VGA INTERFACE using ATMEGA16

**Project Report**

*Submitted by*

*Aditya Savaliya 160230111051*

*Mohit Agrawal 160230111033*

*Dharmik Padhara 160230111037*

*In partial fulfillment for the award of the degree*

*of*

BACHELOR OF ENGINEERING

in

*Electronics and Communication Engineering*

*Dr.S S. and S.S. Ghandhy College of Engineering and Technology*



Gujarat Technological University, Ahmedabad

2019 – 2020

Contents

[**Deceleration** 4](#_Toc42684184)

[**Certificate** 5](#_Toc42684185)

[**Acknowledgement** 6](#_Toc42684186)

[**ABSTRACT** 7](#_Toc42684187)

[Crux of the literature 10](#_Toc42684188)

[VGA TIMINGS 12](#_Toc42684189)

[INTRODUCTION 12](#_Toc42684190)

[H-SYNC and V-SYNC 13](#_Toc42684191)

[PICTURE STRUCTURE OF VGA TIMINGS 15](#_Toc42684192)

[Insight ‘zero’ 16](#_Toc42684193)

[HARDWARE DESIGN 17](#_Toc42684194)

[Why AVR atmega16? 17](#_Toc42684195)

[75Ω terminal impedance 18](#_Toc42684196)

[Circuit diagram for monochromatic image 19](#_Toc42684197)

[Limitations of atmega16 and alternatives. 20](#_Toc42684198)

[SOFTWARE DESIGN 21](#_Toc42684199)

[INTRODUCTION 21](#_Toc42684200)

[Generating H-SYNC 22](#_Toc42684201)

[GENERATING V-SYNC 23](#_Toc42684202)

[PROGRAM FLOW 26](#_Toc42684203)

[DATA OUT 27](#_Toc42684204)

[DATA OUT USING SHIF LOGIC (Monochromatic Image) 29](#_Toc42684205)

[GRAPH - MONOCHROMATIC IMAGE 30](#_Toc42684206)

[IMAGE PROCESSING 31](#_Toc42684207)

[monochromatic image to hex 34](#_Toc42684208)

[IMAGE FROM MONITOR 35](#_Toc42684209)

[INSIGHT 1 36](#_Toc42684210)

[PRELOADING DATA DURING THE SYNC PULSE & Back Porch 36](#_Toc42684211)

[Using EEPROM 37](#_Toc42684212)

[DMA (DIRECT MEMORY ACCESS) 38](#_Toc42684213)

[SPI (SERIAL PERIPHERAL INTERFACE) 39](#_Toc42684214)

[Hardware Design (Polychromatic Image) 42](#_Toc42684215)

[R-2r Resistor ladder network 42](#_Toc42684216)

[CIRCUIT DIAGRAM FOR POLYCHROMATIC IMAGEGRAPH (POLYCHROMATIC) 44](#_Toc42684217)

[DATA OUT Using PRE-LOADING (POLYCHROMATIC IMAGE) 46](#_Toc42684218)

[Polychromatic image to hex 47](#_Toc42684219)

[Ambitious Ideas 48](#_Toc42684220)

[STS code V4 48](#_Toc42684221)

[.GIF 48](#_Toc42684222)

[HARDWARE 50](#_Toc42684223)

[ATMEGA16 50](#_Toc42684224)

[IDE 54](#_Toc42684225)

[PROGRAM MEORY 57](#_Toc42684226)

[CODE 63](#_Toc42684227)

[CODE 69](#_Toc42684228)



DR. S. & S.S. Ghandhy College of Engineering and Technology

*Ring Rd, Majura Gate, Surat, Gujarat 395008*

## **Deceleration**

We hereby declare that the PSAR Reports, submitted along with the Project Report for the project entitled “\_\_*VGA INTERFACE using ATMEGA16*\_\_” submitted in partial fulfillment for the degree of  *Bachelor of Engineering in Electronics and Communication*

to Gujarat Technological University, Ahmedabad, is a bonafide record of the project work carried out at DR. S. & S.S. Ghandhy College of Engineering and Technology

under the supervision of Rana soham and that no part of any of these PSAR reports has been directly copied from any students’ reports or taken from any other source, without providing due reference.

*Name of The Students Sign of Students*

*1.Mohit Agrawal*

*2. Aditya Savaliya*

*3.Dharmik Padhara*



DR. S. & S.S. Ghandhy College of Engineering and Technology

*Ring Rd, Majura Gate, Surat, Gujarat 395008*

## **Certificate**

This is to certify that the PSAR reports, submitted along with the project entitled VGA INTERFACE using ATMEGA16 has been carried out by Aditya Savaliya, Mohit Agrawal and Dharmik Padhara under my guidance in partial fulfillment for the degree of: Bachelor of Engineering in *Electronics and Communication* 7th & 8th Semester of Gujarat Technological University, Ahmadabad during the academic year 2019-20. These students have successfully completed PSAR activity under my guidance.

Internal guide Head of epartment

## **Acknowledgement**

I firstly thank Gujarat Technological University for giving me this opportunity to pursue a project that would enable me to learn so much about the practical aspects of the subjects that I learn in academic curriculum. I would like to express my sincere thanks to Chaitanya Thumar, Scientist, ISRO for his valuable guidance throughout the process of realizing ideas in to the project, without which it would’ve been just a cool project idea which I saw on internet. I thank my friend Utkarsh Raval for giving me the appropriate resources required for the completion of my project. I am also thankful to my teachers and faculty members without whom this project would never made it to the GTU portal in due time. I would also like to thank my family and friends for their constant criticism and appreciation for all of my endeavours.

## **ABSTRACT**

Our objective is to build a cost-effective VGA interface using atmega16 which should be capable of displaying some sort of image as for the proof of concept. The VGA signals are generated using the timers of microcontroller, which are also connected back to the microcontroller in order trigger an interrupt. Upon the interrupt, the microcontroller pushes the image stored in program space on to the data pin of VGA connector (DB-15). The image resolution we realized is 160x80 monochromatic image on screen resolution of 640x480 with refresh rate of 60Hz. The program code was drafted on AVR Studio 5.1, simulated on Proteus 8 and hex file was burned using standard ISP Programmer.

INTRODUCTION

We have come a long way from using panel of light bulbs just to display the register contents of the state machine to using 4k densely packed LEDs to display Big Boss, indeed humans are strange.

Back in the day when electric computers were blooming among the engineers, there was limitation on tracking of the program’s operation unlike the result of the program itself which can easily be monitored using panel of bulbs.

Discovery of CRT screens eventually took over the bulbs, which was again surpassed by LCDs and LEDs.CRT screens may not be around, but the functional principle of LED screens is same as in the aspect of timings of the VGA signals. That concludes my justification on the importance and the choice of the problem statement, stated in following section.

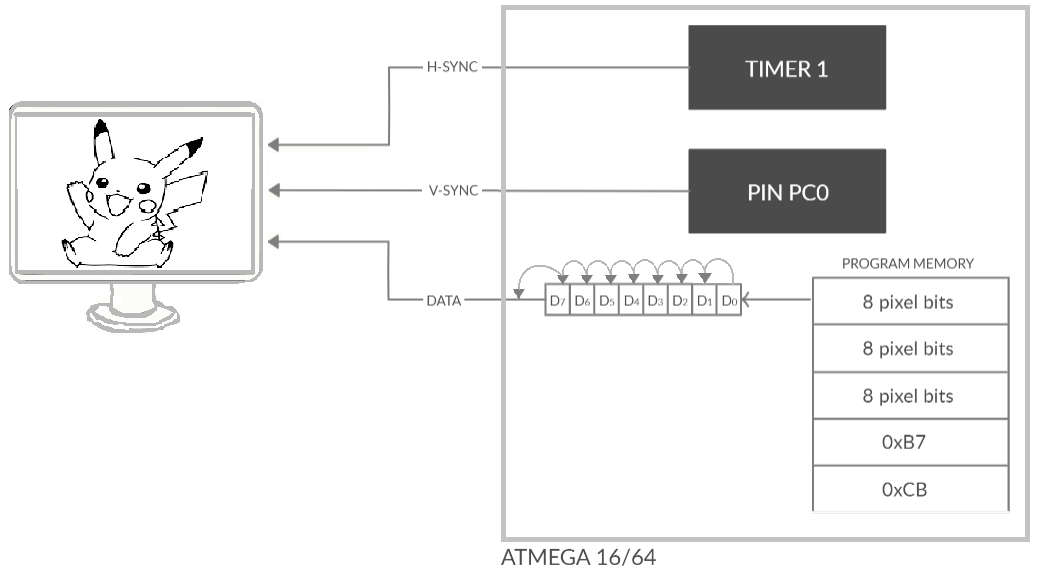
Problem Statement: The idea here is to build a prototype hardware which should provide proof of the concept of VGA interface by generating appropriate pattern which concise of the pixel synchronization pulses and image data.

I believe this project is good opportunity to learn the fundamentals of VGA signal generation and gain practical insights on the actual capabilities and efficiency of AVR microcontroller which is a RISC chip.

I will sequentially write about my learnings and findings under the title of ‘Insight’.

My study and findings may not have any commercial application but it surely can serve educational purpose for the institute.

*“useless as it stands, but interesting none the less.”*

Related LITERATURE REVIEW

Program

Image data

I have referred the text by **Ben Eater** on building a video card from scratch where he uses *fundamental digital electronics components* like Ripple Counter to count the pulse generated by 10MHz external crystal oscillator, couple of NAND gates and EEPROM to store the image data. He also used the resistor ladder instead to DAC to create palate of 16 colours.

This text helped to understand the crude functionality, standards and timing of VGA. However, as his goal was to build it from simplest components possible, it did not include information about the data timings and detailed image extraction process.

REFERENCE: <https://eater.net/vga>

The other text I came across was from **LUCID SCIENCE**, website by electronics hobbyist.

They implemented the same idea using *atmega644p* clocked at 20MHz and even overclocked a bit to obtain better resolution. Their code was not very efficient as the toggling a PIN for H-SYNC and V-SYNC were on timer compare interrupt mode and to generate time delays like front porch they used too many NOP instruction, which resulted in precise time operation but I don’t think it was a wise approach and even they thought that, as website read *“This cycle counting and equalizing process is the real chore, and can be a huge challenge when your code becomes more complex.”*

A different approach would be, using **PWM Mode** of timer for H-SYNC and V-SYNC which leaves processor free from performing chores.

The method employed here for printing a pixel was simple as pushing entire byte to a PORT for polychromatic image (it requires two instruction per pixel), Compared to ours where image is monochromatic due to limited memory space which make code more complex where rolling bits over a single pin followed by out instruction is used to print every pixel (also takes two instructions).

They have two versions to this; in the second version they use external SRAM which act like pseudo DMA support.

REFERENCE: <http://lucidscience.com/pro-vga%20video%20generator-1.aspx>

## Crux of the literature

What I learnt form the referring various texts is, that in order to paint a pixel on the screen which follows VGA standards, we need to send in five basic signals.

**H-SYNC**: Horizontal Synchronization (**Digital** *0V-5V*)

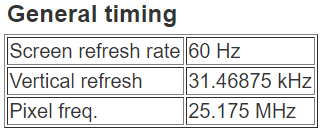
**V-SYNC**: Vertical Synchronization (**Digital** 0V-5V)

**R**: Red (**Analog** 0V-0.7V)

**G**: Green (**Analog** 0V-0.7V)

**B**: Blue (**Analog** 0V-0.7V)

V-SYNC and H-SYNC are timing signals which helps to synchronize the pointing cursor of monitor with the data being received from the memory space of microcontroller or external SRAM. They also help monitor to identify the resolution which will eventually decide the incoming data rate along with the refresh rate. Monitor decodes frequency of incoming H-SYNC and V-SYNC to set the logical pathways for pixel printing at that specific rate.

Timing of H-SYNC and V-SYNC pulse is pre-defined according to VGA standards for and given particular resolution. A good monitor would support couple of different resolutions.

The resolution we choose for our project is industry standard **640x480**, and if you look closely in the figure, then you can see it requires the **pixel clock of 25MHz.** i.e. it expects a pixel every **0.0397 µS** from the hardware connected.

0.0397µS is tiny time window for sending information (image data) from a logical circuit like Atmega16 as it only supports the **maximum clock frequency of 16MHz**. i.e. **0.0625 µS for most RISC instruction**, even if there exists an instruction which could send a bit from memory space to a PORT in single instruction it would still be **1.5 times slower**.

Unfortunately, **Atmega series doesn’t have DMA** (Direct Memory Access) so it would take couple of instructions to send a bit to any of the POAT.

Timing signals are relatively easy to generate but pushing maximum data at the right time is challenging. Because of pixel clock feasible on the hardware is much slower than required one so we focused on drafting the most efficient code possible. The code is far from being perfect as there are plenty of ideas which are yet to be explored and examined.

This review will not explore why the timing signals are still same as that of CRT monitor even with the availability contemporary logical circuit which are faster than ever before.

Design and Methodology

# VGA TIMINGS

## INTRODUCTION

VGA standards were developed around this concept of synchronization pulse.

In earlier days, before the advent of LED or LCD monitors, the pixels were illuminated with the help of particle beams. And when I say illuminated, literally illuminate as the particle beam was projected on the phosphorescent screen.

The intensity of the pixel is controlled by the accelerator grid and the location where it is to be pointed is controlled by an electromagnetic coil.

Appropriate delay time is required for the beam to swing from left to right and again from right to the starting of next line. This time delay allows the beam to settle from the DC voltages in the line before it starts reprinting.

Particle beams may not be around but VGA standard timing for H-SYNC and V-SYNC are still based on the method for painting a pixel from L to R using an electron beam.

**So how are the pixels being printed nowadays?**

There are three basic components to the VGA signal

1. *Image data*
2. *Horizontal synchronization pulse*
3. *Vertical synchronization pulse*

Both Horizontal and Vertical synchronization pulses have three segments each.

1. *Front porch*
2. *Sync pulse*
3. *Back porch*

**Front porch**: Brief time between sync pulse and colour burst which indicates that electron beam is done painting the line.

**Sync pulse**: Electron beam is swinging back to the left in order to start a new row.

**Back porch**: The *back porch* is the portion of each scan line between the end (rising edge) of the horizontal sync pulse and the start of active image. It allows sync pulse to settle down before the next data is to be received.

## H-SYNC and V-SYNC

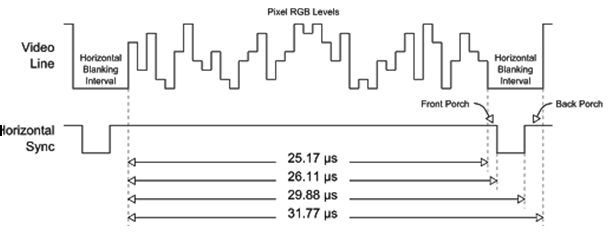
The horizontal sync pulse tells the display which **line** is being printed.

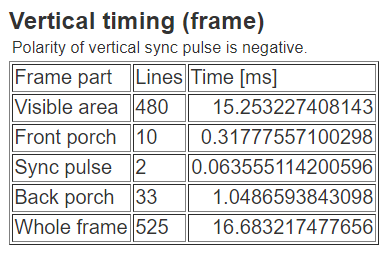
Fundamental H-SYNC is generally considered as a line.

If resolution is 640x480, then a **line is 640 pixels wide** and monitor takes **TH-SYNC time to print the whole line**.

Higher resolution requires monitor to print greater number of pixels hence the time period of H-SYNC for that particular resolution will increase.

Timing of H-SYNC will affect the resolution directly, more the TH-SYNC less the TV-SYNC for a fixed refresh rate.





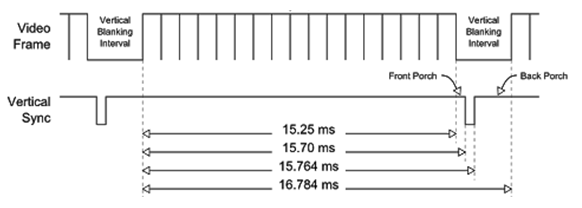
The vertical sync pulse tells the display when each frame ends.

It is generally counted or calculated in terms of line (THSYNC).

Resolution **640x480 have 525 line between two consecutive sync pulses** **Out of those 525 lines 480 are visible** and rest are for blanking time of sync pulse, back porch and front porch.

