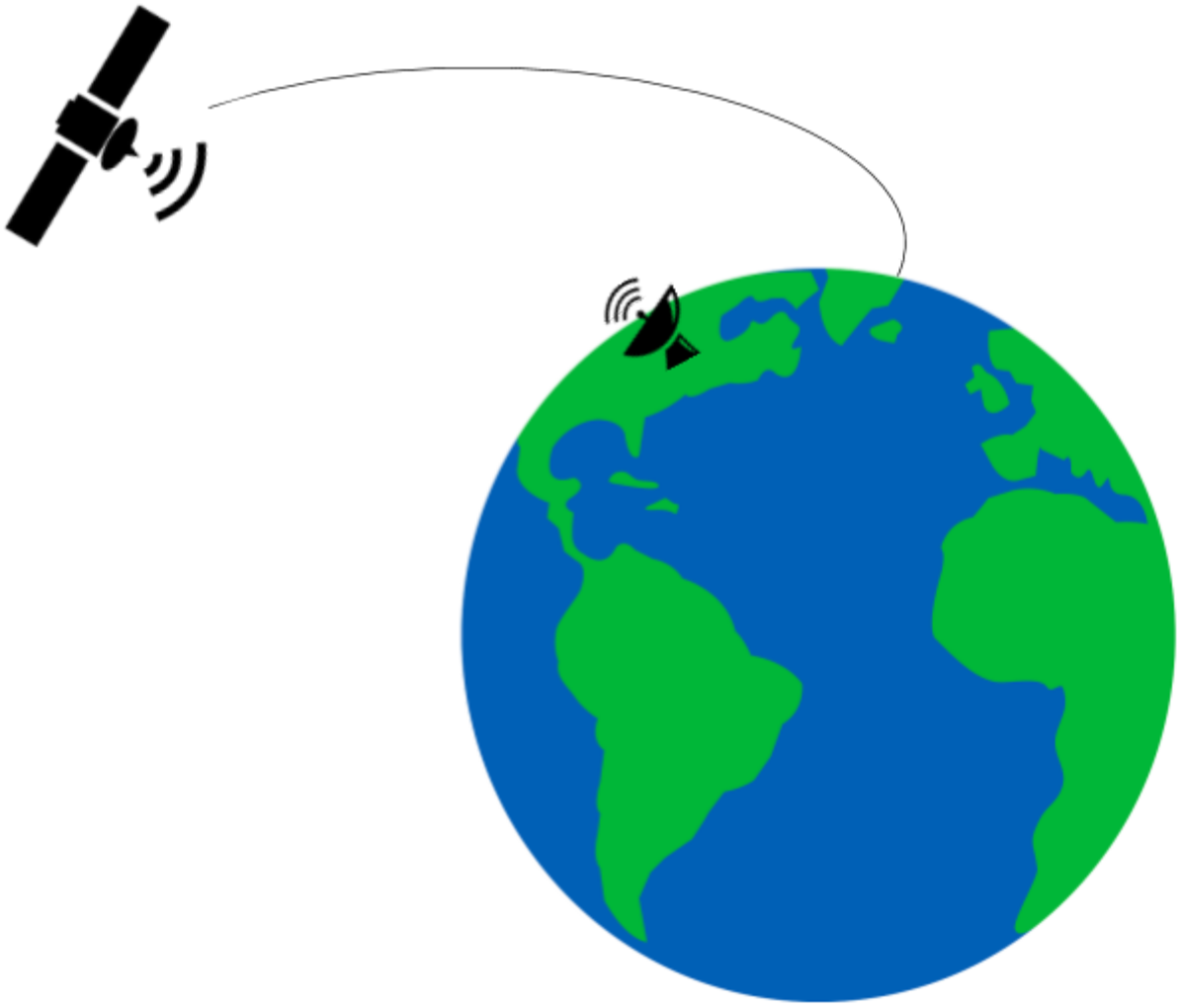
ECSE 4970 Final Project

Automatic Picture Transmission (APT) Decoder



Patrick Love
Nathan Mascari

Background and Motivation

Our project focuses on demodulating, decoding, and displaying an Automatic Picture Transmission (APT) signal. APT is an analog image transmission system developed for use on NOAA weather satellites. More specifically the satellite broadcasts an FM radio frequency signal which encodes a 2.4 kHz audio frequency subcarrier. This subcarrier is amplitude modulated with the final pixel data. Our system decodes everything from the RF demodulation onward.

The motivation for the project was eventually being able to do a full image downlink from a NOAA satellite ourselves. Nathan has acquired a HackRF Software Defined Radio module which can be configured to demodulate the RF carrier and feed the subcarrier audio to the STM32 decoder.

The actual data modulated on the subcarrier is composed of two image channels (A and B), telemetry information, and synchronization data as shown in Figure 1. This data is chopped up into lines, with each line having a length of 2080 pixels. Each image channel use 909 pixels, leaving 262 for telemetry and synchronization. Two lines are transmitted every second, equating to 4160 baud. For reference, an actual decoded image annotated with the various data sections is shown in Figure 2 as well.

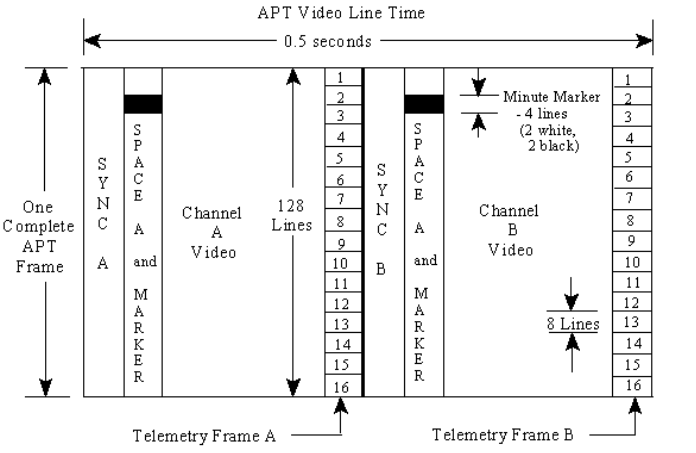


Figure 1: APT Frame Format

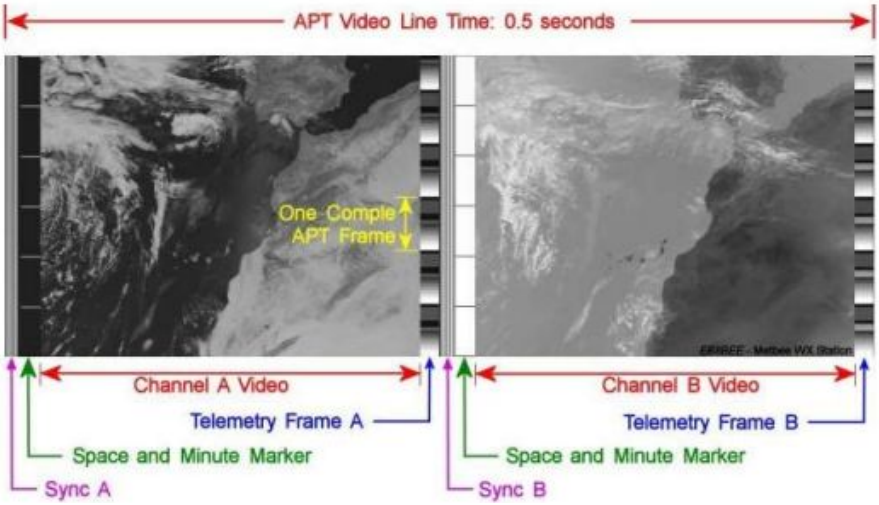


Figure 2: APT Format Example

Design Process

Analog Input Stage

One of the first requirements on the system was the need to digitize the incoming baseband audio signal in preparation for the signal processing stage. At first, we considered utilizing the line-in connector built onto the STM32 Discovery board itself. This connector does not directly feed into the processor though, and an external audio codec (WM8994ECS/R) chip serves as the analog front end. One benefit of using this method would be the higher resolution of the codec ADCs (24 bits), however the external chip limits us to a particular set of supported sampling rates. To avoid the need for rather complex non-integer resampling, it is highly desirable for the sampling rate to be a multiple of the APT symbol baud rate (4160). Unfortunately this is not the case for the rates supported by the WM8994ECS/R, and was ultimately the reason this approach was not chosen. The onboard ADC allowed us to select the sampling frequency arbitrarily, and the 12 bit resolution still gives us 8 levels for each of the 256 possible APT symbol amplitudes.

