AN02036: Making an LED glow using Pulse Density Modulation

Publication Date: 2026/4/15
Document Number: XM-015273-AN v1.1.1

IN THIS DOCUMENT

This application note describes how to control the brightness of an LED by implementing a Pulse Density Modulator, or PDM. This note focuses on the *xcore* aspects (I/O and timing) and we give a brief explanation on how a PDM is implemented.

Descriptions are provided of different, equally valid, methods to implement this on an *xcore*. The method chosen for ultimate deployment will typically depend on how it is integrated in a system.

1 Modulation

The code below uses Pulse Density Modulation to glow the LED. In Pulse Density Modulation you, on a clock beat, drive either a high signal or a low signal. You do this so that the average signal over many clock beats is the desired output.

The code we use implements a very basic Pulse Density Modulator that is suitable for glowing an LED. Let us assume that the level that we want the LED to glow at is a real number in the range [0..1] inclusive, where 0 means completely off, and 1 means completely on.

The trick is to maintain a fraction of time that we have yet to be on for. This is an accumulated error. On every clock-beat we add the level that we want to be at to the accumulated error, and if the result is 1.0 or more we switch the LED on for that next beat. Otherwise we switch the LED off for that beat:

- Add level to the accumulated error
- If the accumulated error is 1.0 or more then the LED must be ON for a clock-beat, and subtract we record this by subtracting 1.0 from the accumulated error
- Else LED is OFF for one clock-beat

Assuming that we want to drive 0.65 as a value, we go through the following sequence:

Error	Level	Error+Level	Output	New Error
0.0	0.65	0.65	OFF	0.65
0.65	0.65	1.30	ON	0.30
0.30	0.65	0.95	OFF	0.95
0.95	0.65	1.60	ON	0.60
0.60	0.65	1.25	ON	0.25
0.25	0.65	0.90	OFF	0.90
0.90	0.65	1.55	ON	0.55
0.55	0.65	1.20	ON	0.20
0.20	0.65	0.85	OFF	0.85

..and so on. You can see that the LED seems to be mostly going through a sequence ON ON OFF, (2/3, or 0.667); with the occasional extra OFF it will be on for 13 out of every 20 clock beats

In order to implement this, we have not used real numbers but instead integers in the range [0..100] inclusive, where 0 is OFF, and 100 is OFF. This means that the level can be seen as a percentage:

```
#include <xcore/port.h>
#include <xcore/clock.h>
#include <platform.h>

port_t led = PORT_LEDS;

void glow_fixed_simple(int percentage) {
    int accumulator = 0;
    for(int i = 0; i < 10000000; i++) {
        int on;
        accumulator += percentage;
        if (accumulator >= 100) {
            on = 1;
            accumulator -= 100;
        } else {
            on = 0;
        }
        port_out(led, on);
    }
}
```

There is one line in the code above which is not standard `C`, which is the function call `port_out(led, on)`. This call outputs (drives) the value found in the variable `on` onto the port found in `led`. In this case, `led` is a global variable that refers to the port on which the LEDs are connected. If `on` is the value `1`, then a high VDDIO signal will be driven, if `on` is the value `0`, then VSS (0 V) will be driven.

More information on *xcore* ports can be found in AN03000: xcore Input and Output and AN03007: xcore Ports.

2 Glow an LED Using a Timer

The above modulator switches the LED on and off, but the period of ON and OFF depends on the length of time executed in each of the branches of the IF statement.

We can make this precise by using a timer. We increment the timer at some fixed interval, and we wait for each next interval before we drive the port on or off.

```
#include <xcore/hwtimer.h>
#include <xcore/port.h>

void glow_fixed_timer(int percentage) {
    hwtimer_t tmr = hwtimer_alloc();
    int accumulator = 0;
    int next_time = hwtimer_get_time(tmr);
    for(int i = 0; i < 5000; i++) {              // five seconds
        int on;
```

(continues on next page)

(continued from previous page)

```
        accumulator += percentage;
        if (accumulator >= 100) {
            on = 1;
            accumulator -= 100;
        } else {
            on = 0;
        }
        next_time += 100000;                        // 1 ms
        hwtimer_set_trigger_time(tmr, next_time);
        (void) hwtimer_get_time(tmr);
        port_out(led, on);
    }
    hwtimer_free(tmr);
}
```

There are four calls involved in the timer. The first call `hwtimer_alloc()` allocates a hardware timer from the pool of timers. There is an associated `hwtimer_free(tmr)` at the end of the function to put the timer back in the pool.

