XX CHAPTER

Miscellaneous

This book has attempted to cover every major topic of C programming, and hopefully the information was useful and understandable. It is impossible, however, to cover every possible aspect of something as complex as the computer in a book as succinct as this. Therefore, this chapter is devoted to providing a mixed bag of questions and answers covering areas that might have fallen through the cracks.

XX.1: How are command-line parameters obtained?

Answer:

Every time you run a DOS or Windows program, a Program Segment Prefix, or PSP, is created. When the DOS program loader copies the program into RAM to execute it, it first allocates 256 bytes for the PSP, then places the executable in the memory immediately after the PSP. The PSP contains all kinds of information that DOS needs in order to facilitate the execution of the program, most of which do not apply to this FAQ. However, there is at least one piece of data in the PSP that does apply here: the command line. At offset 128 in the PSP is a single byte that contains the number of characters of the command line. The next 127 bytes contain the command line itself. Coincidentally, that is why DOS limits your typing at the DOS prompt to 127 characters—it allocates only that much to hold the command line. Unfortunately, the command-line buffer in the PSP does not contain the name of the executable—it contains only the characters you typed after the executable's name (including the spaces).

For example, if you type

```
XCOPY AUTOEXEC.BAT AUTOEXEC.BAK
```

at the DOS prompt, XCOPY.EXE's PSP command-line buffer will contain

```
AUTOEXEC.BAT AUTOEXEC.BAK
```

assuming that the xcopy program resides in the DOS directory of drive C. It's difficult to see in print, but you should note that the space character immediately after the `XCOPY` word on the command line is also copied into the PSP's buffer.

Another negative side to the PSP is that, in addition to the fact that you cannot find your own program's name, any redirection of output or input noted on the command line is not shown in the PSP's command-line buffer. This means that you also cannot know (from the PSP, anyway) that your program's input or output was redirected.

By now you are familiar with using the `argc` and `argv` argument parameters in your C programs to retrieve the information. But how does the information get from the DOS program loader to the `argv` pointer in your program? It does this in the start-up code, which is executed before the first line of code in your `main()` function. During the initial program execution, a function called `_setargv()` is called. This function copies the program name and command line from the PSP and DOS environment into the buffer pointed to by your `main()` function's `argv` pointer. The `_setargv()` function is found in the *x*LIBCE.LIB file, *x* being S for Small memory model, M for Medium memory model, and L for Large memory model. This library file is automatically linked to your executable program when you build it. Copying the argument parameters isn't the only thing the C start-up code does. When the start-up code is completed, the code you wrote in your `main()` function starts being executed.

OK, that's fine for DOS, but what about Windows? Actually, most of the preceding description applies to Windows programs as well. When a Windows program is executed, the Windows program loader creates a PSP just like the DOS program loader, containing the same information. The major difference is that the command line is copied into the `lpszCmdLine` argument, which is the third (next-to-last) argument in your `WinMain()` function's parameter list. The Windows C library file *x*LIBCEW.LIB contains the start-up function `_setargv()`, which copies the command-line information into this `lpszCmdLine` buffer. Again, the *x* represents the memory model you are using with your program. If you are using QuickC, the start-up code is contained in the *x*LIBCEWQ.LIB library file.

Although the command-line information between DOS and Windows programs is managed in basically the same way, the *format* of the command line arrives in your C program in slightly different arrangements. In DOS, the start-up code takes the command line, which is delimited by spaces, and turns each argument into its own NULL-terminated string. You therefore could prototype `argv` as an array of pointers (`char * argv[]`) and access each argument using an index value of 0 to *n*, in which *n* is the number of arguments in the command line minus one. On the other hand, you could prototype `argv` as a pointer to pointers (`char ** argv`) and access each argument by incrementing or decrementing `argv`.

In Windows, the command line arrives as an `LPSTR`, or `char _far *`. Each argument in the command line is delimited by *spaces*, just as they would appear at the DOS prompt had you actually typed the characters yourself (which is unlikely, considering that this is Windows and they want you to think you are using a Macintosh by double-clicking the application's icon). To access the different arguments of the Windows command line, you must manually walk across the memory pointed to by `lpszCmdLine`, separating the arguments, or use a standard C function such as `strtok()` to hand you each argument one at a time.

If you are adventurous enough, you could peruse the PSP itself to retrieve the command-line information. To do so, use DOS interrupt 21 as follows (using Microsoft C):

```
#include <stdio.h>
#include <dos.h>

main(int argc, char ** argv)
{
    union REGS regs;                 /* DOS register access struct */
    char far * pspPtr;               /* pointer to PSP */
    int cmdLineCnt;                  /* num of chars in cmd line */

    regs.h.ah = 0x62;                /* use DOS interrupt 62 */
    int86(0x21, &regs, &regs);       /* call DOS */
    FP_SEG(pspPtr) = regs.x.bx;      /* save PSP segment */
    FP_OFF(pspPtr) = 0x80;           /* set pointer offset */

    /* *pspPtr now points to the command-line count byte */
    cmdLineCnt = *pspPtr;
}
```

It should be noted that in the Small memory model, or in assembly language programs with only one code segment, the segment value returned by DOS into the BX register is your program's code segment. In the case of Large memory model C programs, or assembly programs with multiple code segments, the value returned is the code segment of your program that contains the PSP. After you have set up a pointer to this data, you can use this data in your program.

Cross Reference:

XX.2: Should programs always assume that command-line parameters can be used?

XX.2: Should programs always assume that command-line parameters can be used?

Answer:

These days, you can usually assume that the command-line parameters can be used by your program. Before DOS 2.0, the command-line information stored in the PSP was slightly different (it *didn't* strip input and output redirection data from the command line). In addition, the data pointed to by `argv[0]` did not reliably contain the executable's pathname until DOS 2.0. The DOS interrupt 62 that retrieves the PSP segment wasn't available (or at least documented) until DOS 3.0. You therefore can at least assume that you can get consistent command-line information on PCs running DOS 3.0 or newer.

After you have determined that you are running DOS 3.0 or greater, you can basically do whatever you want to the command-line data, because the information is placed on the stack for you to play with (via `argv`). Of course, normal data manipulation rules apply to command-line data as they do to all data arriving on the stack. The real problems arise when you don't have `argv` provided by your compiler. For example, you could write your program in assembly language, or in some archaic compiler that does not provide `argc` and `argv`. In these cases, you will have to find some method of retrieving the command-line information yourself. That is where the DOS interrupt 62 comes in handy.

If you use DOS interrupt 62 to retrieve a pointer to the command line, you must be aware that you are pointing to data that is used by DOS and for DOS. Although the data is there for you to see, you should not assume that the data is there for you to alter. If you need to use the command-line information to make decisions at various times throughout your program, you should copy the data into a local buffer before actually using the data. This technique enables you to have complete control of the data without worrying about stepping on DOS's toes. In fact, this applies also to C programs that supply `argv`. It is not uncommon for a function outside `main()` to need access to the command-line data. For it to get access to the data, your `main()` must save it globally, or pass it (once again) on the stack to the function that needs it. It is therefore good programming practice to save the command-line information into a local buffer if you intend to use it.

Cross Reference:

XX.1: How are command-line parameters obtained?

XX.3: What is the difference between "exception handling" and "structured exception handling"?

Answer:

Generally speaking, the difference between a structured exception and exception handling is Microsoft's implementation of exception handlers themselves. So-called "ordinary" C++ exception handling uses three statements added to the C++ language: `try`, `catch`, and `throw`. The purpose of these statements is to allow a piece of software (the exception handler) to attempt a safe bailout of the application that was running when the exception occurred. The exception handler can trap exceptions on any data type, including a C++ class. The implementation of the three statements is based on the ISO WG21/ANSI X3J16 C++ standard for exception handling. Microsoft C++ supports exception handling based on this standard. Note that this standard applies only to C++ and not to C.

On the other hand, structured exception handling is an extension to the Microsoft C/C++ compiler. Its single largest advantage is that it works with either C or C++. Microsoft's structured exception handling design uses two new constructs: `try-except` and `try-finally`. These two constructs are not a subset or superset of the ANSI C++ standard; instead, they are a different implementation of exception handling (leave it to Microsoft to forge ahead on its own). The `try-except` construct is known as *exception handling*, and `try-finally` is known as *termination handling*. The `try-except` statement allows an application to retrieve the state of the machine when the exception occurred. This is very handy for displaying information about the error to the user, or for use while you are debugging your code. The `try-finally` statement enables applications to guarantee execution of cleanup code when normal code execution is interrupted. Although structured exception handling has its advantages, it also has its drawbacks. Because this is not an ANSI standard, code using structured exception handling is not as portable as code using ANSI exception handling. A good rule of thumb is if your application is going to be a C++ program, you are advised to stick to ANSI exception handling (use the `try`, `catch`, and `throw` statements).

Cross Reference:

None.

XX.4: How do you create a delay timer in a DOS program?

