
Mono Developer Documentation Documentation

Release 1

Monolit ApS

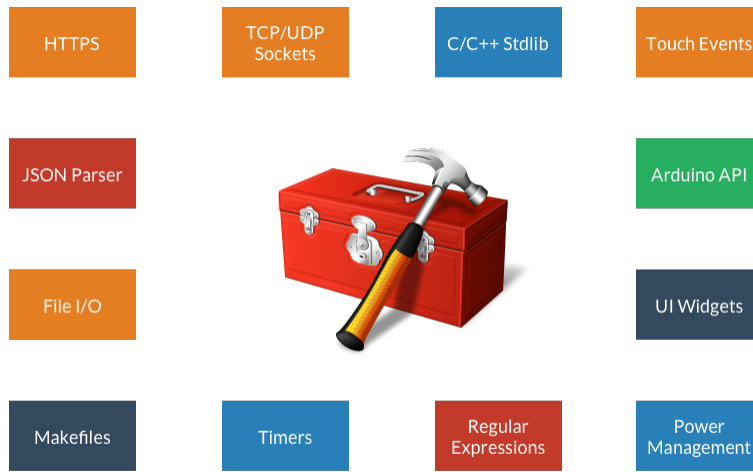
Sep 18, 2017

Contents

1	Content
----------	----------------

3

This is the developer resources for OpenMono SDK. You can find getting started tutorials, downloads, background articles and API references.



Getting Started

These guides will help you get started on creating your first mono app. We begin with guides that help you setup the toolchain and environment.

Lets start: Installing Mono Framework

In this guide we go through the steps of installing the Mono toolchain on your computer.

Download

First we begin with downloading the installer package, that will install the framework on your computer:

Download the installer package that fits your OS. Run the installer and follow the steps to install Mono's developing tools on your system.

The installer contains all you need to install apps on mono, and to develop your own apps. The installer package contains:

- **Mono Framework code:** The software library
- **GCC for embedded ARM:** Compiler
- **Binutils** (Windows only): The `make` tool
- **monoprog:** Tool that uploads apps to Mono via USB
- **monomake:** Tool that creates new mono application projects for you

Check installation

When the installer package has finished, you should check that have the toolchain installed. Open a terminal:

Mac & Linux

Open the *Terminal* application, and type this into it:

```
$ monomake
```

If you have installed the toolchain successfully in your path, the monomake tool should respond this:

```
ERR: No command argument given! You must provide a command
Usage:
monomake COMMAND [options]
Commands:
  project [name]  Create a new project folder. Default name is: new_mono_project
  version         Display the current version of monomake
  path           Display the path to the Mono Environment installation dir
```

Congratulations, you have the tool chain running! Now, you are ready to crate your first *Hello World* project in the next tutorial.

Any problems?

If you do not get the excepted response, but instead something like:

```
-bash: monomake: command not found
```

It means monomake is not in your PATH. Check if you can see a mono reference in your PATH, by typing:

```
$ echo $PATH
```

Look for references to something like `/usr/local/openmono/bin`. If you cannot find this, please restart the *Terminal* application to reload bash profile.

Windows

Open the Run command window (press Windows-key + R), type `cmd` and press Enter. The *Command Prompt* window should open. Check that the toolchain is correctly installed by typing:

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\stoffer> monomake
```

Like on Mac and Linux, monomake should respond with:

```
ERR: No command argument given! You must provide a command
Usage:
monomake COMMAND [options]
Commands:
  project [name]  Create a new project folder. Default name is: new_mono_project
  version         Display the current version of monomake
  path           Display the path to the Mono Environment installation dir
```

If you get this: Congratulations! You have the toolchain installed, and you can go ahead with creating your first *Hello World* app, in the next tutorial.

Any problems?

On the other hand, if you get:

```
'monomake' is not recognized as an internal or external command,  
operable program or batch file.
```

It means `monomake` is not in the environment variable `PATH`. Check that you really did install the tool chain, and that your system environment variable `PATH` does indeed contain a path like this:

```
C:\Program Files\OpenMono\bin
```

You can see your `PATH` environment variable by typing:

```
C:\Users\stoffer> echo %PATH%
```

The *Hello World* project

Now, let us create the obligatory *Hello World* project, that does not do anything else than verify your installation works.

Prerequisites

By now I assume you have installed the *OpenMono SDK*, as described in the previous tutorial. Also, it is best if you are familiar with *object oriented programming*. If you are not, then you might find yourself thinking “what the heck are classes and inheritance!” But read on anyways, but I will recommend to read our *C programmers guide to C++*.

Create a new project

Mono comes with a tool called `monomake`, that does one thing - and one thing only: creating new mono projects. Let's try it!

Open a terminal

- Mac/Linux: Open the Terminal application
- Window: Press Windows-key + R, and type `cmd` then hit Enter

Create project

In the terminal, navigate to the directory where you would like to create the project. Then:

```
$ monomake project hello_world
```

Hit Enter and `monomake` will create a new folder called `hello_world` with 3 files inside:

- `app_controller.h`
- `app_controller.cpp`
- `Makefile`

These 3 files are required for all mono applications. I will not go into too many details here, but just tell you that `app_controller.h` defines the class `AppController`, that is the application entry point. It replaces the `main()` function.

Now, `cd` into the project folder `hello_world`:

```
$ cd hello_world
```

Compile

The project already contains code that compiles, so the only thing you need to do is:

```
$ make
```

Now the tool chain compiles the application:

```
Compiling C++: app_controller.cpp
Compiling C++: System default mono_default_main
Linking hello_world.elf
```

Voila, your mono application compiled and the executable is `hello_world.elf`. This is the file that can be uploaded to Mono.

If you already have mono connected via USB, you can upload your new application to it by:

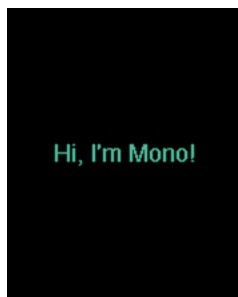
```
$ make install
```

The `install` command will search to any connected Mono's, reboot it and upload the application. If everything went smoothly you should see the text *Hi, I'm Mono* on the display.

The code

Okay, we got the code running on Mono - but what really happens in the code? In this section we will look at the template code in `AppController`.

First, let just describe what the application does. It creates a text on the screen that says: *"Hi, I'm Mono"*. That's it. More specific, it creates a `TextLabel` that gets the text content, and renders on the screen. I have includes a picture of the application below:



Header file

As said, all Mono applications needs an *AppController*, because it is the entry point for all mono applications. Let's take a look at the code in `app_controller.h`:

```

#include <mono.h>           // 1

using namespace mono;       // 2
using namespace mono::ui;

class AppController : public mono::IApplication { // 3

    TextLabelView helloLabel; // 4

public:

    AppController();          // 5

    void monoWakeFromReset(); // 6

    void monoWillGotoSleep(); // 7

    void monoWakeFromSleep(); // 8

};

```

I have added numbers to the interesting code lines in comments. Let's go through each of the lines, and see what it does:

1. We include the framework. This header file, is an umbrella that include all the classes in Mono framework. Every mono application need this include.
2. All mono framework classes exists inside a namespace called `mono`. We include namespace in the context, to make the code less verbose. This allows us to write `String()`, instead of `mono::String()`. (And yes, `mono` has its own string class!)
3. Here we define the `AppController` class itself. It inherits from the abstract interface called `IApplication`. This interface defines the 3 methods the `AppController` must have. We shall examine them shortly.
4. Here we define the `TextLabel` object that will display our text on the screen. It is defined as a member of the `AppController` class.
5. We overwrite the `default constructor` for our `AppController` class, to allow us to do custom initialization. You will see later why.
6. This is a required overwrite from the `IApplication` interface. It is a method that is called when `mono` is reset.
7. Also a requirement from `IApplication`. It defines a method that is called just before `mono` is put into sleep mode.
8. As required by `IApplication`, this method is called when `mono` wake up from sleep mode.

All `AppController`'s are required to implement 6,7 and 8, but you may just leave them blank.

Implementation

Now, the contents of: `app_controller.cpp` file:

```

#include "app_controller.h"

using namespace mono::geo;

```

```
AppController::AppController() :  
    helloLabel(Rect(0,100,176,20), "Hi, I'm Mono!")           // 1  
{  
  
    helloLabel.setAlignment(TextLabelView::ALIGN_CENTER); // 2  
  
    helloLabel.setTextColor(display::TurquoiseColor);       // 3  
}  
  
void AppController::monoWakeFromReset()  
{  
    helloLabel.show();                                       // 4  
}  
  
