Colecovision Multicart Manual

SPARKLETRON

October 27, 2024

Jay Convertino

Contents

1 Usage

1.1 Introduction

This manual describes how to use the Multicart project and its architecture. This cart was design to be cheap, and easy to use. By using a very basic microcontroller and some inline assembly a reliable multicart has been created. This cart allows you to select between 15 different ROMs. These 15 ROMs can be any Colecovision, or Colecovision Super Game Module game you wish. Only caveat is it has to be 32 KiB or less. This cart does not support MEGA ROMs.

1.2 Dependencies

Tested with Ubuntu 22.04

- KiCAD v7.X
- xc8 v2.31 or greater
- sdcc v4.0.0 or greater
- make (build-essential)

1.3 Production File Location

These files will only exist once a build is completed (excluding Gerbers and STL files).

- **Gerbers** schematic/gerber/colecovision_multicart.zip
- **PIC firmware** src/pic_coleco_addr_sel/exe/colecoMultiCart.hex
- **ROM File** src/RODAC/apps/multicart/complete_coleco_multicart.bin
- **STL Bottom** models/bottom.stl
- **STL Top** models/top.stl

1.4 Building

The Easy Way To accomplish the build with a single command, navigate to the src folder that is located in the root of this repo. Once in the source folder simply run...

```
$ make
```

This will build the microcontoller code and Colecovision ROM, if all dependencies are installed. The results of which are located in there respective folders. For the PIC this would be pic_coleco_addr_sel/exe/colecoMultiCart.hex . For the Colecovision this would be RODAC/apps/multicart/complete_coleco_multicart.bin.

1.4.1 Colecovision ROM

The build system used for the Colecovision is called RODAC (Retro Only Device Application Creation). This is a pre-alpha state set of C libraries that use sdcc. The difference between this build system and others is the goal of targeting multiple systems with base libraries and very few ifdefs for switching out code. All paths given below are from the src/RODAC parent directory.

RODAC uses a makefile based build system. The all default target will build nothing. The makefile to build the apps is located in each app. For instance the HELLO_WORLD application is located at apps/hello_world. To execute the build you would simply navigate to apps/hello_world, and run the make command with your target system. Example is the Colecovision would be

```
$ make coleco
```

The Colecovision with a super game module would be

```
$ make coleco_sgm
```

The app of interest is multicart, located at apps/multicart. Details about the code can be seen at 3.2 . For now all we care about is building the code. Building is automated with a makefile that calls two python scripts. rom_header_gen.py, which alters the roms.h header string array with the names of the ROMs given to it. rom_file_gen.py is executed at the end and takes all of the ROM files, including the multicart program and generates the flash ROM image. This currently defaults to 15 ROMs + Multicart for a total of 512 KiB. Each ROM is padded if it is less then 32 KiB. To execute this just simply run make coleco. This, again, will call all the python generators and build all the source code for the multicart and generate the final ROM image.

Burning ROM To burn the ROM you will need to use the file complete_coleco_multicart.bin. This will be located in src/RODAC/apps/multicart. Any programmer compatible with the SST39SF040 will work without issue. For the minipro with a TL866A the example command is

```
$ minipro -p SST39SF040 -w complete_coleco_multicart.bin
```

1.4.2 Microcontroller Code

PIC16F648A is the target microcontroller. Building the code is handled by a makefile that calls xc8-cc. Currently using versions 2.31, but anything greater than that should work fine but has not been tested. Simply navigate to src/pic_coleco_addr_sel and run

```
$ make
```

If xc8-cc is installed and available in the the PATH the build will execute. The resulting outputs are in the exe folder located in the same location. In the exe folder the output needed is colecoMultiCart.hex .

Program PIC Easiest way without install all of MPLAB is to install MPLAB IPE only. This tool will allow you to use a PICKIT programmer with your PC and upload code to controller. The flash board does not have a header for programming the PIC. Easiest way is to wire it up a breadboard with the chip and upload the code with a PICKIT and MPLAB IPE. Again the file you will need is located at src/pic_coleco_addr_sel/exe/colecoMultiCart.hex.

