Questions and Answers

The following is a list of questions asked most often by **Let's C** users, and suggestions for solving problems. If you have a problem with **Let's C**, look here first.

Programming problems

Why doesn't **cpp** *execute?* **cpp** *execute?*'>=29

Most likely, **cc** cannot find **cpp** because it is in another directory and you did not tell **cc** where to look for it. If this is the case, use the **-xc** option in the **cc** command line, as described in the Lexicon entry for **cc**. Also see the sub-section *Setting the environment*, in section 1 of this manual.

Can I keep the compiler and source code on separate disks?

Yes, when you use the **-x** options on the **cc** command line. See the description of these options in the Lexicon entry for **cc**. Also see the sub-section *Setting the environment*, in section 1 of this manual.

My program won't read a carriage return from a file. Why? read a carriage return from a file. Why?'>=29

When you open a file stream, by default it is opened in ASCII mode. A file stream opened in ASCII mode will handle only alphanumeric characters plus the newline character '\n'. All other characters, including the carriage return character '\r' will be dropped from the file stream. To read a file that contains a carriage return or other non-alphanumeric characters correctly, open the file in binary mode.

To read from a binary file you must open it in binary mode; for example:

```
fopen("filename", "rb")
```

For more information, see the entry for **fopen** in the Lexicon.

My automatic large array is corrupted. Why?

Most likely, you did not allocate enough stack when you compiled your program. **Let's C** by default sets aside two kilobytes of memory for stack, but your program may require more. To increase stack size, use the **-ys** option to **cc**, or make the array static by moving it outside of the body of the program. The **-ys** option takes the number of bytes in decimal; for example, **-ys 10000** gives you 10,000 bytes worth of stack.
Note that using too much stack space can itself cause other unpredictable results to appear.

Can I reduce the size of my compiled modules?

Yes. A number of techniques will save space in your programs. For example, try making automatic variables into register variables. This also increases the speed of execution. Use register variables only for heavily used data items. Normally, the compiler uses registers SI and DI for intermediate work. The first register variable you assign uses SI, the second uses DI. After that, the **register** typing is ignored.

Another technique is to perform repetitive function calls by means of a table definition. This will eliminate the code needed to make each of the function calls except one. Also, try removing the modules' symbol tables with the command **strip**. This will reduce their size significantly.

If your program does not use any STDIO routines, you can compile it with the **-ns** option.

The order of evaluation is not what I expected. Why?

Note the following passage from *The C Programming Language*: "C, like most languages, does not specify in what order the operands of an operator are evaluated .... In any expression that involves side effects, there can be subtle dependencies on the order in which variables taking part in the expression are stored. One unhappy situation is typified by the statement

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

"The question is whether the subscript is the old value of **i** or the new. The compiler can do this in different ways, and generate different answers depending on its interpretation... The moral of this discussion is that writing code which depends on order of evaluation is a bad programming practice in any language."

printf *gives incorrect output when rounding numbers. Why?*

For example:

```
printf("%6.0f", (double) 9/10.0);
```

yields 0 instead of 1. This, however, represents a misunderstanding of how **printf** works. The instruction **%6.0f** tells **printf** to truncate the value and print it, not round it; because 9/10 is less than 1, the output is zero, not one.

Where does **make** *look for* **mactions** *and* **mmacros**?

mmacros and **mactions** are files of preset macros and definitions that **make** uses by default. **make** looks for them first in the directories named by the environmental variable **LIBPATH**. If this variable is not set, it then looks in directory **lib**; if there is no directory with this name, it finally looks in the current directory.

My Tandy 2000 cannot access my printer with **lpt1**. *Why?*

The Tandy 2000 is not fully IBM-compatible. One way in which it differs from the IBM PC is that it cannot recognize the logical device **lpt1**. Use **prn** to access the printer on this machine.

What does the error message **temporary file write error** *mean?*

Let's C is a multiphase compiler in which each phase performs a different task. Because each phase stands alone (as a single program), it must write its output to a temporary file that is read by the following phase. Thus, this error means that a phase could not write out its temporary file. This is usually due to a hardware problem such as a full disk, or a write-protected disk. To overcome this problem, it might be necessary to write temporary files to another directory or to another disk drive.

What does the error message **lvalue required** *mean?*

Your program uses a constant where it should use a variable. A *variable* is the name of any data element whose value can change; for example

```
int foo;
```

declares the variable **foo**. A *constant*, on the other hand, is any number or fixed address. The name of an array, for example, is a fixed address and cannot be altered. The code

```
int foo[];
int *bar;
    ...
foo = bar;
```

will generate an **lvalue required** error message, because the name of an array is a constant rather than a variable. On the other hand, the code

```
int *foo;
int *bar;
   ...
foo = bar;
```

will not, because both **foo** and **bar** are pointers and, therefore, are variables. See the Lexicon entries for **lvalue** and **rvalue** for more information.