void AppController::monoWillGotoSleep()  
{  
  
}  
  
void AppController::monoWakeFromSleep()  
{  
    helloLabel.scheduleRepaint();                             // 5  
}
```

Again, I have numbered the most interesting code lines:

1. This the default constructor overwrite. We overwrite the constructor to construct the *TextLabel* object with specific parameters. (See *TextLabelView* reference) We set the labels position and size on the screen (using the *Rect class*), and its text content.
2. In (1) we defined the text labels width to be the entire screen (176 pixels). We want to center the text on the screen, therefore we tell the label to center align its text content.
3. To make application look fancy, we set the text color to be an artzy turquoise color.
4. The method `monoWakeFromReset` is automatically called upon reset. Inside here we tell the text label to be visible. All UI widgets are hidden by default. You must call `show()` to render them.
5. `monoWakeFromSleep` is called when Mono wakes from sleep mode. Here we tell the label to repaint (render) itself on the screen. Sleep mode might have cleared the display memory, so we need to render the label again. `scheduleRepaint` will render the text, when the display signals its time to update.

That is all the code you need to draw on the screen. Notice that we left the method `monoWillGotoSleep` empty. We do not need any clean up code, before mono goes to sleep.

Sleep mode

But how and when will Mono go into sleep mode? Easy: By default the side-button on Mono will trigger sleep and wake. You do not have to do anything! Sleep mode will turn off all peripherals and halt the CPU execution. Only a button press will wake it. Sleep mode is only way you can turn off Mono!

Further reading

- *Your first App* : Build a Tic Tac Toe game (Part 1)
- *Archectural Overview* : Learn more about sleep/wake and `IApplication`

- *Display System Architecture* : An in-depth look on details of the display system.

Tic-tac-toe for Mono

In this tutorial I will teach you how to program Mono's display and touch device by creating a tiny game.

Anatomy of a Mono application

Mono apps can be written inside the Arduino IDE, but if you really want be a pro, you can write Mono apps directly in C++. For that you will need to implement an `AppController` with at least three functions. So I will start there, with my `app_controller.h` header file:

```
#include <mono.h>

class AppController
:
    public mono::IApplication
{
public:
    AppController ();
    void monoWakeFromReset ();
    void monoWillGotoSleep ();
    void monoWakeFromSleep ();
};
```

My matching `app_controller.cpp` implementation file will start out as this:

```
#include "app_controller.h"

AppController::AppController ()
{
}

void AppController::monoWakeFromReset ()
{
}

void AppController::monoWakeFromSleep ()
{
}

void AppController::monoWillGotoSleep ()
{
}
```

Now I have a fully functional Mono application! It does not do much, but hey, there it is.

Screen and Touch

Tic Tac Toe is played on a 3-by-3 board, so let me sketch out the layout something like this:

```

Tic Tac Toe
+---+ +---+ +---+
|   |   |   |
+---+ +---+ +---+
```

```
+---+ +---+ +---+
|   | |   | |   |
+---+ +---+ +---+
+---+ +---+ +---+
|   | |   | |   |
+---+ +---+ +---+
```

I will make the `AppController` hold the board as an array of arrays holding the tokens X and O, and also a token `_` for an empty field:

```
class AppController
{
    ...
    enum Token { _, X, O };
    Token board[3][3];
};
```

For simplicity, I do not want Mono to make any moves by itself (yet); I just want two players to take turns by touching the board. So I need to show the board on the screen, and I want each field of the board to respond to touch.

This kind of input and output can in Mono be controlled by the `ResponderView`. It is a class that offers a lot of functionality out of the box, and in my case I only need to override two methods, `repaint` for generating the output and `touchBegin` for receiving input:

```
class TouchField
:
    public mono::ui::ResponderView
{
    void touchBegin (mono::TouchEvent &);
    void repaint ();
};

class AppController
{
    ...
    TouchField fields[3][3];
};
```

Above I have given `AppController` nine touch fields, one for each coordinate on the board. To make a `TouchField` able to paint itself, it needs to know how to get hold of the token it has to draw:

```
class TouchField
{
    ...
public:
    AppController * app;
    uint8_t boardX, boardY;
};
```

With the above information, I can make a `TouchField` draw a circle or a cross on the screen using the geometric classes `Point`, `Rect`, and the underlying functionality it inherits from `ResponderView`. The `ResponderView` is a subclass of `View`, and all `Views` have a `DisplayPainter` named `painter` that takes care of actually putting pixels on the screen:

```

using mono::geo::Point;
using mono::geo::Rect;

void TouchField::repaint ()
{
    // Clear background.
    painter.drawFillRect(viewRect,true);
    // Show box around touch area.
    painter.drawRect(viewRect);
    // Draw the game piece.
    switch (app->board[boardY][boardX])
    {
        case AppController::X:
        {
            painter.drawLine(Position(),Point(viewRect.X2(),viewRect.Y2()));
            painter.drawLine(Point(viewRect.X2(),viewRect.Y()),Point(viewRect.X(),
↪viewRect.Y2()));
            break;
        }
        case AppController::O:
        {
            uint16_t radius = viewRect.Size().Width() / 2;
            painter.drawCircle(viewRect.X()+radius,viewRect.Y()+radius,radius);
            break;
        }
        default:
            // Draw nothing.
            break;
    }
}

```

Above, I use the View's `viewRect` to figure out where to draw. The `viewRect` defines the View's position and size on the screen, and its methods `X()`, `Y()`, `X2()`, and `Y2()` give me the screen coordinates of the View. The method `Position()` is just a shortcut to get `X()` and `Y()` as a `Point`.

With respect to the board, I index multidimensional arrays by [row-major order](#) to please you old-school C coders out there. So it is `board[y][x]`, thank you very much.

Well, now that each field can draw itself, we need the `AppController` to setup the board and actually initialise each field when a game is started:

