The Techie's Guide to C Programming

Looking for something hotter than BASIC for those technical programs? Try the power of C.

STEVE RIMMER

Programming is a lot like hardware design, really — except that you can't burn your fingers on the soldering iron. Much of the process of getting a program together is similar — at least in concept — to designing circuitry. Of course, programming offers you considerably more choices to cloud the issue with at the onset.

One of the obvious choices is that of the language you're going to program in. If you're predominately interested in hardware, and have only gotten into programming so much as it has supported your hardware efforts, you've probably developed what code you've needed in BASIC. While good for the odd ten line test routine, BASIC has more limitations than most wombats have fleas.

Wombats have a lot of fleas.

There are better development tools than BASIC — in fact, come to think of it, there are few worse ones that spring readily to mind. As you might have gathered, I consider that one of the best ones going is the C language. As we'll get into, it's a powerful platform on which to develop hardware related code — and just about anything else.

In addition to this, it looks cryptic and scares people who don't know what it's about. For example, if I park Horatio, the office cat, in front of a monitor full of C code, he just runs away.

This is the first of a series of features about C programming for technicians — don't sweat it, we're not going to blast through the entire ordeal in this magazine. It's based on Borland's Turbo C running on a PC compatible, which is a really handy development system and extremely cheap. However, one of the principal attributes of C is its lack of dependence on any one compiler or, to a large degree, on any specific sort of computer. As such, what we discuss here will be largely applicable to whatever computer and compiler you happen to have on hand.

I Don't Like Mondays

There are several strata of languages, and, as with most things, each stratum entails some trade offs. BASIC might be regarded as the highest stratum in a sense. It offers you complete protection from the nastiness of the system — you can't, in theory, crash your computer from BASIC. It's very easy to use. However, it arrives at this state by being tediously slow and by denying you access to about ninety percent of what your computer is capable of.

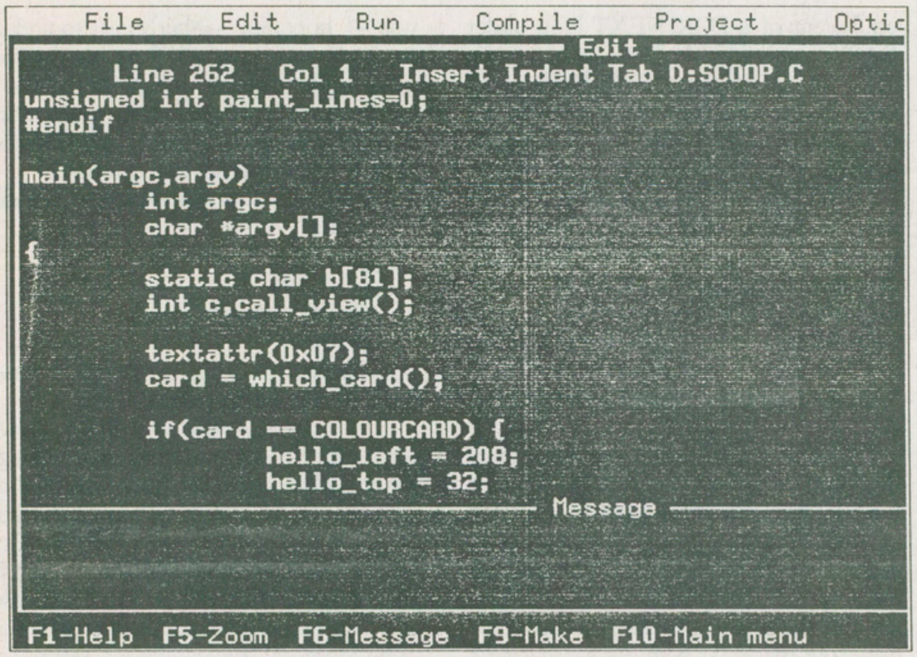
A pox on BASIC.

