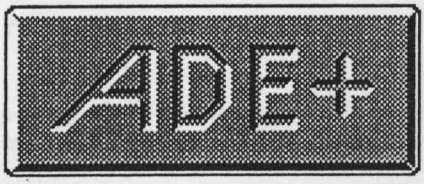
The Ultimate Assembly Language Development Tool . . .



Reference and Tutorial Guide

SYSTEM SOFTWARE SOUTH YORKSHIRE SYSTEMS FOR TRAINING AND MANAGEMENT LIMITED. 12 COLLEGIATE CRESCENT SHEFFIELD S10 2BA. Tel. (0742) 682321

Published in the United Kingdom by:
South Yorkshire Systems for Training Education and Management Limited,
12 Collegiate Crescent,
Sheffield, S10 2BA,
England.

First Published 1986

A companion volume to this user guide, *The ADE plus Technical Reference Guide*, is sold separately and obtainable from the above address. The Technical Reference Guide explains in detail how the user can extend the ADE plus toolkit by writing additional ROM modules with the basic kit.

ADE plus

Release Note 2 March 1987

For the latest information please consult the file called README on the ADE plus disc.

The files shipped with this version are:

$.A4080
This file copies the 40 track DFS disc on to an 80 track blank formatted disc. It is only shipped with the 40 track disc.

CONVERT
A BASIC program to convert BASIC assembler source into ADE format. Type CH "CONVERT" (from BASIC) and follow the instructions on the screen. You may still need to edit the resultant file.
Many thanks to Dr Oliver Blatchford who improved this program.

BOOT
An EXEC file to load the ROM images MMU and ASM into sideways RAM. Not shipped on the 40 track disc.

MMU
ASM
The ADE plus ROM images. Not shipped on the 40 track version.

DEBUG
The debugger in ROM image form. Once this is loaded press crtl-break and enter ADE plus. The ROM is now accessible through the DEBUG command.

DEBUGL
RAM version of DEBUG. Memory used is 6C00-7C00 and 88-8F. To start this debugger check you are in MODE 7 or a slowdown mode and type *DEBUGL (or use LBUG).

DEBUGH
RAM version of DEBUG for second processor. Memory limits E800-F800, 88-8F. To run this debugger (with second processor) type *DEBUGH.

LBUG
Loader for DEBUGL. This allows a sideways RAM page to be selected. A decimal parameter is specified, eg
*LBUG 6.
to debug page 6. DEBUGL must be present.

PRSRAM
Utility to protect all ROM images in sideways RAM from the ADE print buffer.

DS
A symbolic disassembler. The source for this program is in T.DS. To use the program enter

*DS <code file> (<symbol file>)

where <symbol file> is a file of symbols output by the ADE linker.

FILTER
The file filter program as detailed in the user guide. The source for this is T.Filter.

MKDS
A batch file to assemble and link the symbolic disassembler.

MKF
A batch file to assemble and link the FILTER program.

Directory T

T. FILTER
Source for L. filter.

T.DS
Source for L.DS.

T.MACLIB
Source for M.MACLIB.

T.TEST
Source of all 65C00 series op codes
used by ADE plus.

T.ADV
Example assembly program.

T.DEMO
Demo program as mentioned in the
user guide.

T.LBUG
Source for LBUG.

T.PRSRAM
Source for PRSRAM.

Directory H

H.ADELIB
Header file for the library
L.ADELIB.

Directory M

M.MACLIB
Example macro library
See T.MACLIB to find out
what is in it.

Directory L

L.ADELIB
Example linker library
used by DS and FILTER.

L.DS
Linker module for DS.

L.FILTER
Linker module for FILTER.

L.LBUG
Linker module for LBUG.

CONTENTS

Acknowledgements

The authors wish to thank all those who have helped in the development of ADE plus. Thanks are due to all the original developers of ADE and customers over the years who have made valuable suggestions. We have tried to include all of the best ideas that you have come up with. Thanks are due (again) to Ray for trials work and to Nigel for trials and suggestions for this user guide. Programming was by Steve with helpful assistance from Dave, who also wrote the BASIC program conversion utility.

Chapter 1 Introduction

1.1 On receiving ADE plus...

The first thing you need to do is install the software in your machine. You should have received the software on disc, EPROM or cartridge. Follow the notes below for the package you have.

ADFS and DFS discs

The minimum system consists of two 16K ROM images, the ADE plus MMU and the 65C00 series macro assembler. The disc also contains ROM images for the SPY debugger and example software. These latter two programs can be loaded as required. To load the main system you will need two pages of sideways RAM. The disc is designed for use with the MASTER series microcomputers and you should check that your MASTER is configured for at least this amount of RAM by correctly setting the ROM/RAM links inside the machine. This is explained in the user guide. If you have just received your MASTER it will be correctly configured. If you are working with a BBC B with no sideways RAM, or only 16K on a ROM board, you cannot use this version of the software and you will need to exchange it for the EPROM version.

Insert the disc in the drive and SHIFT-BREAK it. The disc should then load into sideways RAM using the SRLOAD command on the MASTER MOS. Press CTRL-BREAK at the end of this procedure so that the software registers with the MOS. You will need to follow this procedure every time you power up. You can now proceed to Section 1.2.

EPROMs

You will have received two 16K EPROMs and a disc in DFS format (or ADFS by request). The disc contains the SPY debugger and the demonstration software. The debugger is supplied in two versions, one running in main memory and one in sideways RAM. Plug the two EPROMs into any two slots in your machine. Make sure that the ADE plus MMU is in a higher priority socket than the 65C00 series assembler. Ideally both EPROMs should be in a lower priority socket than BASIC because the ADE plus MMU needs to be initialised with a *ADE plus command rather than simply being the first ROM seen and entered on a hard break or power up. When the ROMs are installed the title ADE plus should appear on the screen at power up or after a BREAK.

Cartridge
Plug the cartridge into either of the sockets on the MASTER. The software occupies 32K in two consecutive 16K pages.

.2

The Disc Software

Having loaded your main system as outlined above you may also need to use some of the additional software on the disc. There are two directories for this software. Directory H contains versions of the additional software running at &8000. This software will only run in sideways RAM because it uses RAM workspace in the range &8000-&C000. The other directory is L. Programs in this directory will run in main RAM between &2000 and &7C00. The programs include:

DEBUG: a debugger
FILTER: A file character filter program.
DEMO: A trivial program used in the next section as an introduction to ADE plus.

These programs are in the main directory. Directory T contains the source. Directory M contains the macro library. Directory L contains linker libraries and modules.

A BASIC program CONV is supplied. This will do much of the work in converting BASIC assembly language programs to ADE plus format. Simply CH."CONV" and follow the instructions. The input file that this utility converts is expected to be a BASIC program in internal format stored on disc.

The disc also contains a file called 'README', which can be loaded into the ADE plus editor for viewing. This file contains further information on using ADE plus and the disc utilities, as well as any alterations to this User Guide.

It should be noted that extensive use is made of Macros in the examples in this guide. It is , therefore, necessary to ensure that a disc containing the appropriate Macro library is present before trying to assemble the examples.

1.3 Introducing ADE plus

ADE plus is a modular program development toolkit for developing programs in machine code to run on the BBC microcomputer series. The word *modular* implies that the system comes in discreet parts and can be expanded. Later you should be able to add a Z80 assembler module, an advanced editor and an advanced debugger. You may also obtain the ADE plus technical guide which explains how the system operates and how you can write your own modules to 'plug in'. *Toolkit* implies that ADE plus is more than just another macro assembler. All the modules in the ADE plus system interlink to give you complete flexibility about the way in which you develop your programs. ADE plus contains full linking and library facilities which mean that once you have written and debugged a routine you need never write it again. Your programs can be built in pieces and linked together with library functions supplied by you or other users. Throughout the tutorial, dialogue with the computer is shown in `courier` type and user input is in *italics*. Unless otherwise specified, enter RETURN after each line of input. This introductory tutorial assumes you have used the BASIC assembler, or some other assembler, and are familiar with concepts such as "two pass" and "code origin" etc. If you are an absolute beginner you should read an introductory text or the notes in the BBC micro user guide before proceeding to use ADE plus.

System overview
The two 16K programs you have loaded into your computer in ROM or RAM pages consist of the following:

The ADE plus memory management unit (MMU) central to the operation of the system. The MMU is the first thing you will encounter on entering ADE plus.

A 65C00 series macro assembler which will produce programs ready to run or linker modules to be linked.

A text editor to create new programs and edit old ones.

A macro librarian to create macro libraries.

A linker librarian to create linker libraries.

Getting started
To enter the ADE plus system, type *ADE. This command can also be used to re-initialise the system if you get into trouble. If you have a clock in your machine (eg MASTER) or the time of day is available from the network then you can skip the next

section. If not, you will see the message:

```
ADE plus

Time, date:_
```

Enter the time and date, just the time, or neither. To enter neither (the time will be set to midnight, 00:00) press return. To enter only the time, enter the hours (24 hour clock) followed by a colon followed by the minutes:-

```
Time, date: 10:15  RETURN
```

To enter the time and date, enter the time as above, type a comma and the date, which may be in any format up to 16 characters. (The date is just a string stored in memory to ADE plus.)

```
Time, date: 10:15, Fri 10th Oct 86  RETURN
```

If ADE plus gets the time from your system you will not see the above message. Everyone will then see the ADE plus MMU prompt:

```
10:15 =>_
```

(Or whatever time you entered.) ADE plus has set the system timer clock from your input so that all BBC microcomputers will function in the same way. This clock may not be as accurate as a built-in CMOS clock or Econet clock but is good enough to tell you that you have been working too long and should take a break! The assembler timings are also taken from this clock and may not be as accurate as a quartz stop watch, depending on how much time your filing system steals from the system clock due to missed interrupts.

Above the prompt you will see a screen of status information. This shows the state of the MMU and you can recall this screen at any time the MMU prompt is displayed by typing STAT. Try it now; the screen is repeated, and the latest time displayed with the next prompt. STAT is an ADE plus MMU command. Understanding the status screen is the key to making best use of your computing resources with the ADE plus system. A typical display is:

```
SYSTEM ADE plus 1.0
10th Oct 86

Available memory 83K : 16K protected

Input  23K
Output 14K
Printer unbuffered

Workspace 30K

Assembler options: NONE
Linker options   : NONE

ADE/Linker in slot 15
65C00 assembler in slot 2
Advanced editor not installed
Debugger not installed

10:16 =>_
```

Lets look at each line in turn. The date is as entered or does not appear. The amount of available memory is the total RAM found in the system less the amount used for the screen display. If you are using a shadow screen, this is zero. In the above example the total was made up from 23K of free memory in the BBC B used, 16K of sideways RAM (on a rom card) and 44K of free memory on the second processor attached to the system. The status screen tells you how the memory management unit has divided up that memory. 16K is marked 'protected'. This is because on this system the ADE plus MMU program was in the 16K of sideways RAM. ADE plus is smart enough to avoid writing over itself with printout or assembler code! The three lines 'Input', 'Output' and 'Printer' refer to the three principle buffers used by the ADE plus system. Please read the next section carefully as a good understanding of how ADE plus buffers things will help you get the best out of your software.

A buffer is an area of memory that the MMU has reserved for a specific task. The input buffer is a continuous block of memory reserved for source programs and linker modules being linked. In other words, data travelling *into* the ADE plus system goes through the input buffer. The larger this buffer is, the faster the system will perform because disc accesses will be fewer in number. If you do not have a second processor the input buffer will be zero initially and show as 'unbuffered'. You can alter this with the INPUT command. ADE plus MMU picks the largest available block of memory that is not in sideways RAM (other than the workspace, see below) to be the input buffer. When using a second processor the input buffer will be on the IO processor between PAGE and the bottom of screen memory unless this is less than 14K, in which case 14K of second processor memory will be used. Changing screen mode changes

your own advanced editor or patch VIEW to work with ADE plus. Details about how to do this are provided in the *ADE plus Technical Reference Guide*.

Getting down to work

We have so far covered a few of the basic MMU commands. You should be able to view the status screen and alter the buffer sizes if you do not have a second processor. A full list of MMU commands can be obtained by typing the word COMMANDS (or C.). This replaces the conventional *HELP page, since the MMU commands are not available outside the ADE system. You will notice the RESET command. This sets the buffers to the initial values if you get in a mess with them.

Make sure you have an input buffer of at least 2K. Put the demonstration disc in the drive and enter:

```
10:30 => EDIT T.DEMO RETURN
```

In a second the screen will go into mode 3 and the file T.DEMO will be displayed. You are now in the ADE plus editor. We will not edit the program yet, just read it and get the idea about what is going to happen. The program has an ORG of &2000 so this is where it will run when we assemble it. The QSTR statement is one of the many assembler pseudo ops, as is ORG. When you assemble the program it will prompt you to enter your name, stored under 'name'. Later in the program the use of an inline string variable, $name, inserts this string into the program, printing it out with the rest of the message. It is important to realise it is the assembler that will ask you for your name, not the program when it runs. To run the program press escape. You are now in the command mode of the editor. The name of the file is given at the top of the screen together with free memory. The file is currently in the workspace, and the free memory is the amount of workspace left. Enter the editor command:

```
> RUN RETURN
```

The editor now copies the file to the input buffer and calls the assembler. Enter your name:

```
ADE6500

What is your name? Urblunk RETURN

Pass 2..

(various program statistics and timings)
```

```
Greetings Urblunk, welcome to ADE plus!

10:35 =>_
```

The assembly has completed and the program run. Let's assemble the program again without going back into the editor. Type the command:

```
10:36 => ASM =*,G RETURN
```

Again you may enter your name or a different name and the program runs at the end of the assembly. You have just entered an assembly (ASM) command. If it didn't work check you typed a space before =. The * is the name given by ADE plus to the file currently in the input buffer, so we are saying "assemble the file in the input buffer and run it". The running bit comes from the G which must be preceded by a comma. This is a temporary option, the G or GO option, telling the assembler to run the program at the end of pass 2. You can also enter your name on the assembler command line instead of during assembly. Type:

```
10:37 => ASM =*,G/Fred RETURN
```

The assembler now gets on with the job of assembling without pausing to query you. The name printed will be Fred. Fred is preceded by a slash on the assembler command. The slash tells the assembler that the options (if any) have finished and that the following text is a list of answers to assembler QUERY and QSTR statements.

To edit the program in the input buffer, type:

```
10:39 => EDIT * RETURN
```

(E.* will suffice). The program is displayed. Press escape. Now type:

```
> SEARCH message RETURN
```

The text is redisplayed with the cursor at the start of the word 'message' in the comment on the line beginning "start". If it came up "Not found" then check that you typed the word "message" in lower case. Now, you could press function key 1 to go to the next occurence (next line) but the ADE plus editor has a special key called the GOTO label key, function key 2. Press this and the cursor goes to the label 'message' in the program. This key will become indispensable as you get to use this editor. Move the cursor along to the word Greetings with

the available memory and causes ADE plus to reassign its buffers. Enter the command:

```
10:16 =>MODE 3 RETURN
```

If you get an error, check that there is a space between MODE and 3. Now type STAT and the screen should be redisplayed. The input buffer may now be 14K and the output buffer 8K. If not, you are using a shadow screen and gaining more memory.

The output buffer is used for the output of the assembler, linker and librarians. Again the bigger this memory size the faster the system will perform. The output buffer is in this case taken initially from 14K of memory in the second processor. After typing MODE 3, 8K of free memory is left on the IO processor so this is used as the output buffer and 14K is used as the input buffer.

The printer buffer is made up of all the available sideways RAM that is not marked 'protected'. The print spool system interrupts the normal OS calls to buffer printout and stores it in the sideways RAM. Once initialised this print spooling system will work in BASIC or from any place since it works in line with the operating system. The print spooling system uses the cassette file workspace in page 3 and so cannot be used with cassette systems.

ADE plus will *LOAD and *SAVE files to the buffers wherever possible. Failing that, it will fill or empty the buffers using OSGBPB. If your buffers are always bigger than the source and object files the system will perform at peak efficiency. It is often faster to split assembler source programs into *include* files than to try to assemble one long source. Also note that the ADE plus editor will work with the input buffer, allowing programs to be edited, assembled, run and re-edited without recourse to disc at all.

If your system shows 'unbuffered' for the input and output buffers you do not have a second processor (or an active co-processor on the Master Turbo). You may then set the amount of memory you wish to allocate using the INPUT and OUTPUT commands. This memory will be taken away from the workspace. If you do not allocate any, the ADE plus modules will allocate 1K per buffer from the workspace on a temporary basis each time they are called (to do an assembly for example). Do *not* try to change the buffer sizes if you are using a second processor since they are already optimised. Many ADE plus MMU commands cause the buffers to be re-calculated because the amount of total available memory has changed. To set the

input buffer to 5K, for example, type the command:

```
10:20 => INPUT 5 RETURN
```

You can alter the size of the print spool buffer by protecting or unprotecting pages of sideways RAM. If you unprotect pages with ADE plus software on them the system will eventually crash as the software is eaten by the printout. The command PRINT 0 turns off print spooling and PRINT 1 turns it on.

The next line on the ADE plus status screen, below the buffer allocations, shows the amount of workspace. This is memory on the main processor which is initially between OSHWM and HIMEM, though some may be allocated to the buffers as outlined above.

Two lines then detail the 'Assembler options' and the 'Linker options'. There are 52 flags maintained by the ADE plus MMU that can be set to true or false. 26 of these are allocated to the assembler and 26 to the linker. Specific assemblers may not use all the available flags, and details of which are relevant are given in the reference guide for the assembler you are using. The assembler flags are set with the OPT command; the linker flags with the LOPT command. These flags are *global*, that is, they apply to every assembly or linking operation. Local options may be set in the assembler command line or source program or linker command line that only apply to one operation. These flags are never altered by the assembler or linker, only by the MMU. For example, to set option L on the assembler type:

```
10:22 => OPT L RETURN
```

Now type STAT again. The assembler option list shows L. This means that every assembly will produce a listing unless cancelled by an option -L on the assembly command line. To reset an option (set the flag to false) precede it by a minus sign.

```
10:23 => OPT -L RETURN
```

The status display now reads NONE again (when you next type STAT).

Finally, on the status screen, a list of ROM modules (or RAM images) is given. ADE plus MMU looks for the advanced editor, a 65C00 series assembler, a Z80 assembler and a debugger. If these are found they are listed. In the event of the advanced editor not being available, ADE plus will direct EDIT commands to its own basic text editor. You can write

the arrow key and type 'Warmest'. Note the rest of the line moves along to accommodate the new text. This is because you are in 'insert' mode in the editor. Press function key 0 and you will be able to overtype the word 'Greetings'. Press f0 followed by ' wishes'. We now have an unwanted 'gs' on the end of the line. Press function key 9 twice to delete two characters in front of the cursor. Now press function key 6 to insert a line. The current line moves down to accommodate a new line. Press shift and the left arrow key to move the cursor to the start of the line and type "; my message". This line, starting with a colon, is an assembler comment line. Press escape. Note that the editor status now says "Insert OFF" because you pressed f0. Pressing f0 again in editing mode will turn the insert feature back on. Type RUN to run this amended program. The editor is fully explained in chapter 3.

Using the disc

Next, we will assemble a file from and to disc. The demonstration disc contains a program called ADV in the T directory. Assemble this program to AD (in the root directory, $).

```
11:00 => ASM AD=T.ADV RETURN
```

If all goes well you should end up with the assembler report, at the end of pass two of the assembly, and a program called AD ready to *RUN. Run the program to satisfy your curiosity then assemble it again and examine the assembler report:

```
End of absolute assembly
    0 error(s)
    0 warning(s)

Assembly time   : 5.46
CPU time (ADE)  : 0.47
FS time  (MOS)  : 4.99

Free space 28679 bytes

ADE plus

11:00 =>
```

The timings may differ on your system. The actual time depends on the speed of the DFS and the position of the file on the disc. If it is near the middle of the disc, the program will take longer to assemble due to the time spent simply moving the disc head.

Firstly the report tells us this was an absolute assembly. This means that the program output can be *RUN. The alternative type of assembly is a linker assembly, in which case the output must be processed by the linker before it can be *RUN. No errors or warnings were given. The timings show that most of the time was spent accessing the disc even though the program was *LOADed and SAVED. That is because ADE plus is a *fast* assembler! There is still 28K of free memory for symbols and macros.

It is also possible with the 65C00 assembler to assemble from disc but produce no output or output to memory. To output to memory and run the program, enter:

```
11:02 => ASM =T.ADV,G RETURN
```

Remember the G option? There is no output filename in front of the equals sign so no file is produced. All the ADE file processing programs take commands in the form <output>=<input>. If you omit the G option then no output at all is generated and the program is simply scanned for errors.

Next, edit the program T.ADV and introduce some errors. Check the screen is in mode 3 (type MODE 3) then type:

```
11:05 => EDIT T.ADV RETURN
```

When the file is displayed, press escape (edit command mode) and type:

```
> GOTO 16 RETURN
```

This is a command to tell the editor to go to line 16. This GOTO editor command is useful to debug programs because the assembler gives you the line number in its error reports as we shall see. The screen displays the text with the cursor on the line:

```
        BEQ   DONE
```

Change this to

```
        BEQ   DOME
```

Move the cursor to the next line and change CMP to CPM. Change the label WHITE to +WHITE in line 4 and change the X register to a Z register in line 14. Now press escape and type:

```
> QUIT RETURN
```

The MMU prompt reappears. Enter the command:

```
11:10 => ASM =*  RETURN
```

This performs a syntax check on the program in memory. During pass 2 the listing will display:

```
>S 0000:         4 +WHITE EQU  RED+7
>S 200A:        14        LDA  TEXT,Z
>U 200F:F0EE    16        BEQ  DOME
>O 2011:        17        CPM  #'*
>U 2035:        38        CPX  #WHITE+1
```

These are lines containing errors. The errors are U for unknown symbol and O for illegal opcode and S for syntax error. The two syntax errors are from very different causes and the reports are all brief. This kind of report is often satisfactory to more experienced programmers who do not want reams of output. However, get the assembler to give you a more detailed report:

```
11:11 => ASM =*,E  RETURN
```

You should end the line by pressing ctrl-N before return to paginate the VDU output, otherwise the flood of information will disappear off the top of the screen. The two syntax errors, for example, now show as:

```
>S 0000:         4 +WHITE EQU  RED+7
-----------------^
***** ERROR 8BD7 : line starts with illegal char
>S 200A:B1F5  14        LDA  TEXT,Z
-----------------------------------^
***** ERROR 8D44 : Y register expected
```

This information should be sufficient for the beginner to sort out text problems in the source code. Of course X can be used instead of Y, Y here means 'index register'. The error code, in hex, is of use if you think the assembler is mistaken and should not have produced the error. It is the location in the software from which the error was generated. Please send in this number with any queries to SYSTEM Ltd. about bugs you think may exist in the assembler. More experienced programmers, especially enthusiasts, can use the reference to look at the software and fix the bug. Please write to us with your fixes!

To obtain an error summary (only) enter the command line in the form:

```
11:15 => ASM =*,S  RETURN
```

Option E stands for extended error reporting. Option S for error summary. The summary is printed at the end of pass 2. The error codes are stored in the workspace, so the room available for symbols will be slightly reduced. In this case the errors list as:

```
File:*  line  4 line starts with illegal character
File:*  line 38 unknown symbol
```

Many programmers like to list the errors to a file while they go away and make tea. This is possible. A listing file will capture all output on pass 2. If the program is assembled with no 'list on' commands then this file will be an error file:

```
11:20 => ASM  /ERRS=T.ADV,S RETURN
```

The error file is called ERRS. It is placed on the left-hand side of the equals sign in the command line because it is an output file. It is preceded by a slash to distinguish it from the object output file. Both files may be included, eg: ASM AD/ERRS=T.ADV,S

Enter the command:

```
11:25 => TYPE ERRS RETURN
```

to list the error file. TYPE is a disc based filing system command. ADE plus MMU refers unknown commands to the current filing system in two ways. Firstly, if a file of the command name exists the file is executed as a batch file of ADE plus commands. Secondly, if no file is found the command is passed to the filing system directly through OSCLI. This may execute a filing system command, such as *TYPE. Being able to store ADE plus MMU commands in files and execute them by typing the name of the file will come to be second nature as you progress with the system. The assembler command line is complex because of the flexibility of options and files allowed. Thus complex and often used command lines can be saved in a file and executed. For example, to make the last assembly a single command, enter the editor by typing EDIT with no file name. If an old file appears, or rubbish, or the editor goes into command mode, type NEW from command mode and press escape. You now have a blank sheet!

Enter one line: *ASM /ERRS=T.ADV,S* RETURN

Press escape and type the command:

```
>SAVE ASM1 RETURN
```

Now type the command QUIT to take you back to the ADE plus MMU prompt. Enter the line:

```
11:50 => ASM1 RETURN
```

The command stored in the file ASM1 is executed. In the course of program development the same assembly needs to be repeated often, so this facility will prove to be very useful. A batch file, as this is called, can contain many lines containing input to your assemblies (to QUERY and QSTR) as well as linker commands. The file name is simply *EXECed if it exists. You cannot use one of the inbuilt ADE plus command names as a file name for a batch file, but you can use FS command names such as TYPE and LIST. However these make the FS commands inaccessible. To overcome these two restrictions, any command preceded by a slash will be *EXECed (ie treated as a batch file) and any command preceded by an asterisk will, of course, be passed straight to the operating system command line interpreter.

You should now have a good grasp of the way the system functions. If you are not going to use the linker facilities straight away, read the assembler reference section and begin writing your own programs. Chapter 3 explains all the editor function keys if you are using the inbuilt editor.

The next section in this tutorial guide explains how the linker operates.

1.4 Introducing the Linker

You should be familiar with the normal operation of the assembler before you proceed with this section.

A simple assembler such as the inbuilt BASIC assembler or other assemblers running on the BBC micro translates source statements into machine code a line at a time. This is done in two passes. The first simply ascertains the length of each instruction and assigns a value to each symbol declared in the program. In the second pass the assembler takes the symbolic values and substitutes them in the source program producing the final machine code. Everything must be known in pass two or errors will result. This means that if one program wishes to refer to another, all the entry points in the second program must be precisely defined with EQUates in the first program. In a program development environment the values of symbols will change continually as the program grows or shrinks. The only way of writing large programs was as a single gigantic source which could take many minutes to assemble. The purpose of the linker is to allow you to write programs in pieces, where the value of the symbols may change but where one piece, or module, can find out the correct values of symbols in another module. By this method sources can be kept reasonably short and assembled quickly. The assembler produces a "semi machine code" output file for each module. The linker combines all of these modules to produce an executable program. The linker modules output by the assembler are only slightly larger than pure machine code files, so they take up much less space on the disc than the source programs. Consequently linking, which is mostly disc access and only a little calculation, is much faster than assembly.