```

using mono::ui::View;

void AppController::startNewGame ()
{
    // Clear the board.
    for (uint8_t x = 0; x < 3; ++x)
        for (uint8_t y = 0; y < 3; ++y)
            board[y][x] = _;
    // Setup touch fields.
    const uint8_t width = View::DisplayWidth();
    const uint8_t height = View::DisplayHeight();
    const uint8_t fieldSize = 50;
    const uint8_t fieldSeparation = 8;
    const uint8_t screenMargin = (width-(3*fieldSize+2*fieldSeparation))/2;
    uint8_t yOffset = height-width-(fieldSeparation-screenMargin);
    for (uint8_t y = 0; y < 3; ++y)
    {

```

```
yOffset += fieldSeparation;
uint8_t xOffset = screenMargin;
for (uint8_t x = 0; x < 3; ++x)
{
    // Give each touch field enough info to paint itself.
    TouchField & field = fields[y][x];
    field.app = this;
    field.boardX = x;
    field.boardY = y;
    // Tell the view & touch system where the field is on the screen.
    field.setRect(Rect(xOffset,yOffset,fieldSize,fieldSize));
    // Next x position.
    xOffset += fieldSize + fieldSeparation;
}
// Next y position.
yOffset += fieldSize;
}
continueGame();
}
```

Above I space out the fields evenly on the bottom part of the screen, using the `DisplayWidth()` and `DisplayHeight()` to get the full size of the screen, and while telling each field where it should draw itself, I also tell the field which board coordinate it represents.

Before we talk about the game control and implement the function `continueGame`, let us hook up each field so that it responds to touch events:

```
using mono::TouchEvent;

void TouchField::touchBegin (TouchEvent & event)
{
    app->humanMoved(boardX,boardY);
}
```

Above the touch event is implicitly translated to a board coordinate (because each field knows its own board coordinate) and passed to the `AppController` that holds the board and controls the game play.

Game status display

To inform the players what is going on, I want the top of the display to show a status message. And I also want to keep track of which player is next:

```
class AppController
{
    ...
    ...
    mono::ui::TextLabelView topLabel;
    Token nextToMove;
};

using mono::ui::TextLabelView;

AppController::AppController ()
:
    topLabel(Rect(0,10,View::DisplayWidth(),20),"Tic Tac Toe")
{
}
```



```

topLabel.setAlignment(TextLabelView::ALIGN_CENTER);
}

```

A `TextLabelView` is a `View` that holds a piece of text and displays this text inside its `viewRect`. I can now change the label at the top of the screen depending on the state of the game after each move by using `setText()`, followed by a call to `show()` to force the `TextLabelView` to repaint:

```

void AppController::continueGame ()
{
    updateView();
    whosMove();
    if (hasWinner())
    {
        if (winner() == X) topLabel.setText("X wins!");
        else topLabel.setText("O wins!");
    }
    else if (nextToMove == _) topLabel.setText("Tie!");
    else if (nextToMove == X) topLabel.setText("X to move");
    else topLabel.setText("O to move");
    topLabel.show();
}

```

The `updateView()` function simply forces all the fields to repaint:

```

void AppController::updateView ()
{
    for (uint8_t y = 0; y < 3; ++y)
        for (uint8_t x = 0; x < 3; ++x)
            fields[y][x].show();
}

```

Game control

I now need to implement functionality that decides which player should move next and whether there is a winner. First, I can figure out who's turn it is by counting the number of game pieces for both players, and placing the result in `nextToMove`. If `nextToMove` gets the value `_`, then it means that the board is full:

```

void AppController::whosMove ()
{
    uint8_t xPieces = 0;
    uint8_t oPieces = 0;
    for (uint8_t y = 0; y < 3; ++y)
        for (uint8_t x = 0; x < 3; ++x)
            if (board[y][x] == X) ++xPieces;
            else if (board[y][x] == O) ++oPieces;
    if (xPieces + oPieces >= 9) nextToMove = _;
    else if (xPieces <= oPieces) nextToMove = X;
    else nextToMove = O;
}

```

Finding out whether there is a winner is just plain grunt work, checking the board for three-in-a-row:

```

bool AppController::hasThreeInRow (Token token)
{
    // Check columns.
    for (uint8_t x = 0; x < 3; ++x)

```

```
        if (board[0][x] == token &&
            board[1][x] == token &&
            board[2][x] == token
        ) return true;
    // Check rows.
    for (uint8_t y = 0; y < 3; ++y)
        if (board[y][0] == token &&
            board[y][1] == token &&
            board[y][2] == token
        ) return true;
    // Check diagonal.
    if (board[0][0] == token &&
        board[1][1] == token &&
        board[2][2] == token
    ) return true;
    // Check other diagonal.
    if (board[0][2] == token &&
        board[1][1] == token &&
        board[2][0] == token
    ) return true;
    return false;
}

