



Column 150, July 2008 by Jon Williams:

Putting the Brakes to It

Even if you've never been to Los Angeles chances are that you've heard of the 405 freeway. This infamous chunk of 10-lane Hell extends from the San Fernando Valley south through the west side of Los Angeles, past LAX and all the way down into Orange County where it reconnects with Interstate 5. The 405 spent some time in the news a few years ago for all the gun play associated with "road rage." As a Los Angeleno who frequently travels the 405 I smile at everybody – no matter how discourteous they might be; you never know who's having a really bad day. The bigger problem, though, is that in a sea of brake lights I have – far more times than I want to remember – found myself cringing as I hear the squeal of tires on the pavement behind me; usually someone on a cell phone that wasn't paying attention. Thankfully, I haven't been hit yet and I fully intend to keep it that way.

The problem with being stuck in traffic is that you are in fact stuck and there is no way to avoid being hit from behind by that person who's paying more attention to their cell phone conversation than the cars in front of them. On my long drive home from the Maker Faire I noticed that the flashing lights of a police car in my peripheral vision immediately got my attention (no, the police car was not for me). So I started thinking... there were a lot of lights in my rear view mirror, so why did I notice the police lights?

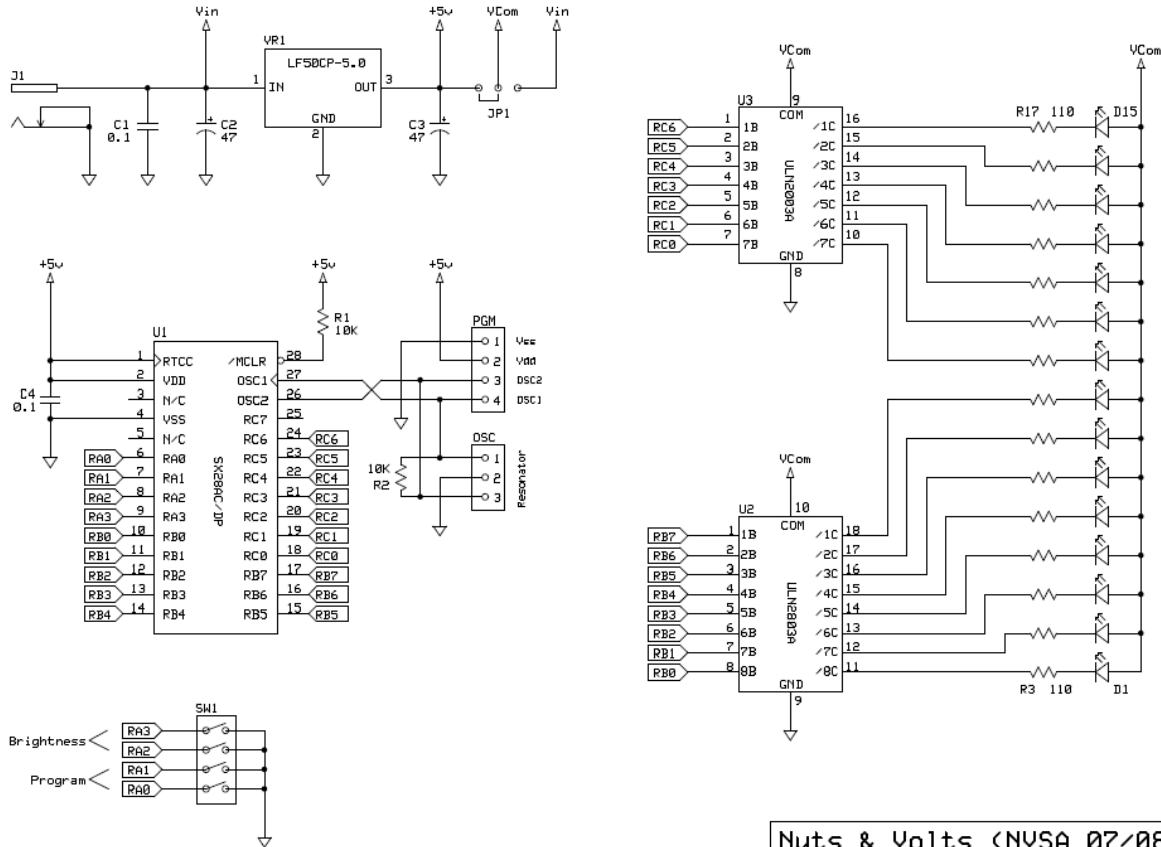
I believe the answer is simple: motion. Perhaps it's a remnant of the reptilian brain of our ancestors, but it seems that we notice motion first, then evaluate what that motion is and means second – this makes perfect sense considering the fight-or-flight response mechanism that has helped our species survive all these years. Quick detection of motion is valuable as there was a time we had to trek into the woods instead of Trader Joe's to get dinner supplies.

