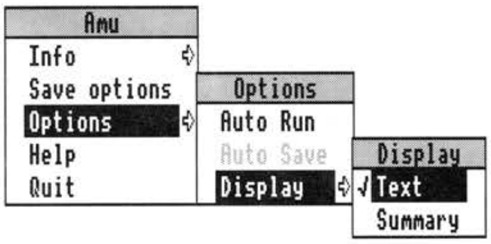
The **Command line** option on the above menu has the standard purpose for non-interactive DDE tools as described in chapter entitled *General features* on page 117.

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 to 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

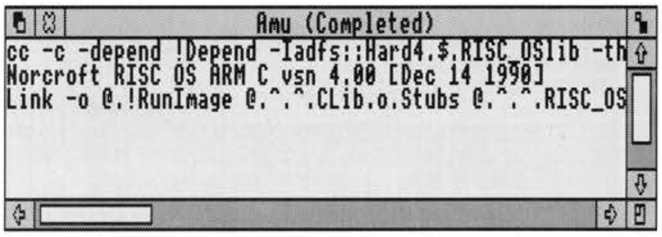
Clicking Menu on the AMU application icon on the icon bar gives access to the following options:



For a description of each option in the application menu see chapter entitled *General features* on page 117.

## 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:



```
cc -c -depend !Depend -Iadfs::Hard4.$.RISC_OSlib -th
Norcroft RISC OS ARM C vsn 4.00 [Dec 14 1990]
Link -o @.!RunImage @.^.^.CLib.o.Stubs @.^.^.RISC_OS
```

## 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:

| | |
|---|---|
| `-f` *`makefile`* | **Makefile** name (makefile defaults to `Makefile` if omitted) |
| `-i` | **Ignore return codes** |
| `-k` | **Continue after errors** |
| `-n` | **Don't execute** |
| `-o` *`commandfile`* | Specify **Command file** as on SetUp menu |
| `-s` | **Silent** |
| `-t` | Equivalent to **Stamp** on the SetUp menu |

*`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.

# 10 Common

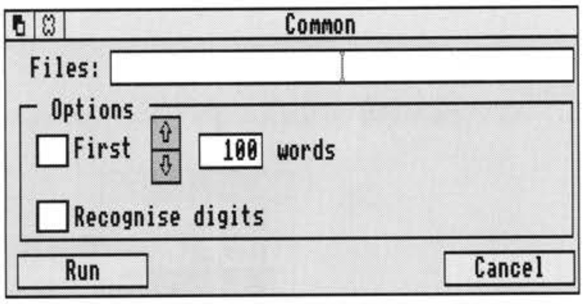
This application tool counts the frequency of words in a file. It allows you to choose between:

- displaying the number of times every word in a file occurs (default);
- displaying the number of times only the most common words in a file occur.

You can also choose whether or not to treat numerics as words.

## The SetUp dialogue box

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



The **Files** writable icon allows you to specify the names of files to be processed (typed in or dragged from a directory display).

### SetUp options

**First** allows you to display only the most common words in a file. You can specify how many of the most common words are to be displayed by:

- using the adjacent arrow icons to increase or decrease the number of words appearing in the **words** box;
- editing the **words** box by clicking Select inside it and typing in a number.

If **First** has been chosen and several different words occur the same number of times in a file, then Common will display the frequency of each of the different words but, for the purposes of **words**, treat them all as if they were only one word.

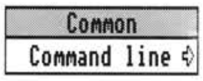
The default is off (ie consider every word in the file).

**Recognise digit**, if chosen, will force Common to count numerics as words.

The default is off (ie ignore digits).

### The SetUp menu

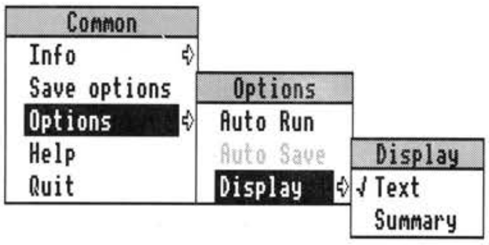
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the Common **Command line** option see the section entitled *Command line interface* on page 132.

## The Application menu

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



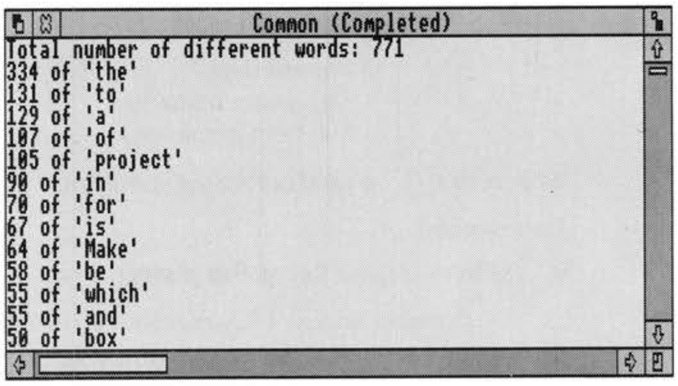
For a description of each option in the application menu see the chapter entitled *General features* on page 117.

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

## Example output

The output of Common appears in one of the standard non-interactive tool output windows. For more details of these see the section entitled *Output* on page 121.

The following is an example of a large text file analysed by Common and displayed in an output text window:



```
Total number of different words: 771
334 of 'the'
131 of 'to'
129 of 'a'
107 of 'of'
105 of 'project'
90 of 'in'
70 of 'for'
67 of 'is'
64 of 'Make'
58 of 'be'
55 of 'which'
55 of 'and'
50 of 'box'
```

Common always returns the following information in the output text window:

- The total number of different words. This is shown at the top of the output text window, eg:

  Total number of different words: 771

- A list of all the words (or, if **First** has been chosen, only the most common words) ranked in order of frequency. In the file analysed above the most common word is `the`, occurring 334 times.

In the above example the words `which` and `and` both occur 55 times; if the **First** option had been set these two words would be considered as one word.

## Command line interface

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

`Common [-f` *`nwords`*`] [-n±]` *`filename`*

| | |
|---|---|
| `-f` | if present Common will only look at the number of word counts specified by *`nwords`*. |
| `-n±` | if present then<br>– = ignore numerics.<br>+ = treat numerics as words. |
| *`filename`* | a valid pathname specifying a file. |

For example:

- If **First** (`-f`) and **Recognise digits** (`-n`) are not chosen:

```
Common adfs::Username.Testfile
```

- If **First** and **Recognise digits** are chosen, and *`nwords`* set to 50:

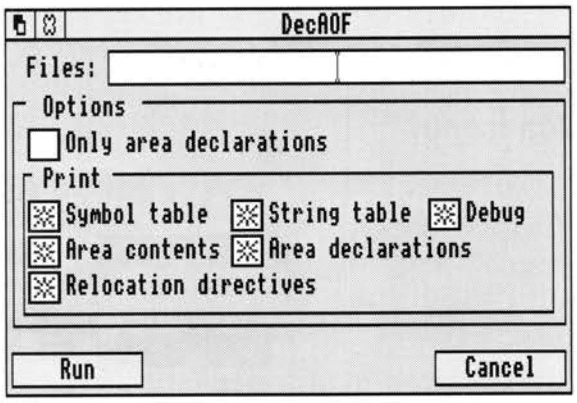
```
Common -f 50 -n+ adfs::Username.Testfile
```

# 11 DecAOF

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 Acorn 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.

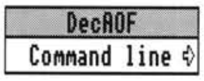
**Area contents** prints the area contents in hex.

**Area declarations** prints the area declarations.

**Relocation directives** prints linker relocation directives.

### The SetUp menu

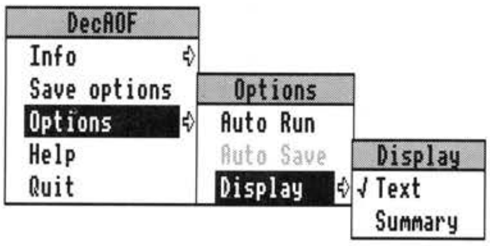
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the DecAOF **Command line** option see the section entitled *Command line interface* on page 135

## The Application menu

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



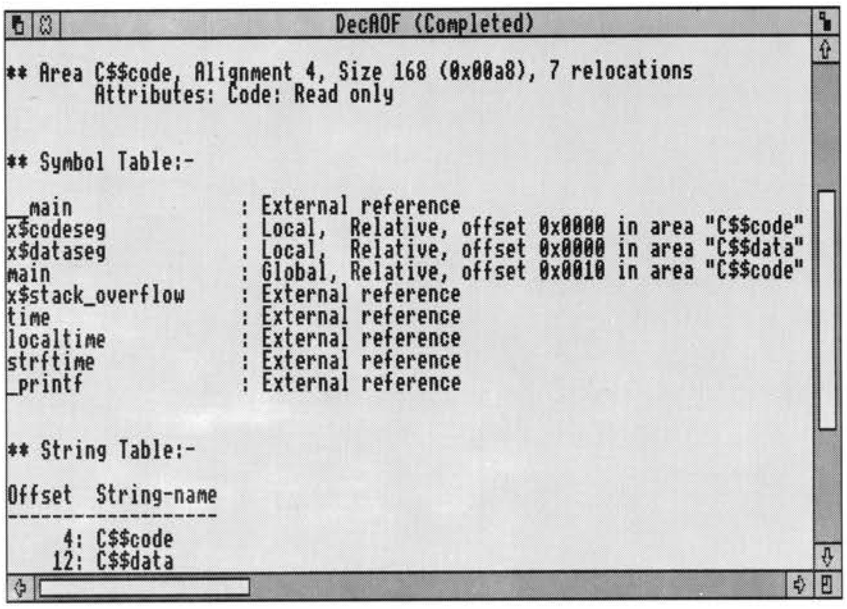
For a description of each option in the application menu see the chapter entitled *General features* on page 117.

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 entitled *Output* on page 121.

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



```
** Area C$$code, Alignment 4, Size 168 (0x00a8), 7 relocations
        Attributes: Code: Read only

** Symbol Table:-

__main              : External reference
x$codeseg           : Local,  Relative, offset 0x0000 in area "C$$code"
x$dataseg           : Local,  Relative, offset 0x0000 in area "C$$data"
main                : Global, Relative, offset 0x0010 in area "C$$code"
x$stack_overflow    : External reference
time                : External reference
localtime           : External reference
strftime            : External reference
_printf             : External reference


** String Table:-

Offset  String-name
-------------------
     4: C$$code
    12: C$$data
```

## 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

| | |
|---|---|
| `-b` | Print only the area declarations |
| `-a` | Print area contents in hex (implies `-d`) |
| `-d` | print area declarations |
| `-r` | print relocation directives (implies `-d`) |
| `-g` | print debug areas |

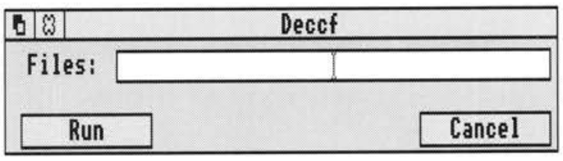
| | |
|---|---|
| `-s` | print symbol table |
| `-t` | print string table |
| *`filename`* | a valid pathname specifying an AOF file |

# DecCF

DecCF analyses one or more object or library files and returns information about the chunks in each file.

## The SetUp dialogue box

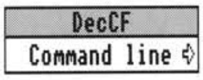
Clicking Select on the application icon or dragging the name of a file from a directory display to the icon brings up the SetUp dialogue box:



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 Acorn Library Format (ALF) or Arm Object Format (AOF) files.

### The SetUp menu

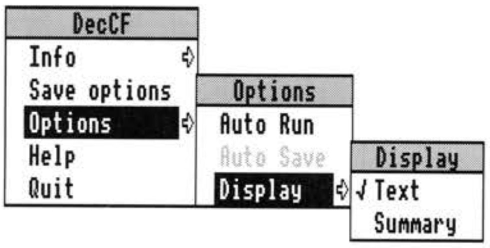
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the DecCF **Command line** option see the section entitled *Command line interface* on page 139.

## The Application menu

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



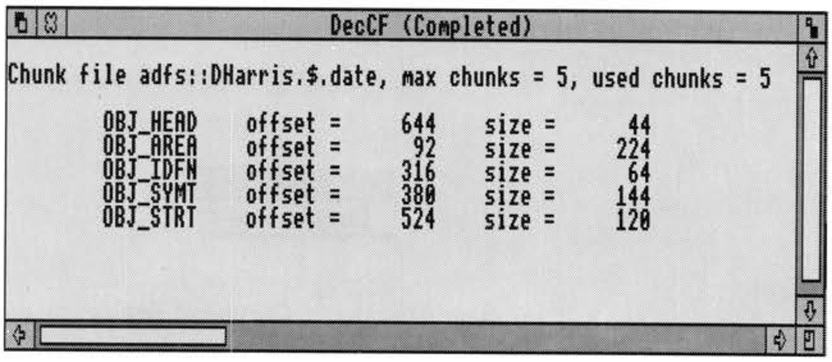
For a description of each option in the application menu see the chapter entitled *General features* on page 117.

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

## Example output

The output of DecCF appears in one of the standard non-interactive tool output windows. For more details of these see the section entitled *Output* on page 121.

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



```
Chunk file adfs::DHarris.$.date, max chunks = 5, used chunks = 5

        OBJ_HEAD    offset =    644   size =     44
        OBJ_AREA    offset =     92   size =    224
        OBJ_IDFN    offset =    316   size =     64
        OBJ_SYMT    offset =    380   size =    144
        OBJ_STRT    offset =    524   size =    120
```

For each file in the **Files** writable icon DecCF will return:

- the maximum number of permissible chunks;
- the number of used chunks;
- the offset and size of each chunk.

## Command line interface

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

`DecCF` *`filename`* `[`*`filename...`*`]`

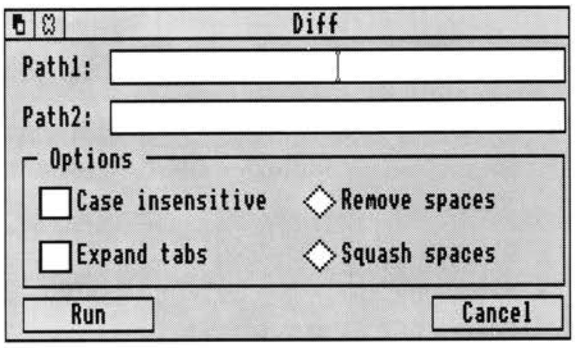
| | |
|---|---|
| *`filename`* | a valid pathname specifying an ALF or AOF file |

# 13 Diff

Diff displays the textual differences between two files 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:



**Path1** and **Path2** allow you to specify the names of files 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 option was chosen.

**Expand tabs** substitutes tabs by multiples of eight spaces.

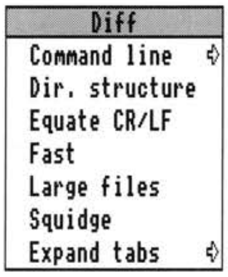
**Remove spaces** removes all spaces before comparing lines. This is useful if you wish to examine two files you have been editing but are not interested in any extra spaces you may have introduced.

**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, eg assembler code, and you want to display lines where spaces have been deleted or added, it is essential to ensure that neither **Remove spaces** or **Squash spaces** have been chosen.

## The SetUp menu

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



**Command line** enables you to examine or edit the actual command line. For more information on this option see the section entitled *Command line interface* on page 145.

**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.

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

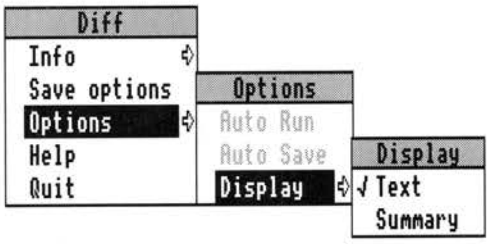
**Large files** is helpful where very large files are being compared. It sometimes happens that two files differ completely over a large section of text because, for instance, you may have edited in several paragraphs or even several pages of text. Ordinarily Diff would not be able to detect this and 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:



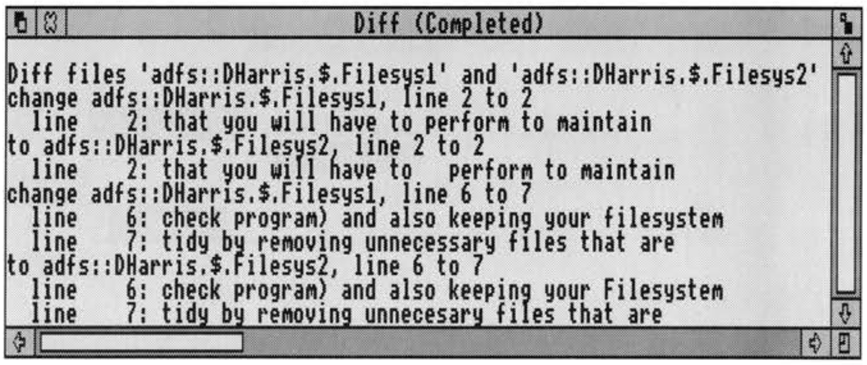
For a description of each option in the application menu see the chapter entitled *General features* on page 117.

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

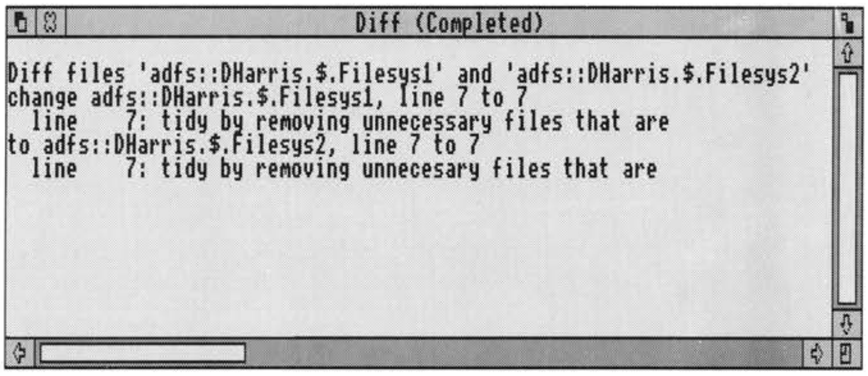
## Example output

The output of Diff appears in one of the standard non-interactive tool output windows. For more details of these see the section entitled *Output* on page 121.

