Lattice 8086/8088 C Compiler

SECTION 1   The MS-DOS Implementation

The Lattice 8086/8088 C compiler 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, it lacks the ability to address this memory directly and efficiently. This implementation therefore restricts the size of C programs to 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). Even with this restriction, programs of considerable complexity and power (including the compiler itself) can be developed.

1.1 Operating Instructions

     The Lattice compiler is supplied under MS-DOS as a package consisting of the following files:

                    LC1.EXE   C compiler (phase 1)
                    LC2.EXE   C compiler (phase 2)
                    FXU.EXE   Function extract utility
                    C.OBJ   C program entry/exit module
                    LC.LIB   Run-time and I/O library
                    LC.BAT   Batch file to execute phases 1 and 2
                    STDIO.H   Standard I/O header file
                    CONIO.H   Console I/O header file
                    CTYPE.H   String macro header file
                    FTOC.C   Fahrenheit-to-Celsius sample program
                    CAT.C   File concatenate sample program
                    SIEVE.C   Eratosthenes sieve sample program
                    IO.ASM   Sample assembler program

     These disk files take up about 160 kilobytes of disk storage. Each phase of the compiler itself has about 50K bytes of program section, and each requires a minimum of an additional 14K bytes of data area. Thus, the compiler needs about 64K bytes of working memory in addition to that taken up by MS-DOS itself, and additional memory will be needed to compile large source files. (On the MS-DOS system used to develop the compiler, the MS-DOS components required slightly more than 16K bytes.)

LC1 and LC2 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. The batch procedure file LC.BAT is provided to execute LC1 and LC2 in succession, using the same file name (the normal sequence). The compilation process can be diagrammed 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 a .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 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 1.3.2 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:

```
C.OBJ + user.OBJ + ... + LC.LIB  ->  LINK  ->  user.EXE
```

The special files required are C.OBJ and LC.LIB. First, the file C.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 LC.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, here are the commands necessary to compile, link, and execute the Fahrenheit-to-Celsius sample program. This example assumes that all of the .EXE files (LC1, LC2, and LINK) reside on the same disk. The commands will be shown in upper case, although lower case commands will work just as well. (Note: the linker prompts described here are those for Version 1.10 of the Microsoft linker; consult Microsoft's documentation if they are different for the version you have. Generally, the default responses are correct.)

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

```
                        LC1 FTOC<CR>
```

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<CR>
```

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

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

```
                        LINK C FTOC<CR>
```

Note that C (meaning C.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. Respond to the other linker prompts as follows:

```
                Run File [C.EXE]: FTOC<CR>
                List File [NUL.MAP]: <CR>
                Libraries [.LIB]: LC<CR>
```

These responses cause the run file to be named FTOC.EXE, skip the generation of a link map, and cause LINK to search LC.LIB for external references.

STEP 4: Execute the .EXE file by typing

```
                        FTOC<CR>
```

A list of Fahrenheit temperature values and their Celsius equivalents will be written to the user's console.

Note that the first two steps could have been accomplished with the single command

```
        LC FTOC<CR>
```

which uses the LC.BAT batch file to execute LC1 and LC2 in succession. Note also that the file FTOC.OBJ still exists and should probably be erased.

Detailed instructions for compiling, linking, and executing programs are presented in the following sections. See Section 1.3 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. Study the examples at
the end of each section to see how actual commands should look.

1.1.1 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 1.3.1 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]<CR>

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
1.1.4).

=stack   The first option is used to override the number of
          bytes reserved for the stack (see Section 1.4 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 a lot of embedded
          statements (ifs within ifs within ifs, etc.) causes the
          first phase to die mysteriously in the middle of a
          compilation, or to complain of errors which don't
          exist, or to exhibit other unusual behavior, increasing
          the stack size MAY solve the problem. On the other
          hand, you may simply have discovered a compiler bug;
          see Appendix B for the procedure used to report such
          problems. 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 signon message and any error or warning

messages which may be generated. The full filename must be specified, including extension. 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 file name 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 is used unless some other drive is specified; the quad file is created on the same drive as the source file unless the -o option is used (see below). Alphabetic characters may be either upper or lower case in file names.

options   Compile time options are specified as a minus sign 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 minus sign 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 is almost never required unless the programmer is doing something tricky with pointers; see Section 1.3.4 for more information.

-b   Forces byte alignment for all offset calculations. The first phase normally aligns all objects which are not "char" on a word boundary. This insures efficient data fetches on an 8086 (fetching a word on an odd byte boundary on the 8086 processor requires four additional clock periods). This option is provided to allow space efficient programs for the 8088 processor. It is also useful for certain structure declarations where word items must be placed at odd byte offsets in order to conform to specific record layouts (for example, the FCB structure used in MS-DOS contains a long integer which falls on an odd byte 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 records are
            interspersed with the normal quads. This allows the
            second phase to produce a table of information relating
            input line numbers to program section offsets. If this
            option is used, the quad file is NOT deleted by the
            second phase. (Note: this option is not implemented
            on some earlier versions of the compiler.)

-od   Creates the output file (the quad file) on drive "d",
            where "d" is a single alphabetic character, either
            upper or lower case, specifying a disk drive ("a" for
            A:, "b" for B:, etc.). The drive letter must be
            adjacent to the "-o" (no intervening blanks).

-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: 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. The option is
            provided for use on programs written for the BDS C
            compiler; see Appendix C for more information.

EXAMPLES

LC1 XYZ -ob -x

Execute the first phase of the compiler using file XYZ.C as
input, creating file XYZ.Q on B:, and interpret all external
declarations without a storage class as being "extern"
declarations.

lcl =4096 >tns.err tns

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

1.1.2   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 1.3.2 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]<CR>

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 1.1.4 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
           drive is used unless some other drive name is
           specified, and the object file is created on the same
           drive as the quad file unless the -o option is used
           (see below).

options   Compile time options are specified as a minus sign
           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 minus sign and letter (that
           is, they cannot be combined as they can for certain
           UNIX programs). Current options include:

-f   Causes code for floating point operations to be
           generated using the 8087 numeric data processor. Note
           that this option must be used for all functions which
           perform any floating point operations which are to be
           included in a program, that is, functions compiled with
           the -f option cannot be combined with (floating point)
           functions compiled without the -f option. (Note: this
           option is not implemented on some earlier versions of
           the compiler.)

-od   Creates the output file (the object file) on drive "d",
           where "d" is a single alphabetic character, either
           upper or lower case, specifying a disk drive ("a" for
           A:, "b" for B:, etc.). The drive letter must be
           adjacent to the "-o" (no intervening blanks).

EXAMPLES

LC2 A:NXF -f

Execute the second phase of the compiler using file A:NXF.Q as

input, creating file NXF.OBJ on drive A:, and generate code for
all floating point operations to use the 8087 floating point
processor.

lc2 u790 -oc

Execute the second phase of the compiler using file U790.Q as
input, creating file U790.OBJ on drive C:.

1.1.3 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
1.5.)   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. The file C.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. The file LC.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 "C" 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 "C" specification.

      3. Typing "LC" 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, you will get a message to the effect that "Unsatisfied external reference(s)" exist, followed by a list of the external names which were not satisfied. DO NOT ATTEMPT TO EXECUTE A PROGRAM WITH UNSATISFIED EXTERNAL REFERENCES unless you are quite sure that the missing functions will never actually be called.

See Section 1.2.2 for a discussion of external names. See Section 1.4 for a technical description of the object code features used in this implementation. If the version of the linker supplied with your system has different prompts than those illustrated here, consult Microsoft's documentation. Generally, the default responses to other prompts are correct. If your linker allows generation of a public symbol map, you may want to create a .MAP file and look at the components present in the resulting load module.

EXAMPLE

LINK C XYZ QRS

    Run File [C.EXE]: XYZ<CR>
    List File [NUL.MAP]: <CR>
    Libraries [.LIB]: LC<CR>

Execute the linker, producing XYZ.EXE as an executable program, and include files XYZ.OBJ and QRS.OBJ in the program.

1.1.4 Program Execution

When a C program is executed, the function "main" is called
to begin execution. Two important services are performed for it
before it ever 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 input and output using the
standard "getchar" and "putchar" macros.

The simplest way to execute a C program is just to type the
name of the .EXE file (without the .EXE extension), followed by a
return. 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] <CR>

Everything after "pgmname" is optional, as the brackets indicate.
The various additional items, if present, must be specified in
the order shown.

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 1.4 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, must be specified.   See Section
1.5.2 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 1.5.2 for a list of valid device names.   The
file is opened as a new file, which discards its
previous contents if it 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 a
convenience 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 1.5.4 for details.

EXAMPLES

CPROG =8000 <INPUT.R PQP 12

Execute CPROG.EXE, setting the stack size to 8000 decimal bytes,
with "stdin" connected to 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

Execute 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.

1.1.5 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 program FXU.EXE is provided so that
groups of small functions may be kept in a single source file and
object modules produced for them individually. The utility
operates by extracting the source text for a specified function
and creating a single source module which can then be compiled to
produce an object module. The format of the command to invoke
the utility is as follows:

        FXU filename function <CR>

where "filename" is the name of the file containing several
functions and "function" is the name of the particular function
to be extracted. The first file name must be specified WITH an
extension, if one is defined; the second name (that of the
function) should be specified without any extension. If the

named function is found, a file of the same name with a .C extension is created; otherwise, an error message is generated. The following limitations of the utility should be noted:

1. The function name must be specified exactly as it appears in its definition; if alphabetic characters are lower case in the source file, they should be lower case in the command. The name of the file created, however, will have all lower case letters converted to upper case.

2. The user must be careful not to specify a function with the same name as the original source file, that is, if "xyz" is being extracted from XYZ.C the original contents of the file will be lost.

3. The text extracted consists of all the characters between the closing brace of the previous function, up to and including the closing brace of the extracted function. Obviously, there are problems with functions that refer to external data locations defined in the source module; in general, FXU should be used only for groups of functions which do not refer to any external data locations defined in the same module.

4. The program counts braces defined in the body of the function in order to determine when it has reached the end of that 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.

EXAMPLE

fxu sfuncs.c movstr

Extract the function called "movstr" from the text file "SFUNCS.C", and create a new file "MOVSTR.C" to contain the text of that function.

1.2 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. See Section 2 of the manual for a description of the machine-independent features of the Lattice implementation of the language.

1.2.1 Data Elements

    The standard C data types are implemented according to the
following descriptions. All data elements are normally aligned
on a word boundary, with the exception of "char" variables; as
noted in Section 1.1.2, 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	-2×10^{9} to 2×10^{9}
float	32	$+/- 10^{-37}$ to $+/- 10^{38}$
double	64	$+/- 10^{-307}$ to $+/- 10^{308}$

"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 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 a C
program cannot exceed 64K bytes, the address of an item is fully

specified in 16 bits. Pointers to functions consist of the 16-bit offset of the first byte of the code defining the function. Again, since the combined size of all the functions in a C program cannot exceed 64K bytes, the address of the function is fully specified in 16 bits.

1.2.2 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.

1.2.3 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 do the header files
included with the compiler package. Alphabetic characters in a
file name may be specified in either upper or lower case. The
file must be present on the currently logged-in disk unless a
drive specifier is included in the file name (not recommended).
The file name is retained internally by the compiler for error
reporting (see Section 1.3.3).

1.2.4 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 well-behaved. The following 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.

1.2.5 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 "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 1.2.1). 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. Beware, however, of 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); such code may consider (falsely) negative zero to be not zero.

As noted in Section 1.1.2, a compile time option selects whether code is generated to perform floating point operations using the 8087 co-processor. The default option generates calls to library functions for arithmetic and comparison operations. Note that the two classes of code generation cannot be combined in the same program; in other words, all functions in the same program which use floating point variables must be compiled with one option or the other. Combining functions compiled with different floating point options will have disastrous results.

Otherwise, the calculations performed by either option should be very nearly equivalent. The library functions used if the 8087 is not present perform arithmetic calculations using 64 fraction bits and a 16 bit exponent, just as the 8087 does. Intermediate results, however, must be converted back to the "double" representation, while on the 8087 they can be left in the more precise "temporary real" format. This may cause some loss of precision in certain cases. For example, in the sequence

