COMP 2659 Course Project – Stage 10b: Burning Custom Game ROMs

Released: Tuesday, April 7, 2015
Target Completion Date: Thursday, April 16, 2015 (at the latest)

Overview

This stage involves removing TOS *completely* from the emulated Atari ST, and replacing it with your own custom game ROMs! This will prove that your game has no reliance on the operating system at all, and that it is in full control of the underlying hardware.

Note: the "quit" option on the splash screen's main menu should be disabled if this stage is completed, since there will be no shell or desktop to quit back to.

Background: The TOS ROMs

When a computer boots, its CPU is reset. Upon reset, among other things, the CPU initializes its program counter with the address of the first instruction to execute. This code is in ROM. On many modern systems, this code ("firmware") is a small BIOS ("Basic Input/Output System") which loads the O/S boot-loader (e.g. from disk), and jumps to it. The boot-loader then loads the actual O/S and jumps to it.

On the Atari ST, the *entire* TOS O/S is stored in ROM[1]! This is what enables such fast boot times. In fact, when booting, an ST doesn't require that any drives be connected at all.

Briefly (assuming 4M of RAM installed, which is the maximum supported on the ST), the 16M address space layout is:

`$FF0000-$FFFFFF`	I/O registers (protected, sparsely populated)
`$FC0000-$FEFFFF`	O/S ROM – code & static read-only data
`$FA0000-$FBFFFF`	cartridge ROM
`$400000-$F9FFFF`	(unused)
`$000800-$3FFFFF`	RAM – misc.
`$000400-$0007FF`	RAM – misc. (protected)
`$000000-$0003FF`	RAM[2] – vector table (protected)

Besides the vector table, the protected RAM in low memory can be used by the O/S for system variables, supervisor stack space, etc. The unprotected RAM is then available for user code and data, including a user stack.

Background: Swapping the ROMs

In theory, if you had a real ST and a hardware lab with a ROM burner, you could burn your own O/S ROMs, pull the TOS ROMs out of the ST motherboard, and replace them with your own! Then, when you booted the system, the CPU would jump to the first instruction of *your* "O/S" code. Your code would have complete control over the hardware. In fact, there wouldn't be any other software in memory!

[1] … although updated versions can be loaded from disk if present. ROM TOS checks for the presence of boot media containing newer versions of TOS at startup.
[2] Actually, the first two longwords in the address space are ROM: the initial PC and the initial SSP values to load on CPU reset.

STeem allows you to swap the virtual ROMs.

Under STeem's "Options" window (brought up by clicking on the wrench icon in the toolbar), there is a "TOS" menu item. Note that the "`tos104us.img`" file, found on your local PC's drive, is simply a file that contains the TOS v1.4 US ROM image. By selecting "Add to list", you can browse for an alternative ROM image and add it to the list of known images[3].

If you've built your game as a valid ROM image, you can add it to the list, select it, and then select "Perform cold reset now". On power-up, the ST will boot with your own game ROM image, not TOS!

Of course, you can always swap the original TOS image back in and reset the ST.

Important suggestion: Before swapping out the TOS image, take a memory snapshot from a known, stable state. Once you've installed your game, you'll be able to flip back to the original TOS-based state quickly by simply loading that snapshot. Actually, you can go one better: turn off the virtual ST's power, select your ROM image as the default and finally save a second memory image, before turning on the power. Now, you can flip back and forth between a TOS-based setup and a game-based setup (ready to run as soon as the power is turned on) by simply selecting which snapshot to load.

General Guidance

Before converting your working game to a ROM image, it must satisfy all of the following requirements:

- It is (hopefully) working, well tested and stable.
- The program must not contain any system calls, *with the exception* of `Super`.
- The program should probably not contain any standard library calls[4].
- The program must not assume that any hardware has been initialized, except as noted below. In particular, the 68901 MFP's timers and USART are off and are un-configured.
- The program must not assume that any RAM has been initialized, except as noted below. In particular, it mustn't assume the frame buffer contents have been initialized, even to a blank screen.
- The program must be prepared to perform all needed I/O and vector table configuration, beyond what is noted below.

A program called "`burnroms`" has been provided. It opens an ordinary Atari Executable-format program named "`os.prg`" and converts it to a corresponding ROM image named "`os.img`" (the input file is unchanged). The latter file is suitable for installation as a STeem-recognized ROM image.

What is "`os.prg`"? Essentially, this will be your executable game.