While retrieving my suitcase from the back of the SUV I noticed something else: that the middle brake light housing in the rear window had a perfectly-flat surface on which I could mount something. Hey, how about a panel blinking lights to get the attention of the drivers behind me? My favorite TV creator, Joss Whedon (*Buffy*, *Angel*, *Firefly*), once said in an interview: "Blinky lights mean science." Well, for me, "blinky" lights mean a better chance of surviving the 405 without getting run into!

Hardware

The circuit is simple: an SX28 that drives 15 LEDs through ULN2x03s. The reason for the ULN2x03 is to provide enough current so that all LEDs can be on at the same time – at about 20 mA each it's just too much to sink or source out of the SX without help. Also, this lets me re-use this circuit for other applications, even those that drive higher current devices (the ULN2803 can sink better than 150 mA per channel with all eight operating). I'm thinking I may re-visit my electronic Menorah project with small incandescent lamps this winter and this circuit will work perfectly for it.

Figure 150.1 shows the schematic for my “brake light buddy” – everything is as straightforward as you would imagine. You may be wondering why I went with 15 LEDs instead of 16. The reason is that I’m going to use bit 15 of the output data as a flag to mark the end of a sequence. If you modify the board you might choose to simplify parts selection by using two ULN2803s instead of the combination as I have done – I keep both in stock for my control projects so the design as presented made sense for me.



Nuts & Volts (NYSA 07/08)		
Brake Light Buddy		
J. Williams	Rev 1.0 05/07/2008	Page 1 of 1

Figure 150.1

One note about the circuit: JP1 is not intended to be moved; this is a solder-in selection. As this project is running LEDs I’m using +5v from the regulator as the common voltage for the anodes. If you use this circuit to power valves or relays you would probably choose a 12v external source and use Vin to run them.

Firmware

This program is a simple sequencer that actually started out even simpler. In the beginning I just read a four-position DIP switch to select a sequence and then jumped to a routine to handle it. Two things happened: 1) There is no way I need a selection of 16 sequences for a brake-light accessory and, 2) The LEDs I used are blindingly bright. What I ultimately decided to do is used two of the DIP switches for the sequence selection and two for LED brightness level. I also modified the sequence data to include step timing for that sequence instead of using a single global value as in the original program; this makes the sequence generation more dynamic, especially when two or more sequences are combined.

DATA Storage, SX/B Style

For those of us who started with the BASIC Stamp we're used to the **EEPROM** (BS1) and **DATA** (BS2) directives that are used to store values in the Stamp's EEPROM. With SX/B we have two directives: **DATA** (for bytes) and **WDATA** (for words). As with the BASIC Stamp we can use **READ** to access these values. Unlike the BASIC Stamp, however, there is no **WRITE** instruction in SX/B and once programmed **DATA** and **WDATA** values cannot be modified.

The reason that we can modify **EEPROM** and **DATA** values in the BASIC Stamp is that these values are written into an external (of the processor) EEPROM. With the SX, however, these values are actually "burned" into the code space of the SX and this space cannot be modified at run time. You could of course add an external EEPROM to the SX (which easy to do with SX/B's I2C instructions) should you need to have store non-volatile values that can be changed.

