MCC PASCAL

26-9-86

[illegible]

METACOMCO

METACOMCO plc welcomes comments and suggestions about its software from users. Please write to our offices at:

26 Portland Square, Bristol, BS2 8RZ, England.

5353 Scotts Valley Drive, Scotts Valley, CA 95060, USA.

MCC PASCAL 68000

Contents

Table of Contents

Preface

MCC Pascal 68000 for the Atari ST is a powerful package incorporating a full screen editor, a one-pass Pascal compiler which conforms to the ISO standard, and a linker. In addition, there are libraries which allow access to GEM graphics. Each of these components is described in the following five sections of the manual.

As this manual is primarily intended to serve as a reference manual, each topic is usually presented in full technical detail as it occurs. Some reference to topics not yet encountered is therefore unavoidable, but some care has been taken to keep these references to the minimum and to make specific cross reference wherever necessary.

Making a backup copy

It is strongly recommended that you make a backup copy of the supplied disk before using the programs.

Section A: The Screen Editor

The Screen Editor

Introduction

The Metacomco screen editor ED may be used on the Atari ST computer to create a new file or to alter an existing one. The text is displayed on the screen, and can be scrolled vertically or horizontally as required.

After inserting and opening the disk, the editor can be invoked by double-clicking on ED.TTP and typing the required parameters. The format is

```
[FROM] <file>,[SIZE <n>]
```

where <file> is any file name.

An attempt is made to open the file specified as FROM, and if this succeeds then the file is read into storage and the first few lines displayed on the screen. Otherwise a blank screen is provided, ready for the addition of data to the new file. The text buffer used to hold the file may be altered by specifying a suitable value as the SIZE keyword. The default is 16500 words, which is sufficient to edit a file of about 60K bytes. It may be increased as required (subject to the amount of available memory).

If invalid arguments are given then ED will re-prompt for the command line giving the user an opportunity to exit from Gem.

When the editor is running the bottom line of the screen is used as a message area and command line. Any error messages are displayed here, and remain displayed until another editor command is given.

Editor commands fall into two categories - immediate commands and extended commands. Immediate commands are those which are executed immediately, and are specified by a single key or control key combination. Extended commands are typed in onto the command line, and are not executed until the command line is finished. A number of extended commands may be typed on a single command line, and any commands may be grouped together and groups repeated automatically. Most immediate commands have a matching extended version.

Control key combinations are described as CTRL/N meaning control N, and some keys are described by the dedicated key on the keyboard. In some cases dedicated function buttons on the ST keyboard can be used instead of the control key combinations if required.

The editor attempts to keep the screen up to date, but if a further command is entered while it is attempting to redraw the display, the command is executed at once and the display will be updated later, when there is time. The current line is always displayed first, and is always up to date.

The mouse of the ST is not supported by the editor and therefore should not be used during an editor session. It should also be noted that the non-standard underline key (next to BACKSPACE) is not supported.

Immediate commands

Cursor control

The cursor is moved one position in either direction by the cursor control keys. If the cursor is on the edge of the screen the text is scrolled to make the rest of the text visible. Vertical scroll is done a line at a time, while horizontal scroll is done ten characters at a time. The cursor cannot be moved off the top or bottom of the file, or off the left hand edge of the text.

The HOME key will take the cursor to the right hand edge of the current line unless the cursor is already there, in which case it will be moved to the left hand edge of the line. The text will be scrolled horizontally if required. In a similar fashion CTRL/E (or key F10) places the cursor at the start of the first line on the screen unless already there, in which case it is placed at the end of the last line on the screen. It should be noted that HOME takes no account of margin settings.

The control combinations CTRL/T (or key F9) and CTRL/R (or key F8) take the cursor to the start of the next word or to the space following the previous word respectively. The text will be scrolled vertically or horizontally as required. The TAB key moves the cursor to the next tab position, which is a multiple of the tab setting (initially 3).

Inserting text

Any letter typed will be added to the text in the position indicated by the cursor, unless the line is too long (there is a maximum of 255 characters in a line). Any characters to the right of the text will be shuffled up to make room. If the line exceeds the size of the screen the end of the line will disappear and will be redisplayed when the text is scrolled horizontally. If the cursor has been placed beyond the end of the line, for example by means of the TAB or cursor control keys, then spaces are inserted between the end of the line and any inserted character.

The RETURN key causes the current line to be split at the position indicated by the cursor, and a new line generated. If the cursor is at the end of a line the effect is simply to create a new, empty blank line after the current one. Alternatively CTRL/A (or INSERT) may be used to generate a blank line after the current, with no split of the current line taking place. In either case the cursor is placed on the new line at the position indicated by the left margin (initially column one).

If the cursor is at the beginning of a line, a RETURN has the effect of creating a blank line before the current line. The cursor will stay at the beginning of the current line. This can be used to generate a line above the first line of a file.

A right margin may be set up so that RETURNs are automatically inserted before the preceding word when the length of the line being typed exceeds that margin. In detail, if a character is typed and the cursor is at the end of the line and at the right margin position then an automatic newline is generated. Unless the character typed was a space, the half completed word at the end of the line is moved down to the newly generated line. Initially there is a right margin set up at column 80. The right margin may be disabled by means of the EX command (see later).

If text has been entered in the wrong case, the control combination CTRL/F (or key F5) will change the case. Move the cursor to the offending character and press CTRL/F, any lower case letters will then change to upper case, and upper case letters will change to lower case.

Deleting text

The BACKSPACE key deletes the character to the left of the cursor and moves the cursor left one position unless at the start of a line. The text will be scrolled if required. CTRL/N (or DELETE) deletes the character at the current cursor position without moving the cursor. As with all deletes, characters remaining on the line are shuffled down, and text which was invisible beyond the right hand edge of the screen may now become visible.

The action of CTRL/O (or key F2) depends on the character at the cursor. If this character is a space then all spaces up to the next non-space character on the line are deleted. Otherwise characters are deleted from the cursor, and text shuffled left, until a space is found. The CTRL/Y command (or key F4) deletes all characters from the cursor to the end of the line. The CTRL/B command (or key F3) deletes the entire current line.

Scrolling

Besides the vertical scroll of one line obtained by moving the cursor to the edge of the screen, the text may be scrolled 12 lines vertically by means of the commands CTRL/U (or key F6) and CTRL/D (or key F7). CTRL/D (or key F7) moves to previous lines, scrolling the text down; CTRL/U (or key F6) scrolls text up moving to lines further on in the file. The CTRL/V command rewrites the entire screen, which is useful if the screen is altered by another program besides the editor.

Repeating commands

The editor remembers any extended command line typed, and this set of extended commands may be executed again at any time by simply typing CTRL/G (or key F1). Thus a search command could be set up as the extended command, and executed in the normal way. If the first occurrence found was not the one required, typing CTRL/G (or key F1) will cause the search to be executed again. As most immediate commands have an extended version, complex sets of editing commands can be set up and executed many times. Note that if the extended command line contains repetition counts then the relevant commands in that group will be executed many times each time the CTRL/G key (or F1 key) is pressed.

Extended commands

Extended command mode is entered by pressing the ESC key. Subsequent input will appear on the command line at the bottom of the screen. Mistakes may be corrected by means of DELETE in the normal way. The command line is terminated by either ESC or RETURN. In the former case the editor remains in extended mode after executing the command line, while in the latter case it reverts to immediate mode. An empty command line is allowed, so just typing RETURN after typing ESC will return to immediate mode.

Extended commands consist of one or two letters, with upper and lower case regarded as the same. Multiple commands on the same command line are separated from each other by a semicolon. Commands are sometimes followed by an argument, such as a number or a string. A string is a sequence of letters introduced and terminated by a delimiter, which is any character besides letters, numbers, space, semicolon or brackets. Thus valid strings might be

```
/happy/     !23 feet!     :Hello!:     "1/2"
```

Most immediate commands have a corresponding extended version. See the table of commands for full details

Program control

The command X causes the editor to exit. The text held in storage is written out to file with the current extension, and the editor then terminates renaming the old file (if it exists) with .BAK. Alternatively the Q command terminates immediately without writing the buffer; confirmation is requested in this case if any changes have been made to the file. A further command allows a 'snapshot' copy of the file to be made without coming out of ED. This is the SA command. SA saves the text to a named file or, in the absence of a named file, to the current file. For example:

```
*SA /saved.tex/
```

or

```
*SA
```

This command is particularly useful in areas subject to power failure or surge. It should be noted that SA followed by Q is equivalent to the X command. Any alterations made between the SA and the Q will cause the message

```
Edits will be lost - type Y to confirm:
```

to be displayed; if no alterations have been made the program will be quitted immediately with the file saved in that state. SA is also useful because it allows the user to specify a file name other than the current one. It is therefore possible to make copies at different stages and place them in different files or with different extensions.

The U command causes the last change to be 'undone'. The editor takes a copy of the line the cursor is currently on, and modifies this when characters are added or deleted. The changed copy is replaced back into the file when the cursor is moved off the current line (either by cursor control, or by deleting or inserting a line). The copy is also replaced when

any scrolling, either vertically or horizontally, is performed. The U command causes the changed copy to be discarded and the old version of the current line to be used instead. Note that the U command is equivalent to pressing the UNDO key.

The SH command shows the current state of the editor. Information such as the value of tab stops, current margins, block marks and the name of the file being edited is displayed. Note that the SH command is equivalent to pressing the HELP key. Tabs are initially set at every three columns; this can be changed by the command ST, followed by a number n, which sets tabs at every n columns. The left margin and right margin can be set by SL and SR commands, again followed by a number indicating the column position. The left margin should not be set beyond the width of the screen. The EX command may be used to extend the margins; once this command is given no account will be taken of the margins; once the cursor is moved off the current right margin on the current line. Once the cursor is moved off the current line margins are enabled once more.

Block control

A block of text can be identified by means of the BS (block start) and BE (block end) commands. The cursor should be moved to the first line required in a block, and the BS command given. The cursor can then be moved to the last line wanted in the block, by cursor control commands or in any other way, such as searching. The BE command is then used to mark the end of the block. If any change is made to the text the block start and block end become undefined once more.

Once a block has been identified, a copy of it may be moved into another part of the file by means of the IB (insert block) command. The previously identified block is replicated immediately after the current line. Alternatively a block may be deleted by means of the DB command, after which the block start and end values are undefined.

Block marks may also be used to remember a place in a file. The SB (show block) command resets the screen window on the file so that the first line in the block is at the top of the screen.

A block may also be written to a file by means of the WB command. The command is followed by a string which represents a file name. The file is created, possibly destroying the previous contents, and the buffer written to it. A file may be inserted by the IF command. The file name given as the argument string is read into storage immediately following the current line.

Movement

The command T moves the cursor to the top of the file, so that the first line in the file is the first line on the screen. The B command moves the cursor to the bottom of the file, so that the last line in the file is the bottom line on the screen if possible.

The commands N and P move the cursor to the start of the next line and previous line respectively. The commands CL and CR move the cursor one place to the left or one place to the right,

while CE places the cursor at the end of the current line, and CS places it at the start.

