FEATURE

Techie's Guide to C Programming Part 6

This month we're going to get a bit less dignified about C and discuss some of the dirty tricks ones has to play on the PC to make it work out right.

STEVE RIMMER

In its most perfect and useless state, C is a pure language. It can't play music, it can't print in red text and it can't draw bar graphs. All it can do is process. It takes data in through a mysterious channel and sends data out through an equally mysterious channel and knows nothing of the world beyond its door.

This is C on a mainframe. Anyone who tried to get away with selling a C compiler like this for the IBM PC would be hooted out of the known universe, of course.

In order to make C into a language which is actually good for something, the designers of PC-based compilers for it always wind up adding huge collections of functions which relate specifically to the PC's peculiar hardware. These handle things like fast screen updating, sound and graphics as well as a dedicated interface to the PC's operating system and BIOS.

It's useful to know how to make these things work because, while they represent the cheapest, nastiest, most reprehensible and ill-behaved aspects of C programming, they are essential to writing programs which actually work.

This month we're going to look at some dirty tricks. Some of these things are specific to Turbo C, but you'll find nearly identical functions in most other popular PC based C compilers.

Vas Ist DOS?

There are two levels of machine specific dirty tricks under C. The low level tricks give you access to the machine's guts. The high level tricks let you do hardware specific things, such as graphics. We'll deal with the low level things here.

The lowest software level of the PC is its BIOS. This is the firmware which starts up the computer and makes it do simple tasks, such as fetching a keyboard character or printing a byte to the screen.

The PC's BIOS is not very bright and extremely slow, and for this reason very few applications have to deal with it directly. There are occasions, however, wherein only the BIOS can help you. C on the PC provides an interface directly to the BIOS.

The BIOS provides a number of groups of functions, all of which are gotten at through software interrupts. These are roughly grouped together by interrupt number. It isn't really necessary to know how interrupts work or even what they do, so long as you know what the BIOS is up to and what numbers to put in the appropriate places to make it do its thing.

In order to make the BIOS get up and dance, from any language which supports calling it, you will need a listing of its entry points. The best one of these which I've encountered to date is *IBM ROM BIOS* by Ray Duncan, published by Microsoft. Read through it carefully and computers will never trouble you again — your eyes will be too shot to see them.

The most useful BIOS function is 10H, which deals with the screen. Ordinarily you will not want to use this to print characters to the screen because it's too slow. However, there are several areas in which it can be of considerable use. It can position the cursor on the screen and it can scroll or clear windows exceedingly quickly.

Let's look at a simple problem. One of the few reliable ways to make the cursor of the PC disappear from within a program is to put it on line 26 of the screen. Astute readers will note that there is no line 26 on a 25-line screen, which is why the cursor can't be seen when it's down there. This works well, except that the screen position functions found in most C compilers, usually *gotoxy*, won't allow you to pass them illegal values.

In this case, we have to use the BIOS to move the cursor to line 26. This is accomplished with BIOS call 10H, function 2.

In machine language, this would be done with the following bit of code.

```
MOVAH,0FH
INT10H
MOVAH,02
MOVDX,1A00H
INT10H
```

Hands up everyone who made any sense of this at all.

The first rule of low level calls to the PC's guts is that they're never even marginally intelligible without a bit of inside information.

Techie's Guide to C Programming, Part 7

The definition of the cursor mover call of the BIOS says that the AH register has to contain the number 2, since this is BIOS function 2. This accounts for the line MOV AH,02. The DH register is to contain the vertical position and the DL register the horizontal position. These two registers can be combined into the DX register. The horizontal position will be zero. The vertical position will be 26. In hexadecimal, 26 is represented by the number 1AH. Hence, we MOV DX,1A00H.

The BH register is supposed to hold the video page number. The CGA card, for example, actually has four pages of video memory. In order to find out which of these is in use at the moment, we have to use a second BIOS call before we get to the first one. Function 0FH returns the page number in the BH register of the processor, just where we want it.

