Installation Notes

for

FlexeLint

A Diagnostic Facility

for

C and C++

Software Version 9.00 and Later
Document Version 9.00

August, 2008

Gimpel Software
3207 Hogarth Lane
Collegeville, PA 19426
(610)584-4261 (Voice)
(610)584-4266 (fax)

www.gimpel.com

1. GETTING STARTED

FlexeLint is distributed on a single CD ROM. The root directory of this CD contains the following files and directories.

`inotes.pdf`	This document which is in PDF (Portable Data Format).
`flexelint.pdf`	The FlexeLint manual in PDF.
`readme.txt`	Additions and modifications to the manual.
`msg.txt`	Messages and explanations in ASCII format.
`src/`	A directory bearing all the source code that needs to be compiled.
`supp/`	Supplementary information subdivided into different categories.

There are four steps in the installation of FlexeLint.

1) Copy the files to your hard drive.
2) Customize the source code if necessary.
3) Compile the source files to produce an executable.
4) Select an appropriate set of options so that FlexeLint will be able to find and process your compiler's header files and successfully process C/C++ programs written for your compiler. You should also package these FlexeLint options into a convenient shell script.

These 3 steps are discussed in the following sections:

Section 1.1 Copying the Files
Chapter 6. CUSTOMIZATION
Section 1.2 Compiling the Source
Section 1.3 Adapting FlexeLint

1.1 Copying the Files

It will be necessary to create a directory in which to house FlexeLint. Copy into this directory the four files on the root directory of the FlexeLint CD (**`inotes.pdf`**, **`flexelint.pdf`**, **`readme.txt`**, **`msg.txt`**). Also copy all source code (from directory ***`src/`***). You may place this in a subdirectory of the FlexeLint directory so as to mirror the structure of the distribution CD.

It is impossible to give detailed instructions on how to create directories or on how to copy source files from the FlexeLint CD to your host system. These will vary from host to host. Special considerations are required when installing on a system that uses the EBCDIC character set. See Section 5.9.1 EBCDIC. In virtually all cases the copying seems to proceed smoothly as there generally is a body of accumulated local expertise to resolve the copy problem.

1.2 Compiling the Source

Step 2 is compiling the source code. For this step and especially the next step, you will need to have C and/or C++ experience. If you are a system administrator assigned the task of installing FlexeLint now would be a good time to contact your client programmer.

The source code for FlexeLint consists of the following files:

```
vers.h
custom.h
custom.c
a2.c
a3.c
a4.c

...
```

The precise number of source files can vary. There are approximately 30 C modules (files ending in `.c`). Compile and link together all of the modules. On Unix and similar systems you need simply type

```
cc  *.c
```

On other systems, things can be a little more complicated. Note however that we do not provide a makefile script as such scripts are operating system dependent. One example of such a script provided by one of our customers is:

```
GOAL=flint9.0B8

all: clean
      gcc -c *.c
      gcc -o $(GOAL) *.o -lm
      ./$(GOAL) -v

clean:
      -/bin/rm -f *.o
      -/bin/rm -f $(GOAL)
```

You will also need to name the executable either as part of the compilation process or by renaming the default executable. We suggest a name of `flint`. Any other name consistent with your administration policies would also be satisfactory insofar as the software is concerned. We also suggest that you place the executable `flint` in your FlexeLint directory (as opposed to the `src/` subdirectory). You may need to make it available on a system wide basis. This could be done by placing `flint` in some generally accessible directory in the PATH of a group of interested users. These practices will vary with the site.

Beware that these modules are C modules and cannot be compiled by a C++ only compiler. On many systems it is best to compile with optimization off, as optimization can interfere with correct behavior. The modules `#include` the header file `stdio.h` and the resulting object modules should be linked in the presence of the standard library. While compiling, you may experience some warning messages such as "label cannot be reached". These are the result of the shrouding process and should be ignored. It would be very rare but when linking, you may experience unresolved references for such functions as `memcpy`. If you do, check `custom.c` to see if the functions are defined there. A function such as `memcpy` is in the ANSI/ISO standard for C and is widely available but may simply not be defined for your system. For this reason we have supplied alternative versions in `custom.c` that may be activated by altering a conditional compilation statement.

Customization is possible by modifying the file `custom.c` or `custom.h`. This is described in Chapter 6. CUSTOMIZATION. Except for these files, the source code is not intended to be read and understood by humans. The source code is intended only to be compiled and then only for the intended machine and operating system. The source has been passed through a processor called The C Shroud (a trademark of Gimpel Software) and the resultant code is sometimes referred to as "shrouded source". If you are experiencing difficulties with this source code, you are urged to contact your vendor. Very often the source code can be reprocessed through The C Shroud with different parameters correcting the problem much more easily than by hand modification. In any event, the difficulty that you are experiencing should be reported so that the problem can be corrected.

Having compiled FlexeLint, you are ready to pass on to step 3.

1.3 Adapting FlexeLint

In Step 3 we select an appropriate set of options, and package these into a convenient script file. Chapter 2. COMPILER DEPENDENCIES discusses generally the issue of option selection.

Chapter 4. SETUP discusses setup procedures for your operating system and in particular the creation of a `lin` script.

The directory `supp/` (Supplementary files) contains mostly machine-dependent, compiler-dependent, or system-dependent information further partitioned into directories. These are described in Chapter 5. SYSTEM DIFFERENCES. Some of this information was provided by FlexeLint users. There is a file named `readme.txt` in the directory and in many of the subdirectories.

One of the subdirectories provides for pure ANSI C headers. Another directory provides for pure POSIX C headers. See Section 3.2 Portability Checking.

2. COMPILER DEPENDENCIES

Once you have a working FlexeLint executable (referred to as `flint` in these notes) it is then necessary to have it understand the dialect of C or C++ understood by your host compiler. No two C/C++ compilers are identical. They differ in their adherence to language standards, especially in extensions to these standards. FlexeLint has a wide range of options and a flexible error-message suppression capability that can adapt to a wide range of compilers. In particular, see Section 5.8.3 Customization Facilities, in the FlexeLint Reference Manual (`flexelint.pdf`). This chapter will discuss additional ways of adapting FlexeLint to your operating environment.

2.1 co.lnt

We recommend a file (whose extension necessarily is "`.lnt`") to describe Compiler Options for your compiler. Its name is unimportant (save for the extension); we will use the name `co.lnt`. `co.lnt` will contain the necessary options to enable FlexeLint to process typical programs for your compiler. In particular, it will enable FlexeLint to process cleanly the header files provided by your compiler. For example, if your compiler expects `UNIX` to be predefined then `co.lnt` should contain

```
-dUNIX
```

(For an automated method of generating appropriate pre-definitions see the option `-scavenge`, in the FlexeLint User Manual.)

