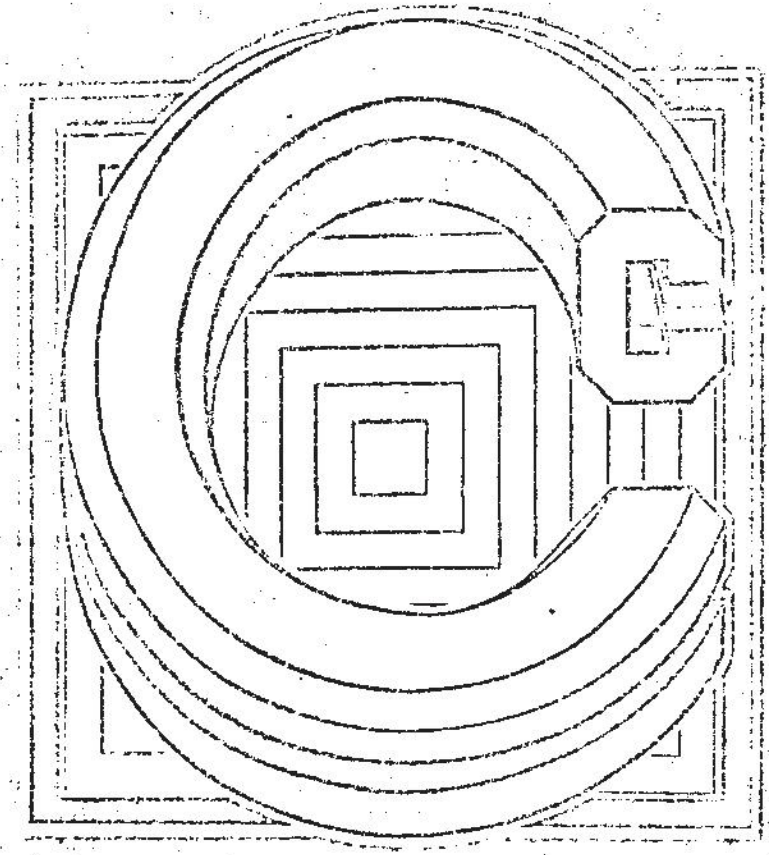
QLC DEVELOPMENT KIT



METACOMCO

QL C DEVELOPMENT KIT

- q l c
- 1° - flpt_
- 2° - flpt_ nombre proy.
- 3° - Opciones - Enter - -

Exec flpt_ link

cadena de comando.

flpt_ nombre/del/proyecto flpt_ proy_ link

Ej: flpt_ cuenta_0

METACOMCO

26 Portland Square, Bristol BS2 8PZ.

QL C Development Kit

Contents

Chapter 1: Using QL C

This manual provides a functional description of an implementation of the Lattice C compiler, a portable compiler for the high level programming language called C, on the Sinclair QL computer. It makes no attempt to discuss either programming fundamentals or how to program in C itself. Extensive reference is made to the definitive text "The C Programming Language", by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, Inc., Englwood Cliffs, New Jersey, 1978), which also provides an excellent tutorial introduction to the language. An alternative tutorial is "C at a Glance" by Adam Denning, published by Chapman and Hall

This manual is divided into six sections. This first describes the operating instructions for using the compiler on the QL. The second details the screen editor provided as part of this development kit. The third details the linker. The fourth section is a description of the language accepted by the compiler, which differs from the standard in only a few minor details. The fifth section presents the portable library functions in functional groups with calling sequences and examples. The final chapter describes QL C's general strategy for code generation Details of all error messages can be found in the appendices.

As this manual is intended to serve as a reference manual, each topic is usually presented in full technical detail as it is encountered. Some reference to sections not yet encountered is therefore unavoidable. I these references are specifically noted.

1.1 Lattice C on the QL

The QL C compiler generates programs to be executed on the QL. It accepts text files containing programs written in the C programming language and produces relocatable machine code in Sinclair format which is then linked to produce an executable file.

In your development kit you should have the following items:

1. Microdrive cartridge 1: C compiler phase 1
2. Microdrive cartridge 2: C compiler phase 2
3. Microdrive cartridge 3: Screen Editor, C run-time system plus linker
4. An EPROM containing part of the QL C
5. The "QL C Development Kit" manual

We strongly recommend that you make backup copies of the three microdrives cartridges and keep the master copies in a safe place.

Use of the EPROM cartridge

As has been mentioned above, part of the QL C is provided on an EPROM. The QL C EPROM is encased in a plastic cartridge which can be inserted into the machine whenever the language is required.

To insert your EPROM, first POWER DOWN your QL. This is very important! Then remove the cover from the QL socket marked "ROM" which is located on the left at the rear of the computer. The EPROM cartridge can now be pushed carefully into this socket. Once the cartridge is in, the machine can be powered up.

Having selected the required screen (either TV or monitor) the EPROM can now be verified. It is not essential to do this; if the title comes up, it should be working. However, as part of it may be missing or damaged, it is a good idea to check. To verify your EPROM, type:

```
ROM
```

The EPROM will then run a check on itself. If it is working, the message "QLC VERSION", followed by the version number, will appear on the top, left hand corner of the screen. If the EPROM is faulty then the message

"BAD ROM" will appear and you should contact Metacomco for further assistance. After the EPROM has been verified, the language can be used.

1.2 Module Compilation

The compiler is implemented via two SuperBasic procedures, LC1 and LC2. These load each phase of the compiler from any specified device (eg microdrive, floppy). Each phase performs a portion of the compilation process and must be invoked by separate commands; LC1 does not automatically load LC2 when it completes its processing. Normally, LC2 should be executed immediately after LC1 if there are no errors in the source file. The compilation process can be diagrammed as follows:

```
file_C -> LC1 -> file_Q
file_Q -> LC2 -> file_O
```

LC1 reads a C source file, whose name must end in the two characters __C, and (provided there are no fatal errors) produces an intermediate file of an identical name, except that it ends in the two characters __Q. LC2 reads an intermediate file created by LC1 and produces a binary file of the same name but ending in __O. The intermediate file is deleted by LC2 when it completes its processing. Each phase normally creates its output file on the same device as the input file. Note that if a source file defines more than one function, so does its resulting object file. Individual functions cannot be broken out from the object file when a program is linked.

The object file produced by the compiler must be incorporated with other object files and with run-time support subroutines in order to produce executable 68000 code. This can be accomplished by using LINK. The use of LINK is described in section 1.3 and more fully in chapter 3.

During compilation on an unexpanded (128K) QL you will notice the compiler using the screen-area of memory as workspace. This will not usually happen on a QL with memory expansion.

1.2.1 Phase 1

The first phase of the compiler reads a C source file and produces an intermediate file of logical records called quadruples, or quads. See section 1.5 for a more detailed discussion of the processing performed.

The microdrive cartridge containing the first phase of the compiler (Cartridge 1) should be inserted into one of the drives and the cartridge containing your C program should be inserted into the other drive. The format of the command to invoke the first phase of the compiler is:

```
LC1 "device [=stack] [%workspace] [>listfile]
      [options] filename"
```

The various command line specifiers are shown in the order they must appear in the command. Required specifiers are shown in bold type; optional specifiers are shown enclosed in brackets.

device — Tells LC1 which device to load the first phase of the compiler from. This can be any legal QDOS device name (e.g. mdv1_)

=stack — Overrides the number of bytes reserved for the compiler's execution stack. The default is 2048 (decimal) bytes, which is sufficient to compile most programs. If present, the stack size override field must be the first field after the device name. It is specified as an equals sign followed by a decimal number (for example, =4096 specifies a value of 4096 decimal bytes). Since the compiler uses recursion to process C statements, heavily-nested statements cause the compiler to use more stack space than straightforward, linear sequences. If a source program with many embedded statements (ifs within ifs within ifs, etc.) causes the first phase to terminate execution with a STACK OVERFLOW error message, the program should compile successfully if LC1 is re-executed using an increased stack size, such as 4096. Some experimentation may be required to determine the necessary stack size. On systems which are cramped for memory, the stack size may be trimmed down in an attempt to eliminate a 'Not enough memory' error; there is no guarantee, however, that the compilation will be successful, particularly if the stack size is reduced below 1024 bytes.

%workspace — Overrides number of bytes allocated for the compiler's workspace. The default is 20480 (decimal) bytes. If a source program causes the first phase to terminate execution with a NOT ENOUGH MEMORY error message, the program should compile successfully if LC1 is re-executed using an increased workspace size, such as 40000.

>listfile — Causes the first phase messages to be written to a specified file. These messages include the compiler sign-on message and any error or warning messages which may be generated. The full file name must be specified. If the file already exists it is overwritten. This option is useful for reviewing long lists of error messages.

options — Compile time options are specified as a hyphen followed by a single letter; in some cases, additional text may be appended. The option letter may be supplied in either upper or lower case. Each option must be specified separately, with a separate hyphen and letter. Available options are:

-b — This option is only necessary if postion-independent code is desired. It informs the first phase that all static and external data is to be addressed using a base register, either A5 or A6, thus limiting the total size of static data objects to 64K bytes. Which address register will be used depends on whether the -f option is specified on LC2 (see section 1.2.2 below); A5 is the default.

-c[flags] — Controls the various compatibility modes of the compiler, which allow it to accept source files

compiled with a previous version of the Lattice compiler. Each flag is specified as a single letter in either upper or lower case; more than one flag may be attached to the -c (for example -cusw), but no blanks are permitted. The flag letters recognized are:

c — Allows comments to be nested; the default is that comments do not nest, in accordance with the generally accepted convention.

d — Allows the dollar sign ($) to be used as an embedded character in identifiers.

m — Permits the use of multiple character constants (for example, 'XX').

s — Causes the compiler to generate only one copy of identical string constants. By default, the compiler generates unique copies of all string constants, even if they are identical.

u — Forces all char declarations to be interpreted as unsigned char.

w — Shuts off the warning generated for return statements which do not specify a return value inside an int function. (Functions which do not return a value should be declared void.)

-d — Causes debugging information to be included in the quad file. Specifically, line separator quads are interspersed with the normal quads. This allows the second phase to collect information relating input line numbers to program section offsets. A special debugger is required in order to use this information, which is not supplied with the package.

-dsymbol
-dsymbol=value — Causes the identifier "symbol" to be defined, as if the compiler had encountered a #define command for it. One of two forms of the option may be used. The first form merely defines the symbol with a null substitution text; the equivalent C statement is

```
#define symbol
```

The second form uses an equal sign to attach a substitution text "value"; its equivalent is

```
#define symbol value
```

Several definitions can be made in the same LC1 command; however, macros with arguments cannot be defined from the command line. This feature allows source files containing conditional compilation directives (#ifdef, #ifndef, #if, #else, #endif) to be used to produce different results without modifying the source file, simply by defining the appropriate symbol on the LC1 command.

-l — Forces alignment of all data elements except char and short to a byte offset divisible by four. This option is provided in the event that code is to be generated for a later version of the 68000 series which may perform long operations more efficiently if the values are stored at modulo 4 addresses.

-n — Causes the compiler to retain up to 31 characters for all identifier symbols, including #define symbols. The default symbol retention length is 8 characters

-oprefix — Specifies that the output filename (the __Q or quad file) is to be formed by prepending the input filename (the __C file which is being compiled) with the string prefix. Any device prefix originally attached to the input filename is discarded before the new prefix is added. No intervening blanks are permitted in the string following the o.

-u — Cancels all automatic symbol definitions for the current compilation. Certain #define symbols are normally pre-defined by the compiler (see below); this flag cancels all of those definitions.

-x — Changes the default storage class for external declarations (made outside the body of a function) from external definition to external reference. The usual meaning of an external declaration for which an explicit storage class is not present is to define storage for the object and make it visible in other files. The -x option causes such declarations to be treated as if they were preceded by the extern keyword, that is, the object being declared is present in some other file.

filename — Specifies the name of the C source file which is to be compiled. The filename should be specified without the __C extension; the first phase supplies the extension automatically. Alphabetic characters may be supplied in either upper or lower case. Note that only files with a __C extension can be compiled; if some other extension is specified, the compiler ignores it and tries to find name.C. (#include files, on the other hand, must be fully specified with extensions.) The quad file is created on the same device as the source file unless the -o option is used.

Include files may be specified enclosed in double quotes ("filename") or angle brackets (<filename>); the two forms have exactly the same effect. The name between the delimiters is taken at face value; the extension must be specified if one is defined for the file. The usual convention is to use __H for all header files. Alphabetic characters in a file name may be specified in either upper or lower case.

As an assistance to conditional compilation, the compiler automatically #defines several symbols, which can be tested in #if, #ifdef, or #ifndef directives to select appropriate code sequences according to the target processor or operating system. Two symbols are always defined in the compiler:


```
#define M68000 1
#define SPTR 1
```

If the -d flag was specified (as "-d", not "-dsymbol"), the followi symbol is defined:

```
#define DEBUG 1
```

The automatic definition of these symbols can be prevented by use of t -u option, which cancels all of the above definitions.

EXAMPLES

```
LC1 "mdv2_  mdv1_myfile"
```

This command executes the first phase of the compiler, loading it fro mdv2__, using file mdv1__myfile__c as input, and creating fi mdv1__myfile__q.

```
LC1 "mdv1_ -omdv1_ -x mdv2_xyz"
```

This command executes the first phase of the compiler, loading it fro mdv1__, using file mdv2__xyz__c as input, and creating fi mdv2__xyz__q; it causes all external declarations without a stora class to be interpreted as extern declarations.

```
LC1 "mdv2_ =4096 mdv1_tns_err -ccuw mdv1_tns"
```

This command executes the first phase of the compiler, loading it fro mdv2__, using file mdv1__tns__c as input, and creating fi mdv1__tns__q; it sets the compiler's stack size to 4096 decimal bytes, a creates a file mdv1__tns__err to contain all the messages generated b the compiler. The compiler will allow nested comments, interpret c char declarations as unsigned char, and suppress warning messages f return statements with no return value in int functions.

1.2.2 Phase 2

The second phase of the compiler reads a quad file created by the first phase and creates an object file in the Lattice format. See section 1.5 for a more detailed discussion of the processing performed. To perform the second phase, the microdrive cartridge containing the second phase of the compiler (Cartridge 2) should be inserted into one of the drives and the cartridge containing your quad file should be inserted into the other drive. The format of the command to invoke the second phase of the compiler is:

```
LC2 "device [=stack] [%workspace] [>listfile]
      [options]" filename"
```

The various command line specifiers are shown in the order they must appear in the command. Required specifiers are shown in bold type; optional specifiers are shown in enclosed in brackets.

=stack	see LC1 command
%workspace	see LC1 command
>listfile	see LC1 command
device	Tells LC2 which device to load the second phase of the compiler from. This can be any legal QDOS device name.
options	Compile time options are specified as a hyphen followed by a single letter; in some cases, additional text may be appended. The option letter may be supplied in either upper or lower case. Each option must be specified separately, with a separate hyphen and letter. Available options are:
-fn	Specifies an address register to be used for the stack frame pointer. Only two values for "n" are allowed: 5 indicates that register A5 is to be used, 6 that register A6 is to be used. The address register used if the -f option is not specified is A6. If the -b

	option is specified on LC1, whichever of these two registers is not used as the stack frame pointer is used as the base register for addressing static and external data.
-oprefix	Specifies that the output filename (the __O file) is to be formed by prepending the input filename (the __Q file which is being compiled) with the string prefix. Any device prefix originally attached to the input filename is discarded before the new prefix is added. No intervening blanks are permitted in the string following the o.
-p	Causes the code generator to insert a special instruction known as a "stack probe" at the entry of each function for which code is generated.
-r	Forces all function calls to use PC-relative addressing, thus limiting the range of function calls to plus or minus 32K. This option must be used if position-independent code is desired.
filename	Specifies the name of the intermediate file from which code is to be generated. This intermediate file is a quad file with a __Q extension, created by the first phase of the compiler. The file name should be specified without the __Q extension; the second phase supplies the extension automatically. Alphabetic characters may be supplied in either upper or lower case. The object file is created on the same device as the quad file unless the -o option is used.

EXAMPLES

```
LC2 "mdv1_ -omdv1_ mdv2_u790"
```

This command executes the second phase of the compiler, loading it grom mdv1__, using file mdv2__u790__q as input, causing the file mdv1__u790__o to be created.

```
LC2 "mdv2_ -E5 mdv1_test4"
```

This command executes the second phase of the compiler, loading it from mdv2__, using file mdv1__test4__q as input, causing the file mdv1__test4__o to be created. Address register A5 will be used as the stack frame pointer in the code generated.

1.2.3 The QLC command (Compiler Driver)

Supplied with the compiler as part of the EPROM code is a SuperBasic routine called QLC which loads and executes both phases of the compiler; QLC prompts the user for all the information needed to run the compiler. It is invoked by typing

```
QLC
```

A window is opened and a sequence of prompts is generated. These ask for:

1. Name of the device that the compiler is to load from.
2. Name of C source file.
3. LC1 options. This may include the =stack, %workspace and >listfile facilities.

After these are answered the first phase of the compiler is executed. When this phase is completed, control returns to QLC.

At this point the user has the choice of quitting from QLC if the first phase failed, or executing the second phase of the compiler if the first phase was successful. Continuation generates a further prompt asking for the LC2 options. The second phase of the compiler is then executed and QLC terminates when LC2 has completed.

1.3 Program Linking

After all of the component source modules for a program have been compiled, they must be linked together to produce an executable load

module, or program file. This step is necessary for several reasons. First, the object file produced by the second phase of the compiler is not in a state suitable for execution. Second, most programs make use of external functions not defined in the same module; before such programs can execute, they must be "connected" with those other functions. The functions may be defined by the user, in which case they must be compiled and available as object files during linking, or they may be defined in the library supplied with the compiler. In particular, the assembly language run-time support subroutines to which the C compiler generates internal calls are supplied in the file QLC__L, which should be searched whenever C object modules are linked.

Although the usual concept of linking involves external function calls, C also permits functions to access data locations defined in other modules. This kind of reference is possible because the external linkage mechanism supported by the object code associates an external symbol with a memory location; this symbol is the identifier used to declare the object in a C program. The programmer must be careful to declare an object with the same attributes in both the module which defines it and the module which refers to it, because linking does not verify the type of reference made -- it simply connects memory references using external symbols. The use of include files for common external declarations will usually prevent this kind of error.

Linking also provides a way to attach initialization code needed to set up for execution of the C program.

1.3.1 How to link a C module

A detailed explanation of how to use the linker is given in chapter 3, for now we will concern ourselves with the procedure required to link one C module.

Firstly, invoke the linker by typing

```
EXEC mdv2_link
```

Now type CTRL-C, this will switich you into the linker. Now type a command line of the form:

```
mdv1_mycprog_o -with mdv2_prog
```

This will take the object file mdv1__mycprog__o, created by the second phase of the compiler and produce an executable file mdv1__mycprog__o__bin. It will also create a list file mdv1__mycprog__o__map. To quit the linker just enter a blank command line, and type CTRL-C to re-enter SuperBasic.

1.4 Program Execution

After linking, the resulting code can be executed by one of the SuperBasic commands EXEC, EXEC_W.

There are also two additional SuperBasic routines CRUN and CRUN_W in the ROM which will allow you to pass command line arguments into your program by typing:

```
CRUN "dev_myprog <arguments>"
```

The text within the quotes can then be accessed via ARGV and ARGC (see Kernighan & Ritchie 5.11). There are certain UNIX type lead-in characters which have special signifance within the CRUN string.

<device	Redefines STDIN to be the device specified. By default STDOUT is set-up to be the same device as STDIN. Therefore this will also redefine STDOUT.
>device	Redefines STDOUT to be the device specified. Allows STDIN and STDOUT to be independent devices.
=stack	Overrides the number of bytes reserved for the run-time stack.
%workspace	Overrides the number of bytes reserved for memory allocation. The default causes all but about 4K of the largest contiguous chunk of memory available to be allocated to program execution.

Another method of overriding the number of bytes reserved for memory allocation is by defining the external location __mneed to be the value required. For example:

```
int _mneed = 5120;
```

in one of your C modules. this statement must appear outside the body of the function to be considered external.

1.5 Compiler Processing

QL C is implemented as two separately executable programs, each performing part of the compilation task. This section discusses the structure of the compiler in general terms, and describes the processing performed by both phases. Special sections are devoted to a discussion of the topics of error processing and code generation.

1.5.1 Phase 1

The first phase of the compiler performs all pre-processor functions concurrently with lexical and syntactical analysis of the input file. It generates the symbol tables, which contain information about the various identifiers in the program, and produces an intermediate file of logical records called quadruples, which represent the elementary actions specified by the program. When the entire source program has been processed (assuming there are no fatal errors), the intermediate file (also called the quad file) is reviewed, and locally common sub-expressions are detected and replaced by equivalent results. Selected symbol table information is written to the quad file, for use by the second phase. The first phase is thus very active as far as disk I/O is concerned. Generally, if the disk activity stops for more than a few seconds, it is reasonable to assume that the compiler has failed. See Appendix B for the compiler error reporting procedure if this happens.

When the first phase begins execution, it writes a sign-on message to the standard output which identifies the version of the compiler which is being executed.

1.5.2 Phase 2

The second phase of the compiler scans the quad file produced by the first phase, and produces an object file in the Sinclair relocatable object format. This object code supports all of the necessary relocation and external linkage conventions needed for C programs (see section 6.3 for details). A logical section of code text specifying the machine language instructions which make up the executable portion of the program is generated first, followed by a section of data-defining text for all initialized static items and a logical section specifying the size of uninitialized static data. Unlike the first phase, the code generator is not always actively performing disk I/O. Each function is constructed in memory before its object code is generated, so that there may be fairly sizable pauses during which no apparent disk activity is taking place. In general, these delays should not last more than several seconds. If no activity occurs for more than about 30 seconds, the compiler has probably failed; see Appendix B for information about reporting compiler problems.

When the second phase begins execution, it writes a sign-on message to the standard output which identifies the version of the code generator which is being executed. When code generation is complete, the second phase writes a message of the form

```
Module size P=pppp D=dddd U=uuuu
```

to the standard output (usually the user's console). pppp indicates the size in bytes of the program or executable section of the module generated, dddd the size in bytes of the initialized data section, and uuuu the size in bytes of the uninitialized data section; all values are given in hexadecimal. These sizes include the requirements for all of the functions included in the original source file. Note that the sizes define the amount of memory required for the module once it is loaded (as part of a program) into memory; the _O file will require more space because it contains relocation information.