Movement with P and N is relative to the current line (the line the cursor is pointing at), that is to say, you can move so many lines before or after the current line. However, it is sometimes useful to be able to move to a specific line in a file. If there is an error at line 222,

```
T;221 N
```

will move to the errant line (T is superfluous if you are at the top already). The top is reached almost instantly; the following lines are then scrolled until the final line is shown. Large files make this a tedious method. The command M takes a line number, moving to it instantly which saves time and frustration.

```
M 222
```

Searching and Exchanging

Alternatively the screen window may be moved to a particular context. The command F is followed by a string which represents the text to be located. The search starts at one place beyond the current cursor position and continues forwards through the file. If found, the cursor is placed at the start of the located string. To search backwards through the text use the command BF (backwards find) in the same way as F. BF will find the last occurrence of the string before the current cursor position. To find the earliest occurence use T followed by F; to find the last use B followed by BF.

The E (exchange) command takes a string followed by further text and a further delimiter character, and causes the first string to be exchanged to the last. So for example

```
E /wombat/zebra/
```

would cause the letters 'wombat' to be changed to 'zebra'. The editor will start searching for the first string at the current cursor position, and continues through the file. After the exchange is done the cursor is moved to after the exchanged text. An empty string is allowed as the search string, specified by two delimiters with nothing between them. In this case the second string is inserted at the current cursor position. No account is taken of margin settings while exchanging text.

A variant on the E command is the EQ command. This queries the user whether the exchange should take place before it happens. If the response is N then the cursor is moved past the search string, otherwise the change takes place as normal. This command is normally only useful in repeated groups.

All of these commands normally perform the search making a distinction between upper and lower case. The command UC may be given which causes all subsequent searches to be made with cases equated. Once this command has been given then the search string "wombat" will match "Wombat", "WOMBAT", "WoMbAt" and so on. The distinction can be enabled again by the command LC.

Altering text

The E command cannot be used to insert a newline into the text, but the I and A commands may be used instead. The I command is followed by a string which is inserted as a complete line before the current line. The A command is also followed by a string, which is inserted after the current line. I and A are uneffected by margin settings.

The S command splits the current line at the cursor position, and acts just as though a RETURN had been typed in immediate mode. The J command joins the next line onto the end of the current one. Where the left hand margin is greater than 1 the J command will include the number of spaces from column 1 to wherever the margin has been set.

The D command deletes the current line in the same way as the CTRL/B key in immediate mode, while the DC command deletes the character at the cursor in the same way as CTRL/N.

Repeating commands

Any command may be repeated by preceding it with a number. For example,

```
    4 E /slithy/brillig/
```

will change the next four occurrences of 'slithy' to 'brillig'. The screen is verified after each command. The RP (repeat) command can be used to repeat a command until an error is reported, such as reaching the end of the file. For example,

```
    RP E /slithy/brillig/
```

will change all occurrences of 'slithy' to 'brillig'.

Commands may be grouped together with brackets and these command groups executed repeatedly. Command groups may contain further nested command groups. For example,

```
    RP ( F /bandersnatch/; 3 IB; N )
```

will insert three copies of the current block whenever the string 'bandersnatch' is located.

Note that some commands are possible, but silly. For example,

```
    RP SR 60
```

will set the right margin to 60 *ad infinitum*. However, any sequence of extended commands, and particularly repeated ones, can be interrupted by typing any character while they are taking place. Command sequences are also abandoned if an error occurs.

WARNING: This screen editor supports HIGH and MEDIUM resolution only. Do not attempt to use this editor on a LOW resolution monitor; if you do, certain undefined effects may occur.

Section B: Running the Compiler

Immediate Commands

CTRL/A or Insert	Insert line
CTRL/B or F3	Delete line
CTRL/D or F7	Scroll text down
CTRL/E or F10	Move to top or bottom of screen
CTRL/F or F5	Change character case
CTRL/G or F1	Repeat last extended command
CTRL/H or 'Left arrow'	Cursor left
CTRL/I or Tab	Tab
CTRL/J or 'Down arrow'	Cursor down
CTRL/K or 'Up arrow'	Cursor up
CTRL/M or Return	Return
CTRL/N or Delete	Delete character at cursor
CTRL/O or F2	Delete word or spaces
CTRL/R or F8	Cursor to end of previous word
CTRL/T or F9	Cursor to start of next word
CTRL/U or F6	Scroll text up
CTRL/V	Verify screen
CTRL/X or 'Right arrow'	Cursor right
CTRL/Y or F4	Delete to end of line
CTRL/[or Esc	Escape (enter extended mode)
CTRL/] or ClrHome	Home (cursor to start/ end of line)

Extended Commands

This is a full list of extended commands, including those which are merely extended versions of immediate commands. In the list, /s/ indicates a string, /s/t/ indicates two exchange strings and n indicates a number.

A /s/	Insert line after current
B	Move to bottom of file
BE	Block end at cursor
BF	Backwards find
BS	Block start at cursor
CE	Move cursor to end of line
CL	Move cursor one position left
CR	Move cursor one position right

CS	Move cursor to start of line
D	Delete current line
DB	Delete block
DC	Delete character at cursor
E /s/t/	Exchange s into t
EQ /s/t/	Exchange but query first
EX	Extend right margin
F /s/	Find string s
I /s/	Insert line before current
IB	Insert copy of block
IF /s/	Insert file s
J	Join current line with next
LC	Distinguish between upper and lower case in searches
M n	Move to line n
N	Move cursor to start of next line
P	Move cursor to start of previous line
Q	Quit without saving text
RP	Repeat until error
S	Split line at cursor
SA	Save text to file
SB	Show block on screen
SH	Show information (= Help key)
SL n	Set left margin
SR n	Set right margin
ST n	Set tab distance
T	Move to top of file
U	Undo changes on current line (= Undo key)
UC	Equate U/C and l/c in searches
WB /s/	Write block to file s
X	Exit, writing text to file

Running the Compiler

Running the Compiler

After inserting and opening the MCC Pascal 68000 compiler disk, the compiler can be invoked by double-clicking on PASCAL.TTP and typing the required parameters (see below). If you type illegal parameters then the compiler will re-prompt for the command line, giving you a chance to exit to GEM.

The format of the command line is:

```
<source-file>[TO <object-file>][<options>]
```

<source-file> is a self-contained MCC Pascal 68000 program. <object-file> is the object code program produced by the successful compilation of <source-code>. Note that <object-file> is optional; this allows <source-file> to be passed through the compiler without producing a corresponding object code program.

Options

Up to eight options are available:

1. LIST <list-file>	If specified, an output source compilation listing is generated by the compiler. The list file gives useful information about the compilation. It supplies the name of the file compiled; a listing of the source code with each line, statement and level of logical nesting numbered, any compilation errors found, positioned at the relevant place in the source listing; details of the block structure of the program, procedures, functions and associated storage; details of the identifiers declared and their associated storage.
2. NOCHECK	If specified, run-time range-checking is disabled for the resulting object-code program. With range-checking disabled, programs execute much more efficiently but are prone to unpredictable results if data type mismatches are encountered. A finished program should always be compiled with this option.
3. EXTEND	If specified, makes available to the programmer the MCC Pascal 68000 extensions to the ISO standard. (See Chapter 8 for a list of these extensions.) You will need to use this option if you wish to access the graphics library.

4. CEM	if specified, gives Continuous Error Messages to the console. This is useful if the compiler is being run from a batch file and console interaction is not required. Note that if the VER option is used (see below) then CEM is automatically set on.
5. WS <size>	if specified, the program is compiled within the given workspace size, <size>. The size given should be in Kbytes and the default is 30K.
6. CASE	if specified, the compiler will be case sensitive towards variable names. This option can only be used in conjunction with the EXTEND option (see above).
7. VER <ver-file>	if specified, a file is created and all output that the compiler would normally sent to the console is sent to the verification file instead. Note that CEM (see above) is automatically set on if verification file is given.
8. OBJ <code-letter>	this option determines the type of module format produced. The choice of object module will depend upon which linker is used (see below). If specified, the <code-letter> must be either G to produce GST format compatible with the LINK program supplied with this package or J to produce CP/M-68K format compatible with the linker called LINK68 supplied with the Atari Software Developer's Kit. The default is G.

9. ERR <drive>	if specified, the compile time error message file PASCAL.ERR is assumed to be on the given drive <drive>. The default drive is 'A:'.

Compilation

Any errors apart from warnings detected by the MCC Pascal 68000 compiler cause suppression of further code generation. An error message together with the erroneous portion of source text are displayed on the console (or in the verification file if the VER option is specified). Also, if the LIST option is specified at compile time, the error message is output at the appropriate point in the compilation listing. Unless the Continuous Error Message (CEM) option or verification file (VER) option is specified, compilation continues after hitting the carriage return key, with errors being reported as necessary.

After each error the user is given the chance to abort the compilation (unless the CEM or VER option is specified). Pressing control B at any point during the compilation also causes the compilation to abort.

Examples

To compile an MCC Pascal 68000 source file FRED.PAS, which is written in standard (ISO) Pascal, creating the list file FRED.LST but without creating an object file, the command line is;

```
FRED.PAS LIST FRED.LST
```

To compile an MCC Pascal 68000 source file PETE.PAS, which contains extensions to ISO Pascal, to a GST format object file PETE.BIN which is compatible with the program LINK, without stopping after any errors to wait for console input, the command line is;

```
PETE.PAS TO PETE.BIN CEM EXTEND
```

To compile an MCC Pascal 68000 source file BERT.PAS written in standard (ISO) Pascal to a CP/M-68K format object file BERT.O, using a workspace of 40K, the command line is:

```
BERT.PAS TO BERT.O OBJ J WS 40
```

Linking a program

The object-file produced by the MCC Pascal 68000 compiler must be linked with the Pascal startup sequence STARTUP and the Pascal run-time library PASLIB to produce an executable program. .

As mentioned above, the type of object module format produced by the MCC Pascal 68000 Compiler depends upon which linker is to be used. The linker supplied with this package accepts GST object module files as input. If you have the Atari Development Kit then you may wish to use the LINK68 linker which accepts CP/M-68K object module files as input. (In this case see the documentation supplied with the Development Kit for details of how to run LINK68.)

The following insutructions assume that you have produced a GST format object module and that you are using the supplied linker LINK.TTP. (For full details of this, see Section C.)

The linker is invoked by double-clicking on LINK.TTP and typing the required command line. The simplified format for the command line for LINK is just:

```
<object-file> <command-file>
```

where <object-file> is the name of the GST object module file produced by the Pascal compiler and <command-file> is a suitable command file for the linker. Although full details of the layout of command files are given in Section C, the file PLINK.LNK supplied on the Pascal Linker

disk will be suitable for Pascal programs which do not require user-written external files to be linked in. This file simply tells the linker to link STARTUP.BIN, the <object-file> from the command line, PASLIB.BIN, and FINISH.BIN. It also specifies the stack space to be allocated to your executable file (this is set to 10K and should be changed as appropriate).

Note: The Pascal libraries STARTUP and PASLIB have been supplied in both CP/M-68K and GST object-module format. Ensure that you match the correct versions with the linker:

for the CP/M-68K linker, LINK68, use STARTUP.O, PASLIB.O and FINISH.BIN;

for the GST linker, LINK, use STARTUP.BIN, PASLIB.BIN and FINISH.BIN.

