Tursi's Overview of Bank Switching on the Coleco With SDCC

First off, I apologize if anything seems condescending. I'm going to start with the basics just to make sure I don't miss anything. Feel free to skim or skip sections you already know.

There is a second gotcha - I use a customized toolchain - in particular I have modified the SDCC linker. I probably should make a standalone tool, but I didn't expect anyone else to use this. ;) The net effect is, for my process to work, you need to use my version of **sdldz80.exe** and once you do that, the output file will only be usable by **makemegacart.exe**. I have provided Windows binaries for SDCC3.5.0 and 4.2.0, and a patch file for anyone else. Source for makemegacart is also included. It may require some tweaking as I didn't set it up to build under Linux, but it doesn't do anything Windows specific.

I previously mentioned SDCC's built-in banking support. I have reviewed it in the 4.2.0 documentation, and it basically provides some jump functions and some special keywords that indicate when they are needed. It does appear to require you, the programmer, to track when it is needed manually. Since it's not much simpler than my system, if any, I will continue to use my system shown here.

The ColecoVision cartridge space is 32k in size. The Megacart works by dividing this into two 16k blocks. The lower 16k block is fixed and never changes, while the upper 16k block can be changed to any block within a 1MB space by simply reading or writing to one of the highest addresses in the address space.

For example, keeping in mind that the cartridge space ranges from 0x8000 to 0xFFFF, the Megacart reserves the last 64 bytes for banking. Data CAN be stored here, but accessing it has side effects, so it is best not to use addresses higher than 0xFFBF. With that in mind, accessing 0xFFFF will select bank 0, 0xFFFE selects bank 1, 0xFFFD selects bank 2, etc. If your cartridge is not the full 1MB, just stick to the addresses that suits your cartridge size - other addresses in this range will cause the data to appear to repeat.

In a normal Coleco cartridge, then, you create a single binary image up to 32k in size, and all this data is always available to the title. In a Megacart however, only 16k of data is guaranteed available, the rest is only available while it is selected.

Generally the best way to manage this is to create 16k blocks of data. The boot code, startup header, and anything that needs to be available at all times (such as the C runtime!) should be stored in the fixed block. Anything else can go into banked blocks. Banked blocks INCLUDE the fixed block 0 - you can actually have two copies of it active at once (although there is no clear reason why you would want to).

The last thing to know about a Megacart is that it orders these 16k blocks in the opposite order of a normal, non-banked ROM. That is, if you look at a normal ROM, the boot header is at the first byte of the image. In a Megacart, it is in the last 16k block, and the blocks are ordered from the top down. You don't normally need to deal with this, but it's important to know if you are building your own tools or examining the ROM itself.

In some cases, you can fit all your code in 16k - then you only need to bank for the purpose of accessing data. This is the easiest approach, since you do not need to worry about the program flow and ensuring

that the right bank is in memory for execution - you can just switch to the bank containing the data you need. For systems where the data is level data or other similar concepts, you could conceivably just switch banks and play a different level for each selected bank.

For more complicated programs, you can just as easily bank switch program code -- but the onus is on you to make sure that the correct bank is selected when you call a function, and that you remember that function can NOT change banks again while it is executing, not even to get data! (You can work around this by putting the data for such a function in the same block.)

The most complicated cases involve a function running in a bank that needs to get data from another bank, or call a function in another bank. Again, you have to manage this manually, so it's up to you how much complexity you want to support. But you make this work by simply using what's called a "trampoline" function (because you jump down and bounce back up). Trampoline functions always exist in the fixed memory space, and their job is just to remember what bank you came from, switch to the new bank, call the function or collect the data, switch back to the first bank, and then return.

I'll show examples of all these cases. That's enough theory, let's get to some practice.

The first thing you need to do is update your toolchain. Remember if you follow these steps, you have to use **makemegacart.exe** to convert the ihex to a binary - objcopy won't work anymore. You can switch back and forth or use my linker alongside the original if you prefer, but this document will cover replacement. Makemegacart.exe will also write normal ColecoVision cartridges if the image isn't banked, so you aren't locked out of building old images.

On most systems, SDCC's binaries install in **C:\Program Files\SDCC\BIN** - browse to this folder and locate "sdldz80.exe". (On 64-bit, it may be in C:\Program Files (x86)\SDCC\BIN). Rename this file to "sdldz80_orig.exe" so you always have it handy. My current linker is built off SDCC 3.5.0 or SDCC 4.2.0. It didn't change much between those versions, so it may "just work" if you have a different version.

Drop in my replacement **sdldz80.exe** into the same folder. You also need to put **makemegacart.exe** somewhere - this folder may be convenient, up to you.

