32000/ASSEMBLER

REFERENCE
MANUAL

32000 ASSEMBLER

ACORN SCIENTIFIC

PART NO 0410, 005
ISSUE NO 1
JULY 1985

© Copyright Acorn Computers Limited 1985

Neither the whole or any part of the information contained in, or the product described in, this manual may be reproduced in any material form except with the prior written approval of Acorn Computers Limited (Acorn Computers).

The product described in this manual and products for use with it, are subject to continuous developments and improvement. All information of a technical nature and particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith.

In case of difficulty please contact your supplier. Deficiencies in software and documentation should be notified in writing, using the Acorn Scientific Fault Report Form to the following address:

Sales Department
Scientific Division
Acorn Computers Ltd
Fulbourn Road
Cherry Hinton
Cambridge
CB1 4JN

All maintenance and service on the product must be carried out by Acorn Computers' authorised agents. Acorn Computers can accept no liability whatsoever for any loss or damage caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this manual, or any incorrect use of the product.

Published by Acorn Computers Limited,
Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN.

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

NOTE: A User Registration Card is supplied with the hardware. It is in your interest to complete and return the card. Please notify Acorn Scientific at the above address if this card is missing.

ISBN 0 907876 36 6 Acorn Scientific

Contents

1 Introducing the Acorn 32000 assembler

This document is a reference guide to the Acorn 32000 macro assembler. It is not a tutorial guide, and therefore the reader is assumed to be familiar with:

32000 Assembly language.
This is described in the *Instruction Set Reference Manual* which is available from dealers. Although the mnemonics used by the assembler are to the National standard, pseudo-operations (assembler directives) are specific to the Acorn Assembler.

The Panos operating system.
The use of the operating system environment under which the assembler runs is described in a document supplied with the system, the *Panos Guide to Operations* and the *Panos Programmer's Reference Manual.* The user is also assumed to know how to use the command to call the assembler.

Acorn Object Format (AOF)
is mentioned frequently in this guide; the output produced by the assembler will usually be in this form, although knowledge of the details is unlikely to be required. A full description of AOF is given in the *Panos Technical Reference Manual.* See also the various User Guides for introductory material.

Features of the assembler include:

- complete support of the NS32000 instruction set including Memory Management and Floating Point extensions.
- support of all nine categories of the general addressing modes of the NS32000.
- two types of object file:
  1. an image in Acorn Object Format suitable for linking into a Panos relocatable image using the system linker.
  2. a simple binary image suitable for immediate execution from the Pandora * prompt.
- powerful macro defining capability. The user may define macro instructions in the source which may be called to insert common

sequences of 32000 mnemonics or assembler directives. Macros may call other macros, and recursion is possible.

- conditional assembly. The ability to assemble parts of the source conditionally is made even more useful by the ability to set 'flag' symbols on the command line so that different versions may be assembled from the same source file.

1.1 Installation

The assembler is supplied on a 5¼ inch floppy disc in Acorn DFS format. This needs to be installed even if it is intended for use in conjunction with the DFS. Refer to the appropriate User Guide supplied with the hardware for details about installing the assembler.

1.2 Assembler commands

This section summarises the arguments of the assembler command. See the beginning of chapter 2 for a breakdown of the metasyntax used here.

(-source) filename (-asm)
: This names the source file to be assembled. The extension '-asm' will be appended if no other extension is given. Multiple files may be assembled using the `CHAIN` directive.

-list (name)
: An assembly listing may be sent to a file called source-lis, or to another named file or device. The format of the listing is described in section 1.4. See figure 1 for a demonstration of this option.

-error (name)
: Assembly errors are reported to the initial error stream by default (i.e. the messages usually go to the screen). The 'error' argument names an alternative destination for errors.

-aof (filename)
: The output from the assembler is put into a file source-aof by default. The 'aof' argument allows an alternative file to be named. Note that the file may not in fact be an AOF file, if the assembly was carried out in absolute or relative binary mode.

-opt options

Several options are provided to change the behaviour of the assembler. These are described in section 1.3 below.

-get "mapping {, mapping}*"

This argument is used to specify a mapping between filenames specified in GET and CHAIN directives, and the actual filename to be used. The word 'get' is followed by a string in double quotes which is a comma-separated list of mappings from GET (or CHAIN) names to filenames. For example:

```
-> asm32 -source fred -get "fpStuff=fp-asm,debug=db-asm"
```

With this mapping, a "GET fpStuff" directive would access file fp-asm.

-identify

Specifying this argument causes the assembler to print its version number.

-help

Specifying this argument causes the assembler to produce a summary of the arguments which may appear on the command line.

1.3 Assembler options

The -opt argument is followed by a list of letters which are used to flag various options. A flag letter preceded by a '+' enables the option; a '-' sign disables. The exception is '$', which is followed by the name of the symbol to be set or reset. Note that if the first option letter is preceded by a '-', then the whole option string must be enclosed in double quotes e.g. -opt "-l-m".

c Usually upper and lower case are treated as distinct characters in identifiers. Quoting opt +c causes cases to be equated upon reading each source line, so that fred and FRED are the same symbol.

l Usually source files are loaded into store during the first pass (if there is enough space) to minimise disc accesses on subsequent passes. This occasionally causes the assembler to run out of room. Quoting opt -l will disable loading and thus prevent the no room error (unless there genuinely isn't enough memory for the assembly).

m By default the assembler will try to optimise the size of the output file by taking many passes over the source. Giving the opt -m option causes the assembler to make only enough passes to resolve symbol references, at the expense of producing non-optimally sized output code. This option only applies when absolute binary rather than AOF is generated.

p Usually the assembler produces 'packed' style AOF files. Quoting opt -p causes general format AOF files to be produced.

