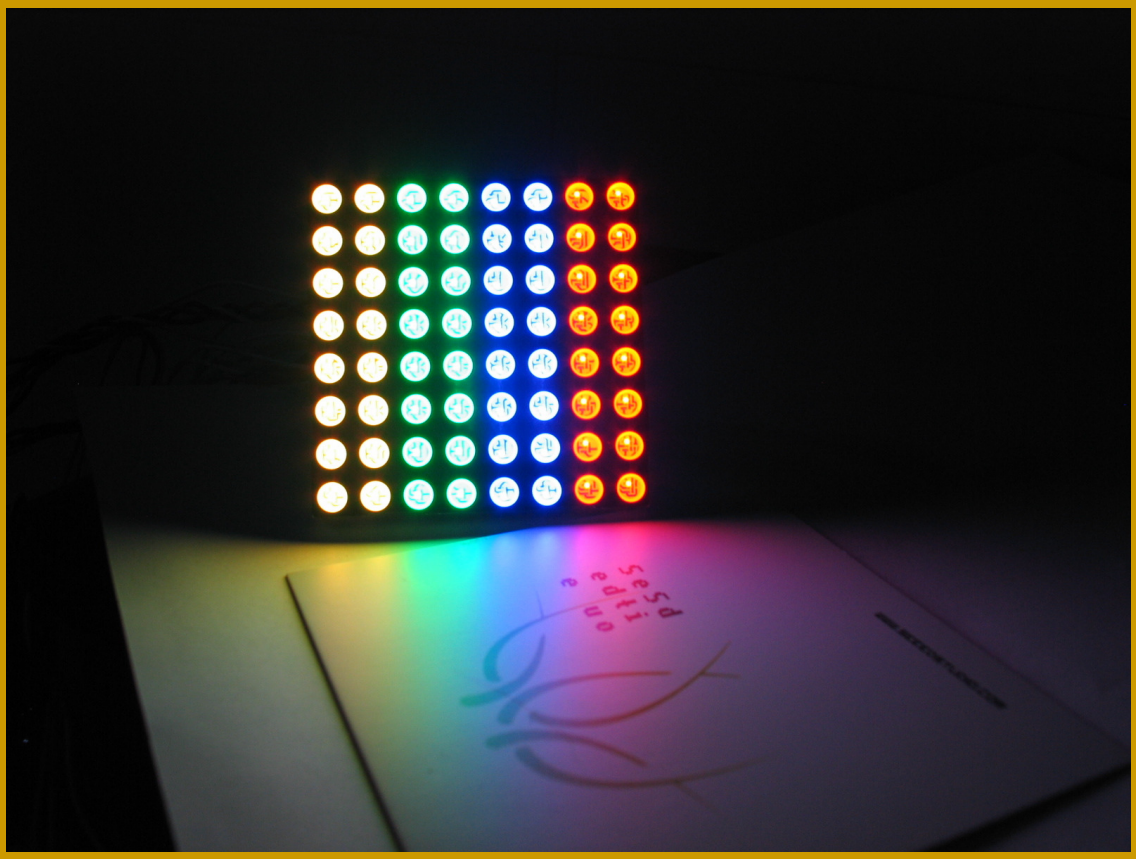
Microtronics Pakistan

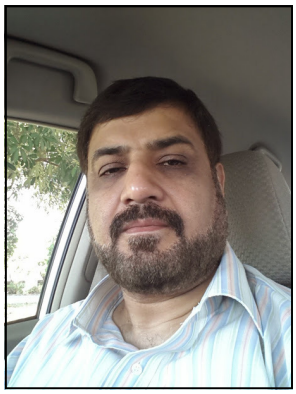
Lets Make LED Matrix Displays

Its Fun to Learn and Explore



Amer Iqbal Qureshi

Acknowledgments



I am so fortunate to have a great collection of friends and family who have been very supportive over the years. As my Electronics experience and knowledge have matured I have been blessed with forming some great relationships in the community. I have to thank everyone who has been supportive in this over the years. We have a group of similar minded friends, all professionals in their own respective fields yet having one thing in common, selfless and dedication to electronics and promoting the knowledge. I will strongly suggest to form similar groups in your local community as nothing is more rapid than sitting together and talking about.

With that said, there are some specific people to thank. First, Numan, who has been a great friend to me personally, and who is always excited when good things happen for me. Umer, you really helped me keep my composure and perspective in the past year with so many things coming at me. Shahzad, took many hours out of his busy job to make pieces of odd hardware that group needed and I would be failing in my duties if I don't mention Irfan who is always there to share the industry experiences and arranging components from the local market. Everyone in group contributed significantly in suggesting the hardware design and software making.

Nothing of this magnitude is without challenges and they have been helpful in solving the many challenges that arose. Producing a book is no small task, but ultimately rewarding. It is always an honor to have the opportunity to share knowledge and experience about something you love with others on the scale possible with a book like this. I know I have learned a lot and I hope to keep learning and sharing.

This is not a textbook and does not contain every little detail of LED displays. Professionally I am a Cardiovascular Surgeon, with lots of passion for electronics. This is from a hobbyist to other hobbyists. I hope this will act as a starting point for many who want to see how best we can use our spare times making things, and enjoying the fun.

Amer Iqbal Qureshi
ameriqbalqureshi@yahoo.com

Introduction

Dear reader thank you for picking up this book and welcome to the wonderful and exciting world of LEDs. The idea for this book was born when I wanted to make my own LED moving signs, although internet was full of knowledge and ideas yet no where I could find a complete step by step solution as to help me in getting started. So I decided to compile my experiences in one place, helping students and hobbyists to begin. Moreover this project acts as a base point to learn many abstract ideas about programming and take a proper channel to develop applications in a "Laired" manner. The concepts and programming methodology you will learn in this book will be helpful in solving many other electronics projects you come across.

There are many different hardware and software solutions to drive the LED matrices, I would however discuss the simplest one, that requires minimal hardware, and the one that is easily available anywhere in world. Secondly emphasis is on learning by doing, I shall take you step by step, to explain the concepts. It is very easy to publish the final software and hardware design, but that does not explain the hidden algorithms and brilliant ideas behind. You can always modify the code and experiment your way to extend whatever you learn in this book.

Many experiments in the book will be mentioned to highlight the concepts and may not be actually used in the final code, yet they act as building sound understanding of what is happening under the hood. One special requirement for this project that I ultimately settled was to minimize the hardware requirements and try to implement the things through software techniques. Nevertheless I will talk a little bit about the additional hardware that can reduce the programmer's work but add to the complexity of circuit design and therefore manufacturing. One of the major advantages of microcontrollers is to reduce the physical hardware and do the needful done in software.

My second requirement was to make the hardware pluggable into any microcontroller lines, may it be PIC, AVR or the commonly used Arduino. I will however use PIC microcontroller, for it is easily available and easy to program.

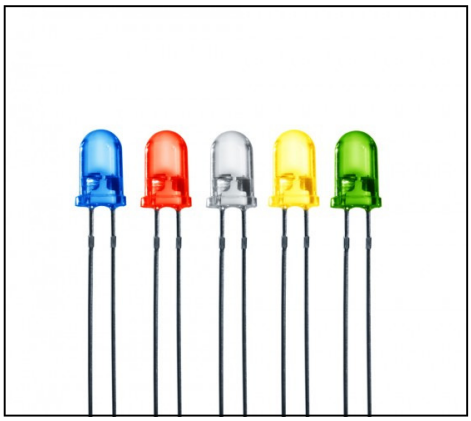
What you need?

You need two sets of tools to explore all this. First a hardware that we will shortly discuss. This includes an LED matrix module, a few shift registers and a few current limiting resistors. You can assemble the hardware on breadboard or as I prefer on a perf-board. Even more useful is if you can find a pre-built hardware. I however suggest making your own.

The second set of tools needed are software compilers and simulators. Simulator is optional but really helpful in experimenting before making real hardware. We will be using MikroC, a popular compiler from microelectronica (www.mikroe.com). Demo version will be sufficient for this project and this can be downloaded from their site. For simulations I will use Labcenter Proteus ISIS version 7.10

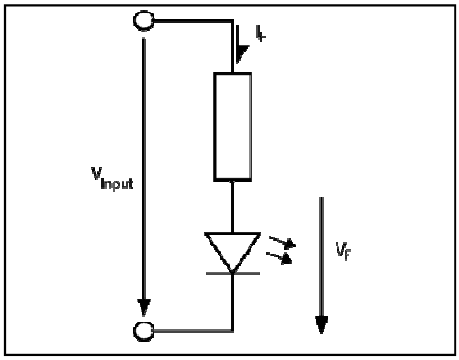
As far as microcontroller is concerned you use any, but I will use PIC18F2550, running at 48MHz. The advantage of this controller is that it has built-in USB connectivity that I use to program directly without need of an external programmer, you can use your own microcontroller setup, and programmer as you like, but I will certainly dedicate a section on how to use and setup the PIC18F2550 platform.

What is an LED ?



So lets start from the basics, as we are going to do with LEDs, it is better to have an understanding about these small devices or may be components. This knowledge is essential in the sense that you might need to tweak a little bit with your hardware in case your LEDs have differing specs.

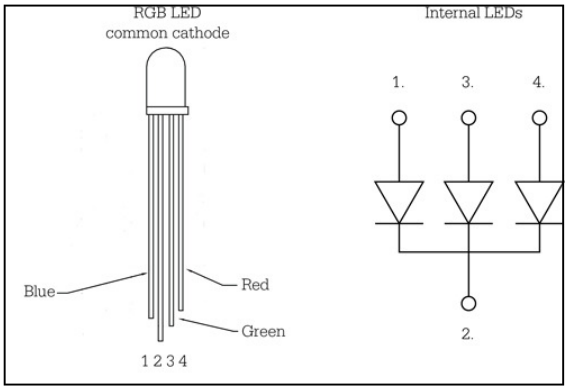
So LED is basically a diode, as you all know, it has a PN junction and like any PN junction it does conduct in one direction only. The second thing is that there is certain level of threshold voltage that needs to be applied to make it conduct. Before that voltage it has very high resistance, so you can not assess its resistance when its conducting and therefore the current being drawn. Even more important is the fact that an over -current will kill the PN junction, if your source is able to provide upto many 100mA current, when the breakthrough voltage is applied the resistance of LED falls to almost negligible, and it conducts all the current that source can provide, this high current would certainly damage the LED. It is therefore important to know the current requirement of the LED, and depending upon your supply voltage calculate a series current limiting resistor. Most of the LEDs commonly used consume 25mA of current, however the bigger and brighter LEDs might be consuming more current. So you got to check out either with manufacturer or measure with a meter. Keep it in mind that LEDs are current driven devices, not voltage driven. To keep the constant illumination they need a constant current source, which is beyond this discussion.



In most projects we use microcontrollers and therefore the supply voltage available is 5V DC. Using Ohm's law I=V/R we calculate the value of R for 25mA. Since all diodes have a voltage drop across them which is roughly 0.8V so driving voltage is 5-0.8= 4.2V thus R is calculated as 168 Ohms. We can round it off to 150 Ohms. The resistor has to be in series and therefore anywhere on anode or cathode side.

The colors in LEDs is determined by the type of PN gap, and impurities added to them. It is not determined by the plastic casing color, you might have encountered transparent LEDs with various colors when illuminated.

Multicolored LEDs are also common these days, they are actually two or three LEDs manufactured in a single casing having one end as common electrode and others are independently driven. There are Bi-color or Tri-color LEDs available. The brightness of an LED is a function of current flowing, and remember this not a linear relation, on top of that the sensitivity of our eye to detect a change in brightness is also not linear, it can detect even minor changes in intensity at lower brightness levels than at higher brightness. Same is true about different colors, the sensitivity to appreciate red is more than green. Thus when making different shades of light by mixing different colored LEDs does not bear uniform response as mathematics. So if you drive red and green LEDs together the with same current, the result is not orange or yellow, but a slight orange, with more tinge of green. Therefore if we want to mix the colors and make different shades we have to drive the LEDs with varying currents. This is achieved dynamically with reducing the driving voltages. Changing the driving voltage using digital systems is

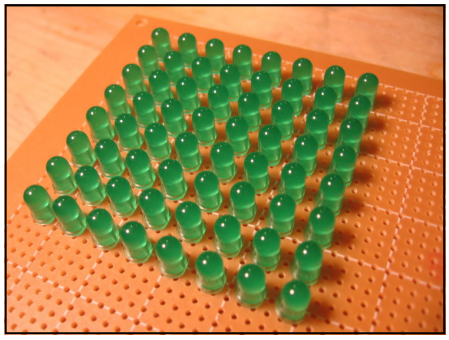


achieved through “pulse width modulation” or PWM. We will talk about this in a later chapter when different shades are to be made.

So if you are going to use multicolor LEDs and want to mix the colors, find a way to modify the current flow to get optimum results, and if you are good at programming you can use PWM to programmatically control this. I personally used a 100 Ohms resistor with green LEDs and 220 or 330 Ohms resistors with Red LEDs. Since I used Bi-color modules, I am not sure what to do with blue LEDs.

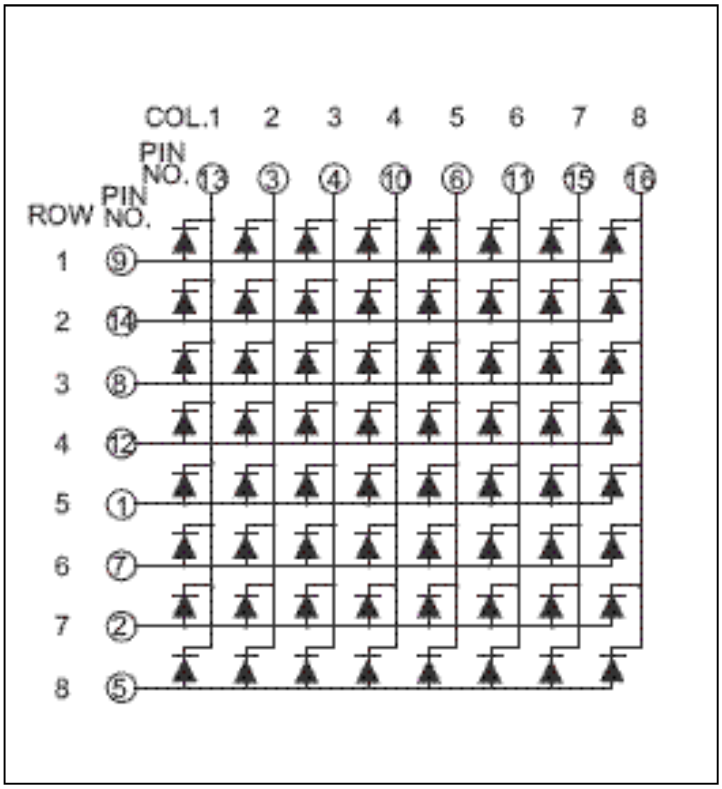
What is an LED Matrix

Driving individual LEDs as separate device needs a separate control line for each LED, that means a lots of wires and huge hardware. LED matrix is a special arrangement of LEDs in the form of an array, or matrix. Where Each LED has a row and a column.



Its not only that LEDs are arranged in manner, they still have two leads each. So if we have an 8 rows and 8 columns , an 8x8 LED matrix we have a total of 64 LEDs and have to manage 128 leads. OK we can connect all the cathodes together, still we have 64 anodes to take care of.

In matrix LEDs we play another clever trick while wiring them. So that all the anodes in a row are connected together as



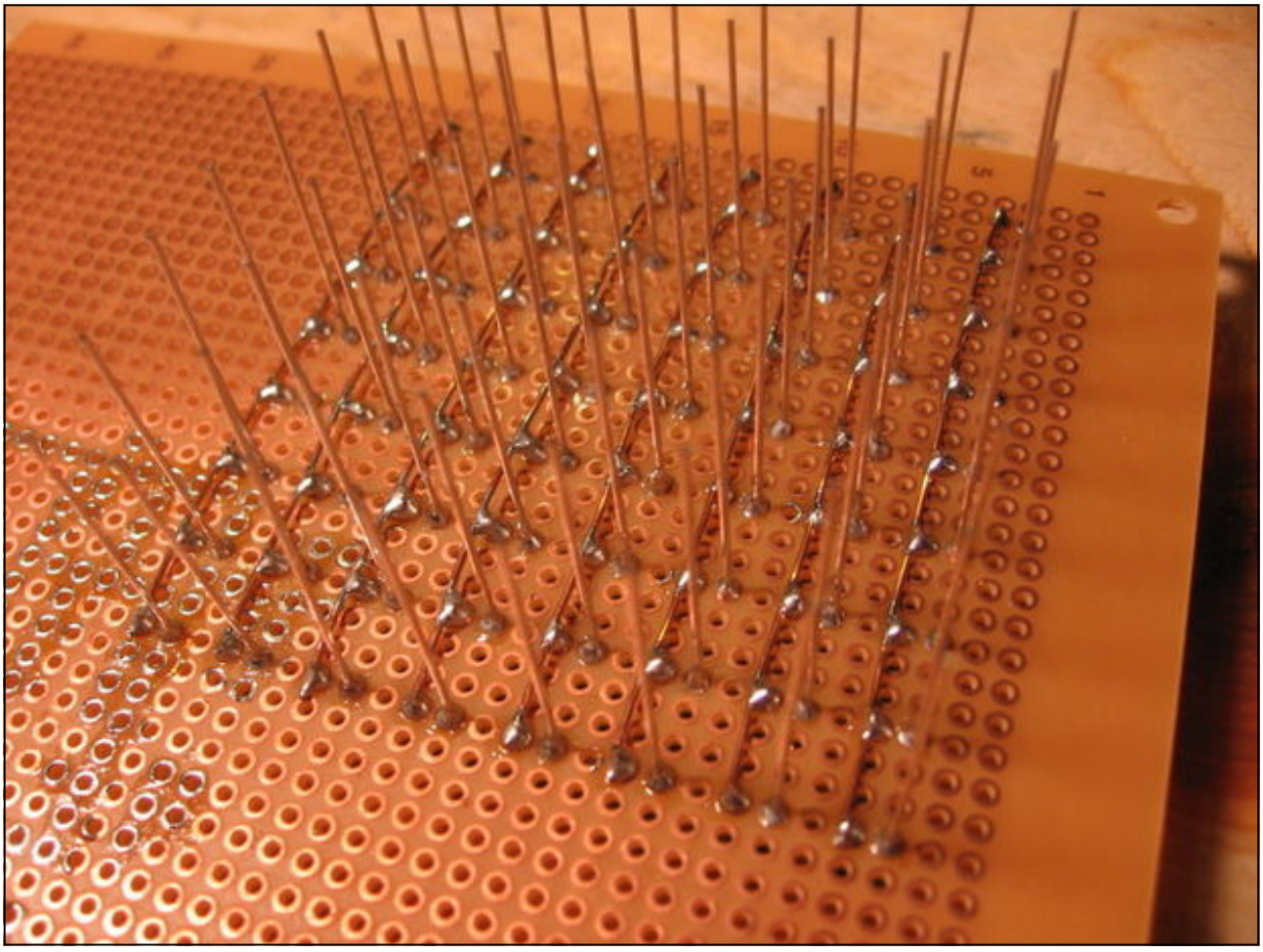
one common anode, and all the cathodes in a column are connected as a single cathode. Now if we have an 8x8 matrix we will have eight wires for the 8 rows and 8 wires for the 8 columns. You can see in this connection diagram if lets say we give positive supply to row 1 and negative or ground to column 1 the upper left led glow. In this way we can control any single LED, but there are some limitations you can quickly figure out some combinations of LEDs that can not be simultaneously turned on without affecting others. Luckily there is a solution for that, and here nature comes to help us, we tend to illuminate a single row at a time and do it so fast that our eyes can not make a track of the change, and we see the entire matrix glowing. This is called “persistence of vision” or

POV in brief. (http://en.wikipedia.org/wiki/Persistence_of_vision).

Making the Matrix

The first thing is to wire up the LEDs. There are two options do decide first. Weather we want our rows to be connected to anodes of LEDs, or Cathodes to be connected. This does not matter significantly as we can compensate for this in our software. However certain hardware designs and driver ICs may require one or the other situation. I would be using anode connected to the rows and cathodes to the columns as shown in the matrix wiring diagram above.

The second question is if you want to wire the LEDs yourself or want to buy pre-made modules. The choice is yours. When making with hand on a veroboard, make sure all LEDs have equal brightness, as there are different batches of available LEDs with varying intensity on the same current.



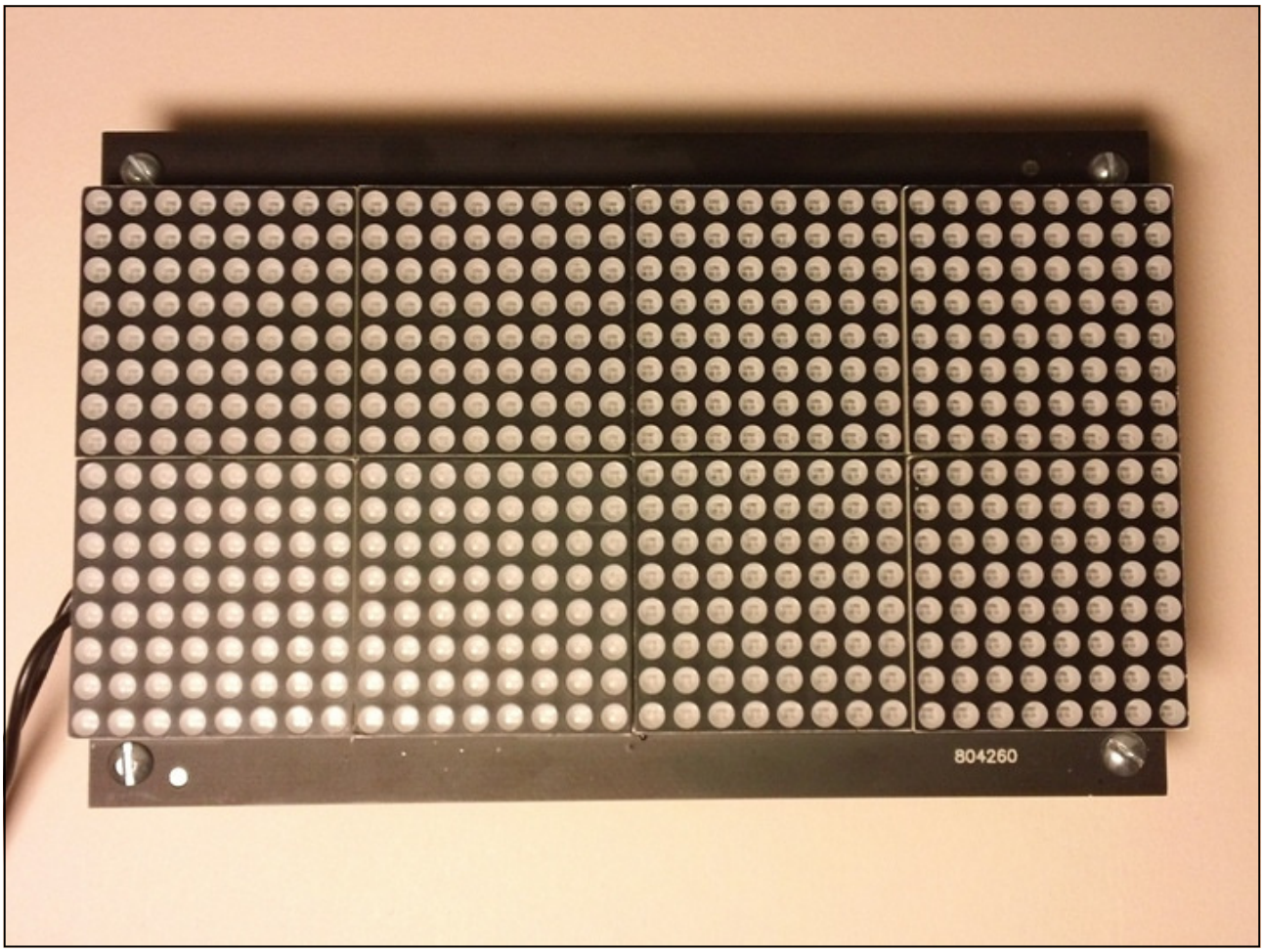
This is really a time consuming job, but a fun to do. The things tend to become even more complex if you want to use multicolored LEDs. Another important point while selecting these LEDs is that use the dull LEDs, and not the ones with crystal clear lens in front, because they tend to glow so sharply that the image quality is rendered useless. So use good illumination but dull LEDs.

Pre-Built LED Modules

Today many commercially made LED modules, having, single, double or tricolor LEDs are available. The have all the necessary wiring done inside them and only rows and columns pins are

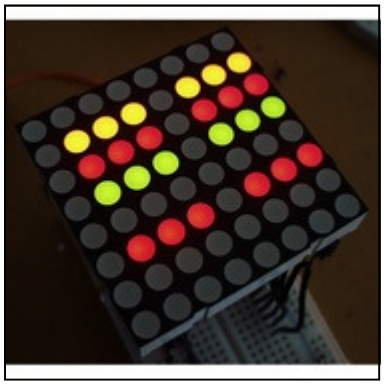
brought out for further interfacing.

They are available in different sizes and configurations like 5x7, 8x8, 16x16 and so on. Once you understand the basic interfacing techniques you can make larger displays by joining multiple modules side by side and connecting them together as one matrix.



The image above shows 8 modules of 8x8 connected together to make a one large matrix 32 x 16 LEDs. Once the matrix is made the software will deal with it as one matrix, and mot as 8 separate modules.