You should have an existing project that already builds under the old toolchain - so the first thing to do is verify that the tools build and link it correctly. Most projects use Makefiles - you should not need to make any changes to the compile or link stages (this is why I replace sdldz80 instead of sitting beside it - if you chose not to replace it, then yes, you need to change the linker filename). However the final step that creates the cartridge image is usually (I believe) objcopy or sdobjcopy. You have to replace that command with makemegacart.exe. For a standard 32k cart, no arguments except the ihx filename and rom output filename are needed:

makemegacart crt0.ihx output.rom

When you build, you will get an output something like this:

```
2 banks detected
This is not a megacart - writing normal Coleco cart ROM
```

```
#  SWITCH  ROM_AD   COL_AD  FREE   NAME
=  ======  =======  ======  =====  ===============
X  n/a     0x00000  0x8000  31349  BOOT
```

This is just a little helper table to tell you how your cart is doing for space. Since this is not a megacart, there's just one 32k bank, so no "#", and no "SWITCH" (which is the address you access to toggle the bank). The ROM_AD is the address in the ROM that the bank is located at (0, since there's only one), and the COL_AD is the address it will appear at on the ColecoVision (0x8000 is the start of cartridge space). FREE tells you how many bytes are left available before the bank is full, and NAME reflects a string that you can optionally set in Megacart banks to help identify them - for non-Megacarts it will always use 'BOOT' here.

You will probably see this warning on the linker stage:

```
No FP - skipping emission of #AREA:_GSINIT
```

As long as you only see the one, it's okay - it's a bug in my linker changes that the first area name can not be emitted because the output file pointer isn't open yet. If you see more than one, something else is wrong.

Back to makemegacart, you MAY see warnings like this:

```
Bad bank address 700E
Bad bank address 700F
Bad bank address 7010
```

This is actually a bug in your code -- makemegacart has been told to place data in the RAM addresses. Usually this means that you have global initialized data, but that the linker has not been told to place that data correctly (so it is both not being saved to the cartridge, AND it is wasting RAM space). The crt0.s that I have does not handle global initialized data (adapted from one by colecovision.eu, admittedly many years ago), so unless yours does, the best approach is to just remove the global initialization. If your does - you may have to check with the author how to place the data in ROM. Anyway, this is just a warning, it does not affect your final output (except for the wasted RAM), and the resulting image should be the same as objcopy provided.

Once you are satified that the toolchain works and produces valid images, you are ready to create a banked image. To do this, you need to add a few concepts to your build.

The first one is to place your code and data into named sections. We will use the named sections to place the data into the appropriate places in the cartridge image. To do this, you add two flags to your SDCC compile line:

```
--codeseg name --constseg name
```

You can use different names, but life will be easier if you use just one section per file. You can use the same section name across multiple files (for example, a section named "bank1", or a section named "graphics"). You can also place multiple sections into a single bank -- which one you chose to do is a matter of personal preference. I prefer to put each file into a separate section, as it makes it a little

easier to move things around when memory starts to get tight. (For instance, I'll put a file named 'boss.c' into a section named 'boss').

This of course brings up the point -- you must ensure that a single file compiles down to less than 16k -- ideally a fair bit less, so you have room to assemble things the way you prefer. This includes data files - large blocks of data may need to be broken up.

Next, you need to define the banks themselves, ie: what data goes into each bank. If you have looked at your crt0.s, you probably (hopefully) will see a section similar to this:

```
;; Ordering of segments for the linker - copied from sdcc crt0.s
.area _HOME
.area _CODE
.area   _GSINIT
.area   _GSFINAL

.area _DATA
.area _BSS
.area _HEAP

 .area _CODE
```

What this is doing is defining the sections in advance of them being linked to give the linker an idea of the order you prefer to see them in -- it's a rough approximation of a linker map. Back in the Makefile, there are attributes that define the start address of some of these entries, and any that aren't defined just continue after the one before them.

I adapted this concept for MakeMegacart and made it recognize some special tokens in this. So the most convenient place to store this linker map ends up being in your crt0.s -- if you don't have one of your own, you will need to copy it into your project.

