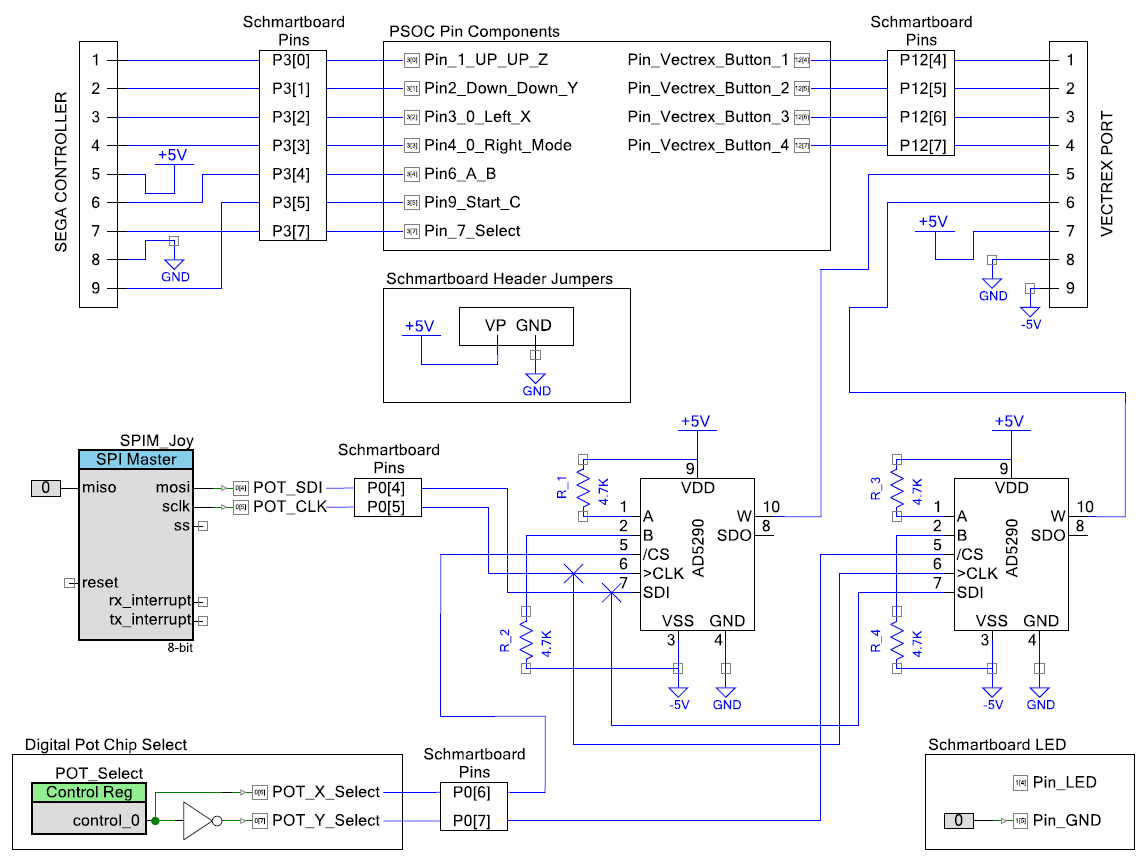
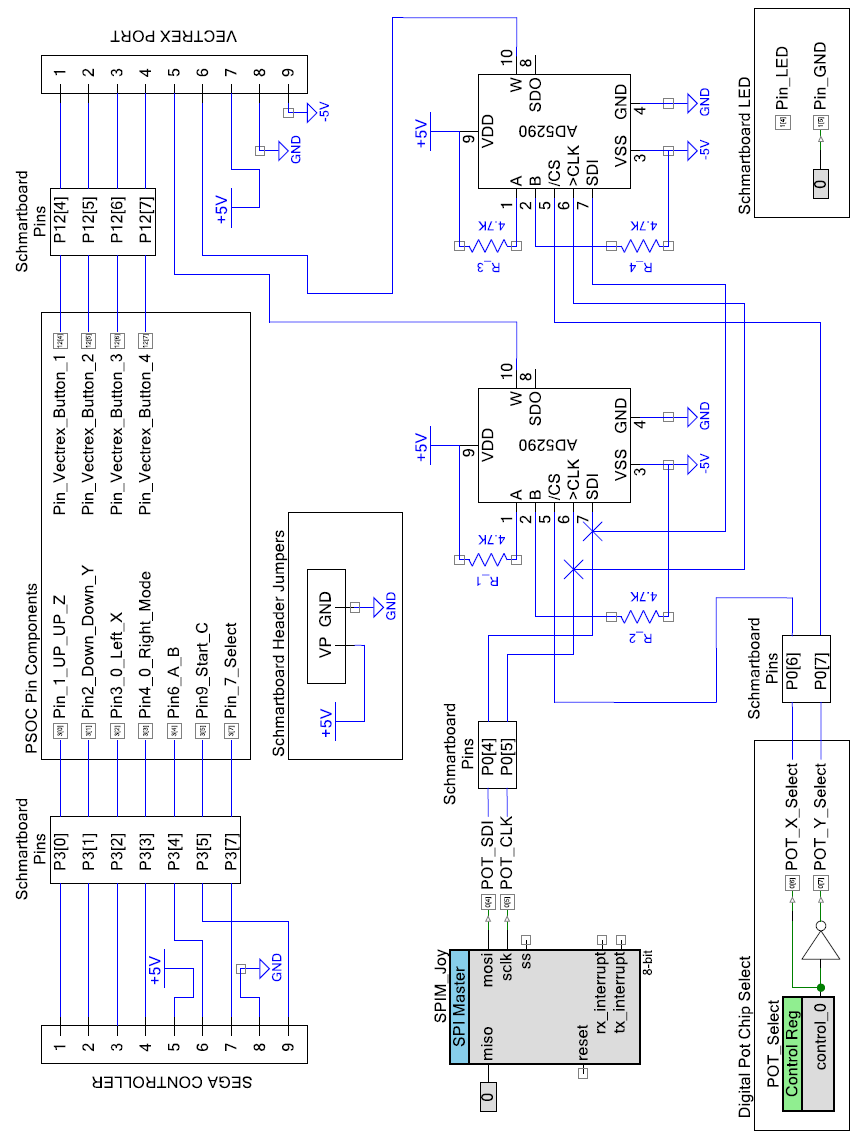
A little while ago I received a Retro-Bit® Super Retro Trio as a gift. This nifty device can play NES, SNES and Genesis games by popping in an original cartridge. Although the Vectrex will always be number one in my heart it was still cool to be able to collect a few games for these other consoles.

