COMP 2659 Course Project – Stage 9: Custom ISRs

Released: Monday, March 30, 2015
Target Completion Date: Monday, April 6, 2015 (at the latest)

Overview

In this stage you will replace two of TOS's ISRs with two of your own. Specifically, you will write custom VBL (vertical blank) and IKBD ISRs. We will refer to the development of each of these ISRs as stages 9a and 9b, respectively. Budget about 1-2 days to complete the former[1], and 3-4 for the latter.

From the user's perspective, the impact of these changes on the game's functionality should be almost non-existent[2]. However, your program's reliance on the operating system will be eliminated almost completely!

The goal of stage 9a is to implement a VBL ISR for your game. This will enable you to reduce the amount of clock polling performed by the main game loop.

In summary, your VBL ISR will:

- time the page flipping for double buffered graphics;
- time the playing of music;
- time any other synchronous events needed by your game[3].

The goal of stage 9b is to implement an IKBD ISR for your game. It will take care of all keyboard and mouse input incoming from the IKBD's 6301 microcontroller.

General ISR Requirements (Applies to Both Stages 9a and 9b)

As you know, an ISR must be implemented in assembly language. However, it is acceptable to delegate one or more subtasks to subroutines written in C. Of course, this may not be necessary for very simple subtasks.

ISRs *must always* be as *fast* as possible! The idea is to do the minimum amount of work and then return quickly to the interrupted task. Remember that other, lower priority (but still important) ISRs may be delayed due to the execution of your ISR.

Time-consuming operations must be left for the main program to perform. Typically, an ISR will simply update some global data (if necessary) and then will somehow notify the main program (e.g. by setting a flag or enqueueing in a buffer) that some extra work needs to be performed.

Ideally, your game supports the ability to quit back to TOS. When an ISR is installed, the previous vector must be saved so that it can be restored on exit.

---

[1] If all previous stages have been designed and implemented well, stage 9a should actually be very quick.
[2] One exception is that the text cursor maintained by TOS will cease to blink in stage 9a. This is because the plotting of the text cursor is performed in the TOS VBL ISR.
[3] Since your game will have a maximum frame rate of 70 Hz, there isn't much point in generating synchronous events using a faster timer. Put another way, why update your model more frequently than you can render views of the model? Regardless, some games may require a higher frequency, e.g. for measuring a user's reaction time. In this case, you can use one of the ST's programmable hardware timers. However, such issues will not be dealt with in phase 2 and will not be discussed here. Seek additional guidance from your instructor as necessary.

Requirement: VBL ISR (Stage 9a)

You must replace the TOS VBL ISR with your own. It must:

- time the page flipping for double buffered graphics;
- time the playing of music;
- time any other synchronous events needed by your game.

Guidance on how to accomplish each of the above are given in the next three sections.

Your main game loop must not perform any clock polling. This includes no longer consulting the TOS vertical blank counter (the longword at address $462, sometimes referred to as "vbclock")[4].

Guidance: Timing of Page Flipping (Stage 9a)

One way of achieving flicker and tear free animation is to perform all screen drawing inside the VBL ISR, i.e. between screen refreshes, when the frame buffer isn't being scanned. However, this only works for very short plotting operations. This technique will *not* usually work for games such as yours because the image rendering takes too long. If your code is still updating the frame buffer when the next screen refresh begins, then image distortion will be seen.

Another common (but incorrect) approach is to render the next frame of animation in the main routine, but to then flip the buffers inside the ISR. The idea (again, incorrect) is that page flips will be synchronized with screen refreshes. This will *not* work because the video index register is reloaded from the video base register at the same time as the VBL IRQ is asserted. By the time your ISR is invoked, it is too late to update the video base.

The correct technique for your game is as follows: 70 times per second, the VBL ISR *requests* that the next frame of animation be rendered, e.g. by setting a global "render request" flag. When the main game loop sees that "render request" has been set, it calls the master render routine to render the next frame of animation into the back buffer. Then, a page flip is scheduled by calling `set_video_base`. The "render request" is then cleared. Note that the pages won't actually flip until the next vertical blank (when the video index register is reloaded with the new video base). If the main game loop is polling "render request" properly, it won't attempt to produce a subsequent frame of animation until after that flip has occurred.

In pseudo-code:

VBL ISR[5]:

```
update music (as necessary)
process synchronous events and update model (as necessary)
set render_request = 1
```

[4] In fact, vbclock is updated by the TOS VBL ISR. So, this clock will cease to tick once the operating system's own VBL ISR has been replaced.

[5] Note: the time elapsed since the last vertical blank, as measured in 70Hz clock ticks, will always be 1.

```
main game loop:

    repeat until game over

        if asynchronous event(s) outstanding
        then
            process and update model as necessary

        if render_request
        then
            render model to back buffer
            schedule page flip by calling set_video_base
            set render request = 0
```

Guidance: Timing of Music (Stage 9a)

Instead of polling the TOS vbclock in the main game loop to decide when it is time to move on to the next note, simply update the music within the VBL ISR. When it is time to move on to the next note, the ISR (or a delegate subroutine such as `update_music`) will simply update the state of the PSG. Since this is a relatively quick operation, it can be done exclusively within the ISR. There must be no song-related event processing left in the main game loop at all.

Be sure to initialize all song-related data structures and variables before installing the VBL ISR.

Note that the playing of sound effects will not necessarily be handled by the VBL ISR, unless they are in response to synchronous events. Sound effects which play in response to asynchronous events (e.g. user input) will be triggered elsewhere.

Important: each region of code which accesses the PSG is a critical section – figure out why[6]! Any main program code which attempts to access the PSG must do so with VBL interrupts masked.