$ This option is followed by a name to be set to TRUE. This name may be accessed in a conditional assembly (IF) directive in the source. For example, -opt $debug sets the symbol 'debug' to TRUE (-1). Following the name by a single quote, e.g. -opt $debug' sets the symbol to FALSE (0).

As implied by the descriptions above, the default state of the options is: +LMP-C.

1.4 Assembler listing format

An assembly listing is produced if required by giving the -list argument on the command line. It has the following format:

llllll b1 b2 b3 b4 b5 b6 nnnn text.....

where:

llllll
is the value of the location counter at the start of the code for the line, printed as a 6-digit hex number.

nnnn
is the source line number.

b1..b6
are the byte values (in hex) of the generated codes. b1 is at the lowest address. Spaces are printed if less than 6 bytes were generated;

Extra bytes are displayed on following lines in the form:

llllll b7 b8 b9 etc.

with at most 6 bytes per line, and llllll being the address of the first byte on each extra line. Lines which came from a macro expansion in the source are marked with a + character at the start of the line.

At the end of the assembly, the assembler sends the following statistics to the output stream, which is the vdu by default (only if the global string Program$Verbosity is set to greater than 1 - see the *Panos Guide to Operations*):

- The number of errors detected.
- The total size (in bytes) of the area(s).
- The number of passes required

Incorrect lines are echoed to both the listing file and the error file. Errors are reported using textual messages printed out before the failing line. Figure 1 gives an example of an error message from an assembly within the Panos editor. The source can be seen in the background, the assembly command appearing in the top window, with the error message contained in the lower window.

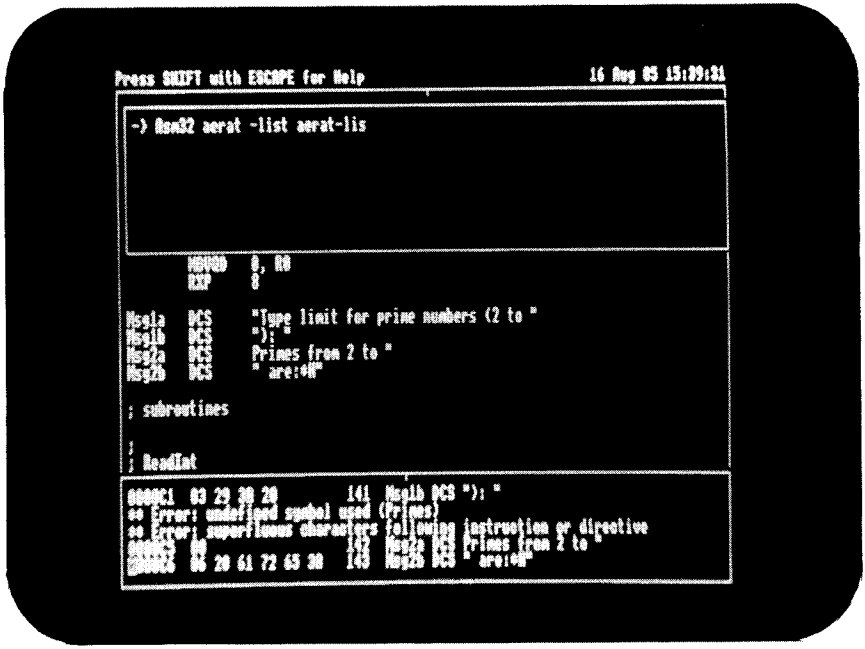

Figure 1 Assembler error message

2 32000 assembler source format

The Acorn 32000 assembler accepts standard National Semiconductor instruction mnemonics, and in addition provides a full set of pseudo-mnemonics (assembler directives) and the ability to define macro instructions.

A source program is a sequence of lines which may contain 32000 assembly language mnemonics, assembler directives, comments, or nothing at all.

Within this document a meta-syntax is used to describe the syntax of assembler source lines. In this meta-syntax, the characters `{ , } , | , *` and `'` have special meanings:

{x}	means 0 or 1 occurrences of x
{x}*	means 0 or more occurrences of x
{x|y}	means 1 occurrence of x or 1 occurrence of y
'c'	where c is a single reserved character, means the literal character c, i.e. any special meaning is disabled. If c is not a single character or not a reserved character then ' stands for itself.
name	is a syntax class-name (i.e. lower case text. Upper case text is used for literal items, e.g. MOVQD, END).

All other symbols stand for themselves.

2.1 Format of source lines

The format of a source line is:

```
{label} {mnemonic {operand {,operand}*}} {;comment}
```

If a label is present, it must start at the beginning of a source line. Any mnemonic must be preceded by at least one space. A comment may start at any position on the line; it is marked by a semi-colon and continues up to the end of the line. There must be at least one space between a mnemonic and any following operands, but no space need precede a comment.