Timers run at 100 MHz (100,000,000 Hz). The function `hwtimer_get_time(tmr)` gets the current value of the timer. We can add 100,000 to this timer value (100,000/100 MHz = 1 ms) and then set this time up as the *trigger*. Now that the trigger is set, getting the time will wait until the trigger time has been reached, and we can ignore the result as we know that it is one millisecond later, and we adjust the LED value. As we only ever add one millisecond at a time to the timer we never miss any time; this code will, on average, run at exactly 1 kHz, with a few nanoseconds of jitter on each edge.

Timers are documented in lib_xcore documentation.

3 Using a Port Clock

The final way to glow an LED we discuss is to use the port-clock. Each port has an associated clock signal that is used to operate the port. The port can only change on a positive clock edge, and by setting the port to be clocked at a known future edge we can achieve the same effect as using the timer. We keep track of the number of the clock edge in a variable `clock_edge_number`:

```
#include <xcore/port.h>
#include <stdint.h>

void glow_clocked(int percentage) {
    static int accumulator = 0;
    static uint16_t clock_edge_number = 0;
    accumulator += percentage;
    clock_edge_number += 50000;
    port_set_trigger_time(led, clock_edge_number);
    if (accumulator >= 100) {
        accumulator -= 100;
        port_out(led, 1);
    } else {
        port_out(led, 0);
    }
}
```

In this code we have removed the for-loop, and instead `glow_clocked()` will have to be called repeatedly. The variables `accumulator` and `clock_edge_number` have been declared `static` so that their value is maintained across function calls. We set the clock-edge-number at which we wish to output using `port_set_trigger_time()`, and we add 50,000 to the clock edge number on every iteration. As a port is clocked by default from a 100 MHz clock, this will result in a PDM that runs at exactly 2 kHz, with negligible jitter on the edges.

Port edges are only counted in a 16-bit number (the counter is therefore declared as a `uint16_t`), so 65,535 is the highest step we can make. More information on clocked ports can be found in AN03001: xcore Clocked Input and Output. In particular, that document tells you how to obtain the current clock-edge, or how to use clocks that are not 100 MHz.

4 Variable Glow

The previous two examples had a fixed level (eg, 65), in this example we pulse the value gently at 1 Hz using a sine waveform. For this we call the function `sinf` in order to calculate a raised sine wave. We use the API we defined earlier and use a clocked port from the previous section for creating a glow:

```
#include <math.h>

void pulse() {
    for(int i = 0; i < 10000; i++) {                // five seconds
        glow_clocked(50 + 50 * sinf(i / 2000.0 * 2 * 3.14));
    }
}
```

5 Example application

5.1 Building the example

This section assumes that the XMOS XTC Tools have been downloaded and installed. The required version is specified in the accompanying `README`.

Installation instructions can be found here.

Special attention should be paid to the section on Installation of Required Third-Party Tools.

The application is built using the xcommon-cmake build system, which is provided with the XTC tools and is based on CMake.

The `an02036` software ZIP package should be downloaded and extracted to a chosen working directory.

To configure the build, the following commands should be run from an XTC command prompt:

```
cd an02036
cd app_an02036
cmake -G "Unix Makefiles" -B build
```

All required dependencies are included in the software package. If any dependencies are missing, they will be retrieved automatically during this step.

The application binaries should then be built using `xmake`:

```
xmake -j -C build
```

Binary artifacts (.xe files) will be generated under the appropriate subdirectories of the `app_an02036/bin` directory — one for each supported build configuration.

For subsequent builds, the `cmake` step may be omitted. If `CMakeLists.txt` or other build files are modified, `cmake` will be re-run automatically by `xmake` as needed.

5.2 Running the example

From an XTC command prompt, the following command should be run from the `an02036/app_an02036` directory:

```
xrun ./bin/app_an02036.xe
```

Alternatively, the application can be programmed into flash memory for standalone execution:

```
xflash ./bin/app_an02036.xe
```


6 Summary

This application note provides an overview of several techniques for controlling LED brightness while demonstrating key *xcore* port operations. It covers the creation of a modulator, the use of timers and port clocks, and methods for achieving smooth and visually consistent LED illumination.