**ECE 395**

Fall 2019

Final Project

# **Digital Guitar Pedal Audio Platform**

Marco Palella and Will Newman

Tuesday at 3 PM

# **Abstract**

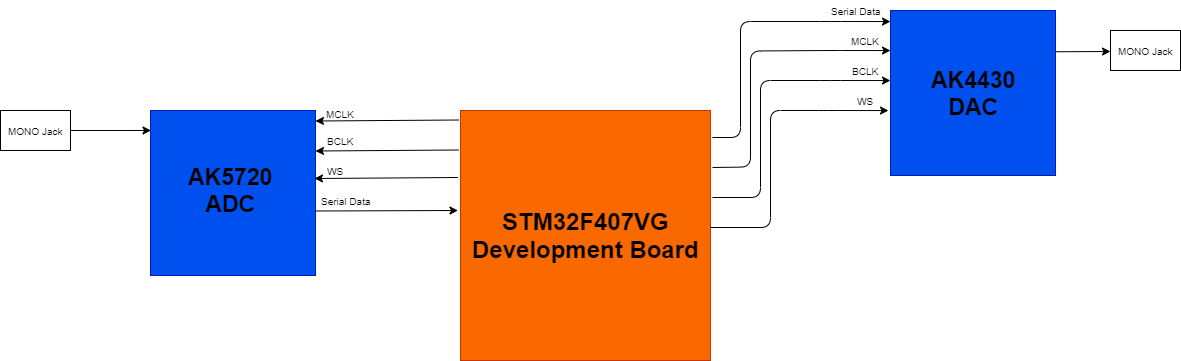
The goal of this project is to design a hardware platform capable of digitally processing an analog signal in the same manner as a guitar pedal. This would require using an ADC to convert analog data from a TS ¼” jack to digital data. This digital data would then be sent to an ARM processor to be digitally processed. In the future, the job of this processor is to produce different guitar effects such as reverb, delay, equalizer, flanger, and distortion effect. The other job of this processor is to take in several GPIO inputs connected to potentiometers and map them to several different parameters on these effects. These will be future goals for the processor. As of now, the goal of this semester was for the processor to take audio in and pass it out of the processor. Since time permitted, we were also able to create a basic distortion effect as a proof of concept of our device. After processing the data, the processor will send it to a DAC, which will output the data as an analog signal through another ¼” TS jack. This platform would need to be designed to ensure a clean signal through the system (ie. low SNR). Because of this, there were several design considerations involved to mitigate SNR, along with mitigating latency and signal distortion.

# 

# **Hardware Components**

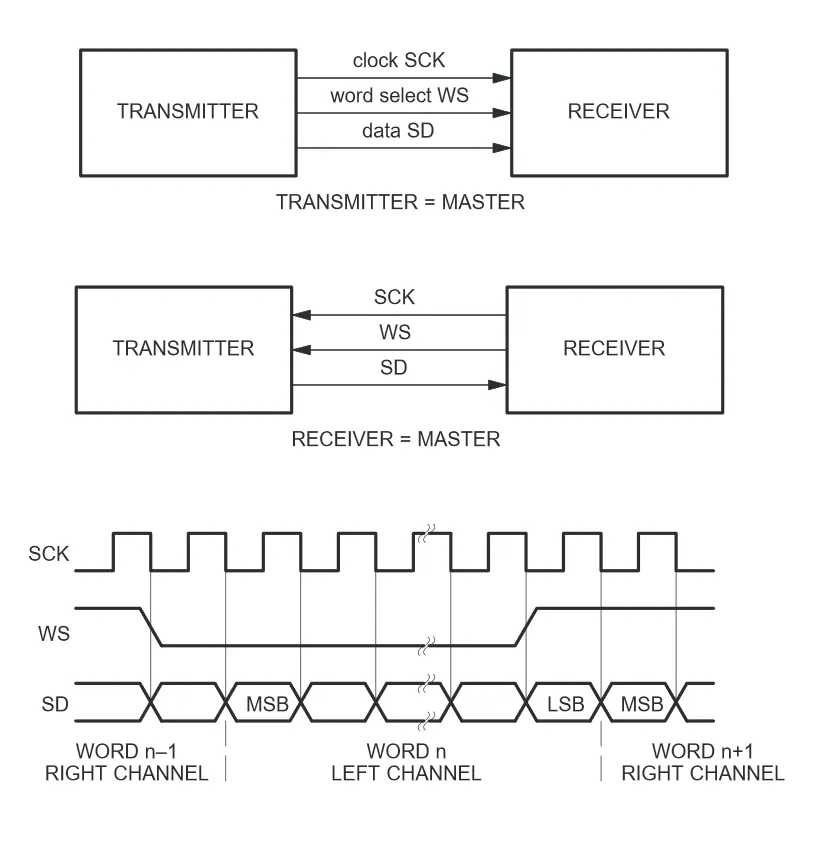
**Overall Design:**

**Block Diagram:**

****

Our project used the STM32F407VG development board. This board made it easier to work with the processor and made our hardware setup/testing much simpler. Using I2S communication, Serial data is received and transmitted by the STM32 processor to an external DAC/ADC. The circuit components that are important on the Development board are the external clock circuit, the processor unit, the power regulation circuit, and the circuitry used to program the processor.

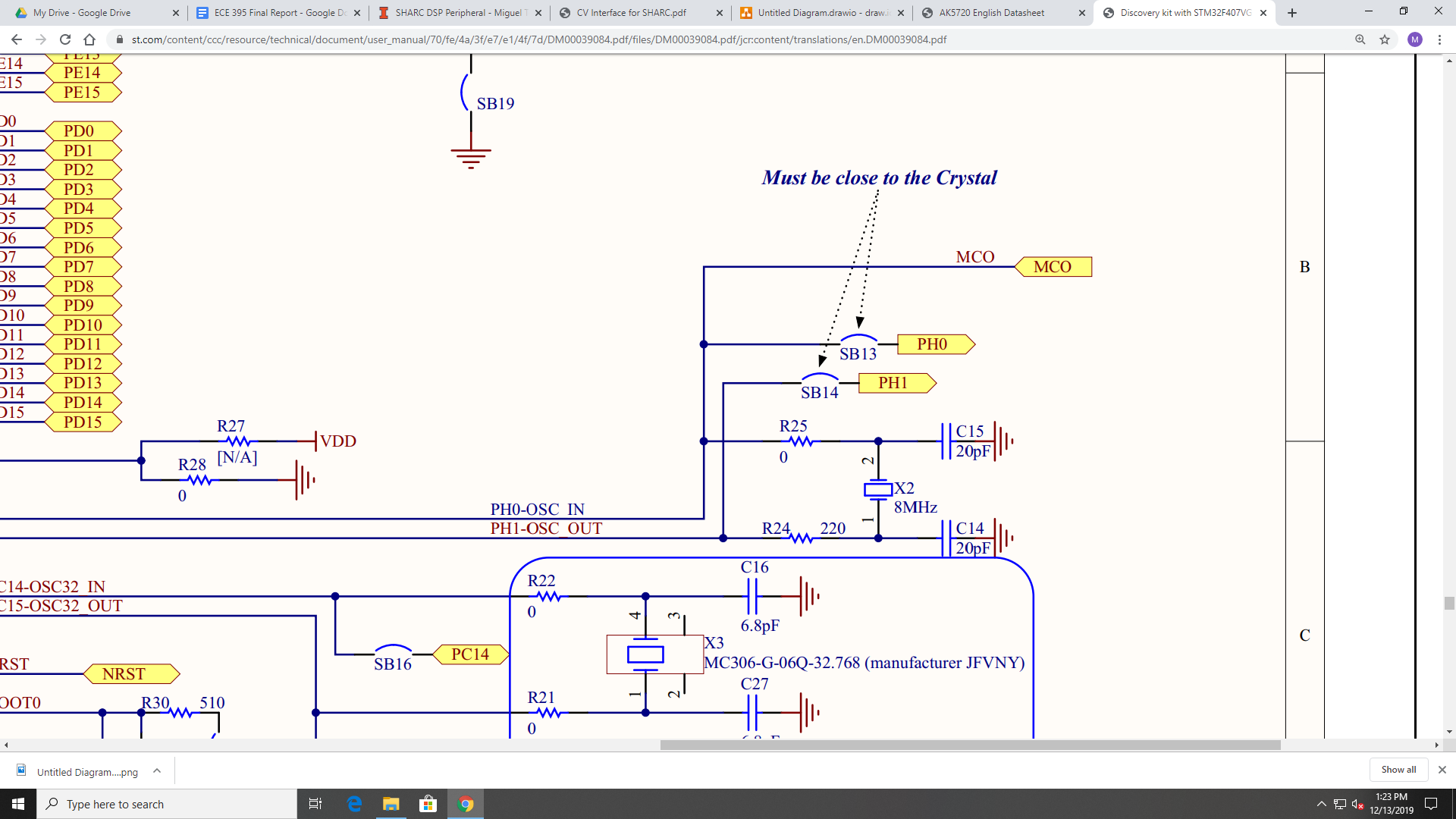
# **I2S Communication:**



I2S serial communication is used for audio applications to transmit digital audio data serially. There are three channels used in I2S: the Serial Data channel (SD), the WS (aka LRCLK) channel (word select), and the SCK or BCK channel (sample/bit clock). Data is transmitted via the SD channel, with the BCK telling the master when each bit is being sent to it. The WS channel tells the master which channel that data is coming from, which is useful for transmitting stereo signals. The word is sent in little endian order to the processor. Another clock that is typically used in I2S communication is the Master Clock(MCK). This clock is used as a clock for the external clock, and typically runs much faster than the bit clock.

# 

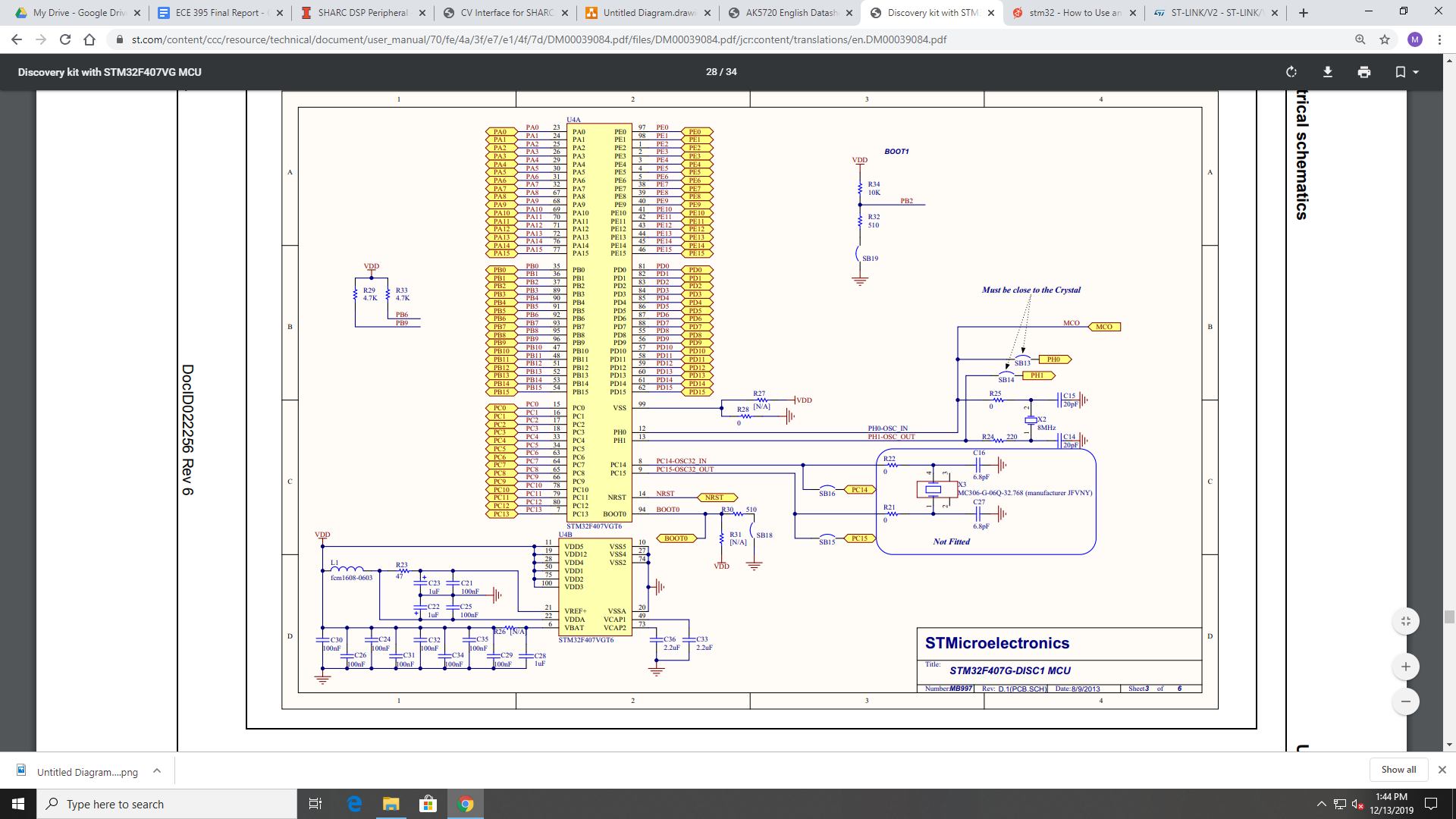
# **External Clock Circuit:**

****

This external clock circuit is necessary for audio processing because it provides a faster clock to the processor. Because our audio processing is real time, the internal processor clock is not fast enough. This 8MHz clock provides a clock speed fast enough for what we are doing.

