MESSAGE MINDER

Windows 3.x is all the rage these days, and it seems that plain old DOS has become passe, right? Well, there has never been a more inaccurate statement. I use DOS exclusively, as does most everyone else I know. The latest Windows craze also makes it difficult for new programmers to "catch up" to top notch programmers because all the journals are singing the praises of Windows, with little attention paid to programming in DOS. I am a college student speaking from experience here. I found it necessary to go through magazines five years old to catch up to what people are talking about in the various programming journals published today!

The most difficult topic of DOS programming was also the most difficult thing for me to find information on: Terminate and Stay Resident programs, or TSRs. Another area which many programmers find daunting is dealing with the PC's serial port. Well, here is a program I wrote which deals with BOTH concepts. If TSR programming is old hat to you, perhaps you might find the code which uses the serial port of interest.

THE PROGRAM: What does it do?

When I decided to undertake the creation of a TSR utility, I didn't want another "me too" note pad or it's ilk; I wanted to create something I hadn't seen before. My program, simply called "message", will allow messages to be exchanged between two computers connected by their serial ports with a null modem cable. If "message" is loaded on both machines, a two-way conversation can take place. No longer would it be necessary to walk from your office to a friend's to remind her of a lunch date, or to ask for quick advice. Once "message" is loaded it records the message strings from the sending computer and displays them in a window at the upper left corner of the screen when the entire message is received. It will then pause until the user presses a key. To send a message the user would press the hot-key <ctrl> w, resulting in a window being displayed at the top of the screen. Type in your message here, followed by return. If you would prefer to load "message" on only one machine, messages could (optionally) be sent by the quick and dirty program I wrote called "send". It's use could not be simpler:

```
C:\send Well, do you think Clinton will win?
```

Send works by taking each command line argument (the words of your message) and copying them into a single string. The string is then sent out the serial port, one character at a time.

To un-install the program the user would enter the following:

C:\message u

THE SERIAL PORT: It isn't really THAT bad!

OK! Before delving into how to make a TSR I feel compelled by guilt to talk a little about the serial port. PLEASE NOTE that this article is about TSR programming, and the information provided on the serial port is intended to give you an idea of how that portion of the program functions.

Two things in the program listing that you should be looking at are 1) the serial() function in "Message", and 2) the "Send" program.

If you don't really care how the serial port works you still need to know how to connect two machines together for "Message" to work. You can let someone rip you off by paying far too much for a null modem connector, or you can be frugal and make something yourself. Here is how:

Serial ports come in two flavors: 9 pin connectors and 25 pin connectors. Both serve the same function, the 9 pin just eliminates the unnecessary connections. In order for "Message" to work you will only need to run two wires between the two machines. The other wires are there to communicate with your modem so "Message" does not need them.

On a 25 pin connector pin 2 is used to transmit data, and pin 3 is used to receive data; on a 9 pin connector pin 2 is used to receive data and pin 3 is used to transmit data. Two connect two PCs with the same type of serial connector together, run a wire from pin 2 of each machine to pin 3 of the other machine. I also recommend that you connect the ground pins (pin 7 on a 25 pin connector) of each PC together. "Message" will usually work without connecting the ground pins, but it is a good (and safe) practice to connect them.

Connectivity aside, how does one send and receive characters with the serial port? One possible method is the use of the BIOS, but it is far too slow. For any real speed we must directly use the PCs UART. UART stands for Universal Asynchronous Receiver-Transmitter.

The UART is a chip inside of your computer, generally the 8250. You talk to this chip via **port addresses**. A port address is similar to a memory location except it is connected to some external hardware. The **base** port address of the PCs first serial card (COM 1) is usually 3F8 hex. Like your microprocessor, the 8250 has internal registers. You access these registers by their port address. 3F8 is called the base address because it is the location of the first register. The second register would be located at port

3F9, etc...

THE WORKINGS OF A TSR PROGRAM.

The first thing "message" does when it is loaded is to determine how much memory it occupies. This information will be saved so that when the program terminates it can tell DOS how much memory to reserve for it.
Before we undertake the task of determining the size of the program a word or two should be said about the PSP, or Program Segment Prefix. When DOS loads a program into memory it creates a 256 byte block of memory containing information about that program. This block of memory always starts at offset 0000. As assembly language programmers know, code usually starts after the PSP at offset 100h, 256 decimal.