Use of the onboard ADC of the STM32 did come with one fairly substantial caveat, namely that the audio signal must be superimposed with a DC offset in order to lie within the ADCs 0-3.3V conversion range. As a result, it was necessary to design a custom analog front-end that performed the level shift. It also served to average the left

and right audio channels from the ⅛” audio jack used to connect to the audio source. The final circuit schematic is shown in Figure 3.

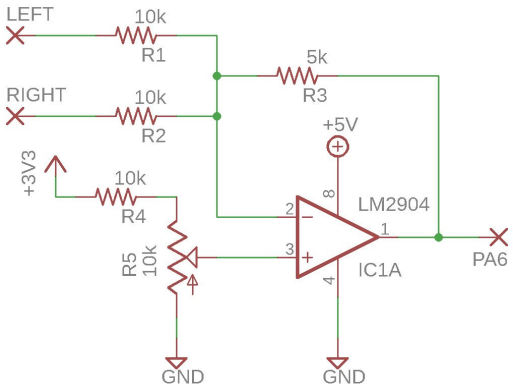


Figure 3: Aux Input Level Shifting Circuit

The level shift is achieved using a fairly standard sum/difference amplifier with an adjustable DC term which can be trimmed to achieve the precise level shift required. Were this not adjustable, achieving a precise offset would require much higher resistor tolerances than were readily available. The STM32 5V supply was used to power the op-amp in order to ensure the saturation voltage was above 3.3V and the full operating range of the ADC could be used. The signal level from a laptop source at max volume was measured and found to be close to correct, so no additional gain was added to the audio signal path. The amplifier does invert the audio signal, but that does not affect the amplitude modulation. The board is designed to slot as an expansion board into the arduino headers on the discovery board, and uses the A0 analog pin which is internally connected to pin PA6 of the STM32. Figure 4 shows the adapter board connected into the STM32 Discovery board.

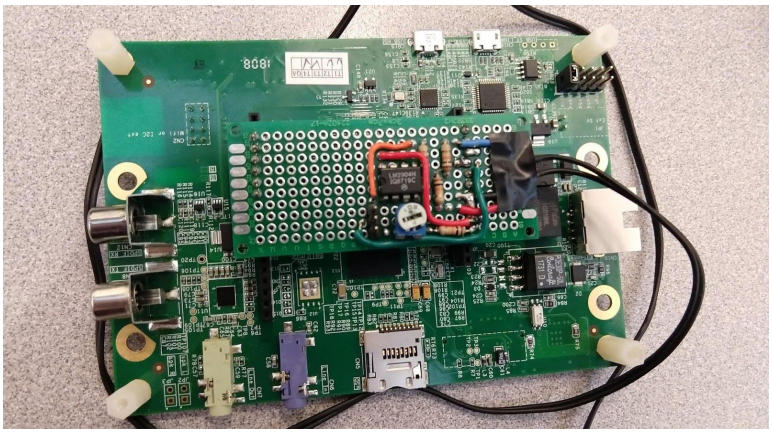


Figure 4: Adapter Board attached to the STM32

Signal Processing

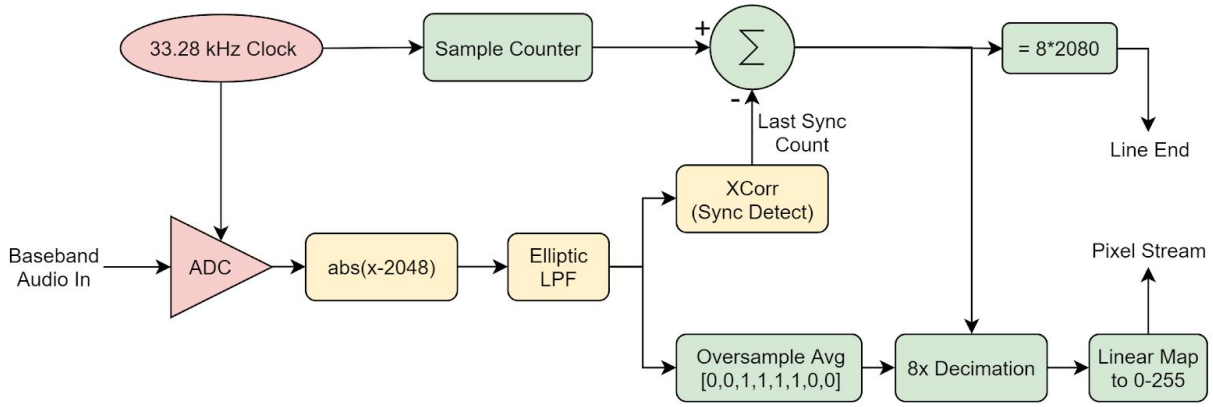


Figure 5: Signal Processing Block Diagram

The high level block diagram of the signal processing pipeline is given in Figure 5. The decoding process was broken down into 4 major stages. First, the audio signal sampled by the ADC at 33.28 kHz. This corresponds to 8x oversampling of the APT baud rate of 4160 Hz. The sampled audio is then demodulated to recover the actual signal levels. A synchronization detector picks out the synchronization pattern in order to correct for clock drift as well as to initially lock on to the signal. Finally, a framing decoder uses the sync information to produce the pixel data stream that is fed to the LCD. The entire signal processing chain was implemented in MATLAB prior to implementation on the STM32 in order to verify the validity of the approach.

The colors of the blocks in Figure 5 indicate how that section is implemented. The red blocks are hardware peripherals within the STM32. We use timer 6 for the sampling clock source configured with a prescaler of 0 and a period of 3244, corresponding to a frequency of 108e6/(3244+1) = 33,282 Hz, which is acceptably close to 33.28 kHz. The ADC used is ADC1 configured in continuous 12-bit mode with external triggering on Timer 6. It is set up to read from channel 6 which corresponds to pin PA6 which also had to be configured in analog input mode. The ADC interrupt generation is enabled to notify the main program of new samples.
The yellow blocks are implemented within that ADC interrupt routine since they contain critical filters and accumulators which cannot tolerate missed samples.

The remaining green blocks are implemented as part of the main processing loop. These components can tolerate missed samples if they have to as they can correct for the offset at the next sync pulse. That said, the sampling rate is sufficiently slow and

the main loop fast enough that the program does not miss samples in the current implementation.

Demodulation

The baseband audio signal for APT is itself a 2.4 kHz amplitude modulated carrier for 256-level analog video data at 4160 baud. At first this may seem like the modulation is of a higher frequency than the carrier, however the 4160 px/s rate actually only requires a 2080 Hz bandwidth since the highest frequency would be alternating high and low samples, with a period of 2 bits. Nonetheless, the difference between the carrier frequency and modulation bandwidth is still quite low. This means that a simple peak hold/envelope detection method will likely be unable to produce high quality results because there are so few carrier peaks per bit.

To demodulate the signal instead we employ a superheterodyne demodulation approach. The biggest challenge here is the need to recover an in phase copy of the original carrier. Fortunately, the APT video signal is always positive, which means the sign of the carrier is preserved through the modulation step. If we apply the signum function to the input we obtain a square wave which has the same phase and frequency of the original carrier. As shown in Figure 6, using the square wave version of the carrier only adds additional weak image signals, but does not affect the baseband image, so this recovery is sufficient. Furthermore, multiplying the sign of the input with itself is the same as just computing the absolute value which is used in the actual implementation (following offset subtraction).