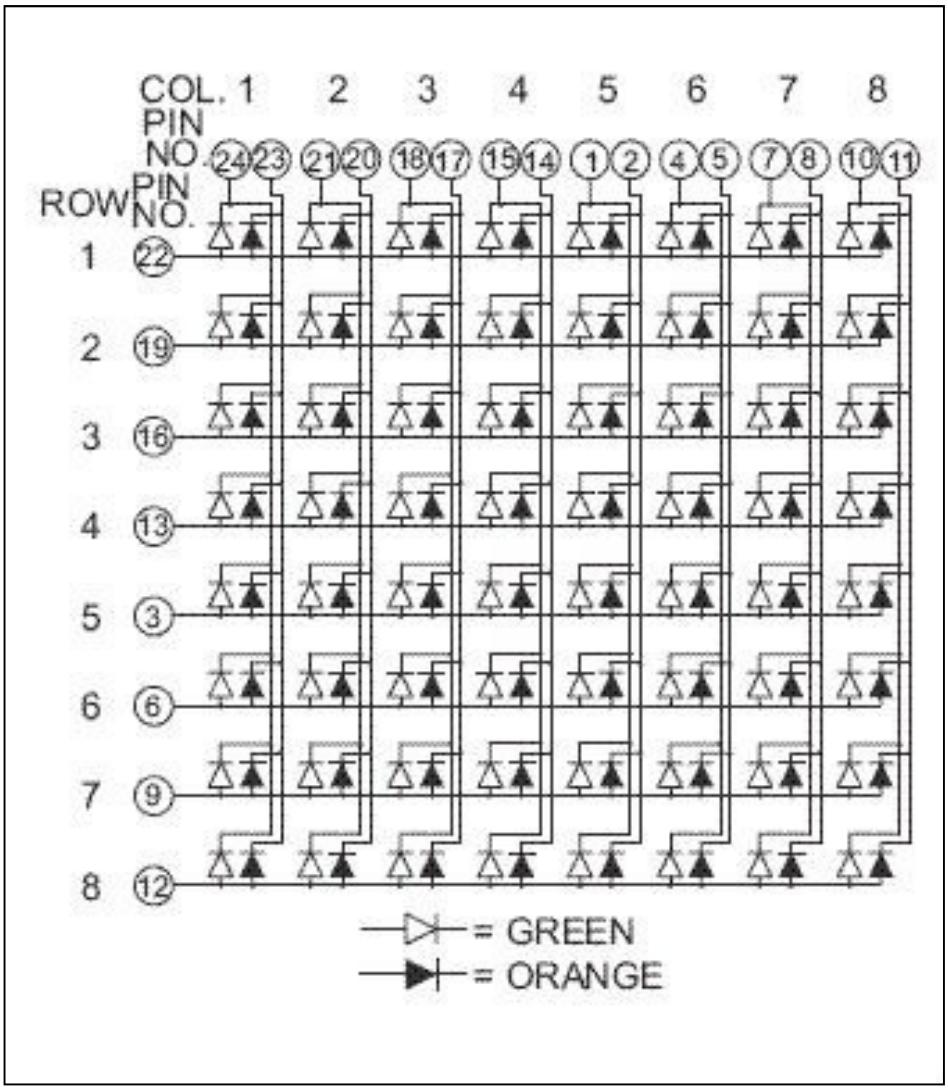
In this book I will use single module of 8x8 LED matrix having bi -color LEDs in each pixel. We will be able to make three colors red, green, yellow and off-course black. Later we will use PWM techniques to make even more shades by altering the intensities of LEDs dynamically through software.



So each pixel of our module will have actually two LEDs in it. A total of 128 LEDs will be there. We can consider for design purpose as two matrices, The anodes of all the LEDs in a row are tied together as rows and cathodes are separately for each column of red and green separately.

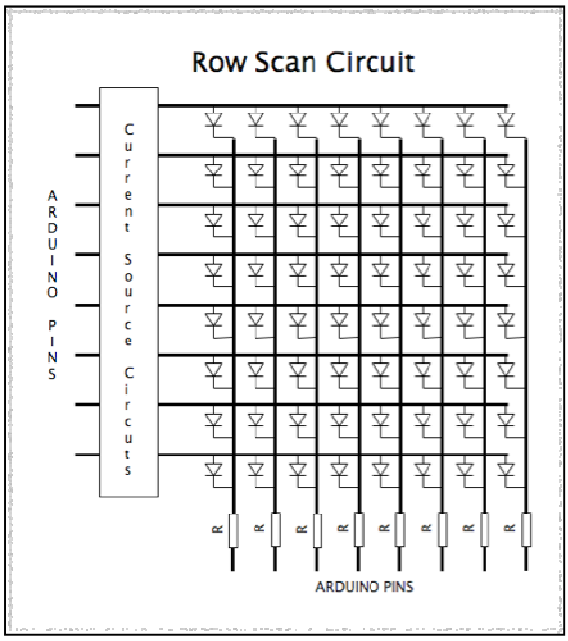
This will therefore give us 8 pins for the rows and 16 pins for the columns. A total of 24 pins. The arrangement of pins in modules are not as rows and columns, but they are mixed up. So you have to find out manually using a battery with current limiting resistor to find out the row pins and column pins for each color. This is a daunting task but necessary, be-

cause a slight mistake will spoil the entire project.



The image above shows the wiring of module I am using check with your module, as this might vary from vendor to vendor.

Scanning The Display

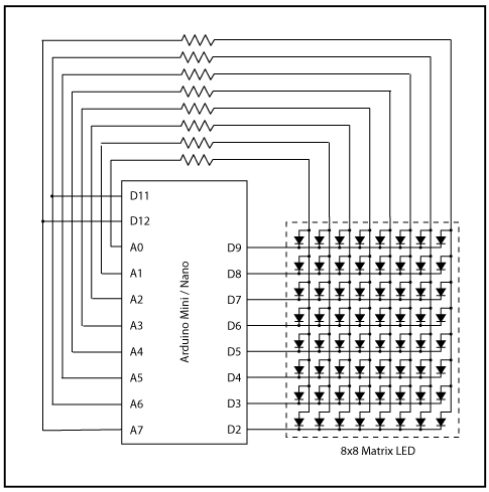


As I have mentioned previously this type of display technology (and Even Televisions) use POV as a tool to give an illusion of display. Now when we have the matrix we have two options, either we illuminate one row at a time, with a pattern of LEDs On or Off, give it some time to cast an image on our eyes, and then turn it off and move to next row. Repeat the same process again and again. Alternately we can set the LEDs in a column and then scan the columns. Both techniques are equally good. For smaller displays like this the column scanning is easy as you don't have to manipulate the data bits much, but as you increase the length of display matrix, more and more columns have to be scanned and the display becomes jerky. I have therefore chosen the row scan method so that the concepts

gained on this small scale be applied easily to larger scale displays. As you can see when a single row is displayed, and all LEDs in that row are supposed to be ON, there can be huge demand of current. Like if we have 16 LEDs in one row (8 red and 8 green) and each one needs about 25mA for full brightness, then we need at least 16 x 25= 400mA of current. So the current source circuit must be able to supply this current, falling short of this the LEDs would appear dim. Commonly a transistor can be used here to power the necessary current. Driving them through microcontroller pins can be dangerous as this can damage the IO pin circuitry. Or you can use an intervening buffer that can power the entire row. The current limiting resistors will be on cathode side as we have to limit the current through red more than green.

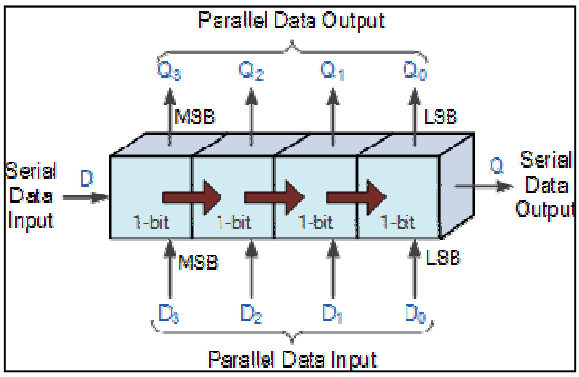
Hardware Design Options

Now that we have decided we will go for row scanning, we need to find out the best solution for selecting rows and passing data to columns. Remember we have 8 rows and 16 columns (8 for red and 8 for green). If we use a microcontroller with fairly large number of digital I-O lines we can use two 8 bit ports for the columns and one 8 bit port through a reasonable buffer to address the rows. This will need a total of 24 digital I-O lines. Programming will be simple as setting column data will take two bytes to be passed on two ports and row can be selected by setting the appropriate port bit to 1 and then shifting down by one position on every data change. This design will therefore require a controller with large number of I-O lines and the display can not be used with controllers having smaller number of I-O lines. As shown in this figure as our display has two matrices we need another set of 8 digital I-O lines with this arrangement, secondly if we need to expand the display, like having 4 or five modules together in one long row, there is hardly any commonly available controller that has so many I-O lines. So this method is out of our scope.



Shift Registers

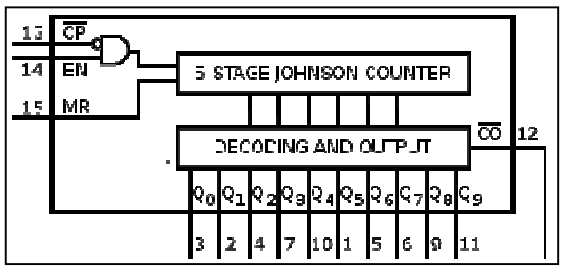
Shift registers are another option, these registers can take serial data one bit at a time and shifts the bits like a conveyer belt, giving out the result as parallel output on its pins. These are also called serial to parallel shift registers. Another beauty of these registers is that they can be cascaded together to make a long daisy chain. All they need is three wires for input, one for data bit, second for clock bit to shift data and third latch bit, which is given a pulse when all data has been sent, and pulsing will make the data appear on output pins. Many such registers are available, the 74HC595 being most commonly used. So if we put two 8 bit shift registers in series with each other, and tie their output lines to columns of our matrices, we will be able to send 16 bits of data on just three digital I-O lines, which is not a big problem for even a small microcontroller.



Scanning Rows

Now the issue to select rows, since we have 8 rows and we have to select one row at a time, we have many options. We can use CD4017, which is a decade counter, it takes a single I-O line to clock it. It will set its outputs high one after the other after every pulse, after 10 pulses the first one

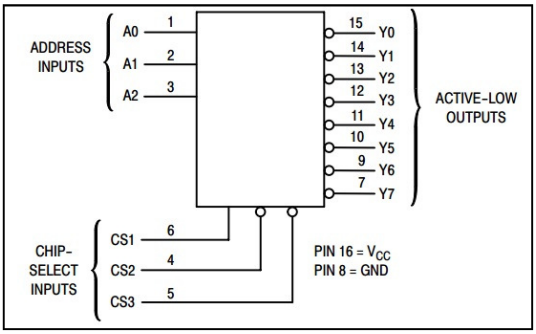
will again start the cycle. But there are two problems, first it has 10 outputs and we want to clock it back to first line after 8, OK we can give two extra pulses after 8, not a big issue, secondly when it is first powered on, the outputs are in unknown state, we can never know how to get the first I-O high. Fortunately there is a Reset pin, that when clocked will reset the chip to state 0.this will be ok for us, as it will take an additional two I-O lines one for clock and other for reset, and 8 outputs Q0 to Q7 can be used to select the individual rows.



What if we later decide to use two rows of 8x8 matrices to make a 16x32 screen? One can think of cascading the two CD4017 chips, that's fair, but when we will clock 11th time the next chip will get a pulse and step forward, but first chip will reset back to select row 0. thus we will have two rows selected and on each this is not a desirable behavior. So this arrangement is good for this project at-least but not a good one if we want to expand the design.

3 to 8 Line decoder (74HC138)

Another option is to use a 3 to 8 line decoder. This chip takes three digital inputs, and a binary number set to these three lines will select the corresponding output. Like if you set 101 then this binary number corresponds to decimal 5 and Y4 will be selected. This particular chip (74HC138) sets the selected pin low and all others high, which is not what we need, however its good arrangement to drive eight corresponding PNP transistors. We can also find 3 to 8 decoder IC with active high outputs that can be used directly to drive our display rows. Similarly we have 74HC154 which is 4 to 16 bit decoder. It can accept 4 bit binary coded decimal number on its 4 inputs and can select one of its 16 outputs. Moreover with little extra circuitry we can make more input bits and output bits. This sounds a reasonable solution. So if we use it we need three more I-O lines for row selecting and three I-O lines for shift registers to send column data, a total of 6 I-O lines. Well a reasonable solution.

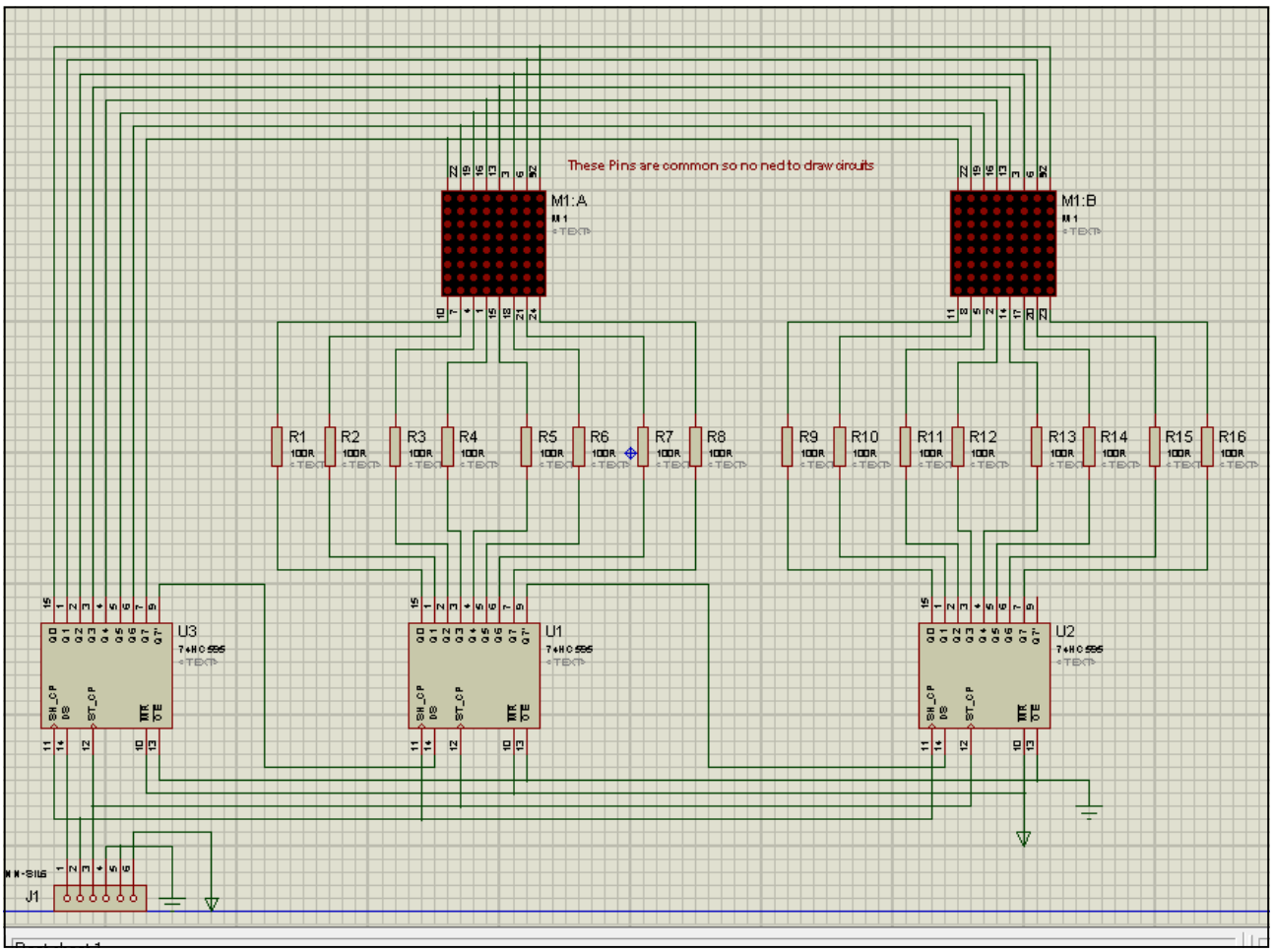


Using 3 Cascaded Shift Registers

The last option that comes to my mind is to use another shift register for row selecting. Since row is to be selected every time column data has been sent, there is no need to drive this shift register separately, and we can cascade all three shift registers together, clocking data in first shift register and sending in three bytes (24 bits) first two bytes contain column data and last byte row select. This way our interface will need only three I-O lines from your microcontroller to drive the entire display, even if you add more modules to make it longer, only add more shift registers, but still you will need only three I-O lines, Data, Clock and Latch.

I would use this configuration, for this project. To select a row the corresponding bit of row register will be set to 1, and to select a column the corresponding bit of shift register will be set to 0. I have used 74HC595 and its I-O lines have sufficient current handling capability to power an entire row with 16 LEDs turned ON. You can add the transistor buffers for rows if you need to add more modules together as this will need more energy to light many LEDs in the same row. I personally did not felt the need of transistor buffers for this single bi-color LED module, although adding transistors improves brightness.

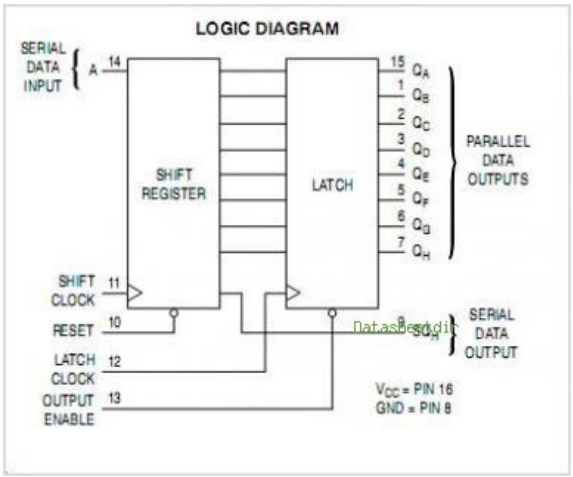
So here is our final schematic of the display. Note this only the display circuit, and does not include the microcontroller circuit. As we discussed previously I do not want to restrict users to any



one microcontroller, so that interfacing remains open.

In the above schematic the matrix is shown as two separate modules, whereas in fact these are packaged as one module, but must be considered as two separate modules with anodes of rows as common, and cathodes separate. Current limiting resistors have been shown here as 100 Ohms, on both modules, as previously said we need to reduce the brightness of red little bit more than green, so better use a slightly higher resistors for the red module, I used 220 Ohms for red and 100 Ohms for green.

Understanding 74HC595 Shift Register



Since our display is going to depend heavily on the shift registers, it will be appropriate here to get a little review of this component. Shift registers are also called serial to parallel converters, they take in serial data on a single data line, and show the results on its output as a complete byte.

The 74HC595 shift register has a series of 8 memory locations that can pass the bits sent to first memory into the next one, just like a “push” when new bit arrives the all bits are shifted one step to next level.

Latch is a special memory that when activated takes the snapshot of shift registers and show them on outputs. After that if the shift register values are changed the change is not reflected on the output pins until the latch is activated again. This is a good mechanism, as this will not show

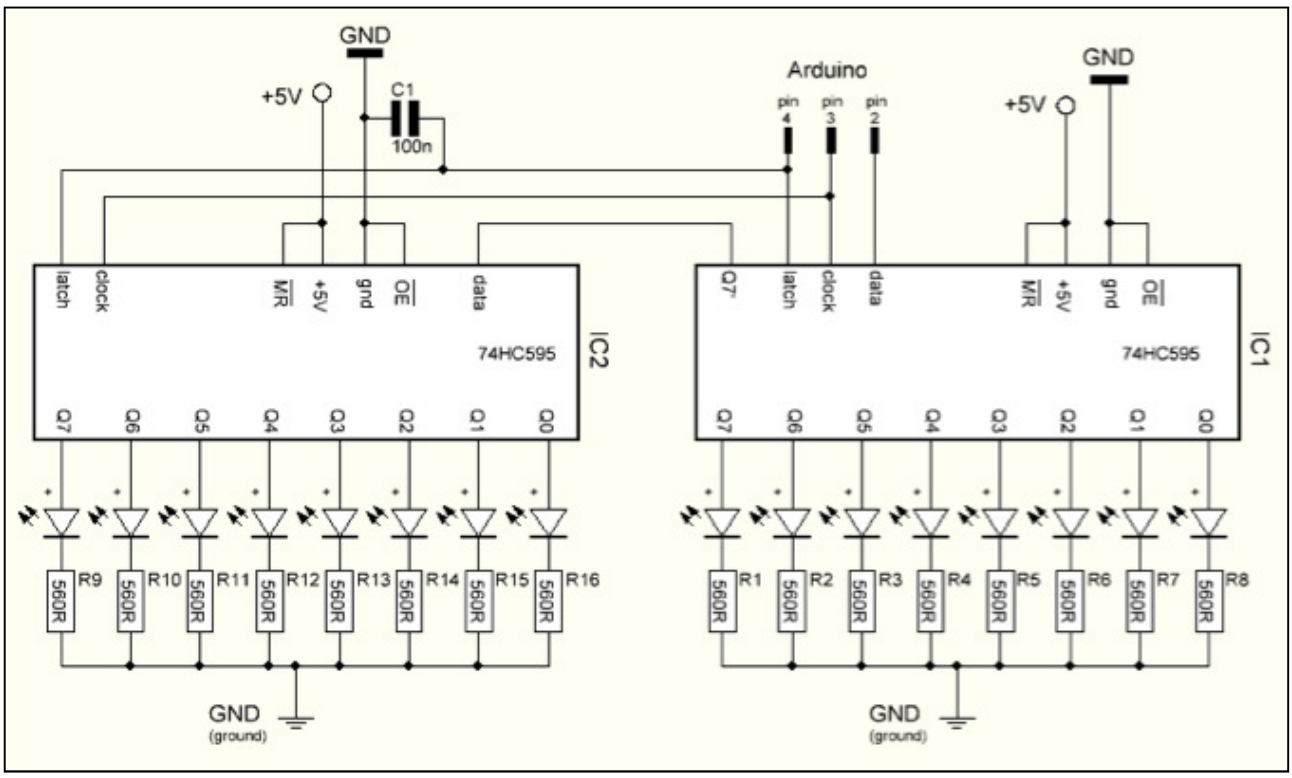
up the shifting bits as garbage on our output, only when we are finished sending the data, we want it to be reflected on outputs. The outputs of latch memory have an additional control line called "output enable". When this line is low the memory in latches as 1s and 0s is reflected on outputs, when this line is high the output pins are practically disconnected from memory, they are neither 1, nor 0. Simply speaking they are in third state, which is called indifferent. We have permanently tied this pin on all shift registers to GND. So that the output is always enabled. To be honest if you want to control the entire brightness of display through software you can connect this pin to a PWM output to control the brightness, this would however need another IO line of microcontroller. Nevertheless it's a good idea to have separate Output enables for red and green to individually control the brightness. The RST pin is a reset pin and when given a small pulse of logic 0 will set all shift registers to 0. we have connected these pins to logic 1 permanently so we do not want to reset the chips.

Now we are left with three pins DS is data pin, whatever data present either 1 or 0 will be taken into first memory location of the shift registers. This happens when the shift clock line is given a pulse of 0 to 1 and back to 0. This pulse will also shift the contents of other registers to next levels by 1 state.

All this shifting will be taking place inside the shift registers and the results will not appear on output pins. When the necessary data has been stuffed in a pulse of 0 to 1 and pack to 0 on latch pin will copy the shift register data on output pins and keep it there no matter what is happening down into the shift registers.

Shifting Data to next Chip

Now that we can shift data bits inside the 74HC595 shift register, how to daisy chain them, so that data bits can be shifted to next chip. This will effectively convert one 8 bit shift register into 16, 24 or even 32 bits shift register.



The shift register has 8 outputs, Q0 to Q7 and an additional Q7' output. This is not connected to latch but a direct extension of the internal shift registers. So when 8 bits have been transferred,

shifting in of 9th bit will cause 8th bit to appear on Q7' pin, which can be fed in directly into the DS pin of next chip. The shift clock and latch pins of all pins can be connected together and clocked simultaneously. In this way an endless daisy chained 74HC595 can be connected together to get a long parallel outputs.

74HC595 can be clocked data at a maximum frequency of 100MHz, which is more than what we need to update our display.

Microcontroller Setup

This publication is not meant to go into details of setting up your microcontroller board, you can use any microcontroller with this. All you need is three lines of digital I-O from controller, one for data, one for data clock and one for latch.

You are free to use any controller, this publication however will use Microchip PIC microcontroller PIC18F2550. This controller has many more pins than needed for interfacing with this display board, other pins can be used for interfacing with other chips like external EEPROM, real time clock or USB communication from PC to get new messages for display etc. as far as driving the display is concerned you can get away with as simple as 16F84 or even smaller controller.

