Introduction to coding on the MSP 430 Processor Family
Overloading I/O Ports with a simple LED Matrix
Adam Rogacki – 11/13/11

Executive Summary: This will provide a basic introduction to processor coding, focusing on TI’s MSP430 processor, with its affordable Launchpad development board. The main focus of this paper will be an example driving multiple outputs with less input output points, in this case a simple LED matrix.

Keywords: MSP430, Launchpad, Processor coding, Charlieplexing, Light Emitting Diodes, C programming.

INTRODUCTION
To get started programming an MSP430 MCU, we will look at driving a simple LED matrix by using GPIO (General Purpose Input Output) to selectively choose which LED to drive at a time. A cheap implementation of the MSP430 line, is the MSP430 launchpad, which is very easily integrated to all versions of windows. Since the number of GPIO pins is usually very limited, we will look at a way with LEDs to selectively choose an LED without additional hardware such as a multiplexer or other more complicated driving circuitry that involves transistors, or other ICs.

We will look into the background of this issue, and see what could work to solve this problem. An introduction to coding I/O on a processor will be given, along with coding examples. We will also look into some downsides to using this overloading idea, and some ways to get around it.

THEORY BEHIND DRIVING LEDS
An LED is a Light-Emitting Diode, and as such, it acts like a diode, with the voltage-current relationship being described by the Shockley Equation:

\[ i_D = I_s \left( e^{\frac{V_D}{nV_t}} - 1 \right) \]  (1)

In the above equation, \( I_s \), \( n \), and \( V_t \) are all constants, being the Saturation Current, the Emission Coefficient, and the Thermal Voltage respectively.

By measuring the LEDs on a curve tracer, this graph is easily seen.

Figure 1 - V vs I relationship of LED

Figure 2 - Illustrate V/Div and I/Div
The above figures were obtained from a Type 575 Transistor Curve Tracer, by Tektronix. As we can see from Figure 2, the axes in Figure 1 would be 0.5 V/Horizontal Division, and 5mA/Vertical Division. Therefore, the turn on voltage for these diodes is approximately 1.6V. Once these diodes pass their respective turn-on voltages, the current increases exponentially with relatively small increases in the voltage. This leads to current runoff which can quickly burn out the diodes, and whatever driving circuitry is involved. For example, one of these diodes ran with the Vcc and GND pins from the MSP430 yielded an insane 174mA of current draw. This amount of current puts incredible strain on the processor, and the components themselves.

The MSP430 has a built in current regulator to monitor the output pins. However, it isn’t good to rely on these to provide any current limiting for a number of reasons. If the processor fails, for example, or goes into some ‘always on’ loop, it could overload the LED and burn it out.

response is different enough, one light may be extremely bright, with the other being extremely dim. Therefore, a series resistance can be used to modulate both brightness, and keep current flow at an acceptable level.

**PIN OVERLOADING – CHARLIEPLEXING**

Since the number of available GPIO pins is oftentimes extremely limited in an MCU, these pins are extremely useful, and in any sort of hardware implementation their use needs to be optimized for the application. Perhaps one of the easiest hardware answers to driving multiple outputs with a few pins would be using a multiplexor. This device enables the processor to ‘select’ an output, and change the voltage out to that specific output. However, even though a multiplexor is indeed a possible solution, it is beyond the scope of this Application Note. We would like to focus primarily on how to code GPIO ports. Another way of overloading IO ports is a technique called Charlieplexing.

![Figure 3 - Example: 2 LEDs in parallel](image)

Figure 3 - Example: 2 LEDs in parallel

Figure 2 illustrates the other reason why you cannot just leave the LEDs without some sort of limiting resistance. Since all the LEDs are different, and might have different turn on voltages, their V vs. I response (Figure 1) will be slightly different. Therefore, if there were two in parallel, as shown in Figure 3, if the V vs. I

![Figure 4 - Charlieplexing 2 LED Trivial Example](image)

Figure 4 shows a trivial example with just 2 LEDs, and 2 IO pins. So, with 2 LEDs, this technique does not save you any IO points. But we will use this just to examine what this method will do on a smaller level.
As can be seen in this example, if Pin 1 is set to High, and Pin 2 is set to Low, D1 will be on, while D2 is off. If we switch this, so Pin 1 is Low, and Pin 2 is High, D1 will be off, and D1 will be on. Let’s examine what will happen if we expand this idea to 3 input/output pins.

**THREE PINS CHARLIEPLEXING EXAMPLE**

Figure 5 gives a schematic of how a 3 pin configuration would look for this idea. To reduce complexity of the schematic (and that of Figure 4), current limiting resistors are left out (or just assumed to be part of the LEDs themselves).