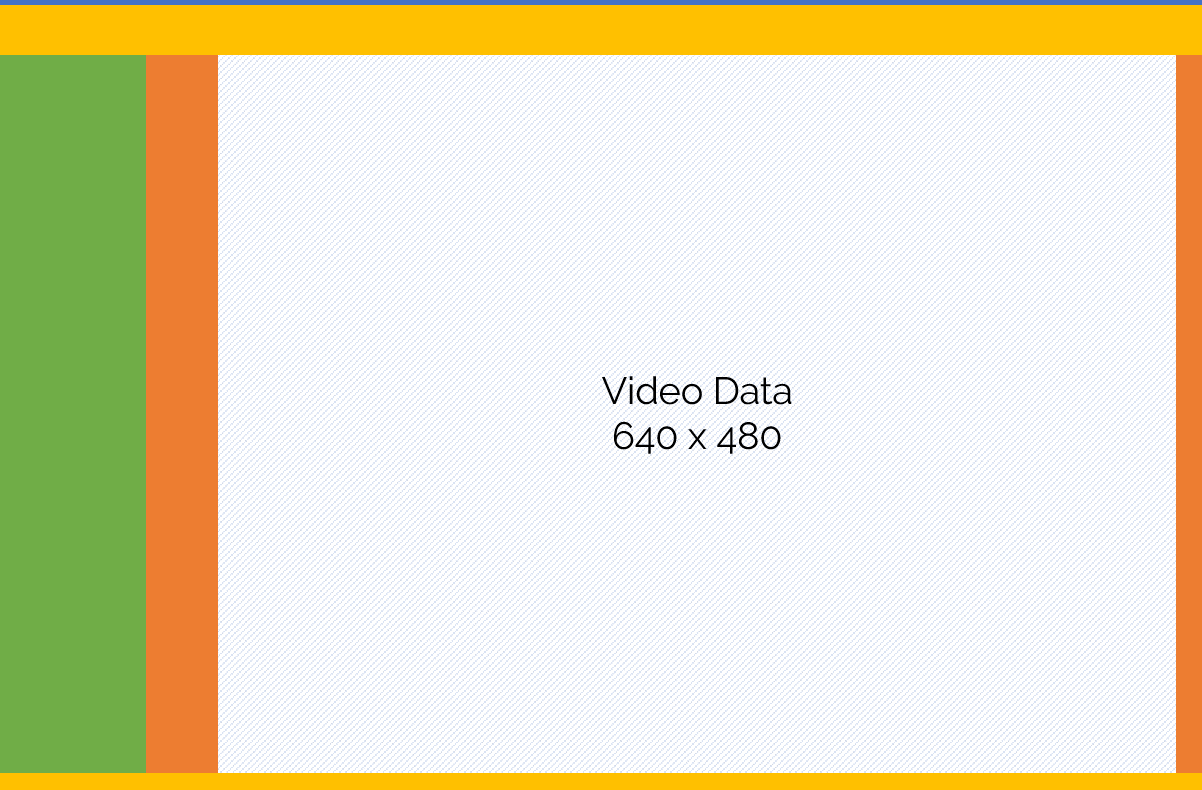
Hence total time taken to print the whole frame is **TH-SYNC (31.77 µS) ×525 = 16.6832 mS**.

**1÷16.6832 mS = Refresh Rate of 60Hz.**



## PICTURE STRUCTURE OF VGA TIMINGS

Sync pulse (V-SYNC)



Back Porch (V-SYNC)

Front Porch (H-SYNC)

Front Porch (V-SYNC)

Sync Pulse (H-SYNC)

Back Porch (H-SYNC)

This diagram sets the record straight, about when to send the data and when not to. If this prototype is to be demonstrated using CRT monitor, sending data at wrong time could damage the monitor as beam is activated while pointing at a place when it should not be active.

However, it’s not possible to damage LED or LCD monitor by sending in data at incorrect time as long as it is below 0.7 Volts.

The following numbers will be given for a 640x480 resolution, complete and detailed information about other resolution of the VGA standard can be found on

REFERENCE: <http://tinyVGA.com/VGA-timing>.

At the beginning of our frame there is the vertical sync, it accounts for a total of 2 lines. The other is horizontal synchronization pulse, accounting for 96 pixels.

**Note that the horizontal pulse must continue even during the vertical sync pulse, front, and back porch.**

The other structuring elements are the back and front porch. They can be found for both directions; the back porch is generated before the video data and the front after. In **yellow above are both the vertical back porch and the vertical front porch**, the first is 33 lines wide and the second 10. Finally, in orange are the horizontal back porch for 48 pixels and horizontal front porch for 16.

## Insight ‘zero’

From above literature we can conclude that, there are three key signals required to generate any sort of image on monitor.

*PWM Signals*

1. *H-SYNC*
2. *V-SYNC*
3. *DATA Analog Signal*

In order to generate PWM signals we need some type of counter or timer.

DATA signal is image in analog form. Hence, we need memory to store image and right mechanism which can push image data onto the data pin at right time.

Using microcontroller would be good idea as it meets the requirement, i.e. Timer and Memory.

# HARDWARE DESIGN

This section mainly deals with the appropriate selection of hardware components for the task at hand.

Displaying 640×480 image requires pixel clock of 25MHz as data needs to be pushed at every 0.04 µS. That means, we need digital circuit which could operate at that frequency range. A microcontroller relies on external crystal for providing high frequency clock to its CPU. NOTE: There is a limit on frequency for any particular microcontroller to execute safe and reliable operations. However, different microcontroller has different limits.

Atmega16 in particular have maximum operable frequency of 16MHz and has two PWM timers (to generate H-SYNC and V-SYNC), program memory of 16Kbytes (which could be used to store image data) and external interrupt support (to detect the edge of H-SYNC and send data out).

*REFERNCE VGA CABLE AND PIN DIAGRAM* <https://en.wikipedia.org/wiki/VGA_connector>

## Why AVR atmega16?

**AVR Microcontroller are based on RISC instruction set, i.e. most instructions are executed in one clock pulse.**

The main reason we opt for Atmega16 was because, it was already introduced to us in university curriculum.

So, we started experimenting on it and later realized that microcontroller (Atmega16) was bottleneck for the code as it doesn’t support CPU clock more than 16MHz, compared to required pixel clock of 25 MHz in order to display full 640×480 image. However, it was still capable of displaying some sort of monochromatic image and polychromatic image and that was just enough to solidify the learned concepts in more challenging and practical way.

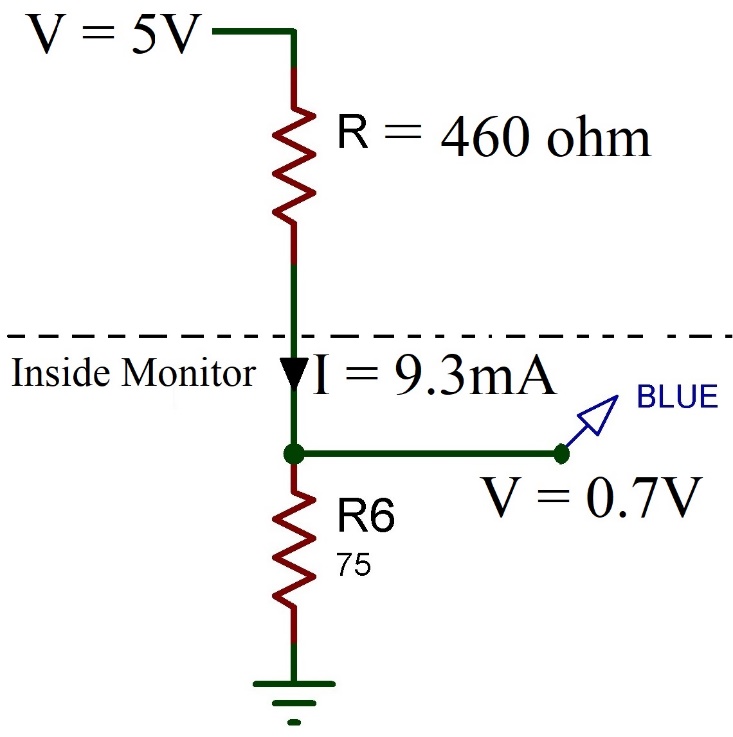
There could be many alternatives to Atmega16 but not in the Mega AVR series.

That would force us to shift to PIC or Xmega AVR series microcontroller or other microprocessors like ARM Cortex which can be clocked much higher than pixel clock itself and would also support DMA (Direct Memory Access) which would allow external device to access it’s memory, speeding up the pixel printing in process.

However, using those would be extravagant compare to simple task at hand.

**More on Atmega16 in APPEND A**

## 75Ω terminal impedance



Monitor have terminal resistance of 75 Ω, for maximum brightness it requires voltage of 0.7V.

0.7 V = I × 75Ω, I = 9.3mA.

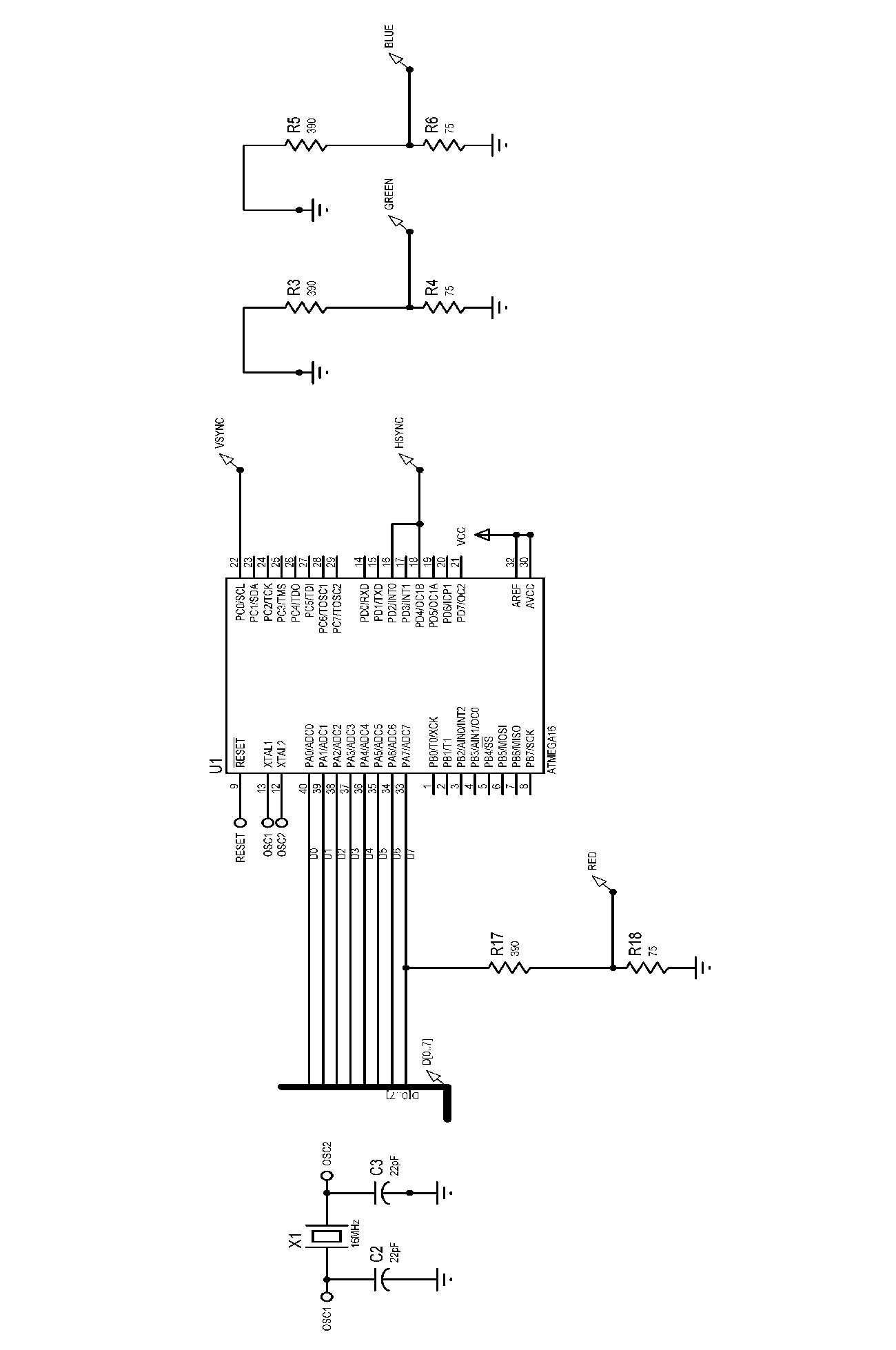
As output of our microcontroller is 5V. Hence, we need to add series resistance to drop voltage.

Using Voltage divider for maximum brightness,

0.7 V =

**R** turns out to be 460Ω.

## Circuit diagram for monochromatic image



## Limitations of atmega16 and alternatives.

Atmega16 have its own set of limitations, which are discussed below:

1. *Lower clock speed.*
2. *No DMA support.*
3. *Limited memory of 16Kbyte.*

Upon doing some basic research we found many alternative microcontrollers which could have realized this project in far more efficient and elegant way. Some of them are discussed in following section.

**ATxmega128A4U** is far more supreme when compared to Atmega16, as there are five 16-Bit timer/counters, 128Kbyte of program memory, DMA controller and can operate at maximum frequency of 32MHz which is 1.28 times of the required pixel clock itself.

**PIC32** is 32-bit microcontroller which could be clocked at 50MHz i.e. twice that of the pixel clock and also has Audio and Graphical interface. As it has two 32-Bit counter (clocked at 50MHz) can generate H-SYNC and V-SYNC with far greater resolution compared to Atmega series microcontroller.

**STM32** is another very powerful microcontroller based on same 32-bit ARM architecture. It can be clocked as high as 480MHz (H7 series). These microcontroller and other ARM based microprocessors are kind of overkill if task is just displaying image onto screen using VGA interface. Moreover, they far more capable when it comes to computer graphics i.e. displaying an image over HDMI interface or generating graphics upon the inputs from other peripheral devices like keyboard. Example, Raspberry pi 4.

# SOFTWARE DESIGN

This section documents how the code was sketched using AVR Studio 5 (IDE). The code was drafted with modular approach, i.e. each module was developed individually and independently to an extent. The following sections will discuss how problems were approached in each module and simulated graphs to support the obtained results.

## INTRODUCTION

Before we start pushing bits onto the data pin, we need to set up courser which points where data is to be printed. That is generating H-SYNC and V-SYNC with adequate timing w.r.t VGA standards. There are multiple modes for AVR timer, but best go would be using them in Fast PWM mode. The problem with other modes like CTC and Normal mode is, they can’t generate PWM signal without assistance of ALU i.e. to toggle a PIN or PORT.

This is problem because while processor is serving an interrupt it can’t render image or do other things.

Fast PWM mode generates PWM wave independent of the service provided by microprocessor.

We are using external crystal oscillator for Atmega16 which is calibrated at 16MHz.

**REFERENCE ON HOW TO UPLOAD SKETCH TO THE MICROCONTROLLER APPENDIX B**

## Generating H-SYNC

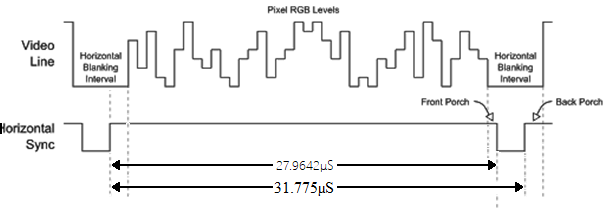
**H-SYNC is generated using Timer1 of Atmega16. Timing Computation in order to load appropriate bits into Timer/Counter Register.**

CPU frequency=16MHz, that means counter increases a count in TCNTx evert 1/16M seconds.

1/16M=0.0625µS.

0.0625µS × 65536 (max count in 16-bit counter) = 4096µS, that gives maximum possible delay that could be generated using this timer. 4 mS is more than total width of H-SYNC so we not need to pre scale the clock input.

Length of H-SYNC excluding SYNC pulse (3.8133 µS) is 27.9642µS out of 31.777µS.



That means we need to pull the pin low after 27.9642µS since the start of H-SYNC.

27.9642µS ÷ 0.0625µS = 447.4272 decimal count, 447 to hex system is 0x01BF.

Hence, we need to load 0x01BF to the compare register.

Then pin is again toggled after 3.8133µS at the end of the H-SYNC.

Hence load the top value of the counter as 0x01FB, As 31.7775µS ÷ 0.0625µS = 508.4 decimal.

