LATTICE, INC.
P. O. BOX 3072 • GLEN ELLYN • ILLINOIS 60138 • 312/858-7950 • TWX 910-291-2190

TECHNICAL BULLETIN
TB841101.001

DATE: November 1, 1984
PRODUCT: 8086/8088 C Compiler
SUBJECT: Known Bugs in Version 2.14

Recently three bugs concerning the Version 2.14 libraries and startup modules have come to light. Two of these pertain to the sensing and use of the 8087 co-processor, and the other concerns the use of the "malloc" and "getmem" functions under MS-DOS 1 in the S and P models of the compiler. In addition some code was omitted from the file "_main.c".

1. Floating Point Bugs

(a) Sensing of the 8087

The technique employed in the run-time library to sense the presence of the 8087 co-processor does not work in V2.14 because the 8087 is not initialized prior to the test. Correcting this problem is rather easy. You need only add a line of code to the file "c.asm" provided with the compiler and reassemble this file to replace the current "c.obj".

In the file "c.asm" you will find a line:

```
CALL _MAIN
```

just prior to this line, insert the following line

```
DB 0DBh,0E3h    ; FNINIT instruction
```

Then reassemble "c.asm" for each of the memory models as follows:

-- Copy "c.asm" to the appropriate model subdirectory (i.e., \lc\s, \lc\d, \lc\p, or \lc\l) so that it is in the same directory as "dos.mac" for the appropriate memory model.

-- Use the command

```
masm c;
```

to assemble "c.asm" into "c.obj".

The instruction is specified here as a "DB" pseudo-op rather than an "FINIT" instruction since many assemblers do not properly handle floating point instructions. Should your assembler be able to handle such instructions, be sure to use an "FNINIT" instruction rather than "FINIT" or else the assembler will generate a "WAIT" instruction prior to the "FINIT" and resulting programs will wait endlessly on machines not containing a co-processor.

(b) Floating Point in the D and L Models

In attempting to repair an earlier bug concerning the 8087 we accidentally broke its use in the D and L models. To correct the problem you can use "debug" to patch the libraries "lcmd.lib" and "lcml.lib" as follows. (Underlined portions are what you must type. Each line must be entered with a carriage return.)

(i) Patching lcmd.lib

```
>debug lcmd.lib
-eA97A   10.12
-eA9CC   78.76
-w
-q
>
```

(ii) Patching lcml.lib

```
>debug lcml.lib
-eBB71   12.14
-eBBC7   3D.3B
-w
-q
>
```

(c) Bug in 8087 Library Divide Routine

Under certain circumstances the floating point division function CXd55 will return the incorrect value when an

8087 is present since this function fails to save and restore the DI register. This bug will be repaired in a future release.

(2) Malloc and Getmem under MS-DOS 1

The S and P model library memory allocation functions "malloc" and "getmem" will not return the correct values in an application running under MS-DOS 1 when a call to one of these would cause the total memory allocated to exceed 64K. In particular the failure value of NULL will not be returned in this case. This problem will be fixed in a future release, and it does not effect the D and L models.

(3) Omission in "_main.c"

The last "exit(0);" statement should be replaced with

```
#ifndef TINY
exit(0);
#else
_exit(0);
#endif
```

*** END ***

LATTICE, INC.

P. O. BOX 3072 • GLEN ELLYN • ILLINOIS 60138 • 312/858-7950 • TWX 910-291-2190

TECHNICAL BULLETIN
TB841010.001

DATE: October 10, 1984
PRODUCT: 8086/8088 C Compiler
SUBJECT: Version 2.14 Update

Version 2.14 of the 8086/8088 C compiler has been released to correct the following problems:

1. In Version 2.13, the S model libraries were built in such a way that the floating point operations did not use the 8087 chip if present. The other memory models worked correctly.

2. While fixing the PUTC problem in Version 2.13, we broke the level 2 buffer flush function in such a way that buffers were sometimes written twice. This would occur when you did several FSEEKs without intervening reads or writes.

3. UNGETC did not always work correctly under the D and L models, depending on where the buffer was located.

END

LATTICE, INC.

P. O BOX 3072 • GLEN ELLYN • ILLINOIS 60138 • 312/858-7950 • TWX 910-291-2190

TECHNICAL BULLETIN
TB841010.002

DATE: October 10, 1984
PRODUCT: 8086/8088 C Compiler
SUBJECT: Insufficient Memory Message

In bulletin TB840914.001 we mentioned a problem observed during the testing of Version 2.13 that resulted in an "Insufficient memory" message. At that time, we believed that the message was originating from MS-DOS. Well, we must admit with a blush that the message was coming from C.ASM, our start-up module.

The situation leading to this message was that we forked a child process whose .EXE file just barely fit into the available memory. After loading the .EXE file, MS-DOS passed control to the child process at the C.ASM entry point. C.ASM then attempted to allocate stack and heap space for the C program, but there was not enough memory. When that occurred, C.ASM displayed the "insufficient memory" message and aborted with a non-zero exit code.

The mistake we made in our test program was to examine only the return code from the FORKL function. This will indicate an error only if MS-DOS was unable to obtain the needed space and load the child program. We should have also used the WAIT function to check the exit code from the child.

END

LATTICE, INC.
P O BOX 3072 • GLEN ELLYN • ILLINOIS 60138 • 312/858-7950 • TWX 910-291-2190

TECHNICAL BULLETIN
TB841010.003

DATE: October 10, 1984
PRODUCT: 8086/8088 C Compiler
SUBJECT: Combining LC and LCM Libraries

In TB840914.001 we gave a simple procedure for combining the LC and LCM libraries. Some people who tried this procedure became concerned about the duplicate symbol messages that resulted. Here is a better procedure:

1. Use your favorite editor to create the file MIXLC.LNK containing the following:

```
bu lcc.lib
fi lcm.lib
fi lc.lib
exc _pfmt,cprintf,fprintf,printf,sprintf
exc _sfmt,cscanf,fscanf,scanf,sscanf
```

2. For each memory model, copy MIXLC.LNK into the \lc\x directory, where x is s, p, d, or l. Then use CD to get into that directory, and execute PLIB86 as follows:

```
plib86 @mixlc
```

This procedure creates a combined library LCC.LIB which includes the complete versions of the PRINTF/SCANF functions. If you want a combined library that contains the abbreviated versions, simply interchange the two "fi" commands in MIXLC.LNK.

END

LATTICE, INC.

P. O. BOX 3072 • GLEN ELLYN • ILLINOIS 60138 • 312/858-7950 • TWX 910-291-2190

TECHNICAL BULLETIN
TB840914.001

DATE: September 14, 1984
PRODUCT: 8086/8088 C Compiler
SUBJECT: Version 2.13 Update

Version 2.13 of the 8086/8088 C compiler has been released to correct the following problems:

1. STRCMP and STRNCMP did not return the correct results when comparing strings of unequal lengths.

2. The various forms of PUTC did not work correctly under the D and L models.

3. The results of a floating-point divide-by-zero operation with an 8087 installed were not the same as when the 8087 was removed.

4. Several problems existed in the various FORK functions:

   (a) The environment string array was not passed correctly to the child process under the S and P memory models.

   (b) An extra backslash was appended to the PATH variable. This was usually accepted by PCDOS, but was rejected by some MSDOS implementations.

   (c) The construction of default FCBs did not stop when an argument beginning with a slash was reached. This caused problems for some older programs that relied on the FCB setup done by the MSDOS command processor.

   (d) The fork logic searched for a .EXE file before a .COM file. This has been reversed because some people keep the .EXE version around even after they have generated a .COM.

(e) A carriage return was not appended to the generated command string, which caused some forked programs to fail.

These problems usually showed up when you tried to fork the command processor or when the child process attempted to use the inherited environment. To simplify the invocation of the command processor, we've added a UNIX-compatible SYSTEM function. The function has one argument, the command string, and returns the same results as FORKL. For example,

```
system("dir a:")
```

calls the command processor to display the directory from drive A.

While testing the fork functions, we observed what appears to be an MS-DOS bug. When the operating system cannot obtain enough memory to load the child program, it's supposed to return error code 8 that we will then pass back to you in _oserr. However, in some cases, MS-DOS displays the message "Insufficient memory" on the screen and returns a success code. This has been observed on PC-DOS 2.1 and 3.0 running on the IBM-XT and IBM-AT and in the version of MS-DOS currently running on the TANDY 2000. We are pursuing a solution, but if anybody can give us any further information, we would appreciate the help.

5. The SETJMP/LONGJMP functions did not work correctly under the P model. Note that this correction required a change in the SETJMP.H header file and that ALL PROGRAMS USING THIS HEADER MUST BE RE-COMPILED regardless of which memory model they employ.

6. Detection of the 8087 did not work correctly on systems using an 80286, such as the IBM-AT.

7. The POW function did not always return the correct result.

8. Several people complained that the libraries became way too large when we added the math functions. Therefore, we have split the library for each memory model into two pieces. LCx.LIB contains the "core functions" for memory model x, and LCMx.LIB contains the floating point math

functions. If you need to use both libraries, make sure that LCM is mentioned before LC at link time. If you are also using our CFOOD *Smorgasbord*, its library (LCX) should be mentioned before LCM. At this time we are pretty confident that the library was split in the correct place, but if you find any interdependence problems that we overlooked, please report them to us. If you want to re-combine the two libraries, use PLIB86, as follows:

```
ren lc.lib lcc.lib
plib86 bu lc.lib fi lcc.lib,lcm.lib
```

Note that the batch files that we've been supplying for linking, named LINKx.BAT, only use LC.LIB. The release disks include some new batch files named LINKMx.BAT that use both LCM.LIB and LC.LIB.

You should also be aware that two versions of the PRINTF and SCANF families are provided. If you don't use LCM.LIB, you'll get a version that does not support floating point conversions, which saves about 3 Kbytes in the load module. With LCM.LIB, you get the full PRINTF and SCANF conversion capabilities.

9. We've eliminated the need for TINYMAIN.C by putting conditional compilation statements into _MAIN.C, and the release disks now contain pre-compiled versions of the abbreviated _MAIN under the names _MAINx.OBJ, where x is the memory model (S,P,D, or L). If you need to recompile _MAIN, use the LC option -dTINY=1 to get the abbreviated version.

The LC libraries still contain the full version of _MAIN, which is the default. Because this full version supports the stdin, stdout, and stderr files, it forces all of the level 2 I/O functions to be included. If you don't need these standard files and don't use any level 2 I/O, your load module size will be reduced by using the abbreviated version.

END

LIFEBOAT
associates

Lattice C Manual

Lattice 8086/8088
C Compiler

Functional Description Manual

Released May 25, 1982

This document describes Revision 2 of the compiler and library.

Note:

Supplement to Lattice C v. 2.1 appended to this document May 29, 1984.

Published by:

Lifeboat Associates
1651 Third Avenue
New York, New York 10128

Tele: (212) 860-0300
Telex: 424490 (LBSOFT UI)

PREFACE

Lattice, Inc., a developer of portable software products based in Chicago, Illinois originally developed Lattice C for its own internal use on a minicomputer. When the IBM PC was introduced in 1981, the company recognized the potential for developing a full implementation of the C programming language for 16-bit microcomputers. Lifeboat Associates, a New York-based software publisher, then provided Lattice with funding which enabled the company to make the conversion. It is believed that Lattice C was the first minicomputer product successfully "ported" to the IBM PC. In 1983, Microsoft, based in Bellevue, Washington, selected Lattice C to become Microsoft C. Also in 1983, Lattice developed a version of its compiler for the Motorola 68000 microprocessor and a Z80 cross-compiler for 8-bit CP/M-based microsystems. With Revision 2, Lattice C implements MS-DOS 2.0 pathnames and large memory models, giving software developers the ability to create programs and data structures which can better utilize the large memory available on 8086/8088-based systems.

TABLE OF CONTENTS

SECTION 1:
Introduction

This document provides a functional description of an implementation of the Lattice C compiler, a portable compiler for the high level programming language called C. 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., Englewood Cliffs, New Jersey, 1978). This description of Lattice C is incomplete without the Kernighan and Ritchie text, as it is called, which also provides an excellent tutorial introduction to the language.

1.1 Documentation

The manual is divided into five sections. First, this introduction. Second, the language accepted by the compiler, which differs from the standard in only a few minor details, is described. The third section presents the portable library functions in functional groups with calling sequences and examples. Fourth, the details of the compiler and run-time program execution are presented for this implementation, including detailed operating instructions, machine dependencies, and program structure. Fifth, the operating system interfaces are described in terms of the portable library functions (file naming conventions, etc.) and the special functions provided with this implementation.

As this document 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 unavoidable, but these references are specifically noted.

To get an overview of the compiler, read the first portion of each of the major subsections in the implementation description (Section 4), the language summary at the beginning of the language definition (Section 2), and the function summaries at the beginning of the library groups (Section 3), and the introductions to the subsections in the system interface information (Section 5). Error messages are described in Appendix A and error reporting procedures in Appendix B. Methods of converting C programs written for CP/M microcomputer systems are described in Appendix C. A list of files shipped with Lattice C are contained in Appendix D.

SECTION 2:
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.

2.1 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.

2.1.1 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:

- Comments normally can be nested in the Lattice compiler; in the standard, they cannot. A compile-time option forces the compiler back to the standard non-nesting mode.

- 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.

- The dollar sign ($) is permitted as an embedded (i.e., not the first) character in identifiers.

- Identically written string constants refer to the same static storage location; that is, only one copy of the string is generated by the compiler. This is in contrast to the statement in Kernighan and Ritchie that all strings are distinct, even when written identically.

- Multiple character constants are accepted by this compiler; in the standard, only a single character enclosed in single quotes is legal. The resulting value may be short or long, and its exact value is machine-dependent.

- 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 2.3, 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.

2.1.2 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 size, in bytes, of any declared object is the largest positive integer which can be represented as an int. This implies, for example, a maximum size of 32767 bytes for

16-bit int machines. The total size of all objects declared with the same storage class is also subject to the same restriction.

- 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 (32766 for a 16-bit int).

- The total size of the formal parameters for any function is limited to a maximum of 256 bytes. Thus, the maximum number of formal parameters depends on their sizes.

- The maximum size of a string constant is 256 bytes.

- Macros with arguments are limited to a maximum number of 8 arguments.

- The maximum level of #include file nesting is 4.

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.

2.2 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.

2.2.1 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 are not generally performed for any of the pre-processor commands, although nesting of macro definitions is possible since substituted text is always re-scanned for new #define symbols.

An exception occurs for the #if command, which is processed differently. As noted in the list of differences, sizeof cannot be used in #if expressions, and the expression must appear entirely on a single line. These restrictions result from a desire to keep #if expressions simple, and because the pre-processor generally has no information about the size of declared

objects. One other clarification should be noted: if a symbol appears in an #if expression which has not been defined in a #define command, it is interpreted as if a value of zero had been specified. This seems consistent with #ifdef usage and permits the use of 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 2.1.1, 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".

2.2.2 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:

```
short or short int
char
unsigned or unsigned int
long or long int
float
double or long float
```

Note that in this implementation, **unsigned** is not a modifier but a separate data type.

The natural size of integers for the target machine (the machine for which code is being generated) is indicated by a plain **int** type specifier; this type will be identical to either **short** or **long**, depending on the architecture of the target machine. Although the size of all these objects is technically machine-dependent, the Lattice compiler assumes the target machine has an 8-bit, 16-bit, or 32-bit architecture and that the fundamental storage quantity is an 8-bit byte. Only in connection with bit fields does this assumption ever become important.

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.

2.2.3 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 2.2.4). 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.

2.2.4 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 2.2.6). 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. The implementation of register is machine-dependent and may not be supported in some cases.

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.

2.2.5 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 2.1) 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.

2.2.6 Initializers

Objects which are of the **static** storage class (as defined in Section 2.2.4) 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 2.1, 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.

2.2.7 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.

2.2.8 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**, **do**, 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.

2.3 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. This feature makes it easy to "comment out" large sections of code which themselves contain comments. The c compile-time option forces the compiler to process comments in the standard, non-nesting mode.

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 int or long, depending on the number of characters, and its value is machine-dependent. 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,

```
'\xf9'
```

generates a character with the value 0xF9.

CRM 2.5 Strings

The Lattice compiler recognizes 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.

CRM 2.6 Hardware characteristics

See the implementation section of this manual for hardware characteristics.

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 sealed (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 2.2.4 of this document, 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 10.1 External function definitions

As noted in the text, formal parameters declared float are actually interpreted as double; similarly, formals declared char or short are read as int. For consistency, the Lattice compiler applies the same rules to functions: a function declared to return float is assumed to return double, and char or short functions to return int.

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 2.2.1 of this document, 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.

Future versions of the compiler may support structure assignment, but the value of other operations (such as passing aggregates directly to or returning them from functions) seems questionable.

Section 3:
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. Although not all of the features of these functions can be implemented on every system supported by the compiler, all systems must be able to provide 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.

3.1 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 actual amount of memory available is system-dependent and usually depends on the size of the program. In most systems the memory made available for dynamic allocation by these functions is the same memory used for the run-time stack (used for function calls and auto variables). On these systems a default number of bytes is reserved for the stack, and the remainder of the memory is used by the memory allocation functions. In order to allow programs to adjust the amount of memory reserved for the stack (and thus the amount available for dynamic allocation), the main program usually supports a special =n option to override the default stack size; alternatively, a program may define the size internally. Check the implementation section of the manual for details. The user is cautioned that on many systems there is no check against the stack overrunning its allotted size and destroying portions of the memory pool.

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.

3.1.1 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 efficiently utilized if the level 2 functions are used instead.

NAME

malloc -- 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);
```

NAME

calloc -- allocate memory and clear

SYNOPSIS

```
p = calloc(nelt, eltsiz);

char *p;            block pointer
unsigned nelt;      number of elements
unsigned eltsiz;    element size in bytes
```

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));
```

NAME

free -- 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 ... }
```

3.1.2 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.

NAME

getmem, getml -- 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

Gets 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.

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));
```

NAME

rlsmem, rlsml -- 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

Releases 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.

RETURNS

ret = Ø 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.

NAME

allmem, bldmem -- 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 **getmem** can be freed by a call to the **rstmem** 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.

NAME

sizmem -- 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.

NAME

rstmem -- 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.

3.1.3 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.

NAME

sbrk, lsbrk -- 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 ln bytes;      long number of bytes to be allocated
```

DESCRIPTION

Allocates a block of memory of the requested size, if possible. These functions form the basic UNIX memory allocator. The first time one of them is called, it will allocate the largest available block of high memory. Then the requested number of bytes is subtracted from the low end of the block for use by the caller.

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.

NAME

rbrk -- reset memory break point

SYNOPSIS

```
rbrk();
```

DESCRIPTION

Resets the memory break point to its original starting-point. This effectively returns all memory to the free space block.

CAUTIONS

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

3.2 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. The major system dependency arises in connection with text files, where some systems perform certain translations to accommodate the particular text file representation used in the local environment. Although the translation is generally transparent at the higher levels, I/O at the lowest levels, particularly I/O involving binary data, must be aware of the translation. Check the implementation section of this manual for the details appropriate to a particular system.

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.

The system functions discussed in this section are concerned with program exit. Additional system functions are described in the implementation section of the manual.

3.2.1 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 on most systems) which generally 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 on most systems 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 may -- depending on the system -- perform a text mode translation so that end-of-line sequences will conform to local conventions. 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. Check the implementation section of this manual for more information about the file access modes.

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 most 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.

NAME

fopen -- open a buffered file

SYNOPSIS

```
fp = fopen(name, mode);

FILE *fp;           file pointer for specified file
char *name;         file name
char *mode;         access mode
```

DESCRIPTION

Opens a file for buffered access; the translated mode is the default mode but may be overridden as described in the introduction to this section. 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  to read a file
w  to write a file
a  to append to a file
r+ to update a file (read and write)
w+ to create a file for update
a+ to append to and update a file
```

The mode character 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. On most systems, no more than 16 files (including stdin, stdout, and stderr, if those are opened for main) can be opened using fopen.

When a file is opened for update, both reading and writing may be performed on the file pointer. In order to switch modes, an fseek or rewind must be executed. Opening the file to append forces all data to be written to the current end of file, regardless of previous seeks.

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.

NAME

freopen -- reopen a buffered file

SYNOPSIS

```
fpr = freopen(name, mode, fp);

FILE *fpr;          file pointer after re-opening
char *name;         file name
char *mode;         access mode
FILE *fp;           current file pointer
```

DESCRIPTION

Reopens a buffered file; that is, attaches a new file to a previously used file pointer. This function is useful for programs which must open several files, but only one at a time; this avoids using up file pointers unnecessarily. The previous file is automatically closed before the file pointer is reused. The name and mode arguments are the same as those for fopen.

RETURNS

```
fpr = NULL if error
    = fp if successful
```

CAUTIONS

The return code should be checked for NULL; the same errors defined for fopen may occur.

NAME

fclose -- 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 3.2.4) 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

NAME

getc, getchar -- 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

These are implemented as macros, so beware of side effects.

NAME

putc, putchar -- 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

These are implemented as macros, so beware of side effects.

NAME

fgetc, fputc -- get/put a character

SYNOPSIS

```
r = fgetc(fp);
r = fputc(c, fp);

int r;              return character or code
char c;             character to be sent (fputc)
FILE *fp;           file pointer
```

DESCRIPTION

These functions get (fgetc) or put (fputc) a single character to the indicated file. Since they are functions, they are often recommended for use rather than the corresponding macros (getc and putc) in two types of situations: (1) if many calls are made and/or (2) if the programmer is concerned about the amount of memory used in the macro expansions. The tradeoff is the usual one: the macro executes more quickly because it saves a function call; the function requires less memory since its code is present in the program only once.

RETURNS

r = character if successful (c, for fputc)
  = EOF if error or end-of-file

NAME

ungetc -- push character back on input file

SYNOPSIS

```
r = ungetc(c, fp);

int r;              return character or code
char c;             character to be pushed back
FILE *fp;           file pointer
```

DESCRIPTION

Pushes back a character to the specified input file. The character supplied must be the character most recently obtained by a getc (or getchar, in which case fp should be supplied as stdin) invocation.

RETURNS

r = character if successful
  = EOF if previous character does not match

NAME

fread, fwrite -- 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.

NAME

gets, fgets -- 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, so the input buffer must be large enough to accommodate the string.