Examples

Having compiled an MCC Pascal 68000 source file PETE.PAS to produce a GST format object file PETE.BIN this must be linked with the Pascal startup sequence STARTUP.BIN and the Pascal library PASLIB.BIN and FINISH.BIN to produce an executable file PETE.PRG using the following LINK command line:

```
PETE PLINK
```

Running the Pascal program

To run an executable .PRG file, double-click on it from the desktop.

As discussed in Chapter 6, it is possible to match external files at run-time with main program parameters. To do this the .PRG file should be installed as an application by opening the disk, selecting .PRG and using the Install Application option from the desktop. .PRG should be given the "TOS-takes-parameters" application type. (For more information on this see the Atari ST User Guide.) Alternatively, renaming the .PRG file as .TTP has the same effect.

To run the executable .PRG file double-click on it from the desktop and match the main program parameters as required.

Example (see last example program of Chapter 6)

If the first line of the Pascal source file is

```
PROGRAM Transfer (output,FileIn,FileOut)
```

then having run the MCC Pascal 68000 compiler and the linker to produce the executable file TRANSFER.PRG, install and run this file as above. To match the parameters type,

```
FileIn = B:FRED.REL FileOut = A:BILL.REL
```

where we wish to use B:FRED.REL as the data file to copy and A:BILL.REL as the name of the copy.

The Linker

[illegible] the Linker

Loading the Program

To run the linker from the desktop, insert the disk into either drive and double-click on the appropriate icon to obtain a directory listing, next double-click on the directory entry LINK.TTP to run the linker program.

The linker program is loaded and will prompt for a command line. See below for details of the command line format.

Command Line Format

The format of the command line is:

```
    module [control [listing [program]]] [option]
```

The various command line specifiers are shown in the order in which they must appear. Optional specifiers are shown enclosed in brackets.

module	Specifies the name of an object file to be used as input to the linker.
control	Specifies the name of a control file from which a list of instructions are input and acted upon.
listing	Specifies the name of the listing file which the linker will generate. This shows the commands used in the production of the link and a map of the layout of the executable file. The map will also show a list of all global symbols and their values an option cross reference listing the modules which reference them.

The Linker

How to Run the Linker

Loading the Program

To run the linker from the desktop, insert the disk into either drive and double-click on the appropriate icon to obtain a directory listing. next double-click on the directory entry LINK.TTP to run the linker program.

The linker program will load and will prompt for a command line. See below for details of the command line format.

Command Line Format

The format of the command line is:

```
    [module [control [listing [program]]]] [option]
```

The various command line specifiers are shown in the order in which they must appear. Optional specifiers are shown enclosed in brackets.

module	Specifies the name of an object file to be used as input to the linker.
control	Specifies the name of a control file from which a list of instructions are input and acted upon.
listing	Specifies the name of the listing file which the linker will generate. This shows the commands used in the production of the link and a map of the layout of the executable file. The map will also show a lt of all global symbols and their values an option cross reference giving the modules which reference them.

program	Specifies the name of the executable program file to be generated by the linker.

Options

Linker options are specified as a hyphen followed by a word; in some cases, additional text may be appended. The option word may be supplied in either upper or lower case. Each option must be specified separately with a separate hyphen. Options available are:

-WITH[filename]	take the following name as the control file name. A file name must be given with this option.
-NOPROG	do not generate a program file.
-PROG[filename]	generate a program file (default).
-NOLIST	do not generate any listing output.
-LIST	generate a listing (default).
-NODEBUG	do not append a symbol table to the binary file output (default).
-DEBUG[filename]	append a symbol table to the binary file output.
-NOSYM	do not generate a symbol table listing in the listing file.
-SYM	generate a symbol table listing. The listing will be alphabetically sorted with the value of the symbol with the section and module name in which it was defined (default).

-CRF	generate a cross reference form of symbol table listing. If this option is requested a cross reference form of the symbol table is generated instead of the symbol table list.
-PAGELEN n	specify the number of lines per page for paginated output. If this option is not supplied the value will default.

If an option is followed by a file name (where applicable) the file name will override the corresponding positional file name (if given) on the command line. If an option specifies that a file will not be generated (-NOPROG, -NOLIST) then the file will not be generated even if a positional file name has been given.

Where conflicting options are given on the command line then the last option coded will take effect; for example:

```
-NOPROG  -PROG  FRED.PRG
```

will produce a program file, whereas

```
-PROG  FRED.PRG  -NOPROG
```

will not.

Command Line Processing

For Pascal, the minimum command line just consists of the name of the object file produced by the compiler followed by the name of the control file. The control file must tell the linker to link in the Pascal startup sequence, STARTUP.BIN, the object file specified on the command line, the Pascal library, PASLIB.BIN and any other files or libraries (such as graphics) needed by the Pascal program.

A standard control file, PLINK.LNK, is supplied with the linker and should be used to link your programs.

Construction of Output File Names

If a module file name is given then the file name is examined. If the file name does not contain a dot then the full file name becomes the base file name, otherwise the file name with the file type (from the dot onwards) stripped off becomes the base file name.

If no module file name is given then the control file name is examined. If the file name does not contain a dot then the full file name becomes the base file name, otherwise the file name with the file type stripped off becomes the base file name.

The default names are then constructed from the base file name as follows:

1) The listing file name is the base file name, with ".MAP" appended.

2) The program file name is the base file name with ".BIN" appended.

3) The debug file name is the base file name with ".SYM" appended.

If an output file name is given explicitly either as a positional parameter or in an <option> then the file name will override the corresponding default name. Any file name given explicitly must be given in full as the file name will be used exactly as entered.

Input File Name Defaults

The linker has two types of input file: the control file, which tells the linker what to do (if more information is needed than can be coded in the command line) and relocatable binary files, which are the output files from the compiler that contain the program to be linked.

For a module file name (or library file name), if the module file name contains a file type then the linker will use the file name exactly as given. If the file name does not contain a file type then ".BIN" will be

appended to the file name; If an open error occurs on this file then the original file name is used instead (by stripping off the ".BIN" again).

This defaulting will apply to all module input commands in the control file as well as to any relocatable binary file name given on the command line.

If the control file name contains a file type then the linker will use the control file name exactly as given. If the file name does not contain a file type then ".LNK" is appended to the file name; if an open error occurs on this file then the original file name is used as the control file name.

Command Line Examples

```
MYPROG PLINK -NOLIST
```

Link the file MYPROG.BIN according to the instructions in PLINK.LNK. The program is called MYPROG.PRG.

```
-WITH FRED
```

Take FRED.LNK as the control file, place the program in FRED.PRG and place the full listing output in FRED.MAP.

```
-WITH BERT -LIST LST: -NOPROG
```

Take BERT.LNK as the control file, do not generate a program file but print the listing as it is produced.

```
-WITH PETE -PROG PETE.TTP
```

Take PETE.LNK as the control file, place the program in PETE.TTP and place the listing output in PETE.MAP.

Termination:

When the link has finished, and if there have been no operating system errors, the linker will issue a message giving the status of the link.

Linker Inputs and Outputs

The linker uses the following inputs and outputs.

Command Line Input

When run interactively, the linker will read a command line from the keyboard to tell it what to do. Any errors in the command line will result in an error message followed by a reprompt of the command line. See Command Line Format above for full details of the command line.

Control File

If the command line includes a control file name the linker will expect as input a single text file containing a list of instructions to perform.

The control file is described in detail in The Control File below.

Relocatable Binary Files

The linker, on instruction from the command line and/or control file, will read the Pascal startup sequence, the relocatable binary file produced by the compiler and then scan the library.

The files are opened for random access to allow modules to be extracted independently (for EXTRACT and LIBRARY commands).

Screen Output

The linker writes information to the screen to inform the user what is happening. This includes a sign-on message identifying the program, and a prompt for a command line.

The linker writes all errors and warning messages to the screen and on completion of the link will print a summary of the number of errors and warnings and the number of undefined symbols (if any).

The linker also tells you when it is starting to read the relocatable binary files for the first time and when it is starting to read the relocatable binary files for the second time. The second pass can be expected to take a lot longer than the first pass if listings and/or program output are wanted.

The linker finally gives a message indicating the completion status of the link and if run interactively prompts again for another command line.

Linker Listing Output

An optional linker listing will be generated, showing the commands used in the production of the link and a map of the layout of the executable file. The map will also show a list of all global symbols and their values and an optional cross reference giving the modules which reference them.

Program File Output

The linker will optionally generate a program file which will be the result of combining the relocatable binary files. This file can be run by TOS as a program.

The linker will optionally append a symbol table to the program file output for use by a symbolic debugger program.

The Control File

The control file is a text file which gives a series of instructions to the linker. The complete set of instructions to the linker will be given here for completeness; however some of them are pretty obscure and are not necessary for linking normal programs.

Supplied with the compiler is a standard control file, PLINK.LNK. This should be used to link your Pascal programs with the Pascal startup sequence, the Pascal library and with the graphics libraries if these are being used.

Unlike the command line input the control file input is not interactive and any errors in the control file will cause the link to be abandoned.

All letters in control file commands and command parameters may be in either case as case is not significant.

Comments in the Linker Control File

The linker accepts comments in the linker control file to explain to the reader what a particular control file does. A line will be considered a comment if the first character in the line is a star (*), semicolon (;) or an exclamation mark (!). A blank line is also considered to be a comment.

The use of comments in a control file may assist you in editing the control file to suit your particular program.

Standard Control File PLINK.LNK

```
*
*    Standard control file for linking MCC Pascal
*    68000 programs
*
*    Step 1 - initialisation
*    ***********************
*
*    Pascal initialisation must be included first
*
INPUT STARTUP.BIN
*
*    Step 2 - user module
*    ********************
*    Now include the program output by the Pascal
*    compiler (from the command line)
*
INPUT
*
*    Step 3 - Pascal library
*    ***********************
*    Pascal library - must always be included.
*
LIBRARY PASLIB.BIN
*
*    For each extra module you want to include in
*    the link, include a line of the form:
*    INPUT <file name>
*
*    Step 4 - termination
*    ********************
*    Include the Pascal termination
*    module must be last.
*
INPUT FINISH.BIN
*
*    Step 5 - GEM graphics library
*    *****************************
*    GEM graphics library is only included if your
*    program is trying to access graphics routines
```

contiene el modulo finish.bin (ver listado)

```
*
*    (by uncommenting the line).  Must appear after
*    FINISH.BIN
*
*    LIBRARY GEMLIB.BIN
*
*    Step 6 - define stack size
*    ==========================
*    Request 10K of data space for run-time
*    stack (change this value as required)
*
DATA 10K
```

Module Input Commands

There are three commands to instruct the linker to read in modules from relocatable binary files. All these commands use the same defaults for file names as the module input file in the command line.

These commands are:

```
INPUT <file name>
LIBRARY <file name>
EXTRACT <module name> FROM <file name>
```

and most users will rarely need to use any other commands.

(a) `INPUT <file name>`

This command instructs the linker to read the file named and place all modules encountered in the file into the link. Include one command for each file that you wish to include in the link.

Example:

```
INPUT STARTUP.BIN
```

will include the file STARTUP.BIN.