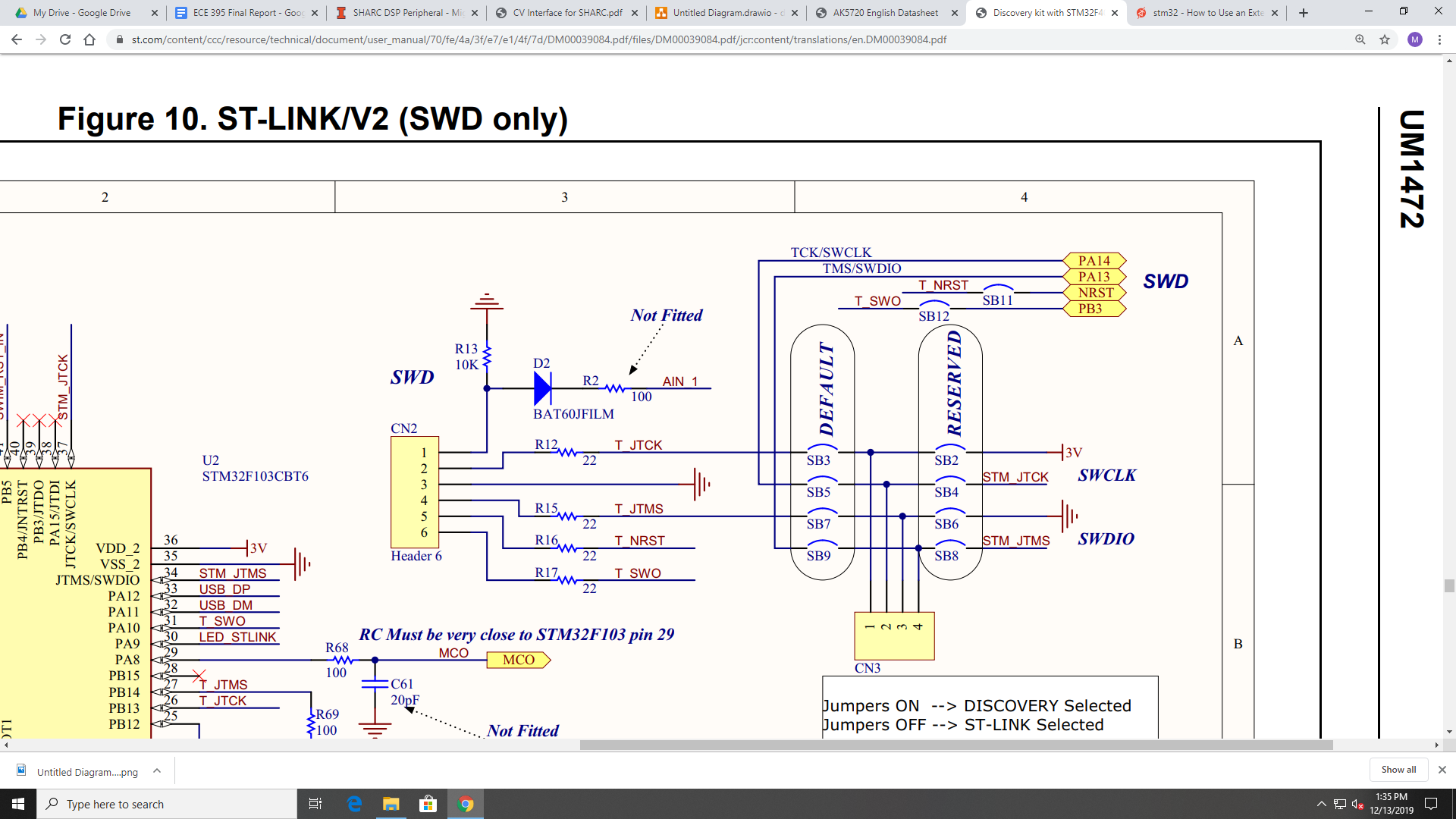
# 

# **Processor Setup:**

****

The processor has some external components connected to it, which should be considered when setting up a PCB.

# **Programming the Board with SWD:**

****

In the future, a method to program the board is necessary to reprogram and update the effects processing on the processor. Using an external [ST-Link](https://www.st.com/en/development-tools/st-link-v2.html), it is possible to program the processor using only four connections. The development board uses a mini-USB to program the board, but using an external link would remove a lot of components from our PCB. For a tutorial on how to program the board using ST-Link, see the following [link](https://www.st.com/en/development-tools/st-link-v2.html).

# **ADC:**

**Description:**

With the help of Professor Haken, we decided to use the AK5720 ADC for our project. This ADC has a resolution of 24-bits and a sample frequency of 48 kHz with I2S serial communication. The ADC also has a built in amplifier, which means that we don’t need a preamplifier circuit. It took a while to get this chip working; it needs to be configured in a very specific way, so if anyone looking at this document is going to try and change the setup, please read page 4 and page 16 thoroughly before making adjustments. Also, keep in mind that the large gain is going to boost your signals considerably. We were seeing clipping of a signal at a peak to peak voltage of 100 mV. If you are going to be working with larger signal, its recommended that the GSEL pin be grounded. This will keep the ADC from creating such a large gain. Also, make sure to look at the analog characteristics sheet on page 6 to see the typical operating input voltage range that the ADCs made for. Serial Data is sent from the ADC via the SDTO pin, and the clock pins are BICK(SCK), MCLK, and LRCK(WS).

**Schematic:**

[Datasheet](https://www.akm.com/content/dam/documents/products/audio/audio-adc/ak5720vt/ak5720vt-en-datasheet.pdf)



# **DAC:**

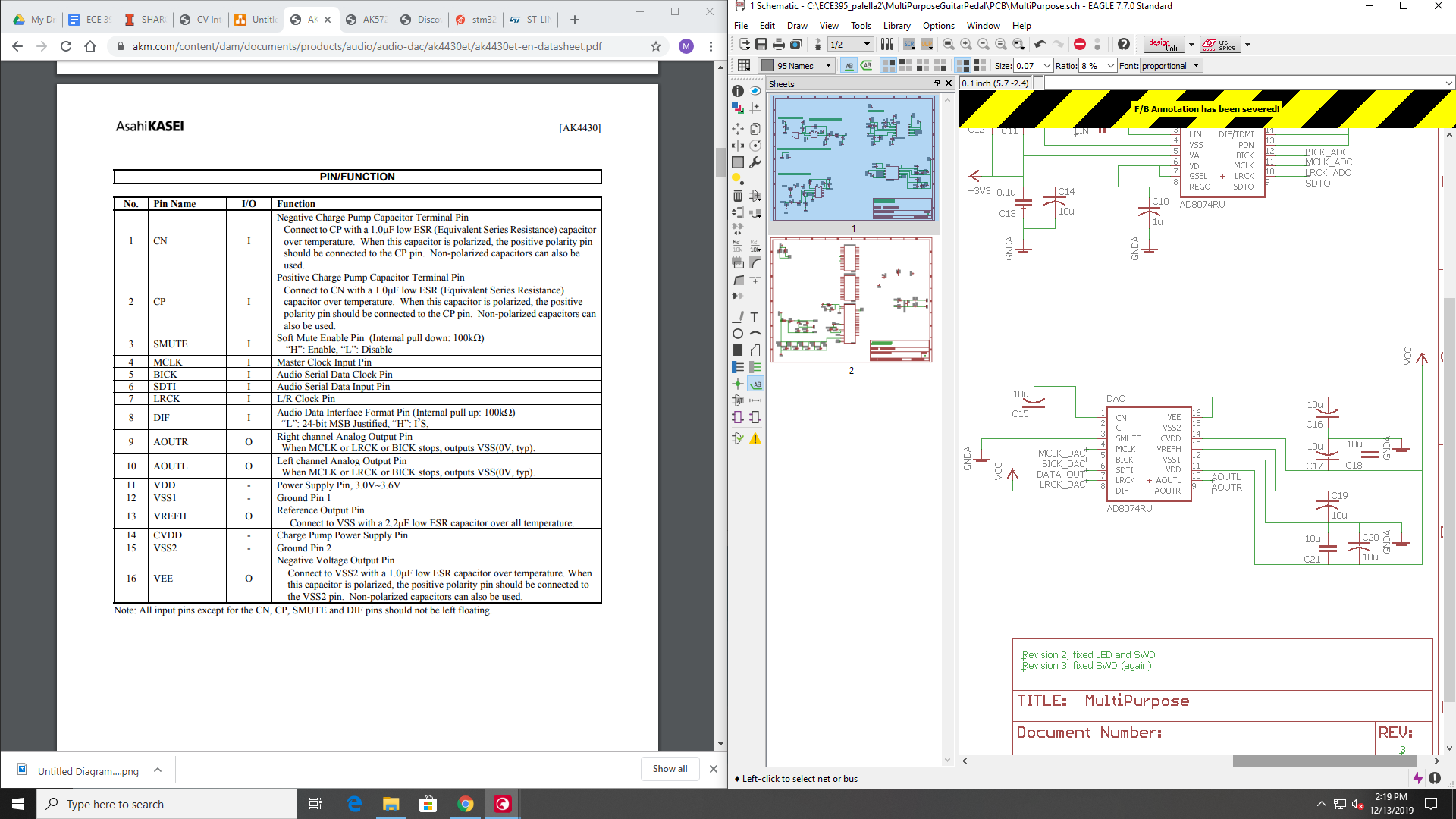
**Description:**

\*note: if any datasheet links are no longer working, all of the appropriate datasheets for all components used are located in the datasheets folder in the provided Github Repository.

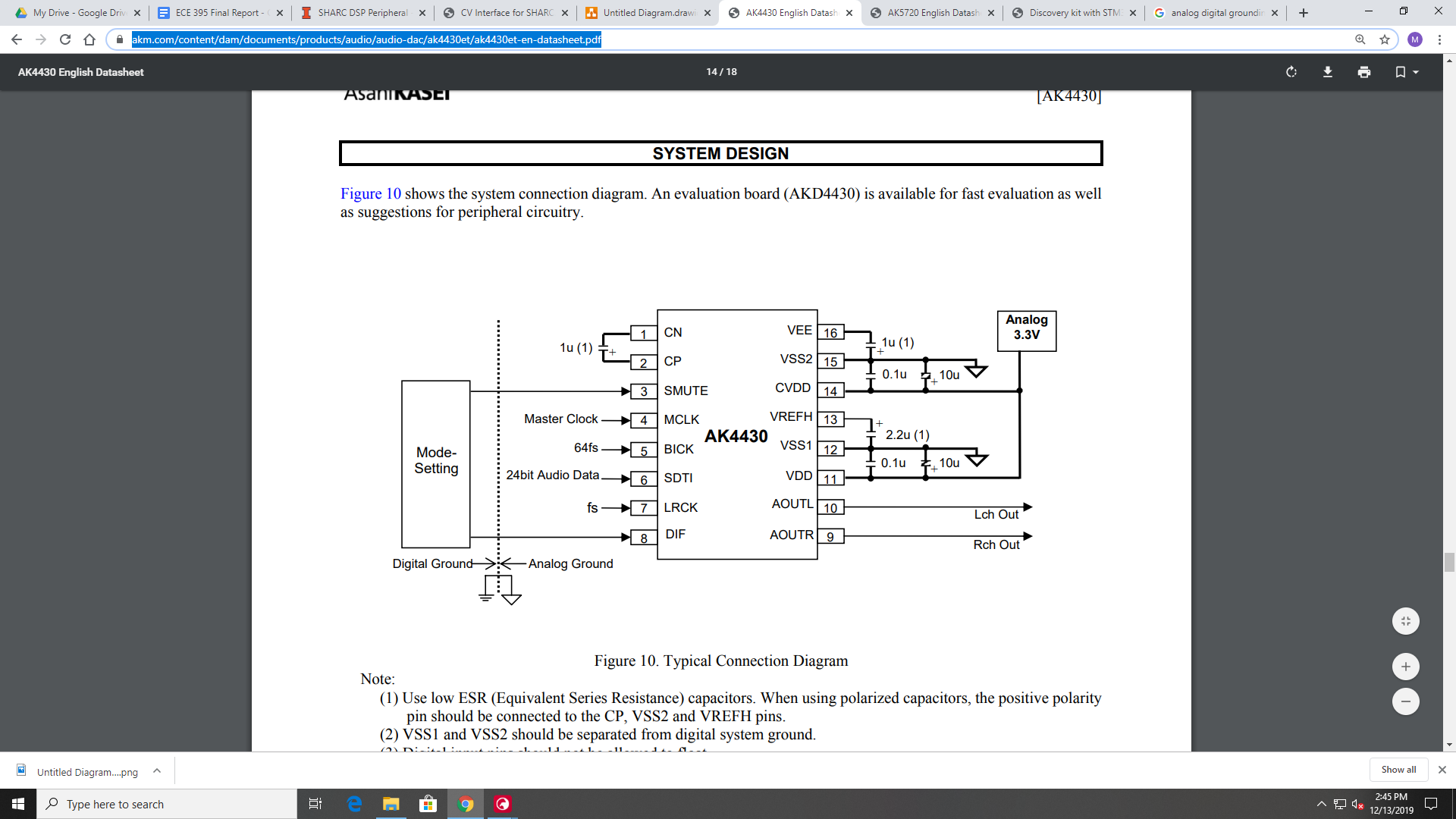
[Datasheet](https://www.akm.com/content/dam/documents/products/audio/audio-dac/ak4430et/ak4430et-en-datasheet.pdf)

The DAC we used is the AK4430. This DAC has the same specs as the AK5720 (24-bit resolution, 48 kHz sampling rate, and I2S capabilities). The DAC is configured similarly to the ADC, but there are far less config pins than the ADC, so it was easier to get the thing working. Serial data is sent to the SDTI pin, and the clocks on it are MCLK, BCLK(SCK), and LRCK(WS).