Let's see how this is done in C. It's a bit easier.

```
HideCursor()
{
union REGS r;

r.x.ax = 0x0f00;
int86(0x10,&r,&r);

r.x.ax = 0x0200;
r.x.dx = 0x1a00;
int86(0x100,&r,&r);
}
```

I said it was a bit simpler — not blindingly clear.

C provides us with a function called *int86*, which simply executes software interrupts. Its first argument is the number of the interrupt we want executed, as found in the ROM BIOS book or in some other suitable list. Its second and third arguments are pointers to a peculiar kind of struct called a "union". The REGS union is defined in the DOS.H file of your compiler. It just holds the values of all the important machine registers. The second argument is the registers going into the call and the third one is the registers coming out. It's allowable to return the registers in the same REGS variable as they start in, as we've done here.

We'll look at the notation of unions in greater detail later on in this series. For now, you can apply this approach to having your C program throw any software interrupt you have a need of.

Here's a slightly more involved example of a BIOS call. This routine scrolls a window on the screen up by a defined number of lines. If the number of lines is zero, the window is blanked. We'll assume that the screen text is in white on black text, which has a screen attribute of 07H. We'll discuss this in a future episode too.

```
ScrollUp(left,top,right,bottom,how_much);
int left,top,right,bottom,how_much;
{
union REGS r;

r.x.ax = 0x0600h + how_much;
r.x.bx = 0x0700;
r.x.cx = (top * 256) + left;
r.x.dx = (bottom * 256) + right;
int86(0x10,&r,&r);
}
```

Working backwards, the definition for this interrupt tells us that its function number is 6. The function number always goes in the AH register, or the upper byte of the AX register. The AL register, or the lower byte of the AX register, contains the number of lines to scroll up by or zero to clear the window. The BH register contains the screen attribute for the window, seven in this case.

The CL and CH registers contain the horizontal and vertical co-ordinates for the upper left corner of the window. The DL and DH registers contain the co-ordinates for the lower right corner.

Snakes and Ladders

As you will have noted, printing to the screen of a PC in the "correct" way, which involves using several interrupt calls strung together, is tediously slow. The way which most applications manage it is to locate the memory where the screen display lives and poke the requisite screen text right into the buffer.

This is cheating, of course, but it works very well. Most C compilers feature high speed screen printing which is a lot more elegant than what we're going to look at here. However, being able to treat the screen just like another chunk of memory has its uses.

In order to do high speed screen access from C, we have to be able to find the screen and figure out which locations correspond to which characters. Regrettably, the screen memory moves around depending on what sort of display card is in memory. CGA, EGA and VGA cards place it at segment B800H. Hercules cards put it at segment B000H.

A segment is just part of a memory address, as we'll see.

In order to find out where the screen memory is, we have to find out what sort of card is in the computer our program is running on. We can do this with a BIOS call.

```
ScreenSegment()
{
union REGS r;

int86(0x11,&r,&r);

if((r.x.ax & 0x00c0) == 0x00c0)
return(0xb000);
else return(0xb800);
}
```

This uses a different BIOS call, interrupt 11H. This interrupt returns a word in the AX register with bits that indicate whether there's a math coprocessor in the machine, how many floppy drives are on line and what the video card type is. In essence, it reads the configuration DIP switches. If bits four and five are both set, we have a monochrome or a Hercules card. If they aren't, it must be one of the other cards.

Now, here's where we start getting tricky. In the large model of Turbo C, and any other C compiler for the PC, we can create a pointer which points anywhere in memory. The function for doing this is MK_FP, for "make far pointer". This bit of code will always point to the beginning of the screen buffer.

```
char *p;
int s_segment;

s_segment = ScreenSegment();
p = MK_FP(s_segment,0);
```

If we now say **p='A'*, the very first character in the upper left corner of the screen will become the letter A. This is the location specified by the screen segment and an offset of zero, that is, the very first byte in the segment which defines the screen memory.