Operands are separated by commas and may contain spaces. These are ignored, except within string constants. Expressions are therefore allowed to

contain blanks, which are ignored. However, spaces are not allowed in tags (see later), numeric constants, and compound symbols such as >=.

A source line may contain up to 255 characters. The assembler will stop with an error if more than this number of characters occur without a line-break, since this would suggest an erroneous source file, e.g. a file which is not a text file.

2.2 Character set

The character set consists of letters (upper and lower case), digits, the underscore character (_) and other special characters. Upper and lower case are distinct, except in instruction mnemonics, directives, and macro names. The use of option c will cause the assembler to equate upper and lower case in identifiers.

2.3 Symbols

Symbols consist of letters, digits and underscores, starting with a letter or underscore. Symbols are significant to 63 characters.

A relocatable symbol is one which is defined as a label in a relocatable area. All other symbols are either absolute or external.
Some symbols are reserved and so cannot be redefined. These are:

```
R0, R1, .. R7,  F0, F1, .. F7
TOS, EXTERNAL, FP, SP, SB, PC
The MMU registers
```

Mnemonics, e.g. END and MOVQB, are allowed as label names however, so lines such as:

```
END  END
```

are allowed but not advisable.

Note that the letters used in the option field of the string instructions (`MOVSi, CMPSi` etc.) and the `SETCFG` instruction are not reserved; they are marked by the fact that they appear in this specific context (inside square brackets).

2.4 Constants

Integers may be given as unsigned decimal numbers or in the forms #Xhhhh, #Bbbbb, #Odddd, for hexadecimal, binary and octal representations respectively. Note that the letters A through F used in hex numbers may occur in either upper or lower case. An alternative representation of hexadecimal numbers is the form :hhhh .

All integers are interpreted as 32-bit quantities.

Floating point constants are optionally signed and have an optional exponent. Examples are:

```
100
1.1
-.1
-1e4
1.234E-1
```

Floating point constants are allowed only in the `DCF` and `DCL` directives, and as floating point immediate operands.

String constants are delimited by single quotes in the simple case, and double quotes to generate a counted-string form, i.e. the bytes in the text preceded by a byte containing the length of the text. The `DCB` directive (described later) accepts either form, creating the appropriate stream of bytes.

Character constants are also valid in integer expressions, where their length is limited to 4 characters in the simple form, and 3 characters in the counted-string form, the length byte forming part of the value in the latter case.

The value of a multi-character constant as an integer is calculated using the same store interpretation adopted by the 32000 architecture, i.e. the least significant byte is the byte at the lowest address, which is the leftmost character of a string, or the length byte in counted strings.
Hence:
`'A' = #X41, "A" = #X4101, 'AB' = #X4241, "AB" = #X424102 etc.`
Within string constants, the asterisk * is used as an escape character. If the character that follows is an N or n, then the actual byte value stored at the

current position is determined by the value of the NLSYM option (see directive descriptions later); the default value is 10 (ASCII LF = NL).

If the following one or two characters are valid hex digits, then the number they represent is planted as a byte value. This enables the simple insertion of control characters within strings. For example A*N generates the bytes #X41,#X0A; *03*FEA generates #X03,#XFE,#X41. If neither of these cases holds, the following character is planted without interpretation; hence a single asterisk is represented in a string as **. Because this mechanism allows the representation of the newline character in strings, it is forbidden for strings to cross line boundaries.

The special symbol '$' is used to stand for the program counter. For example:

```
BR  $       ;infinite loop
```

2.5 Expressions

All expressions are calculated to 32 bits and overflow is ignored. Evaluation is ordered according to the priority below, and left-to-right for operators of the same precedence. Bracketed sub-expressions are evaluated first. The arithmetic and comparison operators treat their operands as signed quantities; the 6 operators in the latter group return TRUE (-1) or FALSE (0).

Operator	Priority	Functions
-	8	Unary minus
~	8	Bitwise complement (unary)
<<	7	Logical left shift (0s shifted in from right)
>>	7	Logical right shift (0s shifted in from left)
&	6	Bitwise AND
|	6	Bitwise OR
~	6	Bitwise exclusive OR
*	5	Multiply
/	5	Divide (as defined by QUOD instruction)
%	5	Remainder (Modulus - as defined by REMD instruction)
-	4	Subtract
+	4	Add

=	3	Equal-to
<>	3	Not-equal-to
<	3	Less-than *
>	3	Greater-than *
<=	3	Less-than-or-equal-to *
>=	3	Greater-than-or-equal-to *
!	2	Conditional NOT
&&	1	Conditional AND
|	1	Conditional OR

The comparison operators marked * perform signed comparison.
Note: the only operators which may have one or both operands relative (or external) are + and - (unary and binary). Relative + relative-relative evaluates to relative. In object module (AOF) mode, two relative operands must have the same relocation base (i.e. they must be defined as labels in the same area).

2.6 Mnemonic conventions

The assembler accepts all standard National Semiconductor instruction mnemonics (as described in the *Cambridge Series Instruction Set Reference Manual*), including flaoating point unit (FPU) and memory management unit (MMU) instructions.