The following two windows show examples of the output from Diff:



```
Diff files 'adfs::DHarris.$.Filesys1' and 'adfs::DHarris.$.Filesys2'
change adfs::DHarris.$.Filesys1, line 2 to 2
  line    2: that you will have to perform to maintain
to adfs::DHarris.$.Filesys2, line 2 to 2
  line    2: that you will have to   perform to maintain
change adfs::DHarris.$.Filesys1, line 6 to 7
  line    6: check program) and also keeping your filesystem
  line    7: tidy by removing unnecessary files that are
to adfs::DHarris.$.Filesys2, line 6 to 7
  line    6: check program) and also keeping your Filesystem
  line    7: tidy by removing unnecesary files that are
```



```
Diff files 'adfs::DHarris.$.Filesys1' and 'adfs::DHarris.$.Filesys2'
change adfs::DHarris.$.Filesys1, line 7 to 7
  line    7: tidy by removing unnecessary files that are
to adfs::DHarris.$.Filesys2, line 7 to 7
  line    7: tidy by removing unnecesary files that are
```

In the first example two text files have been analysed by Diff without any options being set. Three differences have been found:

- on line 2 of the second file there are two extra spaces before the word `perform`.
- on line 6 of the second file `Filesystem` has been spelt with a capital `F`.
- on line 7 of the second file `unnecessary` has been spelt with only one s.

In the second example the same two files are compared but the **Case insensitive** and **Remove spaces** options have been chosen. The result is that only the different spelling of the word `unnecessary` 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

| | |
|---|---|
| `-d` | Show only the directory structure, do not display any differences |
| `-e` | Equate CR and LF |
| `-f` | Perform a fast Diff, all options except `-d` ignored, do not display any differences |
| `-l` | Handle large files more effectively (but more slowly) |
| `-n` | Ignore case sensitivity when comparing letters |
| `-r` | Remove all spaces before comparing lines |
| `-s` | Squash sequences of spaces to one space |
| `-t` | As for `-r`, but `-s` when between two alphanumeric characters |
| `-x` | Expand tabs to spaces (tab stops at multiples of 8) |
| `-X`*n* | Expand tabs to spaces (tab stops at multiples of *n*) |

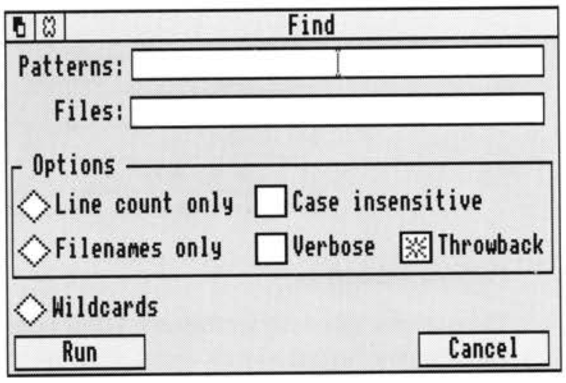
# 14 Find

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.

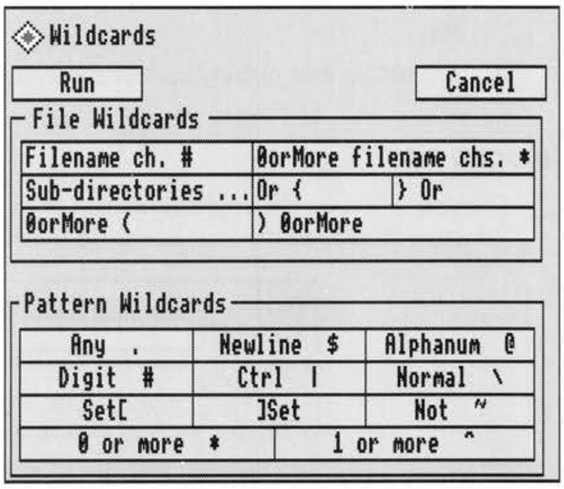
**Filenames only** lists only the names of files matching the pattern.

**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:



### 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.

| | |
|---|---|
| **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 |**

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`.

**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::Fred#` | will search files `Fred1` and `Freda`, but not `Fred13`, `Frederick` etc. |
| `Find adfs::Fr#d` | will search files `Fred` and `Fr2d`, but not `Freld`, `Freed` etc. |

**0orMore filename chs. *** inserts an asterisk immediately before the caret. This character will match any sequence of filename characters except **.**, `{`, and `}`.

For example:

| | |
|---|---|
| `Find adfs::Fred*` | will search files `Fred1` and `Freda`, and also `Fred13`, `Frederick` etc. |
| `Find adfs::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 adfs::W.rel.{atype,btype,ctype}
```

would search all three files inside the braces, ie `atype`, `btype` and `ctype`.

**0orMore (** inserts a left bracket immediately before the caret.

**) 0orMore** inserts a right bracket immediately before the caret.

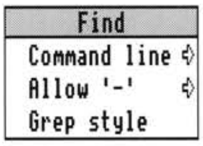
The preceding two options insert opening and closing brackets into the **Files** writable icon. You can then manually insert one or more filename characters between these brackets and Find will search any files with none, one or more occurrences of the characters you put inside the brackets.

For example:

| | |
|---|---|
| `Find adfs::Fr(e)d` | will search files `Frd`, `Fred` and `Freed`, but not `Frid`. |
| `Find adfs::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 following menu on the screen:



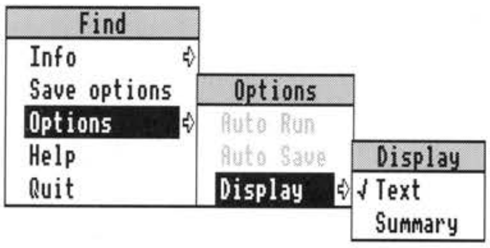
For a description of the Find **Command line** option see the section entitled *Command line interface* on page 153.

The **Allow '–'** option enables you to specify a second pattern which will be matched even if it begins with a –. This second pattern will be searched for in conjunction with the pattern you have inserted into the **Patterns** writable icon.

**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:



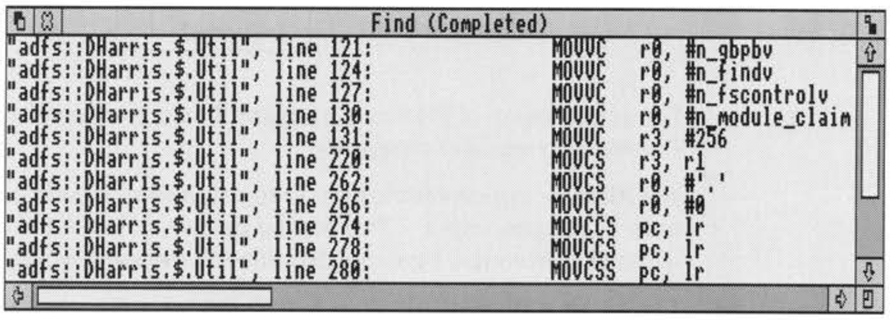
For a description of each option in the application menu see the chapter entitled *General features* on page 117.

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 entitled *Output* on page 121.

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


```
Find (Completed)
"adfs::DHarris.$.Util", line 121:                  MOVVC   r0, #n_gbpbv
"adfs::DHarris.$.Util", line 124:                  MOVVC   r0, #n_findv
"adfs::DHarris.$.Util", line 127:                  MOVVC   r0, #n_fscontrolv
"adfs::DHarris.$.Util", line 130:                  MOVVC   r0, #n_module_claim
"adfs::DHarris.$.Util", line 131:                  MOVVC   r3, #256
"adfs::DHarris.$.Util", line 220:                  MOVCS   r3, r1
"adfs::DHarris.$.Util", line 262:                  MOVCS   r0, #'.'
"adfs::DHarris.$.Util", line 266:                  MOVCC   r0, #0
"adfs::DHarris.$.Util", line 274:                  MOVCCS  pc, lr
"adfs::DHarris.$.Util", line 278:                  MOVCCS  pc, lr
"adfs::DHarris.$.Util", line 280:                  MOVCSS  pc, lr
```

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 `MOVC` in an assembler source file. Instructions where the fourth letter was not a C or V, such as `MOVS`, `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 `Util` 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. |

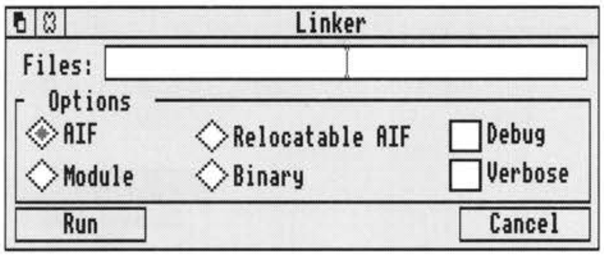
# 15 Link

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 libraryfiles 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 (eg 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 entitled *Libraries* on page 159.

**AIF** generates Application 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 the RISC OS *Programmer's Reference manual*, ANSI C *Release* 4, and the section entitled *Relocatable modules* on page 164 for more details on relocatable modules.

**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 entitled *Relocatable* AIF *images* on page 163 for more details.

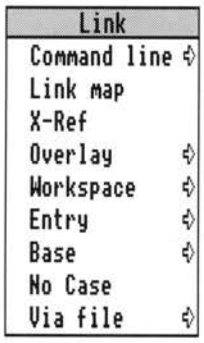
**Binary** generates a plain binary image (without an image header or any specific image format). Generally it is only used when writing completely in assembler. Programs written entirely or partly in C or other high level languages cannot usually use this format.

**Debug** allows you to debug a program with the desktop debugger DDT. See the chapter entitled *Desktop debugging tool* on page 31 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 entitled *Output* on page 157 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 following menu on the screen:



**Command line** allows you to specify the command line to be presented to the underlying `Link` command line tool. Refer to the section entitled *Command line interface* on page 166 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 entitled *Link map option* on page 162 for more information. For details on linker areas, see the section entitled AOF in *Appendix* E - *Code file formats*.

**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 entitled X-*Ref option* on page 162 for more details.

**Overlay** generates an overlaid image using the specified overlay description file. For details of overlay description files, see the section entitled *Overlay description files* on page 161. 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 &8000 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.

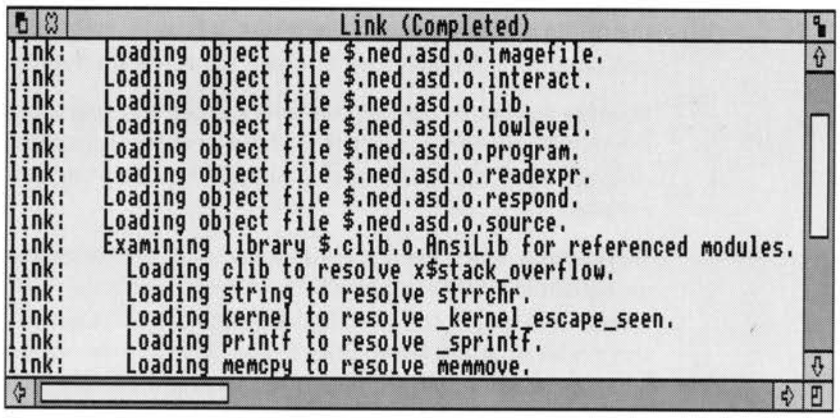
**Note:** The **Base**, **Workspace** and **Entry** options require a numeric argument to be entered in the associated submenu. You can prefix this argument by `&` or `0X` to specify a hexadecimal value. You can postfix it by `k` for 2^10 and `m` for 2^20.

## Output

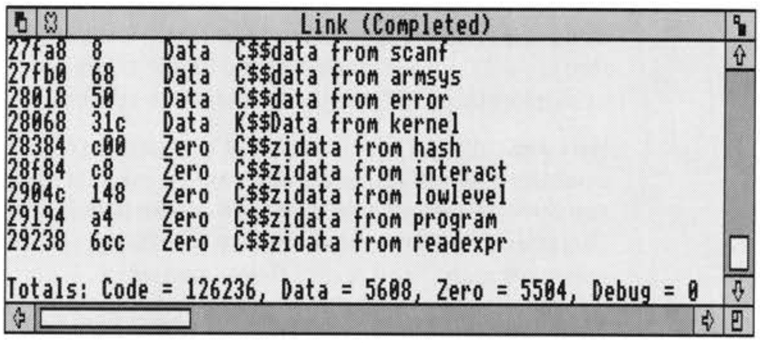
The Output window displays information printed when you have selected the **Verbose**, **Link map** or **X-Ref** options. It also displays any error messages generated while linking.

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 entitled X-*Ref option* on page 162.

**Verbose** output:


Link (Completed)
```
link:   Loading object file $.ned.asd.o.imagefile.
link:   Loading object file $.ned.asd.o.interact.
link:   Loading object file $.ned.asd.o.lib.
link:   Loading object file $.ned.asd.o.lowlevel.
link:   Loading object file $.ned.asd.o.program.
link:   Loading object file $.ned.asd.o.readexpr.
link:   Loading object file $.ned.asd.o.respond.
link:   Loading object file $.ned.asd.o.source.
link:   Examining library $.clib.o.AnsiLib for referenced modules.
link:     Loading clib to resolve x$stack_overflow.
link:     Loading string to resolve strrchr.
link:     Loading kernel to resolve _kernel_escape_seen.
link:     Loading printf to resolve _sprintf.
link:     Loading memcpy to resolve memmove.
```

**Link map** output:


Link (Completed)
```
27fa8  8     Data  C$$data from scanf
27fb0  68    Data  C$$data from armsys
28018  50    Data  C$$data from error
28068  31c   Data  K$$Data from kernel
28384  c00   Zero  C$$zidata from hash
28f84  c8    Zero  C$$zidata from interact
2904c  148   Zero  C$$zidata from lowlevel
29194  a4    Zero  C$$zidata from program
29238  6cc   Zero  C$$zidata from readexpr

Totals: Code = 126236, Data = 5608, Zero = 5504, Debug = 0
```

## 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.

For example, if you are using the libraries RISC_OSLib and ANSILib, you must drag RISC_OSLib first and then ANSILib to ensure they appear in the right order. If you are using the shared C library stubs instead of ANSILib the order is unimportant, since the shared C library stubs is an object file which defines all of the symbols in the shared C library. Also note that, because of the ordering constraints, libraries containing circular references cannot easily be linked.

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, `$.clib.o.stubs`) or the unshared library, `$.clib.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* F - ARM *procedure call standard*
- the library's initialisation routines have been called.

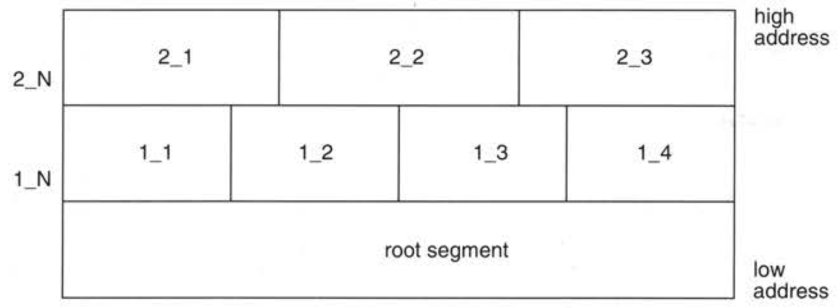
Refer to the chapter entitled *Machine-specific features* in ANSI C *Release* 4 for details on how to initialise the common run-time kernel distributed with the C library.

## Generating overlaid programs

An introduction to overlays is given in ANSI C *Release* 4. 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 ANSI C *Release* 4) 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 make it smaller.

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

On disc, an overlaid program is organised as a RISC OS application. The components of the application (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 absolute 1_2 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:

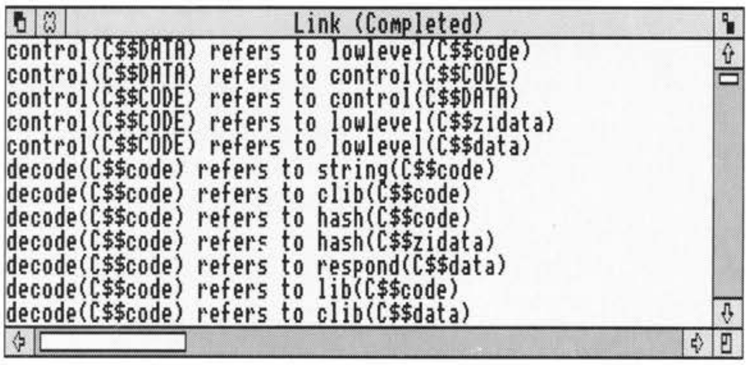
```
1_1 edit1 edit2 editdata(C$$code,C$$data) sort
```

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**:



```
control(C$$DATA) refers to lowlevel(C$$code)
control(C$$DATA) refers to control(C$$CODE)
control(C$$CODE) refers to control(C$$DATA)
control(C$$CODE) refers to lowlevel(C$$zidata)
control(C$$CODE) refers to lowlevel(C$$data)
decode(C$$code) refers to string(C$$code)
decode(C$$code) refers to clib(C$$code)
decode(C$$code) refers to hash(C$$code)
decode(C$$code) refers to hash(C$$zidata)
decode(C$$code) refers to respond(C$$data)
decode(C$$code) refers to lib(C$$code)
decode(C$$code) refers to clib(C$$data)
```

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 indirects 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 ANSI C *Release* 4) 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 `$.clib.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 &8000 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.

### Utilities

Utility or transient programs (filetype `FFC`) can be linked as relocatable AIF images. Use the SetType command to set the filetype correctly after linking:

```
*SetType image Utility
```

**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*.

## 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|
        ...
```

**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.

## 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 (eg `C$$code`) Link generates two symbols:

| | |
|---|---|
| *`area_name`*`$$Base` | Address of the start of the area. |
| *`area_name`*`$$Limit` | Address of the byte beyond the end of the area. |
| *`area_name`* | is 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.

| | |
|---|---|
| `Image$$RO$$Base` | Address of the start of the read-only (code) area. |
| `Image$$RO$$Limit` | Address of the byte beyond the end of the code area. |
| `Image$$RW$$Base` | Address of the start of the read/write (data) area. |
| `Image$$RW$$Limit` | Address of the byte beyond the end of the data area. |
| `Image$$ZI$$Base` | Address of the start of the zero-initialised (bss) area. |
| `Image$$ZI$$Limit` | Address of the byte beyond the end of the bss area. |
| `Image$$RW0$$Base` | Address of the start of the debugging tables. |
| `Image$$RW0$$Limit` | Address of the byte beyond the end of the debugging tables. |

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 (eg strings, floating-point constants etc).