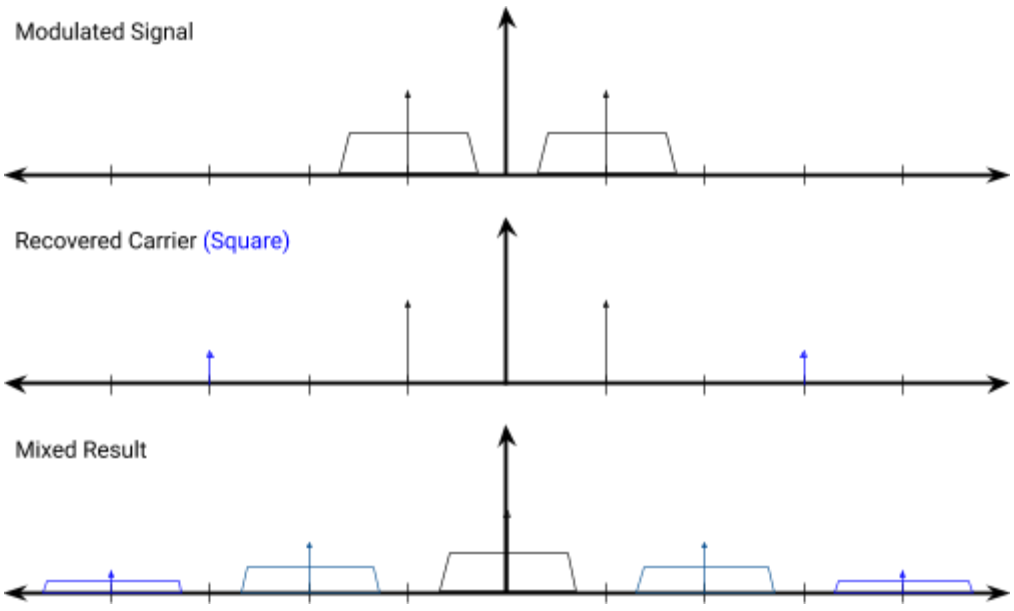
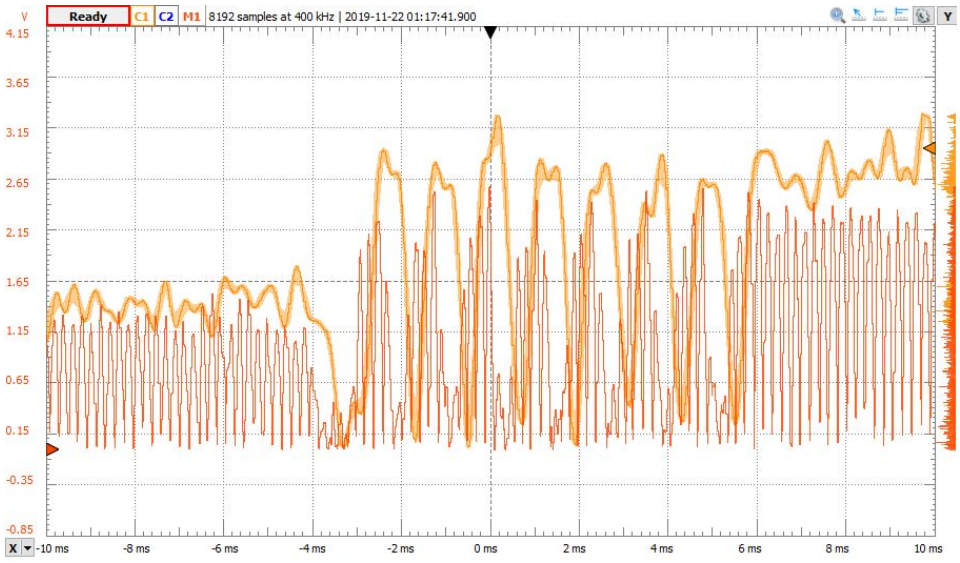


Figure 6: Superheterodyne AM Demodulation

After obtaining the baseband image, a sharp lowpass filter is applied to eliminate the image signals. An elliptic filter was used for its unparalleled rolloff as the filter needed to transition from passband to stopband in a span of only around a tenth of a decade. The filter was designed using MATLAB's designfilt tool to have 0.5 dB passband ripple up to 2100 Hz (just above the signal bandwidth of 2080 Hz) and a stopband attenuation of -68 dB which was calculated contribute no more than one tenth of one level (amplitude loss of 1/2560). The stopband start was set to 2700 Hz, right before the start of the first image signal at 2720. The resulting filter was 8th order and the specific coefficients are included in the Appendix.

The filter is implemented in the ADC conversion interrupt using floating point math. While a fixed point implementation could provide some performance benefit, ultimately since a hardware floating point unit is available and the sampling rate is relatively slow the performance gains were not worth the added implementation complexity. The values are converted back to 16 bit integers after filtering for more compact storage and more efficient processing in the later stages. An added gain of 64 (2^6) was also added to the filter such that the 12 bit inputs utilize the full 16 bit output range after passing through the filter[1]. The storage and retrieval of historical elements was handled using an 8 element circular buffer. All historical values for the filter were stored as floats to avoid the need to perform the same conversions every time the filter was evaluated.



[1] The gain of 64 is a combination of x32 to convert 11 to 16 bit (the signed input after subtraction is only really 11 bit), and an additional gain of 2 to compensate for the halving of amplitude inherent to superheterodyne demodulation

Figure 7: Demodulation vs Abs(Input)

To verify that the demodulation was working correctly, the filtered samples were output through the DAC (after scaling back to 12 bits) and measured with an Analog Discovery board. The input audio was also measured and its absolute value plotted to show peak heights. These were scaled appropriately to show the correspondence. The results are presented in Figure 7. Aside from the delay introduced by the filter, the peaks do follow the variations in the input data quite well. The delay is not a problem for actual decoding since all subsequent stages only refer to the filtered data.

Sync Detection

Now that we have the demodulated levels, the pixel sampling needs to be aligned with the center points of the incoming pixels. This is especially important since the level transitions are not nice steps owing to the extremely limited bandwidth available with only a 2.4 kHz carrier. Fortunately, the APT specification contains very distinctive synchronization patterns with large signal swings to create more defined edges. There are two slightly different sync patterns shown in Figure 8. These patterns are transmitted before each line of each image channel, and the slight difference allows identification of which channel is which. For this project we only are only using Sync A.

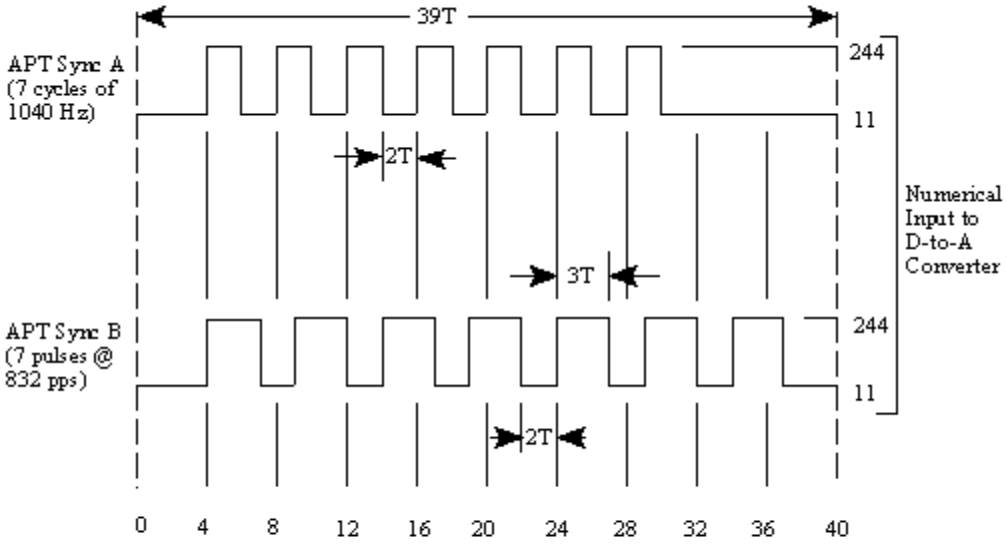


Figure 8: APT Synchronization Signals

The purpose of the sync detector is to produce the cross-correlation of the demodulated signal with the idealized sync pattern. The cross-correlation is the integral of the product of the signals as one is slid across the other. In this case that means the sum of the products between the historical signal values and the reference signal. In this way the sliding is accomplished by the shifting of the incoming samples under the

reference signal (alternatively it could be thought of as a FIR filter with impulse response equal to the time reversed reference signal). It is important the reference signal has zero mean, so a +1/-1 square wave is used.