**Schematic:**

****Analog/Digital Grounding:

I’d like to briefly discuss analog and digital grounding, which is a technique used to decrease added noise on the audio signal when going through an audio processing unit. Here’s a really good [article](https://www.analog.com/en/analog-dialogue/articles/staying-well-grounded.html#) that discusses different analog/digital grounding techniques. The basic concept of digital/analog grounding is to separate the grounds of digital circuitry from the analog circuitry in your design. Digital circuitry in this case would mean anything involving the STM32 processor, while analog circuitry would be the ADC, DAC, and any other channel that is transmitting the audio signal. The two grounds are then connected in some way. In addition, the power lines for the analog/digital circuits must also be separate.



*Diagram of DAC with digital/analog grounding configuration*

# 

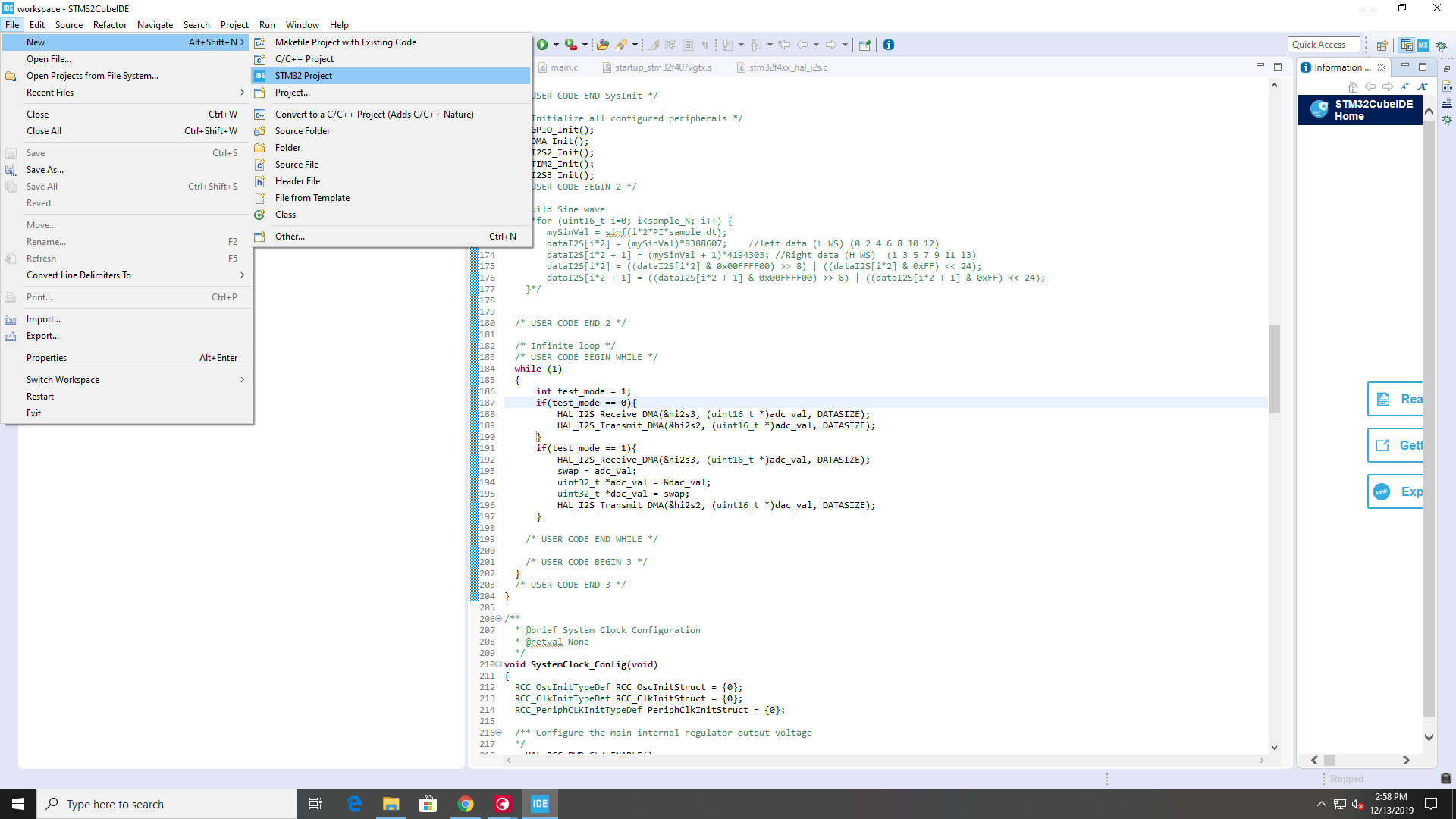
# **STM32F407G Processor:**

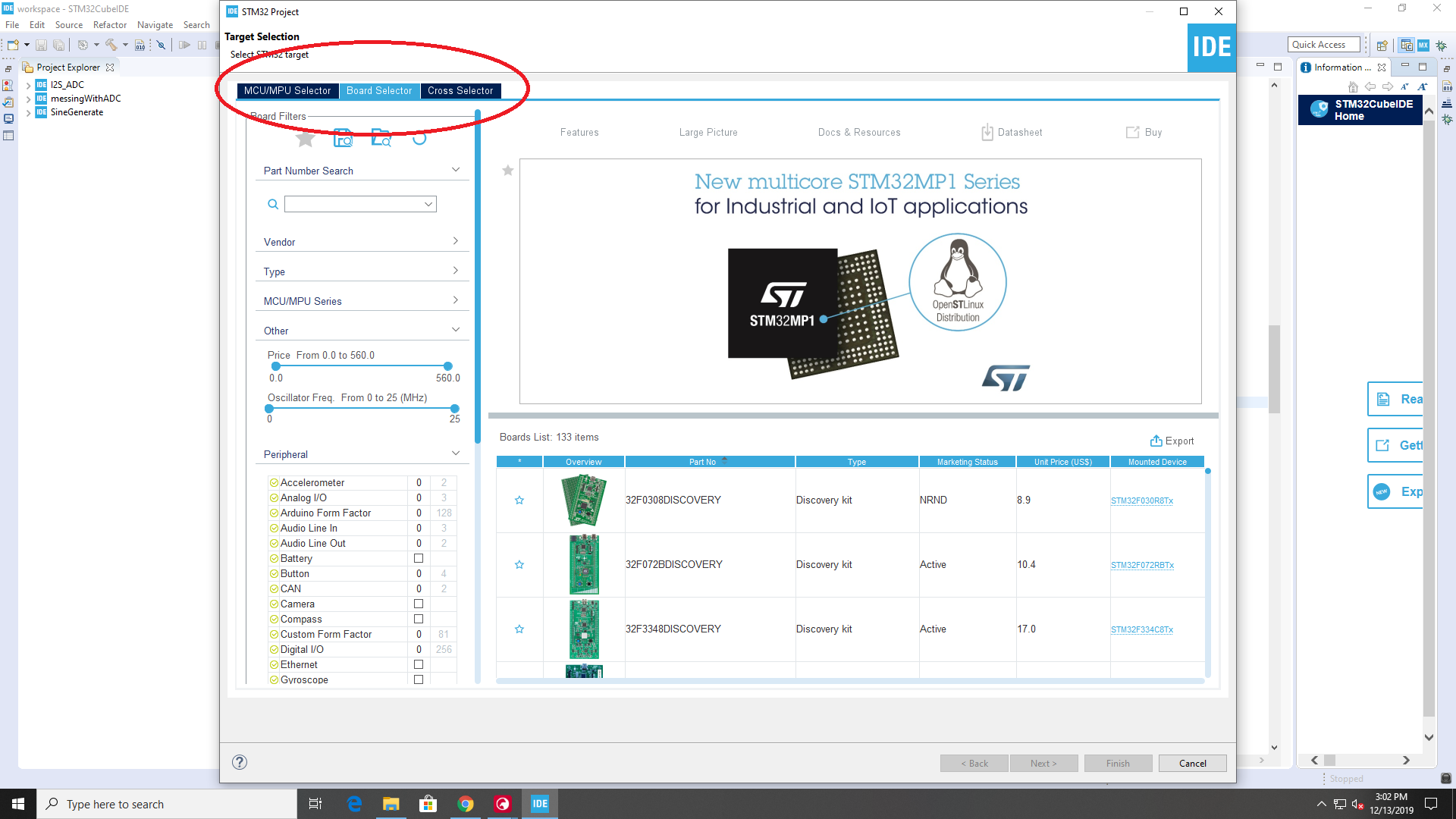
**STM32CubeMX/IDE:**

All coding and processor configuring was done using the [STM32Cube IDE](https://www.st.com/en/development-tools/stm32cubeide.html), an open source IDE that ST has made, which allows for easy configuration of their chips. There are 4 steps involved in programming any ST chip:

1. *Create a new project:*

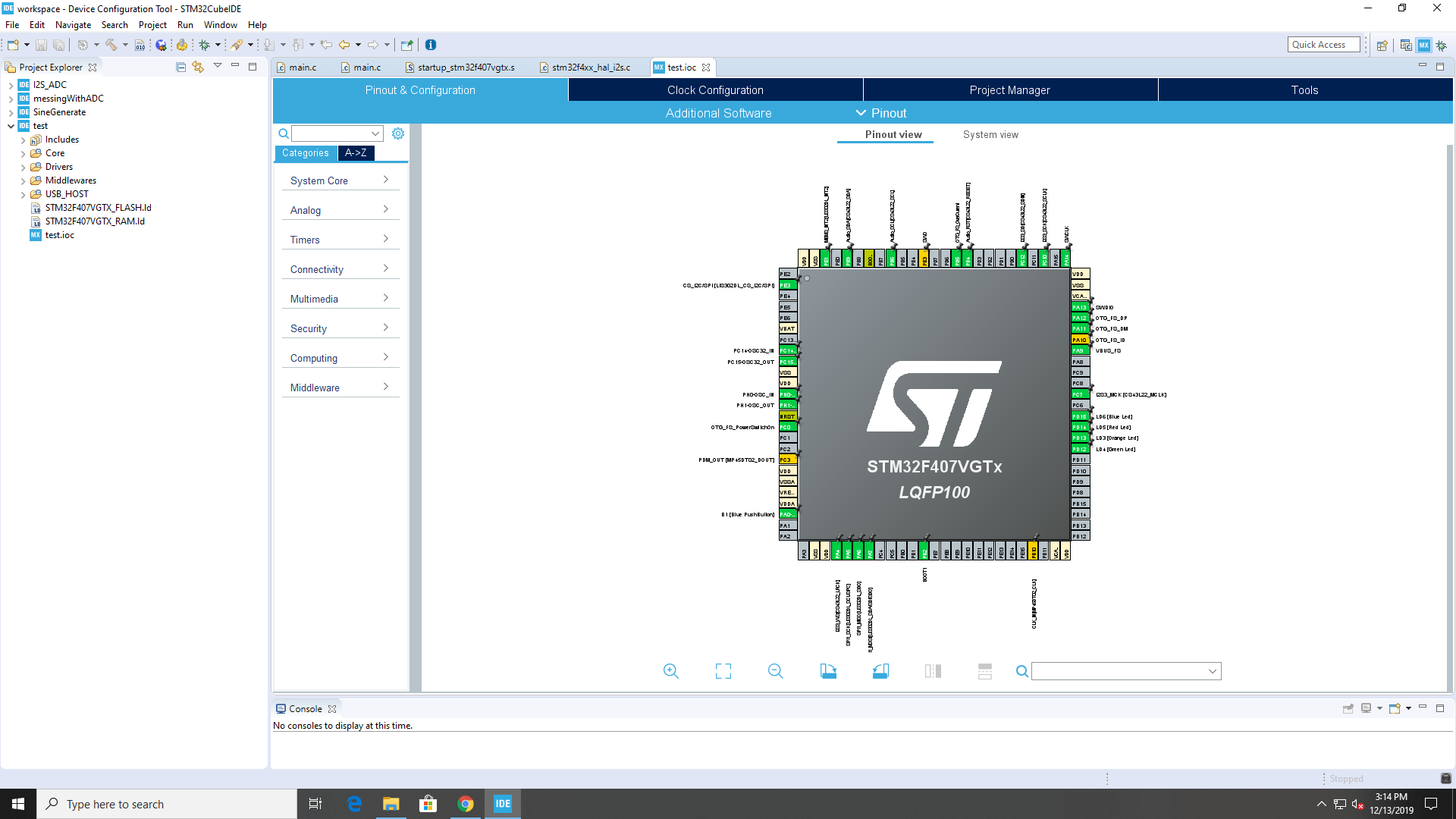
Create a new project by going to new, and selecting an STM32 project. Select the development board or processor that you are using by using the search bar. You can search by board or by processor by changing the bars in the top left corner. Once you have selected your board/processor, click next, name your project, and the IDE will create an IOC file for you to edit. The IOC file is where all the configuration stuff is done.