## Command line interface

The format of the Link command is:

```
Link options file_list
```

### Options

| | |
|---|---|
| `-h` | Display a screen of help text |
| `-o` *`image_file`* | Place output in named *`image_file`* |
| `-d` | Include debugging tables in the output image suitable for use by the desktop debugger |
| `-ov` *`overlay_file`* | Generate an overlaid application as directed by commands in *`overlay_file`* |
| `-m` | Generate relocatable module output |
| `-r` | Generate relocatable AIF output |
| `-aif` | Generate normal AIF output (default) |
| `-aof` | Generate partially linked AOF output suitable for inclusion in a subsequent link step |
| `-bin` | Generate a plain binary image |
| `-w` *`n`* | Reserve *`n`* bytes of workspace for a relocatable image |
| `-e` *`n`* | Set the image entry point to the address specified by *`n`* |
| `-b` *`n`* | Set the image base to the address specified by *`n`* |
| `-c` | Make matching of symbols case insensitive |
| `-map` | Generate a map of the base and size of each AREA and display totals for code, data, zero-init and debugging AREAs |
| `-x` | Display a list of references between linker areas |
| `-v` | Display messages indicating progress of the link operation |
| `-via` *`via_file`* | Take further input file names from *`via_file`* |
| `-s` *`symbol_file`* | Produce a symbol table dump in *`symbol_file`* |

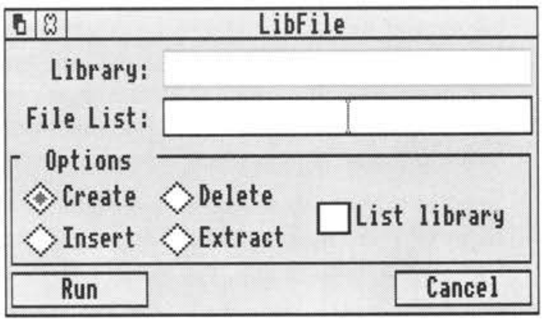
# 16 LibFile

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* E - *Code file formats*.

LibFile supersedes the ObjLib tool previously distributed with the Software Developers Toolkit. Refer to the section entitled *Command line interface* on page 171 for more details.

## The SetUp dialogue box

Click Select on the application icon. This displays the SetUp dialogue box:



Each of the options in the SetUp dialogue box is described overleaf.

### 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.

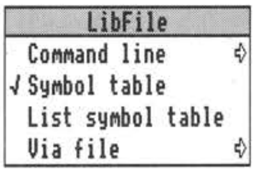
**Insert** adds the files in **File List** to the specified library. Files of the same name in the library will be replaced.

**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 entitled *Command line interface* on page 171.

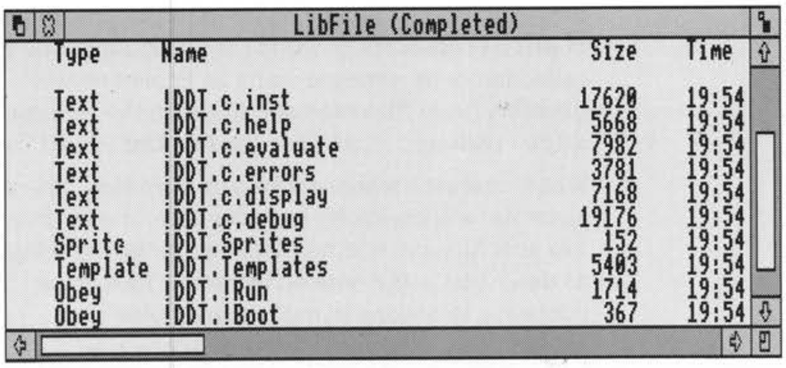
**Symbol table** adds an external symbol table, as used by Link, to the library. External symbols in any object files in the library are placed in the symbol table. Non object files are ignored. By default, this option is on.

**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.

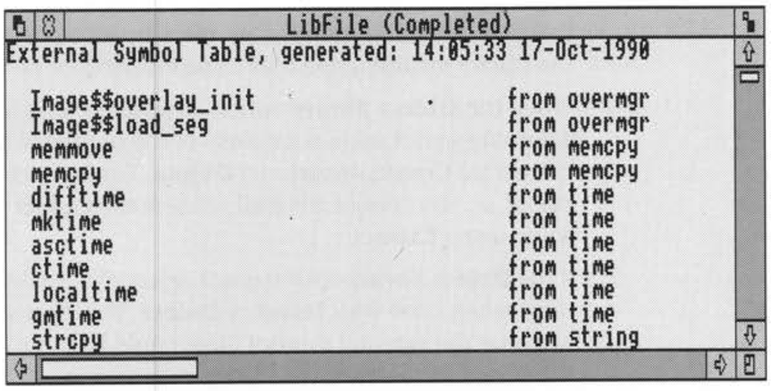
**Via file** allows you to set up a list of files to be used in one file called a Via file. When creating or maintaining libraries with a large number of files it may become tedious having to drag all the files to the **File List** every time, especially if they are in different directories. Enter the 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.



```
Type     Name                    Size    Time
Text     !DDT.c.inst            17620   19:54
Text     !DDT.c.help             5668   19:54
Text     !DDT.c.evaluate         7982   19:54
Text     !DDT.c.errors           3781   19:54
Text     !DDT.c.display          7168   19:54
Text     !DDT.c.debug           19176   19:54
Sprite   !DDT.Sprites             152   19:54
Template !DDT.Templates          5403   19:54
Obey     !DDT.!Run               1714   19:54
Obey     !DDT.!Boot               367   19:54
```



```
External Symbol Table, generated: 14:05:33 17-Oct-1990

  Image$$overlay_init            from overmgr
  Image$$load_seg                from overmgr
  memmove                        from memcpy
  memcpy                         from memcpy
  difftime                       from time
  mktime                         from time
  asctime                        from time
  ctime                          from time
  localtime                      from time
  gmtime                         from time
  strcpy                         from string
```

Notes:

1 Any directories in the **File List** to be archived will be recursively archived (ie 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 (ie 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.$.PDUtils`, it would not be very useful to someone called Ian or to someone using an Econet network. Instead, set your current directory (from the command line with the *Dir command) to `adfs::Edward.$` 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 **Object library** 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 **Object library** option is deselected when using **Extract**.

**7** If the **Object library** 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-1990`. 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 *`file_list`*. |
| `-i` | **Insert** files in *`file_list`*, replace existing members. |
| `-d` | **Delete** the members in *`file_list`*. |
| `-e` | **Extract** members in *`file_list`* 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 timestamps of all bits 0 when creating or updating library. |
| `-v` *`file`* | Take additional arguments from *`file`*. |
| `-csd` *`dir`* | Place relative filenames in *`dir`* when extracting file. |

**Notes**:

1 Multiple options may be specified on a single options argument. For example, `-clso` 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 `-csd` option. This option means "behave as though the directory specified after the `-csd` 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 `-csd` 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.

**5** LibFile supersedes the ObjLib tool previously distributed with Software Developer's Toolkit. If you have makefiles which depend on the use of ObjLib, you can use the following alias to define an ObjLib command:

```
Set Alias$ObjLib LibFile -o %*0
```

This will work provided you do not use the `-List` or `-File` options with ObjLib. If you do use these options, edit the makefile and use the appropriate LibFile command.

## 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 -csd :Ian.$.PDUtils :0.PDLib *
```

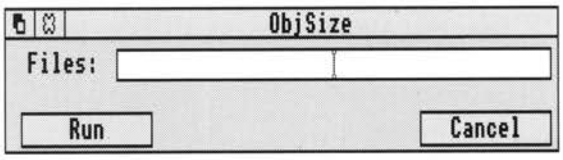
Extract all the files from `:0.PDLib` and put them in the directory `:Ian.$.PDUtils`.

# 17 ObjSize

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

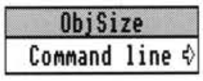
Clicking Select on the application icon or dragging the name of a file from a directory display to the icon brings up the SetUp dialogue box:



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

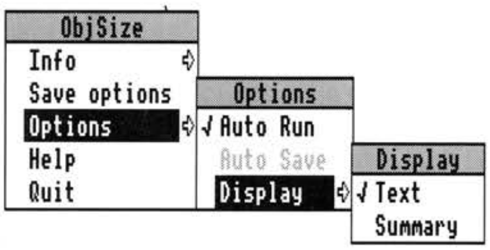
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the ObjSize **Command line** option see the section entitled *Command line interface* on page 175.

## The Application menu

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



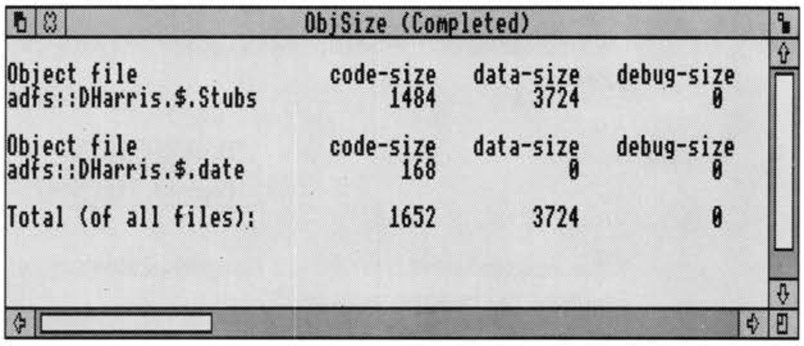
For a description of each option in the application menu see the chapter entitled *General features* on page 117.

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 entitled *Output* on page 121.

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



```
Object file                 code-size  data-size  debug-size
adfs::DHarris.$.Stubs            1484       3724           0

Object file                 code-size  data-size  debug-size
adfs::DHarris.$.date              168          0           0

Total (of all files):            1652       3724           0
```

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. |

# 18 Squeeze

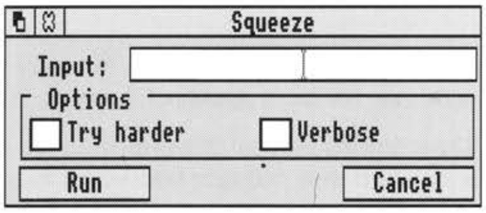
Squeeze compresses an executable ARM-code program, saving disc space and often making the program load faster.

Relocatable modules can be squeezed but must be run rather than RMLoaded.

Squeeze converts a module to 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.

## 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:



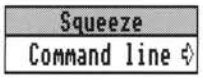
The **Input** writable icon allows you to specify the name of a file to be processed (typed in or dragged from a directory display). This file must be an AIF file.

**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

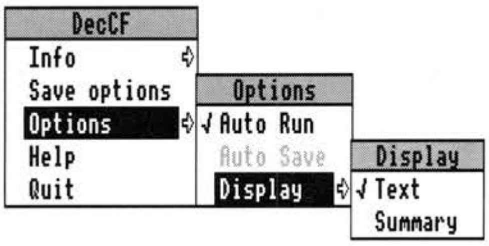
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the Squeeze **Command line** option see the section entitled *Command line interface* on page 179.

## The Application menu

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



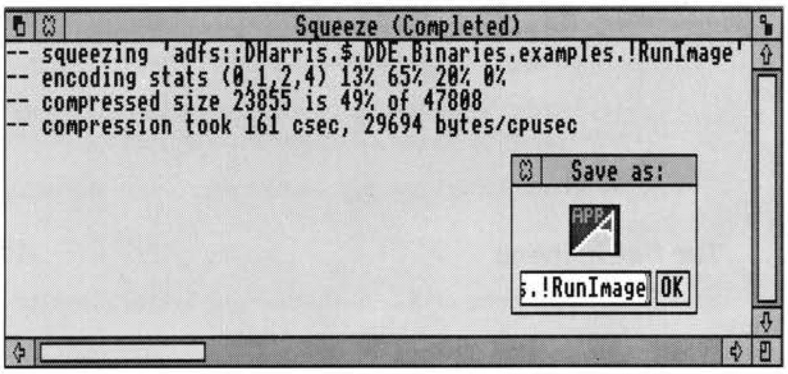
When **Auto save** is enabled, a squeezed file is saved to a suitable place automatically without producing a save dialogue box for you to drag the file from. **Auto save** is off by default, whereas **Auto Run** is on by default.

For a description of each option in the application menu see the chapter entitled *General features* on page 117.

## 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 entitled *Output* on page 121.

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):



```
-- squeezing 'adfs::DHarris.$.DDE.Binaries.examples.!RunImage'
-- encoding stats (0,1,2,4) 13% 65% 20% 0%
-- compressed size 23855 is 49% of 47808
-- compression took 161 csec, 29694 bytes/cpusec
```

## 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

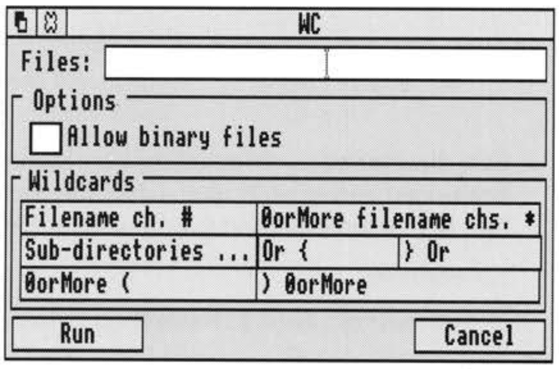
| | |
|---|---|
| `-f` | compress file regardless of size. |
| `-v` | output messages and compression statistics. |
| *`unsqueezed-file`* | a valid pathname specifying an input AIF file. |
| *`squeezed-file`* | a valid pathname specifying an output AIF file. |

# 19 WC

WC analyses one or more files and returns the number of lines, words, alphanumerics and characters in each file.

## 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 a file to be processed (typed in or dragged from a directory display).

### SetUp options

**Allow binary files** enables WC to analyse binary files (ignored by default).

The options offered under the heading of **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 the # character immediately before the caret. This character will match any single filename character except dot (`.`).

For example:

| | |
|---|---|
| `WC adfs::Fred#` | will search files `Fred1` and `Freda`, but not `Fred13`, `Frederick` etc. |
| `WC adfs::Fr#d` | will search files `Fred` and `Fr2d`, but not `Fre1d`, `Freed` etc. |

**0orMore filename chs. *** inserts the * character immediately before the caret. This character will match any sequence of filename characters except dot(`.`) or braces (`{}`).

For example:

| | |
|---|---|
| `WC adfs::Fred*` | will search files `Fred1` and `Freda`, and also `Fred13`, `Frederick` etc. |
| `WC adfs::Fr*d` | will search files `Fred` and `Fr2d`, and also `Freed`, `Fr123d` etc. |

**Sub-directories ...** inserts three dots immediately before the caret. It must be positioned immediately after a directory name. WC will then search all nominated files in that directory and in any subdirectories in that structure.

For example:

```
WC 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 curly brackets into the **Files** writable icon. You can then manually insert one or more filename characters between these brackets, separating each filename with a comma. Find will search all filenames inside the brackets.

For example:

```
Find adfs::W.rel.{atype,btype,ctype}
```

would search all three files inside the brackets, ie `atype`, `btype` and `ctype`.

**0orMore (** inserts an opening bracket immediately before the caret.

**) 0orMore** inserts a closing 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 WC will search any files with none, one or more occurrences of the characters you put inside the brackets.

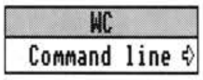
For example:

`WC adfs::Fr(e)d` will search files `Frd`, `Fred` and `Freed`, but not `Frid`.

`WC adfs::Fr(ie)d` will search files `Frd`, `Fried` and `Frieied`, but not `Frid`, `Frieed` or `Fred`.

## The SetUp menu

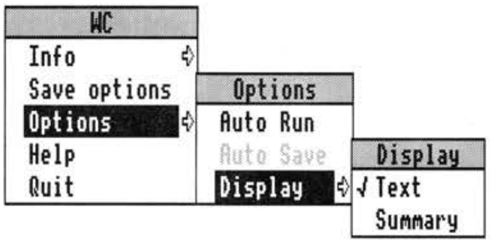
Clicking Menu on the SetUp dialogue box displays the following menu on the screen:



For a description of the WC **Command line** option see the section entitled *Command line interface* on page 184.

## The Application menu

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



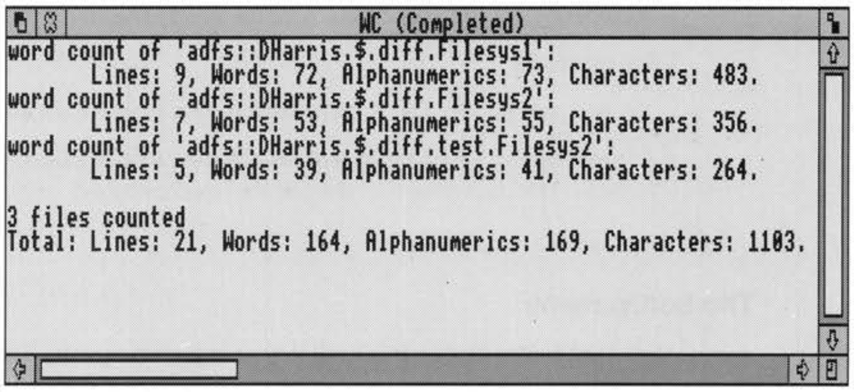
For a description of each option in the application menu see the chapter entitled *General features* on page 117.

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

## Example output

The output of WC appears in one of the standard non-interactive tool output windows. For more details of these see the section entitled *Output* on page 121.

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



```
word count of 'adfs::DHarris.$.diff.Filesys1':
        Lines: 9, Words: 72, Alphanumerics: 73, Characters: 483.
word count of 'adfs::DHarris.$.diff.Filesys2':
        Lines: 7, Words: 53, Alphanumerics: 55, Characters: 356.
word count of 'adfs::DHarris.$.diff.test.Filesys2':
        Lines: 5, Words: 39, Alphanumerics: 41, Characters: 264.

3 files counted
Total: Lines: 21, Words: 164, Alphanumerics: 169, Characters: 1103.
```

## Command line interface

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

```
WC [options] filepattern [filepattern...]
```

### Options

| | |
|---|---|
| `-b` | Do not ignore binary files. |

### Filepattern

| | |
|---|---|
| `#` | Match any filename character except dot (`.`) |
| `*` | Match 0 or more filename characters other than dot (`.`) or braces (`{}`). |
| `...` | Look in all sub-directories beneath the specified 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. |

# 20 Extending the DDE

The components of the DDE 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 components of the DDE in your products, or intending to add further tools to the DDE, 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 entitled *General features* on page 117. 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 DDE 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 components of the DDE:

- 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 (eg 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 messages file, and a !help file (also a !SetUp file if this is to be used by Make - see later 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.