The normal 32000 operand forms are accepted for all 'general' type operands, with the following conventions:

- An expression on its own is normally treated as a code-area address and is assembled as a PC-relative operand (or (SB) or EXTERNAL when the assembler is in AOF mode). The type of the expression must match the current code-area type, i.e. an absolute expression will be faulted in a relocatable area. This rule also applies to branch-type operands, i.e. of 'disp' class.

- Immediate mode operands are specified by preceding an absolute expression with the equals-sign =. In the case of floating point immediates, only a constant may follow the =, not an expression.

- Absolute operands are specified by prefixing an absolute expression with the at-sign @.

- Operands for a 'quick' type argument must be absolute expressions, optionally prefixed by an equals-sign.

In addition, the following special cases are accepted, as shown by these examples:

```
MOVSW     [U,B]
ENTER     [R0, R1, R3], 24
RESTORE   [R0-R4] ; (all registers between R0 and R4 inclusive)
SETCFG    [I]
CMPSD     []
```

3 Assembler directives

This chapter describes the directives acted upon by the assembler. Most of these are general purpose and may occur anywhere in the source. Others are specific to the production of Acorn Object Format files and should only be used after a MODULE directive.

3.1 Absolute and relocatable modes

At any time the assembler is in one of three modes - absolute, relocatable or AOF. The default mode is absolute. In any assembly, one (and only one) of the directives ABSORG, RELORG or MODULE may occur, at most once. If one does occur, it must be before any code or data has been generated, or any label defined, otherwise it is treated as an error.
The form of these directives is:

```
ABSORG   expression
RELORG   expression
MODULE   name
```

The value of the expression must be absolute, and defined by the time the directive is first encountered. It may not change between passes. The effect of the directive is to set the assembler into the specified mode, and the location counter to the value of the expression.

The MODULE directive sets AOF mode, and the optional name is planted in the output file as the module name. Once in AOF mode the assembler will allow the special directives described in section 3.3.

3.2 Standard directives

The following directives are handled by the assembler. As noted in the section on symbols, they may occur in either, or any combination of, upper or lower case, as may the names of user-defined macros and instruction mnemonics.

ABSORG	RELORG	GET	CHAIN	EQU	EQUR	SET	NLSYM
IF	ELSE	ELIF	FI	IFDEF	IFNDEF	MACRO	MEND
DCB	DCW	DCD	DCS	DCF	DCL	ALLOCB	ALLOCW
ALLOCD	ALIGN	END	ENTRY	TITLE	OPTIONS		

plus MODULE directives described in section 3.3

The directives listed in the table are now described in turn, apart from ABSORG and RELORG detailed above.

3.2.1 GET

Syntax: `GET filename`

On encountering this, the assembler suspends processing of the current file and starts to read input from file 'filename' (or the file to which this name is mapped via the -get command line option. See section 1.2). The name should be enclosed in single quotes as above. On reaching the end of the file specified, processing resumes at the point in the first file where it had been suspended. GET may be used in a MACRO definition, but note that the GET operation happens when the macro is expanded, not when the body is read in. A file read in by this means may itself contain a GET directive, but the level to which this process may recurse is dependent upon the state of the Panos I/O environment, the limitations of the filing system involved, and the assembler itself which has a restriction of five levels.

3.2.2 CHAIN

Syntax: `CHAIN filename`

This is similar in effect to GET, except that the current file is closed and processing continues with the named file. This directive will commonly occur at the end of a source file, if the text of the program is too large to fit conveniently in a single file. CHAIN may not occur in a MACRO definition.

3.2.3 EQU

Syntax: `ident EQU expression`

Defines symbol 'ident' to have the value of the expression, which may be absolute or relocatable, but not external.

3.2.4 EQUR

Syntax: `ident EQUR register_name`

Defines symbol 'ident' to be a synonym for the named integer or floating point register (which may itself have been defined by `EQUR`).

3.2.5 SET

Syntax: `ident SET expression`

This directive has the same effect as `EQU`, except that a symbol defined using `SET` may be redefined using `SET`, i.e. the identifier is an assembler 'variable'.

3.2.6 NLSYM

Syntax: `NLSYM expression`

This sets the value (in the range 0 to 255) to be planted on encountering the character pair `*N` in a string constant. The initial value is 10 (ASCII LF = NL).

3.2.7 IF

Syntax:

```
IF expression_1
                section_1
ELIF expression_2
                section_2
ELIF expression_3
                section_3
                .
                .
ELSE
                section_n
FI
```

The `IF..ELIF..ELSE..FI` construct is directly analogous to the same construct in high-level programming languages, but here is used to control the conditional assembly of different sections of code according to whether the expressions evaluate to TRUE (not 0) or FALSE (0). Note that these constructs may be nested, with the obvious interpretation, and that any or all of the `ELIF` and `ELSE` clauses may be omitted. The section which is assembled (if any) must be the same on all passes of the assembler. To ensure this, the expressions must be absolute, must not contain any forward references, and must not use any symbols set as labels or derived from labels. Note that if the compact code option has been disabled (using opt -m on the command line), the label restriction does not apply. IF constructions must not be split over source files or macros.

