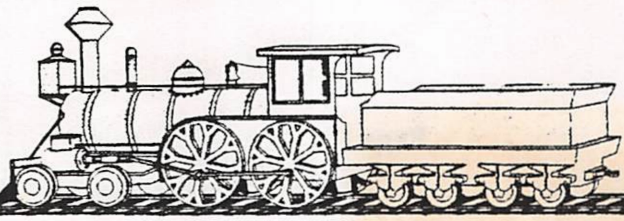
MARCH  1994

The 6809 EXPRESS

OFFICIAL PUBLICATION
OF THE PENN-JERSEY
COLOR COMPUTER CLUB

CONTENTS

THE LIBRARY CAR

by Alan J. Wagner, Sr.

Welcome once again to the 6809 Express library car. The last two times we discussed a short "C" program and you were to attempt to get it to compile. I will assume that you met with some success. This time we will proceed by discussing the five basic variable types available in "C". We will find later that some of them can be expanded upon by the addition of a modifier, but for now we will stick to the basics. We will also need to expand a little on the printf() function from the standard library. This function has many features that we will be seeing as they need to be brought up. For the present, accept them as they show up. We will need to have a session where printf() is all we discuss to tie everything together, but that will be a session or two from now.

TYPE	MEANING	KEYWORD
Character	Character data	char
Integer	Signed whole numbers	int
Float	Floating-point numbers	float
Double	Double precision floating-point numbers	double
Void	Valueless	void

Unlike some languages, in "C" all variables must be declared before they can be used. The most important part of declaring a variable is that we are telling the compiler just what kind of data we are going to store in that variable. This allows the compiler to allocate an appropriate sized chunk of memory to hold the data. Each of the different types may use a different amount of memory. I say "may", because each compiler on each different type of computer is free to represent the data in memory almost anyway the designer of the compiler sees fit. Some types may in fact use the same amount of memory. As an example, the compiler I use on my Coco 3 uses 1 byte for char, 2 bytes for int, 4 bytes for float, and 8 bytes for double. (void was not yet being used when my compiler was written. An ANSI preprocessor handles changing it to something my compiler can handle.) In addition to the above my compiler also has two more non-basic variable types. They are: unsigned integer using 2 bytes and long integer using 4 bytes. Don't worry about what each does for the moment. I mention them only to illustrate that there are two integer types that use 2 bytes and one integer type that uses 4 bytes as does the float type. It is important to keep in mind that once a variable is declared as a specific type, we should not attempt to store/retrieve an incorrect type in/from it. If one would do so without first taking special precautions, some very strange and unexpected results can be obtained.

Let's look at each of the basic types in turn. The char type stores characters such as would be expected to be printed on a printer or the monitor. Because char is usually a one byte variable, on most systems there can be 256 different symbols stored in them. The first 128 of them are defined by the ASCII character set, if that code is used by your system. (Most do use the ASCII set but not all. Check in the docs for your system to see what it uses.) With some compilers the type char can also be used to store whole numbers from zero to 255. For reasons of portability this should be avoided.

Next is the signed integer. An integer is assumed signed unless specifically called as unsigned. It typically is a two byte or 16 bit variable. A signed 16 bit integer can have a value from -32768 to 32767, but may not include any fractional part. That is, an integer may hold only whole numbers. If the integer is specifically called as unsigned, it may hold numbers from 0 to 65,535. Some systems have modifiers that can be applied to type integer such that it may hold values, as unsigned, as high as 4,294,967,295! To know the values of which your system is capable, you must check your manual.

A float type variable can contain a decimal or fractional part. A float number is typically a 32 bit quantity. It is expressed in scientific notation and if it is a 32 bit quantity, with a range from 10 to the negative 38th power to 10 to the positive 38th power. The number of significant digits can vary from 2 to 16, but is usually 6.

A double is, as the name implies, a variable that is double the size of a float. It is a floating point number in that it can have a decimal or fractional part. It may have double the significant digits as well as double the range.

Void is a valueless variable. You may ask of what use is a variable that has no value? A function must be declared as being of some data type and whether or not it takes arguments when it is declared in ANSI C. This allows ANSI compilers to keep watch over the program during compiling to make sure that you as the programer don't ask for a value to be returned from a function that shouldn't be returning one or that you don't try to pass a value to a function that shouldn't get one. If a function is not to return any value or if it is not to take any argument values, how would you declare it?