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.

The *FrontEnd_SetUp command is similar, except it calls its new instantiation `FrontEnd%M`*`toolname`*, 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 entitled *Template files* on page 188.

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.

## 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 DDE non-interactive tools, like !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 FPEmulator 0 RMLoad System:modules.fpemulator
*RMEnsure FPEmulator 2.80 Error You need FPEmulator 2.80 to run !Diff
*RMEnsure SharedCLibrary 0 System:modules.clib
*RMEnsure SharedCLibrary 3.70 Error You need Clib 3.70 to run !Diff
*RMEnsure FrontEnd 0 System:modules.frontend
*RMEnsure Frontend 1.07 Error You need version 1.07 of the FrontEnd
module torun !Diff
*RMEnsure TaskWindow 0 System:modules.task
*RMEnsure TaskWindow 0.29 Error You need version 0.29 of the taskwindow
module to run !Diff
*RMEnsure DDEUtils 0 System:modules.ddeutils
*RMEnsure DDEUtils 1.30 Error You need version 1.30 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:

- `*Wimpslot -min 16K -max 16K`
- `*Set` *`toolname`*`$Dir <Obey$Dir>`
- `*RMEnsure FrontEnd`
- `*FrontEnd_Setup -app %0 -desc %1 -task %2 -handle %3`

Again, examples of a !SetUp file can be found in the set of non-interactive DDE 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 (eg !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 | Should be as standard Acorn applications information boxes.<br>Icon #1 must be indirected text, with a buffer size large enough to accept the application name.<br>Icon #4 must be indirected text, with a buffer size large enough to accept the version string. |
| SetUp | Mandatory | 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, plus a space and the string `(Completed)`. |

| | | |
|---|---|---|
| | | Icon #0 must be indirected text (buffer size 12 bytes), and have a button type of `Menu icon` and an ESG of 1, and should contain the text `Run`. It is used to invoke the command line tool with the chosen options.<br>Icon #1 must be text, and have a button type of `Menu icon` and an ESG of 1, and should contain the text `Cancel`. 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.<br>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 entitled *Writing an application description* on page 190. |
| CmdLine | Mandatory | 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 `Command Line:`.<br>Icon #0 must be indirected text with buffer size 12 bytes, with button type `Menu icon` and ESG of 1, and containing the text `Run`. It is used to invoke the command line tool with the shown command line.<br>Icon #1 must be indirected text with buffer size typically at least 256 bytes, and with a button type of `Writeable`. |
| Help | Optional | 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 `Auto-redraw` flag set. |
| query | Mandatory | Used to ask the user if they really want to kill off a task which is running.<br>Icon #0 must be text, button type `Menu icon`, an ESG of 1, and is used to reply `Yes`.<br>Icon #1 must be indirected text, buffer size 256 bytes. |

| | | |
|---|---|---|
| | | Icon #2 must be text, button type `Menu icon`, an ESG of 1, and is used to reply `No`. |
| Output | Optional | Used to display in a scrolling window, the textual output of the command line tool. The window's `Auto-redraw` flag must not be set. |
| 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 `Menu icon`, ESG of 1, containing the text `Abort`. It is used to abort the task. Icon #3 must be indirected text, with a buffer size large enough to hold strings `Pause` and `Continue`, button type `Menu icon`, ESG of 1. It is used to pause and continue the task. |
| xfer_send | Mandatory if user is able to save anything | Used both as a save dialogue box for the textual output of a tool, and to save the result file generated by running the tool. Icon #0 must be text, with button type `Menu icon`, ESG of 1, containing the text `OK`. Icon #2 must be indirected text, with a buffer size of 256, and button type `writeable`. Icon #3 must be indirected text. |
| save | Optional | As for xfer_send, but is used to save the result file generated by running the tool. It should also have a `close` 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 grammar is given in *Appendix B - FrontEnd protocols* on page 211. 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 B - 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.

### 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 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 Wimpslot of 640k, under the Task Window module. If you want this value to be different, then use the Wimpslot command in the description.

Since the limit on RISC OS command lines is 256 characters, 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 DDE 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.

### 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 (designed using an application such as FormEd).

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 contributes to the command line if it is selected. The mapping onto the command line is given after the keyword `maps_to`; this may begin with an optional string literal (eg `-f`), optionally followed by keywords `string` or `number`. These latter keywords are used for writable indirected text icons, and refer to their contents. 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.

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 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 Menu icon.

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 wildcard 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.

## 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 used 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 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`.

### DESELECTIONS section

The DESLECTIONS section allows you to state which Options when enabled should disable other options. This can be done for both icons in the main options dialogue box and for entries in its attached menu. For example:

```
icn 3 deselects icn 4, icn 5, menu 3;
```

means that if icon 3 is selected, then icons 4 and 5 and menu entry 3 will be deselected.

### EXCLUSIONS section

The EXCLUSIONS section is similar to the DESELECTIONS 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.

### 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.

### 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 DDE 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 256 byte command line 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* C - DDE*Utils*.

## 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* D - *SrcEdit file formats*.

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* D - *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* B - *FrontEnd protocols* on page 211) containing the desired command line.

In order to achieve this, a tool which is being added to the DDE must append six lines to the file `<Make$Dir>.choices.tools` of the form:

| | |
|---|---|
| *`tool_name`* | |
| *`extension`* | (the string used to identify a source written in this language, eg c for the C language) |
| *`make_defaults`* | (the default options for this tool when in a makefile) |
| *`conversion_rule`* | (ie how to convert source files to object files) |
| *`description_file`* | (full pathname of file containing application description) |
| *`setup_file`* | (full pathname of file containing SetUp actions for when tool is invoked via Make). |

# Appendices

# 21 Appendix A - 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.

## 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 earlier chapter entitled *Make*. 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 wimp slot under the desktop, and a program (such as the C compiler) to be run by amu needs more memory than is left, you can instruct amu not to execute commands directly, but to write them to 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 255 characters. The DDE 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.

## 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 the later section entitled *Makefiles constructed by Make*). 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 definition lines are lines of the form:

*`macro-name = some text to the end of the logical line`*

For example:

```
CC = ncc
CFLAGS = -fah -c -I$.clib
LD = link
LIB = $.CLib.o.clxlib $.CLib.o.Stubs
CLX = $.301.clx
```

The = can be surrounded with white space, or not, to taste. Thereafter, wherever `${`*`name`*`}` or `$(`*`name`*`)` is encountered, if *`name`* is the name of a macro then the whole of `${`*`name`*`}` is replaced by its definition. A reference to an undefined macro simply vanishes. An example which uses the above macro definitions, and which is taken from the makefile for amu itself, is:

```
amu:       amu.o $(CLX).o.clxlib
           $(LD) -o amu ${LFLAGS} o.amu ${LIB}
```

which expands to

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

Note that `${LFLAGS}` expands to nothing.

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 makefiles constructed by Make.

## Advanced features

### File naming

To help you move MS-DOS and UNIX makefiles to RISC OS, or to develop makefiles under RISC OS for export to MS-DOS or UNIX, both amu and the C compiler accept three styles of file naming:

| | | |
|---|---|---|
| RISC OS native: | `$.301.cfe.c.pp` | `^.include.h.defs` |
| UNIX-like: | `/301/cfe/pp.c` | `../include/defs.h` |
| MS-DOS-like: | `\301\cfe\pp.c` | `..\include\defs.h` |

(All three of these examples refer to the same two RISC OS files.)

The linker offers more limited support; in essence, it recognises `thing.o` and `o.thing` as referring to the same RISC OS file (`o.thing`). In practice, object files almost always live locally (that's the only place the RISC OS and UNIX C compilers will put one) so this support is fairly complete.

Amu will even accept a mixture of naming styles, though this practice should be discouraged.

The mapping between different naming styles cannot be complete (consider the UNIX analogue of `adfs::0.$.Library` or `net#1.251:src.amu`). However, it is usually sufficient to take much of the hard work out of moving reasonably portable makefiles.

### VPATH

Usually, amu looks for files relative to the 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

o.cg: c.cg # ^.mip.c.cg, via VPATH
```

## Rule patterns, .SUFFIXES, $@, $*, $< and $?

All the examples given so far have been written out longhand, with explicit rules for making targets. In fact, amu can make inferences if you supply the appropriate rule patterns. These are specified using special target names consisting of the concatenation of two suffixes from the pseudo-dependency `.SUFFIXES`. This sounds very complicated, but is actually quite simple. For example:

```
.SUFFIXES:      .o .c
amu:            o.amu ...
.c.o:;          $(CC) $(CFLAGS) -o $@ c.$*
```

(Note the order here: `.c.o` makes a `.o`-like thing from a `.c`-like thing).

The rule pattern `.c.o` describes how to make `.o`-like things from `.c`-like things. If, as in the above fragment, there is no explicit entry describing how to make a `.o`-like thing (`o.amu`, in the above example) amu will apply the first rule it has for making `.o`-like things. Here, order is determined by order in the `.SUFFIXES` pseudo- dependency. For example, suppose `.SUFFIXES` were defined as `.o .c .f` and that there were two rules, `.c.o:...` and `.f.o:...`. Then amu would choose the `.c.o` rule because `.c` precedes `.f` in the `.SUFFIXES` dependency. In applying the `.c.o` rule, amu infers a dependence on the corresponding `.c`-like thing - here `c.amu`. So, in effect, it infers:

```
o.amu:          c.amu
                $(CC) $(CFLAGS) -o o.amu c.amu
```

Note that, in the commands, `$@` is replaced by the name of the target and `$*` by the name of the target with the 'extension' deleted from it. In a similar fashion, `$<` refers to the list of inferred prerequisites. So the above example could be rewritten using the rule:

```
.c.o:;          $(CC) $(CFLAGS) -o $@ $<
```

However, if a `VPATH` were being used, this second form is obligatory. Consider, for example, the fragment:

```
VPATH=^.arm ^.mip ^.cfe
cc:             .... o.pp ....
.c.o:;          $(CC) $(CFLAGS) -o $@ $<
```

There is no explicit rule for making `o.pp`, so amu will apply the rule pattern `.c.o:?`. This might expand to:

```
o.pp:           ^.cfe.c.pp
                $(CC) $(CFLAGS) -o o.pp ^.cfe.c.pp
```

which has a much more useful effect than:

```
                $(CC) $(CFLAGS) -o o.pp c.pp
```

Finally, `$?` can be used in any command to stand for the list of prerequisites with respect to which the target is out of date (which may be only some of the prerequisites).

#### Use of `::`

If you use `::` to separate targets from prerequisites, rather than `:`, the 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 in makefiles constructed by Make.

### Prefix$Dir

The DDEUtils module provides an environment variable `Prefix$Dir` set to the work directory. This is provided to allow you to execute binaries placed in the work directory, since `Run$Path` cannot otherwise specify the work directory.

## Makefiles constructed by Make

A makefile constructed by Make, ie 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`.

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: `toolFLAGS = ....` |
| `# Final targets` | This section contains the rules for making the final targets of the project. For example: `!RunImage: link $(linkflags) -o !RunImage -via objects)` |
| `# 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 `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 (eg `cc -depend`). |

## 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
```

# 22 Appendix B - 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_ENTRYLIS::=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_EXLUSIONS::=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` |

# 23 Appendix C - 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 (eg between AMU and Link).
- **Throwback**. Throwback allows a language processor (eg 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 (&42580)`

| | |
|---|---|
| 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. |

## 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 (&42581)`

| | |
|---|---|
| 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 (&42582)`

| | |
|---|---|
| Entry: | R0 = Pointer to zero terminated command line tail |
| Exit: | All registers preserved |
| Error: | Possible errors are<br>`CLI buffer not set`<br>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 (ie the command line without the name of the program to be run). |

`DDEUtils_GetCLSize (&42583)`

| | |
|---|---|
| Entry: | don't care |
| 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 (&42584)`

| | |
|---|---|
| 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. |

## Throwback SWIs

`DDEUtils_ThrowbackRegister (&42585)`

| | |
|---|---|
| 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 (&42586)`

| | |
|---|---|
| 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 (&42587)`

| | |
|---|---|
| Entry: | don't care |
| 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.

`Throwback_Send (&42588)`

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 when processing a file was found. Then it should be called once for each error found, with the reason

```
Throwback_ReasonErrorDetails
```

DDEUtils_ThrowbackEnd (&42589)

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 (&42580) |

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>
                               ErrorFoundIn {ErrorFoundIn}

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 (&42581)` |
| +20 | `Nul-terminated filename` |

| Byte Offset | Contents |
|---|---|
| +16 | `DDEUtils_ErrorsIn (&42582)` |
| +20 | `Nul-terminated filename` |

| Byte Offset | Contents |
|---|---|
| +16 | `DDEUtils_ErrorDetails (&42583)` |
| +20 | `Line number` |
| +28 | `Severity`<br>`= 0 for warning`<br>`= 1 for error`<br>`= 2 for serious error` |
| +32 | `Nul-terminated description` |

| Byte Offset | Contents |
|---|---|
| +16 | `DDEUtils_ThrowbackEnd (&42584)` |

| Byte Offset | Contents |
|---|---|
| +16 | `DDEUtils_InfoforFile (&42585)` |
| +20 | `Nul-terminated filename` |

| Byte Offset | Contents |
|---|---|
| +16 | `DDEUtils_InfoDetails (&42586)` |
| +20 | `Line number` |
| +28 | `must be 0` |
| +32 | `Nul-terminated 'informational' message` |

# 24 Appendix D - SrcEdit file formats

## Language File Format

*language_name*

| | |
|---|---|
| *searchpath* | this line can be blank. |
| *helppath* | this line can be blank. |
| *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 (eg `C:`), or be terminated by a dot. |
| *helppath* | is the full pathname of language help file. |

## 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.

# 25 Appendix E - Code file formats

This appendix defines three file formats used by DDE tools to store processed code and the format of debugging data used by DDT:

- AOF – Arm Object Format
- ALF – Acorn Library Format
- AIF – RISC OS Application Image Format
- ASD – ARM Symbolic Debugging Format.

DDE 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 least significant byte has the lowest address (DEC/Intel *byte sex*, sometimes called *little endian*). The address of a half word (ie 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 least significant byte has the lowest address (DEC/Intel byte sex, sometimes called little endian). The address of a word (ie 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.

For emphasis: a word consists of 32 bits, 4-byte aligned; within a word, the least significant byte has the lowest address. This is DEC/Intel, or little endian, byte sex, **not** IBM/Motorola byte sex.

## Undefined Fields

Fields not explicitly defined by this appendix are implicitly reserved to Acorn. It is required that all such fields be zeroed. Acorn may ascribe meaning to such fields at any time, but will usually do so in a manner which gives no new meaning to zeroes.

## Overall structure of AOF and ALF files

An object or library file contains a number of separate but related pieces of data. In order to simplify access to these data, and to provide for a degree of extensibility, the object and library file formats are themselves layered on another format called **Chunk File Format**, which provides a simple and efficient means of accessing and updating distinct chunks of data within a single file. The object file format defines five chunks:

- header
- areas
- identification
- symbol table
- string table.

The library file format defines four chunks:

- directory
- time-stamp
- version
- data.

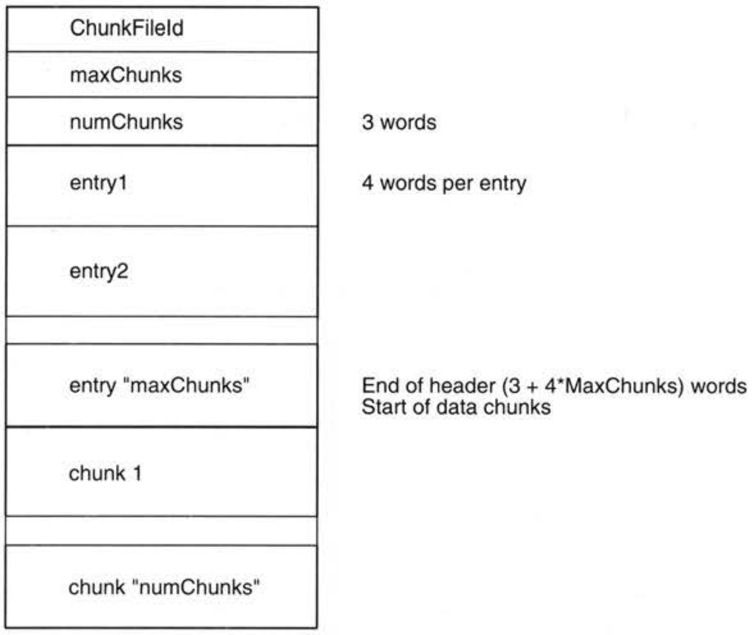
There may be many data chunks in a library.

The minimum size of a piece of data in both formats is four bytes or one word. Each word is stored in a file in little-endian format; that is the least significant byte of the word is stored first.

## 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 C3CBC6C5 hex. 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:

| | |
|---|---|
| `chunkId` | a two word field identifying what data the chunk file contains |
| `Offset` | a one word field defining the byte offset within the file of the chunk (which must be divisible by four); an entry of zero indicates that the corresponding chunk is unused |
| `size` | 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 (in the first word) contain a universally unique name allocated by a central authority (Acorn). The remaining four characters (in the second word) can be used to identify component chunks within this universal domain. In each part, the first character of the name is stored first in the file, and so on.

For AOF files, the first part of each chunk's name is `OBJ_`; the second components are defined later. For ALF files, the first part is `LIB_`.

# AOF

ARM object format files are output by language processors such as CC and ObjAsm.

## 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 | OBJ_HEAD |
| Areas | OBJ_AREA |
| Identification | OBJ_IDFN |
| Symbol Table | OBJ_SYMT |
| String Table | OBJ_STRT |

Only the `header` and `areas` chunks must be present, but a typical object file will contain all five of the above chunks.

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.