NAME

puts, fputs -- 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

NAME

scanf, fscanf, sscanf -- 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);
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 a long 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. 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.

NAME

printf, fprintf, sprintf -- 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; 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, so it 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.

NAME

fseek -- 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 3.2.2) 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.

NAME

ftell -- 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 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.

NAME

ferror, feof -- 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
```

NAME

clrerr -- clear error flag for file

SYNOPSIS

```
clrerr(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.

NAME

fileno -- 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.

NAME

rewind -- 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.

NAME

fflush -- 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.

NAME

setbuf -- change buffer for level 2 file I/O

SYNOPSIS

```
setbuf(fd,buf);

FILE *fd;
char *buf;
```

DESCRIPTION

This function attaches a private buffer to the file whose descriptor is fd. The length of the buffer is assumed to be the same as _bufsiz, which is defaulted to the constant BUFSIZ in stdio.h.

If the buffer pointer is NULL, then this function is the same as setnbf.

CAUTIONS

buf must be large enough to handle the data specified in _bufsiz.

NAME

setnbf -- 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, so this function should only be used for those "files" which are definitely known to be devices.

3.2.2 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. On some systems the two modes are identical. 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 so that any required translation of text file end-of-line sequences can be performed automatically even by the lowest level operations (read and write functions), while at the same time a program may disable the translation, as needed, when working with binary files. The problem is that not all systems use the standard C end-of-line delimiter, the newline (\n); the translated mode converts the newline to whatever the local delimiter may be. Since this may involve expansion or contraction of the number of bytes read or written, the count returned by read or write may not correctly reflect the actual change in the file position. In the binary mode, this problem

does not occur since no translation is performed.

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.

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 exact nature of the I/O performed is system-dependent, but it is generally unbuffered and may have different effects, depending on whether the translated or untranslated mode is in effect. 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 3.2.3.

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.

NAME

open -- 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
                   options (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	Forces 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 3.2.2)

The current file position is set to zero if the file is successfully opened. On most systems, no more than 16 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.

NAME

creat -- 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 for 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 system-dependent and on some systems may be 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.

NAME

unlink -- remove file name from file system

SYNOPSIS

```
ret = unlink(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.

NAME

read -- 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.

NAME

write -- 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.

NAME

lseek -- 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 logical end-of-file (mode 2). Not all implementations support offset 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.

NAME

close -- 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.

3.2.3 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
```

Several system dependencies arise in connection with the direct console functions. Whether or not characters are echoed as they are input is system-dependent but there is usually a mechanism to enable or disable the echo. On some systems the characters that are typed when the program is not actually waiting for input are saved, and then presented to the getch function when it requests input. Often only one character is saved; however some systems may save none while others retain several. The presence of type-ahead, as this feature is usually called, rarely affects the program itself, although its absence may be a source of irritation to users who have to communicate with the program.

NAME

getch, putch -- get/put character directly from/to console

SYNOPSIS

```
c = getch();
putch(c);
int c;          character received/sent to console
```

DESCRIPTION

These functions get (getch) or put (putch) single characters from or to the user's console.

RETURNS

c = character received (getch)

CAUTIONS

There is no notion of an end of file or error status, but some implementions may use EOF (-1) as an error return.

NAME

ungetch -- 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 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

NAME

kbhit -- 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.

RETURNS

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

NAME

cgets -- 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. Characters are input until a system-dependent terminator (usually CR, 0x0D) is encountered. The carriage return is replaced by a NULL byte.

RETURNS

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

CAUTIONS

Check the implementation section of this manual for details of the operation of this function.

NAME

cputs -- 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. Does not automatically generate a carriage return or linefeed.

NAME

cscanf, cprintf -- 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.

RETURNS

n = number of input items matched (cscanf)

CAUTIONS

cscanf performs its I/O directly using getch, so there are none of the usual input conveniences such as back spacing or line deletion. If an implementation's version of cgets provides some of these conveniences, it may be better to call cgets and then use sscanf to decode the resulting string.

3.2.4 Program Exit Functions

The program entry mechanism, that is, the means by which the **main** function gains control, is sufficiently system-dependent that it must be described in the implementation section of this manual. Program exit, however, is somewhat more general, although not without its own implementation dependencies.

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. They offer the advantage of allowing any function -- not just **main** -- to cause termination of the program, and in some systems, they allow information to be passed to other programs.

NAME

exit -- 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.

NAME

_exit -- terminate execution immediately

SYNOPSIS

```
_exit(errcode);

int errcode;        exit error code
```

DESCRIPTION

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

3.3 Utility Functions and Macros

The portable library provides a variety of additional functions useful for many of the common data manipulations performed by C programs. Three utilities provide fast memory transfers; a set of macros allows quick testing of character types; and several utility functions facilitate character string handling.

3.3.1 Memory Utilities

The three utility functions described here are usually implemented in machine language for maximum efficiency. These are the equivalent of the almost universal FILL and MOVE subroutines defined in many other languages.

NAME

setmem -- 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. On many systems a hardware block fill instruction is used to perform the initialization. This function is useful for the initialization of auto char arrays.

CAUTIONS

Some systems may distinguish between char * pointers and pointers of other types, so it is good practice to use a cast operator when arrays or pointers of other types are used for the p argument.

NAME

movmem -- 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. On many systems a hardware block move instruction is used to perform the transfer.

CAUTIONS

Some systems may distinguish between char * pointers and pointers of other types, so it is good practice to use a cast operator when arrays or pointers of other types are used for the s and d arguments.

NAME

repmem -- 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

Some systems may distinguish between **char *** pointers and other types of pointers, so it is good practice to use a cast operator when arrays or pointers of other types are used for the s and v arguments.

3.3.2 Character Type Macros

The character type header file, called ctype.h on most systems, 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, 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.

Here are the macros defined in the character type header file ctype.h. Note that many of these will evaluate argument expressions more than once, so 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.

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.

3.3.3 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, four of the most common functions are provided with str names; these are the functions mentioned in Kernighan and Ritchie: strlen, strcpy, strcat, and strcmp.

NAME

strlen, stclen -- measure length of string

SYNOPSIS

```
length = strlen(s);
length = stclen(s);

int length;          number of bytes in s (before NULL)
```

DESCRIPTION

Counts the number of bytes in s before the NULL terminator. The terminator itself is not included in the count.

RETURNS

length = number of bytes in string before NULL byte

NAME

strcpy, stccpy -- copy one string to another

SYNOPSIS

```
strcpy(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;          sizeof(to) (stccpy only)
```

DESCRIPTION

Moves the NULL-terminated source string to the destination string. strcpy does not get a length parameter, so all of the source string is copied unconditionally. For stccpy, if the source is too long for the destination, its rightmost characters are not moved. The destination string is always NULL-terminated.

RETURNS

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

CAUTIONS

As noted above, strcpy does not get a length parameter, so the destination string must be large enough. Use stccpy if this causes problems.

NAME

strcat -- concatenate strings

SYNOPSIS

```
strcat(to, from);

char *to;          string to be concatenated to
char *from;        string to be added
```

DESCRIPTION

Concatenates from to the end of to. The result is always NULL-terminated.

CAUTIONS

No length parameter is present, so the destination string must be large enough to receive the combined result.

NAME

strcmp, stscmp -- compare two strings

SYNOPSIS

```
status = strcmp(s, t);
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
```

DESCRIPTION

Compares two NULL-terminated strings, byte by byte, and returns 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.

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.

NAME

stcu_d -- 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.

NAME

stci_d -- 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.

NAME

stch_i -- 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.

NAME

stcd_i -- 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 Ø-9. The first character may be + or -.

RETURNS

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

CAUTIONS

No check for overflow is made during processing.

NAME

stpblk -- 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.

NAME

stpsym -- 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.

NAME

stptok -- 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.

NAME

stpchr -- find specific character in string

SYNOPSIS

```
p = stpchr(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

Scans the specified string to find the first occurrence of the specified character. If the NULL terminator byte is hit first, a NULL pointer is returned.

RETURNS

```
p = NULL if c not found in s
  = pointer to first c found in s (from left)
```

CAUTIONS

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

NAME

stpbrk -- find break character in string

SYNOPSIS

```
p = stpbrk(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

Scans 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

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

NAME

stcis, stcisn -- measure span of a character set

SYNOPSIS

```
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 stcis, the character set consists of all characters in b, while for 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

NAME

stcarg -- 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

NAME

stcpm -- 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.

NAME

stcpma -- 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
```

NAME

stspfp -- 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 slashes. Each slash is replaced by a NULL byte, and the beginning index of that node is placed in the index array. For example, the pattern /abc/de/f has three nodes, and their indexes are 1 for abc, 5 for de, and 8 for f. Note that the leading slash, if present, is skipped. Note also that a slash that is part of a node name (usually unwise) must be preceded by a backslash. 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
```

3.3.4 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.

SECTION 4:
Compiler and Run-time Implementation

A version of the Lattice C compiler for the 8086/8088 runs under Microsoft's MS-DOS operating system. It accepts programs written in the C programming language (the full language -- not a subset) and produces relocatable machine code in Intel's 8086 object module format, suitable for use by Microsoft's program linker. The library defines a comprehensive set of I/O subroutines which implement under MS-DOS most of the UNIX-compatible standard functions described in the text by Kernighan and Ritchie.

The 8086 instruction set is well-suited to the implementation of a high-level language like C, and the Lattice compiler generates machine code which takes full advantage of its features. Although the 8086 architecture supports up to 1 megabyte of addressable memory, its segmented addressing approach works most efficiently with 64K-byte program and data address spaces. In order to provide the most flexibility, the compiler supports four different memory addressing environments, or models, from which the programmer can select the combination of efficiency and addressability required for a particular application. These models are discussed in more detail in Section 4.4; initially, only the simplest and most efficient model will be presented in examples: the so-called S model in which a program may have a maximum of 64K bytes of program section (functions), plus a maximum of 64K bytes of data section (including static data, auto or stack data, and dynamically allocatable memory). Despite these limitations, programs of considerable complexity and power (including the compiler itself) can be developed.

4.1 Operating Instructions

See Appendix D for the most current list of the files supplied with the compiler package. The executable files LC1.EXE and LC2.EXE make up the actual compiler. Each 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. A batch procedure file can be used to execute LC1 and LC2 in succession, using the same file name (the normal sequence). The compilation process can be diagramed as follows:

```
file.C  ->  LC1  ->  file.Q
file.Q  ->  LC2  ->  file.OBJ
```

LC1 reads a C source file, which must have a .C extension, and (provided there are no fatal errors) produces an intermediate file of the same name with a .Q extension. LC2 reads an intermediate file created by LC1 and produces an object file of the same name with an .OBJ extension. The .Q file is deleted by LC2 when it completes its processing. Each phase normally creates its output file on the same drive and directory 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; see Section 4.3.2 for more information.

The .OBJ file must be supplied as input to the linker in order to produce an executable program file. Two special files must also be involved in the linking process, in addition to any .OBJ files created by the user. The linking process can be diagrammed as follows:

(Note that the actual filenames used depends upon the memory model selected; see Section 4.4 for more information. In this discussion and in the example below, the S model will be used to illustrate the linking process.)

CS.OBJ + user.OBJ + ... + LCS.LIB -> LINK -> user.EXE

The special files required are CS.OBJ and LCS.LIB. First, the file CS.OBJ must be specified as the first module on the LINK execution command; this module defines the execution entry and exit points for any program generated using the Lattice C compiler. Second, the file LCS.LIB must be specified as the library; this file defines all of the run-time and I/O library functions included as part of the Lattice C package. The user must also specify at link time the names of any .OBJ files which are to be included, as well as the name of the .EXE file which will be created by the linker.

To illustrate the program generation sequence, the following commands necessary to compile, link, and execute the Fahrenheit-to-Celsius sample program (FTOC.C). This example assumes that all of the .EXE files (LC1, LC2, and LINK) reside on the same disk and directory. The commands will be shown in upper case, although lower case commands will work as well. (Note: the linker prompts described here are for Version 1.10 of the Microsoft linker; for LINK.EXE versions other than 1.1, and for use with linkers other than the Microsoft linker, appropriate documentation should be consulted. Generally, the default responses are correct.)

STEP 1: Execute the first phase of the compiler by typing

LC1 FTOC<ENTER>

Note that the .C extension is not supplied (although the command will work properly even if it is).

STEP 2: When the MS-DOS prompt is issued after LC1 has completed its processing, execute the second phase of the compiler with

LC2 FTOC<ENTER>

Again, no extension is specified; LC2 supplies the .Q

extension.

STEP 3: When the prompt is issued after LC2 has completed its processing, the linker is invoked by typing

```
LINK CS+FTOC,FTOC,NUL,LCS
```

Note that CS (meaning CS.OBJ) is specified as the first object module on the LINK command; this is required for the linking of any C program. Then FTOC (meaning FTOC.OBJ, which was just produced by LC2) is specified as an additional object module. The second FTOC causes the run file to be named FTOC.EXE, NUL skips the generation of a link map, and LCS causes LINK to search LCS.LIB for external references.

STEP 4: Execute the .EXE file by typing

```
FTOC<ENTER>
```

The program writes a list of Fahrenheit temperature values and their Celsius equivalents to the user's console.

Detailed instructions for compiling, linking, and executing programs are presented in the following sections. See Section 4.3 for a detailed discussion of the processing performed by the compiler phases.

In presenting the various command line formats, the term field will be used to describe a sequence of non-white space characters in the command line. Optional fields will be shown enclosed in square brackets []; the brackets are not to be included when the actual command is typed. Examples are provided at the end of each section.

Versions of Lattice C designed to take advantage of MS-DOS version 2.0 recognize the full Version 2 pathnames for all filenames. The name can be specified on the command line, as in:

```
LC1 b:\lowlevel\file
```

(which specifies b:\lowlevel\file.c for compilation), or it can be specified in #include statements, as in

```
#include "b:\headers\stdio.h"
```

See option -id below for further uses of command line pathnames.

4.1.1 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 4.3.1 for a more detailed discussion of the processing performed. The format of the

command to invoke the first phase of the compiler is:

LC1 [=stack] [>listfile] **filename** [options]<ENTER>

The various command line specifiers are shown in the order they must appear in the command. Required specifier are shown in emphasized type. Optional specifiers are shown enclosed in brackets. The first two options are part of the general command line options for all C programs (see Section 4.1.4). This allows the use of if expressions in batch files, such as:

```
LC1 %1
if errorlevel 1 goto errs
```

=stack The first option is used to override the number of bytes reserved for the stack (see Section 4.5 for a complete description of the structure of C programs). The default is 2048 (decimal) bytes, which is sufficient for most programs. If present, the stack size override field must be the first field after the name of the first phase (LC1). 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.

>listfile The second option is used to direct the first phase messages to a specified file. These messages include the compiler sign-on message and any error or warning messages which may be generated. The full filename must be specified, including extension, if any. If the file already exists, it is truncated and reused. This option is useful for reviewing long lists of error messages.

filename This is the only command line field which must be present; it 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. 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 default drive and directory (hereafter drive/directory) are used unless a pathname preceding the filename specifies another drive/directory; the quad file is created in the same drive and directory as the source file unless the -o option is used (see below). Alphabetic characters may be either upper or lower case in filenames.

options Compile time options are specified as a hyphen followed by a single letter. The letter must be typed in lower case; the corresponding upper case option will have no effect. Each option must be specified separately, with a separate hyphen and letter (that is, they cannot be combined as they can for certain UNIX programs). Current options include:

-a Causes the compiler to assume worst-case aliasing, that is, to abandon any optimizations based on favorable assumptions about pointers. Normally, the compiler assumes that objects referenced through pointers are not the same as objects being referenced directly in the same section of the program; this option cancels that assumption. The -a option additionally forces all assignment statements to be performed (i.e., the actual store to memory) before execution of the next statement. Normally, the code generated for assignment causes a value to be loaded to a register, but it may not be stored immediately; the -a flag now forces the store operation. This is important only in (1) unions, where a value is stored and then immediately inspected or passed to a function via another member; (2) real-time processing where shared data values are used as "lock" words, and immediate execution of an assignment statement is critical to subsequent actions; and (3) memory-mapped I/O assignments, where values must be stored repeatedly in the same "memory" location.

-b Forces byte alignment for all offset calculations. The first phase normally aligns all objects which are not pointers, structures, or unions on a word boundary.

-c Causes comments to be processed without nesting. The Lattice compiler normally assumes that comments may be nested; this allows large sections of code to be commented out very easily. This option allows the user to force the compiler to the standard, non-nesting mode of operation.

-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. If this option is

used, the object file produced will contain line number/offset records, and can be processed by the Object Module Disassembler to produce an intermixed source code and machine code listing (see Section 4.1.6 below). Note that the -d option does not affect the size of the function itself, only the object file.

-iprefix Specifies that #include files are to be searched for by prepending the filename with the string prefix, unless the filename in the #include statement is already prefixed by a drive or directory identifier. Up to 4 different -i strings may be specified. Note that when an unprefixed #include filename is encountered, the current drive/directory is searched; then drive/directories are searched using prefixes specified in -i options, in the same left-to-right order as they were supplied on the command line. The drive/directory specification should follow DOS 2.0 naming conventions (see example below). No intervening blanks are permitted in the string following the i.

-mM Causes the compiler to generate code for the specified memory model. The model can be specified as a single letter, either upper- or lower-case, naming the model; or a numeric indicator from 0 to 3 may be used (S=0, P=1, D=2, L=3). The model specifier must be adjacent to the m (no intervening blanks). (See Section 4.4.2).

-n Causes the compiler to retain up to 39 characters for all identifier symbols, including #define symbols. The default symbol retention length is 8 characters.

-oprefix Specifies that the output file (the .Q or quad file) is to be formed by prepending the input filename (the .C file which is being compiled) with prefix. The drive is specified by a single alphabetic character, either upper or lower case, followed by a colon. Thus -ob: causes prepending with b:. Any drive or directory prefixes attached to the input filename are discarded before the prepending is performed. No intervening blanks are permitted in the string following the o.

-s Changes the way code is generated for four-byte pointers in the D and L models; see Section 4.4.5.

-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: i.e., external definition. 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. This option is provided for use on programs written for the BDS C compiler; see Appendix C for more information.

EXAMPLES

LC1 xyzfile -b:\headers\

This command executes the first phase of the compiler using file XYZFILE.C as input, creating file XYZFILE.Q in the current directory. Any #include files not found in the current drive/directory will be searched for in the directory B:\HEADERS. Note the trailing backslash on the prefix attached to the -i flag; it is not automatically assumed by the compiler.

LC1 XYZ -ob: -x

This command executes the first phase of the compiler using file XYZ.C as input, creating file XYZ.Q on B:; it sets all external declarations without a storage class to be interpreted as extern declarations.

LC1 =4096 >tns.err tns

This command executes the first phase of the compiler using file TNS.C as input, creating file TNS.Q on the currently logged-in disk; it causes the stack size to 4096 decimal bytes, and create a file TNS.ERR to contain all of the messages generated by the compiler.

4.1.2 Phase 2

The second phase of the compiler reads a quad file created by the first phase and creates an object file in the standard MS-DOS format. See Section 4.3.2 for a more detailed discussion of the processing performed. The format of the command to invoke the second phase of the compiler is:

LC2 filename [options]<ENTER>

The command format is very similar to that for the first phase. The stack size override and listfile options can also be used, but they are generally less useful and will not be described here in any detail. Note that neither phase of the compiler does any processing of the standard input, so the < option has no effect on either phase (see Section 4.1.4 for the general C program execution options).

filename This field must be present; it specifies the name of the intermediate file for 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 default directory is used unless another drive/directory name is specified, and the object file is created in the same drive as the quad file unless the -o option is used (see below).

options Compile time options are specified as a hyphen followed by a single letter. The letter must be typed in lower case; the corresponding upper case option will have no effect. Each option must be specified separately, with a separate hyphen and letter (that is, they cannot be combined as they can for certain UNIX programs). Current options include:

-ggroup Assigns a name of the user's choice to be used for the code group in the .OBJ module. group may be 15 or fewer characters in length, and must be adjacent to the -g (no intervening blanks).

-oprefix Specifies that the output file (the .OBJ file) is to be formed by prepending the input filename (the .Q file which is being compiled) with prefix. The drive is specified by a single alphabetic character, either upper or lower case, followed by a colon. Thus -ob: causes prepending with b:. Any drive or directory prefixes attached to the input filename are discarded before the prepending is performed. No intervening blanks are permitted in the string following the o.

-ssegment Assigns a name of the user's choice to be used for the code segment in the .OBJ module. segment must be 15 or fewer characters in length, and must be adjacent to the -s (no intervening blanks).

-v Causes the code generator to omit the code at the entry to each function which checks for stack overflow (See Section 4.5.5).

The -g and -s options for LC2 are provided to override the default code group and segment names. Only users who need to interface to very specialized applications (other languages, etc.) will need to make use of these options.

EXAMPLE

LC2 u790 -oc:

This command executes the second phase of the compiler using file U790.Q as input, causing the file U790.OBJ to be created on drive C:.

4.1.3 Program Linking

After all of the component source modules for a program have been compiled, they must be linked together to form an executable 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 functions not defined in the current module; before such programs can execute, they must be "connected" with those other modules. These external functions may be defined by the user, in which case they must be compiled and be available as .OBJ files, or they may be defined in the library supplied with the compiler. (The portable functions are described in Section 3; others defined only under MS-DOS are described in Section 5.5.) Third, although C normally defines the function called main to be the execution point of a C program, there is usually a considerable amount of system-dependent processing which must be performed before main is actually called; the module to perform this processing is integrated into the program when it is 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 the linker cannot 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.

The linking process in a general sense requires that all of the components of a program be specified, either directly or indirectly, as input to the linker. Three types of input are required.

1. A start-up file CS.OBJ (or CP.OBJ, CD.OBJ, or CL.OBJ) must be specified as the first module included by the linker. This file defines the MS-DOS entry point for all C programs compiled using the Lattice C compiler.

2. Functions generated by the user must be specified as additional modules to be included. These modules include the main module, as well as any additional functions defined in other source modules.

3. A library file LCS.LIB (or LCP.LIB, LCD.LIB, or LCL.LIB) must be specified as the library to be searched during linking.

In the case of the Microsoft linker supplied with MS-DOS, these inputs are specified by:

1. Making CS (or CP, CD, or CL) the first module on the LINK command.

2. Including the names (without the .OBJ extension) of the user's object files on the LINK command, after the CS (or CP, CD, or CL) specification.

3. Typing LCS (or LCP, LCD or LCL) in response to the Libraries prompt from the linker.

Note that for step (2), one of the files included on the LINK command must be the main module.

If the linker cannot find one of the .OBJ files mentioned on the LINK command, it will stop processing without creating a .EXE file. Another error condition can arise if the linker cannot find all of the external items referred to in the .OBJ files specified. In this case, the message Unresolved Externals will be generated by the linker, followed by a list of the external names which were not defined. No attempt to execute a program with unresolved externals should be made unless it is certain that the missing functions will never be called.

See Section 4.2.2 for a discussion of external names. See Section 4.4 for a discussion of the startup and library files used in the four memory models. See Section 4.5 for a technical description of the object code features used in this implementation. If the linker being used allows generation of a public symbol map, a .MAP file may be created, allowing the examination of the components in the resulting load module.

EXAMPLE