Now that ANSI has provided us with a data type that has no value, the answer is easy, use type void! Void also has another use when used to describe a pointer, it generates a pointer capable of pointing to any other type. Don't worry about what a pointer is just yet, but it is one of the more powerful features of the C language and will be discussed at length in future articles.

Now that we've discussed some data types, let's take a look at a program that uses some of them.

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300 */

main(void)
{
   int fahr, celcius;
   int lower, upper, step;

   lower = 0;      /* lower limit of temperature table */
   upper = 300;    /* upper limit */
   step  = 20;     /* step size */

   fahr = lower;
   while (fahr <= upper) {
      celsius = 5 * (fahr-32) / 9;
      printf("%d\t%d\n", fahr, celsius);
      fahr = fahr + step;
   }
}
```

This is a program for those that want to start learning metric temperature measurement. It will print out a table consisting of two columns of numbers. The first will be the temperature in Fahrenheit, the second will be the temperature in Celsius. Admittedly, this program isn't very fancy. But, let's take a look at it and then later we can spruce it up a bit.

First off, what's with the verbiage between the /* and */ marks? These are comments. They can be put anywhere a blank or newline character could appear in a C program and the compiler will ignore anything between the /* and */ marks. A comment could even be put in the middle of a statement or formula, but that is not good practice because though it won't confuse the compiler, people are more easily distracted. Comments are generally put at the end of a line or in block by themselves. Use them more often than you think you need them to explain what you are trying to do in this or that part of the program. Two years from now when you look back on your project, you'll be scratching your head wondering just exactly why you put that statement in your program at that spot. A comment now could save hours later trying to reinvent your code.

Another use for comment marks is to remove a portion of the program. Let's say you are having a problem getting a portion of your program to count something correctly and it seems to be exiting a loop too soon. You could put in a line that prints the loop count to the screen with each pass to help follow the flow of the program. To remove it, just enclose the printout portion of the program with comment marks and the compiler will ignore it. BUT, it will still be in your source code to remind you where there was a trouble spot, waiting to have the comment marks removed and be sent back into service if a problem ever arises there again.

However, be cautioned, comments can be a two edged sword. If you open a comment and then don't close it properly, you could comment out the rest of your program. If suddenly part of your program seems to have disappeared, check to see that all comments have been opened and closed correctly. For instance, if your program had a fragment like this:

```
   lower = 0;      /* lower limit of temperature table
   upper = 300;      upper limit
   step  = 20;      step size */
```

The upper = 300 and step = 20 have become part of the comment and would not compile. This could cause the program to give a wrong result. The proper way to comment is shown in the original program above.

The lines "int fahr, celsius;" and "int lower, upper, step;" are declarations. A declaration announces the properties of a variable and must occur before a variable is used. The declarations usually occur at the beginning of the function in which they are used. "Special effects" can be obtained by declaring a variable elsewhere, but let's not confuse the issue just yet.

Now we actually get into manipulating some of our variables. The next four lines contain assignment statements. "lower" is set equal to zero, "upper" is set equal to 300, "step" is set equal to 20 and "fahr" is set equal to "lower" which makes it equal zero. These statements are important in C as the compiler does not assign any particular value to a variable, only a memory location. There could be anything left over in that location from the last program that was assigned there. After being declared, ALL C variables must be initialized to the value you need them to have going into the program. Do NOT assume they will be zero. You may run the program several times and the variables just happen to find a location that was zero and it works perfectly. Then just as you think you have this program running flawlessly and call in a friend to see how great it is, it crashes. Thus, proving Murphy's Law is alive and well and operates perfectly on C variables not properly initialized.

Next we come upon a new statement in C, the while conditional loop. The command while is followed by a condition to be met stated within the parentheses. In this case, that fahr is to be less than or equal to upper. If this condition is met, then the body of the while loop is executed. The body of the while loop can be one or more statements enclosed in braces, as in the temperature conversion program, or a single statement without braces, as in the following fragment.