1.4.3 Parts List

- 1 pic16f648a
- 1 74HC148
- 1 SST39SF040
- 3 100nF 10v or greater ceramic capacitor
- 1 bottom case
- 1 top case
- 1 double sided PCB
- 4 #4-24 x 1/2" Philips screws (FASTENAL 31206)

1.4.4 PCB

PCB design was created with KiCAD version 7.X. This is a double sided PCB for the Colecovision only. The design contains three vias for a few traces that needed to switch sides. Exported gerbers are located at schematic/gerber/. This also contains a zip file for production use.

1.4.5 3D printed Case

There are two STL files for 3D printing a cartridge case. The file bottom.stl is the bottom, and top.stl is the top. These files have been printed with ABS at 100.5 percent. The extra half I find offsets any shrinkage in the ABS I own. PLA and such should not need this 0.5 percent increase in size. The case requires 4 screws. I used #4-24 x 1/2" Philips screws from Fastenal, part number 31206.

1.5 Directory Guide

```
│ docs/
│ ├── datasheets
│ └── manual
├── schematic/
│ ├── gerber
│ └── step
└── src/
```

```
│
│   ├── pic_coleco_addr_sel
│   └── firmware
└── model/
```

The listing shows the important directories for this project. Below describes each section.

1. **docs** Contains all documentation related to this project.
   - **datasheets** Contains all IC datasheets used for the hardware.
   - **manual** Contains user manual and wiki that are generated from the same latex source.
2. **schematic** Contains KiCAD schematic for the project, currently version 7.x of KiCAD.
   - **gerber** Export of schematic gerbers are placed here.
   - **step** Export of 3D step model here for case fitment.
3. **src** Contains all source code related to the project.
   - **pic_coleco_addr_sel** pic16 source code and build system, uses xc8 compiler. Any version over 2.X should work fine.
   - **firmware** Colecovision development kit that uses SDCC. The multicart app will generate the project and build rom file.
4. **models** Contains all STL models for cartridge case.

2 Architecture

2.0.1 General Description

The overall architecture is fairly simple and has the following steps.

1. Boot Colecovision in standard banner mode. Display multicart title.
2. Display selection screen in TMS text mode.
3. Highlight current user selection, defaulting at the 1st entry.
4. Once fire is pressed, the highlighted entry is selected for bank switching.
5. Colecovision code will load a new banner telling the user to reset the console.
6. Colecovision will spam the address E001 + entry(0 to 14) 4 times to activate PIC.
7. When E000n enable line goes low and the PIC catches it, the PIC will then set its output address lines to the rom to the input address lines.
8. User will reset console, console will now read the bank switched ROM from multicart.

2.0.2 Excuses

One of the major items you may notice is the need for the user to reset the console. This is due to my design, being a pain to time correctly. By making the user reset the console, which is much slower than the console, the Colecovision and the PIC can be synced easily. Yes that PIC is the reason for this. The PIC at 20 MHz takes 200ns to execute each instruction. Currently my algorithm (see 3.1) will take 1000ns to sample, test the sample and then decide if E000n is active. The E000n pulse was measured to only be 620ns. Since the PIC is simply in a loop sampling, this means the pulse can be missed by the microcontroller. Easy way to fix this is to spam a few reads, in this case four, from the Colecovision Multicart program.

Now the PIC has to set the address. This of course is fairly quick, sometimes too quick and results in corruption. Mostly with Atari games. This resulted in the idea of using a halt instead of reset. Basically the Colecovision code spams the read four times, then goes to a halt. The PIC picks up the read, captures the address and then waits for 1 second. After than second it bank switches the ROM. Since the console is halted this presents no issue and the user can now reset the console to start their game.

Getting the timing between the two perfect could be achieved, heck others have done it. Frankly, this was the easy way out for now. Maybe in the future I'll revisit it. Though the goal is small and cheap. Three IC's is fairly small for the multicart, and as it is now works perfectly for my usecase.

3 Code Highlights

This section is parts of the code I think are interesting or just a bit more information about the files.