In Microsoft C there exists a global variable **_psp** which contains the segment address of the PSP of the current program. With this information we now know where in memory "message" begins. "Message" takes advantage of this right away by setting the **huge** pointer tsrbottom to point to _psp:0000. This address is where our program starts. A huge pointer is used so that C will do our pointer arithmetic for us.

Programs created with Microsoft C in the small memory model are followed in memory by their stack space, then the heap space used for run-time memory allocation. NOTE - Borland swaps the heap and the stack area! With Microsoft C the address pointed to by SP:SS plus a maximum heap size is an effective estimate for the end of the program; For Borland the end of your program is simply SP:SS. To obtain this information you must have access to register values. "Message" accomplishes this by using the Microsoft **_asm** keyword. The following lines , taken from "Message", will copy the stack address into the huge character pointer tsrstack:

```
137  _asm mov WORD PTR tsrstack[0], sp  // save stack ptr
138  _asm mov WORD PTR tsrstack[2], ss  // and  stack seg
```

Then establish a far pointer to our PSP.

```
139  FP_SEG(tsrbottom) = _psp;// save PSP segment
140  FP_OFF(tsrbottom) = 0;   // offset of 0
```

To obtain the program size in bytes simply subtract the PSP address from the stack address, and add to this byte value to maximum amount of memory you will allocate with malloc(). DOS expects this value in paragraphs, not bytes. This makes it necessary to divide our byte size by 16. This can easily be accomplished by shifting the byte value 4 bits to the right (2 ^ 4 = 16).
Here is how "Message" does it:

```
  141  tsrsize = ((tsrstack - tsrbottom) >> 4) + 1;
```

Another useful consequence of saving our stack address is that when our TSR is activated we can use our own stack, rather than using the stack of the interrupted process. This is beneficial because we have no way of knowing how much stack space is left in the process we are interrupting. The doit() function shows how setting up the stack can be accomplished.

Whew! The next procedure **"Message"** undertakes is the initialization of the necessary interrupts which the TSR will use. Before we discuss the specifics of what "message" must do with interrupts we should talk about interrupts in general.

INTERRUPTS

In every DOS machine there exits a reserved area of memory called the **interrupt vector table**. This table is 1024 bytes in size and starts at memory address 0000:0000. This table contains 256 4 byte **far** pointers, that is to say 32 bit pointers which may point to any memory location DOS has access to. These pointers are actually addresses of functions. Some of these you are probably familiar with, such as the VIDEO interrupt 10 hex.

Interrupt 10h is known as a **software** interrupt, meaning you call it yourself in your program. There is another type of interrupt which you may not be familiar with known as a **hardware interrupt**. One example of a hardware interrupt would be interrupt 8, which is called 18.2 times per second to update the system time. You may be wondering how interrupt 8 is called. You, the programmer, did not call it. Hardware interrupts are called by some external event, independent of your program. These external interrupts are controlled by the 8259 Programmable Interrupt Controller chip inside your PC. Another hardware interrupt which we are concerned with is interrupt 0Ch, which is assigned to COM port 1. We configure the computer to call this interrupt whenever it receives a character thru COM 1.

All you need to know about hardware interrupts for now is that the 8259 chip must be acknowledged, or the computer will crash. This is accomplished by sending the value of 20h to port address 20h. See the serial() function in the program listing for a demonstration of this.

By using DOS functions programmers may set interrupt vectors to point to functions of their own creation. Microsoft C has a key word **interrupt** which should be used in the declaration of any function which shall be called by the interrupt process.

Here is an example prototype:

```
   void interrupt _far my_function(void);
```

The interrupts which we are concerned with are:

09h - This interrupt is called any time a key is pressed. The scan code of the key is available at port 60h. See the keybd() function in listing.

1Ch - The timer tick interrupt. This is called 20 times per second by hardware interrupt 9. See the clock() function in the program listing for an example.

28h - The DosOk interrupt. DOS calls this interrupt when it is idle waiting for keyboard input.

0Ch - This interrupt is dedicated to COM 1. We configure the computer's UART to call this interrupt whenever a character is received.

The Microsoft C functions pertinent here are:

```
       _disable()-  disables maskable interrupts
       _enable() -  re-enables interrupts
       _dos_getvect() -  uses DOS function to get int vector
  _dos_setvect() -  uses DOS function to set int vector
```

More on these functions later.