```
while (x <= y)
   x = x + 2;
```

Before we go any further let's stop for a moment and discuss style. No, not which type of jeans are in this year or which tie you should wear with that new suit, but program entry style. Note how in the programs we have looked at so far there are portions of them that are indented. This is to show us humans which parts go together with what other parts. For instance, everything in the body of the while loop is indented between the braces that define its ends. This allows one to see at a glance what goes with the while. These indentations don't mean anything to the compiler and are ignored by it, but make life much easier on us humans. That having been said, let's carry on.

Those of you who remember the conversion formula from Fahrenheit to Celsius may wonder why the formula is 5 * (fahr-23)/9, when it is usually expressed as (fahr-32)*(5/9). The reason is that in this program everything is integer math. With 5/9's being less than one, it would be truncated to zero and all the answers to the formula would be zero! By multiplying the result of (fahr-32) by 5 first and then dividing by 9, the formula is not changed, but the integer math comes out correct.

Next we see the printf() function call. That's right, printf() is a function defined in the standard library as defined by the ANSI Standard. There is really no input or output that is part of the C language. This is one of the reasons C is transportable to so many computers; each implementation must define its input and output via the standard library functions defined by ANSI and written to access the hardware upon which each implementation is destined to operate.

This call of printf() introduces some new formating arguments. The %d's tell printf() to expect decimal numbers. (That is in contrast to float, double, hex or octal, which printf() can also accept. I know that can be confusing because we said this is all integer stuff and integers can't have a decimal.) The \t is a tab request. While the \n is the newline we saw in the first program. One thing to keep in mind when using printf() with % constructions, each construction requires a corresponding second argument that matches the expected type and number of constructions. As an example, in our statement there are two % constructions calling for decimal (integer) values. In the latter half of the parenthetical part of the statement, two integer variables are supplied, thus fulfilling the requirements.

The next line should look fairly familiar. Here we are incrementing the variable fahr by the value of the variable step. This is identical to how it would be done in BASIC.

Next we come to an indented closing (or right) brace. This is the end of the block of code that makes up the while loop. Encountering this brace, the program goes back to the while statement itself and checks the conditional part of the statement and if true, executes the loop again. The last line of the program has another closing brace. This matches the opening brace right after the opening statement of this program, main. Please note that even though our indenting lines up and helps us to see which braces go together, the compiler couldn't care less how far a brace is indented. It simply matches an opening brace with the next closing brace it comes upon using the method of last in first out. A missing opening or closing brace can give the program real fits and generate long lists of errors as the compiler attempts to make sense of the code.

This brings us to the end of this session. Next time we will make this program a little more professional looking. Even though it will work as it is, there are ways to say the same thing in shorter and faster ways. 'Till then, Happy computing.

SYSOP REPORT 3/1/94

by Rick Hengeveld, Sysop

The Maverick BBS has logged in excess of 1000 calls. We also are currently carrying over 240 files for download. Recently a few on-line games were added and these are proving to be very popular, stop in and try your skills against the other users!

In the last 6809 we discussed the various features of VBBS. I hope that will help those that had both questions and problems using the system. This month we'll look into downloading files from the BBS. This is the cheapest and easiest way of getting new files for your Coco or MS-DOS machine. Some have been scared off of this, thinking that all this data transfer is to complicated. Once you have the hang of it downloading is very simple and a great help to your software library.

Due to the various types of terminal software in use it is very difficult to give a step by step procedure to the end user.(you) So I'll attack the subject in general terms. First once your on line and have moved to the transfer section you can view the different software directories via the "C" change directory command. With over 200 titles available it is necessary to categorize the available programs to assist the user in finding those files that would interest them. There are 3 areas each for RS-DOS, OS9 and MS-DOS users. These are divided into entertainment, productivity and utility sections. Once you've selected the directory your interested in (just follow the prompts) you can view the files offered for download. To list the files you select "L"ist from the main prompt. You will then be presented with a prompt that says "File Mask" What this is asking is "Do you want me to list only certain files?" Simply pressing enter will show you all the available files in the current directory, typing *.rep would show you only files with a "REP" extension. This is simply another way to narrow down a search for a specific file. Those that are new to downloading should simply press enter at the File Mask prompt and list all the files in the current directory. This will give you a better overview of what is available. The system will scroll up a single screen full of files and then pause so you can study the list and choose any files you may want to receive. At the bottom of the list you will see a prompt asking you to hit