If your compiler allows the preprocessor command `#ident` you will want to place the option

```
+ppw(ident)
```

in `co.lnt`. If your compiler allows the reserved word `fortran` then you may want to place the option

```
+rw(fortran)
```

within `co.lnt`. Alternatively you may redefine the identifier `fortran` with the option

```
-dfortran=
```

which defines it to null.

FlexeLint will want to know the sizes of fundamental data items such as the `short int` or plain `int`. Happily these are picked up automatically by the compilation process so they will match your compiler. But if you are cross-linting you may want to place options of the form

```
-st#
```

(where ***`t`*** is a type and # is an integer) to denote the assumed size (in bytes) of type ***`t`***. See Section 5.3 Size and Alignment Options, in the FlexeLint Reference Manual.

There are a number of models for 64 bit compiling. If you are using 64 bit integers make sure you direct FlexeLint to employ 64 bit arithmetic. This could well happen automatically but it may not. See Section 5.1, 64 Bit.

In addition to such elementary definitions, you may want to inhibit messages resulting from compiler headers. For example, if your compiler header contains the construct

```
#endif   name
```

this will result in message **`544`**. To comply with Standard C/C++, the identifier ***`name`*** should have been placed in a C/C++ comment. You can inhibit the warning message with **`-e544`**. But it is probably wiser to inhibit the message only for library headers using

```
-elib(544)
```

This way you will still be alerted to this situation in your own code. Alternatively, and this is the practice we have more recently adopted, we use the option **`-wlib(1)`** which sets the Warning level to 1 (Errors only, no Warnings) when processing library headers.

Another example of an error suppression option that you may want to place in **`co.lnt`** is as follows: The statement

```
printf("hello world\n");
```

would normally result in a warning because **`printf`** is declared as returning an **`int`** and the return value is ignored resulting in message 534. You may suppress this message for just the **`printf`** function using the option

```
-esym(534,printf)
```

Several such options are found in the file **`/supp/generic/co.lnt`**. You are invited to add to or subtract from the list of functions.

The file **`co.lnt`** can be used on the FlexeLint command line before the list of modules comprising a program. For example:

```
flint  co.lnt  test.c
```

or

```
flint  co  alpha.c  beta.c
```

or, for unit checkout

```
flint  co  -u  alpha.c
```

Chapter 4. "SETUP" in this manual contains a further discussion on setup.

2.2 sl.c

Note: You may skip this section if your compiler supports prototypes in its header files. You certainly want to skip this section if you are linting C++ code.

File `sl.c`, or some variation of it, is used to specify the Standard Library for compilers that do not yet support ANSI prototypes. If your compiler is an ANSI compiler, its header files should contain all the necessary prototypes to describe its library and `sl.c` is not needed.

Like `co.lnt`, the precise name of `sl.c` is unimportant. `sl.c` is organized along the lines described in Section 6.2 Library Modules, in the FlexeLint user manual. Briefly, it contains a prototype for each function in the compiler's library and this is matched against any attempted use of a library function. If no prototype is provided for such a function, you will receive an appropriate diagnostic (Warning 628). Print out the file `sl.c` from the generic subdirectory to see an example of what you can customize for your compiler.

You should use `sl.c` on the command line along with `co.lnt`. For example:

```
flint  co  sl  file1  file2  file3
```

Normally, on extensionless files, extension "`.lnt`" is tried first, then "`.cpp`", then "`.cxx`" and finally "`.c`".

2.3 custom.c

To further customize FlexeLint for your environment, you may modify the files `custom.c` and/or `custom.h`. This module is fully commented and is provided in its original non-obfuscated source form so that local modifications can be carried out. Additional information is provided in Chapter 6. CUSTOMIZATION.

3. MISCELLANY

3.1 Default Value for Options

There are a relatively large number of options. These are described fully in the FlexeLint Reference Manual (`flexelint.pdf`). A summary of the options can be obtained by typing `flint` on the command line without arguments. As in:

```
flint
```

Alternatively you may use a `?` as an argument. This is handy for capturing the options in a file. Since options are sent to standard error you need to redirect standard error using the `-oe` option. Thus:

```
flint -oe=file ?
```

captures options in file ***`file`***. The default value of the size options and the flags are provided in the option list produced in this way. FlexeLint is designed so that an installation may preset certain flags or preset certain options by setting a global variable (`sys_olist` in `custom.c`). The values of options given in the option summary reflect the values *after* `sys_olist` is processed. In this way options set by a system administrator appear as part of the FlexeLint system.

3.2 Portability Checking

You may check your C code for ANSI/ISO portability by using our supplied ANSI C header files. This will ensure that no strictly local facilities are being used, since these header files have been prepared in accordance with and limited to ANSI/ISO specifications. They may be found in the directory `supp/ansi`

Similarly, you can check your code for POSIX C conformance by using our supplied POSIX C header files. These may be found in the directory `supp/posix`

4. SETUP

4.1 The lin Script File

You may find it more convenient to access FlexeLint through a command script file. On UNIX, this file could be a shell script called `lin` containing something like the following:

```
lin:

     flint +v -i/u/flint std.lnt $1 $2 $3 $4 $5 ... >temp
     more temp
```

The `-i` option is followed by a directory that is used to search for files not found in the current directory. This applies to all files: header files, modules, and indirect files (`.lnt` files).

Here we have assumed that directory `/u/flint` houses FlexeLint. If you have chosen some other directory, make the appropriate change in the script. (Multiple `-i` options can be given, and, for some systems, the environment variable `INCLUDE` is also supported. See Section 4.2 INCLUDE Variable.).

`std.lnt` is described in Section 16. LIVING WITH LINT, in the FlexeLint Reference manual. Its contents are something like:

```
std.lnt:

     /* a standard std.lnt */
     co.lnt
     options.lnt
```

`co.lnt` contains options appropriate to a compiler. `options.lnt` contains site-wide error suppression options. Both of these files are also placed in the common search directory along with `std.lnt`.

Running FlexeLint in this fashion (i.e. through `lin`) dumps the error messages into a file (named `temp`) which is fed a screenful at a time to your console. If your system doesn't support redirection, you may have to use the option `-os(temp)`; since options are processed in order (unlike redirection) place this option before all module names on the command line. The `+v` makes sure that in-progress messages are sent to the screen as well as the file.