The code generator produces a single _O module for a given source module, regardless of how many functions were defined in that module. These functions (if more than one is defined) cannot be separated at link time; if any one of the functions is needed, all of them will be included.

Functions must be separated into individual source files and compiled to produce separate object modules if it is necessary to avoid this collective inclusion.

1.5.3 Error Processing

All error conditions (with the exception of internal compiler errors) are detected by the first phase. As soon as the first fatal error is encountered, the compiler stops generating quads; it then deletes the quad file just before it terminates execution. This prevents the second phase from attempting to generate code from an erroneous quad file. When the compiler detects an error in an input file, it generates an error message of the form:

```
filename line  Error nn: descriptive text
```

where filename is the name of the current input file (which may not be the original source file if #include files are used); line is the line number, in decimal, of the current line in that file; nn is an error number identifying the error; and descriptive text is a brief description of the error condition. (Appendix A provides expanded explanations for all error and warning messages produced by the compiler.) A message similar to the one above but with the text Warning instead of Error is generated for non-fatal errors; in this case, generation of the quad file continues normally. In some cases, an error message will be followed by the additional message:

```
Execution terminated
```

which indicates that the compiler was too confused by the error to be able to continue processing. The compiler uses a simple error recovery scheme which may sometimes cause a single error to induce a succession of subsequent errors in a "cascade" effect. In general, the programmer should attempt to correct the obvious errors first and not be overly concerned about error messages for apparently valid source lines (although all lines for which errors are reported should be checked).

Error messages which begin with the text CXERR are internal compiler errors which indicate a problem in the compiler itself. See Appendix B for the compiler error reporting procedure. The compiler generates a few other error messages that are not numbered; they are usually self explanatory. The most common of these is the

```
Not enough memory
```

message, which means that the compiler ran out of working memory.

Chapter 2: The Screen Editor

2.1 Introduction

The screen editor ED may be used 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. The size of the program is about 20K bytes and it requires a minimum workspace of 8K bytes.

The editor is invoked using EXEC or EXEC_W as follows

```
EXEC_W mdv1_ed
```

The difference between invoking a program with EXEC or EXEC_W is as follows. Using EXEC_W means that the editor is loaded and SuperBasic waits until the editing is complete. Anything typed while the editor is running is directed to the editor. When the editor finishes, keyboard input is directed at SuperBasic once more.

Using EXEC is slightly more complicated but is more flexible. In this case the editor is loaded into memory and is started, but SuperBasic carries on running. Anything typed at the keyboard is directed to SuperBasic unless the current window is changed. This is performed by typing CTRL-C, which switches to another window. If just one copy of ED is running then CTRL-C will switch to the editor window, and characters typed at the keyboard will be directed to the editor. A subsequent CTRL-C switches back to SuperBasic. When the editor is terminated a CTRL-C will be needed to switch back to SuperBasic once more. More than one version of the editor can be run concurrently (subject to available memory) if EXEC is used. In this case CTRL-C switches between SuperBasic and the two versions of the editor in turn.

Once the program is loaded it will ask for a filename which should conform to the standard QDOS filename syntax. No check is made on the name used, but if it is invalid a message will be issued when an attempt is made to write the file out, and a different file name may be specified then if required. All subsequent questions have defaults which are obtained by just pressing ENTER.

The next question asks for the workspace required. ED works by loading the file to be edited into memory and sufficient workspace is needed to hold all the file plus a small overhead. The default is 12K bytes which is sufficient for small files. The amount can be specified as a number or in units of 1024 bytes if the number is terminated by the character K. If you ask for more memory than is available then the question is asked again. The minimum is 8K bytes.

You are next asked,if you wish to alter the window used by ED. If you type N or just press ENTER then the default window is used. If you type Y then you are given a chance to alter the window. The current window is displayed on the screen and the cursor keys can be used to move the window around. The combination ALT and the cursor keys will alter the size of the window although there is a minimum size which may be used. Within this constraint you can specify a window anywhere on the screen, so that you can edit a file and do something else such as run a SuperBasic program concurrently. When you are satisfied with the position of the window press ENTER.

Next, an attempt is made to open the file specified, and if this succeer 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.

Immediate commands use the function keys and cursor keys on the QL in conjunction with the special keys SHIFT, CTRL and ALT. For example, delete line is requested by holding down the CTRL and ALT keys and then pressing the left arrow key. This is described in this document as CTRL-ALT-LEFT. Function keys are described as F1, F2 etc.

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.

2.2 Immediate Commands

Cursor control

The cursor is moved one position in either direction by the cursor control keys LEFT, RIGHT, UP an DOWN. 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 carried out a line at a time, while horizontal scroll is carried out 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 ALT-RIGHT combination will take the cursor to the right hand edge of the current line, while ALT-LEFT moves it to the left hand edge of the line. The text will be scrolled horizontally if required. In a similar fashion SHIFT-UP places the cursor at the start of the first line on the screen, and SHIFT-DOWN places it at the end of the last line on the screen.

The combinations SHIFT-RIGHT and SHIFT-LEFT 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 can also be used. If the cursor position is beyond the end of the current line then TAB moves the cursor to the next tab position, which is a multiple of the tab setting (initially 3). If the cursor is over some text then sufficient spaces are inserted to align the cursor with the next tab position, with any characters to the right of the cursor being shuffled to the right.

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. Although the QL keyboard generates a different code for SHIFT-SPACE and SHIFT-ENTER these are mapped to normal space and ENTER characters for convenience.

An ENTER 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-DOWN 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).

A right margin may be set up so that ENTERs 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 the right hand edge of the window used by ED. The right margin may be disabled by means of the EX command (see later).

Deleting text

The CTRL-LEFT key combination deletes the character to the left of the cursor and moves the cursor left one position. If the cursor is at the start of a line then the newline between the current line and the previous is deleted (unless you are on the very first line). The text will be scrolled if required. CTRL-RIGHT 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 combination SHIFT-CTRL-RIGHT may be used to delete a word or a number of spaces. The action of this 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-ALT-RIGHT command deletes all characters from the cursor to the end of the line. The CTRL-ALT-LEFT command 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 ALT-UP and ALT-DOWN. ALT-UP moves to previous lines, moving the text window up; ALT-DOWN moves the text window down moving to lines further on in the file. The F1 key rewrites the entire screen, which is useful if the screen is altered by another program besides the editor. Remember that you can switch out of the editor window and into some other job by typing CTRL-C at any point, assuming that there is another job with an outstanding input request. SuperBasic will be available only if you entered the editor using EXEC rather than EXEC_W. If there is enough room in memory you can run two versions of ED at the same time if you wish.

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 pressing F2. 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 F2 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 F2 key is pressed.

2.3 Extended Commands

Extended command mode is entered by pressing the F3 key. Subsequent input will appear on the command line at the bottom of the screen. Mistakes may be corrected by means of CTRL-LEFT and CTRL-RIGHT in the normal way, while LEFT and RIGHT move the cursor over the command line. The command line is terminated by pressing ENTER. After the extended command has been executed the editor reverts to immediate mode. Note that many extended commands can be given on a single command line, but the maximum length of the command line is 255 characters. An empty command line is allowed, so just typing ENTER after typing F3 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 (section 2.4).

Program control

The command X causes the editor to exit. The text held in storage is written out to file, and the editor then terminates. The editor may fail to write the file out either because the filename specified when editing started was invalid, or because the microdrive becomes full. In either case the editor remains running, and a new destination should be specified by means of the SA command described below. 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 /mdv2_savedtext/
```

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 ED to request confirmation again; 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 filename other than the current one. It is therefore possible to make copies at different stages and place them in different files.

The SA command is also useful in conjunction with the R command. Typing R followed by a filename causes the editor to be re-entered editing the new file. The old file will be lost when this happens, so confirmation is requested (as with the Q command) if any changes to the current file have been made. The normal action is therefore to save the current file with SA, and then start editing a new file with R. This saves having to load the editor into memory again, and means that once the editor is loaded the microdrive containing it can be replaced by another.

The U command "undoes" any alterations made to the current line if possible. When the cursor is moved from one line to another, the editor takes a copy of the new line before making any changes to it. The U command causes the copy to be restored. However the old copy is discarded and a new one made in a number of circumstances. These are when the cursor is moved off the current line, or when scrolling in a horizontal or vertical direction is performed, or when any extended command which alters the current line is used. Thus U will not "undo" a delete line or insert line command, because the cursor has been moved off the current line.

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. 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 margins; once this command is given no account will be taken of the right margin on the current line. Once the cursor is moved off the current line, margins are enabled once more.

Block control

undefined once more. The start of the block must be on the same line, or a line previous to, the line which marks the end of the block. A block always contains all of the line(s) within it.

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. It is not possible to insert a block within itself.

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 filename given as the argument string is read into storage immediately following the current line.

Movement

The command T moves the screen 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 screen 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.

It is common for programs such as compilers and assemblers to give line numbers to indicate where an error has been detected. For this reason the command M is provided, which is followed by a number representing the line number which is to be located. The cursor will be placed on the line number in question. Thus M1 is the same as the T command. If the line number specified is too large the cursor will be placed at the end of the file.

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 string after F and BF can be omitted; in this case the string specified in the last F, BF or E command is used. Thus

```
*F /wombat/

*BF
```

will search for 'wombat' in a forwards direction and then in a reverse direction.

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. If the response is Y or ENTER then the change takes place; any other response (except F2) will cause the command to be abandoned. This command is normally only useful in repeated groups; a response such as Q can be used to exit from an infinite repetition.

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. It is possible to add control characters into a file in this way.

The S command splits the current line at the cursor position, and acts just as though an ENTER had been typed in immediate mode. The J command joins the next line onto the end of the current one.

The D command deletes the current line in the same way as the CTRL-ALT-LEFT command in immediate mode, while the DC command deletes the character at the cursor in the same way as CTRL-RIGHT.

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.

2.4 Command List

In the extended command list, /s/ indicates a string, /s/t/ indicates two exchange strings and n indicates a number.

Immediate commands

F2	Repeat last extended command
F3	Enter extended mode
F4	Redraw screen
LEFT	Move cursor left
SHIFT-LEFT	Move cursor to previous word
ALT-LEFT	Move cursor to start of line
CTRL-LEFT	Delete left one character
CTRL-ALT-LEFT	Delete line
RIGHT	Move cursor right
SHIFT-RIGHT	Move cursor to start of next word
ALT-RIGHT	Move cursor to end of line
CTRL-RIGHT	Delete right one character
CTRL-ALT-RIGHT	Delete to end of line
SHIFT-CTRL-RIGHT	Delete word to right
UP	Move cursor up
SHIFT-UP	Cursor to top of screen
ALT-UP	Scroll up
DOWN	Move cursor down
SHIFT-DOWN	Cursor to bottom of screen
ALT-DOWN	Scroll down
CTRL-DOWN	Insert blank line

Extended Commands

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
R /s/	Re-enter editor with file s
RP	Repeat until error
S	Split line at cursor
SA /s/	Save text to file
SB	Show block on screen
SI	Show information
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
UC	Equate U/C and l/c in searches
WB /s/	Write block to file s
X	Exit, writing text back

Chapter 3: The Linker

3.1 Introduction

3.1.1 Loading the Program

The linker is found on Cartridge 3 of the QL C Development Kit. It may be loaded and run in one of two ways.

a) Interactive mode

In this mode the linker will identify and prompt the user for a command line. Upon completion of a link the linker will prompt for another command line (unless a fatal etrror has occurred). You may run the linker in interactive mode by any of the following commmands where "dev_" is the device from which the linker is to be loaded.

To run in parallel with the SuperBasic interpreter

```
EXEC_W dev_link
```

or

```
EX dev_link
```

To wait for completion of the linker

```
EXEC_W dev_link
```

or

```
EW dev_link
```

b) Non-interactive mode

In this mode the linker receives its command directly from the SuperBasic interpreter and does not interact with the user. On completion of the link the linker will exit to allow the SuperBasic interpreter to continue. You may run the linker in non-interactive mode

by one of two commands.

To run in parallel with the SuperBasic interpreter

```
EX dev_link;"<command line>"
```

To wait for the link to complete

```
EW dev_link:"<command line>"
```

The quotes round the command line are required for the SuperBasic interpreter to accept the line. Note that the EX and EW commands are only available in the QL toolkit and are not part of the standard SuperBasic.

3.1.2 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 giving the modules which reference them.
program	Specifies the name of the executable program file to be generated by the linker.

3.1.3 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 MDV1_FRED_PRG
```

will produce a program file, whereas

```
-PROG MDV1_FRED_PRG -NOPROG
```

will not.

3.1.4 Command Line Processing

The minimum command line then just consists of the name of one module file. In this case the linker will generate a program file (whose name is constructed as below from the module name) and a full listing file (whose name is also also constructed as below).

This method cannot be employed to generate a C program. The standard control file supplied with the compiler (PROG_LINK) must be used to link your programs.

3.1.5 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 an underscore then the full file name becomes the base file name, otherwise the file name with the file type (from the underscore 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 an underscore 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.

3.1.6 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 "__LINK" 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.

3.1.7 Command Line Examples

```
MDV2_MYPROG_O MDV1_C -NOLIST
```

Link the file MDV2__MYPROG__O according to the instructions in MDV1__C__LINK. The program is called MDV2__MYPROG__O__BIN.


```
-WITH MDV1_FRED
```

Take MDV1__FRED__LINK as the control file, place the program in MDV1__FRED__BIN and place the full listing output in MDV1__FRED__MAP.

```
-WITH MDV1_FRED -LIST SER1 -NOPROG
```

Take MDV1__FRED__LINK as the control file, do not generate a program file but print the listing as it is produced.

```
-WITH MDV1_FRED -PROG MDV2_FRED_BIN
```

Take MDV1__FRED__LINK as the control file, place the program in MDV2__FRED__BIN and place the listing output in MDV1__FRED__MAP.

3.1.8 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.

3.2 Linker Inputs and Outputs

The linker uses the following inputs and outputs.

3.2.1 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 section 3.1.2 above for full details of the command line.

3.2.2 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 section 3.3.

3.2.3 Relocatable Binary Files

The linker, on instruction from the command line and/or control file, will read one or more relocatable binary files (each of which may contain one or more modules).

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

3.2.4 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.

3.2.5 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. The list file is described in detail in section 3.4.

3.2.6 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 QDOS as a program.

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

3.3 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 for linking your C modules with the library and startup code called PROG_LINK. This must be used to link your C programs.

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.

3.3.1 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 PROG_LINK

```
*
*    Standard control file for linking Lattice C
*    modules
*
*    Step 0 - Space allocation
*    =========================
*
*    Allocate space for sections
*
```

```
RELOC   startup
SECTION text
SECTION data
SECTION udata
SECTION end
*
*     Step 1 - initialisation
*     =======================
*
*     C initialisation must be included first.
*
INPUT mdv2_startup
*
*     Step 2 - user modules
*     =====================
*
*     Now include a single user module (from the
*     command line)
*
INPUT *
*
*     For each extra module you want to include in
*     the link, include a line of the form:
*
*     INPUT <file name>
*
*     Step 3 - C library
*     ==================
*
*    C library - must always be included.
*
LIBRARY mdv2_qlc_l
*
```

3.3 2 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 FILE1_BIN
INPUT FILE2_BIN
```

will include all modules in the files FILE1_BIN and FILE2_BIN. Lattice C places only one module in each relocatable binary file.

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 module output from the compiler with all the required libraries, and this is the standard one PROG_LINK already shown. The template file is then used by a command line of the following form (the -WITH is optional):

```
<module file name> [-WITH] <device>prog
```

(b) `LIBRARY <file name>`

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 Lattice C 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 Lattice C.

(d) DATA <value>[K]

Some languages need to use this command to specify the amount of data space to reserve for a program for the stack and heap. This is not needed for C since it allocated space itself for the stack and heap. The value may be decimal or hexadecimal. This value is written to the header of the program file and 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).

3.3.3 Space allocation

The method of allocating space for C given in the standard control file should always be adhered to. Thus this section of commands is only here for completeness.

The previous section described the commands for determining which input modules are to included in the link. This section describes briefly

how the linker allocates space for the modules in the output file and the linker commands which may effect this allocation.

Initially the default allocation mechanism will be described and later the effects of each command on this allocation mechanism will be considered.

References in the following sections to low addresses, start addresses and absolute sections are referring to their positioning in the program file and not to their position in memory when run. This mapping may however be altered by the OFFSET command.

Generally an object module consists of absolute sections and/or at least one relocatable (or common section). The allocation of each section type is as follows:

(a) absolute sections

Absolute sections are allocated space first in the output file from their start address (relative to the start of the file and modified by any OFFSET command). The linker will issue a warning if any absolute sections overlap in the link.

(b) relocatable sections

As each input module is read in turn (as ordered by the INPUT, EXTRACT and LIBRARY commands) the linker builds up a list of relocatable sections in the order in which they are first encountered.

Once the size of each relocatable section is known then the allocation of space is made such that each relocatable section starts at the lowest possible address following the previous relocatable section while avoiding any absolute sections already allocated. The start of each relocatable section is word aligned.

The SECTION command may be used to alter the default ordering of relocatable sections.

(c) common sections

By default common sections are treated as relocatable sections except that contributions towards the same common section from different modules are overlaid rather than concatenated.

The SECTION command may be used to alter the default ordering of common sections.

The COMMON command may be used to alter the default treatment of all common sections.

Space allocation commands

The following commands alter the mechanism by which the linker allocates space for each section:

```
SECTION <section name>
COMMON <common option>
OFFSET <value>
```

(a) SECTION <section name>

This command names a section which is to be placed in a particular order in the output program file. The effect on the storage allocation is that named sections are allocated space first in the order declared with any unnamed sections allocated space following (as with the default case).

(b) COMMON <common option>

The COMMON command instructs the linker how to allocate space for common sections. In the default case common sections are treated as if they were relocatable sections for the purposes of address allocation.

The following <common option>s are available:

(i) END

This option instructs the linker to allocate space for common sections after all relocatable sections have been placed. This

means that the common sections appear at the high end of the memory allocation.

If any common sections have been named by a SECTION command then they are allocated space first followed by the other common sections in the order first encountered in the input files. The allocation of common sections is such that they avoid any absolute sections as with the normal relocatable sections.

(ii) DUMMY

This option instructs the linker to build a separate allocation for common sections. The allocation starts from address zero and ignores any allocation taken by relocatable or absolute sections.

The linker will use the dummy allocation to resolve global symbols in common sections relative to the start of the common area so that a run time system can allocate memory separate from the program for the purposes of storing common. The global variables are then used as offsets from the start of the common region.

Note that with this option no space is made in the program file for the common sections so they may not be initialised. Any attempt to place data bytes in common sections with this option in effect will cause an error.

(c) OFFSET <value>

This command does not affect the output program file unless there are absolute sections present in the link. It is only likely to be useful for linking programs which are to live at fixed addresses, and such programs will probably be blown into PROM.

See section 3.3.4 below for the definition of <value>.

The OFFSET instructs the linker to start the allocation of section starting at the <value> given instead of at address 0.

The effect on the allocation of space is as follows:

(1) absolute sections

The <value> given in the OFFSET command is subtracted from the load address of an absolute section to determine the offset within the program file at which the absolute section will be linked. Any absolute sections which start below this address are not written to the output file and a warning message is output.

(2) relocatable sections

Relocatable section allocation begins from the address given in the OFFSET command instead of at zero. This still means that relocatable sections are allocated from the start of the output program file onwards.

(3) common sections

If COMMON DUMMY is in effect then the allocation of common sections starts from address 0 regardless of the value given in the OFFSET command. For all other COMMON options the allocation is as described under the COMMON command.

3.3.4 Defining symbols at link time

Normally symbols that the linker knows about are declared and given values from within relocatable binary files. Sometimes however it is useful to be able to define symbols from the linker control file: examples of why this might be useful are:

(a) a subroutine name has been accidentally spelt differently in two different modules: as a temporary fix (until one of them is recompiled) the two symbols can be made equivalent using the DEFINE command.

(b) some subroutines have not been written yet but it is desired to test the part of the program that has been written; the missing symbols can be made equivalent to an error routine with the DEFINE command.

(c) a number contained in a library module, such as a memory requirement figure, may need to take different values in different links; these values may be assigned with the DEFINE command.

The DEFINE command is:

```
DEFINE <symbol> [=] <expression>
```

where:

```
<expression> = [ - ] <term> { <op> <term> }

<op>  =  - | +

<term> = <symbol> | <value>

<value> = <digit>{<digit>} |
          $<hexdigit>{<hexdigit>}

<digit> = 0|1|...|8|9

<hexdigit> = <digit>|A|...|F|a|...|f
```

A symbol used on the right-hand side of the DEFINE command may be a defined in a relocatable binary file or in a previous DEFINE command. A forward reference to a symbol to be defined by a future DEFINE command is illegal and will produce an error message. The symbol defined by a DEFINE command may not also be used on the right-hand side of the same DEFINE command.

If a symbol used in an expression remains undefined after all modules have been read in a warning is issued by the linker. The value of the symbol is then undefined.

Examples:

```
DEFINE SCREEN = $28000

DEFINE MAXPAR = 10
```

```
DEFINE USERSPACE = 1000

DEFINE TOTALSPACE = LOCAL+GLOBAL+USERSPACE
```

(where LOCAL and GLOBAL are declared in relocatable object modules)

3.3.5 Summary of the control file

This section is a quick summary of the commands possible in the linker control file.

Lines beginning with '*', ';' or '!' are comments and are ignored by the linker.

All letters in the control file input can be in either case and case is not significant.

(a) `INPUT <file name>`

Include all modules from the named file in the link.

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

Include the named module from the named file in the link.

(c) `LIBRARY <file name>`

Instructs the linker to search the library from start to finish. Any modules in the library which satisfy any currently unresolved references are included in the link.

(d) `SECTION <section name>`

Declares a section to the linker. All declared sections are allocated space before any undeclared sections.

(e) `COMMON <common option>`

Instructs the linker how to handle COMMON sections (if any are encountered).

(f) `OFFSET <value>`

Instructs the linker to start address allocation and to write the output file from the address given in the value parameter.

(g) `DEFINE <symbol> [=] <expression>`

Defines a symbol at link time. If the expression includes a symbol which has not already been defined then the linker expects to find it in a relocatable object module.

3.4 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 followir g 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.

3.5 Actions of the Linker

This section 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.

3.5.1 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.

3.5.2 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.

3.5.3 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.

3.5.4 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.

3.5.5 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.

3.5.6 Post processing

After pass 2 the symbol table is written to the listing file and if required a symbol table is appended to the output file. Upon completion of the symbol table the linker issues a summary message stating the number of errors and warnings and the number of undefined symbols. The linker then issues a final message indicating the completion status of the link. In interactive mode the linker will then

reprompt for another command line. Entering a blank line at this stage terminates the linker. In non-interactive mode the linker will exit.

Chapter 4: Language Definition

The Lattice portable C compiler accepts a program written in the C programming language, determines the elementary actions specified by that program, and eventually translates those actions into machine language instructions. Although the final result of these processes is highly machine-dependent, the actual language accepted by the compiler is, for the most part, independent of any system or implementation details. This section presents the language defined by the Lattice C compiler using the Kernighan and Ritchie (K&R) text "The C Programming Language" as a reference point. Since this language conforms closely to that described in the text, only the major differences are first presented. The major features of the language are then discussed, not in any attempt at completeness, but simply for the sake of showing them from a different perspective. Finally, a comparison with the Kernighan and Ritchie "C Reference Manual" is made to show more precisely how the Lattice implementation differs from the standard.

4.1 Summary of Differences

There are two classes of differences that appear in a discussion of an implementation of a programming language. The first class is that of actual semantic differences, that is, variations which cause the meaning of language constructs to differ. The second class is merely a reflection of the practical limitations to which all programs -- including compilers -- are subject. Each of the following subsections presents the respective details for the Lattice implementation of C.

4.1.1 Differences from the Standard

Deviating from a standard has its own peculiar set of perils and rewards. On the one hand, the differences create problems for those who have conformed to the standard in the past; on the other, they may make life easier for those who take advantage of them in the future. Most of the differences listed below were prompted by a desire to make the language both more portable and more comprehensible. The vast majority of programs will not encounter these potential troublespots; those that do

will in most cases be improved by adjusting to conform to them. Here, then, is a summary of the major differences:

- Pre-processor macro substitutions using arguments must be specified on a single line; for example, when max(a,b) is used, the invocation text from max to the final closing parenthesis must be defined within a single input line.

- In processing structure and union member declarations, the compiler builds a separate list of member names for each structure (or union). Thus, identical names may be used for members in different structures, even though both the offset and the attributes may be different in each declaration. The specific structure being referenced determines which member name (and therefore which offset and set of attributes) is meant. The typing rules for structure member references are strictly enforced so that the particular list of valid member names can be determined. In other words, the expression in front of the . or -> operators must be identifiable by the compiler as a structure or pointer to a structure of a definite type.

- Implicit pointer conversion (by assignment) is legal but generates a warning message; this occurs whenever any value other than a pointer of the same type or the constant zero is assigned to a pointer. A cast operator can be used to eliminate the warning. A more stringent requirement is enforced for initializers, where the expression to initialize a pointer must evaluate to a pointer of the same type or to the constant zero; any other value is an error.

- If a structure or union appears as a function argument without being preceded by the address-of operator &, the compiler generates a warning message and assumes that the address of the aggregate was intended.

- An array name may be preceded by the address-of operator &; the meaning, however, is not that of a pointer to the first element but of a pointer to the array. This construct allows initialization of pointers to arrays.

- The constant expression following an #if conditional statement may not contain the sizeof operator and must be completed in less than a single line.

A more systematic and detailed explanation of the above differences is presented in section 4.3, but some of the most important items above deserve immediate clarification.

The intent behind making the structure and union member names a separate class of identifiers for each structure is twofold. First, the flexibility of member names is greatly increased, since now the programmer need not worry about a possible conflict of names between different structures. Second, the requirement that the compiler be able to determine the type of the structure being referenced generally improves the clarity of the code, and disallows such questionable constructs as

```
int *p;
. . .
p->xyz = 4;
```

which is considered an error by this compiler. Those who grumble about this restriction should note that one can accomplish the equivalent sequence in Lattice C by using a cast:

```
((struct ABC *)p)->xyz = 4;
```

The parentheses are required since the -> operator binds more tightly than the cast. The idea is not that such code should be prohibited unconditionally but that any such constructs should be clearly visible for what they are; the cast operator serves this purpose nicely.

Exactly the same intent is present in the pointer conversion warning. By using a cast operator, the programmer can eliminate the warning; the conversion is then explicitly intentional, and not simply the result of sloppy coding. In addition, there is a more important reason for the warning. Although many C programs make the implicit assumption that pointers of all types may be stored in int variables (or other pointer types) and retrieved without difficulty, the language itself makes no guarantee of this. On word-addressed machines, in fact, such conversions will not always work properly; the warning message provides a gentle (and non-fatal) reminder of this fact.

Finally, the warning generated when a structure or union is used as a function argument without the address-of operator is intended to remind programmers that this compiler does not allow an aggregate to be passed to a function -- only pointers to such objects.

4.1.2 Arbitrary Limitations

Although the definition of a programming language is an idealized abstraction, any real implementation is constrained by a number of factors, not the least of which is practicality. The Lattice compiler imposes the following arbitrary restrictions on the language it accepts:

- The maximum value of the constant expression defining the size of a single subscript of an array is one less than the largest positive int.
- The maximum length of an input source line is 256 bytes.
- The maximum size of a string constant is 256 bytes.
- Macros with arguments are limited to a maximum number of 8 arguments.
- The maximum length of the substitution text for a #define macro is 256 bytes
- The maximum level of #include file nesting is 3.

These limitations are imposed because of the way objects are represented internally by the compiler; our hope is that they are reasonably large enough for most real programs.

4.2 Major Language Features

The material presented in this section is meant to clarify some of the language features which are not always fully defined in the Kernighan and Ritchie text. These are features which depend on implementation decisions made in the design of the compiler itself, or on interpretations of the language definition. Those language features which are specifically machine dependent are described elsewhere in this manual.

4.2.1 Pre-Processor Features

The Lattice C compiler supports the full set of pre-processor commands described in Kernighan and Ritchie. Most implementations perform the pre-processor commands concurrently with lexical and syntactic analysis of the source file, because an additional compilation step can be avoided by this technique. Other versions of the compiler incorporate a separate pre-processor phase in order to reduce the size of the first phases of the compiler. In either case, the analysis of the pre-processor commands is largely independent of the compiler's C language analysis. Thus, #define text substitutions a symbols which may or may not be defined. Otherwise, #if expressions support the full range of operations described in section 15 of Appendix A of Kernighan and Ritchie.

The #define command, as noted in section 4.1.1, has the limitation that the macro invocation text must all be contained on a single input line. Because the compiler uses a text buffer of fixed size, a particularly complex macro may occasionally cause a line buffer overflow condition; usually, however, this error occurs when more than one macro reference occurs in the same source line, and can be circumvented by placing the macros on different lines. Circular definitions such as

```
#define A B
#define B A
```

will be detected by the compiler if either A or B is ever used, as will more subtle loops. Like many other implementations of C, the Lattice compiler supports nested macro definitions, so that if the line

```
#define XYZ 12
```

is followed later by

```
#define XYZ 43
```

the new definition takes effect, but the old one is not forgotten. In other words, after encountering

```
#undef XYZ
```


the former definition (12) is restored. To completely undefine XYZ, an additional #undef is required. The rule is that each #define must be matched by a corresponding #undef before the symbol is truly "forgotten".

4.2.2 Arithmetic Objects

Six types of arithmetic objects are supported by the Lattice compiler; along with pointers, these objects represent the entities which can be manipulated in a C program. The types are:

```
char
short
int
long
float
double
```

Note that unsigned is used as a modifier.

The natural size of integers, as indicated by a plain int type specifier, on the Sinclair QL is 32-bits; this type is identical to long.

The compiler follows the standard pattern for conversions between the various arithmetic types, the so-called "usual arithmetic conversions" described in the Kernighan and Ritchie text. The only exception to this occurs in connection with byte-oriented machines, where expansion of char to int may be avoided if both operands in an expression are char, and the target machine supports byte-mode arithmetic and logical operations.

4.2.3 Derived Objects

The Lattice C compiler supports the standard extensions leading to various kinds of derived objects, including pointers, functions, arrays, and structures and unions. Declarations of these types may be arbitrarily complex, although not all declarations result in a legal object. For example, arrays of functions or functions returning aggregates are illegal. The compiler checks for these kinds of declarations and also verifies that structures or unions do not contain instances of themselves. Objects which are declared as arrays cannot have an array length of zero, unless they are formal parameters or are

declared extern (see section 4.2.4). All pointers are assumed to be the same size -- usually, that of a plain int -- with one exception. On word-addressed machines, pointers which point to objects which can appear on any byte boundary are assumed to require twice as much storage as pointers to objects which must be word-aligned.

Note that the size of aggregates (arrays and structures) may be affected by alignment requirements. For example, the array

```
struct {
    short i;
    char c;
    } x[10];
```

will occupy 40 bytes on machines which require short objects to be aligned on an even byte address.

4.2.4 Storage Classes

Declared objects are assigned by the compiler to storage offsets which are relative to one of several different storage bases. The assigned storage base depends on the explicit storage class specified in the declaration, or on the context of the declaration, as follows:

External — An object is classified as external if the extern keyword is present in its declaration, and the object is not later defined in the source file (that is, it is not declared outside the body of any function without the extern keyword). Storage is not allocated for external items because they are assumed to exist in some other file, and must be included during the linking process that builds a set of object modules into a load module.

Static — An object is classified as static if the static keyword is present in its declaration or if it is declared outside the body of any function without an explicit storage class specifier. Storage is allocated for static items in the data section of the object module; all such locations are initialized to zero unless an initializer expression is included in the declaration (see section 4.2.6). Static items declared outside the body of any function without the static keyword are visible in other files, that

is, they are externally defined. Note that string constants are allocated as static items, and are treated as unnamed static arrays of char.

Auto — An object is classified as auto if the auto keyword is present in its declaration, or if it is declared inside the body of any function without an explicit storage class specifier (it is illegal to declare an object auto outside the body of a function). Storage is presumably allocated for auto items using a stack mechanism during execution of the function in which they are defined.

Formal — An object is classified as formal if it is a formal parameter to one of the functions in the source file. Storage is presumably allocated for formal items when a function call is made during execution of the program.

Note that the first phase of the compiler makes no assumption about the validity of the register storage class declarator. Items which are declared register are so flagged, but storage is allocated for them anyway against either the auto or the formal storage base.

Note also that if the x compile-time option is used, the implicit storage class for items declared outside the body of any function changes from static to extern. This allows a single header file to be used for all external data definitions. When the main function is compiled, the x option is not used, and so the various objects are defined and made externally visible; when the other functions are compiled the x option causes the same declarations to be interpreted as references to objects defined elsewhere.

4.2.5 Scope of Identifiers

The Lattice compiler conforms almost exactly to the scope rules discussed in Appendix A of the Kernighan and Ritchie text (pp. 205-206). The only exception arises in connection with structure and union member names, where (as noted in section 4.1) the compiler keeps separate lists of member names for each structure or union; this means that additional classes of non-conflicting identifiers occur for the various structures and unions. Two additional points are worth clarifying.

First, when identifiers are declared at the beginning of a statement block internal to a function (other than the first block immediately following the function name), storage for any auto items declared is allocated against the current base of auto storage. When the statement block terminates, the next available auto storage offset is reset to its value preceding those declarations. Thus, that storage space may be reused by later local declarations. Rather than generate explicit allocate and deallocate operations, the compiler uses this mechanism to compute the total auto storage required by the function; the resulting storage is allocated whenever the function is called. With this scheme, functions will allocate possibly more storage than will be needed (in the event that those inner statement blocks are not executed), but the need for run-time dynamic allocation within the function is avoided.

Second, when an identifier with a previous declaration is redefined locally in a statement block with the extern storage class specifier, the previous definition is superseded in the normal fashion but the compiler also verifies compatibility with any preceding extern definitions of the same name. This is done in accordance with the principle expressed in the text, namely that all functions in a given program which refer to the same external identifier refer to the same object. Within a source file, the compiler also verifies that all external declarations agree in type. The point is that in this particular case -- where a local block redefines an identifier as extern -- the declaration effectively does not disappear upon termination of the block, since the compiler now has an additional external item for which it must verify equivalent declarations.

4.2.6 Initializers

Objects which are of the static storage class (as defined in section 4.2.4) are guaranteed to contain binary zeros when the program begins execution, unless an initializer expression is used to define a different initial value. The Lattice compiler supports the full range of initializer expressions described in Kernighan and Ritchie, but restricts the initialization of pointers somewhat. An arithmetic object may be initialized with an expression that evaluates to an arithmetic constant which, if not of the appropriate type, is converted to that of the target object.

The expression used to initialize a pointer is more restricted: it must evaluate to the int constant zero or to a pointer expression yielding a pointer of exactly the same type as the pointer being initialized. This pointer expression can include the address of a previously declared static or extern object, plus or minus an int constant, but it cannot incorporate a cast (type conversion) operator, because pointer conversions are not evaluated at compile time (exception: a cast operator can be used on an int constant but not on a variable name). This restriction makes it impossible to initialize a pointer to an array unless the & operator is allowed to be used on an array name, because the array name without the preceding & is automatically converted to a pointer to the first element of the array. Accordingly, as noted in section 4.1, the Lattice compiler accepts the & operator on an array name so that declarations such as

```
int a[5], (*pa)[5] = &a;
```

can be made. Note that if a pointer to a structure (or union) is being initialized, the structure name used to generate an address must be preceded by the & operator.

More complex objects (arrays and structures) may be initialized by bracketed, comma-separated lists of initializer expressions, with each expression corresponding to an arithmetic or pointer element of the aggregate. A closing brace can be used to terminate the list early (see Appendix A of Kernighan and Ritchie for examples. Unions may not be initialized under this implementation, although the first part of a structure containing a union may be initialized if the expression list ends before reaching the union. A character array may be initialized with a string constant which need not be enclosed in braces; this is the only exception to the rule requiring braces around the list of initializers for an aggregate.

Initializer expressions for auto objects can only be applied to simple arithmetic or pointer types (not to aggregates), and are entirely equivalent to assignment statements.

4.2.7 Expression Evaluation

All of the standard operators are supported by the Lattice compiler, in the standard order of precedence (see p. 49 of Kernighan and Ritchie). Expressions are evaluated using an operator precedence

parsing technique which reduces complex expressions to a sequence of unary and binary operations involving at most two operands. Operations involving only constant operands (including floating point constants) are evaluated by the compiler immediately, but no special effort is made to re-order operands in order to group constants. Thus, expressions such as

```
c = 'A' + 'a'
```

must be parenthesized so that the compiler can evaluate the constant part:

```
c + ('a' - 'A')
```

If at least one operand in an operation is not constant, the intermediate expression result is represented by a temporary storage location, known as a temporary. The temporary is then "plugged into" the larger expression and becomes an operand of another binary or unary operation; the process continues until the entire expression has been evaluated. The lifetimes of temporaries and their assignment to storage locations are determined by a subroutine internal to the first phase of the compiler, which recognizes identically generated temporaries within a straight-line block of code and eliminates recomputation of equivalent results. Thus, common sub-expressions are recognized and evaluated only once. For example, in the statement

```
a[i+1] = b[i+1];
```

the expression i+1 will be evaluated once and used for both subscripting operations. Expressions which produce a result that is never used and which have no side effects, such as

```
i+j;
```

are discarded by this same subroutine.

Within the block of code examined by the temporary analysis subroutine, operations which produce a temporary result are noted and remembered so that later equivalent operations may be deleted, as noted above. Two conditions (other than function calls, which may have undetermined side effects) cause the subroutine to discard an

operation and no longer check for the equivalent operation later: (1) if either of its operands appears directly as a result of a subsequent operation; or (2) if a subsequent operation defines an indirect (i.e., through a pointer) result for the same type of object as one of the original operands. The latter condition is based on the compiler's assumption that pointers are always used to refer to the correct type of target object, so that, for example, if an assignment is made using an int pointer only objects of type int can be changed. Only when the programmer indulges in type punning -- using a pointer to inspect an object as if it were a different type -- is this assumption invalid, and it is hard to conceive of a case where the common sub-expression detection will cause a problem with this somewhat dubious practice. Such inspections are generally better left to assembly language modules in any case.

With the exception of this common sub-expression detection, which may replace an operation with a previous, equivalent one, expressions are evaluated in strict left-to-right order as they are encountered, except, of course, where that is prevented by operator precedence or parentheses. It is best not to make any assumptions, however, about the order of evaluation, since the code generation phase is generally free to re-order the sequence of many operations. The most important exceptions are the logical OR (||) and logical AND (&&) operators, for which the language definition guarantees left-to-right evaluation. The code generation phase may have other effects on expression evaluation; usually, some favorable assumptions about pointer assignments are made, though these can be shut off by a compile-time option. Check the implementation section of this manual for full details.

4.2.8 Control Flow

C offers a rich set of statement flow constructs, and the Lattice compiler supports the full complement of them. Some minor points of clarification are noted here. First of all, the compiler does verify that switch statements contain (1) at least one case entry; (2) no duplicate case values; and (3) not more than one default entry. In addition, the first phase of the compiler recognizes certain statement flow constructs involving constant test values, and may discard certain portions of code accordingly. (Even those portions ultimately discarded are fully

analyzed, lexically and syntactically, before being eliminated.) If an if statement has a constant test value, only the code for the appropriate clause (the then or else portion) is retained; while and for statements with zero test values are entirely discarded.

The code generation phase generally makes a special effort to generate efficient sequences for control flow. In particular, the size and number of branch instructions is kept to a minimum by extensive analysis of the flow within a function, and switch statements are analyzed to determine the most efficient of several possible machine language constructs. Check the implementation section of this manual for the details regarding this particular code generator.

4.3 Comparison to the Kernighan & Ritchie "C Reference Manual"

The most precise definition of the C programming language generally available is in Appendix A of the Kernighan and Ritchie text, which is entitled C Reference Manual. This section presents, in the same order defined in the text, a series of amendments or annotations to that manual; this commentary explicitly states any deviations of the Lattice C language implementation from the features described. Because this implementation is very close to the Kernighan and Ritchie standard, many of the sections apply exactly as written; these sections will not be commented upon. Any section not listed here can be assumed to be fully valid for the language accepted by the Lattice C compiler.

CRM 2.1 Comments

The Lattice compiler allows comments to be nested, that is, each /* encountered must be matched by a corresponding */ before the comment terminates by using the C compile-time option. This feature makes it easy to "comment out" large sections of code which themselves contain comments. The default is to process comments in the standard, non-nesting mode.

CRM 2.4.3 Character constants

Two extensions to character constants are provided. First, more than one character may be enclosed in single quotes; the result may be short

or long, depending on the number of characters. Second, if the first character following the backslash in an escape sequence is x, the next one or two digits are interpreted as a hexadecimal value. Thus,

```
'\xE9'
```

generates a character with the value 0xE9 (see -c option for LC1).

CRM 2.5 Strings

The Lattice compiler can be made to recognize identically written string constants and only generates one copy of the string. (Note that strings used to initialize char arrays -- not char * -- are not actually generated, because they are really just shorthand for a comma-separated list of single-character constants.) The same \x convention described above can be employed in strings, where it is generally more useful (see -c option for LC1).

CRM 7.1 Primary expressions

The Lattice compiler always enforces the rules for the use of structures and unions for the simple reason that it cannot otherwise determine which list of member names is intended. Recall from section 2.1 that the compiler maintains a separate list of members for each type of structure or union. Therefore, the primary expression preceding the . or -> operator must be immediately recognizable as a structure or pointer to a structure of a specific type.

CRM 7.2 Unary operators

The requirement that the & operator can only be applied to an lvalue is relaxed slightly to allow application to an array name (which is not considered an lvalue). Note that the meaning of such a construct is a pointer to the array itself, which is quite different from a pointer to the first element of the array. The difference between a pointer to an array and to an array's first element is only important when the pointer is used in an expression with an int offset, because the offset must be scaled (multiplied) by the size of the object to which the pointer points. In this case the target object size is the size of the whole array, rather than the size of a single element, if the pointer points to the array as a whole.

CRM 7.6 Relational operators

When pointers of different types are compared, the right-hand operand is converted to the type of the left-hand operand; comparison of a pointer and one of the integral types causes a conversion of the integer to the pointer type. Both of these are operations of questionable value and are certainly machine-dependent.

CRM 7.7 Equality operators

The same conversions noted above are applied.

CRM 8.1 Storage class-specifiers

The text states that the storage class-specifier, if omitted from a declaration outside a function, is taken to be extern. This is somewhat misleading, if not plainly inaccurate; in fact (as the text points out in CRM 11.2), the presence or absence of extern is critical to determining whether an object is being defined or referenced. As noted in section 4.2.4 of this manual, if extern is present, then the declared object either exists in some other file or is defined later in the same file; if no storage class specifier is present, then the declared object is being defined and will be visible in other files. If the static specifier is present, the object is also defined but is not made externally visible. The only exception to these rules occurs for functions, where it is the presence of a defining statement body that determines whether the function is being defined.

The Lattice compiler can be forced to assume extern for all declarations outside a function by means of the x compile time option. Declarations which explicitly specify static or extern are not affected.

CRM 8.5 Structure and union declarations

The Lattice compiler treats the names of structure members quite differently from Kernighan and Ritchie. The names of members and tags do not conflict with each other or with the identifiers used for ordinary variables. Both structure and union tags are in the same class of names, so that the same tag cannot be used for both a structure and a union. A separate list of members is maintained for each structure; thus, a member name may not appear twice in a particular structure, but the same name may be used in several different structures within the same scope.

CRM 8.7 Type names

Although a structure or union may appear in a type name specifier, it must refer to an already known tag, that is, structure definitions cannot be made inside a type name. Thus, the sequence