Answer:

Fortunately for us programmers, the folks at Microsoft thought it would be a good idea to create a hardware-independent delay timer. The purpose, of course, is to allow a delay of a fixed amount of time, regardless of the speed of the computer on which the program is being run. The following example code demonstrates how to create a delay timer in DOS:

```
#include <stdio.h>
#include <dos.h>
#include <stdlib.h>


void main(int argc, char ** argv)
{
    union REGS regs;
    unsigned long delay;

    delay = atol(argv[1]);    /* assume that there is an argument */

    /* multiply by 1 for microsecond-granularity delay */
    /* multiply by 1000 for millisecond-granularity delay */
    /* multiply by 1000000 for second-granularity delay */
    delay *= 1000000;

    regs.x.ax = 0x8600;
    regs.x.cx = (unsigned int)((delay & 0xFFFF0000L) >> 16);
    regs.x.dx = (unsigned int)(delay & 0xFFFF);

    int86(0x15, &regs, &regs);
}
```

The example uses DOS interrupt 0x15, function 0x86, to perform the delay. The amount of delay is in microseconds. Due to this, the delay function assumes that you might want a really big number, so it expects the high-order 16 bits of the delay value in `CX`, and the low-order 16 bits of the delay value in `DX`. At its maximum, the delay function can stall for more than 4 billion microseconds, or about 1.2 hours.

The example assumes that the delay value will be in microseconds. This version of the example multiplies the delay by one million so that the delay entered on the command line will be turned into seconds. Therefore, a “delay 10” command will delay for 10 seconds.

Cross Reference:

XXI.2: How do you create a delay timer in a Windows program?

XX.5: Who are Kernighan and Ritchie?

Answer:

Kernighan and Ritchie are Brian W. Kernighan and Dennis M. Ritchie, authors of *The C Programming Language*. This book is known affectionately throughout the world as the “K&R Manual,” the “white book,” the “K&R Bible,” and other similar names. The book was originally published by Prentice-Hall in 1978. Dennis developed the C programming language for the UNIX operating system running on the DEC PDP-11 mainframe computer. He and Brian were working for AT&T Bell Laboratories in the early 1970s when both C and the K&R Manual were developed. C was modeled somewhat after the B programming language, written by Ken Thompson in 1970, and BCPL, written by Martin Richards in 1969.

Cross Reference:

None.

XX.6: How do you create random numbers?

Answer:

Well, actually, there’s no such thing as a truly random number generator on a computer. However, a point can be reached where the number repetition pattern is so large that the number appears to be random. That is the ultimate goal of a random number generator. Such a device is called a *pseudo-random number generator*.

There is a lot of theory about how to generate random numbers. I will not discuss in this section the theory and mathematics behind creating random number generators. Entire books have been written dealing with this subject alone. What can be said about any random number generator is that no matter which implementation you use, you must provide the algorithm some value to get it “started.” It helps if this number is also random, or at least pseudo-random. Fast counting registers or shift registers are often used to create this initial number, called a *seed*, that is fed into the generator.

For this book, I will demonstrate the use of the random number generator provided by the C language. Modern C compilers include a pseudo-random number generator function to help you produce a random number, based on an ANSI standard. Both Microsoft and Borland support this standard via the `rand()` and `srand()` functions. Here’s how they work: You provide the seed for the `srand()` function; this seed is an `unsigned int`, so the range is 0 to 65,535. After you have fed the seed to `srand()`, you call `rand()`, which returns a random number (in the range of 0 to 32,767) based on the seed value provided to `srand()`. You can call `rand()` as many times as you want after seeding `srand()`, and you’ll continue to get back random numbers. You can, at any time, seed `srand()` with a different value to further “randomize” the output of `rand()`.

This process sounds simple enough. The problem is that if you feed `srand()` the same seed value each time you call it, you will get back the same series of “random” numbers. For example, you call `srand()` with a seed value of 17. When you call `rand()`, you get the random number 94. Call `rand()` again, and you get 26,602. Call `rand()` a third time, and you get 30,017. Seems fairly random (although this is a painfully small set of data points). If, however, you call `srand()` again, seeding it with 17 again, `rand()` will return 94, 26,602, and 30,017 on its first three calls, along with all the remaining numbers you got back in the first series of calls to `rand()`. Therefore, you still need to seed `srand()` with a random number so that it can produce a random number.

The following example shows a simple, but quite effective, method for generating a fairly random seed value—the time of day.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/timeb.h>

void main( void )
{
    int i;
    unsigned int seedVal;
    struct _timeb timeBuf;

    _ftime(&timeBuf);

    seedVal = ((((((unsigned int)timeBuf.time & 0xFFFF) +
                 (unsigned int)timeBuf.millitm)) ^
                 (unsigned int)timeBuf.millitm));

    srand((unsigned int)seedVal);

    for(i = 0; i < 10; ++i)
        printf("%6d\n", rand());
}
```

The function calls `_ftime()` to retrieve the current time of day in seconds elapsed since January 1, 1970 (no kidding), placed into the `timeBuf.time` structure member. After the call, the `timeBuf` structure also contains the number of milliseconds that have elapsed in the current second in the `millitm` member. Note that in DOS, `millitm` actually holds the number of *hundredths* of a second that have elapsed in the current second. The number of milliseconds is added to the elapsed time in seconds, and the total is XORed with the millisecond count. You could apply many more logical mathematical functions to these two structure members to control the range of `seedVal` and further its apparent randomness, but this example is sufficient.

Note that in the preceding example, the output of `rand()` has not been scaled to a specific range. Suppose that you want to pretend to create a lottery number-picking machine whose values ranged from 1 to 44. You could simply ignore any output from `rand()` that did not fall into this range, but it could take a long time before you acquired the necessary six lottery numbers. Instead, you can scale the output from `rand()` to any numeric range you want. Assume that you have produced a satisfactory random number generator that provides a random number in the range of 0 to 32,767 (as in the case of the earlier example) and that you want to scale the output down to 1 to 44. The following example shows how to accomplish this task:

```
int i, k, range;
int min, max;
double j;

min = 1;           /* 1 is the minimum number allowed */
max = 44;          /* 44 is the maximum number allowed */
range = max - min;    /* r is the range allowed: 1 to 44 */

i = rand();   /* use the above example in this slot */

/* Normalize the rand() output (scale to 0 to 1) */
/* RAND_MAX is defined in stdlib.h */
j = ((double)i / (double)RAND_MAX);

/* Scale the output to 1 to 44 */
i = (int)(j * (double)range);
i += min;
```

This example places a restriction on the random output to a range from 1 to 44. Here's what the function does: It gets a random number whose range is from 0 to `RAND_MAX` (32,767) and divides that random number by `RAND_MAX`. This process produces a *normalized* value—that is, a value whose range is 0 to 1. Next, the normalized value is scaled up by the range of allowed values (43 in this case—44 minus 1). This produces a value from 0 to 43. Next, the minimum amount allowed is added to this number to place the scaled value in the proper range—1 to 44. To experiment, replace the `min` and `max` numbers with different values, and you'll see that the example properly scales the random number to the new `min` and `max` values.

Cross Reference:

None.

XX.7: When should a 32-bit compiler be used?

Answer:

A 32-bit compiler should be used on a 32-bit operating system. The 32-bit compiler creates 32-bit programs that run your PC much faster than 16-bit programs—which is why 32-bit *anything* is hot.

With all the different versions of Microsoft Windows out there, which compiler is best for which operating system? This passage looks at what Microsoft offers, and assigns the correct compiler to the correct operating system. Windows 3.1 and Windows for Workgroups 3.11 are 16-bit operating systems; Microsoft Visual C++ 1.x is a 16-bit compiler. The compiler produces code that will run on Windows 3.1. Microsoft Windows NT and Windows 95 are 32-bit operating systems; Visual C++ 2.0, the latest compiler from Microsoft, is a 32-bit compiler created to produce 32-bit code for these operating systems. The 16-bit programs that Visual 1.x produces will run on Windows NT and Windows 95 as well as Windows 3.1.

The opposite is *not* true, however—32-bit code produced by Visual 2.0 will not run on Windows 3.1. This fact presents a problem for Microsoft, which wants everyone to use its 32-bit compiler but has 60 million PCs out there running a version of Windows that can't run these new 32-bit programs. To get around this obstacle, Microsoft created a translation library called *Win32s* that performs the 32- to 16-bit "thunking," as it is called, to allow 32-bit programs produced by Visual C++ 2.0 to run on Windows 3.1 and Windows for Workgroups. Win32s is specifically designed to run on Windows 3.1 (and WFW)—it is not meant for Windows 95 and NT because they do not need to "thunk" to run 32-bit code. Using Win32s, it is possible to produce a single program using Visual C++ 2.0 that will run on Windows 3.1 (and WFW) as well as Windows NT.

The only remaining gotcha is the compilers themselves. Visual C++ 1.x is a 16-bit Windows program—it will run on Windows 95 and NT and will produce the same 16-bit program as if it were running on Windows 3.1. However, Visual C++ 2.0, being a 32-bit Windows program, will not run on Windows 3.1. It won't even run if you install Win32s, because Microsoft conveniently makes sure you are running Windows 95 or NT before Visual 2.0 will start.

To summarize, run Visual C++ 1.x (version 1.51 is the latest) on Windows 3.1, Windows for Workgroups, Windows NT, or Windows 95 to create 16-bit programs that will run on all versions of Windows; run Visual C++ 2.0 to create 32-bit programs that will run fastest on Windows 95 and NT but will also run very well on Windows 3.1 (and WFW).

For you Borland C/C++ users, Borland's Turbo C++ Version 3.1 is the latest 16-bit compiler, and Borland C++ Version 4.5 is the 32-bit Windows compiler (note that the 32-bit compiler doesn't have the "Turbo" moniker). Both compilers contain the compiler, Borland's OWL C++ classes, and an excellent integrated debugger.

Cross Reference:

None.

XX.8: How do you interrupt a Windows program?

Answer:

There is no trivial way to interrupt a Windows application so that you can perform a necessary task (if that is your goal). Windows 3.x is a nonpreemptive multitasking operating system. Put another way, it is a *cooperative* multitasking operating system. It is cooperative partly because there is no way to simply steal time away from a Windows program that is currently in control of the processor. If you look at the Windows API, you'll see functions such as `PeekMessage()` and `WaitMessage()`. As the Microsoft help documentation says concerning these functions:

> The `GetMessage`, `PeekMessage`, and `WaitMessage` functions yield control to other applications. Using these functions is the only way to allow other applications to run. Applications that do not call any of these functions for long periods prevent other applications from running.

All that having been said, it still can be done. One method is to create a timer in your Windows program that "goes off" every so often. As long as your program is alive, you will get time to perform whatever task you want to when the timer goes off. However, this action is not technically *interrupting* a program in process—it is simply using the cooperative multitasking features of Windows. If you need to interrupt a Windows program, you can do so with a *filter function* (also called a *hook function*). A filter function in Windows is analogous to an interrupt service routine in DOS (see FAQs XX.12 and XX.17). Using a filter function, you can hook certain Windows events and perform tasks when that event occurs. In fact, you can use a filter function to monitor nearly every message that exists in Windows using one or more of the available hooks. This example, however, uses the keyboard hook because it enables you to interrupt a program at will by entering a specific key combination. The following example hooks the keyboard event chain and puts up a message box when the Ctrl-Alt-F6 key is pressed. This method works regardless of the application that is currently running.