POINTERS TO FUNCTIONS

Another important aspect of the C language which must be understood is the **pointer to** a **function**. When we set an interrupt vector to point to our own function, it is imperative that we **chain** the interrupt. To chain an interrupt the following must occur:

1) We should save the original interrupt vector in a pointer to a function.
2) When our function is called (our function is referred to as an **Interrupt Service Routine**), it must call the original interrupt function.
3) When the original interrupt function is done, we can then perform our intended action.

Here is an example prototype of a pointer to an interrupt function:

```
   void (interrupt _far *)intfunc (void);
```

Getting and setting of vectors is easily accomplished thru the use of the _dos_getvect() and _dos_setvect() functions, as shown in the program listing. It should be noted that functions with the prefix _dos are merely interrupt 21h calls packaged in a C function. The _disable ()
function is used to temporarily disable interrupts while setting them so that the interrupt is not called while you are in the process of changing it. Interrupts are then allowed again with the _enable() function.

The init() function is responsible for getting the address of the InDos flag. This flag is a non-zero value whenever a DOS service is being performed. This is important to know because DOS is not **re-entrant**. DOS is a single tasking operating system; it can only process one service at time. If a DOS function were to be called from inside your TSR while DOS was already busy the computer would crash. DOS in no way attempts to keep track of your TSR, but the InDos flag is used by DOS itself to determine whether or not DOS is busy. Our program can use the InDos flag to it's advantage by chaining into the system timer and checking the InDos flag 18.2 times per second, executing the required operation if and when DOS is not busy.
A problem with relying entirely on the InDos flag is that if DOS is not doing anything important, such as waiting for a keystroke, the InDos flag indicates that DOS is busy. The creators of DOS have provided a solution to this problem: When DOS is idle waiting for keyboard input it repeatedly calls interrupt 28h. This is normally a do-nothing routine, but we can replace this with one of our own.
Since DOS is off-limits to a TSR program our old friends such as printf() can no longer be used. This is no great problem. The video bios can still be used for character output as well as keyboard input. "Message" does not use the bios for text display, instead it places characters on the screen by copying the characters directly into video memory. This is known as a **direct screen write**.
Direct screen writes are done by establishing a far pointer to video memory, which starts at segment B800h on color graphics adapters, B000h for monochrome adapters. The video memory in PCs is configured such that each character in memory is followed by that character's attribute. In order to calculate where in memory a character would be for a particular row and column the following formula could be used:
B800:0000

```
   video address = base address + (row * 160) + (column * 2)
```

Where each row occupies 160 bytes in memory, 80 bytes for the characters themselves, and 80 more for the attributes. The column number is multiplied by two to account for the memory required to store the attribute of

each column up to and including the specified column in a row.

IDENTIFICATION and UN-INSTALLATION

1) IDENTIFICATION:

OK, here is the tough part! All TSR programs should provide some method of determining whether our not a copy of itself is currently resident, as well as a way to return the memory it occupies to DOS. There are a few options available to the programmer as to how to perform these functions. The easiest method is to use an interrupt which is normally unused by anything else, such as interrupt 60h. Interrupt 60's vector is set to 0000 when the machine is booted, a fact which the TSR could take advantage of. When the program instates itself, it would set int 60's vector to point to it's un-install procedure. By setting the vector to this non-zero value we have a method of seeing if the program is resident. When you wish to un-install the TSR, simply call interrupt 60. It should be noted that this is a rather quick and dirty way of checking to see if the program is resident; if another TSR uses interrupt 60 conflicts will occur. For our purposes interrupt 60 is not safe enough.

Another way to determine whether or not the TSR is resident is to search through memory for the program's signature. There are two ways for the programmer to do this:
1) Check EVERY segment of DOS memory to see if it is your TSR program.
2) Go through the DOS Memory Control Block chain to search for your program.
"Message" uses the first method for two reasons. First, it is less complicated, and second going through the MCB chain is a "frowned" upon process because it uses an undocumented DOS function call that we can avoid using. If you are interested in how to go through the MCB chain investigate function 52h of DOS interrupt 21h.

Now to explain what "Message" is doing. You will probably want to refer to the check_install() function if the following is to make any sense. In our program we create a structure containing a pointer to the character string "_MESSAGE", and a pointer to the un-install function. The check_install() function then calculates how many bytes from the start of the program this structure is. This next line is where all the "magic" happens, where "resident" is the structure and "tsrbottom" contains our PSP address:

```
719  /* find offset for tsrinfo structure */
720  string_check = (struct tsrinfo _based(seg) *)
721((char _huge *)resident - tsrbottom);
```

The variable string_check is equal to the number of bytes the signature structure is from the start of the program (_psp).