The other end of the spectrum is occupied by assembly language, which involves programming in the computer's native tongue. Assembly language programming allows you to make the computer do absolutely everything it can possible get up to, but you have total responsibility for handling the machine. Assembly language programs under development crash a lot, and when assembly language programmers get together for a brew and a few laughs after a long day down in the pits, they often talk about how colourfully they can blow up a program. Weird souls, these.

In addition to all this, assembly language code takes ages to develop, because you have to write stuff for absolutely every tedious little function you want to include in your program.

The middle ground is occupied by a plethora of languages, of which C is probably the most popular. These languages represent a trade off. They give you much of the speed of assembly language, some of the protection of BASIC and what are called "libraries". A library is a collection of low level routines which you can include in programs you write to keep you from having to re-invent the wheel every time you start a new project.

Pascal is another of these languages, by the way. We could just as easily be talking about Pascal here, except that nobody really likes it and it involves a lot more typing than C. It has been said that real men don't program in Pascal. Far be it for



```
    File      Edit      Run      Compile      Project      Optic
═══════════════════════════════ Edit ═══════════════════════════
      Line 262   Col 1   Insert Indent Tab D:SCOOP.C
unsigned int paint_lines=0;
#endif

main(argc,argv)
        int argc;
        char *argv[];
{
        static char b[81];
        int c,call_view();

        textattr(0x07);
        card = which_card();

        if(card == COLOURCARD) {
                hello_left = 208;
                hello_top = 32;
────────────────────────────── Message ──────────────────────────

 F1-Help  F5-Zoom  F6-Message  F9-Make  F10-Main menu
```

The illustrations in this article show the development of a program in the C language.

me to argue with such obvious wisdom.

Unlike BASIC, C is a compiler language. This may take a bit of explaining. When you run a BASIC program under BASICA or GWBASIC or whatever happened to come with your DOS disk, the BASIC language walks through your program *interpreting* everything line. If it finds a line that says

PRINT "ZEBRA LUST"

it trucks off somewhere in the interpreter, finds the routine that prints, tells it where the string to be printed is and does the deed. Then it finds the next line to interpret.

This process is very slow.

Under a compiler, you write the program as a text file using a word processor or, in the case of something like Turbo C, with the text editor built into the language system. You then run the compiler program, which translates each action in your program into corresponding machine code. When the compiler is done, you have an authentic EXE file, all ready to run from MS-DOS.

This makes the code a lot faster than it would have been under an interpreter. However, the process of getting into the text editor, editing your program, getting out of the text editor again, running the compiler, running the EXE program and then repeating the process is a bit deadly. While I learned to program in C this way, you won't have to because integrated environments like Turbo C mash the whole ugly process together. We'll get into this in greater detail in a future installment of this series.

In addition to this, because the result of a compilation is effectively machine language, we can write what are called "hybrid" programs, that is, ones which are comprised of both C language and assembly language routines. This allows you to have the ease of development of C and the speed of assembly language in those few cases that you really need it. We'll get into this in a future article as well.

The most important feature of C, however, is a bit intangible. C is structured. BASIC can be structured, and some of the newer BASIC environments, such as QuickBASIC, have lifted quite a lot of structure from C in an effort to overcome the tendency of BASIC programmers to write spaghetti code. However, C is inherently structured, and by simply letting its normal structure flow out through your fingers and into your computer, you'll write tight, easy understandable programs and dance past about three quarters of the bugs that BASIC programs are heir to.

Trust me.