```
#include <dos.h>
#include <windows.h>

DWORD FAR PASCAL __loadds KeyBoardProc(int, WORD, DWORD);
static FARPROC nextKeyboardFilter = NULL;

BOOL shiftKeyDown, ctrlKeyDown;

#define REPEAT_COUNT    0x000000FF      /* test key repeat */
#define KEY_WAS_UP      0x80000000      /* test WM_KEYUP */
#define ALT_KEY_DOWN    0x20000000      /* test Alt key state */

#define EAT_THE_KEY         1           /* swallow keystroke */
```

```
#define SEND_KEY_ON         0           /* act on keystroke */

BOOL useAltKey = TRUE;                  /* use Alt key in sequence */
BOOL useCtrlKey = TRUE;                 /* also use Ctrl key */
BOOL useShiftKey = FALSE;               /* don't use Shift key */

/* Entry point into the DLL. Do all necessary initialization here */

int FAR PASCAL LibMain(hModule, wDataSeg, cbHeapSize, lpszCmdLine)
HANDLE  hModule;
WORD    wDataSeg;
WORD    cbHeapSize;
LPSTR   lpszCmdLine;
{

    /* initialize key state variables to zero */
    shiftKeyDown = 0;
    ctrlKeyDown = 0;

    return 1;
}

/* The keyboard filter searches for the hotkey key sequence.
   If it gets it, it eats the key and displays a message box.
   Any other key is sent on to Windows. */

DWORD FAR PASCAL __loadds
KeyBoardProc(int nCode, WORD wParam, DWORD lParam)
{
    BOOL fCallDefProc;
    DWORD dwResult = 0;

    dwResult = SEND_KEY_ON;    /* default to send key on */
    fCallDefProc = TRUE;           /* default to calling DefProc */

switch(nCode){
    case HC_ACTION:
    case HC_NOREMOVE:

        /* If key is Shift, save it */
        if(wParam == (WORD)VK_SHIFT){
            shiftKeyDown = ((lParam & KEY_WAS_UP) ? 0 : 1);
            break;
        }

        /* If key is Ctrl, save it */
        else if(wParam == (WORD)VK_CONTROL){
            ctrlKeyDown = ((lParam & KEY_WAS_UP) ? 0 : 1);
            break;
        }

        /* If key is the F6 key, act on it */
        else if(wParam == (WORD)VK_F6){

            /* Leave if the F6 key was a key release and not press */
            if(lParam & KEY_WAS_UP) break;

            /* Make sure Alt key is in desired state, else leave */
            if( (useAltKey) && !(lParam & ALT_KEY_DOWN) ){
                break;
            }
            else if( (!useAltKey) && (lParam & ALT_KEY_DOWN) ){
                break;
            }

            /* Make sure Shift key is in desired state, else leave */
            if(useShiftKey && !shiftKeyDown){
                break;
            }
            else if(!useShiftKey && shiftKeyDown){
                break;
            }

            /* Make sure Ctrl key is in desired state, else leave */
            if(useCtrlKey && !ctrlKeyDown){
                break;
            }
            else if(!useCtrlKey && ctrlKeyDown){
                break;
            }

            /* Eat the keystroke, and don't call DefProc */
            dwResult = EAT_THE_KEY;
            fCallDefProc = FALSE;

            /* We made it, so Ctrl-Alt-F6 was pressed! */
            MessageBox(NULL, (LPSTR)"You pressed Ctrl-Alt-F6!",
                             (LPSTR)"Keyboard Hook", MB_OK);
            break;
            }

        default:
            fCallDefProc = TRUE;
            break;
    }

if( (nCode < 0) || (fCallDefProc && (nextKeyboardFilter != NULL)))
    dwResult = DefHookProc(nCode, wParam, lParam,
                                        &nextKeyboardFilter);

    return(dwResult);
}

/* This function is called by the application to set up or tear
   down the filter function hooks. */

void FAR PASCAL
SetupFilters(BOOL install)
{
    if(install){
        nextKeyboardFilter = SetWindowsHook(WH_KEYBOARD,
                                        (FARPROC)KeyBoardProc);
    }
    else{
        UnhookWindowsHook(WH_KEYBOARD, (FARPROC)KeyBoardProc);
        nextKeyboardFilter = NULL;
    }
}
```

Microsoft strongly recommends placing filter functions in a DLL rather than your application (notice the presence of a `LibMain()` and the lack of a `WinMain()`). To complete this application, you need to write an ordinary Windows application that calls the `SetupFilters()` function with `TRUE` as the argument to start monitoring keystrokes, and `FALSE` as the argument to stop monitoring keystrokes. While your application is alive and you have called `SetupFilters(TRUE)`, the callback function `KeyBoardProc()` is receiving all keystrokes, even while you are running other Windows applications. If you press Ctrl-Alt-F6, a small message box appears on-screen informing you that you pressed those keys. Presto, you have just interrupted whatever Windows application was running at the time you pressed those keys!

Note that the keyboard filter function will not receive keystrokes while in a DOS shell. However, the filter function will receive, and can interrupt, a system modal dialog box like the one that asks whether you really want to exit Windows.

Cross Reference:

XX.12: How can I pass data from one program to another?

XX.17: Can you disable warm boots (Ctrl-Alt-Del)?

XXI.10: What is dynamic linking?

XX.9: Why should I use static variables?

Answer:

Static variables are excellent for use as a local variable that does not lose its value when the function exits. For example, you might have a function that gets called numerous times, and that part of the function's job is to count how many times it gets called. You cannot accomplish this task with a simple local variable because it will be uninitialized each time the function is entered. If you declare your counter variable as static, its current value will be maintained, just as with a global variable.

So why not just use a global variable instead? You could, and there's nothing wrong with using a global variable. The problem with using global variables is that maintaining a program with lots of global variables becomes cumbersome, particularly if you have numerous functions that uniquely access one global variable. Again, there's nothing wrong with doing it that way—the issue simply becomes one of good code design and readability. By declaring such variables as static, you are informing yourself (or another person who might be reading your code) that this variable is local but is treated like a global (maintains its value). If the static variable was instead declared as a global, the person reading the code must assume that the global is accessed in many places when in fact it might not be.

In summary, the reason you should use static variables is that it is good programming practice when you need a localized variable that maintains its state.

Cross Reference:

II.17: Can `static` variables be declared in a header file?

XX.10: How can I run another program after mine?

Answer:

Of course, the easiest way to run another program after yours is to put them both in a batch file, one after the other. When you run the batch file, the programs listed will execute in order. But you already knew that.

There is no specific method for performing a "when my program is finished, run this one" function call in C or in DOS. However, C provides two sets of functions that will allow a program at any time to run another program that basically ends the first program upon execution of the second. If you were to put such a call at the end of your first application, you could do exactly what you need. The two functions provided are `exec()` and `spawn()`. In reality, each function is actually a family of functions, each function allowing a unique twist over the other functions in the family. The `exec` family of functions is `execl()`, `execle()`, `execlp()`, `execlpe()`, `execv()`, `execve()`, `execvp()`, and `execvpe()`. The following list shows what the `e`, `l`, `p`, and `v` additions mean to the function:

`e`	An array of pointers to environment parameters is explicitly passed to the child process.
`l`	Command-line arguments are passed individually to the executable.
`p`	Uses the `PATH` environment variable to find the file to be executed.
`v`	Command-line arguments are passed to the executable as an array of pointers.

Which combination of options you choose in your application is entirely up to you and the needs of the application you want to launch. Following is an example of a program that calls another application whose name is specified on the command line:

```
#include <stdio.h>
#include <process.h>

char *envString[] = {                /* environment for the app */
   "COMM_VECTOR=0x63",               /* communications vector */
   "PARENT=LAUNCH.EXE",              /* name of this app */
   "EXEC=EDIT.COM",                  /* name of app to exec */
   NULL};                            /* must be NULL-terminated */

void
main(int argc, char ** argv)
{

   /* Call the one with variable arguments and an environment */
   _execvpe("EDIT.COM", argv, envString);

   printf("If you can read this sentence, the exec didn't happen!\n");
}
```

In the preceding short example, `_execvpe()` is called to execute EDIT.COM, the DOS file editor. The arguments to `EDIT` come from the command line to the example. After the `exec` has occurred, the original application has gone away; when `EDIT` exits, you are returned to the DOS prompt. If the `printf()` statement is displayed on-screen, something went awry in the `exec` attempt, because you will not get there if the `exec` succeeds. Note that the environment variables for EDIT.COM are completely meaningless. However, if you *did* `exec` a program that needed the environment variables, they would be available for use by the application.

There is a second way to accomplish this same task—`spawn()`. The spawn function family is `spawnl()`, `spawnle()`, `spawnlp()`, `spawnlpe()`, `spawnv()`, `spawnve()`, `spawnvp()`, and `spawnvpe()`. The `e`, `l`, `p`, and `v` appendages mean the same as they do with `exec`. In fact, this function family is identical to `exec` except for one small difference—`spawn` can either launch another application while killing the original, or launch another application and return to the original when the second application is complete. The argument list for the `spawn` functions is identical to that for `exec` except that one additional argument is required: you must use `_P_OVERLAY` (original application dies) or `_P_WAIT` (return to original when finished) as the first argument to the `spawn` functions. The following example demonstrates the same task as the preceding example:

```
#include <stdio.h>
#include <process.h>

char *envString[] = {            /* environment for the app */
"COMM_VECTOR=0x63",              /* communications vector */
"PARENT=LAUNCH.EXE",             /* name of this app */
"EXEC=EDIT.COM",                 /* name of app to exec */
NULL};                           /* must be NULL-terminated */

void
main(int argc, char ** argv)
{

   /* Call the one with variable arguments and an environment */
   _spawnvpe(_P_OVERLAY, "EDIT.COM", argv, envString);

   printf("If you can read this sentence, the exec didn't happen!\n");

}
```

The only difference here is the name change from `exec` to `spawn`, and the additional *mode* argument. What is nice about `spawn`'s capability to overlay versus wait is the ability to make a runtime decision regarding waiting or leaving during the `spawn`. In fact, the `_P_WAIT` argument answers the next FAQ.

Cross Reference:

XX.11: How can I run another program during my program's execution?

XX.11: How can I run another program during my program's execution?

Answer:

As seen in the preceding example, the `spawn` family of functions allows one application to start another application, then return to the original application when the first one is finished. Read FAQ XX.10 for a good background dose of `spawn` and for example code (all you have to do is change `_P_OVERLAY` to `_P_WAIT` and you're done).

However, there is another way to accomplish this task that needs to be mentioned. This other method involves the `system()` call. The `system()` function is similar to, but still different from, the `exec` or `spawn` functions. In addition to suspending the current application to execute the new one (instead of killing it), `system()` launches the COMMAND.COM command interpreter (or whatever command interpreter is running on your machine) to run the application. If COMMAND.COM or its equivalent cannot be found, the application is not executed (this is not the case with `exec` and `spawn`). The following example is yet another version of a call to EDIT.COM to open a file, whose name arrives from the example program's command line:

```
#include <stdio.h>
#include <process.h>
#include <stdlib.h>

char argStr[255];

void
main(int argc, char ** argv)
{
   int ret;

   /* Have EDIT open a file called HELLO if no arg given */
   sprintf(argStr, "EDIT %s", (argv[1] == NULL ? "HELLO" : argv[1]));

   /* Call the one with variable arguments and an environment */
   ret = system(argStr);

   printf("system() returned %d\n", ret);
}
```

As it was with the earlier example (using `_P_WAIT`), the `printf()` statement after the `system()` call is executed because the initial program was merely suspended and not killed. In every case, `system()` returns a value that signifies success or failure to run the specified application. It does *not* return the return code from the application itself.

Cross Reference:

XX.10: How can I run another program after mine?

XX.12: How can I pass data from one program to another?

Answer:

You can accomplish this task in a couple of basic ways—you can pass the data via a file or via memory. The steps for any of the methods are fairly straightforward: you define where the data is to be placed, how to get it there, and how to notify the other program to get or set the data; then you get or set the data in that location. Although the file technique is simple to define and create, it can easily become slow (and noisy). This answer will therefore concentrate on the memory data-transfer technique. The parts of the process will be detailed one at a time:

Define where the data is to be placed. When you write the two applications (it takes two to share), you must build into them a way to know where the data is going to be when you need to retrieve it. Again, there are several ways you can solve this part. You can have a fixed data buffer internal to one (or each of the programs) and pass a pointer to the buffer back and forth. You could dynamically allocate the memory and pass a pointer to the data back and forth. If the data is small enough, you could pass the information through the CPU's general purpose registers (this possibility is unlikely due to the pitifully few number of registers in the x86 architecture). The most flexible and modular method is to dynamically allocate the memory.

Define how to get the data there. This part is straightforward—you use `_fmemcpy()` or an equivalent memory copy routine. This naturally applies for both getting and setting the data.

Define how to notify the other program. Because DOS is not a multitasking operating system, it is obvious that one (or both) of the programs must have some part of the software already resident in memory that can accept the call from the other program. Again, several choices are available. The first application could be a device driver that is referenced in CONFIG.SYS and loaded at boot time. Or it could be a TSR (terminate-and-stay-resident) application that leaves the inter-application part of the program resident in memory when it exits. Another option is to use the `system()` or `spawn()` calls (see FAQ XX.11) to start the second application from within the first one. Which option you select depends on your needs. Because data passing to and from a DOS device driver is already well documented, and `system()` and `spawn()` calls were documented earlier, I'll describe the TSR method.

The following example code is a complete program but is admittedly thin for the purposes of grasping only the critical pieces of the process (see FAQ XX.15 regarding example code). The following example shows a TSR that installs an interrupt service routine at interrupt 0x63, then calls the terminate-and-stay-resident `exit` function. Next, another program is executed that simply initiates an interrupt call to 0x63 (much like you make DOS int21 calls) and passes `"Hello World"` to that program.

```
#include <stdlib.h>
#include <dos.h>
#include <string.h>

void SetupPointers(void);
void OutputString(char *);

#define STACKSIZE       4096

unsigned int _near OldStackPtr;
unsigned int _near OldStackSeg;
unsigned int _near MyStackOff;
unsigned int _near MyStackSeg;
unsigned char _near MyStack[STACKSIZE];
unsigned char _far * MyStackPtr = (unsigned char _far *) MyStack;

unsigned short AX, BX, CX, DX, ES;

/* My interrupt handler */
void _interrupt _far _cdecl NewCommVector(
        unsigned short es, unsigned short ds, unsigned short di,
        unsigned short si, unsigned short bp, unsigned short sp,
        unsigned short bx, unsigned short dx, unsigned short cx,
        unsigned short ax, unsigned short ip, unsigned short cs,
        unsigned short flags);

/* Pointers to the previous interrupt handler */
void (_interrupt _far _cdecl * comm_vector)();

union REGS regs;
struct SREGS segregs;

#define COMM_VECTOR         0x63    /* Software interrupt vector */

/* This is where the data gets passed into the TSR */
```

```
char _far * callerBufPtr;
char localBuffer[255];              /* Limit of 255 bytes to transfer */
char _far * localBufPtr = (char _far *)localBuffer;

unsigned int ProgSize = 276;     /* Size of the program in paragraphs */

void
main(int argc, char ** argv)
{
    int i, idx;

    /* Set up all far pointers */
    SetupPointers();