```
double a,b,c;
    .   .   .
a = a * b / c;
```

the intermediate "a * b" result remains in the expanded temporary format on the 8087 register stack but requires conversion back to

"double" in the default case. Please note that the library
functions which perform the arithmetic operations without using
the 8087 were coded for accuracy, not speed, using
straightforward, unsophisticated algorithms. If the speed of
floating point arithmetic is a major consideration, the user
should obtain a system with the 8087 co-processor and use the -f
option for compiling floating point modules. (Note: this option
is not implemented on some earlier versions of the compiler.)

     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

          0 = invalid operation (8087 only)
          1 = underflow
          2 = overflow
          3 = divide by zero

Note that "invalid operation" is detected only for 8087
operations, and signals that an operand was a NAN or a result
indeterminate.

     The standard version of CXFERR supplied in LC.LIB simply
ignores all error conditions. The user may write a different
version (in either C or assembly language), if desired, 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   Non-8087: set the result equal to zero. 8087:
               denormalize the result.

Overflow   Set the result to plus or minus infinity.

Zerodivide   Non-8087: set the result equal to zero. 8087: set
               the result to plus or minus infinity.

     Consult the 8087 description for more information about the
floating point formats and the other special features of the
8087.

1.2.6 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.

1.2.7 Register Variables

The current version of the compiler does not implement register variables, although declarations using "register" are accepted if properly made. Storage is reserved for these objects as if they had been declared "auto". Future versions of the compiler may elect to support register variables.

1.3 Compiler Processing

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

1.3.1 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 subexpressions 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's a pretty safe bet that the compiler has crashed. Consult Appendix B for the compiler bug reporting procedure if this happens.

When the first phase begins execution, it writes a signon 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 (due to lack of directory space). This message identifies the version of the compiler which is being executed. No other messages are generated unless the source file

contains errors; see Section 1.3.3 for information about error processing. Note that the quad file is deleted if any fatal errors are detected.

1.3.2   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 1.4 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 persist more than several seconds. Anything longer than a thirty 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 signon 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 (due 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 1.1, 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. A special utility program (FXU.EXE) is provided so that multiple functions may be stored in a single .C file and extracted individually for compilation; see Section 1.1.5.

1.3.3 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. This prevents the second phase from attempting to generate code from an erroneous quad file, in the event that it is executed next (as in the procedure LC.BAT). 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 useful for obtaining an expanded explanation of the error from Appendix A; and "error message text" is a brief description of the error condition. 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 1.1.1). A message similar to the one above but with the text "Warning" instead of "Error" is generated for non-fatal warnings; 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 sort of "cascade" effect. In general, the programmer should attempt to correct the obvious errors first and not be too 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. Refer to 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.

1.3.4 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 is subject to the same 64K byte data space limitation as are all C programs generated by the Lattice compiler, 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 of the sources of the extra overhead in buffering the function in memory derives from the fact 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 which passes control directly to another branch instruction is re-routed to branch directly to the target location.

2. The combination of a conditional branch instruction which 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 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 sequence described next.

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 1.4.3 for more information.

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; this
forces the compiler to assume worst-case aliasing (which is the
compiler jargon for this pointer overlap we have been discussing)
when generating code. We have designed the compiler to operate
this way because we believe that the cases of 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 all but a few cases. The "-a" option is then
provided for use on programs which violate that assumption.

One 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.

1.4   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.

Without mention of the specific object code details used to
create it (which will be divulged in subsequent sections), the
general structure of a C program is illustrated by the following
diagram.

1.4.1 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.

PROG is the segment which includes the instructions which perform the actions specified by any functions defined in the source file.

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.)

Both segments are defined to be combinable with other segments of the same name. PROG 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 1.1.1) 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 1.4.4.

1.4.2 Linkage Conventions

In order to guarantee that both the program and data portions of the final linked program do not exceed 64K bytes, two groups are defined in the object code.

PGROUP = BASE segment + PROG segment

　　　　　　　DGROUP = DATA segment + STACK segment

The PROG and DATA segments are obtained from the C modules in
program, as discussed in the previous section. The other
segments are defined in the module C.OBJ, which must be the fi
module encountered during linking. The BASE segment serves
purposes: (1) it forces PGROUP lower in memory because it is
first segment within C.OBJ, and (2) it contains text wh
identifies the current compiler revision number. The lat
feature allows programs to be examined with the debugger
determine the revision of the library used when the program
linked. The STACK segment has a dual role as well: (1)
defines the base of the stack and dynamic memory portion of
data section of the program, and (2) it satisfies the linke
need for a segment of type STACK (if one is not encountered,
linker generates a warning message). With these gr
definitions, the address of a function is its offset from
base of PGROUP, and the address of a data element is its off
from the base of DGROUP.

　　　　The module C.OBJ also defines its own PROG and D
segments. The PROG segment defines the initial execution addr
of the linked program. The segment registers are initializ
and the amount of memory remaining above the STACK segment
determined. The stack pointer is adjusted to its maximum val
as noted in the discussion at Section 1.4. In the DATA segm
of C.OBJ, the address of the stack base and top are saved for
by the memory allocation functions. At the top of the stack,
address of the program segment prefix is saved so that an orde
return to MS-DOS can be made when the program terminates.
characters from the command line which executed the program
transferred from the program segment prefix to the stack.
pointer to this copy of the command line is then passed to
function "_main", which begins execution of the program (
Section 1.5.4).

　　　　As noted in Section 1.2.2, external names differ f
ordinary identifiers in C in that upper and lower case lett
are equivalent. The relocation information in the object c
defines all external names relative to either PGROUP (functio
or DGROUP (data). All external names are defined as
unspecified type, that is, there is no set of attribu
associated with the name; it is simply an offset within one
the other of the two defined groups. It is therefore an error
define two items with the same external name in the same progr
It is the programmer's responsibility to prevent this occurre
and also to make sure that programs refer to external names i
consistent way (i.e., a function should not refer to "xyz"
"long" when it is actually defined as "int" in some ot
module). External definition and reference from assem
language modules are discussed in Section 1.4.4.

　　　　Consult the appropriate linker documentation for informat
as to how to obtain a public symbol map for a linked program.
a convenience, the DGROUP segments are defined with class n

DATA and the PGROUP segments with class name PROG.
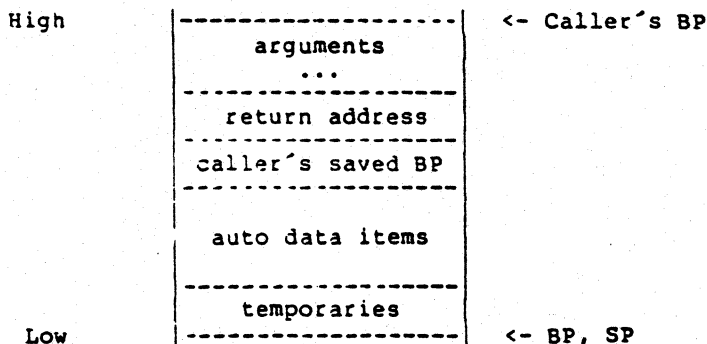
1.4.3 Function Call Conventions

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 because all functions are defined within 64K bytes. 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. The stack pointer SP is moved into BP to allow addressing of the elements on the stack: function arguments, "auto" storage, and temporaries.

The offsets of the various components are indicated by the following diagram. Note that of the registers used by the calling program, only BP is saved.

```
High        |--------------------|    <- Caller's BP
                    arguments
                      ...
            |--------------------|
                return address
            |--------------------|
             caller's saved BP
            |--------------------|

              auto data items

            |--------------------|
                temporaries
Low         |--------------------|    <- BP, SP
```

During execution of a C function, BP and SP normally contain the same value. The temporaries are allocated closest to BP, followed by the "auto" elements declared, in the order of their declaration. This addressing scheme has the disadvantage that

1-28

the arguments supplied to the function are at an off
determined in part by the amount of "auto" storage declared.
the function declares more than about 124 bytes of "au
storage, the arguments require an additional offset byte in
instructions which refer to them.

    The compensating advantage to this mechanism appears whe
function calls another function and supplies it with argum
values.   Because a C function may in special cases hav
variable number of arguments ("printf" is the classic examp
the called function cannot de-allocate the stack space usec
pushing the argument values; the calling function must do so.
retaining the normal SP value in BP, Lattice C functions
restore the stack pointer after a function call with the two-l
instruction

          MOV SP,BP

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

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

    When a function returns to its caller, it first loads
function return value, if any, into predefined registers.
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 l
of the value.   Double precision functions compiled with the
option return the function value as the top of the 8087 regis
stack.

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

1.4.4 Assembly Language Interface

    Programmers may write assembly language modules
inclusion in C programs, provided that these modules adher(
the object code, linkage, and function call conventions descri
in the preceding sections.   An assembly language module wr

                                                              ]

defines one or more functions to be called from C must begin with
the statements

```
        PGROUP GROUP    PROG
        PROG   SEGMENT BYTE PUBLIC 'PROG'
               ASSUME  CS:PGROUP
```

followed by PUBLIC declarations of the function(s):

```
        PUBLIC  AFUNC
          .
          .
          .
    AFUNC  PROC    NEAR
```

The function itself must be declared NEAR, as shown above, and
must conform to the conventions detailed in the preceding
section. If a value is to be returned by the function, it must
be placed in the appropriate register(s).

    A module may similarly define data locations to be accessed
(using "extern" declarations) in C programs by defining a DATA
segment, as in the following example:

```
        DGROUP GROUP    DATA
        DATA   SEGMENT WORD PUBLIC 'DATA'
               ASSUME  DS:DGROUP
               PUBLIC  DX1,DX2,DX3
        DX1    DW       4000H
        DX2    DW       8000H
        DX3    DB       'Text string'
        DATA   ENDS
```

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.)

    To call a C function from assembly language, an EXTRN
declaration for the function must be included after the SEGMENT
directive, and the caller must supply any expected arguments in
the proper order (see Section 1.4.3).

```
                    EXTRN     CFUNC:NEAR
                      .
                      .
                      .
                    CALL      CFUNC
```

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 applied to external reference:

```
                    DW        DGROUP:XD1
```

must be used to define the address of XD1.

Remember that 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".

The following example (a portion of one of the operating system interface routines used in the MS-DOS implementation) illustrates many of the requirements discussed above, and shows how a STRUC may be used to address elements on the stack.