```
(struct { int high, low; } *) x
```

is not permitted, but

```
struct HL { int high, low; };
. . .
(struct HL *) x
```

is acceptable.

CRM 10.2 External data definitions

The Lattice compiler applies a simple rule to external data declarations: if the keyword extern is present, the actual storage will be allocated elsewhere, and the declaration is simply a reference to it. Otherwise, it is interpreted as an actual definition which allocates storage (unless the x option has been used; see the comments on CRM 8.1).

CRM 12.3 Conditional compilation

As noted in section 4.2.1 of this manual, the constant expression following #if may not contain the sizeof operator, and must appear on a single input line.

CRM 12.4 Line control

Although the filename for #line is denoted as identifier, it need not conform to the characteristics of C identifiers. The compiler takes whatever string of characters is supplied; the only lexical requirement for the filename is that it cannot contain any white space.

CRM 14.1 Structures and unions

The escape from typing rules described in the text is explicitly not allowed by the Lattice compiler. In a reference to a structure or union

member, the name on the right must be a member of the aggregate named or pointed to by the expression of the left. This implementation, however, does not attempt to enforce any restrictions on reference to union members, such as requiring a value to be assigned to a particular member before allowing it to be examined via that member.

Chapter 5: Portable Library Functions

In order to provide real portability, a C programming environment must provide -- in a machine-independent way -- not only a well-defined language but a library of useful functions as well. The portable library provided with the Lattice C compiler attempts to fulfill this requirement. It provides the basic functions of memory allocation, file input/output, and character string manipulation; otherwise, the compiler itself could not be implemented. An important side benefit of presenting the functions from a machine-independent viewpoint is that it helps the programmer think of them as such.

When referring to the function descriptions presented in this section, remember that the compiler assumes that a function will return an int value unless it is explicitly declared otherwise. Any function which returns any other kind of value must be declared as that kind of function in advance of its first usage in the same file.

5.1 Memory Allocation Functions

The standard library provides memory allocation capabilities at several different levels. The higher level functions call the lower levels to perform the work, but provide easier interfaces in exchange for the extra overhead. The amount of memory available for dynamic allocation on the Sinclair QL depends on the size of the program. The default is for the program to grab all but 4K of the memory not used for code, data or bss. This memory is used both for dynamic allocation and for the run-time stack (used for function calls and auto variables). It is necessary to leave 4K for QDOS to have room to work in.

A default of 2048 decimal bytes is reserved for the stack, and the remainder of the memory is used by the memory allocation functions. In order to allow program to adjust the amount of memory reserved for the stack, a run-time option = stack is provided to override the default stack

size (see section 1.4); alternatively a program may define the size internally by defining the extend location __stack to be the required value for example:

```
int _stack = 4096
```

Similarly the total amount of memory grabbed by the program maybe overridden either by using the run-time option %workspace (see section 1.4) or by defining the external location __mneed to be the required value, for example:

```
int _mneed = 5120
```

Warning: There is no check against the stack overrunning its allotted size and destroying portions of the memory pool or static data.

All of the memory allocation functions return a pointer which is of type char *, but is guaranteed to be properly aligned to store any object.

5.1.1 Level 3 Memory Allocation

The functions described in this section provide a UNIX-compatible memory-allocation facility. The blocks of memory obtained may be released in any order, but it is an error to release something not obtained by calling one of these functions. Because these functions use overhead locations to keep track of allocation sizes, the free function does not require a size argument. The overhead does, however, decrease the efficiency with which these functions use the available memory. If many small allocations are requested, the available memory will be more efficently utilized if the level 2 functions are used instead.

MALLOC

Purpose:

UNIX-compatible memory allocation

Synopsis:

```
p = malloc(nbytes);
char *p;              block pointer
unsigned nbytes;      number of bytes requested
```

Description:

Allocates a block of memory in a way that is compatible with UNIX. The primary difference between malloc and getmem is that the former allocates a structure at the front of each block. This can result in very inefficient use of memory when making many small allocation requests.

Returns:

p = NULL if not enough space available
= pointer to block of nbytes of memory otherwise

Cautions:

Return value must be checked for NULL. The function should be declared char * and a cast operator used if defining a pointer to some other kind of object, as in:

```
char *malloc();
int *pi;
. . .
pi = (int *)malloc(N);
```

CALLOC

Purpose:

Allocate memory and clear

Synopsis:

```
p = calloc(nelt, eltsiz);
char *p;            block pointer
unsigned nelt;      number of elements
unsigned eltsiz;    element size in bytes
```

Description:

Allocates and clears (sets to all zeros) a block of memory. The size of the block is specified by the product of the two parameters; this calling technique is obviously convenient for allocating arrays. Typically, the second argument is a sizeof expression.

Returns:

p = NULL if not enough space available
= pointer to block of memory otherwise

Cautions:

Return value must be checked for NULL. The function should be declared char * and a cast used if defining a pointer to some other kind of object, as in:

```
char *calloc();
struct buffer *pb;
. . .
pb = (struct buffer *)calloc(4, sizeof(struct
      buffer));
```

FREE

Purpose:

UNIX-compatible memory release function

Synopsis:

```
ret = free(cp);
int ret;            return code
char *cp;           block pointer
```

Description:

Releases a block of memory that was previously allocated by malloc or calloc. The pointer should be char * and is checked for validity, that is, verified to be an element of the memory pool.

Returns:

ret = 0 if successful
= -1 if invalid block pointer

Cautions:

Remember to cast the pointer back to char *, as in:

```
char *malloc();
int *pi;
. . .
pi = (int *) malloc(N);
. . .
if (free((char *)pi) != 0) { ... error ... }
```


5.1.2 Level 2 Memory Allocation

The functions described in this section provide an efficient and convenient memory allocation capability. Like the level 3 functions, allocation and de-allocation requests may be made in any order, and it is an error to free memory not obtained by means of one of these functions. The caller must retain both the pointer and the size of the block for use when it is freed; failure to provide the correct length may lead to wasted memory (the functions can detect an incorrect length when it is too large, but not when it is too small). An additional convenience is provided by the sizmem function, which can be used to determine the total amount of memory available.

The level 2 functions maintain a linked list of the blocks of memory released by calls to rlsmem, called the free space list. Initially, this list is null, and getmem acquires memory by calling the level 1 memory allocator sbrk. As blocks are released by the program, the free space list is created; when a block adjacent to one already on the list is freed, it is combined with any adjacent blocks. Thus, the size of the largest block available may be smaller than the total amount of free memory, due to breakage.

GETMEM/GETML

Purpose:

Get a memory block

Synopsis:

```
p = getmem(nbytes);
p = getml(lnbytes);
char *p;            block pointer
unsigned nbytes;    number of bytes requested
long lnbytes;       long number of bytes requested
```

Description:

These functions get a block of memory from the free memory pool. If the pool is empty or a block of the requested size is not available, more memory is obtained via the level 1 function sbrk. On the Sinclair QL getml is the same as getmem.

Returns:

p = NULL if not enough space available
= pointer to memory block otherwise

Cautions:

Return value must be checked for NULL. The function should be declared char * and a cast used if defining a pointer to some other kind of object, as in:

```
char *getmem();
struct XYZ *px;
. . .
px = (struct XYZ *)getmem(sizeof(struct XYZ));
```

RLSMEM/RLSML

Purpose:

Release a memory block

Synopsis:

```
ret = rlsmem(cp, nbytes);
ret = rlsml(cp, lnbytes);
int ret;            return code
char *cp;           block pointer to be freed
unsigned nbytes;    size of block
long lnbytes;       size of block as long integer
```

Description:

These functions release the memory block by placing it on a free block list. If the new block is adjacent to a block on the list, they are combined. On the Sinclair QL rlsml is the same as rlsmem.

Returns:

ret = 0 if successful
= -1 if supplied block is not obtained by getmem or getml or if it overlaps one of the blocks on the list

Cautions:

Return value should be checked for error. If the correct size is not supplied, the block may not be freed properly.

ALLMEM/BLDMEM

Purpose:

Allocate level 2 memory pool

Synopsis:

```
ret = allmem();
ret = bldmem(n);
int ret;            return code
int n;              maximum number of 1 kilobyte blocks
```

Description:

The bldmem function uses the level 1 function sbrk to allocate up to n 1 kilobyte blocks of memory. If n is 0, then all available memory is allocated.

The allmem function merely calls bldmem with n set to 0.

Subsequent getmem and getml calls will make allocations from this memory pool. All of the memory allocated by getmem calls following a call to allmem or bldmem can be freed by a call to the rstmem function described below.

The size of the memory pool allocated by either one of these functions can be obtained by a call to the sizmem function described below.

Returns:

ret = -1 if first sbrk fails
= 0 if successful

Cautions:

Should be called only once during the lifetime of the program.

SIZMEM

Purpose:

Get memory pool size

Synopsis:

```
bytes = sizmem();
long bytes;                number of bytes
```

Description:

Returns the number of unallocated bytes in the memory pool used by getmem and getml. Note that getmem and getml dynamically expand the pool by calling sbrk whenever a request cannot be honored. Therefore, the value returned by sizmem does not necessarily indicate how much memory is actually available. If used after calling allmem, however, the actual memory pool size will be returned.

Returns:

bytes = (long) number of bytes in memory pool

Cautions:

Note that this function returns a long integer, and must be declared long before it is used.

RSTMEM

Purpose:

Reset memory pool

Synopsis:

rstmem();

Description:

Resets the level 2 memory pool to its initial state. All memory allocated by calls to getmem and getml made after allmem was called is released by rstmem; memory allocated before allmem was called is not affected. This function makes it possible to make a certain number of initial sbrk, getmem, or getml calls, and then to initialize a memory pool by calling allmem. Any allocations made after the call to allmem are freed by rstmem, but the preceding sbrk or getmem calls are not affected.

Cautions.

This function cannot be used if any files have been opened after the immediately preceding allmem call for access using any of the level 2 I/O functions, because these functions use getmem to allocate buffers. Files should be opened before the allmem call to avoid this problem.

5.1.3 Level 1 Memory Allocation

The two functions defined at the lowest level of memory allocation are primitives which perform the basic operations needed to implement a more sophisticated facility; they are used by the level 2 functions for that purpose. sbrk treats the total amount of memory available as a single block, from which portions of a specific size may be allocated at the low end, creating a new block of smaller size. rbrk merely resets the block back to its original size. The "break point" mentioned here should not be confused with the breakpoint concept used in debugging; this term simply refers to the address of the low end of the block of memory manipulated by sbrk.

SBRK/LSBRK

Purpose:

Set memory break point

Synopsis:

```
p = sbrk(nbytes);
p = lsbrk(lnbytes);
char *p;              points to low allocated address
unsigned nbytes;      number of bytes to be allocated
long lnbytes;         long, number of bytes to be allocated
```

Description:

These functions allocate a block of memory of the requested size, if possible; they form the basic UNIX memory allocator. Memory is allocated by advancing the "memory break point," which is simply the base address of a block of memory whose location is system-dependent. The previous break point address is then returned to the caller. On the Sinclair QL lsork is the same as sbrk.

Returns:

```
p   = -1 if request cannot be fulfilled (sbrk only)
p   = 0 if request cannot be fulfilled (lsbrk only)
    = pointer to low address of block if successful
```


Cautions:

For consistency with the UNIX function, sbrk returns -1 if it cannot satisfy the request, although the rest of the memory allocators return NULL. Both functions should be declared char * and a cast used if defining a pointer to some other kind of object.

RBRK

Purpose:

Reset memory break point

Synopsis:

rbrk();

Description:

Resets the memory break point to its original starting-point. This effectively frees all memory allocated by any of the memory allocation functions.

Cautions:

Like rstmem above, this function cannot be used if any files are open and being accessed using the level 2 I/O functions.

5.2 I/O and System Functions

The standard library provides I/O functions at several different levels, with single character get and put functions and formatted I/O at the highest levels, and direct byte stream I/O functions at the lowest levels.

Three general classes of I/O functions are provided. First, the level 2 functions define a buffered text file interface which implements the single character I/O functions as macros rather than function calls. Second, the level 1 functions define a byte stream-oriented file interface, primarily useful for manipulation of disk files, though most of the same functions are applicable to devices (such as the user's console) as well. Finally, since one of the most common I/O interfaces is with the user's console, a special set of functions allows single character I/O directly to the user's terminal, as well as formatted and string I/O.

In general, these functions adhere to the UNIX convention for reporting errors. When a failure indication from an I/O function is obtained, programmers can inspect the global integer errno, which will contain one of the error codes defined in the header file error__h. Additional information may be available from the global integer __oserr, which contains the QDOS error code, if applicable.

The system functions discussed in this section are concerned with program termination and transfer of control.

5.2.1 Level 2 I/O Functions and Macros

These functions provide a buffered interface using a special structure, manipulated internally by the functions, to which a pointer called the file pointer is defined. This structure is defined in the standard I/O header file (called stdio__h) which must be included (by means of a #include statement) in the source file where level 2 features are being

used. The file pointer is used to specify the file upon which operations are to be performed. Some functions require a file pointer, such as

```
FILE *fp;
```

to be explicitly included in the calling sequence; others imply a specific file pointer. In particular, the file pointers stdin and stdout are implied by the use of several functions and macros; these files are so commonly used that they are opened automatically before the main function of a program begins execution. Other file pointers must be declared by the programmer and initialized by calls to the fopen function.

The level 2 functions are designed to work primarily with text files. The usual C convention for line termination uses a single character, the newline (\n), to indicate the end of a line. Unfortunately, many operating environments use a multiple character sequence (usually carriage return/line feed, but occasionally even more exotic delimiters). In order to allow all C programs to work with text files in the same way, the Lattice functions support the standard newline convention but perform a text mode translation so that end-of-line sequences will conform to local conventions. (No text translation is actually required on the Sinclair QL, but this facility is kept for compatibility with versions on other operating systems.) This translation is usually beneficial and transparent but may cause problems when working with binary files. Normally, all files accessed through the level 2 functions are opened in the text, or translated mode, but the programmer may override this mode by defining the external location

```
int _fmode = 0x8000;
```

in one of the functions in the program (this statement must appear outside the body of the function itself in order to be considered an external definition). The value at __fmode is passed to the level 1 function open or creat when the file is opened. If zero, the file is opened in the text mode; if 0x8000, the file is opened in the binary, or untranslated mode. Note that if __fmode is defined as above, the stdin, stdout, and stderr files opened for the main function will also be opened in the binary mode. If this is undesirable, __fmode can be initialized with zero and then set to 0x8000 before specific fopen calls are made; in this way, different files may be opened in different modes.

Files may also be explicitly opened in either the text or binary mode by using a special parameter in the fopen call; see below for details.

The actual I/O operations are performed by the level 2 functions through calls to the level 1 I/O functions described in the next section. The normal mode of buffering, designed to support sequential operations, performs read and write functions in 512-byte blocks.

Normally the level 2 functions acquire buffers via the level 2 memory allocator unless the file is on a device other than a disk. Alternatively, the setbuf function allows a private buffer to be attached. This function assumes that the buffer is the standard size, which is defined via the BUFSIZ constant in stdio_h. If for some reason operating the level 2 I/O functions in the buffered mode is not desirable, the setnbf function can be called. This is done automatically for non-disk files or if setbuf is called with a NULL buffer pointer.

In the descriptions below, some of the function calls are actually implemented as macros; these are noted explicitly. The reason the programmer should be aware of the distinction is because many macros involve the conditional operator and may, under certain conditions, evaluate an argument expression more than once. This can cause unexpected results if that expression involves side effects, such as increment or decrement operators or function calls. In addition, unlike functions, macros do not have addresses, making it impossible for pointers to them to be passed to other functions.

FOPEN/FREOPEN

Purpose:

Open/re-open a buffered file

Synopsis:

```
fp = fopen(name, mode);
fp = freopen(name, mode, fpx);
FILE *fp;          file pointer for specified file
char *name;        file name
char *mode;        access mode
FILE *fpx;         existing file pointer (freopen only)
```

Description:

These functions open a file for buffered access; the translated mode is the default mode but may be overridden as described in the introduction to this section, or by using the appropriate access mode described below. fopen returns a file pointer after finding a free slot in a pre-defined array (__iob), while freopen uses the file pointer supplied by the caller (useful for opening stdin, etc.); note that this file pointer is automatically closed by freopen before being reused. No more than 20 files (including stdin, stdout, and stderr) can be opened using fopen or freopen.

The null-terminated string which specifies the filename must conform to local file naming conventions. The access mode is also specified as a string, and may be one of the following:

r open for reading (mode set according to __fmode)
ra open for reading (translated)
rb open for reading (untranslated)

w open for writing (mode set according to __fmode)
wa open for writing (translated)
wb open for writing (untranslated)

a open for appending (mode set according to __fmode)
aa open for appending (translated)
ab open for appending (untranslated)

The mode characters must be specified in lower case. The a option adds to the end of an existing file, or creates a new one; the w option discards any data in the file, if it already exists; the r option simply reads an existing file. Opening the file to append forces all data to be written to the current end of file, regardless of previous seeks. If the character following the read/write mode indicator is "a", the file is opened in text or translated mode; if it is "b", the file is opened in binary or untranslated mode; if it is neither, the file is opened in the mode specified by __fmode as described in the introduction to this section.

Any of the above strings may be appended with a plus sign + to indicate opening for update (both reading and writing). In this mode, both reads and writes may be performed on the file; in order to switch between reading and writing, however, an fseek or rewind must be executed. If a file is opened for reading with a plus, then the file must already exist; but if a file is opened for writing with a plus, the file will be created anew. Opening for appending with a plus will permit reads to take place from any position in the file, but all write operations will occur at the end of the file.

Returns:

```
fp   = NULL if error
     = file pointer for specified file if successful
```

Cautions:

The return code must be checked for NULL; the error return may be generated if an invalid mode was specified or if the file was not found, could not be created, or too many files were already open.

FCLOSE

Purpose:

Close a buffered file

Synopsis:

```
ret = fclose(fp);
int ret;          return code
FILE *fp;         file pointer for file to be closed
```

Description:

Completes the processing of a file and releases all related resources. If the file was being written, any data which has accumulated in the buffer is written to the file, and the level 1 close function is called for the associated file descriptor. The buffer associated with the file block is freed. fclose is automatically called for all open files when a program calls the exit function (see section 5.2.4) or when the main program returns, but it is good programming practice to close files explicitly. As the last buffer is not written until fclose is called, data may be lost if an output file is not properly closed.

Returns:

ret = -1 if error
= 0 if successful

GETC/GETCHAR

Purpose:

Get character from file

Synopsis:

```
c = getc(fp);
c = getchar();
int c;              next input character or EOF
FILE *fp;           file pointer
```

Description:

Gets the next character from the indicated file (stdin, in the case of getchar). The value EOF (-1) is returned on end-of-file or error.

Returns:

c character
   EOF if end-of-file or error

Cautions:

Implemented as macros.

PUTC/PUTCHAR

Purpose:

Put character to file

Synopsis:

```
r = putc(c, fp);
r = putchar(c);
int r;              same as character sent, or error code
char c;             character to be output
FILE *fp;           file pointer
```

Description:

Puts the character to the indicated file (stdout, in the case of putchar). The value EOF (-1) is returned on end-of-file or error.

Returns:

r = character sent if successful
   = EOF if error or end-of-file

Cautions:

Implemented as macros.

FREAD/FWRITE

Purpose:

Read/write blocks of data from/to a file

Synopsis:

```
nact = fread(p, s, n, fp);
nact = fwrite(p, s, n, fp);
int nact;       actual number of blocks read or written
char *p;        pointer to first block of data
int s;          size of each block, in bytes
int n;          number of blocks to be read or written
FILE *fp;       file pointer
```

Description:

These functions read (fread) or write (fwrite) blocks of data from or to the specified file. Each block is of size s bytes; blocks start at p and are stored contiguously from that location. n specifies the number of blocks (of size s) that are to be read or written.

Returns:

nact = actual number of blocks (of size s) read or written; may be less than n if error or end-of-file occurred

Cautions:

Return value must be checked to verify that the correct number of blocks was processed. The ferror and feof macros can be used to determine the cause if the return value is less than n.

GETS/FGETS

Purpose:

Get a string

Synopsis:

```
p = gets(s);
p = fgets(s, n, fp);
char *p;        returned string pointer
char *s;        buffer for input string
int n;          number of bytes in buffer
FILE *fp;       file pointer
```

Description:

Gets an input string from a file. The specified file (stdin, in the case of gets) is read until a newline is encountered or n-1 characters have been read (fgets only). Then, gets replaces the newline with a null byte, while fgets passes the newline through with a null byte appended.

Returns:

p = NULL if end of file or error
= s if successful

Cautions:

For gets, there is no length parameter; thus, the input buffer provided must be large enough to accommodate the string.

PUTS/FPUTS

Purpose:

Put a string

Synopsis:

```
r = puts(s);
r = fputs(s, fp);
int r;          return code
char *s;        output string pointer
FILE *fp;       file pointer
```

Description:

Puts an output string to a file. Characters from the string are written to the specified file (stdout, in the case of puts) until a null byte is encountered. The null byte is not written, but puts appends a newline.

Returns:

r = EOF if end-of-file or error

SCANF/FSCANF/SSCANF

Purpose:

Perform formatted input conversions

Synopsis:

```
n = scanf(cs, ...ptrs...);
n = fscanf(fp, cs, ...ptrs...);
n = sscanf(ss, cs, ...ptrs...);
int n;          number of input items matched, or EOF
FILE *fp;       file pointer (fscanf only)
char *ss;       input string (sscanf only)
char *cs;       format control string
---- ...ptrs...; pointers for return of input values
```

Description:

These functions perform formatted input conversions on text obtained from three types of files:

1 the stdin file (scanf);
2 the specified file (fscanf); or
3 the specified string (sscanf).

The control string contains format specifiers and/or characters to be matched from the input; the list of pointer arguments specify where the results of the conversions are to go. Format specifiers are of the form

```
%[*][n][l]X
```

where

1 the optional * means that the conversion is to be performed, but the result value not returned;
2 the optional n is a decimal number specifying a maximum field width;
3 the optional l (el) is used to indicate an int or long float (i.e., double) result is desired;
4 X is one of the format type indicators from the following list:

d -- decimal integer
o -- octal integer
x -- hexadecimal integer
h -- short integer
c -- single character
s -- character string
f -- floating point number

The format type must be specified in lower case. White space characters in the control string are ignored; characters other than format specifiers are expected to match the next non-white space characters in the input. The input is scanned through white space to locate the next input item in all cases except the c specifier, where the next input character is returned without this initial scan. Note that the %s specifier terminates on any white space. See the Kernighan and Ritchie text for a more detailed explanation of the formatted input functions.

Returns:

n = number of input items successfully matched, i.e., for which valid text data was found; this includes all single character items in the control string
= EOF if end-of-file or error is encountered during scan

Cautions:

All of the input values must be pointers to the result locations. Make sure that the format specifiers match up properly with the result locations. If the assignment suppression feature (*) is used, remember that a pointer must not be supplied for that specifier.

PRINTF/FPRINTF/SPRINTF

Purpose:

Generate formatted output *

Synopsis:

```
Printf(cs, ...args...);
fprintf(fp, cs, ...args...);
n = sprintf(ds, cs, ...args...);
int n;              number of characters (sprintf only)
FILE *fp;           file pointer (fprintf)
char *ds;           destination string pointer (sprintf)
char *cs;           format control string
--- ...args...;     list of arguments to be formatted
```

Description:

These functions perform formatted output conversions and send the resulting text to:

1 the stdout file (printf);
2 the specified file (fprintf); or
3 the specified output string (sprintf).

The control string contains ordinary characters, which are sent without modification to the appropriate output, and format specifiers of the form

```
%[-][m][.p][l]X
```

where

1 the optional - indicates the field is to be left justified (right justified is the default);
2 the optional m field is a decimal number specifying a minimum field width;
3 the optional .p field is the character . followed by a decimal number specifying the precision of a floating point image or the maximum number of characters to be printed from a string;
4 the optional l (el) indicates that the item to be formatted is long (an int on the Sinclair QL); and
5 X is one of the format type indicators from the following list:

d -- decimal signed integer

u -- decimal unsigned integer

x -- hexadecimal integer

o -- octal integer

s -- character string

c -- single character

f -- fixed decimal floating point

e -- exponential floating point

g -- use e or f format, whichever is shorter

The format type must be specified in lower case. Characters in the control string which are not part of a format specifier are sent to the appropriate output; a % may be sent by using the sequence %%. See the Kernighan and Ritchie text for a more detailed explanation of the formatted output functions.

Returns:

n = number of characters placed in ds (sprintf only), not including the null byte terminator

Cautions:

For sprintf, no check of the size of the output string area is made; thus, the buffer provided must be large enough to contain the resulting image. In all cases, the format specifiers must match up properly with the supplied values for formatting.

FSEEK

Purpose:

Seek to a new file position

Synopsis:

```
ret = fseek(fp, pos, mode);
int ret;          return code
FILE *fp;         file pointer
long pos;         desired file position
int mode;         offset mode
```

Description:

Seeks to a new position in the specified file. See the lseek function description (section 5.2.2) for the meaning of the offset mode argument.

Returns:

ret = 0 if successful
= -1 if error

Cautions:

If mode 1 is specified, the file position established for files being accessed in the translated mode may be incorrect.

FTELL

Purpose:

Return current file position

Synopsis:

```
pos = ftell(fp);
long pos;          current file position
FILE *fp;          file pointer
```

Description:

Returns the current file position, that is, the number of bytes from the beginning of the file to the byte at which the next read or write operation will transfer data.

Returns:

pos = current file position (long)

Cautions:

The file position returned takes account of the buffering used on the file, so that the file position returned is a logical file position rather than the actual position. Note that text mode translation may cause an incorrect file position to be returned, since the number of characters in the buffer is not necessarily the number that will be actually read or written because of the translation.

FERROR/FEOF

Purpose:

Check if error/end of file

Synopsis:

```
ret = feof(fp);
ret = ferror(fp);
int ret;           return code
FILE *fp;          file pointer
```

Description:

These macros generate a non-zero value if the indicated condition is true for the specified file.

Returns:

ret = non-zero if error (ferror) or end of file (feof)
= zero if not

CLRERR/CLEARERR

Purpose:

Clear error flag for file

Synopsis:

```
clrerr(fp);
clearerr(fp);
FILE *fp;          file pointer
```

Description:

Clears the error flag for the specified file. Once set, the flag will remain set, forcing EOF returns for functions on the file, until this function is called.

FILENO

Purpose:

Return file number for file pointer

Synopsis:

```
fn = fileno(fp);
int fn;          file number associated with file pointer
FILE *fp;        file pointer
```

Description:

Returns the file number, used for the level 1 I/O calls, for the specified file pointer.

Returns:

fn = file number (file descriptor) for level 1 calls

Cautions:

Implemented as a macro.

REWIND

Purpose:

Rewind a file

Synopsis:

```
rewind(fp);
FILE *fp;          file pointer
```

Description:

Resets the file position of the specified file to the beginning of the file.

Cautions:

Implemented as a macro.

FFLUSH

Purpose:

Flush output buffer for file

Synopsis:

```
fflush(fp);
FILE *fp;          file pointer
```

Description:

Flushes the output buffer of the specified file, that is, forces it to be written.

Cautions:

This macro must be used only on files which have been opened for writing or appending.

SETBUF

Purpose:

Change buffer for level 2 file I/O

Synopsis:

```
setbuf(fd,buf);
FILE *fp;          file pointer for file
char *buf;         pointer to buffer to be attached
```

Description:

Attaches a private buffer to the file whose file pointer is fp. The length of the buffer is assumed to be the same as bufsiz, which is defaulted to the constant BUFSIZ defined in stdio__h.

If the buffer pointer is NULL, then this function is equivalent to setnbf.

Cautions:

Buf must be large enough to accommodate __bufsiz charcacters.

SETNBF

Purpose:

Set file unbuffered

Synopsis:

```
setnbf(fp);
FILE *fp;          file pointer
```

Description:

Changes the buffering mode for the specified file pointer from the default 512-byte block mode to the unbuffered mode used for devices (including the user's console). In this mode, read and write operations are performed using single characters.

Cautions:

Although the unbuffered mode may be used without difficulty on files, the standard buffering mode is generally more efficient. Thus, this function should only be used for those "files" which are definitely known to be devices.

5.2.2 Level 1 I/O Functions

These functions provide a basic, low-level I/O interface which allows a file to be viewed as a stream of randomly addressable bytes. Operations are performed on the file using the functions described in this section; the file is specified by a file number or file descriptor, such as

```
int fd;
```

which is returned by open or creat when the file is opened. Data may be read or written in blocks of any size, from a single byte to as much as several kilobytes in a single operation. The concept of a file position is key: the file position is a long integer, such as

```
long fpos;
```

which specifies the position of a byte in the file as the number of bytes from the beginning of the file to that particular byte. Thus, the first byte in the file is at file position 0L. Two distinct file positions are maintained internally by the level 1 functions. The current file position is the point at which data transfers take place between the program and the file; it is set to zero when the file is opened, and is advanced by the number of bytes read or written using the read and write functions. The end of file position is simply the total number of bytes contained in the file; it is changed only by write operations which increase the size of the file.

The current file position can be set to any value from zero up to and including the end of file position using the lseek function. Thus, to append data to a file, the current file position is set to the end of the file using lseek before any write operations are performed. When data is read from near the end of file, as much of the requested count as can be satisfied is returned; zero is returned for attempts to read when the file position is at the end of file.

The level 1 functions operate in one of two mutually exclusive modes: the text or translated mode, and the binary or untranslated mode. The desired mode is specified when the file is opened or created, and remains in effect until the file is closed. The two modes are provided for compatibility with operating systems that require translation of text file end-of-file sequences.

A public symbol called __iomode presets the translation mode. Normally, __iomode is 0 And translated mode is used unless O__RAW is specified (see open function). If __iomode is changed to 0x8000, then the untranslated mode is used unless O__RAW is specified. In other words, O__RAW toggles the meaning of __iomode. Note that, since the level 2 I/O functions call open, this change to __iomode affects the meaning of the corresponding options on fopen calls.

Although the level 1 functions are primarily useful for working with files, they can be used to read and write data to devices (including the user's terminal), as well. The I/O performed is unbuffered. The lseek function has no effect on devices, and usually returns an error status. Direct I/O to the user's terminal may also be performed using the functions described in section 5.2.3.

The actual I/O operations on disk files are buffered, but at a level that is generally transparent to the programmer. The buffering makes close operations a necessity for files that are modified.

OPEN

Purpose:

Open a file

Synopsis:

```
file = open(name, rwmode);
int file;          file number or error code
char *name;        file name
int mode;          indicates read/write mode and other modes
                   (see below)
```

Description:

Opens a file for access using the level 1 I/O functions. The file name must conform to local naming conventions. The mode word indicates the type of I/O which will be performed on the file. The header file fnctl__h defines the codes for the mode arguments:

```
O_RDONLY    Read only access
O_WRONLY    Write only access
O_RDWR      Read/write access
```

Also, the following flags can be ORed into the above codes:

```
O_CREAT     Create the file if it doesn't exist
O_TRUNC     Truncate (set to zero length) the
            file if it does exist
O_EXCL      Force create to fail if file exists
O_APPEND    Seek to end-of-file before each write
O_RAW       Use untranslated I/O (see
            introduction to section 5.2.2)
```

The current file position is set to zero if the file is successfully opened. No more than 20 files (including any which are being accessed through

the level 2 functions, such as stdin, stdout, etc.) can be open at the same time. Closing the file releases the file number for use with some other file.

Returns:

file = file number to access file, if successful
= -1 if error

Cautions:

Check the return value for error.

CREAT

Purpose:

Create a new file.

Synopsis:

```
file = creat(name, pmode);
int file;       file number or error code
char *name;     file name
int pmode;      access privilege mode bits; bit 15 has same
                meaning as for open
```

Description:

Creates a new file with the specified name and prepares it access via the level 1 I/O functions. The file name must conform to local naming conventions. Creating a device is equivalent to opening it. The access privilege mode bits are largely ignored; however, bit 15 is interpreted in the same way as for open: if set, operations are performed on the file without translation. If the file already exists, its contents are discarded. The current file position and the end-of-file are both zero (indicating an empty file) if the function is successful.

Returns:

file = file number to access file, if successful
= -1 if error

Cautions:

Check the return value for error. creat should be used only on files which are being completely rewritten, since any existing data is lost.

UNLINK/REMOVE

Purpose:

Remove file name from file system

Synopsis:

```
ret = unlink(name);
ret = remove(name);
int ret;          return code: 0 if successful
char *name,       name of file to be removed
```

Description:

Removes the specified file from the file system. The file name must conform to local naming conventions. The specified file must not be currently open. All data in the file is lost.

Returns:

ret = 0 if successful
= -1 if error

Cautions:

Should be used with care since the file, once removed, is generally irretrievable.

READ

Purpose:

Read data from file

Synopsis:

```
status = read(file, buffer, length);
int status;       status code or actual length
int file;         file number for file
char *buffer;     input buffer
int length;       number of bytes requested
```

Description:

Reads the next set of bytes from a file. The return count is always equal to the number of bytes placed in the buffer and will never exceed the length parameter, except in the case of an error, where -1 is returned. The file position is advanced accordingly.

Returns:

status = 0 if end-of-file
= -1 if error occurred
= number of bytes actually read, otherwise

Cautions:

If fewer than the requested number of bytes remain between the current file position and the end-of-file, only that number is transferred and returned. The number of bytes by which the file position was advanced may not equal the number of bytes transferred if text mode translation occurred.

WRITE

Purpose:

Write data to file

Synopsis:

```
status = write(file, buffer, length);
int status;          status code or actual length
int file;            file number
char *buffer;        output buffer
int length;          number of bytes in buffer
```

Description:

Writes the next set of bytes to a file. The return count is equal to the number of bytes written, unless an error occurred. The file position is advanced accordingly.

Returns:

status = -1 if error
= number of bytes actually written

Cautions:

The number of bytes written may be less than the supplied count if a physical end-of-file limitation was encountered.

LSEEK

Purpose:

Seek to specified file position

Synopsis:

```
pos = lseek(file, offset, mode);
long pos;       returned file position or error code
int file;       file number for file
long offset;    desired position
int mode;       offset mode:
                0 = relative to beginning of file
                1 = relative to current file position
                2 = relative to end-of-file
```

Description:

Changes the current file position to a new position in the file. The offset is specified as a long int and is added to the current position (mode 1) or to the end-of-file (mode 2).

Returns:

pos = -1L if error occurred
= new file position if successful

Cautions:

The offset parameter must be a long quantity; therefore a long constant should be indicated when supplying a zero. In most cases, the return code should be checked for error, which indicates that an invalid file position (beyond the end-of-file) was specified. Note that the current file position may be obtained by

```
long cpos, lseek();
. . .
cpos = lseek(file, 0L, 1);
```

which will never return an error code.

CLOSE

Purpose:

Close a file

Synopsis:

```
status = close(file);
int status;          status code: 0 if successful
int file;            file number
```

Description:

Closes a file and frees the file number for use in accessing another file. Any buffers allocated when the file was opened are released.

Returns:

status = 0 if successful
= -1 if error

Cautions:

This function must be called if the file was modified; otherwise, the end-of-file and the actual data on disk may not be updated properly.

5.2.3 Direct Console I/O Functions

These functions provide a direct I/O interface to the user's console. Because there is no buffering of characters, the functions are particularly useful for applications which use cursor positioning to define special screen formats or which implement special single character responses to program prompts. In order to distinguish these functions from the corresponding level 2 functions, different names are used for them. This allows programs to make use of both kinds of I/O, if desired. Programs which perform console I/O exclusively can use the #define mechanism to establish the following equivalencies for some of the level 2 functions:

```
#define getchar getch
#define putchar putch
#define gets cgets
#define puts cputs
#define scanf cscanf
#define printf cprintf
```

GETCH/GETCHE/PUTCH

Purpose:

Get/put character directly from/to console

Synopsis:

```
c = getch();        get character with no echo
c = getche;         get character with echo
putch(c);           put character
int c;              character received/sent to console
```

Description:

These functions get (getch, getche) or put (putch) single characters from or to the user's console. putch puts a carriage return character in front of each newline. getch does not echo the characters typed at the console, while getche does echo them.

```
c = character received (getch, getche)
```

Cautions:

There is no notion of an end of file or error status.

UNGETCH

Purpose:

Push character back to console

Synopsis:

```
r = ungetch(c);
int r;              return code
char c;             character to be pushed back
```

Description:

Pushes the indicated character back on the console. Only one character of pushback is allowed. The effect is to cause getch (or getche) to return the pushed-back character next time it is called.

Returns:

r = EOF if a character has already been pushed back
= c if successful

KBHIT

Purpose:

Check for keyboard hit

Synopsis:

```
hit = kbhit();
int hit;                0 if no hit
```

Description:

Returns a non-zero value if a keyboard character is available (i.e., if the next call to getch or getche will return immediately with an input character).

Returns:

hit = 0 if no character available
= non-zero if character available

CGETS

Purpose:

Get string directly from console

Synopsis:

```
p = cgets(s);
char *p;                returned string pointer
char *s;                input string buffer
```

Description:

Gets a string directly from the user's console by making calls to the getch function. Characters are input until a carriage return is encountered. The carriage return is replaced by a zero byte.

Returns:

p = pointer to string received, which does not include the terminating carriage return

Cautions:

There is no check on the length of the input string; thus, the buffer supplied must be large enough to accommodate the result. Input strings larger than 256 bytes are not supported.

CPUTS

Purpose:

Put string directly to console

Synopsis:

cputs(s);
char *s; string to be output

Description:

Puts a null terminated string directly to the user's console by making calls to the putch function. Does not automatically generate a carriage return or linefeed.

CSCANF/CPRINTF

Purpose:

Formatted I/O directly to console

Synopsis:

Same as scanf and printf

Description:

These functions perform the equivalent of scanf and printf, but characters are sent directly to or received directly from the console using putch and getch.

Returns:

n = number of input items matched (cscanf)

Cautions:

scanf performs its I/O directly using getch, so there are none of the usual input conveniences such as back spacing or line deletion.

5.2.4 Program Exit and Jump Functions

The simplest way to terminate execution of a C program is for the main function to execute a return statement, or -- even simpler -- to "drop through" its terminating brace. In many cases, however, a more flexible program exit capability is needed; this is provided by the exit and _exit functions described in this section. The offer the advantage of allowing any function -- not just main -- to cause termination of the program, and they allow information to be passed to other programs.

In some cases, it is useful for a program to be able to pass control directly to another part of the program (within a different function) without having to go through a long and possibly complicated series of function returns. The setjmp and longjmp functions provide a general capbility for achieving this.

EXIT

Purpose:

Terminate execution of program and close files

Synopsis:

```
exit(errcode);
int errcode;          exit error code
```

Description:

Terminates execution of the current program, but first closes all output files which are currently open through the level 2 I/O functions. The error code is normally set to zero to indicate no error, and to a non-zero value if some kind of error exit was taken.

Cautions:

Note that exit only closes those files which are being accessed using the level 2 functions. Files accessed using the level 1 functions are not automatically closed.

_EXIT

Purpose:

Terminate execution immediately

Synopsis:

```
_exit(errcode);
int errcode;          exit error code
```

Description:

Terminates execution of the current program immediately, without checking for open files.

SETJMP/LONGJMP

Purpose:

Perform non-local goto

Synopsis:

```
ret = setjmp(save);
longjmp(save,value);
int ret;              return code
int value;            return value
jmp_buf save;         context save buffer
```

Description:

The setjmp function saves the current stack mark in the buffer area specified by save and returns a value of 0. A subsequent call to longjmp will then cause control to return to the next statement after the original setjmp call, with value as the return code. If value is 0, it is forced to 1 by longjmp.

The jmp_buf descriptor is defined in the header file called setjmp_h.

This mechanism is useful for quickly popping back up through multiple layers of function calls under exceptional circumstances. Structured programming gurus lose a lot of sleep over the "pathological connections" that can result from indiscriminate usage of these functions.

SETMEM

Purpose:

Initialize memory to specified char value

Synopsis:

```
setmem(p, n, c);
char *p;        base of memory to be initialized
unsigned n;     number of bytes to be initialized
char c;         initialization value
```

Description:

Sets the specified number of bytes of memory to the specified byte value. This function is useful for the initialization of auto char arrays.

Cautions:

It is good practice to use a cast operator when arrays or pointers of other types are used for the p argument.

MOVMEM

Purpose:

Move a block of memory

Synopsis:

```
movmem(s, d, n);
char *s;        source memory block
char *d;        destination memory block
unsigned n;     number of bytes to be transferred
```

Description:

Moves memory from one location to another. The function checks the relative locations of source and destination blocks, and performs the move in the order necessary to preserve the data in the event of overlap.

Cautions:

It is good practice to use a cast operator when arrays or pointers of other types are used for the s and d arguments.

REPMEM

Purpose:

Replicate values through memory

Synopsis:

```
repmem(s, v, lv, nv);
char *s;        memory to be initialized
char *v;        template of values to be replicated
int lv;         number of bytes in template
int nv;         number of templates to be replicated
```

Description:

Replicates a set of values throughout a block of memory. This function is a generalized version of setmem, and can be used to initialize arrays of items other than char. Note that the replication count indicates the number of copies of v which are to be made, not the total number of bytes to be initialized.

Cautions:

It is good practice to use a cast operator when arrays or pointers of other types are used for the s and v arguments.

5.3.2 Character Type Macros and Functions

The character type header file, called ctype__h, defines several macros which are useful in the analysis of text data. Most allow the programmer to determine quickly the type of a character, i.e., whether it is alphabetic, numeric, punctuation, etc. These macros refer to an external array called __ctype which is indexed by the character itself, and so they are generally much faster than functions which check the character against a range or discrete list of values. Although ASCII is defined as a 7-bit code, the __ctype array is defined to be 257 bytes long so that valid results are obtained for any character value. This means that a character with the value 0xB1, for instance, will be classified the same as a character with the value 0x31. Programs that need to distinguish between these values must test for the 0x80 bit before using one of these macros. Note that __ctype is actually indexed by the character value plus one; this allows the standard EOF value (-1) to be tested in a macro without yielding a nonsense result. EOF yields a zero result for any of the macros: it is not defined as any of the character types.

The following list presents the macros defined in the character type header file ctype__h. Note that many of these will evaluate argument expressions more than once; beware of using expressions with side effects, such as function calls or increment or decrement operators. Note that the file ctype__h must be included if any of these macros are used; otherwise, the compiler will generate a reference to a function of the same name. Those macros marked with a '*' are also available in function form. In order to use the function form, do not #include the ctype__h header file in that source file. If some of the other capabilities of ctype__h are needed, the header file should be included anyway; #undef directives can be used for the specific macros that need to be treated as functions.

isalpha(c)	non-zero if c is alphabetic, 0 if not
isupper(c)	non-zero if c is upper case, 0 if not
islower(c)	non-zero if c is lower case, 0 if not
isdigit(c)	non-zero if c is a digit 0-9, 0 if not
isxdigit(c)	non-zero if c is a hexadecimal digit, 0 if not (0-9, A-F, a-f)
isspace(c)	non-zero if c is white space, 0 if not
ispunct(c)	non-zero if c is punctuation, 0 if not
isalnum(c)	non-zero if c is alphabetic or digit
isprint(c)	non-zero if c is printable (including blank)
isgraph(c)	non-zero if c is graphic (excluding blank)
iscntrl(c)	non-zero if c is control character
isascii(c)	non-zero if c is ASCII (0-127)
iscsym(c)	non-zero if valid character for C identifier, 0 if not
iscsymf(c)	non-zero if valid first character for C identifier, 0 if not
toupper(c)	converts c to upper case, if lower case
tolower(c)	converts c to lower case, if upper case

Note that the last two macros generate the value of c unchanged if it does not qualify for the conversion.

5.3.3 String Utility Functions

The portable library provides several functions to perform many of the most common string manipulations. These functions all work with sequences of characters terminated by a null (zero) byte, which is the C definition of a character string. A special naming convention is used, which works as follows: The first two characters of a string function are always st, while the third character indicates the type of the return value from the function:

stc	function returns an int count
stp	function returns a character pointer
sts	function returns an int status value

Thus, the name of the function shows at a glance the type of value it returns.

For compatibility with other C implementations, several other functions whose names begin with str have also been provided.

STRCAT/STRNCAT

Purpose:

Concatenate strings

Synopsis:

```
to = strcat(to, from);
to = strncat(to, from, max);

char *to;          destination string
char *from;        source string
int max;           maximum number of characters
```

Description:

These functions append the "from" string to the "to" string. For strncat, no more than the specified maximum number of characters will be appended. The result is always null-terminated.

Returns:

to = pointer to result (same as original to argument)

Cautions:

strncat should be used if there is any question that the destination string might not be large enough to hold the result. Either function must be declared char * if the return value is to be used.

STRLEN/STCLEN

Purpose:

Measure length of string

Synopsis:

```
length = stclen(s);
length = strlen(s);
int length;          number of bytes in s (before null)
```

Description:

These functions count the number of bytes in s before the null terminator. The terminator itself is not included in the count. strlen is provided for compatibility with other implementations.

Returns:

length = number of bytes in string before null byte

STRCPY/STRNCPY/STPCPY/STCCPY

Purpose:

Copy one string to another

Synopsis:

```
to = strcpy(to, from);
to = strncpy(to, from, length);
to = stpcpy(to, from);
actual = stccpy(to, from, length);
int actual;          actual number of characters moved (stccpy only)
char *to;            destination string pointer
char *from;          source string pointer
int length;          maximum length of copy
```

Description:

These functions move the null-terminated source string to the destination string. For strncpy and stccpy, if the source is too long for the destination, its rightmost characters are not moved. The destination string is always null-terminated.

Returns:

to = pointer to destination string (same as original to argument) (strcpy, strncpy, stpcpy)

actual = actual number of characters moved, including the null terminator (stccpy only)

Cautions:

strncpy or stccpy should be used if there is any question that the destination string might not be large enough to hold the result. Functions returning char * must be so declared before being used.

STRCMP/STRNCMP/STSCMP

Purpose:

Compare two strings

Synopsis:

```
status = strcmp(s, t);
status = strncmp(s, t, length);
status = stscmp(s, t);
int status;          result of comparison >0 if s>t, 0 if s==t, <0
                     if s<t
char *s;             first string to compare
char *t;             second string to compare
int length;          length of comparison (strncmp only)
```

Description:

These functions compare two null-terminated strings, byte by byte, and return an int status indicating the result of the comparison. If zero, the strings are identical, up to and including the terminating byte. If non-zero, the status indicates the result of the comparison of the first pair of bytes which were not equal. For strncmp, no more than the specified number of characters will be compared.

Returns:

status = 0 if strings match
< 0 if first string less than second string
> 0 if first string greater than second string

Cautions:

The result of the comparison may depend on whether characters are considered signed, if any of the characters is greater than 127.

STCU__D

Purpose:

Convert unsigned integer to decimal string

Synopsis:

```
length = stcu__d(out, in, outlen);
int length;          output string length (excluding null)
char *out;           output string
unsigned in;         input value
int outlen;          sizeof(out)
```

Description:

Converts an unsigned integer into a string of decimal digits terminated with a null byte. Leading zeros are not copied to the output string, and if the input value is zero, only a single 0 character is produced.

Returns:

length = number of characters placed in output string, not including the null terminator

Cautions:

If the output string is too small for the result, only the rightmost digits are returned.

STCI__D

Purpose:

Convert signed integer to decimal string

Synopsis:

```
length = stci__d(out, in, outlen);
int length;          output string length (excluding null)
char *out;           output string
int in;              input value
int outlen;          sizeof(out)
```

Description:

Converts an integer into a string of decimal digits terminated with a null byte. If the integer is negative, the output string is preceded by a -. Leading zeros are not copied to the output string.

Returns:

length = number of characters placed in output string, not including the null terminator

Cautions:

If the output string is too small for the result, the returned length may be zero, or a partial string may be returned.

STCH_I

Purpose:

Convert hexadecimal string to integer

Synopsis:

```
count = stch_i(p, r);
int count;      number of characters scanned
char *p;        input string
int *r;         result integer
```

Description:

Converts a hexadecimal string into an integer. The process terminates only when a non-hex character is encountered. Valid hex characters are 0-9, A-F, and a-f.

Returns:

count = 0 if input string does not begin with a hex digit
= number of characters scanned

Cautions:

No check for overflow is made during the processing.

STCD_I

Purpose:

Convert decimal string to integer

Synopsis:

```
count = stcd_i(p, r);
int count;      number of characters scanned
char *p;        input string
int *r;         result integer
```

Description:

Converts a decimal string into an integer. The process terminates when a non-decimal character is found. Valid decimal characters are 0-9. The first character may be + or -.

Returns:

count = 0 if input string does not begin with a decimal digit
= number of characters scanned

Cautions:

No check for overflow is made during the processing.

STPBLK

Purpose:

Skip blanks (white space)

Synopsis:

```
q = stpblk(p);
char *q;        updated string pointer
char *p;        initial string pointer
```

Description:

Advances the string pointer past white space characters (space, tab, or newline).

Returns:

q = updated string pointer (advanced past white space)

Cautions:

Must be declared char *, as the stp prefix indicates.

STPSYM

Purpose:

Get a symbol from a string

Synopsis:

```
p = stpsym(s, sym, symlen);
char *p;        points to next character in s
char *s;        input string
char *sym;      output string
int symlen;     sizeof(sym)
```

Description:

Breaks out the next symbol from the input string. The first character of the symbol must be alphabetic (upper or lower case), and the remaining characters must be alphanumeric. Note that the pointer is not advanced past any initial white space in the input string. The output string is the null-terminated symbol.

Returns:

p = pointer to next character (after symbol) in input string

Cautions:

Must be declared char *, as the stp prefix indicates. If no valid symbol characters are found, p will equal s, and sym will contain an initial null byte.

STPTOK

Purpose:

Get a token from a string

Synopsis:

```
p = stptok(s, tok, toklen, brk);
char *p;        points to next char in s
char *s;        input string
char *tok;      output string
int toklen;     sizeof(tok)
char *brk;      break string
```

Description:

Breaks out the next token from the input string. The token consists of all characters in s up to but not including the first character that is in the break string. In other words, the break string defines a list of characters which cannot be included in a token. Note that the pointer is not advanced past any initial white space characters in the input string. The output string is the null-terminated token.

Returns:

p = pointer to next character (after token) in input string

Cautions:

Must be declared char *, as the stp prefix indicates. If no valid token characters are found, p will equal s, and tok will contain an initial null byte.

STPCHR/STRCHR/STRRCHR

Purpose:

Find specific character in string

Synopsis:

```
p = stpchr(s, c);
p = strchr(s, c);
p = strrchr(s, c);
char *p;        points to c in s (or is NULL)
char *s;        points to string being scanned
char c;         character to be located
```

Description:

The stpchr and strchr functions scan the specified string to find the first occurrence of the specified character, while the strrchr function scans for the last occurrence of the character. In either case, a NULL pointer is returned if the character is not found in the string.

Returns:

p = NULL if c not found in s
  = pointer to first c found in s (stpchr, strchr)
  = pointer to last c found in s (strrchr)

Cautions:

These functions must be declared char *.

STPBRK/STRPBRK

Purpose:

Find break character in string

Synopsis:

```
p = stpbrk(s, b);
p = strpbrk(s, b);
char *p;        points to element of b in s
char *s;        points to string being scanned
char *b;        points to break character string
```

Description:

These functions scan the specified string to find the first occurrence of a character from the break string b. In other words, b is a null-terminated list of characters being sought. If the terminator byte for s is hit first, a NULL pointer is returned.

Returns:

p = NULL if no element of b is found in s
= pointer to first element of b in s (from left)

Cautions:

These functions must be declared char *.

STRSPN/STRCSPN/STCIS/STCISN

Purpose:

Find longest initial span

Synopsis:

```
length = strspn(s, b);
length = strcspn(s, b);
length = stcis(s, b);
length = stcisn(s, b);
int length;     span length in bytes
char *s;        points to string being scanned
char *b;        points to character set string
```

Description:

These functions compute the number of characters at the beginning (left) of s that come from a specified character set. For strspn and stcis, the character set consists of all characters in b, while for strcspn and stcisn, the character set consists of all characters not in b.

Returns:

length = number of characters from the specified set which appear at the beginning (left) of s

STCARG

Purpose:

Get an argument

Synopsis:

```
length = stcarg(s, b);
int length;     number of bytes in argument
char *s;        text string pointer
char *b;        break string pointer
```

Description:

Scans the text string until one of the break characters is found or until the text string ends (as indicated by a null character). While scanning, the function skips over partial strings enclosed in single or double quotes, and the backslash is recognized as an escape character.

Returns:

length = number of bytes (in s) in argument
= 0 if not found

STCPM

Purpose:

Pattern match (unanchored)

Synopsis:

```
length = stcpm(s, p, q);
int length;     length of matched string
char *s;        string being scanned
char *p;        pattern string
char **q;       points to matched string if found
```

Description:

Scans the specified string to find the first substring that matches the specified pattern. The pattern is specified in a simple form of regular expression notation, where

```
?       matches any character
s*      matches zero or more occurrences of s
s+      matches one or more occurrences of s
```

The backslash is used as an escape character (to match one of the special characters ?, *, or +). The scan is not anchored; that is, if a matching string is not found at the first position of s, the next position is tried, and so on. A pointer to the first matching substring is returned at *q.

Returns:

length = 0 if no match
= length of matching substring, if successful

Cautions:

Note that the third argument must be a pointer to a character pointer, since this function really returns two values: a pointer to, and the length of, the first matching substring.

STCPMA

Purpose:

Pattern match (anchored)

Synopsis:

```
length = stcpma(s, p);
int length;          length of matching string
char *s;             string being scanned
char *p;             pattern string
```

Description:

Scans the specified string to determine if it begins with a substring that matches the specified pattern. See the description of stcpm for a specification of the pattern format.

Returns:

length = 0 if no match
= length of matching substring if successful

STSPFP

Purpose:

Parse file pattern

Synopsis:

```
error = stspfp(p, n);
int error;          return code: -1 if error
char *p;            file name string
int n[16];          node index array
```

Description:

Parses a file name pattern which consists of node names separated by underscores. Each underscore is replaced by a null byte, and the beginning index of that node is placed in the index array. For example, the pattern mdv1__ace__c has three nodes, and their indexes are 1 for mdv1, 6 for ace, and 10 for c. The last entry in the node array n is set to -1 (in the example above, this causes n[3] to be -1).

Returns:

error = 0 if successful
= -1 if too many nodes or other error

5.3.4 Utility Macros

The standard I/O header file stdio__h defines three general utility macros which are useful in working with arithmetic objects. They are:

```
max(a,b)        returns the maximum of a and b
min(a,b)        returns the minimum of a and b
abs(a)          returns the absolute value of a
```

Several important restrictions must be noted.

First, since these are macros which use the conditional operator, arguments with side effects (such as function calls or increment or decrement operators) cannot be used, and the address-of operator cannot be applied to these "functions". Second, beware of using the macro names in declarations such as

```
int min;
```

because the compiler will try to expand min as a macro, and an error message complaining of invalid macro usage will be generated. Third, only arithmetic data items should be used as arguments to these macros; max and min should be supplied two arguments of the same data type, although conversion will be performed if necessary.

5.4 Mathematical Functions

The functions described here include a large proportion of the floating point math functions that are usually provided with UNIX. Detailed specifications are given in the following manual pages. Note that the header files math_h and limits_h should generally be included when using these functions.

EXP/LOG/LOG10/POW/SQRT

Purpose:

Exponential/logarithmic functions

Synopsis:

```
r = exp(x);        compute exponential function of x
r = log(x);        compute natural log of x
r = log10(x);      compute base 10 log of x
r = pow(x,y);      compute x to power y
r = sqrt(x);       compute square root of x

double r;          result
double x,y;        arguments
```

Description:

These functions return the result of the indicated exponential, logarithmic, and power computations on double operands.

For log, log10, and sqrt, the x argument must be positive, and for pow, the y argument must be an integer if x is negative.

Cautions:

These functions must be declared double, which can be accomplished simply by including math_h.

SIN/COS/TAN/ASIN/ACOS/ATAN/ATAN2

Purpose:

Trigonometric functions

Synopsis:

```
x = sin(r);      compute sine of r (r in radians)
x = cos(r);      compute cosine of r
x = tan(r);      compute tangent of r
r = asin(x);     compute arcsine of x
r = acos(x);     compute arccosine of x
r = atan(x);     compute arctangent of x
r = atan2(y,x);  compute arctangent of y/x

double r;        result
double x,y;      arguments
```

Description:

The sin, cos, and tan functions compute the normal trigonometric functions of angles expressed in radians.

The asin function computes the inverse sine and returns a radian value in the range -PI/2 to + PI/2.

The acos function computes the inverse cosine and returns a radian value in the range 0 to PI.

The atan function computes the inverse tangent and returns a radian value in the range -PI/2 to + PI/2.

The atan2 function computes the inverse sine of y/x and returns a radian value in the range -PI to + PI.

Cautions:

These function must be declared double, which can be accomplished simply by including math_h.

SINH/COSH/TANH

Purpose:

Hyperbolic functions

Synopsis:

```
x = sinh(y);        compute hyperbolic sine
x = cosh(y);        compute hyperbolic cosine
x = tanh(y);        compute hyperbolic tangent

double x;           result
double y;           argument
```

Description:

These functions compute and return the value of the indicated hyperbolic functions.

Cautions:

These functions must be declared double, which can be accomplished simply by including math_h.

RAND,SRAND

Purpose:

Simple random number generation

Synopsis:

```
x = rand();
srand(seed);

int x;              random number
unsigned seed;      random number seed
```

Description:

The rand function returns pseudo-random numbers in the range from 0 to the maximum positive integer value. At any time, srand can be called to reset the number generator to a new starting point. The initial default seed is 1. See the description of drand for more sophisticated random number generation.

DRAND

Purpose:

Generate random numbers

Synopsis:

```
x = drand48();          generate double (internal seed)
x = erand48(y);         generate double (external seed)

z = lrand48();          generate positive long (internal seed)
z = nrand48(y),         generate positive long (external seed)

z = mrand48();          generate long (internal seed)
z = jrand48(y);         generate long (external seed)

srand48(z);             set high 32 bits of internal seed
p = seed48(y);          set all 48 bits of internal seed
lcong48(k);             set linear congruence parameters

double x;               double precision random number
short y[3];             48-bit seed supplied by caller
long z;                 long integer random number
short *p;               pointer to internal seed array
short k[7];             linear congruence parameters array
```

Description:

These functions generate pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic. The normal versions (drand48, lrand48, mrand48) utilize an internal 48-bit storage area for the seed value. Special versions (erand48, nrand48, jrand48) are provided for cases where several seeds are in use at the same time, in which case the user provides the seed storage areas.

The drand48 and erand48 functions return values uniformly distributed over the interval from 0.0 up to but not including 1.0.

The lrand48 and nrand48 functions return non-negative long integers uniformly distributed over the interval from 0 to 2**31-1.

The mrand48 and jrand48 functions return signed long integers uniformly distributed over the interval from -2**31 to 2**31-1.

The srand48 and seed48 functions allow initialization of the internal 48-bit seed value to something other than the defaults. For srand48, the specified long value is copied into the high 32 bits of the seed, and the low 16 bits are set to 0x330e. For seed48, the entire 48-bits are loaded from the specified array, and the function returns a pointer to the internal seed array.

The lcong48 function allows a much more intricate initialization of the linear congruential algorithm. The algorithm is of the form:

```
X[n+1] = (a * X[n] + c) mod m
```

where m is 2**48 and the default values for a and c are 0x5deece66d and 0xb, respectively. The array passed to lcong48 contains the value for X[n] in k[0] to k[2], the value for a in k[3] to k[5], and the value for c in k[6]. When seed48 is called, a and c are reset to their original default values.

Cautions:

The functions drand48 and erand48 must be declared double; the functions lrand48, nrand48, mrand48, and jrand48 must be declared long; and the seed48 function must be declared short *.

CEIL/FABS/FLOOR /FMOD/FREXP/LDEXP/MODF

Purpose:

Float conversions

Synopsis:

```
x = ceil(y);          get ceiling integer
x = fabs(y);          get absolute value
x = floor(y);         get floor integer
x = fmod(y,z);        get mod value
x = frexp(y,p);       split into mantissa and exponent
x = ldexp(y,i);       load exponent
x = modf(y,p);        split into integer and fraction

double x;             result
double y,z;           operands
int i;                binary exponent value
double *p;            for return of additional value
```

These functions convert floating point numbers into various other forms.

The floor and ceil functions return the integer values that are just below and just above the specified value, respectively.

The fmod function returns y if z is zero. Otherwise, it returns a value that has the same sign as y, is less than z, and satisfies the relationship

```
y = i * z + x
```

where i is an integer.

The frexp function splits y into its mantissa and exponent parts. The exponent is placed into the area pointed to by p, while the mantissa is returned by the function.

The ldexp function returns y * (2 ** i).

The modf function returns the fractional part of y with the same sign as y and places the integer portion into the area pointed to by p.

Cautions:

These functions must be declared double, which can be accomplished simply by including math_h.

ATOF/ATOI/ATOL

Purpose:

Simple ASCII conversions

Synopsis:

```
x = atof(p);        ASCII to floating point
i = atoi(p);        ASCII to integer
l = atol(p);        ASCII to long integer

double x;           double precision result
int i;              integer result
long l;             long integer result
char *p;            pointer to ASCII string
```

Description:

These functions skip over any leading white space (i.e., blanks, tabs, and newlines) and then perform the appropriate conversion. The conversion stops at the first unrecognized character, and no check is made for overflow.

For atof, the ASCII string may contain a decimal point and may be followed by an e or an E and a signed integer exponent. For all functions, a leading minus sign indicates a negative number. White space is not allowed between the minus sign and the number or between the number and the exponent.

Cautions:

The function atof must be declared double, and the function atol must be declared long.

STRTOL

Purpose:

Convert ASCII to long integer

Synopsis:

r = strtol(s,p,base);

```
long r;          result
char *s;         string to be scanned
char **p;        returns pointer to terminating character
int base;        conversion base
```

Description:

Converts an ASCII string into a long integer, using the specified number base for the conversion. Leading white space (blanks, tabs, and newlines) is skipped, and the conversion proceeds until an unrecognized character is hit. The pointer to the unrecognized character is returned in p. If no conversion can be performed, p will contain s, and the result will be 0.

The conversion base can be in the range from 0 to 36. If it is non-zero, then the ASCII string may contain digit characters from 0 through 9 and from the letter A through as many letters as necessary, with no distinction made between upper and lower case. For example, if base is 13, then the allowable digit characters are 0 through 9 and A,B, and C or a, b, and c. If base is 16, then a leading "0x" or "0X" may appear in the string.

If base is 0, then the leading characters of the string are examined to determine the conversion base. A leading "0" indicates octal conversion (base 8), while a leading "0x" or "0X" indicates hexadecimal conversion (base 16). A leading digit from 1 to 9 indicates decimal conversion (base 10).

ECVT

Purpose:

Convert floating point to ASCII

Synopsis:

p = ecvt(value,ndig,dec,sign);

char *p;	pointer to ASCII string
double value;	value to convert
int ndig;	number of digits in string
int *dec;	returns position of decimal point
int *sign;	non-zero if negative

Description:

Converts the specified value into a null-terminated ASCII string containing the specified number of digits. The integer pointed to by dec will then contain the relative location of the decimal point, with a negative value meaning that the decimal is to the left of the returned digits. The actual decimal point character is not included in the generated string.

Cautions:

The pointer returned points to a static array which is overwritten by each call to ecvt; thus, it should be copied elsewhere if necessary. The function must be declared as returning char *.

MATHERR

Purpose:

Handle math function error

Synopsis:

code = matherr(x);

int code;	non-zero for new return value
struct exception *x;	math exception block

Description:

This function is called whenever one of the other math functions detects an error. Upon entry, it receives the exception block that describes the error in detail. This structure is defined in math_h, as follows:

```
struct exception
{
int type;       error type
char *name;     name of function having error
double arg1;    first argument
double arg2;    second argument
double ret;     proposed return value
};
```

The error type names defined in math_h are:

```
DOMAIN     =>   domain error
SING       =>   singularity
OVERFLOW   =>   overflow
UNDERFLOW  =>   underflow
TLOSS      =>   total loss of significance
PLOSS      =>   partial loss of significance
```

The standard version of matherr supplied in the library places the appropriate error number into the external integer errno, and returns zero. When matherr is called, the function that detected the error will have placed its proposed return value into the exception structure. The zero return code indicates that return value should be used.

Programmers may supply their own version of matherr, if desired. On particular errors, it may be desirable to cause the function detecting the error to return a value other than its usual default. This can be accomplished by storing a new return value in ret of the exception structure, and then returning a non-zero value from matherr, which forces the function to pick up the new value from the exception structure.

5.5 QL Specific Functions

The functions described here provide low-level access to QDOS. Note that the header file qdos_h should be included when using these functions.

QDOS1/QDOS2/QDOS3

Purpose:

QDOS trap functions

Synopsis:

```
e = qdos1(inregs,outregs);
e = qdos2(inregs,outregs);
e = qdos3(inregs,outregs);
int e;                  error code
struct REGS *inregs;    input registers
struct REGS *outregs;   output registers
```

Description:

These functions perform the QDOS traps. qdos1 performs trap #1, qdos2 performs trap #2 and qdos3 performs trap #3. The values of the registers are passed in the structure inregs and returned in the structure outregs (see qdos_h for format). The error code returned is the value of D0 resulting from the trap.

GETCHID

Purpose:

Get QDOS channel ID for level-1 file.

Synopsis:

```
chid = getchid(fd);
char *chid;     QDOS channel ID for specified file
int fd;         Level-1 file decriptor
```

Description:

Returns the QDOS channel ID associated with the specified level-1 file.

FGETCHID

Purpose:

Get QDOS channel ID for level-2 file.

Synopsis:

```
chid = fgetchid(fp);
char *chid;        QDOS channel ID for specified file
int fp;            Level-2 file pointer
```

Description:

Returns the QDOS channel ID associated with the specified level-2 file.

IO__FBYTE

Purpose:

Fetches a byte.

Synopsis:

```
error = io__fbyte(chanid,time,cptr);
int error;         QDOS error code (zero if no error)
char *chanid;      QDOS channel ID
int time;          timeout value
char *cptr;        pointer to character returned
```

Description:

Fetches a byte from the specified channel.

IO__SBYTE

Purpose:

Send a byte

Synopsis:

```
error = io__sbyte(chanid,time,c);
int error;          QDOS error code (zero if no error)
char *chanid;       QDOS channel ID
int time;           timeout value
char c;             character to send
```

Description:

Sends a byte to the specified channel.

IO__PEND

Purpose:

Check for pending input

Synopsis:

```
error = io__pend(chanid,time);
int error;          QDOS error code (zero if no error)
char *chanid;       QDOS channel ID
int time;           timeout value
```

Description:

Checks the specified channel for pending input. No data is actually read.

SD__CURE

Purpose:

Enable cursor

Synopsis:

```
error = sd__cure(chanid,time);
int error;          QDOS error code (zero if no error)
char *chanid;       QDOS channel ID
int time;           timeout value
```

Description:

Enables the cursor in the window defined by the channel ID.

SD__CURS

Purpose:

Suppress cursor

Synopsis:

```
error = sd__curs(chanid,time);
int error;          QDOS error code (zero if no error)
char *chanid;       QDOS channel ID
int time;           timeout value
```

Description:

Suppresses the cursor in the window defined by the channel ID.

SD_POS

Purpose:

Position cursor

Synopsis:

```
error = sd_pos(chanid,time,col,row);
int error;      QDOS error code (zero if no error)
char *chanid;   QDOS channel ID
int time;       timeout value
int col;        column number
int row;        row number
```

Description:

Positions the cursor at an absolute position on the screen using character co-ordinates.

SD_TAB

Purpose:

Tab cursor

Synopsis:

```
error = sd_tab(chanid,time,col);
int error;      QDOS error code (zero if no error)
char *chanid;   QDOS channel ID
int time;       timeout value
int col;        column number
```

Description:

Tabs the cursor to the specified column.

SD__NL

Purpose:

Moves cursor to newline

Synopsis:

```
error = sd__nl(chanid,time);
int error;          QDOS error code (zero if no error)
char *chanid;       QDOS channel ID
int time;           timeout value
```


SD__PCOL

Purpose:

Move cursor to previous column

Synopsis:

```
error = sd__pcol(chanid,time);
int error;          QDOS error code (zero if no error)
char *chanid;       QDOS channel ID
int time;           timeout value
```

SD__NCOL

Purpose:

Move cursor to next column *

Synopsis:

```
error = sd__ncol(chanid,time);
int error;          QDOS error code (zero if no error)
char *chanid;       QDOS channel ID
int time;           timeout value
```

Description:

Moves the cursor one character cell to the right.

SD__PROW

Purpose:

Move cursor to previous row

Synopsis:

```
error = sd__prow(chanid,time);
int error;          QDOS error code (zero if no error)
char *chanid;       QDOS channel ID
int time;           timeout value
```

Description:

Moves the cursor to the character row above its current location.

SD__NROW

Purpose:

Move cursor to next row*

Synopsis:

```
error = sd__nrow(chanid,time);
int error;          QDOS error code (zero if no error)
char *chanid;       QDOS channel ID
int time;           timeout value
```

Description:

Moves the cursor to the character row below its current location.

SD__PIXP

Purpose:

Position cursor using pixel co-ordinates

Synopsis:

```
error = sd__pixp(chanid,time,xcol,xrow);
int error;          QDOS error code (zero if no error)
char *chanid;       QDOS channel ID
int time;           timeout value
int xcol;           pixel column number
int xrow;           pixel row number
```

Description:

Position the cursor at an absolute position on the screen using pixel co-ordinates.

Chapter 6: 68000 Code Generation

Any processor with a sufficiently rich instruction set allows implementation of high-level language constructs in a variety of ways, and the 68000 is no exception. This chapter presents the general strategy used by the Lattice compiler in generating code for the 68000, with a view toward clarifying the machine-dependent aspects of the language, the compiler's choice of machine language instructions, and the interface to user-written assembly language modules.

6.1 Machine Dependencies

The C language definition does not completely specify all aspects of the language; a number of important features are described as machine-dependent. This flexibility in some of the finer details permits the language to be implemented on a variety of machine architectures without forcing code generation sequences that are elegant on one machine and awkward on another. This section describes the machine-dependent features of the language as implemented on the 68000 series. See chapter 4 of the manual for a description of the machine-independent features of the Lattice implementation of the language.

6.1.1 Data Elements

The standard C data types are implemented according to the following descriptions. The only data elements which do not require alignment to a word offset are characters and character arrays; as noted in section 1.2.1, this word alignment can be forced to a long word (four-byte) alignment for all objects larger than two bytes by a compile time option. In all cases, regardless of the length of the data element, the high order (most significant) byte is stored first, followed by successively lower order bytes. This scheme is consistent with the general byte ordering used on the 68000. The following table summarizes the characteristics of the data types:

Type	Length in Bits	Range
char	8	-128 to 127 (ASCII characters)
unsigned char	8	0 to 255
short	16	-32768 to 32767
unsigned short	16	0 to 65535
int	32	-2147483648 to 2147483647
unsigned int	32	0 to 4294967295
long	32	-2147483648 to 2147483647
unsigned long	32	0 to 4294967295
float	32	+/- 10E37 to +/- 10E38
double	64	+/- 10E-307 to +/- 10E308

char defines an 8-bit signed integer. Text characters are generated with bit 7 reset, according to the standard ASCII format.

unsigned char defines an 8-bit unsigned integer.

int
long
long int all define a 32-bit signed integer.

unsigned
unsigned int
unsigned long all define a 32-bit unsigned integer.

short
short int both define a 16-bit signed integer.

unsigned short defines a 16-bit unsigned integer.

float defines a 32-bit signed floating point number, with an 8-bit biased binary exponent, and a 24- bit fractional part which is stored in normalized form without the high-order bit being explicitly represented. The exponent bias is 127. This representation is equivalent to approximately 6 or 7 decimal digits of precision.

double or long float

defines a 64-bit signed floating point number, with an 11-bit biased binary exponent, and a 53- bit fractional part which is stored in normalized form without the high-order bit being explicitly represented. The exponent bias is 1023. This representation is equivalent to approximately 15 or 16 decimal digits of precision.

Pointers to the various data types and to functions are four bytes in length, and contain the absolute address of the first byte of the target object.

The total size of all objects declared within the same storage class is limited according to the particular class, as follows:

Storage Class	Maximum total size of objects declared
extern	1048575
static	1048575
auto	524287
formal	255

6.1.2 External Names

External identifiers may be up to 8 characters in length in the default case, if the -n option is used on LC1, they may be as long as 31 characters. Upper and lower case letters are distinct, i.e., ca-e is significant. A user may define external objects with any name that does not conflict with the following classes of identifiers:

_******* Certain library functions and data elements (defined in modules written in C) are defined with an initial underscore.

CX*** Run-time support functions (written in assembly language) which implement C language features such as long integer multiply and divide, floating point arithmetic, and the like are defined with CX as the first two characters.

The likelihood of collision with library definitions is remote, but users should be aware of these conventions and avoid applying these types of identifiers to external, user-defined functions and data.

6.1.3 Arithmetic Operations and Conversions

Arithmetic operations for the integral types (floating type operations are discussed in the next section) are generally performed by in-line code. Integer overflows are ignored in all cases, although signed comparisons correctly include overflow in determining the relative size of operands. Short integer division by zero generates a trap; long integer division by zero simply generates a result of zero. Division of negative integers causes truncation toward zero, just as it does for positive integers, and the remainder has the same sign as the dividend. Right shifts are arithmetic, that is, the sign bit is copied into vacated bit positions, unless the operand being shifted is unsigned; in that case, a logical (zero-fill) right shift is performed.

Function calls to library routines are generated only for long integer multiplication and division (both signed and unsigned).

Conversions are generated according to the "usual arithmetic conversions" described in Kernighan and Ritchie, and are generally trouble free. The following points should be noted:

1 char objects may be signed or unsigned in this implementation. Thus, sign extension may or may not be per formed during expansion to int. Note that all char declarations may be forced to be interpreted as unsigned char by means of a compile time option; see section 1.2.1.

2 Conversion of short to long causes sign extension, while conversion of unsigned short to long does not. The inverse operations simply truncate the result, which is undefined if its absolute value is too large to be represented.

3 Expansion of char and short operands to int may not be performed by the compiler if those operands only participate in operations with other operands of the same type, resulting in increased efficiency for sequences like

```
char a, b, c;
. . .
a = b + c;
```

Note that expansion is, however, always performed for function call arguments.

4 Conversions from integral to floating types are fairly straightforward. The inverse conversions cause any fractional part to be dropped.

5 Conversion from float to double is well-defined, but the inverse operation may cause an underflow or overflow condition since double has a much larger exponent range. Considerable precision is also lost, though the fraction is rounded to its nearest float equivalent.

6 In general, the presence of any unsigned type operand in an expression causes the result also to be unsigned.

6.1.4 Floating Point Operations

In accordance with the language definition, all floating point arithmetic operations are performed using double precision operands, and all function arguments of type float are converted to type double before the function is called. The formats used are identical to the 32-bit and 64 bit formats defined by the proposed IEEE standard for floating point representations. Legal floating point operations include simple assignment, conversion to other arithmetic types, unary minus (change sign), addition, subtraction, multiplication, division, and comparison for equality or relative size. Note that, in contrast to the signed integer representations, negative floating point values are not represented in two's complement notation; positive and negative numbers differ only in the sign bit. This means that two kinds of zero are possible: positive and negative. All floating point operations treat either value as true zero and generally produce positive zero, whenever possible. Note that code which checks float or double objects for zero by type punning (that is, examining the objects as if they were int or some other integral type) may assume (falsely) negative zero to be not zero.

Floating point arithmetic and comparison operations are performed by generating calls to library routines written in assembly language. Floating point exceptions are processed by a library function called CXFERR that is called according to the following convention:

```
CXFERR(errno);
int errno;
```

where errno can be

```
1 = underflow
2 = overflow
3 = divide by zero
```

The standard version of CXFERR supplied in the libraries simply ignores all error conditions. A different version can be written (in either C or assembly language) to print out an error message and terminate processing, or take any other action. If CXFERR returns to the library function which called it, each exception is processed as follows:

```
Underflow     Sets the result equal to zero.
Overflow      Sets the result to plus or minus
                infinity.
Zerodivide    Sets the result equal to zero.
```

6.1.5 Bit Fields

Bit fields are fetched on a long word basis, that is, the entire word containing the desired bit field is loaded (or stored) even if the field is 16 bits or less in size. Bit fields are assigned from left to right within a machine word; the maximum field size is 31 bits. Bit fields are considered unsigned in this implementation; sign extension is not performed when the value of a field is expanded in an arithmetic expression. If a structure is declared

```
struct {
     unsigned x : 20;
     unsigned y : 9;
     unsigned z : 2;
     } a;
```

then a occupies a single 32-bit word, a.x resides in bits 31 through 12, a.y in bits 11 through 3, and a.z in bits 2 and 1. Bit fields of only a single bit are tested and assigned constant values using the BTST, BSET, or BCLR instructions.

6.1.6 Register Variables

A register variable declaration may be accepted for any pointer or other data object with a size of no more than 4 bytes. Up to four pointers may be assigned to address registers starting with A5 down through A2; up to four simple data elements may be assigned to data registers starting with D7 down through D4. The registers are assigned in the same order in which they appear in the function declaration, with formal parameters being assigned first. Naturally, if A5 is used as a frame pointer via the -f option described in section 1.2.2, it is not available for use as a register variable.

The use of register variables affects the entry sequence at the start of the function in which they are declared, by requiring an additional instruction to save the previous registers' values before they are used in the function. See section 6.3.3 for more information.

6.2 General Code Generation Strategies

When the code for a function is buffered in memory before being written to the object file, branch instructions are not explicitly represented in the function image. Instead, they are represented by special structures denoting the type and target of each branch. When the function has been completely defined, the branch instructions are analyzed and several important optimizations are performed:

1 Any branch instruction that passes control directly to another branch instruction is re-routed to branch directly to the target location.

2 A conditional branch instruction that branches over a single unconditional branch is replaced by a single conditional branch instruction of the opposite sense.

3 Sections of code into which control does not flow are detected and discarded.

4 Each branch instruction is coded in the smallest possible machine language sequence required to reach the target location.

Most of these optimizations are applied iteratively until no further improvement is obtained.

The code generator also makes a special effort to generate efficient code for the switch statement. Three different code sequences may be produced, depending on the number and range of the case values.

1 If the number of cases is three or fewer, control is routed to the case entries by a series of test and branch instructions.

2 If the case values are all positive and the difference between the maximum and minimum case values is less than twice the number of cases, the compiler generates a branch table which is directly indexed by the switch value. The value is adjusted, if necessary, by the minimum case value and compared against the size of the table before indexing. This construction requires minimal execution time and a table no longer than that required for the type of sequence described in No. 3.

3 Otherwise, the compiler generates a table of {case value, branch address} pairs, which is linearly searched for the switch value.

All of the above sequences are generated in-line without function calls because the number of instruction bytes is small enough that little benefit would be gained by implementing them as library functions.

Aside from these special control flow analyses, the compiler does not perform any global data flow analysis or any loop optimizations. Thus, values in registers (except for register variables) are not preserved across regions into which control may be directed. The compiler does, however, retain information about register contents after conditional branches which cause control to leave a region of code. Throughout each section of code into which control cannot branch (although it may exit via conditional branches), values which are loaded into registers are retained as long as possible so as to avoid redundant load and store operations. The allocation of registers is guided by next-use information, obtained by analysis of the local block of code, which indicates which operands will be used again in subsequent operations.

This information also assists the compiler, in analyzing binary operations, in its decision whether to load both operands into registers or to load one operand and use a memory reference to the other. Generally, the result of such an operation will be computed in a register, but sequences like

```
i += j;
```

will load the value of j into a register and compute the result directly into the memory location for i (but only if i is not used later in the same local block of code).

The hardware registers D0 through D7 are used as general purpose accumulators, while A0 through A4 (and A5, if not used as a frame pointer) are used for pointer values, allowing access to indirect operands. Either A5 or A6 is used to address the current stack frame; see section 6.3.3 for more information. The use of registers for register variables is described in section 6.1.6.

6.3 Run-time Program Structure

This section describes the run-time environment which is implicitly assumed by the 68000 code generator and its effect on the interface between C and assembly language. Some knowledge of the architecture of the 68000 processor and of basic object code and linkage concepts is required in order to understand much of the information presented.

The C programming language provides for three basic kinds of memory allocation: the instructions which make up the executable functions, the static data items which persist independently of any of the functions which refer to them, and the automatic data items which exist only while a function is invoked. Because the 68000 processor has a linear address space, no special assumptions about the location of any of these components are needed. Thus, in the general case, functions and data may be placed anywhere in memory because of the following conventions:

1 All function calls are generated using a JSR instruction with a 32-bit absolute address.

2 All static and external data elements are accessed via 32- bit absolute addresses.

3 All automatic data elements are allocated and accessed relative to address register A7; if the offset of a data element exceeds the directly addressable range of 32K bytes, it is accessed by transferring the frame pointer register to another address register and adding in the offset value.

Note that no intialization of A7 and no stack overflow checking is performed by the generated code. The initialization routine, STARTUP supplies an initial value for A7 that will avoid possible collision with the memory occupied by functions or static data elements.

If the -r option on LC2 is used, function calls are generated using a JSR instruction with a 16-bit PC-relative offset. Thus, no function call can reach farther than 32767 bytes above or below itself. If the -b option of LC1 is used, static and external data elements are accessed using 16-bit offsets from an address register (A5 or A6). This limits the size of static and external data to a maximum of 65535 bytes. (Note that the linker uses 0x8000 for its initial offset, allowing the full 64K range and requiring the address register to contain an address 32K bytes greater than the actual base.)

These rather severe restrictions are necessary in order to produce true position-independent code. In addition, note that pointers cannot be initialized by static declarations such as

```
char *p = "string";
```

if position-independent code is desired; at compile time, the only way to initialize such an object is to generate a relocatable address reference that will result in an absolute address at run time. This problem illustrates another limitation of position-independent C code on the 68000: once the address of an object is computed and assigned to a pointer, the target object may not be moved to another location without destroying the validity of such a pointer.

6.3.1 Object Code Conventions

The object file created by the second phase of the compiler is in Sinclair relocatable object format, and defines the instructions and data

necessary to implement the module specified by the C source file; it also contains relocation and linkage information necessary to guarantee that the components will be addressed properly when the module is executed or referenced as part of a linked program. While the addressing conventions used by the code generator permit data and functions to be scattered randomly throughout memory, it is usual to force functions and data to be collected together at link time into two contiguous blocks. The object module produced by the compiler is designed to facilitate this grouping by placing functions and data into two separate named control sections. At link time, all elements in the same control section are placed in contiguous regions of memory.

"text" is the control section which includes the instructions which perform the actions specified by any functions defined in the source file.

"data" is the control section which includes all explicitly initialized static data items which are defined in the source file, and "udata" is the logical control section which specifies the size of all uninitialized static data items. Static data in this sense includes not only those data items explicitly declared static but also items declared outside the body of a function without an explicit storage class identifier. String constants are considered initialized static data, and are placed in the "data" section. (Note that automatic data items are simply allocated on the stack at run time and are not explicitly defined in the object file.)

The net effect of these control section assignments is to force, at link time, all functions to be collected together and all static data items to be similarly combined. There are two advantages to this structure. In the event of a program error which addresses an array out of range, the effect is usually less catastrophic if it is only data (not instructions) which are destroyed. In addition, some processors may support memory management hardware which will allow protection or mapping of contiguous portions of memory; separating program and data portions of a program facilitates use of such capabilities.

6.3.2 Linkage Conventions

As noted in section 6.1.2, external identifiers may be up to 31 characters in length, depending on whether the -n option was used when the module was compiled. The relocation information in the object file defines all external names as an unspecified type, that is, there is no set of attributes associated with the name; it is simply an address within the

memory defined by the final load module. It is therefore an error to define two items with the same external name in the same program. It is the programmer's responsibility to prevent this occurrence, and also to make sure that modules refer to external names in a consistent way (i.e., a function should not refer to "xyz" as short when it is actually defined as int in some other module). External definition and reference from assembly language modules are discussed in section 6.3.4.

In addition to collecting instructions and data into separate sections, the 68000 load module constructed by linking object modules must place either the "text" section or the "data" section lower in memory within the load module. Which of these alternatives is most desirable will depend on the application. In general, the default arrangement places the "text" section below the "data" section. The "udata" section is usually placed immediately above the "data" section.

6.3.3 Function Call Conventions

When a C function makes a call to another function, it first pushes the values of any arguments onto the stack and then makes a call to that function. For external functions, a JSR instruction with an absolute long address is normally used; compiling with the -b option causes the compiler to use a JSR instruction with a 16-bit PC-relative address; both forms are resolved at link time. For functions defined in the same module, a BSR instruction is used, which is resolved at compile time. The arguments are pushed in right-to-left order because the stack grows downward on the 68000; this allows the called function to address the arguments in the natural left-to-right (low-address to-high-address) order. Note that the C language definition requires all char and short arguments to be expanded to int, so that a minimum of four bytes is pushed for each argument. The first actions taken by the called function are as follows:

1 The value of the frame pointer register (either A5 or A6) is saved on the stack, the stack pointer is transferred to the frame pointer register, and the stack pointer is then adjusted downward by the number of bytes of stack space required by the called function (this local storage is also called a stack frame). This sequence of operations is accomplished by a simple LINK instruction, if the needed storage is less than 32K bytes; or by explicit instructions to push the frame pointer register, transfer to it the stack pointer, and allocate the needed storage, if more than 32K bytes. The

frame pointer is either A5 or A6, depending on the -f option used to compile the function (see section 1.2.2).

2 If any of the registers D4-D7 or A2-A5 are used in the function, a MOVEM.L instruction is used to save their current values on the stack. These registers may be in use by the calling program as register variables; see section 6.1.6.

3 If the -p option was used to compile the function, a special instruction called a stack probe will be generated: TST.B -144,A7. This instruction is used by some operating systems (such as Xenix) to ensure that sufficient memory has been allocated for the run-time stack.

The offsets of the various components on the stack are indicated by the following diagram.

```
High    |-------------------|
        |     arguments     |
        |        ...        |
        |-------------------|
        |   return address  |
        |-------------------|
        |  caller's saved FP|
        |-------------------|  <- FP (A5 or A6)
        |                   |
        |   auto data items |
        |                   |
        |-------------------|
        |    temporaries    |
Low     |-------------------|  <- SP (A7)
```

Function arguments are addressed positively from the frame pointer, while the auto data elements are addressed negatively. Temporaries (used for storage of reusable intermediate expression results) are placed below all of the auto data items but are addressed positively from A7. Only as much storage as is actually needed is allocated for temporaries. Note that the first 32K of automatic data is addressed much more efficiently than any subsequent elements; thus, it is advantageous for functions to use no more than 32K bytes of local storage.

When a function returns to its caller, it first loads the function return value, if any, into predefined registers. The size of the value returned determines the register(s) used:

```
8 bits    D0.B (low byte of D0, char functions only)
16 bits   D0.W (low word of D0, short functions
               only)
32 bits   D0.L (all other integral types and
               pointers)
64 bits   (D0.L, D1.L) (double functions only)
```

For double precision values, the high order bits are contained in D0. Note that functions returning aggregates (structures or unions) actually return a pointer to a static copy of the aggregate. Because this copy persists only long enough to assign the return value, such functions are nonetheless recursively reentrant (but not multi-tasking reentrant).

After the return value is loaded, the function restores any of the registers D4-D7 or A2-A5 which were saved on entry, by another MOVEM.L instruction. The UNLK instruction is then used to restore both the frame pointer register and the stack pointer, and an RTS is executed to return to the calling function. As is customary in C, that function is responsible for de-allocating any stack space used to push arguments. An ADDQ.L instruction is used if four or eight bytes of arguments were pushed; otherwise, an instruction like

```
LEA  A7,12(A7)
```

is used to restore A7 if more than eight bytes of arguments were pushed on the stack. (Note: LEA is used instead of ADDA.W because it is faster by four clock cycles.)

6.3.4 Assembly Language Interface

Programmers may write assembly language modules for inclusion in C programs, provided that these modules adhere to the object code, linkage, and function call conventions described in the preceding sections. An assembly language module which defines one or more functions to be called from C should observe the following:

1 Each function entry must be declared in an XDEF statement in order to be accessible in the C modules:

```
XDEF    AFUNC
.
.
.
.AFUNC          ; start of function
```

2 Each data element must be declared in an XDEF statement in order to be accessible in the C modules:

```
XDEF    DX,DY,DZ
DX      DC.L    $4000
DY      DC.W    $8000
DZ      DC.L    DX
```

3 Any of the registers D4-D7 or A2-A6 must be preserved by the module, and the return value loaded into the appropriate data registers.

To call a C function from an assembly language module, an XREF declaration for the function must be included. Before calling the function (via JSR), the caller must supply any expected arguments in the proper order (see section 6.3.3). After control returns from the called function, the stack pointer must be adjusted by the caller to account for pushed arguments.