The MakeMegacart "linker map" recognizes areas name "_bank1", "_bank2", "_bank3", etc, as special tags that indicate where in the final image the bytes are meant to be stored. Underneath each bank index, you simply list with .area commands the sections you want to be included in that bank. Anything before the first "_bankN" tag (or after _ENDOFMAP) is placed into the fixed bank, which means that your startup header, initialization code, and C runtime will be always available. For example, this is an exerpt of Super Space Acer's map:

```
.area _HOME
.area _CODE
       .ascii "LinkTag:Fixed\0"    ; ensure there is data BEFORE the bank LinkTags
       .area _ssa
       .area _tramp
       .area _music

.area _INITIALIZER
.area _GSINIT
.area _GSFINAL

;; banking
```

```
.area _bank1
       .ascii "LinkTag:Bank1\0"
       .area _enemy
       .area _songpack1

.area _bank2
       .ascii "LinkTag:Bank2\0"
       .area _human
       .area _songpack2

.area _bank3
       .ascii "LinkTag:Bank3\0"
       .area _wineasy
       .area _winmed
       .area _winhard
       .area _songpack4
       .area _zenithc
       .area _zenithp

;; end of list - needed for makemegacart. Must go before RAM areas.
; This isn't used by anything else and should not contain data
.area _ENDOFMAP

;; RAM locations
.area _DATA
.area _BSS
.area _HEAP
```

Some of these, such as _CODE and _DATA you should recognize from your existing crt0.s. But there are a couple of additional tags I did not yet mention.

First, you may have noticed the ".ascii" directives. This is just storing a string directly in memory - it can be anything you want. There needs to be something in the _CODE area, even a single byte, just to work around the bug that drops the first area tag (mentioned earlier), but if the string starts with "LinkTag:", then MakeMegacart will extract it as a named space for the output report I mentioned above. If you do that, you MUST include the "\0" byte at the end, or the name will pick up extra characters and may crash (that's not terribly likely, but possible).

The other thing I didn't mention earlier was the ".area _ENDOFMAP". This is not actually used by the linker or the output code, but it tells MakeMakecart to stop scanning. Otherwise it will see the RAM locations and think they are also meant to be stored in banks, and generate lots of 'bad bank address' errors. Note that the indentation has no meaning, I just did it to make it easier to read. It's the ORDER of the .area directives that matters.

There's one more change you may want to make to the startup code, and that is to select a specific bank on startup. You don't have to do this, but you must do it at some point before accessing anything in the upper 16k, as the Megacart chip starts up with a random bank active. Just find your startup code (called _startprog:" in my crt0.s), and before the first call (**call gsinit** in mine), insert these two lines:

```
ld     bc,#0xFFFE                ; switch in code bank
ld     a,(bc)                    ; note that this does NOT set the local pBank
```

If you aren't comfortable modifying the assembly, it should be safe to just make sure you set the bank in your C code.

The last thing you need to do in order to build is make sure that you have the section addresses set correctly. This goes back in the Makefile. The code and data sections still start at 0x8100 as they always have (sometimes these are moved, but the point is they are not banked), but you need to tell the linker that each of the _bank areas you defined in the map all start at 0xC000. This ensures that all functions and data will be correctly linked and any calls will go to the correct address.

It may be a bit confusing - but this is an important distinction. The ADDRESS will be correct, but the toolchain has no tracking for banks, so if the wrong bank is selected at the time of access, *the wrong data will be fetched without any indication of error*. This can cause anything from strange behavior (often *sometimes*) to outright crashes (again often only *sometimes*). You have to keep it straight yourself.

The bit to add to the build line is again pretty simple. For each bank, just add a string like this (including the quotes!):

```
"-Wl -b_bank1=0xc000"
```

This passes a command to the linker that the area named "_bank1" is to start at address 0xC000. You can (and will) have multiple of these, all based at 0xC000, and this is just fine.

We'll bring up a few examples, then I'll talk a bit about writing the code and adapting an existing project. My Makefiles assume a cygwin shell with cygwin make and SDCC/bin in the path - you may need to adapt to your own make system and/or set the path before running make. They are designed to function without any libraries or other code.

bank_example0 is presented as a simple 'Hello World' that tests the toolchain. It should generate a standard non-banked Coleco cart which switches into text mode and displays 'Hello World'.

If all goes well, make should do this:

```
$ make
sdasz80 -plosgff crt0.rel crt0.s
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o main.rel main.c
sdcc -mz80 --no-std-crt0 --code-loc 0x8100 --data-loc 0x7000 crt0.rel
main.rel
No FP - skipping emission of #AREA:_GSINIT
makemegacart.exe crt0.ihx example0.rom
2 banks detected
This is not a megacart - writing normal Coleco cart ROM

#  SWITCH  ROM_AD   COL_AD  FREE   NAME
=  ======  =======  ======  =====  ===============
X  n/a     0x00000  0x8000  32271  BOOT
```

(Note the warning about area _GSINIT I mentioned earlier - that's my bug but it's harmless as long as it only shows one such warning).

The resulting ROM file example0.rom should look like this:

One thing to notice in the output folder is all the temporary files are preserved. These can be useful for tracking down bugs by verifying that the output is what you expect - but no file is more valuable than the map (**crt0.map** in this case). Reading this file will tell you the final address of every global object in your project, and this is very helpful for making sure things land where you expect.

bank_example1 expands on this concept by adding data-only banking. All the code lives in the 16k fixed bank, and that code accesses two other banks to print strings to the screen.