```
BDOS_OPEN EQU 0FH          ;Open existing file
BDOS_CREATE EQU 16H        ;Create new file

PGROUP  GROUP    PROG
;
; Dynamic storage layout for XCMAKE, XCFIND
;
DYNS    STRUC
        DB       ?
INTNO   DB       ?                 ;Save BDOS interrupt number here
OLD_BP  DW       ?                 ;Caller's BP save
RETN    DW       ?                 ;Return address from call
ARG1    DW       ?                 ;First argument
ARG2    DW       ?                 ;Second argument
ARG3    DW       ?                 ;Third argument
DYNS    ENDS

DYNSIZE EQU      2                 ;Size of storage to be allocated
```

```
;
; name          XCMAKE -- create new file
;
; synopsis      iret = XCMAKE(filename, pmode, area);
;               int iret;       return code: 0 if successful
;               char *filename; file to be created
;               int pmode;      access privilege mode bits
;               int *area;      for return of file pointer data
;
;description  This function is called by "creat" to create
;             a new file.  The access privilege mode bits are
;             ignored in this implementation.  The "area"
;             pointer gets a pointer to a device or file block
;             for the new file.  It is then supplied to XCIOS
;             when the connect call is made.
;
; returns       iret = 0 if file successfully created
;                    = -1 if error
;
PROG    SEGMENT BYTE PUBLIC 'PROG'
        PUBLIC  XCMAKE,XCFIND
        EXTRN   XCFINT:NEAR,XCFTRM:NEAR
        ASSUME  CS:PGROUP
XCMAKE  PROC    NEAR
        PUSH    BP
        SUB     SP,DYNSIZE
        MOV     BP,SP
        MOV     [BP].INTNO,BDOS_CREATE
        MOV     AX,[BP].ARG3    ;MOVE AREA ARG TO ARG2
        MOV     [BP].ARG2,AX
        JMP SHORT XCF01
;
; name          XCFIND -- find existing named file
;
; synopsis      iret = XCFIND(filename, area);
;               int iret;       return code: 0 if successful
;               char *filename; file to be found
;               int *area;      for return of file pointer data
;
; description   This function finds an existing file (or device)
;               and returns a pointer to the file block or device
;               block for it.  Except for the privilege mode
;               bits, the arguments have the same meaning as for
;               XCMAKE.
;
; returns       same as XCMAKE
;
XCFIND  PROC    NEAR
        PUSH    BP
        SUB     SP,DYNSIZE
        MOV     BP,SP
        MOV     [BP].INTNO,BDOS_OPEN
XCF01:  PUSH    [BP].ARG1       ;PASS FILE NAME TO XCFINT
        CALL    XCFINT          ;GET FILE/DEVICE BLOCK PTR
```

```
            MOV     SP,BP
            TEST    AX,AX           ;TEST RETURN VALUE FROM XCFINT
            JNZ     XCF02           ;BRANCH IF POINTER NOT NULL
            NOT     AX
            JMP     SHORT XCF05     ;RETURN -1
XCF02:      MOV     BX,AX
            MOV     DI,[BP].ARG2
            MOV     AL,[BX]         ;GET FIRST BYTE OF BLOCK
            TEST    AL,80H
            JNZ     XCF04           ;BRANCH IF DEVICE
            MOV     DX,BX           ;FCB ADDRESS INTO DX
            MOV     AH,[BP].INTNO
            INT     21H             ;OPEN/CREATE FILE
            TEST    AL,AL
            JZ      XCF04           ;BRANCH IF SUCCESSFUL
            PUSH    BX
            CALL    XCFTRM          ;FREE THE FILE ACCESS BLOCK
            MOV     SP,BP
            MOV     AX,-1
            JMP     SHORT XCF05
XCF04:      MOV     [DI],BX         ;RETURN PTR TO BLOCK
            XOR     AX,AX           ;SET GOOD STATUS
XCF05:      ADD     SP,DYNSIZE
            POP     BP
            RET                     ;RETURN TO CALLER
XCFIND      ENDP
XCMAKE      ENDP
            END
```

1.5 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 1.1.5 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 1.5.2. The structure of Lattice C programs (see Section 1.4) allows the full set of memory allocation functions, although care must be taken to provide sufficient space for the stack, as Section 1.5.3 warns. The basic program entry and exit functions are described in Section 1.5.4, and some special functions unique to the MS-DOS implementation are presented in Section 1.5.5. As additional functions will probably be provided as the compiler evolves, the programmer should check the addendum for the current version of the compiler.

1.5.1 File I/O

File names are specified according to the following format:

d:filename.ext

where "d:" is an optional drive 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. The
file name is specified without trailing blanks, if less than 8
characters, and 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 file names use
upper case letters only. Only those characters which are legal
for file names under MS-DOS are acceptable; consult the MS-DOS
documentation for details. Certain names are recognized as
devices rather than files; see the next section.

When a file is opened using "open" or "creat", a file access
block is allocated using "getmem", and library functions which
process read and write calls use this block to transfer data
between the file and the caller's area. The file access block
contains information such as the current file position and the
logical end of file position; it also includes a 128-byte buffer
which is used for MS-DOS read and write functions on the file.
Because the relative block number is kept as a 16-bit unsigned
integer, the maximum size of a file which can be accessed using
the "read" and "write" functions in this implementation is 65535
X 128, or about 8 megabytes. Note that the file I/O functions
maintain an exact end of file, even though portions of the file
are accessed in 128-byte blocks. When data is written to the
file, it is copied into the block buffer and not actually written
to disk until the buffer is full, the file position changed to
another block, or the file is closed. The memory used for the
file access block is released (via "rlsmem") when the file is
closed.

Note that all of the standard I/O functions ultimately call
"open", "creat", "read", and "write", so the above description
applies to "printf", "scanf", "putchar", "getchar", and all the
other upper level functions (if used for file I/O). Programs
with open files cannot use the "rstmem" or "rbrk" functions (see
Section 3.1) because the file I/O system allocates the file
access block from the same memory pool. This restriction does
not apply to open files which are actually devices (see next
section), because a file access block is not allocated for device
I/O. Note that the level 2 functions are subject to a separate
but similar restriction 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 16 open
files, including devices, and including the three (stdin, stdout,
stderr) which are automatically opened for the "main" program.

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. 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). On "write" calls, the count returned may be greater than the number of bytes specified in the count argument (because of CR insertions). Note that on read operations translation is performed only if the CR is immediately followed by an LF character; isolated carriage returns are not affected. Similarly, on write operations translation is performed only if the LF is NOT preceded by a CR.

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, since otherwise data bytes which happen to take on CR and LF values will be translated incorrectly. Since the high-level I/O routines call "open" or "creat" themselves, these routines cannot be used on files in the untranslated mode; they always operate in text mode.

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 an exact end of file in its directory, this mode can be and is implemented in this version.

1.5.2 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, with or without a trailing ":".   The
following table lists the devices and the corresponding BDOS
functions used for read and write operations, in translated and
untranslated modes.

| Device | Translated Mode | | Untranslated Mode | |
Name	Read FN	Write FN	Read FN	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
write 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) on output 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 1.5.5
for details.

     If one of these devices is opened for access using "fopen",
output is normally buffered, which means that no data is actually
sent to the device until 512 bytes have accumulated or the file
is closed. The buffer can be flushed using "fflush" or the file
can be changed to the line-buffered mode using "setnbf"; see
Section 3.2.2 for more information.

1.5.3 Memory Allocation

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

1. The reset functions "rstmem" and "rbrk" cannot be used if any of the standard 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. (They also cannot be used if either files or devices are open through the level 2 I/O functions; see Section 3.2.) A file may be closed, then re-opened after the reset function is called; however, any file descriptors or 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. Programs 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 1.1.4) or by defining an external "int" location called "_stack" and initializing it with a desired value. For example, the statement

        int _stack = 10000;

will provide for 10000 bytes of stack space. (Note: in order to qualify as an external definition, this statement must appear OUTSIDE the body of any function defined in the same module.) The default value for "_stack" (supplied from the library) is 2048. See Section 1.4 for information about the structure of programs.

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

            extern char *_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 1.4).

            extern char *_top;

contains the offset of the top of the stack, either X'FFF0' or whatever was determined to be the highest usable offset. 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.

1.5.4   Program Entry/Exit

The C.OBJ module 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 LC.LIB analyzes the command line for all of the elements
described in Section 1.1.4, and then passes the command-line
arguments to "main". If the stack override and file specifier
features are not needed, the following function may be used
instead.   Note that the function must be compiled and the
resulting object file included as one of the .OBJ files named on
the LINK command.

```
#include "STDIO.H"
#include "CTYPE.H"
#define MAXARG 32          /* maximum command line arguments */

_main(line)
char *line;
{
static int argc = 0;
static char *argv[MAXARG];

while (isspace(*line)) line++;      /* find program name */
while (*line != ´\0´)
    {                       /* get command line parameters */
    if (argc == MAXARG) break;
    argv[argc++] = line;
    while (*line != ´\0´ && isspace(*line) == 0) line++;
    if (*line != ´\0´) *line++ = ´\0´;
    while (isspace(*line)) line++;
    }
main(argc,argv);                    /* call main function */
exit(0);
}
```

The program exit functions "exit" and "_exit", described in
Section 3.3, are implemented under MS-DOS but the error code and
error message arguments are both ignored.

1.5.5   Special Functions

The functions discussed in this section provide serial I/O
capabilities at various levels. At the lowest level, the
function