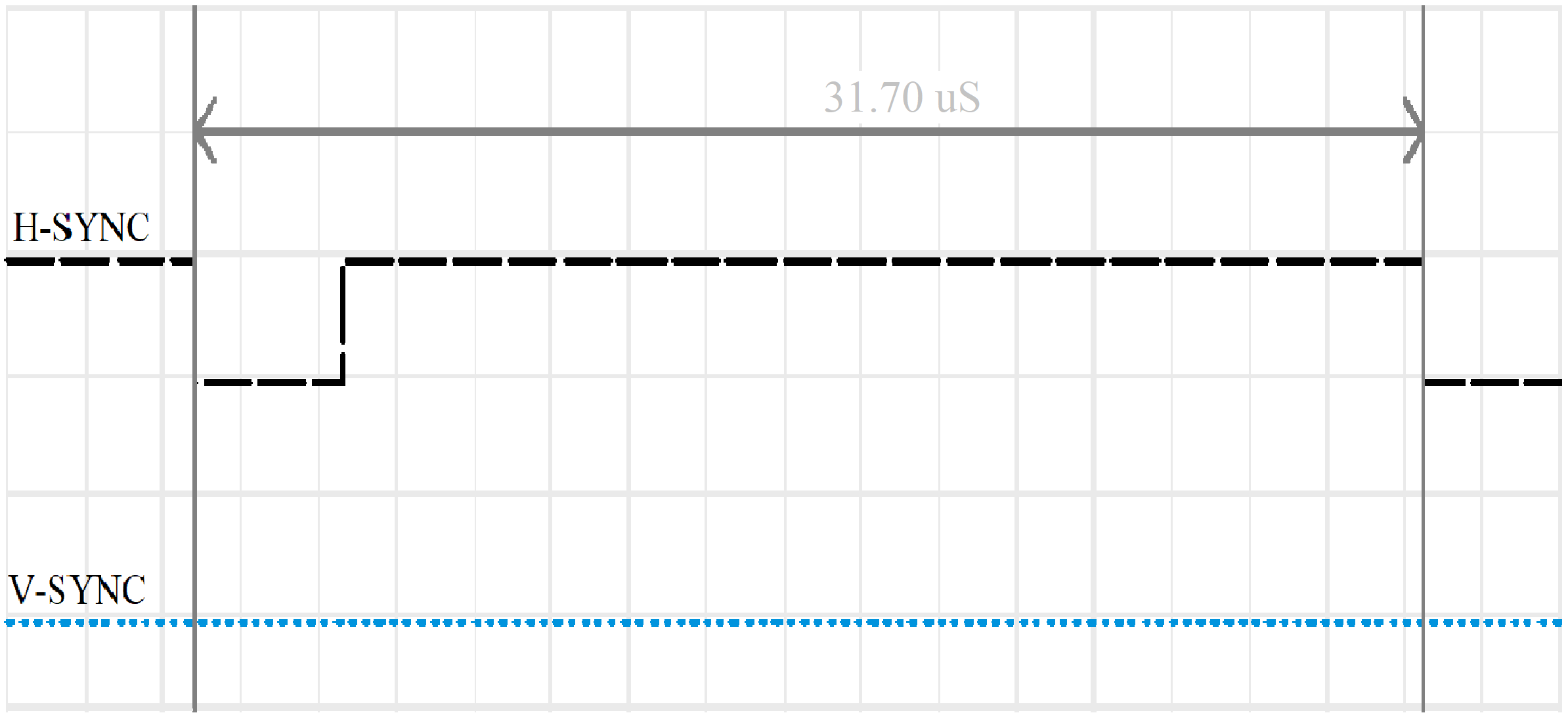
##### Counter DIAGRAM

##### OUTPUT

Code Reference: APPENDIX G

NOTE: The image below is output of the simulated AVR code on Proteus 8.

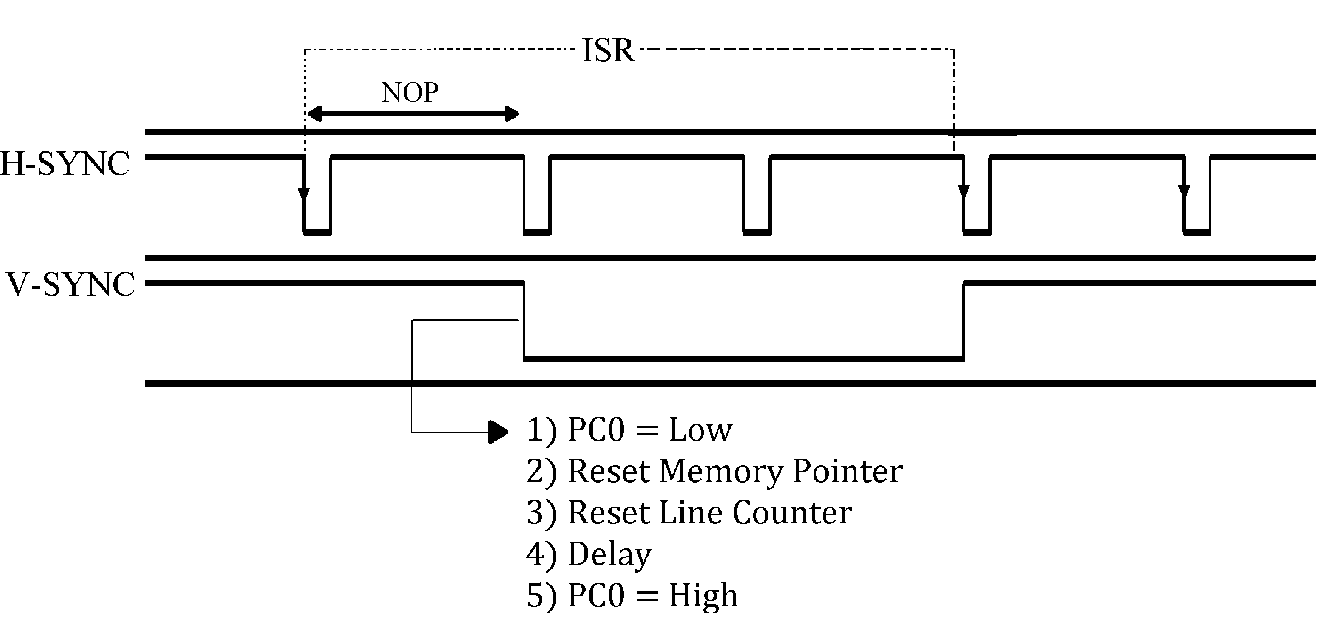
The timings are not accurate compared to VGA standards but are adequate enough to produce an almost stable image.



## GENERATING V-SYNC

H-SYNC (PWM signal) is looped back to interrupt pin of microcontroller, an interrupt is triggered at every falling edge and program counter is redirected to vector location of ISR (interrupt service routine).

We already know that there are 523 lines (out of 525 lines in total) between two consecutive V-SYNC, as two coincide the SYNC pulse of the V-SYNC.



1st

2nd

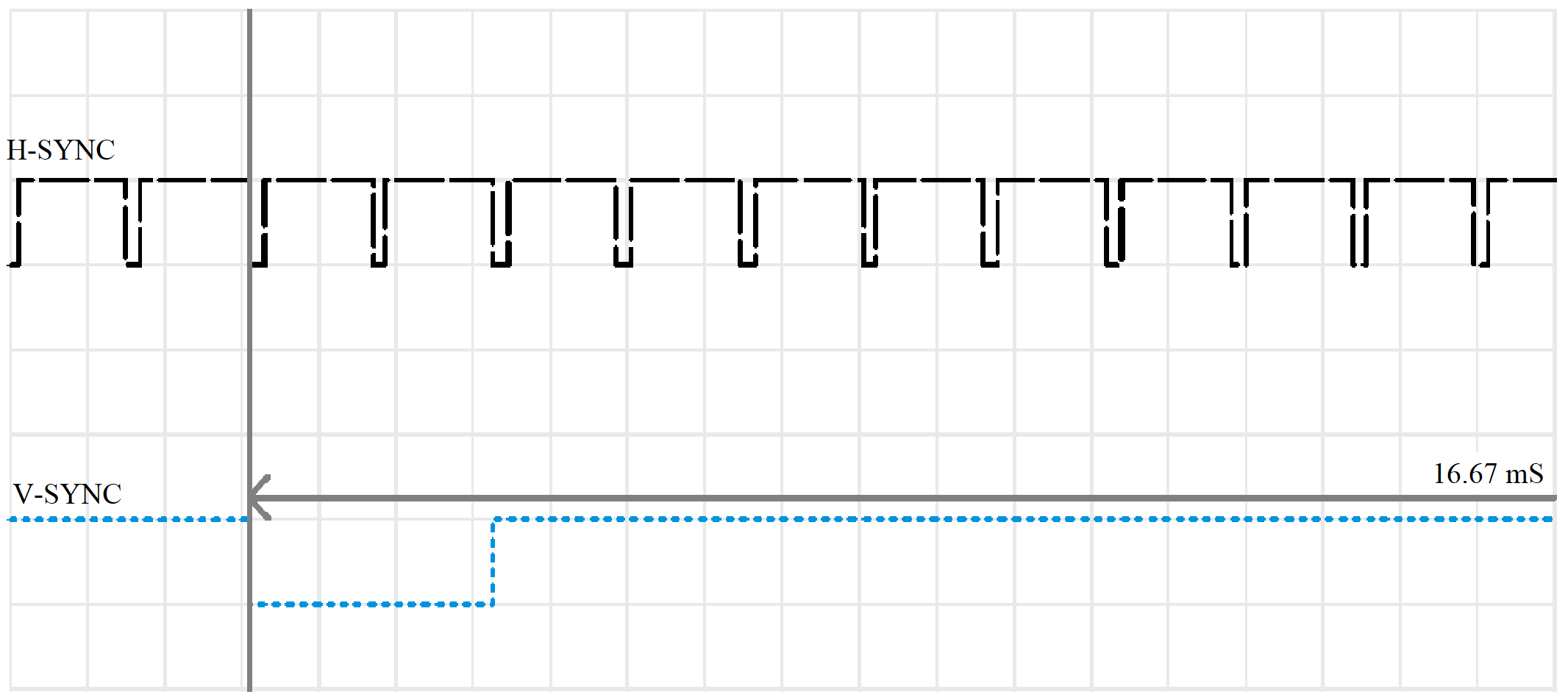
523rd

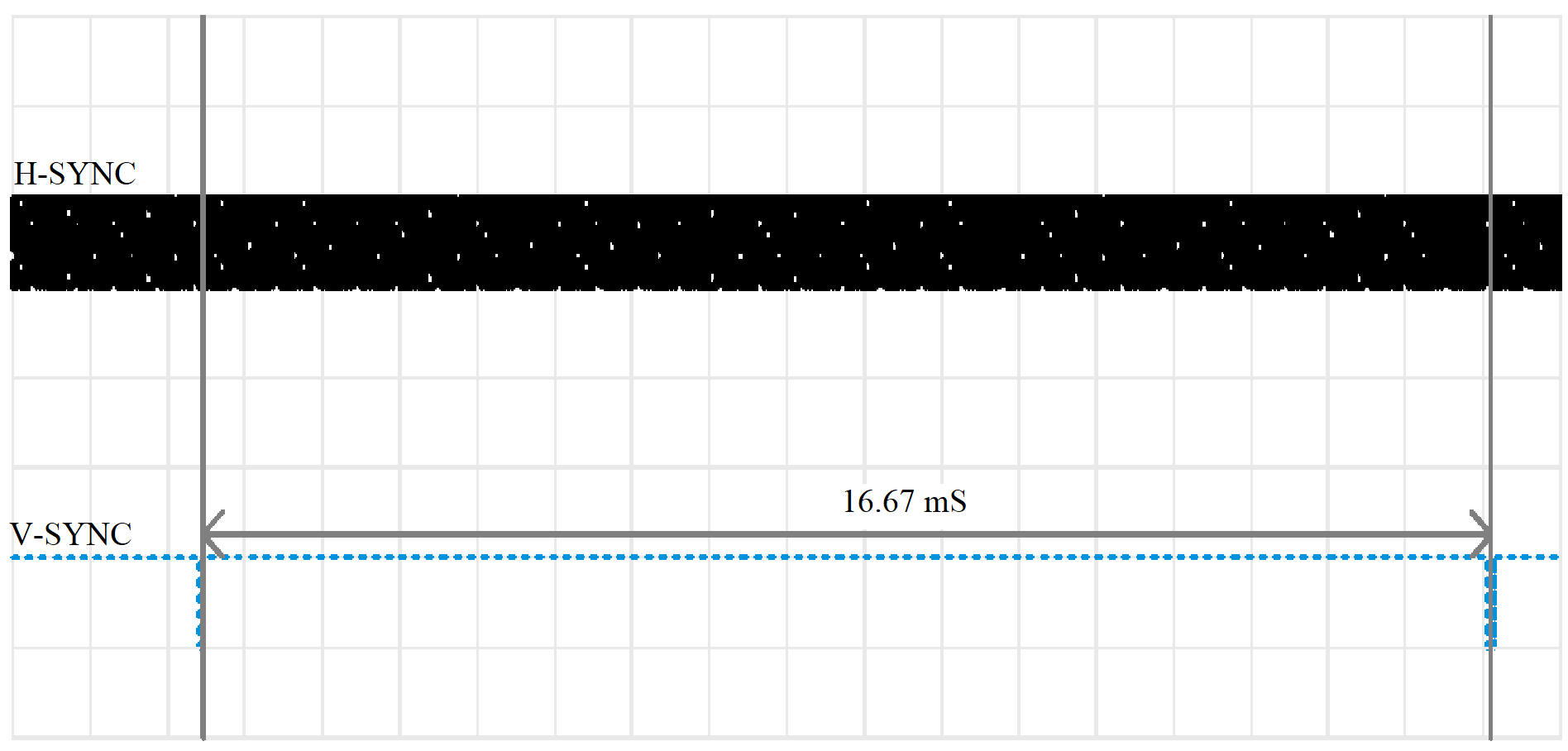
So, we count 523 H-SYNC pulses and then generate the SYNC pulse of V-SYNC by toggling the PIN PC0. Toggling using instruction takes (0.0625µS ×2) time, which is negligible time frame compared to the time period of V-SYNC (in mS).

Toggling using instruction could have been a problem in case of H-SYNC as we are already dealing in terms of µS.

CODE REFERNCE: APPENDIX G

##### OUTPUT



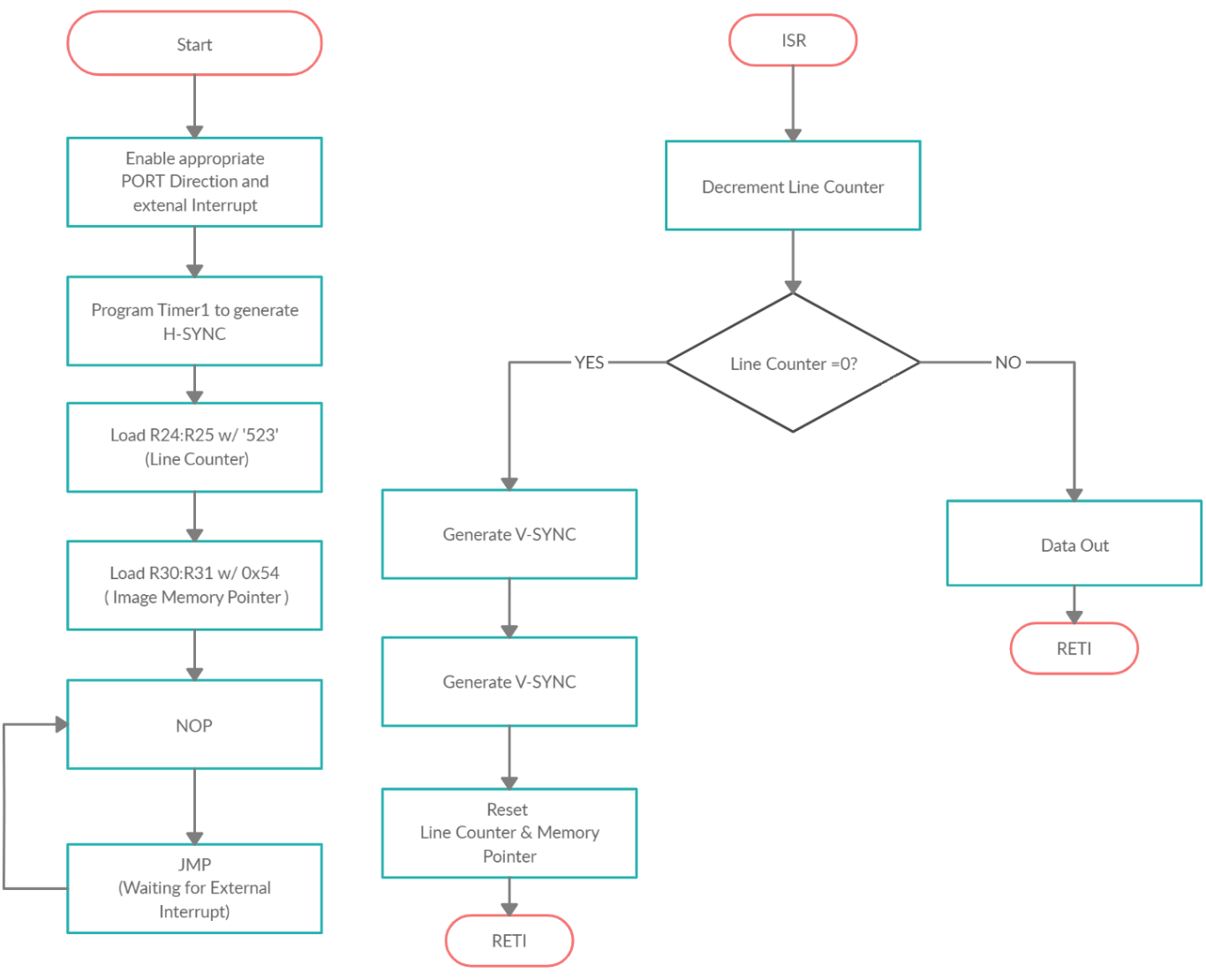


##### VERIFICATION

How to check if the timings of the generated V-SYNC and H-SYNC are correct or are within the error tolerance margin?