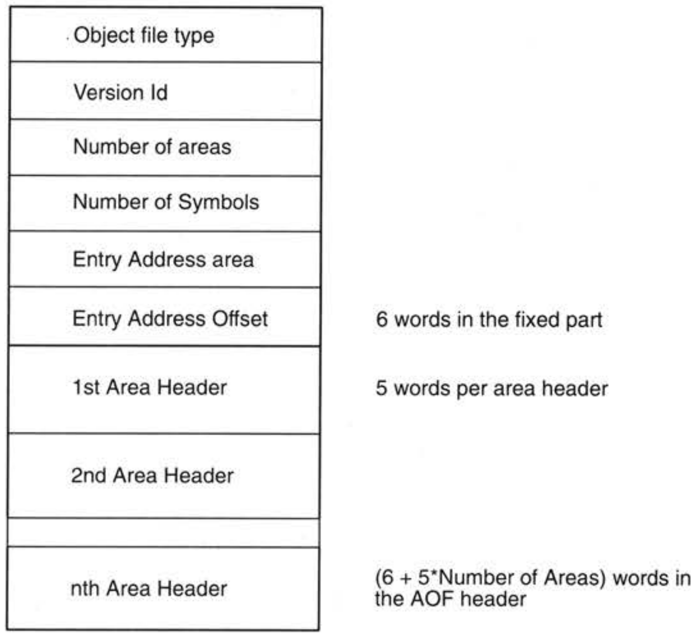
A language translator or other system utility may add additional chunks to an object file, for example a language-specific symbol table or language-specific debugging data, so it is conventional to allow space in the chunk header for additional chunks; space for eight chunks is conventional when the 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 has the following format:



### Object file type

C5E2D080 (hex) marks an object file as being in relocatable object format

### Version ID

This word encodes the version of AOF to which the object file complies: AOF 1.xx is denoted by 150 decimal; AOF 2.xx by 200 decimal.

### 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 for any program which is linked to include this file. If so, the entry address is specified as an `<area-index, offset>` pair, where `area-index` is in the range 1 to `Number of Areas`, specifying the nth area declared in the area declarations part of the header. The entry address is defined to be the base address of this area plus `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) |
| zeros | AT | AL | |
| Area size | | | |
| Number of relocations | | | |
| Unused - must be zero | | | 5 words in total |

### Area name

Each name in an object file is encoded as an offset into the string table, which stored in the OBJ_STRT chunk. This allows the variable-length characteristics of names to be factored out from primary data formats. Each area within an object file must be given a name which is unique amongst all the areas in that object file.

### AL

This byte must be set to 2; all other values are reserved to Acorn.

### AT (Area attributes)

Each area has a set of attributes encoded in the AT byte. The least-significant bit of AT is numbered 0.

Link 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.

When ordered by attributes, Read-Only areas precede Read-Write areas which precede Debug areas; within Read-Only and Read-Write Areas, Code precedes Data which precedes Zero-Initialised data. Zero-Initialised data may not have the Read-Only attribute.

### Bit 0

This bit must be set to 0.

### Bit 1

If this bit is set, the area contains code, otherwise it contains data.

### Bit 2

Bit 2 specifies that the area is a common block definition.

### Bit 3

Bit 3 defines the area to be a (reference to a) common block and precludes the area having initialising data (see Bit 4, below). In effect, the setting of Bit 3 implies the setting of Bit 4.

Common areas with the same name are overlaid on each other by Link. The `Size` field of a common definition 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. In a link step there may be at most one area of the given name with bit 2 set. If none of these have bit 2 set, the actual size of the common area will be size of the largest common block reference (see also the section entitled *Linker defined symbols* on page 239).

### Bit 4

This bit specifies that the area has no initialising data in this object file and that the area contents are missing from the OBJ_AREA chunk. This bit is typically used to denote large uninitialised data areas. When an uninitialised area is included in an image, Link 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 the section on Bit 5, below).

**Note**: Whether or not a zero-initialised area is re-zeroed if the image is re-entered is a property of Link and the relevant image format. The definition of AOF neither requires nor precludes re-zeroing.

### Bit 5

This bit specifies that the area is read-only. Link 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 4.

### Bit 6

This bit must be set to 0.

### Bit 7

This bit specifies that the area contains symbolic debugging tables. Link groups these areas together so they can be accessed as a single contiguous chunk at run-time. It is usual for debugging tables to be read-only and, therefore, to have bit 5 set too. If bit 7 is set, bit 1 is ignored.

## Area size

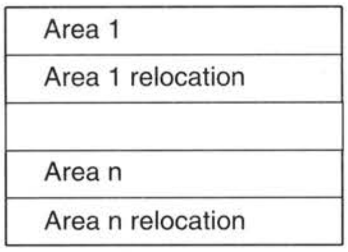
This field specifies the size of the area in bytes, which must be a multiple of 4. Unless the `Not Initialised` bit (bit 4) is set in the area attributes, there must be this number of bytes for this area in the OBJ_AREA chunk.

## Number of relocations

This specifies the number of relocation records which apply to this area.

## Format of the areas chunk

The areas chunk (OBJ_AREA) contains the actual areas (code, data, zero- initialised data, debugging data, etc.) plus any associated relocation information. Its chunkId is OBJ_AREA. Both an area's contents and its relocation data must be word-aligned. Graphically, an area's layout is:



An area is simply a sequence of byte values, the order following that of the addressing rules of the ARM, that is the least significant byte of a word is first. An area is followed by its associated relocation table (if any). An area is either completely initialised by the values from the file or not initialised at all (ie it is initialised to zero in any loaded program image, as specified by bit 4 of the area attributes).

## Relocation directives

If no relocation is specified, the value of a byte/halfword/word in the preceding area is exactly the value that will appear in the final image.

Bytes and halfwords may only be relocated by constant values of suitably small size. They may not be relocated by an area's base address.

A field may be subject to more than one relocation.

There are 2 types of relocation directive, termed here type-1 and type-2. Type-2 relocation directives occur only in AOF versions 1.50 and later.

Relocation can take two basic forms: *Additive* and PC*Relative*.

Additive relocation specifies the modification of a byte/halfword/word, typically containing a data value (ie constant or address).

PCRelative relocation always specifies the modification of a branch (or branch with link) instruction and involves the generation of a program- counter-relative, signed, 24-bit word-displacement.

Additive relocation directives and type-2 PC-relative relocation directives have two variants: `Internal` and `Symbol`.

Additive internal relocation involves adding the allocated base address of an area to the field to be relocated. With Type-1 internal relocation directives, the value by which a location is relocated is always the base of the area with which the relocation directive is associated (the Symbol IDentification field (SID) is ignored). In a type-2 relocation directive, the SID field specifies the index of the area relative to which relocation is to be performed. These relocation directives are analogous to the TEXT-, DATA- and BSS-relative relocation directives found in the a.out object format.

Symbol relocation involves adding the value of the symbol quoted.

A type-1 PCRelative relocation directive always references a symbol. The relocation offset added to any pre-existing in the instruction is the offset of the target symbol from the PC current at the instruction making the PCRelative reference. Link takes into account the fact that the PC is eight bytes beyond that instruction.

In a type-2 PC-relative relocation directive (only in AOF version 1.50 and later) the offset bits of the instruction are initialised to the offset from the base of the area of the PC value current at the instruction making the reference – thus the language translator, not Link, compensates for the difference between the address of the instruction and the PC value current at it. This variant is introduced in direct support of compilers that must also generate Unix's `a.out` format.

For a type-2 PC-relative symbol-type relocation directive, the offset added into the instruction making the PC-relative reference is the offset of the target symbol from the base of the area containing the instruction. For a type-2, PC-relative, internal relocation directive, the offset added into the instruction is the offset of the base of the area identified by the SID field from the base of the area containing the instruction.

Link itself may generate type-2, PC-relative, internal relocation directives during the process of partially linking a set of object modules.

## Format of Type 1 relocation directives

Diagrammatically:

| | | | | |
|---|---|---|---|---|
| Offset | | | | |
| O | A | R | FT | SID |

### Offset

Offset is the byte offset in the preceding area of the field to be relocated.

### SID

If a symbol is involved in the relocation, this 16-bit field specifies the index within the symbol table (see below) of the symbol in question.

### FT (Field Type)

This 2-bit field (bits 16 – 17) specifies the size of the field to be relocated:

| | |
|---|---|
| 00 | byte |
| 01 | halfword |
| 10 | word |
| 11 | *illegal value* |

### R (relocation type)

This field (bit 18) has the following interpretation:

| | |
|---|---|
| 0 | Additive relocation |
| 1 | PC-Relative relocation |

#### A (Additive type)

In a type-1 relocation directive, this 1-bit field (bit 19) is only interpreted if bit 18 is a zero.

A=0 specifies Internal relocation, meaning that the base address of the area (with which this relocation directive is associated) is added into the field to be relocated. A=1 specifies Symbol relocation, meaning that the value of the given symbol is added to the field being relocated.

#### Bits 20 - 31

Bits 20-31 are reserved by Acorn and should be written as zeroes.

### Format of Type 2 relocation directives

These are available from AOF 1.50 onwards.

| | | | | |
|---|---|---|---|---|
| Offset | | | | |
| 1000 | A | R | FT | 24-bit SID |

The interpretation of Offset, FT and SID is exactly the same as for type-1 relocation directives except that SID is increased from 16 to 24 bits and has a different meaning – described below – if A=0).

The second word of a type-2 relocation directive contains 1 in its most significant bit; bits 28..30 must be written as 0, as shown.

The different interpretation of the R bit in type-2 directives has already been described in the section entitled *Relocation directives* on page 234.

If A=0 (internal relocation type) then SID is the index of the area, in the OBJ_AREA chunk, relative to which the value at Offset in the current area is to be relocated. Areas are indexed from 0.

### Format of the symbol table chunk

The `Number of Symbols` field in the header defines how many entries there are in the symbol table. Each symbol table entry has the following format:

| | |
|---|---|
| Name | |
| | AT |
| 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.

### AT

This is a 7 bit field specifying the attributes of a symbol as follows:

#### Bits 1 and 0

(10 means bit 1 set, bit 0 unset).

| | |
|---|---|
| **01** | The symbol is defined in this object file and has scope limited to this object file (when resolving symbol references, Link will only match this symbol to references from other areas within the same object file). |
| **10** | The symbol is a reference to a symbol defined in another area or another object file. If no defining instance of the symbol is found then Link 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 value the base address of the common area. |
| **11** | The symbol is defined in this object file and has global scope (ie when attempting to resolve unresolved references, Link will match this symbol to references from other object files). |
| **00** | Reserved by Acorn. |

#### Bit 2

This attribute is only meaningful if the symbol is a defining occurrence (bit 0 set). It specifies that the symbol has an absolute value, for example, a constant. Otherwise its value is relative to the base address of the area defined by the `Area Name` field of the symbol table entry.

#### Bit 3

This bit is only meaningful if bit 0 is unset (that is, the symbol is an external reference). Bit 3 denotes that the reference is case-insensitive. When attempting to resolve such an external reference, Link will ignore character case when performing the match.

#### Bit 4

This bit is only meaningful if the symbol is an external reference (bits 1,0 = 10). It denotes that the reference is **weak**, that is that it is acceptable for the reference to remain unsatisfied and for any fields relocated via it to remain unrelocated.

**Note**: A weak reference still causes a library module satisfying that reference to be auto-loaded.

### Bit 5

This bit is only meaningful if the symbol is a defining, external occurrence (ie if bits 1,0 = 11). It denotes that the definition is **strong** and, in turn, this is only meaningful if there is a non-strong, external definition of the same symbol in another object file. In this scenario, all references to the symbol from outside of the file containing the strong definition are resolved to the strong definition. Within the file containing the strong definition, references to the symbol resolve to the non-strong definition.

This attribute allows a kind of link-time indirection to be enforced. Usually, strong definitions will be absolute and will be used to implement an operating system's entry vector which must have the **forever binary** property.

### Bit 6

This bit is only meaningful if bits 1,0 = 10. Bit 6 denotes that the symbol is a common symbol – in effect, 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 below). Link treats common symbols much as it treats areas having the common reference bit set – 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 merge and 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 pseudo-area.

### Value

This field is only meaningful if the symbol is a defining occurrence (ie bit 0 of AT set) or a common symbol (ie bit 6 of AT set). If the symbol is absolute (bit 2 of AT set), this field contains the value of the symbol. Otherwise, it is interpreted as an offset from the base address of the area defined by `Area Name`, which must be an area defined in this object file.

### Area name

This field is only meaningful if the symbol is not absolute (ie if bit 2 of AT is unset) and the symbol is a defining occurrence (ie bit 0 of AT is set). In this case it gives the index into the string table of the character string name of the (logical) area relative to which the symbol is defined.

### String table chunk (OBJ_STRT)

The string table chunk contains all the print names referred to within the areas and symbol table chunks. The separation is made to factor out the variable length characteristic of print names. A print name is stored in the string table as a sequence of ISO8859 non-control characters terminated by a NUL (0) byte and is identified by an offset from the table's beginning. The first 4 bytes of the string table contain its length (including the length word – so no valid offset into the table is less than 4 and no table has length less than 4). The length stored at the start of the string table itself is identically the length stored in the OBJ_STRT chunk header.

### Identification chunk (OBJ_IDFN)

This chunk should contain a printable character string (characters in the range [32..126]), terminated by a NUL (0) byte, giving information about the name and version of the language translator which generated the object file.

## Linker defined symbols

Though not part of the definition of AOF, the definitions of symbols which the AOF linker defines during the generation of an image file are collected here. These may be referenced from AOF object files, but must not be redefined.

### Linker pre-defined symbols

The pre-defined symbols occur in Base/Limit pairs. A Base value gives the address of the first byte in a region and the corresponding Limit value gives the address of the first byte beyond the end of the region. All pre-defined symbols begin `Image$$` and the space of all such names is reserved by Acorn.

None of these symbols may be redefined. The pre-defined symbols are:

| | |
|---|---|
| `Image$$RO$$Base`<br>`Image$$RO$$Limit` | Address and limit of the Read-Only section of the image. |
| `Image$$RW$$Base`<br>`Image$$RW$$Limit` | Address and limit of the Read-Write section of the image. |
| `Image$$ZI$$Base`<br>`Image$$ZI$$Limit` | Address and limit of the Zero-initialised data section of the image (created from areas having bit 4 of their area attributes set and from common symbols which match no area name). |

If a section is absent, the Base and Limit values are equal but unpredictable.

| | |
|---|---|
| `Image$$RO$$Base` | includes any image header prepended by Link. |
| `Image$$RW$$Limit` | includes (at the end of the RW section) any zero-initialised data created at run-time. |

The `Image$$xx$${Base,Limit}` values are intended to be used by language run-time systems. Other values which are needed by a debugger or by part of the pre-run-time code associated with a particular image format are deposited into the relevant image header by Link.

### Common area symbols

For each common area, Link defines a global symbol having the same name as the area, except where this would clash with the name of an existing global symbol definition (thus a symbol reference may match a common area).

## Obsolescent and obsolete features

The following subsections describe features that were part of revision 1.xx of AOF and/or that were supported by the 59x releases of the AOF linker, which are no longer supported. In each case, a brief rationale for the change is given.

### Object file type

AOF used to define three image types as well as a relocatable object file type. Image types 2 and 3 were never used under Arthur/RISC OS and are now obsolete. Image type 1 is used only by the obsolete Dbug (DDT has Dbug's functionality and uses Application Image Format).

| | | |
|---|---|---|
| AOF Image type 1 | C5E2D081 hex | (obsolescent) |
| AOF Image type 2 | C5E2D083 hex | (obsolete) |
| AOF Image type 3 | C5E2D087 hex | (obsolete) |

### AL (Area alignment)

AOF used to allow the alignment of an area to be any specified power of 2 between 2 and 16. By convention, relocatable object code areas always used minimal alignment (AL=2) and only the obsolete image formats, types 2 and 3, specified values other than 2. From now on, all values other than 2 are reserved by Acorn.

### AT (Area attributes)

Two attributes have been withdrawn: the Absolute attribute (bit 0 of AT) and the Position Independent attribute (bit 6 of AT).

The Absolute attribute was not supported by the RISC OS linker and therefore had no utility. Link in any case allows the effect of the Absolute attribute to be simulated.

The Position Independent bit used to specify that a code area was position independent, meaning that its base address could change at run-time without any change being required to its contents. Such an area could only contain internal, PC-relative relocations and must make all external references through registers. Thus only code and pure data (containing no address values) could be position-independent.

Few language processors generated the PI bit which was only significant to the generation of the obsolete image types 2 and 3 (in which it affected AREA placement). Accordingly, its definition has been withdrawn.

## Fragmented areas

The concept of fragmented areas was introduced in release 0.04 of AOF, tentatively in support of Fortran compilers. To the best of our knowledge, fragmented areas were never used. (Two warnings against use were given with the original definition on the grounds of: structural incompatibility with Unix's `a.out` format; and likely inefficient handling by Link. And use was hedged around with curious restrictions). Accordingly, the definition of fragmented areas is withdrawn.

# ALF

ALF is the format of linkable libraries (such as the C RISC OS library RISC_OSLib).

## Library file format types

There are two library file formats described here, termed *new-style* and *old-style*. Link can read both formats, though no tool will actually generate an old-style library.

Currently, only the Acorn/Topexpress Fortran-77 compiler generates old-style libraries (which it does instead of generating AOF object files). Link handles these libraries specially, including every member in the output image unless explicitly instructed otherwise.

Old-style libraries are obsolescent and should no longer be generated.

## Library file chunks

Each piece of a library file is stored in a separate, identifiable, chunk, named as follows:

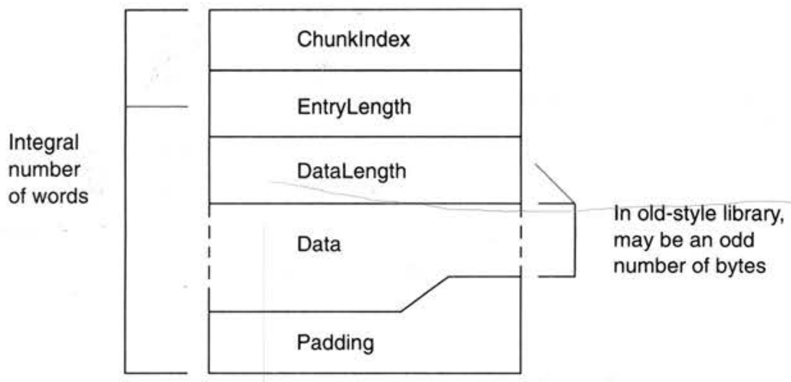
| **Chunk** | **Chunk Name** | |
|---|---|---|
| Directory | `LIB_DIRY` | |
| Time-stamp | `LIB_TIME` | |
| Version | `LIB_VSRN` | – new-style libraries only |
| Data | `LIB_DATA` | |
| Symbol table | `OFL_SYMT` | – object code libraries only |
| Time-stamp | `OFL_TIME` | – object code libraries only |

There may be many LIB_DATA chunks in a library, one for each library member.