I assume you are aware of programming the microcontroller, using your preferred programmer and development board. I will connect the display board to PIC as:

DS pin : RC0

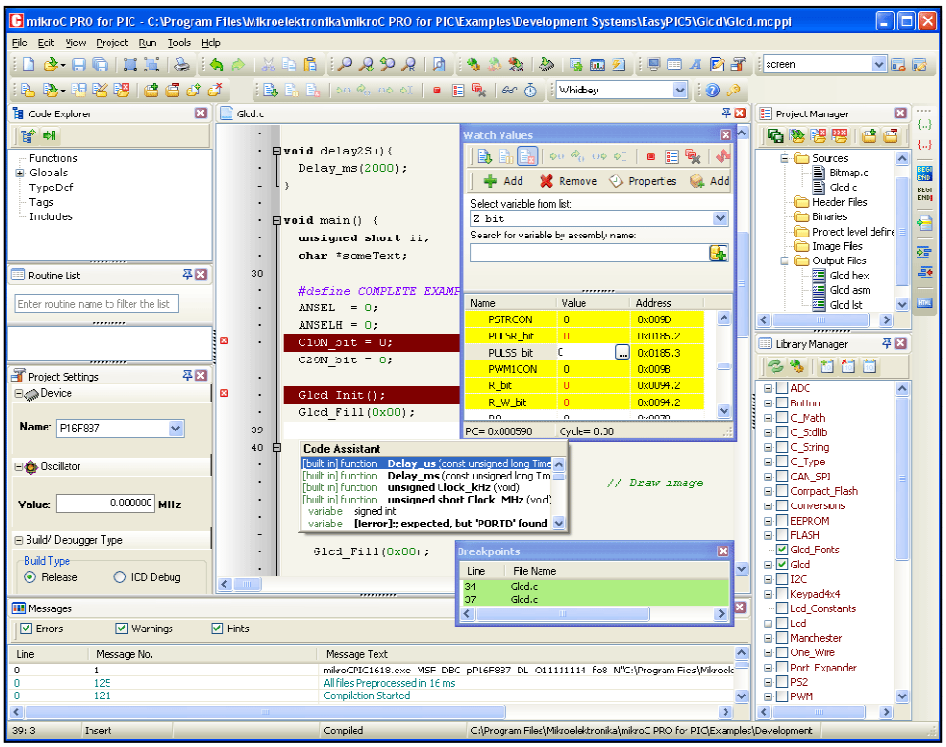
CLK pin : RC1

Latch pin: RC2

The display board will be powered from 5V from the microcontroller board. In case you want to give it external 5-12V supply connect GND to the MCU board as well.

Compiler

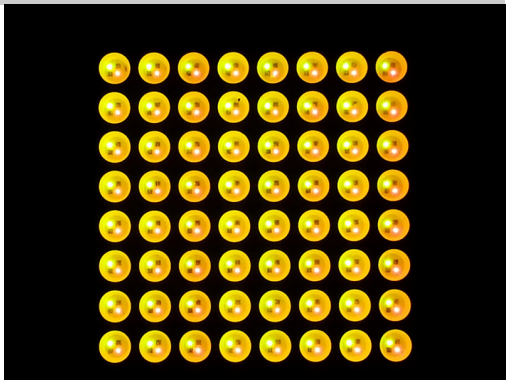
You are free to use any compiler you are comfortable with, may it be BASIC language, or C. The concepts mentioned here will be generic applicable to all compilers. There can be little differences among compilers how the address the ports and port pins etc. you can therefore accommodate these changes as required. I will be using Microelectronica MikroC V6.0 compiler and integrated development environment. MikroC can be downloaded from www.mikroe.com, the demo version is sufficient for this example project however.



Testing Display

Our display board is now ready and we are position to give it a test run. As you know we have three shift registers in series. We will always be shifting out 3 bytes of data, first byte will be for green cathodes, second byte will be for red cathodes and third byte to select the anodes common row. The red and green bytes will contain logic 0 to activate a column, and the row select bit will be logic 1 to power a row. Microcontrollers usually have an SPI module, which is called serial to parallel interface that can accept a byte of data and shift out the byte one bit at a time from a designated SPI data pin along with necessary clock signals from another pin. As we want to make our software as much generic as possible we will write our own custom code to accept 3 bytes of data and then shift them out one bit at a time along with clock signals. The data will be sent using three globally defined pins, in our case RC0, RC1 and RC2.

```
void Shout(unsigned char d)
{
short j;
for(j=0;j<8;j++)                    // Shift out 8 bits
 {
  PORTC.B0= d & 1  ;                // test least significant bit if its 1
  PORTC.B1=1;PORTC.B1=0;            // clock
  d=d >> 1;                         // shift bits to right by 1 to get next bit
 }
}

void main() {
TRISC.B0=0;TRISC.B1=0;TRISC.B2=0;   // make the 3 pins output
PORTC.B0=0;PORTC.B1=0;PORTC.B2=0;   // initialize values
Shout(0);                           // turn on all red
Shout(0);                           // turn on all green
Shout(255);                         // select all rows
PORTC.B2=1;PORTC.B2=0;              // Pulse the latch pin
}
```

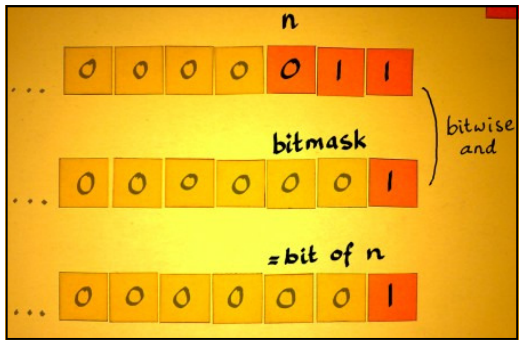


This test program should produce an output as shown in this image. If you look closely you will appreciate that each pixel has two light sources, one red and other green.

Lets dissect this very basic program. In main function we have simply configured the tree pins connected to display as output pins and initialize their values to 0. next we call the shout() function that is the heart of this software. First we shifted out a value of 0, then another 0 and finally 255. this will push first 0 to all cathodes of red, next 0 to all cathodes of green and last 255 to all anodes of all rows. This should light up all 128 LEDs on display and it should appear yellow, due to mixing of red and green colors.

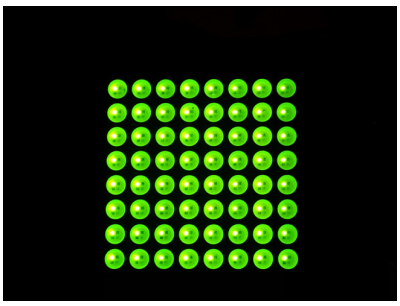
Now we need to discuss the Shout() function, that how it is sending out the bits from a byte. It is accepting a byte of data in local variable d as an unsigned data because we want to use all 8 bits.

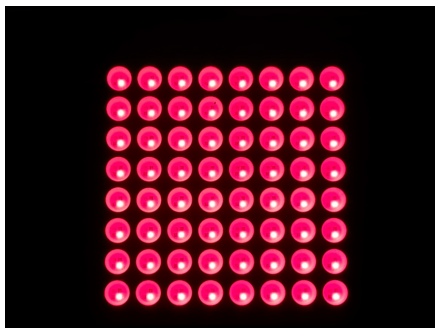
Then we want to shift out the least significant bit of the variable. To test the least significant bit if its 1 or 0 we do a binary operation of AND with number 1 (0000001). The answer will be 1 if the least significant bit of d is 1 and 0 if this bit is 0. we set the result of this test into the data bit, PORTC.B0 after setting the bit to appropriate level we give a clock pulse of logic 1 and back to 0 on PORTC.B1. This will transfer the first bit to shift register. Now we want to repeat the same process to second bit of variable d. we shift the variable d to right by 1, d >> 1, this will move the bits in variable d by 1 place in d and now the second bit is in least significant bit position. We repeat the same process again and now pushes second bit into shift registers. We do it 8 times so that entire byte is shifted out. After calling this function 3 times from the main program, we give a pulse to the latch pin and that will transfer the 24 bits data to shift register pins and then to the display module.



Testing Red and Green Modules

Sending out 0 on first shift register will select all red columns. Sending 255 to the next shift register will turn all green off. You can reverse this by sending 255 first and 0 to second, this will turn on all green LEDs.

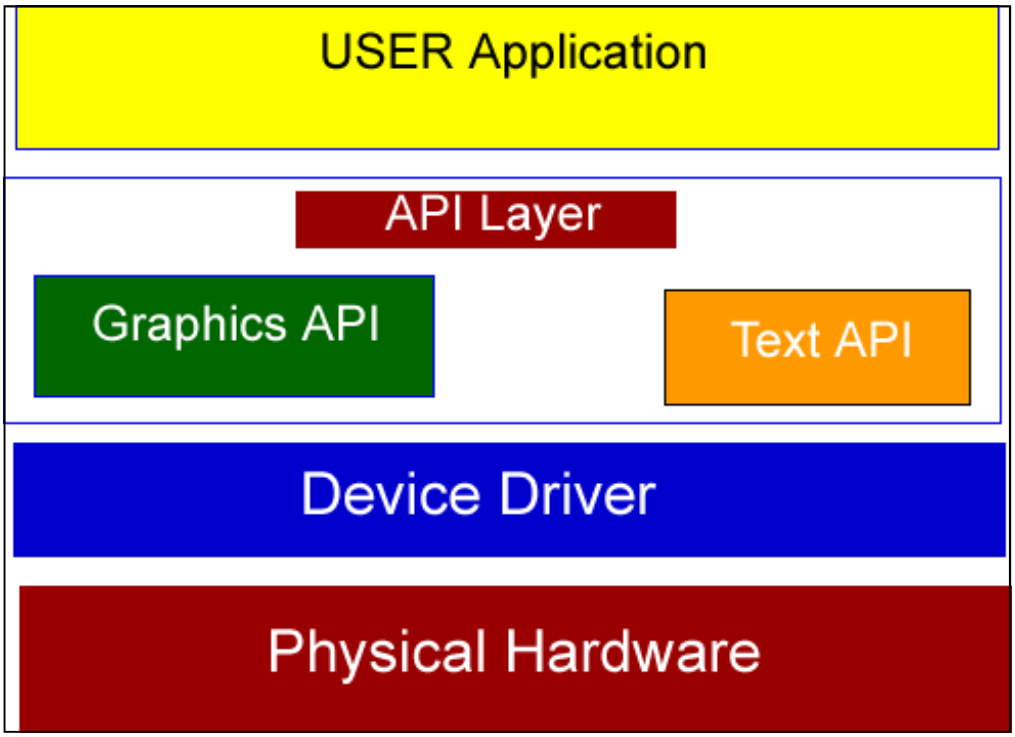




```
void Shout(unsigned char d)
{
short j;
for(j=0;j<8;j++)                    // Shift out 8 bits
 {
  PORTC.B0= d & 1  ;                // test least significant bit if its 1
  PORTC.B1=1;PORTC.B1=0;            // clock
  d=d >> 1;                         // shift bits to right by 1 to get next bit
 }
}

void main() {
TRISC.B0=0;TRISC.B1=0;TRISC.B2=0;   // make the 3 pins output
PORTC.B0=0;PORTC.B1=0;PORTC.B2=0;   // initialize values
While(1) {
Shout(0);                           // turn on all red
Shout(255);                         // turn OFF all green
Shout(255);                         // select all rows
PORTC.B2=1;PORTC.B2=0;              // Pulse the latch pin
Delay_ms(2000);
Shout(255);                         // turn OFF all red
Shout(0);                           // turn ON all green
Shout(255);                         // select all rows
PORTC.B2=1;PORTC.B2=0;              // Pulse the latch pin
Delay_ms(2000);
}

}
```

Best Programming Practices

Well by now hopefully you have tested your hardware, with a very basic and rudimentary code. Remember the underlying code will remain the same, or almost same, but we will need to organize it in a manner that is helpful in making it usable by many different level of users as well as adapt the same code for larger displays with minimal or no changes at the user code. What we are going to accomplish here is to break the code into many different levels or layers, sitting one on top of the other. This kind of approach is truly professional. Although its very easy to build your application in a single file, it is more professional to distribute the code into at least three levels.



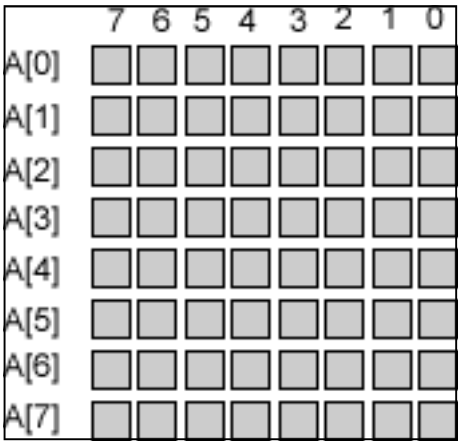
This diagrammatic representation shows the hierarchy of the code libraries. The core idea of this type of organization is to abstract the hardware level details from end users or even intermediate library developers. In any hardware manufacturing service, the electronics engineers work independently from the software developers, though in contact but they design the hardware according to the specifications and cost constraints and the software developers have to adapt the software accordingly. The super or end user of the hardware, who in our case is also a programmer and want to use the device in his projects must be facilitated to use the device without core understanding of the hardware details. A second and equally important aspect is, if we change the hardware internally, the end user must not have to re-write his applications, all he needs to do is plug in the appropriate device driver layer, recompile the application and it should work with new hardware.

We will therefore work in this fashion, we have got the hardware ready, and know its interfacing requirements, as we experimented in previous examples, now lets begin step by step. So the first layer to develop is the device driver, which is in immediate contact with the hardware physically. This driver will be built to expose the basic functionality of driving the display and a few

basic functions to drive it.

Video Memory

Previously we have seen how we can send data directly to the shift registers and then coordinate them with the row scanning to make an image. Remember we can not turn all rows ON at the same time, as we can send data over two shift registers for one row at a time. We therefore need a multiplexing system, where we send data for one row, activate that row, send data for next row, deactivate previous row and turn on next row. Thus we will be sending data for one row at a time. All this will be done so fast that our eye will not be able to perceive the difference and see the image as one. In order to do that we abstract the 8x8 matrix as a memory map of 64 bits, each bit representing one LED. Since this map will be in memory variables, they can be manipulated in anyway we like by our software. The device driver will be smart enough to copy the contents of this memory map onto the physical display continuously. Since we have two 8x8 matrices in one piece, we will consider them as separate maps. All we need is a 2 dimensional array with 8 rows and 8 columns. We need two such arrays, one for red and other for green. An 8x8 array of character type will require 64 bytes of RAM and both arrays will take 128 bytes. So make sure your controller has this much RAM available.



We can simplify this down a little bit, as right now we are not going to implement varying intensities of colors, we just need that red LED in a particular row and column is ON or OFF, same is the case with green. This can be represented by a bit only. So we need 8 bytes, one for each row. The bits in each byte will represent the columns.

The array rows can be accessed using the index number and columns can be accessed by individual bits in the selected array. Note since the values stored in bit pattern will be sent to cathodes shift registers, a value of 0 will turn on the corresponding LED. The usual practice however is to represent an ON led with logic 1. we will therefore implement this later idea, that in the memory map the LED that needs to be turned ON will be indicated by a logic 1 in the corresponding bit and a logic 0 will turn it OFF. Our device driver will invert the bits before sending them to the shift registers. This is the first abstraction we have decided, to make the user, or even device driver writer to think the logical way of logic 1 is ON and logic 0 is OFF.

So we will have two such arrays in memory, one for red and the other for green, for any given row, one byte from each array will be sent to the respected shift register.

Selecting a Row

Now we need to address the data for third shift register, which is going to select a row in the matrix. We have total of 8 rows, and we have to select one at a time. A logic 1 in the corresponding bit of shift register will select the row. To select first row we need to send 00000001 for next row it will be 00000010 for third 00000100. Notice how the 1 from bit 0 is being shifted to left one at a time. We can simply implement this by setting a decimal value of 1 to the variable and then shifting left by 1 every time we want to select next row. This is a workable solution, but what if we have decided to increase the rows to 16? The eight bit microcontroller can not do 16bits math. The solution is to store the equivalent values in another 8 items array, and using a counter variable to use the respective value every time we want to select a row.

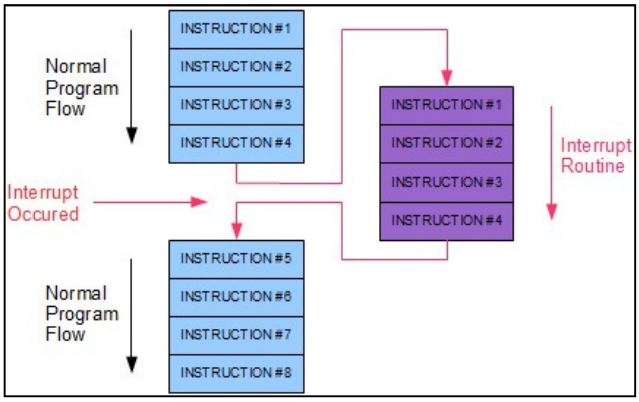
Automatic Frame Refresh

In order to keep the display updated, we need to copy the map arrays to the display matrix peri-

odically. This can be done easily within a loop. However when our program or application needs to do another job, like reading serial port, or some other task, this loop will be interrupted and display will not look uniform. The display must be refreshed automatically at pre-defined intervals. This is implemented using interrupt system. Most controllers have this interrupt system, and can be configured on many different triggers. We will use Timer0 of PIC microcontroller to generate a trigger periodically when we will update the display.

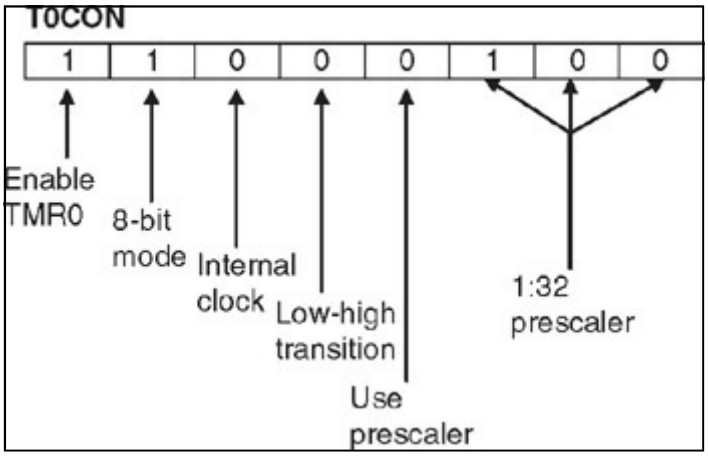
An Overview of Interrupts

Although this is in itself a long subject, but in this book our objective is learning and exploring the core technologies. It will be therefore very much in time to have a brief overview of the interrupts. As you know any processor can do only one task at a time, if we want to manage multiple tasks we need to switch periodically among tasks. This is timed by an internal timer or some other trigger. Once the interrupted task is completed the control returns back to the main procedure from where it was interrupted.



We will be using Timer0 of PIC microcontroller to trigger this interrupt. On every interrupt we will send one row data to our display. In PIC microcontrollers there are many registers that need to be configured in order to set the interrupt system. Moreover we have to make a function that will handle the interrupt routine.

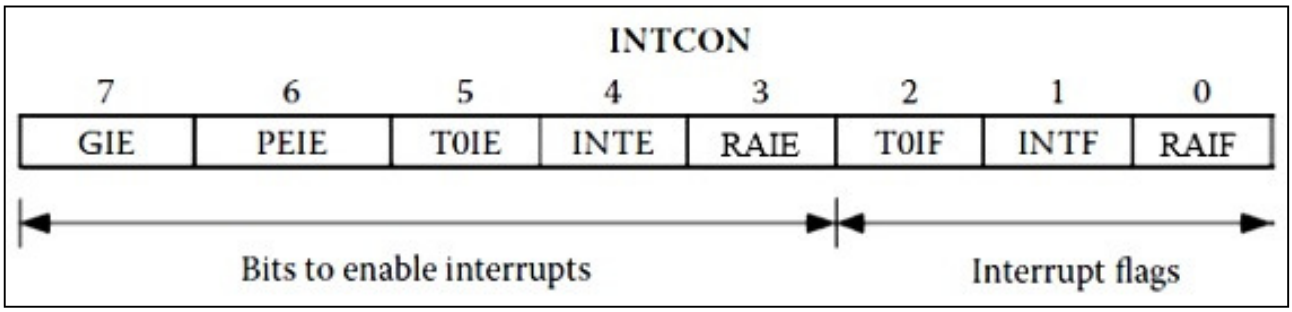
Different families of microcontrollers have different registers to set the behavior of these timers. I am using PIC18F2550 @48MHz so all my calculations here will be according to this controller. You can adjust the values and registers accordingly.



T0CON is the first register to configure the behavior of timer0. this is an 8 bits register and setting each bit has specific meaning for its configuration.

In PIC18F2550 Timer 0 can be configured as 8 bit timer or 16 bits timer, the timer can also be configured to take pulses from internal oscillator or pulses from an external pin. It can also use a prescaler, which is actually a pulse divider to slow down the count. We will be using this timer as 16 bits so that after 65536 pulses it will generate an interrupt. We shall not use the prescaler.

Second important register to configure is the interrupt system configuration register. This is named INTCON. This register controls the main interrupts. As you can see it has bit 5 named T0IE bit, setting this bit to 1 will enable interrupts on Timer0. Bit 7 called GIE is general Interrupt

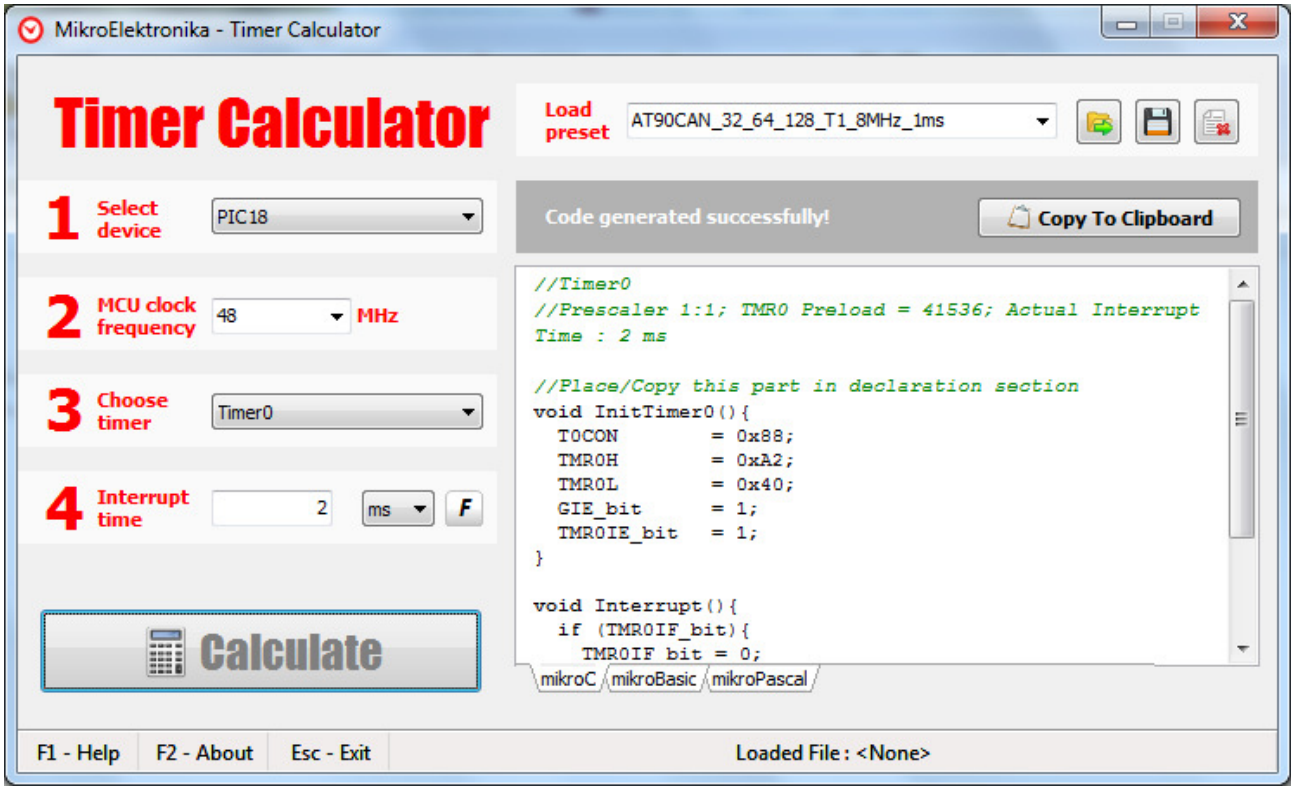


enable, and setting this bit to 1 will enable the entire interrupt system. This is actually the main switch of interrupts. Also notice bit 2, T0IF. This is a flag which is set automatically to 1 when the Timer 0 has generated an interrupt. We have to clear this bit back to 0 in our interrupt handling routine to indicate that interrupt has been serviced.

Frame Rates

Now that we have talked about the interrupts, and we know we can configure an automatic system to refresh the physical display from contents in our shadow memory or you can call video memory. The question arises how frequently we should do that. If we do it too frequently, we tend to spend extra time in servicing interrupts and our main application will perform poorly. If we do it too slowly the display will look jerky and might be flickering. So we have to find an appropriate frame rate. The human eye can not detect a change if frame rate is above 30 frames per second. We will choose a rate of 50 to give our display more stable look. So we need the display to be refreshed 50 times per second. One frame will be updated in 1/50 = 0.02 seconds or 20mS. As we will be updating one row on every interrupt the 20mS time would be divided in 8 rows, which is 2.5mS; we can round it to 2mS that will give a slightly higher frame rate but ok for our purpose.

The firing of interrupt depends upon the clock speed, and the length of counter. The interrupt will fire when counter rolls back to 0. This is 256th pulse on 8 bit timer and 65536th pulse on 16 bits timer. Calculating exact number of ticks required to generate a 2mS interval needs little math. In general for the above speed, timer 0 in 16 bit configuration will overflow and generate an interrupt in 5.4mS. We can reduce this time by preloading the timer registers with some values so that they don't start off from 0, but from a fixed value and therefore will reach 65535 earlier. Fortunately microelectronica have a small utility to calculate the best settings for timer interrupts.