```
LINK CS XYZ QRS<ENTER>

Run File [CS.EXE]: XYZ<ENTER>
List File [NUL.MAP]: <ENTER>
Libraries [.LIB]: LCS<ENTER>
```

This command executes the linker, producing XYZ.EXE as an executable program, and causes the files XYZ.OBJ and QRS.OBJ to be included in the program. Answers to the prompts from the linker used for this compilation are also shown.

Alternatively, these linker instructions can appear on a single command line:

```
LINK CS+XYZ+QRS,XYZ,NUL,LCS<ENTER>
```

4.1.4 Program Execution

When a C program is executed, the function main is called to begin execution. Two important services are performed for main before it receives control.

1. The command which executed the program is analyzed, and information from the command line is supplied as parameters to main. The analysis performed and the nature of the parameters supplied will be discussed in detail below. This feature is designed to make it easier to process command line inputs to the program.

2. The buffered text files stdin (standard input), stdout (standard output), and stderr (standard error) are opened and thus available for use by the program. Normally, all three units are assigned to the user's console, but stdin and stdout may be assigned elsewhere by command line options described below. This feature allows flexibility in the use of programs which work with text file I/O using the standard getchar and putchar macros.

The simplest way to execute a C program is to type the name of the .EXE file (without the .EXE extension), followed by a return (<enter>). Since the command line provides a convenient way to supply input to a program, a program execution request will often contain other information. The general format of the command line to execute a C program is:

pgmname [=stack] [<infile] [>outfile] [args]<ENTER>

Everything after pgmname is optional, as the brackets indicate. The various additional items (=stack, <infile and >outfile), if present, must appear before all other command line arguments following the program name. Note that these three items do not contribute to the argument count.

pgmname This field names the program to be executed; it is the name of the .EXE file created when the program was linked. It must be specified without the .EXE extension.

=stack The first optional field is used to specify a decimal number of bytes to be reserved for the stack when the program executes. The default value used if this field is not present is 2048 bytes. The stack size is specified as a decimal number immediately preceded by an equals sign. All objects declared auto are allocated from the stack, but the memory used for these allocations is freed when the function in which they are declared returns to its caller. The dynamic nature of this allocation makes it generally difficult to predict how much stack space is actually needed for a particular program. The stack size option on the command line allows the user to adjust the amount of memory reserved for the stack without having to recompile the program. The memory reserved for the stack affects the amount of memory available for dynamic allocation by the various library functions

described in Section 3.1. See Section 4.5 for more information about the structure of C programs.

<infile The second optional field names a file or device to which the standard input (stdin) is to be assigned. This option is useful only if the program being executed actually uses the standard input (that is, it processes text input using getchar or scanf or makes explicit getc or fscanf calls using stdin). The file or device name must be immediately preceded by a < character; if a file, the full name including extension, if any, and pathname, if any, must be specified. See Section 5.2 for a list of valid device names. The file must exist, or the program will be aborted with the error message Can't open stdin file.

>outfile The third optional field names a file or device to which the standard output (stdout) is to be assigned. This option is useful only if the program being executed actually uses the standard output (that is, it generates text output using putchar or printf or makes explicit putc or fprintf calls using stdout). The file or device name must be immediately preceded by a > character; if a file, the full name including extension, if any, must be specified. See Section 5.2 for a list of valid device names. The file is opened as a new file, which discards its previous contents if they already existed and creates an empty file. If the filename specified is invalid or not enough directory space is available to create the new file, the program is aborted with the error message Can't create stdout file.

If two > characters are used instead of one, the file is opened for appending, and any output is added on to the end of the file. This option is useful for accumulating logging information. The file is created if it does not exist.

args Any additional fields beyond the program name and the three optional fields are extracted and passed to the function main as two arguments:

```
main(argc, argv)
int argc;       /* number of arguments */
char *argv[];   /* array of ptrs to arg strings */
```

Each arg string is terminated by a null byte. On most systems which support C, argv[0] is the name by which the program was invoked. Unfortunately, under MS-DOS the program name is not readily available, although all of the other information from the command line is. A dummy argv[0] is therefore supplied (all programs are named c according to argv[0]) but subsequent elements of argv are defined properly. Arguments appear in argv

in the same order in which they were found on the command line. Note that the optional stack and file specifiers are not included in the argv list of strings.

Although all of the above features are intended as conveniences for writing utility programs under MS-DOS, many of the library I/O functions are forced to be a part of the program because of this processing (specifically, the opening of the buffered input and output files). For programs which were going to use the buffered I/O functions anyway, this does not present a problem, even though these functions add a substantial number of bytes of code to the size of the linked program. Users who must be concerned about program size and who are not using these functions can avoid including the extra modules by supplying a special version of _main, the library function which calls main. See Section 5.4 for details.

EXAMPLES

```
CPROG =8000 <INPUT.R PQP 12
```

This command executes CPROG.EXE, sets the stack size to 8000 decimal bytes, and connects stdin to the file INPUT.R. The main function will be supplied an argc value of 3, with strings c, PQP, and 12 in the argv array.

```
errlog >>errors.log data
```

This command executes ERRLOG.EXE with stdout connected to ERRORS.LOG for appending (adding to the end of file). The main function will be supplied with an argc value of 2, with strings c and data in the argv array.

4.1.5 Function Extract Utility

Because the compiler generates a single, indivisible object module for all of the functions defined in a source file, the Function Extract Utility (FXU) is provided so that groups of small functions may be kept together in a single source file and object modules produced for them individually. The utility operates by extracting the source text for a single, specified function, thus creating a source module which can then be compiled to produce an object module defining only that specific function.

Those who are somewhat puzzled by the need for this utility may find the following example helpful. Suppose that one user has a module called STRING.C, which defines several string handling functions, and that a program calls one of those functions (say, strcnt). If STRING.C is compiled as a single source module, the resulting object module defines strcnt along with several other functions. When the program is linked, then, the machine code for strcnt is included (as part of the object module produced

when STRING.C was compiled), but the code for all of the other functions is included as well, even though the program does not make use of them. Only by compiling strcnt as the only function defined in its source module will the compiler produce an object module which defines only that function. FXU can be used to produce such a source file.

The format of the command to invoke the Function Extract Utility is

FXU [<header-file] [>output-file] filename function<enter>

The various command line specifiers are shown in the order they must appear in the command; optional specifiers are shown enclosed in brackets. The first two options are part of the general command line options for all C programs (see Section 4.1.4).

<header-file The first option specifies a file which will be copied to the output file when the specified function is found. The entire file is copied before any text from the function is written. If only the function itself is to be written to the output file, the **<NUL** option should be used. If this option is omitted, text will be read from the user's console and copied to the output file until a control-Z is typed.

>output-file The second option specifies the output file which will contain the text of the extracted function (preceded by the header file text, if any). If this option is omitted, text is written to the user's console.

filename Specifies the name of the file containing the function to be extracted.

function Specifies the name of the function to be extracted from the specified file. The function name must be specified exactly as it appears in its definition, except that alphabetic characters may be specified in either upper or lower case.

The Function Extract Utility counts braces defined in the body of the functions in order to determine when it has reached the end of a function. Although it recognizes comments and will not make the mistake of counting any braces which might be enclosed in them, it assumes that comments can be nested, which is the same assumption normally made by the compiler. The compiler, however, can be requested by command line option to process comments as if they did not nest; FXU has no such option.

The text extracted consists of all the characters between the closing brace of the preceding function, up to and including the closing brace of the extracted function. If the specified

function is the first one defined in the source file, then all characters from the beginning of the file to the function's closing brace are included. Note that functions which refer to external data items defined in the source module cannot be easily processed with the function extract utility. As the example below illustrates, however, the header file option can be used to avoid this limitation.

If the specified function is not encountered in the specified source file, the output file will receive the single error message Named function not found. Note that FXU works on only a single function, not a list of functions. A source module defining more than one extracted function can be generated, however, by executing FXU repeatedly and then combining the extracted texts using the CAT program, which is supplied as an example source file.

The supplied version of FXU uses an internal buffer to store characters between functions, while it scans for the next. The buffer size can be expanded, if necessary, by a simple modification to the source text, which is supplied as FXU.C.

EXAMPLES

FXU <NUL STRING.C strcnt

This command extracts the function called strcnt from the text file STRING.C and causes the extracted text to be written to the user's console.

FXU <IOS.H >INPUT.C IOFUNC.C input

This command extracts the function called input from the text file IOFUNC.C, prepends the output with the text from the file IOS.H and writes the resulting text to INPUT.C. If each function in IOFUNC.C can refer to the external locations flag1 and flag2, for example, and needs the information from the standard I/O header file, then IOS.H should include the text

```
#include <stdio.h>
extern int flag1, flag2;
```

A similar technique can be used for functions which need more extensive external references.

4.1.6 Object Module Disassembler

For programmers who wish to debug C modules at the machine code level, the Object Module Disassembler (OMD) provides a listing of the machine language instructions generated for a particular C source module. If the module is compiled with the *-d* option so that line number/offset information is included in the object file, the disassembler utility can produce a listing with interspersed source code lines. This listing can then be used in

association with the link map for the program to perform interactive debugging using Microsoft's DEBUG.

The format of the command to invoke the object module disassembler is

```
OMD [>listfile] [options] objfile [textfile]
```

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

>listfile The first option is used to direct the listing produced by OMD to a specified file or device. If this option is omitted, the listing output is written to the user's console.

options Four override options can be specified; each consists of a hyphen followed by a single letter which indicates the value to be overridden, and a string of decimal digits specifying the override value. There must be no embedded blanks in any single option, but each must be specified as a separate field. The valid options are:

-Pnnn Overrides the default size provided for the program section of the object module being processed. nnn specifies a decimal number of bytes of storage to be allocated for the program section. The default value is 1024 bytes.

-Dnnn Overrides the default size provided for the data section of the object module being processed. nnn specifies a decimal number of bytes of storage to be allocated for the data section. The default value is 1024 bytes.

-Xnnn Overrides the default maximum number of external items which can be processed by OMD; this number applies separately to both external definitions and external references. nnn specifies a decimal number of external items which can be processed. The default value is 200.

-Lnnn Overrides the default size for the line number and offset information tables. These tables are used only if the object file was produced with the -d option; line number/offset information from the file is placed in these tables. The default size (which defines the maximum number of line number/offset pairs which can be processed) is 100.

objfile Specifies the name of the object file, produced by the compiler, which is to be processed by OMD. The full name including the .OBJ extension must be specified.

textfile Specifies the name of a C source code file which is to be listed along with the disassembled instructions. If this option is present, the object file must have been compiled using the -d option for the LC1 command. The full name including the .C extension must be specified.

OMD processes only a single object module. The entire module is read and loaded into memory before the listing is generated. The various override options are useful for processing very large object modules, or for reducing the amount of memory needed by OMD on systems which are cramped for memory.

If the textfile option is used, only the source text from the specified file is listed; if it refers to any #include files, they will not be listed. Some limitations of the textfile option should be noted. First, the code generated for the third portion of for statements is placed at the bottom of the loop; that code will appear in front of the next statement after the end of the loop. Second, the compiler tends to defer storing registers until the last possible moment, so that the code shown for assignment statements often consists merely of loading values into registers; the registers will be stored later. Finally, the code generated for entry to a function will often be displayed in front of the source lines defining that function. Thus, inspection of the surrounding code may be necessary to determine the actual code generated for a source file construct.

EXAMPLES

OMD -P2048 -D8000 QRS.OBJ

This command disassembles the object module QRS.OBJ and writes the listing to the user's console; it causes 2048 decimal bytes of storage to be allocated for the program section defined in the object module, and 8000 decimal bytes for the data section.

OMD >TEMP.LST -X400 XYZ.OBJ XYZ.C

This command disassembles the object module XYZ.OBJ and writes the listing to the file TEMP.LST; it causes the source code lines from XYZ.C to be placed in the listing, provided that line number and offset information is present in the object file. It also provides for a maximum number of 400 external items (same limit for both external definitions and external references).

ERROR MESSAGES

A variety of error conditions are detected by the Object Module Disassembler; all cause early termination of the output file and result in the writing of an appropriate error message to stderr. These messages are self-explanatory for the most part. If one of the run-time-specifiable options is not sufficiently large, the

error message will indicate the specific option which was not large enough; for example, if the module defines too many words of program section, the message

```
Program section overflow
```

will be produced. Note that OMD was designed specifically for use with modules generated by the C compiler; attempts to use it with other object modules will probably cause an error message to be generated.

4.2 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 8086/8088. See Section 2 of the manual for a description of the machine-independent features of the Lattice implementation of the language.

4.2.1 Data Elements

The standard C data types are implemented according to the following descriptions. The only data elements which free alignment to a word offset are pointers, structures, and unions; as noted in Section 4.1.2, this alignment can be disabled by a compile time option. In all cases, regardless of the length of the data element, the low order (least significant) byte is stored first, followed by successively higher order bytes. This scheme is consistent with the general byte ordering used on the 8086, and with the memory formats expected by the 8087 numeric data processor. The following table summarizes the characteristics of the data types:

Type	Length in Bits	Range
char	8	0 to 255 (ASCII character set)
int	16	-32768 to 32767
short	16	-32768 to 32767
unsigned	16	0 to 65535
long	32	-2E9 to 2E9
float	32	+/- 10E37 to +/- 10E38
double	64	+/- 10E-307 to +/- 10E308

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

int defines a 16-bit signed integer; short and short int are synonyms.

unsigned or
unsigned int — defines a 16-bit unsigned integer. Note that in this implementation, unsigned is not a modifier but a separate data type.

long or
long int — defines a 32-bit signed 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 are either two bytes or four bytes in length, depending on the memory addressing model used. See Section 4.4 for more information.

4.2.2 External Names

External identifiers in the MS-DOS implementation differ from ordinary identifiers in one important respect: the MS-DOS linker treats upper and lower case letters as if they were the same. This means that, although the compiler will consider **main** and **MAIN** to be two different functions, the linker will not. External names may be up to 8 characters in length, and the underscore is a valid character. Since the compiler always assumes that external names have the same characteristics as ordinary identifiers, programmers must be careful not to define external names which the compiler believes are different but which the linker will interpret as the same name. A safe rule is to use lower case letters only for all externally visible items, including functions and data items which are to be defined for reference from functions in other source files.

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.

XC**** Low-level operating system interface functions (written in assembly language) are defined with XC 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.

4.2.3 Include File Processing

Include files may be specified as:

```
#include "filename.ext"
```

or

```
#include <filename.ext>
```

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, as was done with the header files included with the compiler package. Alphabetic characters in a filename may be specified in either upper or lower case. The file must be present in the default drive/directory unless a drive specifier or pathname is included in the filename (not recommended). The -i option (see Section 4.1.1) is the recommended method for specifying a different drive and/or directory path. The filename is retained internally by the compiler for error reporting (see Section 4.3.3).

4.2.4 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 16-bit signed comparisons correctly include overflow in determining the relative size of operands. Division by zero generates an interrupt which is processed by MS-DOS; on the operating system used to develop the compiler, the message Integer overflow is generated and execution of the offending program aborted. 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, division, and comparison. Product overflow is ignored. Division by zero yields a result of zero. The sign of the remainder is the same as the sign of the

dividend. Comparison is signed but does not take account of overflow.

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

1. char objects are unsigned in this implementation. Sign extension is not performed during expansion to int; instead, the high byte is simply set to zero. Code sequences such as

```
char i;
  . . .
  for (i=8; i >= 0; i--)
```

will not work (in this case, the loop never terminates).

2. Conversion of int or short to long causes sign extension. The inverse operation is a truncation; the result is undefined if its absolute value is too large to be represented.

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

4. 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.

4.2.5 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 short real and long real formats expected by the 8087 numeric data processor (the formats are described in Section 4.2.1). 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 functions. These functions do not make use of the 8087, although the floating-point formats are compatible with the 8087. Note that these functions were designed for accuracy, not speed, using straightforward, unsophisticated algorithms.

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 library file LCS.LIB (and LCP.LIB, LCD.LIB, and LCL.LIB) simply ignores all error conditions. You may write a different version (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.

Consult the 8087 description for more information about the floating-point formats.

4.2.6 Bit Fields

Bit fields are fetched on a word basis, that is, the entire word containing the desired bit field is loaded (or stored) even if the field is 8 bits or less in size. Bit fields are assigned from left to right within a machine word; the maximum field size is 15 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 : 5;
     unsigned y : 4;
     unsigned z : 3;
     } a;
```

then a occupies a single 16-bit word, a.x resides in bits 15

through 11, a.y in bits 10 through 7, and a.z in bits 6 through 4. Because of the way bytes are ordered on the 8086, this results in a.y being split between the low and high bytes.

4.2.7 Register Variables

This version of the compiler does not implement register variables because of the comparatively limited number of registers available on 8086/8088 microprocessor. However, declarations using register are accepted if properly made. Storage is reserved for these objects as if they had been declared auto.

4.3 Compiler Processing

The Lattice C compiler under MS-DOS 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.

4.3.1 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. The intermediate file (also called the quad file) is reviewed as it is written, and locally common sub-expressions are detected and replaced by equivalent results. When the entire source program has been processed (assuming there are no fatal errors), 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, unless (1) the specified source file could not be found, or (2) a quad file with a .Q extension could not be created (owing to lack of directory space). The sign-on message identifies the version of the compiler which is being executed. The MS-DOS 2.0 implementation returns an exit code of zero if no errors were detected, and a code of 1 otherwise. This allows the use of if expressions in batch files, such as:

```
LC1 %1
if errorlevel 1 goto errs
```

See Section 4.3.3 for more information about error processing.

Note that the quad file is deleted if any fatal errors are detected.

4.3.2 Phase 2

The second phase of the compiler scans the quad file produced by the first phase, and produces an object file in the Intel 8086 format. This object code supports all of the necessary relocation and external linkage conventions needed for C programs (see Section 4.5.2 for details). A logical segment of code specifying the 8086 machine language instructions which make up the executable portion of the program is generated first, followed by a segment of data-defining code for all static items. 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 there may be fairly sizable pauses during which no apparent activity is taking place. In general, these delays should not last more than several seconds. Anything longer than a 30-second delay can safely be assumed to be a crash; 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, unless (1) the specified quad file could not be found, or (2) an object file with a .OBJ extension could not be created (owing to lack of directory space). When code generation is complete, the second phase writes a message of the form

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

to the standard output (usually the user's console). pppp indicates the size in bytes of the program or executable portion of the module generated, and dddd indicates the size in bytes of the data portion; both 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 .OBJ file requires more space because it contains additional relocation information.

As noted in the introduction to Section 4.1, the code generator produces a single .OBJ 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. As previously mentioned, a Function Extract Utility (FXU.EXE) is provided so that multiple functions may be stored in a single .C file and extracted individually for compilation; see Section 4.1.5.

4.3.3 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 and deletes the quad file just before it terminates execution. This prevents the second phase from attempting to generate code from an erroneous quad file. As mentioned in Section 4.3.1, under DOS 2.0 the compiler returns a zero if no errors are detected, and a 1 otherwise. 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.) All error messages are written to the standard output, which is normally the user's console but can be directed to a file if desired (see Section 4.1.1). 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 very simple-minded error recovery technique which may 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.

4.3.4 Code Generation

The code generation phase reads the quad file and builds an image of the instructions for each function in working memory, before writing the instructions to the object file. This implies that at least as much working memory must be present as is required by the largest function in the source file; actually, considerably more memory (as much as several times that size) is required

because of the additional overhead used by the compiler. Since the compiler uses the S memory model which has a 64K byte data space limitation, there is a definite limit to the size of a function which can be compiled even when the maximum amount of memory is available. Nonetheless, all of the compiler's own source modules -- some of which contain very large functions -- can be compiled without difficulty. In any case, C is a language which encourages modularity; most programs consist of numerous functions, most of them small. It is therefore doubtful that the function size limitation will prove to be a problem.

One reason for the extra overhead in buffering the function in memory is that 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 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 AX, BX, CX, and DX are used as general purpose accumulators, while SI and DI (along with BX) are used for access to indirect operands. BP is used to address the current stack frame; see Section 4.5.3 for more information. In the D and L memory addressing models, the ES segment register is used for indirect pointer references, see Section. 4.4.

In order to generate the most efficient code for the largest number of source language constructions, the compiler usually makes a favorable assumption about pointer variables. Specifically, it assumes that the actual objects accessed using pointer variables are not the same as other objects which can be accessed directly. This allows the compiler to avoid discarding register contents (thus forcing them to be reloaded, perhaps unnecessarily, at a later time) whenever a result is assigned using a pointer. Consider the following example:

```
int i, j, k, *pi;
. . .
  i = j+2;
*pi = j;
```

```
k = i*4;
```

In the general case, it is quite possible that pi might actually point to i, which would change the value assigned to k in the next statement. In the vast majority of C programs, however, i will be a local variable to which it is not possible for pi to point. The compiler normally makes this assumption, that is, that *pi cannot be equivalent to i, and therefore can retain the value computed in the first statement for i in a register, which saves having to reload it to perform the multiply operation in the third statement.