No Deposit, No Return

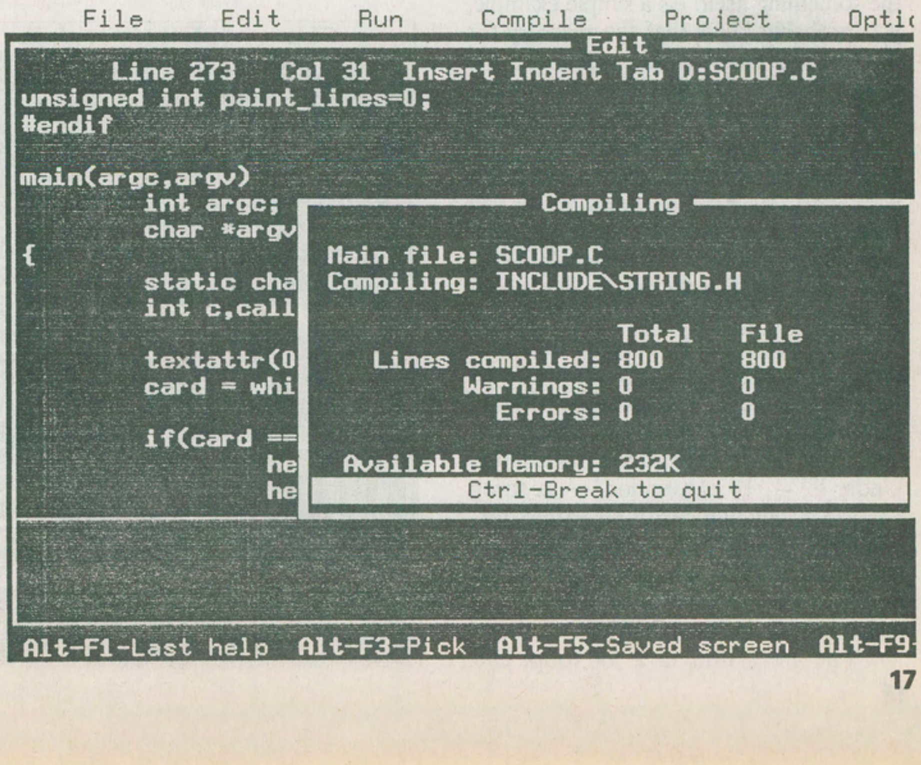
A C program consists entirely of functions. Under BASIC, a function is, by definition, a pretty simple thing. Under C, it's the essential building block of any program.

Under C, a function is a routine which takes in zero or more arguments, does something and optionally returns a result. Functions can call other functions, which in turn can call still other functions. When a C program runs, it starts off by calling a function named *main*, which in turn calls all the other functions of the program.

This is a very simple C program.

```
#include "stdio.h"

main()
{
/* print a string, Billy */
printf("Hello, planet");
}
```



```
    File      Edit      Run      Compile      Project      Opti
═══════════════════════════════ Edit ═══════════════════════════
      Line 273   Col 31  Insert Indent Tab D:SCOOP.C
unsigned int paint_lines=0;
#endif

main(argc,argv)
        int argc;     ═══════════ Compiling ═══════════
        char *argv
{                    Main file: SCOOP.C
        static cha   Compiling: INCLUDE\STRING.H
        int c,call
                                          Total    File
        textattr(0      Lines compiled: 800       800
        card = whi          Warnings: 0         0
                              Errors: 0         0
        if(card ==
                he    Available Memory: 232K
                he           Ctrl-Break to quit

 Alt-F1-Last help  Alt-F3-Pick  Alt-F5-Saved screen  Alt-F9-
```

and when *print* finishes doing its stuff, they'll disappear. If *print* is called a second time, they'll come back. Another function could also have two *int*s called *i* and *l*, but they'd be different variables, and the two functions would not interact. This eliminates an additional hive of bugs that plague BASIC programs, wherein all variables are global.

The C language allows for global variables as well — ones which can be accessed by all functions — but we'll check them out another time.

Under C, we pass small objects to functions directly. For instance, an *int* is a small object. A string is a big object, and, while I suppose we could pass it in its entirety, this would make our programs slow and ugly. As such, we pass pointers to strings in most cases.

Pointers are one of the things under C which crawl into the inner ears of beginning C programmers and sing off key into their brains until they go mad. No foolin'. They're a bit hard to get your head around at first, and you'll have to use a bit of faith in this case to understand what this one is up to.

Under C, the notation **s*, as it's used in the second line of the *print* function, tells C that what is going to be passed to *print* should be regarded as a pointer to a string, or, more properly, to an array of *chars*. Thereafter, the variable *s* will stand in for the string that was passed to *print*. In this case, *print* has to trust that it was actually passed a string.