As you can see in this image it has calculated to stuff a value of 41536 (0xA240) to achieve an exact 2ms interrupt.

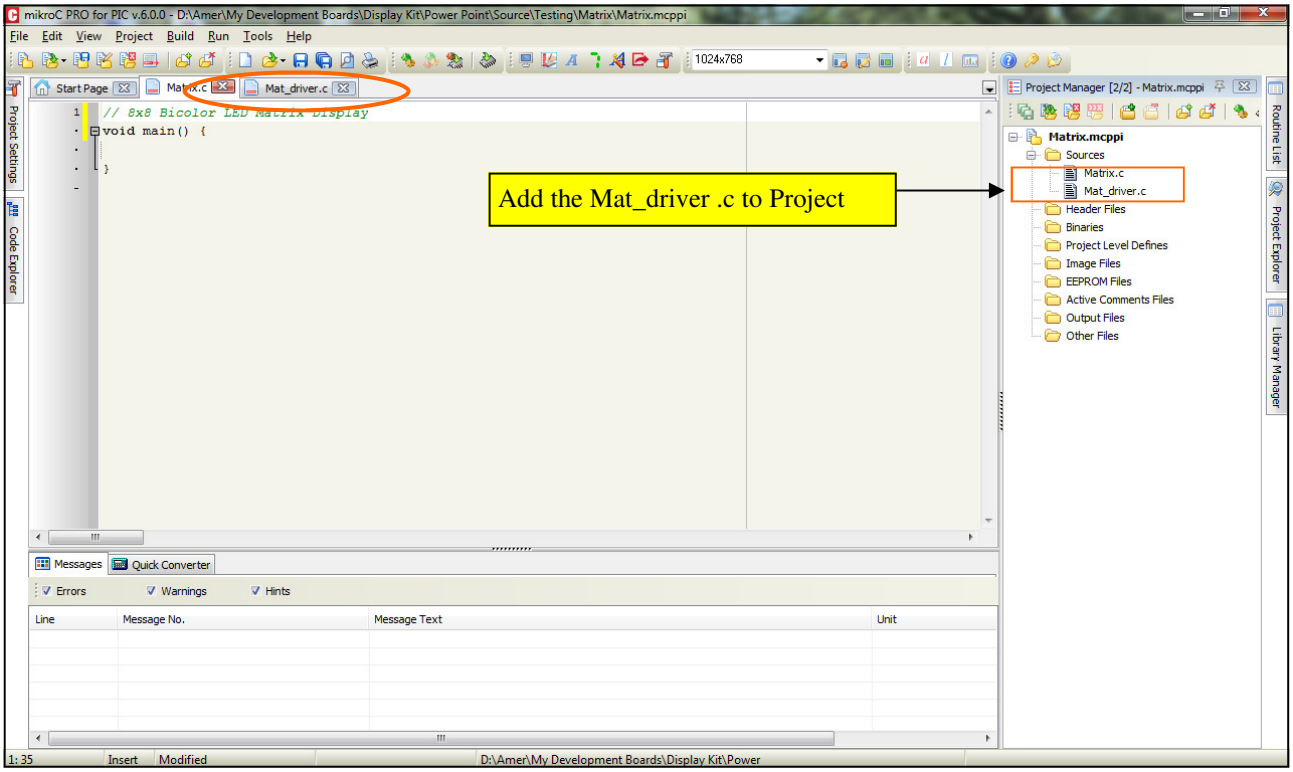
Now with all this basic background knowledge we are ready to code our display driver.

Level 1: Matrix Device Driver

Now that we have all the background knowledge and understanding we are ready to start coding. First we are going to write the core device code, that is the display driver. This part will be responsible for making its video memory arrays, setup the interrupt routines and copy the contents of memory into the shift registers, step through the rows and keep track of the row count. It will encapsulate the display and give us a logical view of the display as a matrix of rows and columns. So that we will be able to address each pixel or LED by its coordinates, and then based upon that basic hierarchy we will build our next level of library.

Start MikroC integrated development environment and create a new project with settings according to your microcontroller hardware. The display device can be connected to microcontroller on any three digital lines, and our device driver software will have the capability to adapt accordingly. For simplicity I would connect the Data line to RC0, Clock line to RC1 and Latch line to RC2.

When you start a new project a new text editor opens with main() function. Although we don't need this for our device driver, but will be used to test the things. In your project add a new file named “Mat_Driver.c”; this will be our device driver and contain all the necessary code.



Our project is named as Matrix therefore Matrix.c will be the file with main function to test the display. The underlying code will be developed in separate files. Now lets come to the Mat_driver.c file. Also add a “matrix.h” a header file that will contain the prototype definitions of our functions and the parameters they need.

```
#include "matrix.h"
// 8x8 Bicolor LED matrix display driver
// SPI Mode Data, Clock , Latch
// Three Bytes : Row Select, Green, Red
// red is least byte
//  mat_row, mat_g and mat_r are reserved global arrays these names can not be
      used in any other part

extern sfr sbit Mat_DS;
extern sfr sbit Mat_CLK;
extern sfr sbit Mat_LAT;

extern sfr sbit Tris_Mat_DS;
extern sfr sbit Tris_Mat_CLK;
extern sfr sbit Tris_Mat_LAT;
unsigned char mat_row[8]={1,2,4, 8,16,32,64,128 };
unsigned char mat_i=0;
unsigned char mat_r[8]={0,0,0,0,0,0,0,0};
unsigned char mat_g[8]={0,0,0,0,0,0,0,0};
```

Start this code, we have included the matrix.h file that will later contain the function definitions as we proceed.

```
extern sfr sbit Mat_DS;
```

And other similar statements indicate that we have a variable or symbol named Mat_DS which is a bit type and points to a bit in some special function register on the microcontroller. The actual value of this bit is not defined in this file, and has been termed as extern, meaning some other calling program will define it. We will define the connections of our device as Mat_DS, Mat_CLK and Mat_LAT for data, clock and latch pins. Corresponding to them are the TRIS register pins that define the direction of these pins, either as input or output, we need them as output in any case.

```
unsigned char mat_row[8]={1,2,4, 8,16,32,64,128 };
unsigned char mat_i=0;
```

Next we have defined an array of unsigned characters, having eight elements. They have been assigned the values as shown. If you convert these values to binary numbers you will see they have a logic 1 which is shifting to left. These values will be used to select the row, and corresponding mat_i is an index to track the current row.

```
unsigned char mat_r[8]={0,0,0,0,0,0,0,0};
unsigned char mat_g[8]={0,0,0,0,0,0,0,0};
```

Similarly there are two arrays mat_r[] and mat_g[] both with 8 elements each. These are the video memory for red and green matrices. Each element corresponds to a row and eight bits in each will correspond to pixels.

Note mat_r[], mat_g[], mat_row[] and mat_i are all global variables, as they have been declared outside any function. They can be accessed from any part of the program, and no other part of program should try to redefine them.

```
// Initialize the matrix, set Tris registers, and setup the interrupt system on
Timer 0
void Init_Matrix() {
 Tris_Mat_DS=0;         //set as output
 Tris_Mat_CLK=0;        //set as output
 Tris_Mat_LAT=0;        //set as output

 Mat_DS=0; Mat_CLK=0; Mat_LAT=0;    //clear all pins

  // setup interrupt on timer 0 2ms ineterrupt
```

```
    T0CON         = 0x88;     // start timer, 16 bit, no prescaler
    TMR0H         = 0xA2;     // Initial values to get interrupt at 2mS
    TMR0L         = 0x40;
    GIE_bit       = 1;        // Enable General Interrupt
    TMR0IE_bit    = 1;        // Enable Timer0 Interrupt
  }
```

Now the first function we want is to initialize our display driver. This function will set the directions of pins to which display has been attached and also set the Timer0 and its interrupt system.

Now since we have added a function to the driver file, add its prototype to the matrix.h file:

```
void Init_Matrix(void);
```

Next we need a function to transfer three bytes of data into the shift registers. This is the most important function as it accepts three bytes of data as parameters, and transfers them as one bit each into the shift registers. Notice that the data in shift registers for red and green have to be logic 0 to make LED ON. Whereas we want it to be logic 1 in our image map, so we have inverted the data before sending.

```
// Shift out the data to three shift registers
// the active column is 0 on shift registers whereas it is 1 in our array
// therefore the values are inverted first before shifting out

void Shout(unsigned char row, unsigned char green, unsigned char red) {
 short j;
 green = ~green;         // invert bits because led is on when bit is 0 (sink)
 red=~red;
 for(j=0;j<8;j++)        // Shift out red First
 {
  Mat_DS= red & 1  ;      // test least significant bit if its 1
  Mat_CLK=1;Mat_CLK=0;   // clock
  red=red >> 1;          // shift bits to right by 1 to get next bit
 }

 for(j=0;j<8;j++)        // Shift Out Green
 {
  Mat_DS= green & 1  ;      // test least significant bit if its 1
  Mat_CLK=1;Mat_CLK=0;   // clock
  green=green >> 1;          // shift bits to right by 1 to get next bit
 }

// The row bits are 1 to select a row, therefore they don't need to be inverted
 for(j=0;j<8;j++)        // Shift Out Row select
 {
  Mat_DS= row & 1  ;      // test least significant bit if its 1
  Mat_CLK=1;Mat_CLK=0;   // clock
  row=row >> 1;          // shift bits to right by 1 to get next bit
 }

 Mat_LAT=1;Mat_LAT=0;        // Clock Latch to appear data.

}
```

```
green = ~green;
```

This will effectively convert all 1s to 0s and all 0s to 1s. Now add the prototype of this function to the matrix.h file:

```
void Shout(unsigned char row, unsigned char green, unsigned char red);
```

Our next function is the heart of this driver, update display; this function will be called from the

```
// called every 2ms from the interrupt and shifts out one row of red and green
matrix array
// increments the row index mat_i
void update_matrix(){
    TMR0IF_bit = 0;         // reset timer values, calculated for 48MHz
    TMR0H         = 0xA2;
    TMR0L         = 0x40;

    Shout(mat_row[mat_i],mat_g[mat_i],mat_r[mat_i]);
     mat_i++;
     if(mat_i==8)
       mat_i=0;
}
```

interrupt service routine, every 2mS. This function will keep a track of the current row in mat_i variable and transfer the data from mat_row, mat_r and mat_g arrays according to this index. After display the current row it increments the index number so that on next interrupt next row is selected. When the number reaches 8 it resets it back to 0. This this function will be called automatically and will copy the image map from video memory to the physical display. Notice this function also clears the interrupt flag and sets the counter values back to initial values calculated to give 2mS interrupt.

Also add its prototype to matrix.h file.

```
void update_matrix();
```

This completes our basic display driver and it is ready for testing. What this basic driver should do? This will continuously update the contents of display memory on LED matrix. As we had talked previously this approach should abstract the hardware details from user or programmer and allow minimum access to internal architecture. Right now if we want to manipulate the display we need to write something directly into the display memory, and for that we must know the array names, as well as their dimensions. Just to test the driver lets fill the red and green arrays with some known values, like 255 for all ON, 0 for all OFF. It will be more convenient to have another function named fill() to accept two parameters and fill the corresponding matrix arrays.

```
// Basic function to fill the Video RAM with pattrens of red and green

void fill(unsigned char r,unsigned char g) {
 short i;
 for(i=0;i<8;i++){
  mat_r[i]=r;
  mat_g[i]=g;
 }
}
```

As usual add the function prototype to matrix.h file. Now our basic display driver is ready to test. In our main program we need to define the three connection pins, define an interrupt function, and call Initialize and update functions. Lets see how to do it.

```
// 8x8 Bicolor LED Matrix Display
#include "Matrix.h"
sbit Mat_DS at PORTC.B0;
sbit Mat_CLK at PORTC.B1;
sbit Mat_LAT at PORTC.B2;

sbit Tris_Mat_DS at TRISC.B0;
sbit Tris_Mat_CLK at TRISC.B1;
```

```
sbit Tris_Mat_LAT at TRISC.B2;

void interrupt() {
     if(TMR0IF_bit)           // check if interrupt on timer 0
          update_Matrix();    // call update display function
}

void main() {
Init_matrix();                // Initialize matrix function, to setup inter-
rupts
while(1) {
 fill(255,0); delay_ms(2000); // show red for 2 seconds
 fill(0,255); delay_ms(2000); // show gree for 2 seconds
 fill(255,255);delay_ms(2000);// show yellow for two seconds

}
}
```

First you must include the matrix.h file as this file contains the prototype definitions of functions we need to call, next we have to define the location of our three connections, it is important to use the names as defined in our library. Since end user will not be able to see the source code, finally you will have to tell it in your documentation the names of these three global bit connections.

```
sbit Mat_DS at PORTC.B0;
sbit Mat_CLK at PORTC.B1;
sbit Mat_LAT at PORTC.B2;

sbit Tris_Mat_DS at TRISC.B0;
sbit Tris_Mat_CLK at TRISC.B1;
sbit Tris_Mat_LAT at TRISC.B2;
```

The pin connections like PORTC.B0 etc are your own choice and you can connect to any digital I-O lines and in any combination you want, once defined here, the Mat_Init() function will take care of everything. Remember some lines in microcontrollers are analog by default, like PORTA, and PORTB in pic18F2550 so make sure you declare them as digital before using them.

```
void interrupt() {
     if(TMR0IF_bit)           // check if interrupt on timer 0
          update_Matrix();    // call update display function
}
```

Interrupt is a reserved name for interrupt function and will be called every time an interrupt takes place, may it be from timer0 or any other interrupt source.in the interrupt routine just check if the interrupt occurred due to timer0? If yes then call the update_matrix function. You can handle other interrupts in this routine like USB communication etc that we will show later.

```
Init_matrix();           // Initialize matrix function, to setup interrupts
```

This function, Init_Matrix must be called once, in main program to initialize all the connection pins, setup interrupt system and start the display driver. After that our driver is ready and now we can manipulate the display memory directly (not recommended) or through specified functions to take effect.

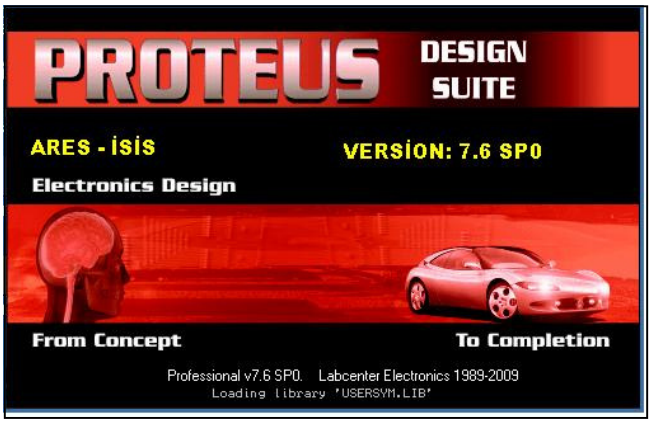
```
 fill(255,0); delay_ms(2000); // show red for 2 seconds
 fill(0,255); delay_ms(2000); // show gree for 2 seconds
 fill(255,255);delay_ms(2000);// show yellow for two seconds
```

We have called the fill function to fill the display memory with different colors, 255 is binary 11111111 so it will turn on All LEDS ON. You can try out different other combinations like,

Fill(0b10101010,0b01010101); decimal 170 and decimal 85. also try fill(240,15). You can play with overlapping bit patterns to turn on yellow color, like 0b11110000, 0b00111100.

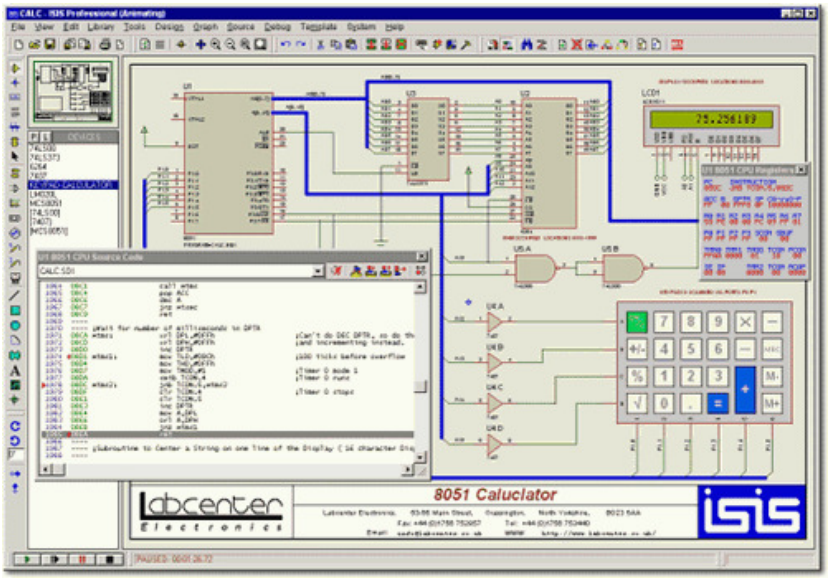
Simulation Software

There is really no-match for the fun and excitement when you get the real hardware working. I personally love this thing, but there are times when a simulator can be handy, like if you do not have access to the real hardware, or before building the hardware want to give a test run. A simulator is good for this purpose. With some limitations it can act as a good starting point, atleast for learning. Many electronics simulators are out there, each having its own merits and demerits. I would however recommend to choose one which has largest community support. This way you can not only share the ideas and design tips but also will be able to see what others are doing.



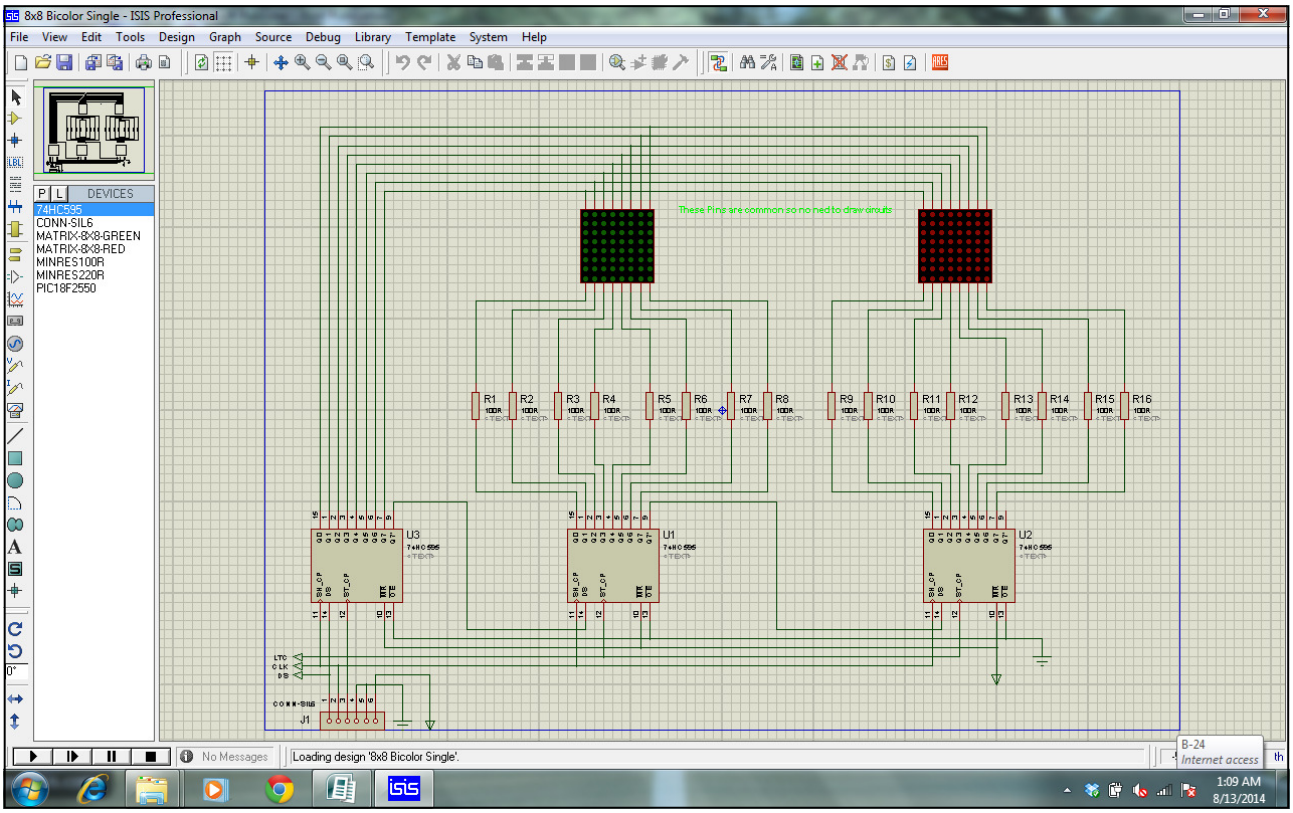
Labcenter electronics Proteus ISIS is one of the most commonly used schematic capture, simulation and PCB designing software. Many commercial as well as hobbyists use this software and share their work. One of the most beautiful things about this software is that it has a large number of microcontrollers in its library and these models can take the compiled .hex files by any compiler and run them in real-time. The real-time performance for high speed code will however require a good PC, yet it's a good choice for almost 90% of your needs.

This section is not going to be a tutorial on using Labcenter proteus software, but will give you an introduction on how to simulate or LED module on this. Proteus is a complete design suite consisting of two major modules. A schematic capture and other PCB layout. Here we are mainly interested in schematic capture module, where you can draw the schematic connections from a large library of available components. Once the schematic is drawn you can run the simulation directly from here and use many virtual tools to monitor the circuit performance and states.

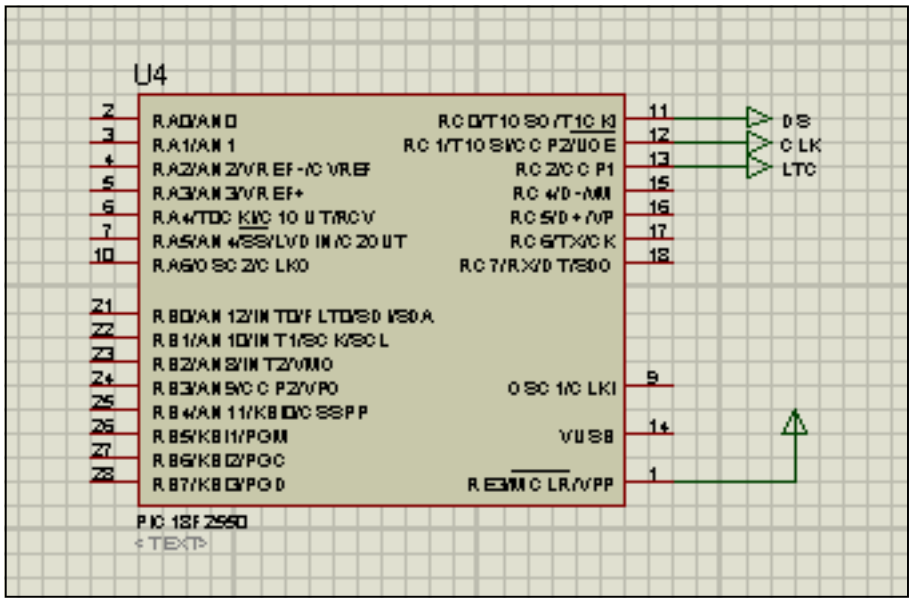


It has a huge library of commonly used microcontrollers as well that can be used in your projects. The microcontroller model when dropped into the schematic can be given settings like clock speed, and the .hex file to be loaded for simulation. The .hex file has to separately generated using any compiler software you like. In case you find something wrong with your application, just make necessary corrections to the source file, recompile it and

run the simulation again. It will automatically load the new compiled .hex file to reflect the changes. Here is the image of our basic display design. The Proteus does not have Bicolor LED

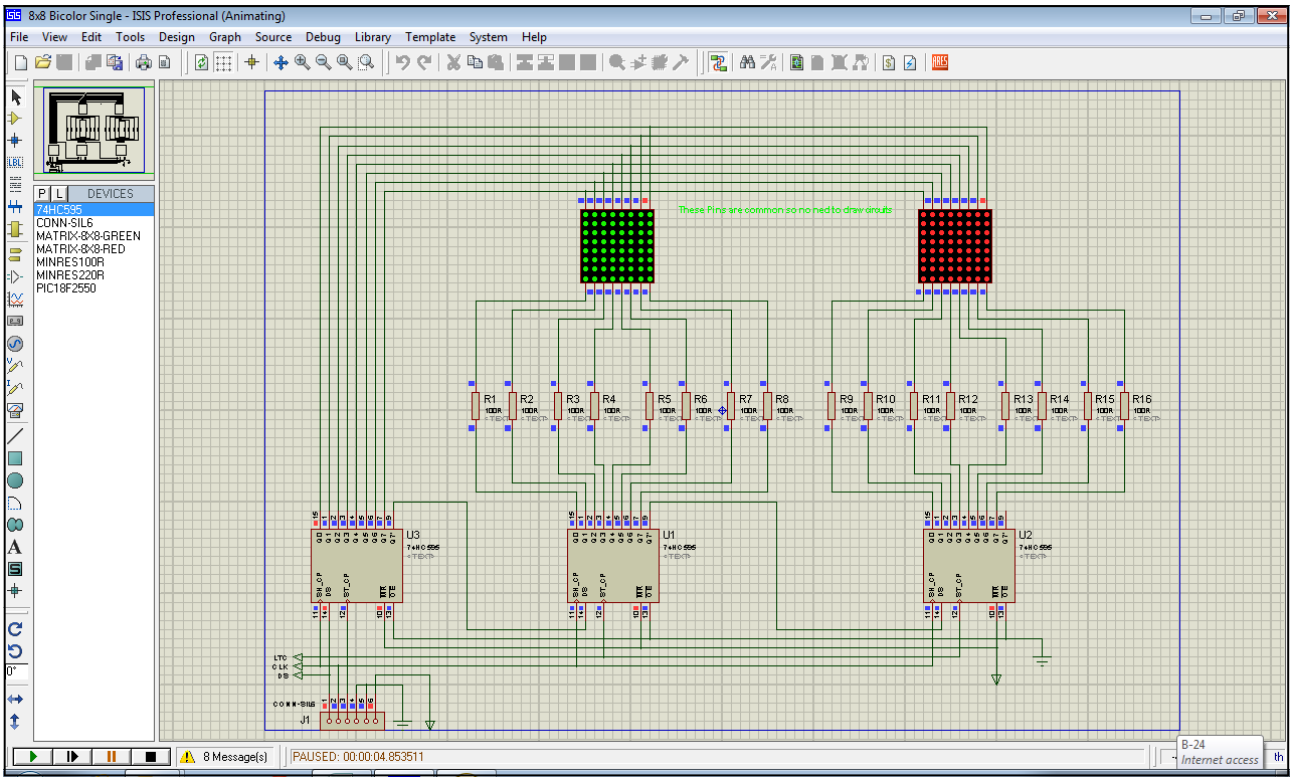


matrix simulation module, so we have used two separate Red and green modules. Although this will not show exactly what we have physically but can serve as a good model to test the functionality of code. The design has two sheets, one shown above with a display module and other with PIC 18F2550 microcontroller connected to this module.



