C Language Programming for Techies

Vectors are one of the more mysterious elements of PC architecture, and one that most programmers happily leave alone. There are, however, a number of fairly risk-free applications of vectors and the mutation thereof under C.

by Steve Rimmer

Down at the very bottom of memory in the dungeon of every PC there lives the vector table. It's a typical dungeon dweller, dark and brooking, full of menace... unfathomable. If you venture down to low memory without all the right spells it will probably bring the whole castle down on top of you.

The low memory vector table of a PC is a list of addresses of important basic functions in your computer. This includes things like the BIOS code that decodes your keyboard, the DOS INT 21H handler that handles files and memory allocation, hardware serial port handlers and the code that tells you there has just been a memory parity error and all your work for the past six hours has been lost.

The mechanism of the interrupt vector table is intimately tied up with that of the 8086 series processors that drive PCs. When the processor executes an INT instruction, what it actually does is to make a sort of indirect far call to where the appropriate entry in the interrupt vector table points.

As a far call requires two words to address it... four bytes... the processor would find the entry for INT 21H by multiplying 33... that's 21H in decimal... by four and leaping to the address pointed to by the two words at 0000:0084H.

All of this seems very low level, and probably should not concern programs written in higher level languages such as C. However, being able to successfully meddle with interrupt vectors can be exceedingly useful. You can get around many of the nastier omissions and holes in the architecture of your system, as well as tapping a few hitherto untapped resources.

No tricks, exercise or unpleasant machine language are required.

Crashing Out

If you hold down the control key and hit the break key while a program is running, the BIOS of your computer will print "^C" on your screen and issue an INT 23H instruction. The vector for this interrupt normally points back into DOS, which will terminate the program in question and return you to the command line.

There are all sorts of instances in which you may not wish to have this facility available to the users of your programs. For one thing, it's not all that professional to have your programs crash back to DOS just because someone hits the wrong combination of keys. In addition, if your program uses extended or expanded memory, crashing out like this may leave some memory handles dangling and blocks of extra memory unrecoverable until you reboot your machine.

The way to get around this problem is to replace the normal vector for INT 23H with one that points to code that you've written. Turbo C has a dedicated function for this, called *ctrlbrk*.

Here's a typical replacement control break handler.

```
int myBreak(void)
{
  gotoxy(1,1);
  printf("Leave the control
  break key alone");
```

```
  return(1);
}
```

We would replace the default control break handler with this one as follows.

```
int myBreak();
ctrlbrk(myBreak);
```

There are a number of fiddly details to consider in this process. The first is that the value returned by *myBreak* is important. If it returns zero, your program will do whatever is in your control break handler and then return to DOS. If it returns a non-zero value, your program will resume where it was when the control break key was belted.

Unfortunately, redirecting the control break vector doesn't stop DOS from printing "^C" wherever the cursor happens to be when control break is hit. If you're writing a program with a formatted screen you probably won't want the default system cursor anyway. You can get rid of it with this function.

```
hideCursor()
{
    union REGS r;

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

  r.x.ax=0x0200;
  r.x.dx=0x1a00;
  int86(0x10,&r,&r);
}
```

This simply sets the cursor onto the line directly below the last line of your screen and keeps it there. If DOS decides to print "^C," it will go undetected. Unfortunately, so will anything else your system decides to print.

In most cases, it's very important that you not alter interrupt vectors and leave them altered when you leave a program. This could result in an interrupt which happens after your program has terminated causing the processor to leap into the middle of a word processing document, for example, and trying to execute the text of your monthly report.

In the case of the control break vector this is not actually a problem... the original vector will be restored for you when your program returns to DOS. We'll look at how to capture and restore other vectors momentarily.

Abort, Retry, Disintegrate

One of the most difficult things to trap for in a complicated program is a user attempting to access a floppy disk which is still in its sleeve five inches immediately to the right of the disk drive it's supposed to be in. The usual result of trying to do this is to have DOS tell you that things are not working out, and ask you if you'd like to abort, retry and so on.

If you select "abort," you'll return to DOS.

If a disk drive runs into trouble accessing a disk, it complains about it by throwing interrupt 24H. This is usually hooked to the code that prints the "abort, retry, ignore" message. However, as with all interrupt vectors, you can redirect it to something which is a bit less catastrophic.

The ideal situation would be to test the drive you wish to access before you go to do so. If it's door is open, you could prompt the user of your program to put a disk in it... rudely, if you're of a mind to.

This process is a bit more involved than the control break handler is because while Turbo C does have the functions to do it, it isn't all that helpful in showing you how to use them.

To begin with, you cannot leave the INT 24H vector dangling when you aren't specifically testing for an open drive... it should always point to the DOS INT 24H handler when you don't have a specific interest in its activities. As such, when you want to test for the status of a drive, you must fetch the old vector and store it somewhere, do what you want to do and then restore the original vector before you move on to better things.

