The Process and Benefits of Programming Pong for the Texas Instruments MSP430G2553 Microcontroller

Joseph A. El-Khouri

Tri-County Early College

Author Note

Joseph A. El-Khouri, Senior Level Student, Tri-County Early College

The product of this research was supported in part by the donation of resources and time of Mr. Brett Nourrcier, Moog Components Group Inc., Murphy, NC

Correspondence concerning this article should be addressed to Attn: MSP430 Pong Inquiry, Joseph El-Khouri, 153 Wells Street, Andrews, NC 28901.

Contact: joseph.elkhouri@gmail.com

jelkhour@unca.edu

Abstract

This paper explores the process of programming Pong, a simple tennis-like game, by utilizing the Texas Instruments (TI) MSP430G2553s Universal Serial Communication Interface (USCI) and setting it up to output the visible region, otherwise known as the Luminance, of the National Television System Committees (NTSC) standard signal for most US televisions (TVs). However, only the USCI is outputting the luminance; through cycle counting, the synchronization and pulse signals are calculated through cycle counted loops and outputted through corresponding pins on the MSP430G2553. This paper also goes over the relationship of programming the Pong game with the preparation of real-world applications and understanding microcontrollers (MCUs) down to their core level.

Keywords: Texas Instruments, universal serial communications interface, National Television System Committee, television, microcontroller

The Process and Benefits of Programming Pong for the Texas Instruments msp430g2553 Microcontroller

You've all heard of it, some have seen it, even a few of you remember when it first came out! That's right, I'm talking about the first video game ever to be released, Pong. Pong is the game that I decided to attempt to replicate on the TI MSP430G2553 MCU. However, I had no idea how to even approach the task of doing so. Through the guidance, resources, and time of my mentor, Mr. Brett Nourrcier, he was able to teach me how to utilize the components in the MSP430G2553 by giving me one on one lessons in assembly code, the programming language of choice for this project, so I could make the idea come to life. This paper examines the process of designing, programming, and benefits of creating a Pong recreation on the MSP430G2553.

The Brief History of Pong

In 1967, a curious man thought to himself, 'must a television set only be used to broadcast one way entertainment'. Thus, the idea of manipulating a television signal for entertainment purposes came to be; the video game was born. This great man was Ralph H. Baer. His idea for a video game came to life with his invention, The Brown Box. The brown box was a box with a circuit board consisting of analog and digital components that displayed two rectangles and a square on the screen. Through the manipulation of hand held dials, you could interact with these rectangles to hit the square 'ball' back and forth. Nowadays, this is a seemingly 'boring' concept compared to the current standard of what a video game is with their advanced 3D graphic capabilities and breathtaking visuals. In 1967 though, it was a jaw dropping revelation that a TV could be used as something besides a device used to watch shows.

Ralph H. Baer knew that he could do something with this idea. In 1970, after refining the brown box, he took the product and design to the electronics company Magnavox. After doing

some preliminary testing to see if production was possible, they finally signed a license agreement in 1971. With the rights to use the concept of Pong handed over to Magnavox, they quickly started up design and production and released the world's first video game console ever, the Magnavox Odyssey.

The Magnavox Odyssey came as a small, top loading (similarly loaded like the Nintendo 64, or SNES), box with two controllers. It came with 27 games distributed on 12 different cartridges, a couple examples being Baseball, Hockey, Invasion, Roulette, Simon Says, and the infamous Tennis (Pong). The cartridges of the Magnavox Odyssey actually didn't have any data on them. The cartridges just changed the way the two paddles and square ball interacted with each other and how you could control them. I assume a bit of imagination was needed to tell the differences in each game.

Almost as soon as the Magnavox Odyssey hit the sales racks, they were sold out. These things were flying off the shelves. They needed a way to allow people to play this game who couldn't find a console or afford one. Recently, a coin-operated game called Computer Space was released and was a huge hit. Magnavox liked the idea and designed a standalone coin-op version of Pong. Pizza joints, arcades, comic book stores, and many other big hangout areas bought the machine like no tomorrow.

By the end of 1975, over 350,000+ Magnavox Odyssey home consoles sold in the US alone. The Odyssey would have had a much greater sales margin if their ads weren't so confusing though. Apparently, many people interpreted the ad so that the console only worked with Magnavox brand television sets. They soon released another ad saying it worked on any TV and sales increased almost two fold. (Winter, 2010)

The TI MSP430G2553 and its Features

Texas Instruments creates many kinds of MCUs which are basically micro computers that can do basic bitwise operations with a very low power draw. The MSP430G2553 is part of the MSP430 line of MCUs. It is an ultra-low power reduced instructions set computer mixed signal microprocessor. This may seem a bit confusing at first but basically, this just means that its list of commands that you can use with it are not as vast as what you'd find on one of Intel's i7's but it has all of the basic bitwise commands that'd you'd need in most cases. The mixed signal part of the name is pointing out that the MCU has both digital and analogue components built into it.