In order to explain how the linker operates a short program will be developed. You can use any editor for this. It is assumed that you have understood the first section of this introduction, so the source program will be presented as text without detailed instructions about activating the editor, pressing escape and so on.

Enter the following text as the first part of the program:

```
        MODULE part_1

text    EXT

SYSEXEC ENT
        LDX     #-1
:loop   INX
        LDA     text,X
        BEQ     :done
        JSR     OSWRCH
        JMP     :loop
:done   RTS
```

Save this file as T.PART1. Let's examine this trivial program a line at a time. The MODULE statement declares that the program is a linker module, not an executable program. The module is called part_1. You can choose any name up to 32 characters. Next, "text" is defined as an *external* symbol using EXT. This means that when the program references "text" it is referring to something in another program that is not defined here. The actual value of "text" is supplied by the linker, not the assembler. SYSEXEC is then declared as a global symbol. This is done using the ENT pseudo-op. You can have as many ENT symbols as you wish, but normally they are the labels for the entry points to routines that may be called from other modules. The symbol name SYSEXEC has special significance because the linker will put the final value of this symbol (if defined) in the catalog exec address for the program. SYSEXEC is the label that the filing system will call when it has loaded the program. :loop and :done are local labels because they start with a colon. Local labels can be re-used even in the same assembly using the block statement. (See assembler reference section.) OSWRCH is called to print the characters of "text". Notice that OSWRCH is not defined in the program, even as an external symbol. The assembler pre-defines all the operating system labels at the start of each assembly unless you tell it not to on the assembly command line. This module will take characters from "text" and print them until it finds a zero. Now let's define a second module of text.

```
        MODULE part_2

; text for linker example

text    ENT
        STR     "Now is the time"
        STR     "For all good men"
        STR     "To come to the"
        STR     "ADE of their"
        STR     "Country!"
        BRK
```

Save this file as T.PART2. We now have the text for two modules on disc. The second module contains "text" as an ENTry point. Symbols may be declared as ENT only once, but many modules may each declare them as EXT. The line starting with a semicolon is a comment line. In the BASIC assembler a comment line begins with a back slash. ADE plus will accept a back slash or a semicolon as a comment marker. STR is a pseudo-op to generate a string of text bytes with a RETURN character on the end of each (ASCII 13).

Now each module is separately assembled. From the ADE plus MMU prompt enter:

```
ASM part1=T.part1
ASM part2=T.part2
```

In both cases the programs should assemble without error. Notice that the assembler report now reads "End of linker module assembly". If any errors have occured, correct the programs and reassemble them. To link the two modules, enter:

```
Link prog=part1,part2
```

The display should look somthing like this:

```
ADE plus Linker V 1.0
program error summary..

prog

No linker errors

Linker symbol table
SYSEXEC  <P> 0800    text    <P> 080F
```

The actual values for the symbols may vary. The source programs contained no ORG statements to tell the assembler where to put the program. The output to the linker from the assembler is *relocatable*, that means it can be put anywhere in memory by the linker. We did not tell the linker where to put the program so it placed it in the lowest available free memory. In this case a second processor was used so the program went at 0800 hex. The linker has placed the text at the end of the little routine and assigned the value 80F to "text". This value will be substituted in the main program loop, in the statement LDA text,X. Try to run the program:

```
*RUN PROG
country!DE oftheire
```

The output is not what we expected. This is because, although there were no linker errors, there was a programming error. The routine OSWRCH treats the ASCII RETURN character as "return to start of line". A line feed is also needed. The correct routine to call was OSASCI. Only T.PART1 needs to be changed and assembled then the program can be re-linked. When you are dealing with big programs the virtues of all this will become apparent. Amend T.PART1 to call OSASCI, then re-link as shown above. Remember to reassemble the program or you will link the old unchanged module! The program should run correctly. If you got an error message "Not linker file" check you are linking part1 and not T.part1!

To demonstrate the relocation function of the linker, enter:

```
LINK prog=part2,part1
```

The symbol table shows the text at the start of the program and the label SYSEXEC as 0847 hex. Check the catalog information with *INFO. The exec address in the catalog is correctly set to 0847 so the program will still run with *PROG.

Clearly this is a trivial example, but using the linker with various command line options allows a lot of new and exciting programming techniques to be developed. Firstly, libraries of preassembled modules can be used This allows preassembled modules written by others to be used in your program - floating point libraries, filing system libraries and so on. The routines will be automatically relocated and woven into your program. By specifying a different library, debugging routines may be called. Once debugging is complete, switch to the "run time" library and the final program is produced. The linker also allows large programs to be split into "overlays". A main program is written and then the subroutines split into groups which are largely independent. Each group of subroutines is held in a separate file on disc and loaded when required, allowing common memory to be used.

To understand how to program with overlays and libraries, it is necessary to understand how the linker functions and what is meant by *sections* in a program. The linker recognises four kinds of data:

1. Relocatable program data. This is machine code and binary data (such as the above two examples). The linker strings all the relocatable program data end to end to make one continuous block of code that is saved with the file name given on the left of the equals sign in the linker command line.

2. Relocatable zero page. The linker will allocate zero page variables on a "first come first serve" basis. Using this method libraries may use zero page without conflicting with user routines because the linker sorts out the actual zero page addresses used. These variables are defined with the RZP pseudo-op in the assembly source program.

3. Absolute program data. This is machine code and data written with an ORG in what is termed an *absolute section*.. Each absolute section produces an additional output file from the linker. These files are meant to be used as overlays, so each absolute section may start from the same address.

4. Absolute zero page. Variables may be defined (with EQU) to have absolute zero page addresses. These should be variables used by the MOS and so on whose location is fixed by another program you cannot change. Make all your variables relocatable zero page using RZP to avoid accidental conflicts.

Linker sections are defined in the source program with the RSECT and ASECT statements. RSECT stands for "relocatable (or relative) section" and ASECT for "absolute section". All RSECTs are strung end to end. The MODULE statement is an implicit RSECT so there is no necessity to use RSECT and ASECT statements unless you wish to program with ASECTs for overlays or some other use of absolute data. An ASECT statement must be followed by an ORG statement showing where the section is to be assembled. Here is an example of an overlay scheme:

An adventure game is to be produced. There is a main program loop which will operate on a large data base. To facilitate special graphics effects, part of the data base may be small machine code routines that are called from the main program. The game has five levels, so there are five overlay files for the data and one file for the main program. The main program will be the RSECT so that it can span several linker modules. Each overlay will be an ASECT in one linker module. The ASECTs all begin at &3000, say, so a check needs to be made that the main program does not go beyond this. Relative symbols (labels defined in the RSECT or implicit RSECT after a MODULE) are offset from zero, so if we assemble our main program at &1900, we must put a check in the last module that the last statement is less than &1700 bytes from the first.

The assembler does not know the absolute address, but all relative symbols are given an *offset* value from the start of the module. Offsets are shown in the assembly listing followed by a ' character, so that JSR LOOPZ for example might produce the

output 20 06 09' showing that a symbol &906 bytes from the start of the program has been referenced.

The ASECT files will be given names by the linker based on the main program name followed by a three digit number (up to 255). The first four letters from the main program name are used. If the name is less than four characters it is padded with zeros.

1.5 Advanced Linker Techniques

Producing a memory map
To produce a memory allocation map from the linker, specify the M option with LOPT at ADE command level or with M on the linker command line. For example:

```
LINK =part1,part2;M
```

A semicolon separates the linker module list from the options. The output may be:

```
Program memory map and error summary

Module: part_1
 RSECT: 0800 - 080F

Module: part_2
 RSECT: 080F - 0856
```

Producing a cross reference listing
Specify the X option on the linker command line:

```
LINK =part1,part2;X,M
```

Several options may be listed after the semicolon, separated by commas.

```
Cross reference listing

Module: part_1
    text:part_2
```

Each reference consists of <symbol name>:<module name>. The module names are the ones specified in the MODULE statement, not the file names.

Dumping the linker symbols to disc
A symbol table dump of all global symbols defined with ENT may be made. The symbol table file name is on the left of the equals sign in the linker command line and is preceded by a comma. This file could be used by a program to generate BASIC statements, for example, defining all the call addresses in

a piece of code to BASIC.

Using a symbol file

The linker U option is followed by a file name enclosed in square brackets. This file is a symbol file (produced as outlined above). It is loaded at the start of linker pass 1 and allows modules to reference external labels not actually in the code, for example the operating system labels on a different type of micro computer. To assemble such a table, the symbols are declared in a source program with GEQU:

```
        MODULE tab
WRITE   GEQU   &A000
READ    GEQU   &A006
KEYSC   GEQU   &A010
```

These are then assembled to a linker module:

```
ASM tab=t.tab
```

Linked to produce a symbol table file S.tab, which remains for all time to be used with programs requiring these symbols:

```
LINK ,S.tab=tab
```

```
LINK prog=mod1,mod2;U[s.tab]
```

Specifying the load and execution address

This is done using the A (address) option and the B (begin at..) option. Both allow 32bit hex addresses following them, enclosed in square brackets.

```
LINK prog=part1,part2;A[FFFF4000]
```

Note that options requiring parameters (always in square brackets) cannot be set with LOPT since this only turns a flag on or off.

Using libraries

Any number of libraries may be used in a linking operation. A linker library is a collection of modules strung end to end in a file. Libraries are made with the LLIB command. At the end of the first pass the linker searches each library in turn and looks for modules in it with symbols still required by the modules being linked. If such a module is found, that module is also linked in. Care must be taken that library modules only forward reference modules coming later in the same library.

As an example, the program FILTER on the demonstration disc uses a library containing three modules. The source for the program is T.FILTER. The aim is to produce a program that can be *RUN and take parameters from the command line as follows:

*FILTER <infile> <outfile> <ASCII code>

Such as *FILTER T.FILE N.FILE 10
to remove all line feeds from a file before using it with the ADE plus editor. The file is assembled to a linker module L.FILTER. This is linked with the library L.ADELIB by the command:

```
LINK FILTER=L.FILTER/L.ADELIB
```

The slash precedes a list of libraries (there may be several separated by commas). The library consists of two modules, a filing system module and a maths module. These have been prepared from the sources T.FSLIB and T.MATHLIB. The assembler output the linker modules L.FSLIB and L.MATHLIB from these files then the linker librarian command LLIB was used to make the library:

```
LLIB L.ADELIB=L.FSLIB,L.MATHLIB
```

It is important to note that when a library module is identified as containing a missing symbol, the whole module is included in the linker output but not necessarily the whole library. Also note that a single linker module file will pass as a one module library. The LLIB command simply verifies the files are linker modules and concatenates them. The best user libraries will be lists of many short modules containing one or two routines each. This allows the inclusion of only the required code, making the output file as small as possible.

Conditional linking
The linker allows each module or library to be tagged with a conditional label. This label is the name of a symbol, which must exist. If the value of the symbol is zero then the module or library is not included. For example:

```
LINK
PROG=L.MODA,L.MODB/L.ADELIB,L.DEBUG[BUGFIX]
```

The last library, L.DEBUG, is only included if the label BUGFIX is non zero. This label can be easily altered to include debugging routines or not.

Chapter 2
2.1 ADE plus memory management unit Command level

ADE plus is initialised by the MOS command *ADE. Entering the ROM in any other way will not initialise the memory management unit correctly. During the initialisation stage the memory management variables kept on page 4 are set up. None of the units in the ADE plus system interfere with these variables, they are 'read only'. If you run a program that corrupts page 4 you will need to enter *ADE to reset the system. ADE plus then prompts for a command with the time and an arrow (see Chapter 1). This level is ADE plus command level. The following pages give full details of each command that you may use.

ASM

ASM <parameters>

The ASM command invokes the 65C00 series macro assembler. The remainder of the command line, which must be separated from ASM by at least one space, is passed to the assembler for interpretation. See Chapter 4. Alternative assemblers may be accessed from this command. Full details are given in the *ADE plus Technical Reference Guide* available from SYSTEM.

CLOSE

CLOSE

The CLOSE command closes any open files. It is the same as CLOSE #0 in BASIC. If a program under test has opened files and crashed, issue this command or the assembler or linker may not be able to open enough files. The ADE plus modules always close all files if a fatal error occurs. This has the side effect of halting an exec file, which in the event of a fatal error is probably a good thing.

COMMANDS

COMMANDS

Typing COMMANDS (or C.) displays a help page giving the full list of ADE plus commands with a guide to the expected syntax of the remainder of the command line in each case. This replaces the *HELP facility found on many ROMs since the ADE plus MMU commands do not function as * MOS commands and cannot be accessed outside the ADE plus command level.

DEBUG

DEBUG <parameters>

This command calls an advanced debugger that is written to work within the ADE plus system. The remainder of the command line, which must be separated from DEBUG by at least one space, is passed to the debugger for interpretation. Full details are given in the *ADE plus Technical Reference Guide*.

EDIT

EDIT (<parameters>)

The EDIT command first looks for an advanced editor in the ADE plus system. If one is found then the remainder of the command line, which must be separated from EDIT by at least one space, is passed to the advanced editor for interpretation. Full details of the interface to the advanced editor are given in the *ADE plus Technical Reference Guide*. If an advanced editor is not present then EDIT commands are passed to a small screen editor within the ADE plus MMU chip. This editor is fully explained in Chapter 3. All editors will accept * as a parameter meaning the file in the input buffer and all editors will leave the edited file in the input buffer so that it can be assembled directly, unless the file is too large to fit in the buffer.

GO

GO <hex addr>

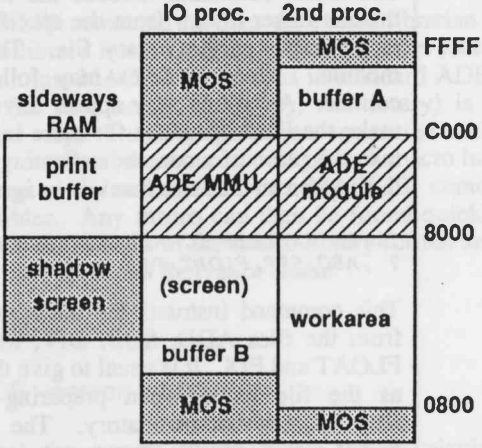
The GO command calls a machine code routine at <hex addr>. The registers are undefined on entry to the routine and need not be preserved, though the routine should exit with interrupts enabled and the decimal flag clear, as normal. The address is not preceded by an ampersand. On the second processor, for example, GO F800 will call the MOS command line interpreter from which ADE plus can be restarted by typing *ADE (not *GO 8000).

INPUT

INPUT <size in K>

This command sets the input buffer size. INPUT 0 will leave the input buffer unallocated. Normally, when a second processor is used, this command need not be issued since the

largest free memory area outside the main work space is given to the input buffer. Whenever input files are smaller than the input buffer they will be *LOADed otherwise they will be read a buffer-full at a time with OSGBPB. If your filing system's implementation of OSGBPB is efficient the system will run faster with a larger input buffer because the number of disc accesses will be reduced.



This diagram shows the full range of memory managed by the ADE plus MMU. With a second processor attached the input buffer will be at least 14K (buffer A). If buffer B is larger than 14K then the input buffer will be buffer B. Without a second processor buffer B is split between the input and the output and its size is variable, set by the INPUT and OUTPUT commands.

LINK

LINK <parameters>

The LINK command calls the ADE plus linker. The remainder of the command line, which must be separated from LINK by at least one space, is passed to the linker for interpretation (see Chapter 5). The linker is built into the ADE plus MMU chip and it is not possible to substitute a different linker. The linker may link the output from compilers and other assemblers. Full details of the linker data formats are supplied in the *ADE plus Technical Reference Guide.*

LLIB

LLIB <lib>=<mod>{,<mod>...} (-)

The LLIB command invokes the linker librarian which will make a linker library from the specified list of modules. <lib> is the name of the library file. There must be at least one module. More modules may follow, each preceded by a comma. A hyphen may appear anywhere in the line and will make the librarian pause for more input. The hyphen does not take the place of a comma separator. Any characters following the hyphen on the same line will be ignored. For example:

```
11:20 => LLIB L.MATH=ADD,SUB,DIV,MUL,LOGIC-
? ,ASC,STR,FLOAT,FIX
```

This command instructs the librarian to make a linker library from the files ADD, SUB, DIV, MUL, LOGIC, ASC, STR, FLOAT and FIX. It is usual to give the modules the same name as the file name when preparing modules for the linker librarian, but not mandatory. The library feature will work best if each file in the library contains only one or two routines. This is because the linker will include the whole of a module in its output if it finds a required symbol in the module's entry list. Libraries should contain groups of related modules. If modules A, B and C say all use common subroutines, put these in a module D and include D in the library *after* A, B and C. Also, make sure all the labels in the modules' entry lists do not get duplicated in programs using the library. For example, filing system routines could all start with "FS_". Also remember when using libraries that the linker distinguishes between lower and upper case. Stick to one convention for your labels. The suggested convention is that library routines are labelled in UPPER CASE, whilst program labels are normally in lower case but ENTry points begin with an Upper case letter. This will avoid any duplication of symbol names.

LOPT

LOPT (-)<opt>{,<opt>...}

The LOPT command sets and resets the linker option flags. A global set of these flags is held by the MMU in its work space. The linker copies these into its local variables when a LINK command is interpreted. Preceding an option by - resets the option (to false or off). Otherwise the option is set (to true or on). The options remain in force until changed by another LOPT command or reset when *ADE is typed. The current options are displayed with the STAT command. Some linker

options take parameters enclosed in square brackets. These options cannot be set with the LOPT command.

MLIB

MLIB <lib>=<source>

The MLIB command invokes the macro librarian to make up a macro library from a single source file. The source file should contain just MACRO definitions in standard ADE plus format. This file (and hence, indirectly, the library) is edited in the normal way. The library file is constructed with a catalogue giving details of the whereabouts of each macro in the file. This catalogue is loaded with the MACLIB command by the assembler. Any macro can then be found quickly by random access. Full details of the macro library format are found in the *ADE plus Technical Reference Guide.*

MODE

MODE <screen mode>

Change the current screen mode and re-calculate the buffer sizes. Do not change mode by any other means. The ADE plus editor selects mode 3 or mode 7 on entry, depending on whether you were using a 40 or 80 column screen and returns to the original mode on exit. The assembler and linker base their workspace calculations on details given by the MMU, which takes the top of free memory in the IO processor each time MODE is changed.

OPT

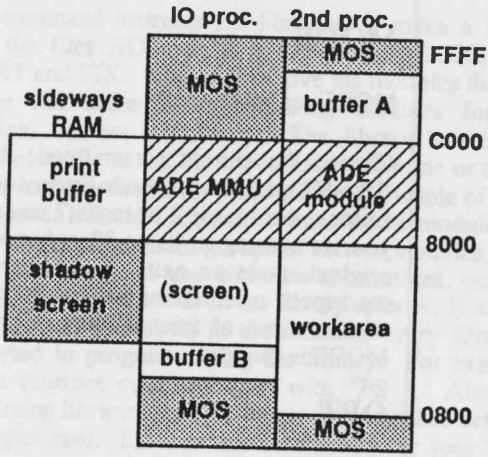
OPT (-)<opt>{,<opt>...}

Set the assembler options globally with the OPT command. Unlike the linker, all the assembler options consist of flags and can all be set permanently with OPT. The options consist of the letters A to Z. Preceding an option with a minus sign resets that option. (See LOPT). A full list of assembler options is given in Chapter 4.

OUTPUT

OUTPUT <size in K>

This command sets the output buffer size. OUTPUT 0 will leave the output buffer unallocated. Normally, when a second processor is used, this command need not be issued since the second largest free memory area outside the main workspace is given to the output buffer. Whenever output files are smaller than the output buffer they will be *SAVEed otherwise they will be written a buffer-full at a time with OSGBPB. If your filing system's implementation of OSGBPB is efficient the system will run faster with a larger output buffer because the number of disc accesses will be reduced.



This diagram shows the full range of memory managed by the ADE plus MMU. With a second processor attached the output buffer will be 14K (buffer A) if buffer B is larger than 14K. Otherwise the output buffer will be buffer B. Without a second processor buffer B is split between the input and the output and its size is variable, set by the INPUT and OUTPUT commands.

PRINT

PRINT <size>

Set the print buffer size. The only valid sizes are 0, which disables the print buffering, or a non-zero value which enables it. Print buffering will only be enabled if there is sideways RAM in the system that has not been protected (see PROT, UNPROT). All the unprotected sideways RAM is used as a

print buffer. The buffer is switched on and off through normal MOS routines once the buffer system is initialised. ADE plus initialises the system, so if you go into BASIC at a later stage the print buffer will still be active and BASIC will be able to spool its print out in the same way that ADE plus does. Full details of how the print buffer operates are supplied in the *ADE plus Technical Reference Guide*. The print spooling system uses the cassette file workspace on page 3 to hold its variables so it cannot be used at the same time as the cassette filing system.

PROT

PROT <rom id>{,<rom id>...}

The PROT command protects pages of sideways RAM from use by the print spooling system. <rom id> is a number in decimal between 0 and 15 referring to the respective page of sideways RAM. If any ADE plus module is in sideways RAM then that page is automatically protected at start up, but you could unprotect it with UNPROT, with dire consequences. Protected RAM is used for user programs and other ROM images.

RESET

RESET

Reset all the ADE plus MMU buffers to their initial values. This will depend on whether a second processor is connected and what screen memory is being used. All sideways RAM will be available to the print spooler except for RAM used to hold ADE plus modules. ADE plus does an implicit RESET command when the screen memory is changed with MODE. This command is useful if the buffers or ADE plus variables on page 4 have been corrupted.

STAT

STAT

The STAT command displays the values of the ADE plus MMU variables in a comprehensible form. It is fully described in Chapter 1.

TIME

TIME (<hh:mm>)(,<date>)

Alter the time and/or the date. The time must be entered as hh:mm (eg 10:36). The date may be entered in free format as a string of up to 16 characters. If only the date is required then it is entered preceded by a comma, such as TIME ,22nd July 87. There must in that case be a space between TIME and the comma.

UNPROT

UNPROT <rom id>{,<rom id...>}

Reverse the action of PROT and enable a page of sideways RAM for use by the print spooling system. The syntax is identical to PROT. It is possible to unprotect RAM used to hold ADE plus modules. The effect of this will become apparent during a printout when the code of the ADE plus module is eaten up by the spooling system. You could lose a valuable edit, so take special care using UNPROT. The STAT command tells you which slots (rom ids) are used by ADE plus modules. The ADE plus MMU is in the same slot as the linker.

ZASM

ZASM <parameters>

The ZASM command calls a Z80 assembler if one is found in the ADE plus system and passes to it the remainder of the command for interpretation. In fact any cross assembler, such as 68000 or 6803, could be used here as long as it was recognised by ADE plus. Details of how ADE plus recognises the two different assemblers are provided in the *ADE plus Technical Reference Guide*. The cross assembler must preserve the ADE plus variables on page 4 and should make use of the ADE plus memory management information and procedures.

2.2 Minimum abbreviations for ADE plus commands

AS.	**ASM**	**CL.**	**CLOSE**
C.	**COMMANDS**	**D.**	**DEBUG**
E.	**EDIT**	**G.**	**GO**
I.	**INPUT**	**L.**	**LINK**
LL.	**LLIB**	**LO.**	**LOPT**
M.	**MLIB**	**MO.**	**MODE**
O.	**OPT**	**OU.**	**OUTPUT**
P.	**PRINT**	**PR.**	**PROT**
R.	**RESET**	**S.**	**STAT**
T.	**TIME**	**U.**	**UNPROT**
Z.	**ZASM**		

2.3 Errors reported by the ADE plus MMU

1. ***Insufficient buffer space***
   Too little memory to allocate buffer

2. ***Out of range***
   Bad parameter in ADE plus command

3. ***Bad parameters***
   Bad or missing parameters in command

4. ***Module not present***
   Required ROM or image in RAM not found

5. ***Buffer allocated***

6. ***Too big***

7. ***Out of memory***
   Errors allocating buffer space

8. ***File not found***
   Unable to open library source file

9. ***Can't write to output file***
   Unable to open library output file

10. ***Bad file specs***
    Librarian unable to interpret command line

11. ***Missing ENDM***

12. ***Too many macros***
    MLIB errors

Chapter 3 ADE plus Editor

The ADE plus memory management unit recognises two types of editor. Firstly, an editor module that has been loaded into sideways RAM or is in a ROM socket or cartridge. This editor is probably a sophisticated mouse-based text and programming editor that you have added to your ADE plus system. This editor is called the *Advanced Editor* on the ADE plus status screen. Secondly, if no such editor exists, a basic text editor is built into the ADE plus MMU ROM itself. Although this editor only performs basic functions it has one or two special features for editing assembly language text which make it very usable. These include block move, copy and delete and a special GOTO LABEL key that will find the point in the text where a symbol has been defined. This chapter explains how to use the editor. You may, of course, prefer to use VIEW or another third party editor in which case this chapter may be removed from the manual.

3.1 Getting started

The ADE plus basic editor is called using the EDIT command from ADE plus command level. The command may take one of three forms:

EDIT <filename>
EDIT *
EDIT

EDIT <filename>
The named file is loaded into the memory work area. The file must fit in this space or an error occurs. The editor validates the text (see OLD below) and goes into editing mode so that the first page of text will be shown on the screen.

EDIT *
The editor attempts to load text into the work area from the INPUT BUFFER. Before using this command you should understand how the ADE plus MMU functions and what is meant by the input buffer. The text is transferred into the work area. The editor validates the text and all being well goes into editing mode with the first page of text displayed. An error occurs if the text is too big for the work area.

EDIT

The editor does not try to load any text. It tries to validate an existing text file in the work area (in case you left editing by mistake) and, if it can, will go into editing mode displaying the first page of the 'old' text that it found. You can issue a NEW command if this was not what you intended. Often the memory will look like a blank text file to the editor so after typing EDIT a blank screen appears with the cursor in the top corner. You can begin to enter text or you can press escape to go into editor command mode.

Editing mode

The normal mode for the editor is editing mode. The whole screen is used to display text. The cursor shows where the next text will be inserted. All the editing commands described below may be used. When you want to issue a command to the editor such as LOAD or SAVE, press the escape key to go into editor command mode.

Command mode

If the editor fails to find valid text when you issue an EDIT command it will go into command mode with the message No text. You can always get from editing mode into command mode by pressing the escape key. You can only leave the editor from command mode. When in command mode you can go back into editing mode by pressing the escape key.