```
        v = inp(p);
        int v;
        int p;
```

returns the 8-bit value "v" (expanded to 16 bits by padding with zero) from input port "p", while the function

        outp(p,v);

sends the 8-bit value "v" to output port "p". These functions perform the equivalent of the assembly language instructions

        IN   AL,P
        OUT   P,AL

The functions can be used to perform I/O directly from C.

    Access to the BDOS function entries of MS-DOS is provided by

        iret = bdos(fn, dx);
        int iret;   value returned in AL by BDOS function
                         (expanded to 16 bits by zero padding)
        int fn;   the BDOS function number
        int dx;   (optional) value to be placed in DX

Obviously, not all of the BDOS functions can be called with this interface; still, a sizable number of them are accessible, including all of the single character I/O functions. The "bdos" function is used to implement the special function

        iret = kbhit();
        int iret;   0 if a character was typed,
                         non-zero otherwise

which returns a value indicating whether or not a character has been typed at the user's console.

    Some of the most frequent I/O requests on any system which supports C are to the user's terminal, where it is often necessary to perform character I/O on a single-character basis. Two library functions provide this capability.

        c = getch();
        int c;

returns the ("int"-expanded) character from the console. The character is NOT echoed or checked for the program interrupt character (control-C or control-BREAK).

        putch(c);

sends the specified character directly to the console. Program interrupt is also NOT checked for by this function.

    Since the standard I/O functions are buffered even when "stdin" and "stdout" are the user's console, mixing "putch" calls with "printf" or "puts" calls can cause definite problems (similar problems occur on input). These problems can be avoided by using the direct console I/O functions described in Section

3.2.4, along with the special header file "CONIO.H". Note that
the "putchar" and "getchar" functions used in these modules are
the same as the "putch" and "getch" functions described above;
they do not echo characters received on input or check for the
program interrupt character. If this presents a problem, the
user can define local versions of "putch" and "getch" (inside one
of the user's modules) which send and receive characters using
some different mechanism (such as the BDOS functions 1 and 2,
which can be called with the "bdos" function described above).

     Two special console I/O functions are provided for input and
output of text strings. The function

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

uses the BDOS function 10 to get an input string. The first byte
(character) of "s" must be initialized by the caller to contain
the number of bytes, minus two, in "s". The string pointer
returned is "s+2", which is the first byte of input data. The
carriage return (which the user at the console must type to
terminate the operation) is replaced by a null byte. Note that
"s+1" will contain the number of characters in the string.
Characters typed are echoed, and the full range of editing
capabilities (such as backspacing, etc.) are available to the
user.

     Text string output can be performed using

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

which uses the BDOS function 9 to write to the console. A
carriage return or linefeed is NOT appended; they must be
included in the string, if desired. This function locates the
terminating null byte, changes it to a '$' (0x24), then changes
it back to the null byte before returning. This points out the
function's two limitations: (1) the string to be printed cannot
itself contain a '$' and (2) the string to be output cannot
reside in read-only memory (ROM).

SECTION 2   Language Definition

The Lattice 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 portable C compiler using the
Kernighan and Ritchie text 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, the C reference manual is "amended" to
show more precisely how the language differs from the standard.

2.1 Summary of Differences

    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 run
afoul of 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:

o   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.

o   Pre-processor macro substitution 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.

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

o   Identically written string constants refer to the same
     static storage locations, 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.

o   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.

o   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.

o   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.

o   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.

o   An array name may be preceded by the address-of operator (&)
     in this implementation; 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.

o   The maximum size of any declared object is the largest
     positive integer which can be represented as an "int". This
     implies a maximum size of 32767 for 16-bit "int" machines.

o   The maximum value of the constant expression defining the
     size of a single subscript of an array is 32767.

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 some 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.

2.2 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 in 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, but some
of the characteristics of the commands depend on how the compiler
is implemented. 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. The exception occurs with the #if command, which is processed differently depending on whether pre-processor functions are performed concurrently or in a separate phase. In the former case, the pre-processor module "borrows" the compiler's expression analyzer to evaluate the #if expression, so that #define substitutions are performed and the "sizeof" operator can be used. If evaluated during a separate pre-processor phase, #if expressions are more restricted; #define substitutions are not performed, and the "sizeof" operator cannot be used because the pre-processor phase has no knowledge of declared objects. To be safe, one should keep #if expressions as simple as possible; better still, avoid #if altogether and use #ifdef or #ifndef.

The #define command, as noted in Section 2.1, 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 like

        #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 immportant.

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 boundary.

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:

(1) 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 a 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.

(2) 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 a 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 a 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".

(3) 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 a 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.

(4) 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 at all in some cases.

Note also that if the "-x" option is used, the implicit storage class for items declared outside the body of a 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 clarification.

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 zeroes 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. As noted in Section 2.1, 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). 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 like

        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 in order for the compiler to 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, usually just called 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 lifetime of temporaries and their assignment to temporary storage locations are determined by a subroutine internal to the first phase of the compiler. This subroutine recognizes identically generated temporaries within a straight-line block of code and eliminates recomputation of equivalent results. Thus, common subexpressions 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 appear 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 subexpression 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 subexpression 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) no
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   Amendments to the C Reference Manual

The most precise definition of the C programming language generally available is 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 that 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 compile time option "-c" forces the compiler to process comments in the standard, non-nesting mode.

CRM 2.4.3 Character constants

Two extensions to character constants are provided. First, more than one character may be enclosed in single quotes; the result may be "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" (lower case 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.) The same "\x" convention described above can be employed in strings, where it is generally more useful (especially for those of us who have never understood how octal came to be used on 16-bit machines).

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 Lattice 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.

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 sc-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 sc-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. 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 in all cases contain "sizeof",
depending on the compiler implementation.

CRM 12.4 Line control

Although the file name 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 file name 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 will probably 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 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
supply that need.   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 benefit of presenting the functions
from a machine-independent viewpoint is that it helps the
programmer think of them in this light, as well.

When referring to the function descriptions presented in this
section, remember that the compiler assumes a function to return
an "int" value unless it is explicitly declared otherwise. Don't
forget to declare any functions which return other kinds of
values BEFORE you call them.

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 sufficiently
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 arbitrary order, but it is an
error to release something not obtained by 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 a lot of small allocations are requested, the level 2
functions will be considerably more efficient.

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 if you make lots of small 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

DESCRIPTION

    Allocates and clears (sets to all zeroes) 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

        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

    Check the return code; if -1, it could help you track down a
    coding problem. 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 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 -- get a memory block

SYNOPSIS

    p = getmem(nbytes);
    char *p;   block pointer
    unsigned nbytes;   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 -- release a memory block

SYNOPSIS

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

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 = 0 if successful
= -1 if supplied block not obtained by "getmem", or 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 -- allocate all available memory

SYNOPSIS

    ret = allmem();
    int ret;   return code

DESCRIPTION

    Uses the level 1 function "sbrk" to get all available
    memory and attach it to the memory pool used by "getmem".

RETURNS

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

NAME

    sizmem -- get memory pool size

SYNOPSIS

    words = sizmem();
    unsigned words;   number of words (sizeof(int))

DESCRIPTION

    Returns the number of unallocated words (i.e., number of
    units of size "sizeof(int)") in the memory pool used by
    "getmem". Note that "getmem" dynamically expands 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

    words = (unsigned) number of "int" objects in memory pool

CAUTIONS

    Note that the value returned is in words, not bytes.

NAME

    rstmem -- reset memory pool

SYNOPSIS

    rstmem();

DESCRIPTION

    Resets the memory pool used by "getmem" and "rlsmem" to its
    initial state. All previously allocated memory is released,
    and the maximum amount of memory is once again available.

CAUTIONS

    This function cannot be used if any files are open and being
    accessed using any of the level 2 I/O functions, because
    these functions use "getmem" to allocate buffers. On some
    systems, this restriction applies to the level 1 I/O
    functions as well; check the implementation section of the
    manual to see if this caution is valid for this system.
    Note that "sizmem" will return a value of zero after
    "rstmem" is called.

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. Do not confuse the "break
point" mentioned here 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 -- set memory break point

SYNOPSIS

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

DESCRIPTION

Allocates a block of memory of the requested size, if
possible. This function is the basic UNIX memory allocator.
The first time it is called, it will allocate the largest
available block of high memory. Then the requsted number of
bytes is lopped off the low end of the block for use by the
caller.

RETURNS

p = -1 if request cannot be fulfilled
= 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. The function 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 back 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. On some systems, the same restriction applies to
    use of level 1 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 this 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. Unlike the corresponding functions under
UNIX, these functions are buffered even when performing I/O to
and from the user´s console (although they are buffered on a line
basis, rather than the 512-byte block buffering used for disk
files). 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 allow 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 (usually 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. Note that a file pointer may be used to read a file or

to write a file, but it is not legal to perform both operations
on the same file.

    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 by 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; a buffer of that size is allocated (using "getmem")
when the file is opened. The buffering technique allows single
character I/O operations to be implemented efficiently as macros,
and reduces the number of actual I/O requests. Since output is
buffered, a file that is being written must be closed so that the
data in the buffer is actually written to the file. A different
buffering scheme is used for devices such as the user´s terminal.
The same buffer is allocated, but the read operation which fills
the buffer terminates on a newline, and the write operation which
flushes the buffer is initiated when a newline is received. This
scheme manages to reduce the I/O overhead while at the same time
performing the I/O in a more timely fashion. The line buffering
scheme is enabled using the "setnbf" function described in this
section, which simply sets the _IONBF flag for the file (note:
of the various flags defined in the standard I/O header file,
only this one is currently implemented). Line buffering is also

the mode set up for "stderr", and for "stdin" and "stdout" in the
event that they default to the user's terminal. See Section
3.2.3 for information about the direct, unbuffered interface to
the user's terminal.

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 file name must conform to local 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

    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".

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 specifed, 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
    as 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 your own files explicity. 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.
    Remember that for devices (such as the user's console) input
    is buffered on a line basis, that is, the read operation
    that fills the buffer does not terminate until a newline is
    received. See Section 3.2.3 if direct single character I/O
    to the console is needed.

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.
    Remember that output for devices (such as the user's
    console) is buffered on a line basis, that is, the write
    operation that flushes the buffer is not actually performed
    until a newline is sent. See Section 3.2.3 if direct single
    character I/O to the console is needed.

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 may be used in placed of the corresponding macros (getc
    and putc) in the event that a lot of calls are made, and the
    programmer is concerned about the memory used up in the
    macro expansions. The tradeoff is the usual one: the macro
    is more efficient timewise because it saves a function call,
    but the function is more efficient spacewise 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

    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 had better 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
    "fputs") 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

CAUTIONS

    Remember that output to a device (such as the user's
    console) is buffered on a line basis, that is, the write
    operation that flushes the buffer is not performed until a
    newline is sent. See Section 3.2.3 for an equivalent to
    "puts" that sends characters directly (without buffering) to
    the user's console.

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 (1) the "stdin" file ("scanf"); (2) the
    specified file ("fscanf"); or (3) the specified string
    ("sscanf"). The control string contains format specifiers
    and/or characters to be matched from the input; the list of
    pointer arguments specify where the results of the
    conversions are to go. Format specifiers are of the form

        %*nlX

    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" (the letter ell) is used to
    indicate a "long int" or "long float" (i.e., "double")
    result is desired; and (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 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.plX

    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" (letter ell)
    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", 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 had better be large enough to contain
    the resulting image. In all cases, make sure that the
    format specifiers 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

    The file position may be affected by text mode translation,
    since the translation may change the number of actual data
    bytes read or written.

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

    setnbf -- force line buffering for file

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 line buffering
    mode used for devices (including the user's console). In
    this mode, read operations to fill the buffer terminate when
    a newline is read, and write operations to flush the buffer
    are initiated whenever a newline is received.

CAUTIONS

    Although the line-buffered 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, but it is illegal to
seek to a position beyond the end of file. 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.

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. On some systems, the buffers used for the level 1
functions are allocated using "getmem", which restricts the use
of the memory allocation functions "rstmem" and "rbrk". Check
the implementation section of the manual to determine whether
this restriction applies, for information about the translated
and binary modes, and for the details of device I/O on this
particular system.

NAME

    open -- open a file

SYNOPSIS

    file = open(name, rwmode);
    int file;   file number or error code
    char *name;   file name
    int rwmode;   read/write mode, where 0=read, 1=write,
                           2=read/write, and bit 15 indicates the
                           desired mode (text=0, binary=1)

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 low order bits specify whether read or
    write operations (or both) are to be allowed, as follows:

        0 = read only access
        1 = write only access
        2 = read/write access

    If bit 15 (the 0x8000 bit) of the mode word is set, then all
    operations will be performed without text file translation
    (if such translation is normally performed for the system).
    If this bit is reset (the default mode used by the level 2
    functions), some translation of data may occur, the exact
    nature of which is system dependent. 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. "open" can be used only
    on existing files; use "creat" to access a new file.

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. If text mode translation occurs, the returned count may be greater than the supplied count due to the addition of characters during translation. The returned count is always the same as the number of characters by which the file position was advanced.

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, so don't
    forget to indicate a "long" constant 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 console I/O header file (called "conio.h" on most systems) which defines several of the level 2 functions in terms of the direct console functions, a feature which is most convenient for programs written for other C environments where I/O to the user's terminal is always unbuffered. The equivalencies defined by "conio.h" are

          getchar = getch
          putchar = putch
          gets = cgets
          puts = cputs
          scanf = cscanf
          printf = cprintf

The functions on the right side of the equals signs are described in this section.

A couple of 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, but 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. Check the implementation section of the manual for more information about console I/O.

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 systems may implement 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 a
     single level 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

    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. On some systems, the first byte of
"s" must be initialized to contain some kind of byte count
indicating the size of the buffer.

RETURNS

    p = pointer to string received, which may or may not
(depending on the system) include the terminating carriage
return

CAUTIONS

    Check the implementation section of the 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 string directly to the user's console. On some
    systems an additional character (usually CR, 0x0D) is
    appended.

CAUTIONS

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

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 "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[ ,message]);
int errcode;        exit error code
char *message;      exit message (optional)
```

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. The optional exit message may not be implemented on some systems.

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 [,message]);
    int errcode;   exit error code
    char *message;   exit message (optional)

DESCRIPTION

    Terminates execution of the current program immediately,
    without checking for open files. The arguments are the same
    as for "exit" (which calls "_exit" after checking the level
    2 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 allow quick testing of character
types; and several utility functions facilitate character string
handling. Almost none of these functions are system dependent.

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 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 "d" 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 0xbl, for
instance, will be classified the same as a character with the
value 0x31. Programs who wish 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. Don't forget to include "ctype.h" if you use any of
these macros; 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 digit, 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
iscsymf(c)	non-zero if valid first character for C identifier
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" indicates the function returns an "int" count
     "stp" indicates the function returns a character pointer
     "sts" indicates the 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 had better 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 had better 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 are 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 zeroes 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 zeroes 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

    Performs an anchored scan of the input string to convert a
    hexadecimal value into an integer. The scan terminates when
    a non-hex character is found. 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

    Performs an anchored scan of the input string to convert a
    decimal value into an integer. The scan terminates when a
    non-decimal character is found. Valid decimal characters
    are 0-9. The first character may be ´+´ or ´-´.