As you can see the DS, CLK and LTC symbols connect to similar symbols on display board with RC0, RC1 and RC2 pins of microcontroller. Double clicking the microcontroller on schematic will open its properties dialog, there we have setup the clock speed as 48MHz, the same as my physical hardware is using and the .hex source file pointed to the same file generated by Mik-

roC and used to burn on my physical device. This image shows the running code paused at a mo-



ment when both led modules, red and green were ON , so that on physical hardware they would appear as yellow.

I would strongly recommend you to use this software not only for schematic drawing but also learning and experimenting before actually building the hardware. After a preliminary test if possible always go to physical hardware, as that is the ultimate destination.

Level 2: Graphics API Library

In the previous section we have completed the basic device driver, our first layer of firmware on top of the physical hardware, that is responsible for updating the display from contents of display memory. Now we are going to start our second layer, which is the API library to facilitate our end user. In this first part of our library we will write some basic code to address individual bits in the display memory and expose the functionality to end user or programmer to think of display as an array of rows and columns, individually addressable. Based upon this basic functionality we will add more useful and power commands or procedures that can be used in a variety of graphic tasks. So lets get started.

Just like you made a spate file for driver, make another file named "graphics.c". This file will contain all the necessary code for managing graphics on our display. The graphics library however depends upon the driver code, which has the declarations for video memory. Add a reference to "matrix.h" file so that we can add new prototypes to it. First we need to add a reference to the video memory, which is a common place for all data. This memory has been declared in Mat_driver.c file, so in our graphics.c we need to declare this reference as external, so that we can manipulate it, and linker would actually fix its location.

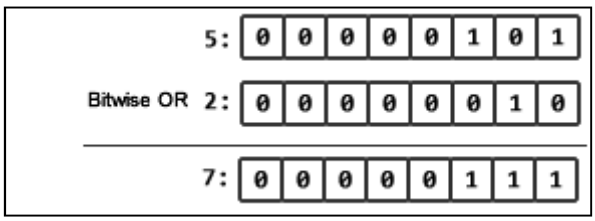
```
extern unsigned char mat_r[];
extern unsigned char mat_g[];
```

Notice we have used the same names and data types as were used in mat_driver.c file. The keyword extern indicates that the arrays have been declared somewhere else but can be used and referenced in this file.

Turning a Pixel ON

As you know physically our one row consists of an entire byte, and individual bits in it correspond to the physical LEDs on display. To turn an LED ON we must set the corresponding bit to 1 and to turn it OFF we must set it to 0. This is accomplished by bitwise operators in C programming language. Individual compilers might give you some direct methods of setting the bit values, I shall however use the generic C methodology to keep an independence from compilers.

This can be accomplished simply by OR operator. The given byte can be OR with a known mask, the bit to be set high is set to 1 in the mask, so that the OR operation will leave all other bits unmodified and set the required bit to logic 1.



Like in this example we want to set bit no 1 as high, we will OR it with mask 0b00000010 here all other bits are 0 and only bit we are interested in is 1. Now the trick is to make a desired mask. So if we want to set n^{th} bit as 1 then we need 8 different masks. Fortunately we have a shift left operator, that will shift the bits to left by n bits. Thus 0b00000001 << 5 will be 0b00100000 now OR the original number with this and we get the desired bit set.

```
unsigned char Set_bit(unsigned char ch, short n)
{
ch=ch | (1 << n);
```

```
return ch;
}
```

This dummy function shows how to do it. Clearing a bit something different. You can not use the same procedure to clear a bit. To clear a bit you have to AND with a mask, where required bit is set to 0 and all other bits are set to 1. For example to clear bit 5 we need a mask 0b11011111 now when you AND it with the data all other bits will remain unaffected and only bit 5 will be cleared to 0.

```
unsigned char clear_bit(unsigned char ch, short n)
{
ch= ch & ~(1 << n);
return ch;
}
```

In this function first we have shifted number 1 to left n times this will produce a mask similar to previous function, next we invert it by ~ operator that will reverse all bits, the 1s will be 0 and 0s will be 1s. Now we can AND this value to our number to clear the required bit.

Abstracting Colors

We have a display with two colors, and theoretically we can make many shades if we can alter their brightness. Right now we will not fiddle with brightness but to turn one color or both will give us 4 possible states, Black, Red, Green, Orange. Black is when both LEDs are OFF. Orange is when Both LEDs are ON. So to make our life easier lets give them some arbitrary numbers, or even names so that writing applications become little more easy. Lets say we define that Black=0, Red=1, Green=2 and Orange=3. We can pass these values as symbols to our functions and the function will interpret which LED to turn ON or OFF. We can do it by using #define in our header file, but there is another better way in C called enumerations.

```
enum colors {black,red,green,orange};
```

Enter this line to your Matrix.h file this will define colors as black=0, red=1, green=2 and orange=3. then we will use these names in our functions to select a color.

```
// Sets the pixel of column and row to color specified
void pset(short row,short col, short clr) {
 switch (clr) {
  case red:    mat_r[row]=mat_r[row] | (1 << col); // set red
               Mat_g[row]=Mat_g[row] & ~(1 << col);// clear green
               break;
  case green:  mat_g[row]=mat_g[row] | (1 << col); // set green
               Mat_r[row]=Mat_r[row] & ~(1 << col); // clear red
               break;
  case orange:
               mat_r[row]=mat_r[row] | (1 << col);
               mat_g[row]=mat_g[row] | (1 << col);
               break;
  case black:
                Mat_r[row]=Mat_r[row] & ~(1 << col);
                Mat_g[row]=Mat_g[row] & ~(1 << col);
                break;
  }
}
```

The row and column concept; from now onwards we will be using the concept of rows and columns, the row is y^{th} element of the video array, and column is x^{th} bit in that byte. Enter the code above in your graphics.c file, and remember to enter the prototype entry in matrix.h file. This function pset accepts three parameters, the row and column describe the coordinates of a pixel and clr the desired color. Based upon the clr value sent in calling program the function switches between one of the 4 possible states and sets the bits accordingly.

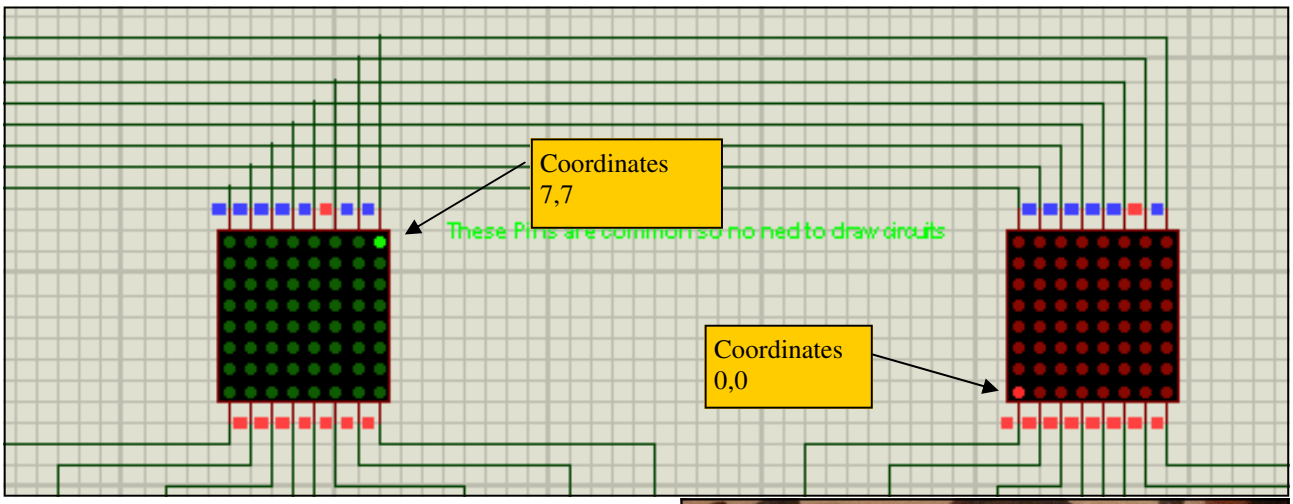
Now lets test it. We have the device driver in place and we have one very elementary basic pro-

```
// 8x8 Bicolor LED Matrix Display
#include "Matrix.h"
sbit Mat_DS at PORTC.B0;
sbit Mat_CLK at PORTC.B1;
sbit Mat_LAT at PORTC.B2;

sbit Tris_Mat_DS at TRISC.B0;
sbit Tris_Mat_CLK at TRISC.B1;
sbit Tris_Mat_LAT at TRISC.B2;

void interrupt() {
     if(TMR0IF_bit)            // check if interrupt on timer 0
          update_Matrix();     // call update display function
}

void main() {
Init_matrix();               // Initialize matrix function, to setup interrupts
pset(0,0,red);
pset(7,7,green);
}
```

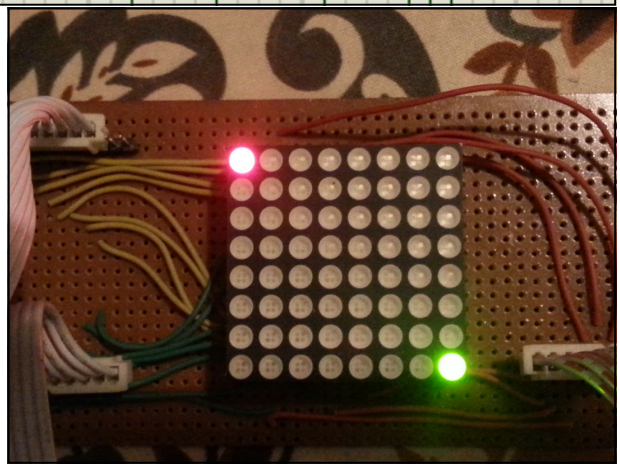
cedure to set or clear the pixel selected by row and column coordinates. Switch to you main function where all other things, like initialization and interrupt handling will remain the same, just change the code for the loop.

```
pset(0,0,red);
pset(7,7,green);
```

These are the two lines required by our end user or programmer to turn on the respective LEDs

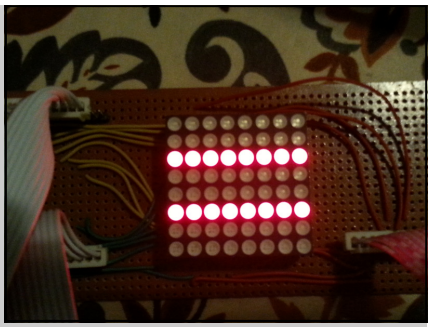


on display. Notice we have used the enumeration names of red and green to indicate color, this makes program reading more easy, we could also simply replace the clr with 1 and 2 to indicate red and green.



Here is the output from simulator, off course these will be one matrix in real hardware. Notice that row 0 is at bottom and row 7 is at top, as far as geometry is concerned this is OK, but conventionally in displays the 0,0 is top left corner and 7,7 should be the bottom right. You can easily correct

it by changing the order of rows wiring, or even in software. For now its OK for us.



```
// 8x8 Bicolor LED Matrix Display
#include "Matrix.h"
sbit Mat_DS at PORTC.B0;
sbit Mat_CLK at PORTC.B1;
sbit Mat_LAT at PORTC.B2;

sbit Tris_Mat_DS at TRISC.B0;
sbit Tris_Mat_CLK at TRISC.B1;
sbit Tris_Mat_LAT at TRISC.B2;

void interrupt() {
     if(TMR0IF_bit)           // check if interrupt on timer 0
          update_Matrix();    // call update display function
}

void main() {
short x;
Init_matrix();              // Initialize matrix function, to setup interrupts
for(x=0;x<8;x++) {
 pset(2,x,red);
 pset(5,x,red);
}
}
```

Now your top level programmer can easily play with pixels anyway he likes, can draw lines, rectangles or even bitmaps, wow great. Lets see how we can draw two horizontal lines on red ma-

```
// Draw a horizontal line on a given row from col1 to col2
void hline(short row, short col1, short col2, short clr) {
 short x;
 for(x=col1;x<=col2;x++) {
  pset(row,x,clr);
 }
}
```

trix. The code below shows how simple it is, just we had a loop with variable x, and on two rows, we have drawn single pixels at every x column. Now that we know how to draw a horizontal line, it will be a nice idea to implement this in our graphics library, for future use, so that we just give

```
void main() {
short x;
Init_matrix();              // Initialize matrix function, to setup interrupts
for(x=0;x<8;x++) {
 hline(x,0,x,green);
  }
}
```

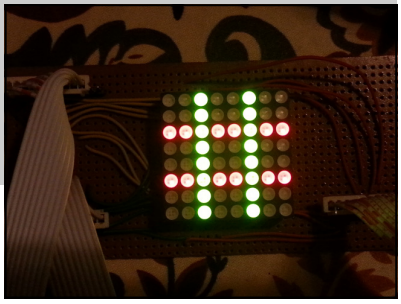
coordinates and it draws the line. Make a new function named hline in your graphics.c file and also add its prototype to the matrix.h file. The function is easy to understand, similar to you code above it, accepts the four parameters, and draws a horizontal line.

The above code draws a series of horizontal lines, of incrementing widths making you appear a traiangle. The objective is to show now by combining the pset function we have made a new function to draw the horizontal line. Similarly we can make a vertical line drawing function, and name it as vline. This time column number will be fixed and starting and ending row numbers will be used to draw the line.

```
// Draw a verticle line on a given column from row1 to row2
void vline(short col, short row1, short row2, short clr) {
 short x;
 for(x=row1;x<=row2;x++) {
  pset(x,col,clr);
 }
}
```

As usual write the code in your graphics.c file and add its prototype to the matrix.h file. I hope you understand the prototype defines the function name and its required parameters in the header file, so that the compiler knows what to expect.

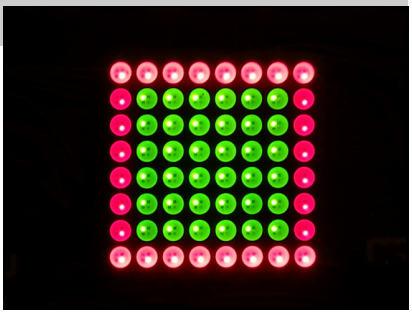
```
void main() {
short x;
Init_matrix();              // Initialize matrix function, to setup interrupts
      hline(2,0,7,red);
      hline(5,0,7,red);
      vline(2,0,7,green);
      vline(5,0,7,green);
}
```



You can see how easy it is to draw the horizontal and vertical lines, just by combining the pset command appropriately, now we can combine the hline and vline functions to draw a rectangle. We will accept two pairs of coordinates and draw two horizontal and two vertical lines and this will make a rectangle. In addition we want to be able to fill the rectangle or box with a color of our

```
// Draw a rectangle between two coordinates with border color and fills the
rectangle with a fill color
void rect(short row1, short col1, short row2, short col2, short clr, short
fill_clr) {
short x,y;
 hline(row1,col1,col2,clr);
 hline(row2,col1,col2,clr);
 vline(col1,row1,row2,clr);
 vline(col2,row1,row2,clr);
// fill rectangle with fill color
 for(y=row1+1;y<row2;y++){
  for(x=col1+1;x<col2;x++) {
   pset(y,x,fill_clr);
  }
 }
```

```
void main() {
short x;
Init_matrix();               // Initialize matrix function, to setup interrupts
rect(0,0,7,7,red,green);
}
```



choice, so we provide another parameter, fill color.

Now all you need is one line of code in your main program to draw a rectangle with filled interior. If you want to fill with black, just give black as fill color. Make sure you enter the function prototypes in matrix.h file as well.

It would be a great idea to have a function to clear whatever is on display, we can use the fill command we made for our driver for this purpose, however its more elegant to have this function

```
// Clear screen and fill the entire screen with a clr
void cls(short clr) {
    switch (clr) {
     case black: fill(0,0);break;
     case red: fill(255,0);break;
     case green: fill(0,255);break;
     Case orange: fill(255,255);break;
    }
}
```

```
void main() {
short x;
Init_matrix();               // Initialize matrix function, to setup interrupts
cls(black);
while(1){
for(x=0;x<5;x++) {
      rect(x,x,7-x,7-x,red,green);
      delay_ms(100);
   }
}

}
```

as part of our graphics library, we will name it as Cls and give it a color parameter to fill the screen with.

This code will produce a nice animation of multiple filled rectangles, being drawn again and again.

Scrolling Display

Scrolling display in either direction, will need shifting of data within the memory array. This can be accomplished directly by having access to the mat_r and mat_g arrays. The arrays have been declared in mat_driver.c file and are public. We can access them in graphics.c file but for that we must declare them as external. Alternately we can write the scrolling functions directly in the

```
// scroll the video memory to left by x positions
void scroll_left(short x,short speed){
 short i,p;
 for(p=0;p<x;p++) {
 vdelay_ms(speed);
 for (i=0;i<8;i++) {
  mat_r[i]=mat_r[i] >> 1;
  mat_g[i]=mat_g[i] >> 1;

 }
 }
}
```

driver file. Both options are equally valid. I would like to add the functions of scrolling to mat_driver.c file. These functions will later be used to scroll the text, or even graphic data.

This function iterates through both memory arrays, and shifts the bits to right. In between the shifts there is a variable degree of delay that can be used to control the speed of scrolling. Shift right operation will scroll the display to left because when shifting data to shift registers we had shifted least significant bit first. Exactly opposite will be scroll right, and very similar will be scroll up and scroll down. The scroll up and down do not need to manipulate buts, but the entire

```
void scroll_right(short x, short speed){
 short i,p;
 for(p=0;p<x;p++) {
 vdelay_ms(speed);
 for (i=0;i<8;i++) {
  mat_r[i]=mat_r[i] << 1;
  mat_g[i]=mat_g[i] << 1;

 }
 }
}

void scroll_up(short y,short speed) {
 short i,p;

 for(p=0;p<y;p++) {
  vdelay_ms(speed);
  for(i=0;i<7;i++) {
    mat_r[i]=mat_r[i+1];
    mat_g[i]=mat_g[i+1];
  }
  mat_r[7]=0;
  mat_g[7]=0;
 }
}


void scroll_down(short y,short speed) {
 short i,p;

 for(p=0;p<y;p++) {
  vdelay_ms(speed);
  for(i=7;i>0;i--) {
    mat_r[i]=mat_r[i-1];
    mat_g[i]=mat_g[i-1];
  }
  mat_r[0]=0;
  mat_g[0]=0;
 }
}
```

array elements can be shifted within the array.

Enter these functions in mat_driver.c and also corresponding entries in matrix.h file.

Reading status of a Pixel

Sometimes it is necessary to determine directly from display memory if the location of a pixel is already on or off. We need a function to return the status of pixel directly from display memory.

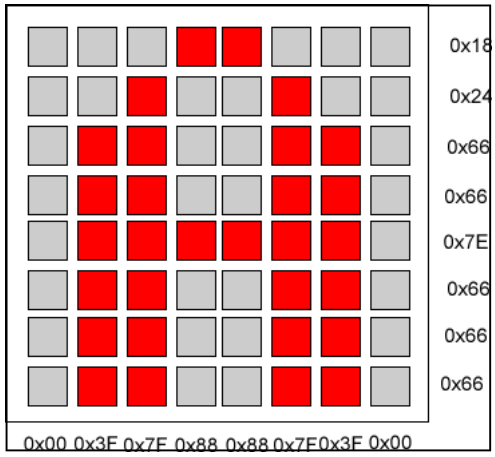
```
// get status of a pixel
char pget(char x,char y) {
 if( ((Mat_r[y] >> x) & 1) && ((mat_g[y]>>x) & 1) )
     return orange;
 if( (Mat_r[y] >> x) & 1)
     return red;
 if( (Mat_g[y] >> x) & 1)
     return green;
return black;
}
```

Level 2: Displaying Text

So far we have seen how to draw graphics using pixels on our display. Similarly we can display bitmap images, and sequential display of bitmap images makes animations. The present section is about displaying text. Text has two components or phases. One is the text itself, which consists of ASCII characters, stored in some string, the second phase is its graphic or bitmap representation. The bit map representation of a given character set is called font. You can define your own font to represent a character set. The standard text character set consists of alpha numeric characters, 0-9, A-Z and a-z along with some special character like comma, semicolon, space and period etc. first we have to decide the character set. We an decide a small limited number of characters, like only capital A-Z and ignoring special characters like $ # etc. or choose a full ANSI character set. Remember the larger the character set larger will be our bitmap and of-course the storage space for it.

Bitmap of a character

Lets first talk about bit map of a character. Most of the character display systems, like character LCDs or PC terminals use a 5x7 array of bit matrices to represent and display a character. This is a fairly good matrix size to display and represent most of the commonly used characters. We have an 8x8 matrix that can certainly accommodate the 5x7 matrix bitmap easily, or we can have the liberty of using entire 8x8 space and make more detailed bitmaps. The first step is to draw a matrix of empty squares on paper and fill the appropriate boxes with pixels you want to turn on. Now you have 8 bytes of data, each bit representing one row, or may be a column as you like. So each character for an 8x8 font will require 8 bytes of data. The entire bitmap will need to be stored somewhere, you can store it as part of the program memory as constants, or in microcontrollers EEPROM. If your hardware permits the best place is to use an eternal storage, like i2C EEPROM or and SD card. In this example we will try to minimize the external hardware dependence and will store the character map bit map in the program memory as constants.

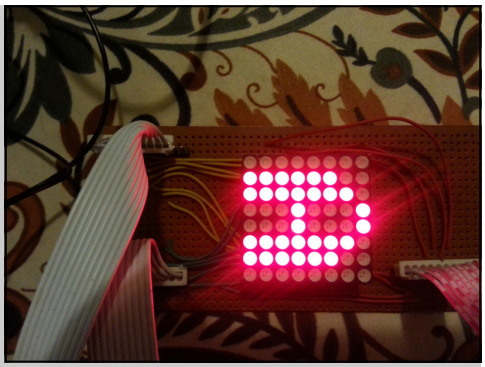


Look at the character map of letter A. internally this will be stored as text as ANSI, decimal 65, but as a display map this will occupy 8 bytes of storage, We can store the map as vertical bits, or as horizontal bits. The best option depends how our hardware is configured. Since we are scanning as rows, it will be very convenient to store the bit map as rows, the corresponding numbers are shown on the right. If we store the vertical configuration then the letter will appear as rotated on our display, as it will display the columns maps as rows. For now lets experiment a bit and see how it goes.

```
// 8x8 Bicolor LED Matrix Display
#include "Matrix.h"
const unsigned char font[8]={0x18,0x24,0x66,0x66,0x7E,0x66,0x66,0x66};
extern unsigned char mat_r[];
sbit Mat_DS at PORTC.B0;
sbit Mat_CLK at PORTC.B1;
sbit Mat_LAT at PORTC.B2;

sbit Tris_Mat_DS at TRISC.B0;
sbit Tris_Mat_CLK at TRISC.B1;
sbit Tris_Mat_LAT at TRISC.B2;

void interrupt() {
     if(TMR0IF_bit)           // check if interrupt on timer 0
          update_Matrix();    // call update display function
}

void main() {
short x;
```

```
Init_matrix();                // Initialize matrix function, to setup inter-
rupts
cls(black);
for(x=0;x<8;x++) {
      mat_r[x]=font[x];
   }

}
```

The code looks simple, we have defined an array named font[8] having 8 bytes of data. The elements are pre filled with rows data from the bitmap image. In the main program, using a loop we have filled the red matrix with each element from the row, and here is the result. It is displaying letter A correctly, just as we constructed it in the bitmap. This gives you another lesson, that not only alphanumeric characters you can make any symbols you like and make their bitmaps.
Now what will happen if we replace the fonts[] array with vertical data.



```
// 8x8 Bicolor LED Matrix Display
#include "Matrix.h"
const unsigned char font[8]=
{0x00,0x3F,0x7F,0x88,0x88,0x7F,0x3F,0x00};
extern unsigned char mat_r[];
sbit Mat_DS at PORTC.B0;
sbit Mat_CLK at PORTC.B1;
sbit Mat_LAT at PORTC.B2;

sbit Tris_Mat_DS at TRISC.B0;
sbit Tris_Mat_CLK at TRISC.B1;
sbit Tris_Mat_LAT at TRISC.B2;

void interrupt() {
     if(TMR0IF_bit)           // check if interrupt on timer 0
          update_Matrix();    // call update display function
}

void main() {
short x;
Init_matrix();                // Initialize matrix function, to setup inter-
rupts
cls(black);
for(x=0;x<8;x++) {
      mat_r[x]=font[x];
   }

}
```

Same program, but just by changing the bitmap array coding from horizontal to vertical our display has been rotated by 90 degrees. Had we been scanning the columns instead of rows, this would appear absolutely perfect, but then the rows data would appear rotated. So the take home message is we must have a bitmap coded as rows, because we are displaying one row at a time. Of course we can always compensate for this in our software, and we will do this shortly.