What does the message **Identifier** *string* **is being redeclared** *mean?*
If you use a function without declaring it, **Let's C** assumes that it is an integer. If later in your program you declare that function to be something other than an integer, your declaration will clash with the implicit declaration you made earlier, and so trigger the error message. You should check that your functions do not contradict themselves. It is a good programing practice to declare explicitly all functions and variables your program will use.

What does the error message **out of space** *mean?*
Most likely, you created a function that is too large for the compiler to process. Break the routine into smaller components.

If this error is generated by **cc2**, try recompiling your program with the option **-vcc2l**. This tells **cc** to use a LARGE-model version of **cc2**; this version runs more slowly than the normal SMALL-model version of **cc2**, but can handle larger programs.

What does **Bad value in debug** *mean?*
This error occurs only when you have used the **-VCSD** option, so that you can debug your program with **csd**. Your program probably declared a pointer to a structure tag that does not exist. Check the declaration of the structure or pointer.

How can I estimate how much stack I need?
Automatic variables and passed parameters go on the stack. Register variables do not go onto the stack. Each level of call requires eight bytes in SMALL model, and 10 bytes in LARGE model. The runtime startup routine needs about 200 bytes of stack, and **printf** about 100 bytes.

The default stack size is two kilobytes (2,048 bytes). To change it, use the **-ys** option; for example, to compile a program with 4,096 bytes (four kilobytes) of stack, use the following command:

```
cc -ys 4096 example.c
```

How do **execall** *and* **system** *differ?*
execall sends a command and its list of arguments, or "tail", directly to MS-DOS; **system**, on the other hand, sends a command through **command.com**.

execall looks for the executable file, loads it, executes it with the given tail as its arguments, and returns its exit status code. Thus, it only works if command exits to its caller rather than by executing the MS-DOS warm boot. MS-DOS built-in commands, such as **dir**, do not work with **execall** for this reason. **system** passes a command line to **command.com**, loads it, and executes it as if it had been typed at the MS-DOS command level. **system** can be used with the MS-DOS built-in commands, as well as with commands that rely on MS-DOS to parse the command line into the formatted parameter area. Note, too, that **system** runs more slowly than **execall**, and it cannot pass to the calling program what the called program returned upon exiting.

See the Lexicon entries for **execall** and **system** for more information and for example programs that use these routines.

What does the runtime startup routine do?

This is a routine that is linked with a C program as the first part of the executable object program. It initializes the stack and saves information necessary to return to the calling program, and calls the C library function **_main** to parse the MS-DOS command tail into the arguments **argv**, **argc**, and **envp**, which are expected by the C program.

How can I make ROMable code?

Use the following steps:

1. Use the option **-VROM** to move constant strings into the code segment.
2. Put the data segment into ROM, copy it to RAM, and have the data segment point to where it is in RAM. This must be done explicitly, i.e., you must write a new runtime startup routine.
3. STDIO routines are linked into a program even if they are not required. Use the **cc** option **-ns** to exclude them from your program. This also gives the program a different version of the **exit** command, which does not call **fflush** or **fclose**.
4. Tools for converting to Intel hex format and for burning PROMs must be purchased from third-party vendors. Mark Williams Company does not supply them at present.

How can I declare an array of (row)(col) elements?*

Declare and initialize it to **Array[row-1][col-1]**. The first element of the array is **Array[0][0]**.

How can I redirect error messages into a file?

Use the greater-than sign '>' with MS-DOS. For example,

```
cc filename.c > errfile
```

will work for one file. For multiple compiles, say:

```
cc file1.c > errfile
cc file2.c >> errfile
cc file3.c >> errfile
```

This appends the error messages from subsequent compilations onto the error file.

The **-A** option to **cc** automatically redirects error messages into a buffer, and invokes the MicroEMACS editor so you can fix your source file "on the spot". You may find this to be more convenient than redirecting the error messages into a file. For more information on this option, see the Lexicon entry for **cc**.

How can I redirect an object file to another directory?

The option **-o** *filename* redirects the object file into *filename*, whereas the option **-xo** *directory* redirects it into *directory*.

How can I build pointers for segment and offset functions, like **copy***?*

Use the function **ptoreg** to convert C pointers to processor register pairs. **ptoreg** converts a pointer *p* relative to segment *seg* and stores the resulting segment:offset pair in the register pair *segreg:offreg*.

The functions **csreg**, **dsreg**, **esreg**, and **ssreg** return the current segment register values. To turn a register pair into a C pointer, use the function **regtop**.

Can I compile a program from within MicroEMACS?

Yes. Use the -A option to the **cc** command line. If an error occurs, you will be returned to MicroEMACS automatically, with your source code displayed in one window and the compiler's error messages displayed in the other. When you have corrected the problem, exiting from the editor with either the **<ctrl-X><ctrl-S>** or **<ctrl-Z>** automatically recompiles your program.

Problems with running programs

My data are being corrupted inexplicably.

My computer is hanging.

My program is generating garbage.