RETURNS

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

CAUTIONS

    No check for overflow is made during 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.

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", it 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 matching 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

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) disk.

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) disk.

File name missing

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

Intermediate file error

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

Invalid command line option

    An invalid command line option (beginning with a "-") was
    specified on either the LC1 or the LC2 command. See
    Sections 1.1.1 and 1.1.2 for the valid command line options.
    The option is ignored, but the compilation is not otherwise
    affected. In other words, this error is not fatal.

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 1.3.3 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 (note that '$' is valid for ordinary identifiers but
     not for pre-processor symbols).

2   The end of an input file was encountered when the compiler
     expected more data. 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 a #include command was not found
     on the currently logged-in disk.

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). May occur if control characters, or
     other garbage is detected in the source file. 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. 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 current version of the compiler supports #include
     nesting to a maximum depth of 4.

8   An invalid arithmetic or pointer conversion was specified.
     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). 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). Will occur if the closing " is
     omitted in specifying the string.

12   The expression preceding the "." or "->" structure reference
     operator was not recognizable by the compiler as a structure
     or pointer to a structure. May occur 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 "." or "->" operator.

14   The indicated identifier was not a member of the structure
     or union to which the "." or "->" referred. 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. 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. 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.
     May occur if an operand was omitted for a binary operation.

19   The numbers of opening and closing parentheses in an
      expression were not equal. May occur if a macro was poorly
      specified or improperly used, but is generally due to the
      obvious error.

20   An expression which did not evaluate to a constant was
      encountered in a context which required a constant result.
      May occur if one of the operators not valid for constant
      expressions was present (see Kernighan and Ritchie, Appendix
      A, p. 211).

21   An identifier declared as a structure, union, or function
      was encountered in an expression without being properly
      qualified (by a structure reference or function call
      operator).

22   (non-fatal warning) An identifier declared as a structure or
      union appeared as a function argument without the preceding
      & operator. Expression evaluation continues with the &
      assumed (i.e., a pointer to the aggregate is generated).

23   The conditional operator was used erroneously. 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. 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. 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. 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. 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   (non-fatal warning) In an assignment statement defining a
      value for a pointer variable, the expression on the right
      side of the = operator did not evaluate to a pointer of the
      exact same type as the pointer variable being assigned,
      i.e., it did not point to the same type of object.   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.
      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 evalute to a
      constant.   May require some experimentation to determine the
      exact cause of the error.

35   During processing of an initializer list or a structure or
      union member declaration list, the compiler expected a
      closing right brace but did not find it.   May 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 numbers of opening
      left braces ({) and closing right braces (}) were not equal.
      May occur if the compiler got "out of phase" due to a
      previous error.   ,

39   One of the C language reserved words appeared in an invalid
      context (e.g., as a variable name).   See Kernighan and
      Ritchie for a list of the reserved words (p. 180).   Note
      that "entry" is reserved although it is not implemented in
      the compiler.

40   A "break" statement was detected that was not within the
      scope of a "while", "do", "for", or "switch" statement.   May
      occur due to an error in a preceding statement.

41   A "case" prefix was encountered outside the scope of a
     "switch" statement. 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. May occur due to
     an error in a preceding statement.

45   A "default" prefix was encountered outside the scope of a
     "switch" statement. 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. 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. 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   This error is generated when the expression following the
     "return" keyword could not be legally converted to the type
     of the value returned by the function. May be generated if
     that 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 (:). 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 (;). May 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 not found (as in a "while" or
      "for" loop). May 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. May 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. May occur
      if a preceding error caused the compiler to "get out of
      phase" with respect to the 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.
      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. May be
      generated if the "*" is forgotten on a structure pointer
      declaration, or if (due to some intertwining of structure
      definitions) the structure actually contains an instance of
      itself.

70   A function's formal parameter was declared illegally, that
      is, it was declared as a structure, union, or function. The
      compiler does not automatically convert such references to
      pointers, which is what is usually intended.

71   Reserved for expansion. Check the latest addendum to see if
      this error has been newly defined; otherwise, treat it as a
      compiler error and report it according to the directions in
      Appendix B.

72   An external item has been declared with attributes which
      conflict with a previous declaration.   May occur if a
      function was used earlier, as an implicit "int" function,
      and was then declared as returning some other kind of value.
      Functions which return a value other than "int" must be
      declared before they are used so that the compiler is aware
      of the type of the function value.

73   In processing the declaration of objects, the compiler
      expected to find another line of declarations but did not in
      fact find one. This error may be generated if a preceding
      error caused the compiler to "get out of phase" with respect
      to declarations.

74   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. 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. 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. 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 Lattice 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.

APPENDIX B   Compiler Errors

This appendix describes the procedure to be used for reporti
compiler errors.   These are errors that result not from t
user's incorrect specifications but from the compiler itse
failing to operate properly.   There are four general kinds
errors which can occur:

    1. The compiler generates an error message for a sour
    module which is actually correct.

    2. The compiler fails to generate an error message for
    incorrect source module.

    3. The compiler detects an internal error condition a
    generates an error message of the form

        CXERR: nn

    where "nn" is an internal error number.

    4. The compiler dies mysteriously (crashes) while compili
    a source module.

    5. The compiler generates incorrect code for a corre
    source module.

The last type of error is of course the most difficult
determine and the most vexing for the programmer, who has
indication that anything is wrong until something inexplicab
doesn't work and only concludes that the compiler is at fau
after a long and painstaking study of his or her own code.

    We would like to know about and repair any compiler erro
as soon as possible, so please help us out by reporting a
problems promptly.   The difficulties you suffer may be spared t
next programmer if you do so.   In order to maintain a mo
precise record of the bugs that are discovered, we would like a
problems to be reported in writing.   You can send the probl
reports to Lifeboat, but the problem will be attended to mo
quickly if you send it to Lattice directly at this address:

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

    In all cases, include the following items of information
your package:

    1. A listing of the source module for which the err
occurred.   Don't forget to include listings of any #include fil
used (and watch out for #include file nesting; don't forget t
inner files as well).   Supplying the source on disk will possib

save us some typing, but there's no guarantee we'll be able to
read it (unless it's IBM PC format), and if you don't want to
spare a disk just send listings.

2. The revision of the compiler and when it was purchased.

3. Your name and address and, if you're willing to talk
about the problem with a technical person, a phone number with
the times you will be available.

4. A description, brief or lengthy as it suits you, of the
problem, along with any other information you think may be
helpful. Obviously, errors of type 3 (see above) don't need
anything more than a terse "Causes CXERR 23." If you've
investigated the problem yourself to some extent, let us know
what you found.

Once you have determined that there is a definite compiler
problem, put together a problem report and ship it off to us.
We'll try to get to it as soon as possible, and we are attempting
to institute a liberal update policy, especially for those who
report bugs. Meanwhile, try coding 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 we use them for
our own development work, so we're well motivated to fix any
problems you find.

APPENDIX C Conversion of CP/M Programs

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

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

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

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

          char *cfunct();

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

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

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

     Be careful not to compile more than one of the modules
the program without using the -x option; otherwise, the lin
will inform you that multiple definitions of the external it

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 Sections 1.5.5 and 3.2.3. In general, it
is best to review all of the functions supplied in both libaries
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. Still,
judicious use of #defines may eliminate some problems.

INDEX

Note: index references containing punctuation (such as functi
names, which are enclosed in double quotes, as in "printf") a
listed at the beginning of the references for each letter.
sure to check the entire list for each letter when searching f
a particular reference.

Lattice 8086/8088 C Compiler

MANUAL SUPPLEMENT FOR VERSION 2.00 OF COMPILER

1.0 DIFFERENCES FROM PREVIOUS VERSIONS

The following list summarizes the most important differences between Version 2.00 and previous versions for users who are upgrading their compiler. For complete information about the new features, refer to the latest manual and supplement.

1.1 Compiler Differences

The meaning of some previously defined compile time flags has been changed, and several new options have been added.

1.1.1 Effect of the -a flag

The effect of the -a flag has been extended so that it forces all assignment statements (that is, the actual store operation) to be performed before the execution of the next statement. 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.

1.1.2 Alignment of data elements

The alignment of storage for arithmetic objects has been changed. Now, the only data elements which force alignment to a word offset are pointers, structures, and unions. (In previous versions of the compiler, all objects except simple "char" variables were word-aligned.) The -b flag still has the effect of dropping alignment requirements for all objects.

1.1.3 Extensions to -i flag

The -i option has been generalized to accept a prefix which is to be prepended to file names from ∤include statements; up to 4 -i options may be specified. Note that the current directory is always searched first before the -i options are checked.   ,

1.1.4 Optional long identifiers flag

A special -n option has been added which, if used, forces the compiler to retain up to 39 characters for all identifiers (including pre-processor ∤define symbols).

1.1.5 Stack overflow checking

The compiler now, by default, generates code at the beginning of
each function to check for stack overflow. The code for stack
overflow detection can be eliminated by compiling your source
module with the new -v option on LC2. Library functions are
supplied with stack overflow detection included.

1.1.6 Expanded memory addressing

The -m and -s flags on LC1 are new compile time options used for
the new expanded memory addressing feature of the compiler. Four
different memory "models" are supported, allowing a range of
addressing capabilities for compiled programs. Note that a sin-
gle program must be compiled and linked according to one and only
one of the memory models, that is, functions compiled according
to different memory models may not be combined in a single pro-
gram.

1.1.7 Code group and segment name override

Two new flags (-g and -s) on LC2 allow the user to specify code
group and segment names in the generated object file.

1.2 Run-time and Library Differences

In addition to new versions of the library to support the new
memory addressing capabilities, the implementation of many of the
library routines has been improved, resulting in some differences
in their operation.

1.2.1 Processing of =, <, and > specifiers on command line

The special command line specifiers "=", "<", and ">" are now
processed by C.OBJ instead of "_main", and must appear before all
other command line arguments following the program name.

1.2.2 New version of stdio.h and level 2 I/O functions

The level 1 and level 2 I/O functions have been upgraded but are
compatible with the old functions; however, any program using
level 2 I/O (i.e., any that #included "stdio.h") must be
recompiled because the stdio.h definitions have changed.

1.2.3 Extensions to "open" and "fopen" functions

The level 1 I/O "open" function has been extended to support a
number of new flags, defined in the new header file "fcntl.h".
The level 2 I/O "fopen" function has been extended to accept a
"+" after the mode character to indicate that both reads and
writes are allowed on the file. To switch from one mode to the
other, you must execute "fseek" or "rewind" on the file pointer.

1.2.4 Implementation of level 2 I/O buffering

The level 2 I/O functions now perform I/O for all devices (inclu-
ding the console) in true unbuffered fashion. The old "line
buffered" mode supported by the previous version has been
scrapped, so that "printf" to the console sends its characters
immediately, whether or not a newline is sent. Similarly, input
from devices is also normally unbuffered, but buffered console
input is supported and processed using the BDOS function. The
new version of "_main" sets up "stdin" to be buffered, so back-
space and line cancel features will now work on reads from
"stdin" when assigned to the console.

1.2.5 Implementation of level 1 I/O buffering

The level 1 I/O functions (open, read, write, lseek, close) do
not acquire buffers via the level 2 memory allocator, as they did
in previous versions. This means that you no longer have to
worry about messing up files when you do a "rstmem", and that the
"_block" external is no longer supported. Under MS-DOS, buffer-
ing is now performed by the operating system itself, resulting in
improved performance for large read/write data transfers.

1.2.6 Support for MS-DOS Version 2.0

The compiler and library for MS-DOS Version 2.0 supports the new
path names available under that operating system. Your programs
should not require any changes unless they are sensitive to the
file name format. I/O redirection now works properly under DOS
Version 2 for programs compiled using this new library.