****

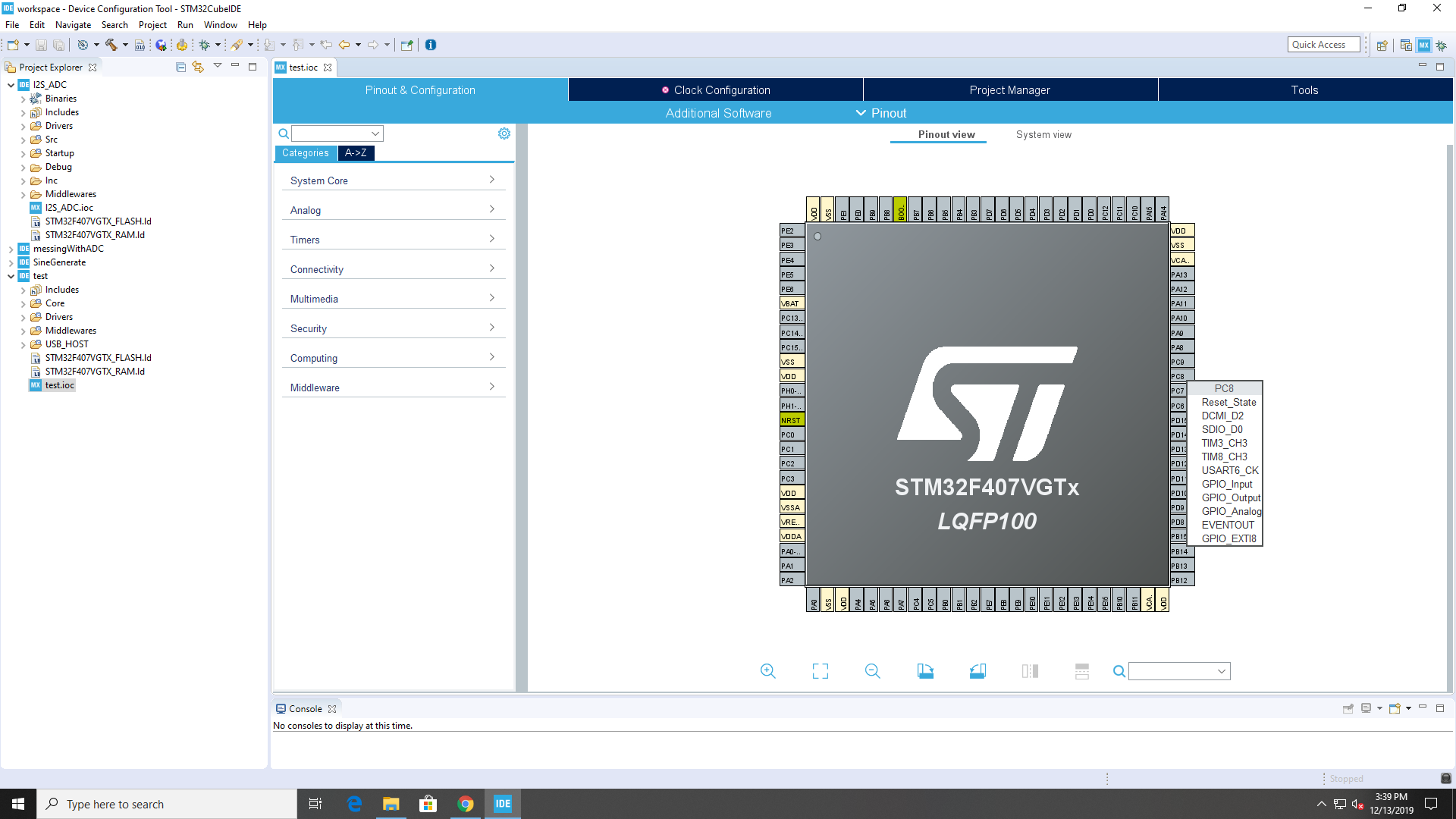
****

1. *Configuring the chip:*

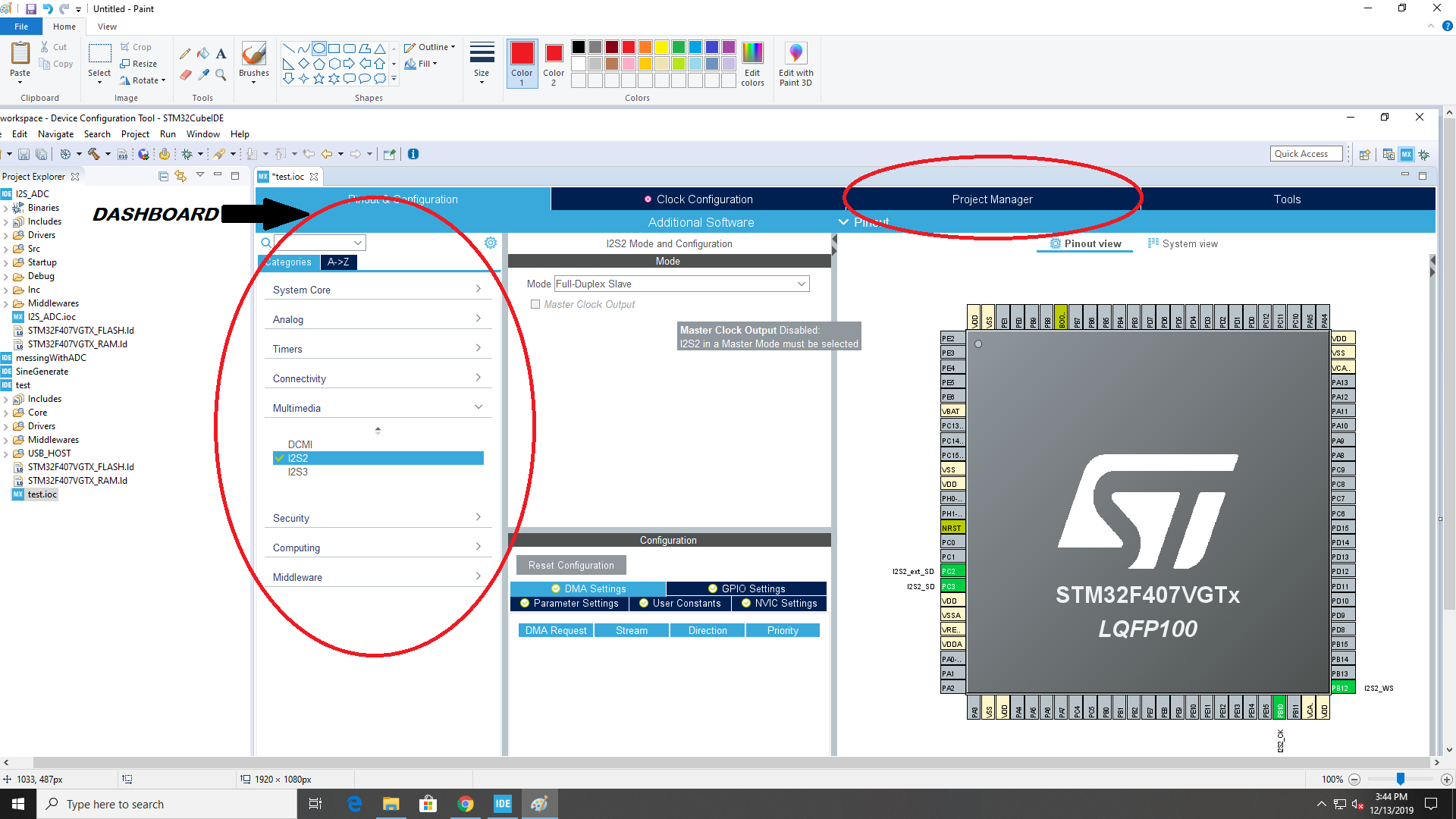
Configuring your chip involves setting up all the necessary pins required for your projects, setting up the clock configuration, and changing the settings for each of your selected pins. When you first open your project, the following screen should appear:

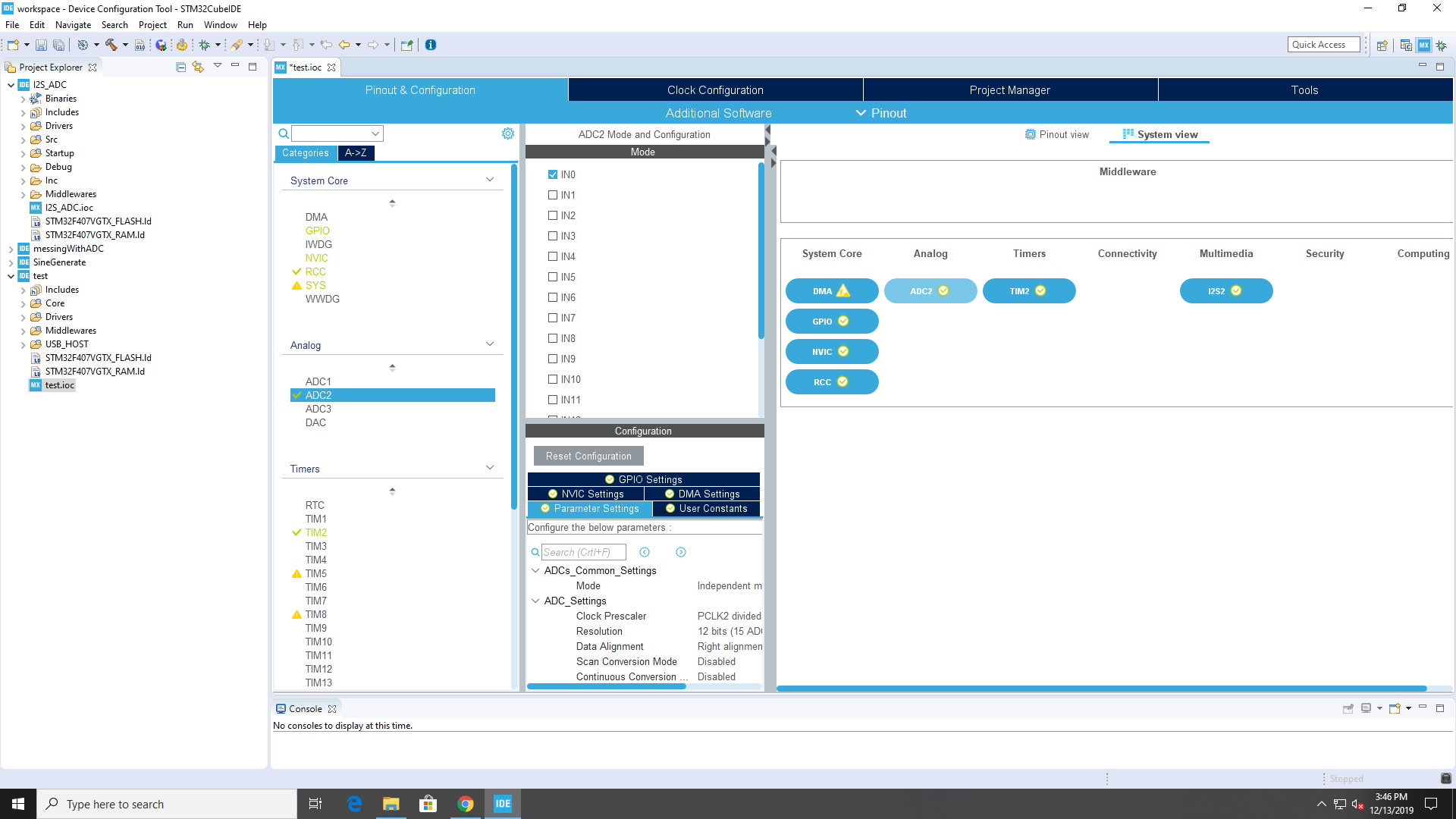


This is the default configuration for the development board. If you choose an individual MCU, then the chip should start with no configurations. Since many of the pins on the processor on the development are being used by the components on the development board, many of the pins are being used. For most projects, a lot of these pins won’t actually be used, so it’s best to start from scratch and build what you need. To clear the pinout, just select Pinouts->Clear Pinouts. To add pins, use the dashboard on the left, or left click on any of the pins on the chip image. If you left click on any of the pins, it will display all the possible uses for the pin. Most pins have several functionalities, and the pinouts should be optimized for your project.



*Selecting the configuration for an individual pin*

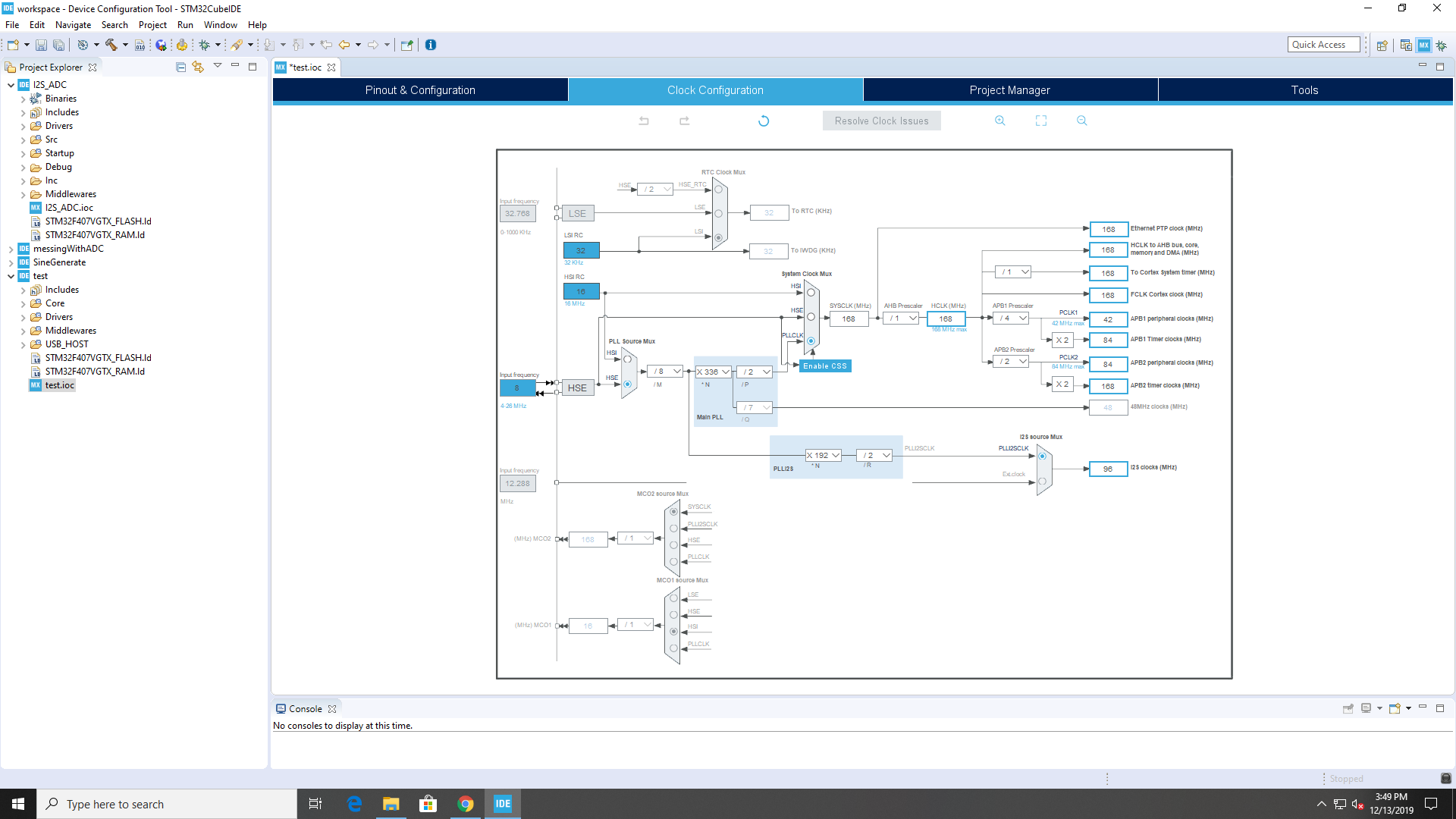




*System View of Pinouts*

The dashboard will automatically add any pins required for the items functionality, and it allows you to change the settings for any of the pinouts. You can also change the pinout settings by going to the system viewer.