On the other hand, there are rare cases where this assumption is not valid. C programmers almost never code sequences such as:

```
pi = &i;
*pi = 12;
```

but more subtle cases of pointer overlap can occur, particularly when both the pointer and its target are externally defined. For these cases, the -a compile-time option is provided (Section 4.1.1); this forces the compiler to assume worst-case aliasing (which is compiler jargon for pointer overlap) when generating code. The compiler is so designed because instances of pointer overlap are more the exception than the rule. Thus, rather than default to worst-case assumptions that produce correct code in all cases and unnecessary inefficiencies in most cases, the compiler normally makes a favorable assumption that produces efficient code which works correctly in nearly all cases.

A final note on this subject: even when the -a option is used, the compiler assumes that only objects of the pointed-to type can be changed in pointer assignments. Thus, if an int pointer is used in an indirect assignment, only registers containing int values will be discarded.

4.4 Memory Addressing Models

The segmented architecture of the 8086 and 8088 processors presents special problems for the implementation of high level languages. In order to provide programmers with the ability to select the combination of efficiency and addressability needed for a particular application, the compiler supports four different sets of memory addressing assumptions, called memory models. Each is identified by a single capital letter, and reflects a different view of the addressing of functions and data within a C program. These views can be expressed by the limitation on the size of the respective space for program text (the functions) and data objects (all of the declared or allocated data structures), as follows:

Model	Program Address Space	Data Address Space
S	64K	64K
P	up to 1M	64K
D	64K	up to 1M
L	up to 1M	up to 1M

The D and L models use four-byte pointers, and the P and L models generate FAR calls and returns. The S and P models produce compact, efficient code limited to addressing a 64K data area, while the D and L models allow access to all of the 1 megabyte of available memory.

4.4.1 Choosing the Memory Model

The compiler is most efficient (both in terms of code size and execution speed) for the S model. All of the examples in previous sections have shown commands for compiling with the S model, and this memory model is particularly recommended for beginning C programmers.

All of the functions in a single program must be compiled and linked with one and only one of the available memory models. In other words, functions compiled for different models may not be combined. It becomes important, therefore, to choose the right memory model for the particular application. The tradeoff is between efficiency and memory addressability. There are two choices that must be made.

1. Will the combined size of the functions in the program be greater than 64K bytes? If not, one of the models that uses NEAR calls (the S or D models) should be selected, as these are faster and require less code. Otherwise, a model that supports FAR calls (the P or L models) should be selected.

2. Does the application require more than 64K bytes of data storage? If not, one of the models that uses 2-byte data pointers (the S or P models) should be selected, because pointer operations are performed much more efficiently in these models. If the program simply needs access to specific memory locations beyond the program's 64K address space, the library functions peek and poke can be used, allowing the program to retain the efficient 2-byte pointers. Otherwise, if data storage in excess of 64K bytes is a must, a model that uses the 4-byte data pointers (the D or L models) must be selected, even though this will produce somewhat less efficient code.

4.4.2 Compiling for the Memory Models

Generation of code for the various models is controlled by a compile-time option specified on the first phase (LC1) of the

compiler. The -m option must be followed immediately (no spaces) by a letter (either lower or upper case) specifying the desired memory model. The model may also be specified as a single numeric digit from 0 to 3. If no -m option is present, code is generated for the S model.

S model:	LCl filename	(no flags)
	LCl filename -mS	
P model:	LCl filename -mP	
	LCl filename -m1	
D model:	LCl filename -mD	
	LCl filename -m2	
L model:	LCl filename -mL	
	LCl filename -m3	

4.4.3 Linking Programs

When using the various memory models, care must be taken to link with the appropriate library (LCS.LIB, LCP.LIB, LCD.LIB, or LCL.LIB). The compiler generates code segments with different names for each model, which allows examination of the LINK map to determine if code for different models has been erroneously mixed. Only one of the following segment names should appear on the link map.

S model:	PROG	(code group PGROUP)
P model:	_CODE	
D model:	CODE	(code group CGROUP)
L model:	_PROG	

Note that for the P and L models, several segments with the name _CODE (or _PROG) will be included (one for each separately compiled module containing functions).

4.4.4 Code Generation for Pointer Operations

In the S and P models, pointers to the various data types consist of the 16-bit offset of the low order (least significant) byte of the data element. Since the combined size of the data elements in these models cannot exceed 64K bytes, the address of an item is fully specified in 16 bits. Indirect data references are made by loading the pointer into one of the indexing registers SI, DI, or BX.

Function pointers differ in each of the memory models. In the S and D models, pointers to functions consist of the 16-bit offset of the first byte of the code defining the function. In the S model, this pointer is stored in two bytes, while in the D model, it is stored as the first two of four bytes (the last two are zero since they are not used, but are merely required to conform to the four-byte size of other pointers in the D model). In the P model, a two-byte pointer (required because of the two-byte size

of data pointers) is used to store the offset of a four-byte function address contained in the data section. This function pointer has the same format as function pointers in the L model, where the first two bytes contain a 16-bit offset and the next two contain the 16-bit segment base for the function.

The code generated by the D and L models uses four-byte pointers and can therefore address any location in memory. These pointers are stored as an offset portion in the low two bytes, followed by a base portion in the high two bytes (the format expected by the machine language instructions LDS and LES). Objects are addressed from these pointers by loading the base portion into the extra segment register ES, the offset portion into an index register, and using the segment override prefix for ES to force the indexed operation to refer to the correct memory location. Since there is only one ES register, such common operations as copying from one pointer to another require ES to be reloaded for each step in the copying process. Pointer references are therefore less efficient than in the 2-byte memory models.

The four-byte pointers used in the D and L models are manipulated according to the following rules:

1. Pointer arithmetic is performed by adding or subtracting a 32-bit offset to the pointer, using a call to a library routine. Thus, dynamically allocated arrays (addressed by subscripting a pointer variable) may be larger than 64K, and address manipulations work properly for all offset values. Note that, since the compiler requires statically declared arrays (extern, static, or auto) to be less than 64K bytes in size, only a 16-bit offset is used in accessing elements of these arrays, resulting in more efficient code.

2. When two 4-byte pointers are subtracted, a library routine is called which returns a long result.

3. Conversions between long integers and 4-byte pointers are automatically performed, again by calling library routines.

4. Comparison of pointers for equality or relative rank is performed by calling a library routine which converts the pointers to normalized (canonical) form before comparing. Thus, two pointers which have different base and offset portions, but which actually point to the same location will be recognized as equal.

5. Any function which returns a pointer as its return value calls a library routine which converts that pointer to normalized (i.e., offset in the range 0 to 15) form.

4.4.5 The -s Option for Four-byte Pointers

While the above rules generally describe use of four-byte pointers, the additional overhead of library routine calls can be inefficient if a significant amount of pointer manipulation is being performed. A special compile-time option (specified on LC1) is provided for knowledgeable users who are willing to work within certain restrictions. Adding the -s flag to LC1 causes the following changes in the above rules:

1. Pointer arithmetic is performed by adding or subtracting a 16-bit offset to the pointer. Thus, no single object may be greater than 64K bytes in size.

2. Pointer arithmetic affects only the offset portion of the pointer (not the base). When pointers are compared for equality, an exact match of both base and offset portions is required. When compared for relative rank, only the offset portions are compared, so the comparison is meaningful only if they are pointers to the same array.

3. When two pointers are subtracted, only the offset portions participate in the operation, and the result is a short.

4. Pointers and long integers are not converted when one is assigned to the other; instead, a simple copy operation is performed.

5. The return value from a function which returns a pointer is not normalized.

Most functions can be safely compiled with the -s option, resulting in improved code generation quality. In fact, all of the library functions written in C supplied in the libraries are compiled with the -s option, except for the memory allocation functions. Note that the -s flag has no effect on the S and P models.

As noted above, the biggest potential problem when converting code to use the four-byte pointers of the D and L models is that pointers and integers are no longer the same size. While it may appear that a program's source code does not depend in any way on this fact, programmers must be alert for subtle problems that might relate to this. Here are three important cautions:

1. When supplying pointer arguments to C functions, it is common practice to supply a null pointer (i.e., one that does not point to anything) as the #define constant NULL, which is defined as 0 by stdio.h. When compiling code for the D or L models, NULL must be changed to 0L so that the null pointer value supplied to functions is the same size as the pointer argument. Failure to do this will cause the called function to incorrectly address its parameters, resulting in serious problems.

2. The sbrk memory allocator is supposed to return a value of -1 when no more memory is available (for compatibility with other implementations). Under the D and L models, the result of casting -1 into a character pointer depends on whether the -s option was used (see Sections 4.4.4). Since the library function was compiled without the -s option, the -1 gets converted to the four-byte pointer format. The result is that a function compiled with the -s option cannot properly test for the -1 value! All of these problems can be avoided by using the library function **lsbrk**, which accepts a long integer number of bytes and returns zero if no more space is available (see Section 3.1).

3. The four-byte pointers implemented under the D and L models allow direct access to all of the memory on the machine. This can be extremely useful, but it can also be extremely dangerous. Memory on the 8086 and 8088 processors is not protected, and storing values via an uninitialized pointer can crash the system -- or worse. MS-DOS stores a number of very important system elements in lower memory, so that use of an uninitialized pointer to store data can have disastrous consequences (such as destroying the File Allocation Table (FAT) for a hard disk!). Programmers should exercise extreme caution when using these memory models. Beginning programmers are advised to use the S or P models, where uninitialized pointers are much less likely to access critical locations.

4.4.6 Creating an Array Greater than 64K Bytes

Since static data in all of the memory models is limited to a maximum of 64K bytes, the only way to create an object of greater size is through the memory allocation functions.

Suppose that an array of 10,000 double precision values must be allocated; 80,000 bytes of storage will be required for such an array. First, a pointer must be declared which will contain the array's address after allocation:

```
double *d;
```

Note that a simple double pointer is all that is needed, despite the fact that it will actually point to an array. Next, the memory allocation function

```
char *getml();
```

must be declared.

Also note that the memory allocation function must be declared to return a pointer; otherwise, the compiler will assume it returns

an int and the cast operation shown below will not work correctly. The array is then allocated by the expression:

```
d = (double *) getml(80000L);
```

Note the L (el) specifier on the constant. The size could also be specified as (**10000L * sizeof(double)**). (Note: if the size argument for **getml** is computed using a multiplication expression, be sure that one of the operands is a long constant or is cast to a long before the multiplication; otherwise, the compiler will perform the multiplication in short arithmetic and obtain an incorrect result. If the example above is written as ((long)(**10000 * sizeof(double)**), the size argument is incorrectly computed as 14464!

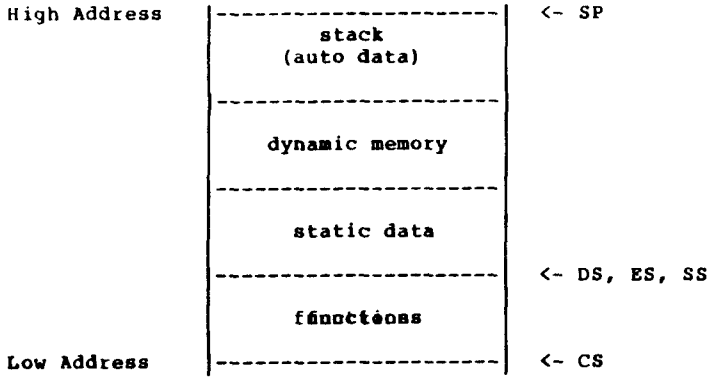
The returned pointer, of course, must be checked for NULL (zero) before use; NULL is returned if there is not enough memory available for the requested allocation. The variable d can now be subscripted as if it were an array, i.e., **d[12]** will address the thirteenth (13th) element of **d**, etc. In this example, the number of elements in the array is less than 64K, so ordinary int variables can be used as subscripts; if a char array had been allocated, long integers would be needed to subscript an array of this size. Also note that since an object with a size greater than 64K is being addressed, the -s option cannot be used.

4.5 Run-time Program Structure

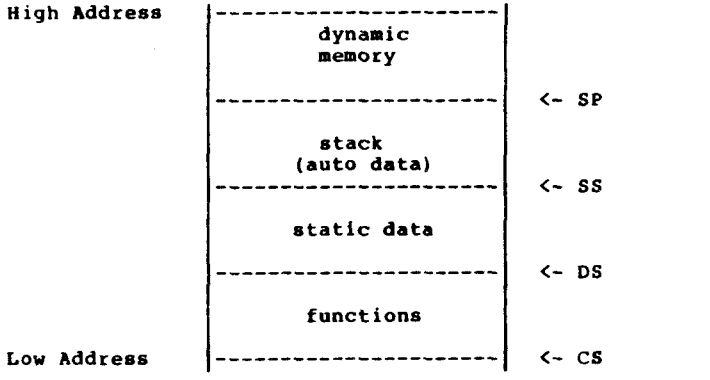
This section describes the structure of C programs under the 8086/8088 MS-DOS implementation of the Lattice C compiler. Some knowledge of the architecture of the 8086 processor and of the 8086 object code and linkage concepts is required in order to understand much of the information presented. Readers who are not interested in the precise technical details of the hardware implementation may safely skim through or skip over this section; it is primarily intended for programmers who must provide an interface between C and assembly language.

Postponing discussion of the specific object code details used to create it (see Section 4.5.1 below), the general structure of a C program is illustrated by the diagrams on the next page.

S and P Models



D and L models



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. Most implementations (including this one) support, through library functions, an additional dynamic memory allocation facility which returns pointers to objects not explicitly declared. The diagrams above show the way these allocations are made; as one might expect, the auto data items are allocated on the 8086 hardware stack.

Two different memory layouts are used, depending on the size of pointers. In either case, the functions are grouped together in the lowest portion of the address space defined by the program, and the static data items are grouped together immediately above the functions. In the S and P models, the segment registers **DS**, **ES**, and **SS** all contain the same value, which is the base segment address for all of the static data items in the program. The stack pointer SP is initialized to point to the highest available offset relative to this segment; this value is X'FFF0' if sufficient memory is available, and is adjusted accordingly if less than 64K bytes of memory remain above the data segment base. A certain number of bytes is reserved for the stack, which grows downward. The remainder of memory between the end of the static data items and the lowest address allotted to the stack is available for dynamic memory allocation using library functions. The stack overflow detection mechanism described in Section 4.5.5 can be used to prevent stack allocations from exceeding the allotted space and colliding with the dynamic memory pool or the static data items. The stack size override feature described in Section 4.1.4 allows the number of bytes to be reserved for the stack to be specified when a program is executed.

In the D and L models, the stack resides immediately above the static data area, and the free memory pool (allocated by the functions described in Section 3.1) is above the stack, which can be as large as 64K bytes. Segment register **DS** points to the base of the static data area, **SS** points to the base of the stack, and **ES** is undefined. The stack pointer SP is initialized to contain the number of bytes allocated for the stack (defined in _stack; see Section 4.5.5). As noted above, the stack overflow detection mechanism prevents collision with the static data elements.

4.5.1 Object Code Conventions

The object file created by the second phase is in the standard MS-DOS object code format, which is compatible with the Intel 8086 object module format. The object file defines the instructions and data necessary to implement the module specified by the C source file, and 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. In order to force the parts of the module into the proper locations after linking, the object file defines two logical segments which are marked for concatenation with other segments of the same name.

The program segment is the segment which includes the instructions which perform the actions specified by any functions defined in the source file. As noted in Section 4.4.3, the segment name used for the program segment depends on the memory model.

DATA is the segment which includes all static data items which are defined in the source file. This includes not only those data items explicitly declared **static** but also items declared

outside the body of a function without an explicit storage class specifier, string constants, and double precision constants. (Auto data items are simply allocated on the stack at run time and are not explicitly defined in the object file.)

The DATA segment is defined to be combinable with other segments of the same name. In the S and D models, the program segment is also made combinable. Program segments combine with byte-alignment, that is, as closely as possible; DATA segments combine with word-alignment. Thus, no space is wasted when functions are combined during linking, and the word alignment of elements within a particular DATA segment is preserved after combination. This alignment of data items is important for efficient data fetches on the 8086, where word fetches from an odd byte address require an additional four clock periods. Note that although a compile-time option (described in Section 4.1.1) allows the alignment requirement for data items within a particular module to be relaxed, the word alignment of DATA segments during linking is not affected.

The net effect of these segment definitions is to force, at link time, all functions to be collected together and all static data items to be similarly combined. This achieves the most important part of the program structure diagrammed above. The segment directives needed to combine assembly language modules with C modules are shown in Section 4.5.4.

4.5.2 Linkage Conventions

The 8086 group concept is used to guarantee that the data portion of the final linked program does not exceed 64K bytes; in the S and D models, it is similarly used to force the combined program section to fit into 64K. The groups which may be defined are:

PGROUP = BASE segment + PROG segment (S model)
CGROUP = BASE segment + CODE segment (D model)
DGROUP = DATA segment + STACK segment (all models)

The PROGRAM and DATA segments are obtained from the C modules in the program, as previously discussed. The other two segments are defined in the startup module (CS.OBJ, CP.OBJ, CD.OBJ, or CL.OBJ), which must be the first module encountered during linking. The BASE segment serves two purposes: (1) it forces PGROUP (or CGROUP) lower in memory because it is the first segment within the startup module, and (2) it contains a byte which identifies the memory model used. The latter feature allows programs to be examined with a program debugger to determine the memory module used when the program was linked. The STACK segment has a dual role as well: (1) it defines the base of the stack and dynamic memory portion of the data section of the program, and (2) it satisfies the linker's need for a segment of type STACK (if one is not encountered, the linker generates a warning message).

The startup module (CS.OBJ, CP.OBJ, CD.OBJ, or CL.OBJ) also defines its own program and DATA segments. The PROGRAM segment defines the initial execution address of the linked program. The segment registers are initialized, and the amount of memory remaining above the STACK segment is determined. The stack pointer is adjusted to its initial value, as noted in the discussion in Section 4.5. In the DATA segment of the startup module, the address of the stack base and top are saved for use by the memory allocation functions. At the top of the stack, the address of the program segment prefix is saved so that an orderly return to MS-DOS can be made when the program terminates. The characters from the command line which executed the program are transferred from the program segment prefix to the stack. A pointer to this copy of the command line is then passed to the function _main, which begins execution of the program (see Section 5.4).

As noted in Section 4.2.2, external names differ from ordinary identifiers in C in that upper and lower case letters are equivalent. All external names are defined as an unspecified type, that is, there is no set of attributes associated with the name; it is simply an offset within one or the other of the two defined groups. 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 programs refer to external names in a consistent way (i.e., a function should not refer to xyz as long when it is actually defined as int in some other module). External definition and reference from assembly language modules are discussed in Section 4.5.4.

See the appropriate linker documentation for information on how to obtain a public symbol map for a linked program. As a convenience, the DGROUP segments are defined with class name DATA, the PGROUP segments with class name PROG, and the CGROUP segments with the class name CODE.

4.5.3 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. A NEAR call (which changes IP but not CS) is used in the S and D models; a FAR call (which changes both IP and CS) is used in the P and L models. The argument values are pushed in reverse (right-to-left) order because the stack grows downward on the 8086; this allows the called function to address the arguments in the natural left-to-right (low-address-to-high-address) order. The first actions taken by the called function are:

1. The BP register is pushed onto the stack; this saves the value of BP used by the caller.

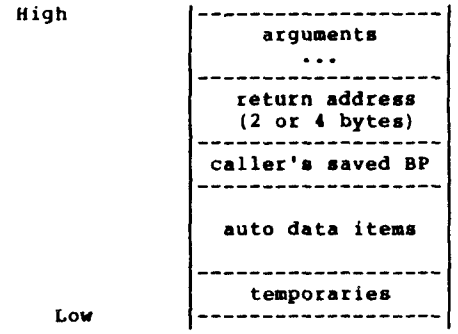
2. The stack pointer SP is reduced (i.e., a value is subtracted from it) by the number of bytes of stack

space required by the called function. This value is rounded to the nearest word so that the stack pointer is always word-aligned. The stack space includes all auto data elements declared in the function, and also may include additional space for the temporary storage locations which are often required during expression evaluation. If no auto items or temporaries are needed, this step is skipped; SP is unchanged.

3. If the function was compiled with stack overflow detection, it then checks the stack pointer for a legal value, as described in Section 4.5.5. If stack overflow is detected, control is routed to the entry point XCOVF (defined in the startup module) by means of a jump instruction.

4. The stack pointer SP is moved into BP to allow addressing of the elements on the stack: function arguments, auto storage, and temporaries.

he offsets of the various components are indicated by the ollowing diagram. Note that of the registers used by the alling program, only BP is saved.



uring execution of a C function, BP and SP normally contain the ame value. The temporaries are allocated closest to BP, ollowed by the auto elements declared, in the order of their eclaration. This addressing scheme has the disadvantage that he arguments supplied to the function are at an offset etermined in part by the amount of auto storage declared. If he function declares more than approximately 124 bytes of auto torage, the arguments require an additional offset byte in the nstructions which refer to them.

he compensating advantage to this mechanism appears when a unction calls another function and supplies it with argument alues. Because a C function may in special cases have a ariable number of arguments (printf is the classic example), the alled function cannot deallocate the stack space used in pushing

the argument values; the calling function must do so. By retaining the normal SP value in BP, Lattice C functions can restore the stack pointer after a function call with the two-byte instruction:

```
MOV  SP,BP
```

If BP is not set up in this way, a value must be explicitly added to SP, which requires a three- or four-byte instruction.

A second advantage to this technique is that it is easy to implement assembly language functions (to be called from C) with a variable number of arguments. Since the caller's BP contains the value in SP before argument values were pushed (as the diagram shows), it defines the upper limit for the address of any arguments. In other words, only the space between the saved return address and the address in the caller's BP register can contain arguments.

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:

16 bits	AX register
32 bits	(AX,BX) register pair
64 bits	(AX,BX,CX,DX) register quadruplet

In the multiple register returns, AX contains the high order bits of the value. Note that in the D and L models, this means that the segment portion of a pointer return value is contained in BX and the base portion in AX.

After the return value is loaded, the function adds to SP the same value that was subtracted on entry. Then BP is popped, restoring the caller's base pointer, and a near return is executed. The calling function now regains control, and must restore SP if any argument values were pushed.