1.2.7 Extensions to level 2 memory allocation

The memory allocation functions have been extended to support the
new extended addressing capabilities of the compiler. In
particular, the "sizmem" function has been changed to return a
"long" integer, and two new functions ("getml" and "lsbrk") have
been added which support a "long" integer requested block size.
The other functions are compatible with the old versions.

1.2.8 New utility functions

Several new utility functions have been added to the library
which allow access to all of the features of the 8086/8088
processors, including (1) software interrupts (useful for making
direct ROM BIOS calls); (2) BDOS functions; (3) access to segment
register contents; (4) inter-segment memory transfers; and (5)
"peek" and "poke" functions for examining and setting any memory
locations.

3

2.0 MANUAL ERRATA

Please note the following errors in the original version of the
manual.

2.1 Missing ENDS statement

On page 1-33 of the manual, the last four statements of the
assembly language example should read as follows:

```
        XCFIND ENDP
        XCMAKE ENDP
        PROG   ENDS
               END
```

Inclusion of the ENDS statement is critical; if omitted, the
linker will produce an invalid EXE file when the module is
linked. Note that the assembly language interface is slightly
different for each of the memory models; see below.

2.2 Call "_exit", not "exit"

On page 1-38 of the manual, a short version of "_main" is presen-
ted; however, the final statement before the closing brace should
read:

```
        _exit(0);
```

If "exit" is called, the level 2 I/O functions are included in
the program. Note that the correct version of this function has
now been supplied as TINYMAIN.C.

2.3 "kbhit" function described incorrectly

Page 1-39 of the manual describes a function called "kbhit",
which returns a status indicating whether a character has been
typed at the user's console. Please note that the action of the
function as described in the manual is exactly opposite to its
actual characteristics: it returns zero if a character has NOT
been typed at the keyboard, and non-zero if a character is wai-
ting input.

3.0 MAJOR NEW FEATURES

The most important new features present in Version 2.00 of the
compiler are (1) expanded memory addressing capabilities; (2)
run-time stack overflow checking; and (3) support for MS-DOS
Version 2.0.

3.1 Expanded memory addressing

The compiler now supports program and data spaces greater than
64K bytes. Four different "memory models" are defined, 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. None of this requires you to
change any of your C code if you have played by the rules. The
main pitfall is any assumption that a pointer will fit into an
integer, since integers are still only two bytes under all mo-
dels.

In all of the models, a single DATA segment is used to contain
all statically allocated data within the program, thus restric-
ting the combined total of static data to 64K bytes or less.
Similarly, the stack segment where "auto" data elements are
allocated can never be greater than 64K. In the D and L models,
however, dynamic memory can be allocated without restriction, and
pointers can be used to access any location in memory.

3.1.1 Choosing the memory model

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

(1) Will the combined size of the functions in your program be
    greater than 64K bytes? If not, select one of the models
    that uses NEAR calls (the S or D models), which are faster
    and require less code. Otherwise, you must select a model
    that supports FAR calls (the P or L models), unless your
    application is suitable for program section overlays, which
    can be created using the Phoenix Software Associates PLINK86.

(2) Does your application require more than 64K bytes of data

storage? If not, select one of the models that uses 2-byte
data pointers (the S or P models), because pointer operations
are performed much more efficiently in these models. If you
simply need access to specific memory locations beyond the
program's 64K address space, you can probably use the new
library functions "peek" and "poke" and retain the efficient
2-byte pointers. Otherwise, if data storage in excess of 64K
bytes is a must, you must select a model that uses the 4-byte
data pointers (the D or L models) and pay the price of
somewhat less efficient code.

3.1.2 Compiling for the memory models

Generation of code for the various models is controlled by a new
compile time option specified on the first phase (LC1) of the
compiler. The -m option must be followed immediately (no spaces,
please) 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:	LC1 filename	(no flags)
	LC1 filename -ms	
	LC1 filename -m0	
P model:	LC1 filename -mp	
	LC1 filename -ml	
D model:	LC1 filename -md	
	LC1 filename -m2	
L model:	LC1 filename -ml	
	LC1 filename -m3	

3.1.3 Linking programs

When using the various memory models, you must be careful 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 you to examine the LINK map
and determine if you have erroneously mixed code for different
models. 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 com-
piled module containing functions).

3.1.4 Run-time program structure

Two different memory layouts are now used. For the small data
case, the layout remains exactly as described in the manual. For
the large data cases, the stack resides immediately above the
static data area, and the free memory pool (allocated by "sbrk")
is above the stack. DS is the base of the static data, SS is the
base of the stack, and ES is undefined. The public symbol
"_base" still contains the base of the stack relative to DS (it
specifies the number of static data bytes). The symbol "_top"
contains the top of the stack relative to SS (it contains the
number of bytes allocated for the stack).

It is important that your programs reserve sufficient stack space
in the large models, since there is no free space to absorb stack
overflows. (Stack overflow checking is now a default option; see
below.) If you define a public unsigned integer called "_stack",
that value will be used as the number of bytes in the stack.
Note that the stack can be as large as 64K bytes in the D and L
models.

3.1.5 Code generation for pointer operations

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 ad-
dressed 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 there-
fore much more efficient in the 2-byte memory models, and if your
application can live with a 64K data space, use the S or the P
model.

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. The compiler
    also performs pointer arithmetic for constant offsets which
    fit in 16 bits by performing the operation on the offset
    portion of the pointer, and then adjusting the base portion
    by 1000H if overflow occurs.

(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.

3.1.6 The -s option for four-byte pointers

While the above rules provide complete generality in the 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.

We expect that most functions can be safely compiled with the -s option, with the result of improved code generation quality. In fact, all of the library functions written in C supplied in our 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.

3.1.7 Assembly language interface

While C functions can be adapted for a memory model simply by
changing a compile-time switch, assembly language functions pre-
sent a bigger problem, because you must change your code to
process long pointers or use FAR linkages or both. The pointers
can be handled in the following way:

```
    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
```

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).

The use of FAR linkages requires the PROC statements to be
changed. Note also that the position of any EXTRN statements for
external functions is critical. For FAR linkages, they must
appear before the SEGMENT definition for the code segment, while
for NEAR linkages, they must appear after the code segment defi-
nition. These rules can be summarized in the following example:

```
    S/D model:        define code segment
                      EXTRN      XYZ:NEAR, ...
              ABC     PROC       NEAR

    P/L model:        EXTRN      XYZ:FAR, ...
                      define code segment
              ABC     PROC       FAR
```

To deal with this messy problem, we've put conditional statements
into our assembly language functions. This keeps a single source
file for each function, and it is then re-assembled with a dif-
ferent macro library for each model. Our libraries (SM8086.MAC,
DM8086.MAC, PM8086.MAC, and LM8086.MAC) are included and should
be self-explanatory. We put the following statement at the
beginning of each assembly language module:

                  INCLUDE DOS.MAC

Then, we merely copy the appropriate library to DOS.MAC before
assembling for a particular model. The source files for C.ASM
and IO.ASM, which are supplied, illustrate the use of these
macros.

3.1.8 Special cautions on using the D and L 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 you may

think that your code does not depend in any way on this fact, you
may find that the assumption has crept into your implementation
without your being aware of it. Be alert for problems that might
relate to this. Here are three other important cautions you
should take note of:

(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". If you compile code for the D or
    L models, you MUST change NULL to be 0L so that the null
    pointer value supplied to functions is the same size as the
    pointer argument. If you fail to do this, the called
    function will not correctly address its parameters, and all
    sorts of chaos will result.

(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 3.1.5 and 3.1.6). 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! Avoid all of these problems
    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 5.3 of this supplement).

(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; we have
    already heard one horror story about a user who destroyed the
    File Allocation Table for his hard disk by using a garbage
    pointer to set up a structure. All we can do is caution you
    to be extremely careful. Our advice to beginning C users is
    to stick with the S and P models, where uninitialized poin-
    ters are much less likely to access critical locations.

3.1.9 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. A quick check
of the manual, however, shows that the allocation functions only
accept a size argument of "unsigned" type, which has a maximum
value of 64K. We have overcome this shortcoming by adding two
new allocation functions which accept a size argument which is a
"long" integer. These functions are described in more detail in
section 5.3 below, but an example here will illustrate the use of
one of them.

Suppose that we must allocate an array of 10,000 double precision values; 80,000 bytes of storage will be required for such an array. First, declare a pointer 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, declare the memory allocation function:

        char *getml();

Note that you MUST declare the memory allocation function to return a pointer; otherwise, the compiler will assume it returns an "int" and the cast operation shown below will not work correctly. The only difference between "getml" and "getmem" is that "getml" gets an argument which is a "long" integer. The array is then allocated by the expression:

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

Note the "L" specifier on the constant. The size could also be specified as "(10000L * sizeof(double))". (Special note: if you compute the size argument for "getml" 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 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 we had allocated a "char" array, "long" integers would be needed to subscript an array of this size. One final note: since we are addressing an object with a size greater than 64K, the -s option cannot be used.

3.2 Stack overflow checking

The compiler now, 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 immedi-
ately.

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 like

        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 like

        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. You will need at least as
much storage as the largest amount of "auto" storage in any of
the functions included in the program (i.e., if one of your
functions has an "auto" array of 4000 bytes, you will need at
least that much stack space, 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. Our 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 com-
piling your source file with the -v option on LC2.   Library
functions are supplied with stack overflow detection included.

3.3 Support for MS-DOS Version 2

If you specified DOS Version 2 when you ordered the compiler, you
received a version of the compiler and library set up for that
operating system. The compiler and the programs generated will
execute only under DOS Version 2.

3.3.1 Compiler support for MS-DOS Version 2

The compiler now recognizes the full Version 2 path names for all file names. 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"

The -i option has been extended to support an alternative form of the above, where the command line might specify, for example:

        LC1 xyzfile -ib:\headers\

and the #include statement could then read

        #include "stdio.h"

Note that the trailing backslash must be supplied on the prefix attached to the -i flag; it is not automatically supplied by the compiler.

A maximum of 4 -i prefixes may be specified on the LC1 command. When an #include statement is encountered naming a file that is not already prefixed by a drive or directory specification, the current directory is searched first for the file; if not found, each -i prefix (in the same order specified on the LC1 command) is prepended to the #include file name and searched for, in turn, until the file is located. An error message is produced if none of the searches is successful. No spaces, please, between the -i and the desired text to be used for prepending.

The -o option for both LC1 and LC2 has been similarly extended, allowing the output file to be written directly to another direc-tory, if desired.

Special versions of the utility programs FXU and OMD for MSDOS Version 2 have also been provided.

3.3.2 Library support for MS-DOS Version 2

The main change in the library for MS-DOS Version 2 is that the I/O functions recognize path names because they use the new UNIX-like file interface provided in Version 2. Your programs should not require any changes unless they are sensitive to the file name format. The I/O redirection mechanism has been adjusted to work properly for Version 2 programs. The "exit" and "_exit" functions pass the exit code back to the operating system, and the value can be tested in a batch file command such as

        if errorlevel "value"

4.0 COMPILER AND RUN-TIME CHANGES

The following items are new or substantially changed, as compared
to the descriptions in the manual. Some minor changes to the
language accepted by the compiler are not remarked upon if they
were not explicitly mentioned in the manual (or unless they are
incompatible with previous versions).

4.1 Compile time options for LC1

Here are the new or changed compile time options on the first
phase of the compiler.

-a   Same as in previous version, but 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
            statemen is critical to subsequent actions; and (3)
            memory-mapped I/O assignments, where values must be
            stored repeatedly in the same "memory" location.