The MSP430 series MCUs by TI are all 16-bit. I'm sure most people have heard of bits and bytes when referring to computers, but not everyone knows what it means. Describing what they mean is outside the relevancy of this paper and further research of your own should be conducted to fully understand the concept. Most MCUs are either 8-bit or 16-bit. Saying a MCU is 16-bit means that one of the machine instructions uses up 16-bits and registers and memory data can contain up to 16-bits of data which is literally 257 times more data than an 8-bit processor can hold in one of its registers. The internal timers have caps at 0xFFFF (65535 in standard decimal form) whereas in an 8-bit MCU they would top out only at 255. This is one of the main obvious benefits of using the MSP430G2553 for me.

Aside from the amount of data it can hold in a register, it has many components and features built into it for the taking. The first thing I should go over is the 16kb of space for a program which is a pretty huge amount for a small processor like the MSP430G2553. It also has a good amount of Random Access Memory (RAM) to use for storing things, for example, my scoreboard is stored entirely in the RAM which is read and written to about 30 times per second. The MSP430G2553 has (if I remember correctly) 14 different interrupts. An interrupt is when all

processing is stopped, a specific sub-routine is executed and when a return from interrupt instruction is processed, it goes right back to where it was when the interrupt started and continues back to what it was doing before.

The MSP430 series MCUs are notoriously known for their extreme low power use. They have 4 different low power modes. The most basic one only turning off one of the internal oscillators, the most extreme turning everything off, including the MCU itself, only to be awoken through a non-maskable interrupt which is the highest level interrupt there is.

The number of instructions for the MSP430 might not be as many as Intel's processors but it has all the basics and most can be recreated with the combination of two or more of these basic instructions. It has 50 instructions that you can type into Code Composer Studio (CCS) [I'll go over it in greater detail later]. 24 of which are emulated instructions. For example, CLR dst, which means clear destination, is really just MOV #0, dst which is moving the number 0 into the destination but is included for quicker access and easier organization and understanding of the written code.

Another wonderful feature of the MSP430 is the analogue to digital converter. Although I didn't use this in my application, it's a very powerful part of the chip. It allows you to read from analogue components like a thermometer or a pressure sensor by converting their values into a digital number that the chip can read and process like any other number. *("Msp430g2x53, msp430g2x13 mixed", 2013)*

You've heard it mentioned a few times already, but you might not know exactly what the timers in the MSP430G2553 are. There are two of them in the MSP430G2553 and they can be set either to go off when they reach their max count, or at a user specified number. When they go off, they trigger a reset specified by the programmer. I'll go over what it triggers when I describe

the how of this project later on in this paper. Besides the standard timers that come in this MCU, there's a timer called the watchdog timer. This is a vital timer for many applications it's used in. The watchdog timer starts counting up as soon as the processor turns on, and when the timer reaches its max, it resets the MCU and it starts over from the beginning. At first glance, this may seem a bit redundant because if it keeps resetting the processor, how could anything get done? Well, luckily there's a handy option to disable it when programming it, but, especially for medical applications, you don't want to disable it. Instead, you keep resetting the timer before it resets the processor itself so in case the medical equipment fails, instead of getting stuck in an infinite loop of malfunctioning, the watchdog timer will go off and reset it, usually fixing the problem.

The last, but definitely not least feature that I'm going to go over (there's more, but these are the most relevant to the topic) is the USCI. The USCI is an extremely handy feature that I found to be extremely amazing. The USCI has a transmitter buffer which you load with 8 bits of information. That's 8 1s and 0s. Once you do that, it immediately puts the info from the transmitter buffer into a shift register which starts outputting the data on a USCI compatible pin that you define in the code. It outputs it at a rate based off of a clock of choice, and then continues to hold the pin either high (1) or low (0) depending on whatever the last byte sent out is equal to. This is the component I used to output the visible region of the NTSC signal which I'll go over in greater detail later. (“*Msp430x2xx family user's”, 2012)*

Very Brief Look at Code Composer Studio

The wonderful program used to program all of the code for the Pong project was Code Composer Studio (CCS) which is a free (if used with the msp430 series MCUs) compiler for C and Assembly code made for TIs line of MCUs. It's packed full of features like the debugger,

dissembler, and ability to read ports, registers, flash, RAM, and everything else while a compatible TI chip is plugged up.