3.2.8 IFDEF and IFNDEF

Syntax:
```
IFDEF symbol_name
IFNDEF symbol_name
```

The `IFDEF` and `IFNDEF` directives are alternatives to `IF` in the general conditional assembly constructs. If the named symbol has been defined on the current assembly pass by the time that an `IFDEF` directive is encountered, the effect is the same as that with "IF true-expression". `IFNDEF` provides the converse effect - i.e. for when the symbol has NOT been defined on the current pass.

3.2.9 MACRO...MEND

Syntax:

```
MACRO
macro_call_template
macro_body
  .
  .
  .
MEND
```

The directive MACRO introduces the definition of a textual macro. It appears by itself on a (possibly commented) source line. The macro_call_template looks like:

```
{%label}  macro_name  {param_def {, param_def}*}
```

The macro_name may be the same as an existing instruction, directive or macro. To access the old instruction from within the macro definition, it should be preceded by a @. For example, within a macro called MODULE, the MODULE directive must be written @MODULE.

A param_def looks like:
%param_tag {=default_value}

A param_tag is defined as one of the following items:

(a) an ordinary symbol name
(b) a 1- or 2-digit decimal number
(c) a single asterisk *

The label field is optional, and if present must start at the beginning of the line. It has the same syntax as an ordinary symbol, preceded by a percent sign.

The macro_name must be supplied - its syntax is that of ordinary symbols, with the exception that it will be recognised in any combination of upper and lower case.

The parameter list, if present, follows after at least one space. There may be 0 or more parameter definitions. The parameter tags may be any combination of types (a) and (b) above, or 0 or more of type (a) followed by a single parameter of type (c). A parameter of type (c) is used for passing arbitrary lists of items.

The default value of a parameter, if supplied, is a piece of text which will be treated as having been supplied in the actual call if that parameter was omitted. It has the same syntax as actual parameters (see below). Note that a parameter of type (c) may not take a default value. If, for a given parameter, no value is supplied at the call, and there is no default value, the parameter is treated as a null string '' .

When a macro using a type (c) parameter is called, it is as if the macro had a parameter list ending with the sequence %1, %2, %3, ... where the number of such parameters in the formal list matches the number of such parameters in the call.

When a macro name is enountered during assembly, its call template is matched against the current line in the source file, and the parameters assigned, with appropriate defaults. The label field is treated specially, in that if there is a label on the line containing the macro call, but there is no label in the macro call template, then the label is treated in the normal way and defined as a symbol whose value is that of the location counter at the start of the line. If however there is a label field in the macro template, then the text of the actual label is assigned to the label parameter and is not entered as a symbol at this point.

Actual parameters are treated as uninterpreted textual information. All spaces in parameters are removed, except for those occurring within quotes. The text of the parameter is preserved in respect of the case of letters and the occurrence of special characters. To get a space into a parameter the whole parameter must be placed in quotes. Note that quote characters surrounding a parameter are stripped during processing - when the parameter is substituted during expansion the quotes will not appear. To get a quote character into a parameter, the parameter is enclosed in quotes and two quotes are used (as in string constants). Note that the contents of a quoted parameter are NOT treated as if it was a string - e.g. `'*N'` is not translated into a newline character.

Assembly then continues with the source text being read out of the body of the macro, rather than from the source file. On encountering a '%' character during reading of a macro, the assembler reads the next item which should be a `param_tag` as described above. It is an error for the item not to be a tag defined in the call of the current macro, unless the assembler is currently treating its input as a comment, in which case this error is ignored.

In order to increase the usefulness of macros, it is possible to concatenate a parameter with a following piece of text which would otherwise be taken as part of the parameter name. If a parameter is followed by a full-stop (.) character, the assembler will ignore it but terminate its parsing of the parameter name. Hence if `%T1='B'`, `%dest='R0'`, `%source='STEP(SB)'` then:

```
MOVZ%T1.D  %source, %dest
```

would expand to

```
MOVZBD  R0, STEP(SB)
```

There is a special case of parameter instantiation which is related to type (c) parameters. If the parameter `%*` occurs within the macro body, it expands to the parameters %1, %2, %3.. separated by commas. In addition if `%*` is immediately followed by a 1- or 2-digit number (N say), it expands to parameter %N, %N + 1 etc. These forms are only valid in macros which have a `%*` type parameter in the call template.

The item `%#param` expands to the number of characters in param as a textual string, which is useful for testing for null string parameters:

```
IF %#string = 0
.....
```

Also `%(absexpr)` converts absexpr into a decimal string representation of the value of the expression. The expression may not include forward references or macro parameters. However such an expression could be SET to a label before using `%(label)`.

In addition to normal parameters created on macro call, the assembler maintains two psuedo-parameters connected with macros. These are:

%MCOUNT - Macro count.

> This parameter when substituted returns an integer (as text) which is the number of macro calls made so far on this pass, up to and including the point at which the macro currently being expanded was called. On each macro call, a global variable is incremented, and assigned to a (local) psuedo parameter which is entered into the symbol table with the other parameters for this macro. Hence within a given macro expansion, the value of %MCOUNT may be used to create local labels, if some simple naming convention is adopted.

%PCOUNT - Parameter count.