Leaving the editor

You can exit from the editor back to ADE plus command level by typing the QUIT command or by typing the RUN command, in which case ADE plus will attempt to assemble and run the program in memory. Both of these commands attempt to copy the text from the work area to the input buffer so that it may be assembled directly. If there is no input buffer or the text is too big then it will not be possible to assemble the current program directly from memory. Remember ADE plus allows you to specify the size of the input buffer.

The text window

When the editor is entered the screen mode will be 3 or 7 depending on the mode in ADE MMU command level. In mode 3 the window shows 25 lines of 79 characters. In mode 7 only 39 characters are shown. The current line sideways scrolls if you type beyond the last column. The tab stops are pre-defined every 8 columns.

3.7 The command screen

When in command mode the editor prints status information at the top of the screen. This information includes some or all of the following:

Bytes free
Free space in the text buffer in characters (or No text)

File
The current file being edited or * if the input buffer has been loaded or No file

Insert ON or OFF
The editor is initially in insert ON mode (see below)

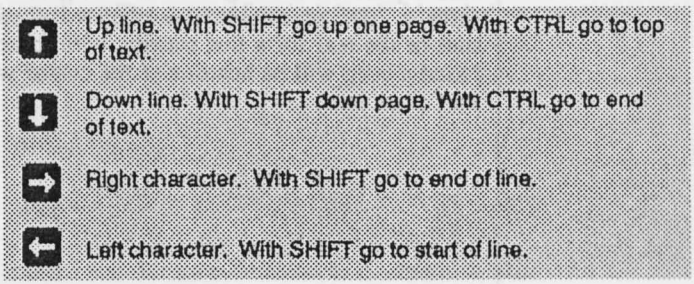
n markers
The number of text markers set (1 or 2) - not displayed if no markers set

hh:mm
The time (when the editor status was last printed)

The editor prompts for a command with ">". A MOS (*) command may be issued or any of the editor commands described below.

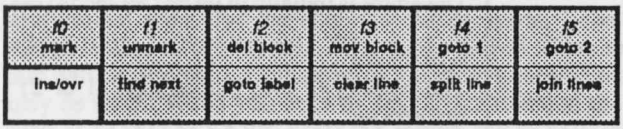
3.8 Editing text

The text is edited by moving the cursor round with the cursor arrow, shift and control keys and by using the function keys to perform special tasks. Normally the editor is in INSERT mode which means that text entered will be inserted before any remaining text on the line. The maximum line length is 128 characters. By pressing function key zero the editor toggles between insert and overtype mode. In overtype mode text entered will replace existing text on a line. You cannot overtype the end of a line. The cursor arrow keys are used as follows:

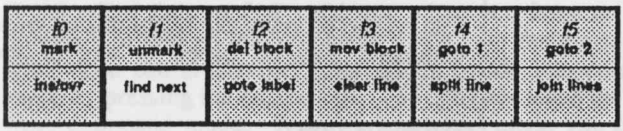


When moving up and down the editor keeps track of the current column. However, remember that a tab character counts as a single character so that when editing with tabs the cursor may move a little as you go from line to line. When moving to a new line the editor adds spaces to get to the correct column. When leaving the line the editor always removes trailing spaces.

The delete key deletes backwards, but cannot delete beyond the start of a line. Function key 9 deletes forwards on a line. Function key 8 inserts a space character in front of the cursor. The remaining function keys perform actions on whole lines.

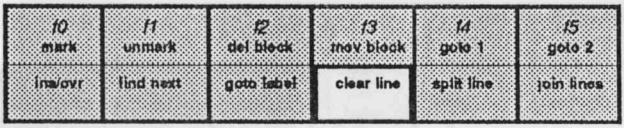


Toggle insert and overtype mode. The status screen (press ESCAPE) shows which mode is currently active.

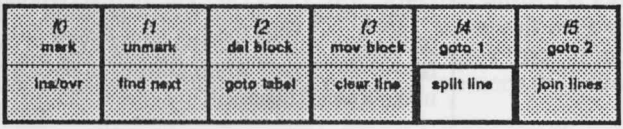


Find next occurrence of search string. The search resumes from the last occurrence, not from the current cursor position. A beep means not found, else the cursor is moved to the new string. See SEARCH command.

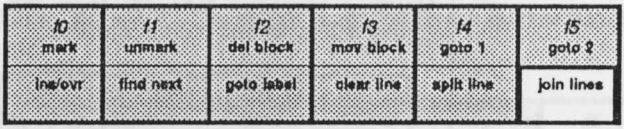
Go to a program label. The editor extracts a label from the cursor position looking for delimiting characters that are not valid as ADE plus assembler labels. It then searches from the start of the text for a line starting with these characters. If not found a beep will be heard, else the cursor will be moved to the start of the line.



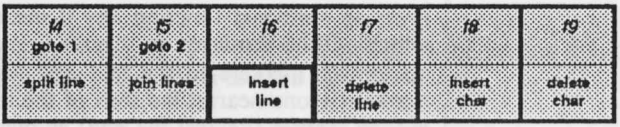
Clear to end of line. All the characters from the cursor to the end of the current line are deleted.



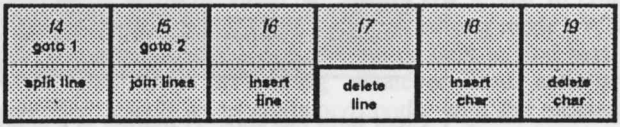
Split line at cursor position. The line is split. The cursor remains on the end of the line and the remaining characters will appear on the line below. If the cursor was at the end of the line then a blank line will be inserted.



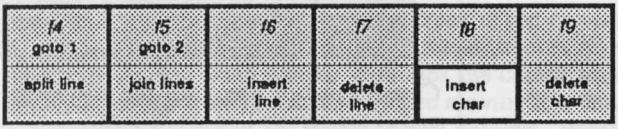
Join the current line with the one below. The line below is appended to the end of the current line.



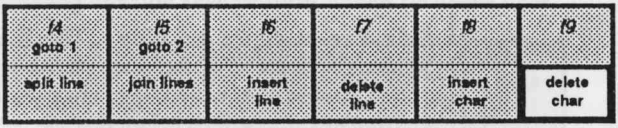
Insert blank line. The current line becomes a blank line. Lines below move down.



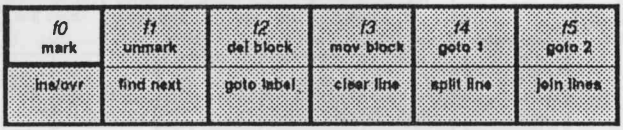
Delete line. The current line is deleted. Lines below move up.



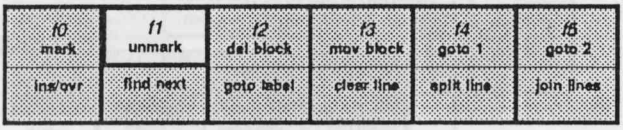
Insert character at the cursor position. The characters to the end of the line will be moved along even in overtype mode. This key is normally used when editing in overtype mode and desiring to make a small insertion.



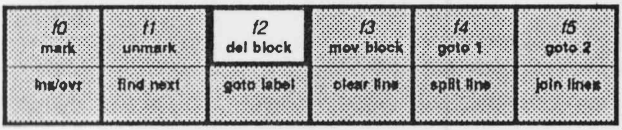
Delete a character forwards from the cursor position. You cannot delete the end of line character.



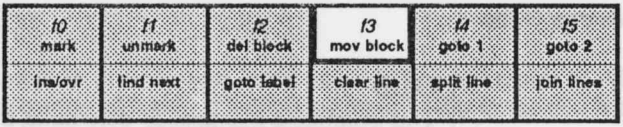
Pressing shift and function key 0 inserts a marker in the text which will be shown as an inverse up arrow. The editor allows up to two markers in the text and always refers to the first marker (ie one nearest the start of the text) as marker 1 and the second marker (nearest the end) as marker 2, regardless of the order that the markers were inserted.



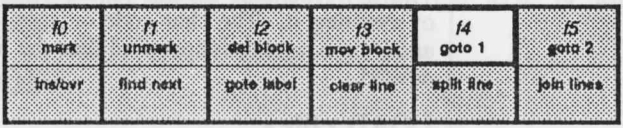
Clear both markers. A marker is a character that can be deleted by typing over it or using the delete key. Delete markers with shift f1 wherever they are in the text. This is done automatically before the text is saved to disc or the input buffer.



Delete the marked block of text. All text between the markers and the markers themselves are deleted. The cursor must be outside the marked block. The editor will warn with a beep if the cursor is in the block or there are less than 2 markers set.



Move the marked block. The block is moved to the current cursor position, same conditions as above. The block may be copied by pressing the COPY key. Pressing shift-f3 is identical to pressing COPY then shift-f2. Block move will only work if there was enough memory to perform the copy first. If move or copy cannot function because of lack of memory the editor will beep.



Go to marker 1. The cursor is moved to the marker nearest the start. If no markers set then the editor beeps.

Go to marker 2. The cursor is moved to the second marker. If less than two markers set then the editor beeps.

3.9 Editor commands

The following commands may be issued from command level in the editor. To get from editing mode to command mode press the escape key. Pressing the escape key whilst in command mode causes the editor to go back into editing mode. When the EDIT command is given, if the editor fails to find text in the work area after performing whatever load actions were specified, it will go to command mode, with the message "No text" in place of the "Bytes free" status. In such a case type NEW.

NEW

This command clears the text buffer. It may be reversed immediately afterwards by typing OLD.

OLD

This command attempts to determine the extent of text in the buffer. If the text is not valid then the editor will be in a No text state and you will need to type NEW. Note that this editor considers text containing line feeds to be invalid. A line feed filter program is included on the demonstration disc. To run this program, have it in the disc drive and type

*FILTER <file> (<new file>) (<char code>) (<new char>)

The new file will default to the same name as the old file overwriting it. The char code defaults to 10 (filter line feeds) and if no new char is specified then the line feeds will simply be deleted.

LOAD <file>
L <file>

Load the specified file from disc into memory. The file must be small enough to fit or a Too big error will occur. The editor validates the file and puts the cursor at the start. Press escape to go into editing mode and change the file.

SAVE <file>

Save the file under the name specified on disc. If the file exists then the message "Replace? (Y/N):" will appear. Press Y or N.

CLEAR

Clear markers from the text

SEARCH <text>
S <text>

Search the text from the start for a line containing <text>. The text need not be delimited, in which case it starts with the first non blank. A tab character may be included by using the double bar symbol, which the editor will translate into a tab. To include leading spaces delimit the string with "quotes". A double quote character will cause a quote to be placed in the quote string, for example:

S "LDA #""A"""

The current string is printed on the status. If no string is specified the editor searches for the next occurrence of the string. If the string is found, the editor goes into editing mode with the cursor at the string. If not, the message "Not found" is printed.

QUIT

Attempt to save the text in the input buffer and return to ADE plus command level. If there is no input buffer or the text is too big, the message "Quit? (Y/N):" is displayed and you can press Y or N as you please.

RUN

Save the text in the input buffer. If this can't be done the command aborts with the error "Can't run". Then call the assembler with the command line ASM =*,G. This will attempt to assemble the text and run it. Make sure your text is at a suitable address to avoid corrupting the ADE or MOS variables.

MODE

This command toggles the editor screen between Mode 3 and Mode 7. Only use Mode 7 if you have a poor quality monitor.

3.10 User defined keys

The function keys f0 to f9 are set to deliver the text defined with *KEY n when they are pressed together with SHIFT and CTRL.

Chapter 4 The Macro Assembler

4.1 DESCRIPTION OF THE ASSEMBLER

The assembler is designed to facilitate assembly language programming on the BBC series of microcomputers. The assembler contains an extensive set of pseudo-ops that cater for every conceivable programming requirement. Standard 65C00 series mnemonics and address mode syntax are used. The source program may reside on disc but will be loaded into memory if sufficient RAM is available. The size of the source program is not limited. The assembler uses main RAM for its workspace. This workspace holds the symbol table, the file buffers, the macro text and the assembler's own variables. The assembler is not used to create the source program. A text editor must be used to do this. Any text editor may be used that produces a standard ASCII text file. Word processors that embed control and text formatting commands in the text will not work. VIEW will work if *format* and *justify* are turned off. The assembler WILL recognise VIEW **rulers** and use them to format the source code listing on pass 2. Otherwise only text is accepted.

Assembly is initiated from the ADE plus MMU prompt by the ASM command, or by the editor run command, as described below. The assembler translates the source program into either a machine code file that can be directly run using the *RUN command or a linker file of relocatable hexadecimal code that can be linked with other files to produce the machine code object file. During pass 1 the assembler generates a symbol table containing the numerical values of all symbols defined by the user. The length of each instruction is determined and any forward references noted. These are always assumed to be non-zero page addresses so that the assembler will generate three byte instructions where it might have generated a two byte instruction if no forward reference was encountered. Thus all zero page labels should be defined before they are used. Macros are read into main memory during pass 1. A macro must be defined before it is used so that the assembler can work out the correct number of bytes that each use of the macro will entail. Macros are stored in memory in text form so they should be defined as briefly as possible.

A second pass re-reads the source program and generates the output file substituting the actual numerical values of all symbols fully defined. If the output is a linker file then a list of addresses to be relocated and a list of external (undefined) symbols and the places where they occur is also output. A listing will also be generated in pass 2 and the assembler will flag any

errors that it finds in the source program. The assembler does NOT detect *logical* errors in the program so a successful assembly does not mean the program will run correctly.

The source file may be split up into *include* files and *chain* files. Include files are inserted between two lines of another source file, one that contains an INCLUDE statement. This 'parent' file may not itself be an *include* file. A *chain* file follows on from the last statement of a preceding source file that CHaiNed it. This feature is included for compatibility with ADE versions 1 and 2. Newcomers to this system should use *include* files as they offer the flexibility of a 'control file', that is a parent file or main source file which is simply a list of INCLUDE statements together with all the conditional flags to be set for the assembly.

Aborting an assembly

An assembly may be terminated at any stage by pressing ESCAPE. When this is detected all files opened by the assembler are closed.

The assembler works with any filing system that supports random access through OSGBPB, catalogue entries through OSFIND and multiple file access. Usually the filing system used will be disc (DFS, ADFS) or econet.

The Assembly Command Line

The assembler is started from an ASM command after the ADE plus prompt. The ASM command is always followed by at least one space. The rest of the line specifies the file to be assembled and various options supplied to the assembler. This part is called the assembler command line. Thus the general format for starting an assembly is:

hh:mm => *ASM {object}{/listing}=source{[,opt]}{/ans[,ans]}*

or

hh:mm => *A {object}{/listing}=source{[,opt]}{/ans[,ans]}*

User input italicised

The meaning of each item on the assembly command line is as follows. Items in curly brackets are optional. Items in square brackets may be repeated. *Object* is the object file, or output file, which may be in relocatable or absolute form according to the contents of the source file. *Listing* is a file to contain the listing or error messages output on pass 2. The inclusion of this file does not generate a listing. That is controlled by the LST pseudo-op. If no listing file is specified then the listing goes to

the screen or printer according to which options are used. A '/' must precede the listing file name and is not part of the name. The object file name may be omitted and in such a case the source code is only scanned for errors. A listing file may be generated when no object file is being generated. An equals sign always follows any object or listing files (even if neither are present). The equals sign separates the *output* parameters from the *input* parameters. *Source* is the name of the first source file to be assembled. This must be present. The source file may contain INCLUDE files and a CHN file. *Opt* is an assembler option. If options are specified they follow the source file name and each option is preceded by a comma, including the first option. *Ans* is an answer to a QUERY statement in the source program. Any number of answers may be specified and each answer is separated by a comma. An answer is an expression which may contain symbols but not contain forward references or externals. The first answer is preceded by a '/'. The first answer is supplied to the first QUERY and so on. When the assembler runs out of answers it asks for them from the keyboard. (See QUERY). If an invalid answer is detected the message *Re-enter:* is displayed and you must type in the answer on the keyboard.

If a command line is not correctly interpreted the message:

Invalid command line

is displayed and a prompt for a new line issued. This will usually occur if no source file is specified or the equals sign is absent. If the source file does not exist the error *file not found* will be reported.

The symbol * may be used for the source file when the source file is the text currently in the input buffer. This may be a repeat of the last assembly or a file left there by an ADE plus editor.

4.3 Assembler Options

A number of options may be entered on the command line. These options can also be entered in the source code using the OPT pseudo-op. Each option is a letter A..Z (or a..z) and controls the value of one of 26 flags in the assembler variables. Each time the option is specified the flag is toggled on or off. Only some of the flags affect the behaviour of the assembler. The unused letters are reserved for future expansion.

Option C - conditionals

Conditional listing. Statements that are conditioned out of the assembly will not be listed if the C option is in effect. These statements normally have an S (skipped) in column 18. Statements containing errors will always be listed regardless of the options.

Option E - extended error messages

The assembler will produce full text error messages and a pointer to the part of the source statement giving the error. Additional messages are generated for syntax errors and linker rule violations.

Option F - Fix ASCII size

The assembler will fix ASCII constants as 16 bit values after OPT F. This means that 'XY' could be specified as a word value for example. Note however that after OPT F some macro substitutions may not work as expected,

```
IF '@1'='
```

would have to be replaced by

```
IF '@1'=""
```

Also, using 16 bit ASCII the second quote must ALWAYS be specified whereas it is optional with the default 8 bit.

Option G - go

The assembler will attempt to run the object program provided the assembly was an absolute assembly and that there were no errors. The assembler will obtain the object code from the output buffer if possible, otherwise the object file will be *RUN.

Option H - halt exec file

The assembler will close the exec file if an error occurs so that bad output is not linked or run.

Option L - listing

Force a listing. A listing is generated regardless of any LST OFF statements in the source code. The assembler still distinguishes between LST ON and LST FULL statements.

Option N - no listing

No listing is generated regardless of LST ON statements in the source code.

Option O - omit operating system labels

The assembler uses a list of default symbols as defined in the operating system as well as TRUE and FALSE. These are "assembled" before pass 1 unless OPT O is specified in the assembler command line or globally in ADE plus OPT command. It is impossible to specify OPT O in the source program.

Option P - printer

Direct output to the printer on pass 2. Output stops after the symbol table or a fatal error. Any listing file is unaffected.

Option Q - syntax check

The assembler does not generate any output file on pass 2. This option is the same as omitting the object file name from the command line and will probably be most useful in an OPT statement in a source that you are not going to compile immediately.

Option R - reduced instruction set

The assembler will use only the standard 6502 instructions. 65C00 series instructions will be flagged as illegal op codes unless you provide a macro for them. Thus for 6502 work PHX may still be used if you specify a macro (typically TXA, PHA). The output file is identical to the 65C00 format so you may link different modules together, some of which contain only 6502 instructions.

Option S - error summary

The assembler will generate a summary of error messages at the end of pass 2. Each error message will give the file name, the line number within the file and the text of the extended error message.

Option U - upper case translation

The assembler will translate all symbols into upper case. This gives compatability with ADE versions 1 and 2.

Option W - wait after error

The assembler will pause after an error, listing the extended error message and the text *Press any key*.

.4 Example Command Lines

```
=T.MYTEXT
```

Perform a syntax check on the file T.MYTEXT and possibly generate a listing if LST ON is specified. The listing goes to the screen.

```
=T.MYTEXT,N
```

Perform a syntax check but produce no listing except for lines containing errors.

```
/ERRORS=T.MYTEXT,N
```

Perform a syntax check and send errors to the file 'ERRORS' as well as to the screen.

```
MYCODE=T.MYTEXT
```

Assemble T.MYTEXT and produce an object file MYCODE.

```
MYCODE=T.MYTEXT,L,P
```

Produce an object file and force a listing of all the source to the printer.

```
MYCODE=T.MYTEXT,L,P/&1900,256+FRED,&65
```

Assemble, list to printer and use the response &1900 for the first QUERY, 256+FRED for the second QUERY and &65 for the last query.

```
MYCODE/LISTFILE=*,L,C/&1900,YES
```

Assemble from memory and send a listing to the file LISTFILE. Do not list sections of code that are conditioned out. Answer the first two QUERYs with &1900 and YES.

4.5 General Considerations

Use a listing file with a hard disc or RAM disc based filing system. On floppy based systems it will slow down the assembly. Double density systems may have enough room on disc but large programs will soon fill a single density floppy disc. A listing file with the N option is useful to capture all the errors. The P option sends all the listing to the printer but you can be more selective using the LLST

pseudo-op. Commands for assembly and linking may be put in an EXEC file. If ADE plus does not recognise a command it tries to execute the file of the given name.

4.6 Format of the Source Program

<label> <opcode> <operand> <comment>

The source program consists of one or more files of ASCII characters. Each file consists of a number of lines. A line is a sequence of characters ending in ASCII carriage return (&0D), ASCII CRLF (&0A,0D) or the physical end of file. The assembler contains directives to generate text data in the object file with or without bit 7 set. Each line is divided into four fields: the label, the opcode, the operand and the comment. Every field is optional though certain opcodes and pseudo-ops require certain operands and possibly require a label. The simplest line is a blank line. A comment line is allowed, and a line beginning with an asterisk is a comment line. Fields are separated by 'white space' which means one or more space characters or tab characters. The tab character is expanded to spaces in the listing and initially tab stops are set every 8 columns but this may be overriden by including a *View* format ruler in the text. Rulers will not be listed. Only the tab stops on the ruler apply, counted from the left hand margin. A statement may be simply a label. An operand with no opcode is not allowed. The label must be the first item on the line and must not be preceded by any white space. Statements must not contain line numbers (except as comments).

4.7 The Label

The label is usually-an optional field. A label must not begin with a number but may contain any of the following characters:

A..Z, a..z, 0..9, full stop, unserscore, query (?)

Labels may not contain embedded blanks. A label that begins with a colon is a local label. A global label may also be defined with the ENT pseudo-op. The assembler differentiates between lower and upper case in labels unless the U (uppercase) option is specified. The underscore character is a valid character and is always included as part of the label. A label may not start with a numeric character. The label may be any length. Note that the colon in front of a local label is part of the label name and must be used whenever the label is referenced in the operand field. Labels may be redefined under the following conditions: a label defined with the = pseudo-op may be redefined at any point with another = statement. These labels are normally used for counters in macros and so on. A local label may be

redefined provided it is in a different BLOCK. No other labels may be redefined.The length of the label is only limited by the length of the statement. In order to avoid redefining labels in macros a special parameter is provided within the scope of the macro called @0. @0 is a five digit decimal number which is incremented each time a macro is invoked. Like the user's parameters (@1 to @9), @0 is saved when macros are nested. It should be appended to labels that are used in macros. Of course a label can be passed to a macro as a parameter.

The assembler keeps a counter called the 'location' counter which is the address at which the instruction will be assembled. In a linker file the location counter is the offset from the start of the section (except in ASECT). The label is normally given the value of the location counter at the start of the line. Thus if a label is defined on its own it simply receives the value of the location counter. Another label on the next line will have the same value. The EQU,= and QUERY pseudo-ops redefine the value of a label to the value of the operand field.

When a label is used in an operand it is referred to as a symbol. The assembler keeps all its labels, macros and other items needed to assemble the source in a table called the symbol table. When the symbol is referenced in an operand the assembler substitutes the value from the symbol table. All absolute symbols with values greater than 255 can be defined anywhere in the program. Zero page symbols must be defined before they are used. A third type of symbol is an external one. When an external symbol is referenced the assembler makes a note of the place in the object program that the symbols value should go. The linker then substitutes the value into the program. If you put a zero page label in an operand before it is defined, or any external label, the assembler will always generate a long form (3 byte) instruction. It is good practice to include zero page symbols at the start of the program in a DSECT. All unknown symbols in pass 2 will be flagged as errors.

4.8 The Opcode

The second field of each statement is the opcode. It is separated from the label by white space. If there is no label there must be at least one space or tab before the opcode. The same characters used in labels are used in opcodes. Firstly the assembler tries to match the opcode to one of the standard 65C00 series mnemonics, plus the alternative mnemonics for some branching instructions allowed with this assembler. At this stage only three character opcodes are considered and they are converted to upper case. If the opcode is not recognised the assembler searches for a pseudo-op, again converting the characters to upper case. If the opcode is not a pseudo-op it must be a macro. The assembler looks at the macros defined to date and then, if it is not among them, in the current macro library. If the macro is found in the library it is loaded into memory. If there is no specified macro library or the opcode is not in it the assembler gives up and flags an unknown instruction error in pass 2. This sounds a lengthy process but most statements will be 65C00 mnemonics and so they will be recognised immediatly. Macro names can be mixed upper and lower case and the case is significant unless the U option is in effect. The assembler recognises tokens for the 65C00 series opcodes, in the range 128 to 255. Full details are given in the *ADE plus Technical Reference Guide.*

4.9 The Operand

The operand field of the statement is required in a precise format by each mnemonic or pseudo-op. If the format is wrong the assembler flags a format error in pass 2. The operand contains an expression formed out of constants, symbols and operators plus syntax information such as an indication of which addressing mode is to be used. The standard addressing modes are used, as used in the BASIC mini-assembler. The arithmetic operations have no precedence except for the use of brackets and are evaluated from left to right. The only arithmetic error flagged is division by zero.

4.10 The Comment

The comment field is used to document the program. The assembler prints the comment in the listing but otherwise it is ignored. A comment starts with a semicolon or back slash and ends at the end of the line. The assembler also recognises statements whose first non-blank character is an asterisk as comment lines and ignores them.

4.11 Expressions

In the description of the pseudo-ops that follow, the term <expr> is used to denote an expression. Expressions consist of one or more *terms* separated by arithmetic operators. Each term is a constant or symbol which may be signed, or it is another expression in brackets (...). Expressions are long (16 bit) or short (8 bit, less than 256). If the assembler finds a forward reference in an expression then the expression is long even if its value turns out to be less than 256. The length of the expression is used to generate long or short form instructions. Expressions between &FF80 and &FFFF generate long instructions but can be used as negative values -1..-128 in operands that should be one byte long.

4.12 Constants

The assembler recognises four types of constant: ASCII constants and three types of numeric constant. Numeric constants may be binary, decimal or hexadecimal. The first character of a constant shows what type of constant it is.

Decimal numbers

Decimal constants consist of a sequence of ASCII digits 0..9. These represent the integer values 0..65535. Overflow beyond 65535 is ignored so the value mod 65536 will be used.

Hexadecimal numbers

Hexadecimal constants represent numbers in base 16 and consist of the digits 0..9, the letters A..F and an identifier $ or &. The $ or & character precedes the hexadecimal digits. The value is an integer in the range &0000 to &FFFF. Once again overflow is ignored. The $ or & alone represent the value zero.

