SHUFHOOK manual

This document has been compiled on 2021-10-19.

Contents

Section 1: Overview

This program allows shuffling around interrupt handlers by utilising IBM Interrupt Sharing Protocol (IISP) headers and Alternate Multiplex Interrupt Specification (AMIS) interrupt lists.

Currently, only one operation is implemented. It is called "Bubbling". It is specified by the /B switch. To bubble means to swap around handlers so as to expose a specific handler as the topmost handler.

Section 2: The long story

This is a copy of the same chapter in the KEEPHOOK manual. If you know some of these concepts already you may wish to skip some or all of the following sections.

2.1 What is an interrupt?

Interrupts originate as a control-flow feature, originally for interrupting normal execution of a program upon special conditions. These are called hardware interrupts. For example, a keyboard device may initiate a hardware interrupt request (IRQ) to the CPU to indicate it has new data available. That is, in the case of a keyboard, one or several new keypress or key release events.

The keyboard is actually numbered as the second IRQ (IRQ #1) on a typical 86-DOS system. IRQ #0 is used for a timer tick. It is raised by default with a frequency of 18.2 Hz (nearly 64 kilo-binary = 65_536 times per hour). There are several other IRQ sources, up to 15 of which can be differentiated by the dual-PIC setup of a typical IBM-PC-compatible 86-DOS machine.

When an IRQ is being serviced by the CPU, the CPU waits until such a time that it comes upon an instruction boundary. (A repeated string operation counts as multiple instructions for these purposes.) Then it saves some machine state onto the current stack. On an 8086 processor those are the 16-bit flags register, the code segment, and the instruction pointer, for a total of 6 bytes. This interrupt stack frame can be used to return the control flow to the point that is being interrupted. There is a dedicated "interrupt return" (`iret`) instruction that performs this action. Upon execution of an interrupt, after the stack frame is written a few control flags are modified. On the 8086, the Interrupt Flag (IF) and Trace Flag (TF) are cleared.

Finally, to service the interrupt, a certain "far" address (16-bit code segment value and 16-bit instruction pointer value) is read from a specific entry of a table, the Interrupt Vector Table. The IVT is located in the random-access memory at linear address zero for the 8086. Loading the `CS:IP` from the IVT is effectively an inter-segment branch to some code somewhere in the 1088 KiB segmented address space. This code is called the interrupt handler, or Interrupt Service Routine (ISR).

Typically, an interrupt handler servicing an IRQ will respond to the interrupt condition in some way, such as to modify some memory. Aside from the interrupt stack frame (FL, CS, IP) the handler may push further registers onto the stack to preserve their values. Later it can pop the registers in the reverse order and then return to the interrupted code using the `iret` instruction.

During their design of the 8086, Intel provided for up to 256 different interrupt numbers. Each interrupt number has a corresponding entry in the IVT. There is also an instruction, called `int`, which can cause the processor to branch to a specified interrupt handler as if it was servicing an IRQ. As opposed to the hardware interrupts, an interrupt caused by an `int` instruction is called a software interrupt.

When 86-DOS was developed, it was decided that several software interrupt numbers would be used for the purpose of a "system call" operation. The most well-known and common of these

software interrupts is interrupt 21h. It is used for most DOS service calls. Using a software interrupt is useful to DOS due to several reasons.

The IVT can be used by any process without having to set up a segment register to point to any OS data (freeing them up to be all used by the application), and also without having to copy any sort of dispatcher or jump table into each process's code segment. Besides, the interrupt stack frame allows for some state to be restored upon an interrupt return without more OS-side setup.

Software interrupts used as service dispatchers often will be utilised to return results in one or more registers or status flags. They can also clobber registers. Unlike the typical use of hardware interrupts not all registers need to be preserved by software interrupt handlers. This is because the application knows when and that it uses an `int` instruction, and so can treat it like a function call following a certain calling convention. Hardware interrupts can occur at most instruction boundaries and are thus less alike ordinary function calls.

2.2 What does it mean to hook an interrupt?

Hooking an interrupt generally means to bend the address stored in an IVT entry so as to point to a new interrupt handler. Once the address is updated, subsequent interrupt calls to the affected interrupt will call the new handler.

DOS itself hooks several interrupts on start-up, including interrupt 21h. In this case, generally DOS will not remember the address previously found in the IVT entry for that interrupt number.

However, there is a class of system extensions that falls into the category "Terminate and Stay Resident" (TSR) programs. A TSR installs a resident part before terminating another part, called the transient part. Thus it leaves resident a part of itself. Such programs may hook interrupts, such as the DOS service interrupt 21h. More often than not, such resident software will retain a segmented far pointer to the "next handler". This pointer is obtained from the corresponding IVT entry before the entry is modified to hook the interrupt.