    /* Use a cheap hack to see if the TSR is already loaded
       If it is, exit, doing nothing */
    comm_vector = _dos_getvect(COMM_VECTOR);
    if(((long)comm_vector & 0xFFFFL) ==
                                 ((long)NewCommVector & 0xFFFFL)){
        OutputString("Error: TSR appears to already be loaded.\n");
        return;
    }

    /* If everything's set, then chain in the ISR */
    _dos_setvect(COMM_VECTOR, NewCommVector);

    /* Say we are loaded */
    OutputString("TSR is now loaded at 0x63\n");

    /* Terminate, stay resident */
    _dos_keep(0, ProgSize);
}

/* Initializes all the pointers the program will use */

void
SetupPointers()
{
    int idx;

    /* Save segment and offset of MyStackPtr for stack switching */
    MyStackSeg = FP_SEG(MyStackPtr);
    MyStackOff = FP_OFF(MyStackPtr);

    /* Initialize my stack to hex 55 so I can see its footprint
       if I need to do debugging */
    for(idx=0;idx<STACKSIZE; idx++){
        MyStack[idx] = 0x55;
    }

}
void _interrupt _far _cdecl NewCommVector(
        unsigned short es, unsigned short ds, unsigned short di,
        unsigned short si, unsigned short bp, unsigned short sp,
        unsigned short bx, unsigned short dx, unsigned short cx,
        unsigned short ax, unsigned short ip, unsigned short cs,
        unsigned short flags)
{

    AX = ax;
    BX = bx;
    CX = cx;
    DX = dx;
    ES = es;

    /* Switch to our stack so we won't run on somebody else's */

    _asm{
                                    ; set up a local stack
        cli                         ; stop interrupts
        mov     OldStackSeg,ss      ; save stack segment
        mov     OldStackPtr,sp      ; save stack pointer (offset)
        mov     ax,ds               ; replace with my stack s
        mov     ss,ax               ; ditto
        mov     ax,MyStackOff       ; replace with my stack ptr
        add     ax,STACKSIZE - 2    ; add in my stack size
        mov     sp,ax               ; ditto
        sti                         ; OK for interrupts again
    }

    switch(AX){
        case 0x10:      /* print string found in ES:BX */
            /* Copy data from other application locally */
            FP_SEG(callerBufPtr) = ES;
            FP_OFF(callerBufPtr) = BX;
            _fstrcpy(localBufPtr, callerBufPtr);

            /* print buffer 'CX' number of times */
            for(; CX>0; CX--)
                OutputString(localBufPtr);
            AX = 1;     /* show success */
            break;

        case 0x30:      /* unload; stop processing interrupts */
            _dos_setvect(COMM_VECTOR, comm_vector);
            AX = 2;     /* show success */
            break;
default:
            OutputString("Unknown command\r\n");
            AX = 0xFFFF;        /* unknown command -1 */
            break;
    }

    /* Switch back to the caller's stack */

    _asm{
        cli                         ; turn off interrupts
        mov     ss,OldStackSeg      ; reset old stack segment
        mov     sp,OldStackPtr      ; reset old stack pointer
        sti                         ; back on again
    }

    ax = AX;            /* use return value from switch() */

}

/* avoids calling DOS to print characters */
void
OutputString(char * str)
```

```
{
int i;

    regs.h.ah = 0x0E;
    regs.x.bx = 0;

    for(i=strlen(str); i>0; i--, str++){
        regs.h.al = *str;
        int86(0x10, &regs, &regs);
    }
}
```

The preceding section is the TSR portion of the application pair. It has a function, `NewCommVector()`, installed at interrupt 0x63 (0x63 is typically an available vector). After it is installed, it is ready to receive commands. The `switch` statement is where the incoming commands are processed and actions are taken. I arbitrarily chose 0x10 to be the command “copy data from `ES:BX` and print that data to the screen `CX` number of times.” I chose 0x30 to be “unhook yourself from 0x63 and stop taking commands.” The next piece of code is the other program—the one that sends the commands to 0x63. (Note that it must be compiled in the Large memory model.)

```
#include <stdlib.h>
#include <dos.h>

#define COMM_VECTOR     0x63

union REGS regs;
struct SREGS segregs;

char buffer[80];
char _far * buf = (char _far *)buffer;

main(int argc, char ** argv)
{
    int cnt;

    cnt = (argc == 1 ? 1 : atoi(argv[1]));

    strcpy(buf, “Hello There\r\n”);

    regs.x.ax = 0x10;
    regs.x.cx = cnt;
    regs.x.bx = FP_OFF(buf);
    segregs.es = FP_SEG(buf);

    int86x(COMM_VECTOR, &regs, &regs, &segregs);

    printf(“ISR returned %d\n”, regs.x.ax);

}
```

You might think that this short program looks just like other programs that call `int21` or `int10` to set or retrieve information from DOS. If you did think that, you would be right. The only real difference is that you use 0x63 rather than 0x21 or 0x10 as the interrupt number. This program simply calls the TSR and requests it to print the string pointed to by `es:bx` on-screen. Lastly, it prints the return value from the interrupt handler (the TSR).

By printing the string `"Hello There"` on-screen, I have achieved all the necessary steps for communicating between two applications. What is so cool about this method is that I have only scratched the surface of possibilities for this method. It would be quite easy to write a third application that sends a command such as "give me the last string that you were asked to print." All you have to do is add that command to the `switch` statement in the TSR and write another tiny program that makes the call. In addition, you could use the `system()` or `spawn()` calls shown in FAQ XX.11 to launch the TSR from within the second example program. Because the TSR checks to see whether it is already loaded, you can run the second program as often as you want, and only one copy of the TSR will be installed. You can use this in all your programs that "talk" to the TSR.

Several assumptions were made during the creation of the TSR. One assumption is that there is no important interrupt service routine already handling interrupt 0x63. For example, I first ran the program using interrupt 0x67. It loaded and worked, but I could no longer compile my programs because interrupt 0x67 is also hooked by the DOS extender used by Microsoft to run the C compiler. After I sent command 0x30 (unload yourself), the compiler ran flawlessly again because the DOS extender's interrupt handler was restored by the TSR.

Another assumption was in the residency check. I assumed that there will never be another interrupt handler with the same near address as `NewCommVector()`. The odds of this occurring are extremely small, but you should know this method is not foolproof. Although I switched stacks in `NewCommVector()` to avoid running on the calling program's stack, I assume that it is safe to call any function I want. Note that I avoided calling `printf` because it is a memory hog and it calls DOS (`int21`) to print the characters. At the time of the interrupt, I do not know whether DOS is available to be called, so I can't assume that I can make DOS calls. Note that I *can* make calls (like the one to `OutputString()`) that do not use the DOS `int21` services to perform the desired task. If you must use a DOS service, you can check a DOS busy flag to see whether DOS is callable at the time. A note about `_dos_keep()`. It requires you to tell it how many paragraphs (16-byte chunks) of data to keep in memory when exiting. For this program, I give it the size of the entire executable in paragraphs, rounded up a bit (276). As your program grows, you must also grow this value, or strange things will happen.

Cross Reference:

XX.10: How can I run another program after mine?

XX.11: How can I run another program during my program's execution?

XX.15: Some of your examples are not very efficient. Why did you write them so badly?

XX.13: How can I determine which directory my program is running from?

Answer:

Fortunately for us DOS programmers, the DOS program loader provides the full pathname of the executable file when it is run. This full pathname is provided via the `argv[0]` pointer, which is pointed to by the `argv` variable in your `main()` function. Simply strip off the name of your program, and you have the directory from which your application is running. A bit of example code demonstrates this:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void main(int argc, char ** argv)
{
    char execDir[80];
    int i, t;

    /* set index into argv[0] to slash character prior to appname */
    for(i = (strlen(argv[0]) - 1);
           ( (argv[0][i] != '/') && (argv[0][i] != '\\') ); --i);

    /* temporarily truncate argv[] */
    t = argv[0][i];
    argv[0][i] = 0;

    /* copy directory path into local buffer */
    strcpy(execDir, argv[0]);

    /* put back original character for sanity's sake */
    argv[0][i] = t;
}
```

Cross Reference:

XX.1: How are command-line parameters obtained?

XX.14: How can I locate my program's important files (databases, configuration files, and such)?

Answer:

DOS provides a pair of functions that enable you to search a directory for a file of any type. You can search for files that are normal, archive, hidden, system, read-only, directories, and even volume labels. Following is a short example of how to search for a particular file in the current directory:

```
#include <stdio.h>
#include <dos.h>

void main( void )
{
    struct _find_t  myFile;

    _dos_findfirst( "MYFILE.INI", _A_NORMAL, &myFile );

    while(_dos_findnext(&myFile) == 0)
       printf("Found file %s of size %s\n", myFile.name,
                                            myFile.size);
}
```

This example demonstrates how the `_dos_findfirst()` and `_dos_findnext()` functions work. You can go into a directory and use these two functions as shown to find a file of a particular name. These functions also allow the * and ? wildcards. They will return every file in a directory if you use *.* as the filename. To find every file on the hard drive, place the preceding example code in a recursive function that will go into subdirectories to search for the specified file.