First note the addition of a new variable to the Makefile, **$BANKS**:

```
# this rule links the files into the ihx
buildcoleco: $(objs)
	$(CC) -mz80 --no-std-crt0 $(BANKS) --code-loc 0x8100 --data-loc 0x7000 $(objs)

# banks - all defined the same way, we just need to declare them
BANKS = "-Wl -b_bank1=0xc000" "-Wl -b_bank2=0xc000"
```

Note how bank1 and bank2 are both defined with the same address - this pattern repeats for however many you have. Using a separate variable just helps keep the build line cleaner. Also note that we only need to list the "bankN" areas here - as long as we have defined our specific areas under them in the crt0 map area.

We also update the source file build lines - in this case adding lines for text1 and text2. For all three files, we now define a **codeseg** (where the code goes) and **constseg** (where constant data goes), with matching names just to make tracking easier.

```
main.rel: main.c
	$(CC) $(CFLAGS) -o main.rel main.c --codeseg main --constseg main

text1.rel: text1.c
	$(CC) $(CFLAGS) -o text1.rel text1.c --codeseg text1 --constseg text1

text2.rel: text2.c
	$(CC) $(CFLAGS) -o text2.rel text2.c --codeseg text2 --constseg text2
```

We also add text1 and text2 to **$objs**

```
objs = crt0.rel main.rel text1.rel text2.rel
```

Note that the makemegacart line has changed, as well:

```
	makemegacart.exe -map crt0.s crt0.ihx example1.rom
```

There's now a **-map crt0.s** option - this tells makemegacart to read the crt0.s and extract a link map from it. Let's have a look at that.

```
	.area _HOME
	.area _CODE
		.ascii "LinkTag:Fixed\0"    ; also to ensure there is data
		.area _main

	.area _INITIALIZER
	.area _GSINIT
	.area _GSFINAL

	;; banking (must be located before the RAM sections)
	.area _bank1
```

```
        .ascii "LinkTag:Bank1\0"
        .area _text1

.area _bank2
        .ascii "LinkTag:Bank2\0"
        .area _text2

;; end of list - needed for makemegacart. Must go before RAM areas.
; This isn't used by anything else and should not contain data
.area _ENDOFMAP
```

The first thing we've done is inserted an explicit link to area _main under the _CODE section, which will be placed in the fixed bank, along with _INITIALIZER, _GSINIT, _GSFINAL, etc.

Then we've added two new areas, named _bank1 and _bank2. These are magic names that Makemegacart will look for, but they only exist here and in the makefile's BANKS line. The actual code files we build will, in this case, be named segments _text1 and_text2, and because of the order we have listed them here, they will be placed in their respective banks.

text1.c and **text2.c** both contain string data which will be banked, while **main.c** contains the code in the fixed bank. Note that externally accessing the addresses of these variables is completely normal C using *extern*, however, it is our job to make sure the data at the pointed to address is correct when it is needed.

To do that, we need to read (or write) the magic address that triggers the bank switch. These are the macros that I use, and added to main.c:

```
#define SWITCH_IN_BANK1     (*(volatile unsigned char*)0)=(*(volatile unsigned
char*)0xfffe);
#define SWITCH_IN_BANK2     (*(volatile unsigned char*)0)=(*(volatile unsigned
char*)0xfffd);
```

0xfffe selects bank 1, and 0xfffd selects bank 2. (0xffff selects bank 0, which is the same as the fixed bank, so is arguably not very useful). I make them defines so that the code is placed inline, which makes it faster than a function call and available to the optimizer in certain cases.

If you aren't familiar with volatile, its purpose is to tell the compiler that this variable (or in this case memory address) may never be cached and may never be accessed out of sequence - in short, the memory address must be accessed at this point in the code. This is very important for hardware access and even more so for bank switching.

The one possibly mystery may be why there is a write to address 0 (which is what the first half of each of those are). The reason is that SDCC sometimes optimized a pure read away, so that it never happened. I'm not sure if that is a bug as I'm not sure what the C standard says about a read that is never assigned, even volatile, but turning it into a write only adds one more instruction and it always works. If you coded these in assembly instead, you would not need to write the result anywhere. I write to 0 because by default, it's ROM, but if you were working with the SGM or such you might choose a different target to avoid corrupting RAM data. It's only the read that matters.

Once all that setup is complete, accessing the data is very simple:

```
SWITCH_IN_BANK1;
writeString(0x1800+80+4, strBank1);
SWITCH_IN_BANK2;
writeString(0x1800+120+4, strBank2);
```

The pointers work like ordinary addresses, but you have to have the correct bank in place!