When we feed these generated signals into the monitor, we could see it comes out of the standby mode, then exploring the menu of monitor can display the current resolution which monitor thinks it is in.

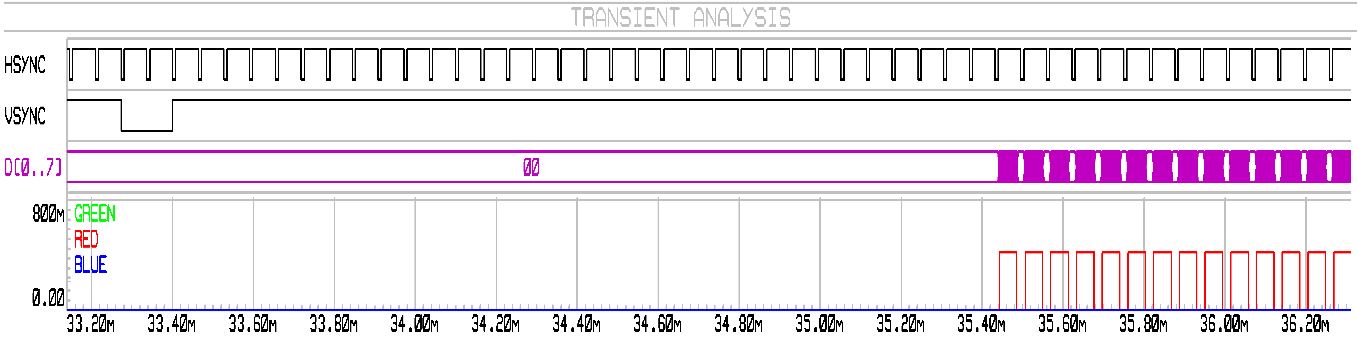
## PROGRAM FLOW



CODE REFERENCE VOID MAIN (): APPENDIX G

## DATA OUT

There are 525 lines in each frame, out of which 480 are visible. Our objective now is to send data on to pin 1 (RED) within the visible period of H-SYNC.



First 33 lines should have no data as they are part of back porch (V-SYNC). Therefore, we need to load 0x00 in first 33 × 20 (Bytes per line) memory spaces.

Atmega16 have SRAM of 1 Kbytes which not enough to store an image of resolution 160×480 holding 20 (bytes per line) × 525 bytes i.e.10.5KB, so it was convenient to store image in program space.

On side note: If there exists a controller where SRAM is more than Program Space or in case of using external SRAM, reading data from *program memory* takes 3 clock cycles (LPM) whereas reading data from *data space* (SRAM) takes 2 clock cycles (LDS). This would not make much difference (of 9 pixels) in the resolution of the image.

We can also manipulate program space just like data space with different instructions taking more time as, Program space is ROM where as Data space is SRAM.

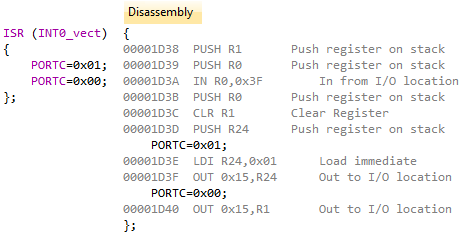
AVR-Lib provides an attribute called PROGMEM to declare array in program space, **more on that in APPENDIX F**

However, sending data is time critical operation which is not preferred to code in C for two reasons

1. Slower execution because of overhead instruction which is again because of poor code optimization by IDE.
2. Size of the code will be more than assembly counterpart.

*“C is the language of choice and assembly language is included in situations where it is either necessary or convenient.”*

For time or space critical applications, it can often be desirable to combine C code (for easy maintenance) and assembly code (for maximal speed or minimal code size) together.



1 Clock Pulse

2 Clock Pulse

6 Clock Pulse

Writing simple code for toggling a Pin on PORT C take about 9 Clock cycle which is 0.0625 × 9 = 0.56 uS.

Whereas same code would take only three clock cycles in assembly. i.e. 0.1875 uS, 1/3 of the C counterpart.

**Combing assembly and C in AVR studio are discussed in APPENDIX C.**

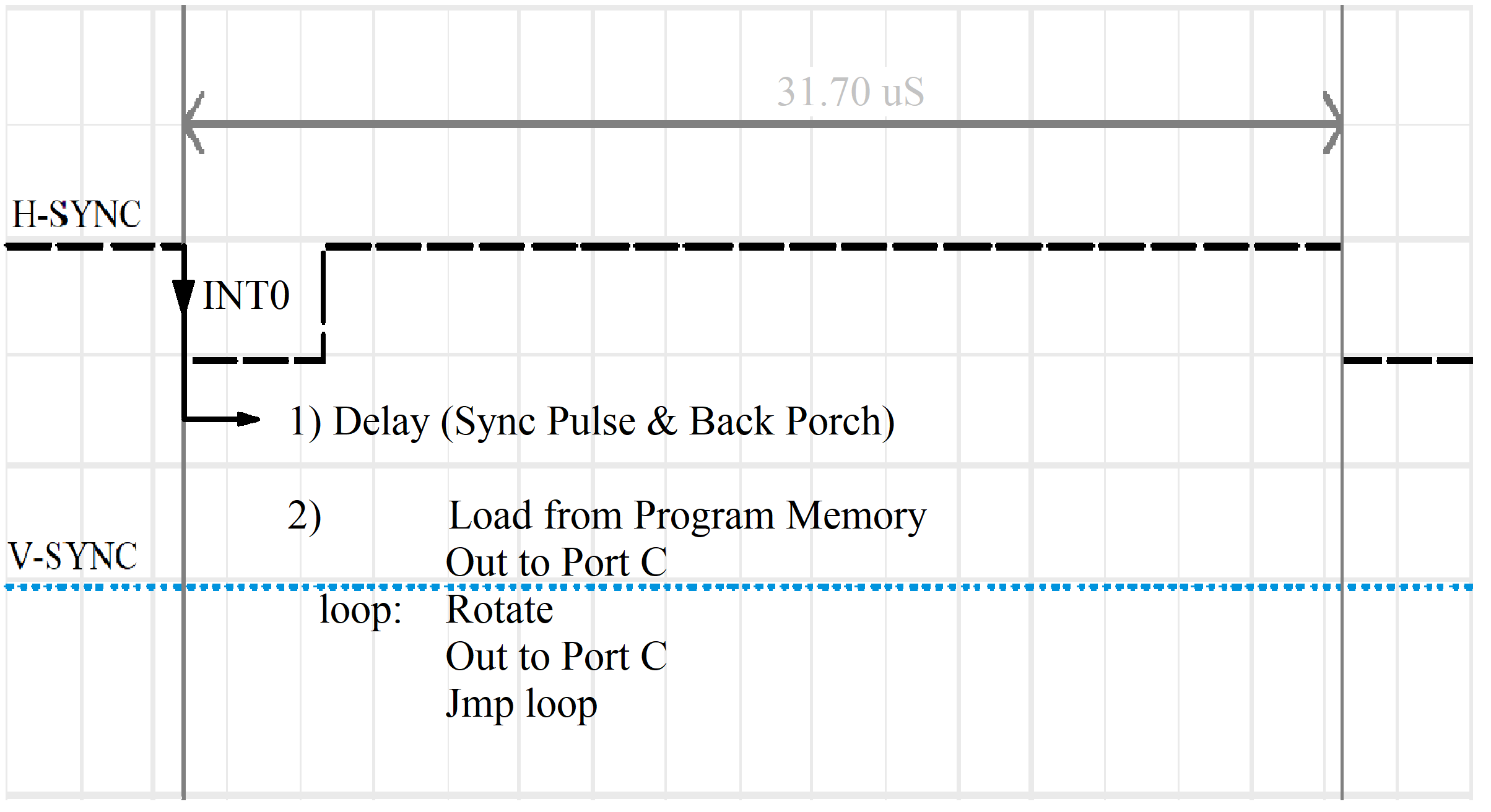
## DATA OUT USING SHIF LOGIC (Monochromatic Image)

Upon the falling edge of H-SYNC, an interrupt is triggered which routes the program counter to vector location of ISR.

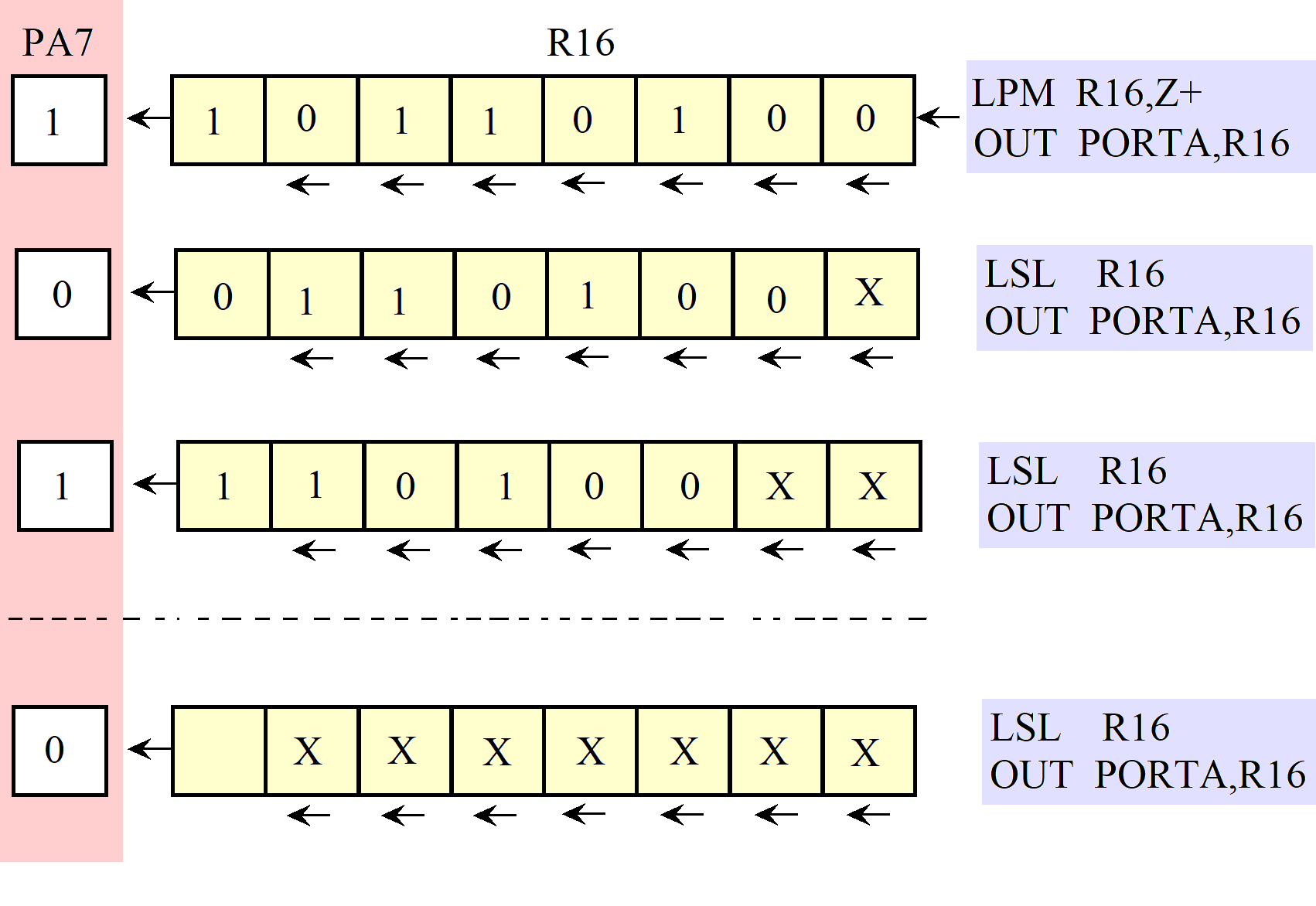
Upon entering the ISR the program checks, if the line count = 523. If yes then, it generates V-SYNC else it jumps to data macro.

##### DATA MACRO

The byte is loaded into the temporary register from program space (where image is stored) using LPM instruction.



34th



**CODE REFERENCE: APPENDIX G**

## GRAPH - MONOCHROMATIC IMAGE

# IMAGE PROCESSING

In digital raster image, pixel is the smallest accessible element, also known as picture element.

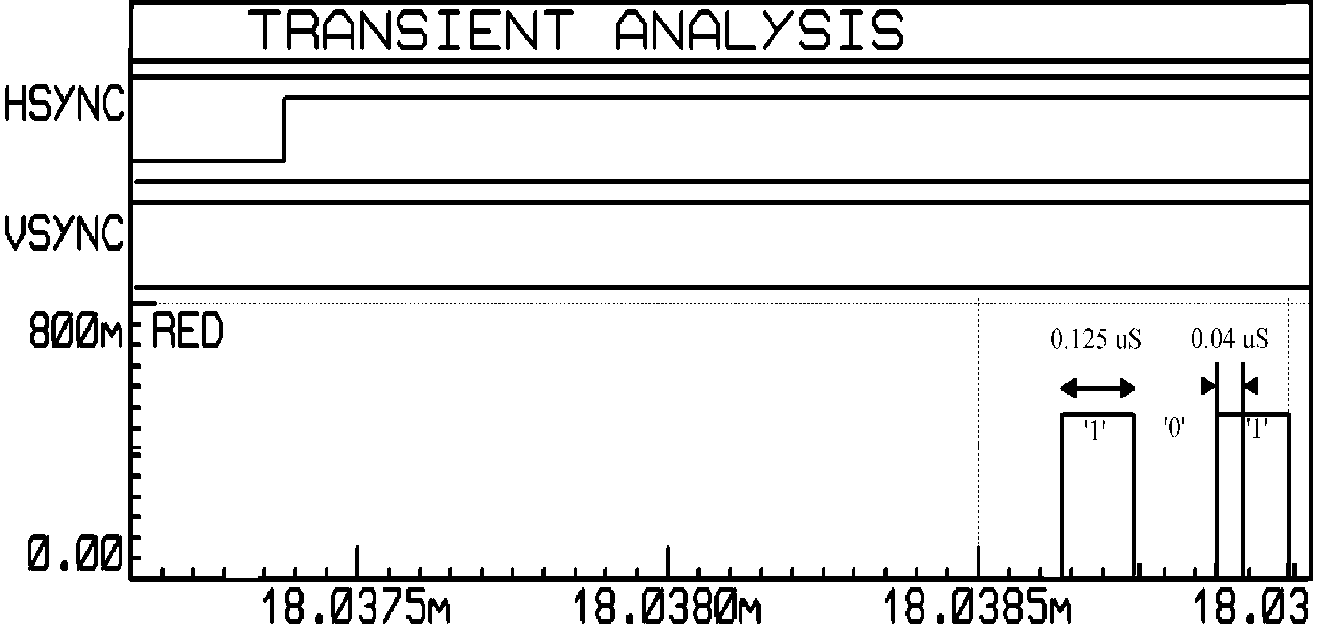
**What is shape of the pixel?**

Graphically it has to be square, generally as the image data is processed based on that. Physically it doesn’t need to be square, it is sometimes diamond or dot or even a line.