Cross Reference:

None.

XX.15: Some of your examples are not very efficient. Why did you write them so badly?

Answer:

Although some of the examples are complete programs because they are tiny, it is not necessary, nor is it practical, to display complete, fully documented programs. They are distracting in that they break up the flow of the book, and they do not help you in your basic goal, which I hope is to attain specific, informative, but brief answers to your questions.

Cross Reference:

XX.12: How can I pass data from one program to another?

XX.14: How can I locate my program's important files (databases, configuration files, and such)?

XX.16: How do you disable Ctrl-Break?

Answer:

There are several ways to disable the Ctrl-Break function. I will discuss two popular techniques for accomplishing this task.

The first technique is to use DOS to disable Ctrl-Break. DOS interrupt 0x21, function 0x33 enables you to get and set the Ctrl-Break check flag, that is, the flag that tells DOS whether to process the Ctrl-Break keys when they are pressed. The following example illustrates this method:

```
#include <stdio.h>
#include <dos.h>


void main(int argc, char ** argv)
{
    union REGS regs;
    int ctrlBreakFlag;

    /* find out the current state of the flag */
    regs.x.ax = 0x3300;           /* subfunction 0 gets flag state */
    int86(0x21, &regs, &regs);
    ctrlBreakFlag = regs.h.dl;    /* save flag value from DL */

    /* set the state of the flag to disable Ctrl-Break */
    regs.x.ax = 0x3301;
    regs.h.dl = 0;              /* disable checking */
    int86(0x21, &regs, &regs);  /* subfunction 1 sets flag state */

}
```

In the preceding example, DOS was called to query the current state of the Ctrl-Break check flag, which was saved into `ctrlBreakFlag`. The return value from the DOS call, found in DL, will be 0 if Ctrl-Break checking is disabled, and 1 if checking is enabled. Next, the code clears DL and calls the DOS Set Ctrl-Break flag function to disable Ctrl-Break checking. This will be true until it is reset again by another call to this function. Not shown in the preceding example is subfunction 02 (AX = 0x3302), which simultaneously gets and sets the state of the Ctrl-Break flag. To perform this task, put 0x3302 into AX, put the desired Ctrl-Break state in DL, do the interrupt, and save the previous state from DL into `ctrlBreakFlag`.

The second major method is to use a *signal*. A signal is a carryover from the old UNIX days (hey, I'm not that old!). The purpose of the signal function is to allow the programmer to be called when certain events occur. One of these events is the user interruption, or Ctrl-Break in DOS-land. The next example demonstrates how to use the Microsoft `signal()` function to trap and act on the Ctrl-Break event (assume that Ctrl-Break checking is enabled):

```
#include  <stdio.h>
#include  <signal.h>

int exitHandler(void);

int main(int argc, char ** argv)
{
    int  quitFlag = 0;

    /* Trap all Ctrl-Breaks */
    signal(SIGINT, (void (__cdecl *)(int))exitHandler);

    /* Sit in infinite loop until user presses Ctrl-Break */
    while(quitFlag == 0)
        printf("%s\n", (argc > 1) ? argv[1] : "Waiting for Ctrl-Break");
}

/* Ctrl-Break event handler function */

int exitHandler()
{
    char  ch;

    /* Disable Ctrl-Break handling while inside the handler */
    signal(SIGINT, SIG_IGN);

    /* Since it was an "interrupt," clear keyboard input buffer */
    fflush( stdin );

    /* Ask if user really wants to quit program */
    printf("\nCtrl-Break occurred. Do you wish to exit this program?
    ➥(Y or N) ");

    /* Flush output buffer as well */
    fflush(stdout);

    /* Get input from user, print character and newline */
    ch = getche();
    printf("\n");

    /* If user said yes, leave program */
    if(toupper(ch) == 'Y')
        exit(0);

    /* Reenable Ctrl-Break handling */
    signal(SIGINT, (void (__cdecl *)(int))exitHandler);

    return(0);
}
```

The beauty of this example is that you have a function that gets called every time Ctrl-Break is pressed. This means you have the choice of what you want to do. You can ignore the event, which is essentially "disabling" Ctrl-Break. Or you can act on it in any way you want. Another advantage to this method is that when your program exits, normal operation of Ctrl-Break resumes without manual intervention.

Cross Reference:

XX.17: Can you disable warm boots (Ctrl-Alt-Delete)?

XX.17: Can you disable warm boots (Ctrl-Alt-Delete)?

Answer:

Yes. Disabling warm boots is not particularly easy to figure out, but it is not difficult to actually code. To trap a Ctrl-Alt-Delete key sequence, you must trap the keyboard's *interrupt service routine* (ISR). From a high-level perspective, it works like this: You monitor (trap) all keystrokes, waiting for the dreaded Ctrl-Alt-Delete combination. If the keystroke is not Ctrl-Alt-Delete, you pass the keystroke on to the "computer" (remember, I'm speaking in high-level terms). When you see the Ctrl-Alt-Delete combination, you can swallow it (remove it from the keyboard's character buffer) or change it to some other keystroke (one that your program might be expecting to mean that the user just tried to reboot the machine). When your program is finished, it stops monitoring the keystrokes and resumes normal operation. This is how TSRs work—they chain themselves to the keyboard and other interrupts so that they can know when to pop up and do that voodoo that they do so well.

"So how do you do this?" you might ask. With a C program, of course. The next example shows a simplified form of trapping all Ctrl-Alt-Delete keystrokes and doing "nothing" when that combination is pressed:

```
#include <stdlib.h>
#include <dos.h>

/* function prototypes */
void (_interrupt _far _cdecl KbIntProc)(
        unsigned short es, unsigned short ds, unsigned short di,
        unsigned short si, unsigned short bp, unsigned short sp,
        unsigned short bx, unsigned short dx, unsigned short cx,
        unsigned short ax, unsigned short ip, unsigned short cs,
        unsigned short flags);

void (_interrupt _far _cdecl * OldKbIntProc)(void);
```

```
unsigned char far * kbFlags;    /* pointer to keyboard flags */

int key_char, junk;             /* miscellaneous variables */

/* keyboard scancode values */
#define ALT         0x8
#define CTRL        0x4
#define KEY_MASK    0x0F

#define DELETE      0x53

void
main(int argc, char ** argv)
{
    int i, idx;

    /* Save old interrupt vectors */
    OldKbIntProc = _dos_getvect(0x9);

    /* Set pointer to keyboard flags */
    FP_SEG(kbFlags) = 0;
    FP_OFF(kbFlags) = 0x417;

    /* Add my ISR to the chain */
    _dos_setvect(0x9, KbIntProc);

    /* Print something while user presses keys... */
    /* Until ESCAPE is pressed, then leave */
    while(getch() != 27){
        printf("Disallowing Ctrl-Alt-Delete...\n");
    }

    /* Remove myself from the chain */
    _dos_setvect(0x9, OldKbIntProc);

}


void _interrupt _far _cdecl KbIntProc(
        unsigned short es, unsigned short ds, unsigned short di,
        unsigned short si, unsigned short bp, unsigned short sp,
        unsigned short bx, unsigned short dx, unsigned short cx,
        unsigned short ax, unsigned short ip, unsigned short cs,
        unsigned short flags)
{

    /* Get keystroke input from keyboard port */
    key_char = inp(0x60);

    if( ((*kbFlags & KEY_MASK) == (CTRL | ALT))
                        && (key_char == DELETE) ){

        /* Reset the keyboard */
        junk = inp(0x61);
        outp(0x61, (junk | 0x80));
        outp(0x61, junk);
        outp(0x60, (key_char | 0x80));
        outp(0x60, 0x9C);
```

```
          }

          /* Reset the interrupt counter */
          outp(0x20, 0x20);


          /* Now call the next ISR in line */
          (*OldKbIntProc)();

     }
```

There are only two sections to this program: the `main()` body and the keyboard interrupt service routine `KbIntProc()`. The `main()` uses `_dos_getvect()` to retrieve the address of the function (ISR) that is currently servicing keystrokes from the keyboard. Next, it uses `_dos_setvect()` to replace the keyboard servicing function with your own—`KbIntProc()`. A `while` loop that constantly gets keystrokes prints the same message repeatedly until the Escape key (decimal 27) is pressed. When that event occurs, the program calls `_dos_setvect()` to restore the original keyboard servicing program into place.

However, `KbIntProc()` is where all the fun takes place. When installed (by the `_dos_setvect()` call described previously), it gets to look at all keystrokes before anyone else. Therefore, it has the first crack at manipulating or entirely removing the keystroke that came in. When a keystroke arrives, the keyboard is checked to see whether the Ctrl and Alt keys are down. Also, the keystroke itself is checked to see whether it is the Delete key (hex 53). If both cases are true, the Delete key is removed by resetting the keyboard. Regardless of whether the keystroke was ignored, manipulated, or removed by you, the ISR always calls the original keyboard interrupt service routine (`OldKbIntProc()`). Otherwise, the machine immediately grinds to a halt.