The result of this computation is a measure of how closely the signal resembles the reference pattern. By finding the maximum of this we can identify to the nearest sample where the sync pattern ends. Since we are sampling at 8x the baud rate and the sync signal ends on a level transition, this also identifies the pixel clock phase.

The implementation of the sync detector takes advantage of the fact that the vast majority of the gains are the same to improve calculation efficiency. Instead of summing across the entire history every time, it instead keeps and updates an accumulator only for the samples that changed gain at this step (i.e. subtract 2x a sample that changes gain from +1 to -1). This reduces the computation required to only 16 operations (14 changes, subtract off the oldest sample, and add the current one). It does make the filter extremely intolerant of missed samples though, since if a sample slips through without being subtracted off it will remain in the accumulator indefinitely. This was the primary reason this was made to run as part of the ADC ISR.

The sync signal is 7 cycles with a 4T period, which corresponds to a total duration of 7*4*8 = 224 samples at 8x oversampling. These samples are stored in a 256 entry circular buffer, which is very convenient as the wraparound logic is automatically handled when using a single byte variable for the index. The accumulator is a signed 32 bit integer which can never overflow since in the worst case it will have 7*2*8=112 max 16 bit ints which is only a 23 bit number.

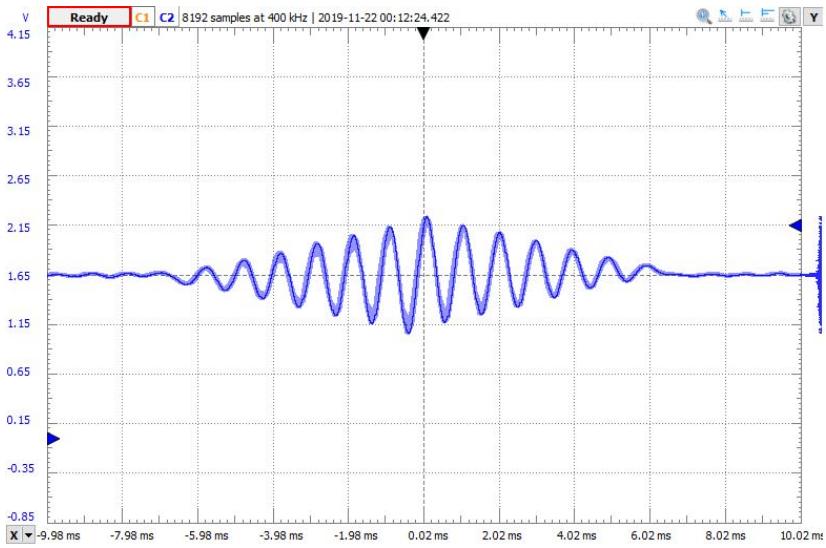


Figure 9: Cross Correlation at Sync A

Similar to the demodulation step the sync correlation signal was output to the DAC in order to verify it was generated correctly. Figure 9 shows the output when the sync occurs. As you can see the correlation oscillates as the signal goes in and out of phase with the reference, and the amplitude increases as more and more peaks are matched. The correlation at the sync pulse is also orders of magnitude higher than during the data period as evidenced by the almost negligible output before and after. To obtain this plot the actual correlation value had to be divided by 4096 before adding the offset and outputting as a 12 bit value to the DAC.

Framing

The framing code is the part of the code concerned with finding and then keeping track of where we are within the APT line and is responsible for generating the final stream of pixels that will be sent to the display. A single APT line consists of 2080 pixels containing two 909 pixel image channels as well at the sync pulses, telemetry and minute markers. The exact layout of a single APT line is provided in Figure 10.

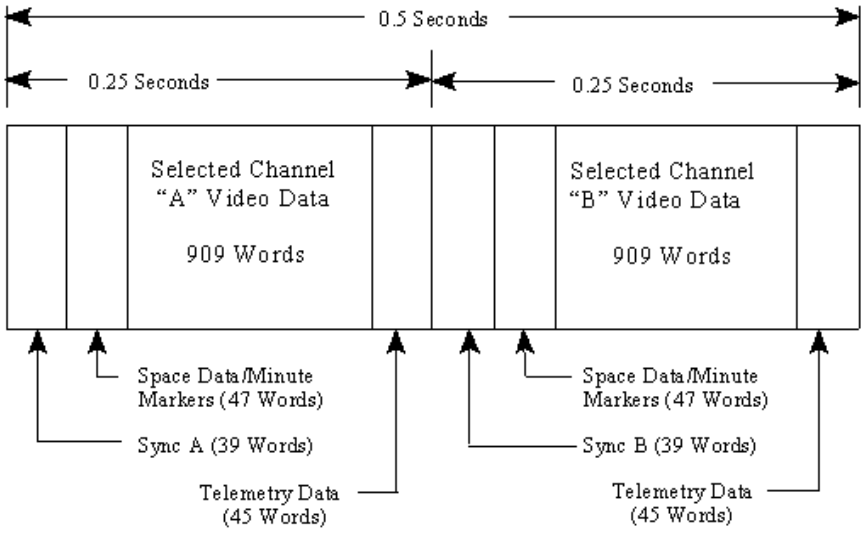


Figure 10: APT Line Format

At the highest level, the framing logic keeps a counter of the current sample offset since the start of this line. Since we are oversampling at 8x, the lower 3 bits of this counter are the current position within the pixel sampling window, and the upper bits are the current word in the line. During the sync A phase (which is overall actually longer than the sync A pattern) the maximum of the sync cross correlation is found and used to correct any drift the counter accumulated since the last line. The space interval right

after is used as a buffer phase so that pixels are not suddenly skipped if the sync corrects the counter forward past the end of section.

At the start of channel A data, the pixel values are sampled and emitted for display until the end of the line. Currently, the sampled pixel value is taken as the average of the 4 central oversamples of the value, at offsets 2, 3, 4, and 5 from the start of the sample window. The other important thing that happens at this stage is the remapping of pixels to the 255 output levels. When summing the 4 central samples the result is an 18 bit integer, which must be mapped to 0-255. The simplest method would be a simple shift by 10 bits, however this requires the assumption that the input audio utilized exactly the full input range of the ADC, which is not the case in general.

The correct way to do this would be to wait for enough lines to come in to see all the telemetry wedges, which contain both max and min modulation. We did not find this approach practical though and instead simply use the max and min sample encountered when the signal is first found. Our rationale was that the sync signal will guarantee that the signal comes pretty close to max and min modulation somewhere in the line, and that these values will prove sufficient for good display of the image. Using a constantly running max/min was considered, but ultimately we thought it would be strange if the brightness range changed part way through drawing the image, so we preferred the use of a consistent value determined at startup.