**Why this is important?**

The pixel clock required for 640x480 resolution is 25MHz. i.e. it expects pixel every 0.04µS.

On the other end, the hardware employed to do this task is sending out data at much slower rate. So single value of the pixel by Atmega16 is reprinted several times.



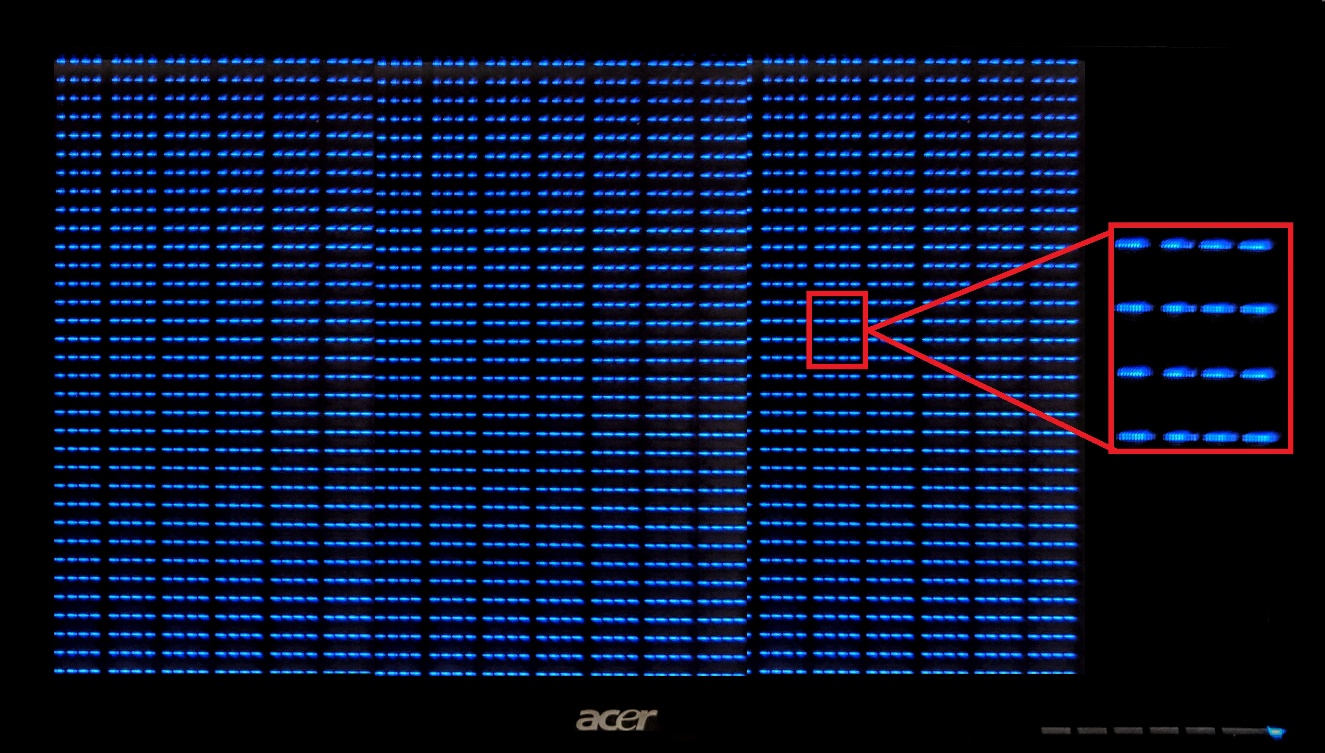
As you can see from the image, the time taken by Atmega16 to write a single bit is 0.249 µS. Whereas the pixel clock is of 0.04 µS.

Hence the number pixels in 25.422µS (Visible time of H-SYNC) should be 230 (25.422÷0.125) but actual number is quite different, 120.

It takes about 3 clock cycles to load image data into general purpose register and another one to send it out. So, data throughout rate is not continuous as LPM is used after every byte is transferred. Hence the actual number of pixels are different than ideal case.

One might ask why can’t it transfer the byte directly to the port without loading it first into the accumulator, saving time in process and improving resolution to great extent?

The answer is that Atmega16 doesn’t support direct memory access nor it have direct memory addressing instruction, i.e. it doesn’t allow any port to read/write contents of its memory. So, its firsts load memory content into the general-purpose register or accumulator before sending it out to any port or peripheral. Due to this time consumption, the monitor prints the received pixel (from atmeag16) 0.125 µS/0.04 µS = 3.125 times.



NOTE: If its printing each pixel three times, we can’t just print it 3 times vertically in order to make pixel square as the speed of printing pixels horizontally is different than speed at which pixels are being printed vertically.

As we know that graphical pixels need to be square hence this stretched pixel problem needs to be tackled.

There are two ways which I can think of to solve this problem.

1. Send data at frequency of 25.7MHz i.e. pixel clock

2. Making pixel square by trading the resolution.

Sending data at faster rate would require faster calibrated CPU clock of 25MHz, but maximum frequency supported by Atmega16 is 16MHz. So, to make this feasible we might need better microcontroller.

The second method is quite simple and feasible, **Calculating pixel dimensions.**

It takes monitor 15.253 mS to print the frame, that is 480 pixels vertically

15.253 mS ÷ 480 = 31.7775 µS, which is equal to time period of H-SYNC as it should be.

31.7775 µS to print 640 pixel, 31.7775 µS ÷ 640 = 0.0467 µS (pixel clock)

Times each pixel is being repeated 0.125 (duration of each pixel in prototype) ÷ 0.0467, which is equal to 3.125 times.

It takes 25.422 µS (visible time of H-SYNC) to print 160 pixels, 25.422 ÷ 160 = **0.15888** average time to print each pixel in prototype, 0.15888 ÷ 0.04 = **3.97 time each pixel is being repeated.**

Multiplying it by ‘x’ factor so that the width of pixel is equal to the length, **W × X = L**, X turns out to be i.e. **Stacking pixels upon each other approx. 8 times would result a square pixel.**

**How this would be trade for resolution?**

Repeating each pixel 6 times would reduce the vertical 480 pixels to actual 80 (480÷6).

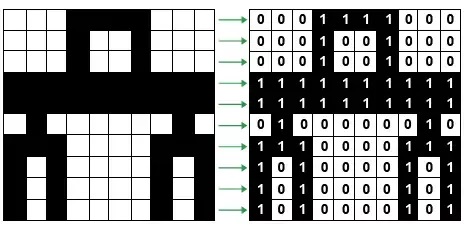
Resolution now stand at 160x80, that is aspect ratio of 2:1.



# monochromatic image to hex

Loading data into program memory is simple but how to load an image? First, we need to extract the image data from the actual image.

A pixel or picture element have a corresponds to a binary equivalent value. but they are not mapped one to one in the 2D array or matrix, rather the bitmap goes through the compression algorithm which changes the bits in a pattern which occupies less space and retains almost the same quality.



Before we load image, we need to get rid redundant bit (redundant in our case as we need raw or bitmap image).

We could have written a MATLAB to extract the image data but found this gem on GitHub

Reference: <https://github.com/riuson/lcd-image-converter>

As the image we are going to display is monochrome, we need to export images which only have colour pallet size of two. (if image is not monochrome then perform grayscale operation)

This application coverts the image into the binary or hex equivalent which is what atmega16 would understand.

As we need each line to repeat 8 times, we can’t directly use LCD converter. Pre loading the image into paint and using the resize option wisely could get around this problem.

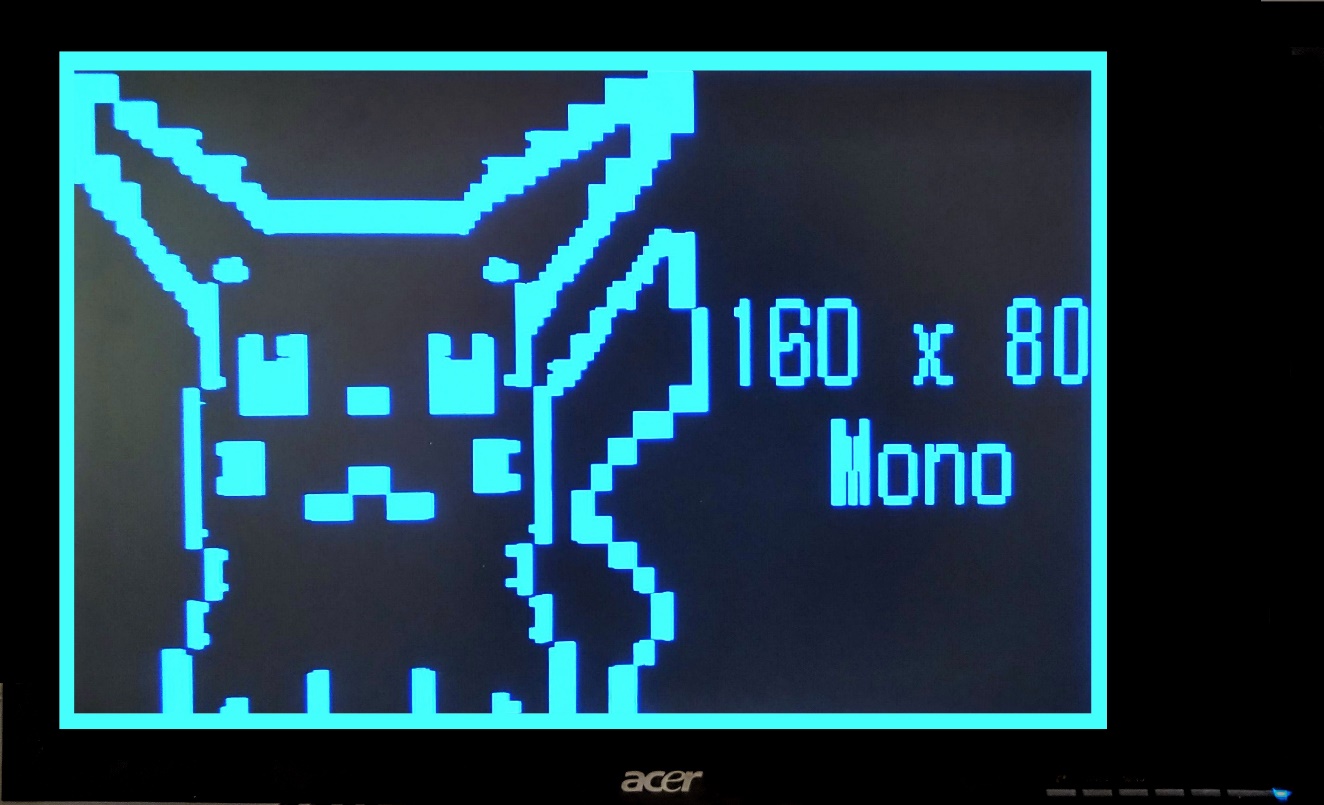
Steps:

1.load the image convert it into 160x80 resolution.

2.again resize the image into 160x480 resolution, disable the maintain aspect ratio button.

3.save and drop the image into the LCD converter, which will spit out the hex equivalent image with resolution 160x80 which have array of 160x480 bytes.

# IMAGE FROM MONITOR



# INSIGHT 1

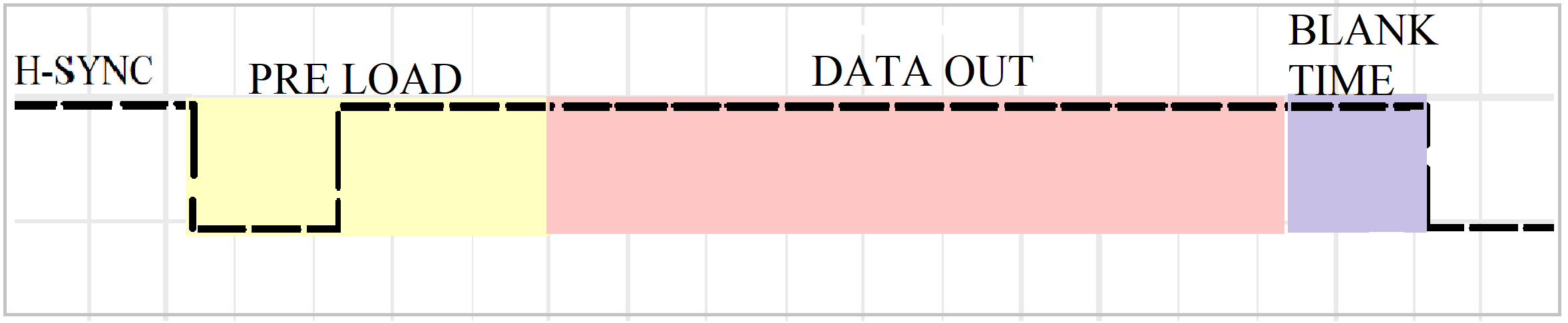
##### HYPOTHESIS ON PuSHING DATA FROM MICROCONTROLLER

Now, there are several ways to send out data to pin 1, 2 or 3 of the DB-15 connectors.

1. *Pre-Load data during SYNC pulse*
2. *Using EEPROM*
3. *DMA (Direct Memory Access)*
4. *SPI*
5. *Program Memory Access*

## PRELOADING DATA DURING THE SYNC PULSE & Back Porch

**Pre-loading the data into general purpose register and using single instruction OUT followed by parallel to serial data conversion.**



The idea is to load the image data into general purpose registers during the SYNC pulse of H-SYNC, as processer is idle.

SYNC pulse of H-SYNC is of duration 3.8µS which have 3.8 µS ÷ 0.125 µS instructions. Which leaves room for 30 instructions and that is 30 bytes, thanks to RISC. But we will leave two registers, as to use them as memory pointer. Upon the rising edge of the H-SYNC we could trigger an interrupt.

INT0\_vect: OUT 0x15,R16 //Send Byte 1 to PORT C

delay //Wait for Parallel to Serial converter

OUT 0x15,R17 //Byte 2

delay

OUT 0x15,R18 //Byte 3

delay

OUT 0x15,R19 // byte 4

…

…

OUT 0x15,R27 //byte 12

RETI

NOTE: This code is designed for displaying only monochromatic image.

##### Operation

Here the OUT instruction sends the image data in byte, to a port C. This is not how monitor expects data, as it has only one data pin as input per colour. And as our image is monochromatic, we only need one pin.

Hence all these 8 parallel bits needs to be converted into 8 serial bit steam. The capacity (which will eventually decide the resolution) of this setup largely depends upon the external hardware as how fast it converts the parallel data into serial bit stream.

**74LV165A IC** 8-bit parallel-in/serial-out shift register could be used.

REFERENCE: <https://assets.nexperia.com/documents/data-sheet/74LV165A.pdf>

## Using EEPROM

Using EEPROM to store image is like using hard drive for storing image, which is a good idea but not so good when we need to access it at faster data rate. Also, that’s why computer have RAMs.

Limitations of using an EEPROM

1. The internal EEPROM of Atmega16 is of 512 bytes which is limiting factor if we wish to produce .gif images on the screen, typical .gif of 3 sec would have 6-10 frames which would require storage far more than just 512bytes.
2. It takes about 1ms to read data from EEPROM which also require 4 instruction operation on EEPROM registers.
3. There is limit on how many times we can read or write data onto the EEPROM i.e. AVR ATmega16 EEPROM has endurance of 100,000 write/erase cycle.

**Hence the idea of using internal EEPROM was scratched out.**

What about external EEPROM?

There are few good reasons not to use external EEPROM.