Running make will look something like this:

```
$ make
sdasz80 -plosgff crt0.rel crt0.s
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o main.rel main.c --codeseg main
--constseg main
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o text1.rel text1.c --codeseg
text1 --constseg text1
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o text2.rel text2.c --codeseg
text2 --constseg text2
sdcc -mz80 --no-std-crt0 "-Wl -b_bank1=0xc000" "-Wl -b_bank2=0xc000" --code-
loc 0x8100 --data-loc 0x7000 crt0.rel main.rel text1.rel text2.rel
No FP - skipping emission of #AREA:_GSINIT
makemegacart.exe -map crt0.s crt0.ihx example1.rom
Succeeded reading map, found 10 segments
Segment _bank1 in bank 1
Segment _bank2 in bank 2
Creating bank 2 for area _bank2
Segment _text1 in bank 1
Segment _text2 in bank 2
3 banks detected
Writing 128k megacart

#  SWITCH  ROM_AD   COL_AD  FREE   NAME
=  ======  =======  ======  =====  ===============
7  0xFFF8  0x00000  0xC000  16128
6  0xFFF9  0x04000  0xC000  16128
5  0xFFFA  0x08000  0xC000  16128
4  0xFFFB  0x0C000  0xC000  16128
3  0xFFFC  0x10000  0xC000  16128
2  0xFFFD  0x14000  0xC000  16096  Bank2
1  0xFFFE  0x18000  0xC000  16096  Bank1
0  0xFFFF  0x1C000  0x8000  15833  Fixed
```

There's a bit of new output from makemegacart to explain here. The first thing you'll note is:

```
Succeeded reading map, found 10 segments
```

This reports both that the map made sense, and that 10 segments (or areas) were identified. At this

point, Makemegacart has only taken notes about where data goes, it hasn't actually created an image yet. (This means you can define areas you have not coded yet without affecting the image).

Next you'll see it working out the banking. These lines indicate the banks it has detected via the magic names. The numbers count in this case, and this output helps you confirm that it got what you intended. Since a normal Coleco cartridge is 32k, bank '1' always exists, but you will see the 'creating' line as it discovers new banks.

```
Segment _bank1 in bank 1
Segment _bank2 in bank 2
Creating bank 2 for area _bank2
```

Next it will list the actual areas found in the ihx file, as they are identified, and emit a note about which bank they were placed in:

```
Segment _text1 in bank 1
Segment _text2 in bank 2
```

Note that _CODE and other segments in bank 0 (fixed) are not displayed.

Next a report about the number of banks detected, and the size of megacart that will be written:

```
3 banks detected
Writing 128k megacart
```

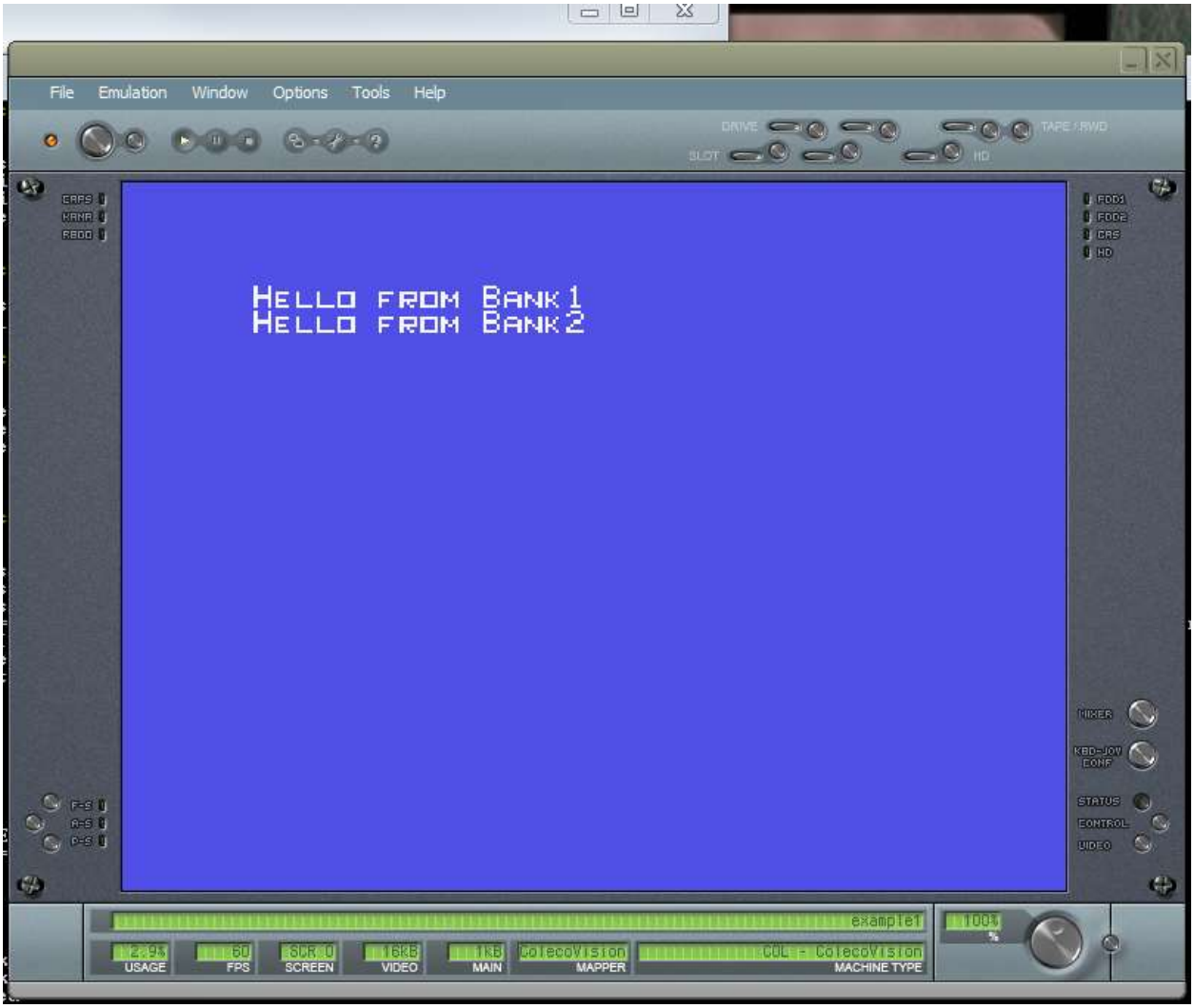
Megacarts currently have a minimum size of 128KB, so anything smaller is padded up. Likewise, the size must be a power of 2, so the only valid sizes are 128k, 256k, 512k, and 1024k. The image will simply be padded to fit.