Eventually I got tired of playing Genesis games with an SNES controller so I purchased a Hyperkin® 6-button Genesis controller for all that authentic Sonic and Earthworm Jim goodness.

I like the feel of the controller so I naturally decided to see if I could make it work as a Vectrex controller. I didn’t want to modify it to only work on a Vectrex and so I got to work on building out a prototype adapter. This article is a recounting of this experience.

These days building an external controller adapter means that you’ll be dealing with a microcontroller. In looking around my workshop I found that I had an extra PSOC-5 Schmartboard lying around. PSOC stands for programmable system-on-a-chip and it’s a rather powerful and flexible little device. There’s certainly a steep learning curve associated with working with any microcontroller but it’s worth it as it gives such immense flexibility in design. It’s so much easier to work a problem out in code rather than have to deal with external logic chips. I could have easily done this project with a PIC or an AT-Mega but, alas, I didn’t so here’s the schematic.





The parts list for this project is fairly small. There’s the PSOC-5 Schmartboard, two Analog Devices digital pots (part#: AD5290YRMZ10), two DB9 female connectors and four 4.7K resistors. Oh, and a breadboard and lots of wire to connect all this stuff together.

The whole thing is powered from the Vectrex so all voltages come from there. The Schmartboard needs to be powered via a couple of jumpers to one of it’s pin headers (VP and GND on the schematic). The Vectrex also powers the Sega controller.

Starting from the top left of the schematic there’s a DB9 connector for the Sega controller which ties in to the P3 bus pins of the Schmartboard. On the right of the schematic is the Vectrex DB9 connector which connects to the P12 bus pins on the Schmartboard. In between is the PSOC where all of the pins are configured.