Generating the Bitmaps

Now comes the difficult question as to how to generate the bitmaps of all the text characters we want to be supported by our display. One obvious method is to take a graph paper and pencil, draw the characters and using a binary calculator generate 8 character codes for each character.
Second somewhat easier method is to find some software, that can do this for us. This is certainly a good option. I have found many such applications, available on internet with a grid of boxes, you can just click the boxes you want to turn on, and it will generate the bitmaps for you. Still you have to draw every character yourself.
There are some applications that will accept the windows font name from you and make a full list of codes for you. This is a nice option. I leave it to you to explore the internet and find a good solution, and share with colleagues and fellows on our forum.
The option that I used was to borrow someone else's work, well a lazy approach but I need to concentrate more on the

programming side than to sit back for days and generate the bitmap. Naturally I started my search using google and soon found a lots of projects that were sharing their work. Most of the projects were made by hobbyists and their hand -crafted bitmaps were rather incomplete. Continuing my search and I ended up on a site where they had a commercial 8x8 led matrix display and had a library of fonts for Arduino. The bitmap was quite extensive and made both for 5x7 as well as 8x8 displays. The code could be easily adapted to our project and suits us. I decided to use the 8x8 font, as

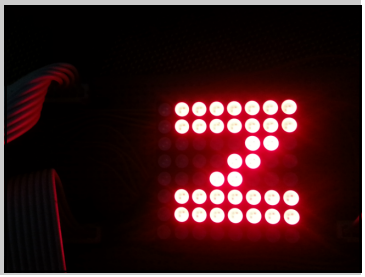
```
// defines 8x8 ASCII characters 0x20-0x7F (32-127)
unsigned const char font[96][8] = {
        {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}, //
        {0x00,0x60,0xfa,0xfa,0x60,0x00,0x00,0x00}, // !
        {0x00,0xe0,0xe0,0x00,0xe0,0xe0,0x00,0x00}, // "
        {0x28,0xfe,0xfe,0x28,0xfe,0xfe,0x28,0x00}, // #
        {0x24,0x74,0xd6,0xd6,0x5c,0x48,0x00,0x00}, // $
        {0x62,0x66,0x0c,0x18,0x30,0x66,0x46,0x00}, // %
        {0x0c,0x5e,0xf2,0xba,0xec,0x5e,0x12,0x00}, // &
        {0x20,0xe0,0xc0,0x00,0x00,0x00,0x00,0x00}, // '
        {0x00,0x38,0x7c,0xc6,0x82,0x00,0x00,0x00}, // (
        {0x00,0x82,0xc6,0x7c,0x38,0x00,0x00,0x00}, // )
        {0x10,0x54,0x7c,0x38,0x38,0x7c,0x54,0x10}, // *
        {0x10,0x10,0x7c,0x7c,0x10,0x10,0x00,0x00}, // +
        {0x00,0x05,0x07,0x06,0x00,0x00,0x00,0x00}, // ,
        {0x10,0x10,0x10,0x10,0x10,0x10,0x00,0x00}, // -
        {0x00,0x00,0x06,0x06,0x00,0x00,0x00,0x00}, // .
        {0x06,0x0c,0x18,0x30,0x60,0xc0,0x80,0x00}, // /
        {0x7c,0xfe,0x9a,0xb2,0xfe,0x7c,0x00,0x00}, // 0
        {0x42,0x42,0xfe,0xfe,0x02,0x02,0x00,0x00}, // 1
        {0x46,0xce,0x9a,0x92,0xf6,0x66,0x00,0x00}, // 2
        {0x44,0xc6,0x92,0x92,0xfe,0x6c,0x00,0x00}, // 3
        {0x18,0x38,0x68,0xc8,0xfe,0xfe,0x08,0x00}, // 4
        {0xe4,0xe6,0xa2,0xa2,0xbe,0x9c,0x00,0x00}, // 5
        {0x3c,0x7e,0xd2,0x92,0x9e,0x0c,0x00,0x00}, // 6
        {0xc0,0xc6,0x8e,0x98,0xf0,0xe0,0x00,0x00}, // 7
        {0x6c,0xfe,0x92,0x92,0xfe,0x6c,0x00,0x00}, // 8
        {0x60,0xf2,0x92,0x96,0xfc,0x78,0x00,0x00}, // 9
        {0x00,0x00,0x36,0x36,0x00,0x00,0x00,0x00}, // :
        {0x00,0x05,0x37,0x36,0x00,0x00,0x00,0x00}, // ;
        {0x10,0x38,0x6c,0xc6,0x82,0x00,0x00,0x00}, // <
        {0x28,0x28,0x28,0x28,0x28,0x28,0x00,0x00}, // =
        {0x00,0x82,0xc6,0x6c,0x38,0x10,0x00,0x00}, // >
        {0x40,0xc0,0x8a,0x9a,0xf0,0x60,0x00,0x00}, // ?
        {0x7c,0xfe,0x82,0xba,0xba,0xf8,0x78,0x00}, // @
        {0x3e,0x7e,0xc8,0xc8,0x7e,0x3e,0x00,0x00}, // A
        {0x82,0xfe,0xfe,0x92,0x92,0xfe,0x6c,0x00}, // B
        {0x38,0x7c,0xc6,0x82,0x82,0xc6,0x44,0x00}, // C
        {0x82,0xfe,0xfe,0x82,0xc6,0xfe,0x38,0x00}, // D
        {0x82,0xfe,0xfe,0x92,0xba,0x82,0xc6,0x00}, // E
        {0x82,0xfe,0xfe,0x92,0xb8,0x80,0xc0,0x00}, // F
        {0x38,0x7c,0xc6,0x82,0x8a,0xce,0x4e,0x00}, // G
        {0xfe,0xfe,0x10,0x10,0xfe,0xfe,0x00,0x00}, // H
        {0x00,0x82,0xfe,0xfe,0x82,0x00,0x00,0x00}, // I
        {0x0c,0x0e,0x02,0x82,0xfe,0xfc,0x80,0x00}, // J
        {0x82,0xfe,0xfe,0x10,0x38,0xee,0xc6,0x00}, // K
        {0x82,0xfe,0xfe,0x82,0x02,0x06,0x0e,0x00}, // L
        {0xfe,0xfe,0x60,0x30,0x60,0xfe,0xfe,0x00}, // M
        {0xfe,0xfe,0x60,0x30,0x18,0xfe,0xfe,0x00}, // N
        {0x38,0x7c,0xc6,0x82,0xc6,0x7c,0x38,0x00}, // O
        {0x82,0xfe,0xfe,0x92,0x90,0xf0,0x60,0x00}, // P
        {0x78,0xfc,0x84,0x8e,0xfe,0x7a,0x00,0x00}, // Q
        {0x82,0xfe,0xfe,0x98,0x9c,0xf6,0x62,0x00}, // R
        {0x64,0xe6,0xb2,0x9a,0xde,0x4c,0x00,0x00}, // S
        {0xc0,0x82,0xfe,0xfe,0x82,0xc0,0x00,0x00}, // T
        {0xfe,0xfe,0x02,0x02,0xfe,0xfe,0x00,0x00}, // U
```

```
        {0xf8,0xfc,0x06,0x06,0xfc,0xf8,0x00,0x00}, // V
        {0xfe,0xfe,0x0c,0x18,0x0c,0xfe,0xfe,0x00}, // W
        {0xc6,0xee,0x38,0x10,0x38,0xee,0xc6,0x00}, // X
        {0xe0,0xf2,0x1e,0x1e,0xf2,0xe0,0x00,0x00}, // Y
        {0xe6,0xce,0x9a,0xb2,0xe2,0xc6,0x8e,0x00}, // Z
        {0x00,0xfe,0xfe,0x82,0x82,0x00,0x00,0x00}, // [
        {0x80,0xc0,0x60,0x30,0x18,0x0c,0x06,0x00}, // "\"
        {0x00,0x82,0x82,0xfe,0xfe,0x00,0x00,0x00}, // ]
        {0x10,0x30,0x60,0xc0,0x60,0x30,0x10,0x00}, // ^
        {0x01,0x01,0x01,0x01,0x01,0x01,0x01,0x01}, // _
        {0x00,0x00,0xc0,0xe0,0x20,0x00,0x00,0x00}, // `
        {0x04,0x2e,0x2a,0x2a,0x3c,0x1e,0x02,0x00}, // a
        {0x82,0xfc,0xfe,0x22,0x22,0x3e,0x1c,0x00}, // b
        {0x1c,0x3e,0x22,0x22,0x36,0x14,0x00,0x00}, // c
        {0x0c,0x1e,0x12,0x92,0xfc,0xfe,0x02,0x00}, // d
        {0x1c,0x3e,0x2a,0x2a,0x3a,0x18,0x00,0x00}, // e
        {0x12,0x7e,0xfe,0x92,0xc0,0x40,0x00,0x00}, // f
        {0x19,0x3d,0x25,0x25,0x1f,0x3e,0x20,0x00}, // g
        {0x82,0xfe,0xfe,0x10,0x20,0x3e,0x1e,0x00}, // h
        {0x00,0x22,0xbe,0xbe,0x02,0x00,0x00,0x00}, // i
        {0x02,0x23,0x21,0xbf,0xbe,0x00,0x00,0x00}, // j
        {0x82,0xfe,0xfe,0x08,0x1c,0x36,0x22,0x00}, // k
        {0x00,0x82,0xfe,0xfe,0x02,0x00,0x00,0x00}, // l
        {0x3e,0x3e,0x30,0x18,0x30,0x3e,0x1e,0x00}, // m
        {0x3e,0x3e,0x20,0x20,0x3e,0x1e,0x00,0x00}, // n
        {0x1c,0x3e,0x22,0x22,0x3e,0x1c,0x00,0x00}, // o
        {0x21,0x3f,0x1f,0x25,0x24,0x3c,0x18,0x00}, // p
        {0x18,0x3c,0x24,0x25,0x1f,0x3f,0x21,0x00}, // q
        {0x22,0x3e,0x1e,0x22,0x38,0x18,0x00,0x00}, // r
        {0x12,0x3a,0x2a,0x2a,0x2e,0x24,0x00,0x00}, // s
        {0x00,0x20,0x7c,0xfe,0x22,0x24,0x00,0x00}, // t
        {0x3c,0x3e,0x02,0x02,0x3c,0x3e,0x02,0x00}, // u
        {0x38,0x3c,0x06,0x06,0x3c,0x38,0x00,0x00}, // v
        {0x3c,0x3e,0x06,0x0c,0x06,0x3e,0x3c,0x00}, // w
        {0x22,0x36,0x1c,0x08,0x1c,0x36,0x22,0x00}, // x
        {0x39,0x3d,0x05,0x05,0x3f,0x3e,0x00,0x00}, // y
        {0x32,0x26,0x2e,0x3a,0x32,0x26,0x00,0x00}, // z
        {0x10,0x10,0x7c,0xee,0x82,0x82,0x00,0x00}, // {
        {0x00,0x00,0x00,0xee,0xee,0x00,0x00,0x00}, // |
        {0x82,0x82,0xee,0x7c,0x10,0x10,0x00,0x00}, // }
        {0x40,0xc0,0x80,0xc0,0x40,0xc0,0x80,0x00}, // ~
        {0x1e,0x3e,0x62,0xc2,0x62,0x3e,0x1e,0x00}, //
};
```

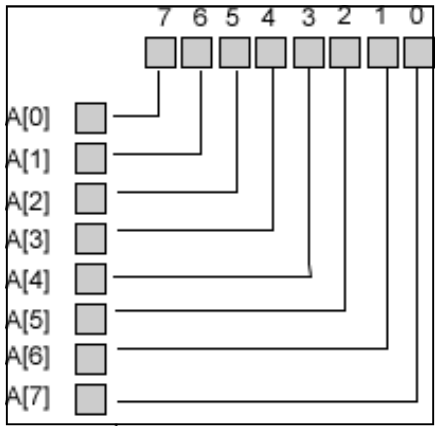
this had rather bold character maps that were good to read from a distance.
This is the entire bitmap for 96 characters, 8 bytes for each character. The array is two dimensional with 96 rows and 8 columns. The first character is ‘space’ which is character 32 on ASCII table. The table therefore is offset by 32 from the ASCII codes of the characters. Thus if the character that we want to display can be located in the map at ASCII value of the character - 32. So letter A has ASCII code of 65, its map will be found at 65 - 32 = 33th element of the fonts array in its rows dimension. Displaying 8 characters from font[33][] will give us character A.
So now make another file named font.h and copy this entire code in it. We will include this file in our text processing API library. This bitmap has been mapped as columns, therefore characters would appear rotated.

```
// 8x8 Bicolor LED Matrix Display
#include "Matrix.h"
#include "font.h"

extern unsigned char mat_r[];
sbit Mat_DS at PORTC.B0;
sbit Mat_CLK at PORTC.B1;
sbit Mat_LAT at PORTC.B2;

sbit Tris_Mat_DS at TRISC.B0;
sbit Tris_Mat_CLK at TRISC.B1;
```

```
sbit Tris_Mat_LAT at TRISC.B2;

void interrupt() {
     if(TMR0IF_bit)           // check if interrupt on timer 0
          update_Matrix();    // call update display function
}

void main() {
short x;
unsigned char ch;
Init_matrix();                // Initialize matrix function, to setup inter-
rupts
cls(black);
ch='N';
for(x=0;x<8;x++) {
      mat_r[x]=font[ch-32][x];
   }

}
```



Lets attempt to use the bitmap table, to display any character of our choice. In this program we have removed the font [] array and included the font.h file, that has the necessary font bitmap table. In our main program we have a character type variable named ch, and we have assigned it a character, lets say 'N'. Notice the character N is enclosed within single quotes. Single quotes in C mean a single character, whereas double quotes like "N" would mean a string which is different structure. So when we say ch='N' it has stored the ASCII value of N that is 78. to access its bitmap in table we subtract 32 from the 78 and this points us to exact location in our font table where bitmap image of letter N is starting. From here we access next 8 bytes and stuff them in our video memory and we get the letter N displayed. You can try with other characters like '1', '9', '*', '&' etc and you will get the appropriate display every time. However the letters are rotated. Now we are going to explore how to display the characters in correct orientation. There are two ways, either we rotate our display, which is not a good option, or to get a character map where rows have been mapped instead of columns; this is again a daunting task. Lets find some software solution to rotate the bitmap while displaying using the same bitmap table.



Rotating the bitmap by 90 degrees

In order to rotate the horizontal bitmap image by 90 degrees, we have to transfer the bits from horizontal map into vertical positions one by one. This will be done for all 8 bytes in the bitmap to all 8 bytes in the video memory.

```
// 8x8 Bicolor LED Matrix Display
#include "Matrix.h"
#include "font.h"

extern unsigned char mat_r[];
sbit Mat_DS at PORTC.B0;
sbit Mat_CLK at PORTC.B1;
sbit Mat_LAT at PORTC.B2;

sbit Tris_Mat_DS at TRISC.B0;
sbit Tris_Mat_CLK at TRISC.B1;
sbit Tris_Mat_LAT at TRISC.B2;

char TestBit(unsigned char x, short i) {
 return ((x >> i) & 1);
}
```

```
void interrupt() {
     if(TMR0IF_bit)           // check if interrupt on timer 0
          update_Matrix();    // call update display function
}

void main() {
short x,i;
unsigned char ch;
Init_matrix();               // Initialize matrix function, to setup inter-
rupts
cls(black);
ch='N';
for(i=0;i<8;i++)
 {
    for(x=0;x<8;x++) {
     if(Testbit(font[ch-32][i],7-x))
         pset(x,i,red);
      else
         pset(x,i,black);
     }

 }
}
```

In this code we have made a new function named TestBit() that will accept a byte data and an index to the bit that we want to test if its 1 or 0. because accordingly we have to set the bits in the column. The code has simply gone through each byte of the bitmap, tested each bit if its 1 or 0 and set the corresponding column bits as red color or black (for OFF) and you can see the result, the letter ‘N’ appears in normal orientation.

Characters and Strings

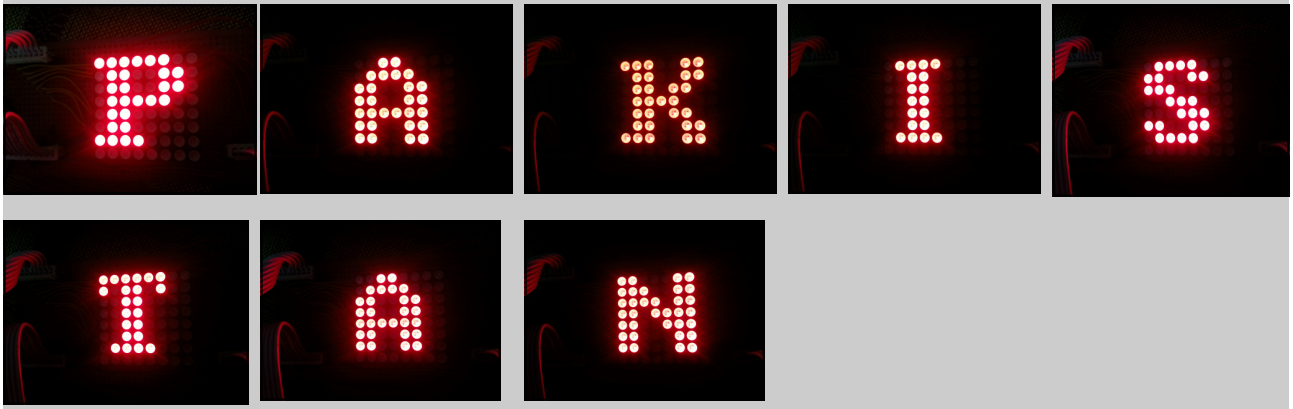
Now that we can display individual character in correct orientation, lets give a brief talk to the strings, which are the standard method of sending textual data to any device. Strings are essentially alphanumeric characters, grouped together to make one data type. The strings are stored internally as an array of characters. One thing special about strings is that their length is not fixed. It can be a single character, or it can have 100 characters. The strings are stored internally as an array, with a special character at the end of data. This special character is binary 0 in the last byte. This is also represented in C as ‘\0’. Essentially this is a single byte containing 0 in it. Remember character ‘0’ and binary 0 are different.

Thus if we have a string “PAKISTAN” it has 8 characters it will be represented in memory as an array of 9 bytes, first 8 bytes will have ASCII codes for the data characters and last byte will have ‘\0’ to indicate end of string.

In order to display string data on our display we need to scan the string array one character at a time, display it and then proceed to next one till we encounter ‘\0’.

```
// 8x8 Bicolor LED Matrix Display
#include "Matrix.h"
#include "font.h"

extern unsigned char mat_r[];
sbit Mat_DS at PORTC.B0;
sbit Mat_CLK at PORTC.B1;
sbit Mat_LAT at PORTC.B2;

sbit Tris_Mat_DS at TRISC.B0;
sbit Tris_Mat_CLK at TRISC.B1;
```

```
sbit Tris_Mat_LAT at TRISC.B2;

char TestBit(unsigned char x, short i) {
 return ((x >> i) & 1);
}

void interrupt() {
     if(TMR0IF_bit)           // check if interrupt on timer 0
          update_Matrix();    // call update display function
}

void main() {
short x,i,y;
char ch;
char message[]="PAKISTAN";
Init_matrix();               // Initialize matrix function, to setup interrupts
cls(black);
y=0;
do {
ch=message[y];
for(i=0;i<8;i++)
 {
    for(x=0;x<8;x++) {
     if(Testbit(font[ch-32][i],7-x))
         pset(x,i,red);
       else
         pset(x,i,black);
     }

 }
 delay_ms(2000);
 y++;
 } while(message[y] != '\0');
}
```



In this program we have defined a character typed array named message[] and put a string data , “PAKISTAN” in it. Notice the data is enclosed in double quotes. Compiler automatically appends a ‘\0’ at the end of string. In program we have defined a variable y, to track the characters in string, and put a loop that iterates through the message string one byte at a time, displays it, waits for 2 seconds and then advances y to get next character. This action is repeated till we encounter ‘\0’ in the message string.

I think now we have covered all the basics of text display on our display, and you understand how to manipulate data. Now you can get any text data formatted in the form of a string and dis-

play it on the matrix. If you combine some sort of communication between your controller and PC, like serial, USB or Ethernet you can easily write applications that will get the string data from internet and update the display.

Now is the time to write our appropriate API library to display text data, using this background knowledge, so that our user does not need to go through all this in his projects. In your source files create a file named “text.c”; this file will contain the API functions to manipulate text, just like our graphics.c manipulated graphics. Text.c will use some functions defined in graphics.c to draw the pixels.

The first function that is to be internally used is the Testbit, as we have seen it previously to test if a bit is high or low. This will be used to rotate the character map.

```
char TestBit(unsigned char x, short i) {
 return ((x >> i) & 1);
}
```

Note this function has type char, unlike many other functions that are typed void. This is because this function will accept a character data, which will be one byte long, and an integer value which indicates the bit we want to test. This value will be between 0 and 7. the function will apply the bit test on appropriate bit and return a character 1 if the bit is already on, and a 0 if the bit is off.

Next logical function to manipulate text will be to display one character. The function will accept the ACSII code of the character and fetch its bitmap from the font array, rotate the map and display character on display.

```
void show_character(short y,char c,unsigned char clr,unsigned char backclr) {
 char ch;
 short i;
 short x;
 ch=c-32;
 for(i=0;i<8;i++)
 {
    for(x=0;x<8;x++) {
     if(Testbit(font[ch][i],7-x))
         pset(x,i+y,clr);
      else
         pset(x,i+y,backclr);
     }

 }

}
```

The parameter y indicates the column number on matrix from where matrix should be displayed, this will allow us to position the character anywhere on display. The clr and backclr are the color names for the character as well as the color of pixels that are supposed to be off. This will effectively make a background color and allow us to display text on any background. The code does not need discussion as we have previously discussed the logic behind, just to recap you, we subtract 32 from the ASCII code of character to get the index value in our bitmap. Then as we did previously extract each bit from the bitmap and set on the appropriate column. Please make sure you write the functions prototype definitions in matrix.h file.

So far so good, now most of the times the strings are larger than display width, no matter how long you make a display, its always a good idea to scroll the text. For simplicity we ware dealing with English characters only therefore they need to slide in from right and shift leftwards slowly.

Our next function is to slide a single character one column at a time from right and slide in to left.

```
// slide in a character from right side and shift leftwards
void slide_L(char c, unsigned char clr, unsigned speed, unsigned char backclr)
{
 char ch;
 short i;
 short x;
 ch=c-32;
 for(i=0;i<8;i++)
 {
    for(x=0;x<8;x++) {
     if(Testbit(font[ch][i],7-x))
         pset(x,7,clr);
      else
         pset(x,7,backclr);
     }

     scroll_left(1,speed);

 }
}
```

This function draws the character one column at a time on right most column of display, the same way as we did to display the character. After every column draw we scroll the display to left by one column and then repeat the same process with next column bitmap. This will give an animated effect of scrolling a character.

Naturally next target will be to scroll an entire string. After the above function scrolling the entire string becomes fairly easy, as we just need to extract one character at a time from the string and send it for scrolling.

```
// scroll an entire string
void Scroll_text(unsigned char * str, unsigned char clr, unsigned
speed,unsigned char backclr) {
 short y=0;
 do {
      Slide_L(str[y],clr,speed,backclr);
   y++;
 } while (str[y] != '\0') ;
}
```

Now calling the scroll text function API from our main program to see the entire library working.

```
void main() {
Init_matrix();            // Initialize matrix function, to setup interrupts
while(1){
      Scroll_text("WELCOME ",red,100,black);
}
}
```

We had previously made functions not only to scroll left but to scroll up, right and down as well, just a demo of a character being scrolled up.

```
void main() {
Init_matrix();             // Initialize matrix function, to setup interrupts
while(1){
show_character(1,'A',green,black);
scroll_up(8,50);
}
```

This code will display a character ‘A’ on screen and scroll the character up by 8 rows giving an animated effect of moving the letter A upwards. You can easily make symbols or graphics like up arrow or down arrow and scroll them to show the same application as we find in elevators.