Next, the bank report is emitted:

```
#  SWITCH  ROM_AD   COL_AD  FREE   NAME
=  ======  =======  ======  =====  ===============
7  0xFFF8  0x00000  0xC000  16128
6  0xFFF9  0x04000  0xC000  16128
5  0xFFFA  0x08000  0xC000  16128
4  0xFFFB  0x0C000  0xC000  16128
3  0xFFFC  0x10000  0xC000  16128
2  0xFFFD  0x14000  0xC000  16096  Bank2
1  0xFFFE  0x18000  0xC000  16096  Bank1
0  0xFFFF  0x1C000  0x8000  15833  Fixed
```

From this you can see that the LinkTags in the crt0 were used for the name, as well as the addresses in the image for each bank, and the number of bytes free. The empty banks are reporting 16128 bytes free because the math assumes 256 bytes of reserved space instead of 64 bytes - you will get a warning if you go over that but the image will still work if you need the space.

Running the image (remember to select Megacart in the emulator) will give you this:



bank_example2 is very similar, but instead of only extracting data, the main code calls functions in the other banks. There is very little change needed to make this work. In fact, the only changes are in main.c, text1.c and text2.c

main.c simply changes the main writes to this:

```
	SWITCH_IN_BANK1;
	showString1(0x1800+80+4);
	SWITCH_IN_BANK2;
	showString2(0x1800+120+4);
```

And just like variables, the function prototypes are references just like normal C. Note how parameter passing is perfectly normal too. (For that matter, since RAM is always mapped in, all global variables are perfectly accessible from any bank, too).

```
// function prototypes
void showString1(unsigned int adr);
void showString2(unsigned int adr);
```

In text1.c, the new function looks like this (and text2.c is very similar):

```
void showString1(int adr) {
	writeString(adr, strBank1);
}
```

Note that the banked code can safely call a function that is in the fixed bank, writeString in this case! Because the fixed code is always available, those functions can always be called, as long as they in turn do not change banks (so that they safely return).