Speaking of startup, the last task the framing logic needed to support was a way to find the signal from scratch whenever the program starts or the signal is lost. Fortunately, this is effectively just an extended version of the usual sync finding algorithm. The system simply keeps track of the time of maximum sync correlation across the equivalent duration of 1 line. No matter where within the line this started, since we return to the same position at the end of the interval, the sync pattern must have occurred somewhere in that time[2]. The count at maximum correlation (minus the sync pattern length) is then subtracted off the sample counter to reference it to the start of the identified line. Processing can now begin as usual on the next line.

[2] Note that since the sync correlator is free running, it does not matter here if the *entire* sync pattern is in the interval (it could be split if it starts during sync), only that the end is (where max correlation occurs).

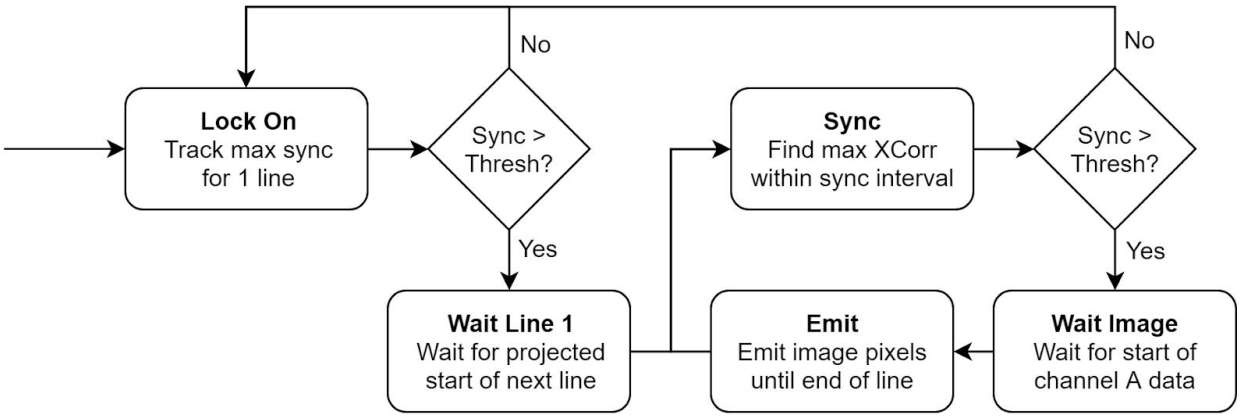


Figure 11: Framing System State Machine

The framing system is implemented as a finite state machine which runs as part of the main code loop. A flag set by the sampling code informs when new samples are available. The overall state diagram is given in Figure 11. The current state is stored in an enum and checked with a switch statement on each pass to select which logic to perform. Most states simply increment the sample counter and wait until a particular threshold, maybe passively updating the sync maximizer as it does. The emit state is the most interesting as it contains its own little state machine per pixel time that handles resetting the accumulator (at sample 0); summing the middle samples at offsets 2, 3, 4, and 5; and mapping and emitting the pixel byte on sample 7.

Something that was not discussed before is the sync threshold, which determines the minimum allowed value for a maximum sync to be considered a real signal. This makes the system more robust in the case the signal is lost and then regained, as it will return to the full line search and can pick up the reacquired signal anywhere within the line. It also has the benefit that it will only output pixels for known strong lines, so the output image will not be filled in with noise or black when no signal is present.

The framing system was initially tested outputting the pixel levels to the DAC. The drift corrections for each line were also printed to the serial port (in DMA mode so as not to affect the main loop speed) to ensure the timing was accurate. Indeed the counter usually only drifts by +/- 1 sample (1/8 of a pixel) over the whole line. Some sample pixel outputs for a whole line, and zoomed around the B sync pattern are given in Figure 12. Note the flat region at the start of the line is the sync and space interval where no pixels are emitted. The high amplitude B sync and space is also clearly visible in the center separating the two image segments. You can also see that the 3 high 2 low pattern of the B sync is correctly captured in the emitted pixel sequence.

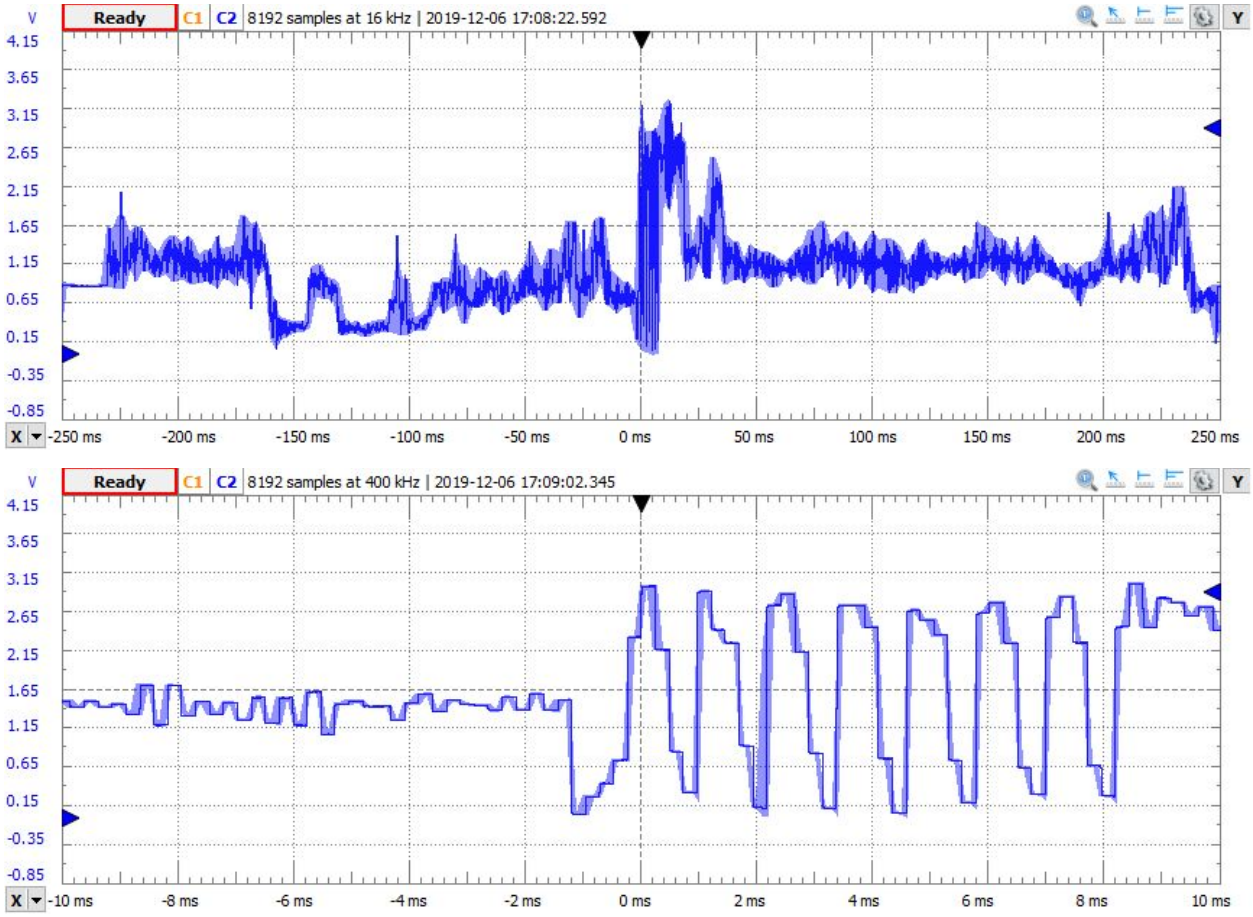


Figure 12: Decoded Pixel Streams (Top: 1 line, Bottom: Sync B)

LCD Display:

The LCD display takes pixels from the DISP_EmitPixel function and draws them using the BSP_LCD_DrawPixel() function. This comes from the built in lcd driver, located in stm32f769i_discovery_lcd.c/.h. This could be done within the HAL, but this was found to be tedious and generally difficult, as the data is stored in SRAM which requires the configuration of that as well.