When this example program is executed, it prints "Disallowing Ctrl-Alt-Delete..." each time you press any key. When you do press Ctrl-Alt-Delete, nothing happens, because the keystroke is removed and the computer has no idea you are pressing those three keys. Exiting the program restores the state of the machine to normal operation.

If you think about it, you will realize that this program can trap any keystroke or key combination, including Ctrl-C and Ctrl-Break. You therefore could legitimately consider this method for trapping the Ctrl-Break key sequence. I should point out that this method is quite intrusive—the tiniest of bugs can quickly halt the machine. But don't let that stop you from learning or having fun.

Cross Reference:

XX.16: How do you disable Ctrl-Break?

XX.18: How do you tell whether a character is a letter of the alphabet?

Answer:

All letters of the alphabet (including all keys on a computer keyboard) have an assigned number. Together, these characters and their associated numbers compose the ASCII character set, which is used throughout North America, Europe, and much of the English-speaking world.

The alphabet characters are conveniently grouped into lowercase and uppercase, in numerical order. This arrangement makes it easy to check whether a value is a letter of the alphabet, and also whether the value is uppercase or lowercase. The following example code demonstrates checking a character to see whether it is an alphabet character:

```
int ch;

ch = getche();

if( (ch >= 97) && (ch <= 122) )
    printf("%c is a lowercase letter\n", ch);
else if( (ch >= 65) && (ch <= 90) )
    print("%c is an uppercase letter\n", ch);
else
    printf("%c is not an alphabet letter\n", ch);
```

The values that the variable `ch` is being checked against are decimal values. Of course, you could always check against the character itself, because the ASCII characters are defined in alphabetical order as well as numerical order, as shown in the following code example:

```
int ch;

ch = getche();

if( (ch >= 'a') && (ch <= 'z') )
    printf("%c is a lowercase letter\n", ch);
else if( (ch >= 'A') && (ch <= 'Z') )
    print("%c is an uppercase letter\n", ch);
else
    printf("%c is not an alphabet letter\n", ch);
```

The method you choose to use in your code is arbitrary. On the other hand, because it is hard to remember every character in the ASCII chart by its decimal equivalent, the latter code example lends itself better to code readability.

Cross Reference:

XX.19: How do you tell whether a character is a number?

XX.19: How do you tell whether a character is a number?

Answer:

As shown in the ASCII chart in FAQ XX.18, character numbers are defined to be in the range of decimal 48 to 57, inclusive (refer to FAQ XX.18 for more information on the ASCII chart). Therefore, to check a character to see whether it is a number, see the following example code:

```
int ch;

ch = getche();

if( (ch >= 48) && (ch <= 57) )
    printf("%c is a number character between 0 and 9\n", ch);
```

```
else
    printf("%c is not a number\n", ch);
```

As in the preceding FAQ, you can check the variable against the number character range itself, as shown here:

```
int ch;

ch = getche();

if( (ch >= '0') && (ch <= '9') )
    printf("%c is a number character between 0 and 9\n", ch);
else
    printf("%c is not a number\n", ch);
```

As before, which method you choose is up to you, but the second code example is easier to understand.

Cross Reference:

XX.18: How do you tell whether a character is a letter of the alphabet?

XX.20: How do you assign a hexadecimal value to a variable?

Answer:

The C language supports binary, octal, decimal, and hexadecimal number systems. In each case, it is necessary to assign some sort of special character to each numbering system to differentiate them. To denote a binary number, use `b` at the end of the number (`1101b`). To denote an octal number, use the backslash character (`\014`). To denote a hexadecimal number, use the `0x` character sequence (`0x34`). Of course, decimal is the default numbering system, and it requires no identifier.

To assign a hexadecimal value to a variable, you would do as shown here:

```
int x;

x = 0x20;     /* put hex 20 (32 in decimal) into x
x = '0x20';   /* put the ASCII character whose value is
                 hex 20 into x */
```

You must know the hexadecimal numbering system in order to know what numbers to assign. Refer to FAQ XX.24 for detailed information on the hexadecimal numbering system.

Cross Reference:

XX.24: What is hexadecimal?

XX.21: How do you assign an octal value to a number?

Answer:

Assigning an octal value to a variable is as easy as assigning a hexadecimal value to a variable, except that you need to know the octal numbering system in order to know which numbers to assign.

```
int x;

x = \033;          /* put octal 33 (decimal 27) into x */
x = '\033';        /* put the ASCII character whose value is
                      octal 33 into x */
```

Refer to FAQ XX.23 for detailed information on the octal numbering system.

Cross Reference:

XX.23: What is octal?

XX.22: What is binary?

Answer:

The binary numbering system is the lowest common denominator in computing. Binary is base 2. Do you remember being taught different numbering systems in elementary or high school? In one of my math classes in grade school, I was taught base 6. You count 1, 2, 3, 4, 5, then 10, 11, 12, 13, 14, 15, then 20, and so on. At least that's the way I was taught. In truth, you should count 0, 1, 2, 3, 4, 5, then 10, 11, 12, 13, 14, 15, and so on. By starting with 0, it becomes slightly easier to see the groupings of six digits—hence the *six* in base 6. Notice that you count from 0 to the number that is one less than the base (you count from 0 to 5 because 6 is the base). After you have counted to 5, you move to two decimal places. If you think about it, our base 10 (decimal) system is similar—you count up to one less than the base (9), and then you go to two digits and resume counting.

In computers, the numbering system is base 2—binary. With base 2, you count 0, 1, then 10, 11, then 100, 101, 110, 111, then 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, and so on. Contrast base 2 with base 6; in base 2, you count from 0 to 1 before going to two decimal places.

Of course, the next question is "Why is base 2 used?" The reason is the transistor. The transistor is what makes modern-day computers possible. A transistor is like a light switch. A light switch has two positions (or states), *on* and *off*. So does a transistor. You could also say that off equals 0, and on equals 1. By doing so, you can count from 0 to 1 using a transistor (or a light switch if you want). It doesn't seem as though you can do any serious computing with only two numbers (0 and 1), but we're not finished yet. Suppose that you had a light switch panel that contained four light switches. Although each switch still has only two states, the four switches, when treated in combination, can have 16 unique positions, or 2^4 (four switches, two states each). Therefore, you can count from 0 to 15 with just four switches, as shown in Table XX.22.

Table XX.22. Binary counting.

Switches	*Decimal Value*	*Power*
0	0	
1	1	2^0
10	2	2^1
11	3	
100	4	2^2
101	5	
110	6	
111	7	
1000	8	2^3
1001	9	
1010	10	
1011	11	
1100	12	
1101	13	
1110	14	
1111	15	

The table demonstrates three important points: (1) By placing the switches side by side, you can count with them—up to 15 in this case (16 total counts); (2) You can consider each switch as a decimal place, or rather a *binary* place, just as you can with the decimal system; (3) When each switch is considered to represent a binary place, that switch also happens to be a *power* of two (2^0, 2^1, 2^2, 2^3, and so on).

Further, notice that in the table where a power of two is shown, the count had to add another binary place. This is the same as the decimal system, in which each time you increase another decimal place, that new decimal place is a power of 10 ($1 = 10^0$, $10 = 10^1$, $100 = 10^2$, and so on). Knowing this, you can convert binary numbers to decimal with minimal effort. For example, the binary number 10111 would be $(1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$, which equals $(16 + 0 + 4 + 2 + 1)$, or 23 in decimal. A much larger binary number, 10 1110 1011, would be $(1 \times 2^9) + (0 \times 2^8) + (1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$, which equals $(512 + 0 + 128 + 64 + 32 + 0 + 4 + 2 + 1)$, or 743 in decimal.

So what does all this nonsense get us? In the realm of computers, there are *bits*, *nibbles*, and *bytes*. A nibble is four bits, and a byte is eight bits. Do you know what a bit is? It's a transistor. Therefore, a byte is eight transistors, side by side, just like the four switches in Table XX.22. Remember that if you had four switches (or transistors) grouped together, you could count to 2^4, or 16. You could have called that a *nibble* of switches. If a nibble is four transistors grouped together, a byte is eight transistors grouped together. With eight transistors, you can count to 2^8, or 256. Looking at it another way, this means that a byte (with eight transistors) can contain 256 unique numbers (from 0 to 255). Continue this a little further. The Intel 386, 486, and Pentium processors are called 32-bit processors. This means that each operation taken by the Intel chip is 32 bits wide, or 32 transistors wide. Thirty-two transistors, or bits, in parallel is 2^{32}, or 4,294,967,296. That's more than 4 billion unique numbers!

Of course, this description does not explain how a computer uses those numbers to produce the fantastic computing power that occurs, but it does explain why and how the binary numbering system is used by the computer.

Cross Reference:

XX.23: What is octal?

XX.24: What is hexadecimal?