Probably the biggest and most used function of CSS was the debugger. It allowed me to step through my program one instruction at a time so I could monitor all the ports, values of the registers, RAM, and everything else in super slow-mo to make sure that there were no irregularities in the execution of the code. From double checking that a pin was set high to reading a particular value in the RAM, I was always on the debugger if I wasn't typing code. ("Code composer studio, 2013)

The Assembly Language

When you're programming, there's usually a few languages you can pick from to make your programs. By far, the most popular and well known language is C which is a high level programming language. When you compile high level languages, like C, they get converted into assembly language. Assembly is a low-level programming language that closely resembles machine code, which is the 1s and 0s that a processor reads to execute instructions. Each instruction written in assembly uses mnemonics to represent this machine code, otherwise known as opcode. Most opcodes require 1-2 operands. An operand is a destination, source, or both that the processor manipulates according to what the opcode is.

To make this easier to comprehend, here's an example. Let's say we have the assembly mnemonic "MOV.W #50, R8" which stands for "Move the word sized (16-bit long version of) decimal 50 into the 8^{th} register." MOV.W means move a specified 16 bit integer into a specified place. The # sign means the standard decimal form of 50, and the R8 is a register. A register is a place to store any number between 0-65535. Now that we know what the code means, let's convert it into machine code. The MOV.W command gets converted to 1010, the 50 gets

converted to 00110010, and R8 gets converted into 1101. The Assembler (the assembly compiler) then puts it all together to make the machine code 1010 0011 0010 1101 and writes that to the chip (This is not an accurate representation of actual MSP430 machine code, it is only a demonstration made to understand the concept easier).

There are many people who argue that assembly is a dead language and that it's not really that important to know, but the fail to realize that all compilers convert their code to assembly first. Because of this, sometimes, the fault is in the converted assembly code and not in the logic of the higher level code. Thus, to debug and fix the problem, you have to go through the assembly and solve the problem. Without the knowledge of assembly, this would be impossible. Another plus of knowing assembly is speed. Programs written in higher level code are almost 100% of the time not going to be as efficient as if they were written in assembly. If speed is critical (which it was in my case), then assembly is more than a perfect choice. Also, assembly language programmers have the opportunity to earn more money in an environment where programming assembly is an option. Lastly, and my most favorite part of writing assembly, is its ability to control the hardware in ways that higher level programming can never do. (Burt, 2004)

NTSC Signal

NTSC is the signal that the USA uses to broadcast television pictures over. It was developed sometime in the late 1940's. The signal can be simplified and separated into 2 parts to where it's easy to understand; a synchronization phase and a visible transmission phase. To properly output the synchronization phase, you send 0.3 volts (v) to the television for 4.7 microseconds. A microsecond (us) is 0.000001 seconds. Since the processor is running at 16mhz, it goes through 16 cycles per 1us which makes this timing easily feasible by multiplying 4.7 microseconds by 16 cycles/us which tells us we need to send 0.3v to the TV for 75.2 cycles.

Since you can't have part cycles, we'll just round it to 75 cycles. Since the signal is actually a little flexible, 75 is well within the range of error for the NTSC signal. After that, to finish the synchronization signal, you drop the voltage going to the TV to 0v for 58.8us. This is 940.8 cycles. Once again, you can round this number to 941 cycles. You repeat this process of sending 0.3v and dropping it to 0v 19 times. Once you do that, you can start the transmitting of the visible portion of the screen. (You can convert the times to cycles from now on yourself if you want. Just multiply 16 by the us) First, you must drop the voltage going to the TV to 0v for 4.7us. This part of the signal is called the horizontal sync pulse. This pulse brings the electron beam in a CRT TV to the left side of the screen so it can prepare to display a line of information. After that, the voltage is brought up to 0.3v for 5.9us. This part of the signal is called the "Back Porch". This part of the signal prepares the TV for displaying information that it's about to receive. In color TV, this is also where the color burst happens. Since the color burst isn't relevant to my project, I'll only touch on it. The color burst is a quick burst of information that contains the color for the upcoming luminance signal. After the back porch is finished, you start sending a variable signal between 0.3v (black) and 1v (white) that contains the brightness, or 'luminance' of the picture for 51.1us. Since I was only displaying white and black, I just switched between 1.0v and 0.3v to make white and black respectively. Once you finish sending the line of data, the last part of the visible lines is sent, the "Front Porch". This is bringing the voltage back to 0.3v for 1.4us. Once you finish that, you repeat the visible portion process 242 times to send a full picture of the screen to the TV. Keep in mind that this whole process of sending 19 synchronization lines and 242 visible lines takes only about $1/30^{th}$ of a second. That's how fast the processor is working at making these demands while at the same time doing all the processing of the pong game. It's really a remarkable chip it is. (Fink, 2012)