To use this with the LCD Driver, the otm8009a.c/.h BSP component needed to be added to the build. Other files from the discovery board BSP folder that needed to be added to the build are as follows: stm32f769i_discovery.c/.h for general I/o classes, and stm32f769i_discovery_sdram.c/.h for the memory configuration and usage for storage of the image. Tha HAL also needed files included into the build, including: stm32f7xx_hal_dma2d.c to store the image, stm32f7xx_hal_dsi.c for the display/serial interface, stm32f7xx_hal_ltdc.c and stm32f7xx_hal_ltdc_ex.c to control the display,

stm32f7xx_hal_sdram.c and stm32f7xx_hal_sram.c for storage, and stm32f7xx_ll_fmc.c, a HAL driver for the flexible memory controller.

This driver is initialized using the function BSP_LCD_Init()in the DISP_Init() function, which first configures the LCD driver to the settings of the connected display. In our case this was the touch screen that came with the discovery board, the otm8009a IC display. These settings include lcd_y_size and lcd_x_size, giving the total height and width respectively. Other display variables include Horizontal Front Porch (HFP) Horizontal Sync (HSYNC), Horizontal Back Porch (HBP), Vertical Front Porch (VFP) Vertical Sync (VSYNC), Vertical Back Porch (VBP). these can be used to calculate Active Width and Active Height. All of these parameters are pictured below in Figure 13.

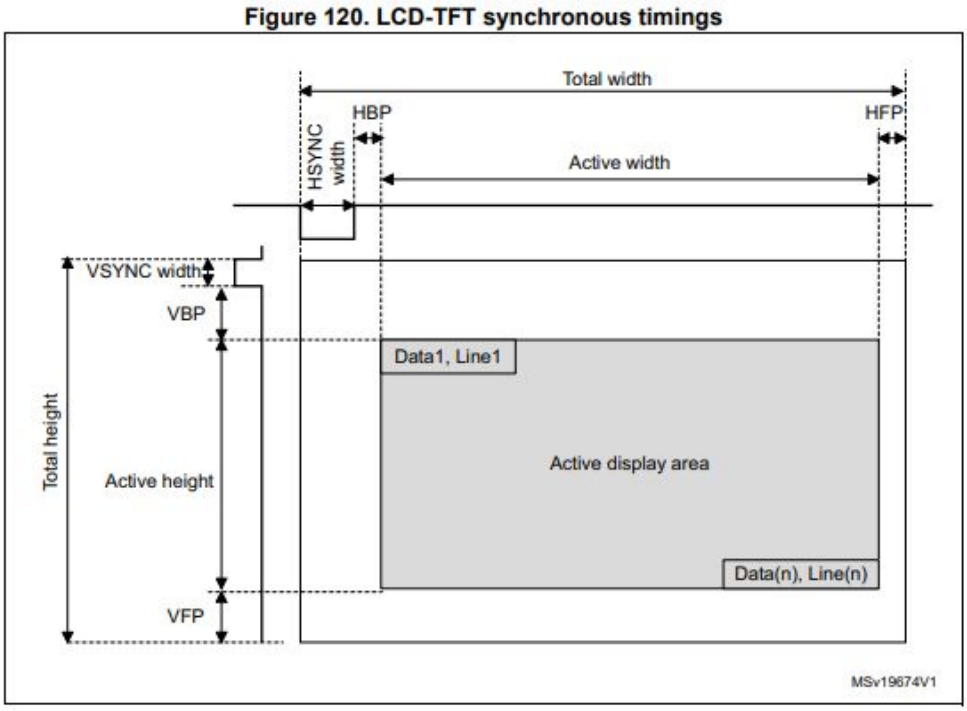


Figure 13: Display ‘area’

These parameters are used to define the timing for each frame of the display, refering to the size of the screen in units of lcdClk for the horizontal variables, and units of line for the vertical parameters. A frame timing diagram is shown below in Figure 14. For our purposes, the LCD clock rate is 27429 kHz

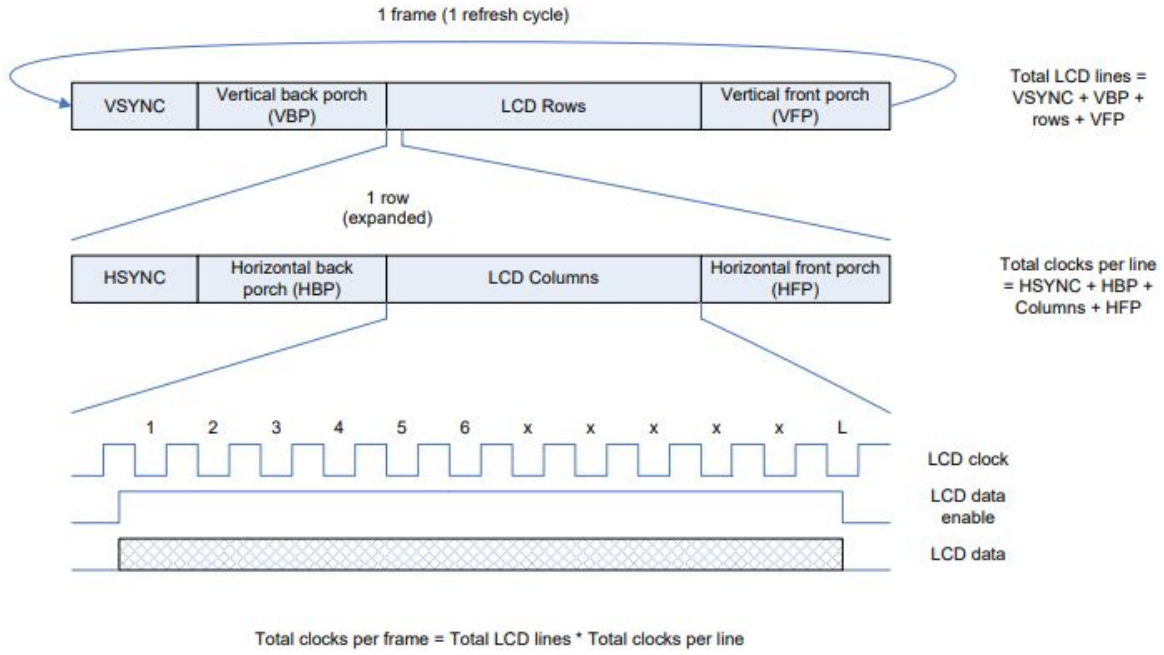


Figure 14: Frame timing

Next, the driver calls BSP_LCD_MspInit() to set IP blocks LTDC, DSI and DMA2D: out of reset, synced to the clock, and enabled NVIC IRQ related to IP blocks. After this the driver uses a DSI_HandleTypeDef and DSI_PLLInitTypeDef (phase lock loop) to configure the data serial interface. This includes setting the number for data lanes and clock speed, as well as PLL parameters like the PLL Input Division Factor and PLL Output Division Factor. The driver then configures a DSI_VidCfgTypeDef with the frame timing parameters, as well as the vertical sync polarity, horizontal sync polarity, data enable polarity, video mode, packet size, null packet size. For our case, the video mode is burst, which time compresses the RGB pixel packets, leaving time to go into a low power mode between line transmissions.
HAL_DSI_Init is called with the DSI_HandleTypeDef and DSI_PLLInitTypeDef as parameters. Both vsync and hsync polarities were set to active high by setting the DSI_HandleTypeDef parameters HSPolarity and VSPolarity , as was the data enable, DEPolarity. Then, each region of the video frame (VBP, HFP, etc) is set to allow commands during low power mode by setting the DSI_VidCfgTypeDef property '.LP[frame parameter]Enable', to DSI_LP_[parameter abr.]_ENABLE for example: DSI_VidCfgTypeDef.LPHorizontalFrontPorchEnable = DSI_LP_HFP_ENABLE.
Finally, the HAL_DSI_ConfigVideoMode function is called with both the DSI_HandleTypeDef and DSI_VidCfgTypeDef by reference.