A special case of the input command is the command

```
INPUT *
```

which instructs the linker to input the relocatable binary file whose name was given on the command line. This feature allows the generation of a template file which can be used to link a single program output from the compiler with all the required libraries, and this is the standard one PLINK.LNK already shown. The template file is then used by a command line of the following form (the -WITH is optional):

```
<module file name> [-WITH] PLINK
```

(b) `LIBRARY <file name>`

Consulta

This command instructs the linker to search the relocatable binary file named from start to finish for modules which satisfy any currently unresolved references in the link. When a module is found which satisfies an unresolved reference it is included in the link and the library search continues form the current position.

All libraries supplied with Pascal are ordered is such a way that they need only be searched once (i.e., only one LIBRARY command). If you create your own libraries, you may need to scan them more than once; this may be achieved by including more than one LIBRARY command specifying the same filename. You must, of course, include at least one LIBRARY command for each library that you wish to search.

(c) `EXTRACT <module name> FROM <file name>`

This command instructs the linker to search the relocatable binary file specified for the module requested. If the module is found it is included in the link. If it is not found an error message is generated and the link is aborted.

Include one extract command for each module that you wish to explicitly include from the relocatable binary file.

This command is not required for linking Pascal.

Todas las librerías suplidas con [illegible] de tal manera que sólo son buscadas una vez. Si creas tu propia librería puedes necesitar [illegible] más de una vez

(d) DATA <value>[K]

The DATA command specifies the amount of data space to reserve for a program for the stack and heap. The value may be decimal or hexadecimal. This value is used by the operating system to allocate room for the stack and heap. The value may be specified in bytes or Kbytes (1024 bytes). An allocation of 10K bytes should be sufficient for most Pascal programs

The Listing File

The listing file consists of a series of reports to indicate what the linker has done with the program file. The following reports are generated:

(a) Command line and control file information

This report indicates the command line used to perform the link and a listing of the control file (if one was used). Any error messages from processing of the control file are also placed in the report.

(b) Object module header information

This report indicates which commands were used for input of modules and the module names read in by the command. Any error messages produced while reading the module files are also printed here.

(c) Load Map

This report generated after pass 1 indicates where the linker has placed everything. The load map is produced in increasing address order with the following format:

(1) For each section a line in the following form

(a) The section type (ABSOLUTE, SECTION, COMMON)
(b) The section start address
(c) The section end address
(d) The section name

(2) For each subsection (contribution from a module) a line of the following form:

(a) The start address of the subsection
(b) The end address of the subsection
(c) The module name

(3) For each entry point in a relocatable or common subsection a line of the following form (in increasing address order)

(a) The entry point address
(b) The entry point name

The load map is then followed by three lists of the following form:

(1) Absolute symbols in address order
(2) User defined symbols in defined order
(3) Undefined symbols in alphabetical order

(d) Symbol table listing

The linker produces a symbol table listing of all global symbols in the link in alphabetical order. For each symbol a line is printed containing the following information:

(1) The value of the symbol (or ???????? for undefined symbols)
(2) The symbol name
(3) The section name the symbol is defined in (or Absolute, defined or undefined)
(4) The module name (if defined within the module).

If the -CRF option is used on the command line then if a symbol is referenced in other modules the symbol infomation is followed by one or more lines of module names which reference the symbol. This cross reference information is followed by a blank line before the next symbol table entry.

Actions of the Linker

This chapter gives a brief description of how the linker functions and the expected actions when errors are encountered. The linker functions are split into several phases which are logically separate although each phase may use information extracted from previous phases.

Command Line Validation

In this phase the linker reads the command line and decides which input and output files to use. If the command line contains any errors the linker will display an error message stating the problem.

If the command line is valid the linker will attempt to open all output files requested and the linker control file (if a name is supplied). If the opening of any file fails the linker will give a message indicating the problem.

If the linker is run interactively it will reprompt for another command line. If not then the linker will display a message indicating an invalid command line supplied and exit.

Control File Validation

If a control file name is given the linker will read the control file line by line validating each command in turn. If any errors are reported at this stage the linker will report the error but continue reading the control file.

If any errors occur in the control file the linker will not perform the link but the message 'Errors in linker command file' will appear. If run in interactive mode then the linker will reprompt for another command line. If run in non-interactive mode the linker will exit.

Pass 1 of Relocatable Binary Files

If the command line and control file (if given) contain valid commands the linker will issue a message saying 'starting pass 1' and will read all the relocatable binary files requested and determine the size of each section to be placed in the output file. During this pass the linker will issue error and warning messages as appropriate to indicate any problems encountered.

If the linker fails to open any requested input files or encounters any errors during this pass the linker will issue an error message stating the problem and will continue processing the rest of the input files.

At the end of pass one if any errors have been encountered the linker will print an error message summary and print the message 'Link completed with errors'. In interactive mode the linker will reprompt for another command line. In non-interactive mode the linker will exit.

If only warnings have been detected the linker will continue with the link.

Between Pass Processing

After pass 1 the linker determines where to place everything in the program file and resolves all global symbols. The load map is generated at this time along with a list of all absolute, user defined and undefined symbols.

Pass 2 Processing

During this pass all the relocatable binary files are reread and the program file created complete with a program header. If any errors are encountered at this stage the link is aborted.

[illegible] Library Interface

This section [illegible] reference guide to the Metacomco Graphics Library [illegible] the GEM and AES graphics libraries (GEMLIB.BIN and [illegible]). Each routine is listed with a brief description of its action. [illegible] description of the routines can be found along with the [illegible] provided on disk (LIB0.ASM to LIB18.ASM). To help you [illegible] description, each routine is listed under the filename of the [illegible] in which it can be found. The header file GEMLIB.INC [illegible] useful macros and structures for use with this library.

LIB1

```
v_opnvwk( work_in, &handle, work_out )
Open virtual workstation

v_clsvwk( handle )
Close virtual workstation

v_clrwk( handle )
Clear screen to current background colour

numfonts = vst_load_fonts( handle, select )
Loads all fonts (other than system fonts, which are already loaded)

vst_unload_fonts( handle, select )
Unloads all fonts

vs_clip( handle, flag, work_in )
Enable clipping to specified rectangle, or disable it
```

Section E: Graphics Library

LIB2

```
v_pline( handle, count, work_in )
Draw lines between coordinate pairs given

v_pmarker( handle, count, work_in )
Draw markers at coordinate pairs given

v_gtext( handle, x, y, string )
Write graphics text

v_fillarea( handle, count, work_in )
Draw a filled polygon

v_cellarray( handle, coords, rowlen, rowsize, nrows,
             mode, work_in )
Draw a rectangular pixel image

v_contourfill( handle, x, y, colour )
Flood fill

vr_recfl( handle, coords )
Draw filled rectangle with no outline

v_bar( handle, coords )
Draw a filled rectangle with an outline

v_arc( handle, x, y, radius, starta, stopa )
Draw an arc

v_pieslice( handle, x, y, radius, starta, stopa )
Draw a pie shape

v_circle( handle, x, y, radius )
Draw a circle
```

Graphics Library Interface

This section provides a quick reference guide to the Metacomco Graphics Library Interface to the GEM and AES graphics libraries (GEMLIB.BIN and GEMLIB.O). Each routine is listed with a brief description of its action. A full description of the routines can be found along with the source which is provided on disk (LIB0.ASM to LIB18.ASM). To help you find the full description, each routine is listed under the filename of the source file in which it can be found. The header file GEMLIB.INC defines useful macros and structures for use with this library.

Lib1

`v_opnvwk( work_in, &handle, work_out )`
Open virtual workstation → [illegible] workstation

`v_clsvwk( handle )`
Close virtual workstation → [illegible]

`v_clrwk( handle )`
Clear screen to current background colour → [illegible]

`numfonts = vst_load_fonts( handle, select )`
Loads all fonts (other than system fonts, which are already loaded)

`vst_unload_fonts( handle, select )`
Unloads all fonts

`vs_clip( handle, flag, work_in )`
Enable clipping to specified rectangle, or disable it.
↓ Permite [illegible]

Lib2

`v_pline( handle, count, work_in )`
Draw lines between coordinate pairs given

`v_pmarker( handle, count, work_in )`
Draw markers at coordinate pairs given

`v_gtext( handle, x, y, string )`
Write graphic text

`v_fillarea( handle, count, work_in )`
Draw a filled polygon

`v_cellarray( handle, coords, rowlen, rowsize, nrows,`
`mode, work_in )`
Draw a rectangular pixel image

`v_contourfill( handle, x, y, colour )`
Flood fill

`vr_recfl( handle, coords )`
Draw filled rectangle with no outline

`v_bar( handle, coords )`
Draw a filled rectangle with an outline

`v_arc( handle, x, y, radius, starta, stopa )`
Draw an arc

`v_pieslice( handle, x, y, radius, starta, stopa )`
Draw a pie shape

`v_circle( handle, x, y, radius )`
Draw a circle

```
v_ellarc( handle, x, y, xradius, yradius, starta,
          stopa )
```
Draw an elliptical arc

```
v_ellpie( handle, x, y, xradius, yradius, starta,
          stopa )
```
Draw an elliptical pie

```
v_ellipse( handle, x, y, xradius, yradius )
```
Draw an ellipse

```
v_rbox( handle, coords )
```
Draw a rectangle with rounded corners

```
v_rfbox( handle, coords )
```
Draw a filled rectangle with rounded corners

```
v_justified( handle, x, y, string, length, wordsp,
             charsp )
```
Display text justified to fit given length

Lib3

```
vswr_mode( handle, mode )
```
Select the writing mode

```
vs_color( handle, colour, rgb )
```
Select the red/green/blue proportions for a particular colour value

```
actual_style = vsl_type( handle, style )
```
Select the line drawing style

```
vsl_udsty( handle, mask )
```
Specify the user defined line drawing mask

```
actual_width = vsl_width( handle, width )
```
Select the line width

```
actual_colour = vsl_color( handle, colour )
```
Select the line colour

```
vsl_ends( handle, starts, ends )
```
Select the line end style

```
actual_type = vsm_type( handle, type )
```
Select the marker type for the polymarker call

```
actual_height = vsm_height( handle, height )
```
Select the marker height for the polymarker call

```
actual_colour = vsm_color( handle, colour )
```
Select the marker colour for the polymarker call

```
vst_height( handle, height, &char_w, &char_h, &cell_w,
            &cell_h )
```
Select the height of characters using coordinate units

```
vst_point( handle, height, &char_w, &char_h, &cell_w,
           &cell_h )
```
Select the height of characters using points units

```
actual_angle = vst_rotation( handle, angle )
```
Select the rotation for characters

```
actual_font = vst_font( handle, font )
```
Select the character font

```
actual_colour = vst_color( handle, colour )
```
Select the character colour for graphic text

```
actual_effect = vst_effects( handle, effect )
```
Select the special effects for graphic text

```
vst_alignment( handle, hin, vin, &hactual, &vactual )
```
Specify the alignment of text with respect to a coordinate point

```
actual_style = vsf_interior( handle, style )
```
Specify the style for filling the interior of polygons etc

```
actual_style = vsf_style( handle, style )
```
Specify the pattern for filling when pattern or hatched selected

```
actual_colour = vsf_color( handle, colour )
```
Specify the colour to be used when filling

```
actual_flag = vsf_perimeter( handle, flag )
```
Turn fill area outline on or off

```
vsf_udpat( handle, pattern, planes )
```
Specify the user definable fill pattern