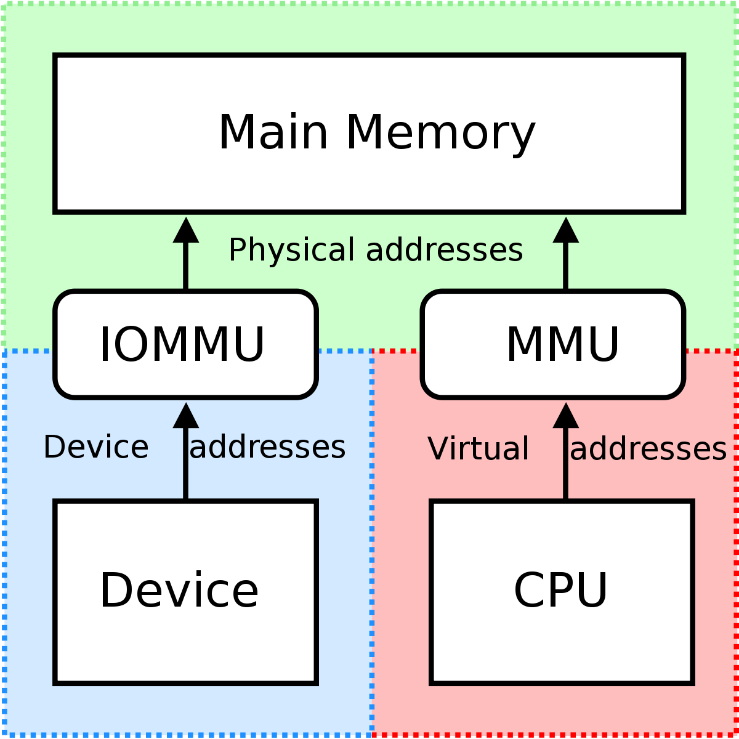
1.To minimize the cost factor

2.Why use external memory when we have program memory of 16kb?

**What would be the technical aspect of program if we were supposed to use external EEPROM?**

Configuring Timer1 and Timer2 in FAST PWM Mode, that leaves CPU free most of the time. This idle time could be used to project the real time timer count to any given PORT. The timer count of timer2 or timer1 states the real time courser location of the monitor and works as address. This address could then feed onto the EEPROM and data could be accessed accordingly, given that the image is stored at the counter pointing location. Which is to be taken care when programming an EEPROM.

## DMA (DIRECT MEMORY ACCESS)

As you can see in the figure, using DMA controller we can have access to main memory of microcontroller which generally have reserved access for processor only.

Unlike Atmega series (Microcont-roller), Microprocessors have DMA. i.e. A feature of computer systems that allows certain hardware subsystems to access main system memory, independent of the central processing unit. In contracts to the function of Atmega16, before data is send out to the port it needs to be loaded in the accumulator or any general-purpose register.

This would take over head instructions in terms of time, as valuable resource as timings are critical for VGA signal.

Atmega16 doesn’t have DMA, but switching to different microcontroller like newer PIC or ARM microprocessor would improve the resolution to greater extend.

There are two reasons for not switching to ARM

1) It is lot fancier than likes of Atmega series which would defy the whole purpose of building video card with underdog and cheap components.

2) I have never worked with ARM processor before so it will have its own learning curve and implementation phase, which might take another one month s

Reference links: [https://community.arduboy.com/t/wip-VGA-out-without-fpga/6612](https://community.arduboy.com/t/wip-vga-out-without-fpga/6612)

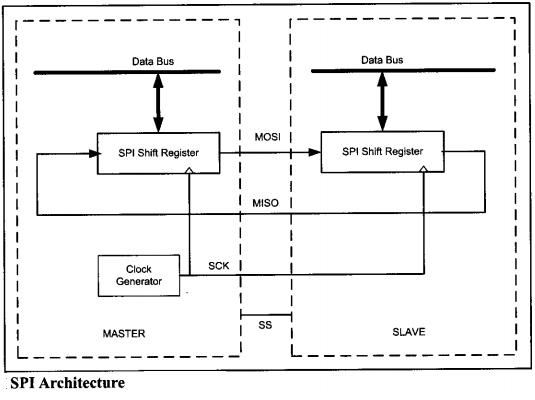
<https://www.st.com/resource/en/datasheet/stm32f103c8.pdf>

## SPI (SERIAL PERIPHERAL INTERFACE)

SPI consist of two shift registers, one on the Master Side and other on the Slave Side.

In SPI, the shift registers are 8 bits long. it means that after 8 clock pulses the contents of one reg is transferred to another. When master wants to send a byte, it places it in the shift register and generates 8 clock pulses.

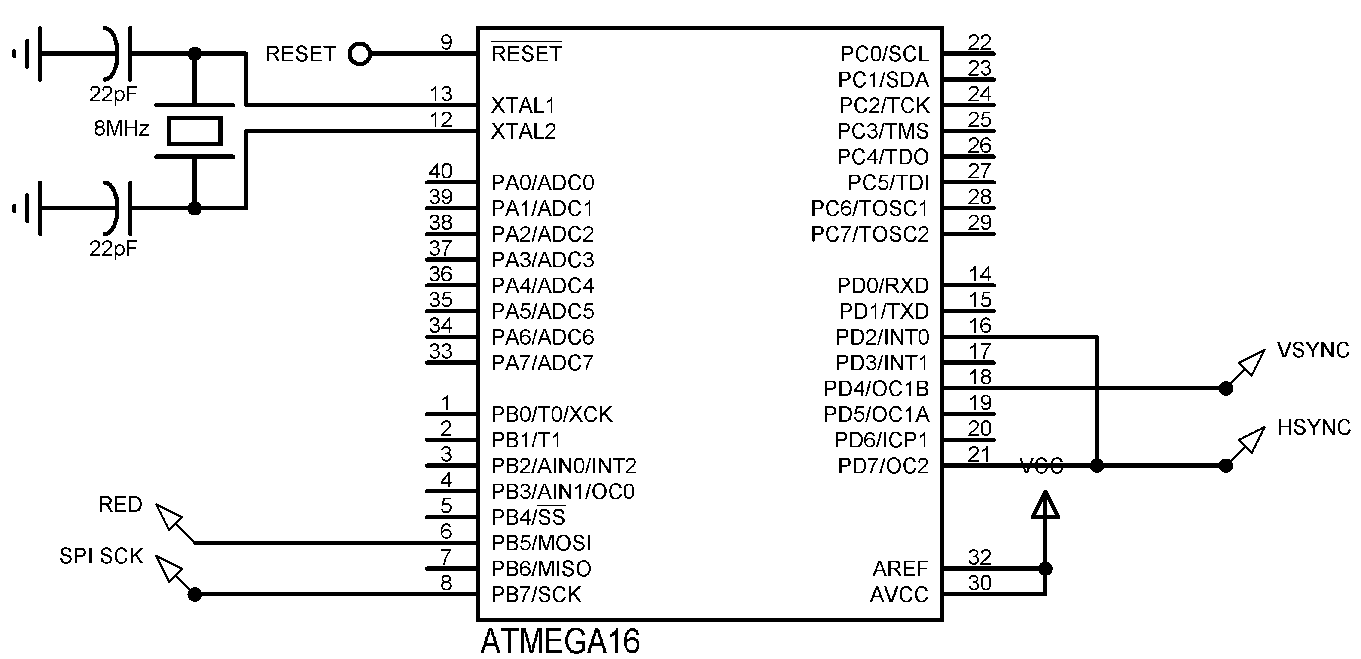
After 8 pulses the byte is transmitted out thought the MOSI pin to the SPI shift register on slave side. We are going to use MOSI pin as output with little variation, instead of shift register on other side we have our data pin of DB-15. MOSI spits out the image byte, bit by bit.



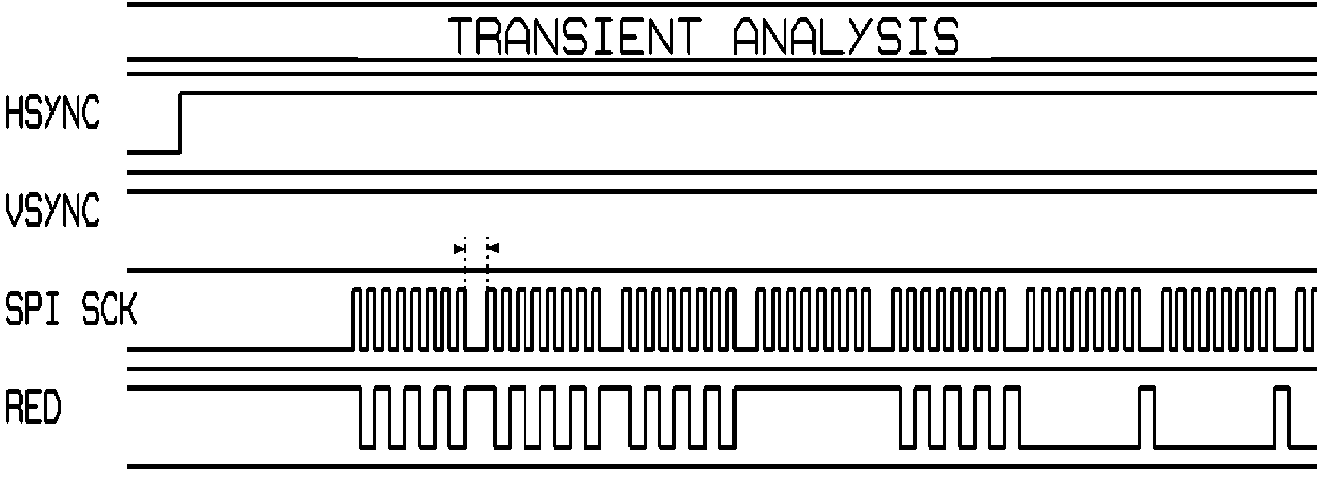
Registers involved in SPI communication

1. *SPSR SPI Status Register*
2. *SPCR SPI Control Register*
3. *SPDR the SPI Data Register*

##### CIRCUIT DIAGRAM



##### OUTPUT



NOTE: Memory Array[5000] ={0x00};

The first method we tried was sending data using SPI (Serial Peripheral Interface) lines, it sure does have its own short comings which were quite visible on the screen as 0.5cm wide blank lines but it left us with many ideas and lessons which eventually pavemented the path for higher resolution image/8-bit colour image.

**More on SPI in APPENDIX E**

**Problems with SPI**

After completion of byte transfer, the data line is set HIGH by microcontroller which results in garbage pixels (“1”) being printed other than the image data.

This problem could be tackled using a NOT GATE and complementing all the data stored in the microcontroller memory.

This way NOT GATE will set data line LOW after the 8-clock pulse, printing black pixels on the screen which should not be problem.

However, this blanking time will produce discontinuous image.

In order to tackle this newly created problem we can use a D-flip flop to latch the last bit of every byte, this would eliminate the blank gaps or discontinuity in the image.

However, this will result in stretching the last bit or pixel compared to the rest of the pixels, hence the whole images will appear it be stretched out horizontally.

“Sometimes you WIN, sometimes you LEARN.”

The way which SPI transfers the data serially from its 8-bit register can be replicated using a simple code with any register.

Example:

Load byte into a register, ***say R16.***

Using shift instruction to shift the bit form L to R or **R to L**, say ***ROL 16.***

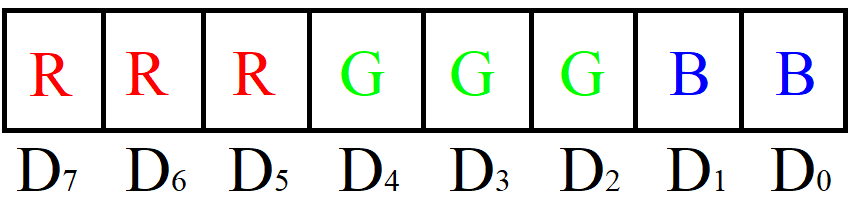
Then sending the contents of r16 to a particular port where the data out pin is connected at fixed point, use ***OUT 0x15, R16.***

# Hardware Design (Polychromatic Image)

## R-2r Resistor ladder network

The output from the microcontroller is digital signal, i.e. it’s either ‘0’ (0V) or ‘1’ (5V). Monochromatic image can be easily displayed using this binary logic as there are only two possible colours in the image, but in order to display colour or grey scale image we need analog values between 0V and 0.7V.

A typical 8-bit image requires one byte per pixel to store its colour information. First three bits is for Red then three for Green and remaining tow for Blue.

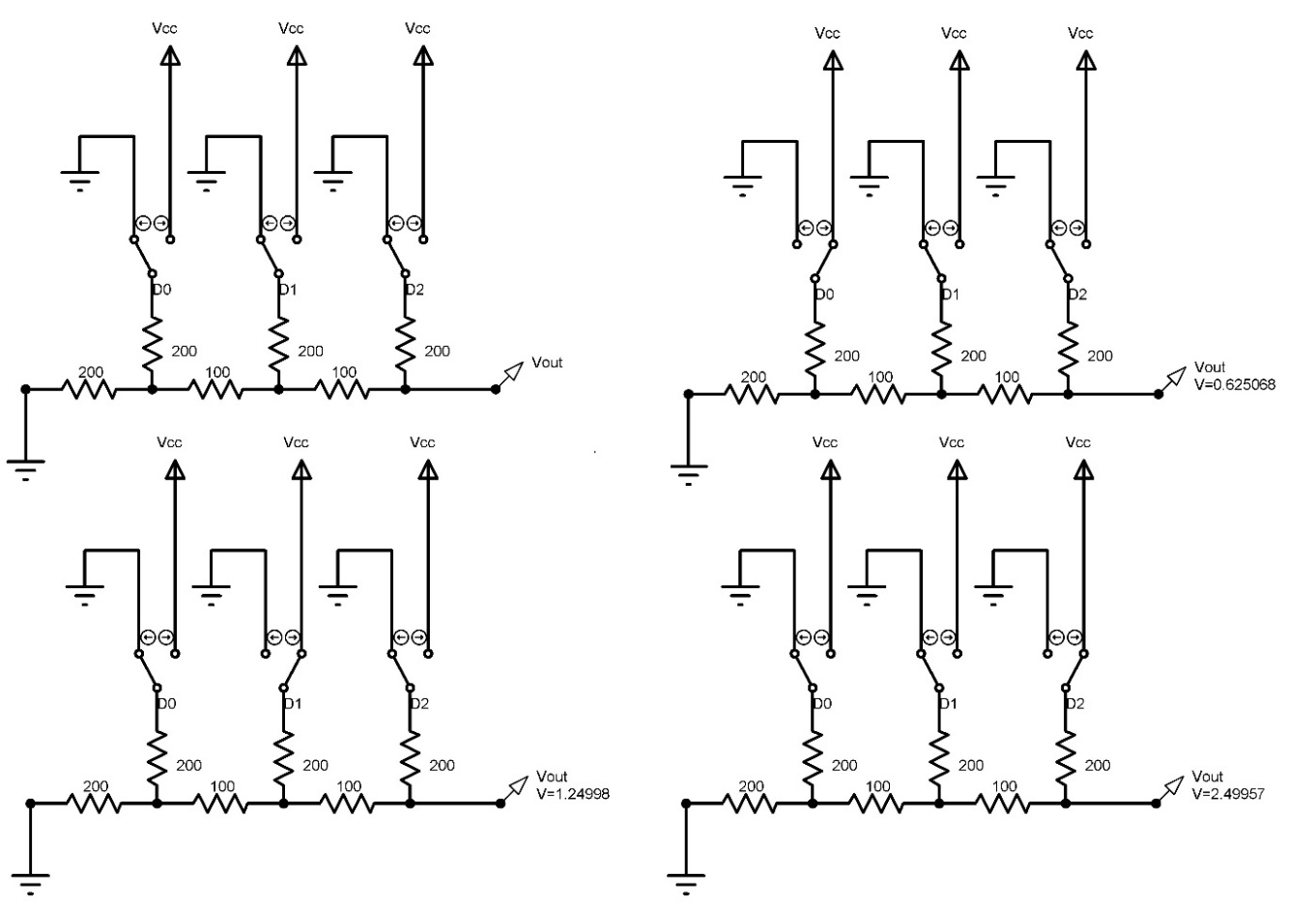
Three bits for Red and Green gives us 23 combination of colours. i.e. 8 possible shades of Red/Green, from Black (000) to bright RED or GREEN (111). Blue is assigned only two bits (four possible states) as human eyes are less sensitive to blue colour.

NOTE: Standard 8-bit image follows this order of *RRRGGGBB*. However, any order could have been realised on same hardware but this order of RGB in particular generates image which is more natural compared to others and this is why it is standard format.