The next handler pointer can be used for three purposes. First, if the resident part is ever to be uninstalled during a session then the program needs to have remembered the prior handler's address to restore that address into the IVT entry. Second, upon whatever handling the resident does when its interrupt handler is called, it may decide to "chain" the call to the prior handler. It can do so using a direct or indirect jump branch. Third, the resident may call the prior handler by emulating an interrupt call complete with a stack frame. This is obtained by running a `pushf` instruction followed by a direct or indirect call branch. If the control flow returns from the prior interrupt handler then the resident may do additional handling afterwards. This allows the resident to do both pre- and post-processing of an interrupt service call.

2.3 How about to unhook an interrupt?

As alluded to, a TSR program may be instructed to uninstall its resident portion. Non-resident software can also hook interrupts and may want to unhook them when it terminates.

In the simplest case, to unhook an interrupt means to take the "next handler" address the program has stashed away, and write it to the IVT entry of the interrupt in question.

However, consider this: Let A be the program that hooked interrupt 21h. If a different program, let's call it B, hooked the same interrupt at a point in time where A's handler already had been hooked into the interrupt, then we end up with a chain of interrupt handlers that goes like `IVT -> B -> A -> DOS`. If A naively restores its next handler pointer (which contains

the DOS's handler address) then the chain would change to be as follows, `IVT -> DOS`. As desired, A's handler no longer is in the chain. But B's handler also is no longer in the chain, without B's awareness or consent.

If B *also* is instructed to uninstall and naively restores its next handler pointer into the IVT then the IVT will point to A's handler next. But by this time, the memory used by the resident A may already have been freed and possibly re-used. Clearly, this is concerning.

In some cases it can be appropriate or desired to uninstall one's interrupt hook in this way. However, more commonly it is better for the program A when trying to uninstall to check whether its resident still provides the top-most handler for the used interrupt number. That is, whether the IVT entry still points to A's resident handler, rather than elsewhere.

This introduces a failure condition: If, like before, B hooks the same interrupt after A has hooked it, and then A is told to uninstall its resident, its IVT entry check will indicate that its resident does not provide the top-most handler. Pending further options this must cause A to fail in part or in full. It should report this and leave resident at least a minimal interrupt handler, in the same spot that held the entrypoint for its full interrupt handler, so as not to disturb the interrupt handler chain. That is, the address stored by B as its next handler pointer must remain valid.

2.4 Can we work around the unhook problem?

The problem with two TSRs both hooking the same interrupt is that one of them remembers a pointer to the other. It is not enough to modify the IVT.

However, what if we had a program (call it C) that indicates in some way where its "next handler" pointer lives? If A is directed to uninstall its resident portion, and C's is the top-most handler, then A can follow C's next handler pointer and determine that it points to A's handler. If C promises to use that particular pointer, and no other, to refer to A, then A gains an option. It can modify C's pointer and update it with the next handler pointer of A itself. So we go from `IVT -> C -> A -> DOS` to the new chain `IVT -> C -> DOS`.

To automate the indication, the IBM Interrupt Sharing Protocol (IISP) was defined. It consists of an 18-byte structure, of which 6 bytes are crucial. Those 6 are comprised first of a word (2-byte) signature, the string "KB", at the address of the interrupt handler entrypoint plus six bytes. And second, at the entrypoint address plus two bytes, a dword (4-byte) segmented far pointer. This pointer is the sole reference to the next handler; if the resident C wants to unhook itself (either from the IVT or another reference) or wants to call or chain to the next handler, it must use the IISP header's field.

2.5 What is this "advanced" deinstallation about?

If you install the resident A first, and then the B mentioned earlier, A can no longer access the reference to its interrupt handler. The same is true if you install A first, then C, and then B. However, there is a way to install B after A and still access the reference to A.

For how to do that, first some words about the Alternate Multiplex Interrupt Specification (AMIS). AMIS was defined by Ralf Brown to allow a common interface to TSRs with several standard functions. Instead of overloading common software interrupts like interrupt 21h (DOS services) or interrupt 2Fh (the traditional "multiplex" interrupt) with ever more special magic functions, AMIS defines a new interface on the previously unused interrupt 2Dh.

Each AMIS program allocates a multiplex number out of 256 possible values upon installation,

choosing a number that is not yet used. To facilitate this, there is a common detection function, the AMIS function 0, called "Installation Check". The multiplex number to check is written to the AH register, and the AMIS function number to call is put into AL. Then the software interrupt 2Dh is called. Function 0 returns with AL set to 0FFh if the multiplex number is in use, but leaves AL at 0 instead if it isn't. (Function 0 also returns a signature and version number in three more registers but these don't matter to us.)