A pointer is simply an object which says where something is, rather than being the something itself. As a simple example, you probably know that the memory for the first character on the screen of a PC usually lives at location zero in segment B800H — you might have experimented with POKEing data to this location to see the screen contents change. If we create a pointer to this location under C, we can alter the screen contents by altering what the pointer points to. This is actually a very useful way to directly access the screen under C.

The first actual line in *print* that does anything is the one which assigns *l* the value of *strlen(s)*. The *strlen* call is a library function which returns an *int* value containing the length of the string passed to it. Under C — by convention — a string consists of any number of characters terminated by a zero byte, or "null". As such, what *strlen* actually does is to start with the first location pointed to by *s* and keep counting 'til it finds a zero byte.

The next line is a *for* loop, the equivalent of a FOR NEXT loop under BASIC. As is typical of C, it's a bit cryptic at first. It translates as follows. First, set the value of *i* to zero. Repeat the loop while *i* is less than *l*. With each iteration of the loop, increase the value of *i* by one.

That last one might not be quite as easy to understand as were the first two. In programming, incrementing and decrementing values by one is a common occurrence, so C gives a short hand way of expressing it. The notation + +*i* means to increment *i*, and –*i* means to decrement it. Wait'll we get into how you increment *i* by two.

The *for* loop executes whatever's in the set of curly brackets associated with it for each iteration of the loop. In this case, we call another library routine called *putchar*, which prints a single character to the screen. We pass it, in turn, each element of the array of characters that comprises our string. If *s* points to a string, *s[0]* is the first element, *s[1]* is the second, and so on. Everything starts with zero in C — there seems to be little point in wasting perfectly good numbers by starting with one.

Unlike as with BASIC, we don't have to tell a C function to return when it's done with. After the last statement inside the outer set of curly brackets of the *print* function has been completed, it will automatically return to whatever called it.

Extra Texture

The *print* function we've looked at was — hopefully — fairly easy to understand. It was, however, very clunky and awkward by C standards. Not only was it inelegant, but it would execute much more slowly than needs be. Here's a quick look at how C *really* parties before we split.

This is the *print* function, version two.

```
print(s)
char *s;
{
while(putchar(*s++));
}
```

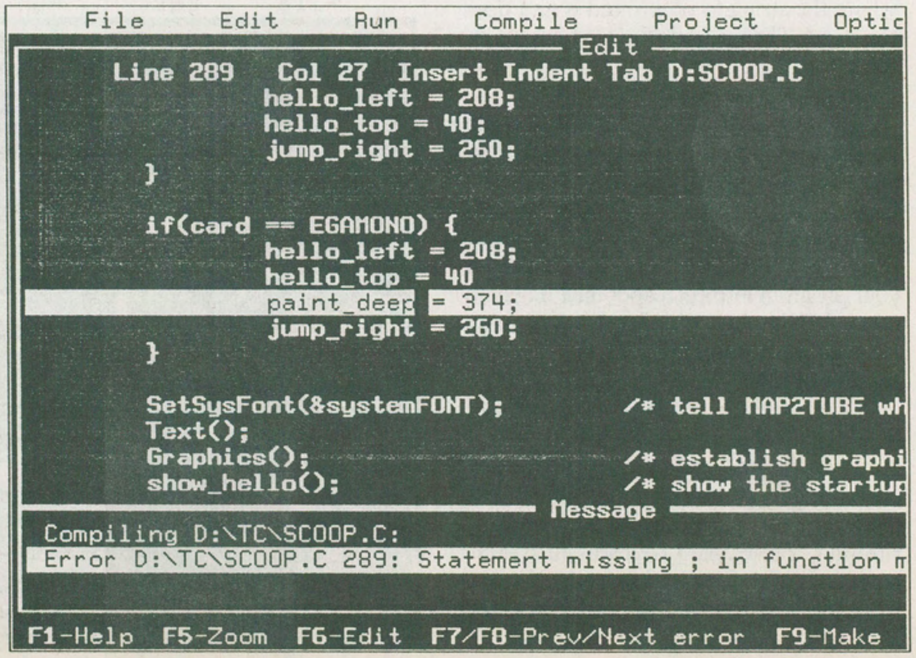
There's nothing up my sleeve — that's all of it. Let's see how it works.