3.1 PIC address selection

Using a PIC causes some issues listed in the 2.0.2 subsection. The xc8-cc compiler will add bank switching to every porta/b read even if it is already at the correct bank. This adds 400ns of overhead at a 20 MHz clock. Using inline assembly I was able to read a port and check it again in 1000ns. Still not perfect, but better than the C generated code. The resulting code is below.

```
bcf STATUS,  5
bcf STATUS,  6
test:
movf   PORTA,w
addlw  0xF0
btfsc  STATUS,0
goto  test
movwf 0x20
```

The bcf STATUS clears the bank select bits to 0 before the loop that constantly reads the port. xc8 would put this in the loop. The address pins and the E000n enable are on the same port. This is done by design. Since if its 0, we know that the address bits also contained in the work register are valid. If we do a read after, we run the risk of missing the bits since we are polling the port. By polling the port we don't know if we are close to the starting edge or ending edge of the pulse. If we are at the end we will miss the address since the next read to the port will be too late. The read from PORTA will always be b000XXXXX (X is unknown). The top 2 bits are always 0 by design per the PIC16F648a manual when using a crystal oscillator. Bit 5 is 0 since it is tied to ground. The rest are sampled from the inputs tied to them. Bit 4 is the E000n select line. Bits 3 to 0 are address lines 3 to 0 from the Colecovision. The reason bit 4 is the select line is if its high, 1, and a add of b11110000 (0xF0) is executed this will overflow. Then the instruction brfsc tests the STATUS bit. If the bit is clear, we skip the next instruction, in this case the goto test that creates the loop. If it is 1, an overflow we continue the loop. Once the carry is 0 we exit the loop and move the work register contents to general register h20. This is is so we don't destroy the address value read earlier.

Next we the code will delay for about one second. This is so the Colecovision can reach the HALT state before the new address value for the bank switch is applied. If this happened before the HALT the code would switch to the new bank causing issues as it would start executing that code. The rest of the code simply pulls the contents of general register h20 back to the work register and writes it to PORTB. Then the PIC loops forever doing nothing.

3.2 Colecovision Multicart

Multicart application is written in C and uses the RODAC development environment. This is a pre-alpha set of libraries targeting retro hardware. This code was fairly easy and doesn't have many oddities. This section will highlight interesting bits.

The only library used is the TMS99XX. This was originally written for testing TMS based chips on a PIC mircocontroller. It worked well and has been ported to SDCC. This port needs polish, but does work well enough for the multicart application. This application does not use the NMI as its such as slow update rate for user input. The TMS is put into TXT mode since it allows for 40 columns, and no sprites are needed for a simple menu. Highlighting is done by writing a inverted character set to the pattern table of the TMS RAM. An example of this code is below.

```
/* invert ascii chars */
for(int inv_index = 0; inv_index < sizeof(c_tms99XX_ascii); inv_index++)
{
  /**** write data to port from array of data at index ****/
  VDP_DATA_PORT = ~(((uint8_t *)c_tms99XX_ascii)[inv_index]);
}
```

The index variable tracks the current selection of the ROM. This is from zero to fourteen. Once interesting bug I found was using mod 15 to do the wrap did not work well. It would sometimes go past in the emulator. Changed this to a ternary operator that does a comparison. This works for both going less than zero or greater than 14. Also the top is dictated by the length of the string array. This allows the multicart program to support any size up to the number of columns available. Scrolling has not been added. Maybe in the future. The filtering is only for the up/down input. The fire button is unfiltered.

The controller read is done in a way to make it quick when the button is held, but also work when quickly pressed by the user. A buffer filters the input presses of the user, and is simply shifted until it is 0. Then there is timeout counter that if nothing is pressed, the buffer is reset to its default value of all ones. There is also a delay at the end of the loop to take a pause between screen updates. An example of this code is below.

```
if(!((controller >> FIRE_BIT) & 0x01))
{
  //do we care?
}
else if(!((controller >> UP_BIT) & 0x01))
{
  buffer = buffer << 1;

  if(!buffer)
  {
    index = (index > 0 ? (index - 1) : num_of_roms-1);

    buffer = (uint8_t)~0;
  }
}
/* when down is pressed, and its 0 shift a buffer till its 0, then increment
else if(!((controller >> DOWN_BIT) & 0x01))
{
  buffer = buffer << 1;

  if(!buffer)
  {
    index = (index < num_of_roms-1 ? (index + 1) : 0);

    buffer = (uint8_t)~0;
  }
}
/* nothing? then do a count to allow quick presses to shift the buffer, and
else
{
  counter += 1;

  if(counter > 32)
  {
    counter = 0;
    buffer = (uint8_t)~0;
  }
}
```

Last is the reason for the reading E001 + index four times. This is dues to the PIC is explained in 2.0.2. Example of the code below.

```
buffer = *(bank_switch + index);
buffer = *(bank_switch + index);
buffer = *(bank_switch + index);
buffer = *(bank_switch + index);
```