Let us look at an example case. It is clearly seen that if Pins 1 through 3 were set either high or low, none of the LEDs would turn on. But what if Pins 1, and 2 were Low, but Pin 3 was high? Then D4 would turn on. But, since Pin 1 is also set to Low, D5 has a positive drop over the diode and would also be on. This isn’t ideal, since it means that we would be unable to select single LEDs to light with this configuration.

Why are two LEDs lit? Figure 6 will help to illustrate this:

By using the pins as outputs, if we set the pin to high, it connects that pin to whatever Vcc is, but if the pin is set to low, the pin is connected to the common ground for the processor. So therefore, if we have 1 pin high, and two pins low in the Figure 5 example, two diodes will be on.

<table>
<thead>
<tr>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 1 - Truth Table for 3 Pin Case**

It is seen by the above table that no matter the selection scheme for the three pin outputs, two of the diodes are going to be lit.
The above picture just shows this. By hooking one of the inputs to Vcc, and the other two to ground (what the processor would be doing, by just dealing with outputs), you can clearly see that two of the LEDs are lit. The next section will look into how we can get around this limitation.

**SETTING HIGH IMPEDANCE**

What we really want, is three states for our GPIO pins. The Vcc and GND states are both important for choosing which pin to turn on, however if we had a High-Z option, this would enable us to choose which LED we would actually want to select. Bringing back the above schematic, and modifying the above table:

![Figure 8 - 3 Pin Example Again](image)

<table>
<thead>
<tr>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
<th>D5</th>
<th>D6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>HI</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>HI</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2 - Example with Hi-Z

This is just a theoretical/ideal result, for example if Output 2 is set to Hi-Z, and pin 1 is set High, with pin 3 set to Low, it looks as if we are selecting D6 to be on. However, D1 and D3 still have a voltage drop of Vcc across both of them, which will result in 0.5 Vcc dropped across both of them. Based on the curve in Figure 1 for each diode (since they are different, even slightly, that turn on characteristic will also be different), 0.5 Vcc may still be on the turn on region, which would mean that though 1 light is selected (D6 in our case), that D1 and D3 may be illuminated somewhat.

This is shown in the following figure.

![Figure 9 - Leakage Illustration](image)

As can be seen above, one LED is much brighter than the rest, this is the LED we are selecting, while the rest are either off, or very dim (I turned off the lights in the room to better illustrate the dim LEDs). Another trick we could play, in order to make these LEDs even dimmer would be to put another diode in series with every LED. This would provide an additional
drop (typically on the order of 0.7 V or so),
which would move us to the left of the graph in
Figure 1, and reduce the brightness of any dim
lights dramatically.

Now that we have examined this whole
implementation, we can start to see how to go
about coding up the MSP 430 to actually drive
this circuit.

**CODING THE MSP430**

First let us examine what the MSP430
launchpad looks like.

The Launpad contains two major parts, there is
a hardware USB emulation on the right third of
the board, and the bottom two thirds contains
the processor in the middle, with two rows of
breakout pins for the user to use.

The easiest configured outputs are P1.0 – P1.7,
all 8 of these IO pins are of the GPIO type that
we discussed earlier, with three usable states.
The Output state can either go Hi (to Vcc) or
Low (to GND), and the Input state looks like a
high impedance back to ground. These inputs
are contained in the Port 1 Register. The two
main commands that we will be looking into are
P1DIR and P1OUT.

P1DIR writes the 8 bits the port 1, and you can
choose which direction you want the pins to go,
with a 1 written to the bit telling the processor
that pin is for output, and 0 for input.

P1OUT will tell the processor what you want to
output. If you want the pin to be high, a one
will be written to that bit, or if you want the pin
to be grounded, a 0 will be written to that bit.

In the example code, we will be utilizing the
pins that correspond with 1.3, 1.4, and 1.5

<table>
<thead>
<tr>
<th>P1.7</th>
<th>P1.6</th>
<th>P1.5</th>
<th>P1.4</th>
<th>P1.3</th>
<th>P1.2</th>
<th>P1.1</th>
<th>P1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 3 - Port 1 Configuration**

The X's in the above table do not mean “don’t
care’s” as they usually do in digital logic. These
are the pins which we are going to change the
value of in order to select data.

**EXAMPLE: TURNING ON 1 LED**

For example, say we wanted to select D1. From
table 2, it is seen that O1 = 1, O2 = 0, O3 = Hi-Z.

In order to implement this, first we need to tell
the processor which pins are inputs and which
are outputs via the P1DIR variable. It is easiest
to write these bits in hexadecimal, rather than a
binary string, in order to reduce confusion. So,
in this case, we have O1, and O2 to be outputs,
while O3 is to be set as input. Therefore, we
would want the binary: 00110000, to be written
to the port (the 1’s correspond with bits 4 and
5, to be our outputs). So the proper C code
would be:

```
P1DIR = 0x30;
```

This selects bits 4 and 5 to be outputs, and the
rest to be high impedance (including the pin 3
which we want).