enter or if you wish to place any of the files in a batch, more on batches later. Pressing enter will list the next screen full of file that are available in the current directory. Most Coco users should write down the complete names of any files they wish to receive during this listing operation, after selecting these files they can then start a download by selecting "D"ownload from the main menu. After pressing "D" you will again see the File Mask prompt. This time enter the name of the file you wish to receive. The system will locate the file and ask which transfer protocol you wish to use. This is the function that confuses many users. Here is the simple rule for Coco users. Select DSZ - Xmodem. All Coco terminals support Xmodem. So what's all this protocol stuff about? Protocols are simply a method to check that the data being sent over the phone line was in fact received and was correct. To do this check both systems have to do the error checking using the same method. Only a few Coco terminal software packages support Ymodem batch and very few if any support the Zmodem protocol. Most MS-DOS terminals support the Y & Zmodem protocols. These protocols are both batch type protocols. So what is meant by "Batch"? Lets go back to the file list where you can select the files you wish to receive. At the end of each page of displayed files you are asked if you wish to "B"atch any of the listed files. By selecting batch the system will tag those files and set them aside for you. After making all your selections you can start a download of all the files you selected. In this way you can download a multitude of files while leaving your terminal unattended! To accomplish this you need to select a protocol that supports batch. Both DSZ Ymodem and DSZ Zmodem support this batching mode. These protocols work much like Xmodem however unlike Xmodem these correction methods also send and write the file names to your directory. In this way multiple files are sent in one shot.

No matter what terminal software you use, The Maverick's prompts and keyboard inputs are the same, however telling the BBS what you want sent and the protocol to send the file in is not enough. You must prepare your system to receive the file that is sent. As I stated near the beginning of this article it is difficult to give a step by step procedure since every users terminal software will need different keyboard inputs to start a download. You must become familiar with your terminal software and know the inputs required to prepare your terminal to receive a download. Speaking for myself I can help those who use the Ultimaterm package, Procomm or Telix. There are a good number of files currently on the Maverick system that are quite unique. And I would seriously doubt that many of you have these files. So give it a try, as with most things you can't hurt the system from your end, and even disconnecting during your call will simply result in the BBS resetting for the next call.

SPURTS FROM a LEAKY PEN

by H. Peter Unks

I have taken some bad advice in my time. One of the dumbest suggestions I ever followed came from a popular CoCo support magazine. This advice had nothing to do with the CoCo or any other computer. It had to do with ice and how to avoid slipping on it. At the same time as gaining instant traction, one could avoid doing harm to concrete driveways and the like. And besides, instead of chemicals getting into the lawn, a little more dirt wouldget added to it. So the advice went. How could I go wrong?

Falling on my butt has never been one of my favorite pastimes, so I did as advised and came home with 50 lbs. of kitty litter. I sprinkled it liberally on the ice over which Beverly and I had to tread. We had instant traction. Then it snowed and hailed and rained some more.

I had instant mud. Semi-frozen instant mud. I applied more kitty litter. Once again, instant traction. Then it snowed and hailed and rained. We were back to instant mud. Slippery, slushy mud.

It didn't take a great deal of skill to figure out where my wife, my dog, or I was at any given time. Just following the muddy footprints would lead right to us.

We gave the remaining 25 lbs. of kitty litter to a friend who owns a cat. Now that the snow has melted we see the first half of the bag is still with us. We have the only blacktop driveway in town with a dried mud trail through it. Our carpeting fared a little better thanks to Beverly's work with the vacuum.

The moral of the story is that CoCo support publications are good sources of CoCo support. But to avoid falling on your behind, get your advice at a hardware store.

Moving along to a different topic, let me say that I could use a software review or two. Hardware reviews would be welcome too. Or, how about an item on a computer show?

Upload your articles to THE MAVERICK BBS in ASCII just like Clyde did with his Atlanta adventure. And speaking of Clyde, HAPPY FIFTIETH to Clyde and Ruth!

See you at the meeting!

FIRST CLASS MAIL

6809 EXPRESS

The Official Publication of The
PENN-JERSEY COLOR
COMPUTER CLUB

H. Peter Unks, Editor

Mr. Clyde Gano

19 USA
This U.S. stamp, along with 25¢ of additional U.S. postage, is equivalent to the 'F' stamp rate
USA 29
KILMER P&DC NJ
PM
02 APR
1994

6809 EXPRESS

The Official Publication of The
PENN-JERSEY COLOR
COMPUTER CLUB

6809 EXPRESS
The "6809 EXPRESS" is the official publication of the PENN-JERSEY COLOR COMPUTER CLUB. The club is based in the greater Lehigh Valley of Northeastern Pennsylvania including sections of Northwestern New Jersey. Any non-profit organization may reprint any part of this newsletter provided credit is given. PJCCC will gladly exchange newsletters with any other computer club. Send requests to EDITOR, 6809 EXPRESS, Penn-Jersey Color Computer Club, 145 Seventh Street, Phillipsburg, NJ 08865. PJCCC assumes no responsibility for errors or omissions. PJCCC assumes no liability for damages resulting from the use of any information or programs contained in this newsletter.

GET ONLINE WITH THE MAVERICK
BBS! DIAL 215-760-0456