To apply FlexeLint (on UNIX) to files `m1.c` through `m3.c` you could then use:

```
     lin  m1  m2  m3
```

Since the errors are saved in a file, you can use any multi-file editor to alternately flip between the error message file (`temp`) and the source file to eliminate the errors.

Please note that by default, modules ending in "`.c`" are taken to be C modules. You may use the option

```
        +cpp(c)
```

to indicate that such modules are C++ modules. Such an option is a candidate to be placed into `options.lnt`

For projects consisting of a large number of modules, create an indirect file containing the names of the modules. For example, suppose a project consists of modules `m1.c`, `m2.c`, `m3.c`, `m4.c` and `m5.c`. Create a file `project.lnt` containing:

```
    project.lnt:

        m1.c  m2.c  m3.c  m4.c  m5.c
```

Then

```
        lin  project
```

will process all the modules with the proper options. To process any subset of the modules, specify them explicitly and use the `-u` option. For example, to process just `m2.c` and `m4.c` use:

```
        lin  -u  m2  m4
```

The `-u` stands for 'unit checkout' and suppresses error messages that are only appropriate if all the modules are being supplied.

To support a locally unique message suppression strategy, you may add to `std.lnt`. For example, assume we wish to suppress message 765 but only for this project. We would create a local copy of `std.lnt` in the project's directory as follows:

```
    std.lnt:

        /*  std.lnt (special version for project X)  */
        std.lnt
        -e765
```

This appears to be a recursive invocation of `std.lnt` but it is not. It is simply a reference to the next file by that name on the directory search list. See Section 5.9 Self-Referencing Options Files, in the FlexeLint Reference Manual.

4.2 INCLUDE Variable

Some (operating systems)/(compiling systems) support environment variables (sometimes called `SET` variables). These include Unix, Linux, OS-9, Windows, OS/2 and MS-DOS. For these systems the `INCLUDE` environment variable may specify a search path in which to search for header files (`#include` files). It specifies a sequence of directories separated by a special character as follows (`SEP2` defined in `custom.c`, see Section 6.2.1 Initial Macros)

MS-DOS	`;`
OS/2	`;`
OS-9	`;`
Unix	`:`
Linux	`:`

Systems also differ as to what character is used as a path separator character (`SEP1` in `custom.c`). For MS-DOS and OS/2, the path separator character is the backslash (`\`) and for Unix and OS-9 the path separator character is the forward slash (`/`). We will provide an example using the forward slash suitable for Unix. For example, on UNIX,

```
set INCLUDE= /include : /etc/include
```

specifies that, should the search for a `#include` file within the current directory fail, a search will be made in the directory `/include` and, on failing that, a search will be made in the directory `/etc/include`. This searching is done for indirection files (`.lnt` files) and modules as well as `#include` files.

Notes:

1. No blank may appear between '`INCLUDE`' and '`=`'. Blanks, otherwise, may be used freely and are ignored.
2. A terminating `SEP2` is ignored. (See Section 6.2.1 Initial Macros)
3. This facility is in addition to the `-i`... option and is provided for compatibility with a number of operating systems and compilers.
4. Any directory specified by a `-i` directive takes precedence over the directories specified via the `INCLUDE`.

4.3 LINT Environment Variable

For those systems supporting environment variables (see `sys_env` in Section 6.2.3 Customizable Functions, if the environment variable `LINT` is set, the associated value is prepended to the arguments with an assumed intervening blank.

4.4 Exit Code

Many operating systems support the notion of an exit code whereby a program may report a byte of information back to a controlling program. FlexeLint will set the exit code to the number of errors encountered (with an upper bound on this exit code of 255). If you want the exit code to always be **`0`** use the option **`-zero`**. See also **`-zero(`*`message-number`*`)`** in Section 5.7 Other Options, in the FlexeLint Reference Manual.

5. SYSTEM DIFFERENCES

This chapter gives helpful information on a number of specific systems on which FlexeLint is being run. Variations in the behavior of FlexeLint for various systems result from differences in the files `vers.h`, `custom.h` and `custom.c`. These are described in Chapter 6. CUSTOMIZATION. See also Chapter 2 COMPILER DEPENDENCIES..

Much of the information contained in this chapter is found in directory `supp/`. This directory may contain additional and/or more current information than this manual.

5.1 64 Bit

Directory `supp/64bit` contains information that addresses 64-bit programming. You can conduct an experiment to see if `custom.h` should be modified to enhance the precision of our integral operations.

5.2 AIX

For the purpose of compiling and running FlexeLint, AIX may be regarded as a dialect of UNIX. Please see Section 5.20 Unix, for information regarding running under Unix.

5.3 ALPHA AXP

Directory `supp/alpha` pertains to the Alpha AXP (64-bit RISC) architecture that originated as the DEC Alpha from Digital Equipment Corp. This directory will provide additional details in its `readme.txt`.

5.4 ANSI/ISO

ANSI/ISO is not an operating system but a standardization body. You can check your programs for ANSI/ISO C compliance by using only the header files found in the directory `supp/ansi`. See Section 3.2 Portability Checking.

5.5 Apollo

Additional information about using FlexeLint for this system appears in the directory `supp/apollo`.

The Apollo compiler supports what we refer to as an abbreviated structure reference and is simulated with `+fab`. It also supports anonymous unions (`+fan`). Accordingly, both options have been placed in the Compiler Options file (`co-apollo.lnt`) in the directory `supp/apollo`.

To accommodate the `__attribute(…)` option of the Apollo compiler, use the FlexeLint option:

```
-d__attribute()=
```

This will cause FlexeLint to ignore the `__attribute(…)` construct. This also appears in `co-apollo.lnt`.

5.6 OS/2

Information about using FlexeLint on OS/2 can be found in the directory `supp/os2`.

5.7 Data General AOS/VS

Information about using FlexeLint for this operating system appears in the directory `supp/dg-aos`.

If you are using the Unix or Linux operating system on Data General equipment go to Section 5.20 Unix.

The command line for FlexeLint follows the conventions for the AOS/VS system. To execute FlexeLint the command line is:

```
x flint Arguments
```

where the ***Arguments*** are as described in Section 4. THE COMMAND LINE, in the FlexeLint Reference Manual.

To redirect the output of FlexeLint on the AOS/VS system you may use a command line such as:

```
x flint/l=FileName Arguments
```

This will redirect Standard Output. To redirect Standard Error as well, use the command:

```
x flint/l=FileName/e=FileName2 Arguments
```

FileName and ***FileName2*** may be the same file. Alternatively, use the `-os(`***filename***`)` option. See `-os` option in Section 5.7 Other Options in the FlexeLint Reference Manual.