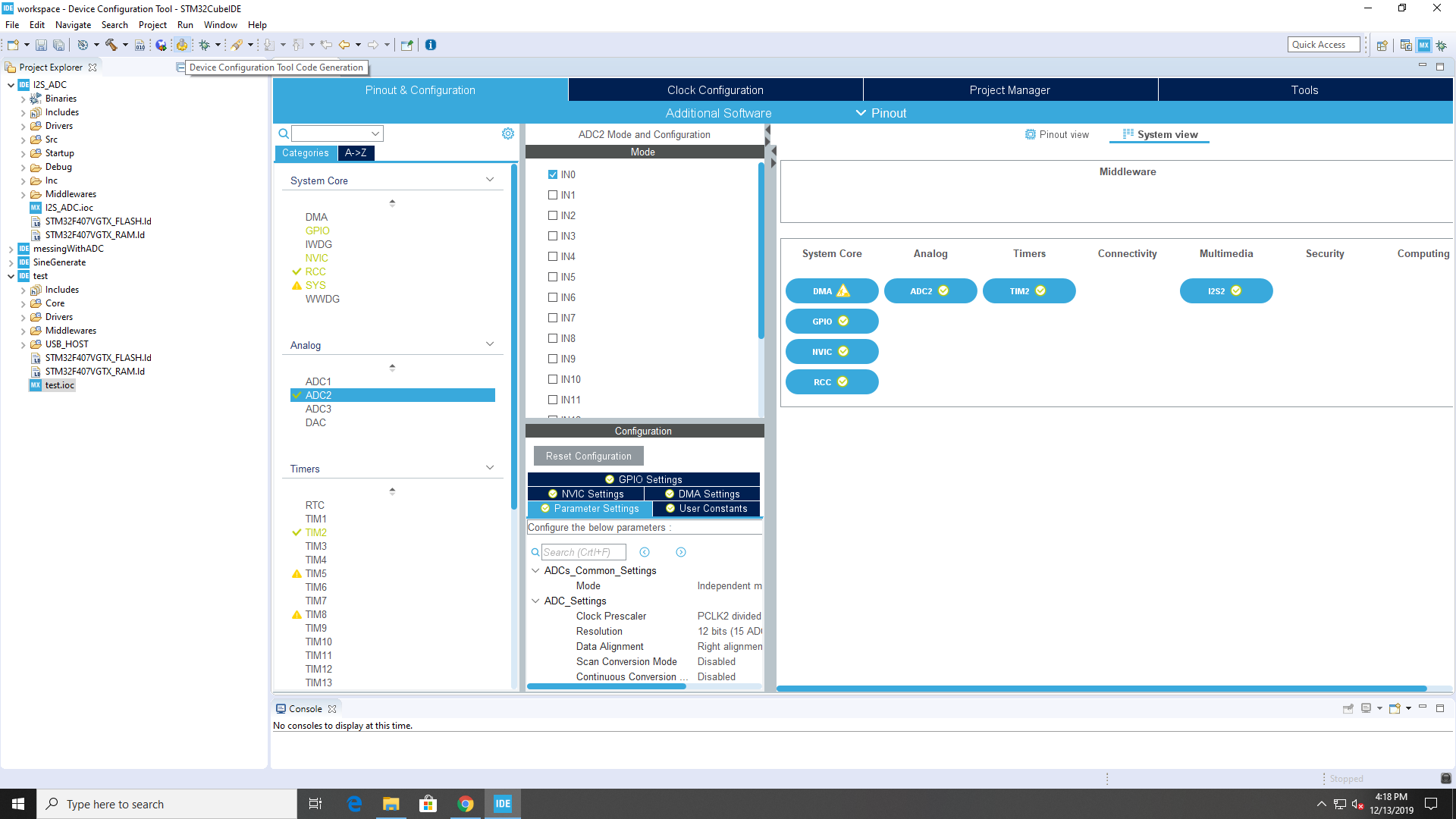
1. *Clock Configuration:*

**

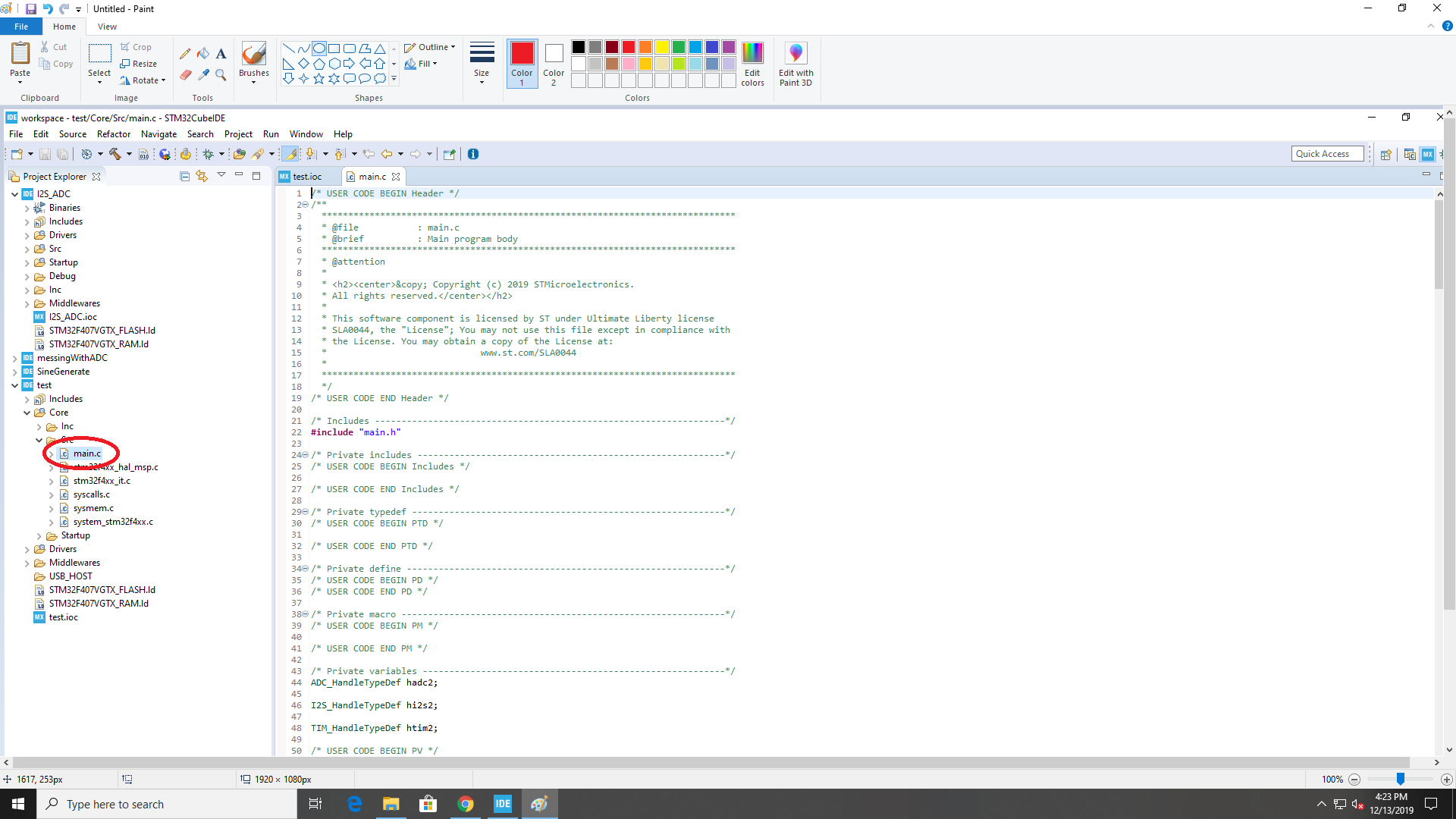
Your Clock Configuration should not change very much. The only thing to make sure about in this section is that the HCLK is set to its maximum (168 MHz). Also, if you are using the external clock, make sure that the PLL source MUX is set to the HSE input.

1. *Creating/Compiling Code:*

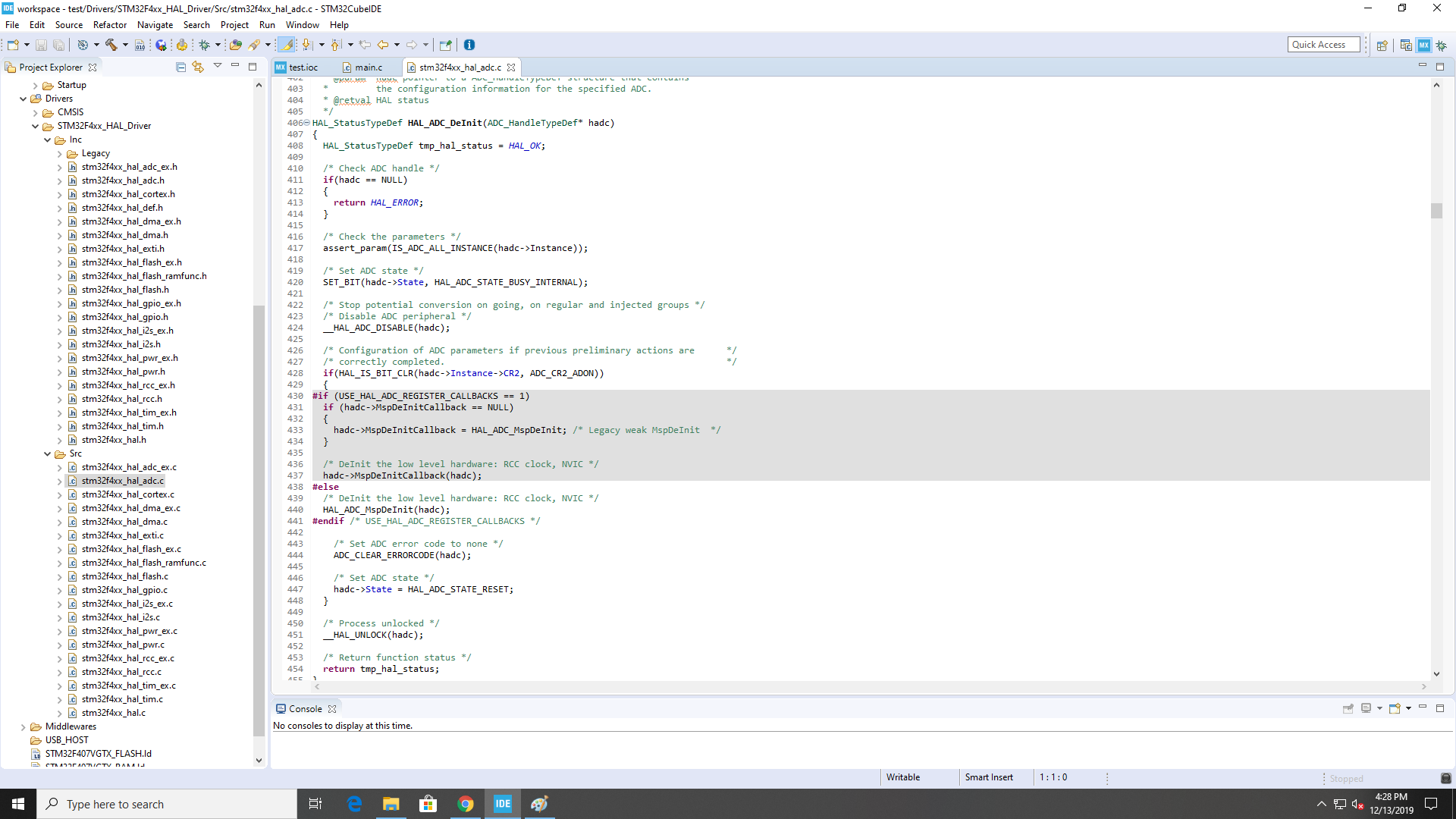
The next step is to generate code. To do this, select the gear icon on the toolbar:



This will compile code that sets up all of the configurations you have selected in the IOC file. Most, if not all of the coding that you will be doing will be done inside of the main.c file.