Binary numbers

Binary numbers are numbers in base 2. They are used most frequently for bit masks. The first character must be % then the digits 0 and 1 may be used. If a digit bigger than 1 is used a bad number error will be flagged. The percent sign on its own represents the number zero.

ASCII constants

ASCII constants represent the ASCII code for a single character. They are preceded by a single or double quote and may optionally be followed by the same quote. A quote alone will assemble a space character (&20).

4.13 Strings

Some assembler pseudo-ops assemble a sequence of ASCII characters, called a string and denoted by <string> in the pseudo-op descriptions. The start and end of a string must be delimited in the source code. This assembler allows any character except ^ and | to be used as a delimiter. The string must start and end with the same character, and this character is not assembled as part of the string. Within the string the 'escape' characters ^ and | are used as follows: The character following ^ is assembled with bit seven set. The character following | has 64 subtracted from its ASCII value making it a control character. Two up arrows assemble an up arrow and two double bars a double bar. Three up arrows assemble an up arrow with bit seven set and three double bars assemble a 'control-double-bar'.

4.14 Reserved words

Only the letter A on its own is treated as a reserved word in this assembler. A is used to denote the accumulator and may be used as an operand. This is optional so that LSR is the same as LSR A.

4.15 Arithmetic Operators

The assembler supports the following arithmetic operators, which must be used as the sole separator between two terms of an expression. Spaces are not allowed in expressions, except as an ASCII constant.

+,-,*/,#	add,subtract,multiply,divide and modulus
&,!	bitwise AND,OR
=,>,<,>=,<=,<>	logical operators
>>,<<	shift right or left (eg 32<<2 is 128)
~,-,+	unary NOT, minus, plus (default)
?	unary symbol type

The asterisk may be used as a term in an expression and it denotes the current value of the location pointer. It is a symbolic term so that in a relocatable assembly an expression using * as a term is a relative expression.

The only unusual operator is ?. ? <expr> or ?<symbol> returns the type of a symbol or expression. The possible values are 0 for absolute, 128 for relocatable and 64 for external. External symbols have a value of 0 and relocatable symbols may be less than 256. The assembler uses the type value to correctly generate long instructions in these cases. You may wish to do the same in macros.

4.16 Relocatable and external expressions

Expressions are categorised into three types. Absolute expressions are the normal type produced in none linker assemblies. When a linker module is being assembled the expressions may contain a relocatable part *or* an external part. All operators may be used in absolute expressions but relocatable and external symbols can only be added or subtracted from an expression. Only one external reference may appear in an expression. If it does, the listing will be marked with a " in the hex code corresponding to the external expression. 0000" signifies a simple external reference but the external may be added or subtracted from an absolute expression and that may use any of the operators. The linker does not check for arithmetic overflow when adding the external to a word value. For example:

```
FARCALL EXT
        JSR FARCALL     ; output is 20 00 00"
        LDA FARCALL+6   ; output is A9 06 00"
```

When an expression uses relocatable symbols, its value is flagged by a '. If several relocatable symbols are added and subtracted the result may be absolute. For example:

```
DAT1    DB  4
DAT2    DB  6
        LDA DAT1        ; output is A9 00 00'
        LDA DAT2        ; output is A9 01 00'
        LDA DAT2-DAT1   ; output is AA 01
        LDA DAT1-DAT2   ; output is A9 FF FF
```

If an expression attempts to add twice the relocation constant , or use more than one external reference, then a L (linker violation) error will be flagged.

High and Low bytes

This assembler uses **>expression** to denote the *low* byte of the expression (<expr> MOD 256) and **<expression** to denote the *high* byte (<expr> DIV 256). For example:

```
    LDX #>string
    LDY #<string
    JSR print_string
```

4.17 The Assembly Listing

The listing produced in pass 2 of the assembly consists of three parts: the program listing, the symbol table and a report on the assembly. The listing of each statement starts with the first error that was flagged for that statement or three blanks followed by the contents of the location counter in hexadecimal. If the code is relocatable the location counter will be followed by an apostrophe to show that its final value will be adjusted. If the location is absolute it is followed by a colon. A space follows and the object code is printed in hexadecimal. Normally only the first three bytes of object code are printed but a list option allows all bytes to be printed on successive lines. This only affects data definitions because an instruction cannot be longer than three bytes. The line number of the current file is listed next in decimal. This can be suppresed by a further list option. The line number is reset to 1 each time a new file is included or chained. This number is for reference purposes only.

The listing is set out on pages. The format of each page is controlled by the PAGE and WIDTH pseudo-op. The default width is 132 characters. The default page length is 66 lines. This includes 55 statements and the headers and footers. The headers and footers can be turned off with a list option, in which case the assembler prints continuous statements with no pagination. Line feeds are sent to the printer at the end of each line; unless, for example, you have set the printer ignore character to ASCII &0A using *FX 6,10. Formfeeds are sent at the foot of each page unless you set the no form feed list option, in which case the footer is made up of blank lines. The header consists of the file name followed by a title if specified by TTL or a default copyright message. This is followed by the date if available and the time if available. The time is the time at the start of the assembly in hours and minutes and will help sort out which is the latest listing. Finally, aligned in the right hand margin, is the page number in decimal. The value of the location counter is omitted on EQU,=,QUERY and conditional pseudo-ops. Instead the operand for the statement is printed followed by an equals sign. Statements that are conditioned out are marked by an S in column 18 unless conditionals are not to be listed, in which case the statements are omitted as they generate no code. In a repeating conditional only the first occurrence of the statement will be listed for a DO though all the code will be listed if the full code list option is in effect. For a REPEAT or WHILE the whole passage is listed over and over.

List options

Option	Meaning	Value
0 -	No pagination	1
1 -	No line nos.	2
2 -	No formfeed	4
3 -	No symbol table	8
5 -	All object code	32

These symbolic names are part of the additional symbol table held in the ROM that will be used unless OPT O is specified. Example:

```
LISTO  8 + 32
```

4.18 Assembler Pseudo-Ops

Assembler pseudo-ops are listed according to the category in which they fall. Each pseudo-op is used like a normal opcode in the source line.

4.19 Assembler Directives

Assembler directives are general purpose pseudo-ops which control the value of various assembler flags and variables. In addition, assembler directives differ according to whether the assembly produces a linker file or an executable file. Linker files have a MODULE statement before any code generating statements. Executable files have an ORG statement before any code generating statements. The common feature of these pseudo-ops is that they do not generate any code.

ORG

(<label>) ORG <expr> (;comment)

The ORG directive establishes the value of the location counter in an assembly producing an executable object file or in the absolute section of a linker file or in a dummy section. In a non-linker assembly ORG sets the load address in the object file's catalogue entry. The expression must be absolute and not contain forward references, so that the exact value of the location counter is known on pass 1. If the load address is to be different from the code origin a second ORG may be used at the end of the program because the value put in the catalogue is the value specified by the last ORG on pass 1. Each time an ORG is encountered in a non-linker assembly, the object file's execution address is also set to the value specified, implying that execution begins at the first byte of the file. If this is not the case then a different execution address may be specified by using the EXEC pseudo-op or the END statement. Remember to put the EXEC statement after the last ORG in the program. The MSW pseudo-op is also used to establish the exact load address on systems with second processors. An ORG statement in a linker file is only valid in a DSECT or ASECT. DSECTs are valid in both linker and non-linker files and the ORG specifies the value of the location counter for the dummy section. ASECTs are only valid in linker modules. ORG sets the value of the ASECT location counter which is the same as that described above for non-linker files. If an ORG is used anywhere else in a linker file it is ignored and produces a warning message.

```
; Examples
            ORG     &2000

start       QUERY   Start address
            ORG     start
```

EXEC

(<label>) EXEC <expr> (;comment)

The EXEC pseudo-op defines the address of the first byte of executable code in a non-linker assembly. The expression *must* be absolute and contain no forward references. The EXEC statement may be the last statement in the program. The value specified is written into the catalogue entry for the object file. If more than one EXEC is specified, or an address specified with the END statement, then previous EXEC statements are ignored. Note that an ORG after an EXEC statement means that the EXEC will be ignored. An EXEC statement in a linker file will be ignored and produce a warning message. In the case of a linker assembly the execution address for the program is set by a global symbol SYSEXEC, which you may define anywhere in one of the modules being linked.

```
;Example
            ORG     &2000
            EXEC    start
            DATA    1,2,3,4
start       JSR     init
```

MSW

(<label>) MSW <expr> (;comment)

The MSW pseudo-op defines the high order address of the object file in a non-linker assembly. Although the 65C00 series use 16 bit addressing, the operating system allows files to have a 32 bit address. The high order bytes specify whether the file is loaded in the IO processor or the second processor. The default value for MSW is 0. If MSW -1 is used then programs will always run on the IO processor. When writing ROM software use MSW -1 if you are loading the program into sideways RAM for testing. MSW is ignored in linker assemblies.

```
; Example
            ORG     &2000
            EXEC    start
            MSW     &FFFF
```

DSECT

(<label>) DSECT (;comment)

The DSECT directive is used to define an area of memory, such as page zero, a data table or jump table, without actually generating any code in the object file. DSECT stands for dummy section. DSECT has its own location counter which may be set anywhere by an ORG following the DSECT statement. At the end of the DSECT the previous location counter is restored. The first DSECT will have an implicit ORG of zero. At the end of the DSECT its location counter is preserved so that the next DSECT will be continuous to it. Use DSECT rather than EQU to define variables because it allows an extra variable to be inserted in a list very simply and shows explicitly the amount of storage required by each variable. DSECTs are allowed in all assemblies. A DSECT is terminated by a DEND in a non-linker assembly or by the occurrence of ASECT or RSECT is a linker assembly. Symbols defined in DSECTs are absolute symbols.

```
;Example
            DSECT
            ORG     &70
var_ptr     DS      2
value       DS      2
            DEND
```

DEND

(<label>) DEND (;comment)

The DEND directive signals the end of the current DSECT and the resumption of code generation from the previous address saved when the DSECT was started. If your program produces no output then look for a missing DEND. When assembling a linker module, any use of ASECT or RSECT causes an implicit DEND.

ASECT

(<label>) ASECT (;comment)

The ASECT directive is only used in linker modules. It specifies that the linker should load the code following at an absolute address, which must be specified by using ORG within the ASECT. The linker will produce a separate output file for each ASECT encountered, whose name is derived from the main program name. This is because the BBC MOS filing system conventions do not allow scatter loading from a single

object file. Conveniently it allows the use of overlays by having one main program (the RSECT) and a number of overlays loading in at the same absolute address. This is explained in section 1.5 *Advanced linker techniques.* An example of an ASECT would be a jump table that is to be at the top of a ROM such as the operating system ROM. ASECT stands for Absolute SECTion. The ASECT is terminated by an RSECT. Symbols defined in ASECTs are absolute symbols.

```
; Example
          ASECT
          ORG     &C00
          HEX     "05F5DA66775A09"
          RSECT           ; resume rsect
```

EMBED

(<label>) EMBED <expr> (; comment)

The EMBED statement is used to embed a section of code inside a program which should be assembled as if it were at a different address. An embedded section is in effect data which must be moved to the correct address at run time before it will execute. The code in an embedded section is absolute code and the <expr> following EMBED must be an absolute expression containing no forward references. The embedded code is terminated by RESUME or an ASECT, DSECT, RSECT or ORG statement. Be careful when programming with embedded sections. The label attached to EMBED will be the old location pointer. The first label following it will be the new location pointer which will initially be the value of <expr>. This example shows a small routine to be moved to zero page and executed there:

```
; Example:
; Search memory buffer for byte in Y reg and leave
zero page
; pointer at &71 pointing to byte found.  Rapid if
buffer is
; large.

fast_fward ldx    #end_code-code_start
:1         lda    code,X
           sta    code_start,X
           dex
           bpl    :1
           sty    cp_byte+1
           jsr    code_start
           rts
code       EMBED  &70
code_start lda    buf_start      ; patched
cp_byte    cmp    #0             ; patched
           beq    :rts
           incw   code_start+1
           bne    code_start
:rts       rts
end_code   RESUME
```

RESUME

(<label>) RESUME (;comment)

RESUME marks the end of an embedded section. If RESUME is specified without EMBED a warning is given. Similarly, RESUME will be done automatically by the assembler if the source code indicates a change of location pointer value through ORG or a change of section type. In this case the assembler will issue a warning that it is performing a RESUME operation.

RSECT

(<label>) RSECT (;comment)

The RSECT directive is used only in linker modules. It specifies a Relative SECTion and is the default section at the start of a linker assembly. Code generated in an RSECT is relocatable and all symbols defined (as labels) are relative symbols. The linker will attempt to string all RSECTs end to end and produce the smallest executable object file. One RSECT may include a special symbol SYSEXEC which will be the execution address of the linked program. SYSECEC may occur in an ASECT or DSECT but normally speaking it will be an address in the code segment which is usually the RSECT. As SYSEXEC is a global symbol it may only occur once in all modules being linked. The end of the RSECT is specified by an ASECT. DSECTs may be embedded in an RSECT since they produce no output. Similarly an RSECT may contain an

EMBEDed section of absolute code. See ASECT, ORG, DSECT, EMBED, RESUME.

END

(<label>) END (<expr>) (; comment)

The END directive specifies the end of the assembly source program. Any lines following it are ignored and can be used to document the module. If an expression is specified it is used as the EXEC address in a non-linker assembly. If END is specified in an include file or a WHILE or REPEAT loop then a warning is issued and the END statement ignored.

```
; Examples
            END
            END     start  ; set exec address
```

MODULE

(<label>) MODULE <name> (; comment)

The MODULE directive specifies that a linker file will be produced as the output from the current assembly. There should be one MODULE statement per source program and it must occur before any code generating statements since the default is for a non-linker assembly. The module name is used by the linker when producing the map and cross reference files. The MODULE statement implies an RSECT but can be followed by a directive indicating any other type of section. The name must be between 1 and 32 characters, contain no spaces and is not enclosed in quotes.

```
; Example
            MODULE DATA
```

EQU

<label> EQU <expr> (; comment)

The EQUate pseudo-op is used to assign a value to a symbolic label. The label must be present on the same source line and the expression must evaluate on pass one. If the expression is absolute then the symbol defined will be an absolute symbol. If the expression is relative the symbol will be relative. Note that an expression containing an even number of relative symbols may be an absolute expression. The use of forward references will produce an error. Symbolic labels should be used in place of numeric constants wherever possible since they make

programs easier to maintain. The operating system calls are already predefined using equates but may be redefined using '='.

```
; Examples

low             EQU     12
hi              EQU     low+20
here            EQU     *       ; PC value
zero,null       EQU     0
```

=

<label> = <expr> (;comment)

The = pseudo-op allows the value of <label> to be defined (like EQU) or redefined. = should be used with caution since it does little type checking. Its main purpose is for setting the value of variables used during the assembly to control conditionals and loops.

```
; Examples

low             =       0
low             =       low+5   ; redefine
```

GEQU

<label> GEQU <expr> (;comment)

The GEQU pseudo-op stands for Global EQUate and may be used only in the assembly of a linker module. It combines the actions of ENT and =. The labels are set to the value specified in <expr> which may contain no forward references or external values. The label is then declared in the module's entry list. See EQU, ENT, EXT.

```
; Examples
base            GEQU    &19     ; absolute
                RSECT
here            GEQU    *       ; relative
```

QUERY

<label> QUERY <prompt>

The QUERY pseudo-op is similar in operation to the EQUate pseudo-op, in that the value of the symbol is defined. In the case of QUERY the expression to be evaluated and assigned to the label must be typed in at the keyboard during pass 1. <prompt> is output to the screen as a prompt to the user. All the characters to the end of the line are output, and the prompt is not

considered to be a delimited string. QUERY allows the values of labels to be changed each time the program is assembled and its main use is for switching various conditional sections in or out by assigning the value 0 or -1 to labels used in IF statements. QUERY may be used to give the program ORG in a non-linker assembly. When the prompt appears the input buffer is flushed and most errors trapped. Entering a blank line or one that contains an undefined symbol will cause the assembler to ask you to type in the value again. The input to QUERYs may be provided in advance on the assembler command line (see section 4.2). See QSTR.

```
; Example
start       QUERY   Program start
yes,YES     =       -1
no,NO       =       0
debug       QUERY   debugging (yes/no)
```

ENT

<label> ENT (;comment)

The ENT pseudo-op creates an entry in the global symbol table passed with the object code to the linker. The symbol is assigned the current value of the location counter which may be in ASECT, RSECT or DSECT. If the ENTry is made in DSECT the symbol's value is absolute as if it were in the ASECT. Normally the entry point is the start of a subroutine that is globally available to all the modules being linked. ENTry symbols can be redefined with '='. Using ENT and = global equates may be made. Global symbols are all given 16 bit values but references to them may be 16 or 8 bit and may reference the MSB or LSB of the symbol. The linker checks that the value fits and will produce an error message if you try and use a global symbol bigger than 255 in a place where an 8 bit quantity is required.

```
; Example
print       ENT
            lda     (ptr),Y
```

ENDM

(<label>) ENDM (;comment)

The ENDM statement defines the end of a macro definition. If you miss the ENDM statement out in the source program the assembler will try to save the whole source in memory as a macro and you will probably get an "Out of memory" error message.

```
; Example
print       MACRO
            jsr     _print
            DATA    @1,&EA
            ENDM
```

QSTR

<label> QSTR <prompt>

The Query STRing pseudo-op issues the given <prompt> as a prompt on the screen. The user must then type in a string which is held in memory under the given <label>, which must be present. The assembler reserves space for the string and inserts the string itself in the object code when the <label> is encountered in the opcode field preceded by a dollar. The <label> must not be used elsewhere in the module being assembled. The string is in the same format as used by the ASC pseudo-op. If a $<label> is encountered as an opcode before the string has been entered the assembly will stop with a fatal error "Unknown in-line string". The string that is input is not delimited.

```
; Example
name        QSTR    Enter name

copyr       ASC     "The owner of this"
            ASC     "program is: "
            $name
```

EXT

<label> EXT (;comment)

The EXTernal pseudo-op specifies to the assembler that the given labels are not to be defined in this module but are global symbols whose value will be supplied by the linker. You may use as many EXT statements as you like but the fewer global symbols used the faster the linking process will be. An external reference can be added or subtracted from an absolute expression or used alone as an operand. Local symbols should not be used as external references because they are not supported by the linker.

```
; Examples
init,graf   EXT             ; graph lib
helio,mult  EXT
fs_open     EXT             ; MOS lib
```

EXZ

The EXternal Zero page pseudo-op is identical to EXT except that the label so defined is zero page and generates short

instructions. This label will be declared elsewhere with RZP or GEQ or ENTs in a dummy section.

MACRO

<label> MACRO (; comment)

The MACRO pseudo-op indicates that the source lines that follow form a macro definition. The obligatory label is the name of the macro. Once defined the macro may be used as an opcode, but not a symbol. A macro is generally a short sequence of instructions used repeatedly. By programming extensively with macros the program looks more like a high level language program and is easier to read. See Section 4.26 for more details about macros. MACRO definitions are held in memory so keep them as brief as possible with comment passages just above the MACRO statement. The macro definition ends with the ENDM statement. Macros may contain any assembler source statement except a further MACRO statement. They may refer to other macros in the opcode field. The total nesting depth for macros is limited by the amount of available memory. Each level uses one page of memory to store parameters and the REPEAT/WHILE stacks. Arguments may be supplied to macros when they are invoked but the arguments are not specified in the MACRO statement. In the definition when an argument is to be used it is given the pre-defined name @n where n is the argument number. Up to nine user arguments may be used in each macro together with an additional argument @0, which is a five byte ASCII number supplied by the assembler. @0 is used to define labels in the macro so that each macro invocation generates different labels. @0 is incremented each time a macro is invoked. When one macro calls another all the arguments including @0 are saved. See ENDM.

GET

(<label>) GET <macro>,(<macro>..) (; comment)

The GET directive fetches a macro definition from the current macro library opened with the MACLIB statement. Warnings will be issued if the named symbols already exist or are not in the library. GET functions differently from ADE versions 1 and 2.

```
; Example
          MACLIB my_macs
          GET    incw,decw,mulw,divw
```

BLOCK

(<label>) BLOCK (; comment)

The BLOCK pseudo-op defines a local symbol block. If the label is local it is part of the previous block. The scope of local labels is limited to the BLOCK in which they are defined. This allows the same names to be used repeatedly in different blocks and means the programmer does not have to think up ever more bizarre names for branch destinations etc. If a local symbol is not found within the current BLOCK during pass 2 a U (unknown symbol) error occurs.

```
; Example
            BLOCK
:1          DEX
            BNE     :1
            BLOCK
:1          DEY
            BNE     :1
```

OPT

(<label>) OPT (-)<option>(,<option>).. (; comment)

The OPT directive sets an assembler option. The options are single letters and may be given on the assembly command line or in the source code. Any letter A..Z may be given to set one of 26 flags in the assembler but only certain flags will have any effect. The current list of options is given in section 4.3. An option may be turned off by preceding it with a minus sign. A plus sign is permitted but ignored.

```
; Example
            OPT     P,G     ; print & go
```

4.20 Listing Directives

The listing directives control the listing in pass 2 of the assembly. They are optional but they can save space and improve the readability of a program. All of the listing directives will accept a label which is assigned to the current value of the location counter. However most listing pseudo-ops are not themselves printed, so the label will not appear in the listing, though it will appear in the symbol table. It is not recommended that labels are used in this manner. Listing directives do not generate any code output.

LST (LIST)

(<label>) LST ON | OFF | FULL (; comment)

The LST directive switches the listing of various source statements on and off. LST ON switches the listing on. LST OFF switches it off. This may be overriden using certain assembler OPTions. LST FULL lists out all source statements in macro expansions whereas LST ON does not. Sections of code that are conditioned out of the assembly will normally be listed and marked with an S. Use OPT C to suspend listing of these conditioned out statements. LIST may be specified instead of LST.

LLST (LLIST)

(<label>) LLST ON | OFF | FULL (; comment)

The LLST pseudo-op is identical to LST except that output goes to the printer instead of the screen. LST and LLST can be used independently.

LLIST may be specified instead of LLST.

TTL

(<label>) TTL <string>

The TTL pseudo-op sets the title up to a maximum of 21 characters to appear at the top of each page. The default title is a copyright string. The string is not delimited; everything up to the end of line is used. If the title is too long for your width of paper, the top of each page may look untidy so keep titles brief and to the point.

```
; Example
          TTL    Support routines
```

SKP

(<label>) SKP <expr> | H (; comment)

The SKiP pseudo-op skips the given number of lines on the output device. SKP H causes a page eject.

PAGE

(<label>) PAGE <expr> (; comment)

The PAGE directive sets the form length for the listing to <expr> lines. The default is 66 lines per page, of which 61 are printed (55 statements plus the header). The assembler always leaves a five line gap between pages, plus a formfeed to the printer, which can be deselected using LISTO. If the assembler thinks the value of PAGE is too small (less than 20) it will ignore it and issue a warning message.

WIDTH

(<label>) WIDTH <expr> (; comment)

The WIDTH directive sets the line width sent out to the printer. The default is 132.

LISTO

(<label>) LISTO <expr> (; comment)

The LISTOption pseudo-op sets flags in an 8 bit variable to control the listing format. The flags are toggled. The current assignments are: Bit 0 (LISTO 1) controls the title line. If bit 0 is zero then no titles or page ejects or page numbers occur. Bit 1 (LISTO 2) controls the line numbering. The default is line numbers and page numbers on. LISTO 3 will toggle both line and page numbers. Bit 2 controls formfeeds. LISTO 4 toggles this. The default is that formfeeds are sent to the printer. Bit 3 controls the symbol table listing. LISTO 8 means that no symbol table is listed. Bit 5 controls the listing of object code over three bytes per line. If LISTO 32 is specified then statements generating more than three bytes of object code will generate multiple lines.

```
; Example
          LISTO   32+8
```

SYSVDU

(<label>) SYSVDU <expr>(,<expr>..) (; comment)

SYSVDU sends the LSB of each expression to OSWRCH. Use with caution. It is basically the same as the VDU command in BASIC. Changing screen mode to a mode that uses more memory will probably crash the assembler. Choose the pass that you want SYSVDU to be used in by enclosing it in a pair of conditional statements using the # logical symbol which is true in pass 2. The SYSVDU allows an optional output stream specifier. This must be the first expression and must be preceded by a #(hash). The stream is selected using FX3 and the original stream restored at the end of the SYSVDU statement.

```
; Example
          IF     #
          SYSVDU 2
          INFO   Temporary version
          FI

          SYSVDU #0,...;Default to VDU
          SYSVDU #Z,...;Disabled
          SYSVDU #8,...;Printer and VDU
          SYSVDU #10,..;Printer only
          SYSVDU #3,...;RS423 only
```

SYSFX

(<label>) SYSFX <expr>(,<expr>(,<expr>)) (; comment)

The SYSFX command issues an OSBYTE call. It can, for example, select different output streams using SYSFX 3... This command is clearly open to misuse, so treat it with caution. The first expression is loaded into the A register; the second (optional) to the X register and the third (optional) to the Y register. No result is returned. Use conditionals and # to decide in which pass SYSFX will be executed.

```
; Example
          IF     ~#      ; pass 1
          SYSFX  6,0
          FI
```

SYSCLI

(<label>) SYSCLI <string>

SYSCLI sends the string to OSCLI. SYSCLI is a powerful pseudo-op that allows the issue of * commands during pass 1 of the assembly. Use with extreme caution. Use conditionals and # to decide in which pass SYSCLI will be executed. The string is not delimited, the remainder of the line is passed to the MOS.

```
; Example
          PAUSE  Insert next disc
          SYSCLI MOUNT 0
          CHN    File_2
```

INFO

(<label>) INFO <prompt>

This pseudo-op sends <prompt> to the console. It should be used with conditional statements to give warning and progress messages. Use with conditionals and # to decide in which pass you wish the string to be sent to the console. <prompt> is not delimited. All characters to the end of the line are sent.

```
; Example
          IF     ~#
          INFO   Assembling data
          FI
```

STOP

(<label>) STOP <prompt>

The STOP pseudo-op causes assembly to be aborted. INFO is called to print the message and the fatal error "Stopped" occurs. Again, this statement should be used with conditionals to detect an abnormal circumstance such as invalid QUERY data or a program exceeding a predetermined memory limit.

```
; Example
          IF     *>&C000
          STOP   Too big for ROM
          FI
```

PAUSE

(<label>) PAUSE <prompt>

The PAUSE statement sends <string> to the console. The assembly pauses until you press the space bar. This may be used, for example, to prompt for a disc change. See SYSCLI, INFO, STOP.