Space occupied by 8-bit image is, eight times that of same monochromatic image as each bit represents a pixel in monochromatic image compared to a Byte in 8-bit image.

AVR microcontroller could send these 8-bits to any defined PORT in a single instruction. In order to convert these binary 8-bits in voltage levels ranging from 0V to 0.7V (analog), we need to employ DAC.

Instead of using DAC IC per each colour, we could simply implement R-2R DAC architecture which is resistor network which takes in binary input and spits out analog values accordingly with equal step size.



NOTE: All the resistance values are either 100Ω (R) or 200Ω (2R), hence this network would function the same with any values of resistance as long as they are multiple of R and 2R. However, the output current is low for the network with equivalent high resistance. So, 00it is wise to use low value of resistors for higher current output.

|  |  |  |  |
| --- | --- | --- | --- |
| D2 | D1 | D0 | VOUT |
| 0 | 0 | 0 | 0 V |
| 0 | 0 | 1 | 0.625 V |
| 0 | 1 | 0 | 1.25 V |
| 0 | 1 | 1 | 1.875 V |
| 1 | 0 | 0 | 2.5 V |
| 1 | 0 | 1 | 3.125 V |
| 1 | 1 | 0 | 3.75 V |
| 1 | 1 | 1 | 4.375 V |

This table is computed using basic network theory concepts (Thevenin’s Theorem in particular).

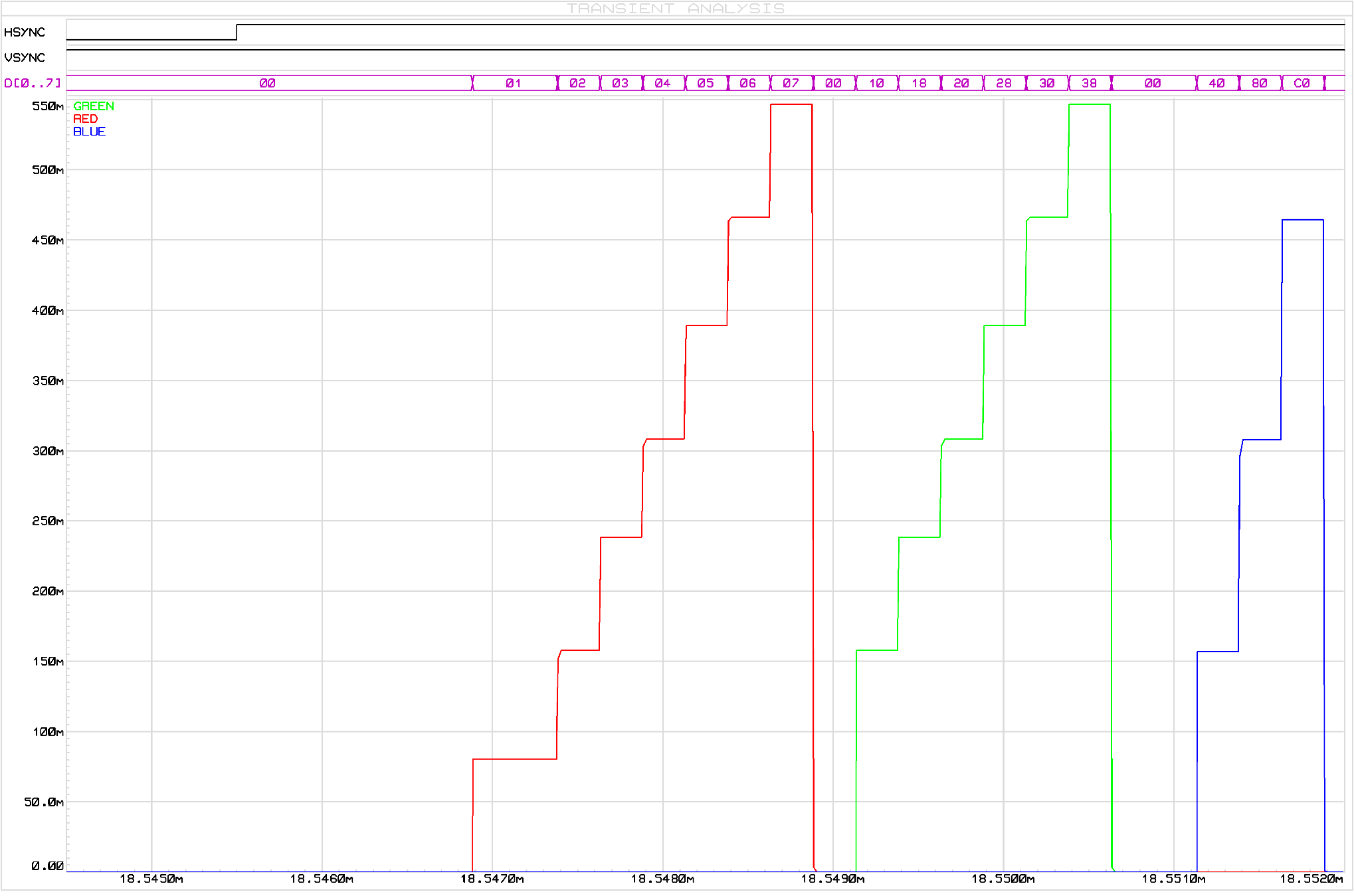
Step size, **∆V = V**ref **+ ∆V** = 5 V +  **=** 0.625 V

Maximum output voltage, **Vo = Vref ×**

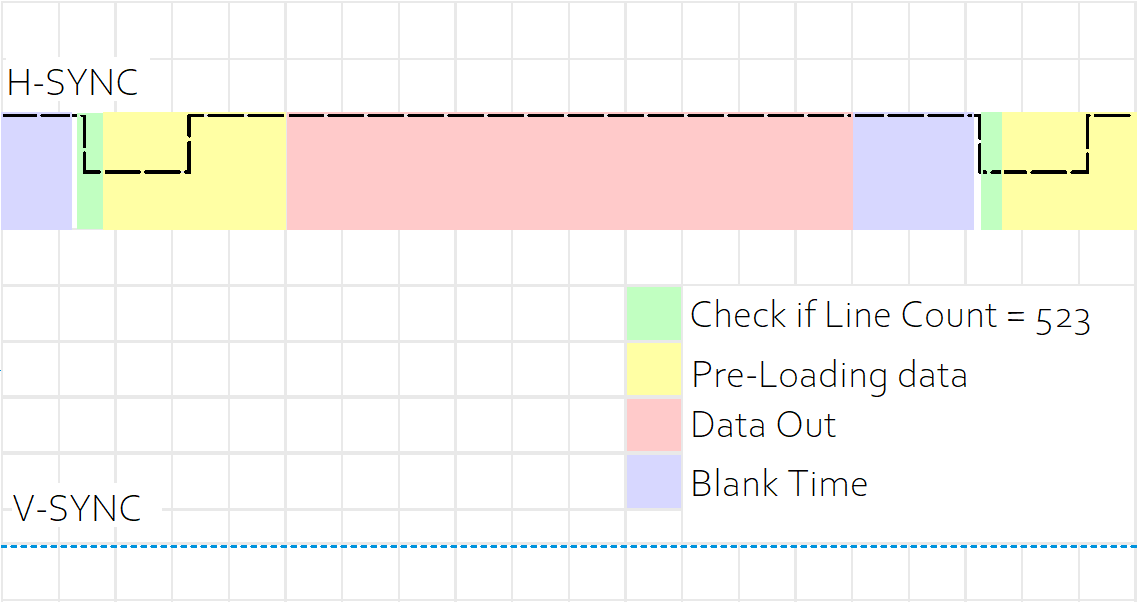
**Vo = 5 V × = 4.375**, which is

also equal to 5V – 0.625 (for ‘000’state).

## CIRCUIT DIAGRAM FOR POLYCHROMATIC IMAGEGRAPH (POLYCHROMATIC)



## DATA OUT Using PRE-LOADING (POLYCHROMATIC IMAGE)



**CODE REFERENCE APPPENDIX G**

# Polychromatic image to hex

Image extraction is different from monochromatic image, as image have 8 bits per pixels which makes image bulkier in terms of size and difficult in terms of data array management.

Using hex editor, we could see the image in hex format. Further more looking at bitmap format we could determine where is the header and where is the image in hex array displayed by hex editor.

We have used Neo hex editor, of which the output is copied onto the notepad. Simple C code to add “ **,** ” after end of the string results in image array which is ready to be loaded into the microcontroller.

# Ambitious Ideas

Here are some ideas which could have been implemented with tweaks in code.

# STS code V4

Using STS instruction, we could fetch the image directly from any other computer or hardware, say PC. Which then could then be directly display onto the monitor in real time.

This could be implemented using another microcontroller for better resolution. Also this would work as microcontroller which can program itself in real time.

## .GIF

.gif stands for **Graphics Interchange Format**, which is just like any other image standards like the JPEG or PNG, well at least technically.

If anything, they’re more like flipbooks. For one, they don’t have sound. Also, the GIF format wasn’t created for animations; that’s just how things worked out. See, GIF files can hold multiple pictures at once, and people realized that these pictures could load sequentially (again, like a flipbook) if they’re decoded a certain way.

Breaking a .gif file will Spit out multiple frame which then can be displayed consecutively to create animation like effect.

Frames from .gif can be extracted using tools available online\

Reference: <https://picasion.com/split-animated-gif/>

The number of frames obtained after the extraction depends upon the time length of the gif animation.

These framed then could be loaded on to memory space consecutively, and accessed in loop like fashion.

This can be realised onto the hardware with more memory space, like Atmega32.

Appendix A

ATMEGA16 IMPORTANT FEATURES

## HARDWARE

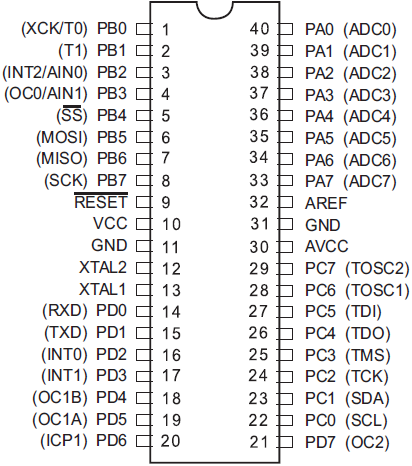
AVR Microcontrollers are Available in three Categories

**TinyAVR:**Less memory, small size, appropriate just for simpler applications

**MegaAVR:** These are the mainly popular ones having a good quantity of memory (up to 256 KB), higher number of inbuilt peripherals and appropriate for modest to complex applications.

**XmegaAVR:** used in commercial for complex applications, which need large program memory and high speed.

## ATMEGA16



Features at glance (only those mentioned here are important from this project’s aspect)

* 16 Kbytes of In-System Self-programmable
* 512 Bytes EEPROM
* 1 Kbyte Internal SRAM
* Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
* Two 8-bit Timer/Counters with Separate Prescalers and Compare Modes
* One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode
* Master/Slave SPI Serial Interface
* Speed Grade, 0 - 16 MHz for ATmega16

Appendix B

Uploading SKETCH PROGRAMMER



Reference: <https://www.ladyada.net/learn/avr/programmers.html>

After writing the code in Atmel Studio (or any other IDE) and then compiling it, generates a hex file.

Hex is what microcontrollers talks in (binary, actually). **In order to upload or program the microcontroller, one would need a programmer.**

It’s a hardware device that allows us to program the microcontroller using a computer.

There are plenty of programmers available in the market for each microcontroller, however a good programmer or development board would support many microcontrollers.

Example: STK 500, AVR DRAGON these are some which promoted by Atmel itself.

The programmer I am using is USB computable ISP (In-system programming) programmer for Robokits India.

Reference: <https://en.wikipedia.org/wiki/In-system_programming>

There are two ways to upload the hex file to the programmer:

1.using Atmel studio or AVRDUDE

2.using the programmer application

We are using the programmer application which is little bit simpler to use, compared to counter parts.

Robokits seem to have discontinued this programmer (released in 2011) and no application of programmer is available in downloadable section, but I do have a copy.

Link: https://drive.google.com/open?id=1Lb8ZcH8WkSmnZ85qeITPmWp1c3Jl4zRJ

There is well a documented text on how to configure the programmer for directly uploading the sketch from the Atmel studio or AVRDUDE.

Appendix C

MIXING ASM AND C

## IDE

Atmel Studio provides development environment from both assembly language and C. Assemble language was important from the aspect of efficient execution and C was convenient for most of the code.

##### MIXING ASSEMBLY AND C

This section of code is one the most important than the rest, at it allowed us to calculate and execute the instructions within the permittable time frame gilded by VGA standards.

Reference on how to do this on Atmel Studio or AVR studio:

<http://ww1.microchip.com/downloads/en/appnotes/doc42055.pdf>

<https://en.wikibooks.org/wiki/Embedded_Systems/Mixed_C_and_Assembly_Programming>

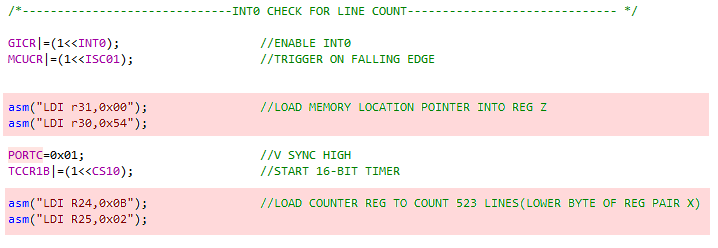
<https://www.nongnu.org/avr-libc/user-manual/group__asmdemo.html>

Reading these documents concluded that there are two ways to include assembly into the main c code.

1. **In line assembly**
2. **Adding .s fine into the main code**

###### In line assembly

One of the most common methods for using assembly code fragments in a C programming project is to use a technique called inline assembly. Inline assembly is invoked in different compilers in different ways. Also, the assembly language syntax used in the inline assembly depends entirely on the assembly engine used by the C compiler.

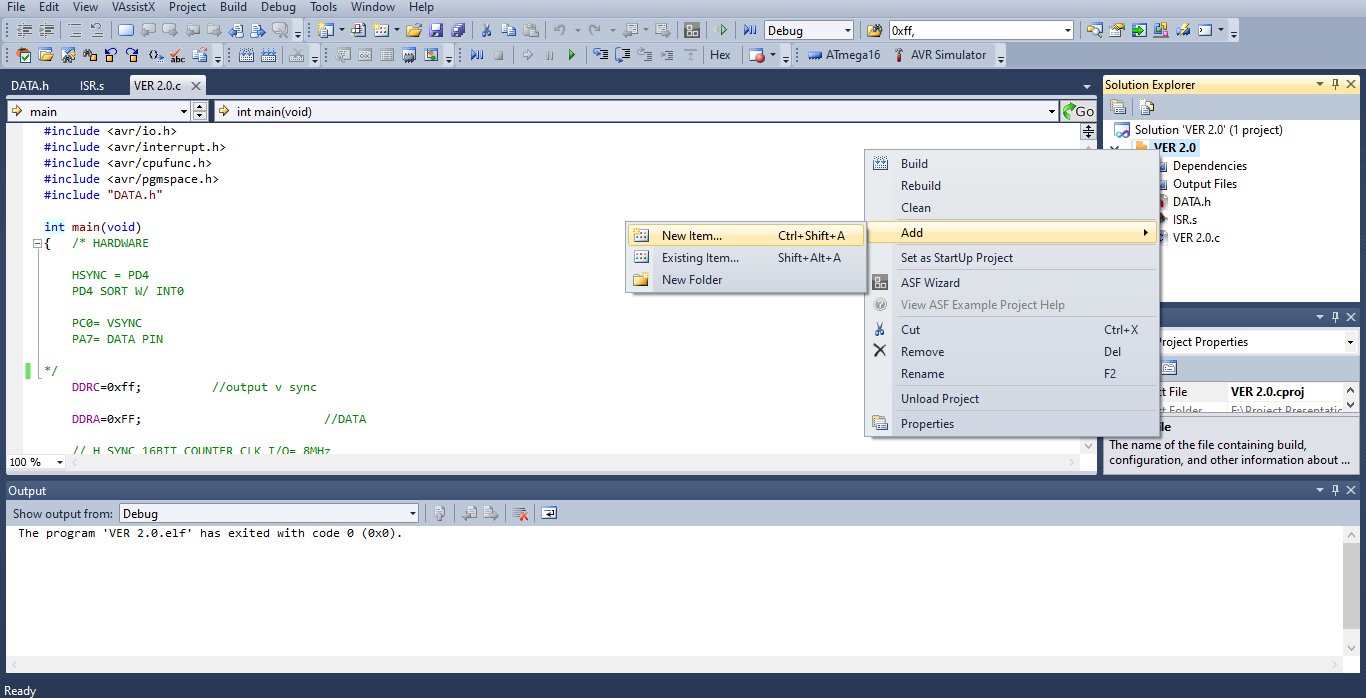


###### Adding .s extension into the main C code

This file is a pre-processed assembly source file. The C pre-processor will be run by the compiler front-end first, resolving all #include, #define etc. directives. The resulting program text will then be passed on to the assembler.