In order to know what this is up to, you'll have to know a few more things about how C gets its act together. One of the concepts which C is very fond of is that of truth and falsehood, which is very philosophical, of course. Under C, zero is false and everything else is true. In the case of a string, then, all of the characters in the string represent true conditions, except for the zero byte at the end. Very useful, this.

This version of *print* uses a new C construct, the *while* loop. This is of the form

```
while(whatever is true) <do this>
```

The notation **s+ +*, designating what gets passed to *putchar*, is splendidly cryptic. It tells C to extract the number which *s* is pointing to, pass it to *putchar* and then to increment *s* to point to the next location in memory, that is, the next character in the string. Under C, if we say

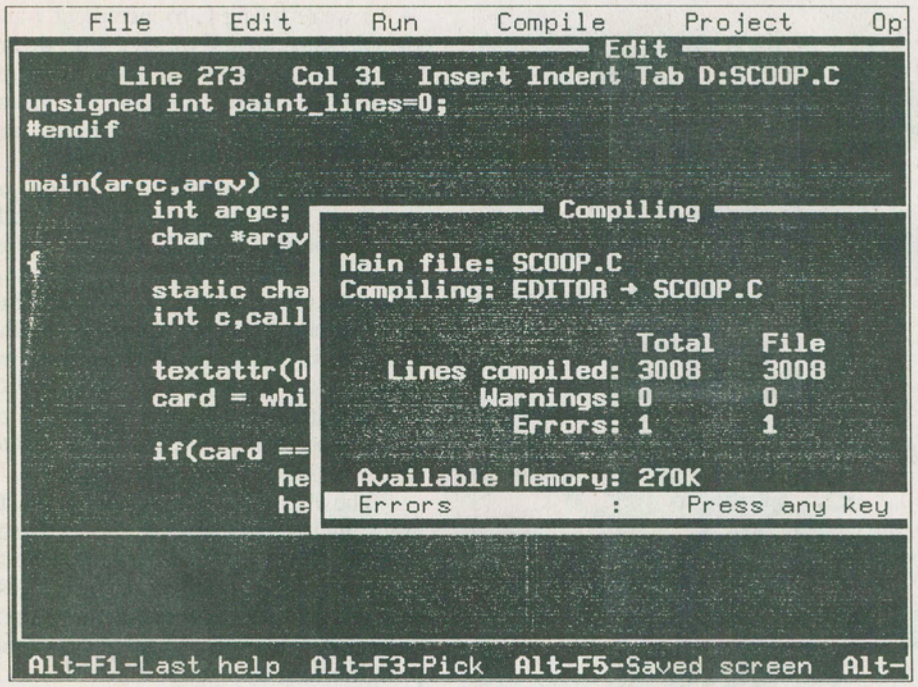


```
      Line 289   Col 27   Insert Indent Tab D:SCOOP.C
                     hello_left = 208;
                     hello_top = 40;
                     jump_right = 260;
        }

        if(card == EGAMONO) {
                     hello_left = 208;
                     hello_top = 40
                     paint_deep = 374;
                     jump_right = 260;
        }

        SetSysFont(&systemFONT);                /* tell MAP2TUBE wh
        Text();
        Graphics();                             /* establish graphi
        show_hello();                           /* show the startup
```

```
Compiling D:\TC\SCOOP.C:
Error D:\TC\SCOOP.C 289: Statement missing ; in function m
```



```
    File      Edit      Run      Compile      Project      Op
═══════════════════════════ Edit ════════════════════════════
      Line 273   Col 31  Insert Indent Tab D:SCOOP.C
unsigned int paint_lines=0;
#endif

main(argc,argv)
        int argc;  ════════════ Compiling ═══════════════
        char *argv
{                 Main file: SCOOP.C
        static cha Compiling: EDITOR → SCOOP.C
        int c,call
                                      Total    File
        textattr(0    Lines compiled: 3008     3008
        card = whi          Warnings: 0        0
                              Errors: 1        1
        if(card ==
                he    Available Memory: 270K
                he    Errors          :     Press any key

 Alt-F1-Last help  Alt-F3-Pick  Alt-F5-Saved screen  Alt-
```

This program defines a function called *main*, which, as we said, will automatically be called by the computer when the program is run. A function under C is always written with parentheses after it. These contain any arguments passed to the function. In this case, you might think that *main* will always have an empty argument list. In does in this case, but it needn't always. C allows for command line arguments to be passed to *main* – but more on this another time.