You may want to allow dollar sign (`$`) in identifiers, especially when processing the header file `paru.h`. Use the option `-$`. The Data General C compiler allows the keyword `$builtin`. This should be defined to be `extern`. The keyword `$asm` should be ignored as should constructs of the form: `$algn(…)`. Accordingly, the options

```
-$
-d$builtin=extern
-d$asm=
-d$algn()=
```

have been placed into `co-aos.lnt` in the directory `supp/dg-aos`.

5.8 HP-UX

Information on using Hewlett Packard’s HP-UX system can be found in directory `supp/hp`. Much of this information has been furnished by HP personnel.

Also, please see Section 5.20 Unix, for general information about operating in a Unix environment.

5.9 IBM VM

Additional information about using FlexeLint for IBM’s VM appears in the directory `supp/ibm-vm`.

5.9.1 EBCDIC

The primary difference between this variant of FlexeLint and other variants is the use of the EBCDIC character set.

The source for FlexeLint is distributed in ASCII. Accordingly, you must first transliterate the ASCII source into EBCDIC before it will compile on your system. We do not provide this transliteration for you, although we could easily do so and we will upon request. The reason we don't, is that the best transliteration to use is the one that your shop is already using to translate ASCII programs into programs acceptable by your C compiler. In that way you can be reasonably assured that your compiler will accept the resulting source code. Also, the characters that FlexeLint itself uses to parse C code will automatically become the same characters used by your compiler.

The transliteration is assumed to be one-one for each of the special characters. For example, the '`{`' should not become a pair of characters since this character is contained in strings and its translation will be subsequently used by FlexeLint as equivalent to '`{`'. However, the carriage-return newline sequences can become end-of-records and tabs can be converted to blanks.

There are also special escape sequences recognized by FlexeLint as representative of certain C source characters and these are shown in the table below. These are in addition to the natural transliteration character alluded to above and the ANSI escape sequences.

C source character	*First Alternate*	*Second Alternate*	*Third Alternate*	*Fourth Alternate*
`{`	`\(`	`(<`	`(:`	`0x8B`
`}`	`\)`	`>)`	`:)`	`0x9B`
`[`	`( 0x4F`			
`]`	`0x4F )`			
`|`	`0x4F`			

In the table above, the hex codes beginning with `0x` represent the EBCDIC character with that hex code. The other characters are those that are the targets of the transliteration for those ASCII characters.

For example, if the ASCII left bracket character, '`[`' is transliterated into hex code `0xAD` then FlexeLint will accept any of the following codes for left bracket:

the single character: `0xAD`	(the transliteration target)
the pair of characters: `( 0x4F`	(from the above table)
the triple of characters: `? ? (`	(a Trigraph)

5.9.2 Options

Some options you may want to select for the IBM mainframe variant are

`+fcu`	Characters are normally Unsigned
`+fdl`	the type of pointer Differences is `long`

These options are placed in the file `co-vm.lnt` in the `supp/ibm-vm` directory.

5.10 LINUX

Additional information about using FlexeLint for this system appears in the directory `supp/linux` but also see `supp/unix`.

For Linux, the source code is compiled with the symbol `os_UNIX` defined as `1` (in the header `vers.h`). This is normally preset in the 'factory'. Please check Section 5.20 Unix, for Unix specifics.

The presumed compiler is the GNU C/C++ compiler. There are at least two compiler options files (`co-gnu.lnt` and `co-gnu3.lnt`) for this compiler in the directory `supp/linux`. But we now recommend the procedure described in directory `supp/gcc`. This is more general than those presented in the linux directory.

5.11 Microsoft Windows

For MS-DOS or Windows, FlexeLint should be compiled in either 32-bit or 64-bit mode. FlexeLint for Windows is available as an executable and is called PC-lint, (see our website for licensing and pricing). PC-lint support files are found in `supp/lnt`.

5.12 NEXT

Additional information about using FlexeLint for this system appears in the directory `supp/next`.

The NEXT operating system is similar to UNIX in regard to compiling and running FlexeLint. See Section 5.20 Unix, for additional information.

5.13 OpenVMS

Additional information about using FlexeLint for this system appears in the directory `supp/openvms`.

FlexeLint has been compiled and run using DEC's VAX-11 C compiler, using the No Optimize option.

5.13.1 Set up

It is not necessary to place FlexeLint into the system command table with DEC's "Command Definition Utility". Indeed it *should* not be placed there. Rather, establish FlexeLint as a foreign command by placing the line

```
lint :== "$ directory lint.exe"
```

in a `.com` script file where ***directory*** is the directory where the executable is located. For example:

```
lint :== "$sys$sysdrive:[user.lint]lint.exe"
```

5.13.2 Options

Options described here are placed in `co-vms.lnt` in the directory `supp/openvms`.

Since the `$` (dollar) symbol is generally recognized by this compiler and because many header files make use of it, you may want to use the option:

```
-$
```

5.13.2.1 VAX Reserved Words

Several reserved words recognized by the VAX-11 C compiler should be enabled using the `rw` (Reserved Word) option as follows:

```
+rw(globalref,globalvalue,globaldef)
+rw(noshare,readonly)
```

Two other reserved words that you may need are not built-in but may be constructed as defines. These appear in `co-vms.lnt` as:

```
-d"variant_struct=/*lint ++fab*/ struct /*lint --fab*/"
-d"variant_union=/*lint ++fab*/ union /*lint --fab*/"
```

You may also build them into FlexeLint by inserting them into `sys_def` within `custom.c` (See Chapter 6. CUSTOMIZATION).

5.13.2.2 VAX Predefined Symbols

The symbol `VAX` is predefined to be `1`. In order to match the expectations of the VAX-11 C compiler you may also want to make the following defines:

```
-dvax=1
-dvms=1
-dVMS=1
-dvaxc=1
-dVAXC=1
-dvax11c=1
-dVAX11C=1
-dCC$gfloat=0
-d_align()=
```

If you do not want `VAX` to be predefined you may use the option:

```
-uVAX
```

on the command line.

5.13.3 VAX Include Files

FlexeLint does not support the peculiar `#include` conventions of the VAX-11 C compiler. However, FlexeLint does support the ANSI convention that allows identifiers to be equated to file strings. For example

```
#define stdio <stdio.h>
...
#include stdio
```

is supported. This `define` can be done with a centrally located option such as:

```
-dstdio=<stdio.h>
```

But it is better to use `-#d` as in:

```
-#dstdio=<stdio.h>
```

as this restricts the macro to uses on `#include` lines (`stdio` is not typically a problem but, e.g., `time` is). A collection of such defines for standard ANSI header files appears in `co.lnt` in the VAX-VMS subdirectory. You will also need to have FlexeLint search a directory in which these header files exist as files (as opposed to members of a text library). For example:

```
-iSYS$LIBRARY:
```

will cause FlexeLint to search the named library.