There's no change to the Makefile (except to change the output name) or crt0.s. Running make gives this:

```
$ make
sdasz80 -plosgff crt0.rel crt0.s
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o main.rel main.c --codeseg main
--constseg main
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o text1.rel text1.c --codeseg
text1 --constseg text1
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o text2.rel text2.c --codeseg
text2 --constseg text2
sdcc -mz80 --no-std-crt0 "-Wl -b_bank1=0xc000" "-Wl -b_bank2=0xc000" --code-
loc 0x8100 --data-loc 0x7000 crt0.rel main.rel text1.rel text2.rel
No FP - skipping emission of #AREA:_GSINIT
makemegacart.exe -map crt0.s crt0.ihx example2.rom
Succeeded reading map, found 10 segments
Segment _bank1 in bank 1
Segment _bank2 in bank 2
Creating bank 2 for area _bank2
Segment _text1 in bank 1
Segment _text2 in bank 2
```

```
3 banks detected
Writing 128k megacart

#  SWITCH  ROM_AD   COL_AD  FREE   NAME
=  ======  =======  ======  =====  ===============
7  0xFFF8  0x00000  0xC000  16128
6  0xFFF9  0x04000  0xC000  16128
5  0xFFFA  0x08000  0xC000  16128
4  0xFFFB  0x0C000  0xC000  16128
3  0xFFFC  0x10000  0xC000  16128
2  0xFFFD  0x14000  0xC000  16076  Bank2
1  0xFFFE  0x18000  0xC000  16076  Bank1
0  0xFFFF  0x1C000  0x8000  15843  Fixed
```

Which should look very familiar by now. Running it gives this result:

bank_example3 is the final sample, and demonstrates the last concept, the use of trampoline functions. When a function running in a switched bank needs to access code or data in another switched bank, the easiest way is for it to ask a function in the fixed bank to do it. It's possible to write dedicated functions for every possible permutation, but in my experience, it's easier to introduce a variable that remembers what the current bank is, and so can use that variable to temporarily change banks, and switch back.

I also like to separate out these functions from the normal code (mostly so I can quickly see in the map file how much space they are taking up), and so for that reason I create a new file **trampolines.c** which contains at first a single global variable **nBank**. Now, there are a few ways to track the current bank, but I use an unsigned int that contains the actual memory switch address, because I find that the most convenient (and it produces the fastest switching code, though it uses a byte more memory than an unsigned char might).

```
unsigned int nBank;
```

We do the usual steps in the Makefile of adding trampolines to $objs, and creating a new build line for it. But then we go and add it to the map in crt0.s, in the code section under main:

```
            .ascii "LinkTag:Fixed\0"
            .area _main
            .area _trampolines
```

For this example, we will have main() call a display function in bank1 that displays three strings: its own string, a string copied from bank2, and a string displayed by calling a function in bank2. While the first string is something you've already seen, the other two will be done by trampoline functions.

The first thing we need to do is setup nBank with valid data. My crt0 sets up a default value, but the easiest and safest answer is to simply force the bank you want active as the first instruction of the code.

```
void main() {
	// on entry, the VDP is set up, the audio is muted, so all we need to do is
	// set a default bank and write our strings
	SWITCH_IN_BANK1;	// doesn't matter which one right here
```

Since we now want banking available in multiple files (main.c and trampolines.c), it's handy to move them into a header file, and so **banking.h** is added. We've also extended the declarations of a bank switch by updating nBank:

```
#define SWITCH_IN_BANK1	(*(volatile unsigned char*)0)=(*(volatile unsigned
char*)0xfffe); nBank=0xfffe;
#define SWITCH_IN_BANK2	(*(volatile unsigned char*)0)=(*(volatile unsigned
char*)0xfffd); nBank=0xfffd;
```

It's really important to double check that nBank is set to the correct switch - if you get this wrong, you'll have some really hard-to-debug behavior!

The magic comes from adding a new define that can switch any arbitrary bank - but we will normally use it to return to an old bank, and thus the name:

```
// switch to an arbitrary bank
#define SWITCH_IN_OLD_BANK(x) (*(volatile unsigned char*)0)=(*(volatile unsigned
char*)x); nBank=x;
```

Why wouldn't we always use this? You could if you passed in constants, and it would optimize fine. I prefer the explicit macros. Personal choice.

So now we're going to change the main code to just call a function in bank1. It's already selected, so we just go.

```
	// call the function in bank1
	showStrings1();
```

The function in bank1 is relatively straightforward. All of the functions it is interested in are in fixed memory (in trampolines.c), so it can just call them:

```
void showStrings1() {
	char buf[32];

	// this function is in the fixed bank, and the data is here in bank 1
	writeString(0x1800+80+4, strBank1);

	// this function is in trampoline.c in the fixed bank, and the data is
	// in bank 2, then copied to our local buffer
	tramp_copyFromBank2(buf, strBank2Copy);
	writeString(0x1800+120+4, buf);

	// this function is in trampoline.c in the fixed bank, and will
	// then call a display function in bank2
	tramp_displayBank2(0x1800+160+4);
}
```

Note again how data is passed between functions completely normally.

tramp_copyFromBank2 is the first trampoline example, and its only job is to get data from another bank into RAM so the first bank can use it. Here, we are copying a string, but since memory is limited, in most cases you will probably only be collecting a byte or a few bytes, or maybe loading VDP memory instead of CPU memory.

```
void tramp_copyFromBank2(char *dest, const char *src) {
	unsigned int old = nBank;

	SWITCH_IN_BANK2;
	while (*src) {
		*(dest++)=*(src++);
	}
	*dest = '\0';
	SWITCH_IN_OLD_BANK(old);
}
```

Ignoring that the string copy is not remotely safe (it assumes the destination buffer is big enough and that the string is NUL terminated), the points to look at here are first the variable **old**, which on entry gets a cache of the current version of nBank.