In this program I ended up using both **DATA** and **WDATA** to store an LED sequence – let's have a look.

```
Zig_Zag:
DATA 100
WDATA %0_000000000000111
WDATA %0_000000000111000
WDATA %0_000000111000000
WDATA %0_000111000000000
WDATA %0_111000000000000
WDATA %0_000111000000000
WDATA %0_000000111000000
WDATA %1_000000000111000
```

We start by using a label to name the sequence; this allows the compiler to resolve the label to an address that is used by the **READ** function to retrieve a table element. The first byte in the sequence, defined with **DATA**, is used as the timing (in milliseconds) for the sequence. What follows are the desired LED output patterns. As these values are words we use **WDATA**. **WDATA** stores values *Little Endian*; you should know this if you are going to retrieve these values as bytes instead of as a word as we will do in this program.

One of the many things I like about SX/B is the flexibility with numeric formatting. SX/B allows an underscore character to be used in a number without any problems, so what I've done with the LED output patterns is separate bit 15 from the others. The reason for this is that the program will use bit 15 bit as a flag for the end of the sequence. As you can see, the last entry in the above table has a 1 in the bit 15 position. I like to use a built-in flag for projects like this as it allows the sequence to be modified without having to keep track of and editing the number of steps. If you wanted to use all 16 LEDs then you can add a second **DATA** statement to the beginning of the sequence that defines the number of steps used – just remember to update this value when you modify the sequence (you'll also need to change the code that plays the sequence, but that's a one-time deal).

So let's look at the code that plays the sequence. This bundled into a subroutine that is called by passing the label of the sequence to play.

```
SUB PLAY_SEQUENCE
tmpW2 = __WPARAM12

READINC tmpW2, tmpB2
DO
  READINC tmpW2, tmpW3
  IF invert = Yes THEN
    tmpW3 = tmpW3 ^ 0x7FFF
  ENDF
  Display = tmpW3
  DELAY_MS tmpB2
LOOP UNTIL tmpW3.15 = 1
ENDSUB
```

We start by capturing the address of the sequence to be played into *tmpW2*. The timing (in milliseconds) for each step in this sequence is retrieved using **READINC**, which is a special form of **READ**. **READINC** is very cool for

code like this as it knows how many bytes to retrieve based on the size of the output variable(s) and then it automatically increments the address pointer by the appropriate value, again, based on the size of the output variable(s). What this means is that after reading the step timing value into *tmpB2* the table pointer in *tmpW2* is now pointing to the LED data for the first step in the sequence.

A **DO-LOOP** is used to play the sequence values and will terminate when bit 15 of the output pattern (which gets read into *tmpW3*) is set. At the top of this loop **READINC** is used to retrieve a pattern from the table and move it into *tmpW3*. Since *tmpW3* is a word, **READINC** will increment the address (in *tmpW2*) by two after the call; this causes the address to be pointing to the next pattern in the **WDATA** table.

For a little extra flexibility I added a global flag to invert the LED outputs. To do this the raw value in *tmpW3* gets XOR'd with 0x7FFF – this inverts all the bits in *tmpW3* except our flag, bit 15. Note, too, that SX/B allows C-style numeric formatting and for hex values; I tend to do this to make my programs useful for those that will translate them to C. Whether it's modified or not *tmpW3* is moved into the LEDs which are ports RB and RC aliased as *Display*. A call to **DELAY_MS** holds the LEDs for the timing (in milliseconds) defined by the sequence.