Towards the middle left of the schematic is the “SPI Master” component that is configured in the PSOC. This component is needed to drive the SPI signals that program the digital pots. These are shown to the right of the “SPI Master” component. The digital pots can swing between positive and negative voltages. 4.7K resistors are used at each end of the digital pot’s wipers to produce a voltage swing between +-2.5V. This is the voltage range used by the Vectrex for it’s analog to digital converter circuits that it uses to read the joystick pots. In this configuration we will be simulating a Vectrex joystick.

On the bottom left of the schematic is a PSOC control register. We can place a 1 or 0 in this register and it is used to drive the CS pins (chip select) on the digital pots. We only want to talk to one digital pot at a time. A NOT gate component is used to ensure this.

At the bottom right of the schematic are the internal PSOC pin components needed to light up the LED that is present on the Schmartboard. This will be controlled via software.

The basic theory of operation is that the PSOC polls the Sega controller which returns the status of all the buttons and D-Pad. The PSOC in turn maps these to the proper Vectrex pins. The digital pots are used to drive the joystick X/Y pins of the Vectrex. The Vectrex will be able to read 6 positions from the D-Pad.

As with most embedded electronic projects the microcontroller firmware is written in C. The code is pretty straightforward and everything takes place in a loop.

Let’s look at each section of the code and describe what’s going on.

|  |
| --- |
| /\* ========================================  \*  \* SEGA 6-Button Controller to Vectrex Adapter  \* By Dan Siewers (Kokovec)  \*  \* ======================================== \*/  #include "project.h"  // Function prototype: SPI Write function for Digital Pots void Write\_To\_SPI\_Pot(uint8 pot, uint8 tx\_data);  // This struct holds all button states struct Button\_States {  uint8 UP;  uint8 DOWN;  uint8 LEFT;  uint8 RIGHT;  uint8 A;  uint8 B;  uint8 C;  uint8 Z;  uint8 Y;  uint8 X; }Button\_State;  uint8 pulse = 0; // iterator variable for SELECT pulses sent to controller |

This section of the code declares the prototype for a function that will handle sending X/Y pot data to the Vectrex. More on that later.

We also declare and initialize a structure (struct) that will hold all of the Sega controller button states. Each member of this struct will hold either a “0” (button pressed) or a “1” (no button pressed). Then we initialize a variable that will be used as a loop counter later on.

|  |
| --- |
| // Main program starts here int main(void) {  Pin\_7\_Select\_Write(1); // We start with controller Select pin High  Pin\_LED\_Write(1); // Light up the green LED on the Schmartboard  CyDelay(1000); // Give controller a second to warm up  CyGlobalIntEnable; // Enable global interrupts  SPIM\_Joy\_Start(); // Start SPI interface for Digital POTs  Write\_To\_SPI\_Pot(0, 127); // Center POT-X  Write\_To\_SPI\_Pot(1, 127); // Center POT-Y  Pin\_LED\_Write(0); // Turn off green LED on Schmartboard |

This section of code takes care of setting up a few things. First we make sure that the “Select” line that goes to the Sega controller is held high. This is to make sure that the controller doesn’t get confused by thinking that we are polling it for information before we’re ready to do so.

Then we light up the small LED on the Schmartboard to indicate that the adapter is setting things up. After that we wait for a second to give the controller some time to settle in.

Then we start the internal interrupt handler (although we don’t really use it in this design). Then we start up the PSOC SPI component. This component handles all of the complexity involved in sending out SPI signals to the digital pots.

Now that the SPI component is running we can talk to the digital pots. Good thing too because the next couple of lines of code call the SPI Write function to center the Vectrex joystick. If we didn’t do this the digital pots would startup in unknown states.

Finally we turn off the Schmartboard LED to tell the world we’re now going to try talking to the Sega controller.