## LIB_DIRY

The LIB_DIRY chunk contains a directory of all 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

The ChunkIndex is a 0 origin index within the chunk file header of the corresponding LIB_DATA chunk. The LIB_DATA chunk entry gives the offset and size of the library module in the library file. A ChunkIndex of 0 means the directory entry is not in use.

## EntryLength

The number of bytes in this LIB_DIRY entry, always a multiple of 4.

## DataLength

The number of bytes used in the Data section of this LIB_DIRY entry. This need not be a multiple of 4, though it always is in new-style libraries.

## Data

The data section consists of a 0 terminated string followed by any other information relevant to the library module. Strings should contain only ISO-8859 non-control characters (ie codes [0-31], 127 and 128+[0-31] are excluded). The string is the name used by the library management tools to identify this library module. Typically this is the name of the file from which the library member was created.

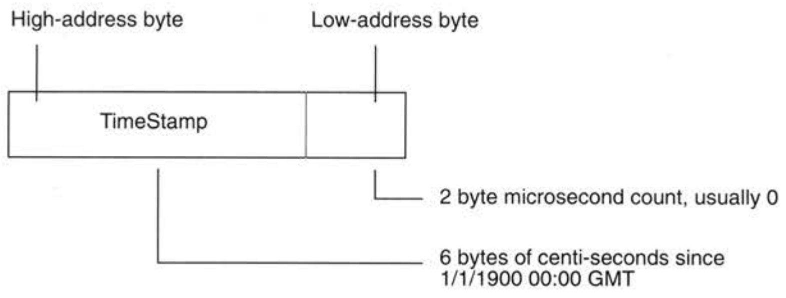
In new-style libraries, an 8-byte, word-aligned time-stamp follows the member name. The format of this time-stamp is described in the section entitled LIB_TIME on page 244. Its value is (an encoded version of) the time-stamp (ie the last modified time) of the file from which the library member was created.

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.

## LIB_TIME

The LIB_TIME chunk contains a 64 bit time-stamp recording when the library was last modified, in the following format:


High-address byte
Low-address byte
TimeStamp
2 byte microsecond count, usually 0
6 bytes of centi-seconds since 1/1/1900 00:00 GMT

## LIB_VSRN

In new-style libraries, this chunk contains a 4-byte version number. The current version number is 1. Old-style libraries do not contain this chunk.

## LIB_DATA

A LIB_DATA chunk contains one of the library members indexed by the LIB_DIRY chunk. No 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 AOF. All libraries you are likely to use with the DDE are object code libraries.

Additional information is stored in two extra chunks, OFL_SYMT and OFL_TIME.

OFL_SYMT 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). OFL_SYMT entries do not contain time-stamps.

The OFL_TIME chunk records when the OFL_SYMT chunk was last modified and has the same format as the LIB_TIME chunk (see above).

# AIF

AIF is the format of executable program files produced by linking AOF files. Example AIF files are !RunImage files of applications coded in C or assembler.

## Properties of AIF

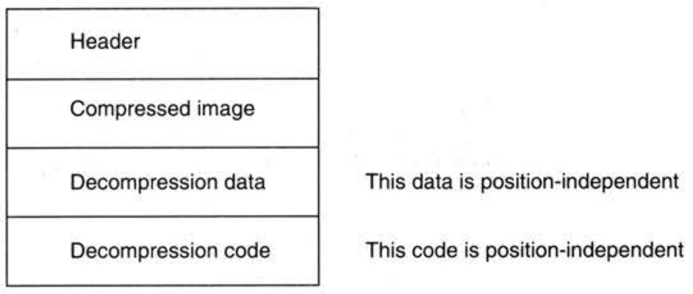
- An AIF image is loaded into memory at its load address and entered at its first word (compatible with old-style Arthur/Brazil ADFS images).
- An AIF image may be compressed and can be self-decompressing (to support faster loading from floppy discs, and better use of floppy-disc space).
- If created with suitable linker options, an AIF image may relocate itself at load time. Self-relocation is supported in two, distinct senses:-
    - One-time Position-Independence: A relocatable image can be loaded at any address (not just its load address) and will execute there (compatible with version 0.03 of AIF).
    - Specified Working Space Relocation: A suitably created relocatable image will copy itself from where it is loaded to the high address end of applications memory, leaving space above the copied image as noted in the AIF header (see below).

  In addition, similar relocation code and similar linker options support many-time position independence of RISC OS Relocatable Modules.
- AIF images support being debugged by the Desktop Debugging Tool (DDT), for C, assembler and other languages. Version 0.04 of AIF (and later) supports debugging at the symbolic assembler level (hitherto done by Dbug). Low-level and source-level debugging support are orthogonal (capabilities of debuggers notwithstanding, both, either, or neither kind of debugging support may be present in an AIF image).

  Debugging tables have the property that all references from them to code and data (if any) are in the form of relocatable addresses. After loading an image at its load address these values are effectively absolute. All 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 can be copied by the debugger.

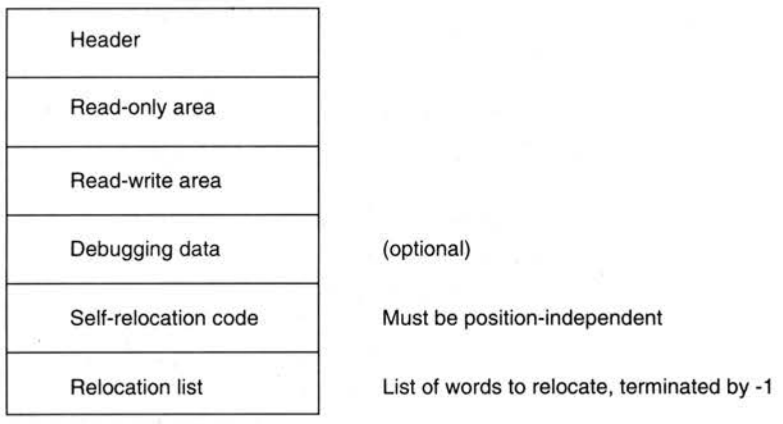
## Layout of an AIF image

The layout of an 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.

Once an image has been decompressed – or if it is uncompressed in the first place – it has the following layout:



Debugging data are absent unless the image has been linked appropriately and, in the case of source-level debugging, unless the constituent components of the image have been compiled appropriately.

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 the debugging data (if present) somewhere safe, otherwise they 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 below).

## AIF header layout

| | |
|---|---|
| BL DecompressedCode | BLNV 0 if the image is not compressed |
| BL SelfRelocCode | BLNV 0 if the image is not self-relocating |
| BL ZeroInitCode | BLNV 0 if the image has none |
| BL ImageEntryPoint | BL to make header addressable via R14 |
| SWI OS_Exit | Just in case silly enough to return |
| Image ReadOnly size | Includes header size and any padding<br>Exact size - a multiple of 4 bytes |
| Image ReadWrite size | Exact size - a multiple of 4 bytes |
| Image Debug size | Exact size - a multiple of 4 bytes |
| Image zero-init size | Exact size - a multiple of 4 bytes |
| Image debug type | 0,1,2 or 3 (see below) |
| Image base | Address of the AIF header - set by link |
| Work space | a self-moving relocatable image<br>Min work space - in bytes - to be reserved by |
| Four reserved words (0) | |
| Zero-init code (16 words) | Header is 32 words long |

BL is used everywhere to make the header addressable via R14 (but beware the PSR bits) in a position-independent manner and to ensure that the header will be position-independent.

It is required that an image be re-enterable at its first instruction. Therefore, after decompression, the decompression code must reset the first word of the header to BLNV 0. Similarly, following self-relocation, the second word of the header must be reset to BLNV 0. This causes no additional problems with the read-only nature of the code segment – both decompression and relocation code must write to it anyway. So, 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 writeable, 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 are reserved by Acorn.

## Zero-initialisation code

The Zero-initialisation code is as follows:

```
    BIC     IP,     LR,     #&FC000003 ; clear status bits -> header + &C
    ADD     IP,     IP,     #8         ; -> Image ReadOnly size
    LDMIA   IP,     {R0,R1,R2,R3}      ; various sizes
    CMPS    R3,     #0
    MOVLES  PC,     LR                 ; nothing to do
    SUB     IP,     IP,     #&14       ; image base
    ADD     IP,     IP,     R0         ; + RO size
    ADD     IP,     IP,     R1         ; + RW size = base of 0-init area
    MOV     R0,     #0
    MOV     R1,     #0
    MOV     R2,     #0
    MOV     R4,     #0
ZeroLoop
    STMIA   IP!,    {R0,R1,R2,R4}
    SUBS    R3,     R3,     #16
    BGT     ZeroLoop
    MOVS    PC,     LR                 ; 16 words in total.
```

### Relationship between header sizes and linker pre-defined symbols

```
AIFHeader.ImageBase                      = Image$$RO$$Base

 AIFHeader.ImageBase +
  AIFHeader.ROSize                       = Image$$RW$$Base

 AIFHeader.ImageBase +
 AIFHeader.ROSize +
  AIFHeader.RWSize                       = Image$$ZI$$Base

 AIFHeader.ImageBase +
  AIFHeader.ROSize +
   AIFHeader.RWSize +
    AIFHeader.ZeroInitSize               = Image$$RW$$Limit
```

## Self relocation

Two kinds of self-relocation are supported by AIF and one by AMF; for completeness, all three are described here.

One-time position independence is supported by relocatable AIF images. Many-time position independence is required for AMF Relocatable Modules. And only AIF images can self-move to a location which leaves a requested amount of workspace.

Why are there three different kinds of self-relocation?

- The rules for constructing RISC OS applications do not forbid acquired position-dependence. Once an application has begun to run, it is not, in general, possible to move it, as it isn't possible to find all the data locations which are being used as position-dependent pointers. So, AIF images can be relocated only once. Afterwards, the relocation table is over-written by the application's zero-initialised data, heap, or stack.
- In contrast, the rules for constructing a RISC OS Relocatable Modules (RM) require that it be prepared to shut itself down, be moved in memory, and start itself up again. Shut-down and start-up are notified to a RM by special service calls to it. Clearly, a RM must be relocatable many times so its relocation table is not overwritten after first use.
- Relocatable Modules are loaded under the control of a Relocatable Module Area (RMA) manager which decides where to load a module initially and where to move each module to whenever the RMA is reorganised. In contrast, an application is loaded at its load address and is then on its own until it exits or faults. An application can only be moved by itself (and then only once, before it begins execution proper).

## Self-relocation code for relocatable modules

In this case there is no AIF header, the code must be executable many times, and it must be symbolically addressable from the Relocatable Module header. The code below must be the last area of the RMF image, following the relocation list. Note that it is best thought of as an additional area.

When the following code is executed, the module image has already been loaded at/moved to its target address. It only remains to relocate location-dependent addresses. The list of offsets to be relocated, terminated by (–1), immediately follows `End`. Note that the address values here (eg `|__RelocCode|`) will appear in the list of places to be relocated, allowing the code to be re-executed.

```
  IMPORT   |Image$$RO$$Base|          ; where the image is linked at...
  EXPORT   |__RelocCode|              ; referenced from the RM header

|__RelocCode|
  LDR      R1,      RelocCode         ; value of __RelocCode (before relocation)
  SUB      IP,      PC,    #12        ; value of __RelocCode now
  SUBS     R1,      IP,    R1         ; relocation offset
  MOVEQS   PC,      LR                ; relocate by 0 so nothing to do
  LDR      IP,      ImageBase         ; image base prior to relocation...
  ADD      IP,      IP,    R1         ; ...where the image really is
  ADR      R2,      End
RelocLoop
  LDR      R0,      [R2], #4
  CMNS     R0,      #1                ; got list terminator?
  MOVLES   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        RelocLoop                  ; and do the next one
RelocCode DCD       |__RelocCode|
ImageBase DCD       |Image$$RO$$Base|
End                                   ; the list of locations to relocate
                                      ; starts here (each is an offset from the
                                      ; base of the module) and is terminated
                                      ; by -1.
```

Note that this code, and the associated list of locations to relocate, is added automatically to a relocatable module image by Link (as a consequence of using Link with the SetUp option Module enabled).

## Self-move and self-relocation code for AIF

This code is added to the end of an AIF image by Link, immediately before the list of relocations (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.

```
RelocCode ROUT
  BIC     IP,      LR,    #&FC000003 ;clear flag bits; -> AIF header + &08
  SUB     IP,      IP,    #8         ; -> header address
  MOV     R0,      #&FB000000        ; BLNV #0
  STR     R0,      [IP,   #4]        ; won't be called again on image re-entry
;does the code need to be moved?
  LDR     R9,      [IP, #&2C]        ; min free space requirement
  CMPS    R9,      #0                ; 0 => no move, just relocate
  BEQ     RelocateOnly
;calculate the amount to move by...
  LDR     R0,      [IP,   #&20]      ; image zero-init size
  ADD     R9,      R9,    R0         ; space to leave = min free + zero init
  SWI     GetEnv                     ; MemLimit -> R1
  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,      %F01              ; intermediate limit for copy-up
;
; copy everything up memory, in descending address order, branching
; to the copied copy loop as soon as it has been copied.
;
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
RelocateOnly
  LDR     R1,      [IP,  #&28]       ; header + &28 = 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
```

```
RelocLoop
  LDR      R0,      R2],  #4       ; offset of word to relocate
  CMNS     R0,      #1             ; terminator?
  MOVEQS   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        RelocLoop               ; and do the next one
  End                              ; The list of offsets of locations to
relocate
                                   ; starts here; terminated by -1.
```

# ASD

Acknowledgement: This design is based on work originally done for Acorn Computers by Topexpress Ltd.

This section describes the format of symbolic debugging data generated by ARM compilers and assemblers running under RISC OS and used by the desktop debugger DDT.

For each separate compilation unit (called a *section*) the compiler produces debugging data an a special AREA of the object code (see the section entitled AOF on page 229 for an explanation of 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 (see the section entitled AIF on page 246 for a description of Application Image Format). Because 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 is 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, not by language processors. These low-level debugging tables appear in an extra section item, as if generated by an independent compilation. 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 data, function, procedure, etc., definition in the source program has a corresponding debug data item and 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 also nested. 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.

## Representation of Data Types

Several of the debugging data items (eg procedure and variable) have a **type** word field to identify their data type. This field contains, in the most significant 3 bytes, a code to identify a base type and, in the least significant byte, 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:

| | | |
|---|---|---|
| void | 0 | (all codes are decimal) |
| 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 of the type such as array bounds and field types. The type code for such types is negative being the negation of the (byte) offset of the special 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 to define the type itself. In that case, the type word will refer to this item.

Enumerated types in C and scalars in Pascal are treated simply as integer sub-ranges of an appropriate size, the name information is not available in the this version of the debugging format. Set types in Pascal are not treated in detail: the only information recorder for them is the total size occupied by the object in bytes.

Fortran character types are supported by a special kind of debugging data item the format of which is yet to be defined.

## 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 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 following codes are defined:-

| | |
|---|---|
| 1 | section |
| 2 | procedure |
| 3 | endproc |
| 4 | variable |
| 5 | type |
| 6 | struct |
| 7 | array |
| 8 | subrange |
| 9 | set |
| 10 | fileinfo |

The meaning of the second and subsequent words of each item is defined below.

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 (the padding value is immaterial, but NUL or ' ' is preferred). The length of a string is in the range [0..255] bytes.

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).

## Section

A section item is the first item of each section of the debugging data.

| | |
|---|---|
| language:8 | one byte code identifying the source language |
| debuglines:1 | 1 => tables contain line numbers |
| debugvars:1 | 1 => tables contain data about local vars |
| spare:14 | |
| debugversion:8 | one byte version number of the debugging data |
| codeaddr | pointer to start of executable code in this section |
| dataaddr | pointer to 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 data of the file information for this section (or 0 if no fileinfo is present) |
| debugsize | total byte length of debugging data for this section |
| name or nsyms | string or integer |

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 main file name (notionally a module name). Its syntax is similar to that for 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.

The following language byte codes are defined:-

| | |
|---|---|
| 0 | Low-level debugging data (notionally, assembler) |
| 1 | C |
| 2 | Pascal |
| 3 | Fortran77 |
| other | reserved to Acorn. |

The fileinfo field is 0 if no source file information is present.

The debugversion field was defined to be 1; the new debugversion for the extended debugging data format (encompassing low-level debugging data) is 2. For low-level debugging data, other fields have the following values:-

| | |
|---|---|
| language | 0 |
| codeaddr | Image$$RO$$Base |
| dataaddr | Image$$RW$$Base |
| codesize | Image$$RO$$Limit - Image$$RO$$Base |
| datasize | Image$$RW$$Limit - Image$$RW$$Base |
| fileinfo | 0 |
| nsyms | number of symbols within the following debugging data |
| debugsize | total size of the low-level debugging data including the size of the section item |

The section item is immediately followed by nsyms symbols, each having the following format:-

| | |
|---|---|
| stridx:24 | byte offset in string table of symbol name |
| flags:8 | (see below) |
| value | the value of the symbol |

The flags field has the following values:-

| | |
|---|---|
| 0/1 | the symbol is a local/global symbol |
| + | (there may be many local symbols with the same name) |
| 0/2/4/6 | symbol names an absolute/code/data/zero-init value |

Note that the linker reduces all symbol values to absolute values. The flags field records the history, or origin, of the symbol in the image.

The string table is in standard AOF format. It consists of a length word followed by the strings themselves, each terminated by a NUL (0). The length word includes the length of the length word, so no offset into the string table is less than 4. The end of the string table is padded to the next word boundary.

## Procedure

A procedure item appears once for each procedure or function definition in the source program. Any definitions with the procedure have their related debugging data items between the procedure item and the matching endproc item. The format of procedure items is as follows:-

| | |
|---|---|
| type | the return type if this is a function, else 0 |
| args | the number of arguments |
| sourcepos | a word encoding the source position of the start of the procedure |
| startaddr | pointer to the first instruction of the procedure |
| bodyaddr | pointer to the first instruction of the procedure body (see below) |
| endproc | offset of the related endproc item |
| fileentry | offset of the file list entry for the source file |
| name | string |

The bodyaddr field points to the first instruction after the procedure entry sequence, that is the first address at which a high-level breakpoint could sensibly be set. The startaddr field points to the beginning of the entry sequence, that is the address at which control actually arrives when the procedure is called.

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 the compiler into strings prefixed by '`$n`'.