Let's look at **DELAY_MS** for a moment as delays are an important aspect of this program and the fact that we're going to use an interrupt to control LED brightness (more on that in a moment) means that we can't simply use **PAUSE**.

```
SUB DELAY_MS
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1
  ELSE
    tmpW1 = __WPARAM12
  ENDIF
  DO WHILE tmpW1 > 0
    tmpB1 = 25
    DO WHILE tmpB1 > 0
      \ CLRB isrFlag
      \ JNB isrFlag, @$
      \ DEC tmpB1
    LOOP
    DEC tmpW1
  LOOP
ENDSUB
```

Whether I'm using interrupts or not, I always use a subroutine called **DELAY_MS** as a replacement for **PAUSE**; without interrupts this allows **PAUSE** to be compiled in one location and saves code space; with interrupts I can fine-tune the routine based on the interrupt rate used. As I just stated we will use interrupts in this program so the structure of **DELAY_MS** presented here will reflect that.

This routine starts by capturing the delay time – specified in milliseconds – to a word variable called *tmpW1*. If a byte is passed to the routine it will show up in *__PARAM1*; if a word value is passed it will show up in *__WPARAM12*. We can determine the type of value passed by examining *__PARAMCNT*; one for a byte, two for a word.

With the milliseconds now held in *tmpW1* we will use that to control a **DO-LOOP** that has internal code that takes one millisecond to execute; the loop will decrement *tmpW1* each time through and will terminate when *tmpW1* reaches zero.

In this program the interrupt is going to run 25,000 times per second so if we counted 25 interrupt cycles we would have one millisecond. We get an assist on counting interrupt cycles by setting a flag inside the interrupt. To count the ISR cycles we start by setting *tmpB1* to 25 then executing another **DO-LOOP** that waits for *tmpB1* to hit zero. Inside this loop we will clear *isrFlag* and then wait for it to be set again (which happens inside the interrupt). When the flag gets set *tmpB1* is decremented. These are done with inline assembly statements just to keep things as trim and zippy as possible. If you use this version **DELAY_MS** with a different interrupt rate be sure to adjust the inner loop so that it runs one millisecond.

Since I've been talking about the interrupt let's have a look at it.

```
Mark_ISR:
    isrFlag = 1

PWM_Ctrl:
    acc = acc + brightness
    IF C = 1 THEN
        \ MODE $0F
        \ MOV !RB, #%00000000
        \ MOV !RC, #%00000000
    ELSE
        \ MODE $0F
        \ MOV !RB, #%11111111
        \ MOV !RC, #%11111111
    ENDIF
RETURNINT
```

This code serves two purposes: 1) It sets *isrFlag* so we can use that for external timing as we saw with `DELAY_MS` above, and, 2) It controls the LED brightness using a strategy normally employed for applying PWM to a single output.

The LED brightness is controlled adding the brightness setting into a running accumulator. When this accumulator rolls over (causing the Carry bit to be set) the LED output pins will be enabled; when there is no rollover the LEDs are disabled by making those pins inputs. If we put a large value into the *brightness* variable the rollover will take place more frequently, hence the LEDs will be on more of the time and therefore be brighter. By manipulating the TRIS registers of the LED pins we are able to handle them all simultaneously. This is a useful trick and could be applied to the segment outputs of a seven-segment display project.

With all the support code in place we can build the main line of the program. We'll start by reading the brightness settings from two of the bits of the DIP switch:

```
Start:
    brightness = %00111111
    brightness.6 = DMode.2
    brightness.7 = DMode.3
```

The first line sets the brightness to 63, or about 25%. The second line can add another 25% and the third can add 50%, based on the position of the DIP switches. The four-position DIP switch is attached to the RA port with all the pull-ups enable, so an open switch will read as one. To dim the LEDs a bit we can close one or both switches. My SUV has tinted glass in the back so I'm leaving them open for maximum brightness.