Lib4

```
vro_cpyfm( handle, mode, coords, source, dest )
```
Copy a rectangular area, opaque

```
vrt_cpyfm( handle, mode, coords, source, dest, colour )
```
Copy a rectangular area, transparent

```
vr_trnfm( handle, source, dest )
```
Transform memory form definition block

```
v_get_pixel( handle, x, y, &value, &colour )
```
Get pixel value and colour

Lib5

```
vsc_form( handle, form )
```
Define mouse cursor

```
vex_timv( handle, new_addr, old_addr, &mspertick )
```
Define timer interrupt routine

```
v_show_c( handle, reset )
```
Show mouse cursor

```
v_hide_c( handle )
```
Hide mouse cursor

```
vq_mouse( handle, &button, &x, &y )
```
Inquire mouse position and state

```
vex_butv( handle, new_addr, old_addr )
```
Define mouse button interrupt routine

```
vex_motv( handle, new_addr, old_addr )
```
Define mouse movement interrupt routine

```
vex_curv( handle, new_addr, old_addr )
```
Define cursor change interrupt routine

```
vq_key_s( handle, &status )
```
Inquire keyboard status

Lib6

```
vq_extnd( handle, flag, work_out )
```
Extended enquire

```
vq_color( handle, colour, flag, rgb )
```
Enquire red/green/blue proportions of specified colour

```
vql_attributes( handle, work_out )
```
Enquire attributes concerning polylines

```
vqm_attributes( handle, work_out )
```
Enquire attributes concerning polymarkers

```
vqf_attributes( handle, work_out )
```
Enquire attributes concerning filling

```
vqt_attributes( handle, work_out )
```
Enquire attributes concerning text

```
vqt_extnt( handle, string, work_out )
```
Enquire size of rectangle enclosing given text.

```
rc = vqt_width( handle, char, &width, &left, &right )
```
Enquire size of character cell

```
index = vqt_name( handle, id, name )
```
Enquire font name and index number

```
index = vq_cellarray( handle, coords, rowlen, nrows,
                      &rowsize, &rowused, &err,
                      work_out )
```
Enquire cell array definition

```
vqin_mode( handle, dev_type, &input_mode )
```
Enquire input mode

```
vqt_fontinfo( handle, &min, &max, dist, &maxwidth,
              work_out )
```
Enquire current font information

Lib7

```
vq_chcells( handle, &rows, &cols )
```
Inquire number of character cells

```
v_exit_cur( handle )
```
Exit alpha mode

```
v_enter_cur( handle )
```
Enter alpha mode

```
v_curup( handle )
```
Cursor up

```
v_curdown( handle )
```
Cursor down

```
v_curright( handle )
```
Cursor right

```
v_curleft( handle )
```
Cursor left

```
v_curhome( handle )
```
Cursor home

```
v_eeos( handle )
```
Erase from cursor to end of screen

```
v_eeol( handle )
```
Erase from cursor to end of line

```
vs_curaddress( handle, row, col )
```
Cursor absolute address

```
v_curtext( handle, string )
```
Print string at current alpha cursor position

```
v_rvon( handle )
```
Turn on reverse video mode

```
v_rvoff( handle )
```
Turn off reverse video mode

```
vq_curaddress( handle, &row, &col )
```
Enquire current alpha cursor position

```
status = vq_tabstatus( handle )
```
Enquire whether a mouse or tablet is available

```
v_dspcur( handle, x, y )
```
Display graphics cursor at specified location

```
v_rmcur( handle )
```
Hide graphics cursor

Lib8

```
id = appl_init()
```
Initialise the AES. This call must precede any use of VDI or AES.

```
rc = appl_read( id, n, buffer )
```
Read bytes from a message pipe

```
rc = appl_write( id, n, buffer )
```
Write bytes to a message pipe

```
id = appl_find( name )
```
Find the id of another application

```
appl_tplay( buffer, n, speed )
```
Replay a recording of user interaction created with appl__trecord

```
n = appl_trecord( buffer, size )
```
Make a recording of user interactions

```
rc = appl_exit()
```
Close down the application library

Lib9

```
keycode = evnt_keybd()
```
Wait for a keyboard event

```
n = evnt_button( clicks, mask, state, &x, &y, &button,
                 &kstate )
```
Wait for a mouse button event

```
evnt_mouse( flags, x, y, w, h, &mx, &my, &button,
            &kstate )
```
Wait for a mouse movement event

```
evnt_mesag( buffer )
```
Wait for a message event

```
evnt_timer( low, high )
```
Wait for a timer event

```
status = evnt_multi( flags, clicks, mask, state,
                     flags1, x1, y1, w1, h1,
                     flags2, x2, y2, w2, h2,
                     buffer, low, high, &x, &y,
                     &button, &kstate, &keycode,
                     &n )
```
Wait for a mixture of events

```
current = evnt_dclick( new, mask )
```
Set or read back the mouse double click delay

Lib10

```
rc = menu_bar( tree, flag )
```
Display or erase the menu bar

```
rc = menu_icheck( tree, item, flag )
```
Display or erase a check mark next to menu item

```
rc = menu_ienable( tree, item, flag )
```
Enable or disable a menu item

```
rc = menu_tnormal( tree, title, flag )
```
Display menu title in normal or reverse video

```
rc = menu_text( tree, item, text )
```
Change the text of a menu item

```
menuid = menu_register( id, text )
```
Install desk accessory menu item string on the Desk menu

Lib11

```
rc = objc_add( tree, parent, child )
```
Add an object to the object tree

```
rc = objc_delete( tree, object )
```
Delete an object from the object tree

```
rc = objc_draw( tree, object, depth, x, y, w, h )
```
Redraw the objects in a subtree using specified clip rectangle

```
obj = objc_find( tree, object, depth, x, y )
```
Find an object under the mouse cursor

```
rc = objc_offset( tree, object, &x, &y )
```
Enquire the position of an object

```
rc = objc_order( tree, object, position )
```
Move an object to a new position in parent's list of children

```
rc = objc_edit( tree, object, char, index, type,
                &newindex )
```
Allow the user to edit text within an object

```
rc = objc_change( tree, object, dummy, x, y, w, h,
                  state, redraw )
```
Change the state of an object, and possibly redraw it

Lib12

obj = form_do(tree, object)
Start an interaction with the user

rc = form_dial(flag, x1, y1, w1, h1, x2, y2, w2, h2)
Handle dialogue boxes

button = form_alert(defbutton, text)
Display an alert message on the screen

button = form_error(n)
Display an alert according to the error number

form_center(tree, &x, &y, &w, &h)
Calculate the position of a form centered on the screen

Lib13

rc = graf_rubberbox(x, y, w, h, &fw, &fh)
Draw a rubber box, expanding and contracting as user drags it

rc = graf_dragbox(bw, bh, bx, by, x, y, w, h, &fx, &fy)
Allow the user to drag a box within a rectangle

rc = graf_movebox(w, h, sx, sy, dx, dy)
Draw a box moving from one place to another

rc = graf_growbox(sx, sy, sw, sh, fx, fy, fw, fh)
Draw an expanding box outline

rc = graf_shrinkbox(fx, fy, fw, fh, sx, sy, sw, sh)
Draw a shrinking box outline

pos = graf_watchbox(tree, object, instate, outstate)
Track mouse moving in and out of object

pos = graf_slide(tree, parent, object, direction)
Track a sliding box within a parent

handle = graf_handle(&cw, &ch, &bw, &bh)
Return the VDI handle used by the AES

rc = graf_mouse(code, buffer)
Change the shape of the mouse cursor

graf_mkstate(&x, &y, &button, &kstate)
Enquire the current position of the mouse and keyboard shift keys

Lib 14

rc = scrp_read(buffer)
Read the current scrap directory name

rc = scrp_write(buffer)
Write a new scrap directory name

Lib 15

rc = fsel_input(pathspec, filename, &button)
Display file selector box and allow the user to interact with it

Lib16

```
wid = wind_create( flags, x, y, w, h )
```
Create (but do not display) a new window

```
rc = wind_open( wid, x, y, w, h )
```
Display a window in initial size

```
rc = wind_close( wid )
```
Close a window. It may be opened again by wind__open if required

```
rc = wind_delete( wid )
```
Delete a window and free the space used

```
rc = wind_get( wid, field, &p1, &p2, &p3, &p4 )
```
Get information about a window

```
rc = wind_set( wid, field, p1, p2, p3, p4 )
```
Set window attributes

```
wid = wind_find( x, y )
```
Find which window is under the mouse cursor

```
rc = wind_update( flag )
```
Interlock window updates

```
rc = wind_calc( mode, flags, ix, iy, iw, ih, &ox, &oy,
                &ow, &oh )
```
Calculate size of work area or entire window

Lib17

```
rc = rsrc_load( filename )
```
Load resource file into memory after allocating space for it

```
rc = rsrc_free( )
```
Free the memory allocated by rsrc__load

```
rc = rsrc_gaddr( type, index, &address )
```
Get the address in memory of an item in the resource file

```
rc = rsrc_saddr( type, index, address )
```
Store the address of a data structure

```
rc = rsrc_obfix( tree, object )
```
Convert the size of an object from character coordinates to pixels

Lib18

```
rc = shel_read( command, tail )
```
Get command name and command tail

```
rc = shel_write( exec, graf, gem, command, tail )
```
Start up another application when this one terminates

```
rc = shel_find( buffer )
```
Find a filename in the current directory and in a preset list

```
rc = shel_envrn( pvalue, buffer )
```
Search environment for a parameter and return pointer to it

Appendix A: Pascal Syntax - Quick Reference Guide

A Pascal program has the following basic outline:

```
(program heading)                              1. The word PROGRAM
                                               and the program's
                                               title

PROGRAM <heading>

(GOTO label declarations)                      2. Declarations of
                                               types, variables,
                                               and so forth
LABEL  1,9999;

(constant definitions)
CONST  <identifier> = <literal>;

(type definitions)                             3. Definitions of
                                               procedures and
                                               functions

TYPE   <identifier> = <type>;

(variable declarations)
VAR    <identifier(s)> : <type>;

(subprogram declarations)

PROCEDURE or FUNCTION <heading>;

BEGIN                                          4. The word BEGIN

(program statements)                           5. Any number of
```

```
                                     statements divided
                                     by semicolons
END.
                                      6. The word END
                                     followed by a period
```

Type definitions

Predefined types:

```
INTEGER BOOLEAN CHAR REAL
```

Enumerated types:

```
TYPE colours = (red, blue, green, yellow);
```

Subrange types:

```
TYPE SomeIntegers = 10..100;
     SomeColours  = red..green;
```

SET types:

```
TYPE NumberSet = SET OF 1..100;
     ColourSet = SET OF SomeColours;
```

ARRAY types:

```
TYPE AnArray  = ARRAY [1..40,char] OF red..green;
     Paintbox = PACKED ARRAY [colours] OF BOOLEAN;
```