[3] Your instructor has noticed a bug under some versions of Microsoft Windows: the list doesn't appear to be updated when a new image is added. Actually, everything seems to be working. The image in question has indeed been selected as the new default – it just isn't visible in the list of images for some reason.

[4] Actually, you can adjust the linker command line such that the C standard library is included, but this has not been done for you. At any rate, the program cannot invoke any library functions which in turn make system calls. To link to the C standard library ("libc"), adjust the link step in the make file as follows:

```
ld –o os.prg start.o init.o main.o crt0.o libc.a
```

A skeleton project for building ROMs has been provided. It includes the `burnroms` program itself, a sample `makefile`, and:

- `start.s` — the entry point into the firmware, written in 68000 assembly language
- `init.c` — basic initialization code, written in C, called by the above
- `main.c` — a sample `main` function, called from within `start.s`, once the minimal initialization has been completed

This project does build. The resulting ROMs can be installed, but they don't do anything exciting (other than not crash, which is actually quite an accomplishment). The `main.c` module is meant to be replaced with something more useful. The former two modules are meant to be used as-is.

Briefly, the `start.s` and `init.c` files perform the following setup for you:

- put the 68000 into a known state;
- assert a hardware reset signal on the system bus (which is delivered to most I/O interfaces, with the notable exception of the 6850s, which don't have RESET pins);
- initialize the memory configuration hardware
- initialize the video hardware (the framebuffer is not cleared, though); in particular the physical base of the framebuffer is set;
- copy the code & static data from ROM into RAM, and jump to this copied code to continue execution there (this is so that any non-constant global variables don't cause a crash on write – they would if they were in ROM);
- set up supervisor and user stack space and stack pointers;
- set up the 68901 MFP's vector register (to the same value as used under TOS, for convenience, including selecting "software end-of-interrupt" mode);
- assert software resets of the 6850s
- perform a basic initialization of the vector table: all interrupts vector to the start of the O/S by default (i.e. reboot), *except* that an analogue of the TOS `Super` system call is installed (trap #1, system call $20);
- put the 68000 into user mode with interrupts masked;
- invoke `main`

Your code will be responsible various additional setups tasks, including:

- initializing your data structures;
- installing your ISRs[5]
- configuring the 68901 MFP (timers such as timer D, USART, interrupt enable registers, …);
- possibly configuring other I/O interfaces and devices as necessary (if not already in the required mode following the hardware reset – floppy/DMA hardware, for example, in the unlikely event that you use it);
- enabling IKBD 6850 IRQs;
- unmasking interrupts at the 68000 when ready.

[5] Note that the GLU will be asserting HBL IRQs. Your code must either mask these or handle them.

When the `main` function is invoked, the memory map will be as follows:

`$FF0000–$FFFFFF`	I/O registers (protected, sparsely populated)
`$FC0000–$FEFFFF`	game ROM
`$FA0000–$FBFFFF`	cartridge ROM (we're not using this)
`$3FFD00–$3FFFFF`	misc. RAM
`$3F8000–$3FFCFF`	initial frame buffer
`–$3F7FFF`	user stack (grows down)
`$0307D0–`	misc. RAM
`$000800–$0307CF`	game shadow RAM[6] (code & static data)
`$000400–$0007FF`	supervisor stack (protected)
`$000000–$0003FF`	vector table (protected)

Miscellaneous Notes

The boot code provided is currently set up so that the game runs in user mode (after a minimal boot sequence), and can call `Super` to switch to supervisor mode for brief periods when needed[7]. An alternative would be to structure the game as a user mode application which runs on top of a minimal supervisor mode O/S with a trap-based system-call interface. The latter would be a little more work, but would more closely resemble the design of a typical real-world system.

The Atari ST supports both a colour monitor (at low & medium resolutions) and a monochrome monitor (at high resolution). It also supports a variety of memory configurations (not just 4M of RAM, which is the maximum). We have configured STeem so that the virtual ST has a monochrome monitor and 4M of RAM. The start-up code provided assumes this hardware configuration, and will not work with alternative settings. The code could be extended to detect the type of monitor connected. It could also be extended to perform a "shadow test algorithm" for RAM configuration detection. However, neither of these has been done.

[6] Again, note that user code and static data is initially only in ROM, but is copied down to low "shadow" RAM right away, and is then executed/accessed there.

[7] As part of the provided code, your instructor has written his own version of the `Super` "system call", so that your game can continue to call `Super` even though TOS is no longer present.