To more fully customize the header includes you may modify function `i_open` within `custom.c`. See Chapter 6. CUSTOMIZATION.

5.13.4 VAX Return Code

The return code (the argument to the exit function) is 1 if there were no errors (or if `-zero` was selected) and `02000000000` otherwise.

5.13.5 Redirecting Error Messages

Redirecting of error messages cannot be achieved using the `>` operator as described in the text. Rather, use the option `-os(`***`filename`***`)`.

5.14 OS-9

Additional information about using FlexeLint for this system appears in the directory **`supp/OS-9`**.

You may want to place the following options on the command line or, equivalently, within your compiler options file.

`-zero`	Use an exit code of **`0`**
`-dOSK`	Predefine the symbol **`OSK`**
`-i/dd/defs/`	Compiler include files are in **`/dd/defs`**
`+rw(remote)`	add the **`remote`** keyword

`INCLUDE` path processing is supported (See Section 4.2 INCLUDE Variable).

5.15 OSF/1 and Tru64 Unix

Additional information about using FlexeLint for these systems appears in the directory **`supp/osf1`**.

These operating systems are similar to UNIX in regard to compiling and running FlexeLint. See Section 5.20 Unix, for additional information.

5.16 POSIX

POSIX is not an operating system but a standard. You can check for compliance with POSIX C by using only the header files found in the directory **`supp/posix`**. See Section 3.2 Portability Checking

5.17 Stratus

Additional information about using FlexeLint for this system appears in the directory **`supp/stratus`**.

The Stratus C compiler is unique in that it allows a **`char_varying`** type. FlexeLint does not support this type directly. If you are using this type, the simplest possible solution is to use the option:

```
-dchar_varying()=char*
```

Then declarations such as

```
            char_varying(10) x;
```

get translated into

```
            char* x:
```

But beware of a trap. That is:

```
            char_varying(10) x, y;
```

gets translated into:

```
            char* x, y;
```

which makes `y` a single `char`. Accordingly it is better to create a `typedef`:

```
            typedef char *_chvar_ptr;
            #define char_varying(x) _char_ptr
```

Place these in a header file (say `stratus.h`) and use the option `-header(stratus.h`). This will automatically include the header in each module processed.

Please note that the `char*` only roughly approximates the `char_varying(`***n***`)`. You may prefer to use an array type as a possibly closer match.

5.18 SUN OS / Solaris

Additional information about using FlexeLint for these systems appears in the directory `supp/sun`.

When compiling on the SUN, compile for Native Mode not for the System V Universe. Subtle problems have arisen when compiling under System V Universe.

For the purpose of compiling and running FlexeLint, SUN OS and Solaris may be regarded as dialects of UNIX. Please see Section 5.20 Unix, for further information.

5.19 Think C for Macintosh

Additional information about using FlexeLint for this system appears in the directory `supp/thinkc`.

For Think C we allow for the fact that it may be difficult to supply command line arguments. If no command line arguments are presented to FlexeLint, FlexeLint will act as if it were given the single command line argument "`flexe.lnt`". This specifies an 'indirect file'. You may place therein command line arguments (options, module names or names of other indirect files) on one or multiple lines. In general, "`flexe.lnt`" follows the rules for all indirect files. See Section 4.1 Indirect (.lnt) Files, in the FlexeLint Reference Manual.

We have had reports that some Symantec compilers do not support the header file `<CursorCtl.h>` which houses a declaration of the function `SpinCursor` ; called in `sys_tick()`. If this is the case, then simply create your own version of `CursorCtl.h` which includes a null define of `SpinCursor()`:

```
#define SpinCursor(x)
```

5.20 Unix

Additional information about using FlexeLint for this system appears in the directory `supp/unix`.

FlexeLint compiles in both 32 bit and 64 bit mode. If your system is LP64 or ILP64 all is well. For LLP64 systems see Section 5.1, 64 Bit.

`INCLUDE` path processing is supported (See Section 4.2 INCLUDE Variable).

The file `co-unix.lnt` in the directory `supp/unix` contains a number of options we have found useful in the Unix environment.

Any option specified in the FlexeLint User Manual may be placed in an indirect file (`.lnt` file) or in a lint comment within source code. However, on the command line some special characters are awkward to use. For Unix-like systems, several alternate characters are allowed in the specification of options. These are useful in command lines. In particular

`.` can be used for `?`
`[` can be used for `(`
`]` can be used for `)`

These may be used in addition to the standard characters. For example:

```
lint -e7.. alpha.c
```

is equivalent to

```
lint "-e7??" alpha.c
```

As another example

```
lint -od[alpha.hh] alpha.c
```

is equivalent to

```
            lint "-od(alpha.hh)" alpha.c
```

If only one subparameter appears between parentheses, you may use an '**=**'. Thus

```
            -od=alpha.hh
```

is equivalent to each of the above **-od** options.

5.20.1 -split Option

For Unix systems, the **-split** option is pre-enabled (see **sys_olist** in **custom.c**). This means that options that take arguments may be split. Thus,

```
            -dapple=1
```

is equivalent to

```
            -d apple=1
```

The **-split** option is not described in the FlexeLint Reference Manual proper as it is intended to be used by maintenance personnel and not by ordinary FlexeLint users.

Also parenthetical groupings used by some options (such as **-esym**) may be dropped so that instead of

```
            -esym(534,alpha,beta)
```

one may use:

```
            -esym  534,alpha,beta
```

Note: no spaces surround the commas.