RECORD types:

```
TYPE ARecord = RECORD
                 {There are 4 fixed fields...}
                 Field1 : INTEGER;
                 Field2 : 'a'..'m';
                 Field3 : (white, grey, black);
                   RECORD
                      .
                      .
                      .
                   END;
                 Field4 : ARRAY [1..4] OF 'a'..'d';
                 {...and one variant field}
                 CASE ATag = ATagType OF
                   Select1 : ( Field5 : REAL );
                   Select2 : ( Field6 : BOOLEAN );
               END;
```

FILE types:

```
TYPE Collection = FILE OF ARecord;

     Somenums   = FILE OF INTEGER;
```

Pointer types:

```
TYPE Location = ^ARecord;
```

Variable Declarations

```
VAR i,num,digits : INTEGER;

       SomeInfo   : ARecord;
```

Procedure and Function Declarations

As for the program block, except for the heading and ending with a ;.

```
PROCEDURE ASubroutine ( i : INTEGER; VAR n : REAL );
VAR j,k : INTEGER;
BEGIN
  .
  .                          (procedure statements)
  .
END;

FUNCTION ASubroutine : REAL;
```

```
VAR i,j,k : INTEGER;
BEGIN
  .                                   (function statements)
  .
  ASubroutine := 5.0
END;
```

Statements

Assignment statements:

```
Answer := Result;

Answer := a * b / c + d;

ASet := [1, 2, 3, x..y, 7];
```

GOTO statements:

```
GOTO 2;

2 : k := y; (target)
```

IF statements:

```
IF (Answer = 5) OR (Result <> 7) THEN
  BEGIN
   .
   .                                  (statements)
   .
  END
 ELSE
  BEGIN
   .
   .                                  (statements)
   .
  END;
```

FOR statements:

```
FOR i := 10 TO 20 DO   (or FOR i := 20 DOWNTO 10 DO)
  BEGIN
   .
   .                                  (statements)
   .
  END;
```

WHILE statements:

```
WHILE NOT (Answer > 5) AND (RESULT < 12) DO
  BEGIN
   .
   .                                  (statements)
   .
  END;
```

REPEAT statements:

```
REPEAT
  .
  .                                   (statements)
  .
UNTIL (Answer <=5) OR (RESULT >=17);
```

CASE statements:

```
CASE Answer OF
1,2 : BEGIN
       .
       .
      END;
{ : <statement>
END;
```

WITH statements:

```
WITH ARecord DO
 BEGIN
   Field := 5;

END;
```

Arithmetic expressions:

```
Num1 + Num2
Num1 - Num2
Num1 * Num2
Num1 / Num2
Num1 DIV Num2      { integers only }
Num1 MOD Num2      {             }
```

Appendix B: Compile-time Error Messages

This appendix lists the error numbers and corresponding messages displayed by the Pascal compiler. When you generate an error, you should receive both the number and the message.

1 :	Illegal character
2 :	Illegal character
3 :	File ends inside quoted string
4 :	File ends inside a comment
5 :	Integer part of number is too large
6 :	PROGRAM expected
7 :	Identifier expected
8 :	')' expected
9 :	';' expected
10 :	A block cannot start with this symbol
11 :	Missing dot at end of program
12 :	Text encountered after end of program
13 :	BEGIN expected
14 :	A procedure has been declared as forward but has not been found
15 :	Syntax error
16 :	A label must be an INTEGER constant
17 :	Label number expected
18 :	'=' expected
19 :	Type has been implicitly declared, but actual definition not found
20 :	':' expected
21 :	Undeclared label
22 :	This kind of identifier cannot be used to start a statement
23 :	Type expected
24 :	'OF' expected

25 : ')' expected
26 : Line too long, it will be truncated
27 : Only two digits are permitted in the E field of a real number
28 : Unexpected end of source file encountered
29 : Commas must be used between labels
30 : A type identifier must follow '^'
31 : ']' expected
32 : '[' expected
33 : Files cannot contain files
34 : END expected
35 : '..' expected
36 : Type mismatch between subrange bounds
37 : The first bound of the subrange is greater than the second
38 : Illegal subrange type
39 : Constant expected
40 : Number expected
41 : Type identifier expected
42 : Identifier already declared in this block
43 : Identifier not declared
44 : Too many elements in type
45 : Type is not countable
46 : Constant must be of another type
47 : Block name expected
48 : The previous forward declaration does not agree
49 : The parameter list should not be repeated
50 : This block has been declared as forward for the second time
51 : Parameter expected
52 : Function return type must be pointer, subrange, real or ordinal
53 : Maximum code size for main procedure exceeded
54 : ';' expected
55 : Cannot READ or WRITE zero items
56 : A field width must be of type integer
57 : Expression cannot be written
58 : This relational operator cannot be used between these types
59 : An expression of type PACKED ARRAY OF CHAR required
60 : The IN operator cannot be used between these types

61 : The '+' and '-' operators can only be used on integer and real types
62 : The OR and AND operators can only be used between boolean operands
63 : The '+' and '-' operators cannot be used between these operands
64 : Unable to reopen file for updating
65 : Unimplemented feature
66 : The MOD and DIV operators may only be used between integer operands
67 : The '*' operator may not be used between these operands
68 : Invalid operand
69 : The NOT operator can only be applied to boolean operands
70 : The '^' symbol may only be used for pointer and file variables
71 : Internal compiler error
72 : A dot follows a variable which is not a record
73 : Field not known
74 : Only arrays may be subscripted
75 : The expression type is incompatible with the index type of this array
76 : ':=' expected
77 : Variable and expression are not assignment compatible
78 : Expression too complex
79 : DO expected
80 : UNTIL expected
81 : THEN expected
82 : The variable of a for loop must be a local variable
83 : The variable of a for loop must be of an ordinal type
84 : TO or DOWNTO expected
85 : Subscript value out of bounds
86 : Division by zero
87 : Case label expected
88 : Empty case statement body
89 : The case constant appears twice
90 : Parameter list expected
91 : Number of parameters does not agree with declaration
92 : Extra comma, it will be ignored

93 : Variable of different type required
94 : An element of a packed structure cannot be used as a VAR parameter
95 : Procedural parameter is not identical to the requirements of the parameter list
96 : Expression of different type required
97 : The argument to NEW or DISPOSE must be a pointer
98 : Only the current function may be assigned to
99 : A boolean expression is required
100 : The empty string is not permitted
101 : Label already defined
102 : Label has been declared but not defined
103 : Label already declared
104 : Placement of label invalidates previous GOTO statement
105 : Label numbers must be in the range 0 to 9999
106 : Label is not accessible from this point in the program
107 : The identifier cannot be redefined in this scope
108 : External procedures may only be declared at the outermost level
109 : RESET and REWRITE may only be applied to files
110 : RESET and REWRITE may not be used on the standard files input and output
111 : EOLN, PAGE, READLN and WRITELN may only be applied to textfiles
112 : Cannot write to input or read from output
113 : Record type required
114 : A file is required here
115 : Items within a set constructor must have identical types
116 : Not enough space - try increasing workspace size
117 : The MOD operator must have a positive, non zero, argument
118 : Unimplemented instruction
119 : Parameter should be of type unpacked array
120 : Parameter should be of type PACKED array
121 : Subscript parameter is incompatible with the subrange of the unpacked array parameter
122 : Array host types are not identical

123 : Same control variable in nested for statements
124 : Cannot assign to a for statement control variable
125 : Cannot pass a for statement control variable as a variable parameter to a subprogram
126 : Cannot call READ or READLN with a for statement control variable as parameter
127 : For statement control variable is threatened by a procedure or function
128 : The argument to DISPOSE must be a variable or function of type pointer
129 : The argument to INCLUDE must be a filename in quotes
130 : Unable to open INCLUDE file for input
131 : INCLUDE cannot be nested to this depth
132 : Too many case constants supplied
133 : Case constants can not be variables
134 : This case constant does not match any of the variants
135 : This case constant is type incompatible with the corresponding variant
136 : A string can not be on more than one line
137 : The / operator may not be used between operands of these types
138 : The left-hand argument of the 'IN' operator must be ordinal
139 : File variables or structured variables with file components cannot be value parameters
140 : The case index must be an expression of ordinal type
141 : Field width must be an expression of ordinal type
142 : This function does not contain an assignment to its identifier
143 : Files and structured types containing files can not be assigned
144 : The actual parameter corresponding to a variable parameter must be a variable access
145 : A pointer variable must be a variable access
146 : The case constant list is incomplete
147 : This parameter cannot denote a field that is the selector of a records variant part
148 : The applied occurrence of the type identifier is within the scope of the field designator of the same name

149 : This case constant can never be reached
150 : Only integer, real or character values can be read from a textfile
151 : Variables in set constructors must be in the range 0..255
152 : Possible unclosed comment
153 : Program parameters can only be defined as variables

Appendix C: Collected Errors

This appendix is a list of collected errors. They are all trapped by the MCC Pascal 68000 run-time system with the exception of those marked by an asterisk (*). These errors mainly involve undefined variables or dynamic storage.

Array Types and Packing

1. An error results if the value of any subscript of an indexed-variable is not assignment-compatible with its corresponding index-type.

2. In a call of the form PACK (Vunpacked, StartingSubscript, Vpacked), it is an error if the ordinal-typed actual parameter (StartingSubscript) is not assignment-compatible with the index-type of the unpacked array parameter (Vunpacked).

3*. In a call of the form PACK (Vunpacked, StartingSubscript, Vpacked), it is an error to access any undefined component of Vunpacked.

4. In a call of the form PACK (Vpacked, StartingSubscript, Vpacked), an error results if you exceed the index-type of Vunpacked.

5. In a call of the form UNPACK (Vpacked, Vunpacked, StartingSubscript), it is an error if the ordinal-typed actual parameter (StartingSubscript) is not assignment compatible with the index-type of the unpacked array parameter (Vunpacked).

6*. In a call of the form UNPACK (Vpacked, Vunpacked, StartingSubscript), it is an error for any component of Vpacked to be undefined.

7. In a call of the form UNPACK (Vpacked, Vunpacked, StartingSubscript), it is an error to exceed the index-type of Vunpacked.

Record Types

8*. An error occurs if you access or reference any component of a record variant that is not active.

9. An error results if any constant of the tag-type of a variant-part does not appear in a case-constant-list.

10. An error results if you pass the tag field of a variant-part as the argument of a variable-parameter.

11*. An error results if a record that has been dynamically allocated through a call of the form NEW (p,C1...,Cn) is accessed by the identified-variable of the variable-access of a factor, of an assignment statement, or of an actual parameter.

File Types, Input and Output

12*. An error results if you change the value of a file variable f when a reference to it's buffer, buffer variable f, exists.

13. An error results if, immediately prior to a call of PUT, WRITE, WRITELN or PAGE, the file affected is not in the 'generation' state

14. An error results if, immediately prior to a call of PUT, WRITE, WRITELN or PAGE, the file affected is undefined.

15. An error results if, immediately prior to a call of PUT, WRITE, WRITELN or PAGE, the file affected is not at end of file.

16. An error results if the buffer variable is undefined immediately prior to the use of PUT.

17. An error results if the affected file is undefined immediately prior to any use of RESET.

18. An error results if, immediately prior to use of GET or READ, the file affected is not in the 'inspection' state.

19. An error results if, immediately prior to use of GET or READ, the file affected is undefined.

20. An error results if, immediately prior to use of GET or READ, the file affected is at end-of-file.

21. An error results if, in a call of READ, the type of the variable-access is not assignment compatible with the type of the value READ (and represented by the affected file's buffer-variable)

22. An error results if, in a call of WRITE, the type of the expression is not assignment compatible with the type of the affected file's buffer-variable.