The next step is to read the sequence selection and then run it.

```
Main:
    show = DMode & %00000011

    IF show = %11 THEN Display_Hold
    IF show = %10 THEN Ziggy
    IF show = %01 THEN Out_In
    IF show = %00 THEN Crazy_Flasher

Ziggy:
    invert = No
    FOR cycles = 1 TO 3
        PLAY_SEQUENCE Zig_Zag
    NEXT
    Display = %0000_0000_0000_1111
    DELAY_MS 100
    GOTO Display_Hold

Out_In:
```

```

FOR cycles = 0 TO 1
  invert = cycles.0
  PLAY_SEQUENCE Explode
  PLAY_SEQUENCE Implode
NEXT
GOTO Display_Hold

Crazy_Flasher:
invert = No
FOR cycles = 1 TO 2
  PLAY_SEQUENCE Dual_Flash
NEXT
GOTO Display_Hold

Display_Hold:
Display = %111111111111111111
GOTO Display_Hold

```

The sequence bits are read from the DIP switch using a mask to isolate them from the brightness bits. From there it is a simple matter of jumping to the appropriate handler. Inside each handler is a little loop to run a sequence one or more times. You can see that the handler at Out_In combines two sequences and inverts them every other cycle; this is the pattern I tend to use most.

As you can see, all of the handlers ultimately jump to the label Display_Hold which simply lights all the LEDs. Remember that the point of this project is to get a driver's attention, not to annoy him/her with non-stop blinking lights. Keep your movement patterns short and interesting and you're less likely to get honked at or [one-finger] saluted by there person behind you.

Construction & Installation

While you could build this as a point-to-point wiring project, I just don't have that much patience anymore and the bumps in the road here in SoCal would likely shake that apart – a PCB is the best way to go. It didn't take long to whip up an ExpressPCB board for it, but note that the board is not the standard mini-board size and will cost more. That said, I grouped the ICs closely so that you could squeeze the project down to mini-board size (with fewer LEDs, of course) should you want to do that. Just one reminder on the board: the ULNs are active-low (the cathode connects to the ULN) so make sure you orient the LEDs in the board properly before soldering.

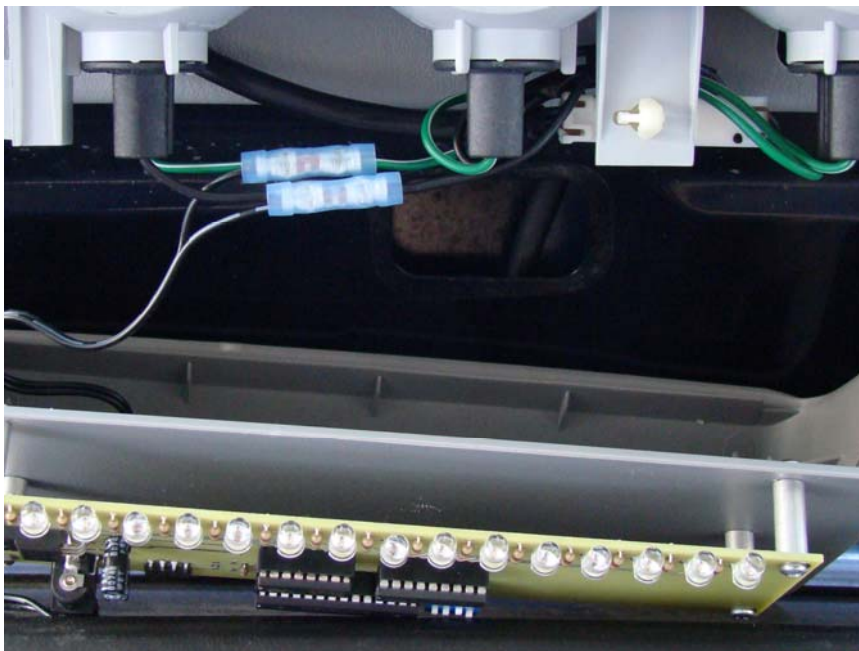


Figure 150.2

Physical installation will depend on your vehicle. As I stated earlier, my third brake light is in a housing that has a horizontal surface. I was able to remove the cover to mount the board and tap into the brake light power. Figure 2 shows how I tapped into my brake lights using common butt splices; I scavenged a power plug from a dead wall-wart for the connection to the board. Be very careful with this last step, you don't want to end up disabling your brake lights – this will definitely get you a ticket and could cause a safety issue. If in doubt, get help from a friend who is experience in automotive electrical systems and always test everything very carefully by taking your car out onto the road. Figure 3 shows the assembly with the brake light cover reinstalled.