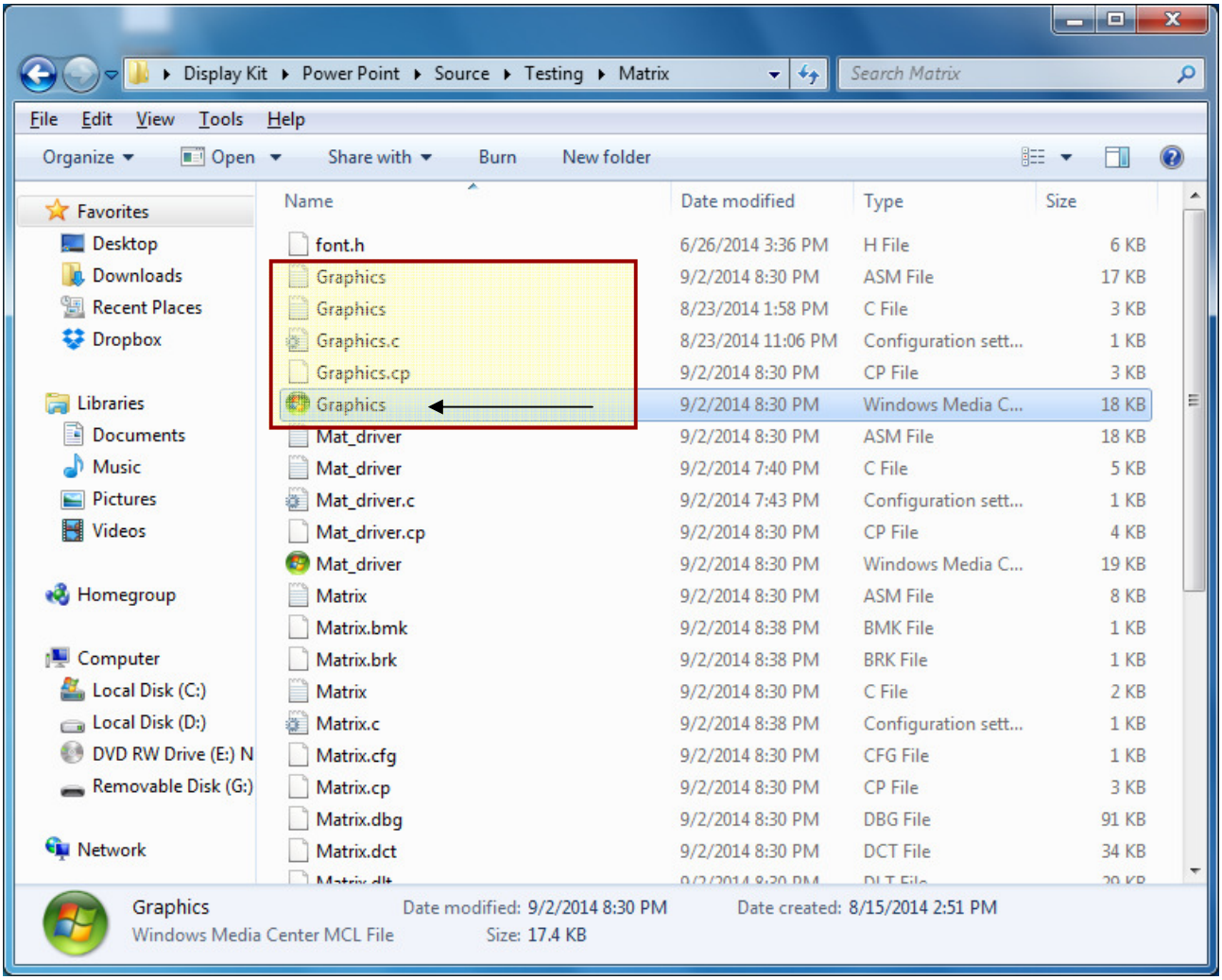
```
void main() {
short x,y,i;
Init_matrix();              // Initialize matrix function, to setup interrupts

 while(1) {
 cls(black);
 Show_character(1,'A',red,black);scroll_up(8,100);
 Show_character(1,'A',red,black);scroll_down(8,100);
 Show_character(1,'A',red,black);scroll_right(8,100);
 Show_character(1,'A',red,black);scroll_left(8,100);
 cls(black);

  Scroll_text("MICROTRONICS ",green,75,black);
  Scroll_text("PAKISTAN ",red,50,green);
  Scroll_text("LED ", orange,50, black);



  cls(black);
  for(x=0;x<8;x++){
   for(y=0;y<8;y++) {
    pset(y,x,((x+y*2) % 3)+1); delay_ms(20);
   }
  }
  delay_ms(1000);
  cls(black);
  for(i=0;i<4;i++) {
   rect(i,i,7-i,7-i,(i%3)+1,black);
   delay_ms(100);
  }
 }

}
```

Here is a rather longish demo to show you most of the features of your library. Now you can expand this library your way, use it in your applications and have fun. Using this library you can make simple animations, games, text message displays and lot more. Make sure you have defined the function prototypes in the matrix.h file.

Making Distributable Binaries

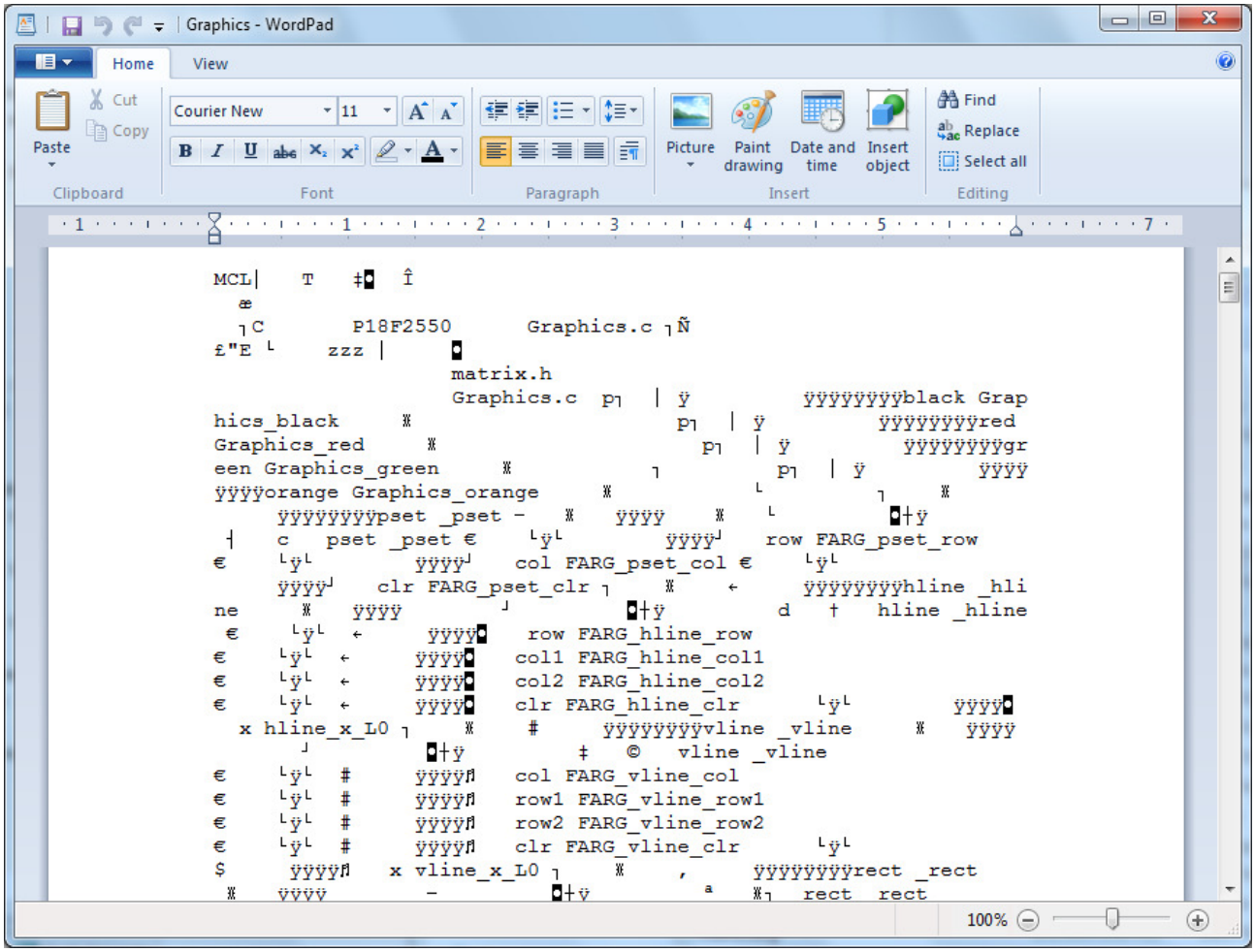
When you are finished developing your application software, and you want to distribute it to others so that they can use the functionality in their applications, its time to learn how to make distributable binaries. As far as your own in-house development is concerned, many companies and individuals develop their own libraries and use them in their applications, in this case a source level inclusion in your projects and recompiling the entire application is not a bad idea. When it comes to commercial redistribution or sharing with colleagues, you don't want to expose the source code. You therefore distribute your work as compiled binaries that can be called in the application program. The calling program only needs to know the format of functions and procedures you have made, and the kind of parameters it needs.

Almost all compilers allow you to make redistributable binary files. In terms of compilers we call them 'Object files'. This is because in classical days of PC programming the compilers used to produce an intermediate file having an .obj extension. This may or may not be the case with your



particular compiler, so you need to read its documentation in order to find the output files of com-

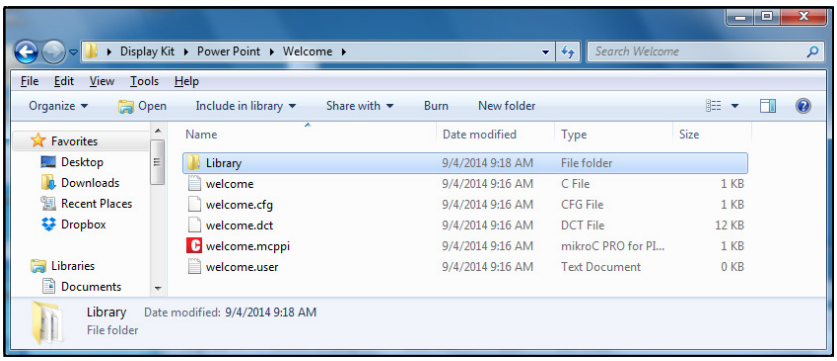
piler. A look at the source folder where we are compiling all our applications, shows a number of files generated by the compiler for each source .c file. As an example for the graphics.c file it has generated an .asm file which is the assembly language code that will be used by assembler, to create the object file. In MikroC the object file has an extension .mcl, therefore graphics.mcl is the object file generated by compiler and assembler as the final binary containing your code.



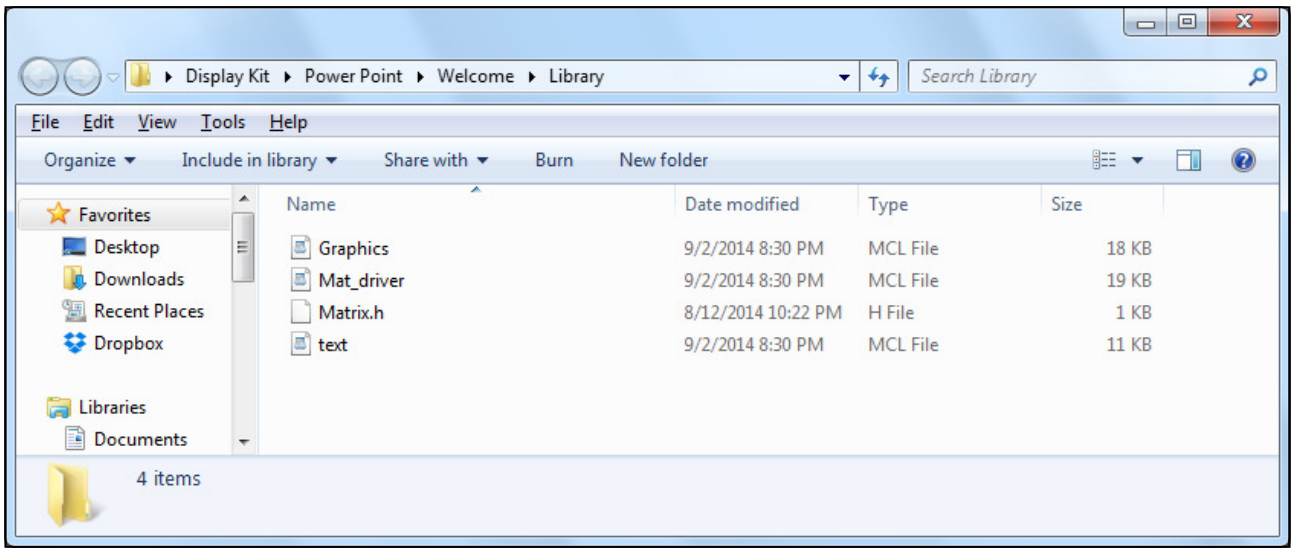
Opening this file in a text editor shows it has strange structure, something that can not make sense straight away to a programmer. This is because the source commands have been converted into binary equivalents.

Depending upon the structure of your library you may have one or more than one .mcl files, one for each source file. In our case we need graphics.mcl, mat_driver.mcl and text.mcl to be distributed. These files will not be compiled again, so you don't need matrix.h and font.h files. However in order to use the functions in our main program we need to declare the prototypes, therefore it will be good to distribute the matrix.h with your package.

Now lets make a new project in a new folder, and copy these files into it and see how to use the libraries as compiled files without need of source files. We made a new projct named welcome in a separate folder named welcome, and created a folder named 'Library' in it.

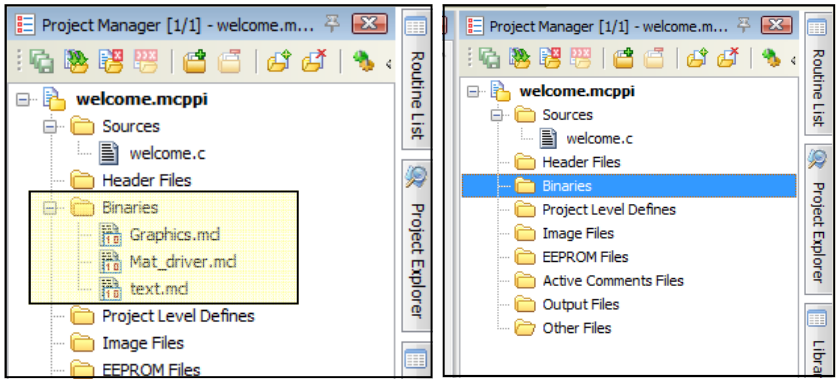


There is no compulsion to name the folder as library, you can choose any name. In this folder we copy the distributable binary files, ready to be used in the main program.

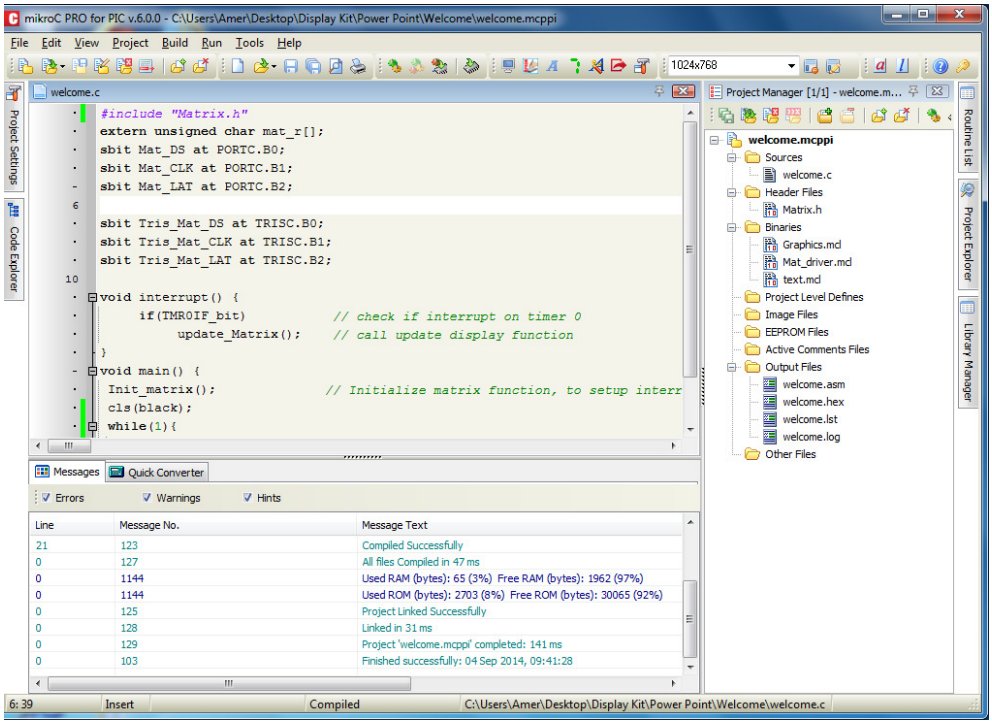


Notice we have not included the font.h file as it has the constant bitmap that is already compiled into text.mcl file.

Next we need our project to know the location of these binary files. In the project explorer panel, notice a node named 'Binaries' which at present has no files, right click over it and in the dialog box select 'Add New Files'. A file dialog box will appear locate the Library folder and include the graphics.mcl, mat_driver.mcl and text.mcl files.



Similarly add the matrix.h file in the includes folder. That all we are done, now lets turn our attention to the main program, as per requirements of our library we have to define the public symbols and the connections to our microcontroller. Make an interrupt handler function, and call the



matrix.setup function. This is all the same we have been doing in our main program throughout this book while testing the library.

```
#include "Matrix.h"
extern unsigned char mat_r[];
sbit Mat_DS at PORTC.B0;
sbit Mat_CLK at PORTC.B1;
sbit Mat_LAT at PORTC.B2;

sbit Tris_Mat_DS at TRISC.B0;
sbit Tris_Mat_CLK at TRISC.B1;
sbit Tris_Mat_LAT at TRISC.B2;

void interrupt() {
     if(TMR0IF_bit)           // check if interrupt on timer 0
          update_Matrix();    // call update display function
}
void main() {
 Init_matrix();              // Initialize matrix function, to setup interrupts
 cls(black);
 while(1){
 Scroll_text("hello World",green, 100,black);
 }
}
```

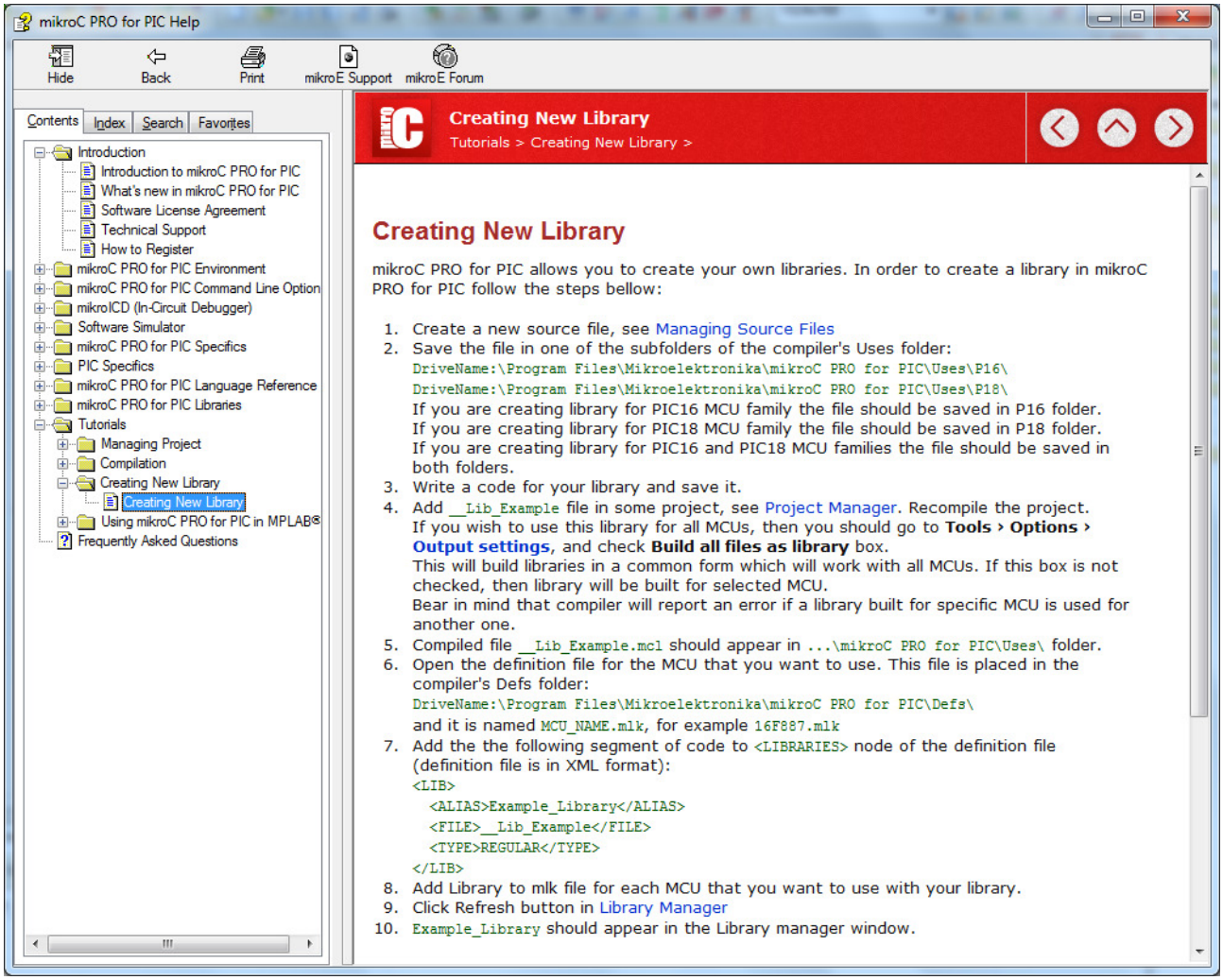
The program compiled successfully as shown in previous image, without need of the source files. You will have to write the documentation for the library to help the user, how to call the various functions and what parameters are expected. You can also keep the Library files in a common location and include in your project as binaries from that common location, as on a network server. Now in any project that needs to implement your matrix display as part of it, users can just include the binary files and use them.

Extending Library Files

The library features we have included in this demonstration may not be perfect or enough. You as programmer and owner of the source code can certainly add more stuff, from time to time and release various versions of binaries. The end user can also make their own libraries with new features using the existing commands of your library. Just like you do using the commands provided by Mikroelectronica as part of MikroC.

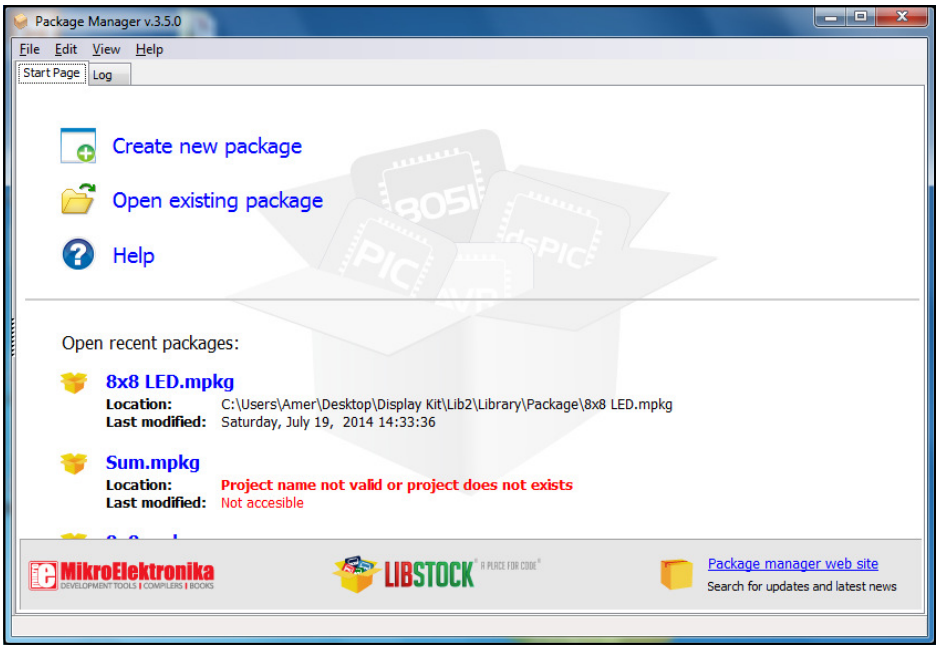
Including The Library in MikroC Permanently

In case you or some of your clients routinely work with your library, you would like to add a reference of this library in your MikroC or other Mikroelectronica products like MikroBasic installation. Just like we have the pre-installed libraries from Mikroelectronica. For this you will have to include the .mcl files in a common folder in the compiler installation and then write some xml file to describe the libraries. On top of that Mikroelectronica have a mechanism to allow only particular controller projects to have access to certain libraries. This becomes more valid in case your library is using some special modules that only certain controllers have. Like if your library needs to use USART, for some communication, not all microcontrollers have hardware USART in them, so it will be pointless to use the library with those controllers. You will have to make an xml file for each controller that an support this library. This is a daunting task, and including bare binaries directly in your project is much more convenient than to go through this entire exercise. The Mikroelectronica help in your compiler gives details about this procedure.