XX.23: What is octal?

Answer:

Octal is base 8. Oh, no, another numbering system? Unfortunately, yes. But there is no need to describe base 8 to the level of detail that was described for base 2. To count in octal, you count 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, and so on. The following two lines count in base 8 and in base 10 side by side for comparison purposes (base 10 is on top):

```
0, 1, 2, 3, 4, 5, 6, 7,  8,  9, 10, 11, 12, 13, 14, 15, 16
0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20
```

Notice that in base 8 (on bottom), you had to increase to two decimal, I mean octal, places after you reached 7. The second octal place is of course 8^1 (which is equal to 8 in the decimal system). If you were to continue counting to three octal places (100 in octal), that would be 8^2, or 64 in decimal. Therefore, 100 in octal is equal to 64 in decimal.

Octal is not as frequently used these days as it once was. The major reason is that today's computers are 8-, 16-, 32-, or 64-bit processors, and the numbering system that best fits those is binary or hexadecimal (see FAQ XX.24 for more information on the hexadecimal numbering system).

The C language supports octal character sets, which are designated by the backslash character (`\`). For example, it is not uncommon to see C code that has the following statement in it:

```
if(x == '\007') break;
```

The `\007` happens to also be decimal seven; the code is checking for the terminal beep character in this case. Another common number denoted in octal is `\033`, which is the Escape character (it's usually seen in code as `\033`). However, today you won't see much of octal—hexadecimal is where it's at.

Cross Reference:

XX.22: What is binary?

XX.24: What is hexadecimal?

XX.24: What is hexadecimal?

Answer:

Hexadecimal is the base 16 numbering system. It is the most commonly used numbering system in computers. In hexadecimal, or hex for short, you count 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F, and so on. Don't get too bothered by the letters—they are merely single-digit placeholders that represent (in decimal) 10 through 15. Remember the rule of counting in different bases—count from 0 to the number that is one less than the base. In this case, it is 15. Because our Western numbers do not contain single-digit values representing numbers above 9, we use A, B, C, D, E, and F to represent 10 to 15. After reaching 15, or F, you can move to two decimal, rather hexadecimal, places—in other words, 10. As with octal and binary, compare hex counting to decimal counting (once again, decimal is on top):

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
1, 2, 3, 4, 5, 6, 7, 8, 9,  A,  B,  C,  D,  E,  F, 10, ...
```

Note that "10", what we historically have called ten, in decimal is equal to "A" in hex. As with the previously discussed numbering systems, when you increase to two or three or more hexadecimal places, you are increasing by a power of 16. 1 is 16^0, 16 is 16^1, 256 is 16^2, 4096 is 16^3, and so on. Therefore, a hex number such as 3F can be converted to decimal by taking $(3 \times 16^1) + (F \times 16^0)$, which equals (48 + 15), or 63 in decimal (remember, F is 15 in decimal). A number such as 13F is $(1 \times 16^2) + (3 \times 16^1) + (F \times 16^0)$, which equals (256 + 48 + 15), or 319 in decimal. In C source code, you will see these numbers shown as 0x3F or 0x13F. The "0x" is used to show the compiler (and the programmer) that the number being referenced should be treated as a hexadecimal number. Otherwise, how could you tell whether the value "16" were decimal or hex (or even octal, for that matter)?

A slight modification can be made to Table XX.22 by adding hex counting to that table (see Table XX.24).

Table XX.24. Binary, decimal, hexadecimal conversion table.

Binary	*Decimal Value*	*Binary Power*	*Hex*	*Hex Power*
0000	0		0	
0001	1	2^0	1	16^0
0010	2	2^1	2	
0011	3		3	
0100	4	2^2	4	
0101	5		5	
0110	6		6	
0111	7		7	
1000	8	2^3	8	
1001	9		9	
1010	10		A	
1011	11		B	

Binary	*Decimal Value*	*Binary Power*	*Hex*	*Hex Power*
1100	12		C	
1101	13		D	
1110	14		E	
1111	15		F	
1 0000	16	2^4	10	16^1

I added one more count at the bottom, taking the total to decimal 16. By comparing binary to decimal to hex, you can see that, for example, ten is "1010" in binary, "10" in decimal, and "A" in hex. In the last line, sixteen is "1 0000" or "10000" in binary, "16" in decimal, and "10" in hex. What does all this mean? Because today's 16-, 32-, and 64-bit processors all have bit widths that happen to be multiples of 16, hexadecimal fits very nicely as the numbering system for those types of computers.

Another added quality is that hex is also a multiple of binary. Note that the last line of Table XX.24 shows the binary value broken into two sections (1 0000). Four binary digits (or four bits) can count up to 15 (16 unique numbers when you include zero). Four bits also equals a nibble. By looking at the table, you can see that one digit of hex can also count up to 15 (16 unique numbers because we include zero). Therefore, one hex digit can represent four binary digits. A good example is (decimal) 15 and 16. Fifteen is shown as 1111 in binary, the highest number four binary digits can count. It is also shown as F, the highest number one hex digit can count. After you go to 16, it takes five binary digits (1 0000) and two hex digits (10). The following lines convert the same numbers used earlier (0x3F and 0x13F) to binary:

```
3F      111111
13F  100111111
```

If you replace the leading spaces with zeros and break the binary digits into groups of four, it might become a little clearer:

```
03F  0 0011 1111
13F  1 0011 1111
```

You are not required to break binary numbers into groups of four—it's simply easier to count when you know that every four binary digits equal one hex digit. To prove that the numbers are equal, I'll convert the binary representations to decimal (because the hex values were already converted in a previous paragraph). 111111 would be $(1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$, which equals (32 + 16 + 8 + 4 + 2 + 1), or 63 in decimal, just as before. The number 1 0011 1111 would be + $(1 \times 2^8) + (0 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$, which equals (256 + 32 + 16 + 8 + 4 + 2 + 1), or 319 in decimal. Hexadecimal and binary fit together like hand and glove.

Cross Reference:

XX.22: What is binary?

XX.23: What is octal?

XX.25: What are escape characters?

Answer:

Escape characters are characters designed to perform a command or task instead of being printed on the computer screen. For example, an escape character could be a character sent to a device that tells the computer screen to draw the next line in red rather than the normal white. The escape character is sent to the device that draws the red line along with the actual characters the device is supposed to draw in red. So how does the device know that the character is an escape character? Typically, the Escape key (decimal 27, octal /033) is sent just before the escape character so that the device knows that an escape character is next to arrive. After the device has the escape character, it acts on the command that the escape character represents, then resumes normal operation—taking characters and printing them on-screen. Because it usually takes two or more characters to pull off the desired command (the Escape key plus the command character itself), this is often referred to as an escape sequence.

I know that this sounds confusing (the Escape key, followed by the escape character), but that is precisely why these are called escape characters. The Escape key is used to inform whoever wants to know that the next character is a command, not an ordinary character. The escape character itself (the one that comes *after* the Escape key) can be any character—it could even be another Escape key. The actual character that represents the desired command to occur is up to the program that is reading these characters and waiting for such commands.

An example of this is the ANSI.SYS device driver. This driver, loaded in your CONFIG.SYS file, intercepted all characters that were printed on-screen and processed these characters for escape sequences. The purpose of ANSI.SYS was to provide a way to print colored, underlined, or blinking text, or to perform higher-level commands such as clear the screen. The advantage to ANSI.SYS was that you didn't have to know what type of monitor or display card you had—ANSI.SYS took care of that for you. All you had to do was embed the escape characters into the appropriate places in the character strings you displayed on-screen, and ANSI.SYS would take care of the rest. For example, if you printed "\033H4Hello there" ANSI.SYS would print "Hello there" on-screen in red. ANSI.SYS would see the Escape key (\033), read the command (which in this case is H4—print remaining characters in red), and print what was left ("Hello there").

Before ANSI.SYS, escape characters were used in the old centralized computing environments (one mainframe computer with a bunch of dumb terminals connected to it). Back in those days, the terminals had no computing power of their own and could not display graphics, and many were monochrome, unable to display color. However, each monitor did have a series of escape characters that the mainframe could send to the monitor to make it do such things as clear the screen, underline, or blink. Programmers would embed the escape characters into their character strings just as you do with ANSI.SYS, and the monitor would perform the desired command.

Today, this type of escape sequence usage has all but died out. On the other hand, many other types of character sequences could be described as escape characters that have been around for just as long, and they are still used heavily today. For example, in the section where I describe how to assign an octal or a hex value to a variable, I am using an escape character (the pair "0x" in the case of hex, and the single `\` character in octal). Note that these characters do not actually use the Escape key as the "hey, here comes something special" notifier, but nonetheless they are used to denote something special about what is to immediately follow. In fact, the backslash character (`\`) is used quite frequently as an escape character. In C, you use `\n` to tell the computer to "perform a linefeed operation here." You can use `\t` to perform a tab advance, and so on.

Cross Reference:

XX.23: What is octal?

XX.24: What is hexadecimal?