For Fortran77, multiple entry points to the same procedure each give rise to a separate procedure item but they all have the same endproc offset referring to a single endproc item.

## Endproc

This item marks the end of the debugging data items belonging to a particular procedure. It also contains information relating to the procedure's return. Its format is as follows:-

| | |
|---|---|
| sourcepos | a word encoding the position in the source file of the end of the procedure |
| endaddı | a pointer to the code byte AFTER the compiled code for the procedure |
| filentry | offset of the file-list entry for the procedure's end |
| nreturns | number of procedure return points (may be 0) |
| retaddrs... | pointers to the procedure-return code |

If the procedure body is an infinite loop, there will be no return point so nreturns will be 0. Otherwise the retaddrs should each 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 in this procedure.

## Variable

This 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). Its format is as follows:-

| | |
|---|---|
| type | a type word |
| sourcepos | a word encoding the source position of the variable |
| class | a word encoding the variable's storage class |
| location | see explanation below |
| name | string |

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 (ie an offset from the frame- pointer) for automatic and var-type arguments; an offset into the argument list for Fortran arguments; and 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 (eg identically named variables in disjoint source-level naming scopes such as nested block in C).

## Type

This item is used to describe a named type in the source language (eg a typedef in C). The format is as follows:-

| | |
|---|---|
| type | a type word (described earlier) |
| name | string |

## Struct

This item is used to describe a structured data type (eg a struct in C or a record in Pascal). Its format is as follows:-

| | |
|---|---|
| fields | the number of fields in the structure |
| size | total byte size of the structure |
| fieldtable... | a table of fields entries in the following format:- |

| | |
|---|---|
| offset | byte offset of this field within the structure |
| type | a type word (interpretation as described earlier) |
| name | string |

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

This 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 (different for C and Fortran). The format is as follows:-

| | |
|---|---|
| size | total byte size of each element |
| arrayflags | (see below) |
| basetype | a type word |
| lowerbound | constant value or stack offset of variable |
| upperbound | constant value or stack offset of variable |

If the size field is zero, debugger operations affecting the whole array, rather than individual elements of it, are forbidden.

The following bit numbers in the arrayflags field are defined:-

| | |
|---|---|
| 0 | lower bound is undefined |
| 1 | lower bound is a constant |
| 2 | upper bound is undefined |
| 3 | upper bound is a constant |

If a bound is defined and not constant then it is an integer variable on the stack and the boundvalue field contains the stack offset of the variable (from the frame-pointer).

## Subrange

This item is used to describe 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). Its format is as follows:-

| | |
|---|---|
| size | half-word: 1, 2, or 4 to indicate byte size of object |
| typecode | half-word: simple type code |
| lwb | lower bound of subrange |
| upb | upper bound of subrange |

## Set

This item is used to describe a Pascal set type. Currently, the description is only partial. The format is:-

| | |
|---|---|
| size | byte size of the object |

## Fileinfo

This item appears once per section after all other debugging data items. The half of the header word which would usually give the item length is not required and should be set to 0.

Each source file is described by a sequence of 'fragments', each of which describes a contiguous region of the file within which the addresses of compiled code increase monatonically 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 that make no use of the #include facility, the list of fragments will have only one entry and all line-number information will be contiguous.

The item is a list of entries each with the following format:-

| | |
|---|---|
| length | length of this entry in bytes (0 marks the final entry) |
| date | date and time when the file was last modified |
| filename | string (or null if the name is not known) |
| n | number of fragments following |
| fragments... | n fragments with the following structure... |

| | |
|---|---|
| fragmentsize | length of this entry in bytes |
| firstline | linenumber |
| lastline | linenumber |
| codeaddr | pointer to the start of the fragment's executable code |
| codesize | byte size of the code in the fragment |
| lineinfo... | a variable number of line number data |

There is one lineinfo half-word for each statement of the source file fragment which gives rise to executable code. Exactly what constitutes an executable statement may be defined by the language implementation; the definition may for instance include some declarations. The half-word can be regarded as 2 bytes: the first contains the number of bytes of code generated from the statement and cannot be zero; the second contains the number of source lines occupied by the statement (ie the difference between the line number of the start of the statement and the line number of the next statement). This may be zero if there are multiple statements on the same source line.

If the whole half-word is zero, this indicates that one of the quantities is too large to fit into a byte and that the following 2 half-words contain (in order) the number of lines followed by the number of bytes of code generated from the statement.

# 26 Appendix F - 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 and the implementation language of Acorn's UNIX product RISC iX, the utility of a new language implementation for the ARM will be related to its compatibility with Acorn's implementation of C.

At the end of this document are several examples of the usage of this standard, together with suggestions for generating effective code for the ARM.

## The purpose of APCS

The ARM Procedure Call Standard is a set of rules, designed:

- to facilitate calls between program fragments compiled from different source languages (eg to make subroutine libraries accessible to all compiled languages)
- to give compilers a chance to optimise procedure call, procedure entry and procedure exit (following the reduced instruction set philosophy of the ARM). This standard defines the use of registers, the passing of arguments at an external procedure call, and the format of a data structure that can be used by stack backtracing programs to reconstruct a sequence of outstanding calls. It does so in terms of *abstract register names*. The binding of some register names to

register numbers and the precise meaning of some aspects of the standard are somewhat dependent on the host operating system and are described in separate sections.

Formally, this standard only defines what happens when an external procedure call occurs. Language implementors may choose to use other mechanisms for internal calls and are not required to follow the register conventions described in this document except at the instant of an external call or return. However, other system-specific invariants may have to be maintained if it is required, for example, to deliver reliably an asynchronous interrupt (eg a `SIGINT`) or give a stack backtrace upon an abort (eg when dereferencing an invalid pointer). More is said on this subject in later sections.

## Design criteria

This procedure call standard was defined after a great deal of experimentation, measurement, and study of other architectures. It is believed to be the best compromise between the following important requirements:

- Procedure call must be extremely fast.
- The call sequence must be as compact as possible. (In typical compiled code, calls outnumber entries by a factor in the range 2:1 to 5:1.)
- Extensible stacks and multiple stacks must be accommodated. (The standard permits a stack to be extended in a non-contiguous manner, in stack chunks. The size of the stack does not have to be fixed when it is created, avoiding a fixed partition of the available data space between stack and heap. The same mechanism supports multiple stacks for multiple threads of control.)
- The standard should encourage the production of re-entrant programs, with writable data separated from code.
- The standard must support variation of the procedure call sequence, other than by conventional return from procedure (eg in support of C's `longjmp`, Pascal's `goto-out-of-block`, Modula-2+'s exceptions, UNIX's signals, etc) and tracing of the stack by debuggers and run-time error handlers. Enough is defined about the stack's structure to ensure that implementations of these are possible (within limits discussed later).

## The Procedure Call Standard

This section defines the standard.

### Register names

The ARM has 16 visible general registers and 8 floating-point registers. In interrupt modes some general registers are shadowed and not all floating-point operations are available, depending on how the floating-point operations are implemented.

This standard is written in terms of the register names defined in this section. The binding of certain register names (the **call frame registers**) to register numbers is discussed separately. We do this so that:

- Diverse needs can be more easily accommodated, as can conflicting historical usage of register numbers, yet the underlying structure of the procedure call standard – on which compilers depend critically – remains fixed.
- Run-time support code written in assembly language can be made portable between different register bindings, if it obeys the rules given in the section entitled *Defined bindings of the procedure call standard* on page 274.

The register names and fixed bindings are given immediately below.

#### General Registers

First, the four argument registers:

```
a1   RN   0    ; argument 1/integer result
a2   RN   1    ; argument 2
a3   RN   2    ; argument 3
a4   RN   3    ; argument 4
```

Then the six 'variable' registers:

```
v1   RN   4    ; register variable
v2   RN   5    ; register variable
v3   RN   6    ; register variable
v4   RN   7    ; register variable
v5   RN   8    ; register variable
v6   RN   9    ; register variable
```

Then the call-frame registers, the bindings of which vary (see the section entitled *Defined bindings of the procedure call standard* on page 274 for details):

```
sl             ; stack limit / stack chunk handle
fp             ; frame pointer
ip             ; temporary workspace, used in
                 procedure entry
sp   RN   13   ; lower end of current stack frame
```

Finally, lr and pc, which are determined by the ARM's hardware:

```
lr   RN   14   ; link address on calls/temporary workspace
pc   RN   15   ; program counter and processor status
```

In the obsolete APCS-A register bindings described below, sp is bound to r12; in all other APCS bindings, sp is bound to r13.

### Notes

Literal register names are given in lower case, eg v1, sp, lr. In the text that follows, symbolic values denoting 'some register' or 'some offset' are given in upper case, eg R, R+N.

References to 'the stack' denoted by sp assume a stack that grows from high memory to low memory, with sp pointing at the top or front (ie lowest addressed word) of the stack.

At the instant of an external procedure call there must be nothing of value to the caller stored below the current stack pointer, between sp and the (possibly implicit, possibly explicit) stack (chunk) limit. Whether there is a single stack chunk or multiple chunks, an explicit stack limit (in sl) or an implicit stack limit, is determined by the register bindings and conventions of the target operating system.

Here and in the text that follows, for any register R, the phrase 'in R' refers to the contents of R; the phrase 'at [R]' or 'at [R, #N]' refers to the word pointed at by R or R+N, in line with ARM assembly language notation.

### Floating-point Registers

The floating-point registers are divided into two sets, analogous to the subsets a1–a4 and v1–v6 of the general registers. Registers f0–f3 need not be preserved by a called procedure; f0 is used as the floating-point result register. In certain restricted circumstances (noted below), f0–f3 may be used to hold the first four floating-point arguments. Registers f4–f7, the so called 'variable' registers, must be preserved by callees.

The floating-point registers are:

```
f0   FN   0   ; floating point result (or 1st FP argument)
f1   FN   1   ; floating point scratch register (or 2nd FP arg)
f2   FN   2   ; floating point scratch register (or 3rd FP arg)
f3   FN   3   ; floating point scratch register (or 4th FP arg)
f4   FN   4   ; floating point preserved register
f5   FN   5   ; floating point preserved register
f6   FN   6   ; floating point preserved register
f7   FN   7   ; floating point preserved register
```

## Data representation and argument passing

The APCS is defined in terms of N (>= 0) word-sized arguments being passed from the caller to the callee, and a single word or floating-point result passed back by the callee. The standard does not describe the layout in store of records, arrays and so forth, used by ARM-targeted compilers for C, Pascal, Fortran-77, and so on. In other words, the mapping from language-level objects to APCS words is defined by each language's implementation, not by APCS, and, indeed, there is no formal reason why two implementations of, say, Pascal for the ARM should not use different mappings and, hence, not be cross-callable.

Obviously, it would be very unhelpful for a language implementor to stand by this formal position and implementors are strongly encouraged to adopt not just the letter of APCS but also the obviously natural mappings of source language objects into argument words. Strong hints are given about this in later sections which discuss (some) language specifics.

## Register usage and argument passing to external procedures

### Control Arrival

We consider the passing of N (>= 0) actual argument words to a procedure which expects to receive either exactly N argument words or a variable number V (>= 1) of argument words (it is assumed that there is at least one argument word which indicates in a language-implementation-dependent manner how many actual argument words there are: for example, by using a format string argument, a count argument, or an argument-list terminator).

At the instant when control arrives at the target procedure, the following shall be true (for any M, if a statement is made about argM, and M > N, the statement can be ignored):

```
arg1 is in a1
arg2 is in a2
arg3 is in a3
arg4 is in a4
for all I >= 5, argI is at [sp, #4*(I-5)]
```

`fp` contains 0 or points to a stack backtrace structure (as described in the next section).

The values in `sp`, `sl`, `fp` are all multiples of four.

`lr` contains the `pc+psw` value that should be restored into `r15` on exit from the procedure. This is known as the *return link value* for this procedure call.

`pc` contains the entry address of the target procedure.

Now, let us call the lower limit to which sp may point **in this stack chunk** SP_LWM (Stack-Pointer Low Water Mark). Remember, it is unspecified whether there is one stack chunk or many, and whether SP_LWM is implicit, or explicitly derived from sl; these are binding-specific details. Then:

> Space between sp and SP_LWM shall be (or shall be on demand) readable, writable memory which can be used by the called procedure as temporary workspace and overwritten with any values before the procedure returns.

`sp >= SP_LWM + 256.`

This condition guarantees that a stack extension procedure, if used, will have a reasonable amount – 256 bytes – of work space available to it, probably sufficient to call two or three procedure invocations further.

### Control Return

At the instant when the return link value for a procedure call is placed in the pc+psw, the following statements shall be true:

fp, sp, sl, v1-v6, and f4-f7 shall contain the same values as they did at the instant of the call. If the procedure returns a word-sized result, R, which is not a floating-point value, then R shall be in a1. If the procedure returns a floating-point result, FPR, then FPR shall be in f0.

### Notes

The definition of control return means that this is a 'callee saves' standard.

The requirement to pass a variable number of arguments to a procedure (as in old-style C) precludes the passing of floating-point arguments in floating-point registers (as the ARM's fixed point registers are disjoint from its floating-point registers). However, if a callee is defined to accept a fixed number K of arguments and its interface description declares it to accept exactly K arguments of matching types, then it is permissible to pass the first four floating-point arguments in floating-point registers f0-f3. However, Acorn's C compiler for the ARM does not yet exploit this latitude.

The values of a2-a4, ip, lr and f1-f3 are not defined at the instant of return.

The Z, N, C and V flags are set from the corresponding bits in the return link value on procedure return. For procedures called using a BL instruction, these flag values will be preserved across the call.

The flag values from lr at the instant of entry must be restored; it is not sufficient merely to preserve the flag values across the call. (Consider a procedure ProcA which has been 'tail-call optimised' and does: CMPS a1, #0; MOVLT a2,

#255; MOVGE a2, #0; B ProcB. If ProcB merely preserves the flags it sees on entry, rather than restoring those from lr, the wrong flags may be set when ProcB returns direct to ProcA's caller).

This standard does not define the values of fp, sp and sl at arbitrary moments during a procedure's execution, but only at the instants of (external) call and return. Further standards and restrictions may apply under particular operating systems, to aid event handling or debugging. In general, you are strongly encouraged to preserve fp, sp and sl, at all times.

The minimum amount of stack defined to be available is not particularly large, and as a general rule a language implementation should not expect much more, unless the conventions of the target operating system indicate otherwise. For example, code generated by the Arthur/RISC OS C compiler is able, if there is inadequate local workspace, to allocate more stack space from the C heap before continuing. Any language unable to do this may have its interaction with C impaired. That sl contains a stack chunk handle is important in achieving this. (See the section entitled *Defined bindings of the procedure call standard* on page 274 for further details).

The statements about sp and SP_LWM are designed to optimise the testing of the one against the other. For example, in the RISC OS user-mode binding of APCS, sl contains SL_LWM+512, allowing a procedure's entry sequence to include something like:

```
CMP  sp, sl
BLLT |x$stack_overflow|
```

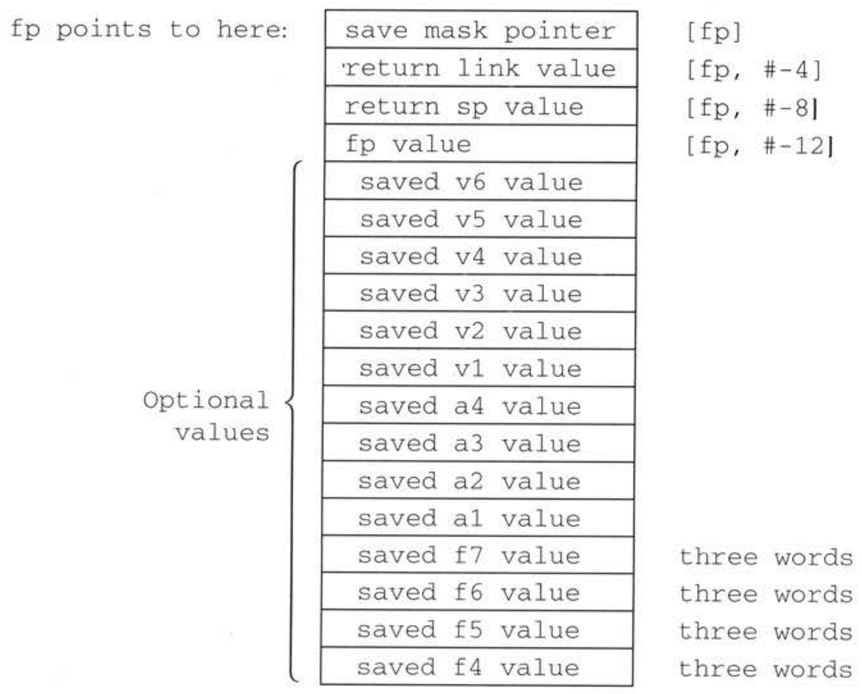
where x$stack_overflow is a part of the run-time system for the relevant language. If this test fails, and x$stack_overflow is not called, there are at least 512 bytes free on the stack.

This procedure should only call other procedures when sp has been dropped by 256 bytes or less, guaranteeing that there is enough space for the called procedure's entry sequence (and, if needed, the stack extender) to work in.

If 256 bytes are not enough, the entry sequence has to drop sp before comparing it with sl in order to force stack extension (see later sections on implementation specifics for details of how the RISC OS C compiler handles this problem).

## The stack backtrace data structure

At the instant of an external procedure call, the value in fp is zero or it points to a data structure that gives information about the sequence of outstanding procedure calls. This structure is in the format shown below:



This picture shows between four and 26 words of store, with those words higher on the page being at higher addresses in memory. The presence of any of the optional values does not imply the presence of any other. The floating-point values are in extended format and occupy three words each.

At the instant of procedure call, all of the following statements about this structure shall be true:

- The **return fp value** is either 0 or contains a pointer to another stack backtrace data structure of the same form. Each of these corresponds to an active, outstanding procedure invocation. The statements listed here are also true of this next stack backtrace data structure and, indeed, hold true for each structure in the chain.
- The **save mask pointer** value, when bits 0, 1, 26, 27, 28, 29, 30, 31 have been cleared, points twelve bytes beyond a word known as the **return data save instruction**.