On the PC's screen, the odd numbered bytes in memory hold the characters. The even numbered bytes hold the attributes for those characters, which define what colours they are, or whether they're flashing, inverse or underlined on a Herc card.

This will draw a line of the letter A across the top of the screen. It assumes that the previous stuff has all been done.

```
int i;

for(i=0;iá;i+=2) p[i]='A';
```

There are 160 bytes in an 80-line screen, and we want to address every odd one. This code will make all the A's in the last line flash in inverse text.

```
for(i=0;iá;i+=2) p[i+1]=0xf0;
```

The attribute 0xf0 is black text flashing on a white background,. quite remarkably hard on the eyes, actually.

Finally, this function will print a string *s* at the location *(x,y)* on the screen using this rather tricky screen interface.

```
printxy(s,x,y)
char *s;
int x,y;
{
char *p;
kcint offset;

offset += (160 * y) + (2 * x);
p= MK_FP(ScreenSegment(),offset);

while(*s) {
*p++ =*s++;
*p++ =0x07;
}
}
```

To use this function we would do this:

```
printxy(10,20,"There once was a hermit named Dave");
```

This call would print the string ten characters in from the left on line 20.

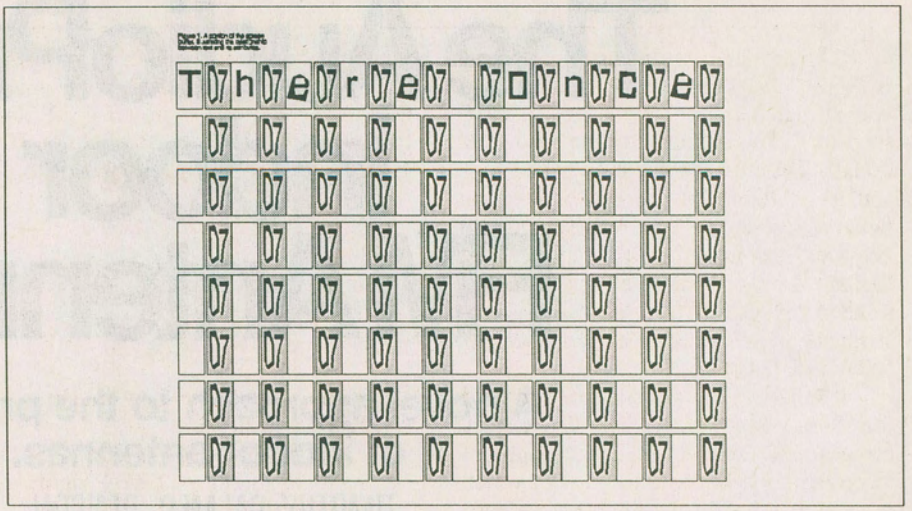
There are a lot of holes in this function, of course. First off, if you have a colour card in your computer, you'll find that executing this creates a blast of snow on your screen for an instant. This is because the colour card is a bit dim witted, and it clashes with the processor if data is jammed into its buffer at the wrong times. In order to avoid this, we would have to write a fairly complex routine in assembly language which reads some of the registers of the card's controller chip and waited until it was just the right moment to access the buffer. Colour cards being rather antediluvian, it hardly seems worth the effort.

More fundamental problems with this function include its inability to do any screen control at all. It cannot respond properly to carriage returns, line feeds, tabs and so on. You might want to try adding these features to it once you understand what it's up to.

Regularly Scheduled Program

It might well be argued that the subjects we've looked at this month are a bit advanced, and might well have been better dealt with later on. In theory this is true... in reality, you invariably have to get at the guts of your PC sooner or later. Once you get started, you'll find yourself exploring all manner of dirty tricks. There are those programmers who are of the opinion that the PC *itself* is a dirty trick, something which IBM created in retaliation for one of those long forgotten antitrust suits which were always snapping at its corporate heels.

In this light it seems somehow appropriate that a few dirty tricks should live on in every program that runs on it. ■



How the character byte and the attribute byte would look like in memory.