This expands to the number of numeric parameters which were created using a type (c) parameter in the current macro. Its main use (as for %*) is in the writing of recursive list-processing macros, which may handle an arbitrary number of parameters. Note that %PCOUNT does NOT include any normal type (a) parameters preceding the %*. It is an error to instantiate %PCOUNT unless a macro with a type (c) parameter is currently being expanded.

Here are a few examples of the use of macros to demonstrate these points:

```
MACRO
    MOVC   %N, %dest        ; move constant to double-word
IF ((%N)>=-8) && ((%N)<=7) ; in range -8..7
    MOVQD   %N, %dest
ELIF ((%N)>=-#XC0000000) && ((%N)<=#X3FFFFFFF) ; OK in up to 30-bit
                                                ; disp field
    ADDR    \%N, %dest
ELSE
    MOVD    =%N, %dest
FI
MEND

MACRO
     Case_Table  %Type, %Base, %*
IF %PCOUNT = 0
    ; end of table
ELSE
    DC%Type  %1-%Base               ; dump 1 element
    Case_Table %Type, %Base, %*2    ; and do the rest of the list
FI
MEND

  MACRO
%Name   ENUM  %Base, %*     ; enumerate list of names as constants.
    IF %PCOUNT > 0
%Name.%1  EQU %Base         ; define this one
%Name   ENUM  %Base+1, %*2  ;  enumerate rest
    FI
  MEND
```

Example of use:

```
Colour_   ENUM   0, black, red, green, blue, white
```

defines Colour_black as 0, Colour_red as 1, etc.

3.2.10 DCB

Syntax: `DCB expression [, expression]*`

This directive causes the planting of bytes of data into store. Each expression is either an absolute integer expression, or a string expression. The null string '' is permitted, for which no bytes are planted. It is an error to plant an integer expression not in the range -128 to 255. The counted-string form may also be used by enclosing the characters to be planted in double quotes. This puts a length byte followed by the actual text.

3.2.11 DCS

Syntax: `DCS expression [, expression]*`

This directive is identical in effect to `DCB`.

3.2.12 DCW

Syntax: `DCW expression [, expression]*`

This directive causes the planting of words of data into store. Each expression must be absolute, and evaluate to an integer in the range -32768..65535.

3.2.13 DCD

Syntax: `DCD expression [, expression]*`

This directive causes the planting of double-words into store. Each expression must be an absolute integer expression.

3.2.14 DCF

Syntax: `DCF fpconst [,fpconst]*`

This causes the planting of list of single precision (four-byte) floating point constants in memory.

3.2.15 DCL

Syntax: `DCL fpconst [,fpconst]`

This acts as `DCF` but plants double precision (eight-byte) constants.

3.2.16 ALLOCB

Syntax: `ALLOCB expression [, expression ]`

This directive reserves a number of bytes of store as determined by the value of the first expression (which must be $>= 0$). The essential effect is to add the value of the expression (which must be absolute) to the location counter. The optional second parameter is the value which will be deposited in each byte of the reserved area. If it is omitted, the value of the allocated bytes will be undefined.

3.2.17 ALLOCW

Syntax: `ALLOCW expression [, expression ]`

This is as for `ALLOCB`, but reserves store in units of words, i.e. it adds twice the value of the expression to the location counter. The optional second parameter is the value which will be deposited in each word of the reserved area.

3.2.18 ALLOCD

Syntax: `ALLOCD expression [, expression ]`

This is as for `ALLOCB`, but reserves store in units of double-words, i.e. it adds four times the value of the expression to the location counter. The optional

second parameter is the value which will be deposited in each double word of the reserved area.

3.2.19 ALIGN

Syntax: `ALIGN expression {, expression}`

This directive is used to set the location counter on an address boundary. The first expression given must evaluate to an absolute, positive quantity, N, which is a power of 2 (i.e. 2,4,8,16 etc). The effect of the directive is to ensure that the location counter is positioned at the next address which is 0 mod N. This is achieved by planting between 0 and N-1 bytes, as padding.

In place of the first expression, the words `BYTE`, `WORD`, `DOUBLE`, or `QUAD` may be used. These stand for 1, 2, 4, and 8 bytes respectively.

The default value used for padding is 0, but if the second expression is supplied, then its value will be used - it must evaluate to an absolute integer in the range -128 to 255. For example in a code area the value #XA2 might be used - this is the one-byte machine instruction NOP.

3.2.20 ENTRY

Syntax: `ENTRY`

In AOF mode, this defines the entry point (at the current location) which is looked for by the linker to determine the 'root' module of an image. In absolute or relative mode, this sets the current location as the execution address of the binary output file produced.

3.2.21 END

Syntax: `END {expression}`

This directive serves two purposes:

- When it is encountered during the processing of an included file (through the use of `GET`), the assembler closes that file and resumes processing the one it was reading from when the `GET` occurred. No expression may be present in this case.

- If no GET was in progress then it causes the current pass of the assembler to complete. If the expression is present then it defines the entry point. This is an alternative to the use of ENTRY; only one of these mechanisms may be used in an assembly.

It is a fatal error in the latter case for END to occur as a directive within the body of a macro. It is also incorrect for a final END to occur while there are open conditional assembly blocks (this implies that a FI has been missed out).