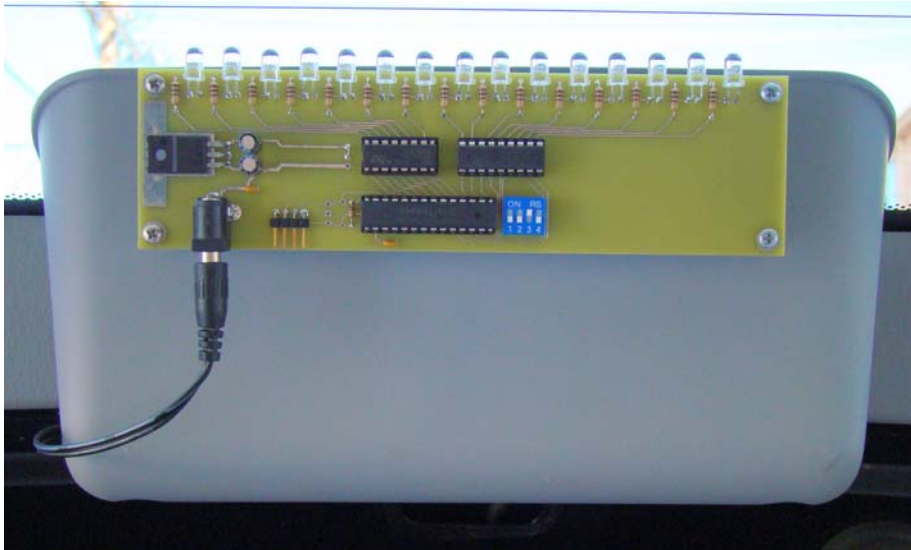


Figure 150.3

You can see in the photo of my completed board I omitted R2 and the resonator. Timing is not critical with this kind of project so the internal 4 MHz oscillator works just fine. If you use this board for a project that requires tighter timing or faster speed, add R2 and an appropriate resonator.

It can be difficult to see in a photograph, but the LEDs I used are every bit as bright as the brake light that the project is attached to; the LEDs are clearly visible in daylight on a sunny southern California day. Figure 4 shows the LEDs light when the brake light is operating.



Figure 150.4

Builder Beware

While I did check with a friend who knows more about California law than me – and he says it’s okay – I do not in fact know if my little brake light add-on is legal to use, and you should check your local laws before putting it on the road. Other important points:

- 1) Integrate the device in your brake system – don’t allow it to operate otherwise.
- 2) Make sure you can easily disable it should you be requested/ordered to do so.
- 3) Use only red LEDs – colored LEDs could be misconstrued as “official.”
- 4) Never point the device forward as you could fool a driver in front of you and, perhaps, be accused of impersonating an official vehicle (which is against the law).

So there you have it, a “simple” project that could prevent an accident and turns out to have lots of useful bits that we can incorporate into other projects. What will you do with your “blinky” lights?

Until next time, Happy Stamping, SX/B-style!