The working bits of a function under C are always contained in curly brackets. As we'll get into, a function can have smaller bits of itself enclosed in still more curly brackets – indenting and nesting pairs of curly brackets is an art form under C, and makes even fairly sloppy code look very elegant.

This *main* function calls one other function, called *printf*. The *printf* function is a standard C function, and is provided with the library of the compiler. It's amazingly powerful, as we'll get to anon. In this case, though, we've used it in a very simple sense. It prints a string to the screen. Strings in C are contained in double quotes, just like under BASIC.

Every line in a C function is ended by a semicolon, and leaving the semicolon off is one of the most common mistakes for beginning C programmers. Except in special cases, a C compiler ignores line returns when it's scanning a program. This function could be written as

```
main() { printf("Hello, planet"); }
```

as far as the compiler is concerned. The former version is a lot easier to read, though.

The very first line of the program is a compiler directive. It tells C to read in and compile a file called STDIO.H before it does anything else. This is called a "header" file. This one is also provided with the compiler, and includes some basic definitions to tell the compiler things like how to find the screen. It's included with every program you write, and you'll probably find that lots of other headers want including too, as you get into more complex programs.

Finally, just above the *printf* call there's a comment. Under C, anything which is enclosed in /* and */ is a comment, and will be ignored by the compiler. This is useful to add comments to your code – C is a little terse, and very hard to read all by itself without a few prompts from the real world. It's also handy for temporarily "commenting out" blocks of code in a program under development.

One of the justifications for *not* putting comments in your code in C is that anything which is difficult to write should also be difficult to read. You can take this any way you feel like.

One last note before we move on – don't mix up slashes and backslashes under C – they mean different things. In fact, it's worth observing that C uses every one of the normal printable ASCII characters for something. Further, C is case sensitive. The compiler regards *printf*, *PRINTF* and *Printf* as being three different things. C programs are traditionally written in lower case.

This next program is a bit more complicated. Note that I've left off the *include* directive at the beginning – we won't bother with these in future, as they can be assumed to be there.

```
main()
{
print("There once was a Hermit
named Dave");
}

print(s)
char *s;
{
int i,l;

l=strlen(s);
for(i=0;i;++i) {
putchar(s[i]);
}
}
```

In this program, we use both library functions and one of our own. The function *print* is defined here as being something which prints a string to the screen – essentially what *printf* was up to in the last example. This one, however, lets us see how the whole thing works.

There's a lot going on here.

Under C, data is stored in different sorts of variables, called *types*. Simple numbers are stored as integers, or *ints*. An *int* is a signed sixteen bit number on a PC. Strings, such as the one we're going to print, are stored as arrays of characters. A single character is of the type *char*. As such, a string is contained in a buffer of sequential *chars*.

Under C – unlike BASIC – every variable you use must be explicitly declared before you use it. You must tell the compiler it exists, and you must say what it will be used for. In *print*, we have declared two *ints* called *i* and *l*. Because we must declare what type of variables these are, C will not allow use to casually put the wrong sort of data in them, a common failing under BASIC. This is called "strong type checking". We can over-ride this when we need to, but it catches a whole seething hive of potential bugs in the normal development of a program.

The two *ints* in *print* are "local" variables. They exist only within this function,

+ +*s* we're telling C to increment *s* and then do something. If we say *s* + +, we are telling C to do something and then increment *s* — which, in this case, means something slightly different, as you can see.