The following options may be split:

```
-append                     -atomic                      -bypclass
-bypdir                     -byph                        -c
-compiler                   -cpp                         -d
-#d                         -deprecate                   -e
-ecall                      -efile                       -efunc
-elib                       -elibcall                    -elibmacro
-elibsym                    -emacro                      -estring
-esym                       -etd                         -etemplate
```

-etype	**-ext**	**-father**
-format	**-format4a**	**-format4b**
-format_specific	**-format_stack**	**-format_summary**
-format_template	**-format_verbosity**	**-function**
-header	**-html**	**-i**
-ident	**-ident1**	**-idlen**
-incvar	**-index**	**-indirect**
-libclass	**-libdir**	**-libh**
-libm	**-limit**	**-lnt**
-lobbase	**-maxfiles**	**-maxopen**
-message	**-od**	**-oe**
-os	**-overload**	**-parent**
-passes	**-plus**	**-ppw**
-ppw_asgn	**-pragma**	**-rw**
-rw_asgn	**-scavenge**	**-sem**
-size	**-specific**	**-specific_climit**
-specific_retry	**-specific_wlimit**	**-stack**
-static_depth	**-strong**	**-t**
-template	**-thread_unsafe_class**	**-tr_limit**
-thread_unsafe_dir	**-thread_unsafe_group_class**	**-typename**
-thread_unsafe_group_dir	**-u**	**-width**
-thread_unsafe_h	**-wlib**	**-xml**
-thread_unsafe_group_h		

5.20.2 #machine

If you need to establish a particular machine (i.e., **`#assert`** a particular machine) you may use the option **`-a#machine(`*`machine-name`*`)`**. See Section 5.8.2 Compiler Codes and Section 15.4.1 #assert, in the FlexeLint Reference Manual.

5.21 VERSADOS

The Versados variant of FlexeLint has been compiled and run with the Whitesmith compiler.

The suffix used to identify FlexeLint indirect files is **`.lt`** rather than **`.lnt`** owing to a Versados restriction on the number of characters after the period.

5.22 Xenix

For Xenix-286 the modules should be compiled using the Large memory model. The default size of **`int`**'s and **`near`** pointers is 2.

For Xenix-386 you should use the native mode (32-bit) for compiling. This will automatically yield a default size for **`int`** of 4 and a pointer size of 4. If you are using **`far`** pointers you may want to enable the Microsoft reserved words **`near`** and **`far`** with

```
            +rw(near,far)
```

Alternatively you may want to enable `_near` and `_far` or `_ _near` and `_ _far`.

The `INCLUDE` environment variable is supported (See Section 4.2 INCLUDE Variable). See also Section 5.20 Unix, for a description of alternate option characters.

6. CUSTOMIZATION

FlexeLint may be customized by modifying files `custom.c`, `custom.h`, and `vers.h`. However, modifying source code should be considered a last resort. You should first explore the numerous options which are described in Chapter 5. OPTIONS of the FlexeLint Reference Manual.

`vers.h` contains a number of compile-time switches which are ordinarily preset for a user's environment. (You might want to take the time to display this small file to confirm that the settings appear to be appropriate to your machine, operating system and/or compiler.)

Of the two remaining files, `custom.c` is more appropriate for user customization. `custom.h` is used mostly for communication purposes between `custom.c` and the remainder of FlexeLint. However, `custom.h` does have some important limit macros that you may need to modify. See Section 5.1, 64 Bit.

The descriptive material below is in approximately the same order as the information within the files that it describes; therefore, you may find it convenient to print out `custom.h` and `custom.c` and refer to these while reading their descriptions. In some cases the files may be more current than the manual and they will almost always contain valuable supplementary detail.

6.1 File custom.h

The header file `custom.h` contains:

A) System-dependent macros
B) System-dependent typedefs
C) Declarations for objects within `custom.c`
D) Declarations for objects used by `custom.c`

6.1.1 A) System-dependent macros

By the word *macro* we mean both object-like macros as in:

```
#define NUMCHARS 128
```

and function-like macros as in:

```
#define isdigit(c) ('0' <= (c) ...
```

6.1.1.1 Limit Macros

`MAXFNM`	is the maximum length of any file name (default `FILENAME_MAX`).

`M_DSTR`	is the maximum depth of define strings (i.e. macros). For example:
	`#define M1 1`
	`#define M2 M1`
	`#define M3 M2`
	`#define M4 M3`
	creates a depth of 4. The default value is 300.

`M_IF`	The maximum depth of nested `#if` statements (default 50).
`M_CONTROL`	The maximum depth of nested control structures created by nested `if`, `while`, `do`, `for` and `switch` statements (default = 100).
`M_LINE`	The maximum input line length (default = 600). (Doubled with each `+linebuf` option).
`M_INC`	The maximum number of nested include files (default = 40).
`M_STRING`	The maximum length of an internal string. (default = 4096). Internal strings are used to hold macro definitions and are used as intermediate storage while producing prototypes (see option `-od`). This internal string length can also be increased using the option `+macros`. See `+macros` in Section 5.7 Other Options, in the FlexeLint Reference Manual.
`M_NAME`	This limit places an upper bound in the length of any name.
`FSETLEN`	The number of bytes needed to represent the set of files. This places a limit in the number of files equal to `FSETLEN * 8` (or however many bits in a byte).
`M_TOKEN`	The maximum length of any indentifier, at this writing it is 200 characters.

6.1.1.2 Character Set Macros

We have tested for two character sets, ASCII and EBCDIC. In doing so we have provided predicate macros `isdigit`, etc. for testing characters for the ASCII set and have relied upon a presumed header file `ctype.h` to provide this functionality for any other character set. If you are using a character set other than ASCII or EBCDIC you will need to at least set `NUM_CHARS` (the number of characters in the character set). There may be other difficulties as well in using a non-standard character set and such difficulties should be brought to the attention of Gimpel Software.

6.1.1.3 Size Macros

It is important not to be too eager to make changes to the size macros as they pertain in most cases to the compile-time environment used to compile FlexeLint. Suppose, for example, you are compiling FlexeLint to run on a VAX, where `CHARLEN` (the number of bits in a character) is 8, and, while running on the VAX, you intend to diagnose programs that will run in some other environment where the character length is not 8. You must keep `CHARLEN 8` because FlexeLint uses this to determine its own data sizes. If you want to indicate, for example, that the target environment will have 9 bits in a byte then you should use the option `-sb9` while linting.

Macros that begin with `SZ_`... can be safely modified since they are used as default values for the size of target machine scalars. In particular, the size of a `long double` (`SZ_LDBL`) can and probably should be set here if it differs from the default setting with your compiler (assuming your compiler actually supports `long double`). But remember that it can also be set with the `-sld#` option as well. The same may be said of the size of a `long long int` (`SZ_LLI`) and of `wchar_t` (`SZ_WCHAR`). On the other hand, the sizes of other quantities such as `int`, `char`, etc. automatically default to the compile-time environment and probably shouldn't be modified. The user of FlexeLint has a right to expect that the default sizes of these quantities are the same as that of the host environment. They can always be changed via the `-s`... options.