Parts List for the VEX Decoder/Servo Driver		
Designator	Value	Source
C1, C4	0.1	Mouser 80-C315C104M5U
C2-C3	47	Mouser 647-UVR1V470MDD
D1-D15	5 mm LED	Mouser 859-LTL2R3SEK
J1	2.1 mm barrel	Mouser 806-KLDX-0202-A
OSC*	0.1 pin socket	Mouser 506-510-AG90D
PCB		ExpressPCB.com
PGM	0.1 R/a header	Mouser 517-5111TG
R1	10k, 1/8 W	Mouser 299-10K-RC
R2*	10k, 1/8 W	Mouser 299-10K-RC
R3-R17	110, 1/8 W	Mouser 299-110-RC
Resonator*	4 MHz	Parallax 250-04050
Resonator*	20 MHz	Parallax 250-02060
Resonator*	50 MHz	Parallax 250-05060
S1	28-pin	Mouser 571-1-390261-9
S2	18-pin	Mouser 571-1-390261-5
S3	16-pin	Mouser 571-1-390261-4
SW1	4-pos DIP	Mouser 106-1204-EV
U1	SX28	Parallax SX28AC/DP
U2	ULN2803A	Mouser 511-ULN2803A
U3	ULN2003A	Mouser 511-ULN2003A
VR1	LF50CP-5.0	Mouser 511-LF50CP

* = optional components

Code Listing

```
' =====
'
' File..... Brake_Light_Buddy.SXB
' Purpose...
' Author.... Jon Williams
'           Copyright (c) 2008 Jon Williams
'           Some Rights Reserved
'           -- see http://creativecommons.org/licenses/by/3.0/
' E-mail.... jwilliams@efx-tek.com
' Started...
' Updated... 15 MAY 2008
'
' =====
'
' -----
' Program Description
' -----
'
' -----
' Conditional Compilation Symbols
' -----
'
' -----
' Device Settings
' -----
'
DEVICE          SX28, OSC4MHz, TURBO, STACKX, OPTIONX, BOR42
FREQ            4_000_000
ID              "Brakes"
'
' -----
' I/O Pins
' -----
'
DMode           PIN      RA  PULLUP           ' brightness & mode
Display         PIN      RBC                    ' output LEDs
'
' -----
' Constants
' -----
'
Yes             CON      1
No              CON      0
'
' -----
' Variables
' -----
'
flags           VAR      Byte
  isrFlag       VAR      flags.0
  invert        VAR      flags.1
'
brightness     VAR      Byte
acc            VAR      Byte
'
show           VAR      Byte           ' from select switch
cycles        VAR      Byte
'
tmpB1          VAR      Byte           ' for subs & funcs
tmpB2          VAR      Byte
tmpW1          VAR      Word
tmpW2          VAR      Word
tmpW3          VAR      Word
```

```

' =====
INTERRUPT NOPRESERVE 25_000
' =====

Mark_ISR:
    isrFlag = 1                                ' mark for timing

PWM_Ctrl:
    acc = acc + brightness                    ' update accumulator
    IF C = 1 THEN                              ' rollover?
        \ MODE $0F
        \ MOV !RB, #%00000000                ' yes, make LEDs outputs
        \ MOV !RC, #%00000000
    ELSE
        \ MODE $0F
        \ MOV !RB, #%11111111                ' no, disable LEDs
        \ MOV !RC, #%11111111
    ENDIF

RETURNINT

' =====
PROGRAM Start
' =====

' -----
' Subroutine / Function Declarations
' -----

DELAY_MS      SUB      1, 2                    ' shell for PAUSE
PLAY_SEQUENCE SUB      2

' -----
' Program Code
' -----

Start:
    brightness = %00111111                    ' set base to 25%
    brightness.6 = DMode.2                    ' add 25%
    brightness.7 = DMode.3                    ' add 50%

Main:
    show = DMode & %00000011                 ' mask off high bits

    IF show = %11 THEN Display_Hold
    IF show = %10 THEN Ziggy
    IF show = %01 THEN Out_In
    IF show = %00 THEN Crazy_Flasher

Ziggy:
    invert = No
    FOR cycles = 1 TO 3
        PLAY_SEQUENCE Zig_Zag
    NEXT
    Display = %0000_0000_0000_1111            ' fill gap in end of pattern
    DELAY_MS 100
    GOTO Display_Hold

Out_In:
    FOR cycles = 0 TO 1
        invert = cycles.0
        PLAY_SEQUENCE Explode
        PLAY_SEQUENCE Implode
    NEXT