4.21 Data Definition Directives

These directives are used to define areas within the program. Directives to build address tables and messages and hex data are provided as well as byte, word and storage allocation. All of these pseudo-ops may have a label, which addresses the first byte of the data. They may all be followed by a comment. These pseudo-ops generate object code output.

ASC

(<label>) ASC <string> (; comment)

The ASC pseudo-op defines an ASCII string within the program. Bit seven of each character is controlled by the up arrow character (^). Control characters may be assembled with a vertical bar as in *KEY commands. Two up arrows assemble an up arrow, two vertical bars assemble a vertical bar character. The string is delimited by the first non-blank character. The last character is the same as this and the delimiters do not form part of the string.

```
; Example
prompt      ASC     "=>"
```

STR

(<label>) STR <string> (; comment)

The STR pseudo-op is identical to ASC except that a carriage return character (&0D) is appended to the string.

```
; Example
            LDX     #>mos_basic
            LDY     #<mos_basic
            JMP     OSCLI       ; exit
mos_basic   STR     "BASIC"
```

DC

(<label>) DC <string> (; comment)

The DC pseudo-op is identical to ASC except that bit seven is set in the last character generated and not set in any other character.

```
; example
keywords    DC      "INPUT"
            DC      "PRINT"
            DC      "LIST"
            DC      "GOTO"
            DB      0
```

MSB

(<label>) MSB ON | OFF (; comment)

Everywhere the assembler generates an ASCII character in the output code bit 7 is first set according to the MSB pseudo-op. The default MSB OFF means that bit seven is always zero (BBC format). MSB ON means that bit seven will be 1. This allows assembly for different machines such as APPLE II. Note that this does not apply to the DC pseudo-op which explicitly strips bit 7 on all characters except the last.

DFB (DB)

(<label>) DFB <expr>(,expr...) (; comment)

The DeFine Byte pseudo-op separately defines one or more bytes of data. Legal values are -127 to +255. A comma separates each defined byte. DB may also be used in place of DFB.

```
; Example
            DB      1,2,255,-127,-10,low+7
```

DW

(<label>) DW <expr>(,<expr>...) (; comment)

The Define Word pseudo-op defines words in 65C00 series lo-hi format. Any label used points to the lo-byte of the first word defined. Several words may be entered on the same line separated by commas. See DB.

DDB

(<label>) DDB <expr>(,<expr>...) (; comment)

The Define Double Byte pseudo-op is used to define words of memory in hilo- format. The label will point to the high byte of the first word defined. See DB, DW.

DS

(<label>) DS <expr>(,<expr>) (; comment)

The Define Storage pseudo-op reserves space in the object program for data. By default the assembler fills the bytes with zeros but a second expression may be used to define the 8 bit quantity used to fill each expression. The expression following DS must not contain forward references since the assembler needs to know the exact space requirements on pass 1. However the byte used to fill the space need not be known until pass 2. If the space allocated is large (bigger than a page) DS should be used in a dummy section. The label points to the first byte of the space. In linker modules DS may be used with complete abandon to define variable sections since the assembler generates linker information about the space reservation rather than lots of bytes of zeros. The file produced by the linker will still contain these bytes but it is possible to produce relocatable un-initialised variable storage by using the linker U option or adjusting the final object file when all the storage is at the end.

```
; Examples
            DS      100
            DS      &100,&AA  ; page of HEX AAs
```

HEX

(<label>) HEX <hex string> (; comment)

The HEX pseudo-op is used to define hexadecimal data tables. The string must be delimited and contain an even number of valid hex characters or else a B (bad string) error will be flagged. The bytes are assembled in the order in which they occur with the label pointing to the first byte.

```
; Example
            HEX     "0A23BCD7"
```

DATA

(<label>) DATA <expr> | <string> (<expr> | <string>..) (;comment)

DATA allows all kinds of data to be defined on the same line. The following conventions apply. <expr> will generate a byte of data. £<expr> will generate a word of data. Strings must be delimited by double quotes. Hex strings must be delimited by a / (slash).

```
; Example
          DATA    0,124,"Mismatch"
          DATA    £err-pm,/AB9F/
```

RZP

<label> RZP <expr> (;comment)

RZP stands for Reserve linker Zero Page. RZP must have a label and creates an external symbol with the label that is marked as a zero page variable of size <expr> bytes. The linker creates absolute addresses for each zero page variable. <expr> must be less than 256. When making linker libraries or working with them it is important to define all zero page variables with RZP to avoid conflicts at link time.

```
; Examples
ptr1,ptr2  RZP    2
zptab      RZP    10
```

4.22 Conditional Assembly Directives

Full conditional assembly is supported by the assembler using IF...ELSE...FI which may be nested up to 255 levels deep. The ELSE statement is optional. Conditional statements may be labelled and may be followed by a comment. Be careful to note which labels will be defined. It is wise not to use labels with conditionals in most cases. The special symbol # may be used as <expr> and is true on pass 2. Only non code generating pseudo-ops should be conditionally assembled on one pass only. See INFO, PAUSE, STOP, SYSVDU, SYSFX, SYSCLI.

IF

(<label>) IF <expr> (; comment)

The IF pseudo-op marks the start of a conditional block. The statements that follow are assembled if the expression is non zero. The expression must contain no forward references or relative symbols.

ÈLSE

(<label>) ELSE (; comment)

The ELSE pseudo-op may only occur inside an IF..FI block or the assembler will flag a C (conditional) error. The result of the corresponding IF is reversed, so that if assembly was being skipped it now resumes.

FI

(<label>) FI (; comment)

The FI directive terminates a conditional block. The assembler returns to the previously nested conditional state or carries on assembling if the FI was the outermost one.

```
; Example, the starred statements are skipped
            IF      -1
            block 1
            IF      0
***         block 2
            ELSE
            block 3
            FI
            block 4
            ELSE
***         IF      -1
***         block 5
***         ELSE
***         block 6
***         FI
            FI
```

DO

(<label>) DO <expr> (; comment)

The DO statement causes the next non-blank line to be assembled <expr> times. Thus, the line following it should not be a pure comment line (no action will result) and should not contain a label as the label will be multiply defined. The expression must be absolute and contain no forward references.

```
;Example, SHR <operand>,<bits>
SHR         MACRO
            IF      @N=1
            DO      @1
            LSR     A
            ELSE
            DO      @2
            LSR     @1
            FI
```

REPEAT...UNTIL

(<label>) REPEAT (;comment)
{<statement>...}
(<label>) UNTIL <expr> (;comment)

REPEAT...UNTIL is a high level assembler construct. The statements between REPEAT and UNTIL are repeatedly assembled until the expression following UNTIL is non-zero. This is done by moving the file pointer back to the statement following REPEAT each time (unless in a macro). The statements will be assembled at least once. REPEAT...UNTIL cannot be nested within the main body of the program text, but a REPEAT loop may occur inside a macro which is called from within a REPEAT. In such a case the corresponding UNTIL must occur in the same macro. REPEATS may not be nested within the macro but a nested macro may have its own REPEAT...UNTIL and so on. In the case of an INCLUDE file within a REPEAT loop, the whole file may be assembled many times. You must ensure no multiple label definitions occur. One way to do this is to use local labels and make the first statement a BLOCK.

```
; Example
DCLIST      MACRO
CNT         =       0       ; counter
            REPEAT
            DC      @[CNT+1]
CNT         =       CNT+1
            UNTIL   CNT>@N
            ENDM
```

WHILE...WEND

(<label>) WHILE <expr> (;comment)
{<statement>...}
(<label>) WEND

The WHILE statement is the converse of REPEAT UNTIL. The expression is evaluated and if it is non zero then the statements following are assembled, otherwise the assembler reads forwards for the statement after WEND. When the <expr> is true, then WEND causes the source file to be wound

back to the WHILE statement and the <expr> is re-evaluated. WHILE...WEND follows the same nesting rules as REPEAT...UNTIL.

```
; Example
DCLIST      MACRO
CNT         =       1
            WHILE   @?[CNT]>0
            DC      @[CNT]
CNT         =       CNT+1
            WEND
            ENDM
```

4.23 File Control

The last group of assembler pseudo-ops deals with the various types of file processed by the assembler. A maximum of three files will be open in pass 1. If a listing file is specified on the command line four files may be open in pass 2. In addition any EXEC file or SPOOL file adds one more file open, so check your filing system can cope before using all of these pseudo-ops.

INCLUDE

(<label>) INCLUDE <file> (; comment)

Include the file following in the source stream. The file is inserted between the INCLUDE statement and the statement following it. Thus a main assembly source file may just be a list of INCLUDE statements. The main source file remains open at the correct point throughout the assembly of the INCLUDE file. A fatal error occurs if the file is not found.

CHN

(<label>) CHN <file> (; comment)

The CHN pseudo-op chains to the next file in the assembly source program. A fatal error occurs if the file is not found. The previous file is closed and the CHN file becomes the current source. This statement makes programs compatible with ADE versions 1 and 2 but new programs should be written using INCLUDE for faster assembly.

MACLIB

(<label>) MACLIB <file> (; comment)

MACLIB defines the file that is used to fetch macro definitions. Macro definitions can be fetched using GET statements. In addition, if an opcode is not found in pass one and MACLIB is defined, then a search will automatically be made. See GET.

NOLIB

(<label>) NOLIB (; comment)

The NOLIB directive cancels the MACLIB directive. After a NOLIB unknown opcodes will not be looked up in the current library file. A new MACLIB may be specified after NOLIB. NOLIB also frees 2K of workspace so, if room is tight, open your MACLIB at the start of the source (pass 1) and read in all the macros you need with GET, then close the library with NOLIB. You will have 2K extra memory for symbols.

OBJ

(<label>) OBJ <file> (; comment)

The OBJ pseudo-op causes the current output file to be closed and a new output file to be opened. No other variables are affected, but by specifying a new ORG a second program may be assembled which can reference the first program because it shares the symbol table. OBJ is *not* valid in linker assemblies and produces a warning message.

```
; Example

          OBJ     jtab
          ORG     &FFE0   ; rom jumps
          JMP     start
          JMP     procA
          JMP     procB
```

Assembler Addressing Modes

This section describes how the various 65C00 and 6502 addressing modes should be specified in the assembler source program. The addressing mode is specified by the *format* of the operand field. Note that not every addressing mode is valid with every instruction. If the assembler detects an invalid addressing mode then a *format* error is flagged during pass 2.

In the descriptions that follow <addr87> is an 8 bit address in the range 0..255. <addr16> is a 16 bit address in the range 0..65535. <expr8> is an 8 bit expression in the range -128 to +255. 'ix' means one of the index registers X or Y.

Operand format	Addressing mode
blank	implied or accumulator
A (or a)	accumulator
#<expr8>	immediate
<expr8>	relative offset
<addr16>	absolute
<addr8>	absolute page zero
(<addr16>)	absolute indirect
(<addr8>)	zero page indirect (not 6502)
<addr16>,ix	absolute indexed
<addr8>,ix	zero page indexed
(<addr8>,X)	pre-indexed indirect zero page
(<addr8>),Y	post-indexed indirect zero page
(<expr16>,X)	pre-indexed indirect absolute (not 6502)

If the assembler detects a zero page address but there is no short form for the instruction then the absolute form will be used if it exists. Some addressing modes only apply to certain instructions when assembling for 65C00 series CPUs. These 'extended' instructions will generate format errors if you set the *restricted instruction set* option either in the source or in the assembly command line.

IMPLIED ADDRESS MODE

There is no operand. Examples TAX, SEC, PHX.

ACCUMULATOR ADDRESS MODE

The operand is an A or an a or there is no operand. Examples ASL, ROL, DEC A, DEC, ASL A.

IMMEDIATE ADDRESS MODE

A # character is followed by an 8 bit expression. Examples LDA #&FF, CMP #Z.

RELATIVE ADDRESS MODE

Used by branching instructions to specify the offset of the branch. The integer is the offset from the next instruction so it effectively lies in the range -126 to +129 bytes.

ABSOLUTE ADDRESS MODE

The operand is a two byte integer stored in lo-hi format and specifies the address to be referenced. Examples LDA &7000, JMP START.

ZERO PAGE ADDRESS MODE

The operand is a single byte specifying an address in the range 0..&FF. Examples LDA &30, ADC acc1, where acc1 is in the permitted range.

ABSOLUTE INDIRECT ADDRESS MODE

The operand is a 16 bit address in brackets. Only JMP indirect uses this mode. Example: JMP (&2000). JMP (vector). The bug in the 6502 processor JMP () instruction has been fixed in the 65C00 series so JMP indirect will behave differently on the two types of CPU if the vector lies on a page boundary.

ZERO PAGE INDIRECT ADDRESS MODE

The operand is an 8 bit address in brackets. This mode is only valid on 65C00 series processors. The addressed location and the one following specify a 16 bit address in the same format as the absolute indirect address mode, but the address of the vector can be specified in one byte. Examples LDA (&40), SBC (&12).

ABSOLUTE INDEXED ADDRESS MODE

The operand is a 16 bit address to which the contents of the X or Y register are added, making a final target address; for example, LDA table,Y, BIT &2000,X. In the restricted instruction set for the 6502 some instructions will not allow either index register to be used (BIT for example).

ZERO PAGE INDEXED ADDRESS MODE

The operand is an 8 bit address to which the contents of X or Y are added. This forms a target address which must still be in page zero. The Y register is only valid to load and store the X register. Examples STX &50,Y, ROR 12,X.

PRE-INDEXED INDIRECT ADDRESS MODE (ZERO PAGE)

The operand is an 8 bit address followed by a comma and an X in brackets. The target address is formed by adding the X register to the 8 bit address and using this (zero page) address as a pointer. Examples LDA (0,X), STA (10,X).

POST-INDEXED INDIRECT ADDRESS MODE

The operand is an 8 bit address in brackets followed by a comma and a Y. The target address is the address stored in the zero page location and the one following. To this address is added the contents of the Y register giving a final address anywhere in memory. Examples STA (ptr),Y, CMP (&67),Y.

PRE-INDEXED INDIRECT ADDRESS MODE (ABSOLUTE)

The operand is a 16 bit address followed by a comma and an X in brackets. This address mode can only be used with JMP instructions on the 65C00 series of processors. Example JMP (vectab,X).

	imp	acc	immd	rel	abs	zp	(abs)	(zp)	ab,x	zp,x	(zp,X)	(zp),Y	(ab,X)
ADC			√		√	√		◊	√	√	√	√	
AND			√		√	√		◊	√	√	√	√	
ASL	»	√			√	√			x	x			
BCC				√									
BCS				√									
BEQ				√									
BIT			◊		√	√			◊x	◊			
BMI				√									
BNE				√									
BPL				√									
BRA				◊									
BRK	√												
CLC	√												
CLD	√												
CLI	√												
CLV	√												
CMP			√		√	√		◊	√	√	√	√	
CPX			√		√	√							
CPY			√		√	√							
DEC	◊»	◊			√	√			x	x			
DEX	√												
DEY	√												
EOR			√		√	√		◊	√	√	√	√	
INC	◊»	◊			√	√			x	x			
INX	√												
INY	√												
JMP					√		√						◊
JSR					√								
LDA			√		√	√		◊	√	√	√	√	
LDX			√		√	√			y	y			
LDY			√		√	√			x	x			
LSR	»	√			√	√			x	x			
NOP	√												
ORA			√		√	√		◊	√	√	√	√	
PHA	√												
PHP	√												
PHX	◊												
PHY	◊												
PLA	√												
PLP	√												
PLX	◊												
PLY	◊												
ROL	»	√			√	√			x	x			
ROR	»	√			√	√			x	x			
RTI	√												
RTS	√												
SBC			√		√	√		◊	√	√	√	√	
SEC	√												
SED	√												
SEI	√												
STA					√	√		◊	√	√	√	√	

Table of opcodes part 1

Table of opcodes - part 1 cont.

	imp	acc	imd	rel	abs	zp	(abs)	(zp)	ab,x	zp,x	(zp,X)	(zp),Y	(abs,X)
STX					√	√				y			
STY					√	√				x			
STZ					◊	◊			◊x	◊x			
TAX	√												
TAY	√												
TRB					◊	◊							
TSB					◊	◊							
TSX	√												
TXA	√												
TXS	√												
TYA	√												

List of abbreviations used in the tables:

Addressing modes

imp	implied
acc	accumulator
imd	immediate
rel	relative
abs	absolute
zp	zero page
(abs)	indirect (absolute)
(zp)	indirect (zero page)
ab,X	absolute indexed
zp,X	zero page indexed
(zp,X)	pre-indexed indirect
(zp),Y	post-indexed indirect
(ab,X)	absolute pre-indexed indirect

Symbols

√	valid on 6502 and 65C00 series
◊	only valid on 65C00 series
x	only valid with X register index
y	only valid with Y register index
"	equivalent to accumulator mode

The first table shows opcodes that are valid for the 65C12 used in the BBC second processor and master series. The second table shows additional 65C00 series instructions supported by the assembler but not valid on the second processor or master unless you upgrade your CPU. These are BBR,BBS,RMB and SMB. The remaining opcodes are alternatives for opcodes in the first table. CLR is an alternative to STZ. DEA is an alternative to DEC A. INA is an alternative to INC A. These are not valid when the reduced instruction set is used. BLT is an alternative to BCC and BGE is an alternative to BCS.

	imp	acc	imd	rel	abs	zp	(abs)	(zp)	ab,x	zp,x	(zp,x)	(zp),Y	(abs,x)
BBR						◊							
BBS						◊							
RMB						◊							
SMB						◊							
CLR					◊	◊			◊x	◊x			
DEA	◊												
INA	◊												
BLT				√									
BGE				√									

Table of opcodes - part 2

4.25 Additional Opcodes for the 65C00 Series

The standard chip now used in Acorn machines is the 65C12. This chip supports additional opcodes listed in table one but does not meet the full 65C00 series specification. The assembler provides the following opcodes that may be used when writing for a CPU that supports them. The assembler will not know whether these instructions are valid or not for your processor so you should use them with caution. When the reduced instruction set option is in effect none of these codes are valid.

BBR Branch on bit reset

This instruction tests a bit of a zero page location and branches if the bit is zero. The syntax for this instruction is:

BBR <bit>,<addr8>,<addr16>

<bit> is a number between 0 and 7 and specifies the bit to be tested. <addr8> is an address in zero page and specifies the location to be tested. <addr16> is the target of the branch and must be in the normal range for branch instructions.

BBS Branch on bit set

This instruction tests a bit of a zero page location and branches if the bit is set to one. The syntax is the same as that for BBR.

RM Reset memory bit

This instruction sets a selected bit of a zero page location to zero. The syntax is:

RMB <bit>,<addr8>

SMB Set memory bit

This instruction sets a selected bit of a zero page location to one. The syntax is the same as RMB.

MACROS

Macros are short sequences of assembly language statements that are grouped together under a single word (the <label> in the MACRO statement). Whenever the macro name is used as an opcode the whole sequence of statements will be assembled. If LST ON or LLST ON is selected only the source statement invoking the macro will be listed, but if LST FULL or LLST FULL is in effect then each statement in the macro will be listed showing how any parameters passed to the macro have been substituted. A macro does not have to generate code, it can simply be a group of pseudo-ops. The amount of code generated by a macro may vary in each invocation because the macro may contain all the conditional statements allowed by the assembler. A macro may invoke other macros by using the = pseudo-op a macro may set a flag on its first invocation and vary its code accordingly. For example, an error macro may define a subroutine on its first invocation and simply call that routine on subsequent invocations.

The use of macros in the source program makes the program easier to read, but their true flexibility comes from the fact that they may be supplied with arguments, or parameters, that vary each time the macro is invoked. The arguments supplied to the macro are known as @1 to @9. Whenever the symbols @1..@9 occur in the macro text the assembler substitutes the relevant parameter. If a parameter is not supplied then the null string is substituted. The assembler generates an additional parameter @0 which is a five digit ASCII string that starts at 00000. The string increments each time a macro is invoked and is used to define symbols in the macro so that multiple definition errors will not occur. The text for macros is held in memory, in the symbol table, so that you should keep the text as brief as possible to avoid running out of room. @N is a one byte ASCII digit giving the number of arguments supplied. (@n may be used). @A to @J return the length of the argument 0..9. @A will always be five of course. The argument number in the macro text can be a variable or expression provided the expression is absolute, contains no forward references, and is in the range 0..9. The assembler must evaluate the expression whilst expanding the line, before the argument is substituted. To tell the assembler to do this the argument number is enclosed in square brackets, for example @[4+2] or @[arg_cnt] where arg_cnt is a pre-defined symbol. Round brackets may be used to substitute a substring of the argument in place of the entire argument. Two expressions are specified separated by a comma. Both must be absolute and evaluate in pass 1. The first is the start character and the second is the number of characters.

If you 'go off the end' of an argument no extra characters are substituted. If you start off the end of the argument a null string will be substituted. @(4,3)9 means start at the fourth character of argument nine and substitute three characters. @(2,2)[arg+7] is a valid argument expression. @?[<expr>] returns the length of an argument. Unlike @A to @J, the argument is variable, specified by <expr>. @?[0] will return 5 always.

The arguments are specified in the macro invocation as strings separated by commas. Leading spaces in each string are ignored. If a string must include a comma or a leading space then it may be enclosed in square brackets. Examples: A,X,Y [A],[X],[Y],[ ] 14,[(ptr),Y]. As arguments can be used anywhere without restriction, labels and macro names may be passed.

4.27 Macro Libraries

Macros may be defined in the program using the MACRO statement or obtained from libraries. A library is a file that contains macro texts put there by the MLIB command of the ADE plus MMU. The library file contains a catalogue of the macros in it and their position in the file so that the assembler can find a macro quickly using random access filing rather than a slow sequential search. The MACLIB statement specifies the library to use. When you use MACLIB the assembler reads the catalogue of macros from the head of the file into the symbol table so that all the macros in the library are available to you. The actual text for each macro is not read into memory until the first time the macro name occurs in the opcode field of an assembly source statement. To save a little time you may force load the text for macros from the library using the GET statement. This is also used if you then wish to open a second library using a further MACLIB statement. You must close the first library file with NOLIB before using a second library. The full format of the library file is specified in the *ADE plus Technical Reference Guide*.

```
;Examples:

        MACLIB  mlib4    ; open library file
        GET     incw,decw,stw   ; get three macros
        NOLIB            ; close library file
        MACLIB  genlib   ; open another library
        push    *        ; this macro in genlib
        incw    scrn_ptr; this in mem from mlib4
```

The push macro might be defined as:-

```
push    MACRO
        IF      "@1"="*"
        pha
        phx
        phy
        ELSE
arg     =       1
        WHILE   @?[arg]>0
        ph@[arg]
arg     =       arg+1
        WEND
        FI
        ENDM
```

4.28 Error Reporting

The assembler understands three kinds of errors. Fatal errors, non-fatal errors and warnings. A fatal error is one which causes the assembler to abort the assembly. A non-fatal error is an error in the source code which means the assembler could not compile it properly but is able to continue. A warning is a misuse of some assembler construct, usually the wrong use of a pseudo-op. A warning means that the assembler is able to continue compilation of the source correctly **at the time the warning is issued.** An example of a warning is when the GET statement includes a macro name not found in the library. Use of that macro later on will produce a non-fatal error.

4.29 Fatal Errors

All errors reported by the operating system to the assembler are fatal errors. Most fatal errors will be errors related to the filing system and IO. Pressing the escape key also generates a fatal error and running out of room for the symbol table generates a fatal error.

4.30 Assembler error messages

The assembler generates the following non-fatal error messages

A Address out of range
B Bad string
C Conditional error
D Divide by zero
E Equate or Entry error
F Format illegal *(eg JMP &1000,Y)*
H Bad hex
I Illegal symbol
L Linker rules violation
M Multiple definition
N Nesting error
O Opcode illegal
P Phase error *(label with different value on pass 2)*
R REPEAT error
S Syntax error
T Term missing
U Unknown symbol
V Value > 255
W WHILE/WEND error
X Macro expansion error
Z Zero page address expected

Extended syntax error messages, when option E is in effect

S Line starts with illegal character
S Y register expected
S X register expected
S Illegal index register
S Comma expected
S Illegal character in expression
S Illegal digit
S Illegal operator
S Missing ')'
S ON or OFF expected
S No statement after DO
S Illegal option
S Illegal bit number

Extended linker rule violations, when option E is in effect

L Branch to external address *(not allowed)*
L Illegal symbol in linker assembly
L Relocatable and external symbols mixed
L Absolute address expected
L Illegal use of external symbol
L More than one external symbol in expression

Fatal errors during an assembly

The assembler produces the following fatal errors. These errors are trapped by the BRK mechanism and are in addition to other fatal errors that may occur in the filing system software. When a fatal error occurs all open files are closed and the assembler returns to the ADE MMU prompt. The error is reported with the line number of the current file and the pass number, 1 or 2. The error numbers specified are the ones reported if the BRK is intercepted.

40. ***Text in memory lost***
An assembly of a RAM file (*) did a CHN or an INCLUDE.

41. ***Can't nest include***
An attempt to INCLUDE a file in an INCLUDED file was made.

42. ***Object buffer overflowed***
OPT G has been specified with no output file and the object code is too big for the output buffer.

43. ***Invalid assembler command line***
The assembler was entered with an invalid comman line.

44. ***Invalid QUERY statement***
Unable to process a QUERY.

45. ***Stopped***
Produced by the STOP command.

46. ***MODULE specified in absolute assembly***

47. ***ENDM expected***
The assembler got to the end of the current source file in a macro definition.

48. ***MACLIB library not found***

49. ***MACLIB read error***
The required macro is not at the position indicated in the MACLIB catalog.

50. ***Can't run linker module***
OPT G and MODULE both specified.

51 ***Can't run null/unbuffered object***
OPT G - the assembler can't find any code to execute.

52. ***Unknown inline string***
The assembler encountered a $<label> in the opcode field where <label> was not defined previously with QSTR.

53. ***Bad WHILE***
The expression after WHILE contains an error.

54. ***Object file is a directory***
An attempt has been made to open an ADFS directory for output.

55. ***Object file has E attribute set***
An attempt has been made to open a file for output that has the E attribute set (ADFS).

56. ***Insufficient workspace***
There is not enough memory to begin an assembly (<3k>).