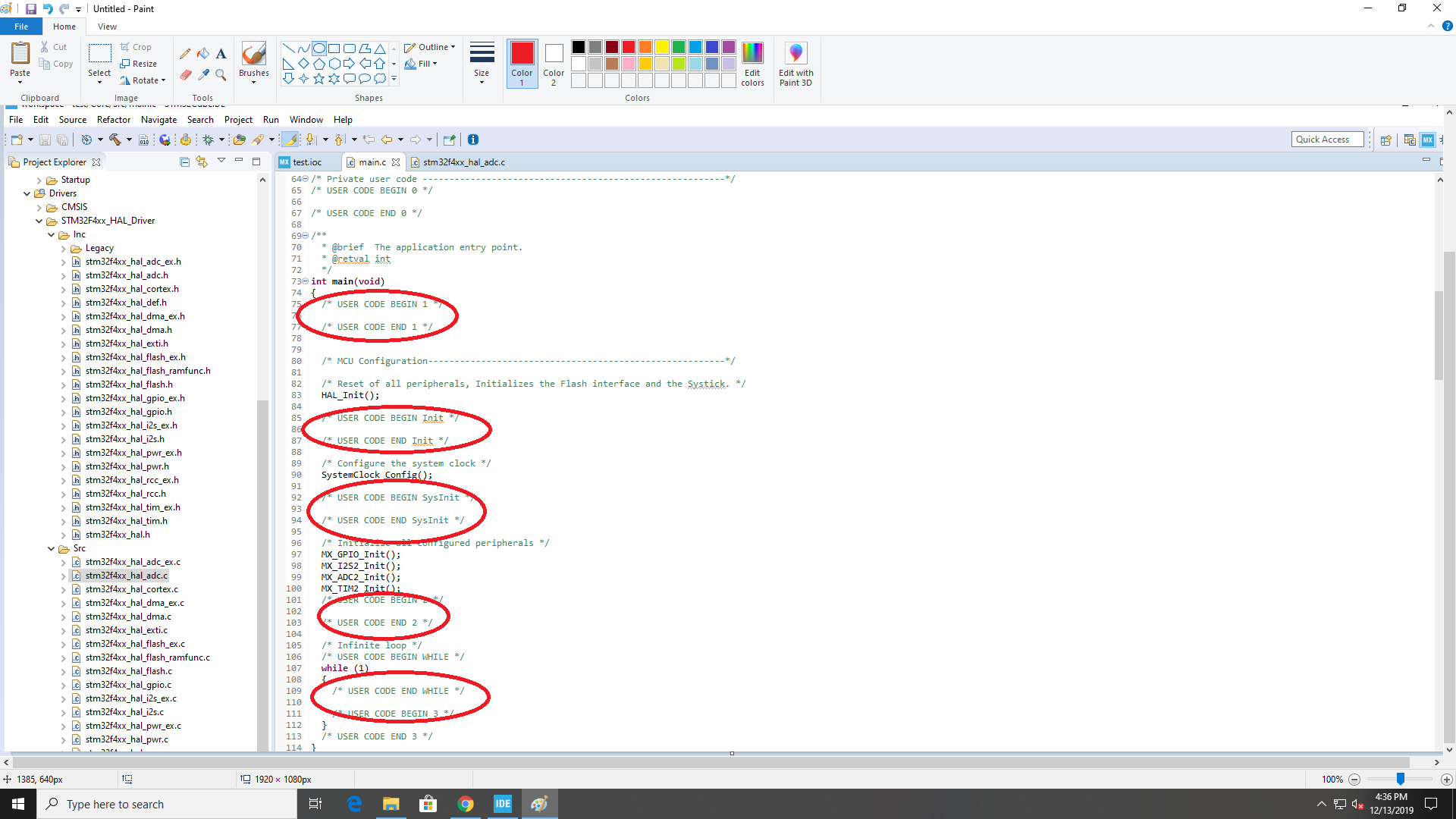


Though, keep in mind that there are many useful functions that are made by the compiler for all functions. All of these functions are located inside the corresponding .c files for the different devices you have initialized. For example, if you have created a pinout for one of the onboard ADCs, there are functions inside the stm32f4xx\_hal\_adc.c file that will initialize the ADC, receive data from it, etc.



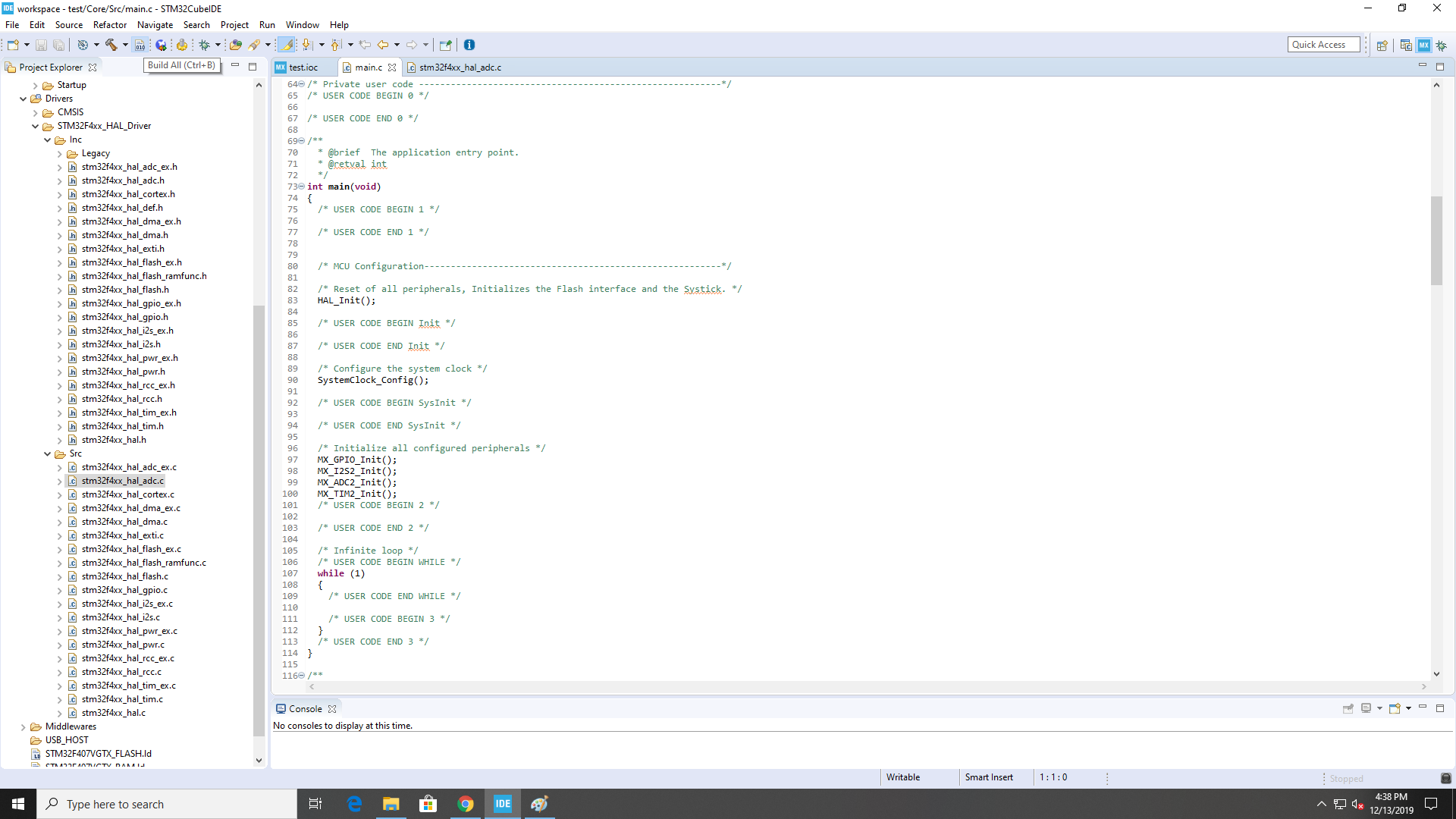
For more information about what individual functions do, you can look at the [provided manual](https://www.st.com/content/ccc/resource/technical/document/user_manual/2f/71/ba/b8/75/54/47/cf/DM00105879.pdf/files/DM00105879.pdf/jcr:content/translations/en.DM00105879.pdf). This manual provides a brief description of each function, along with a description of the function’s inputs/outputs.

Inside of your main.c file, there are several areas where you can add your own code. ***Make sure that any of the code you write falls within these regions***, otherwise when you build it, it will delete it.

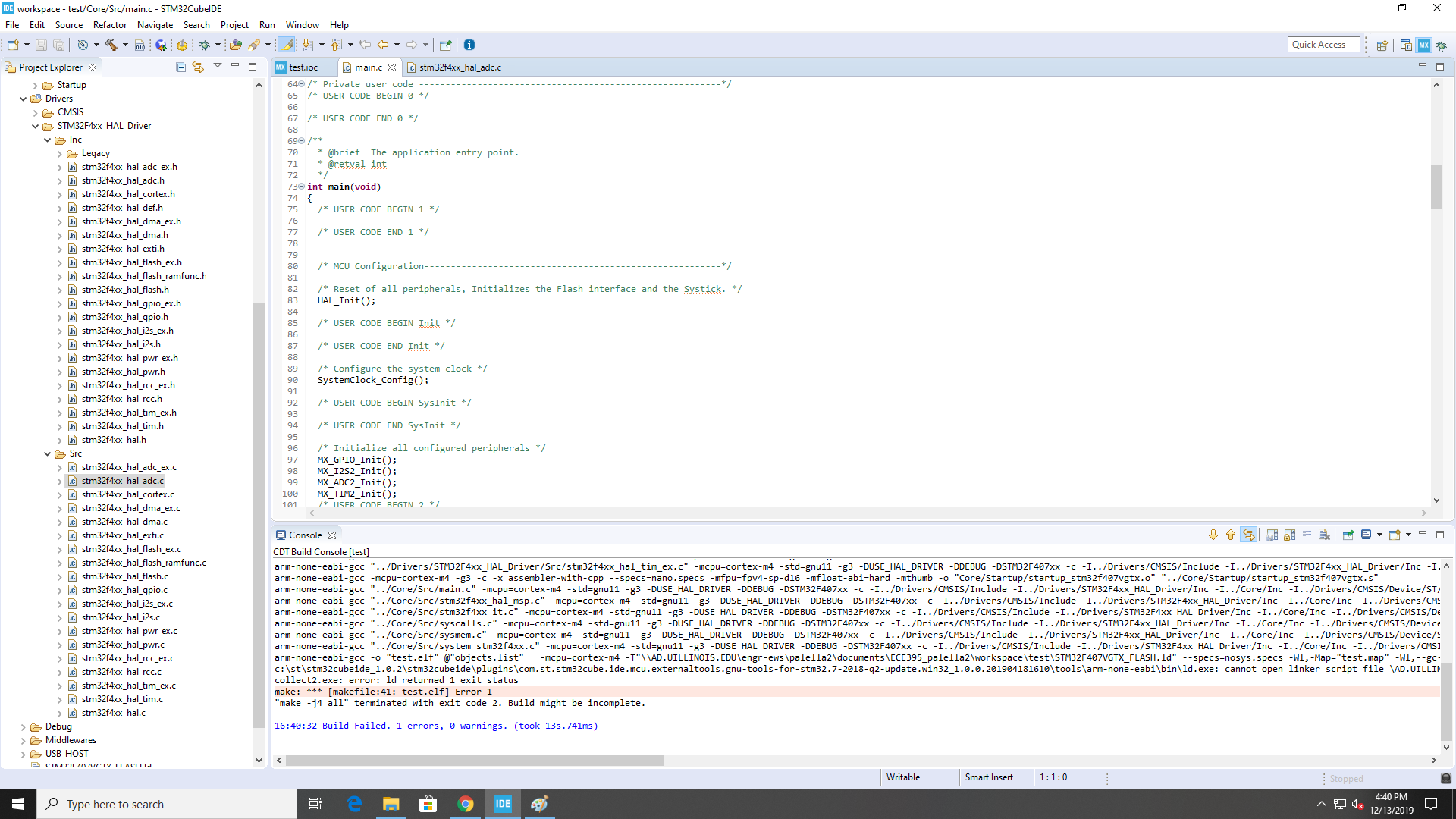


***Make sure your code lies between the commented text!!***

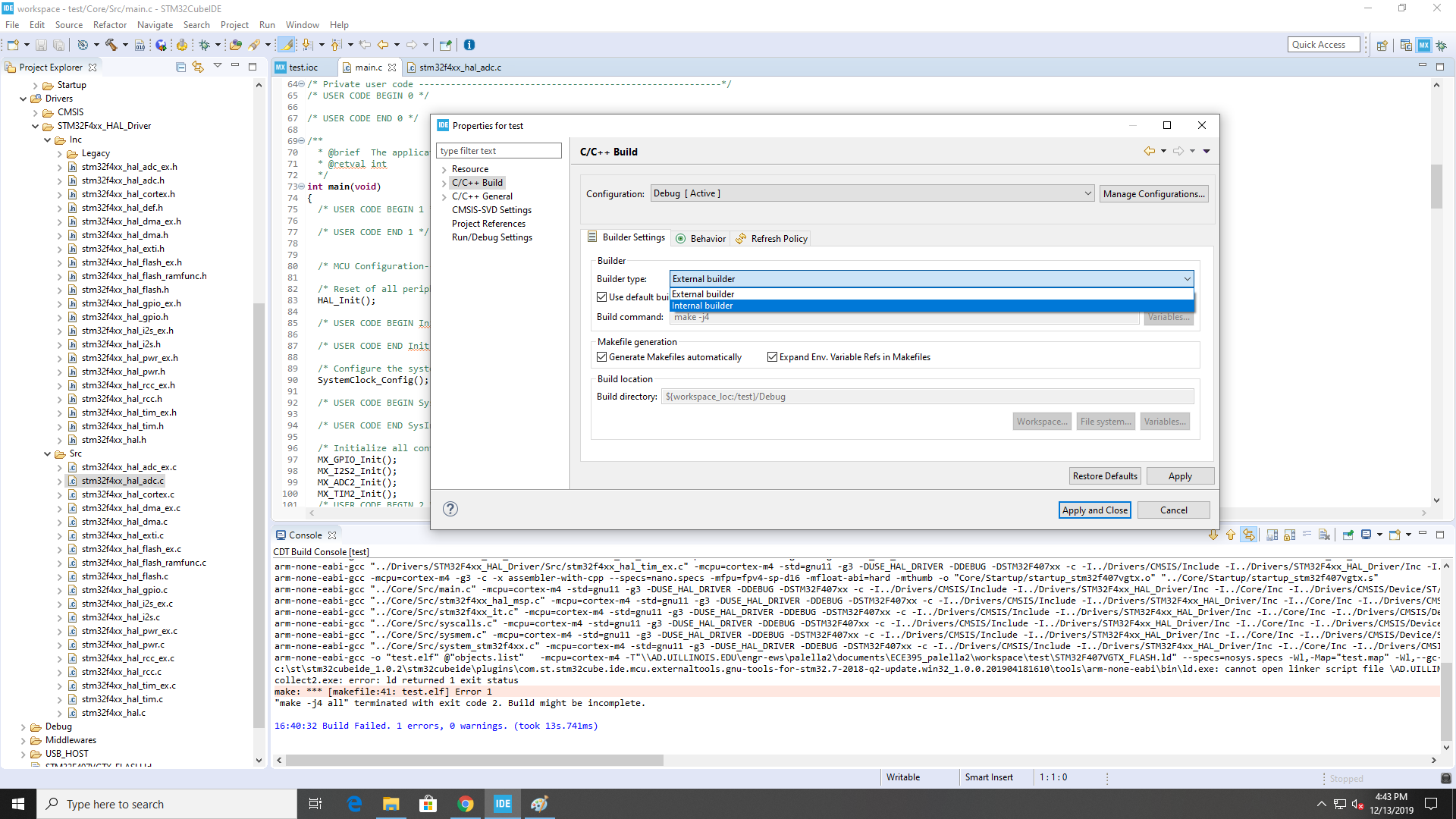
Once you have written your code, you need to build your project by pressing Ctrl-B, or by selecting the build icon in the toolbar:



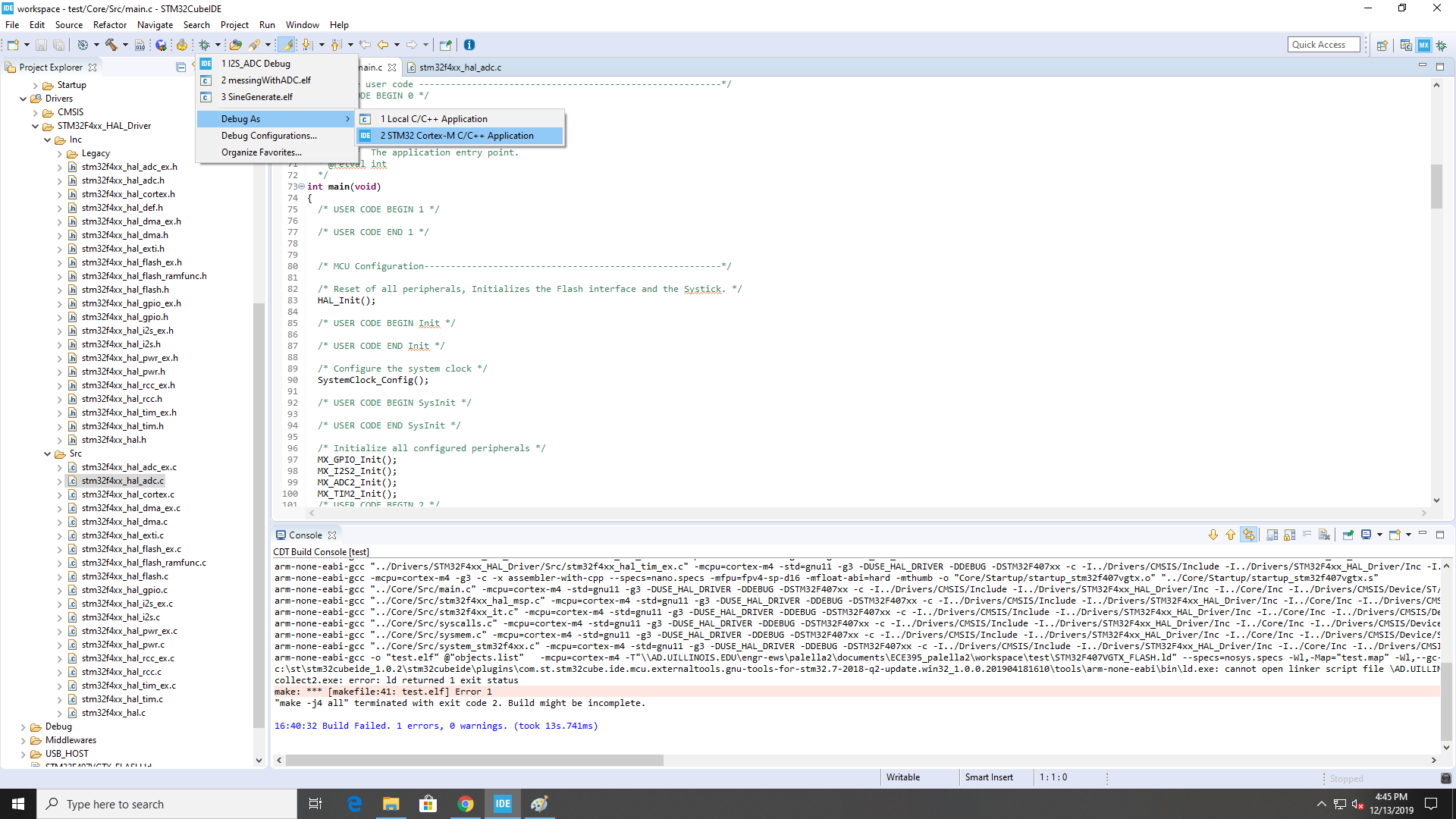
Next, correct any warnings or errors the compiler gives you, and build again. Note: this error is very common when first building the project:



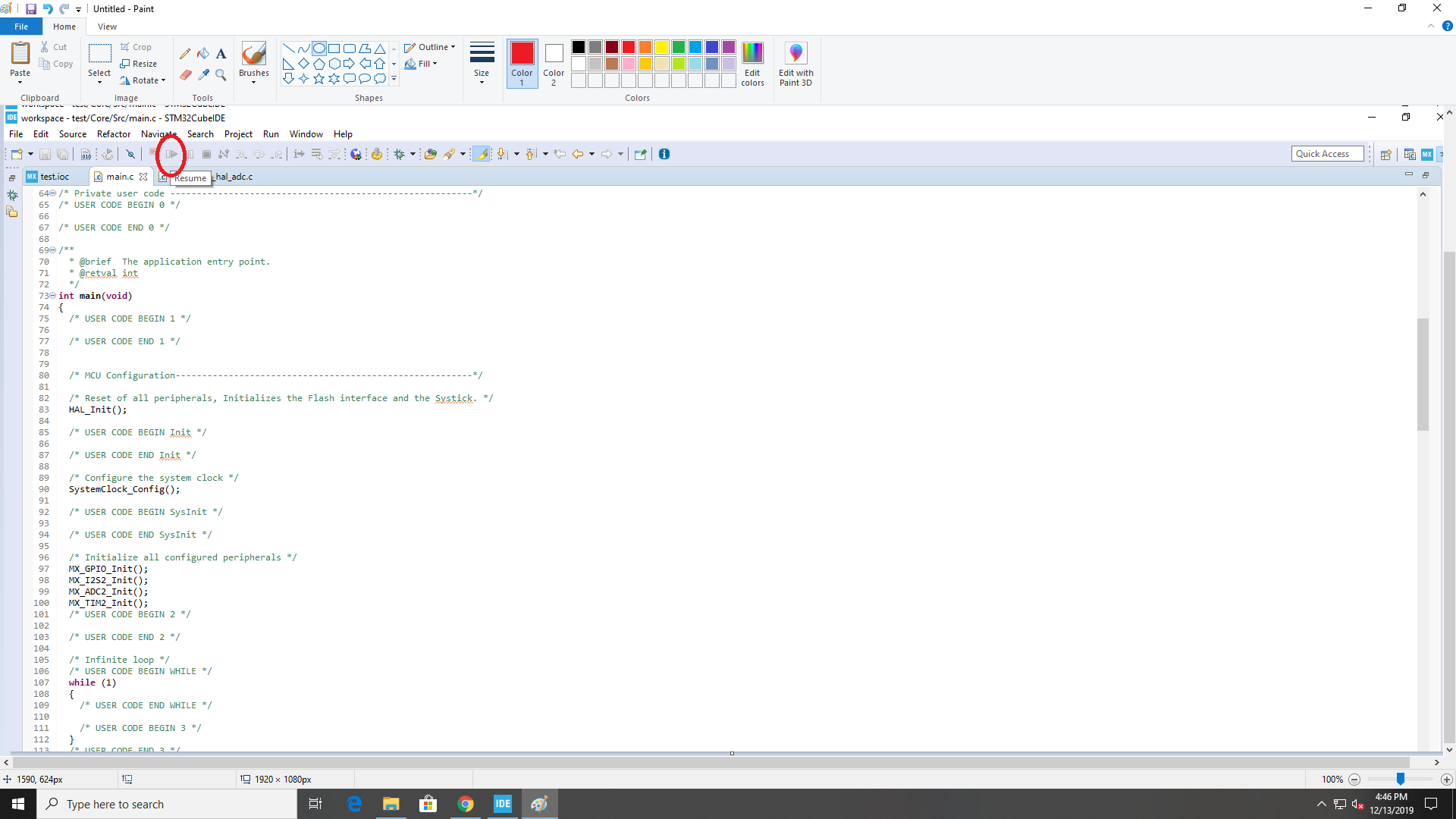
This is caused because for some reason, the IDE automatically tries to generate assuming there is an external .elf being used. To fix this go to Project->properties. Navigate to C/C++ Build and set the builder type to internal builder then apply and close:



After all errors have been fixed, run the program by selecting the debugger icon on the toolbar:

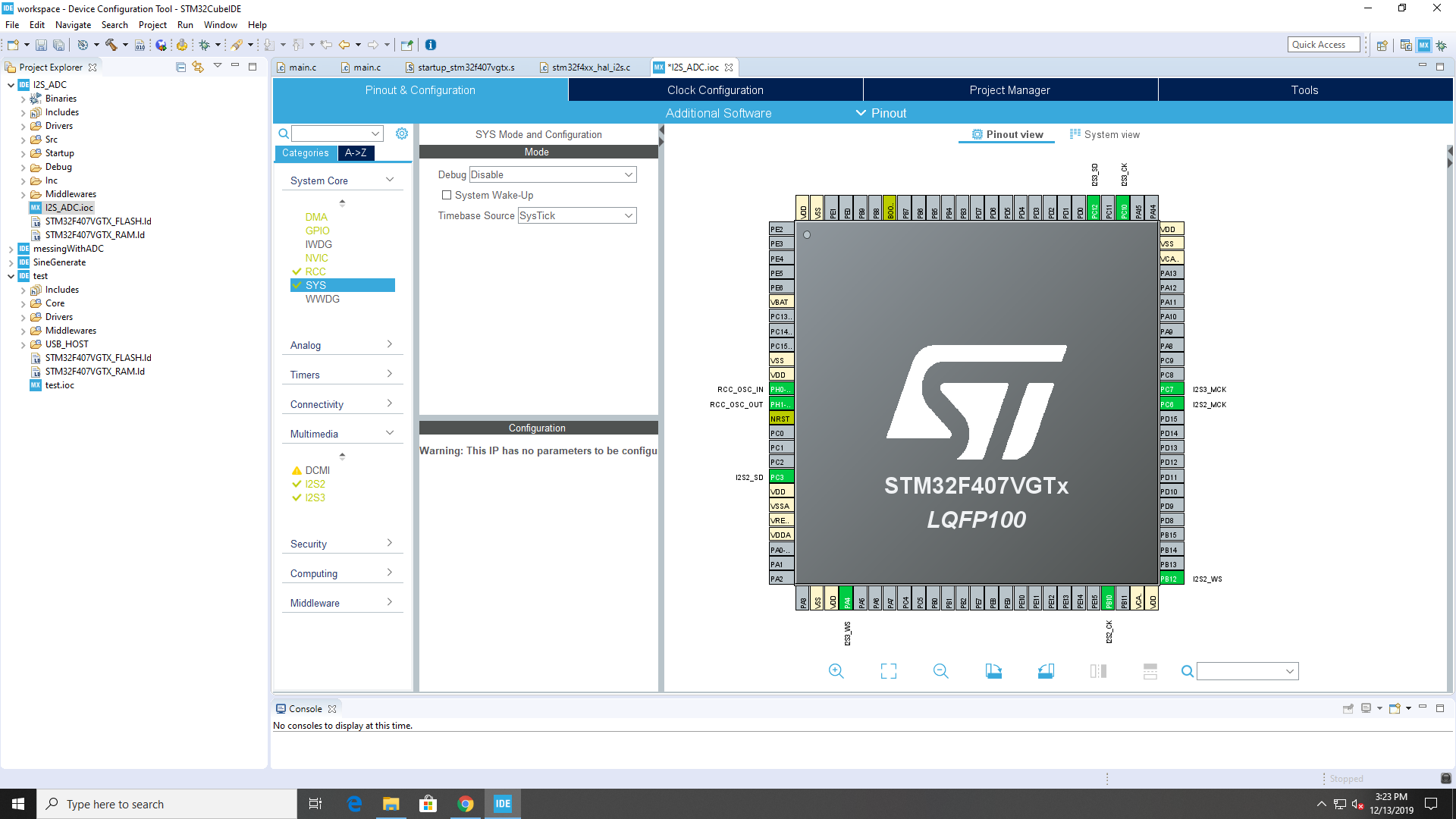


Run your program as an STM32 project, this will send you to the debugger perspective, where you can run the code by pressing the play button on the toolbar:



# 

# **Processor Configuration:**

For our project, we only required the two I2S pinouts on the STM32 to transmit and receive data from the ADC and DAC. We also used the RCC pinout, which is needed to use the external clock.