Having saved the old vector, you can point the INT 24H vector to a safe bit of code which need not print anything nasty to the screen.

Here's the code to test a potential disk drive before you go to actually access it. The argument passed to it is the number of the drive you want to test... zero for drive A, one for drive B and so on. What it actually does is to shanghai the INT 24H vector to point to our own handler, which in turn will set the value of *diskErr* accordingly if a disk error transpires.

It will return zero if the drive is accessible.

Having set everything up, the function forces a drive access by attempting to open a file on the test drive. It doesn't actually matter whether the file exists or not.

You will notice that rather exotic looking declaration for *oldHardErr*. This is how you would create a pointer to an interrupt vector. It will turn up again.

```
int diskErr;
char linebuf[81];

testDisk(n)
  int n;
{
  void interrupt
(*oldHarderr)();
  FILE *fp;
  char b[32];

  oldHarderr=getvect(0x24);
  harderr(diskErrorHandler);
  diskErr=0;
  getcwd(linebuf,80);
  sprintf(b,"%c:\\TEMP.DAT",
  n+'A');
  if((fp=fopen(b,"r")) !=
  NULL) fclose(fp);
  setvect(0x24,oldHarderr);
  return(diskErr);
}

#pragma warn -par
int
diskErrorHandler(errval,ax,
bp,si)
  int errval,ax,bp,si;
{
  if(ax>=0) {
  diskErr=1;
  restoreDir(linebuf);
  }
  hardretn(2);
}

restoreDir(s)
  char *s;
{
  strupr(s);
  if(isalpha(s[0]) &&
s[1]==':')
    setdisk(s[0]-'A');
  chdir(s);
}
```

A Bit of Music

The clock tick interrupt, INT 1CH, is among the most useful of PC interrupt vectors for performing sneaky tasks. It's actually run by the PC's hardware. Eighteen times a second it gets called by one of the timer chips. Normally it just points directly to an IRET... a return from interrupt instruction... up in the BIOS, and as such does nothing. However, you can hook it to do all sorts of interesting things.

The most common use of the INT 1CH vector is to drive a screen clock. Resident programs that display the time in the upper right corner of your screen do so by attaching themselves to this vector.

You can make all sorts of independent background processes work in your programs with surprisingly little effort if you know how to manipulate this useful vector. In the following example we'll check out a function which can play random music in the background of any C program, seemingly without any intervention save for setting it loose.

This is not a terribly practical application of the clock tick vector... the music will drive you to madness fairly shortly... but it illustrates the process.

```
void interrupt beNoisy();/*
little music function */

main(argc,argv)
  int argc;
  char *argv[];
{
  void interrupt
  (*oldTimer)();
  int i,r=0;

  /* save the old timer
  interrupt */
  oldTimer=getvect(0x1c);
  /* set the clock tick to play
  tunes */
  setvect(0x1c,beNoisy);

  /*** your program's guts go
  here ***/
  getch(); /* something to
  pause for a while */

  /* undo the clock tick */
  setvect(0x1c,oldTimer);
  /* make sure the sound is
  off */
  nosound();
}

/* this is an interrupt
driven noisemaker. it's
called
 by the hardware about 18
 times a second */
void interrupt beNoisy()
{
  static int scale[7]=
  {220,247,261,294,330,
  349,392};
  static int count,soundon;

  if(soundon && count > 1) {
  count=0;
  nosound();
  soundon=0;
  }
  else if(!soundon && count
  > 2) {
  count=0;
  sound(scale[rand()%7] *
  (rand()%5));
  soundon=1;
  }
  ++count;
}
```

This bit of code illustrates Turbo C's functions for dealing with interrupt vectors it has not specifically planned for. The *getvect* function returns the current interrupt vector and the *setvect* function replaces the current vector with one of your own devising. Because interrupts can happen unexpectedly, make sure you don't go changing their vectors until you're certain your handler is initialized and ready to handle something. As an example of this, if you exchange the order of the *setvect* and *nosound* functions at the end of the *main* function of this little program, the sound will continue to drone on once in a while, as a timer rick interrupt can occur in between the two calls.

The actual handler, *beNoisy*, uses the Turbo C *sound* function to play tunes. It randomly selects note values from the equally tempered C scale at the top of the function. That's the key of C, not the C language in this case.

Because the handler will be called eighteen times a second... it's actually 18.2, but this doesn't matter for our applications... you can count up the number of calls to the handler function to measure time. Thus, if you wanted the note to change one a second, you would increment *count*, in this case, and do something when it reaches eighteen... resetting *count* to zero in the process.

Interruptus

There are numerous other applications for Turbo C's interrupt manipulation functions. In last month's Computing Now, for example, we looked at a way to write an interrupt driven high speed serial port handler using them.

While you should not go meddling with system interrupts without first understanding what they do, the ability to redirect them can give your programs a whole new set of resources. You'll probably want a few good books on system interrupts to fully understand what they're about. ❐