4.32 Warnings given by the assembler

A warning is issued when the error encountered does not *immediately* lead to invalid object code. However a warning indicates something wrong with the source and should be investigated. The following warning messages may be issued

1. Can't use OBJ in linker assembly
2. Wrong use of ORG *(in RSECT)*
3. EXEC ignored in linker module
4. In DSECT *(produced by DSECT statement)*
5. Not in DSECT *(produced by DEND)*
6. In ASECT *(produced by ASECT statement)*
7. Page length! *(use 20 or more)*
8. MODULE name already specified
9. Illegal module name ignored
10. END in INCLUDE/REPEAT/WHILE *(ignored)*
11. Discontinuing embedded section
12. RESUME without EMBED
13. Can't use MACLIB in macro
14. Not a macro library *(MACLIB handler)*
15. Macro not in library *(GET)*
16. Name already known *(GET - macro not fetched)*

Warnings are produced by pseudo-ops and most warnings cause the pseudo-op to be ignored.

Chapter 5 The Linker

5.1 Description of the Linker

The linker is designed to take separate output files from the assembler, called linker modules, and produce from the information they contain an object program that is ready to run. The difference between this and the assembly process is that most of the assembly in linker modules has been done. The files only need to be relocated to some address in memory and have missing symbol values filled in. Thus linking is much quicker than assembling and linker files are smaller than source files. In a large program it saves time to assemble a small module being worked on and then link it with other code already tested.

The linker also allows the code to be split into sections, common data areas to be used between modules and libraries of assembled modules to be used. With a linker there should never be any need to re-invent the wheel. Once a routine is debugged it can be put in a library and used again and again without the need to reassemble the source.

Linking is initiated from the ADE plus control prompt by the LINK command, as described below. The linker is a two pass linker. On pass one the linker reads a header from each file that contains all the information the linker needs about the contents of the file. The header gives the size of each segment of the file and all the global symbols referenced and all the public symbols declared in the file. This is done for each file in the linker command line. Next, if any symbols have been referenced but not declared and a library specified, then the linker looks in the library for the relevant module and loads the header for that module.

A second pass reads the remainder of each file, the object code. This code has been assembled in linker format and is in a series of code records. Using the information gained in pass 1 the linker can take each code record and produce the required machine code at an absolute address. This code is written to the output file. When this has been done for each file in the linker command line the linker reads any library modules that are required and generates code from them in the output file. The output file is then ready to run.

Aborting a linking operation

A linking operation may be aborted at any stage by pressing the escape key. All files opened by the linker are closed.

5.2

The Linker Command Line

Linking is started from a LINK command after the ADE plus prompt. The LINK command is always followed by at least one space. The rest of the line specifies the files to be linked and various options supplied to the linker. The general format for invoking the linker is:

```
LINK {obj}{,sym}{/list}=mod[c]{,mod[c]…}{;opts}(-)
                {/lib[c]{,lib[c]…}}
```

The meaning of each part of the command line is:

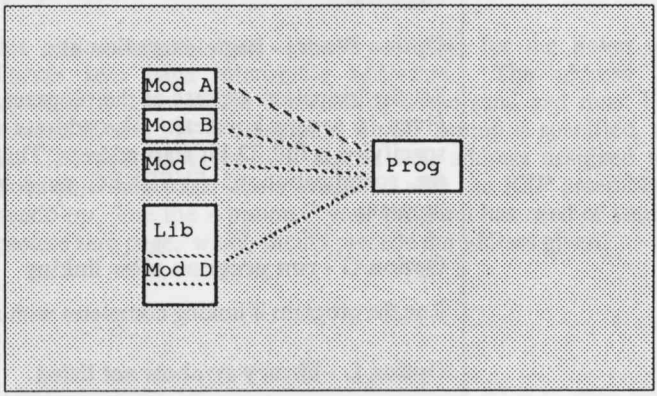
{obj}	The output file. The curly brackets show that it is optional. If not specified then linker output is confined to the symbol tables and cross reference listings etc.
{,sym}	An optional symbol dump. The name of the file is given preceded by a comma.
{/list}	A listing file. All printed output is duplicated to this optional file.
=	The separator between the output and input parts of the command line. This character is mandatory.
mod	A linker module output by the assembler. Any number of modules are specified separated by commas.
{[c]}	A conditional label. The label if specified must immediately follow the linker file name and be enclosed in square brackets. The label must be an absolute symbol that has been defined (globally, to the linker) before the current link file. If the value of the conditional is zero then the module is missed out of the linkage. Thus it is possible to include a module of test routines and run-time routines - the required module being selected by a switch, a conditional label, set in the main program.

{;opts} A linker option. Linker options follow the linker file list and are separated from it by a semicolon. Each linker option is a single letter but may have a parameter following it in square brackets. Options are processed before pass 1.

{/lib} A library file. The first library file is preceded by a slash. This file is searched at the end of pass 1 for missing modules.

{[c]} A conditional label. If this label is zero, then the library file preceding it is not searched.

{-} A hyphen may occur anywhere on the linker command line and indicate that the line continues with the next line, which is read from the keyboard or the EXEC file.



A linking operation involves linking modules and libraries to produce an executable program.

Linker Options

A number of options may be entered on the linker command line. Each option is a letter A..Z (or a..z) and controls the value of one of the linker variables. Several options require a parameter immediately following the option enclosed in square brackets.

Option A[adr] - address to link

The linker begins linking relocatable sections at this address. If the address is not specified then the lowest available address will be used as provided by OSHWM. Thus if the linker is being used on the second processor or master turbo, the program will normally begin at &800. Such a program will not run on the IO processor alone. The address may be 8 hex digits, specifying a full 32 bit load address and is given without a leading ampersand.

Option B[adr] - begin execution at...

Set the execution address of the program. The address must be given in square brackets. The address may be 8 digits long, specifying a full 32 bit start address. This can be overridden by the specification of SYSEXEC as a global symbol in the program.

Option G - run program after linking

Run the program if linking completes with no errors.

Option L - library symbols not listed

Omit symbols obtained from libraries in the symbol table listing

Option M - map file

Produce a memory map during pass one showing where each module will be located in memory.

Option P - echo output to printer

All linker output is sent to the printer.

Option U[file] - use symbol file

Use the specified file name, which must be enclosed in square brackets and be a symbol table file. The symbols from the file are included so that they are known to the linker. Typically these could be a table of entry points to the operating system of some other computer for which the program was being compiled.

Option W[width] - set printer width

Set the width of the printer page in characters. [width] is specified in decimal. The default is W[132].

Option X - cross reference listing

Send the cross reference listing to the screen or printer if selected.

Option Z[adr] - define zero page address space

This option sets the first address available for the linker to allocate zero page relocatable addresses. The address is specified in hex without a preceding ampersand and must be between 00 and FF. The default, if no Z option is specified, is 00. The address given will be the first address assigned to relocatable zero page labels produced with the RZP assembler pseudo-op. Use the Z option if your program has need of some absolute zero page, which should precede the address given.

.4

Example Linker Command Lines

PUZZEL=L.PUZZEL

Simplest linker command line. Produce an executable program, PUZZEL, from the module L.PUZZEL.

=PROG1,PROG2,PROG3 ; M,X /LIB

Test link the files PROG1, PROG2 and PROG3, produce a map, cross reference and symbol list on the screen using the library LIB.

GROM,S.GROM=GR1,GR2,GR3 ; A[8000],Z[40]/L.GLIB

A typical ROM linkage. Link files GR1, GR2 and GR3 to produce GROM with a symbol file S.GROM. The code is to be located at &8000 and zero page allocation to begin at &40. Use L.GLIB as a library.

.5

Linker Sections

This section describes the meaning of each type of linker section. The linker considers the program to be divided into three sections. These are called the dummy section (DSECT), the absolute section (ASECT) and the relocatable section (RSECT). Each section functions in the following way.

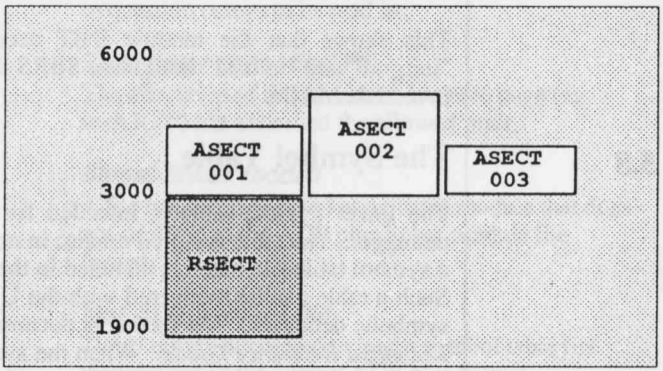
Dummy sections may be defined anywhere in the program with a DSECT command and they generate no code output in the linker file. However, the assembler assembles the statements in the dummy section and throws the output away but remembers the values of all the labels defined. Thus if global symbols are defined in a dummy section then the linker knows about them. The most common use of this would be to define the allocation of zero page storage. The following macro allocates a global name to a specified amount of zero page storage. The name points to the first byte:

```
RESERVE   MACRO
          DSECT
@1        ENT
          DS          @2
          DEND
          ENDM
```

The DEND statement reverts to the previous type of section at the previous location counter value. The initial value of the location counter in each section is zero. The value of the location counter in DSECT may be changed using ORG.

ASECTs may contain ORGs that set the absolute value of the location counter. Each ASECT or ORG (within an ASECT) results in the opening of a new object file. This is because the filing system format used by ACORN will not support 'scatter loading'. Each file takes a name derived from the parent file including a three digit number appended to up to four letters of the original name. For instance, the first ASECT in a linking operation to produce file GAME will be GAME000, the next GAME001 and so on. This convention is the one most often used for overlay programming. You must keep track of the ASECTs in your program. It is best to make each ASECT a separate module so that the numerical order of the ASECT object files will follow the order the modules are specified on the linker command line.

The relocatable section normally forms the bulk of a linker file. There is only one relocatable segment of memory and the linker joins all the RSECTs end to end during the linking process. All the addresses labelled in the RSECT are labelled as an offset from zero and so the linker must add the actual start of the RSECT on to each offset. The linker also has to be clever enough to recognise the use of just the top or bottom byte of a relocatable label. An RSECT is ended by the end of the source or the occurrence of another section. ORG must not be used in an RSECT. If ORG is used a warning message will be given, and the ORG will be ignored. This will not affect the assembler output.



Program with RSECT at 1900 and three ASECTs with ORG &3000, showing a possible overlay structure.

The Linker Map

As the linker comes across each new section it makes an entry in its section table and this table can be printed out. The table is called a map because it is in effect a memory map of the program. Each entry shows the section number and the start address and the length. DSECTs do not appear in the map since they only define labels. The linker prints the module name and section type followed by the first and last address used in the section. The last address is in fact the linker's location pointer value at the end of the section, so this is in reality the first 'unused' address in the section and will, in the case of RSECTs, be the first used address of the next RSECT.

The Cross Reference list

The cross reference table is output during pass 2. A heading is printed for each module and then every external reference encountered in the module is printed together with the module name containing the external reference. For example:

```
Module: INIT
   initgraf:SBRS
   initsnd:SBRS
   initbrk:MAIN
   start:MAIN
```

This shows that the module INIT accesses global symbols "initgraf" and "initsnd" in module SBRS then "initbrk" and start in module MAIN.

The Symbol Table

The linker symbol table is intended for use with a symbolic debugger. The table is stored on disc in non-ASCII format when a symbol table file name is included in the linker command line. Such a table can be recovered with the U option or used with a symbolic debugger. The format is documented in the *ADE plus Technical Reference Guide*. When the symbol table is listed the user has a choice of omitting library routines using the L option. The table is listed on the screen in alphabetical order, with the absolute value of each symbol provided.

5.9 Linker Error Messages

The following fatal error messages may be issued by the linker:

13. ***Invalid linker command line***
The command line passed to the linker by the ADE plus MMU could not be processed correctly.

14. ***Not a linker module***

15. ***Linking abandoned***
The errors reported during pass one of a linking operation mean that the linker is unable to continue.

16. ***Bad module***
A module name is too long or contains illegal characters.

17. ***Escape***

18. ***Conditional missing***

19. ***Illegal conditional***
The linker expected a valid label between square brackets following a linker or library module. Note this error is not detected when the linker command line is processed, but when the module in question comes to be read in.

20. ***Zero page exceeded FF***
The allocation of linker relocatable zero page (eg with RZP) has exceeded the allowed limit.

21. ***Illegal linker record***
A file has been specified as an input module that does not conform to the ADE plus linker module file format.

22. ***Not ADE symbol file***
The U option attempted to load a symbol table file that was not in the correct ADE plus format.

Chapter 6 The Debugger

6.1 Description of the Debugger

A debugging monitor based on the SPY debugger of ADE versions 1 and 2 is supplied on disc with this package. The debugger is supplied in three versions. Version 1 called DEBUGL runs below the mode 7 screen memory in main RAM. Version 2, called DEBUGH, runs in the second processor memory just below &F800. These versions are activated by typing *DEBUGx. The third version is suitable for sideways RAM and called DEBUG. Once loaded (eg with *SRLOAD) and registered (by pressing ctrl-break) it can be accessed from the ADE plus MMU with the DEBUG command. The utility program LBUG loads the debugger and pages in a given RAM slot. For example to debug RAM slot 15 type.

```
*LBUG 15
```

On entering the debugger the screen displays a 64 byte block of memory, the processor registers and a command line. Most commands are re-entered with a single key stroke. If the * key is pressed, the debugger accepts a whole line of input and passes the line to OSCLI. Thus, to exit, you can type *ADE. The memory display can be in hex or ASCII or in disassembly format.

6.2 Debugging commands

Unless otherwise specified, all the commands are single key entry.

L - Display dissasembly

The L key will toggle the memory display between hex and disassembly. The memory pointer will then point to one of eight instructions disassembled on the screen.

TAB - Display in ASCII

Display memory in ASCII text with a full stop printed for characters that are unprintable.

M - Set memory pointer

The highlighted location in the middle of the memory display is called the memory pointer. This is the location that will be updated if you enter commands to type data into memory. Type M followed by the hex address. No ampersand is entered, all data is in hex. An arrow (>) prompt will appear when you press M. This prompt always appears when data is expected to be entered. Press return when you have entered the address. The debugger will now display memory centred on the address entered.

```
M>2A05
```

Note that this can also be achieved by typing

```
2A05M
```

without pressing return.

RETURN - Increment memory pointer

Move the memory pointer to the next highest address.

+ - Increment memory pointer by 8

Move the memory pointer up by one column. Typing ; has the same effect.

/ - Decrement memory pointer

This is the opposite of pressing the return key.

- - Decrement memory pointer by 8

Move the memory pointer back by 8 locations.

U - Update memory pointer from PC

The memory pointer is set to the address contained in the PC register, as shown on the register display at the top of the screen.

@ - Update PC from memory pointer.

The program counter is set to point at the same memory location as the memory pointer. This may be done prior to single stepping, for example.

I - Set memory pointer indirectly

The memory pointer is updated from the contents of the current memory pointer location and the location following it. The current location is considered to be the low byte of the address. This procedure is often called word indirection.

R - Set memory pointer relative

The memory pointer is set as if the contents of the current location pointed to by the memory pointer were the displacement in a branch instruction.

G - Get first occurrence of pattern

N - Get next occurrence of pattern

These two commands enable a search to be made for a specified byte pattern or string. Press G followed by the pattern. This may be entered in hex one byte at a time. Each byte is followed by return. Strings are entered between quotes. Press return on its own to begin the search. The search commences at the byte following the current memory pointer. If the pattern is found the search stops with the memory pointer at the first byte of the pattern. Further occurrences of the pattern can be found by pressing N. When the pattern is not found (memory pointer goes past zero) the message *?err?* is displayed. For example to find the instruction JSR &21AF, enter:

```
G
>20
>AF
>21
>
```

6.3 Altering Memory

To alter the current memory location indicated by the memory pointer, enter a hex number and press the space bar. If you enter a number and press return, the number will be entered and the memory pointer will advance by one. To enter a string into memory, type a quote character and enter the string which will be terminated with a second quote. Each character will be stored and the display updated as each key is pressed. To see text in ASCII on the display, press the TAB key.

P - Fill memory block with byte

This command allows a range of memory to be filled with the same byte. To clear four pages from &1900 to &1CFF, for example, type:

```
P
first: 1900
last:  1CFF
with:  0
```

If the last address is less than the first the debugger displays *?err?* . As the debugger enters the data it reads it back and checks the value. If the data does not verify (eg if you write over a ROM) then the message *?fault?* will be displayed with the memory pointer set at the offending location.

S - Shift memory contents

This command will intelligently move a block of memory of any size to a new starting address. To move a page from &1900 to &1980, enter:

```
S
first: 1900
last:  19FF
to:    1980
```

If the last address is less than the first then the debugger displays *?err?*.

V - Verify two memory blocks

This command compares the contents of two memory blocks and if a difference is found prints *?fault?* with the memory pointer set to the offending address in the first block. If the two blocks are the same *ok* is displayed. To verify that pages &19 and &29 are the same, for example, type:

```
V
first: 1900
last:  19FF
with:  2900
ok
```

6.4 Altering the registers

The changeable register is highlighted and indicated by an arrow on the left. Pressing the full stop key moves the pointer to the next register, or back to the first. The value to put in the register may be entered by typing the data in hex and pressing a full stop. The PC register expects four digits. Two digits may be entered for other registers. The status register shows the individual flag names if the corresponding flag bit is set to one.

6.5 Miscellaneous commands

Z - Single step

Pressing the Z key executes the instruction at the current PC location. This instruction will be displayed in disassembly form between the register(s) and memory display(s). The register(s) and memory display(s) will be updated to show the result of the step.

K - Continue from PC

The program being debugged is allowed to run from the current PC location with the registers set as shown.

J - Jump to address

Execute a program from the address entered after the J command. Type in the address and press return. The debugger does a JSR to the location entered, so subroutines may be tested. If you press J but then decide not to jump, press the escape key. To run a program from &1900, enter:

```
J>1900
```

: - Clear break points

Clear all 8 break points.

, - Toggle break point

Set or clear a break point. Up to 8 break points may be used.

> - Advance to break point

Advance the memory pointer to the next break point.

£ - Advance to next instruction

Advance the memory pointer to the next instruction.

X - Exit

Return to ADE MMU (from ROM debugger).

INDEX

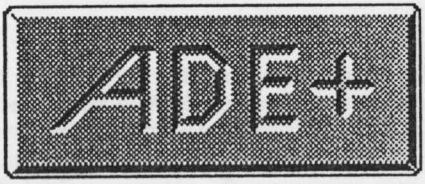
The main section dealing with each subject is shown in **bold** type. Other references are to page numbers.

f0	f1	f2	f3	f4	f5	f6	f7	f8	f9
mark	unmark	del block	mov block	goto 1	goto 2				
ins/ovr	find next	goto label	clear line	split line	join lines	insert line	delete line	insert char	delete char

This function key strip can be cut out and used with the ADE+ mini text editor.

This function key strip can be cut out and used with the ADLE+ mini text editor.

Your Key to Expanding the Power of ...



Technical Reference Guide

SYSTEM SOFTWARE SOUTH YORKSHIRE SYSTEMS FOR TRAINING, EDUCATION AND MANAGEMENT LIMITED, 12 COLLEGIATE CRESCENT, SHEFFIELD S10 2BA. TEL: (0742) 682321

Published in the United Kingdom by:
South Yorkshire Systems for Training Education and Management Limited,
12 Collegiate Crescent,
Sheffield, S10 2BA,
England.

First Published 1986

CONTENTS

Acknowledgements

The authors wish to thank all those who have helped in the development of ADE plus. Thanks are due to all the original developers of ADE and customers over the years who have made valuable suggestions. We have tried to include all of the best ideas that you have come up with. Thanks are due (again) to Ray for trials work and to Nigel for trials and suggestions for this user guide. Programming was by Steve with helpful assistance from Dave, who also wrote the BASIC program conversion utility. Dr Oliver Blatchford made useful additions to the CONVERT program.

1 Introduction

The ADE plus Technical Reference Guide is a supplement to the ADE plus user manual and can be clipped in the back of your ring binder. None of the information contained in this manual is necessary to operate the ADE plus software; that information is already contained in the user guide. What this supplement will do is enable you to understand how the ADE plus system works and expand the system to meet your own requirements. You will need to read all of the information in the subsequent chapters in order to be able to write sideways ROM modules that extend the ADE plus system and take full advantage of the memory management unit (MMU) facilities.

This guide does not explain how to write sideways ROM software. A full explanation of the Acorn MOS calls to sideways ROMs, and the format that a ROM must take, is provided in the *Advanced User Guide for the BBC Microcomputer* published by the Cambridge Microcomputer Centre and the *Reference Manual Part One* to the BBC Master series, published by Acorn. Either of these books provide all of the information required to write generic software. Applications taking specific advantage of the Master's capabilities will need the Acorn manual.

1.1 Generic Software

The philosophy of the ADE plus system, if it has one at all, is that of generic applications. This means that all of the programs supplied with the ADE plus system will work on the BBC B, BBC B+, Master, Master Turbo and Master Compact. The software will operate with the DFS, ADFS, NFS and ANFS filing systems. It will operate with most third party add-ons provided they stick within the rules and meet the full specifications of the Acorn MOS. Sadly many third party filing systems that we have tested fail to do this. Users are strongly encouraged to make their applications generic when extending the ADE plus system. This will allow users to share new modules among themselves without due regard to the system each user has.

Writing generic software does not mean that the system cannot take advantage of the extended capabilities of the Master series. What it does mean is that the programs must be aware of which system they are running on and take appropriate action. For example, when you type *ADE the MMU will obtain the time and date from the CMOS RAM if running on a master or, failing that, from the network (if that is the current filing system) or, as a last resort, will ask you for the information. The main restriction of generic software is that the extended instruction set of the 65C12 or 65C02 microprocessor cannot be used. The

main consequence of this is that programs are slightly larger and very slightly slower than they would otherwise be, but in reality there is no problem, everyone agrees that ADE plus is a fast and extensive system. Of course the potential market for generic software is much bigger than for software tied to a specific machine or filing system.

You will only be able to use the routines listed in the Technical Reference Guide on systems where ADE plus is already present since they all use the MMU facilities.

System Overview

The ADE plus system consists of a memory management unit, which is also an overall system manager, and a number of modules. These modules are either supplied with the system or supplied as additional software or are your own creation. A module is either *resident* in which case it is in sideways ROM or RAM, or *transient* in which case it is held on disc (or RAM disc) and called into memory by the filing system when required. Examples of resident modules are the 65C00 series assembler, the Linker, the Debugger (DEBUG) and the macro librarian. Examples of transient modules are the RAM based Debugger (DEBUGL/H) and the file filter program. Transient modules fall into two categories. The filter program is an example of software that makes no use of the ADE MMU variables and routines. This program will run without ADE plus being present. Other transient modules will make use of the MMU information and can only run alongside ADE plus. In this guide any program that makes use of the MMU, or *is useful* to the MMU is referred to as a module. The golden rule for any module is that it returns control to the MMU variables intact.

The MMU can be in ROM or loaded from disc into sideways RAM (the software is the same). It will be referred to as the MMU ROM. The Linker and Librarian modules are part of the MMU ROM and called directly from the MMU command level. These modules cannot, therefore, be replaced. The assembler and any additional modules are all external to the MMU and can be replaced by your own code if necessary. The MMU follows specific rules for calling modules in sideways RAM. Transient modules are always *RUN by the current filing system.

An exception to the rules described above is the editor module. The MMU will send editing instructions to a ROM based editor module if present. If no such module is found then the MMU will use it's own internal text editor. The MMU will not

immediately recognise VIEW or WORDWISE as an editor module, but if you have facilities to patch these ROMs they can easily be made to appear as the advanced editor that the MMU craves.

3 MMU Variables

This section describes the location and purpose of each of the ADE MMU variables.

The MMU functions as a language. This means that it will run on the second processor if present. It is allocated 1K of workspace by the MOS, from &400 to &7FF. The MMU variables that must be preserved are all held on page 4 between &400 and &490. Modules must not interfere with these variables under any circumstances. It is not good enough to rely on restoring the variables unless you intercept the BREAK key. If your module also functions as a language it will be re-entered on BREAK. It must note that this is a re-entry and not simply save the variables again. If your module is not a language then you should note that the MMU variables will be held on the *language* processor. Thus if a second processor is connected, modules running on the IO processor will not be able to access the MMU variables. To curcumvent this problem, all modules should run on the language processor. This is achieved by setting the top bits of their load and execution address to zero

As well as passing the variable on page 4 to a module, the MMU also passes the command line or command line tail to the module in page 7 of the language processor. Modules that only run from the ADE MMU command level can always find the command line here and do not need to go through MOS calls to access it. Once the command line has been processed by a module it can be thrown away.

A full description of each variable is given though not all variables are useful. Many variables are referred to in the following chapters and their use will become more apparent.

&400 SYSTEM_STATUS

SYSTEM_STATUS is normally zero. Modules can set this variable to &FF to indicate that they have been entered once. The MMU will not reset it, so modules should set it back to zero before exiting to the MMU. Rom based modules can use the ROM workspace byte allocated to them by the MOS to store their status, but the MOS byte is on the IO processor and dependent on

the ROM page that the module sits in, so SYSTEM_STATUS may be used more conveniently.

&401 LINK_ROM_SLOT

This variable contains the ROM id of the ADE MMU, Linker and Librarians. To exit to the MMU load the X register with this variable and perform OSBYTE 142.

&402 ASM_ROM_SLOT

This variable contains the ROM id of the ADE 65C00 series macro assembler. To perform an assembly from another module set up the assembler command line tail (everything *after* the ASM command) at &700 and load the X register with this variable then do OSBYTE 142. Editor modules that contain a RUN command should follow this procedure. When no 65C00 assembler is present this variable contains &FF,

&403 EDIT_ROM_SLOT

This variable contains the ROM id of the advanced editor module or &FF. If it contains &FF then EDIT commands are passed to the MMU's internal text editor. If the variable contains a positive value then the MMU calls this ROM with the EDIT command line (in its entirety) at &700.

&404 DEBUT_ROM_SLOT

This variable contains the ROM id of a ROM based debugger module of &FF. DEBUG commands will be passed to this module, if present, in the same way as EDIT commands are passed.

&405 ZASM_ROM_SLOT

This variable contains the ROM id of a Z80 assembler or &FF. The command line tail of a ZASM command will be passed to this ROM at &700. The ZASM command could be used with any utility module as a way of extending the system, for example a C compiler. The MMU will not process ZASM commands unless a command line tail of at least one non-blank character is present. You will note that the EDIT and DEBUG commands do not *require* parameters and thus the whole command line is passed for analysis. The ASM and ZASM commands *always* require parameters and the MMU moves these to &700 for your convenience.