23. In a call of the form EOF(f), an error results if f is undefined.

24. In any call of the form EOLN(f), an error results if f is undefined.

25. In any call of the form EOLN(f), an error results if EOF(f) is true.

26. When reading an integer from a textfile, an error results if the input sequence (after any leading blanks or end-of-lines are skipped) does not form a signed-integer.

27. When an integer is read from a textfile, an error results if it is not assignment compatible with the variable-access it is being attributed to.

28. When reading a number from a textfile, an error results if the input sequence (after any leading blanks or end-of-lines are skipped) does not form a signed-number.

29. An error results if the appropriate buffer variable is undefined immediately prior to any use of READ.

30. In writing to a textfile, an error results if the value TotalWidth or FractionalDigits, if used, is less than one.

Pointer Types

31. An error results if you try to access a variable through a NIL-valued pointer.

32*. An error results if you try to access a variable through an undefined pointer.

Dynamic Allocation

33*. An error results if you try to dispose of a dynamically-allocated variable when a reference to it exists.

34*. When a record with a variant part is dynamically allocated through a call of the form NEW (p,C1....) Cn, an error results if you activate a variant that was not specified (unless it's at a deeper level than Cn).

35*. An error results if you use the short form of DISPOSE (e.g., DISPOSE (p)) to deallocate a variable that was allocated using the long form (e.g., NEW (p,C1.....Cn)).

36*. When a record with a variant part is dynamically allocated through a call of the form NEW (p,C1....) Cn, an error results if you specify a different number of variants in a call of DISPOSE.

37*. When a record with a variant part is dynamically allocated through a call of the form NEW (p,C1....) Cn, an error to specify a different number of variants in a call of DISPOSE.

38. An error results if you call DISPOSE with a NIL-valued pointer argument.

39. An error results if you call DISPOSE with an undefined pointer argument.

Required Functions and Arithmetic

40*. For a call of the SQR function, an error occurs if the result does not exist.

41 In a call of the form LN (x), an error results if x is less than or equal to zero.

42. In a call of the form SQRT (x), an error results if x is negative.

43. For a call of the function TRUNC, an error occurs if the result is not in the range -MAXINT..MAXINT.

44. For a call of the function ROUND, an error occurs if the result is not in the range -MAXINT..MAXINT.

45. For a call of the function CHR, an error occurs if the result does not exist.

46. For a call of the function SUCC, an error occurs if the result does not exist.

47. For a call of the function PRED, an error occurs if the result does not exist.

48. In a term of the form x/y, an error results if y is equal to zero.

49. In a term of the form i DIV j, an error results if j is equal to zero.

50. In a term of the form i MOD j, an error results if j is zero or negative.

51. An error results if any integer arithmetic operation, or function whose result type is integer, is not computed according to the mathematical rules for integer arithmetic.

Parameters

52. An error results if an ordinal-typed value-parameter and it's actual-parameter are not assignment compatible.

53. An error results if a set-typed value-parameter and it's actual-parameter are not assignment compatible.

Miscellaneous

54*. An error results if a variable-access contained by an expression is undefined.

55*. An error results if the result of a function call is undefined.

56. An error results if a value and the ordinal-typed variable, or function-designator it is assigned to, are not assignment-compatible.

57. An error results if a set-typed variable, and the value assigned to it, are not assignment compatible.

58. On entry to a case-statement, an error results if the value of the case-index does not appear in a case-constant-list.

59. If a for-statement is executed, an error results if the types of the control-variable and the initial-value are not assignment-compatible.

60. If a for-statement is executed, an error results if the types of the control-variable and the final-value are not assignment-compatible.

Order of Evaluation:

The order of evaluation of

a. the indices of multidimensional arrays
b. the constituent members of set-contructors
c. member-designators in set-contructors
d. actual parameters in function and procedure calls
e. either side of assignment statements
f. the parameters of PACK and UNPACK

is generally left to right although the order may depend upon optimization features of the compiler.

In Boolean expressions not all of the operands may need to be evaluated. Thus if operands have side-effects (for example, function calls), the results may not be predictable.

Note: Although program parameters may be of any type, only those of type FILE are bound to be variables supplied at run-time.

Appendix D: Linker Errors and Warning Messages

This appendix lists the error and warning messages which can be produced by the linker in the phases in which they will be encountered.

Command Line Errors

The linker on encountering an error in the command line will display a message indicating the problem and reprompt for another command line. It will not attempt to parse the line following the error.

```
ERROR - 01 file name too long - <file name>
```

Either a file name entered on the command line or a default file name generated from the primary file is too long. The full file name can only be 44 characters long.

```
ERROR - 02 No link file given with the -WITH option
```

A -WITH option has been entered without a link file name. The -WITH option must be followed by a file name.

```
ERROR - 03 Page length missing following -PAGELEN
           option
```

The -PAGELEN expects a value to set the page length to for formatting on a printer.

```
ERROR - 04 Page length is not a number
```

The item following the -PAGELEN option is not a number.

ERROR - 05 Page length too small. Minimum is
            20 lines

As the listing output is formatted with headers, titles and subtitles the minimum realistic page length is 20 lines.

ERROR - 06 No input module or control file given

The linker requires as input either a module file name or a control file name. If neither is given then the linker does not have any input files to act upon.

ERROR - 07 Illegal option given on command
            line <option>

An unrecognised option has been entered. The option parameter indicates which option the linker was unable to recognise.

Control File Errors

The linker will on encountering an error in the control file list the line for which the error has occurred and print a message indicating the cause of the error. The linker will process the rest of the control file but will not proceed with the link.

ERROR - 09 Illegal or unrecognised command <command>

An illegal or unrecognised command has been encountered in the control file. The command parameter is the command that the linker failed to recognise.

ERROR - 0A Too many parameters <parameter>

The linker has encountered too many parameters on the line. The command has been processed but the link will not be performed.

ERROR - 0B Not enough parameters, expecting <item>

The linker did not find enough parameters on the line. The item parameter indicates which item was expected which will be one of the following:

Item	Command
file name	INPUT, EXTRACT or LIBRARY
module name	EXTRACT
FROM keyword	EXTRACT
section name	SECTION
END or DUMMY	COMMON
value	OFFSET
symbol name	DEFINE
expression	DEFINE

ERROR - 0C No module name given in command line for
            INPUT *

The linker has encountered an INPUT * in the control file but no module name was given on the command line.

ERROR - 0D FROM keyword missed out or incorrectly
            spelt

In an extract command the FROM keyword was not found. This keyword must be present.

ERROR - 0E section already exists <section>

The section named in the section command has already been named in a previous SECTION command and so cannot be placed in the order requested.

ERROR - 0F Illegal option, DUMMY or END only allowed

An illegal common option has been given. The linker only recognises the keywords DUMMY and END.

ERROR - 10 Only one COMMON command allowed

Only one common command is allowed in any one link.

ERROR - 11 Symbol was used in DEFINE
command: <symbol>

A symbol being defined in a define command has already been used in a previous define expression. Forward referencing of defined symbols is not allowed.

ERROR - 12 Symbol is being redefined <symbol>

The symbol being defined has already appeared in a provious define command and cannot be redefined.

ERROR - 13 Syntax error in DEFINE command <expression>

The linker has detected an error in the syntax of the define command. The expression following the error message starts from the character position which caused the syntax error.

ERROR - 15 OFFSET value is not a number

The value following the offset command is not a number.

ERROR - 16 Only one offset value is allowed

As the OFFSET value is the start point for allocation of memory for the program only one value is allowed.

Low Level Errors

These errors are detected when parsing the line at a low level. The error messages are followed by a message indicating which command was being processed at the time the error was encountered.

ERROR - 19 numeric overflow

The numeric value following an OFFSET command is too large to fit in a 32 bit word.

ERROR - 1A Syntax error in number

The linker has detected an illegal character while processing a number. This is normally caused by a $ which is not followed by a hexadecimal digit.

ERROR - 1B Invalid character

The linker has detected an illegal character while processing a line.

ERROR - 1C Decimal number overflow

The linker has detected that a decimal number has overflowed to negative.

Processing Errors and Warnings

These errors are detected while processing the link after validation of all command inputs to the linker. The description of the error messages are followed by a description of the actions performed following the error. Warning messages always result in the linker continuing from the current position in the link.

ERROR - 1D EXTRACT module not found

The linker could not find the module requested in an extract command in the file specified. The linker will continue to process all remaining inputs in pass 1 and then prompt for another command line. The program file will not be produced.

ERROR - xx Error in relocatable binary
file <file name>

This error message indicates a problem with the relocatable binary file remaining input files in pass 1 and then prompt for another command line. The program file will not be produced.

ERROR - 2D Attempt to initialise dummy COMMON
in <file>

The linker has detected an attempt to place data into a COMMON section with the COMMON DUMMY option in effect. As no space is saved for the COMMON blocks they may not be initialised in this way. The linker will continue to process all remaining input files in pass 1 and then prompt for another command line. The program file will not however be produced.

ERROR - 2E Absolute section below OFFSET address
in <file name>

This error indicates that an OFFSET command has been given in the linker control file but an absolute section resides below the OFFSET address. The linker will continue but the part of the section below the OFFSET value will not be contained in the file.

ERROR - 31 Phasing error occurred in <file>

This is an internal linker error which should not occur.

ERROR - 32 Out of memory

The linker has run out of memory while trying to allocate more memory for internal tables. The linker will exit after printing this message.

ERROR - 33 Attempt to allocate large record

The linker has attempted to allocate a record which is larger than the current memory allocation. The linker will exit after printing this message. This should never occur.

ERROR - 34 Incompatible section type for
section <section>

This error indicates that a section has been used both as a normal relocatable section and as a COMMON section. The linker will process all remaining input files in pass 1 but no program file will be produced.

WARNING - 35 Insufficient memory for cross reference

This message indicates that the linker cannot allocate sufficient memory for the cross reference listing. The linker will continue but a normal symbol table listing will be given instead of a cross reference.

WARNING - 36 Truncation error at offset <offset>

This warning indicates that a value has had to be truncated to fit into a byte or word expression. The offset value gives the location in the output program at which the truncation has occurred. The linker will continue however the program may encounter problems if run.

WARNING - 37 Undefined symbol was used in DEFINE
expression: <symbol>

This warning indicates that a symbol which was used in the expression part of a DEFINE command is still undefined. This means that the rsult of the DEFINE command is also undefined.

ERROR - 3A Internal error

The linker has detected an internal error (consistency check). This error should never occur.

WARNING - 3B Multiply defined symbol <symbol>

This warning indicates that a symbol has been defined more than once in the link. The first value encountered will be the value used by the link.

```
WARNING - JE Abs section overlaps next one in <file>
```

This warning indicates that two absolute sections overlap each other in the program file. This means that the second absolute section will overwrite the first.