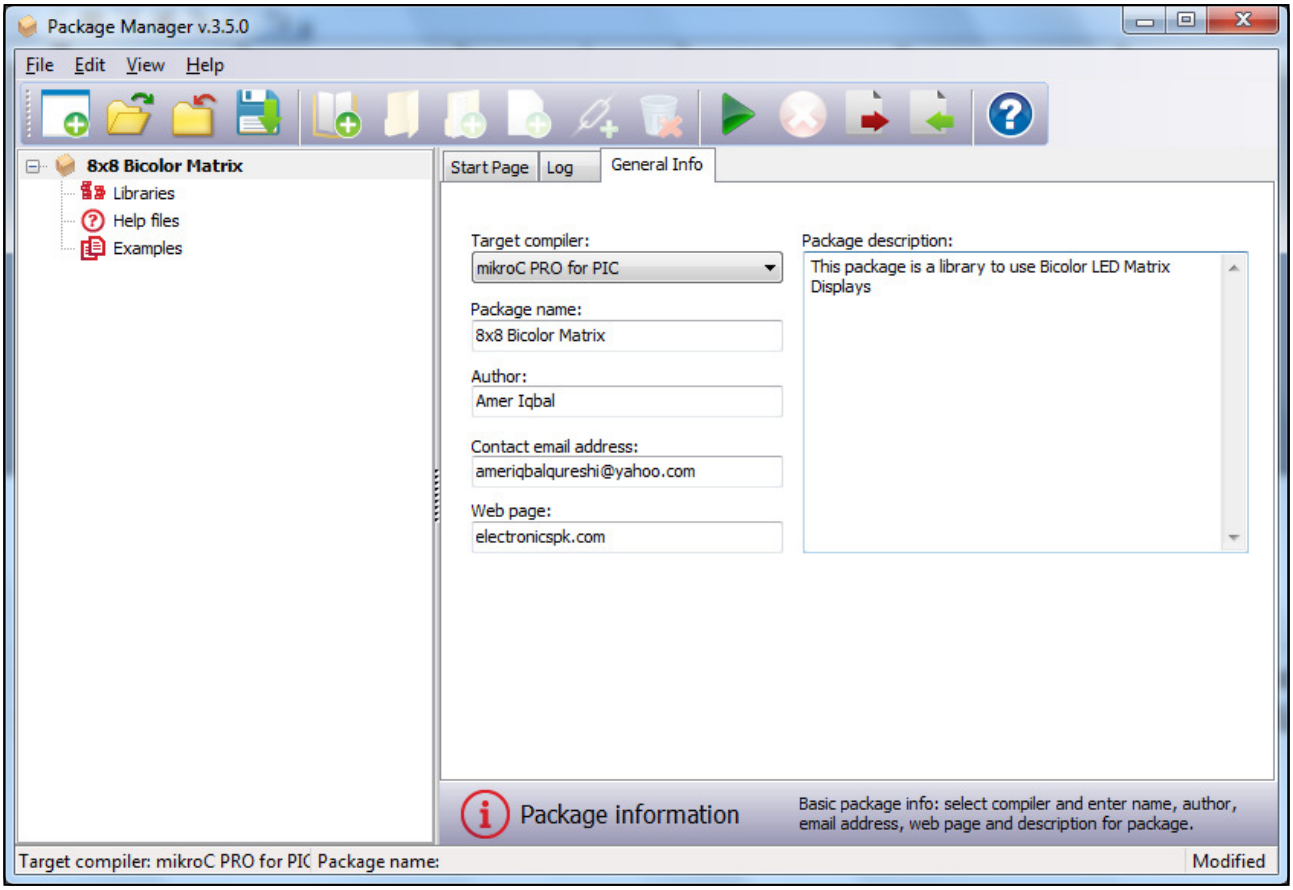
Fortunately folks at Mikroelectronica have realized the importance of this issue and they have created a small tool that will make a library for you, and using that tool you and your users can install the library into their MikroC installation.

This is called 'Packaging tool' and can be downloaded free from mikroe.com the packaging tool will allow you to pack the library files, associated help files and example files all in a single re -distributable package. Lets see this tool in action. Download and install the latest version of pack-

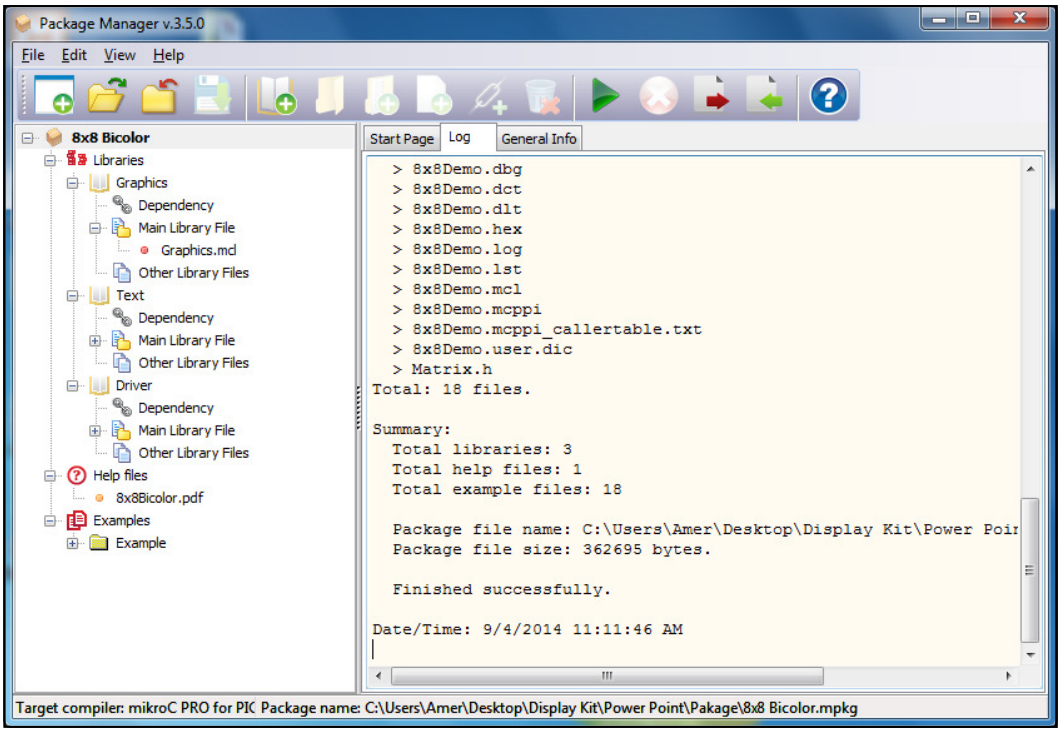


age manager from Mikroe.com after installation, which is a straight forward process, start the package manager. Remember this product is standalone and does not integrate with MikroC.

Now click on the greeting screen to “Create a new package” in the next screen fill relevant information.



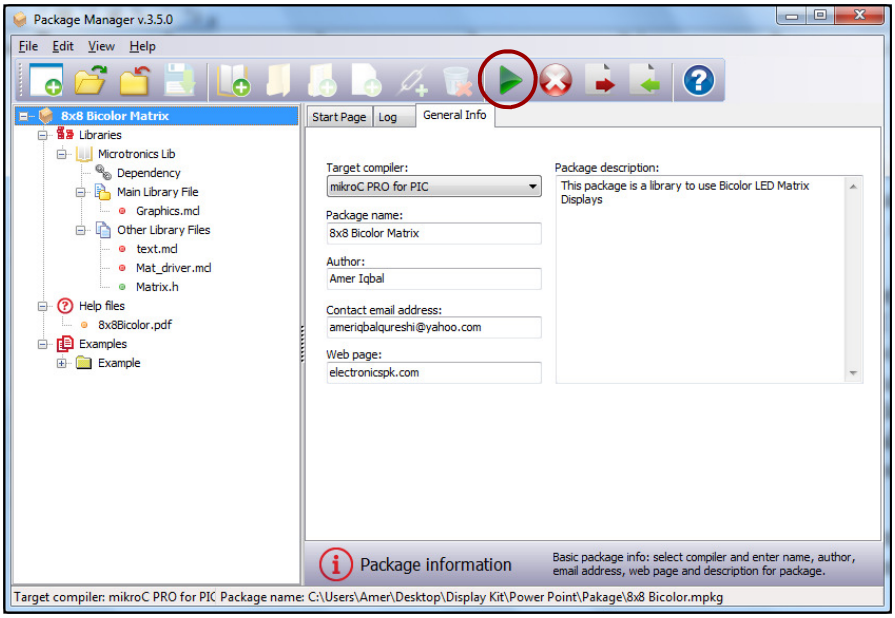
Now in the project tree right click over the libraries node and click “Add a New Library”. This will have three nodes, the dependency node gives you an entire list of controllers supported by the compiler, you can select the controllers for which your library is supported. You can elect all if



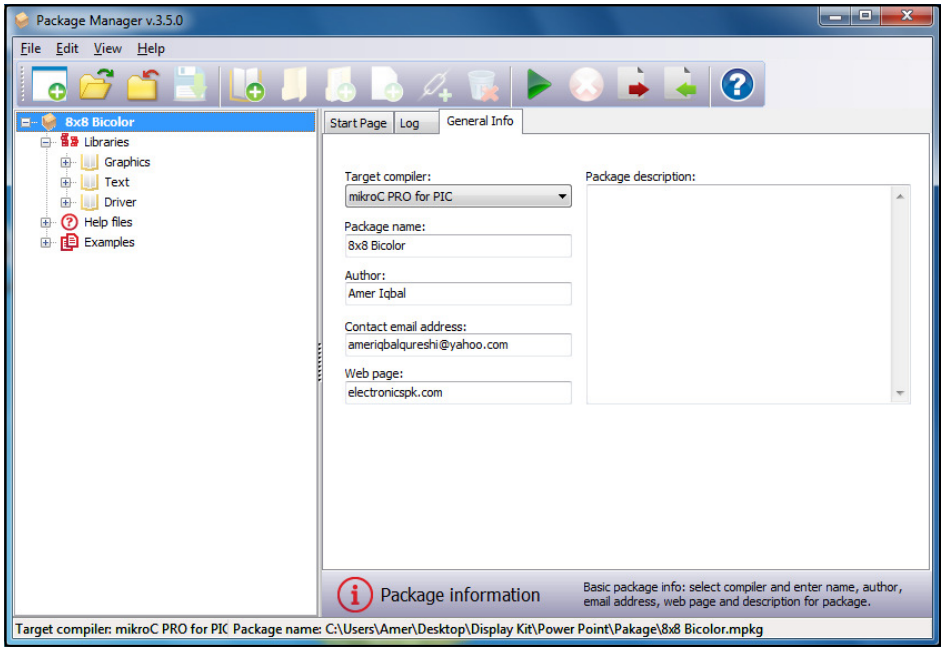
you want but then installation will take a long time, as it will have to adjust libraries for every controller you have selected. Next click over the main library file select one object file that is going to be the main library file, and in other files add other supporting files having functions internally used by the main library. We have three separate libraries so we make three library nodes.

Now you can add the help files and example files as well, help files can be PDF or they should be .hlp files as per windows help system. I leave this section away as truly speaking I have not yet written the help files for this library as yet, but surely you will do it before giving away the package, so that users can find help and support. Just for the purpose of demonstrating I have included a dummy PDF file. And also an examples folder. Now save the package, in a new folder, lets name it package. The package will be saved as 8x8 Bicolor.mpkg file. This file contains all the definitions necessary for installation. Just distribute this file, the user will download the package manager to install this file.

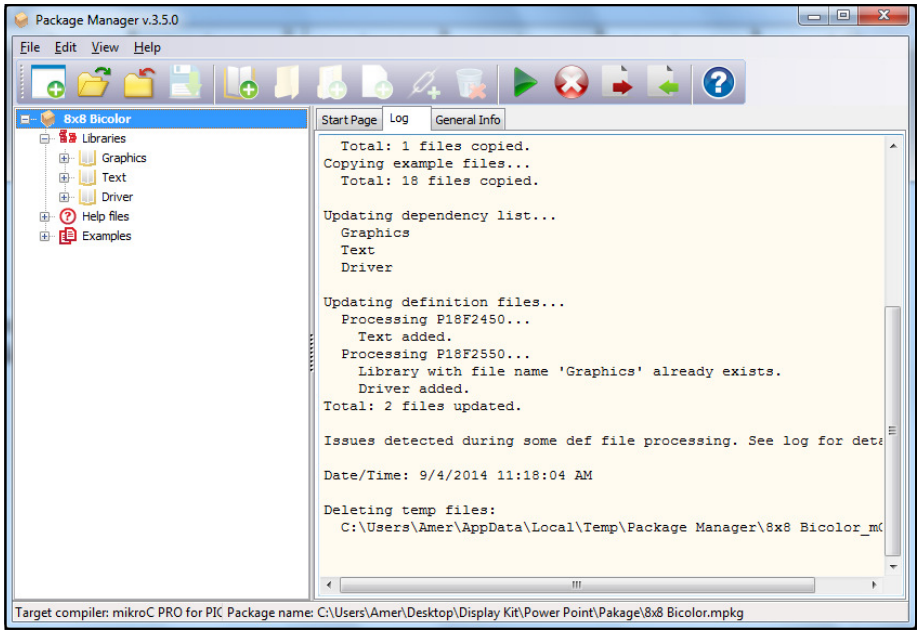
Lets give it a try. Close the package manager, and run it again, this time in welcome screen select “Open Existing Package”, now browse to the package file you have just created and open it.



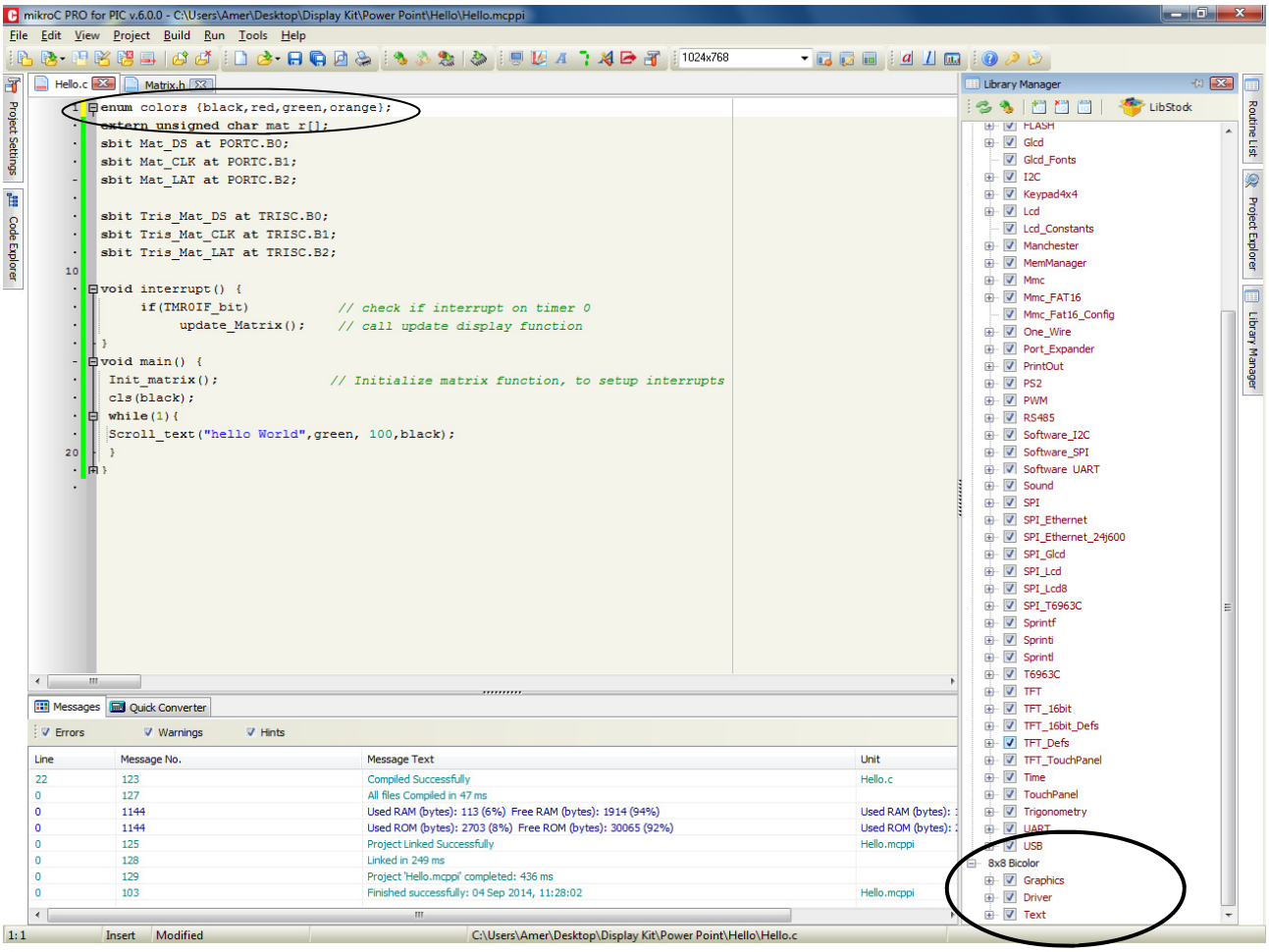
This should open the package file with all the settings as you had given, now click over the Green button on top, called install package, this will extract the package and install it on your installation of MikroC for every controller you had selected.



If you have already installed the package you can uninstall it by pressing the uninstall button, otherwise just click the green button and it should install the libraries.



Now is the time to see if this works. I had made the package file for PIC18F2550 only, so our library will appear only in projects with this controller, of-course you will make the package file for more controllers in mind so that the library is useful for a large number of controllers.



Notice now in our new project, based upon PIC18F2550 the library is available just like Mik-

roE libraries, and you do not need to include the matrix.h file as well, however in matrix.h we had defined the colors as enumeration type, if you want to use the color names, then write an enumeration on top of the file, otherwise you can use 0,1,2 and 3 for black, red, green etc. double clicking the library items will open the help file that you have attached.

Appendix
Complete Library Source Code

Mat_driver.c

```
#include "matrix.h"
// Microtronics Pakistan
// Amer Iqbal
// ameriqbalqureshi@yahoo.com
// 8x8 Bicolor LED matrix display driver
// SPI Mode Data, Clock , Latch
// Three Bytes : Row Select, Green, Red
// red is least byte
//  mat_row, mat_g and mat_r are reserved global arrays these names can not be
used in anyother part
// This is the lowest level driver to send the shadow arrays to led matrix over
SPI interface, but does not use the SPI
// Module, so that even simplest controllers can implement it.


// Connections of the matrix display to Digital I-O lines. These reserved words
must be declared in program
extern sfr sbit Mat_DS;
extern sfr sbit Mat_CLK;
extern sfr sbit Mat_LAT;

extern sfr sbit Tris_Mat_DS;
extern sfr sbit Tris_Mat_CLK;
extern sfr sbit Tris_Mat_LAT;
unsigned char mat_row[8]={1,2,4, 8,16,32,64,128 };   // selecting the row by
index mat_i
unsigned char mat_i=0;
unsigned char mat_r[8]={0,0,0,0,0,0,0,0};   // 8x8 64 bit array for holding red
matrix pattren
unsigned char mat_g[8]={0,0,0,0,0,0,0,0};   // 8 x 8 64 bit array for holding
green matrix pattren

// Initialize the matrix, set Tris registers, and setup the interrupt system on
Timer 0
void Init_Matrix() {
 Tris_Mat_DS=0;
 Tris_Mat_CLK=0;
 Tris_Mat_LAT=0;
 Mat_DS=0; Mat_CLK=0; Mat_LAT=0;

  // setup interrupt on timer 0 2ms ineterrupt
    T0CON         = 0x88;
    TMR0H         = 0xA2;
    TMR0L         = 0x40;
    GIE_bit       = 1;
    TMR0IE_bit    = 1;
}


// Shift out the data to three shift registers
```

```
// the active column is 0 on shift registers whereas it is 1 in our array
// therefore the values are inverted first before shofting out

void Shout(unsigned char row, unsigned char green, unsigned char red) {
 short j;
 green = ~green;         // invert bits because led is on when bit is 0 (sink)
 red=~red;
 for(j=0;j<8;j++)        // Shift out red First
 {
  Mat_DS= red & 1  ;      // test least significant bit if its 1
  Mat_CLK=1;Mat_CLK=0;   // clock
  red=red >> 1;          // shift bits to right by 1 to get next bit
 }

 for(j=0;j<8;j++)        // Shift Out Green
 {
  Mat_DS= green & 1  ;      // test least significant bit if its 1
  Mat_CLK=1;Mat_CLK=0;   // clock
  green=green >> 1;          // shift bits to right by 1 to get next bit
 }

// The row bits are 1 to select a row, therefore they dont need to be inverted
 for(j=0;j<8;j++)        // Shift Out Row select
 {
  Mat_DS= row & 1  ;      // test least significant bit if its 1
  Mat_CLK=1;Mat_CLK=0;   // clock
  row=row >> 1;          // shift bits to right by 1 to get next bit
 }

 Mat_LAT=1;Mat_LAT=0;        // Clock Latch to appear data.

}

// called every 2ms from the interrupt and shifts out one row of red and green
matrix array
// increments the row index mat_i
void update_matrix(){
    TMR0IF_bit = 0;         // reset timer values, calculated for 48MHz
    TMR0H         = 0xA2;
    TMR0L         = 0x40;

    Shout(mat_row[mat_i],mat_g[mat_i],mat_r[mat_i]);
     mat_i++;
     if(mat_i==8)
       mat_i=0;
}

// Basic function to fill the Video RAM with pattrens of red and green
void fill(unsigned char r,unsigned char g) {
 short i;
 for(i=0;i<8;i++){
  mat_r[i]=r;
  mat_g[i]=g;
 }
}

// scroll the video memory to left by x positions
void scroll_left(short x,short speed){
 short i,p;
 for(p=0;p<x;p++) {
 vdelay_ms(speed);
 for (i=0;i<8;i++) {
  mat_r[i]=mat_r[i] >> 1;
```

```
  mat_g[i]=mat_g[i] >> 1;

 }
 }
}


void scroll_right(short x, short speed){
 short i,p;
 for(p=0;p<x;p++) {
 vdelay_ms(speed);
 for (i=0;i<8;i++) {
  mat_r[i]=mat_r[i] << 1;
  mat_g[i]=mat_g[i] << 1;

 }
 }
}

void scroll_up(short y,short speed) {
 short i,p;

 for(p=0;p<y;p++) {
  vdelay_ms(speed);
  for(i=0;i<7;i++) {
    mat_r[i]=mat_r[i+1];
    mat_g[i]=mat_g[i+1];
  }
  mat_r[7]=0;
  mat_g[7]=0;
 }
}


void scroll_down(short y,short speed) {
 short i,p;

 for(p=0;p<y;p++) {
  vdelay_ms(speed);
  for(i=7;i>0;i--) {
    mat_r[i]=mat_r[i-1];
    mat_g[i]=mat_g[i-1];
  }
  mat_r[0]=0;
  mat_g[0]=0;
 }
}
```

Graphics.c

```
#include "Matrix.h"
// Graphics Library for 8x8 matrix display
extern unsigned char mat_r[];
extern unsigned char mat_g[];

// Sets the pixel of column and row to color specified
void pset(short row,short col, short clr) {
 switch (clr) {
  case red:    mat_r[row]=mat_r[row] | (1 << col); // set red
               Mat_g[row]=Mat_g[row] & ~(1 << col);// clear green
               break;
  case green:  mat_g[row]=mat_g[row] | (1 << col); // set green
               Mat_r[row]=Mat_r[row] & ~(1 << col); // clear red
               break;
  case orange:
               mat_r[row]=mat_r[row] | (1 << col);
               mat_g[row]=mat_g[row] | (1 << col);
               break;
  case black:
                Mat_r[row]=Mat_r[row] & ~(1 << col);
                Mat_g[row]=Mat_g[row] & ~(1 << col);
                break;
  }
}

// Draw a horizontal line on a given row from col1 to col2
void hline(short row, short col1, short col2, short clr) {
 short x;
 for(x=col1;x<=col2;x++) {
  pset(row,x,clr);
 }
}

// Draw a verticle line on a given column from row1 to row2
void vline(short col, short row1, short row2, short clr) {
 short x;
 for(x=row1;x<=row2;x++) {
  pset(x,col,clr);
 }
}


// Draw a rectangle between two coordinates with border color and fills the
rectangle with a fill color
void rect(short row1, short col1, short row2, short col2, short clr, short
fill_clr) {
short x,y;
 hline(row1,col1,col2,clr);
 hline(row2,col1,col2,clr);
 vline(col1,row1,row2,clr);
 vline(col2,row1,row2,clr);
 for(y=row1+1;y<row2;y++){
  for(x=col1+1;x<col2;x++) {
   pset(y,x,fill_clr);
  }
 }
}

// Clear screen and fill the entire screen with a clr
```

```
void cls(short clr) {
    switch (clr) {
     case black: fill(0,0);break;
     case red: fill(255,0);break;
     case green: fill(0,255);break;
     case orange: fill(255,255);break;
    }
}

// get status of a pixel
char pget(char x,char y) {
 if( ((Mat_r[y] >> x) & 1) && ((mat_g[y]>>x) &1) )
     return orange;
 if( (Mat_r[y] >> x) & 1)
     return red;
 if( (Mat_g[y] >> x) & 1)
     return green;
return black;
}
```

Text.c

```
#include "font.h"
#include "matrix.h"
// function to test a bit if its 1 or 0
char TestBit(unsigned char x, short i) {
 return ((x >> i) & 1);
}

// display a character c with forecolor clr and backcolor
void show_character(short y,char c,unsigned char clr,unsigned char backclr) {
 char ch;
 short i;
 short x;
 ch=c-32;
 for(i=0;i<8;i++)
 {
    for(x=0;x<8;x++) {
     if(Testbit(font[ch][i],7-x))
         pset(x,i+y,clr);
      else
         pset(x,i+y,backclr);
     }

 }

}


// slide in a character from right side and shift leftwoards
void slide_L(char c, unsigned char clr, unsigned speed, unsigned char backclr)
{
 char ch;
 short i;
 short x;
 ch=c-32;
 for(i=0;i<8;i++)
 {
    for(x=0;x<8;x++) {
     if(Testbit(font[ch][i],7-x))
         pset(x,7,clr);
      else
         pset(x,7,backclr);
     }

     scroll_left(1,speed);

 }
}
 // scroll an entire string
void Scroll_text(unsigned char * str, unsigned char clr, unsigned
speed,unsigned char backclr) {
 short y=0;
 do {
      Slide_L(str[y],clr,speed,backclr);
   y++;
 } while (str[y] != '\0') ;
}
```