*I2S Setup:*

The STM32F4-Discovery Proto board that we were using came packed with on board HAL libraries which made using I2S slightly more convenient. I2S has many ways to interact with the data flow. For our project we chose HAL\_I2S\_Transmit\_DMA and HAL\_I2S\_Recive\_DMA which will use direct memory accesses to log/send the data. Both functions required a buffer, and for our design we chose that they each have a separate buffer. Additionally I2S on the board is by default the master transmitter/receiver so we had to ensure our ADC/DAC were put in slave modes. As mentioned in the RCC set up our choice of clocking allowed us to go up to 94 KHz however audio only need 44 KHz to be clean. To compromise on power consumption and robustness we chose the 48 KHz sampling speed, which gave us some leeway to make sure our sound was clean and didn’t draw a ton of power.

*I2S Implementation:*

To access our data we inserted our audio processing function into the HAL\_I2S\_RxHalfCpltCallback interrupt, which triggers when the buffer to transmit is half full. To access the data correctly we need to go from little endian to big endian, so we take the bottom 4 Bytes and left shift them by 8 and the top 2 Bytes and right shift them by 24. This turns our little endian 32 bit unsigned int into and big endian 32 bit unsigned int. This process is done for each member of the buffer, and the reverse is done so that we can properly transmit the data to the DAC.

*I2S Caution:*

While the STM32F4-Discovery has lot of built in libraries, if you intend to use off board ADC/DAC’s HAL\_I2S\_Transmit\_X and HAL\_I2S\_Recive\_X (X can be either DMA or IT for Interrupt based collection) are the only functions you need. If you intend to communicate to the onboard microphone or 3.5 mm headphone jack/ Cyrus audio DAC you need to target that device with I2C communications which complicates the process. We spent a large amount of time trying to get on board systems to work under the guise that an understanding of them would help us with off board systems, but it had nearly no real correlation.

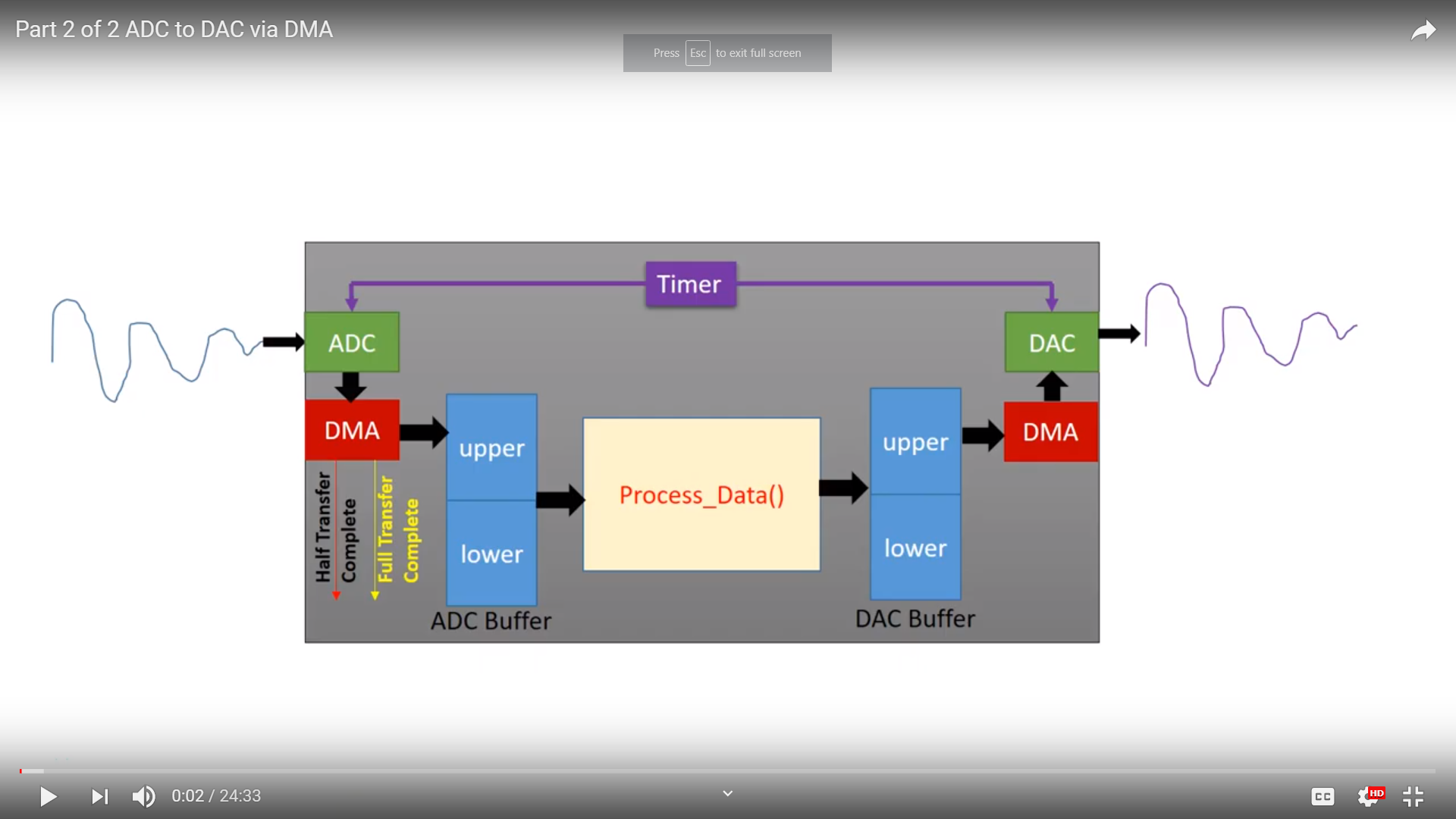
*RCC Setup:*

The RCC is the system clocking for an ARM chip, usually ARM system clocks are slow to reduce power draw, however our project required a higher power clock so we configured our RCC to use the Crystal/Ceramic resonator for the High speed clock and we disabled our low speed clocks. This enables out I2S to sample data upto 94 KHz.

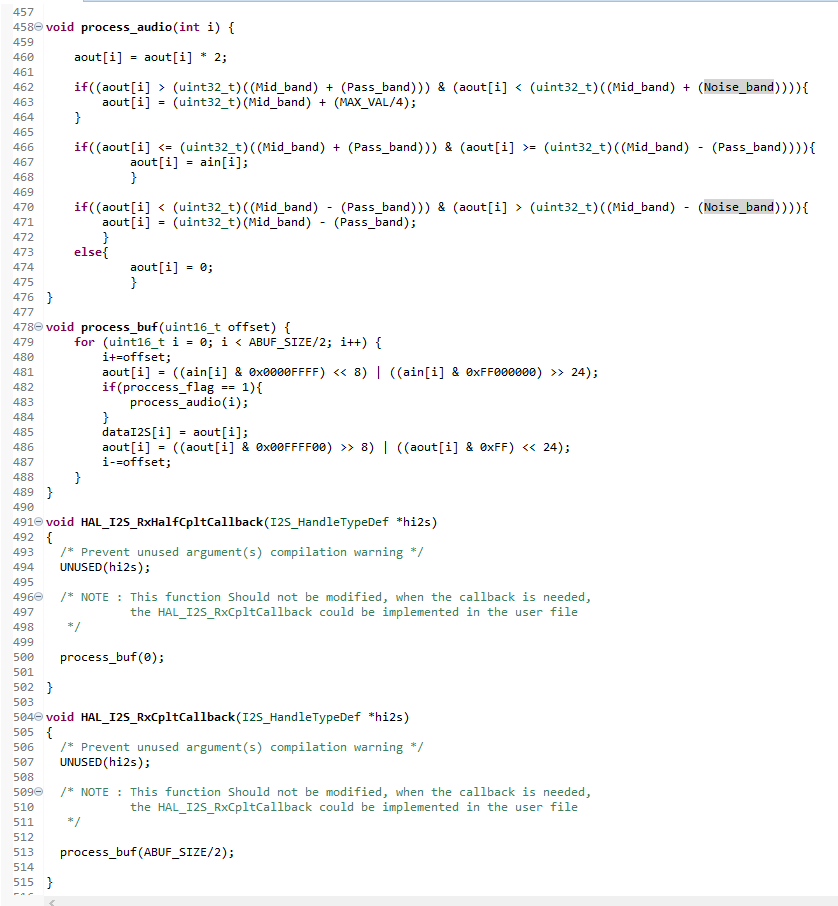
# Software:

*Overall Layout:*

The overall software design of our processor can be visually represented in the block diagram below. The ADC transmits data into a DMA buffer. This Buffer is fed into another buffer which is used to account for the time it will take to process data. A pointer is used to grab data from this buffer and process the data in an external function. The data is then outputted to a DAC buffer, sent to a DAC DMA, and finally sent to the DAC itself. The section of the code that we needed to develop was the data processing function and the two buffers. Everything else is already handled by the compiler. You can send data to and from the ADC/DAC via the Hal\_I2S\_Recieve\_DMA() and the Hal\_I2S\_Transmit\_DMA() functions. The preconfigured DMA functions come with some useful interrupt subroutines that are utilized in our code. The subroutines are used to change the place at which data is being sent to in the ADC and DAC buffers every time the DMA buffer is half and completely full.

**

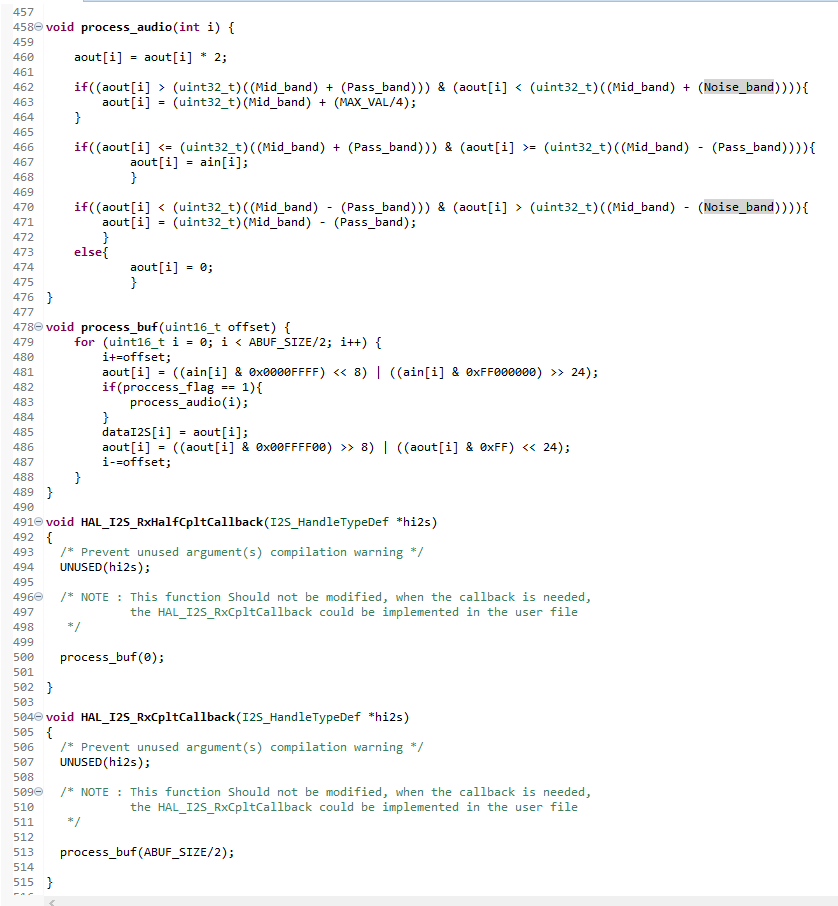
*Source:* [*https://www.youtube.com/watch?v=ThFfI-JSv2Y*](https://www.youtube.com/watch?v=ThFfI-JSv2Y)



*Interrupt Functions being used to change buffer pointers and process data*

*Process\_buf() Function:*

This function is used to adjust the data values inside of the ADC buffer so that they can be processed and also to actually process the data. The function first moves the data values from the ADC buffer to the DAC buffer. During this process we need to also shift the two least significant bytes over by a byte and then move the most significant byte down to the be the least significant byte. The reason for this is that the Transmit/Receive functions shift the data being taken/sent from the ADC/DAC around for whatever reason. This needs to be done before you process audio or else you will be not be processing the correct value.



*Distortion Effect:*

To show off the capabilities of our device, a distortion effect was created inside of the Process() function. Firstly, we had a flag we used to enable/disable the effect, which helped us in testing, and allowed us to keep the process isolated from the rest of the system. After this, 3 bands were created: the pass band, distortion band, and noise band. We created these bands by first getting the max value of a uint32\_t value (the data type of our array) which is 16777216 and divided it by 2, which gave us the middle value of a uint32\_t. All bands were made relative to these two values. Our pass bands were Values <= Middle Value +/- Max value\*8/32, distortion bands were values > Middle Value +/- Max value\*8/32 AND Values <= Middle Value +/- Max value\*31/64, and finally our noise band was any Value > Middle Value +/- Max value\*31/64. The pass band does as the name implies and just sends data through unchanged, the distortion band will bring any of its values up to or down to Middle Value +/- Max value\*8/32, and anything in the noise band is brought to 0. The noise band was a surprising component because before we made the effect there was almost no noise in the circuit, and what noise there was you could only hear at very low volumes/amplitudes such as listening with headphones. It seems that the distortion effect without a noise band would drag up low amplitude noise and make a fax machine sound, the noise band had to be mirrored on both sides due to the fact small negative values are represented as very large uint32\_t values. One of the main drawbacks in doing the effect this way is that it is amplitude based so if one plays too loud the entire sound of the guitar is distorted or nullified but for less than 3 strings at any one time the effect works fine.