Guidance: Timing of Other Synchronous Events (Stage 9a)

Likely, your game must process other types of synchronous events, e.g. enemy ships appearing or balls moving. Use the VBL ISR to trigger these events.

For example, if a ball is moving across the screen at constant velocity then update its position regularly, inside the VBL ISR. Do *not* couple the position updating with rendering. These are separate tasks. Note also that rendering time may not be constant; it may vary depending on the complexity of the image being produced. For the ball to move at a constant rate, its position must be updated regularly within the ISR, regardless of what the main program is busy doing.

[6] Hint: what could happen if the main program was in the middle of modifying PSG registers for a sound effect, due to an asynchronous user input event, and then the VBL ISR interjected and tried to play the next song tone? In particular, think about how the PSG does register selection.

Requirement: IKBD ISR (Stage 9b)

You must replace the TOS IKBD ISR with your own, which. It must:

- handle all key make and break codes incoming from the IKBD;
- handle all mouse packets incoming from the IKBD.

Guidance on how to accomplish each of the above are given in the following sections.

Guidance: Basic Issues Related to Incoming IKBD Data (Stage 9b)

Under TOS, the Atari ST uses vector #70 for IKBD IRQs. The IKBD ISR will be invoked once for each byte received by the IKBD 6850 ACIA, as sent by the IKBD's 6301 microcontroller. Note that it may also be invoked in other circumstances, such Rx overruns. A complete ISR must handle these eventualities.

Don't forget that the IKBD ISR will be invoked once for each key make code, once for each key break code, and three times for each mouse packet. Therefore, so that it doesn't confuse mouse data with scancode data, it must maintain a state. In other words, the ISR will be a little software finite state machine.

Important technical detail: the last step performed by the IKBD ISR must be to clear bit #6 of the "in service B" register of the MFP 68901 chip[7], just before returning. Otherwise, the ISR will run a maximum of one time and will never be invoked again. The reason for this will be discussed further in lecture.

Remember that the ST has two 6850s: one is the IKBD interface and one is the MIDI port controller. They share a common vector #, and hence a common ISR. In addition, there is always the danger of spurious interrupts[8]. Therefore, a complete ISR must check that the IKBD 6850 is actually asserting an IRQ, by checking its status register, before it attempts to service an interrupt.

For this assignment, the safest thing to do is probably to disable MIDI 6850 interrupts at the start of the program, and then to re-enable them at the end.

Important: in this assignment, you may write various routines whose execution may overlap in time (e.g. ISR code vs. interruptible non-ISR code). As discussed in class, ensure that "critical sections" are protected. This requires masking keyboard interrupts at the start of the critical section, and unmasking them at the end. To avoid overly restrictive masking, mask these interrupts at the 68901 MFP, not at the 68000.

Guidance: Handling Incoming Make and Break Codes (Stage 9b)

Should the ISR perform input event handling for key presses? Only if the event handling logic (i.e. the required update to the game world model) is **very** simple and quick. This is discouraged, since overruns becomes very likely.

[7] This register is mapped at address $FFFA11.

[8] Spurious interrupts are due to hardware problems such as excessive electromagnetic interference: noise might induce a signal on an IRQ line even though no device is asserting IRQ. In a well designed system, of course, the danger of interference should have been minimized. At any rate, STeem is an emulator and probably doesn't simulate spurious interrupts. Still, your ISR should assume the possibility and deal with them if they arise.

IKBD event handling should usually be left to non-ISR code. In this case, the ISR must enqueue each key press's scan code in a buffer (a circular queue implemented as an array). As discussed in class, the main game loop must repeatedly check to see if the buffer is non-empty, and if so it must dequeue the key and process the event.

Guidance: Managing Key Auto-Repeat (Stage 9b if necessary)

In some games, key presses are used for "discrete" input events, meaning that it is the depressing of the key which is important, not how long it is held down before being released. For example, a single shot might be fired from a ship each time the 'f' key is struck. The key must be re-struck in order to fire the next shot.

In other games, when a key is depressed, it signals the start of a "continuous" input event which is effect until the key is released. For example, as long as the "up arrow" key is held down, a ship might ascend. When the key is released, the ship stops ascending.

If your game takes input from the keyboard, it must decide whether input is discrete or continuous, and it must handle key make and break codes appropriately. They are both fairly straightforward to implement.

For continuous input, note that a timer ISR such as the VBL ISR will be involved. The exact approach will be game-specific. One common technique: as long as the break code for the most recent make code has not been received, regenerate the corresponding user input event at each timer ISR invocation. In other words, changes in key state are asynchronous events, but while the key is depressed a series of synchronous events are generated.

Guidance: Handling Incoming Mouse Packets (Stage 9b)

Your IKBD ISR must deal with mouse packet bytes, even if the mouse isn't used during game play. One reason is that mouse packet bytes should not be misinterpreted as key make or break codes if the mouse is jiggled during play. Another is that the splash screen in the final version of your game must be mouse-driven.

To process mouse input, maintain global mouse x and y coordinates. Each mouse packet will contain three separate items. The receipt of each by the 6850 will result in separate interrupts. Do **not** wait in the ISR for the arrival of all three bytes. It is guaranteed that the three items will be consecutive, i.e. a make or break from a key press will never occur between the items of a mouse packet. The three parts of a mouse packet are the header and relative mouse movement data. The header indicates which if any mouse buttons are pressed and its value is 1111 10LR, where L means left mouse button and R means right mouse button. The relative mouse movement is the signed change in the x and y positions in pixels since the previous packet (these are commonly referred to as deltas: delta x and delta y). These deltas should be added to the global x and y coordinates. Be careful: the global coordinates will need to be integers, so your code will need to widen the deltas properly before adding.