The asterisk may also be confusing, as it appears to mean different things in different contexts. There's a reason for this, actually — it does. If *s* is a pointer to a string, **s* is the first byte of the string. But wait, you cry — I thought that *s[0]* was the first byte of the string. It is — the two notations are equivalent in this case. We can use either, although in this version of *print* the one with the asterisk allows us to do less typing and write tighter code.

The library function *putchar* prints its argument to the screen and, for the sake of code like this, also returns it. As such, testing the truth of *putchar*, in this case, is just as good as testing the truth of each byte of the string. This means that *putchar* will print the zero byte when it gets there, of course, but since zero bytes aren't printable, nothing will happen.

What this function is actually saying, then is this. While *putchar* has not printed a zero byte, get the next byte of the string, increment the string pointer for the successive iteration and print that byte. We can write this in less compact C notation as

```
while(*s) {
putchar(*s);
+ +s;
}
```

This is easier to read, and, in fact, results in no less acceptable code than our really compressed version above.

No Cigarettes, No Matches

The weird, compact notation of C takes some getting used to, and you shouldn't worry too much about not being able to write fluently in it at first. If you're used to BASIC, learning C will be a bit of a brain trauma to begin with. Stick with it — you'll find that it gets a lot easier as you go. In time, you'll wonder how you could ever have gotten anything to work in a language as funky and unnatural as BASIC.

In the next installment of this series, we'll look briefly at a few of the compiler packages available to let you do some real experimenting in C, talk about some of the better reference books about and look at a bit more of the esoteric little world of C code.

Same time, same station. ■

Value Priced Oscilloscopes

20 MHz DUAL TRACE

OUR MOST POPULAR SCOPE!

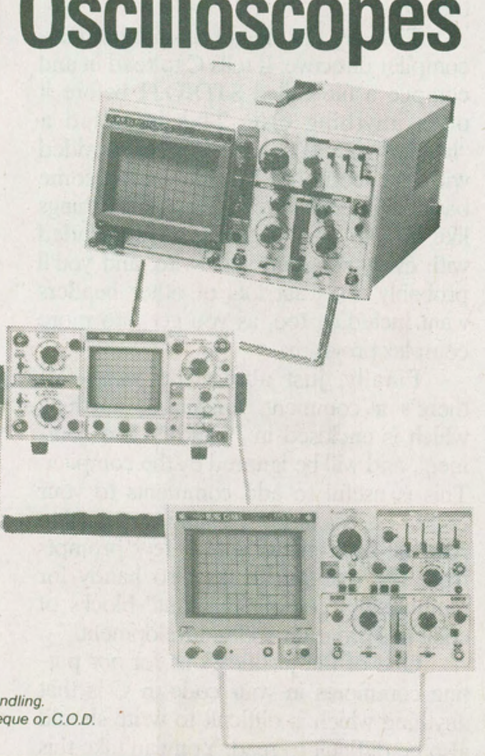
Features built-in component tester ideal for troubleshooting solid state circuits and components with no circuit power; 80 x 100 mm high luminence CRT; TV sync measurement for vertical and horizontal circuitry. Sensitivity 1mV/div. Dimensions: 6.4"x11.5"x14".
MODEL NO. 33330 **$595.**

15 MHz PORTABLE

Dual trace, rechargeable battery or AC operation, 3.5" CRT, TV video sync filter, sensitivity 2mV/div., carry handle.
MODEL NO. OS615S **$829.**

35 MHz DUAL TRACE

Features fast sweep time, high brightness 80 x 100 mm CRT display, stable display of Ch.A and Ch.B waveforms, single trace viewing, sensitivity 1mV/div.
MODEL NO' OS635 **$849.**
Set of 1:1/10:1 switchable probes $59.



50 MHz also available. Please add $15 for shipping and handling.
Order by phone or mail. Credit card, money order, cert. cheque or C.O.D.
Ontario residents add 7% P.S.T.

VISA MasterCard

KB ELECTRONICS
355 Iroquois Shore Road, Oakville, Ontario L6H 1M3
Tel: (416) 842-6888 • Telex: 06-982396 KBELOKVL

ASK FOR OUR FREE CATALOGUE TEST BENCH EQUIPMENT