&406 OPT_A

This is the first assembler global option, as set with the OPT command. The asembler must copy 26 bytes from OPT_A into its local options since these may be modified by the command line or by OPT statements in the source file. OPT_A must *not* be modified. The Z80 and 65C00 assemblers are deemed to share the same list of options and you should stick to the interpretation of them if you write a Z80 assembler, though unused options will be available to you for any purpose .

&420 LOPT_A

This is the first linker global option. Since the linker is internal to the MMU, these options are of no interest to external modules.

&43B HIGH_WATER_MARK

This is a word variable, low byte first. It specifies the value of OSHWM on the language processor. Its main use is internal to MMU.

&43D RAM-TOP

This work variable specifies the top of the workspace RAM on the language processor. The assembler, for example, sets up a stack at RAM TOP for macros and libraries at this location. Transient modules running on the second processor will be passed their start address as a value of HIMEM from the operating system, which is of little use. RAM_TOP will give the true value which will never be greater then &8000.

&43F TUBE_STAT

This variable is set to &FF and is the language processor, a second processor or co-processor.

&440 RAM_STAT

RAM_STAT is a list of 15 variables, one for each page of sideways ROM or RAM starting at page 0. For each page n, the variable at RAM-STAT+n is defined as follows: Bit 7 is set if the page was found to be RAM, otherwise it is reset. Bit 6 is set if the page is protected. Protection is achieved by the PROT command or by default because the RAM page contains an ADE module.

&450 INPUT_BUFFER_ALLOCATED

This variable is set ot &80 when an input buffer is in use. It is set to 0 if no input buffer exists or if the size is set to zero. Changing MODE and issuing several other commands forces the MMU to recalculate its buffer sizes and set this flag. If the flag is reset (0) then the information describing the buffer is undefined. The assembler uses this information to set up a local buffer in the workspace if the MMU does not provide one, so do not rely on the buffer description unless INPUT_BUFFER_ALLOCATED is true (&80).

&451 INPUT_BUFFER_PAGE

This variable indicates the location of the input buffer. The only meaningful values are &10 which indicates the buffer is in the second processor, &11 which indicates that the buffer is on the IO processor but the language processor is the second processor, and &13 which indicates that the buffer is in the IO processor which is also the language processor. Values 0 to &OF would indicate a buffer in sideways RAM, but this facility is not available on Version 1 of the MMU. When the value is &11 then the buffer is said to be *far*. Far buffers are accessed using MMU OSWORD routines to transfer blocks of data across the tube.

&452 INPUT_BUFFER_START

This is the first page of the input buffer. Buffers always start on a page boundary so the lsb of the buffer start address will always be zero. When the INPUT_BUFFER_START will be &CO, indicating that the buffer starts at &C000 on the second processor. If INPUT_BUFFER_PAGE was &11 or &13 and INPUT_BUFFER_START was &19 then the buffer would begin at &1900 on the IO processor.

&453 INPUT_SIZE

This variable gives the size of the input buffer in pages. If the buffer has been set ot 5K with the INPUT command then this variable will be &14. When the second processor is connected the value of this variable is defined by the system memory available and cannot be altered. If INPUT_BUFFER_START is &CO, then the variable will always be &38 indicating a size of 14K, or a buffer extending from &C000 to &F800.

&454 OUPUT_BUFFER_ALLOCATED

This variable is a flag indicating that an output buffer exists. See INPUT_BUFFER_ALLOCATED.

&455 OUTPUT_BUFFER_PAGE

This variable gives the location of the output buffer. See INPUT_BUFFER_PAGE. When a second processor is in use the largest buffer is given to the input and the second largest to the output. Thus one buffer will be on the IO processor, (far: &11) and one on the second processor (&10). If a shadow screen is used then the IO processor usually contains the most free space and will be allocated to the input buffer but otherwise the allocation depends on screen mode.

&456 OUTPUT_BUFFER_START

This variable gives the msb of the start address of the output buffer. See INPUT_BUFFER_START.

&457 OUTPUT_SIZE

This variable specifies the size of the output buffer in pages. See INPUT_SIZE.

&458 PRINT_BUFFER_ALLOCATED

This variable is set ot &80 if the print buffer is allocated. When the print buffer is allocated the print spooling system is kicked into life. The print buffer will be allocated whenever the buffers are checked (eg at *ADE or a mode change) and there is unprotected sideways RAM available. The buffer is de-allocated with the PRINT 0 command or if all the sideways RAM is protected. When this happens, the print spooling system is informed and relinquishes control to the MOS printer driver. However if the buffer is not empty, then the printer spooler informs the MMU that it cannot change the size of the print buffer or de-allocate it.

&459 PRINT_BUFFER_PAGE

The variable gives the first page of sideways RAM available for the print spooling system. The print spooler keeps its own variables in the IO processor so if this information is lost (eg by *BASIC) the print spooler will keep on working.

&45A PRINT_BUFFER_START

This will be set to &80 if the print spooler is active because the sideways RAM pages start at &8000.

&45B PRINT_SIZE

This is the size of the print buffer in K, not in pages. If the size were in pages, then a 64K buffer would have a size of 0! The size will be a multiple of 16.

&45C TUBE_ALLOCATED

This variable is used internally by the MMU. It is set to true (&80) when the tube spare memory from &C000 to &F800 has been allocated to a buffer.

&45D IOP_ALLOCATED

This variable is used internally by the MMU. It is set to true (&80) when spare memory in the IO processor has been allocated to a buffer.

&45E TUBE_FREE_MEMORY

This variable contains the amount of free memory available on the second processor for buffers. Its normal value is 0 (none) or &38 (14K).

&45F IOP_FREE_MEMORY

This variable contains the amount of free memory available on the IO processor for buffers. Its value depends on screen memory usage and the IO processor setting of OSHWMN.

&460 IOP_PAGE

This variable contains the value of OSHWM (msb) on the IO processor. When a second processor is in use, this value is different from OSHWM for the language processor.

&461 WATER_MARK

This is a very important variable. It is the page boundary on which the workspace available to modules (on the language processor) begins. This may be the same as OSHWM but is not necessarily so. If buffers are allocated by the user in the language processor, then they will be allocated between OSHWM and WATER_MARK. You must not use memory below this address if you want to use the MMU buffers for file operations.

&462 SCREEN_MODE

This variable contains the screen mode upon which all the memory calculations are based. Modules may change mode but must set this mode again before returning to the MMU. The screen mode will normally be 128 or greater because the MMU likes to select a shadow screen, if available, in order to maximise the amount of free memory.

&463 - &46F Reserved for future expansion.

&470 DATE

This variable contains the system date as determined from CMOS RAM or Econet or entered by the user. It is in the form of an ASCII string terminated with a CR (ASCII &OD). The maximum length of the string is 16 characters including the CR. If the date was not specified, then the string would be just a CR. When the TIME command is issued with a second parameter, then the parameter is stored here. The date can be in any format since no calculations are done with it.

&480 TIME

This variable holds the MMU time as printed on the prompt line. The time is updated each time an MMU command is entered, so it represents the time that a module was invoked. The time is actually kept in the MOS system timer because this is common to all BBCs. During initialisation the system timer is loaded from the CMOS clock, Econet clock, or from the user's watch. Unfortunately the system clock tends to run a little slow because of missed interrupts, especially in cases of much disc access. However the good news is that when you enter a TIME command only the system timer is changed, not the CMOS of Econet clock so you cannot upset these important time pieces accidentally. The time string is HH:MM <cr> (6 bytes).

&490 - &4FF Reserved for future expansion but may be used by modules until further notice.

The memory between &463 and &46F may be used in Version 1 of the ADE MMU. Modules must not use these memory locations. Memory after TIME will not be used in Version 1.

Operations performed during *ADE

This chapter explains what happens when you type *ADE . It is not sufficient to enter the ADE MMU with OSBYTE 142 unless this initialisation has been performed (by ADE or by your software). *ADE is a service routine that is always run on the IO processor. When it has completed, it enters the ADE MMU language ROM with OSBYTE 142. Modules also restart the MMU with OSBYTE 142. The first thing the MMU (language) does is to examine its workspace byte allocated by the MOS to determine the state of the MMU variables. This workspace byte which may be at any location between &DFO and &DFF on the IO processor according to which page the MMU ROM lies in, is set to one of the following value:

1 COLD_START - set by hard break.
2 WARM_START, data in page 4 is ok. This is set if the BREAK key is pressed.
3 ADE_START - set by *ADE.
4 ADE_CONT - set to this value by the MMU before calling a module.

The ADE MMU ROM responds to the following service calls:

&27 Issued during BREAK key processing on the master.
&03 Issued during BREAK key processing and initialisation. Print ROM title. Set workspace byte to COLD_START for a hard break and WARM_START for a warm break.

&09 If no parameter, give ADE ROM title (*HELP).

&08 Check processing of ADE OSWORD and perform appropriate action.

&04 Check unknown command for *ADE. If *ADE is the command, then perform a cold start on the MMU.

The cold start sequence is:

Initialise scan of sideways ROMs for ADE resident modules.
Scan ROMs for 65C00 Assembler. Result to ASM_ROM_SLOT (&FF= not found).

Scan ROMs for Advanced editor. Result to EDIT_ROM_SLOT.
Scan ROMs for Debugger. Result to DEBUG_ROM_SLOT.
Scan ROMs for Z80 Assembler. Result to ZASM_ROM_SLOT.

The results are only copied into the ROM_SLOTs when the language is started. They are saved in temporary locations during the rest of the initialisation. It is important to remember that the memory between &400 and &7FF could contain the Tube operating software at this stage.

Search system and determine the amount of available memory.

Set ROM workspace byte to ADE_START and issue OSBYTE 142 to start MMU language processing.

ADE next gains control as a language on the language processor. It first reads the ROM workspace byte. If this is set to COLD_START then a hard break has occurred so the MMU issues the *ADE command to initialise properly. If a warm start is issued, then a soft break has occured to the MMU checks the validity of the data on page 4. If it is not valid, then *ADE is issued, otherwise it carries on as if returning from a module. Finally, if none of these reasons apply, then the reason is a start up after the *ADE command so...

Read the ROM slots and RAM status information from the IO processor temporary locations and expand into RAM_STAT and the SLOTs in the variables. Check RAM_STAT against the SLOTs and protect any modules in sideways RAM. Read the tube presence and set TUBE_STAT. Reset all options to zero. Set initial screen mode. Set the initial time. Calculate the buffer sizes and locations (this involves reading OSHWM, possibly on both processors) and call the STAT command; then prompt for a command. Start the print spooler if possible.

At this stage all the ADE MMU variables are set up. When a command is issued the first work on the line is checked. If this is a valid ADE command, then the remainder of the line is analysed for syntax automatically, using the same information as COMMANDS uses to print out the help screen. Then the routine to handle the command, or the external module (such as the assembler) is called. If the command is not recognised, then the MMU asks the filing system if a file exists with the given name. If it does, then the file is *EXECed. Otherwise the command is passed to the MOS. Thus TYPE, for example, will be passed to the filing system after the MMU has already accessed the disc to check for a file of the name TYPE.

When a BRK error occurs the MMU closes all open files, inlcuding the EXEC file. On ADFS systems this can cause another BRK error if there is no catalogue in memory, so the MMU is ready and gives up if another error occurs. Press ESCAPE with no ADFS disc mounted to see this happen.

Calling resident modules

The MMU recognises a module is resident because its SLOT number is positive. The slot numbers were set initially from the IO processor by the *ADE routine. They scan the ROM titles for recognisable sequences of characters. These are the ROMs currently recognised:

ADE 6500
This is the title of the 65C00 series Macro Assembler. The Macro Assembler supplied with ADE plus has no service call to initialise itself. It relies on being called by the MMU. It does support other services, however, in order to recognise that the BREAK key has been pressed, but there is no need in theory to have a service entry point into a ROM module. However to avoid the MOS confusing the ROM with BASIC, it is best to have a service entry even if this is just an RTS.

ADEED
This is the title of advanced editor ROM. Chapter 9 shows how to modify a well-known word processing ROM to function as an advanced editor.

ADEBUG
This is the title of the ADE Debugger when present a a ROM. The program DEBUG that comes with ADE plus has this ROM title. Once it is installed ADE may be retarted with *ADE and the DEBUG command will be operational because ADEBUG will have been spotted.

ADEZ80
This is the title of an ADE Z80 assembler module, or any user module that you write to be accessed with the ZASM command. The list of commands in the MMU can be found by scanning the ROM with DEBUGL. If you are running in sideways RAM you could change the ZASM command to your own. The command will have to be 4 letters and bit 7 of the last letter must be set. The command is followed by a syntax byte (&20) and the address of the routine to call the ZASM module. Do not alter the address. Replacing the syntax byte by zero will mean tht the MMU will not insist on parameters to the command. In this case the whole command line including ZASM will be passed to your module in page 7. As it stands only the command line tail will be passed starting at the first non-blank character.

5 ADE OSWORD

The MMU and modules that communicate with it need some way to pass information, often across the tube. This is done using the OSWROD call mechanism. In order to reduce the possibilities of conflict with other ROMs, a single OSWORD call is used, and the first byte of the parameter block specifies a function number. This chapter describes each function currently implemented. For each function a parameter block of less than 16 bytes is required, pointed to by the YX registers. The first byte of the parameter block is always the function number. Results are returned in the paramter block, but some calls copy memory across the tube as well.

The ADE OSWORD number is 103 (decimal).
(The 65C00 series assembler also uses OSWORD 104)

Function &00 - Read IO processor RAM limits

On entry:

YX	00	Function call

On exit:

YX	00
YX+1	OSHWM for IO processor
YX+2	HIMEM for IO processor

Function &01 - Read MMU ROM page

On entry:

YX	01

On exit:

YX	ROM slot of MMU

This call enables the MMU to determine its ROM slot from across the tube and set LINK_ROM_SLOT correctly.

Function &02 - Activate print spooler

On Entry:

YX	02
YX+1	first page of sideways RAM
YX+2	second page of sideways RAM or &FF
YX+3	third page of sideways RAM or &FF
YX+4	fourth page of sideways RAM or &FF

On exit:
No results are returned. If the parameter block is invalid the message "Print spooler failed to initialise" is printed. If the print spooler is active, then it attempts to change the RAM it uses to the new values given in the parameter block but, if the buffer is not empty, then the message "Printing in progress" will be given.

Function &03 - Kill print spooler

On entry:
YX 03

On exit:
No results are returned. If the print spooler was not active, then no action results. If it was active, then the buffer is flushed (as with FX21,3) and control returned to the old MOS routines.

Function &04 - Blow text to editor

Copy all the text from a file in the input buffer on the IO processor into the workspace on the second processor. This routine is used by the editor when EDIT* is entered as an MMU command. It is assumed that the file handler that loaded text into the input buffer put a zero byte at the end of the text.

On entry:
YX 04
YX+1 input buffer start page (on IO processor)
YX+2 editor workspace start page (on second processor)
YX+3 size of editor workspace in pages

On exit:
YX completion flag

The completion flag is positive if the text was transferred successfully. It is set to &FF if the second processor workspace was filled up before all the text was transferred. A good completion does not guarantee that text was copied. It means bytes were copied and a zero byte was found before the editor buffer was filled. The editor must do its own evaluation of the contents.

FUNCTION &05 - SUCK TEXT FROM EDITOR

Copy text from the editor workspace into the input buffer on the IO processor. This routine is used by the MMU editor when QUIT or RUN is typed. The editor must put a zero byte at the end of the text as an end of text marker.

On entry:
YX 05
YX+1 input buffer start page (on IO processor)
YX+2 editor workspace start page (on second processor)
YX+3 size of input buffer in pages

On exit:
YX completion flag

If the completion flag is negative (&FF), then the transfer failed because the text was larger than the input buffer.

Function 128 - Copy block to second processor

This function copies a 1K block of memory, regardless of contents, across the tube into the second processor. The Linker and Assembler use this function when a source file is in the IO processor memory in order to read it. 1K blocks are large enough to keep the system running at optimum speed and small enough to leave plenty of workspace for labels and macros etc.

On entry:
YX &80
YX+1 destination RAM page in second processor
YX+2 source start page in IO processor

On exit:
The parameter block is unchanged.

Function 129 - Copy block to IO processor

This function is used to copy a 1K block from the second processor to the IO processor memory. The Assembler and Linker use it when the output buffer is in the IO processor and they are running in the second processor.

On entry:
YX &81
YX+1 destination page on IO processor
YX+2 source start page on second processor

The parameter block is unchanged.

Print spooler variables

The print spooler variables are held on page 3 of the IO processor in the cassette file workspace. Thus the print spooler cannot be used at the same time as the cassette filing system or any other utility which wants to pinch this memory. All the print spooler routines are in sideways ROM in the MMU chip so only this chip needs to be resident to operate the spooler (this may assist sideways RAM users). The routines are called using extended vectors.

&380 PRINT_P1
The first page of sideways RAM the spooler may use

&381 PRINT_P2
The next page to use or &FF

&382 PRINT_P3
The third page to use or &FF

&383 PRINT_P4
The last page to use or &FF

&384 PRINT_INSERT_PTR
This is a three byte pointer showing where the next byte should be inserted in the print buffer. The buffer is a circular buffer, so this value wraps round as it is continually incremented. The first byte is an index to the RAM page. It takes the values 0,1,2 or 3. A value of 0 means that PRINT_P1 is the RAM page pointed to, a value of 2 means PRINT_P3 is referred to. The next two bytes specify the address in the RAM page between &8000 and &C000. This pointer is incremented after a character is inserted. If it is 1 less than PRINT_REMOVE_PTR then the buffer is full and the character is not inserted. The MOS will call this routine repeatedly until it inserts the character.

&387 PRINT_REMOVE_PTR
This is a three byte pointer in the same format as PRINT_INSERT_PTR showing where the next byte to be removed is stored. If it is the same as PRINT_INSERT_PTR then the buffer is empty. It is incremented after a character is removed. When it is incremented to the same value as PRINT_INSERT_PTR the output buffer empty event is generated. The MOS calls this routine during the centi-second interrupt.

&38B OLD_INSV
The old contents of INSV are kept here and restored when the buffer is killed by ADE OSWORD.

&38D OLD_REMV
The old contents of REMV are kept here and restored when the buffer is killed.

&38F OLD_CNPV
The old contents of CNPV are kept here and restored when the buffer is killed.

&391 P_COUNT
This is the count of characters currently in the buffer. It is incremented when a character is inserted and decremented when a character is removed because subtracting the pointers is a lengthy process. When a 64K buffer is full this value will (incorrectly) be zero. It is only used by the CNPV routine.

&393 PRINT_WRITER
This is a 16 byte routine to write a byte into sideways RAM and is the counter part of OSRDRM.

6 File Formats

This chapter explains the data format of each file type used by the ADE plus MMU, Linker and Librarians and the 65C00 series Macro Assembler.
There are six types of file used by the system:

Text files
Object files
Linker modules
Linker libraries
Macro libraries
Symbol table files

6.1 Text files

Text files are used by the editor, the assembler and the MMU. Text is represented in ASCII format. The editor always *LOADs text files and thus cannot strip line feeds from them. Files created by the editor will not contain line feeds, but other editors may include them. Line feeds must be removed using the FILTER utility provided on the ADE plus disc before a text file containing line feeds can be loaded into the editor. The editor takes CR (ASCII &OD) to be the end of line character. Zero bytes are used consistently as end of text markers but these are always placed by the file loading or reading routine and are not saved as part of the file itself. This makes the text files compatible with all other text based applications on the BBC micro series. When a text file is saved by the editor, the length field in the catalog is correct but the execution address is, of course, meaningless (as is the load address).

The assembler filters out line feeds as it reads the file byte by byte even if it was *LOADed by the assembler's memory management interface routines (see Chapter 7). This means you should have no difficulty assembling files from other editors, including the previous ADE editor.

6.2 Object files

Object files are executable programs output by the assembler or linker. These are in standard Acorn filing system format. The disadvantage of this format is that a file cannot be scatter loaded in different parts of a machine, so data often has to be placed in a separate file from code. The linker use symbol table option can help with this and facilitate good overlay programming techniques, but the limitations of the filing system tend to prevail when, perhaps, compared with an Apple Macintosh.

The load and execution addresses are specified as a full 32 bit address. The assembler uses the MSW pseudo op and the linker uses the A and B options to set these. The assembler accepts tokens for the 65C00 opcodes that would allow an advanced editor to compress the source program somewhat. The tokens are ASCII characters >= 128. They are:

128	BRK	165	BVC
129	CLC	166	BVS
130	CLD	167	BLT
131	CLI	168	BGE
132	CLV	169	BRA
133	DEX	170	AND
134	DEY	171	EOR
135	INX	172	ORA
136	INY	173	ADC
137	NOP	174	CMP
138	PHA	175	LDA
139	PHP	176	SBC
140	PLA	177	ASL
141	PLP	178	LSR
142	RTI	179	ROL
143	RTS	180	ROR
144	SEC	181	DEC
145	SED	182	INC
146	SEI	183	CLR
147	TAX	184	STZ
148	TAY	185	CPX
149	TSX	186	CPY
150	TXA	187	TSB
151	TXS	188	TRB
152	TYA	189	BIT
153	DEA	190	JMP
154	INA	191	JSR
155	PHY	192	LDX
156	PHX	193	LDY
157	PLY	194	STA
158	PLX	195	STX
159	BCC	196	STY
160	BCS	197	BBR
161	BEQ	198	BBS
162	BMI	199	RMB
163	BNE	200	SMB
164	BPL		

Linker Modules

Linker modules are the most complicated file format in the ADE plus system. Understanding these files will unlock a whole new world in which you will be able to write your own compilers that will interface with the assembly language output from ADE and with libraries of functions. The library file format is essentially the same as the linker module format.

A linker file is considered to be a byte stream, just like a source file. This byte stream is broken down into three sections, a header, symbol declarations and the object output of the assembler. The object output is in a series of records. The linker must process the file at least twice. On the first pass it collects the symbol definitions and measures the length of each object module. It also notes all external symbols. Missing external symbols (not specified in any declaration part) are searched for in any libraries listed on the linker command line. A module containing a wanted symbol is included as if it were a separate linker module file. On the second pass RSECT files are combined and the main program is output as all symbol values are now known. The declaration part in each file is skipped during this process. A third pass is necessary if any ASECTs are included as these have to be separately output due to the limitations of the FS file structure mentioned above. This process is quick because the files are much smaller than source files and there is little calculation to do.

The file header

Each linker file begins with a six byte header specifying 'ADELNK' in ASCII. This is used by the linker to verify that it has got a valid module. After the K of ADELNK the module name, specified in the MODULE statement of the source file, is found. This is in ADCII and terminated by CR (&OD). Thus the header is of variable length and always ends in &OD. Each module name in a linking operation must be different. The module name is associated by the linker with a module number (the order in which modules are introduced) and a table kept in memory with details of each module.

The declaration part

The declaration part of the module consists of a number of declaration records. Each record has the following format:

Byte 0	The length of the symbol name being declared or 0 indicating the end of the declarations
Byte 1	Flags for symbol being declared
Bytes 2,3	Value of symbol being declared (depends on flags)
Byte 4	XTRA byte, available for expansion by compilers

Bytes 5 Symbol name, length as defined in byte 0

Bits 0 and 3 of the flags byte are used by the linker. Other bits may be set by the assembler/compiler but are ignored. If bit 0 is set the symbol is relocatable, otherwise it is absolute. The value of an absolute symbol is specified in the value field of the declaration record. If bit 3 is set the symbol is a zero page declaration from the RXP pseudo-op. If the symbol is a zero page declaration, then the value field represents the number of bytes of storage that the label refers to. The address of the label is found by simply adding up the sizes of each RXP symbol starting from 0 (default) or an address specified with the Z option. If the symbol is a program label, then the start address of the RSECT for the current module is added to the symbol's value to make it absolute. The first RSECT starts from OSHWM (default) or the address specified by the A option. Each new RSECT finds its start address by adding up the lengths of preceding RSECTs. A module may contain a number of RSECTs but the total number of section in the progam may not exceed 256.

The object records

Object records define actual bytes of code and data that will be put into the output file. They are well structured in sections. The data format for ASECTs is the same as that for RSECTs. An object record may refer to one external symbol, and this symbol name is included as part of the object record. Each section in the source file produces a corresponding section in the object records. The section starts with a section type record followed by any number of data records (here data refers to code and program data) followed by a section end record.

The section header record is:

&00	End of linker file
&80yyxx	Absolute section with ORG at xxyy
&81yyxx	Relocatable section offset xxyy from start of first RSECT in file

A section may be an empty section. For example an ORG statement begins a new ASECT on the 65C00 macro assembler, so an ASECT statement followed by an ORG produces two sections in the linker file one of which will be empty (and have a meaningless address field).

The two bytes following an ASECT or RSECT section header are called the address field. In an ASECT this is the origin of the code. For an RSECT this value is actually the offset

from the start of the file but is ignored by the linker since it has arrived at the same value by adding up the lengths preceding RSECTs anyway. It could be used by checking routines.

A section data record is:

&00	End of section record. Every section has at least one of these, even an empty section.
&01..&0F	Absolute data records. This type of record is followed by 1..15 bytes of absolute data that is inserted unaltered into the output. This type of record would be produced by a STR statement for example, but also by instructions to specify the opcode byte.
&10	The byte prefix. A word record follows (3 bytes plus possible external symbol). Extract the lower 8 bits from it and output only one byte. On pass two give an error message if the value is outside the range -127 to +255.
&20	The double byte prefix. A word record follows. Swap the upper and lower bytes and then output two bytes.
&30	Define storage record. Three bytes follow. Bytes 0 and 1 define the amount of storage in lo-hi format (may be zero). Byte 2 defines the 'fill' value to place in the output at this point.
&4n	External word record. This record outputs two bytes unless it is preceded by a byte or double byte prefix. The record consists of an absolute part which is usually zero and a symbol reference. The lower four bits of the record type are modifiers that tell the linker what to do with the record. Bytes 0 and 1 following the record type are the absolute part of the value (the starting point) and byte 2 following the value is the length of the external name. The external name follows immediately in ASCII.
&40	Add the external symbol's value to the absolute part giving a 16 bit value.
&42	Subtract the external symbol's value from the absolute part giving a 16 bit value.