4.5.4 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. In order to facilitate assembly language programming for the various memory models, four macro libraries have been provided as part of the compiler package. These libraries define values for symbols which can be tested for conditional assembly purposes, including:

LPROG	1 for P or L model, 0 otherwise
DATA	1 for D or L model, 0 otherwise
8086	1 for S model, 0 otherwise
8086	1 for P model, 0 otherwise
8086	1 for D model, 0 otherwise
8086	1 for L model, 0 otherwise

Also defined are four special macros, used to delimit the beginning and end of the program and data segments:

PSEG	defines start of program segment
ENDPS	defines end of program segment
DSEG	defines start of data segment
ENDDS	defines end of data segment

In order to use these symbols and macros, an INCLUDE statement must be used at the beginning of the assembly language module. For example:

```
INCLUDE   DM8086.MAC
```

makes available the symbol and macro definitions for the D memory model. By using the generic include file named DOS.MAC, an assembly language module can be assembled under any of the models, as long as the appropriate header file is copied to DOS.MAC before the module is assembled.

An assembly language module which defines one or more functions to be called from C must define the start of the program segment:

```
          PSEG
```

followed by PUBLIC declarations of the functions:

```
          PUBLIC    AFUNC, ...
```

followed by the functions themselves:

```
AFUNC     PROC      NEAR/FAR
          ...
          ENDP
```

Note that the PROC statement must define the function NEAR in the S and D models, and FAR in the P and L models. The function must conform to the conventions detailed in Section 4.5.3. If a value is to be returned by the function, it must be placed in the appropriate register(s).

To call a function from assembly language, an EXTRN declaration must be included for that function, and the caller must supply any expected arguments in the proper order (see Section 4.5.3). Note that the position of the EXTRN statements for functions is critical: for the P and L models, they must appear before the program segment definitions:

```
          EXTRN     XYZ:FAR,
          PSEG
AFUNC     PROC      FAR
```

For the S and D models, they must appear after the definition of the program segment:

```
        PSEG
        EXTRN   XYZ:NEAR, ...
AFUNC   PROC    NEAR
```

An assembly language module may also define data locations to be accessed (using "extern" declarations) from C programs:

```
        DSEG
        PUBLIC  DX1,DX2,DX3
DX1     DW      4000H
DX2     DW      8000H
DX3     DB      'Text string',0
        ENDDS
```

Note that if the address of an item is to be defined, the name must be prefixed with the group name if it is used as the operand of the OFFSET operator or of the DW or DD statements. If DX4 is used to define the address of DX1 in the example above, it must be coded:

```
DX4     DW      DGROUP:DX1
```

Otherwise, a segment-relative offset is generated, which will not be the actual address of the item as it is defined within the context of a C program. (Note: the prefix is not required for the LEA instruction, which refers to the current ASSUME directive.)

Similarly, to refer to data elements defined in a C module, include appropriate EXTRN statements:

```
EXTRN   XD1:WORD,XD2:BYTE
.
.
.
MOV     AX,XD1
```

Note that any EXTRN statements for data elements must be defined within a DATA segment declaration like the one shown previously. The BYTE attribute must be used for external char items. If an element is larger than a word, a STRUC can be used to define it, or its offset can be loaded into an index and used to fetch its component parts. The same caution about addresses requiring a group prefix applies to an external reference. For example:

```
DW      DGROUP:XD1
```

must be used to define the address of XD1. Otherwise, a segment-relative offset is generated, which will not be the actual address of the item as it is defined within the context of a C program. Note that the prefix is not required for the LEA instruction, which refers to the current ASSUME directive.

Upper and lower case letters for external names (and for all symbols within assembly language modules) are equivalent, so an assembly language function **XYZ** can be called from C as either **XYZ** or **xyz**.

Assembly language functions need to preserve BP only, as the compiler does not make any assumptions regarding register contents following a function call (except for return values).

As noted above, DS always points to the base of static storage for any of the memory models, so assembly language functions must be careful not to change DS. In the S and P models, **ES** must also be preserved (but not in the D and L models).

Note the differences between the use of pointers in the S and P models from their usage in the D and L models, as in the following example:

```
     S/P model:          MOV  BX,[BP].ARG1   ;get a pointer arg
                         MOV  AX,[BX]        ;use it

     D/L model:          LES  BX,[BP].ARG1   ;get offset and base
                         MOV  AX,ES:[BX]     ;use it
```

The following example (supplied with the compiler package as the file IO.ASM) illustrates many of the above requirements, and also demonstrates the use of DOS.MAC and conditional assembly for the different memory models.

```
        TITLE   PORT I/O FUNCTIONS
        SUBTTL  Copyright 1982 by Lattice, Inc.
        NAME    PORTIO
        INCLUDE DOS.MAC

        IF      LPROG
X       EQU     6               ;OFFSET OF ARGUMENTS
        ELSE
X       EQU     4               ;OFFSET OF ARGUMENTS
        ENDIF

        PSEG
;**
;
; name          inp -- input byte from port
;
; synopsis      c = inp(port);
;               int c;          returned byte
;               int port;       port address
;
; description   This function inputs a byte from the specified por
;               address and returns it as the function value.
;
;**
        PUBLIC  INP
        IF      LPROG
```

```
INP     PROC    FAR
        ELSE
INP     PROC    NEAR
        ENDIF
        PUSH    BP              ;SAVE BP
        MOV     BP,SP
        MOV     DX,[BP+X]       ;GET PORT ADDRESS
        IN      AL,DX           ;GET INPUT BYTE
        XOR     AH,AH           ;CLEAR HIGH BYTE
        POP     BP
        RET
INP     ENDP
        PAGE
;**
;
; name          outp -- output byte to port
;
; synopsis      outp(port,c);
;               int port;       port address
;               int c;          byte to send
;
; description   This function sends the specified character to
;               the specified port.
;
;**
        PUBLIC  OUTP
        IF      LPROG
OUTP    PROC    FAR
        ELSE
OUTP    PROC    NEAR
        ENDIF
        PUSH    BP              ;SAVE BP
        MOV     BP,SP
        MOV     DX,[BP+X]       ;GET PORT ADDRESS
        MOV     AX,[BP+X+2]     ;GET OUTPUT BYTE
        OUT     DX,AL
        POP     BP
        RET
OUTP    ENDP
        ENDPS
        END
```

4.5.5 Stack Overflow Detection

The compiler, by default, generates code at the beginning of each function to check for stack overflow. The cost in code size for each function is 9 bytes for the S and D models, and 11 bytes for the P and L models. The benefit is elimination of a very nasty class of errors which can be very difficult to find. When stack overflow is detected, the error message:

```
*** STACK OVERFLOW ***
```

is written to the console, and the program terminates immediately.

Stack overflow occurs when the program fails to supply sufficient storage for the run-time stack. The number of bytes of storage for which the stack is set up is defined in the external location _stack, and can be changed when the program is executed by the =nnn option on the command line. The size of the stack can thus be set in any of three ways:

1. If no definition for _stack is found in the user's object modules during linking, the Lattice C library provides a definition of _stack containing 2048 (2K). Thus, the default stack size is 2048 bytes.

2. If one of the user's object modules includes a definition for _stack, that value will be used. All that is required is that a statement such as

   ```
   int _stack = 4096;
   ```

   appear outside the body of a function. That value then becomes the default stack size.

3. Either one of the above methods can be overridden at execution time (after linking) by executing the program with a command such as

   ```
   PROGNAME =8000
   ```

   The decimal value after the equals sign becomes the stack size during execution of the program.

Unfortunately, there is no hard and fast rule for determining how much stack space a program will need. At least as much storage as the largest amount of auto storage declared in any of the functions included in the program will be needed (i.e., if a function has an auto array of 4000 bytes, at least that much stack space is needed, because auto data items are allocated on the stack). Since C functions typically call other functions, the storage needed by the called function must be added to that needed by the caller, and so on. The intention in supplying the various setting mechanisms described above is to make the stack size easily adjustable.

The code for stack overflow detection can be eliminated by compiling your source file with the -v option on LC2 (Section 4.1.2). Library functions are supplied with stack overflow detection included.

SECTION 5:
System Library Implementation

Although the portable library functions described in Section 3 of this manual define a general purpose interface to the typical environment provided for C programs, there are inevitably many details and variations which are system-dependent. In this section, some of the details of the MS-DOS library implementation are presented in order to clarify the peculiarities of this particular environment.

Fortunately, MS-DOS supports a number of powerful features which allow a full implementation of the standard file I/O functions, although the representation of text files presents a minor problem; Section 5.1 discusses file I/O. Several standard device names are also supported by MS-DOS, and the Lattice C I/O interface processes these in special ways, as explained in Section 5.2. The structure of Lattice C programs (see Section 4.5) allows the full set of memory allocation functions, although care must be taken to provide sufficient space for the stack, as Section 5.3 warns. The basic program entry and exit functions are described in Section 5.4, and some special functions unique to the MS-DOS implementation are presented in Section 5.5. As additional functions will probably be provided as the compiler evolves, the programmer should check the addendum for the current version of the compiler.

5.1 File I/O

Filenames are specified according to the following format:

```
d:\pathname\filename.ext
```

where d: is an optional drive specifier, pathname is an optional directory specifier, filename is the name of the file, and .ext is the file extension. If the drive specifier is omitted, the currently logged-in disk is used; if the pathname is omitted, the current directory is used. The filename is specified without trailing blanks if less than 8 characters; the extension (including the ".") must be omitted if one is not defined for the file. Alphabetic characters may be supplied in either upper or lower case; actual filenames use upper case letters only. Only those characters which are legal for filenames under MS-DOS are acceptable; consult the MS-DOS documentation for details. Certain names are recognized as devices rather than files; see Section 5.2.

The level 1 I/O functions perform disk I/O by making direct calls to MS-DOS, so that all buffering is performed by the operating system. Programs using the level 2 I/O functions cannot use the rbrk function, because fopen allocates a buffer using getmem.

In the MS-DOS implementation, both the level 2 (fopen, putc, getc, fclose) and the level 1 (open, creat, read, write, close)

I/O functions are limited to 20 open files, including devices, and including the three (stdin, stdout, stderr) which are automatically opened for the main program. Note that the number of files available under MS-DOS version 2 is also affected by the CONFIG.SYS file, since an MS-DOS file handle is used for every open file (see MS-DOS documentation).

The portable library provides a system-dependent option when a file is opened or created; the programmer may select one of two modes of I/O operation while a file is open. On some systems the modes are in fact the same, but in the MS-DOS implementation they differ in some important details.

Translated or text mode is the default condition. In this mode, the line terminator normally used by C programs (a single newline character, \n or 0x0A) is translated to the MS-DOS line terminator, which consists of the two characters, carriage return and linefeed (0x0D followed by 0x0A). This translation is performed when the file is written using calls to the write library function; the inverse translation is performed when the file is read using the read library function. Programs which use the higher level I/O functions (putchar, getchar, printf, etc.) are usually not affected, but programs which call read and write directly must beware of these translations. On read calls, the count returned may be less than the actual number of bytes by which the file position was advanced (because of CR deletions). Note that all carriage returns are discarded when reading from a file in the translated mode; similarly, all linefeeds are expanded to CR/LF when writing to the file.

Untranslated or binary mode is an option which can be selected when the file is opened or created. By adding 0x8000 to the mode for the open call or to the access privilege mode word for the creat call, the programmer indicates that read/write operations on the file are to be performed without translation. In this mode, bytes are transferred between the caller's area and the file without modification. This option must be used for files containing binary data; otherwise data bytes which happen to take on CR and LF values will be translated incorrectly.

In addition to the file I/O modes discussed above, two other functions should be clarified under the heading of file I/O. The creat function gets a system-dependent argument, the access privilege mode bits; these are ignored under the MS-DOS implementation, except for bit 15 (the 0x8000 bit) which, if set, causes the file to be accessed in untranslated or binary mode. The lseek function has an offset mode, not always implemented, which specified an offset relative to the end of file. Because MS-DOS retains the exact file size in its directory, this mode can be and is implemented in this version.

5.2 Device I/O

Several special file names are checked for by the Lattice I/O interface under MS-DOS, and processed using single character

reads and writes. These device names may be specified in either upper or lower case, but will be recognized by the level 1 open and creat functions only if the trailing colon is supplied. If the colon is omitted, the name is passed to the operating system and may be processed specially by it; however, the level 1 functions will deal with a device reference sans colon as if it were a reference to a disk file. The device names recognized are as follows:

Console	CON:
Printer	PRN:, LST:, LPT:, LPT1
Aux Port	AUX:, COM:, RDR:, PUN:
Null	NUL:, NULL:

I/O is performed to these devices, one character at a time, using the appropriate BDOS function calls. One exception occurs for the console device: if a translated mode read operation requests more than 1 byte, the BDOS buffered console input function is used to read the data (a maximum of 128 bytes per read). Any special editing features supported by the operating system (backspace processing, etc.) will therefore be enabled.

The following table lists the devices and the corresponding BDOS functions used for read and write operations in translated and untranslated modes.

Device Name	Translated Mode Read FN	Translated Mode Write FN	Untranslated Mode Read FN	Untranslated Mode Write FN
CON	1	2	7	6
AUX	3	4	3	4
COM1	3	4	3	4
PRN	-	5	-	5
LPT1	-	5	-	5
NUL	-	-	-	-

A - for the function number indicates that the corresponding operation is not supported for that device. The read function returns end of file (count = 0) if read is not supported. If writing is not supported, the write function returns a normal count indicating success, but does not actually send the data. An additional special device name, specified by a NULL string (""", which consists of just a '\0'), is recognized and processed as if CON: had been specified.

In translated mode, a newline (0x0A) is converted to a carriage return/linefeed sequence. A carriage return on input is converted to a newline, and terminates the read operation even if the byte count is not satisfied. In untranslated mode, characters are sent without modification, and read operations do not terminate until the requested number of characters has been received. Note that a read operation to the console in untranslated mode does not echo the characters received.

Programmers may also perform direct single character I/O

operations using the bdos function, and several additional functions support direct I/O to the console. See Section 5.5 for details.

If one of these devices is opened for access using fopen, input and output are performed in unbuffered mode, which means that single characters are received and sent immediately. The only exception to this rule occurs for stdin, which _main opens in buffered mode so that the buffered console input function can be used. If desired, stdin can be changed to the unbuffered mode using setnbf; see Section 3.2.2 for more information.

5.3 Memory Allocation

The full set of memory allocation functions described in Section 3.1 is provided under MS-DOS. The following cautions apply:

1. The reset functions rstmem and rbrk cannot be used if any of the level 2 I/O functions are also being used on currently open files. Note that only disk files allocate a file access block using getmem; the reset functions may be used if the only open files are actually connected to devices. A file may be closed, then re-opened after the reset function is called; however, any file pointers must be updated if this is done, because there is no guarantee that the same value will be returned when the file is opened again.

2. The dynamic memory used by the memory allocation functions is the same memory used for the run-time stack. Programmers must be careful to provide enough space for the stack to prevent its collision with the dynamic memory pool, either by getting an override value from the command line (see Section 4.1.4) or by defining an external int location called _stack and initializing it with a desired value. See Section 4.5.5 for a complete discussion of the stack size.

3. Programmers who wish to implement their own memory allocation functions can refer to the data locations in the startup module which define the total stack space available:

```
extern unsigned _base;
```

contains the offset (from DS) of the lowest portion of the stack, which is the same as the highest offset of the static data items in the program (see diagram at Section 4.5).

```
extern unsigned _top;
```

contains, in the S and P models, the offset of the top of the stack, either X'FFF0' or whatever was determined to be the highest usable offset; in the D and L models,

it contains the number of bytes allocated for the stack (same as _stack). As noted above, the external location _stack contains the default or specified stack size desired; user-written memory allocators may wish to make use of that value, as a convenience.

5.4 Program Entry/Exit

The startup module CS.OBJ (or CP.OBJ, CD.OBJ, or CL.OBJ) calls _main to begin execution of a C program, and passes to it a copy of the command line which executed the program. Actually, because MS-DOS does not save the program name portion of the command, the command line passed to _main consists of the characters "c " (lower case 'c' followed by a blank) immediately followed by all of the characters typed after the program name. The standard version of _main supplied in the libraries analyzes the command line for all of the elements described in Section 4.1.4, and then passes the command-line arguments to main. If the stack override and file specifier features are not needed, the version supplied as TINYMAIN.C can be used instead. Please note the following important cautions if this is done.

1. The library function printf sends its output to the pre-defined file pointer stdout, which is normally opened by _main. If the code that performs this function is removed, printf calls will produce no visible output (the I/O library functions ignore attempts to read or write unopened files). A similar caveat applies to the use of scanf, which reads from stdin.

2. If the goal is to avoid including the level 2 I/O functions in the linked program, the library function exit should not be called, since it closes all buffered output file before terminating execution and automatically causes level 2 functions to be included. Call _exit instead.

The program exit functions exit and _exit are described in Section 3.2.4. The error code argument is passed back to the operating system, where it can be tested in a batch file using a command such as:

```
if error level 1 goto error
```

5.5 Special Functions

The functions discussed in this section provide low-level access to various system resources. They tend to be machine-dependent and are therefore not portable.

NAME

bdos -- call BDOS function

SYNOPSIS

```
ret = bdos(fn,dx);
int  ret;       returns code
int  fn;        BDOS function number
int  dx;        value to be placed in DX
```

DESCRIPTION

Performs a BDOS call by placing fn in the AH register, dx in the DX register, and operating an INT 21H. The value returned by BDOS in the AX register is passed back as ret.

CAUTIONS

In the D and L models, it is better to use the intdosx function described next, since some BDOS function calls need a pointer defined relative to DS, which may not contain the correct segment base in these models.

NAME

intdos, intdosx -- generate DOS function call

SYNOPSIS

```
ret = intdos(inregs, outregs);
ret = intdosx(inregs, outregs, segregs);

int ret;                    operating system return code
union REGS *inregs;         input registers
union REGS *outregs;        output registers
struct SREG *segregs;       segment registers (intdosx only)
```

DESCRIPTION

Generates a DOS function request to the operating system. The operating system specifications should be checked to determine the DOS functions and calling sequences supported; the values in the registers are used as inputs. In particular, the exact function request is specified by placing a value in one of the registers (under MS-DOS, the function number is specified in AH; under CP/M-86, in CL). inregs must contain the values which will be loaded into the working registers before the function call is made; outregs will receive the values in the registers after control returns from the function request. With intdosx, the values which will be placed in the segment registers before the interrupt may be specified; although the SREGS structure defines all of the segment registers, only DS and ES will actually be loaded. The REGS and SREGS structures are defined in the DOS.H header file.

CAUTIONS

Defining the segment register values for intdosx is best accomplished by calling segread to obtain current values (see below for details on this function).

Note that inregs, outregs, and segregs are shown as pointers above; the usual technique is to declare them directly, and then use the address-of operator to pass a pointer to them.

NAME

int86, int86x -- generate 8086 software interrupt

SYNOPSIS

```
int86(intno, inregs, outregs);
int86x(intno, inregs, outregs, segregs);

int intno;                  interrupt number
union REGS *inregs;         input registers
union REGS *outregs;        output registers
struct SREGS *segregs;      segment registers (int86x only)
```

DESCRIPTION

Performs an 8086 software interrupt of the specified number. The operating system should be checked to determine the interrupts and calling sequences supported; generally, values in the registers are used as inputs. inregs must contain the register values which will be loaded into the working registers before the interrupt is performed; outregs will receive the register values after control returns from the interrupt. With int86x, the values which will be placed in the segment registers before the interrupt can be specified; although the SREGS structure defines all of the segment registers, only DSand ES will actually be loaded. The REGS and SREGS structures are defined in the DOS.H header file.

CAUTIONS

The software interrupts on the 8086 are used to implement multi-level system processing, and invalid input data can cause unpredictable (and occasionally disastrous) results. Defining the segment register values for int86x is best accomplished by calling segread to obtain current values (see below for details on this function).

Note that inregs, outregs, and segregs are shown as pointers above; the usual technique is to declare them directly, and then use the address-of operator to pass a pointer to them.

NAME

segread -- return current segment register values

SYNOPSIS

```
segread(segregs);

struct SREGS *segregs;          structure for return of values
```

DESCRIPTION

Places the current 8086 segment register values into the SREGS structure whose pointer is supplied. Its main purpose is to obtain current values in order to make a subsequent call to int86x or intdosx. The definition for the SREGS structure is found in the DOS.H header file.

NAME

movedata -- move data bytes from/to segment/offset address

SYNOPSIS

```
movedata(sseg, soff, dseg, doff, nbytes);
int sseg;           segment portion of source address
int soff;           offset portion of source address
int dseg;           segment portion of destination address
int doff;           offset portion of destination address
unsigned nbytes;    number of bytes to move
```

DESCRIPTION

Moves the specified number of data bytes from the source to the destination address. The addresses must be specified as (segment:offset) in accordance with the standard 8086 notation. This function is primarily intended for use in programs compiled using the S and P models; in the D and L models, the standard library function movmem can be used. The segread function can be used to obtain segment register values.

CAUTIONS

Memory is not protected on the 8086, so supplying invalid parameters to this function can have disastrous results.

NAME

peek, poke -- examine/modify arbitrary memory locations

SYNOPSIS

```
peek(segment, offset, buffer, nbytes);
poke(segment, offset, buffer, nbytes);

int segment;        segment portion of memory address
int offset;         offset portion of memory address
char *buffer;       local memory buffer
unsigned nbytes;    number of bytes to transfer
```

DESCRIPTION

These functions copy data values between an arbitrary memory location and a local memory buffer: **peek** moves data to the local buffer from a specified memory address, while **poke** moves data from the local buffer to the arbitrary memory address. These functions are primarily intended for use in *programs compiled* using the S and P models; in the D and L models, the standard library function **movmem** can be used.

CAUTIONS

Memory is not protected on the 8086, so supplying invalid parameters to the poke function can have disastrous results.

NAME

inp, outp -- direct port I/O functions

SYNOPSIS

```
v = inp(p);
outp(p,v);

int v;          I/O value
int p;          I/O port number
```

DESCRIPTION

The inp function reads I/O port p and returns whatever data is there. The outp function writes value v to port p.

CAUTIONS

Direct port operations can cause all sorts of problems, including physical damage to some systems. Extreme care should be exercised.

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 .OBJ 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 (or currently logged-in) drive/directory.

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 (or currently logged-in) drive/directory.

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 4.1.1 and 4.1.2 for valid command line options. The option is ignored, but the compilation is not otherwise affected. In other words, this error is not fatal.

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 .OBJ file. This error usually results from an out-of-space condition on the output disk.

