```cpp
#include "PWM.h"

namespace Hardware
{
    /// <summary>
    /// Constructeur
    /// </summary>
    /// <param name="pin">Pin</param>
    PWM::PWM(Pin pin)
    {
        this->pin = pin;

        // Check if PWM cape is loaded, if not load it
        if (!CapeLoaded(PWM_CAPE)) { Write(SLOTS, PWM_CAPE); }

        // Init the pin
        basepath = OCP_PATH;
        switch (pin)
        {
        case Hardware::PWM::P8_13:
            if (!CapeLoaded(P8_13_CAPE)) { Write(SLOTS, P8_13_CAPE_LOAD); }
            basepath.append(FindPath(P8_13_FIND));
            break;
        case Hardware::PWM::P8_19:
            if (!CapeLoaded(P8_19_CAPE)) { Write(SLOTS, P8_19_CAPE_LOAD); }
            basepath.append(FindPath(P8_19_FIND));
            break;
        case Hardware::PWM::P9_14:
            if (!CapeLoaded(P9_14_CAPE)) { Write(SLOTS, P9_14_CAPE_LOAD); }
            basepath.append(FindPath(P9_14_FIND));
            break;
        case Hardware::PWM::P9_16:
            if (!CapeLoaded(P9_16_CAPE)) { Write(SLOTS, P9_16_CAPE_LOAD); }
            basepath.append(FindPath(P9_16_FIND));
            break;
        }

        // Get the working paths
        dutypath = basepath + "/duty";
        periodpath = basepath + "/period";
        runpath = basepath + "/run";
        polaritypath = basepath + "/polarity";

        // Give Linux time to setup directory structure;
        usleep(250000);

        // Read current values
        period = StringToNumber<int>(Read(periodpath));
        duty = StringToNumber<int>(Read(dutypath));
        run = static_cast<Run>(StringToNumber<int>(Read(runpath)));
        polarity = static_cast<Polarity>(StringToNumber<int>(Read(polaritypath)));

        // calculate the current intensity
        calcIntensity();
    }


    PWM::~PWM()
    {
    }

    /// <summary>
    /// Calculate the current intensity
    /// </summary>
    void PWM::calcIntensity()
    {
        if (polarity == Normal)
        {
            if (duty == 0) { intensity = 0.0f; }
            else { intensity = (float)period / (float)duty; }
        }
        else
        {
            if (period == 0) { intensity = 0.0f; }
            else { intensity = (float)duty / (float)period; }
        }
    }

    /// <summary>
    /// Set the intensity level as percentage
    /// </summary>
    /// <param name="value">floating value multipication factor</param>
    void PWM::SetIntensity(float value)
    {
        if (polarity == Normal)
        {
            SetDuty(static_cast<int>((value * duty) + 0.5));
        }
        else
        {
            SetPeriod(static_cast<int>((value * period) + 0.5));
        }
    }

    /// <summary>
    /// Set the output as a corresponding uint8_t value
    /// </summary>
    /// <param name="value">pixel value 0-255</param>
    void PWM::SetPixelValue(uint8_t value)
    {
        if (period != 255) { SetPeriod(255); }
        SetDuty(255 - value);
        pixelvalue = value;
    }

    /// <summary>
    /// Set the period of the signal
    /// </summary>
    /// <param name="value">period : int</param>
    void PWM::SetPeriod(int value)
    {
        string valstr = NumberToString<int>(value);
        Write(periodpath, valstr);
        period = value;

        calcIntensity();
    }

    /// <summary>
    /// Set the duty of the signal
    /// </summary>
    /// <param name="value">duty : int</param>
    void PWM::SetDuty(int value)

    string valstr = NumberToString<int>(value);
    Write(dutypath, valstr);
    duty = value;

    calcIntensity();
}

/// <summary>
/// Run the signal
/// </summary>
/// <param name="value">On or Off</param>
void PWM::SetRun(Run value)
{
    int valInt = static_cast<int>(value);
    string valstr = NumberToString<int>(valInt);
    Write(runpath, valstr);
    run = value;
}

/// <summary>
/// Set the polarity
/// </summary>
/// <param name="value">Normal or Inverted signal</param>
void PWM::SetPolarity(Polarity value)
{
    int valInt = static_cast<int>(value);
    string valstr = NumberToString<int>(valInt);
    Write(runpath,valstr);
    polarity = value;
}

/// <summary>
/// Find the current PWM path in the OCP.3 directory
/// </summary>
/// <param name="value">part a the path name</param>
/// <returns>Returns the first found value</returns>
string PWM::FindPath(string value)
{
    auto dir = opendir(OCP_PATH);
    auto entity = readdir(dir);
    while (entity != NULL)
```

```
        {
            if (entity->d_type == DT_DIR)
            {
                string str = static_cast<string>(entity->d_name);
                if (str.find(value) != string::npos) { return str; }
            }
            entity = readdir(dir);
        }
        return "";
    }

}
```