- The return data save instruction is a word that corresponds to an ARM instruction of the following form:

  ```
  STMDB   sp!, {[a1], [a2], [a3], [a4],
                [v1], [v2], [v3], [v4], [v5], [v6],
                fp, ip, lr, pc}
  ```

  Note the square brackets in the above denote optional parts: thus, there are 12 x 1024 possible values for the return data save instruction, corresponding to the following bit patterns:

  ```
            1110 1001 0010 1101 1101 10xx xxxx xxxx    APCS-R, APCS-U
  or
                              !    !  !
            1110 1001 0010 1100 1100 11xx xxxx xxxx    APCS-A (obsolete)
  ```

  The least significant 10 bits represent argument and variable registers: if bit N is set, then register N will be transferred.

  The optional parts `a1`, `a2`, `a3`, `a4`, `v1`, `v2`, `v3`, `v4`, `v5` and `v6` in this instruction correspond to those optional parts of the stack backtrace data structure that are present such that: for all M, if `vM` or `aM` is present then so is `saved vM value` or `saved aM value`, and if `vM` or `aM` is absent then so is `saved vM value` or `saved aM value`. This is as if the stack backtrace data structure were formed by the execution of this instruction, following the loading of `ip` from `sp` (as is very probably the case).

- The sequence of up to four instructions following the return data save instruction determines whether saved floating-point registers are present in the backtrace structure. The four optional instructions allowed in this sequence are:

  ```
  STFE f7, [sp, #-12]! ; 1110 1101 0110 1101 0111 0001 0000 0011
  STFE f6, [sp, #-12]! ; 1110 1101 0110 1101 0110 0001 0000 0011
  STFE f5, [sp, #-12]! ; 1110 1101 0110 1101 0101 0001 0000 0011
  STFE f4, [sp, #-12]! ; 1110 1101 0110 1101 0100 0001 0000 0011
                                            !
  ```

  Any or all of these instructions may be missing, and any deviation from this order or any other instruction terminates the sequence.

  (A historical bug in the C compiler (now fixed) inserted a single arithmetic instruction between the return data save instruction and the first STFE. Some Acorn software allows for this.)

  The bit patterns given are for APCS-R/APCS-U register bindings. In the obsolete APCS-A bindings, the bit indicated by `!` is 0.

  The optional instructions saving `f4`, `f5`, `f6` and `f7` correspond to those optional parts of the stack backtrace data structure that are present such that: for all M, if `STFE fM` is present then so is `saved fM value`; if `STFE fM` is absent then so is `saved fM value`.

- At the instant when procedure A calls procedure B, the stack backtrace data structure pointed at by `fp` contains exactly those elements `v1`, `v2`, `v3`, `v4`, `v5`, `v6`, `f4`, `f5`, `f6`, `f7`, `fp`, `sp` and `pc` which must be restored into the corresponding ARM registers in order to cause a correct exit from procedure A, albeit with an incorrect result.

**Notes**

The following example suggests what the entry and exit sequences for a procedure are likely to look like (though entry and exit are not defined in terms of these instruction sequences because that would be too restrictive; a good compiler can often do better than is suggested here):

```
entry   MOV   ip, sp
        STMDB   sp!, {argRegs, workRegs, fp, ip, lr, pc}
        SUB     fp, ip, #4
exit    LDMDB   fp, {workRegs, fp, sp, pc}^
```

Many apparent idiosyncrasies in the standard may be explained by efforts to make the entry sequence work smoothly. The example above is neither complete (no stack limit checking) nor mandatory (making arguments contiguous for C, for instance, requires a slightly different entry sequence; and storing `argRegs` on the stack may be unnecessary).

The `workRegs` registers mentioned above correspond to as many of `v1` to `v6` as this procedure needs in order to work smoothly. At the instant when procedure `A` calls any other, those workspace registers not mentioned in `A`'s return data save instruction will contain the values they contained at the instant `A` was entered. Additionally, the registers `f4-f7` not mentioned in the floating-point save sequence following the return data save instruction will also contain the values they contained at the instant `A` was entered.

This standard does not require anything of the values found in the optional parts `a1`, `a2`, `a3`, `a4` of a stack backtrace data structure. They are likely, if present, to contain the saved arguments to this procedure call; but this is not required and should not be relied upon.

## Defined bindings of the procedure call standard

### APCS-R and APCS-U: The RISC OS and RISC iX PCSs

These bindings of the APCS are used by:

- RISC OS applications running in ARM user-mode
- compiled code for RISC OS modules and handlers running in ARM SVC-mode
- RISC iX applications (which make no use of `sl`) running in ARM user mode

- RISC iX kernels running in ARM SVC mode.

The call-frame register bindings are:

```
sl    RN    10    ; stack limit / stack chunk handle
                  ;   unused by RISC iX applications
fp    RN    11    ; frame pointer
ip    RN    12    ; used as temporary workspace
sp    RN    13    ; lower end of current stack frame
```

Although not formally required by this standard, it is considered good taste for compiled code to preserve the value of `sl` everywhere.

The invariants `sp > ip > fp` have been preserved, in common with the obsolete APCS-A (described below), allowing symbolic assembly code (and compiler code-generators) written in terms of register names to be ported between APCS-R, APCS-U and APCS-A merely by relabelling the call-frame registers provided:

- When call-frame registers appear in `LDM`, `LDR`, `STM` and `STR` instructions they are named symbolically, never by register numbers or register ranges.
- No use is made of the ordering of the four call-frame registers (eg in order to load/save `fp` or `sp` from a full register save).

**APCS-R: Constraints on `sl`** (For RISC OS applications and modules)

In SVC and IRQ modes (collectively called module mode) `SL_LWM` is implicit in `sp`: it is the next megabyte boundary below `sp`. Even though the SVC-mode and IRQ-mode stacks are not extensible, `sl` still points 512 bytes above a skeleton stack-chunk descriptor (stored just above the megabyte boundary). This is done for compatibility with use by applications running in ARM user-mode and to facilitate module-mode stack-overflow detection. In other words:

```
sl = SL_LWM + 512.
```

When used in user-mode, the stack is segmented and is extended on demand. Acorn's language-independent run-time kernel allows language run-time systems to implement stack extension in a manner which is compatible with other Acorn languages. `sl` points 512 bytes above a full stack-chunk structure and, again:

```
sl = SL_LWM + 512.
```

Mode-dependent stack-overflow handling code in the language-independent run-time kernel faults an overflow in module mode and extends the stack in application mode. This allows library code, including the run-time kernel, to be shared between all applications and modules written in C.

In both modes, the value of `sl` must be valid immediately before each external call **and each return from an external call**.

Deallocation of a stack chunk may be performed by intercepting returns from the procedure that caused it to be allocated. Tail-call optimisation complicates the relationship, so, in general, `sl` is required to be valid immediately before every return from external call.

**APCS-U: Constraints on `sl`** (For RISC iX applications and RISC iX kernels)

In this binding of the APCS the user-mode stack auto-extends on demand so `sl` is unused and there is no stack-limit checking.

In kernel mode, `sl` is reserved by Acorn.

## APCS-A: The obsolete Arthur application PCS

This obsolete binding of the procedure-call standard is used by Arthur applications running in ARM user-mode. The applicable call-frame register bindings are as follows:

```
sl    RN    13    ; stack limit/stack chunk handle
fp    RN    10    ; frame pointer
ip    RN    11    ; used as temporary workspace
sp    RN    12    ; lower end of current stack frame
```

(Use of `r12` as `sp`, rather than the architecturally more natural `r13`, is historical and predates both Arthur and RISC OS.)

In this binding of the APCS, the stack is segmented and is extended on demand. Acorn's language-independent run-time kernel allows language run-time systems to implement stack extension in a manner which is compatible with other Acorn languages.

The stack limit register, `sl`, points 512 bytes above a stack-chunk descriptor, itself located at the low-address end of a stack chunk. In other words:

```
sl = SL_LWM + 512.
```

The value of `sl` must be valid immediately before each external call and each return from an external call.

Although not formally required by this standard, it is considered good taste for compiled code to preserve the value of `sl` everywhere.

## Notes on APCS bindings

### Invariants and APCS-M

In all future supported bindings of APCS `sp` shall be bound to `r13`. In all supported bindings of APCS the invariant `sp > ip > fp` shall hold. This means that the only other possible binding of APCS is APCS-M:

```
sl      RN      12      ; stack limit/stack chunk handle
fp      RN      10      ; frame pointer
ip      RN      11      ; used as temporary workspace
sp      RN      13      ; lower end of current stack frame
```

This binding of APCS is unlikely to be used (by Acorn).

### Further Restrictions in SVC Mode and IRQ Mode

There are some consequences of the ARM's architecture which, while not formally acknowledged by the ARM Procedure Call Standard, need to be understood by implementors of code intended to run in the ARM's SVC and IRQ modes.

An IRQ corrupts `r14_irq`, so IRQ-mode code must run with IRQs off until `r14_irq` has been saved. Acorn's preferred solution to this problem is to enter and exit IRQ handlers written in high-level languages via hand-crafted 'wrappers' which on entry save `r14_irq`, change mode to SVC, and enable IRQs and on exit return to the saved `r14_irq` (which also restores IRQ mode and the IRQ-enable state). Thus the handlers themselves run in SVC mode, avoiding this problem in compiled code.

Both SWIs and aborts corrupt `r14_svc`. This means that care has to be taken when calling SWIs or causing aborts in SVC mode.

In high-level languages, SWIs are usually called out of line so it suffices to save and restore `r14` in the calling veneer around the SWI. If a compiler can generate in-line SWIs, then it should, of course, also generate code to save and restore `r14` in-line, around the SWI, unless it is known that the code will not be executed in SVC mode.

An abort in SVC mode may be symptomatic of a fatal error or it may be caused by page faulting in SVC mode. Acorn expects SVC-mode code to be correct, so these are the only options. Page faulting can occur because an instruction needs to be fetched from a missing page (causing a prefetch abort) or because of an attempted data access to a missing page (causing a data abort). The latter may occur even if the SVC-mode code is not itself paged (consider an unpaged kernel accessing a paged user-space).

A data abort is completely recoverable provided `r14` contains nothing of value at the instant of the abort. This can be ensured by:

- saving R14 on entry to every procedure and restoring it on exit
- not using R14 as a temporary register in any procedure
- avoiding page faults (stack faults) in procedure entry sequences.

A prefetch abort is harder to recover from and an aborting BL instruction cannot be recovered, so special action has to be taken to protect page faulting procedure calls.

For Acorn C, R14 is saved in the second or third instruction of an entry sequence. Aligning all procedures at addresses which are 0 or 4 modulo 16 ensures that the critical part of the entry sequence cannot prefetch-abort. A compiler can do this by padding all code sections to a multiple of 16 bytes in length and being careful about the alignment of procedures within code sections.

Data-aborts early in procedure entry sequences can be avoided by using a software stack-limit check like that used in APCS-R.

Finally, the recommended way to protect BL instructions from prefetch-abort corruption is to precede each BL by a `MOV ip, pc` instruction. If the BL faults, the prefetch abort handler can safely overwrite `r14` with `ip` before resuming execution at the target of the BL. If the prefetch abort is not caused by a BL then this action is harmless, as R14 has been corrupted anyway (and, by design, contained nothing of value at any instant a prefetch abort could occur).

## Examples from Acorn language implementations

### Example procedure calls in C

Here is some sample assembly code as it might be produced by the C compiler:

```
; gggg is a function of 2 args that needs one register.variable (v1)
gggg    MOV     ip, sp
        STMFD   sp!, {a1, a2, v1, fp, ip, lr, pc}
        SUB     fp, ip, #4              ; points at saved PC
        CMPS    sp, sl
        BLLT    |x$stack_overflow|      ; handler procedure
        ...
        MOV     v1, ...                 ; use a register variable
        ...
        BL      ffff
        ...
        MOV    ..., v1                  ; rely on its value after ffff()
```

Within the body of the procedure, arguments are used from registers, if possible; otherwise they must be addressed relative to `fp`. In the two argument case shown above, `arg1` is at `[fp,#-24]` and `arg2` is at `[fp,#-20]`. But as discussed below, arguments are sometimes stacked with positive offsets relative to `fp`.

Local variables are never addressed offset from `fp`; they always have positive offsets relative to `sp`. In code that changes `sp` this means that the offsets used may vary from place to place in the code. The reason for this is that it permits the procedure `x$stack_overflow` to recover by setting `sp` (and `sl`) to some new stack segment. As part of this mechanism, `x$stack_overflow` may alter memory offset from `fp` by negative amounts, eg `[fp, #-64]` and downwards, provided that it adjusts `sp` to provide workspace for the called routine.

If the function is going to use more than 256 bytes of stack it must do:

```
        SUB     ip, sp, #<my stack size>
        CMPS    ip, sl
        BLLT    |x$stack_overflow_1|
```

instead of the two-instruction test shown above.

If a function expects no more than four arguments it can push all of them onto the stack at the same time as saving its old `fp` and its return address (see the example above); arguments are then saved contiguously in memory with `arg1` having the lowest address. A function that expects more than four arguments has code at its head as follows:

```
MOV     ip, sp
STMFD   sp!, {a1, a2, a3, a4}       ; put arg1-4 below stacked args
STMFD   sp!, {v1, v2, fp, ip, lr, pc}  ; v1-v6 saved as necessary
SUB     fp, ip, #20                 ; point at newly created call-frame
CMPS    sp, sl
BLLT    |x$stack_overflow|
...
...
LDMEA   fp, {v1, v2, fp, sp, pc}^   ; restore register vars & return
```

The store of the argument registers shown here is not mandated by APCS and can often be omitted. It is useful in support of debuggers and run-time trace-back code and required if the address of an argument is taken.

The entry sequence arranges that arguments (however many there are) lie in consecutive words of memory and that on return `sp` is always the lowest address on the stack that still contains useful data.

The time taken for a call, enter and return, with no arguments and no registers saved, is about 22 S-cycles.

Although not required by this standard, the values in `fp`, `sp` and `sl` are maintained while executing code produced by the C compiler. This makes it much easier to debug compiled code.

Multi-word results other than double precision reals in C programs are represented as an implicit first argument to the call, which points to where the caller would like the result placed. It is the first, rather than the last, so that it works with a C function that is not given enough arguments.

## Procedure calls in other language implementations

### Assembler

The procedure call standard is reasonably easy and natural for assembler programmers to use. The following rules should be followed:

- Call-frame registers should always be referred to explicitly by symbolic name, never by register number or implicitly as part of a register range.
- The offsets of the call-frame registers within a register dump should not be wired into code. Always use a symbolic offset so that you can easily change the register bindings.

### Fortran

The Acorn/TopExpress Arthur/RISC OS Fortran-77 compiler violates the APCS in a number of ways that preclude inter-working with C, except via assembler veneers. This may be changed in future releases of the Fortran-77 product.

### Pascal

The Acorn/3L Arthur/RISC OS ISO-Pascal compiler violates the APCS in a number of ways that preclude inter-working with C, except via assembler veneers. This may be changed in future releases of the ISO-Pascal product.

### Lisp, BCPL and BASIC

These languages have their own special requirements which make it inappropriate to use a procedure call of the form described here. Naturally, all are capable of making external calls of the given form, through a small amount of assembler 'glue' code.

### General

Note that there is no requirement specified by the standard concerning the production of re-entrant code, as this would place an intolerable strain on the conventional programming practices used in C and Fortran. The behaviour of a procedure in the face of multiple overlapping invocations is part of the specification of that procedure.

## Various lessons

This document is not intended as a general guide to the writing of code-generators, but it is worth highlighting various optimisations that appear particularly relevant to the ARM and to this standard.

The use of a callee-saving standard, instead of a caller-saving one, reduces the size of large code images by about 10% (with compilers that do little or no interprocedural optimisation).

In order to make effective use of the APCS, compilers must compile code a procedure at a time. Line-at-a-time compilation is insufficient.

The preservation of condition codes over a procedure call is often useful because any short sequence of instructions (including calls) that forms the body of a short `IF` statement can be executed without a branch instruction. For example:

```
if (a < 0) b = foo();
```

can compile into:

```
        CMP     a, #0
        BLLT    foo
        MOVLT   b, a1
```

In the case of a **leaf** or **fast** procedure – one that calls no other procedures –much of the standard entry sequence can be omitted. In very small procedures, such as are frequently used in data abstraction modules, the cost of the procedure can be very small indeed. For instance, consider:

```
        typedef struct {...; int a; ...} foo;
        int get_a(foo* f) {return(f->a);}
```

The procedure `get_a` can compile to just:

```
        LDR     a1, [a1, #aOffset]
        MOVS    pc, lr
```

This is also useful in procedures with a conditional as the top level statement, where one or other arm of the conditional is fast (ie calls no procedures). In this case there is no need to form a stack frame there. For example, using this, the C program:

```
        int sum(int i)
        {
           if (i <= 1)
              return(i);
           else
              return(i + sum(i-1));
        }
```

could be compiled into:

```
sum     CMP      a1, #1 ; try fast case
        MOVSLE   pc, lr ; and if appropriate, handle quickly!
        ; else, form a stack frame and handle the rest as normal code.
        MOV      ip, sp
        STMDB    sp!, {v1, fp, ip, lr, pc}
        CMP      sp, sl
        BLLT     overflow
        MOV      v1, a1                  ; register to hold i
        SUB      a1, a1, #1              ; set up argument for call
        BL       sum                     ; do the call
        ADD      a1, a1, v1              ; perform the addition
        LDMEA    fp, {v1, fp, sp, pc}^   ; and return
```

This is only worthwhile if the test can be compiled using only `ip`, and any spare of `a1-a4`, as scratch registers. This technique can significantly speed up certain speed-critical routines, such as read and write character. At the present time, this optimisation is not performed by the C compiler.

Finally, it is often worth applying the tail call optimisation, especially to procedures which need to save no registers. For example, the code fragment:

```
extern void *malloc(size_t n)
{
    return primitive_alloc(NOTGCABLEBIT, BYTESTOWORDS(n));
}
```

is compiled by the C compiler into:

```
malloc  ADD     a1, a1, #3              ; 1S
        MOV     a2, a1, LSR #2          ; 1S
        MOV     a1, #1073741824         ; 1S
        B       primitive_alloc         ; 1N+2S = 4S
```

This avoids saving and restoring the call-frame registers and minimises the cost of interface 'sugaring' procedures. This saves five instructions and, on a 4/8MHz ARM, reduces the cost of the malloc sugar from 24S to 7S.

# Index

# Reader's Comment Form

*Acorn Desktop Development Environment*

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** |

*Cut out (or photocopy) and post to:*
Dept RC, Technical Publications
Acorn Computers Limited
645 Newmarket Road
Cambridge CB5 8PB
England

**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

Acorn

SKD37