6.1.1.4 Other Macros

The macros `ARB` and `ARBA` are used to explicitly initialize static scalars (`ARB`) or static aggregates (`ARBA`) that could have been initialized arbitrarily. That is, FlexeLint does not rely on the initial value of these objects. Some compilers produce better code for initialized data objects and some compilers actually require initialized objects. For such compilers `ARB` and `ARBA` are given an initializer of `0`. On the other hand FlexeLint is able to give better initialization diagnostics if it knows the program is not relying on the zero initialization.

The codes used as a second argument to `fopen` to read binary files (`"rb"` or `"r"`) are macroized.

The fundamental I/O routines, `sys_fgets()`, `sys_read()` and `sys_write()` are macroized to point to the similarly named routines in the I/O library, but can be redefined to be otherwise.

6.1.2 B) System-dependent typedef's

The type `MAX_INT_t` represents the type used to manipulate user values during value tracking. It ideally should hold the largest integer type available. By default it is typed `long`. If your system supports the LLP64 model you may try substituting `long long` here in place of `long`. See Section 5.1, 64 Bit. See also `supp/64bit`.

If your compiler does not support `unsigned long` or `unsigned short` or `const char*` you will want to alter one or more of the other typedefs in Section B. But most compilers nowadays support these standard features.

6.1.3 C) & D) Declarations

This region is not intended for customization but only to communicate declarations between `custom.c` and the rest of FlexeLint.

6.2 File custom.c

`custom.c` is partitioned into three groups:

	Initial Macros
(I)	Customizable Data
(II)	Customizable Functions
(III)	ANSI equivalents

6.2.1 Initial Macros

This section defines several macros used only by `custom.c`.

`SEP1`	is used to separate directory names in a path. For Unix this is `'/'`.
`SEP1A`	an alternate to `SEP1`.
`SEP1B`	another alternate to `SEP1`.
`SEP2`	is used to separate directories in a list of directories as might be specified in the INCLUDE environment variable.
`FNMAP_UC`	is 1 if filenames should be mapped to upper case. Otherwise it is 0.
`FNMAP_LC`	is 1 if filenames should be mapped to lower case. Mapping to case not only aids in presentation it also aids in file identity detection. But it should not be ON if filenames are case sensitive.
`FNMAP_MIXED`	is 1 if files can be expressed in either case, but map to a case-independent filename. Thus file `"a.h"` is the same as file `"A.H"` but the original case is preserved in messages.

6.2.2 (I) Customizable Data

`ansi_flag`

If the `ansi_flag` is ON (1) then the default startup situation is to restrict the language processed to pure ANSI. If ON, this flag will inhibit several extensions that are common in the MS-DOS community. These include keywords `near`, `far`, `huge`, `pascal`, `fortran`, `cdecl` and `interrupt`. If ON, the `//` style comment will be flagged for C modules. If you want some of these features and not others, then start with the `ansi_flag` OFF and inhibit reserved words with the `-rw` option in your version of the standard library.

`default_ext`

This is an array of strings which gives the default value for the option `+ext()`.

`dot_cpp, dot_cxx`

These specify extents which will indicate that the module is a C++ module as opposed to a C module. By default they are "`.cpp`" and "`.cxx`" (suitably case adjusted) respectively. The `+/-cpp` option can add to or delete from this list.

`dot_lnt, dot_lob`

A FlexeLint indirect file (see Section 4.1 Indirect (.lnt) Files, in the FlexeLint Reference Manual) is distinguished by a terminating "`.lnt`". Since you may add to or delete from this list of extensions using the `+/-lnt` option, you probably shouldn't modify the variable `dot_lnt`.

A filename suffix of "`.lob`" specifies a lint object module. If you need to change the suffix, you must do it here.

`sys_id` and `sys_vers`

The first line of output produced by FlexeLint is of the form:

Identification Version `Copyright Gimpel Software`...

The variable `sys_id` specifies *Identification* and the variable `sys_vers` specifies *Version*.

`aoc_plus, aoc_quest, aoc_quote, aoc_lp, aoc_rp, aoc_cm, aoc_star`

These variables are Additional Option Characters. They do not replace but may be used instead of `'+'`, `'?'`, `'"'`, `'('`, `')'` `','` and `'*'` within options. For example, in Unix, the corresponding additional option characters are, respectively, none, `'.'`, none, `'['`, `']'` none and none. This is to assist the programmer in placing options on the command line where `'?'`, `'('`, and `')'` are meta characters. For Unix, therefore, `-e7..` has the same meaning as `-e7??`.

`sys_olist`

`sys_olist` is a string bearing a list of options. These are pre-evaluated before any argument and before the help screens are displayed.

`sys_odesc`

`sys_odesc` is the array of strings which describes the options. If you are modifying options you may also want to alter this help information. This information is printed as the result of entering the lint command without arguments or by specifying `?` on the command line.

`scr_ht, scr_wth` and `scr_n`

These variables have to do with printing help information (see `sys_odesc` above). When FlexeLint is invoked on a line without arguments (`argc == 1`), several screens of help information are fed to the user. The functions that produce this information rely on `scr_ht` and `scr_wth` to determine the height and width of the screen. `scr_n` is a variable that keeps track of the number of lines written to the screen. If you don't want the help information broken up, set `scr_ht` to 1000.

6.2.3 (II) Customizable Functions

`i_open()`

All file opens for reading are done through the function `i_open()`. At one time, `i_open()` was used only for opening include files (hence the name) but now it is used for opening modules (files on the command line or within an indirect file) as well. This means that the search for modules, by default, employs the same logic as the search for header files, using the usual search path techniques. By modifying `i_open()` you can alter the search path for modules as well as include files. For example, if you want to restrict the search for modules to the current directory then, as the commentary to `i_open()` suggests, you should return `NULL` if the `parent` is `NULL` (signifying a module) and the `ordinal` is other than `1` (signifying that we are retrying from a prior failed attempt to open the file).

One reason for providing **`i_open()`** is that no C standard can prescribe the effect of a **`#include`** statement owing to the great variation in file-naming conventions and directory organizations found in systems supporting the C language.

main()

The **`main()`** function is provided within **`custom.c`** for a number of reasons. The interface presented to **`main()`** by the operating system may not be in the usual **`argc,argv`** form specified by K&R. In that case it will be necessary to construct that form and provide it to **`sys_main()`**.

Another reason is that it is possible to support an enhanced set of options through the adroit manipulation of the argument vector passed to the **`main()`** program. To take just one simple example, suppose that the convention in a particular environment is that a slash question-mark on the command line should result in the display of options for a command. The customizer can search through the arguments to FlexeLint and if it finds this sequence, it can convert it into a single question-mark.