-d   Causes debugging information to be included in the quad
            file. Specifically, line separator quads are inter-
            spersed with the normal quads. This allows the second
            phase to collect information relating input line num-
            bers to program section offsets. If this option is
            used, the object file produced will contain line num-
            ber/offset records, and can be processed by the object
            module disassembler to produce an intermixed source
            code and machine code listing (see 6.1 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 file name with the string "prefix",
            unless the file name in the #include statement is
            already prefixed by a drive or directory identifier.
            If "prefix" is a single character, a colon is added;
            thus, -ia causes prepending with "a:". Up to 4 dif-
            ferent -i strings may be specified. Note that un-
            prefixed #include file names are searched for first in
            the current directory, and then by prepending with the
            prefixes specified in -i options, in the same left-to-
            right order as they were supplied on the command line.

-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).

-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 (.Q) file name is to be
             formed by prepending the input file name (the .C file
             which is being compiled) with "prefix". If "prefix" is
             a single character, a colon is added; thus, -ob causes
             prepending with "b:". Any drive or directory prefixes
             attached to the input file name are discarded before
             the prepending is performed.

-s   Changes the way code is generated for four-byte poin-
             ters in the D and L models; see section 3.1.6.

4.2 Compile time options for LC2

Here are the new or changed compile-time options on the second
phase of the compiler.

-f   This option, described in the manual, has not yet been
             implemented. 8087 support will be provided in a future
             release of the compiler.

-ggroup   Specifies that the name "group" is to be used for the
             code group in the .OBJ module. "group" must be 15 or
             fewer characters in length, and must be adjacent to the
             "-g" (no intervening blanks).

-oprefix   Specifies that the output (.OBJ) file name is to be
             formed by prepending the quad file name (the .Q file
             which is being processed) with "prefix". If "prefix"
             is a single character, a colon is added; thus, -ob
             causes prepending with "b:". Any drive or directory
             prefixes attached to the quad file-name are discarded
             before the prepending is performed.

-ssegment   Specifies that the name "segment" is 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.

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.

4.3 Alignment of data elements

The alignment of storage for arithmetic objects has been changed.
Now, the only data elements which force alignment to a word
offset are pointers, structures, and unions. The -b flag still
has the effect of dropping alignment requirements for all ob-
jects.

4.4 Language definition: arbitrary limitations

This section attempts to clarify some of the limitations of the
compiler which are omitted or not clearly defined in the manual.
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:

o   The maximum size, in bytes, of any declared object is the
     largest positive integer which can be represented as an
     "int". This implies 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.

o   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").

o   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.

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

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

o   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.

4.5 Change in the processing of the #if command

The processing of the #if command has been modified internally,
with two important consequences for programmers. First, as
should be noted in the list of differences from the standard
language, "sizeof" cannot be used in #if expressions, and the
expression must appear 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 a #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.

4.6 New error/warning messages

Two new numbered messages are now generated by the compiler; here
is an explanation of them.

82   The object declared caused the total storage for its
            storage class to exceed 32767 bytes, the maximum legal
            value.

83   This non-fatal warning complains of an indirect refer-
            ence (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 you
            refer to an element beyond the end of an array.

4.7 Exit error code

The MS-DOS Version 2 compiler returns an exit code of zero if no
errors were detected, and a code of one otherwise. This allows
the use of "if" expressions in batch files, such as:

            LC1 %1
            if errorlevel 1 goto errs

5.0 CHANGES TO LIBRARY FUNCTIONS

The following features of the standard library functions provided
with the compiler are new or substantially changed in Version
2.00.

5.1 Program entry/exit functions

Three changes are important in the operation of the program
startup and termination features of the new library.

5.1.1 Processing of =, <, and > options on command line

The special command line specifiers "=", "<", and ">" are now
processed by C.OBJ instead of "_main", and must appear before all
other command line arguments following the program name.

5.1.2 Source for "_main" now supplied

The standard library version of "_main" has been supplied as
MAIN.C, while a smaller version which does not open any buffered
files has been supplied as TINYMAIN.C. Users may modify these

modules to produce their own versions of "_main". Please note, however, the following cautions:

(1) The library function "printf" sends its output to the pre-defined file pointer "stdout", which is normally opened by "_main". If you remove the code that performs this function, don't be surprised when "printf" calls 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 your intention is to avoid including the level 2 I/O functions in the linked program, don't call the library function "exit", because it closes all buffered output files before terminating execution. This will cause the level 2 functions to be included anyway. Call "_exit" instead.

5.1.3 Exit functions under MS-DOS Version 2

The functions "exit" and "_exit" pass the error exit code back to the operating system, where it can be tested in a batch file using a command like:

    if errorlevel 1 goto error

This feature is supported only under DOS Version 2.

5.2 I/O library

Extensive revisions to the I/O library have extended its capabilities and retained compatibility with the previous version. The implementation of some features has changed internally, with effects that are noted in this section.

5.2.1 Upgrades to level 1 and level 2 I/O functions

This version upgrades the level 1 and level 2 functions to be compatible with the latest UNIX releases and with the UNIFORUM draft standard. You will not need to make any program changes, since the new functions are still compatible with the old. However, any programs using level 2 I/O (i.e., any that included "stdio.h") must be recompiled because the stdio.h definitions have changed.

Here is a summary of the extensions and changes:

(1) The "open" function currently accepts a second argument of 0, 1, or 2 for read, write, or update mode, respectively. Now you can include the header file "fcntl.h" which defines the following codes for that second argument:

        O_RDONLY   Same as code 0
        O_WRONLY   Same as code 1
        O_RDWR   Same as code 2

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   Flunk the create if file exists
O_APPEND   Seek to end-of-file before each write
O_RAW   Use untranslated I/O (Lattice addition)

A new public symbol called "_iomode" has been added to
preset the translation mode. Normally, _iomode is 0 and
translated mode is used unless O_RAW is specified. If you
change _iomode to 0x8000, then untranslated mode is used
unless O_RAW is specified. In other words, O_RAW toggles
the meaning of _iomode.

(2)   The "fopen" function recognizes a + after the mode character
      to indicate that both reads and writes are allowed. In
      order to switch from one mode to the other, you must execute
      an "fseek" or "rewind". We've improved on the typical UNIX
      implementation in this area by returning EOF if you fail to
      do this. Many versions of UNIX will silently smear your
      file if you violate the rule.

(3)   Normally the level 2 I/O functions acquire buffers via the
      level 2 memory allocator unless the file is on some device
      other than a disk. We've added the standard UNIX "setbuf"
      function that allows you to attach your own buffer via the
      call:

              char buffer[BUFSIZ];
              FILE *fp;
              setbuf(fp, buffer);

      Note that this function assumes that the buffer is the
      standard size, which is defined via the BUFSIZ constant in
      stdio.h. If you call it with a null pointer, it behaves the
      same as "setnbf" and makes the file unbuffered.

(4)   Level 1 I/O no longer acquires buffers via the level 2
      memory allocator, so you don't need to worry about screwing
      up files when you do a "rstmem". Under MS-DOS, buffering is
      now performed by the operating system itself, resulting in
      improved performance for large read/write data transfers.

5.2.2 Level 1 I/O device processing

Device names are now 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 it as if it were a disk. The device names recognized
are as follows:

          Console   CON:

          Printer   PRN:, LST:, LPT:, LPT1:
          Aux port   AUX:, COM:, COM1:, 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 re-
quests 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.

5.2.3 Implementation of level 2 I/O buffering

The level 2 I/O function "fopen" sets operations for files to be
buffered in 512-byte blocks, as in the previous version. If the
name it passes to "open" is recognized by that function as being
a device, "fopen" sets operations to be unbuffered. The old
"line-buffered" mode supported by the previous version has been
scrapped; now, unbuffered I/O is handled in true single-character
fashion, as in UNIX. Thus, "printf" to the console sends the
characters immediately, whether or not a newline is sent. Input
is also unbuffered, and only a single character is read at a
time. Note, however, that the "_main" function provided in the
library sets up "stdin" to be buffered, which causes buffered
console input to be used.

5.2.4 New functions: fread/fwrite

Two new functions for reading and writing blocks of data to
buffered files have been added to the library. These functions
work with the level 2 I/O functions ("fopen", "fclose"). Here
is the manual page for these new functions.

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 writ-
    ten; may be less than "n" if error or end of file occurred

CAUTIONS

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

5.3 Memory allocation functions

The new versions of the memory allocation functions have been
extended to support the capability of a memory pool in excess of
64K bytes. For the most part, the new functions are compatible
with the old. The exceptions are listed below.

5.3.1 "sizmem" function now returns "long" size in bytes

In the old version, "sizmem" returned the size of the memory pool
as an "unsigned" number of 16-bit words. The new version returns
the available memory in bytes, and must be declared as a "long"
integer function:

        long n, sizmem();

        n = sizmem();

5.3.2 "getml" function

The new function "getml" works exactly like the "getmem" function
except that it accepts a "long" integer argument.

        p = getml(lbytes);
        char *p;   pointer to memory block, or NULL
        long lbytes;   size of desired block, in bytes

In accordance with the usual convention, "getml" returns a null
(zero) pointer if it cannot allocate the requested block. Note
that the function must be declared "char *".

5.3.3 "lsbrk" function

The new function "lsbrk" is similar to "sbrk", but accepts a
"long" integer for the size argument. Its return value on
failure is zero instead of the -1 returned by "sbrk".

        p = lsbrk(lbytes);
        char *p;   pointer to allocated block, or NULL
        long lbytes;   size of desired block, in bytes

5.3.4 Change in the action of the "rstmem" function

According to the manual, the level 2 memory allocation function
"rstmem" restored the memory pool to its empty state by calling
"rbrk". This had the effect of deallocating all of the memory
obtained by calls to "sbrk", as well as calls to "getmem". In
the current version, the action of "rstmem" has been changed so
that only allocations made (by calls to "getmem") after a call to
"allmem" are affected. Thus, the cautions on page 1-37 of the
manual should read:

(1) The reset function "rbrk" cannot be used if any of the
     standard I/O functions are also being used on currently open

22

files.   This restriction applies only to level 1 functions.
Files 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)   A similar restriction applies to use of the function
      "rstmem", except that files opened BEFORE the most recent
      call to "allmem" are not affected. Thus, if a program opens
      all files first and then calls "allmem", it may safely call
      "rstmem" without affecting those open files. Any files
      opened AFTER the "allmem" call must be closed before
      "rstmem" is called.

5.4 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." Sec-
ond, beware of using the macro names in declarations such as

        int min;

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

5.5 New utility functions

Several new utility functions have been added to the library
which allow access to all of the features of the 8086/8088
processors, including (1) software interrupts (useful for making
direct ROM BIOS calls); (2) BDOS functions; (3) access to segment
register contents; (4) inter-segment memory transfers; and (5)
"peek" and "poke" functions for examining and setting arbitrary
memory locations.

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.
Check the system-level documentation for your operating
system 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 inter-
rupt is performed; "outregs" will receive the register val-
ues after control returns from the interrupt. With
"int86x", you can specify the values which will be placed in
the segment registers before the interrupt; 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

The software interrupts on the 8086 are used to implement
all sorts of system level processing, and invalid input data
can cause unpredictable (and occasionally disastrous) re-
sults. 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

    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.
    Check the system-level documentation for your operating
    system 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",
    you may specify the values which will be placed in the
    segment registers before the interrupt; 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

       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 nota-
    tion. This function is primarily intended for use in pro-
    grams 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 regis-
    ter 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 re-
    sults.

6.0 UTILITY PROGRAMS

The function extract utility has been modified, and a new utility
program -- the object module disassembler -- has been added to
the compiler package.

6.1 New version of Function Extract Utility

Extensive modifications to the function extract utility have made
the description in the manual (Section 1.1.5) inaccurate. Here
is the corrected version of that section.

(1.1.5 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.

       Programmers who are a little 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 just defines 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

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
1.1.4).

<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 specifed file. The function name must be
                specified exactly as it appears in its definition,
                except that alphabetic characters may be specified
                in either case (upper or lower).

     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

Extract the function called "strcnt" from the text file STRING.C;
do not include any preceding text; and write the extracted text
to the user's console.