These problems may have a number of causes; the most likely is improper allocation of space. Often, the stack size is too small. If the stack grows too large for the space that has been allocated for it, it will invade and corrupt the static data area. The default value for stack size is two kilobytes (2,048 bytes), which is large enough for most functions; however, but highly recursive functions (such as **qsort**) or programs that use large automatic arrays (such as the sample program on page 29 of *The C Programming Language*, ed. 2) will quickly exhaust the available stack space. There is no way to increase the size of the stack while a program is running. To increase the stack, you must relink the program and allocate more stack by using the **-ys** option to the **cc** command. For more information, see the Lexicon entries for **cc** and **stack**.

Another common cause of data corruption is using a pointer without allocating space for the object to which it points. This is called an *uninitialized pointer*. For instance, if your program declares the variable **str** to be a pointer to a **char**, you cannot assign data to **str** unless you ensure that **str** points to a place that can hold these data; otherwise, the results will be unpredictable. You can make sure that a pointer works correctly either by **initializing** it, or by allocating space with **malloc** or **calloc**.

Another cause of this problem is passing a function the wrong number or type of parameters. Be sure that all functions have the correct number of arguments, and that all arguments are of the correct type.

My output is not going to the screen as I expected.

MS-DOS buffers output to the console, and does not print it until it gets a newline character. You can flush out the buffer whenever you want by using the function call **fflush(stdout)**, or you can use the Mark Williams functions **getcnb** and **putcnb**, which go directly to the console.

getcnb *doesn't work right. Why?* work right. Why?'>=29

Problems will arise when you combine **getcnb** with **printf** or any other normal STDIO function. **Let's C** follows the UNIX protocol, and buffers all of its STDIO functions. For example, when you create a **printf** string, it waits in a buffer until something, such as a newline character or a **fflush** instruction, pushes it out of the buffer and onto your screen. Thus, if you use a **printf** call to print a prompt string, then use a **getcnb** call to get the user's response, you will not see the prompt until you type the carriage return in response to the **getcnb** call. To solve this problem, either use **putcnb** to display the prompt, or follow the **printf** call with **fflush(stdout)**.

How do I clear the screen?

The easiest way to do this is to use the appropriate escape sequences defined in the file **ansi.sys**, provided it is loaded by **config.sys**. You can also call the MS-DOS function that clears the screen. See the Lexicon entry for **ansi.sys** for more information.

Can I open more than the default number of files at a time?

Yes. Simply insert the instruction

```
FILES=n
```

into the file **config.sys**, where *n* is the number of files you want to be able to open at any given time. Because of the way MS-DOS is designed, no more than 20 files can be opened by a program at any one time; this limit *includes* **stdin**, **stdout**, **stderr**, **aux**, and the printer. Make sure that **config.sys** is on your boot disk, and then reboot your system.

Can I call any MS-DOS function or interrupt?

The function **intcall**, which is described in the Lexicon, provides a general interrupt calling routine. See the Lexicon entry for **interrupts** for the number of the interrupt you need to hand to **intcall**, the number of the function you wish to call, and any other information the function requires. Also read the header file **dos.h**, which defines constants for most of the interrupts and function numbers.

For a summary of how to handle interrupts, see the Lexicon entry for **interrupt handling**.

How can I position the cursor?

The easiest way is to use the escape sequences listed in the file **ansi.sys**. See the Lexicon entry for **ansi.sys** for more information. MS-DOS interrupt 10 can also be used to move the cursor. The MicroEMACS editor uses interrupt 10, and its source code (which is included with **Let's C**) demonstrates how to use this interrupt.

Can I link my **masm** *routines with* **Let's C** *output?*

Yes, as long as you observe **Let's C**'s linkage conventions. For more information, see the Lexicon entry on **calling conventions**.

The command **fixobj** lets you edit object modules. With **fixobj**, you can edit modules compiled or assembled by other language tools so that they can be linked with programs generated by **Let's C**. For more information, see the Lexicon entry for **fixobj**.

Where does **Let's C** *put things in memory?*

See the entry on **memory allocation** in the Lexicon.

Limitations in i8086

What are the limits on the size of arrays?

Let's C does not limit the size of an array; however, the architecture of the Intel i8086 microprocessor is such that it forbids the creation of a data structure that is larger than 64 kilobytes.

SMALL-model limitations

Programs are limited to 128 kilobytes of code and data combined. Within the 128 kilobytes, the following limitations apply:

- No program can have more than 64 kilobytes of code.
- No program can have more than 64 kilobytes of data.

Data includes stack (automatic) data, static data, and dynamically allocated memory.

LARGE-model limitations

Programs are limited to one megabyte of code and data combined. Within the one megabyte, the following limitations apply:

- No module can have more than 64 kilobytes of code.
- No module or library can have more than 64 kilobytes of static data.
- The stack size cannot exceed 64 kilobytes.
- No individual data structure can exceed 64 kilobytes.