|  |
| --- |
| // Main loop starts here  for(;;)  {  /\*  \* The Sega 6-Button controller is backwards compatible with the 3-button controllers  \* Because of this we need to send out 3 SELECT pulses in order to capture all of the buttons  \* See table on design schematic for details  \*/    // First 2 SELECT pulses used to capture all buttons except X,Y,Z  for(pulse=0; pulse<3; pulse++) {   Pin\_7\_Select\_Write(1); // SELECT line high   CyDelayUs(12); // "Hold" time (microseconds)  Button\_State.UP = Pin\_1\_UP\_UP\_Z\_Read(); // Read Joy Up  Button\_State.DOWN = Pin2\_Down\_Down\_Y\_Read(); // Read Joy Down  Button\_State.LEFT = Pin3\_0\_Left\_X\_Read(); // Read Joy Left  Button\_State.RIGHT = Pin4\_0\_Right\_Mode\_Read(); // Read Joy Right  Button\_State.B = Pin6\_A\_B\_Read(); // Read button B  Button\_State.C = Pin9\_Start\_C\_Read(); // read button C    Pin\_7\_Select\_Write(0); // SELECT line low  CyDelayUs(12); // "Hold" time (microseconds)  Button\_State.A = Pin6\_A\_B\_Read(); // Read button A  }    // 3rd Pulse used to read X,Y,Z buttons  Pin\_7\_Select\_Write(1); // SELECT line high  CyDelayUs(12);  Button\_State.Z = Pin\_1\_UP\_UP\_Z\_Read(); // Read Button Z  Button\_State.Y = Pin2\_Down\_Down\_Y\_Read(); // Read Button Y  Button\_State.X = Pin3\_0\_Left\_X\_Read(); // Read Button X    // 4th pulse to "reset" controller state (short pulse)  Pin\_7\_Select\_Write(0); // SELECT line low  CyDelayUs(5); // "Hold" time (microseconds)  Pin\_7\_Select\_Write(1); // SELECT line high |

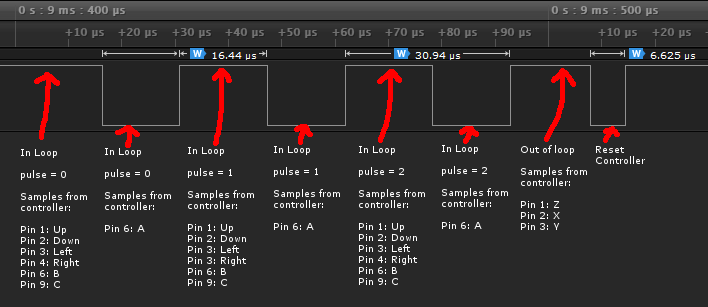
So now things start to get a little more complicated. This bit of code reads all of the buttons and D-Pad states from the Sega controller.

Before we step through the code let’s take a look at what we are trying to accomplish.

When the code shown above runs it sends out pulses to the “Select” line of the controller. What we need to do is send out 4 pulses in total. Each pulse represents a 30.94 us period.

The first 3 pulses are used to read the controller buttons and the 4th resets the controller.

Now, each time the “Select” line changes state between 5V and 0V we take a sample of the controller buttons as shown in the timing diagram shown below.



Looking back at the code we can see that there are two parts to reading the controller buttons and D-Pad. First is a loop that iterates as long as the “pulse” variable hasn’t reach a value of 3. This loop reads the controller button states 6 times as shown in the timing diagram. It’s basically reading the same buttons over and over again until it leaves the loop. After leaving the loop we bring the “Select” line high and the controller will finally return the state of buttons X,Y and Z.

We then send out one more short pulse to let the controller know that we’re done.

|  |
| --- |
| // Map buttons to vectrex pins  Pin\_Vectrex\_Button\_4\_Write(Button\_State.A); // Sega A -> Vectrex 4  Pin\_Vectrex\_Button\_3\_Write(Button\_State.B); // Sega B -> Vectrex 3  Pin\_Vectrex\_Button\_2\_Write(Button\_State.C); // Sega C -> Vectrex 2  Pin\_Vectrex\_Button\_1\_Write(Button\_State.Z); // Sega D -> Vectrex 1 |

The next bit of code maps the captured Sega controller button states (A,B,C,X,Y,Z) to the pins that lead out to the Vectrex. Each line of code sets the corresponding PSOC output pins to match the Vectrex button pins. The PSOC will latch these pins and keep them in that state until we decide to change them.