3.2.22 TITLE

Syntax: TITLE text

This directive is followed by a string which is subsequently printed at the top of each page of assembly listing. In addition the directive causes the listing to move to the top of the next page (a form-feed is sent to the listing file). An example is:

```
TITLE Low-level graphics support routines.
```

3.2.23 OPTIONS

Syntax: {label} OPTIONS { {=|+|-}value}*

where label is an optional label, and value is one of:

IFS	Conditional assembly directives
LIST	Global listing (outside of MACRO and IF)
MDEF	Macro definitions
MEXP	Macro expansion
SKIP	Code skipped by IF

If the value is preceded by +, the class of item controlled by that word is enabled in the listing. If it is preceded by -, that part of the listing is disabled. If = (or nothing) precedes the value, the listing will contain only items of that class. Examples are:

```
OPTIONS LIST  ; Only the global listing
OPTIONS -MDEF ; Turn off macro definition listings
OPTIONS +SKIP ; List code skipped in IFs
```

If the label is present, it is assigned (as with SET) the previous value of the OPTIONS, for use in a later directive, e.g.:

```
oldopt OPTIONS -LIST ; force listing off
....
....
       OPTIONS oldopt ; restore previous state
```

If OPTIONS -LIST is used at the first line of a GET macro library file, and OPTIONS +LIST used on the last line, then no part of the library file will appear in the listing.

3.3 Object module directives

As mentioned at the start of this chapter, the MODULE directive is used to put the assembler in AOF mode. This section describes the directives which are used in this mode.

3.3.1 MODULE

Syntax: `MODULE {{'name' | name}}`

This directive defines the external name to be given to the module. It must occur at most once in the assembly, obeying the same positioning rules as ABSORG and RELORG. It overrides the -m option since compact code is *always* produced.

The name is optional and only has to be enclosed in quotes if it contains a semi-colon, or multiple, or leading spaces.

3.3.2 AREADEF

Syntax:

```
AREADEF namedef [{attribute {,attribute}*}],alignment
```

This allows the user to create and specify the attributes of an area of store into which code and/or data may be planted using the assembler's normal mechanisms. The namedef parameter has the syntax:

```
[name | name = 'external_name']
```

where the second form is used to permit an arbitrary, externally visible name, but is restricted to use with areas marked as COMMON or COMDEF (see below).

The alignment parameter is either an absolute expression which must be a power of 2, or one of the keywords BYTE, WORD, DOUBLE, or QUAD (standing for 1, 2, 4, and 8 respectively). These keywords are recognised only in this context and in the ALIGN directive.

Note that in AOF mode, the ALIGN directive will only accept alignment values smaller than or equal to that specified in the AREADEF directive for the area in which it is used.

The attribute keywords which may be present are defined below. They fall into five groups; at most one keyword from each group may occur in the list (in any order). If no keyword from a given group occurs, the first keyword in that group is the default and will be assumed present. Keywords are recognised in any combination of upper and lower case letters.

DATA / CODE

Defines the use to which information in this area will be put. At most one area may be defined as being a code area.

WRITE / READ

READ specifies that the area should be protected against write access (if possible - this depends on the presence of the MMU). WRITE indicates that the area must be made writeable.

NOPIC / PIC

PIC stands for Position Independent Code. When applied to a code area it indicates that this area contains such code (i.e. re-entrant, pure, and containing no relocation). When applied to a data area it causes the assembler to fault any attempt to generate a relocated object in the area.

PRIVATE / SHARED

Specifying SHARED allows the linker to arrange run-time sharing of the area across different processes using this module. Otherwise the area will only be accessible within a single process space.

CONTIG / COMMON / COMDEF

Defines whether this area will be contiguous with other areas of the same type (CONTIG), or overlap them (COMMON and COMDEF). COMDEF indicates that this module defines the common area named, rather than simply referencing it. For COMMON and COMDEF, the area name may have a different external name, as mentioned above.

Use of this directive also defines a relocatable symbol of the same name as the area at the start of the area (offset 0).

Associated with each area is a location pointer. The only way in which this is changed is by dumping items (including instructions) or allocating space (using ALLOCi, ALIGN etc.) while the area is currently selected.

There is one predefined area. The attributes of this area are as if it had been declared by:

```
AREADEF  , [code], byte   ; null name special to this area
```

This area is special in that although it is marked as the code area, this may be overridden by the user declaring another area to be the code area. This area is the one which is selected at the start of each assembly pass.

3.3.3 AREA

Syntax: `AREA [name]`

This directive selects the area into which items will be dumped in the normal way (i.e. by creating instructions or data). The parameter, if present, must be the name of an already declared area. The effect of the command is to set the location counter to the last-reached point within the named area (which is 0 if the area has not been previously selected). If no name is given then the default area is selected.

3.3.4 AREALEN

Syntax: `label AREALEN name [, offset]`

This assigns the length of the named area to the label. If the offset parameter is present, this is added to the length before the label is assigned. The label may be used only in a general operand or ADDRESS directive, as EXTERNAL mode is used to refer to it.

3.3.5 AREAEND

Syntax: `label AREAEND name [,offset]`

This acts as `AREALEN` except that the end (plus optional offset) address of the area is assigned to label.