Much more elaborate schemes can be imagined whereby the options accepted by **`main()`** are transformed into the standard options accepted by FlexeLint.

scr_need()

This function is called in conjunction with **`scr_out()`** to make sure there are *n* more lines in the output screen. See also **`sys_odesc`**.

scr_out()

The function **`scr_out()`** is called to output help information (the information output when a user supplies no arguments on the command line). This function is provided so that the output format may be altered to a form more appropriate to that expected by a particular installation (using, perhaps, windowing software).

scr_param()

`scr_param()` is called early on and just once. It is a chance to set the screen parameters (**`scr_ht`**, **`scr_wth`**) from run-time considerations. Otherwise they default to static initializations.

`sys_arg()`

`sys_arg()` allows the customizer to modify arguments (either filenames or options) before passing them to FlexeLint.

`sys_def()`

Some C compilers have pre-defined names (just as FlexeLint pre-defines `_lint`). These can be defined in the function `sys_def()`. To define a name, call the function:

```
            scan_sym( name , value );
```

following the examples already present in `sys_def()`. Note, however, that the `-d`... option on the command line or within a `co.lnt` file will have the same effect.

`sys_dircmp()`

This function is called to compare two directory names. This differs from a filename compare because directory names can have `SEP1`'s optionally appended to them.

`sys_env()`

Some operating systems support the notion of an environment variable, whose value can often be obtained by a program via a call to a built-in function called `getenv()`. For example,

```
            SET INCLUDE=...
```

is often used to identify a sequence of directories in which to search for include files.

FlexeLint calls upon `sys_env()` when it wishes to determine the value of `INCLUDE` or when it wants to determine the value of the `LINT` environment variable (see Section 4.3 LINT Environment Variable and Section 4.2 INCLUDE Variable).

`sys_putenv()`

`sys_putenv()` takes a directive of the form "***name=value***" and assigns ***value*** to the environment variable name.

`sys_env_var()`

This function is called to expand environment variables that appear on the command line or within an extension of the command line such as within an indirect file. The environment variable name must be surrounded by special characters. For example `%ALPHA%`, on MS-DOS, refers to the `ALPHA` environment variable. Though `sys_env_var()` is called for each option, the environment variable may be embedded within the option as in `-i%`***`directory`***`%`.

`sys_exit()`

As is indicated in the commentary to `main()`, `sys_main()` does not return but rather it calls the function `sys_exit()` which is expected to exit back to the system with, possibly, a return code.

`sys_expand()`

The function `sys_expand()` is called when FlexeLint needs to know if a name contains wild-card characters (e.g., in the name `*.c`,`'*'` is a wild-card character). In Unix operating systems, wild-card expansion is automatic (is done by the shell) but in other systems it must be done for each program that supports wild-card expansion. For example, assume that there are three files in the current directory matching the pattern `a*.c`, viz. `a1.c`, `a2.c` and `a3.c`. Then the sequence of calls will be:

Call	Returns	Interpretation
`sys_expand( "a*.c" )`	`"a*.c"`	This is expandable
`sys_expand( NULL )`	`"a1.c"`	First name
`sys_expand( NULL )`	`"a2.c"`	Second name
`sys_expand( NULL )`	`"a3.c"`	Third name
`sys_expand( NULL )`	`NULL`	No more names

`sys_fattrib()`

`sys_fattrib()` returns file attributes. The file attributes for our limited purposes include at least the following flags.

`fatt_ORDIN`	=> ordinary file
`fatt_DIR`	=> directory

This is a wrapper function around a system call. For example, on POSIX systems it results in a call to `stat()`.

`sys_fgets()`

To obtain a line of input, FlexeLint calls `sys_fgets()` which is normally defined to be the library function `fgets()`.

`sys_i()`

`sys_i()` is used to tack on a local directory separation character to the `-i` option. Thus for Unix, `-i/usr` becomes `-i/usr/`.

`sys_include()`

The function `sys_include()` will convert the value of the `INCLUDE` environment variable into a sequence of file prefixes that it passes to the function `add_search()`. For example if the `INCLUDE` environment variable contains the value:

```
"/usr/include;/usr/include/sys"
```

and if the `SEP2` symbol (#defined within `custom.c`) has the value `';'` then this string will be decomposed into two strings representing two directories. If the directory prefix string (`SEP1`) is `'/'` then the two strings passed to `add_search()` will be:

```
"/usr/include/"
"/usr/include/sys/"
```

`sys_fncmp()`

This function is called to compare two filenames for equality.

`sys_fnfold()`

This function will fold a mixed-case name into a mono-cased name for comparison purposes.

`sys_fnmap()`

This function is called to map filenames (and directory names) to upper or lower case depending on the status of the macros `FNMAP_UC` and `FNMAP_LC`.

`sys_fnorm()`

This function is called to normalize a filename.

`sys_fnsimple()`

This function returns the filename provided as argument minus any directories that may be part of the name.

`sys_option()`

This function is called whenever an option is to be processed. Normally the option is passed directly into FlexeLint. This gives the programmer an opportunity to change things.

`sys_pathname()`

`sys_pathname()` returns in an argument, the full path name of a file. This usually amounts to prepending the current directory if this is not already a full name. The returned name may have redundancies (e.g. `"/x/../y/z" == "/y/z"`). File normalization is done elsewhere (`sys_fnorm`).

`sys_pragma()`

This function is called for each `pragma` provided the first such call returned 1.

`sys_tick()`

This appears to be an innocuous little function; it is called periodically but for most systems it does nothing. Actually this function could give you the "hook" you need to run FlexeLint in a windowing environment. At `sys_tick()` you could check to see if any messages are outstanding and return when they are processed.

`sys_truncate()`

`sys_truncate()` will truncate a filename to an 8x3 name. As a peculiarity, the names comprising the path are not truncated, nor is the extension.

`sys_vmsg()`

The function `sys_vmsg()` can be used to partially customize the verbosity message. A typical verbosity message produced by FlexeLint is:

```
---Module:    alpha.c
```

If the verbosity option had specified a storage report as in: `-vms` then the verbosity message would resemble:

```
---Module:    alpha.c, consumed 9696 bytes
```

It is the latter portion of the verbosity message that is customizable. For example, if the operating system provided sharp limits as to how much storage were available, the storage report could indicate how many bytes remained rather than how many bytes were so far consumed.

6.2.4 (III) ANSI Functions

The functions in this section of `custom.c` (at this writing `memcpy()`, `memset()` and `strchr()`) are provided in the event that your compiler does not support them. In general the compiler-provided versions of these functions are likely to be more efficient and are to be preferred if available.