Operating System Errors

When the linker gets an error code from TOS the action taken is dependent on what the linker is trying to do when the error is encountered. The linker will take the following action on encountering errors:

(a) Open errors on files

These errors are reported by the linker. If the error occurs on opening the program, listing, debug or control file the linker will reprompt for a command line. If an error occurs on opening a relocatable object file the linker will continue until the end of pass 1 to validate that all other files may be opened.

(b) Read and write errors on files.

If the linker encounters a read or write error on a file (other than end of file on read) the linker will report the error and exit.

(c) Close errors on files

If the linker encounters an error on closing files the linker will report the error and continue.

The linker will display a message indicating the error which has occurred along with the name of the file which encountered the error.

In non-interactive mode all operating system errors will cause the linker to exit (including an open error).

Appendix E: Example Programs

Example Program 1

```
{ This program calls the Metacomco Graphics Interface
  to draw a circle, a triangle, a rectangle and some
  text on the screen. It should be compiled with the
  extend option and then linked with the graphics
  interface routines.                                  }

PROGRAM graphics( input );
CONST
  { Include the graphic constants include file.  Note
    that to save space, a copy of the constants file
    should be tailored to the requirements of your
    application                                        }

  INCLUDE 'grconsts.inc';

  message    = 'M E T A C O M C O ';
  null       = 0;                { Terminator for message  }
  chheight   = 50;               { Character height        }

TYPE
  { Include the graphic type definitions include file.
    Note that to save space, a copy of the types file
    should be tailored to the requirements of your
    application                                        }
```

```
   INCLUDE 'grtypes.inc';

VAR
    returncode,               { Function return code        }
    i          ,              { General purpose index       }
    handle     ,              { Device handle               }
    chh        ,              { Character height            }
    chw        ,              { Character width             }
    cellh      ,              { Cell height                 }
    cellw      : INTEGER;     { Cell width                  }
    workout    : intout;      { Dummy capabilities array    }
    workin     ,              { Environment array           }
    triangle   : intin;       { Triangle co-ordinates       }
    box        : coord4;      { Box co-ordinates            }
    string     : gtextstr;    { Message string for gtext    }
    anything   : CHAR;        { Read keyboard character     }

{ Include the graphics procedures and functions file.
  Note that to save space, a copy of the include file
  should be tailored to the requirements of your
  application                                            }

INCLUDE 'grrtns.inc';

FUNCTION mypack( hi, lo : INTEGER ) : INTEGER;
{ Internal function to pack two integers into 32 bits }
BEGIN
   { The factor moves hi into the high order word }
   mypack := hi * 65536 + lo
END;

BEGIN
   { Set up the environment array with default values }
   FOR i := 1 TO 5 DO workin[i] := mypack( 1, 1 );
   workin[6] := mypack( 2, 0 );
```

```
   { Open a virtual workstation and get its handle }
   v_opnvwk( workin, handle, workout );

   { Clear the screen }
   v_clrwk( handle );

   { Set fill interior style to patterned }
   returncode := vsf_interior( handle, pattern );

   { Draw a circle with brick pattern interior }
   returncode := vsf_style( handle, bricks );
   v_circle( handle, 200, 200, 200 );

   { Change style to woven }
   returncode := vsf_style( handle, weave );

   { Set coordinates of triangle }
   triangle[1] := mypack( 600, 50 );
   triangle[2] := mypack( 450, 300 );
   triangle[3] := mypack( 100, 300 );

   { Draw a filled triangle }
   v_fillarea( handle, 3, triangle );

   { Change style to dots }
   returncode := vsf_style( handle, dots);

   { Set coordinates of rectangle }
   box[1] := mypack( 150, 150 );
   box[2] := mypack( 500, 250 );

   { Draw a rounded rectangle }
   v_rfbox( handle, box );
```

```
    { Set graphics size and style }
    vst_height( handle, chheight, chw,chh,cellw,cellh );
    returncode := vst_effects( handle, thickened );

    { Make 'C' null terminated string }
    string := message;
    string[18] := CHR( null );

    { Output the message }
    v_gtext( handle, 190, 215, string );

    { Close virtual workstation and exit }
    v_clsvwk( handle );

    { Give viewer a chance to study the picture }
    READ( anything );
END.
```

Example Program 2

The following example allows memory to be updated. It should be noted that this is not always a good idea as it may crash your machine. The purpose of this program is to show how assembler programs may be called from Pascal. If you intend to implement a peek/poke program, the Pascal program ought to check for suitable addresses etc. The Pascal program should be compiled with the extend option on and then linked with the assembler routines. A typical LINK68 command line would be:

```
{****************** Pascal Program ******************
*                                                    *
*  An example to show how to call assembler routines *
*  from Pascal. This program must be linked with the *
*  assembler programs Peek and Poke.                 *
*                                                    *
******************************************************}

PROGRAM Memory( input, output );

VAR
   address, contents, value : INTEGER;

{ External assembler procedure to Poke a location }

PROCEDURE Poke( address, value : INTEGER ); EXTERNAL;

{ External assembler function to Peek a location }

FUNCTION Peek( address : INTEGER ) : INTEGER; EXTERNAL;

BEGIN  { Memory }

   WRITELN( 'Input address to change' );
   READLN( address );
```

```
  WRITELN( 'Input new value' );
  READLN( value );

  Poke( address, value );

  WRITELN( 'Long word contents of ', address,
           ' changed to ', value);

  WRITELN( 'Input address to examine' );
  READLN( address );

  contents := Peek( address );

  WRITELN( 'Long word contents of ', address,
           ' is ', contents);

END.  { Memory }


****************** Assembler Program ******************
*                                                     *
* Support routines for the Pascal Memory example.     *
* The Poke routine update the memory location given   *
* with a value. The Peek routine returns the long     *
* word contents of the supplied memory address.       *
*                                                     *
*******************************************************

* The two routines are called with a standard 'C'
* calling sequence (see Chapter 8)

     XDEF      peek,poke
```

```
peek  MOVE.L
      MOVE.L   4(SP),A3        Get address to peek
      MOVE.L   (A3),D0         Return contents
      RTS                      Return to Pascal

poke  MOVE.L   4(SP),A3        Get address to poke
      MOVE.L   8(SP),(A3)      Update memory
      RTS                      Return to Pascal

      END
```

Example Program 3

```
{

 A little program to change the stack (bss) size of
 CP/M-68K object module format program (in particular
 the Pascal compiler PASCAL.TTP).

 This program can be used to update the compiler if
 it runs out of stack during compilation. It reads in
 a stacksize and copies the file matched with oldfile
 to the file matched with newfile changing the stack
 size in the file header.

 This program should only be used with CP/M-68K files.
 Ensure that the new version works before overwriting
 writing the old version !

 Note that this program should be compiled using the
 extend option of the compiler because CHR requires
 a range of 0 .. 255 which is non-standard but
 available as an extension.
                                                    }
```

```
PROGRAM changestack( oldfile, newfile, input, output );

{ Define three useful constants for use in converting
  the stacksize into single byte form }

CONST
   ffffff = 16777216;
   ffff   = 65536;
   ff     = 256;

VAR
   oldfile ,newfile              : TEXT;
   ch                            : CHAR;
   i, stacksize                  : INTEGER;
   byte0, byte1, byte2, byte3    : INTEGER;

{ Routine to read a character from the oldfile and
  write it to the newfile }

PROCEDURE rw;
VAR
   ch : CHAR;
BEGIN
   READ( oldfile, ch );
   WRITE( newfile, ch );
END;

BEGIN
   { Open the oldfile for reading, create the newfile
     and open it for writing }
   RESET( oldfile );
   REWRITE( newfile );
```

```
   { Output a prompt for the new stacksize and read
     it in }
   WRITELN( 'Input the new stack size : ' );
   READLN( stacksize );

   { Read begining of file header }
   FOR i := 1 TO 10 DO rw;

   { Read over old stack size }
   FOR i := 1 TO 4 DO READ( oldfile, ch );

   { Calculate the 4 bytes of the stacksize given }
   byte0 :=  stacksize             DIV ffffff;
   byte1 := (stacksize MOD ffffff) DIV ffff
   byte2 := (stacksize MOD ffff  ) DIV ff     ;
   byte3 :=  stacksize MOD ff                 ;

   { Write out new stack size }
   WRITE( newfile, CHR(byte0), CHR(byte1), CHR(byte2),
          CHR(byte3) );

   { Copy the rest of the oldfile to the newfile }
   WHILE NOT EOF( oldfile ) DO rw;
END.
```

Appendix F: Compliance Statement

MCC Pascal 68000 is an implementation of a standard Pascal which has passed validation by the British Standards Institution under the ISO Standard 7185 "Specification for computer language PASCAL". The implementation-defined features are as follows:

E.1	The value of each char-type corresponding to each allowed string-character is the corresponding ISO character. See ISO 646 (ASCII).
E.2	The subset of real numbers denoted by signed real are the values representable with 32-bit floating point. This is about 7 decimal places.
E.3	The values of char-type are the ISO character set. See ISO 646 (ASCII).
E.4	The ordinal numbers of each value of char-type are the code values given in ISO 646 (ASCII).
E.5	The point at which the file operations REWRITE, PUT, RESET, and GET are performed, determined by the normal conventions of the operating system. Control is not returned to the program until the operation has been completed. Note that there is line by line buffering for normal interactive I/O. However, the lazy I/O ensures that prompts can be written before input is read.
E.6	The value of MAXINT is 2147483646
E.7	The accuracy of the approximations of the real operations and functions is determined by the representation (see E.2), and by

the truncation of intermediate results. This gives approximately 7 decimal digits of precision.

E.8 The default value of TotalWidth for integer-type is 12

E.9 The default value of TotalWidth for real-type is 13

E.10 The default value of TotalWidth for Boolean-type is 5

E.11 The value of ExpDigits is 2

E.12 The exponent character is 'E' (Upper case).

E.13 The case used for output of the values of Boolean-type is upper case.

E.14 The procedure page outputs the form-feed character (ASCII decimal 12). The effect on any particular device depends upon that device.

E.15 File-type program parameters should be bound to the program by the usual operating system mechanism.

E.16 REWRITE does not overwrite previous output to the standard file output. RESET sets the file variable to the first component of the standard file output.

E.17 The equivalent symbol to '' is implemented.
The equivalent symbol to '{' is implemented.
The equivalent symbol to '}' is implemented.

The following errors are not, in general, reported:

D.2, D.4, D.5, D.6, D.19, D.20, D.21, D.22, D.25, D.27, D.30, D.32, D.43, D.48

The following errors are detected prior to, or during execution of a program:

D.1, D.3, D.7, D.8, D.9, D.10, D.11, D.12, D.13, D.14, D.15, D.16, D.17, D.18, D.24, D.23, D.26, D.28, D.29, D.31, D.33, D.34, D.35, D.36, D.37, D.38, D.39, D.40, D.41, D.42, D.44, D.45, D.46, D.47, D.49, D.50, D.51, D.52, D.53, D.54, D.55, D.56, D.57, D.58,

The processor does not contain any extensions to ISO 7185 (such extensions must be enabled by means of a compiling option, not the subject of validation).

Implementation dependent features F.1 - F.7, F.10 and F.11 of Pascal are treated as undetected errors. If the procedure page is used to write to a file then the effect of reading from that file is to read the form-feed character (F.8) The binding of variables denoted by program parameters which are not of file-type is treated as an undetected error (F.9)