Once we have done that, we are safe to change banks. Just like previous examples, after the bank is changed we can safely access that bank's data, so we just copy the string into the requested RAM buffer. RAM is always available, so it's always safe.

Then we use the new macro to switch back to the previously set bank (which would be Bank1), and then it's safe to return to the caller.

There are two nice advantages to saving the old bank in a local variable like this, as opposed to building some form of queue or manual stack. The first is that, if the function is simple enough, SDCC may optimize out the assignment altogether, and simply never change nBank (after all, it changes back to normal at the end of the function - I have observed this optimization). The second is that only as much data storage as needed is actually used, and it works fine even if functions end up calling other trampolines - it all unwinds cleanly. SDCC also does a fairly good job with this code pattern.

You can probably predict **tramp_displayBank2** by now, and it is pretty simple:

```
void tramp_displayBank2(unsigned int adr) {
	unsigned int old = nBank;

	SWITCH_IN_BANK2;
	showString2(adr);
	SWITCH_IN_OLD_BANK(old);
}
```

Exactly the same as previously, except now we are calling a function. Note how the display address is cleanly passed without any need for fancy code or worry - the stack variables are also in RAM and so are always available.

Make now looks like so:

```
$ make
sdasz80 -plosgff crt0.rel crt0.s
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o main.rel main.c --codeseg main
--constseg main
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o text1.rel text1.c --codeseg
text1 --constseg text1
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o text2.rel text2.c --codeseg
text2 --constseg text2
sdcc -mz80 -c --std-sdcc99 --opt-code-speed -o trampolines.rel trampolines.c
--codeseg trampolines --constseg trampolines
sdcc -mz80 --no-std-crt0 "-Wl -b_bank1=0xc000" "-Wl -b_bank2=0xc000" --code-
loc 0x8100 --data-loc 0x7000 crt0.rel main.rel text1.rel text2.rel
trampolines.rel
No FP - skipping emission of #AREA:_GSINIT
makemegacart.exe -map crt0.s crt0.ihx example3.rom
Succeeded reading map, found 11 segments
Segment _bank1 in bank 1
```

```
Segment _bank2 in bank 2
Creating bank 2 for area _bank2
Segment _text1 in bank 1
Segment _text2 in bank 2
3 banks detected
Writing 128k megacart

#  SWITCH  ROM_AD   COL_AD  FREE   NAME
=  ======  =======  ======  =====  ===============
7  0xFFF8  0x00000  0xC000  16128
6  0xFFF9  0x04000  0xC000  16128
5  0xFFFA  0x08000  0xC000  16128
4  0xFFFB  0x0C000  0xC000  16128
3  0xFFFC  0x10000  0xC000  16128
2  0xFFFD  0x14000  0xC000  16058  Bank2
1  0xFFFE  0x18000  0xC000  16032  Bank1
0  0xFFFF  0x1C000  0x8000  15751  Fixed
```

Note how the map now shows 11 segments instead of 10, but because trampolines is in the fixed space, it's not listed as a banked area.

If you run this rom, you get something like this:

Troubleshooting - a debugger like BlueMSX combined with the map file can help a lot, since you can set breakpoints and step through the functions you suspect are causing you problems. The type of a pointer is more important than ever if that pointer's data comes from ROM, so it's helpful to know (in fact, I had such a bug while writing this - see the notes in example3's text2.c).

The most common failure case with the Coleco of a bad bank switch is a reset - it's just the nature of the machine. But improper behaviour can happen, and sometimes it can be very subtle. If you keep your changes incremental and test often, you keep the scope of change small and make it easier to focus on where the error is.

When changing an existing project, the biggest initial challenge may be getting your fixed bank down under 16k (and you want to leave room for trampoline functions). It may be (and likely WILL be) necessary for you to follow the function call path of functions to determine which ones belong in the same bank, and what data they require. When moving code to new banks, take your time and move slowly. Data is usually more forgiving and easier to tell when you get it wrong, but again, take your time.

Once you get your initial framework up and running in a banked manner, it's not so hard to extend that with new functionality and new data, so enjoy!

Thanks for reading, if you got this far and tried the examples. Hopefully this is useful to someone. I'm not saying this is necessarily the best approach and it's certainly not the only approach, but it's the approach that I use and it works for me. ☺

Files should be up at http://harmlesslion.com/software/colecobanking