```

```

GOTO Display_Hold

Crazy_Flasher:
invert = No
FOR cycles = 1 TO 2
    PLAY_SEQUENCE Dual_Flash
NEXT
GOTO Display_Hold

Display_Hold:
Display = %11111111111111111111      ' all on for hold
GOTO Display_Hold                    ' wait for reset

' -----
' Subroutine / Function Code
' -----

' Use: DELAY_MS duration
' -- duration is in milliseconds
' -- replacement for PAUSE

SUB DELAY_MS
IF __PARAMCNT = 1 THEN                ' byte value passed
    tmpW1 = __PARAM1
ELSE                                   ' word value passed
    tmpW1 = __WPARAM12
ENDIF
DO WHILE tmpW1 > 0                    ' milliseconds left?
    tmpB1 = 25                        ' yes, reload 1 ms timer
    DO WHILE tmpB1 > 0                ' let it expire
        \ CLRB isrFlag
        \ JNB isrFlag, @$
        \ DEC tmpB1
    LOOP
    DEC tmpW1                          ' update duration
LOOP
ENDSUB

' -----

' Use: PLAY_SEQUENCE Label
' -- bit 15 of pattern value signifies end-of-sequence

SUB PLAY_SEQUENCE
tmpW2 = __WPARAM12                    ' get start of sequence

READINC tmpW2, tmpB2                  ' read step timing
DO                                     ' run the sequence
    READINC tmpW2, tmpW3               ' get pattern bits
    IF invert = Yes THEN               ' inversion selected?
        tmpW3 = tmpW3 ^ 0x7FFF        ' invert all but bit 15
    ENDIF
    Display = tmpW3                    ' move pattern to LEDs
    DELAY_MS tmpB2                     ' hold
LOOP UNTIL tmpW3.15 = 1               ' if 1 then we're finished
ENDSUB

' -----

' User Data
' -----

' First byte in table is timing (ms) per table step
' Bit15 signifies the last bit in the sequence
' -- pattern is 15 bits wide

Zig_Zag:

```

```
DATA 100
WDATA %0_000000000000111
WDATA %0_000000000111000
WDATA %0_000000111000000
WDATA %0_000111000000000
WDATA %0_111000000000000
WDATA %0_000111000000000
WDATA %0_000000111000000
WDATA %1_000000000111000
```

Explode:

```
DATA 25
WDATA %0_000000000000000
WDATA %0_000000010000000
WDATA %0_000000111000000
WDATA %0_000001111100000
WDATA %0_000011111100000
WDATA %0_000011111110000
WDATA %0_001111111111000
WDATA %0_001111111111100
WDATA %0_011111111111110
WDATA %1_111111111111111
```

Implode:

```
DATA 25
WDATA %0_000000000000000
WDATA %0_100000000000001
WDATA %0_110000000000011
WDATA %0_111000000000011
WDATA %0_111100000001111
WDATA %0_111110000001111
WDATA %0_111110001111111
WDATA %0_111111000111111
WDATA %0_111111011111111
WDATA %1_111111111111111
```

Dual_Flash:

```
DATA 75
WDATA %0_111000011111000
WDATA %0_000000011111111
WDATA %0_111000011111000
WDATA %0_000000011111111
WDATA %0_111000011111000
WDATA %0_000000011111111
WDATA %0_111111100000000
WDATA %0_000111110000111
WDATA %0_111111100000000
WDATA %0_000111110000111
WDATA %0_111111100000000
WDATA %1_000111110000111
```

Tri_Flash:

```
DATA 100
WDATA %0_111110000011111
WDATA %0_000000000000000
WDATA %0_111110000011111
WDATA %0_000000000000000
WDATA %0_111110000011111
WDATA %0_000000000000000
WDATA %0_000001111100000
WDATA %0_000000000000000
WDATA %0_000001111100000
WDATA %0_000000000000000
WDATA %0_000001111100000
WDATA %0_000000000000000
WDATA %0_000001111100000
WDATA %1_000000000000000
```