Now, to complete the example, since we have
selected our output pins, we need to tell the
processor whether to assign Vcc or GND to the
outputs. From table 2, we know that we want
bit 5 to be Vcc, and bit 4 to be GND. That would
 correspond to the binary combination of:
00100000, which would correspond to the
following hex pattern being written to the P1OUT variable:

\[
P1OUT = 0x20;
\]

This will set bit 5 to be Vcc, and bit 4 to be GND, while our previous P1DIR statement told the processor that bit 3 is HI-Z. This selects D1, and thus concludes the example.

**EXAMPLE: CYCLING THROUGH LEDS**

Now we have looked at how to select one LED, we will look at another simple example. How we can iterate through each LED. In order to select various LEDs, we need to cycle through each and configure the P1DIR and P1OUT pins for every LED. We will examine the code that does this on the next page.
#include <msp430g2231.h>

void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;
  for (;;) {
    long j;
    //First 2
    P1DIR = 0x18;
    P1OUT = 0x08;
    j = 50000;
    do (j--);
    while (j != 0);
    P1OUT = 0x10;
    j = 50000;
    do (j--);
    while (j != 0);

    //Second 2
    P1DIR = 0x28;
    P1OUT = 0x08;
    j = 50000;
    do (j--);
    while (j != 0);
    P1OUT = 0x20;
    j = 50000;
    do (j--);
    while (j != 0);

    //Third 2
    P1DIR = 0x30;
    P1OUT = 0x10;
    j = 50000;
    do (j--);
    while (j != 0);
    P1OUT = 0x20;
    j = 50000;
    do (j--);
    while (j != 0);

    //Blank
    P1DIR = 0x00;
    j = 100000;
    do (j--);
    while (j != 0);
  }
}

Figure 11 contains the code for this example. It starts with an include statement, which is changed based on which MSP430 we are using. This contains all the various definitions that are specific to that version of the MSP430. For example, which bits are GPIO, and which are for various other tasks (such as using the built in timers, or ADC functions of the MSP430). Then, as every C program, the code must have a main function.

The first line in the code halts the watchdog timer since we do not need its functionality in this particular example. The following for statement implies that this statement will run for as long as the processor has power.

The code within the for statement is split into three similar parts, for moving the input pin from pin 5, to pin 4, to pin 3. We will examine the first one (labeled “//First 2” in the code).

The first statement P1DIR = 0x18; sets the port direction, stating that pins 3 and 4 are to be output. The next statement brings pin 3 high. Then we go into a delay. The next line, j=50000; is just a variable defining our delay, in this case fifty thousand loops (this delay is approx. 1 second or so). Then the code iterates down from 50000-0, and moves on, by using a do-while loop. So, while j is not 0, decrement j by 1. Once j = 0, the code moves on to reassign P1OUT to turn on another LED, in this case P1OUT = 0x10; which says that pin 4 is now high, while pin 3 is low. Then the delay is run through again. This repeats through the other 4 LEDs, and the process repeats.

FUTURE EXAMPLES

The idea of charlieplexing is as flexible as the number of IO pins you have. In fact, if you have n IO pins, you can drive n*(n-1) LEDs with the pins utilizing this low hardware cost solution.
The idea can be taken up on larger LED arrays, with the MSP430, up to 56 LEDs. If the delay is low enough between cycling the LEDs (50000 in the previous example), you can give the impression that multiple LEDs are in fact lit.

Figure 12 - 'Turning on' all LEDs at once

Figure 12 illustrates this. It may be difficult to see, since not all the LEDs were facing the camera directly, but all are lit with relatively the same light intensity. This was done by changing the j value in the previous example to 100 for all cases. In actuality, the lights are flickering very quickly, but much too fast for the human eye to detect, so there is the illusion that all the LEDs are on at the same time. Using this idea with a larger array of lights can yield itself to outputting any combination of LEDs that you would want, even to the effect of creating logos, or symbols, or messages.

CONCLUDING REMARKS
In this application note we have looked into driving LEDs with an MSP430 as an example looking into basic processor coding techniques, and some drawbacks associated with this hardware example. First we examined how to drive LEDs, and basic Diode theory, and then introduced the topic of Charlieplexing, to keep the hardware side of this example as painless as possible. Then basics of Processor coding was introduced, describing how to set pins as inputs and outputs. Finally utilizing the duty cycle of the LEDs was introduced, to in effect make more than one LED appear to be lit at one time.

REFERENCES – FURTHER READING

LEDs
ECE 302 – Fall 2009 Lecture Notes, by Dr. Gregory Wierzba

Charlieplexing
en.wikipedia.org/wiki/Charlieplexing
www.instructables.com/id/Charlieplexing-LEDs-The-theory/

MSP430
processors.wiki.ti.com/index.php/MSP430_LaunchPad_(MSP-EXP430G2)