```
XREF    cfunc
MOVE.L  D0,-(A7)  ; push argument
MOVE.L  D1,-(A7)
JSR     cfunc     ; call function
ADDQ    #8,A7     ; restore stack pointer
```

Data elements defined in a C module may be accessed via XREF statements, as well:


```
XREF    XD2,XD3
.
.
.
MOVE.L  XD2,D0
```

The following example functions illustrate some of the requirements discussed above.

```
XDEF    inp,outp
;
; c = inp(ioaddr);  returns byte from specified
;                   I/O address
; char c;           returned byte
; char *ioaddr;     I/O address
;
inp     MOVE.L  4(A7),A0    fetch argument
        MOVE.B  (A0),D0     fetch byte
        RTS
;
; outp(c, ioaddr); writes byte to specified
;                  I/O address
; char c;          byte to be written
; char *ioaddr;    I/O address
;
outp    MOVE.L  4(A7),A0    fetch first arg
        MOVE.L  8(A7),D0    fetch second arg
        MOVE.B  D0,(A0)     write byte
        RTS
        END
```

Appendix A: Error Messages

This appendix describes the various messages produced by the first and second phases of the compiler. Error messages which begin with the text CXERR are compiler errors which are described in Appendix B.

A.1 Unnumbered Messages

These messages describe error conditions in the environment, rather than errors in the source file due to improper language specifications.

`Can't create object file`

The second phase of the compiler was unable to create the __O file. This error usually results from a full directory on the output disk.

`Can't create quad file`

The first phase of the compiler was unable to create the __Q file. This error usually results from a full directory on the output disk.

`Can't open quad file`

The second phase of the compiler was unable to open the __Q file specified on the LC2 command, usually because it did not exist on the specified drive.

`Can't open source file`

The first phase of the compiler was unable to open the __C file specified on the LC1 command, usually because it did not exist on the specified drive.

`Combined output file name too large`

The output file name constructed by LC1 or LC2 by combining the source or quad file name with the text specified using the -o option exceeded the maximum file name size of 64 bytes.

File name missing

A file name was not specified on the LC1 or LC2 command.

Intermediate file error

The first phase of the compiler encountered an error when writing to the __Q file. This error usually results from an out-of-space condition on the output disk.

Invalid command line option

An invalid command line option (beginning with a -) was specified on either the LC1 or the LC2 command. See sections 1.2.1 and 1.2.2 for valid command line options. The option is ignored, but the compilation is not otherwise affected. In other words, this error is not fatal.

Invalid symbol definition

The name attached to a -d specifying a symbol to be defined was not a valid C identifier or was followed by text which did not begin with an equal sign.

No functions or data defined

A source file which did not define any functions or data elements was processed by the computer. This error always terminates execution of the compiler. It can be generated by forgetting to terminate a comment, which then causes the compiler to treat the entire file as a comment.

Not enough memory

This message is generated when either phase of the compiler uses up all the available working memory. The only cure for this error is either to increase the available memory on the system, or (if the maximum is already available) reduce the size and complexity of the source file. Particularly large functions will generate this error regardless of how much memory is available; break the task into smaller functions if this occurs.

Object file error

The second phase of the compiler encountered an error when writing to the __O file. This error usually results from an out-of-space condition on the output disk.

Parameters beyond file name ignored

Additional information was present on the command line beyond the name of the source or quad file to be compiled. The compiler option flags must be specified before the name of the file to be compiled.

Unrecognised -c option

One of the characters following the -c option on LC1ally acceptable constructions but do not prevent the creation of the quad file. See section 1.5.3 for more information about error processing.

1 This error is generated by a variety of conditions in connection with pre-processor commands, including specifying an unrecognized command, failure to include white space between command elements, or use of an illegal pre-processor symbol.

2 The end of an input file was encountered when the compiler expected more data. This may occur on an #include file or the original source file. In many cases, correction of a previous error will eliminate this one.

3 The file name specified on an #include command was not found.

4 An unrecognized element was encountered in the input file that could not be classified as any of the valid lexical constructs (such as an identifier or one of the valid expression operators). This may occur if control characters or other illegal characters were detected in the source file. This may also occur if a pre-processor command was specified with the # not in the first position of an input line.

5 A pre-processor #define macro was used with the wrong number of arguments.

6 Expansion of a #define macro caused the compiler's line buffer to overflow. This may occur if more than one lengthy macro appeared on a single input line.

7 The maximum extent of #include file nesting was exceeded; the compiler supports #include nesting to a maximum depth of 4.

8 A cast (type conversion) operator was incorrectly specified in an expression.

9 The named identifier was undefined in the context in which it appeared, that is, it had not been previously declared. This message is only generated once; subsequent encounters with the identifier assume that it is of type int (which may cause other errors).

10 An error was detected in the expression following the [character (presumably a subscript expression). This may occur if the expression in brackets was null (not present).

11 The length of a string constant exceeded the maximum allowed by the compiler (256 bytes). This will occur if the closing " (double quote) was omitted in specifying the string.

12 The expression preceding the . (period) or -> structure reference operator was not recognizable by the compiler as a structure or pointer to a structure. This may occur even for constructions which are accepted by other compilers; see section 4.1.

13 An identifier indicating the desired aggregate member was not found following the . (period) or -> operator.

14 The indicated identifier was not a member of the structure or union to which the . (period) or -> referred.

15 The identifier preceding the (function call operator was not implicitly or explicitly declared as a function.

16 A function argument expression specified following the (function call operator was invalid. This may occur if an argument expression was omitted.

17 During expression evaluation, the end of an expression was encountered but more than one operand was still awaiting evaluation. This may occur if an expression contained an incorrectly specified operation.

18 During expression evaluation, the end of an expression was encountered but an operator was still pending evaluation. This may occur if an operand was omitted for a binary operation.

19 The number of opening and closing parentheses in an expression was not equal. This error message may also occur if a macro was poorly specified or improperly used.

20 An expression which did not evaluate to a constant was encountered in a context which required a constant result. This may occur if one of the operators not valid for constant expressions was present (see Kernighan and Ritchie, Appendix A, p. 211).

21 An identifier declared as a structure, union, or function was encountered in an expression without being properly qualified (by a structure reference or function call operator).

22 (non-fatal warning) An identifier declared as a structure or union appeared as a function argument without the preceding & operator. Expression evaluation continues with the & assumed (i.e., a pointer to the aggregate is generated).

23 The conditional operator was used erroneously. This may occur if the ? operator was present but the : was not found when expected.

24 The context of the expression required an operand to be a pointer. This may occur if the expression following * did not evaluate to a pointer.

25 The context of the expression required an operand to be an lvalue. This may occur if the expression following & was not an lvalue, or if the left side of an assignment expression was not an lvalue.

26 The context of the expression required an operand to be arithmetic (not a pointer, function, or aggregate).

27 The context of the expression required an operand to be either arithmetic or a pointer. This may occur for the logical OR and logical AND operators.

28 During expression evaluation, the end of an expression was encountered but not enough operands were available for evaluation. This may occur if a binary operation was improperly specified.

29 An operation was specified which was invalid for pointer operands (such as one of the arithmetic operations other than addition).

30 (non-fatal warning) In an assignment statement defining a value for a pointer variable, the expression on the right side of the = operator did not evaluate to a pointer of the exact same type as the pointer variable being assigned, i.e., it did not point to the same type of object. The warning also occurs when a pointer of any type is assigned to an arithmetic object. See section 4.1 for an explanation of the philosophy behind this warning. Note that the same message becomes a fatal error if generated for an initializer expression.

31 The context of an expression required an operand to be integral, i.e., one of the integer types (char, int, short, unsigned, or long).

32 The expression specifying the type name for a cast (conversion) operation or a sizeof expression was invalid. See Kernighan and Ritchie, Appendix A, pp. 199-200 for the valid syntax.

33 An attempt was made to attach an initializer expression to a structure, union, or array that was declared auto. Such initializations are expressly disallowed by the language.

34 The expression used to initialize an object was invalid. This may occur for a variety of reasons, including failure to separate elements in an initializer list with commas or specification of an expression which did not evaluate to a constant. This may require some experimentation to determine the exact cause of the error.

35 During processing of an initializer list or a structure or union member declaration list, the compiler expected a closing right brace, but did not find it. This may occur if too many elements were specified in an initializer expression list or if a structure member was improperly declared.

36 A statement within the body of a switch statement was not preceded by a case or default prefix which would allow control to reach that statement. This may occur if a break or return statement is followed by any other statement without an intervening case or default prefix.

37 The specified statement label was encountered more than once during processing of the current function.

38 In a body of compound statements, the number of opening left braces { and closing right braces } was not equal. This may occur if the compiler got "out of phase" due to a previous error.

39 One of the C language reserved words appeared in an invalid context (e.g., as a variable name). See Kernighan and Ritchie for a list of the reserved words (p. 180). Note that entry is reserved although it is not implemented in the compiler.

40 A break statement was detected that was not within the scope of a while, do, for, or switch statement. This may occur due to an error in a preceding statement.

41 A case prefix was encountered outside the scope of a switch statement. This may occur due to an error in a preceding statement.

42 The expression defining a case value did not evaluate to an int constant.

43 A case prefix was encountered which defined a constant value already used in a previous case prefix within the same switch statement.

44 A continue statement was detected that was not within the scope of a while, do, or for loop. This may occur due to an error in a preceding statement.

45 A default prefix was encountered outside the scope of a switch statement. This may occur due to an error in a preceding statement.

46 A default prefix was encountered within the scope of a switch statement in which a preceding default prefix had already been encountered.

47 Following the body of a do statement, the while clause was expected but not found. This may occur due to an error within the body of the do statement.

48 The expression defining the looping condition in a while or do loop was null (not present). Indefinite loops must supply the constant 1, if that is what is intended.

49 An else keyword was detected that was not within the scope of a preceding if statement. This may occur due to an error in a preceding statement.

50 A statement label following the goto keyword was expected but not found.

51 The indicated identifier, which appeared in a goto statement as a statement label, was already defined as a variable within the scope of the current function.

52 The expression following the if keyword was null (not present).

53 The expression following the return keyword could not be legally converted to the type of the value returned by the function. This may be generated if the expression specified a structure, union, or function.

54 The expression defining the value for a switch statement did not define an int value or a value that could be legally converted to int.

55 (non-fatal warning) The statement defining the body of a switch statement did not contain at least one case prefix.

56 The compiler expected but did not find a colon (:). This error message may be generated if a case expression was improperly specified, or if the colon was simply omitted following a label or prefix to a statement.

57 The compiler expected but did not find a semi-colon (;). This error generally means that the compiler completed the processing of an expression but did not find the statement terminator (;). This may occur if too many closing parentheses were included or if an expression was otherwise incorrectly formed. Because the compiler scans through white space to look for the semi-colon, the line number for this error message may be subsequent to the actual line where a semi-colon was needed.

58 A parenthesis required by the syntax of the current statement was expected but was not found (as in a while or for loop). This may occur if the enclosed expression was incorrectly specified, causing the compiler to end the expression early.

59 In processing external data or function definitions, a storage class invalid for that declaration context (such as auto or register) was encountered. This may occur if, due to preceding errors, the compiler began processing portions of the body of a function as if they were external definitions.

60 The types of the aggregates involved in an assignment or conditional operation were not exactly the same. This error may also be generated for enum objects.

61 The indicated structure or union tag was not previously defined; that is, the members of the aggregate were unknown. Note that a reference to an undefined tag is permitted if the object being declared is a pointer, but not if it is an actual instance of an aggregate. This message may be issued as a warning after the entire source file have been processed if a pointer was declared with a tag that was never defined.

62 A structure or union tag has been detected in the opposite usage from which it was originally declared (i.e., a tag originally applied to a struct has appeared on an aggregate with the union specifier). The Lattice compiler defines only one class of identifiers for both structure and union tags.

63 The indicated identifier has been declared more than once within the same scope. This error may be generated due to a preceding error, but is generally the result of improper declarations.

64 A declaration of the members of a structure or union did not contain at least one member name.

65 An attempt was made to define a function body when the compiler was not processing external definitions. This may occur if a preceding error caused the compiler to "get out of phase" with respect to declarations in the source file.

66 The expression defining the size of a subscript in an array declaration did not evaluate to a positive int constant. This may also occur if a zero length was specified for an inner (i.e., not the leftmost) subscript.

67 A declaration specified an illegal object as defined by this version of C. Illegal objects include functions which return aggregates (arrays, structures, or unions) and arrays of functions.

68 A structure or union declaration included an object declared as a function. This is illegal, although an aggregate may contain a pointer to a function.

69 The structure or union whose declaration was just processed contains an instance of itself, which is illegal. This may be generated if the * is forgotten on a structure pointer declaration, or if, (due to some intertwining of structure definitions) the structure actually contains an instance of itself.

70 A function's formal parameter was declared illegally; that is, it was declared as a structure, union, or function. The compiler does not automatically convert such references to pointers.

71 A variable was declared before the opening brace of a function, but it did not appear in the list of formal names enclosed in parentheses following the function name.

72 An external item has been declared with attributes which conflict with a previous declaration. This may occur if a function was used earlier, as an implicit int function, and was then

declared as returning some other kind of value. Functions which return a value other than int must be declared before they are used so that the compiler is aware of the type of the function value.

73 In processing the declaration of objects, the compiler expected to find another line of declarations but did not, in fact, find one. This error may be generated if a preceding error caused the compiler to "get out of phase" with respect to declarations.

74 (non-fatal warning) A string constant used as an intialiser for a char array defined more characters than the specified array length. Only as many characters as are needed to define the entire array are taken from the first characters of the string constant.

75 An attempt was made to define the same function more than once within the same source module.

76 The compiler expected, but did not find, an opening left brace in the current context. This may occur if the opening brace was omitted on a list of initializer expressions for an aggregate.

77 In processing a declaration, the compiler expected to find an identifier which was to be declared. This may occur if the prefixes to an identifier in a declaration (parentheses and asterisks) are improperly specified, or if a sequence of declarations is listed incorrectly.

78 The indicated statement label was referred to in the most recent function in a goto statement, but no definition of the label was found in that function.

79 (non-fatal warning) More than one identifier within the list for an enumeration type had the same value. While this is not technically an error, it is usually of questionable value.

80 The number of bits specified for a bit field was invalid. Note that the compiler does not accept bit fields which are exactly the length of a machine word (such as 16 on a 16-bit machine); these must be declared as ordinary int or unsigned variables.

81 The current input line contained a reference to a pre-processor symbol which was defined with a circular definition, or loop. See section 4.2.1 for an example.

82 The size of an object exceeded the maximum legal size for objects in its storage class; or, the last object declared caused the total size of declared objects for that storage class to exceed that maximum.

83 (non-fatal warning) An indirect pointer reference (usually a subscripted expression) used an address beyond the size of the object used as a base for the address calculation. This generally occurs when an element beyond the end of an array is referred to.

84 (non-fatal warning) A #define statement was encountered for an already defined symbol. As noted in section 4.2.1, the second definition takes precedence, but requires an additional #undef statement before the symbol is truly undefined.

85 (non-fatal warning) The expression specifying the value to be returned by a function was not of the same type as the function itself. The value specified is automatically converted to the appropriate type; the warning merely serves as notification of the conversion. The warning can be eliminated by using a cast operator to force the return value to the function type.

86 (non-fatal warning) The types of the formal parameters declared in the actual definition of a function did not agree with those of a preceding declaration of that function with argument type specifiers.

87 (non-fatal warning) The number of function arguments supplied to a function did not agree with the number of arguments in its declaration using argument type specifiers.

88 (non-fatal warning) The type of a function argument expression did not agreement with its corresponding type declared in the list of argument type specifiers for that function. Note that the compiler does not automatically convert the expression to the specified type; it merely issues this warning.

89 (non-fatal warning) The type of a constant expression used as a function argument did not agree with its corresponding type declared in the list of argument type specifiers for that function. In this case, the compiler does convert the expression to the expected type.

90 The type specifier for an argument type in a function declaration was incorrectly formed. Argument type specifiers are formed according to the rules for type names in cast operators or sizeof expressions.

91 One of the operands in an expression was of type void; this is expressly disallowed, since void represents no value.

92 (non-fatal warning) An expression statement did not cause either an assignment or a function call to take place. Such a statement serves no useful purpose, and can be eliminated; usually, this error is generated for incorrectly specified expressions in which an assignment operator was omitted or mistyped.

93 (non-fatal warning) An object with local scope was declared but never referenced within that scope. This warning is provided as a convenience to warn of declarations that may no longer be needed (if, for example, the code in which the variable was used was eliminated but not its declaration). It may also occur if the only use of the object is confined to statements which are not compiled because of conditional compilation directives (#if, etc).

94 (non-fatal warning) An auto variable was used in an expression without having been previously initialised by an assignment statement or appearing in a function argument list with a preceding & (ie, its address passed to a function). Note that the compiler considers the variable initialised once any statement causes it to be initialised, even though control may not flow from that statement to other subsequent uses of the variable. Note also that this warning will be issued if the third expression in a for statement uses a variable which has not yet been initialised, which may be incorrect if that variable is initialised inside the body of the for statement.

Appendix B: Compiler Errors

This appendix describes the procedure to be used for reporting compiler errors. These are errors that result not from the user's incorrect specifications but from the compiler itself failing to operate properly. There are five general kinds of errors which can occur:

1 The compiler generates an error message for a source module which is actually correct.

2 The compiler fails to generate an error message for an incorrect source module.

3 The compiler detects an internal error condition and generates an error message of the form

```
CXERR: nn
```

where nn is an internal error number.

4 The compiler dies mysteriously (crashes) while compiling a source module.

5 The compiler generates incorrect code for a correct source module.

The last type of error is, of course, the most difficult to determine and the most vexing for the programmer, who has no indication that anything is wrong until something inexplicably doesn't work, and who only concludes that the compiler is at fault after a long and painstaking study of his or her own code.

Both Lattice and Metacomco are anxious to know about and repair any compiler errors as quickly as possible, so please report any problems promptly. The difficulties encountered may be spared the next

programmer if this is done. In order to maintain a more precise record of the bugs that are discovered, all problems should be reported in writing to:

```
Metacomco                Tenchstar Inc
26 Portland Square       5353E Scotts Valley Road
Bristol BS2 8RZ          Scotts Valley, CA 95066
UK.                      USA
```

In all cases, include the following items of information:

1 A listing of the source module for which the error occurred. Don't forget to include listings of any #include files used (and watch out for #include file nesting; don't forget the inner files as well).

2 The revision number of the compiler, when it was purchased and the registration number.

3 Your name and address and, if possible, a telephone number with information about the best time to call.

4 A description of the problem, along with any other information which may be helpful such as the results of your investigation into the problem. Obviously, errors of type 3 (see above) don't need anything more than a terse "Causes CXERR 23."

Appendix C: Linker Errors

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

C.1 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.

C.2 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 previous 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.

C.3 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.

C.4 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 - 3E 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.

C.5 Operating System Errors

When the linker gets an error code from QDOS 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 D: Example Programs

Example Program 1

```
/*
This example program prints a table of temperature
conversions from Celsius to Fahrenheit
*/

main()
{
  int lower,upper,step;
  float fahr,celsius;

  lower = -40;
  upper = 140;
  step = 20;


  /* print a heading */
  printf("Celsius   Fahrenheit\n\n");


  celsius = lower;
  while (celsius <= upper)
  {
    fahr = (celsius * 9.0/5.0) + 32.0;
    /* print the conversions */
    printf("  %4.0f      %6.1f\n",celsius,fahr);
    celsius += step;
  }
}
```

Example Program 2

```
/*
This example program is based on the
"Sieve of Eratosthenes."
*/
#include "mdvl_stdio_h"
#define TRUE 1
#define FALSE 0
#define SIZE 8190
#define SIZEPL 8191

char flags [SIZEPL];
int i,prime,k,count,iter;

main ()
{
  printf ("Hit return to do 10 iterations: /n");
  getchar ();

  for (iter = 1; iter <= 10; iter++)
  {
    count = 0;
    for (i = 0; i <= SIZE; i++)
      flags [i] = TRUE;
    for (i = 0; i <= SIZE; i++)
    {
      if (flags [i])
      {
        prime = i + i + 3;
        k = i + prime;
        while (k <= SIZE)
        {
          flags [k] = FALSE;
          k += prime;
        }
        count++;
      }
    }
  }
  printf ("\n%d primes.\n",count);
}
```


Example Program 3

```
/*
This example program merges upto twenty files into one.
Use CRUN to execute the resulting program file. The
last argument specifies the output file, and the other
arguments specify the input files.
*/

#include "mdvl_stdio_h"

main (argc,argv)
int argc;
char *argv[]
{
  int i, c;
  FILE  *fpin, *fpout, *fopen();

  if  ((argc < 3) || (argc > 22))
  {
    printf ("Bad arguments\n");
    exit (1);
  }
  if ((fpout = fopen (argv [argc - 1], "wb")) == NULL)
  {
    error (argv [argc - 1]);
    exit (1);
  }
  for (i = 1; i < (argc - 1); i++)
  {
    if ((fpin = fopen (argv [i], "rb")) == NULL)
    {
      error (argv [i]);
      fclose (fpout);
      remove (argv [argc - 1]);
      exit (1);
    }
    while ((c = getc (fpin)) != EOF) putc (c, fpout);
    fclose (fpin);
  };
}
```

```
error (string)
char string[];
{
  printf ("Cannot open %s\n",string);
}
```

Example Program 4

```
/*
This example program writes the string "Hello world"
to stdout using the QDOS trap io.sstrg.
*/

#include "mdv1_stdio_h"
#include "mdv1_qdos_h"

main()
{
  int sent;
  char *buffer = "Hello world";
  io_sstrg(fgetchid(stdout),-1,buffer,11,&sent);
}

io_sstrg(chan_id,time_out,b,blen,rlen)
char *chan_id,*b;
int time_out,blen,*rlen;
{
  struct REGS in,out;
  int ret;

  in.D0 = 7;        /* function number */
  in.D2 = blen;
  in.D3 = time_out;
  in.A0 = chan_id;
  in.A1 = b;

  ret = QDOS3(&in,&out);
  *rlen = (short)out.D1;
  return(ret);
}
```