|  |
| --- |
| // Map Sega D-Pad to Vectrex Y pots  if(Button\_State.UP == 0) {  Write\_To\_SPI\_Pot(1, 255); // Set POT-Y to UP position  }  else if(Button\_State.DOWN == 0) {  Write\_To\_SPI\_Pot(1, 0); // Set POT-Y to DOWN position  }  else {  Write\_To\_SPI\_Pot(1, 127); // Set POT-Y to center position  }    // Map Sega D-Pad to Vectrex X pots  if(Button\_State.RIGHT == 0) {  Write\_To\_SPI\_Pot(0, 255); // Set POT-X to UP position  }  else if(Button\_State.LEFT == 0) {  Write\_To\_SPI\_Pot(0,0); // Set POT-X to DOWN position  }  else {  Write\_To\_SPI\_Pot(0,127); // Set POT-X to center position  } |

This snippet of code handles the D-Pad. It’s a series of “If” statements that run through the logic of figuring out if a D-Pad was pressed in any direction. First it looks at the UP/DOWN pads and sends data to the digital pot via the SPI Write function. If UP is pressed then the function will send a value of 255 to the Y digital pot. The digital pot will in turn send +2.5V to the Vectrex Y input (pin 6). If DOWN is pressed then the function sends a value of 0 to the Y digital pot. The digital pot will in turn produce a -2.5V signal to the Vectrex Y input. If neither UP or DOWN were pressed then a value of 127 is sent to the Y digital pot. The digital pot then sends ~0V to the Vectrex Y input which effectively centers it.

The next part of the code does the same thing for D-Pad LEFT/RIGHT and sends the corresponding voltages to the X digital pot.

|  |
| --- |
| // Set this delay for Sega controller polling rate  // Might need adjustment depending on type of controller  // [10 = ~100Hz] [16 = ~60Hz]  CyDelay(10); // Delay between polling cycles (milliseconds)  } } |

This part of the code makes sure that we wait a while before we poll the controller again. It’s a fixed delay and the program sits here until the delay timer runs out. With a setting of 10 milliseconds we are polling the controller at a rate of about 100 times per second. Apparently most Sega Genesis games poll the controller around 60 times a second but most Vectrex games will do it a little faster. So it looks like Vectrex can what Sega don’t. (Please excuse the pun).

|  |
| --- |
| /\*  \* This function handles writing 8 bit values to the digital pots  \* Digital pots: Analog AD5290YRMZ10 (Mouser part#: 584-AD5290YRMZ10)  \* There are 2 digital pots. One fed to Vectrex Pot-X and one fed to Pot-Y.  \* The SDI and CLK lines are looped between digital pots  \* CS lines are fed to each pot individually (via control register POT\_Select)  \* Function params:  \* Inputs:  \* uint8 pot 0 = digital pot chip select for X  \* 1 = digital pot chip select for Y  \*   \* uint8 tx\_data 8 bit data fed to the digital pot via SPI  \*/ void Write\_To\_SPI\_Pot(uint8 pot, uint8 tx\_data) {  uint8 temp;  // Clear the TX buffer  SPIM\_Joy\_ClearTxBuffer();  // Ensure that previous SPI read is done, or SPI is idle before sending data  temp = SPIM\_Joy\_ReadTxStatus();  while(!(temp & (SPIM\_Joy\_STS\_SPI\_DONE | SPIM\_Joy\_STS\_SPI\_IDLE)));  // Select the pot (X or Y)  POT\_Select\_Write(pot);  // Send the data  SPIM\_Joy\_WriteTxData(tx\_data);  // Ensure data is sent before returning from function  while( ! (SPIM\_Joy\_ReadTxStatus() & SPIM\_Joy\_STS\_BYTE\_COMPLETE ) ); }  /\* [] END OF FILE \*/ |

This is the function that is responsible for sending SPI data to the digital pots. Upon entering the function you have to provide two parameters. The first one selects which digital pot we want to talk to. We give it a value of 0 to talk to the X digital pot and a value of 1 to talk to the Y digital pot. The next parameter is the actual value that we want to send to the digital pot selected.  
As you can see there’s a bit of complexity involved here. It’s good practice to clear the internal buffer of the SPI interface to make sure that it’s empty. After that we select which digital pot we want to talk to by setting the “Pot\_Select” control register to an appropriate value (0 or 1). This sets the CS (chip select) pin on the digital pot that needs to be listening and stops the other one from changing its output. The next line of code waits until all of the data has been sent to the digital pot before returning from the function.

Once this code has been programmed into the PSOC it’s time to test it all out.