A.2 Numbered Error Messages

These error messages describe syntax or specification errors in the source file; they are generated by the first phase of the compiler. A few are warning messages that simply remark on marginally acceptable constructions but do not prevent the creation of the quad file. See Section 4.3.3 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 (i.e., high bit set) characters are detected in the source file. This may also occur if a pre-processor command is 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 appears 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 An invalid arithmetic or pointer conversion was specified. This usually results when an attempt is made to convert something into an array, a structure, or a function.

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 is 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 " (quotes) are 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 2.1.

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. This may occur for constructions which are accepted by other compilers; see Section 2.1.

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 This non-fatal warning occurs when 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 is 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 is improperly specified.

29 An operation was specified which was invalid for pointer operands (such as one of the arithmetic operations other than addition).

30 This non-fatal warning occurs when 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. See Section 2.1 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, a structure, or union member declaration list, the compiler expected a closing right brace, but did not find it. This may also occur if too many elements are specified in an initializer expression list or if a structure member was improperly declared.

36 This implementation does not allow initializer expressions to be used for unions.

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 also 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 specifies 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 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 also 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 also occur if too many closing parentheses are included or if an expression is otherwise incorrectly formed.

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 also occur if the enclosed expression is 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 also occur if, due to preceding errors, the compiler begins processing portions of the body of a function as if they were external definitions.

60 A storage class other than register appeared on the declaration of a formal parameter.

61 The indicated structure or union tag was not previously defined; that is, the members of the aggregate were unknown.

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.

7Ø 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 also be generated if a preceding error caused the compiler to "get out of phase" with respect to declarations.

74 During processing of external declarations, an attempt was made to define a function, but it was not the first identifier declared on the input line.

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 previous function in a goto statement, but no definition of the label was found in that function.

79 In processing a list of declared items, the compiler expected a separator (comma or semi-colon) but did not find one. This usually results from an improperly specified list of names being declared, or from an attempt to initialize an object for which initialization is not permitted (such as an extern object).

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 2.2.1 for an example.

82 The size of an object exceeds the maximum legal size (which is the largest positive int); or, the last object declared caused the total size of declared objects for that storage class to exceed that maximum. This may also occur if the size of formal parameters exceeds 256 bytes.

83 This non-fatal warning complains of an indirect reference (usually a subscripted expression) which accesses memory beyond the size of the object used as a base for the address calculation. It generally occurs when an element beyond the end of an array is referred to.

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; who only concludes that the compiler is at fault after a long and painstaking study of his or her own code.

Lattice, Inc. is 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:

Lattice, Inc.
P.O. Box 3072
Glen Ellyn, Illinois
60138

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). Supplying the source on IBM PC-compatible disk format will save time.

2. The revision number of the compiler, when it was purchased and the serial 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."

   Our current policy in cooperation with our publisher, Lifeboat Associates in New York, calls for a free update to the first finder of a bona fide bug!

   Meanwhile, attempt to code around the problem; if that doesn't work, mutter a few curses directed at "lousy compiler writers" and work on something else. Remember, Lattice is in the business of supplying portable C compilers and uses them for its own development work; the motivation to fix the bugs immediately is definitely there.

APPENDIX C:
Conversion of CP/M-80 Programs

Because of its similarity to CP/M-80, it is reasonable to expect that C programs written for that operating system will be transported to MS-DOS without a great deal of difficulty. This appendix attempts to point out some of the pitfalls likely to be encountered when moving source from CP/M to MS-DOS or vice-versa for compilation with the Lattice C compiler.

The least amount of trouble lies in store for those who have written programs for the BDS C compiler. At the source code level, every effort has been made to be compatible. While the Lattice compiler is a little stricter in some things, generally the correction is accepted by the BDS compiler as well, which facilitates keeping one set of source code for both systems. For example, a sequence like

```
char *cp;
. . .
cp = cfunct(i);
. . .
char *cfunct(n)
int n;
{
. . .
```

will cause the Lattice compiler to complain about a mismatch of external attributes, because cfunct is used implicitly as int before it is defined as char *. Inserting

```
char *cfunct();
```

prior to the first use of cfunct eliminates the error, and is acceptable to the BDS compiler as well. As for other coding constructions, the warning generated for structures supplied as function arguments without a preceding & was included specifically for BDS C programs. The problem of external data definitions posed by the BDS implementation's lack of storage class specifiers is solved by the -x compile-time option. Here are the rules for using it on BDS C programs:

1. When compiling the main module, do not specify the -x option. The various external declarations are interpreted as definitions of the objects, and storage is actually allocated for them.

2. When compiling any of the other modules, specify the -x option on the LC1 command. The various external declarations are then interpreted as references to objects defined elsewhere (presumably in the main module).

Be careful not to compile more than one of the modules in the

program without using the -x option; otherwise, the linker will inform you that multiple definitions of the external items were encountered.

At the library level, there are other, more serious difficulties. Although the BDS library does a good job of supplying most of the standard functions described in the Kernighan and Ritchie text, the details of their operation are different from the Lattice functions in a number of small ways. In particular, putchar and getchar are direct console I/O functions under BDS C, whereas they are implemented as macros in Lattice C. This problem can be avoided by using the console I/O functions described in Section 3.2.3. In general, it is best to review all of the functions supplied in both libraries with a view toward locating potential trouble spots. Many of the more specialized CP/M functions have not yet been provided in the Lattice library, but check the latest compiler addendum; others will probably be added as newer versions of the compiler are released.

. . .

Users of the Whitesmiths C compiler are not likely to encounter any problems with source language compatibility, but the library is for the most part completely different. Hint: judicious use of #defines may eliminate some problems.

APPENDIX D:
LIST OF FILES

The following files are supplied as part of the compiler package:

Executable Files

LC1.EXE	C compiler (phase 1)
LC2.EXE	C compiler (phase 2)
FXU.EXE	Function Extract Utility
OMD.EXE	Object Module Disassembler

Run-time and Library Files

CS.OBJ	C program entry/exit module (S model)
CP.OBJ	C program entry/exit module (P model)
CD.OBJ	C program entry/exit module (D model)
CL.OBJ	C program entry/exit module (L model)
LCS.LIB	Run-time and I/O library (S model)
LCP.LIB	Run-time and I/O library (P model)
LCD.LIB	Run-time and I/O library (D model)
LCL.LIB	Run-time and I/O library (L model)

C Source Files

MAIN.C	Standard library version of _main
TINYMAIN.C	Abbreviated version of _main
FTOC.C	Fahrenheit-to-Celsius sample program
CAT.C	File concatenation sample program
FXU.C	Source for function extract utility
CONIO.C	Basic console I/O functions

C Header Files

STDIO.H	Standard I/O header file
CTYPE.H	Character type macros header file
ERROR.H	Header file defining UNIX error numbers
FCNTL.H	Header file defining level 1 I/O codes
IOS1.H	Header file defining level 1 I/O structures
DOS.H	Environment information header file
MSDOS.H	Defines MS-DOS version
SM8086.H	Memory model header file for S model
PM8086.H	Memory model header file for P model
DM8086.H	Memory model header file for D model
LM8086.H	Memory model header file for L model

(Note: in order to use the DOS.H header file, you must copy one of the last four files into M8086.H.)

Assembly Language Source Files

C.ASM	Source for C.OBJ (all versions)
IO.ASM	Sample assembler language function

Assembly Language Macro Files

SM88Ø86.MAC	Macro include file used with S model
PM88Ø86.MAC	Macro include file used with P model
DM88Ø86.MAC	Macro include file used with D model
LM88Ø86.MAC	Macro include file used with L model

(Note: in order to assemble the sample source modules, you must copy one of the last four files into DOS.MAC.)

INDEX

Lattice 8086/8088 C Compiler

MANUAL SUPPLEMENT FOR VERSION 2.10

1.0 SUMMARY OF DIFFERENCES

The following list summarizes the most important differences between Version 2.10 and Version 2.00. Please note that this document is intended as a supplement to the Version 2.00 manual. If you do not yet have that manual, you must contact the publisher from whom you purchased the compiler and make arrangements to obtain it.

1.1 Compiler Differences

The compiler has been been upgraded in a few minor ways, as the following list indicates:

— Extern/static objects as large as 64K now permitted

— New -d flag to #define symbols from command line

— Pre-defined symbols for memory model, operating system

— New -w flag forces word alignment as in Version 1.04

— Larger input lines/macro definitions supported

— Additional warnings issued for questionable constructs

1.2 Library Differences

Most of the differences between Version 2.1 and Version 2.0 are in the library, as summarized in the following list:

— Automatic sensing of MS-DOS 1 vs. MS-DOS 2

— Automatic sensing and use of 8087 math chip

— UNIX-compatible math functions

— FORK/EXEC combination functions

— ASCII/BINARY mode specifiers on FOPEN

— Access to environment strings

— Miscellaneous library additions

We have attempted to keep all of the 2.1 changes upward compatible with Version 2.0, and so you should not have to change any existing programs. However, the addition of MS-DOS version

sensing and 8087 sensing has caused the library to grow a bit. Therefore, programs that were close to a memory limit might be affected. Also, the FORK/EXEC capability forced us to change our approach to memory allocation, which may affect programs that bypass our standard memory management functions. This change in the memory layout for the S and P models forced a change in the code generated to detect stack overflow, which means that programs using these models must be entirely recompiled before they can use the new library. See section 6 for more details.

Version 2.1 also includes bug fixes in both the compiler and the library. These should all be transparent, since none of the documented interfaces were changed.

Finally, we've improved the operating procedures by adding an LC command that invokes both compiler passes and by adding several batch files that copy the release disks to our recommended directory structure on systems with hard disks.

The following sections describe these differences in detail.

TABLE OF CONTENTS

2.0 NEW COMPILER FEATURES

In addition to the usual bug fixes, this version of the compiler has been enhanced in several ways. These enhancements make it easier to perform conditional compilations, and allow a larger class of programs to be accepted by the compiler. At the same time, several helpful warnings have been added which can often point to coding errors.

2.1 New Storage Class Size Limitations

In the previous version of the compiler, no object could exceed 32767 bytes in size; moreover, the combined size of all objects declared for a particular storage class was subject to the same limitation. In Version 2.1, the limit has been changed to 65535 bytes, for static and extern objects only. Structures may be declared which are in excess of 32767 bytes, but the error message "maximum object/storage size exceeded" will be generated if an attempt is made to declare an auto structure of that size.

2.2 Command Line Definition of Pre-processor Symbols

The -d flag has been extended to allow symbols to 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. The -d flag in its simplest form retains the same meaning as in the previous version (i.e., it causes the compiler to include line-number information in the object file). The new forms of the command are

```
-dsymbol
-dsymbol=value
```

where "symbol" is a standard C identifier. 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 used in the same LC1 command; however, macros with arguments cannot be defined from the command line.

2.3 Pre-defined Symbols

As a further assistance to conditional compilation, the compiler now automatically #defines several symbols, which can be tested in conditional compilation statements to select appropriate code sequences for the operating system, memory model, and so forth.

These symbols have also simplified the new version of DOS.H and eliminated the need for the MSDOS.H, SM8086.H, PM8086.H, DM8086.H, and LM8086.H header files supplied with Version 2.00 of the compiler.

Two symbols are always defined in the compiler:

```
#define MSDOS 1
#define I8086 1
```

One of the following symbols is defined, depending on the memory model specified:

```
#define I8086S 1        defined if S model, else undefined
#define I8086P 1        defined if P model, else undefined
#define I8086D 1        defined if D model, else undefined
#define I8086L 1        defined if L model, else undefined
```

One or the other of the following is also defined, depending on the memory model:

```
#define SPTR 1          defined if S or P model
#define LPTR 1          defined if D or L model
```

If the -s option was specified on LC1, the following symbol is defined:

```
#define SFLAG 1
```

Finally, if the -d flag was specified (as "-d", not "-dsymbol"), the following symbol is defined:

```
#define DEBUG 1
```

The automatic definition of these symbols can be prevented by using a new compiler flag:

```
-u
```

Specifying this flag on LC1 cancels all of the above definitions.

2.4 Optional Word Alignment

An option to support alignment of data elements other than char to an even (word) offset has been provided:

```
-w
```

Specifying this flag on LC1 causes all data elements except char items to be assigned to even offsets. This alignment produces more efficient code on an 8086 processor, where fetching a word on an odd byte boundary requires four additional clock periods. The same alignment was used as the default in Version 1.04 of the compiler.

2.5 Expanded Line and Macro Sizes

Formerly, the substitution text for a #define macro was limited to a maximum of 80 bytes; in Version 2.10, the new maximum is 256 bytes. Similary, the previous maximum size of an input source line was 132 bytes; the new maximum is 256 bytes.

2.6 New Warning Messages

Two new warning messages have been added to Version 2.10 of the compiler. They are:

```
Warning 84: redefinition of pre-processor symbol "xxxx"
Warning 85: function return value mismatch
```

The first warning is issued whenever a #define statement is encountered for an already #defined symbol. As noted in the manual, the second definition takes precedence, but requires an additional #undef statement before the symbol is truly undefined.

The second warning is issued whenever the value returned by a function is not of the same type as the function itself. The value specified is automatically converted to the appropriate type; the warning merely serves to notify you of the conversion. The warning can be eliminated by using a cast operator to force the return value to the function type.

Version 2.00 of the compiler generated warning 30 ("pointers do not point to same object") only when the result of an assignment statement was a pointer. In Version 2.10, the same warning is generated when a pointer of any type is assigned to an arithmetic object. A new warning (duplicate declaration of item."xxxx") is now generated when a formal parameter for a function is redeclared at the lowest level inside the function, as in

```
f(x)
char x;
{
int x;
```

Finally, the use of an undefined structure tag in a pointer declaration now causes only a warning, not an error, if the structure is never defined -- as long as no attempt is made to refer to the structure's members, or to perform arithmetic with the pointer.

2.7 New Error Messages

If either operand in a logical OR (||) or logical AND (&&) expression is constant, the compiler now will generate the error message "invalid constant expression". If the end of source file input is detected inside a constant, the error message "unexpected end of file" will be generated.

3.0 MS-DOS VERSION SENSING

The start-up program (see C.ASM) now sets up a global variable named _dos that indicates which version of MS-DOS is active. The library functions then test this variable at appropriate points in order to call the proper low-level operating system service functions.

You can refer to _dos in two ways, as follows:

```
extern char _dos;
extern char _dos[2];
```

The MS-DOS major version number (1 or 2) is then found at _dos (first method) or _dos[0] (second method). The minor version number is found at _dos[1] for the second method only.

As you can see by examining C.ASM, the MS-DOS version information is obtained via operating system call 30, which is fully described in the MS-DOS or PC-DOS Technical Reference. We ensure that _dos will never contain a value of 0 when operating under MS-DOS, and we may use the 0 value to indicate CP/M-86 in a future release.

4.0 AUTOMATIC SENSING AND USE OF 8087 MATH CHIP

The floating point simulation functions in the library have been changed to detect the presence of the 8087 math chip. If the chip is installed, you should notice a large performance improvement in programs that do many floating point operations.

Notice that you do not need to use the -f flag in order to obtain the benefits of the 8087 under Version 2.1. Furthermore, the programs that you generate will run correctly on a system without the math chip. We are still considering the use of the -f flag to stimulate in-line 8087 code generation, but this may be dropped if the bi-modal library approach proves adequate.

5.0 UNIX-COMPATIBLE MATH FUNCTIONS

Version 2.1 includes a large portion 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 usually be included when you are using these functions.

NAME

exp,log,log10,pow,sqrt -- exponential functions

SYNOPSIS

```
r = exp(x);          compute E**x
r = log(x);          compute natural log of x
r = log10(x);        compute base 10 log of x
r = pow(x,y);        compute x**y
r = sqrt(x);         compute square root of x

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

DESCRIPTION

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.

NAME

sin,cos,tan,asin,acos,atan,atan2 -- transcendental 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 arcsin 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,x,y;
```

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.

NAME

sinh,cosh,tanh -- hyperbolic functions

SYNOPSIS

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

double x,y;
```

DESCRIPTION

These functions simply compute the normal hyperbolics.

NAME

rand,srand — 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, you can call srand 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.

NAME

drand -- 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;
unsigned short y[3];
long z;
unsigned short *p;
unsigned short k[7];
```

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 you to initialize 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 you to do 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 you call seed48, a and c are reset to their original default values.

NAME

ceil,fabs,floor,fmod,frexp,ldexp,modf -- 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,y,z;
int i;
double *p;
```

DESCRIPTION

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.

NAME

atof,atoi,atol — 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;
int i;
long l;
char *p;
```

DESCRIPTION

These functions skip over any leading white space (i.e. blanks and tabs) 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.

NAME

strtol -- 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

This function converts an ASCII string into a long integer, using the specified number base for the conversion. Leading white space (i.e. blanks and tabs) 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).

NAME

ecvt -- 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

This function 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.

NAME

matherr -- 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
```

When matherr is called, the function that detected the error will have placed its proposed return value into the exception structure. If you want to substitute a different value, then matherr must return a non-zero code.

If you do not supply a version of matherr, the standard version will put the appropriate error number into errno and return a code of 0.

6.0 FORK/EXEC COMBINATION FUNCTIONS

MS-DOS Version 2 contains a system call that behaves like a combination of the UNIX fork and exec functions. That is, it creates a "child" process which executes a specified load module. The "parent" process can then retrieve the child's completion code via another new system call that is similar to the UNIX wait function.

Of course, MS-DOS Version 2 does not really support multi-programming, and so the parent and child processes do not actually timeshare the computer. The parent remains suspended until the child terminates. However, if multi-programming is added to MS-DOS in the future, we expect that the interface provided in our library will be unaffected except that the parent will no longer be totally out of business while the child executes.

After reviewing the new functions described in the following manual page, you might wonder why we did not provide an exact equivalent of UNIX's fork function. The answer is simply that we could not figure out a general way to replicate the parent process's address space on the 8086. That is, the typical program operating under MS-DOS contains absolute segment numbers, which makes it impossible to move the program for execution in another area of memory.

Memory Management Change

In order to implement the FORK/EXEC capability, we had to change our approach to memory allocation for the S and P memory models. Version 2.0 of Lattice C remained fully compatible with the earlier small-model versions by placing the stack as high as possible in the data segment and by placing the memory pool (i.e. the "heap") between the static data area and the stack. However, for the D and L models introduced with Version 2.0, we placed the stack immediately above the static data and made all remaining memory above the stack available for the heap.

In Version 2.1, the stack is located just above the static data for all memory models. Under MS-DOS 2, then, the start-up program (see C.ASM) returns all remaining memory space to the operating system so that there is room to create a child process. When you call sbrk, it will attempt to obtain the required amount of space back from MS-DOS, and when you call rbrk, the space thus obtained will once again be given back to MS-DOS. In other words, the memory allocation functions will appear to operate as they did under Version 2.0 of the compiler, but in fact they will be much more closely coupled to the operating system.

One fly in the ointment is that a child process can choose to terminate but remain resident in memory. If that happens, the parent may find that its heap cannot grow because the child

process is in the way. To help avoid this problem, we've added a global variable called _mneed. This is simply a long integer that specifies the minimum number of bytes needed in the heap. At start-up time and each time you call rbrk, _mneed is consulted. If sufficient space cannot be allocated, the start-up program aborts or rbrk returns a -1 failure code. Note that you can change _mneed each time you call rbrk.

Finally, if you referenced the pointers _mbase and _mnext directly without going through the memory allocation functions, be aware that these are both 4-byte pointers under all memory models. This implies that under the S and P models these pointers should not be treated as DS-relative offsets. The sbrk function adds _top to the first word of _mnext in order to return a DS-relative offset to you. Also, sbrk ensures that the heap remains within the addressing range of DS.

NAME

fork/wait -- create child process and wait for it

SYNOPSIS

```
error = forkv(name,argv);
error = forkl(name,arg0,arg1,...,argn,NULL);
error = forkvp(name,argv);
error = forklp(name,arg0,arg1,...,argn,NULL);
code = wait();

int error;          0 for success, non-zero for error
int code;           child process return code
char *name;         file name of load module
char *argv[];       argument pointer array
char *arg0,*arg1,.. argument pointers
```

DESCRIPTION

These functions create a child process that executes the specified load module and then passes a return code back to the parent process. The specified arguments are passed to the child's main entry point via the normal argc/argv mechanism. By convention, the first argument (i.e. **arg0** or **argv[0]**) is the name of the child process load module, which is usually the same as name. Note, however, that this first argument is not actually passed to the child process because of limitations in the MS-DOS process creation primitives. Also note that these same limitations restrict the total length of all argument strings to be no more than 127 characters.

The function names have been chosen to match the various forms of the UNIX exec function. The "v" suffix on forkv and forkvp indicates that the arguments are supplied as a vector in the argv form. The last pointer in the vector must be null. The "l" suffix on forkl and forklp indicates that the arguments are supplied as a list of pointers, with the last pointer being null. The "p" suffix on forkvp and forklp indicates that the PATH environment variable should be used if the load module is not found in the current directory. When stepping through the directories, the functions look for "name.COM" and then "name.EXE" in each directory.

RETURNS

If the child process cannot be created, the fork function returns a non-zero result. Under MS-DOS, the global integer _oserr will contain the operating system error code. If it is 0, the error occurred while processing the fork call parameters.

CAUTIONS

Under MS-DOS the arguments are converted into a text string no longer than 127 bytes. The first argument (i.e. **arg0** or **argv[0]**) is dropped, and a blank is placed between succeeding arguments. The resulting string is passed to the child in its command line buffer. If the child is a C program, its startup phase will convert the command line back into an arg list.

7.0 ASCII/BINARY MODE SPECIFIERS ON FOPEN

Since the use of the _fmode global flag has confused some users, we've added another way to specify translated or untranslated mode when opening a level 2 file via fopen or freopen. You can now specify translated mode by placing the letter 'a' in the second position of the mode string. Similarly, the letter 'b' specifies untranslated mode. If the second letter is neither of these, _fmode is used as before.

Note that this new approach is not currently UNIX-compatible. However, several other C compiler packages work this way, and proposals for this approach are floating around some of the standards committees.

In summary, fopen and freopen now recognize the following mode strings:

```
r   -- open for reading (translation according to _fmode)
ra  -- open for reading (translated)
rb  -- open for reading (untranslated)

w   -- open for writing (translation according to _fmode)
wa  -- open for writing (translated)
wb  -- open for writing (untranslated)

a   -- open for appending (translation according to _fmode)
aa  -- open for appending (translated)
ab  -- open for appending (untranslated)
```

You can also place a plus sign after any of these codes to indicate opening for both reading and writing. If you open for reading with a plus, then the file must already exist; but if you open for writing with a plus, the file will be created anew. Opening for appending with a plus will allow you to read from anywhere in the file, but all write operations will occur at the end of the file.

I/O ERROR CODES

When we wrote the Version 2 manual, we forgot to mention the method by which you can determine what went wrong when one of your I/O calls fails. In general, we've tried to adhere to UNIX's technique for reporting errors. When you get a failure indication from an I/O function, consult the global integer errno, which will contain one of the error codes defined in the header file error.h. As a further refinement, you can look at the global integer **_oserr** to see the MS-DOS error code, if any. These codes are described in the MS-DOS and PC-DOS Reference Manuals.

8.0 ACCESS TO ENVIRONMENT STRINGS

MS-DOS Version 2 supports the UNIX notion of "environment strings", which are of the form "name=value" and are usually defined by the **SET** command. The strings are stored one after another in the environment array, and the last one is followed by a null string. Upon entry, the public pointer **_env** points to the current environment array. For the S and P models, the startup program copies the environment into the stack so that you can address it relative to DS. Note that the stack space required for this is added to the value you specify in **_stack** or on the command line.

The UNIX-compatible **getenv** function has been added to the library to enable you to easily find a particular name in the environment.

NAME

getenv -- get environment string by name

SYNOPSIS

```
p = getenv(name);
char *p;        points to value part of matching env string
char *name;     env name
```

DESCRIPTION

This function searches the environment array pointed to by _env and returns a pointer to the value portion of the first string whose name portion matches name. If no match occurs, a NULL pointer is returned.

9.0 MISCELLANEOUS LIBRARY ADDITIONS AND CORRECTIONS

By popular request, we've added more of the string manipulation functions from the proposed UNIX standard. Also, we've added the following functions:

```
remove    -- same as unlink
clearerr  -- same as clrerr
rename    -- rename a file
bdosx     -- bdos function with pointer
getche    -- getch with echo
setjmp    -- save current stack for long return
longjmp   -- make long return
```

The library now contains callable versions of some of the character type macros described in section 3 of the Lattice C Manual. Specifically, the following macros are also available in function form:

```
isalpha(c)     non-zero if c is alphabetic
isupper(c)     non-zero if c is upper case
islower(c)     non-zero if c is lower case
isdigit(c)     non-zero if c is a decimal digit
isspace(c)     non-zero if c is white space
isalnum(c)     non-zero if c is alphanumeric
iscntrl(c)     non-zero if c is a control character
toupper(c)     converts c to upper case if it is lower
tolower(c)     converts c to lower case if it is upper
```

In order to use the function forms, do not #include the **ctype.h** header file in your compilation. If you need **ctype.h** for some reason, you can #undef the specific macros that should be treated as functions.

The following manual pages describe the functions that have been added.

NAME

strcat/strncat -- string concatenation
strcmp/strncmp -- string comparison
strcpy/strncpy -- string copy
strlen -- measure string length
strchr/strrchr -- find first or last occurrence of character
strpbrk -- find break character
strspn/strcspn -- find longest initial span

SYNOPSIS

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

order = strcmp(a,b);
order = strncmp(a,b,max);

to = strcpy(to,from);
to = strncpy(to,from,max);

length = strlen(s);

p = strchr(s,c);
p = strrchr(s,c);

p = strpbrk(s,t);

length = strspn(s,t);
length = strcspn(s,t);

char *to,*from;          destination and source strings
int max;                 maximum number of characters
char *a,*b;              strings to compare
int order;               - if a < b
                         0 if a == b
                         + if a > b
char *s;                 string to test
char *t;                 test string
int length;              result length
char *p;                 result pointer
```

DESCRIPTION

The strcat and strncat functions append the "from" string to the "to" string. For strncat, no more than the specified maximum number of characters will be appended.

The strcmp and strncmp functions perform an unsigned character comparison of the specified strings. For strncmp, no more than the specified maximum number of characters will be compared.

The strcpy and strncpy functions copy the from string to the to string. For strncpy, no more than the specified maximum

number of characters will be copied.

The **strlen** function returns a count of the number of characters in the specified string, not including the terminating null.

The **strchr** function returns a pointer to the first occurrence of the specified character in the specified string. Similarly, **strrchr** returns a pointer to the last occurrence of the character. Both functions return a null pointer if the character is not found in the string.

The **strpbrk** function returns a pointer to the first occurrence in string **s** of any character from string **t**. A null pointer is returned if no character from the test string is found.

The **strspn** function returns the length of the initial segment of string **s** that consists entirely of characters from string **t**. Similarly, **strcspn** returns the length of the initial string of characters not from string **t**.

NAME

rename -- rename a file

SYNOPSIS

```
error = rename(old,new);
int error;              0 for success
char *old;              old file name
char *new;              new file name
```

DESCRIPTION

This function renames a file, if possible. A failure will occur if the new file name already exists or if the old file name does not.

NAME

bdos/bdosx -- call BDOS function

SYNOPSIS

```
ret = bdos(fn,dx,al);
ret = bdosx(fn,dp,al);

int ret;          return code
int fn;           BDOS function number placed in AH
int dx;           value to be placed in DX
char *dp;         pointer to be placed in DS:DX
int al;           value to be placed in AL
```

DESCRIPTION

Performs a BDOS call via interrupt number 0x21. For the S and P memory models, bdos and bdosx behave identically. For the D and L models, you must use bdosx if the BDOS function requires a pointer in DS:DX or bdos if BDOS only wants an integer in DX.

NAME

getch/putch -- get/put character directly to/from console

SYNOPSIS

```
c = getch();        get character with no echo
c = getche();       get character with echo
putch(c);           put character

int c;
```

DESCRIPTION

These functions get and put characters directly to and from the console using the lower-numbered BDOS functions. No special processing is done except that putch puts a carriage return character in front of each newline.

Previous versions of **putch** masked off the high order bit of the character and did not automatically emit carriage returns. Existing programs that generate "\r\n" sequences should still behave the same because the second '\r' emitted by **putch** is merely redundant.

NAME

setjmp/longjmp -- perform non-local goto

SYNOPSIS

```
ret = setjmp(save);
longjmp(save,value);

int ret;            return code
int value;          return value
jmp_buf save;
```

DESCRIPTION

The **setjmp** function saves the current stack mark in the buffer area specified by **save** and returns a value of 0. Then a later call to **longjmp** will 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.

CAUTIONS

Calling **longjmp** with an invalid save area is an effective way to disrupt your system. One common error is to use **longjmp** after the function calling **setjmp** has returned to its caller. If you think about how the stack works, you'll see why this doesn't.

10.0 CONVENIENCE FEATURES

This version contains several things that should make it easier to use the Lattice C Compiler, including:

-- Batch files for loading the compiler onto an IBM-XT

-- A single LC command to invoke both compiler passes

-- Batch files for compiling and linking in standard ways

These are described in the following sections.

10.1 Batch Files for Loading Compiler onto IBM-XT

Since the IBM-XT and equivalent MS-DOS hard-disk machines seem to be very popular with Lattice C users, we've included a batch file that constructs our recommended directory structure and a second batch file that loads the compiler onto the hard disk.

MAKELC.BAT creates a directory structure that will contain the various modules that make up the compiler package. To execute it, place the first release disk into drive A and type A:MAKELC. When the procedure completes, the hard disk will contain the following directory structure:

\lc	Contains compiler, header files, and utilities.
\lc\s	Contains headers, objects, and libraries for S memory model.
\lc\p	Contains headers, objects, and libraries for P memory model.
\lc\d	Contains headers, objects, and libraries for D memory model.
\lc\l	Contains headers, objects, and libraries for L memory model.
\lc\c	Contains headers, objects, and libraries for building .COM files.
\lc\src	Contains source files for utility and demonstration programs.

The LOADLC.BAT procedure copies the information from the release disks to the hard disk. If you don't want to keep a particular memory model online, simply remove its subdirectory before executing the load procedure. To execute LOADLC, place the first release disk into drive A and type:

```
c:
cd \lc
copy a:loadlc.bat
loadlc
```

At the appropriate times you will be prompted to change the disk in drive A.

After loading the compiler modules, you should set the default search path for the command interpreter to include \lc. This can either be done via the PATH command or via the AUTOEXEC.BAT file as described in the MS-DOS reference manual.

10.2 LC Command

The release disk contains a program called LC.COM that uses the new FORK/EXEC functions to call the two compiler passes repeatedly for multiple compilations. The command has the format:

```
LC options files
```

where **options** is a list of compiler options and **files** is a list of files, which can include "wild cards".

In general, the options are the same as for the LC1 and LC2 commands, except where LC1 and LC2 used the same option letter to mean different things. These cases were resolved as follows:

-mds This option specifies the D model with the -s option on LC2.

-m2s Same as -mds.

-mls L model with -s option on LC2.

-m3s Same as -mls.

-qx Specifies prefix for quad files, same as LC1 -o.

In other words, the -s flag for pass 2 appears as a suffix on the **-md** and **-ml** memory model specifiers, and the quad file drive is indicated via -q instead of -o. Note that these changes apply only to the new LC command, not to LC1 and LC2.

LC allows you to put a blank between an option letter and the string that follows it, as in

```
LC -d xyz
```

This is compatible with UNIX, but causes a problem if the -d item is just before the file name part of the command and was intended to indicate debugging mode instead of defining a symbol, as in

```
LC -d program
```

The symbol "program" will be #defined instead of being treated as a file name to be compiled. To get around this problem, use the UNIX convention of ending the options with a single dash:

```
LC -d - program
```

10.3 Batch Files for Compiling and Linking

The release disks contain several batch files that should simplify the most common compiling and linking scenarios.

LCS	Compile for S model
LCP	Compile for P model
LCD	Compile for D model
LCL	Compile for L model
LINKS	Link for S model
LINKP	Link for P model
LINKD	Link for D model
LINKL	Link for L model
LINKC	Link for .COM file

The LCx procedures accept up to 9 arguments consisting of options (as defined for LC.COM) and file names or file name patterns. The options must appear first. The LINKx procedures accept a single argument that is the name of the .OBJ file containing your main program.

11.0 LIST OF FILES

Version 2.10 is normally shipped on disks in the IBM 320K format. As discussed in section 10, the first disk contains batch files that facilitate copying the release disks onto your hard disk if you use an IBM-XT or equivalent. The actual release files are:

Batch Files

MAKELC.BAT	Make hard disk directory structure
LOADLC.BAT	Load Lattice C onto hard disk
LCx.BAT	Compile under memory model x
LINKx.BAT	Link under memory model x

Executable Files

LC.COM	Compiler command line handler
LC1.EXE	C compiler (phase 1)
LC2.EXE	C compiler (phase 2)
FXU.EXE	Function extract utility
OMD.EXE	Object module disassembler
PLIB86.EXE	Object module librarian

Run-time and Library Files

CS.OBJ	C program entry/exit module (for S model)
CP.OBJ	C program extry/exit module (for P model)
CD.OBJ	C program extry/exit module (for D model)
CL.OBJ	C program extry/exit module (for L model)
CC.OBJ	C program entry/exit module (for .COM files)
LCS.LIB	Run-time and I/O library (for S model)
LCP.LIB	Run-time and I/O library (for P model)
LCD.LIB	Run-time and I/O library (for D model)
LCL.LIB	Run-time and I/O library (for L model)

C Source Files

MAIN.C	Standard library version of _main
TINYMAIN.C	Abbreviated version of _main
CONIO.C	Basic console I/O functions
FTOC.C	Fahrenheit-to-Celsius sample program
CAT.C	File concatenate sample program
FXU.C	Function extract utility

C Header Files

STDIO.H	Standard I/O header file
CTYPE.H	Character type macros header file
DOS.H	Environment information header file
ERROR.H	Header file defining UNIX error numbers
FCNTL.H	Header file defining level 1 I/O codes
IOS1.H	Header file defining level 1 I/O structures
MATH.H	Mathematical functions header file
LIMITS.H	Defines limiting values for math functions

Assembly Language Source Files

C.ASM	C program entry/exit module (all versions)
IO.ASM	Sample assembler language function

Assembly Language Macro Files

CM8086.MAC	Macro include file used for .COM files
SM8086.MAC	Macro include file used with S model
PM8086.MAC	Macro include file used with P model
DM8086.MAC	Macro include file used with D model
LM8086.MAC	Macro include file used with L model

(Note: in order to assemble the source modules, one of the above files must be copied into DOS.MAC; use the version appropriate for the memory model desired.)

TECHNICAL BULLETIN
TB840523.001

DATE: May 23, 1984
PRODUCT: 8086/8088 C Compiler, Version 2.10
SUBJECT: Support for .COM files

The Version 2.10 disks contain several files that support the construction of .COM files. However, we neglected to include information in the Version 2.10 addendum about this feature.

If you use our standard installation procedure as described in the addendum, your hard disk will contain a directory "\lc\c" and a batch file "linkc.bat". In the former you will find versions fo "c.obj" and "dos.mac" that must be used when constructing .COM files. The general procedure is:

1. Compile your C modules under the small model (e.g. via the LCS batch procedure);

2. Assemble your assembly-language programs using "\lc\c\dos.mac".

3. Link everything using the LINKC batch procedure. This should cause a "NO STACK SEGMENT" warning message, which can be ignored. LINKC also calls the MS-DOS utility EXE2BIN to convert the linker .EXE output into the desired .COM format.

Note that you cannot produce a .COM file if any of the included modules defines a stack segment or contains segment fixups. Compiling under the small model and linking with the special version of "c.obj" guarantees that these two criteria are met. Assembling with the special version of "dos.mac" does not guarantee that your assembly-language can be used to construct a .COM, so you might want to take a look at "c.asm" to see how we set it up to avoid segment fixups.

END

TECHNICAL BULLETIN
TB840523.002

DATE: May 23, 1984
PRODUCT: 8086/8088 C Compiler, Version 2.10
SUBJECT: Return values for INTDOS and INT86

The Version 2.10 addendum did not mention a change that we made in INTDOS, INTDOSX, INT86, and INT86X as a result of requests from many users. These functions now return the processor status flags instead of the AX register value. Many MS-DOS interrupt functions use the flags (especially the carry flag) to convey information back to the caller, and there was previously no way for the C program to obtain this information after calling one of the above functions. The flags are defined in any 8086/8088 instruction manual.

Note that if your program is assuming that these functions return AX, it must be changed to obtain AX from the "outregs" structure as described in the Lattice C Manual.

We have also had several queries from people who are trying to use INT86 or INT86X to perform absolute disk reads and writes via interrupts 0x25 and 0x26. This will not work because those two interrupts return with the status flags still pushed on the stack, as is discussed in the MS-DOS and PC-DOS Programmer's Reference.

END

TECHNICAL BULLETIN
TB840615.001

DATE: June 15, 1984
PRODUCT: 8086/8088 C Compiler, Version 2.11
SUBJECT: Version 2.12 Update

Version 2.12 of the 8086/8088 C compiler has been released to correct the following problems:

1. Null #defines were not handled correctly. That is, a statement such as

```
#define XYZ
```

caused a spurious error message.

2. The STRNCMP function did not always return the correct value.

3. The character count returned by the STCCPY function did not include the null terminator under some circumstances.

4. Bit fields were not compiled correctly as a result of changes introduced in Version 2.10.

5. The INT86 and INT86X functions did not work correctly in all cases because some DOS interrupts destroyed the BP register.

6. When you defined _fmode to 0x8000, the standard files (stdin, stdout, and stderr) were not switched into raw mode by _main.

7. Because of a packaging error, the Version 2.00 copy of MAIN.C was included on the 2.1 release disks. The correct file is called _MAIN.C and is now included. Do not use the old MAIN.C with Version 2.1.

8. STDIO.H has been changed to define NULL as 0 for the S and P memory models and as 0L for the D and L models.

9. Several people complained about the load module size increase caused by the DOS compatibility feature described in Section 3 of the 2.1 addendum. Therefore, we have changed the standard libraries so that the I/O functions work only with DOS 2. If you still want to be compatible with DOS 1, the release disks contain files named IOS1x.OBJ, where x is the memory model (S,D,P, or L). Include the appropriate copy of IOS1 when you link, and your program will work with both DOS 1 and DOS 2. If you want to save a little more memory, examine _MAIN.C and remove the code that is specific to DOS 1.

10. If you declared a function to return a char or float value, the function would actually return an int or a double, respectively. This has been corrected. Note that this bug is suspected to exist in several other compilers, particularly on UNIX systems, and some people have fallen into sloppy coding practices because of it. The most common pitfall is illustrated below:

In module #1:

```
char func()
{
char c;
return(c);
}
```

In module #2:

```
int x;
x = func();
```

The contents of x's high order byte will be garbage, because the module #2 implicitly declares func to return an int even though it is actually returning a char. On the 8086, what this means is that func places c in the AL register and does nothing with AH, which is a code improvement.

11. Section 2.7 of the 2.1 addendum indicates that constants are not allowed as operands in logical expressions. This restriction has been removed because it broke several existing programs.

12. There is a typo on page 5 of the 2.1 addendum. In the second last line, the word "constant" should be "comment".

13. There is a typo on page 35 of the 2.1 addendum. In the description of the -mds option, the -s flag applies to LC1, not LC2.

14. The -x and -n flags were not recognized by the LC.COM command.

15. The header file SETJMP.H was omitted from the 2.10 and 2.11 release disks.

16. The 2.10 and 2.11 releases contained libraries that were not compiled with the -s option, which resulted in a performance degradation.

17. The 2.1 addendum did not make it clear that LC.COM only recognizes C source files that are in the current directory. If you type

```
LC \stuff\abc
```

the command will not find the source file.

If you have already purchased 2.10 or 2.11, you can receive a free update to 2.12 by simply sending the original release disks to us with a return mailer.

END

Phoenix

Phoenix Software Associates©

PLIB86

Object Library Manager for Intel 8086/8088

by Dave Hirschman

Table of Contents

Plib86 (tm) is a Phoenix Software Associates Ltd. software system that can manipulate libraries of object files. It supplements the PSA linkage editor Plink86 (tm), and is intended for use on the Intel Corporation (1) 8086 (or 8088) processor (tm) under the MS-DOS (2) or CP/M-86 (3) operating systems.

Plib86 handles object files and libraries conforming to the INTEL relocatable file format described in their document "8086 Relocatable Object Module Formats" #121748-001. This format is used in compilers written by Microsoft Corporation, creator of the MSDOS operating system, by most other companies writing compilers for MSDOS, and by a few compilers written for Digital Research's CP/M-86 operating system. However, a different library index is used by Microsoft to achieve faster library searches. Plib86 can read and generate both the Intel and Microsoft library index formats.

The first section of this manual provides an explanation of the "object library" concept and the capabilities of Plib86. User's unfamiliar with library managers would do well to start here. Also, the Plink86 user's guide contains a chapter discussing object files and linkage editors that may be helpful.

The next section of this manual describes how to use Plib86 to handle several common object library situations. At the same time it provides an informal explanation of what the commands do. Those readers experienced with linkage

editors and library managers may wish to skip directly to this portion of the manual: it provides enough information to handle most jobs.

The final portion of the manual is an exhaustive list of the commands and features offered by Plib86. This should be examined when it becomes necessary to go beyond the examples given in the previous section. Side issues such as error codes are generally referred to appendices.

Typically it is convenient (if not essential) to divide a large programming job into smaller pieces called "modules" that can be edited and compiled separately. Actually, compilers available on micro-computers tend to have severe limitations on how many lines of code can be compiled at one time, forcing the programmer to use modularization anyway. On the positive side, modular programming offers a method of organizing a program into manageable pieces that are easier to understand and work with.

After the program modules are created and compiled the programmer must "link" them together with a "linkage editor" to produce the executable program (see Plink86 user's manual).

Once one has created a modular program one may find that some of the modules are useful in a different program. With a little effort these modules can be made more general in function and can be used in many programs. The programmer can gradually build up a "library" of useful routines that can be hooked in by the linkage editor whenever needed.

In fact, virtually all compilers are sold with a "library", since functions like arithmetic on real numbers are often not supported by the hardware and have to be implemented as procedure calls. The compiler library also contains modules that support the high level features of the language such as formatted output in FORTRAN. This library is often called the "run time support" since its modules are required while the program executes.

Other software products in addition to compiler runtime support routines are sold in the form of libraries. An example might be a set of data base management routines that is combined with the application program by the linkage editor to produce a complete system.

Because of the importance of libraries, linkage editors typically have special facilities for handling them. To save memory space, only those modules in the library that are actually required by the program are linked in. Sometimes a library is simply a concatenation of object modules, requiring the linkage editor to search sequentially for the required modules. More sophisticated systems provide a "library index". It contains a list of the public symbols offered by each library module, and the location of the module that defines each symbol. Therefore the linkage editor can rapidly locate the modules that are required. The Microsoft and Intel library formats are indexed structures.

The purpose of the library manager is to create and manipulate object module libraries. It is therefore a useful assistant to the linkage editor.

Plib86 provides commands to create libraries from individual object modules, and to extract a selected module from a library. It can also merge libraries, and can replicate the library search process undertaken by the linkage editor while creating a program. In other words, one can create a library consisting of only those modules that the linkage editor would include in a particular program.

Plib86 also provides a powerful cross-reference function. It optionally generates a report listing each public symbol, the module which defines it, and a list of other modules that refer to it. This may be used to cross-reference a single library or several libraries together, or, in combination with the library search feature described above, to generate a cross-reference of a program that will be created by the linkage editor.

Creating/Merging Libraries

To create a new library use the BUILD command and the FILE command. For example, executing Plib86 and entering

```
BUILD  DB.LIB
FILE   BTREE, SORT, REPGEN,
       FIRSTLIB.LIB;
```

in response to the prompt would create a library named DB.LIB containing the files listed after the FILE command. These files could be single object modules or complete libraries. Everything is merged into a single library. The default file type for the files appearing in the FILE statement is "OBJ".

Normally you can just execute Plib86 and type in commands on as many lines as desired. Then end the last line with a semi-colon to begin processing. Each statement begins with a key word like BUILD or FILE and is followed by arguments, possibly separated by commas. Input is free format, and blank lines are ignored. Also, key words may be abbreviated by leaving off characters at the end. For example, you can use BU and FI instead of BUILD and FILE. An error message will be given if the abbreviation could be confused with another command.