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

Extract the function called "input" from the text file IOFUNC.C,
and prepend the output with the text from the file IOS.H; and
write the resulting text to INPUT.C. If each function in
IOFUNC.C can refer to the external locations "flagl" 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 flagl, flag2;

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

6.2 New utility program: Object Module Disassembler

     For programmers who wish to debug C modules at the machine
code level, the object module disassembler 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 the MS-DOS debug program. The
usefulness of this utility, of course, is limited to those
programmers who are knowledgeable about the 8086 architecture and
instruction set.

     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 the special character "-" 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

Disassemble the object module QRS.OBJ and write the listing to
the user's console. Allocate 2048 decimal bytes of storage for
the program section defined in the object module, and allocate
8000 decimal bytes for the data section.

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

Disassemble the object module XYZ.OBJ and write the listing to
the file TEMP.LST. Include source code lines from XYZ.C in the
listing, provided that line number and offset information was
present in the object file. Provide 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.

7.0 STOCKLIST

Due to the support for the extended memory addressing models, the
list of files supplied with the compiler has grown considerably.
Note that the manual describes the linking process as involving
the files C.OBJ and LC.LIB; in the current release, four differ-
ent versions of each of these files have been supplied. The
procedure for linking programs is the same, but now you must use:
for C.OBJ, either CS.OBJ, CP.OBJ, CD.OBJ, or CL.OBJ; and for
LC.LIB, either LCS.LIB, LCP.LIB, LCD.LIB, or LCL.LIB. Make sure
the same memory model is selected for both files.

You should find the following files on your release disk(s):

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 concatenate 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

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 sample source modules, you
must copy one of the last four files into DOS.MAC)

OBJECT MODULE LIBRARIAN REFERENCE MANUAL

by Phoenix Software Associates, Ltd.

     The Object Module Librarian (OML) manipulates libraries
object files on the Intel Corporation (1) 8086 (or 80:
processor (tm) under the MS-DOS (2) or CP/M-86 (3) operat
systems.

     OML handles object files and libraries conforming to
format generated by the Microsoft compilers for the Intel 80
This is actually the standard Intel format with an enhan
library index. A list of compilers that produce object c
compatible with this format is given later.

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

     The next section of this manual describes how to use OML
handle several common object library situations.   At the s;
time it provides an informal explanation of what the commands
Those readers experienced with linkage editors and library m
agers may wish to skip directly to this portion of the manu
it provides enough information to handle most applications.

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

Trademark Acknowledgment:

(1)   Intel is a trademark of Intel Corporation.

(2)   MS-DOS is a trademark of Microsoft, Inc.

(3)   CP/M-86 is a trademark of Digital Research.

I. LIBRARY MANAGER CONCEPTS

Typically it is convenient (if not essential) to divide a large programing 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.

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 run time support are often sold in the form of libraries. An example might be a data base manager 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 library format is an indexed structure.

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

OML provides commands that can 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 wh
creating a program. In other words, one can create a libr
consisting of only those modules that the linkage editor wo
have included in a particular program.

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

CREATING/MERGING LIBRARIES...

To create a new library use the BUILD and FILE comma
For example, executing OML and entering

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

would create a library named DB.LIB containing the files lis
after the FILE command. These files could be single obj
modules or complete libraries. Everything is merged int
single library.

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

Another way to use OML is to specity its actions on the
command line. For example, the above library could have t
created by entering (on one line):

```
OML BU DU FI BTREE,SORT,REPGEN,FIRSTLIB:LIB
```

Note that the output file type defaults to LIB automa
cally.

SEARCHING A LIBRARY...

Suppose you want to create a library consisting of seve
modules plus those portions of another library that
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 ot

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
also 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 preceding file and
causes the COSINE module in the MATHLIB to be ignored.

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 speci-
fied. 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 commands like 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.

4

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, OML assumes that the currently logged-in disk is to be used.

INITIATING OML

    OML may be used interactively, or input may be given on the command line:

        OML   <statements> <cr>

where <cr> means to press the RETURN key.   This means that OML may be used in .BAT files.

    To use OML in the interactive mode, enter

        OML <cr>

on the console. OML will read statements from the console, prompting with "=>". All input is stored uninspected until a carriage return is typed. The standard line editing features supplied by the operating system are available.

    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,

        OML @<file name> <cr>

creates the library. Sometimes these ".LNK" files may be pre-pared once for a given library and used over and over again, greatly simplifying the whole process.

    OML reads an entire command, checking for syntax only, before any file processing is done.

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, OML assumes that the
currently logged-in disk is to be used.

INITIATING OML

    OML may be used interactively, or input may be given on the
command line:

        OML <statements> <cr>

where <cr> means to press the RETURN key. This means that OML
may be used in .BAT files.

    To use OML in the interactive mode, enter

            OML <cr>

on the console. OML will read statements from the console,
prompting with "=>". All input is stored uninspected until a
carriage return is typed. The standard line editing features
supplied by the operating system are available.

    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,

        OML @<file name> <cr>

creates the library. Sometimes these ".LNK" files may be pre-
pared once for a given library and used over and over again,
greatly simplifying the whole process.

    OML reads an entire command, checking for syntax only,
before any file processing is done.

COMMAND FORMAT

All OML input is free format. Blank lines are ignored, a command may extend to any number of lines. Comments ma} included with input from any source by using a percent sign When this is encountered, all remaining characters on the s line are ignored.

Input is a list of statements:

<statement> <statement>...<statement>

Each statement begins with a key word, and many are follc 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. words may be abbreviated by omitting trailing characters, as : as the abbreviation is unique among the entire group of words. For instance, the previous statement could have b entered as

FI A,B,C

If a syntax error is found, the current input line is ec} with two question marks inserted after the point at which error was detected. This is followed by an error message Appendix). The command must then be re-entered.

If some other error occurs, OML terminates with an er message also listed in the appendix.

OBJECT FILES

OML must be told what object files and libraries to use input and what modules to select from them. The FILE command typically used, and normally causes all modules with the gi files to be processed:

FILE COSINE, SIN, ARCTAN

The LIBRARY and SEARCH commands are similar, but are t only on libraries and select only those modules that defir public symbol that is needed by some other module that already been processed. This is called a "library search" anc a process carried out by most linkage editors. It insures t only those library modules that are actually needed are incl┥ in the program.

LIBRARY MATHLIB
SEARCH FORTRAN

The LIBRARY command causes the given libraries to

searched once. When the SEARCE command is used the libraries may be searched multiple times 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 OML can't find a requested object file it will look on drive A to find it, and will then ask the operator to enter the 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 OML runs out of memory a work file is opened on the default disk, which then may not be removed.

Under MSDOS 2.0 operating systems OML will accept a path name as part of an object file name. Also, if an object file can't be found OML will look for a string named "OBJ" in the environment and append its value to the front of the file name, after stripping any drive id. For example, suppose that the operator enters

    SET OBJ = \OBJECT

and then runs OML. Let us suppose that one of the commands to OML is

    FILE B:TEST.OBJ

and that TEST.OBJ doesn't in fact exist on drive B. OML would strip the B: from the name and then try \OBJECT\TEST.OBJ to obtain the requested file.

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 most recent FILE given. For example,

    FILE MATH1 AS COSINE

would name the module in MATH1 COSINE instead of MATH1.

If you are processing libraries built with Microsoft's library manager you will get several checksum errors. These arise because the Microsoft library manager renames the modules as OML does but does not re-compute the checksum field at the end of the module name record. The messages should no longer appear once the library has been re-built by OML.

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, CONSINE
LIB MATHLIB. DB EXCLUDE.BTREE

The INCLUDE statement causes only those modules listed to
considered for processing, and this selection precedes a libr
search. EXCLUDE is the opposite. The modules listed are
processed. INCLUDE and EXCLUDE apply to the FILE, LIBRARY
SEARCH file immediately preceding. In the second example abo
for instance, the EXCLUDE BTREE applies only to the DB libra
not MATHLIB.

BUILDING A LIBRARY

The BUILD command is used to create a library out of
modules selected from the input files. It is followed by
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
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
EXTRACT command (described next). If no output is requested f
OML (i.e. there is no BUILD, EXTRACT or LIST command) then (
will simply read the input modules and report any errors
finds.

EXTRACTING A LIBRARY MODULE

The EXTRACT command is used to extract a single obj
module from a library file and place it into a separate d
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
input files. Therefore it is usually necessary to use
INCLUDE statement to specify which library module should
extracted. For instance,

9

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 FIMATHLIB 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 currently two reports available:

M - A list of all modules processed in alphabetical order.

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 external symbols). 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 ·

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 exam-
ples:

WIDTH 132
HEIGHT 88

CONTROLLING THE LIBRARY INDEX

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 sever
versions of the same module, for instance a device driver f
some kind of hardware. If you try to place all of the modul
into the library you will get duplicate symbol warnings, and
link time the linkage editor wouldn't be able to select t
desired module.

This can be made to work by using NOINDEX on the modu
entry points. This excludes all of these symbols from t
library index. To get the linkage editor to select the corre
modules insert an un-used but unique dummy symbol into ea
module. At linkage edit time one of these dummy symbols would
accessed in order to create a need for the desired module. 1
linkage editor would then select it when the library is search

Using OML, for instance, one could use a statement like

        DEFINE FOO=DRIVER1

to select the module containing driver 1. An alternative is
rely on the fact that the name of each module is actually in 1
library index as well, followed by an exclamation point. F
example, if the library contains a module named DRIVER1 th
there will be a dummy index entry named DRIVER1!. These symbc
can be used instead of creating a dummy module entry point
discussed above.

more uncommon or obscure errors a number is printed on the console that may be looked up below.

COMMAND SYNTAX ERRORS

The following errors are caused by mistakes made in the input given to OML. The input line causing the problem will be displayed on the terminal, with a couple of question marks inserted at the point where the error was detected. Re-run OML after correcting the problem.

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 rebuilding 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 0 thru 9).

10 — 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.

15 — Expecting "="

16 — Expecting a value. A 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 OML twice with one command in each.

WORK FILE ERRORS

When OML runs out of memory it opens a work file on di
named PLIB86.WRK to hold the description of the library. The
error codes indicate a problem with processing the work file.

30 - The work file can't be created. Probably there is no spa
      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 th
      library (about 50,000 symbols, modules, and so on may
      defined). This library is too large for OML to handle.

INPUT OBJECT FILE ERRORS

The following errors have to do with the object files th
are given to OML to process. Usually they occur when a file h
been corrupted somehow. Try re-compiling to get a new copy
the object file. If it is a library supplied by the compil
manufacturer that is causing the problem, try to get a fresh co
of it.

41 - Premature end of input object file. The end of the ind
      cated file was reached unexpectedly. Possibly, the fi
      was truncated by copying it with a program that assumes
      CNTL-Z (1AH) is the end of the file.

42 - Fatal read error in object input file.

43 - Fatal file position error in object input file. This c
      occur when a library file is truncated.

OUTPUT FILE ERRORS

The following errors are caused by a problem in creati
the output code file or memory map file (when written to disk
Often, they are caused by a full disk or disk directory, a di
that is write-protected, or some kind of hardware problem wi
the disk.

45 - Can't create output disk file. Possibly the disk directo
      is full, or the disk is write protected.

46 - Output file too large. The given modules won't fit in
      the library. You will have to break up the library in
      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. Probably, an irrecoverable hardware error has occurred.

49 – Can't close output file. Probably the disk is write protected, or a hardware error has occurred.

50 – 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 OML. You must have a really tiny memory – better buy more!

SYSTEM ERRORS

These errors indicate that a system error 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 OML again. If the error persists, please gather the relevant information and contact the software distributor from whom you obtained OML.

201 – Expandable array bug.

205 – Seek errors while writing output file (attempt to seek past end of file).

219 – Bad object block (GetBlock).

221 – Invalid object key (Q).

222 – Invalid object key (QM).