Walkthrough of the Code

Wow, I haven't looked at or touched many parts of this code for months due to not needing to. I've mostly been doing the pong logic programming recently so apologizes if this part seems a lot like stream of consciousness. Please refer to the code printout when line numbers are referenced. I'm also not very in-depth, I'm assuming that the reader has a basic understanding of assembly and can visualize processes without much difficulty.

Alright, let's start at line 6 with the initialization routine. This is a line of text declaring a header file that is used throughout the code. It defines a bunch of shortcut commands and names that relate specifically to the MSP430G2553 chip. Lines 9-13 are telling the assembler to put all of this code below into the program memory and line 13 is telling that at startup, to go to line 17 which is labeled RESET. Line 17 tells where to put the stack pointer which is what keeps track of where you currently are in the program. On line 19, the watchdog timer is stopped because we don't need it for this type of application. Lines 21-23 is setting up the processor to work at 16mhz. Lines 25-27 set up the timer to 1060 cycles which is the length of a scan line for NTSC. We also disable the timer initially. We do the same thing with the next timer in lines 29-31. Lines 34-43 set up the USCI for transmitting the visible portion of the screen by setting port 1.7 up for spi mode. It runs it at a speed slower than the 16mhz the clock is running at (4mhz I believe, 16mhz/4 = 4mhz) This allows the USCI to send data while doing normal processing stuff at the same time. Lines 46-54 set up the pins on the physical chip itself as inputs and outputs. Lines 73-85 give registers (places to store numbers) names so it's easier to refer to them later on in the code. Lines 88-108 set up the scoreboard which is displayed at the bottom of the screen. It puts a predefined shape into the space allocated in the RAM by ScoreBoard (which can be found at line 536) by putting the address of ScoreBoard into the register GameReg which is a

register that is used for many many things throughout the code. It's basically a universal register I use when I can. Lines 111-119 start the timers 182 cycles apart from each other. The second timer is used for the back porch timing. I kinda cheated with the NTSC signal's back porch and extended it to position the picture horizontally since it's really just black level (please don't shoot me). Lines 127-132 initialize the positions of the two paddles, the ball, and the velocity of the ball. Lines 133-138 clear the line buffer which is a string of data that is calculated during each back porch before transmitting it through the USCI during the visible portion of the NTSC signal. Line 140 is just starting the line count of the NTSC at 0. This next part is where the power saving magic comes into play. Line 141 turns off the computer and enables general interrupts. Remember, the timers are still going, but the computer's turned off. Once the first timer goes off, it triggers an interrupt which can be found at line 583. It's labeled to go to TIMERRESET when it triggers so let's do that.

TIMERRESET can be found on line 148. It starts out by turning off port 2.0 which is connected to the composite signal as the 0.3 voltage. This starts everything at 0v. Remember, the horizontal sync pulse starts 0v. Lines 149-161 are basically a directory for where to go depending on what line you're on. You may notice that I butchered the NTSC standards, but since they're so flexible, they work just fine. Even digital screens read it just fine. I was playing this on my dad's 40" LCD. Lines 165-172 are executed when the line count goes above 261. It basically does the final visible line as a purely black line and resets the line count. Lines 176-328 do all the logic programming for the paddles, ball, collision, score increasing, goals, etc. Lines 330-337 are blank scan lines. Lines 339-349 are the vertical sync lines. Lines 352-381 do the exact thing as lines 330-337 except output blank lines with the USCI transmitter to 'warm it up'

because of a strange hook caused by inaccurate timing that would appear in the first few lines that it would output. Lines 383-413 is the display code for the score board.