As the C pre-processor strips all C-style comments, pre-processed assembly source files can have both, C-style (/\* ... \*/, // ...) as well as assembly-style (; ...) comments.

One of the source files will need to produce a “main” module for the linker so that the linker knows where to start the application. The most common is a C code file with a function called “main”. However, an assembly file with a subroutine named “main” and declared to be global (using the “.**global**” directive) will also produce a module named “main”.



Appendix D

PROGRAM MEMORY AND LPM

## PROGRAM MEORY

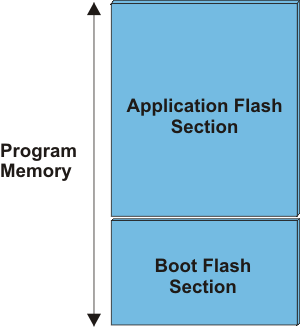
##### INTRODUCTION

**AVR Memory Organization**

The AVR microcontroller's memory is divided into Program Memory and Data Memory. Program Memory (ROM) is used for permanent saving program being executed, while Data Memory (RAM) is used for temporarily storing and keeping intermediate results and variables.

**Program Memory (ROM)**

Program Memory (ROM) is used for permanent saving program (CODE) being executed, and it is divided into two sections, Boot Program section and the Application Program section. The size of these sections is configured by the BOOTSZ fuse. These two sections can have different level of protection since they have different sets of Lock bits.  
Depending on the settings made in compiler, program memory may also be used to store a constant variable. The AVR executes programs stored in program memory only .code memory type specifier is used to refer to program memory.



##### using PROGRAM SPACE FOR ARRAY DECLERATION

**Data in Program Space**

Reference: <https://www.nongnu.org/avr-libc/user-manual/pgmspace.html>

Many AVRs have limited amount of RAM in which to store data, but may have more Flash space available. The AVR is a Harvard architecture processor, where Flash is used for the program, RAM is used for data, and they each have separate address spaces. It is a challenge to get constant data to be stored in the Program Space, and to retrieve that data to use it in the AVR application.

The problem is exacerbated by the fact that the C Language was not designed for Harvard architectures, it was designed for Von Neumann architectures where code and data exist in the same address space. This means that any compiler for a Harvard architecture processor, like the AVR, has to use other means to operate with separate address spaces.

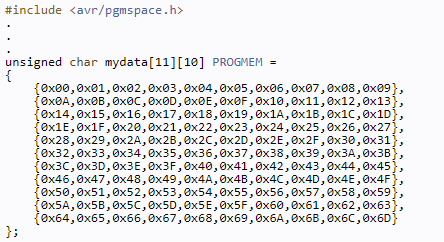
Some compilers use non-standard C language keywords, or they extend the standard syntax in ways that are non-standard. The AVR toolset takes a different approach.

GCC has a special keyword, \_\_attribute\_\_ that is used to attach different attributes to things such as function declarations, variables, and types. This keyword is followed by an attribute specification in double parentheses. In AVR GCC, there is a special attribute called progmem. This attribute is use on data declarations, and tells the compiler to place the data in the Program Memory (Flash).

AVR-Libc provides a simple macro PROGMEM that is defined as the attribute syntax of GCC with the progmem attribute. This macro was created as a convenience to the end user, as we will see below. The PROGMEM macro is defined in the <avr/pgmspace.h> system header file.

It is difficult to modify GCC to create new extensions to the C language syntax, so instead, avr-Libc has created macros to retrieve the data from the Program Space. These macros are also found in the <avr/pgmspace.h> system header file.

Now to data in Program Memory. Use the PROGMEM macro found in <avr/pgmspace.h> and put it after the declaration of the variable, but before the initializer, like so:

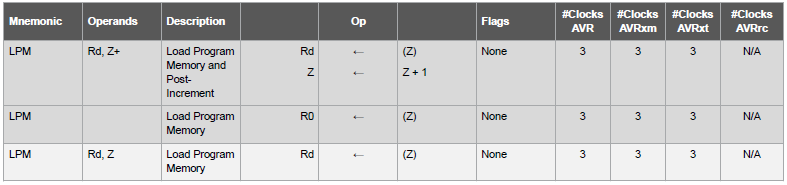


##### asm (“LPM”)

**Description**

Loads one byte pointed to by the Z-register into the destination register Rd. This instruction features a 100% space effective constant initialization or constant data fetch. The Program memory is organized in 16-bit words while the Z-pointer is a byte address. Thus, the least significant bit of the Z-pointer selects either low byte (ZLSB = 0) or high byte (ZLSB = 1). This instruction can address the first 64KB (32K words) of Program memory. The Z-pointer Register can either be left unchanged by the operation, or it can be incremented. The incrementation does not apply to the RAMPZ Register.

Devices with Self-Programming capability can use the LPM instruction to read the Fuse and Lock bit values. Refer to the device documentation for a detailed description.



APPENDIX E

SPI CODE

##### CODE

**.C**

DDRB |= (1<<MOSI) |(1<<SCK) |(1<<SS); // Make MOSI, SCK, SS as Output pin

DDRB &= ~(1<<MISO); // Make MISO pin as input pin

PORTB |= (1<<SS); // Make high on SS pin

SPCR = (1<<SPE) |(1<<MSTR) |(0<<SPR0); // Enable SPI in master mode with Fosc/2

SPSR = (1<<SPI2X); // Speed doubler

**.S**

INT0\_vect: NOP //FRONT PORCH DELAY

NOP

NOP

NOP

NOP

NOP

NOP

NOP

NOP

LDI R26,0X60 //LOAD REG X WITH MEMORY ADDRESS

LDI R27,0x00

LOOP: LD R20,X+ //LOAD CONTENT OF POINTER X INTO R16 THEN INC X

OUT 0x0F,R20 //LOAD VALUE INTO SPDR (SPDR I/O address = 0X0F)

/\* THIS IS DEALY LOOP CALCULATED ON BASE OF TIME TAKEN BY SPDR REGISTER TO FULLY TRANSTER ITS CONTENTS ONTO THE MOSI PIN, ONE LESS NOP AND DATA COLLISION OCCURS \*/

NOP //TO AVOID POLLING LOOP

NOP //13 NOP AS FOSC/2

CPI R26,0x69 //EXIT IF 8 BYTES ARE TRANSFERED ELSE LOOP

BRNE LOOP

RETI

APPENDIx F

CODE VER 2.0

## CODE

There are three parts to the code

1. **.s**
2. **.c**
3. **.h**

**.s**

This is time critical part of the code contains the assembly interrupt service routine which is responsible for pushing out data.

**.c**

This part of the code contains the configuration of the timers and interrupts.

**.h**

It’s the header file which have the array declaration of the image. Viz hex image is stored here.

##### .h

const unsigned char ARRAY[14000] PROGMEM = {

// First 33 lines Back Porch V-SYNC

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

…

…

…

};

##### .c

#include <avr/io.h>

#include <avr/interrupt.h>

#include <avr/cpufunc.h>

#include <avr/pgmspace.h>

#include "DATA.h"

int main(void)

{ /\* HARDWARE

HSYNC = PD4

PD4 SORT W/ INT0

PC0= VSYNC

PA7= DATA PIN

\*/

DDRC=0xff; //output v sync

DDRA=0xFF; //DATA

// H SYNC 16BIT COUNTER CLK I/O= 8MHz

OCR1A = 0x01FB; //SET TOP VALUE IN 16BIT COMPARE REG 507

DDRD |= (1<<PD4); //PWM PIN AS OUTPUT

OCR1B=0x01BF; //LOAD COMPARE VALUE(TOGGLE PD4 ON MATCH) 447

TCCR1A|=(1<<COM1B1); //FAST PWM MODE WITH TOP ORC1A (NO PRE SCALE)

TCCR1A|=(1<<WGM10);

TCCR1A|=(1<<WGM11);

TCCR1B|=(1<<WGM12);

TCCR1B|=(1<<WGM13);

/\*------------------------------INT0 CHECK FOR LINE COUNT------------------------------ \*/

GICR|=(1<<INT0); //ENABLE INT0

MCUCR|=(1<<ISC01); //TRIGGER ON FALLING EDGE

asm("LDI r31,0x00"); //LOAD MEMORY LOCATION POINTER INTO REG Z

asm("LDI r30,0x54");

PORTC=0x01; //V sync high

TCCR1B|=(1<<CS10); //start 16bit timer

asm("LDI R24,0x0B"); //LOAD COUNTER REG TO COUNT 523 LINES(LOWER BYTE OF REG PAIR X)

asm("LDI R25,0x02");

sei();

while(1)

{

\_NOP(); //TODO:: Please write your application code

}

}

##### .s

#include <avr/io.h>

.global INT0\_vect

INT0\_vect : SBIW R24,1 //SUB(1) IMEDIATE TO WORD LOCATED IN REG PAIR X

BREQ GEN\_VSYNC //JUMPS TO GEN\_VSYNC IF A FRAME IS COMPLETED

JMP DATAOUT

GEN\_VSYNC:

nop //delay wait for 523th line to complete

nop

nop

nop

nop

LDI R16,163

LOOP:DEC R16

BRNE LOOP

LDI R16,0x00 //pull PC0 LOW

OUT 0x15,R16 //pull PC0 LOW

LDI R16,255 //wait for two lines

LOOP2:DEC R16

BRNE LOOP2

LDI R16,79

LOOP3:DEC R16

BRNE LOOP3

NOP

NOP

NOP

NOP

NOP

nop

NOP

LDI r31,0x00 //also reset memory pointer

LDI r30,0x54

LDI r24,0x0b //also reset counter

LDI r25,0x02

LDI R18,0x01

OUT 0x15,R18 //pull PC0 HIGH

RETI

DATAOUT : LDI R19,17 //wait sync pulse + back porch

LOOP4:DEC R19

BRNE LOOP4

nop

nop

nop

nop

nop

nop

nop

nop

nop

nop

//1st BYTE

LPM R16,Z+

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

...

...

//20

LPM R16,Z+

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

LSL R16

OUT 0x1B,R16

nop

nop

nop

nop

LDI R16,0X00

OUT 0X1B,R16

nop

nop

nop

DEC R19

NOP

nop

nop

nop

RETI

APPENDIX G

CODE VER 2.9/3.0

## CODE

There are three parts to the code

1. **.s**
2. **.c**
3. **.h**

**.s**

This is time critical part of the code contains the assembly interrupt service routine which is responsible for pushing out data.

**.c**

This part of the code contains the configuration of the timers and interrupts.

**.h**

It’s the header file which have the array declaration of the image. Viz hex image is stored here.

##### .h

const unsigned char ARRAY[15800] PROGMEM = {

//29 pixels x 525

//leave first 33 lines as they are part of Back Porch (V-SYNC)

//line 1

0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

};

##### .c

#include <avr/io.h>

#include <avr/interrupt.h>

#include <avr/cpufunc.h>

#include <avr/pgmspace.h>

#include "DATA.h"

int main(void)

{ /\* HARDWARE

h sync = pd4

pd4 sort w/ int0

portC= v sync

portA= data pin

\*/

DDRC=0xff; //output v sync

DDRA=0xFF; //DATA

// H SYNC 16BIT COUNTER CLK I/O= 8MHz

OCR1A = 0x01FB; //SET TOP VALUE IN 16BIT COMPARE REG 507

DDRD |= (1<<PD4); //PWM PIN AS OUTPUT

OCR1B=0x01BF; //LOAD COMPARE VALUE(TOGGLE PD4 ON MATCH) 447

TCCR1A|=(1<<COM1B1); //FAST PWM MODE WITH TOP ORC1A (NO PRE SCALE)

TCCR1A|=(1<<WGM10);

TCCR1A|=(1<<WGM11);

TCCR1B|=(1<<WGM12);

TCCR1B|=(1<<WGM13);

/\*------------------------------INT0 CHECK FOR LINE COUNT-----------------------\*/

GICR|=(1<<INT0); //ENABLE INT0

MCUCR|=(1<<ISC01); //TRIGGER ON FALLING EDGE

asm("LDI r31,0x00"); //LOAD MEMORY LOCATION POINTER INTO REG Z

asm("LDI r30,0x54");

PORTC=0x01; //V SYNC HIGH

TCCR1B|=(1<<CS10); //START 16-BIT TIMER

asm("LDI R24,0x0B"); //LOAD COUNTER REG TO COUNT 523 LINES(LOWER BYTE OF REG PAIR X)

asm("LDI R25,0x02");

sei();

while(1)

{

\_NOP(); //TODO:: Please write your application code

}

}

##### .s

#include <avr/io.h>

.global INT0\_vect

INT0\_vect : SBIW R24,1 //SUB(1) IMEDIATE TO WORD LOCATED IN REG PAIR X

BREQ GEN\_VSYNC //JUMPS TO GEN\_VSYNC IF A FRAME IS COMPLETED

JMP DATAOUT

GEN\_VSYNC:

nop //delay wait for 523th line to complete

nop

nop

nop

nop

LDI R16,163

LOOP:DEC R16

BRNE LOOP

LDI R16,0x00 //pull PC1 LOW

OUT 0x15,R16 //pull PC1 LOW

LDI R16,255 //wait for two lines

LOOP2:DEC R16

BRNE LOOP2

LDI R16,79

LOOP3:DEC R16

BRNE LOOP3

NOP

NOP

NOP

NOP

NOP

nop

NOP

LDI r31,0x00 //also reset memory pointer

LDI r30,0x54

LDI r24,0x0b //also reset line counter

LDI r25,0x02

LDI R18,0x01

OUT 0x15,R18 //pull PC1 HIGH

RETI

DATAOUT : LPM R0,Z+

LPM R1,Z+

LPM R2,Z+

LPM R3,Z+

LPM R4,Z+

LPM R5,Z+

LPM R6,Z+

LPM R7,Z+

LPM R8,Z+

LPM R9,Z+

LPM R10,Z+

LPM R11,Z+

LPM R12,Z+

LPM R13,Z+

LPM R14,Z+

LPM R15,Z+

LPM R16,Z+

LPM R17,Z+

LPM R18,Z+

LPM R19,Z+

LPM R20,Z+

//1(20)

OUT 0x1B,R0

LPM R0,Z+

OUT 0x1B,R1

LPM R2,Z+

OUT 0x1B,R2

LPM R3,Z+

OUT 0x1B,R3

LPM R4,Z+

OUT 0x1B,R4

LPM R5,Z+

OUT 0x1B,R5

LPM R6,Z+

OUT 0x1B,R6

LPM R7,Z+

OUT 0x1B,R7

LPM R8,Z+

OUT 0x1B,R8

LPM R9,Z+

OUT 0x1B,R9

nop

nop

nop

OUT 0x1B,R10

nop

nop

nop

OUT 0x1B,R11

nop

nop

nop

OUT 0x1B,R12

nop

nop

nop

OUT 0x1B,R13

nop

nop

nop

OUT 0x1B,R14

nop

nop

nop

OUT 0x1B,R15

nop

nop

nop

OUT 0x1B,R16

nop

nop

nop

OUT 0x1B,R17

nop

nop

nop

OUT 0x1B,R18

nop

nop

nop

OUT 0x1B,R19

nop

nop

nop

OUT 0x1B,R20

//2

nop

nop

nop

OUT 0x1B,R0

nop

nop

nop

OUT 0x1B,R1

nop

nop

nop

OUT 0x1B,R2

nop

nop

nop

OUT 0x1B,R3

nop

nop

nop

OUT 0x1B,R4

nop

nop

nop

OUT 0x1B,R5

nop

nop

nop

OUT 0x1B,R6

nop

nop

nop

OUT 0x1B,R7

nop

nop

nop

OUT 0x1B,R8

nop

nop

nop

OUT 0x1B,R9

nop

nop

nop

ldi r28,79

here: dec r28

BRNE here

ldi r16,0X00

out 0X1b,R16

RETI