Another way to use Plib86 is to give the commands as it is executed. For example, the above library could have been created by entering (on one line):

```
PLIB86 BU DB FI BTREE, SORT,
       REPGEN, FIRSTLIB.LIB
```

Note that the output file type defaults to "LIB" automatically.

Library Search

Suppose you want to create a library consisting of several modules plus those portions of another library that are referenced by the modules. Use the LIBRARY command:

```
BU DB FI BTREE, SORT, REPGEN
LIB FIRSTLIB.LIB
```

The portions of FIRSTLIB not referenced by the three.other files are not put into the DB library.

Updating a library

To update a library it is necessary to copy the old library to the output file while omitting the module to be updated, and to include the new module. For example, to replace module COSINE in library MATHLIB, rename the current MATHLIB.LIB to MATHLIB.OLD and enter

```
BU MATHLIB FI COSINE,
   MATHLIB.OLD EXC COSINE
```

The EXCLUDE statement applies to the previous file name given and causes the COSINE module in the MATHLIB to be omitted.

Module Extraction

The EXTRACT statement causes a single object module file to be created. It may not be used at the same time as BUILD. The first object module found in the input files is extracted, so the particular module to be selected from a library must be specified. The object file extracted may be given any file name. The module name remains the same. For example, typing

```
EXT OLDCOS FI MATHLIB.LIB
      INCLUDE COSINE
```

creates file OLDCOS.OBJ containing object module COSINE. The INCLUDE statement is the counterpart of EXCLUDE: it applies to the previous input file and causes only those modules named to be considered for processing. There wouldn't be any point to INCLUDing more than one module in this case since only the first one found is extracted.

Cross reference listing

To create a cross-reference listing use the LIST command with input file statements similar to those given in previous examples. For example,

```
LIST = DB S
FI BTREE, SORT, REPGEN, FIRSTLIB.LIB
```

creates a cross reference report named DB.LST describing the modules in all of the files listed. The "S" selects the cross-reference report. For a description of other reports available see the LIST command description. The "=" specifies that the report is to be put into a disk file. If omitted the report appears on the console.

Input Format

This portion of the manual describes some basic input elements. Later sections show how these are combined to create full statements.

Identifiers

An identifier is the name of some object, such as a module or symbol. An identifier is a sequence of no more than 64 characters containing no spaces, and containing none of the following:

^=;<>/,\!'#&*+-:@ DEL

Lower case letters, when used, are automatically translated into upper case. The first character of an identifier may not be a digit 0 - 9.

The above restrictions on valid identifier characters may be avoided by using the escape character `^`. The character immediately following the escape character is treated as a normal identifier character.

The following are examples of valid identifiers:

```
ProgramI
SORT3
ABC^@       (the `@` is escaped)
```

The following are not valid identifiers:

```
34ABC   - begins with a number
NIM A   - contains a space
```

PROG%1 - starts a comment with `%`

The above identifiers could all be made valid with the escape character:

```
^34ABC
NIM^ A
PROG^%1
```

To include the escape character in an identifier enter two escape characters

Identifiers appearing in object files are truncated to 50 characters for purposes of comparison with other identifiers in the program. Identifiers may be truncated again for inclusion in reports (see the LIST command).

Disk File Names

Plib86 adapts itself to the file name format used by the operating system it is executing under. The first character not allowed to be in a file name terminates the name. The escape character may be used to put any character into a file name.

In this manual, MS-DOS format file names are used for purposes of discussion. These file names are of the form [device:]name[.type], with optional portions in brackets. Here are some examples:

```
MATHLIB.LIB
B:CHESS.OBJ
SCANNER
```

When the "device:" is not given, Plib86 assumes that the currently logged-in disk is to be used.

Initiating Plib86

Plib86 may be used interactively, or input may be given as it is executed:

```
Plib86 statements <cr>
```

where <cr> means to press the RETURN key. This means that Plib86 may used in .BAT files.

To use Plib86 in the interactive mode, enter

```
Plib86<cr>
```

on the console. Plib86 will read lines from the console, prompting with "=>" The standard line editing features supplied by the operating system are available. Plib86 checks input lines for syntax and stores them until a semi-colon ';' is entered at the end of a line. Then processing of the input files begins.

A disk file containing all or only part of a command may be inserted into the input at any point by preceding the disk file name with an "@". The default file type is ".LNK". These disk files can contain further "@" specifications, up to three levels deep. The most common use of this feature is to prepare a file containing a complete command; then, entering

```
Plib86 @file name <cr>
```

creates the library. Sometimes these ".LNK" files may be prepared once for a given library and used over and over again, greatly simplifying the whole process.

Plib86 reads an entire command, checking for syntax only, before any file processing is done.

Command Format

All Plib86 input is free format. Blank lines are ignored, and a command may extend to any number of lines. Comments may be included with input from any source by using a percent sign "%". When this is encountered, all remaining characters on the same line are ignored.

Input is a list of statements:

<statement> <statement> ... <statement>

Each statement begins with a key word, and many are followed by arguments separated by commas. For example, in

FILE A,B,C

FILE is the key word, and A, B, and C are the arguments. Key words may be abbreviated by omitting trailing characters, as long as the abbreviation is unique among the entire group of key words. For instance, the previous statement could have been entered as

FI A,B,C

If a syntax error is found, the current input line is echoed with two question marks inserted after the point at which the error was detected. This is followed by an error message (see Appendix). Plib86 must then be re-executed.

If an error occurs during file processing, Plib86 terminates with an error message also listed in the appendix.

Object Files

FILE, LIBRARY, SEARCH

Plib86 must be told what object files and libraries to use for input and what modules to select from them. The FILE command is typically used, and normally causes all modules with the given files to be processed:

```
FILE COSINE, SIN, ARCTAN
```

The LIBRARY and SEARCH commands are similar, but are used only on libraries and select only those modules that define a public symbol that is needed by some other module that has already been processed. This is called a "library search" and is a process carried out by most linkage editors. It insures that only those library modules that are actually needed are included in the program.

```
LIBRARY MATHLIB
SEARCH FORTRAN
```

The LIBRARY command causes the given libraries to be searched once. When the SEARCH command is used the libraries are searched repeatedly as long as undefined symbols remain. This won't be needed unless two or more libraries are being searched that each refer to symbols defined in the others.

If Plib86 can't find a requested object file, and is running under the MSDOS 2.0 operating system, it will look in the environment for a string named "OBJ". The value of this string is

assumed to be one or more directory path names, separated by semi-colons (just like the MSDOS 2.0 PATH command). These path names are appended to the front of the object file name (any disk drive ID is removed first) one at a time in an effort to find the file. The path name separator '\' is added between the path name and file name. For example, if

```
SET OBJ = \OBJECT; \LIBRARIES
```

were entered before running Plib86, and file TEST.OBJ was being searched for, it would look for \OBJECT\TEST.OBJ and \LIBRARIES\TEST.OBJ. This means that commonly used object files can be left in a directory for use by many programs.

If an input file can't be found by using the OBJ path names, or if MSDOS 2.0 is not the operating system being used, the operator will be asked to enter a file name prefix string (e.g. "A:" or "\OBJECT\" that will be appended to the front of the file name after stripping any drive id . Diskettes may be changed at this time if necessary. Of course, the operator must insure that any diskettes removed do not contain open files like the BUILD or EXTRACT file. Also, if Plib86 runs out of memory a work file is opened on the default disk, which then may not be removed.

AS

If an object file (not a library) is being processed the module it contains is given the same module name as the name of the file it came from. This is done because some compilers don't supply a unique module name. This default may be changed by using the AS statement. It supplies the module name for the last file name given. For example,

```
FILE MATH1 AS COSINE
```

would name the module in MATH1 COSINE instead of MATH1.

INCLUDE, EXCLUDE

The modules selected from a library may be further restricted by using the INCLUDE and EXCLUDE statements. These are followed by a list of module names:

```
FILE MATHLIB INCLUDE SIN, COSINE
LIB MATHLIB, DB EXCLUDE BTREE
```

The INCLUDE statement causes only those modules listed to be considered for processing, and this selection precedes a library search. EXCLUDE is the opposite. The modules listed are not processed. INCLUDE and EXCLUDE apply to the FILE, LIBRARY or SEARCH file immediately preceding. In the second example above, for instance, the EXCLUDE BTREE applies only to the DB library, not MATHLIB.

Building a Library

BUILD

The BUILD command is used to create a library out of the modules selected from the input files. It is followed by the name of the file to create. The file type defaults to .LIB:

```
BUILD DB.LIB
BUILD D:MATHLIB
```

After all modules are output the library index is created.

One must be careful that the output file does not have the same name as any of the input files. For instance, entering

```
BUILD MATHLIB
FI COSINE, ARCTAN, MATHLIB
```

won't work because MATHLIB will be erased before it is read.

The BUILD command may not be used simultaneously with the EXTRACT command (described next). If no output is requested from Plib86 (i.e there is no BUILD, EXTRACT or LIST command) then Plib86 will simply read the input modules and report any errors it finds.

INTEL

By default, the BUILD command constructs a Microsoft format index for the library file under construction. When this statement appears, however, an INTEL format index is constructed instead. No arguments are required. When creating an INTEL format index, the LOWERCASE statement may have to be used to inhibit translation of symbol name characters in the index to upper case. Some compilers using Intel format libraries distinguish between upper and lower case when comparing symbol names.

Extracting a Library Module

The EXTRACT command is used to extract a single object module from a library file and place it into a separate disk file. It is followed by the name of the file to create:

```
EXTRACT COSINE.OBJ
EXTRACT ARCTAN
```

If the file type is omitted OBJ is assumed.

The EXTRACT command extracts the first module found in the input files. Therefore it is usually necessary to use the INCLUDE statement to specify which library module should be extracted. For instance,

```
EXTRACT COSINE FI MATHLIB
```

extracts the very first module in MATHLIB, even if it is not the COSINE module. To get the correct one enter

```
EXTRACT COSINE FI MATHLIB INC COSINE
```

Generating Reports

The LIST command is used to generate reports about the object files being processed. It may optionally be followed by a file name, causing the reports to be directed to that disk file or device. The file name must be preceded by an equal sign. Then a character is entered for each report desired, separated by commas. There are two reports available:

M - A list of all modules processed in alphabetical order. Next to each module is listed all of the symbols defined within it.

S - A list of all public and external symbols in alphabetical order. Each is followed by the name of the module defining the symbol in parenthesis (this will be blank for symbols not defined by any module read). Following this is an alphabetical list of all modules that access the symbol (i.e. this is a cross-reference report).

Here are some examples:

```
LIST M
LIST = DB.LST M, S
LIST = XREF.LST S
```

WIDTH, HEIGHT

The report generator can be re-configured for different size paper. It assumes 80 columns and 66 rows per page as a default. The number of columns may be changed with the WIDTH command, and the number of rows with the HEIGHT command. Here are some examples:

```
WIDTH 132
HEIGHT 88
```

BRIEF

The S option of the LIST command can be quite long. If the BRIEF command is used, however, all undefined symbols are deleted from the report, making it more manageable. These undefined symbols might be from libraries that you did not search in creating the report, and might not be necessary in the report. The BRIEF statement has no arguments.

Controlling the Library Index

NOINDEX

Normally all public symbols from all modules are inserted into the library index. If a duplicate symbol is found library creation continues but a warning message is given and the index entry for that symbol will select the first module defining the symbol.

Sometimes it is useful to exclude certain symbols from the library index. This may be accomplished by using the NOINDEX command. For example,

```
NOINDEX SYM1, SYM2, SYM3
```

excludes SYM1, SYM2, and SYM3 from the index.

Suppose you wish to create a library that contains several versions of the same module, for instance a device driver for some kind of hardware. If you try to place all of the modules into the library you will get duplicate symbol warnings, and at link time the linkage editor wouldn't be able to select the desired module.

This can be made to work by using NOINDEX on most of the module entry points. This excludes all of these symbols from the library index. To get the linkage editor to select the correct module insert an un-used but unique dummy symbol into each one. At linkage edit time one of these dummy symbols would be accessed in order to create a need for the desired module. The linkage editor

would then select it when the library is searched.

Using Plink86, for instance, one could use a statement like

```
DEFINE FOO=DRIVER1
```

to select the module containing the "driver1" dummy entry point. An alternative which works in a Microsoft format library is to rely on the fact that the name of each module is actually in the library index as well, followed by an exclamation point. For example, if the library contains a module named DRIVER1 then there will be a dummy index entry named DRIVER1!. These symbols can be used instead of creating a dummy module entry point as discussed above.

BLOCKS

The Microsoft library index consists of a prime number of hash blocks. Plib86 will choose the amount of index space needed so that everything fits and then adds about a 10% slop factor. The extra is added because the hash blocks are set up as a "scatter table" (see Knuth's volumes of computer programming) and search time can increase dramatically as the blocks become nearly filled. However, if the linkage editor reads most or all of the index into memory when doing a library search (as Plink86 does) this may not matter too much. The extra time spent comparing identifiers is more than made up for by the savings from reading fewer index blocks from disk.

The BLOCKS command functions only when the BUILD command is used, and specifies the number of index blocks to be used. For example,

```
BLOCKS 7
```

forces Plib86 to use 7 blocks. If some of the symbols won't fit into the index Plib86 will print warning messages. If the argument to the blocks command is not a prime number Plib86 will increase it until it is. The limit on the number of library index blocks is 997.

Miscellaneous Commands

VERBOSE

When processing a large library file it is sometimes useful to know what Plib86 is doing. When the VERBOSE statement is used Plib86 will maintain a status line at the bottom of the CRT screen indicating what is going on. This statement should not be used on a hard-copy terminal.

BATCH

If Plib86 can't find an object file or library it will normally prompt the operator to enter the name of a disk drive or directory path name where the file may be found. The BATCH command will cause Plib86 to stop with a fatal error without prompting the operator. It is useful when running Plib86 from within a batch file and no operator is available to respond to a prompt.

LOWERCASE

Any object files and libraries conforming to the Microsoft standard normally use only upper case letters in identifiers. Therefore Plib86 normally translates all lower case letters to upper case. This statement inhibits this translation for all identifiers found in object files, library indices, or Plib86 commands. It is sometimes necessary to use this command when an Intel format library is being built (see the INTEL command).

Appendix A - Warning Messages

Occasionally Plib86 detects a situation that looks like it might be a problem when the input or output object files are processed by the linkage editor. It then issues a warning message and continues to execute. These messages should be self-explanatory, but a number is also given that may be looked up in this appendix to get a more complete explanation of what has happened.

1 - There may be only one definition for each global (i.e. PUBLIC) symbol in the object modules being processed. Plib86 ignores the duplicate definition and retains the first one for use in any library index or reports being generated.

2 - Each record in an object file contains a check field at the end for validation purposes. This message indicates the checksum was bad, but processing continues. These messages are inhibited after a few have been printed. Was the object file patched on disk before Plib86 read it? Typically people who patch object files don't bother changing the checksum. Also, some compilers and other librarians seem to be sloppy about making sure the checksums are correct. If the file is really smashed a fatal error will probably occur soon after this message appears.

3 - Each record in an object file is preceded by a word giving the record size. This error means that Plib86 reached the end of the record and found that the number of bytes processed is different from the specified size. The object file is probably smashed, but Plib86 will attempt to continue reading it.

4 - There is no room in the Microsoft library index being created with the BUILD command for the named symbol. You probably used the BLOCKS command to reduce the size of the index, and now it is too small to hold everything. This warning should never occur if you haven't used the BLOCKS command: contact Phoenix Software if it does.

Appendix B - Error Messages

When a fatal error is detected by Plib86 a console message is printed which should be self-explanatory. However, an error number is also printed which may be looked up in the table below. A longer discussion of the error will be found there.

Command Syntax Errors

These errors are caused by mistakes made in the input given to Plib86. Re-run Plib86 after correcting the problem. The input line causing the problem will be displayed on the terminal, with a couple of question marks inserted after the point where the error was detected. These should aid in locating the problem, but occasionally Plib86 may not detect the error until more text is processed. In other words, if the error location is given as the front of a line, check the end of the previous line.

1 - "@" files are nested too deeply. Only three levels of "@" files may be active at any given time. Do you have a loop in your "@" file references?

2 - Disk error encountered while reading "@" file. Try re-building the file.

5 - The item given for input at this point is too large. The maximum size allowed is 64 characters.

6 - Invalid digit in number (i.e. not Ø thru 9).

1Ø - Invalid file name. The input stream should contain a valid file name for the particular operating system being used.

11 - Expecting a statement. A key word which begins a statement should be present here.

12 - The INCLUDE and EXCLUDE statements may not be used simultaneously on the same input file.

14 - Expecting identifier. A section, module, segment, or symbol name must be entered at this point.

16 - Expecting a value. An expression or 16-bit quantity must appear at this point.

17 - No files were given to process! You must use the FILE statement and specify at least one input file.

18 - The BUILD and EXTRACT commands may not be used simultaneously. You must run Plib86 twice with one command in each.

Work File Errors

When Plib86 runs out of memory it opens a work file on disk named Plib86.WRK to hold the description of the library. These error codes indicate a problem with processing the work file.

30 - The work file can't be created. Probably there is no space in the disk directory.

31 - An I/O error occurred while writing the work file.

32 - An I/O error occurred while reading the work file.

33 - An I/O error occurred while positioning the work file.

34 - There are too many module description objects in this library (about 50,000 symbols, modules, and so on may be defined). This library is too large for Plib86 to handle.

Input Object File Errors

The following errors have to do with the object files that are given to Plib86 to process. Usually they occur when a file has been corrupted somehow. Try re-compiling to get a new copy of the object file. If it is a library supplied by the compiler manufacturer that is causing the problem, try to get a fresh copy of it.

41 - Premature end of input object file. The end of the indicated file was reached unexpectedly. Possibly, the file was truncated by copying it with a program that assumes a CNTL-Z (1AH) is end of file.

42 - Fatal read error in object input file.

Output File Errors

The following errors are caused by a problem in creating the output code file or memory map file (when written to disk). Often, they are caused by a full disk or disk directory, a disk that is write-protected, or some kind of hardware problem with the disk.

45 - Can't create output disk file. Possibly the disk directory is full, or the disk is write protected.

46 - Output file too large. The given modules won't fit into the library. You will have to break up the library into one or more smaller ones.

47 - Fatal disk write error in output file. Possibly the disk is full or write protected, or some kind of hardware error has occurred.

48 - Fatal disk read error in output file. An irrecoverable hardware error has probably occurred.

49 - Can't close output file. The disk is probably write protected, or a hardware error has occurred.

5Ø - Can't create the LIST output file. Possibly the disk directory is full, or the disk is write protected.

Miscellaneous Errors

51 - There are too many symbols to be placed into the library index. You will have to break up the library into one or more smaller ones.

52 - No modules were selected (by library search, INCLUDE, or EXCLUDE) to be placed in the output file (BUILD or EXTRACT).

54 - There isn't enough memory in the computer to run Plib86. You must have a really tiny memory - better buy more!

Plib86 Bugs

These errors indicate a bug in Plib86 has occurred through no fault of your own. They are listed here for completeness in the manual, although it is unlikely that you can do anything to correct them. Try running Plib86 again. If the error persists, please gather the relevant information and contact Phoenix Software Associates.

201 - No NeedRead Buffers (NRnew).

205 - Seek errors while writing output file (attempt to seek past end of file).

210 - Requested record size too large (Newrec).

219 - Bad object block (GetBlock).

221 - Invalid object key (Q).

222 - Invalid object key (QM).

We ask that you make a reasonable effort to solve your difficulties yourself before contacting us, and to phone only if you are trying to deal with an emergency. Otherwise, please report your problem in writing. We can read much more quickly than we can listen. Our address is:

Phoenix Software Associates, Ltd.
1420 Providence Highway
Suite 260
Norwood, MA 02062

Be sure to include with your description of the problem the input you are trying to use and where your object files came from. If you like, send input, object, and library files on diskettes (MSDOS 1.1 or 2.0 5 1/4 format, single or double sided), and instructions on how to run the software to make the error condition occur. Source files are usually unnecessary. We will be happy to sign non-disclosure agreements to protect your software, if having it will help us identify a bug more quickly, and to return the diskettes to you after the problem has been identified.

LIFEBOAT ASSOCIATES SOFTWARE PROBLEM REPORT

Please use this form to report errors or problems in software supplied by Lifeboat Associates. This form is designed to act as a transmittal sheet.

Software Product Name: _____ Media Format: _____

Version No.: _____ Serial No.: _____ Invoice No.: _____

Purchased From: _____

_____

_____

Date of Purchase: _____ Return Authorization #: _____
Has the software registration card been returned? _____

Computer Used: _____ CPU (8080/8085/Z-80): _____

Disk Capacity: _____ Number of Drives: ____ Memory Size: _____

Operating System/Version (If not listed above): _____/_____

Software used with the above product, (e.g. list the BASIC used if you are reporting a problem with a Payroll program that uses it).

Name of Software	Version
_____	_____
_____	_____

Does the software come with sample or test programs? _____
If so, have you been able to use them successfully? _____

Please describe the problem you have encountered. Include references to the manual if appropriate. Try to reduce the problem to a simple test case. Enclose any appropriate programs (preferably on disk). If you feel that the problem may be caused by the disk being defective, you may prefer to return the original disk with this report to achieve the fastest resolution of the problem. (If so, call for a Return Authorization No. A handling charge may be incurred. No handling charge will be made if a product or portion thereof is returned **DUE TO DISKETTE MEDIA DEFECTS** within 30 days from the date of sale).

- Information on product changes, bugs, fixes and current version numbers are published in **Lifelines**, our software newsletter.

PROBLEM DESCRIPTION: (Continue on additional pages if necessary)

Area Phone Num. Ext.

Name: _____ (____) ___-_____(____)

Address: _____ (____) ___-_____(____)

City: _____ State: _____ Zip Code: _____

Return to: Lifeboat Associates
1651 Third Avenue
New York, N.Y., 10028

002prob.bn.09.81

Technical assistance is available Monday - Friday, from 11:00 a.m. to 7:00 p.m., Eastern time.
1-(212) 860-0300
TWX: 710-581-2524 Telex: 640693