This next part is probably what took the longest part out of my time when designing this. Lines 422-527 is the output code for the paddles and ball to the visible screen. I'm going to try to go through it part by part. You can see cycle counting in lines 441-443 to get the timings correctly for the horizontal sync pulse. Now, remember that second timer I set at 181 cycles after the initial timer? Well this is where it comes into play. On 446, I enable the interrupt for the second timer. Quick note, lines 450 and 451 are unused portions of code that I apparently forgot to take out. Please disregard them. In lines 453-460, I check to see if the left paddle is within the range of the current line we're on. If so, put a two pixel wide line on the current line buffer on the first byte of pixels. You can see a lot of cycle counting artifacts that ended up not being used due to the idea of using a second timer for this part. A lot of redundant jumps are used throughout this section of code too. Lines 467-475 do the same thing for the right paddle on the other side of the line buffer on the 20^{th} byte of pixels. Lines 481-507 write the ball (which is 2 pixels wide by 4 tall; The aspect ratio of pixels tall and wide end up making it look square) to the line buffer. It does this by first seeing if the Y position of the ball falls within range of the current line the processor is displaying. This can be seen on lines 482-487. Then it puts the Ball's current X position into the game register and then it clears the first three bits of the game register the rotates it right three times to get what byte it's on. It then moves the line buffer address that many bytes over to get to that byte. Now that that's done, we can use the game register for another purpose. I move a two pixel mask of the ball into the game register. Next I put the ball's X position into the JumpTimer. (this is because I needed a register that wasn't being used at the time so the JumpTimer kinda dual purposed here for a little bit). Then I clear the top 5 bits of the

value in the JumpTimer to get the numerical value of what bit the ball is on. This was an ingenious method thought up by my mentor and definitely made the whole process 1000x easier than my previous method which ended up using too many cycles. If it's already on the correct bit, it goes ahead and writes the ball mask to the byte using a bit set command. Otherwise, it rotates the ball mask right, decrements the JumpTimer, and repeats as many times as necessary to get the ball in the right place. If the ball mask ends up getting cut off, I make up for it by setting the Most Significant Bit (MSB) in the next set of 8 pixels in the line buffer. Line 511 then resets the line buffer address, but this can actually be omitted. I guess I didn't notice I already did that on line 447. Then it returns from the interrupt and awaits orders for the visible area. Lines 518-527 are where the transmitting actually happens. It loads the data of the LineBuffer into the Transmit Buffer of the USCI one byte at a time. Since whatever's loaded into the transmit buffer gets immediately put into the shift register (as long as there's not already something in it), the transmit buffer is loaded in quick succession once then carefully timed afterwards. It outputs all 20 bytes of code and does the whole process over again!

Importance and Relevance of Programming Pong

The main reasons for doing such a ridiculous project like programming pong in assembly is for the experience and getting to better understand a microcontroller and processors in general at their core level. Like I said earlier, all compiled code gets transferred into assembly so it's extremely helpful to understand it especially when trying to debug it. Programming pong has made me actually like assembly a lot and I plan on using it in the near future on personal project.

Employers look at people why get things done, and last time I checked, I don't think many people have programmed in assembly for their senior exit project. I'd consider this getting

things done. You also can't get a job doing something you've never done before, so assembly opens quite a few doors in the job opportunity room. (Kegel, 2012)

Conclusion

If I can't stress this enough, none of this would have been possible without my mentor. He took 2 hours out of his schedule every week to teach me assembly code for about 5 months. That's over 40 hours of dedication! He even donated the resources, the MSP430 Launchpad, the MSP430G2553; everything he's done for me will no doubt help me in my college journeys and job finding quest.

I'd also like to thank my parents and aunt for driving me out to Moog Components Group Inc. every Tuesday and picking me up. Without them, I wouldn't of been able to do this either!

I really hope you enjoyed reading this as much as I enjoyed making the program. Remember, if you need to contact me, send me an email! I'd love to hear what you have to say.

References

Burt, G. (2004). *Why study assembly language?*. Retrieved from http://www.csee.umbc.edu/courses/undergraduate/313/fall04/burt_katz/lectures/Lect01/why.html

Code composer studio (ccstudio) integrated development environment (ide) v5. (2013). Retrieved from http://www.ti.com/tool/ccstudio

Fink, D. (2012). *National television system committee*. Retrieved from http://www.ntsc-tv.com/

Kegel, D. (2012, Dec 18). *How to get hired -- what cs students need to know*. Retrieved from http://www.kegel.com/academy/getting-hired.html

Msp430g2x53, msp430g2x13 mixed signal microcontroller (rev. i). (2013, May 01). Retrieved from http://www.ti.com/litv/pdf/slas735i

Msp430x2xx family user's guide (rev. i). (2012, Jan). Retrieved from http://www.ti.com/lit/ug/slau144i/slau144i.pdf

Winter, D. (2010). *Pong story the site of the first video game*. Retrieved from http://www.pong-story.com/intro.htm