Next, the LTDC is configured; this is done via the LTDC_HandleTypeDef, hltdc_discovery. First, the horizontal timing parameters are configured, including: HorizontalSync (HSA -

1), AccumulatedHBP (HSA + HBP - 1), AccumulatedActiveW (lcd_x_size + HSA + HBP - 1), and TotalWidth (lcd_x_size + HSA + HBP + HFP - 1). The function then sets the LCD pixel width and pixel height, using: hltdc_discovery.LayerCfg->ImageWidth = lcd_x_size and hltdc_discovery.LayerCfg->ImageHeight = lcd_y_size. LayerCfg is an array of LTDC_LayerCfgTypeDef, which handles the individual layers on the screen. Layers are used to overlay images. For our purposes we only use one layer, LTDC_DEFAULT_ACTIVE_LAYER, but there is a maximum of 2 that we can use, not including the background. Next, the LCD Peripheral Clock is configured. This uses a PeriphClkInitStruct that holds PLL parameters, like the PLLSAI division factor for LTDC clock. HAL_RCCEx_PeriphCLKConfig is then called, which actually initializes the RCC extended peripherals clock for the LCD. The last parameters to be set in the function set up the background color, pixel clock polarity, and Instance for the ltdc handle. The parameters include: ltdc_discovery.Init.Backcolor.Blue, hltdc_discovery.Init.Backcolor.Green, and hltdc_discovery.Init.Backcolor.Red for the background colors, hltdc_discovery.Init.PCPolarity is set to LTDC_PCPOLARITY_IPC for input pixel clock, and the instance parameter is set to LTDC.

The BSP_LCD_InitEx function (remember the BSP_LCD_InitEx function that we talked about earlier that we are still in) then gets the LTDC configuration from the DSI configuration via the function HAL_LTDC_StructInitFromVideoConfig, with the LTDC_HandleTypeDef and DSI_VidCfgTypeDef as parameters. This retrieves the polarity and timing parameters from the DSI handle and sets the LTDC handle to the same configuration, which is pretty neat.

Next (still in BSP_LCD_InitEx), the LTDC is initialized by calling HAL_LTDC_Init with the LTDC handle as a parameter. This checks and then sets the configured timing and frame parameters, enables the transfer error interrupt, enables the FIFO underrun interrupt, enables LTDC by setting LTDCEN bit, and initializes the LTDC state to HAL_LTDC_STATE_READY. To avoid any synchronization issues, the DSI is started after enabling the LTDC. HAL_DSI_Start() is called with the DSI handle. If the SDRAM had not been set up, BSP_LCD_InitEx then calls BSP_SDRAM_Init(). Lastly, BSP_LCD_SetFont is called with the LDC default font.

Finally out of BSP_LCD_InitEx(), the initialization is done. The backlight, which is controlled separately from the pixels, is turned on with the BSP_LCD_DisplayOn() function. Next, even though the LCD is initialized, the layer we want to use has to be initialized. To do this we call BSP_LCD_LayerDefaultInit(LTDC_DEFAULT_ACTIVE_LAYER, LCD_FB_START_ADDRESS), where LTDC_DEFAULT_ACTIVE_LAYER has been defined

as Layer 1 in the LCD driver, and LCD_FB_START_ADDRESS has been been defined as 0xC0000000 in the LCD driver. In the function, a LCD_LayerCfgTypeDef is created and configured to the correct size, background color, and PixelPixelFormat. The default size is that of the whole display, with properties WindowX0 set to 0, WindowX1 to BSP_LCD_GetXSize() (which gets the lcd size in x), WindowY0 set to 0, WindowY1 to BSP_LCD_GetYSize() (which gets the lcd size in y). PixelFormat defaults to LTDC_PIXEL_FORMAT_ARGB8888. The figure below shows how ARGB8888 is stored:

Table 38. Data order in memory

Color Mode	@ + 3	@ + 2	@ + 1	@ + 0
ARGB8888	A_0[7:0]	R_0[7:0]	G_0[7:0]	B_0[7:0]

Figure 15: ARGB8888 Format

Once the layer type def is configured, the LTDC layer array mentioned previously needs to be updated to match. This is done by calling HAL_LTDC_ConfigLayer with the parameters of the LTDC handle, the filled out layer type def, and the layer index that the LayerDefaultInit function was called with. To accomplish this, the HAL_ConfigLayerFunction Process locks the LTDC handle, changes the state to HAL_LTDC_STATE_BUSY, asserts that all the parameters are defined, copies the new layer configuration into the LayerCfg array, Configure the LTDC Layer by calling LTDC_SetConfig, changes the state back to HAL_LTDC_STATE_READY, and unlocks the handle. Once done with the DefaultLayerConfig, the Current Drawing Layer properties variable, a LCD_DrawPropTypeDef DrawProp, is set to have a white background, font size of 24, with black letters.

The next step is to set the layer that we configured to be visible. This can happen before, after, or between drawing on the layer, but this is when we did so. To do this we called BSP_LCD_SetLayerVisible(LTDC_DEFAULT_ACTIVE_LAYER, ENABLE). To draw on a layer, you first have to select the layer. This is done with the BSP_LCD_SelectLayer function, called with the layer index you want to select. In our case this was LTDC_DEFAULT_ACTIVE_LAYER. This sets the global variable activelyer in the lcd driver to the layer index we selected, which is used in the draw functions described later. Finally, we clear the screen with BSP_LCD_Clear(LCD_COLOR_BLACK). This is because when the layer is initially created, it is set to whatever state the sram is in, which is not blank on initialization, giving an image like the one in Figure 16 below

Figure 16: SDRAM without clearing

The display.c file keeps track of the output position in the form of a position struct, and has the functions DISP_Init, DISP_EmitPixel, and DISP_NewLine. DISP_init was talked about previously. The framing function discussed earlier calls DISP_EmitPixel to display each pixel, which has the ability to apply any downsampling or screen orientation operations. To downsample, the DISP_EmitPixel function checks if the row and column of the pixel, stored in the position struct, can be divided without remainder by a predetermined hardcoded downsampling coefficient. If there is no remained, the pixel is displayed using the BSP_LCD_DrawPixel() function, with the desired x location, desired y location, and color, as arguments. Since the image is black and white, the red, green and blue values need to be equal;all of them are set to the pixel value, which can range from 0 to 255. The column position is increased by one, and if necessary, the row position is as well using the DISP_NewLine function.

Results and Conclusions

To test the final system, a long sample of APT audio from NOAA-19 was downloaded from the SatNOGS, an open source satellite ground station network. The decoded image from the sample is shown Figure 17. This observation contains data from one pass of the satellite northwards over Europe. Since the image is shown with lines in chronological order, north is down in the image, with the top of africa appearing at the top, Italy and the Mediteranean in the darker water region, mainland Europe in the middle, and the Scandinavian peninsula just visible under the clouds in the lower left.