3.3.6 EXPORT and EXPORTC

Syntax:
```
EXPORT  namedef [,namedef]*
EXPORTC namedef [,namedef]*
```

These two directives are used to make symbols external. The syntax of `namedef` is that given in the description of `AREADEF`. `EXPORT` defines a symbol as being data or absolute, according to whether it was defined by the use of `EQU` (absolute) or as a label (data). `EXPORTC` defines each given parameter as an external code item - it must have been defined as a label in the code area. At most one `EXPORTC` or `EXPORT` command may be applied to any particular name. The `namedef` syntax allows external names to be completely general. These directives must not be labelled.

3.3.7 IMPORTC

Syntax: `IMPORTC namedef [,namedef]*`

`IMPORTC` defines a symbol as being an external code item descriptor, i.e. one which may be used as the operand in a `CXP` instruction. It takes a list of `namedef` parameters, i.e. identifiers with an optional equivalence string. Each name given is defined here, and must not be defined in any other way. A name so defined will normally be used as a `CXP` operand, but may be used in the context of a general operand, in which case external addressing mode will be generated (this would generally be meaningful only as the first operand in an `ADDR` instruction). There must be no label on a line containing this directive.

An extended form of `namedef` is permitted in this context:

```
int_name=['mod']'ext_name'
```

where `int_name` is the internal name of the imported item, i.e. the one to be used in this assembly, `mod` is the (optional) name of the module from which

the symbol is to be imported, and ext_name is the external name of the item to be imported.

3.3.8 IMPORT

Syntax: `IMPORT namedef {, namedef}*`

This directive is similar to IMPORTC, but each external symbol so referenced will be set up in the link table by address (or by value, if the symbol is defined as a constant) rather than as a code descriptor. Names so defined may be used only as general operands, and will generate external addressing mode. (Note that it is possible to specify an offset from such a symbol, as part of this mode). The assembler does not create a link entry for these symbols immediately, but on the first time a symbol is used as an operand; hence external symbols which are not used, or are used only in an ADDRESS directive (or in CDESC after an IMPORTC) will not needlessly take up a link table entry. This directive must not have a label.

The extended form of namedef is also allowed here (see IMPORTC).

3.3.9 HANDLER

Syntax: `HANDLER`

This directive must be unlabelled and marks the entry point of the Panos condition handler code for the module being assembled at the current location. It must be in the code area. See the *Panos Programmer's Reference Manual* for further details on Condition Handler.

3.3.10 SPECSB and DEFSB

Syntax:
```
DEFSB position
SPECSB position
```

One of these directives is used to define the location of the static base for this module. The parameter may be one of three types:

1. an absolute expression
2. an expression evaluating to an address in some area
3. the name of an IMPORT ed symbol {+offset}

Case 1 is used if the static base (SB) should point at some absolute store address. Case 2 is used to set the SB within any declared area. Case 3 is used to set the SB to be relative to the global symbol named. DEFSB may occur at most once in an assembly. If SB-relative addressing is used at all, the SB will normally be defined in this way. If case (b) is used, the assembler will optimise code references to labels in the area in which the SB is defined, so that they use SB-relative addressing, rather than external mode. If the statement:

```
MOVD 6(ABC),R0
```

is encountered, and if ABC is in the SB area, the assembler will generate X(Y(SB)) addressing. This optimisation is disabled, if necessary, by the use of SPECSB instead of DEFSB. These directives may not be labelled.

3.3.11 ADDRESS

Syntax: `ADDRESS expression [, expression]*`

This directive is similar in format to DCD, but causes the assembler to generate (link-time) relocatable objects, rather than constant ones. An expression is one of:

A relocatable address
A symbol defined using IMPORT, i.e. an external name [+offset].
An absolute expression

The assembler generates commands in the output file which instruct the linker to relocate each doubleword when the address of the item is known. The last type of item needs no relocation, but may occur here for convenience - the effect is as for DCD.

3.3.12 CDESC

Syntax: `CDESC expression [, expression]*`

CDESC creates a code descriptor for a local or external code item. If the parameter is a label in the code area, then a local code descriptor will be created; otherwise it must be the name of a symbol appearing in a preceding IMPORTC directive. In either case, the assembler generates commands in the output file to relocate the item at link time.

3.3.13 LINKNO

Syntax: `label LINKNO symbol`

This assigns the link table number allocated to symbol to label. `symbol` must be an external, and must have been defined in an `IMPORT` or `IMPORTC` directive prior to the use of `LINKNO`.

3.4 Treatment of labels

The following points should be borne in mind regarding the assembler's treatment of labels when producing AOF output:

1. The occurrence of a label reference to an area other than the code area generates external-mode addressing, unless the area contains the static base, when SB-relative addressing is generated. This of course applies only in the context of general operands within the code area - it is illegal to reference such labels as PC-relative operands, e.g. BSR or BR targets.

2. `CXP` may take three types of operand:

   The name of a symbol defined using `IMPORTC`. This generates a standard external entry.

   A label in the code area. This causes the assembler to set up a local code descriptor entry in the link table for the label concerned.

   `EXTERNAL` (absexpr). A reference to the external object identified by the given link table number. This should have been determined by the `LINKNO` directive.

Index

Acorn Computers Limited
Scientific Division
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
Telephone 0223 245200