&44	Add the external symbol's value then make the high byte zero, but still output 16 bits. (Eg from a LDX#>EXTERNAL_REF statement)
&46	Subtract the external symbol's value and make the high byte zero, output 16 bits. (Eg from LDX#>0-EXTERNAL_REF)
&48	Add the external symbol's value and shift right 8 bits to give the value DIV 256. (Eg from LDX#<EXTERNAL_REF)
&4A	Subtract the external symbol's value and shift right 8 bits. (Eg from LDX#<0-EXTERNAL_REF)
&8n	A relocatable word record. This record outputs two bytes unless preceded by a byte or double byte modifier. Two bytes follow this record giving an absolute value to which the relocation constant (the linker's location counter at the start of the section) is added or subtracted. The lower four bits specify modifiers as in type &4n records.
&80	Add relocation constant to absolute part.
&81	Subtract relocation constant from absolute part.
&84	Add relocation constant and make top 8 bits zero. (Eg from LDX#>REL_SYMBOL)
&85	Subtract relocation constant and make top 8 bits zero. (Eg from LDX#>0-REL_SYMBOL)
&88	Add relocation constant and value right 8 bits. (Eg from LDX<REL_SYMBOL)
&89	Subtract relocation constant and shift value right 8 bits. (Eg from LDX#<0-REL_SYMBOL)

Note that the bit assignments for external symbols and relocatable symbols and the add or subtract bit in each case, are completely separate. This allows compilers, in theory, to mix external and relocatable records with a type &Cn record. The ADE plus assemblers provided by SYSTEM do not allow this mixing but the linker does. The linker processes the external reference before processing the relocation constant and any overflow outside of 16 bits is ignored.

Summary of bit assignments in linker data records:

bit 0	Subtract relocation constant if bit 7 set
bit 1	Subtract external reference if bit 6 set
bit 2	Set top 8 bits to zero after processing external and relocation parts
bit 3	Shift right 8 bits after processing external and relocation parts
bit 4	Set by DB and #operators in assembler
bit 5	Set by DDB statement in assembler
bit 6	Set if an external reference follows
bit 7	Set if word is relocatable

Bit 6 or 7 is always set except in absolute data records in which case the above assignments do not apply.

Linker Library files

The LLIB command in the MMU makes a linker library out of the specified input files by concatenating them. The hard work of identifying which module to include is done by the linker. In a library file each module name must be different. The header for a library is still ADELNK so a single linker module file can be used as a library.

Macro Library files

Macro libraries are fairly sophisticated in ADE plus, in that they work by random access to speed up operation. Each macro library has a 2K catalog at the start of the file, built up by the macro librarian, MLIB. The librarian contains no options for deleting or inserting into an existing catalog but there is no reason why users should not write a utility to do this since the structure of the random access file is fairly simple. In order to have a good number of macros in a reasonably sized catalog, the file pointers in the catalog are 16 bits and extended to 32 bits by the assembler using them with the addition of leading zeros. This means that a macro library cannot be bigger than 64K.

When a library is specified in an MACLIB statement the 2K catalog is read in on the software stack below RAM_TOP. The assembler searches for unknown or 'GET' opcodes from the beginning of the catalog. When a name is found the file pointer for the maco library is positioned and the text for the macro is read in as if it had been defined in the source code. Macro text is held in the workspace RAM along with labels and every other kind of object the assembler uses (see Chapter 7).

The file begins with the six bytes "MACLIB" followed by an entry for each macro.

Macro library catalog

"MACLIB" {not repeated}

Byte 0	Length of macro name or 0 for end of catalog names
Byte 1	LSB file pointer for macro text
Byte 2	MSB file pointer for macro text
Byte 3	LSB length of macro definition
Byte 4	MSB length of macro definition
Bytes 3..	Macro name length as specified by byte 0

This format is repeated. The number of macros that will fit in the catalog depends on the name length. The end of file is specified by a record with zero in the name length byte.

The text stored at 'file pointer' for each macro starts with the line following the MACRO statement and ends with an ENDM line.

6.6 Symbol File format

The linker allows you to dump the linker symbol table to a file which can be used from a symbolic debugger, included in a future linking operation or used in any user application. The example program DS, on the ADE plus disc, shows how such a file is used. The first six bytes of the file contain the characters 'ADESYM' from which the file is identified. The full format of a symbol file is:

"ADESYM" {not repeated}

Byte 0	length of symbol name
Byte 1	LSB symbol's absolute value
Byte 2	MSB symbol's absolute value
Bytes 3..	Symbol name, length as specified in byte 0

The end of file is specified by a record with a zero in the name length byte.

Assembler objects and variables

This chapter lists the types of objects manipulated by the 65C00 assembler.

The assembler stores all its objects in linked lists. Each list begins with a pointer in zero page or the language workspace and ends with a zero, or null, pointer. The following types of objects are stored.

Symbols
Local symbols
Macro texts
Block markers
In line strings
The software stack

There is a list for local symbols originating in zero page, but this is complicated by block markers. The start of the local symbol list changes as the program proceeds. There is a separate list for each initial character of the other symbol names. This is rather like BASIC stores its variables but is further speeded up by the inclusion of the length as part of the name so that searching is restricted to symbols with names of the required length.

The list of pointers is held on page 5, starting at &500. The first letter of each symbol is implicit, so the actual symbol stored in memory is minus this character. The symbols are stored as:

Bytes 0,1	Link to next symbol in this list or zero
Byte 2	Length of symbol name (less 1)
Byte 3	Symbol flags
Bytes 4,5	Symbol value or pointer
Byte 6	XTRA byte for linker expansion
Bytes 7..	Remainder of name

The flag bits are:

bit 0	Relocatable symbol
bit 2	Macro definition
bit 3	EQUated symbol
bit 4	ENTry symbol
bit 5	EXTernal symbol
bit 6	inline string
bit 7	forward reference bit

Bit 7 is set on pass one when the symbol is defined and reset on pass two when the symbol is 're-defined'. At the same time the value of program labels is checked in pass 2 and phase errors are reported. If the symbol is a macro name or an inline string then the value field points to the text of the macro or string. In the case of an external symbol the value is set to zero, to give correct expression results for the linker.

The assembler uses all of zero page from 0 to 8F and all of the language ROM workspace except for the MMU variables. The temporary copy of the assembler options is at &6E0. Source lines are assembled on page 7 at &700. If the input and output buffers are not in main memory then the assembler allocates a 1K temporary buffer at WATER_MARK for each far buffer. Thus if you break into the assembly process you will find the source text in up to three places. Needless to say the speed performance of the assembler has been optimised for large files that can be buffered in far memory.

Example programs

Example intelligent file read routines

The first example is perhaps the most important. It gives part of a set of routines which will be part of each application that uses the MMU. These routines are always inlcuded with the application because of the requirements for maximum speed of operation. The code shown does not form a complete program but illustrates the code needed to allocate the MMU buffers to your application and open and read an input file. A set of output routines is also needed, the structure of which completely mirrors the input routines. This particular code is taken from the linker.

The program behaves very intelligently. Allocate_buffers is called as part of the application's start-up procedure. This sets OLD_WATER_MARK to the first page of free memory after it has allocated some workspace for itself. Then to read a file, a call is simply made to OPEN_SOURCE with YX pointing to the name. Calls to CHAR_GET return each character in the file byte by byte and set carry when EOF is encountered. Apart from the fact that this routine does not preserve registers it is the same as OSBGET, so once you have the code in your application you can forget about where the source file is and get on with the important job of processing it.

```
***  Allocate buffer space based on MMU map  ***

;  set up six variables to describe IO files

          DSECT
          ORG       &500        ; or wherever
source_stat  DS     1
source_start DS     1
source_size  DA     1
object_stat  DS     1
object_start DS     1
object_size  DS     1
old_water_mark DS 1
          DEND

;  This routine initialises both the input and the
;  output buffer descriptors

Allocate_buffers  ENT
          LDX       #0
;  point input buffer
          LDY       #0
          LDA       WATER_MARK
;  save MMU water mark
```

```
        STA     old_water_mark
:loop   LDA     #far
; assume buffer is far
        STA     source_stat,Y
        LDA     INPUT_BUFFER_ALLOCATED,X
        BPL     :ilocal
; buffer allocated, so ...
        LDA     INPUT_BUFFER_PAGE,X
        CMP     #&11            ; is input far?
        BEQ     :1
        LDA     #near
; input is on same side of tube
        STA     source_stat,Y
        LDA     INPUT_BUFFER_START,X
        STA     source_start,Y
        LDA     INPUT_SIZE,X
        STA     source_size,Y
        BNE     :cont
; unless buffer is 0 pages...
:ilocal LDA     #near
; buffer is near and local
        STA     source_stat,Y
        LDA     old_water_mark ; bump up by 1 K
        STA     source_start,Y
        LDA     #4
        STA     source_size,Y
        STA     INPUT_SIZE,X
; for MMU reader routines
        CLC
        ADC     old_water_mark
        STA     old_water_mark ; next free RAM
:cont   INY
        INY
        INY
        INX
        INX
        INX
        INX
        CPX     #8
        BCC     :loop
; do output buffer then exit
        RTS


*** OPEN FILE FOR INPUT ***

; on entry YX point to the name of the file
; set source stat as follows
; bit 0 set if file *LOADED or read in fully
; bit 6 set if file *LOADed, else need to close
; bit 7 set if input is far

        DSECT
source_handle   DS  13          ; OSGBPB block
osfile_fcb      DS  18          ; OSFILE block
        DEND
```

```
            BLOCK

Open_source     ENT
            LDA     #0
            STA     source_handle
            STX     osfile_fcb
            STY     osfile_fcb+1
            LDX     #>osfile_fcb
            LDY     #<osfile_fcb
            LDA     #5                  ; read dir info
            JSR     OSFILE
            CMP     #1
            BEQ     :open1
            ERR     1,"File not found"

:open1      LDA     osfile_fcb+12
            ORA     osfile_fcb+13   ; len >64K?
            BNE     read_file
            LDX     osfile_fcb+11   ; size in pages
            LDA     osfile_fcb+10   ; plus a bit?
            BEQ     :3
            INX                     ; yes!
:3          CPX     input_size
; will fit in RAM?
            BEQ     :4
            BCS     read_file       ; no, use OSGBPB

; load whole file

:4          LDA     osfile_fcb+10   ; save length
            STA     source_count
            LDA     osfile_fcb+11
            STA     source_count+1
            LDA     INPUT_BUFFER_START
            STA     osfile_fcb+3
            LDX     #0
; load on page boundry
            STA     osfile_fcb+2
            STA     osfile_fcb+6
            BIT     source_stat     ; far load?
            BPL     :5
            DEX
; load at FFFFpage on far IOP
:5          STX     osfile_fcb+4
            STX     osfile_fcb+5
            LDX     #>osfile_fcb
            LDA     #255
            JSR     OSFILE          ; load file
            LDA     source_stat
            ORA     #&41
; set load and eof bits
            STA     source_stat
            JSR     Set_source_pointer
; init reader pointer
            RTS

; Read file in blocks with OSGBPB

read_file LDX   osfile_fcb
            LDY     osfile_fcb+1
            LDA     #&40
```

```
; open file for input
          JSR     OSFIND
          STA     source_handle
; must be ok cos dir info was

; gozzinta...

*** Read next block of source into input buffer

READ_SOURCE       ENT
          LDA     INPUT_BUFFER_START
          STA     source_handle+2
          LDX     #0
          STX     source_handle+1
          STX     source_handle+5
          STX     source_handle+7
          STX     source_handle+8
          STX     buffer_count
          LDA     INPUT_SIZE      ; pages to read
          STA     buffer_count+1
          STA     source_handle+6
          BIT     source_stat     ; far read?
          BPL     :1
          DEX
:1        STX     source_handle+3
          STX     source_handle+4
          LDX     #>source_handle
          LDY     #<source_handle
          LDA     #4                ; read sequential
          JSR     OSGBPB
          JSR     Set_source_pointer
          LDA     source_handle+5
; see if whole file copied in
          ORA     source_handle+6
          BEQ     :2
          INC     source_stat     ; set eof bit
          LDA     buffer_count
; and adjust count
          SEC
          SBC     source_handle+5
          STA     buffer_count
          LDA     buffer_count+1
          SBC     source_handle+6
          STA     buffer_count+1
          RTS

*** set text pointer to start and
*** if necessary read a 1K block

Set_source_pointer ENT
          LDA     INPUT_BUFFER_START
          STA     source_pointer
          LDA     #0
          STA     source_blocks_read
          STA     text_blocks_read
          BIT     source_stat
          BPL     Set_text_pointer
; source is near.. else
```

```
; Read far block accross tube

Read_far_block   ENT
          LDA    #128
          STA    temp_cb
          LDA    source_start    ; near page
          STA    temp_cb+1
          LDA    source_pointer
          STA    temp_cb+2
          CLC
          LDA    #4
; next page address
          STA    text_blocks_read
          ADC    source_pointer
          STA    source_pointer
          JSR    Set_text_pointer

; gozzinta ..

Do_ADE_OSWORD
          LDX    #>temp_cb
          LDY    #<temp_cb
          LDA    #120
          JSR    OSWORD
          RTS

*** set text pointer and offset to
*** start of local buffer

Set_text_pointer ENT
          LDA    #0
          STA    text_pointer
          LDA    source_start
          STA    text_pointer+1
          RTS

*** Read character from source ***

; This routine behaves like a fast OSBGET

          BLOCK

Char_get  ENT
          LDA    buffer_count
          ORA    buffer_count+1
          BNE    :cg1
          SEC
          RTS                    ; EOF

:cg1      LDY    #0
          LDA    (text_pointer),Y
          INC    text_pointer
          BNE    :1
          PHA
          INC    text_pointer+1
          INC    source_blocks_read
          LDA    source_blocks_read
          CMP    input_size
; read all buffer?
          BCC    :2
          LDA    source_stat     ; done all?
```

```
                AND   #1
                BNE   :eof
                JSR   Read_source
;   else read next block
                PLA
                CLC
                RTS

;eof            LDA   #0
;     clear buffer count
                STA   buffer_count
                STA   buffer_count+1
                PLA
;     return last valid char
                CLC
                RTS

:2              BIT   source_stat
                BPL   :3                ; near
                DEC   text_block_read
                BNE   :3
                JSR   Read_far_block ; get next 1K
:3              PLA
:1              DEC   buffer_count
                INC   buffer_count
                BNE   :4
                DEC   buffer_count+1
:4              DEC   buffer_count
                CLC
                RTS
```

8.2 Example advanced editor "patch"

This routine is a patch for View Version A1.4. You should be able to follow it through and examine your favourite word processor to patch it in the same way. The routine relies on View bieng loaded either into sideways RAM or into a ROM and then run on the second processor because it stores the MMU variables in the ROM address space. A QUIT command is added to View which returns control to the MMU with an 'ADE_CONT' type restart (see Section 2). The QUIT command is added by using the ADE debugger to find out where View calls OSWORD 0 to get a command line and replacing this call by another routine which gets the line and checks for QUIT before returning to VIEW. This program does **not** suck and blow the text from the workspace into the input buffer, but you could amend it to do so though this is not trivial. (A look at a disassembly of the MMU editor might help here, all the OSWORD 120 routines and variables used have been documented to this guide).

View A1.4 has at least 4K of free memory between &B000 and &C000, so this is a good place to put the patch. The patch was implemented as follows (addresses may be different on your version of View):

Determine which ROM slot the view ROM is in, say slot 13,
*LBUG 13 to load the ADE plus debuffer with the View ROM
in the address space.

```
8000M <tab>  ...should see 'View' at &8009
S8000<cr>
C000<cr>
2000<cr>     move a copy of the ROM to RAM at &2000
2009M        should see 'View' again
"ADEED"
00<cr>       (quotes needed) ... alter ROM title to ADEED
2001M        adjust language entry point to go to patch, note
             current address
             and call this VIEW_START (&8107 in version
             A1.4)
00<cr>       set address to &B000
B0<cr>
*LOAD patch 5000<cr> ... load the patch routine at the correct
             address: This routine will actually run at &B000
*SAVE ADEED 2000+4000 8000 8000<cr>
             save a ROM image on disc.
```

The ROM image can be loaded into sideways RAM and will be found by ADE plus. It will correctly respond to EDIT <file name> commands.

The code for "patch" is shown below. To use the debugger to find the address PATCH_ADR, set the memory pointer to &8000 and enter:

```
G
&20
&F1
&FF
<cr>
```

On View A1.4 this sets the memory pointer to &827B where we see JSR OSWORD after a parameter block has been set up to read a source line to page 5. Thus PATCH_ADR is &827C. This patch also resets the fill and justify flags each time you go to command level because these are not wanted when editing a progam. On Version A1.4 they were found by experimentation at &4F and &50.

```
          TTL    Patch to View A1.4
; Absolute assembly:
; to generate patch file, save as T.PATCH then
; ASM PATCH=T.PATCH
;-----------------------------------------------

PATCH_ADR EQU    &827C
; could change for other versions
CODE_ADR  EQU    &B000
; B000 - C000 not used by View 1.4
VIEW_START EQU   &8107

SCREEN_MODE EQU  &462
LINK_ROM_SLOT EQU &401

          ORG    CODE_ADR

; control transfered here when the ROM is entered
; as a language after being patched at &8001.
; It is assumed we are in sideways RAM or
; have been copied across the tube.

          PHA
          TXA
          PHA                    ; save entry regs
          CLI
; (IRQ disabled on entry)
          JSR    SAV_VAR         ; save MMU vars
          LDX    #0
; put command in kbd buffer
:0        LDA    LOAD,X
          BEQ    :1              ; insert "LOAD "
          JSR    INSERT
          INX
          BNE    :0
:1        LDX    #0
; get file name from MMU com line
:10       LDA    &700,X
; first look for end of EDIT cmd
          CMP    #13
; no file name supplied
          BEQ    :cancel
          CMP    #32             ; delimiters
          BEQ    :gap
          CMP    #"."
          BEQ    :gap
          INX
; carry on over EDIT, ED. etc
          BNE    :10
:gap      INX                    ; skip delimiters
          LDA    &700,X
; to avoid strain on kbd buffer
          CMP    #32
          BEQ    :gap
          CMP    #13             ; any file name?
          BEQ    :cancel         ; no
:2        JSR    INSERT
; insert file name in kbd buf
          INX
          LDA    &700,X
```

```
        CMP     #13             ; at end of name?
        BNE     :2
        LDX     #0
; now insert FX command
:3      LDA     FXC,X
        BEQ     :done
        JSR     INSERT
        INX
        BNE     :3
:done   PLA
        TAX
        PLA
        JMP     VIEW_START      ; go start view

; if no file name given,
; clear kbd buffer and start

:cancel LDA     #21
        LDX     #0
        JSR     OSBYTE
        JMP     :done

; data to put in kbd buffer

LOAD    ASC     "LOAD "
; must have space at end
        BRK

FXC     DATA    13,"*FX125",13,0
; terminate file name, cause escape

*** Insert character in Kdb buffer ***

INSERT  TAY
; char to insert (Y reg lost)
        TXA
        PHA
        LDX     #0              ; buffer number
        LDA     #138
        JSR     OSBYTE          ; FX138,0,char
        PLA
        TAX
; preserve X over INSERT
        RTS

*** Save MMU variables in ROM address space    ***
*** And intercept View command line entry call ***

SAV_VAR LDX     #0
:s1     LDA     &400,X
        STA     &BF00,X
        INX
        BNE     :s1             ; save a page
        LDA     #>GET_LINE
; patch View OSWORD 0 call
        STA     PATCH_ADR
        LDA     #<GET_LINE
        STA     PATCH_ADR+1
        RTS
```

```
*** Routine to intercept command line entry
*** Check for QUIT command

          BLOCK
GET_LINE  PHA                    ; save A
          LDA    #1
; reset Fill and Justify
          STA    &4F
          LDA    #&FF
          STA    &50
          PLA
          JSR    OSWORD
; get line at &500
          BCS    :RTS            ; escape pressed
          PHP                    ; save all regs
          PHA
          TXA
          PHA
          TYA
          PHA
          JSR    TEST_QUIT
; was it a QUIT command
          PLA
; returned, so it wasnt
          TAY
          PLA
          TAX
          PLA
          PLP
:RTS      RTS
; processes command

TEST_QUIT LDX    #&FF
          LDY    #0
:1        INX
          LDA    &500,X
          CMP    #32
; skip leading blanks
          BEQ    :1
:2        AND    #&5F            ; upper case
          CMP    :QC,Y
          BNE    :RTS
          INX
          LDA    &500,X          ; next char
          INY
          CPY    #4
          BCC    :2

; perform QUIT command

          LDX    #0
:3        LDA    &BF00,X
; restore MMU variables
          STA    &400,X
          INX
          BNE    :3
          LDA    #22
; restore ADE screen mode
          JSR    OSWRCH
          LDA    SCREEN_MODE
```

```
            JSR   OSWRCH
            LDX   LINK_ROM_SLOT
            LDA   #142
            JMP   OSBYTE          ; warm start ADE

:QC         ASC   "QUIT"
```

A linker module disassembler

This BASIC program disassembles a linker module file, output from a module assembly, into the different types of linker data records. It will help explain the structure of linker files and will be a useful debugging aid if you go on to write any language compilers for the ADE plus system. The program could be extended to include a machine code disassembler as well.

```
10  REM LINKER MODULE FILE DISASSEMBLY V1.0
20  GOSUB 160
30  PROCload
40  PROCverify
50  insect=TRUE
60  REPEAT
70    PROCdeclare
80  UNIT NOT insect
90  insect=TRUE
100 REPEAT
110   PROCsection
120 UNTIL NOT insect
130 PROCclose
140 HIMEM=T%
150 END

160 REM initialise
170 T%=HIMEM
180 HIMEM=TOP+&400
190 P%=HIMEM
200 CLS
210 PRINT "Linker file disassembly"'
215 out$=STRING$(80," "):in$=STRING$(40," ")
            :W$="    "
220 RETURN

230 DEFPROCload
240  INPUT "Enter filename",F$
250 in=OPENIN (F$)
260 IF in=0THEN PROCerr("File not found")
270 RAM=TRUE
280 IF EXT#in>T%-HIMEM THEN RAM=FALSE
290 IF RAM THEN CLOSE#in
          :OSCLI("LOAD "+F$+" "+STR$~P%)
300 ENDPROC
```

```
320 DEFPROCverify
330  LOCAL ok
340  ok=TRUE
350  FOR I%=1 TO 6
360   IF FNget<>ASC(MID$("ADELNK",I%))
          THEN ok=FALSE
370  NEXT
380  IF NOT ok THEN PROCerr("Not a linker file")
390  PRINT "File: "F$"  Module: ";
400  REPEAT
410   I%=FNget:VDU I%
420  UNTIL I%=13
430  PRINT
440 ENDPROC

460 DEFFNget
470  IF RAM THEN P%=P%+1:=?(P%-1) ELSE =BGET#in

490 DEFPROCerr(E$)
500  PRINT 'E$'
510  CLOSE#0
520  END
530 ENDPROC

550 DEFPROCdeclare
560  L%=FNget
570  IF L%=0 THEN insect=FALSE:ENDPROC
580  F%=FNget AND 9
590  V%=FNget+256*FNget
600  X%=FNget
610  out$=""
620  FOR I%=1 TO L%
630   out$=out$+CHR$FNget
640  NEXT
650  IF F%=8 THEN PRINT "Zero page symbol:";
660  IF F%=1 THEN PRINT "Relative symbol :";
670  IF F%=0 THEN PRINT "Absolute symbol :";
680  PRINT out$;TAB(32);~V%
690 ENDPROC

710 DEFPROCsection
720  S%=FNget
730  IF S%=0 THEN insect=FALSE:ENDPROC
740  O%=FNget+256*FNget
750  IF S%=128 THEN PRINT '"ASECT at      :";~O%
760  IF S%=129 THEN PRINT '"RSECT offset :";~O%
770  inrec=TRUE
780  REPEAT
790   PROCrecord
800  UNTIL NOT inrec
810 ENDPROC

830 DEFPROCrecord
840  in$="":out$=""
850  R%=FNgetin
860  IF R%=0 THEN inrec=FALSE:ENDPROC
870  IF R%<16 THEN PROCdata
880  IF R%=&10 THEN PROCbyte
890  IF R%=&20 THEN PROCddb
900  IF R%=&30 THEN PROCds
910  IF R% AND &40 THEN PROCext
```

```
920  IF R% AND &80 THEN PROCrel
930  REPEAT
940   IF LEN in$ <8 THEN in$=in$+" "
950  UNTIL LEN in$=8
960  PRINT in$;": ";out$
970 ENDPROC

990 DEFFNgetin
1000  LOCAL c%,h$
1010  c%=FNget
1020  h$=STR$~c%
1030  IF LEN h$=1 THEN h$="0"+h$
1040  in$=in$+h$
1050  IF LEN in$=8 THEN PRINT in$;":":in$=""
1060 =c%

1080 DEFPROCdata
1090  out$="DATA "
1100  FOR I%=1 TO R%
1110   PROCbval
1120   out$=out$+W$
1130  NEXT
1140 ENDPROC

1160 DEBPROCbyte
1170  PROCword
1180  out$="DB   "+W$
1190 ENDPROC

1210 DEFPROCddb
1220  PROCword
1230  out$="DDB  "+W$
1240 ENDPROC

1260 DEFPROCds
1270  PROCwval
1280  out$="DS   "+W$
1290  PROCbval
1300  out$=out$+","+W$
1310 ENDPROC

1330 DEFPROCword
1340  LOCAL R%
1350  R%=FNgetin
1360  IF R% AND &40 THEN PROCextl
1370  IF R% AND &80 THEN PROCrell
1380 ENDPROC

1400 DEFPROCextl
1410  PROCwval
1420  L%=FNgetin
1430  IF R% AND 2 THEN W$=W$+"-" ELSE W$=W$+"+"
1440  FOR I%=1 TO L%
1450    W$=W$+CHR$FNgetin
1460  NEXT
1470  PROCmodify
1480 ENDPROC

1500 DEFPROCrell
1510  PROCwval
1520  W$=W$+"'"
```

```
1530  IF R% AND 1 THEN W$=W$+"(-)"
1540  PROCmodify
1550 ENDPROC

1570 DEFPROCwval
1580  W$=STR$~(FNgetin+256*FNgetin)
1590  REPEAT
1600   IF LEN W$<4 THEN W$=W$+" "
1610  UNTIL LEN W$=4
1620 ENDPROC

1640 DEFPROCbval
1650  W$=STR$~FNgetin
1660  IF LEN W$=1 THEN W$=" "+W$
1670 ENDPROC

1690 DEFPROCext
1700  PROCext1
1710  out$="DW    "+W$
1720 ENDPROC

1740 DEFPROCrel
1750  PROCrel1
1760  out$="DW    "+W$
1770 ENDPROC

1790 DEFPROCclose
1800  PRINT '"End of file"'
1810  IF RAM THEN ENDPROC
1820  CLOSE#in
1830 ENDPROC

1850 DEFPROCmodify
1860  IF R% AND 4 THEN W$=">"+W$
1870  IF R% AND 8 THEN W$="<"+W$
1880 ENDPROC
```