The most important piece of information to get out of the above example is the use of **typecasting**. The incorrect offset would be stored in string_check without specifically telling the compiler HOW we want it to subtract tsrbottom from resident. What do I mean by this? Well, you will recall that the 80x86 family has segment and an offset registers which when added together give the actual memory address. However, the segment and the offset are not added in a logical manner. For example, supposed you had a segment address of 54BE hex and an offset address of 2E3A. You would calculate the actual memory address by adding them in this way:

```
          SEGMENT:   54BE0  **** Notice the "magic zero"
          OFFSET :  + 2E3A
ADDRESS:   57A1A
```

By typecasting, we are simply telling the compiler that we want segment:offset subtraction of the given values, as opposed to integer subtraction.

Once this offset is calculated the program checks every segment in memory to see if it is a PSP by seeing if the memory location contains 20CD hex. Why search for 20CD? Well, the first item in every PSP is an Interrupt 20, which is numerically represented as 20CD hexadecimal.
If a PSP is found, the calculated offset is checked to see if it contains the character string associated with the TSR.

2) UN-INSTALLATION:

If the user called check_install() passing it a value of 1 the un-install function will be called, provided the program is resident. Please note the un-install function is delimited with the interrupt keyword. This is necessary because the function being called is not a part of the current program, it is in the resident version of "Message". If un-install is called, it restores the original interrupt vectors and restores the memory the program occupies by using the _dos_freemem() function. This C function calls a DOS function which frees the memory at the segment address we send it. You will notice that the value we pass to _dos_freemem() is _psp. This segment is the start of our program, and this is the address we must tell DOS to free. You must also release the memory allocated to hold the TSR's environment space. The segment address of the environment is stored in the PSP at offset 002Ch. We pass this segment information to DOS by creating a far pointer to the PSP table, and sending DOS the segment pointed to in the

table, as shown below:

```
844  pspvariable = _psp;  // segment of environ PSP segment
845  environ = (int _based(pspvariable) *)0x2C; //envir. ptr.
846  _dos_freemem(*environ);// free environment memory
847  _dos_freemem(_psp);    // free program memory
```

ROLLING YOUR OWN

Before you go off and start writing the next Side Kick (tm, of course) there are a few more items to cover. It is likely that whatever type of program you write you will want to have pop-up menus. This will require you to perform run time memory allocation to save the portion of the user's screen that your window will overwrite. I have not allocated memory in "Message" because it was not necessary. Since the program always saves the first three lines on the screen (requiring only a small amount of memory) it was easier to simply declare an array of the proper size. In order to use the malloc() function from within your TSR it is necessary to add the maximum amount of memory your program will allocate to the amount of memory you calculate your program occupies, and then reserve this total amount with the _dos_keep() function.

You will probably want your TSR to be activated by some sort of "hotkey". This will require you to chain into one of the keyboard interrupts. Interrupt 9 or 16h should be suitable to this purpose. Interrupt 9 is what "Message" uses, so that I what I will describe.

Every time a key is pressed an interrupt 9 is generated. The **scan code** for this key is located at port 60h. Please note that this code is **not** ASCII. If you do not have a reference book with a list of key scan codes Quick C help contains a list of them. Also note that each key has only one scan code associated with it, that is to say that z and Z would generate the same scan code. You will no doubt want your TSR to respond to some combination of keys, such as the (<ctrl> w) combo "Message" uses. To do this you must check the **keyboard status flag**, located at memory location 0040:0017 hex. Bit three of this flag will be high if the <ctrl> key is down. Here is the code where "Message" checks for (<ctrl> w).

```
   387 if(  // hotkey pressed?
   388( (ch = inp(0x60)) == 0x11) && // w key?
   389!tsr_running && // TSR not active?
   390(*keyboard_status & 4)    // <ctrl> key?
391   )
```

If your program will perform file i/o with the standard C library functions it will be necessary for your program to tell DOS to use the TSR's PSP instead of the interrupted applications. If you are interested in this you should investigate functions 50h and 51h of DOS interrupt 21h.

CONCLUSION

I hope that I have in some way furthered your knowledge of TSR programming. I hope I have not omitted some piece of information you were keeping an eye out for, but I have covered enough to get you started.