Another AMIS function is function 4, called "Determine Chained Interrupts". It can be called once it has been detected that a multiplex number is in use. Again the multiplex number is passed in AH and the function number in AL. Additionally, the querent must provide an interrupt number to check in the BL register. This function can return a pointer to an interrupt handler entrypoint, or the address of an AMIS interrupt list, or other return values. It is assumed that AMIS TSRs' interrupt handlers begin with an IISP header.

The AMIS interrupt list is made up of a number of 3-byte entries: The first byte specifies an interrupt number, the second and third byte form a word that specifies an offset. The offset, combined with the same segment as used to address the interrupt list itself, points to the entrypoint of an interrupt handler that corresponds to the specified interrupt number. The last entry of the list is the one with an interrupt number of 2Dh. (Note that it is not prohibited to have multiple interrupt handlers for the same interrupt number, save interrupt 2Dh.)

Knowing that much, consider a case with a TSR program called D. It is an AMIS-compliant TSR which allocates a multiplex number to be detected by AMIS function 0, and returns an interrupt list upon being queried on AMIS function 4. Aside from interrupt 2Dh, it also hooks interrupt 21h.

Now if you install the TSRs in the order A then D then B, you end up with an interrupt chain for interrupt 21h that goes like `IVT -> B -> D -> A -> DOS`. If you tell the A program to uninstall itself, it will determine that the IVT points to B, and B's handler does not start with an IISP header. The advanced deinstallation method kicks in now: The deinstaller for A can detect all AMIS multiplex numbers that are in use, then ask each resident AMIS TSR about its handler for interrupt 21h using function 4. If it finds such a handler it can search the interrupt chain starting at this handler.

In our case, A will detect the resident D, ask for its interrupt 21h handler with function 4, and receive the interrupt list. The interrupt list contains an entry for D's interrupt 21h handler entrypoint. Searching the chain starting from D's handler, which has an IISP header, allows A to find the reference to A's handler. Then A can uninstall its own handler from the chain by updating the reference in D's IISP header. The interrupt chain is updated from `IVT -> B -> D -> A -> DOS` to now be `IVT -> B -> D -> DOS`. The reference that B holds to D remains valid.

Section 3: Details

Interrupts are specified either as hexadecimal numbers, separated by blanks, or using the keyword ALL.

Interrupt handlers are matched using the /S= switch.

3.1 Bubbling

Bubbling means to switch handlers so that a selected handler is made the topmost handler. This is possible in three cases:

The handler is reachable from the top (only IISP handlers are above it) and it is an IISP handler.

> This is called the Normal case. The handler is unlinked and re-inserted at the top. That is, the IVT is made to point to the handler, the reference to the handler is made to point to the handler's next handler pointer, and the handler is made to point to the previously topmost handler.

The handler is reachable from another, discoverable IISP handler and it is an IISP handler.

> This is called the More Complex case. The operation is the same as for the first case.

The handler is reachable from the top and another new IISP handler is discoverable which comes later in the interrupt chain.

> This is called the Complex case. The handler and its chain up to the new handler is unlinked and re-inserted at the top. The IVT is made to point to the handler, the reference to the handler is made to point to the new handler's next handler pointer, and the new handler is made to point to the previously topmost handler.

There is another scenario which is not currently implemented. This is if the handler is a non-IISP handler, it is not reachable from the top, but it is reachable from another discoverable IISP handler. The problem is that this needs another IISP handler that is reachable, and it must be located in the chain behind the handler to bubble. It is not possible to determine which of all discoverable IISP handlers is located behind the handler to bubble.

3.2 /S=NNNN switch (Set match segment)

Match handlers with a code segment equal to a certain number. NNNN is a hexadecimal number.

Specifying this switch sets up matching so that the code segment of a handler is used to determine which handler to operate on.

Section 4: Requirements

- 86-DOS system compatible to MS-DOS version 2 or later (version 5 or later recommended)
- 8088/8086 processor or higher
- 10 KiB of memory for the transient SHUFHOOK program

Section 5: Building

1. Check out tip revision of the hg repo to a directory
2. Check out lmacros into a sibling directory (that is, the .mac files should be located in the path `../lmacros/` as seen from the shufhook directory)
3. From the shufhook directory run `./mak.sh` (needs bash and nasm)
4. Alternatively, from the shufhook directory run `nasm -I ../lmacros/ transien.asm -o shufhook.com`
5. From the shufhook/doc subdirectory run `./mak.sh` (needs bash and hg and halibut)

Source Control Revision ID

hg adcbf681b221, from commit on at 2021-10-19 18:08:34 +0200

If this is in ecm's repository, you can find it at https://hg.pushbx.org/ecm/shufhook/rev/adcbf681b221