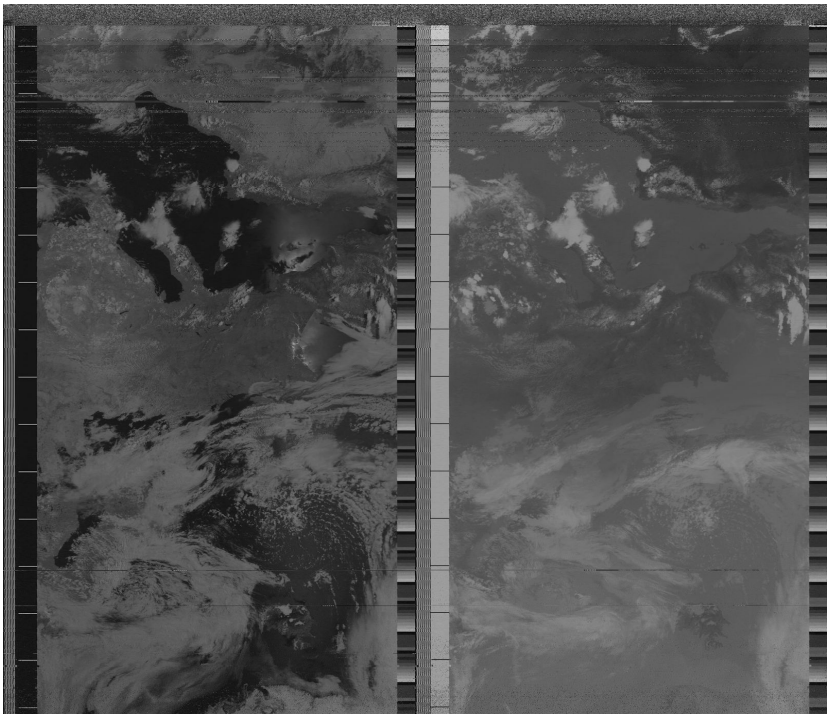


Figure 17: Example Decoded NOAA-19 Image

```
Drifted +1 Samples
Drifted +1 Samples
Lost sync!
LOCKED
Drifted +3 Samples
Lost sync!
LOCKED
Lost sync!
LOCKED
Lost sync!
LOCKED
Drifted +1 Samples
Drifted +0 Samples
Drifted +0 Samples
Lost sync!
LOCKED
Drifted +0 Samples
Drifted +0 Samples
LOCKED
Drifted +0 Samples
Drifted +0 Samples
Drifted +1 Samples
Drifted +0 Samples
Drifted +1 Samples
Drifted +0 Samples
```

Figure 18: Sync Info for Noisy Segment

You can also see some bands of noise near the beginning of the acquisition as the station first acquires the satellite transmission. The audio file also contains skips in these regions. This posed a challenge to the system initially since this would cause it to shift further than the existing sync detection could correct. This motivated the addition of the sync threshold. In the final version, the noisy section is handled quite well. Figure 18 shows the synchronization output for that region. As you can see, the system detects the loss of signal, and starts seeking over the entire line for a new sync. It then re-locks at the next clear line. Also evident is the fact that when we do have a clear signal, our baud rate timer and the transmission baud rate are quite well matched. At we typically only have to correct 1 sample of clock drift over the full 16640 sample line.

Of course the real results are of course the decoded images. Figure 19 shows the decoded image on the STM32 Discovery's LCD screen compared against the reference image on a laptop. As you can see, the replication is excellent. This test is using 3x downsampling to fit the full 1994 px width onto the 800 px screen width (using 664 px). Other tests run at full resolution showed clear reproduction of the visible portion (most of stream A) as well.

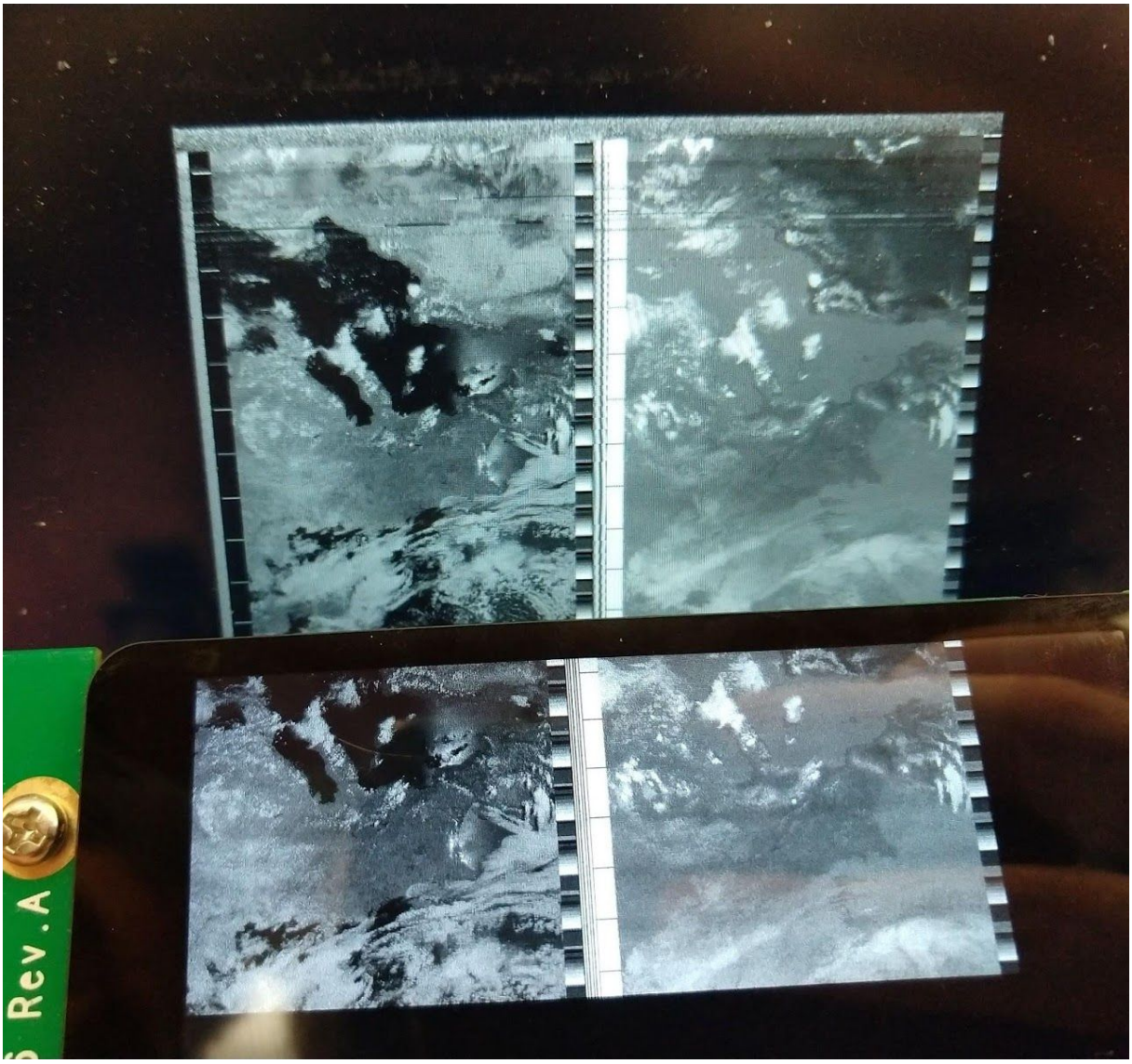


Figure 19: Decoded Image on LCD vs. Reference

Appendix

Elliptic Filter Coefficients:

X[k]	0.000678838494336		
X[k-1]	-0.003534196288993	Y[k-1]	7.200781848522088
X[k-2]	0.008994777919394	Y[k-2]	-23.028289310289551
X[k-3]	-0.014666976055880	Y[k-3]	42.687438666850078
X[k-4]	0.017088053114672	Y[k-4]	-50.139405828371544
X[k-5]	-0.014666976055880	Y[k-5]	38.197599092024909
X[k-6]	0.008994777919394	Y[k-6]	-18.427485269037746
X[k-7]	-0.003534196288993	Y[k-7]	5.146330495211290
X[k-8]	0.000678838494336	Y[k-8]	-0.637004588053818

References

APT References

NOAA KLM User Guide (APT part starts page 225)

Weather Images Of NOAA Satellites

APT - Signal Identification Wiki

Test Sample

SatNOGS Observation

Libre Space Forum with Synchronized Image

LCD Graphics

Introduction to graphics and LCD technologies

DSI Resources

Phase-locked loop

HAL LTDC Driver

STM32 Cube LTDC example nightmare | All About Circuits

STM32F429 LCD TFT controller