AppController::Token AppController::winner ()
{
    if (hasThreeInRow(X)) return X;
    if (hasThreeInRow(O)) return O;
    return _;
}

bool AppController::hasWinner ()
{
    return winner() != _;
}
```

Lastly, I need to figure out what to do when a player touches a field. If the game has ended, one way or the other, then I want to start a new game, no matter which field is touched; If the player touches a field that is already occupied, then I ignore the touch; Otherwise, I place the proper piece on the board:

```
void AppController::humanMoved (uint8_t x, uint8_t y)
{
    if (nextToMove == _ || hasWinner()) return startNewGame();
    else if (board[y][x] != _) return;
    else board[y][x] = nextToMove;
    continueGame();
}
```

Fallen asleep?

To wrap things up, I want Mono to start a new game whenever it comes out of a reset or sleep:

```
void AppController::monoWakeFromReset ()
{
    startNewGame();
}
```

```
void AppController::monoWakeFromSleep ()
{
    startNewGame();
}
```

Well there you have it: An astonishing, revolutionary, new game has been born! Now your job is to type it all in.

Tic-tac-toe for Mono, part II

In the *first part*, you saw how to get Mono to draw on the screen and how to react to touch input.

In this second part, I will show you how to use timers to turn Mono into an intelligent opponent!

Growing smart

To get Mono to play Tic Tac Toe, I will need to give it a strategy. A very simple strategy could be the following:

1. Place a token on an empty field if it makes Mono win.
2. Place a token on an empty field if it blocks the human opponent from winning.
3. Place a token arbitrarily on an empty field.

Well, it is not exactly Skynet, but it will at least make Mono appear to have some brains. In code it translates to the following.

```
void AppController::autoMove ()
{
    timer.Stop();
    // Play to win, if possible.
    for (uint8_t x = 0; x < 3; ++x)
        for (uint8_t y = 0; y < 3; ++y)
            if (board[y][x] == _)
            {
                board[y][x] = O;
                if (hasWinner()) return continueGame();
                else board[y][x] = _;
            }
    // Play to not loose.
    for (uint8_t x = 0; x < 3; ++x)
        for (uint8_t y = 0; y < 3; ++y)
            if (board[y][x] == _)
            {
                board[y][x] = X;
                if (hasWinner())
                {
                    board[y][x] = O;
                    return continueGame();
                }
                else board[y][x] = _;
            }
    // Play where free.
    for (uint8_t x = 0; x < 3; ++x)
        for (uint8_t y = 0; y < 3; ++y)
            if (board[y][x] == _)
            {
                board[y][x] = O;
```

```
        return continueGame();
    }
}
```

The timer is what controls when Mono should make its move; it is a Mono framework `Timer` that can be told to trigger repeatedly at given number of milliseconds. I will make the application fire the timer with 1.5 second intervals:

```
class AppController
{
    ...
private:
    mono::Timer timer;
    void autoMove ();
    void prepareNewGame ();
};

AppController::AppController ()
:
    timer(1500)
{
    ...
}
```

I will control the application by telling `timer` to call various functions when it triggers, and then stop and start the timer where appropriate. Conceptually, I can simply tell `timer` to call a function `autoMove` by

```
timer.setCallback(autoMove);
```

but because `autoMove` is a C++ class member-function, I need to help out the poor old C++ compiler by giving it information about which object has the `autoMove` function, so the incantation will actually be

```
timer.setCallback<AppController>(this, &AppController::autoMove);
```

With that cleared up, I can place the bulk of the control in the `continueGame` function:

```
void AppController::continueGame ()
{
    updateView();
    whosMove();
    if (hasWinner())
    {
        if (winner() == X) topLabel.setText("You win!");
        else topLabel.setText("Mono wins!");
        timer.setCallback<AppController>(this, &AppController::prepareNewGame);
        timer.start();
    }
    else if (nextToMove == _)
    {
        topLabel.setText("Tie!");
        timer.setCallback<AppController>(this, &AppController::prepareNewGame);
        timer.start();
    }
    else if (nextToMove == X)
    {
        topLabel.setText("Your move");
        topLabel.show();
    }
}
```

```

    }
    else
    {
        topLabel.setText("Thinking...");
        timer.setCallback<AppController> (this, &AppController::autoMove);
        timer.start();
    }
}

```

All that is missing now is a `prepareNewGame` function that prompts for a new game:

```

void AppController::prepareNewGame ()
{
    timer.stop();
    topLabel.setText("Play again?");
}

```

And that is it! Now you can let your friends try to beat Mono, and when they fail, you can tell them that *you* created this master mind.

Tic-tac-toe for Mono, part III

In the *first part*, you saw how to get Mono to draw on the screen and how to react to touch input.

In the *second part*, you saw how to use timers to turn Mono into an intelligent opponent.

In this third part, I will show you how to extend battery life and how to calibrate the touch system.

Getting a Good Night's Sleep

It is important to automatically put Mono to sleep if you want to conserve your battery. The battery lasts less than a day if the screen is permanently turned on. On the other hand, if Mono only wakes up every second to make a measurement of some sort, then the battery will last a year or thereabouts. What I will do in this app, is something in between these two extremes.

In it's simplest form, an auto-sleeper looks like this:

```

class AppController
{
    ...
private:
    mono::Timer sleeper;
    ...
};

AppController::AppController ()
:
    sleeper(30*1000,true),
    ...
{
    sleeper.setCallback(mono::IApplicationContext::EnterSleepMode);
    ...
}

void AppController::continueGame ()

```

```
{
    sleeper.start();
    ...
}
```

The `sleeper` is a single-shot `Timer`, which means that it will only fire once. And by calling `start` on `sleeper` every time the game proceeds in `continueGame`, I ensure that timer is restarted whenever something happens in the game, so that `EnterSleepMode` is only called after 30 seconds of inactivity.

It is Better to Fade Out than to Black Out

Abruptly putting Mono to sleep without warning, as done above, is not very considerate to the indecisive user. And there is room for everyone here in Mono world.

So how about slowly fading down the screen to warn about an imminent termination of the exiting game?

Here I only start the `sleeper` timer after the display has been dimmed:

```
class AppController
{
    ...
private:
    mono::Timer dimmer;
    void dim ();
    ...
};

using mono::display::IDisplayController;

AppController::AppController ()
:
    dimmer(30*1000,true),
    ...
{
    dimmer.setCallback<AppController>(this, &AppController::dim);
    ...
}

void AppController::dim ()
{
    dimmer.stop();
    IDisplayController * display = IApplicationContext::Instance->DisplayController;
    for (int i = display->Brightness(); i >= 50; --i)
    {
        display->setBrightness(i);
        wait_ms(2);
    }
    sleeper.start();
}
```

The `dimmer` timer is started whenever there is progress in the game, and when `dimmer` times out, the `dim` method turns down the `brightness` from the max value of 255 down to 50, one step at a time.

Oh, I almost forgot, I need to turn up the brightness again when the the `dimmer` resets:

```
void AppController::continueGame ()
{
```

```
IApplicationContext::Instance->DisplayController->setBrightness(255);
sleeper.stop();
dimmer.start();
...
}
```

So there you have it, saving the environment and your battery at the same time!

Mono for Arduino Hackers

You can use the familiar Arduino IDE to build Mono applications. This guide will take you through the steps of how to do this.

Prerequisites

I assume you are familiar with Arduino, its coding IDE and the API's like `pinMode()` etc. I also assume that you have the IDE installed, and it is version 1.6.7 or above. You do not have to follow any other of the getting started guides. Arduino IDE development for Mono is completely independent. If this is the first Mono guide you read, it is all good.

Overview

You can code Mono using 2 approaches: Native Mono or the Arduino IDE. The difference is the tools you use, and the way you structure your code. In the Arduino approach you get the familiar `setup()` and `loop()` functions, and you use the Arduino IDE editor to code, compile and upload applications to Mono.

Under the hood we still use the native Mono API's and build system, we just encapsulate it in the Arduino IDE.

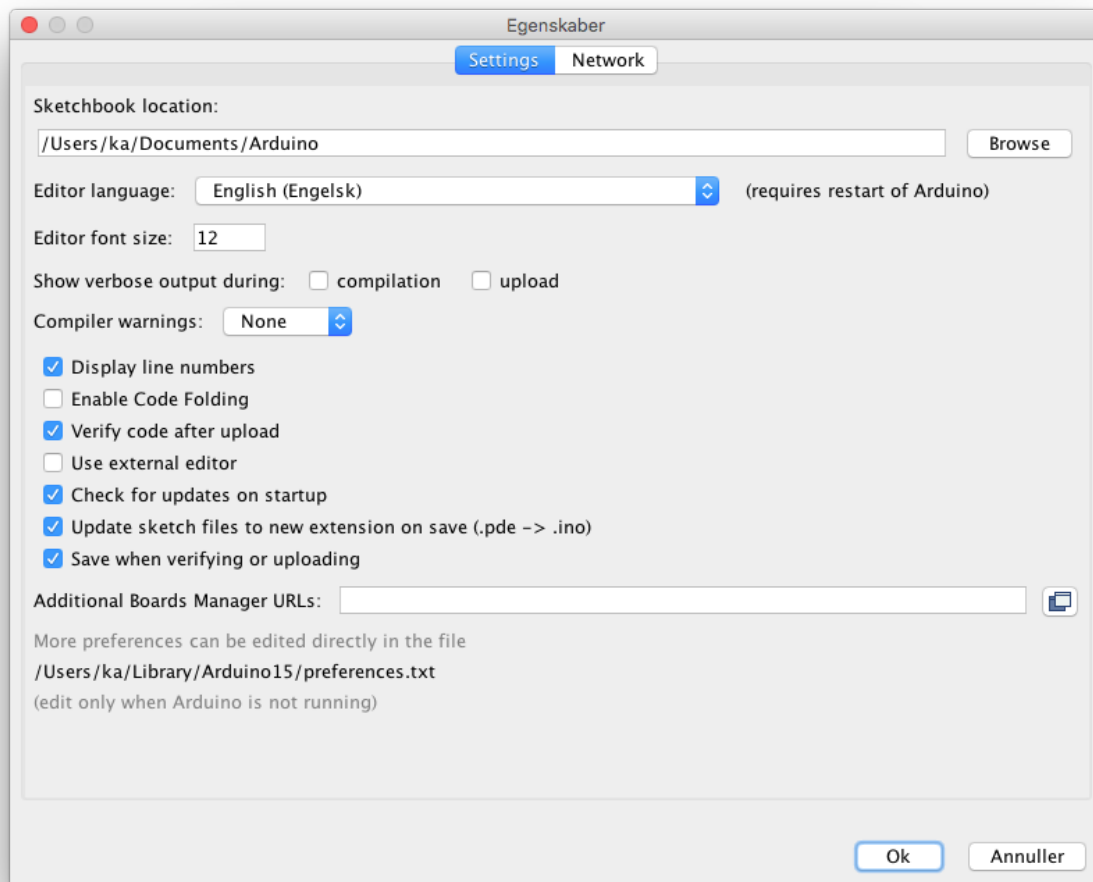
Installation

The Arduino IDE has a plugin system, where you can add support for third-party boards. We use such a plugin, that adds Mono as a target board. To install the plugin we use the *Board Manager*, that can install new target boards.

Note: You need Arduino IDE version 1.6 or above to utilize the Board Manager feature. You can download Arduino IDE here: arduino.cc

Add Mono as a board source

To make the Board Manager aware of Mono's existence, you must add a source URL to the manager. You do this by opening the preferences window in Arduino IDE. Below is a screenshot of the window:



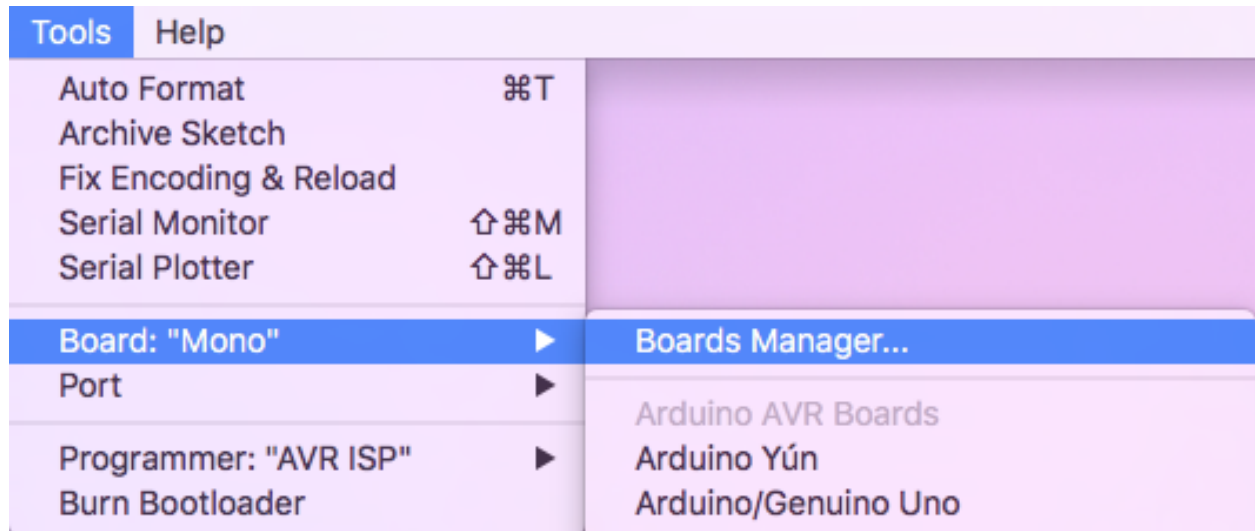
In the text field called *Additional Boards Manager URLs* type in the URL for Mono board package:

```
https://github.com/getopenmono/arduino_comp/releases/download/current/package_
↪openmono_index.json
```

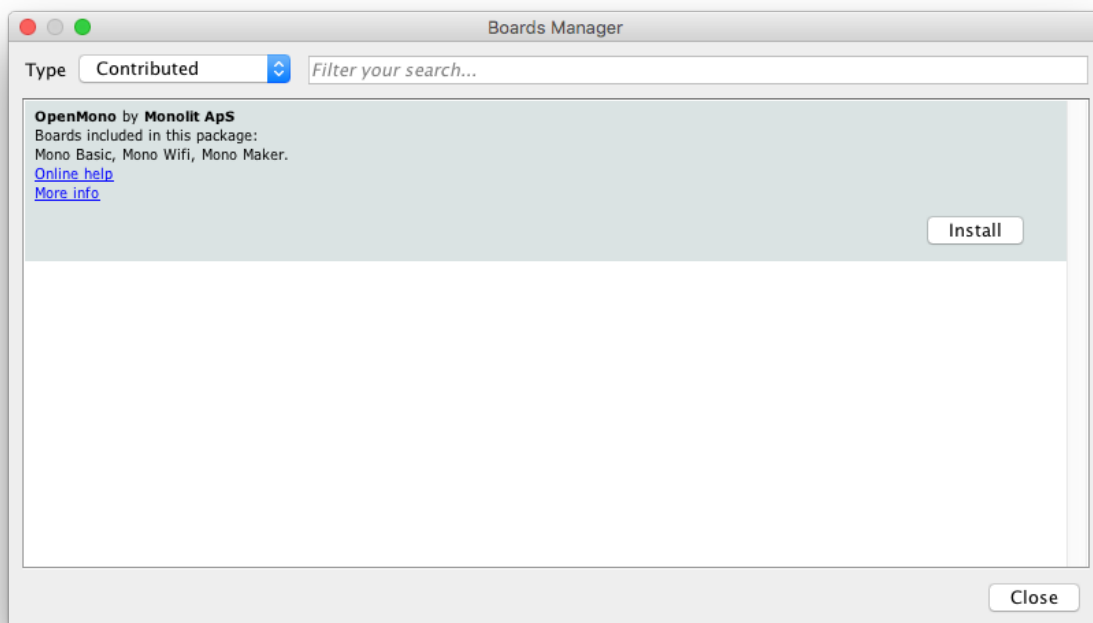
And press *OK*.

Install the board package

Now, open the *Board Manager* by selecting the menu: *Tools -> Boards -> Board Manager*:



The *Board Manager* appears, and query the source URLs for new board types. It will discover a new board type OpenMono. Select the type *Contributed*, in top left corner:



Now click on the *Install* button to download and install all the needed tools to build mono applications. This might take a few minutes.

When the installation is done, you can close the *Board Manager* and return the main window. Now select Mono from the list of available target boards:



Install the USB Serial Port Driver

If you run Windows, there is an additional step. (Mac users, you can skip this section.) Windows need to detect the Mono hardware as an USB CDC device and create an ol' fashion COM port. So download the USB device definition driver by right clicking the link and choosing *Save file as*:

Download Windows Serial Port Driver

Run the installer, and you are ready to use Mono.

Limitations

The standard Arduino boards are much simpler than Mono. For example: They can be turned off and they have bare pin headers. Arduino API are made to make digital and analog communication simple. You have functions like `digitalWrite` and `analogRead`. While you have these functions available, you do not have any pin headers sticking out of Mono chassis! You need the *Arduino Shield Adaptor* or to build your own hardware to really take advantage of the Arduino API.

Mono's API is much more high-level, meaning that you have functions like *Render text on the screen*, and the software library (Mono Framework) will handle all the communication for you. Luckily you can still do this from inside the Arduino IDE.

Our additions to the default Arduino sketch

There are some key differences between Arduino and Mono, most important the power supply. You can always turn off an Arduino by pulling its power supply, but that is not true for mono. Here power is controlled by software.

By default we have added some code to the Arduino sketch template, so it will power on/off when pressing the User button. Also, we added the text *Arduino* to the display, such that you know your Mono is turned on.

Hello World

Let us build a Hello World application, similar to the one in the *The obligatory Hello World project* guide. We start out with the default Arduino project template:

```
void setup() {  
    // put your setup code here, to run once:  
  
}  
  
void loop() {  
    // put your main code here, to run repeatedly:  
  
}
```

We will use the class `TextLabelView` to display text on the screen. A `TextLabel` has a size and a position on the screen, which is defined by the class `Rect` that represents a rectangle.

Context issue

You might think we just create the `TextLabel` in the `setup()` function like this:

```
void setup() {  
    // put your setup code here, to run once:  
  
    mono::ui::TextLabelView helloLbl;  
  
}
```

But this approach will deallocate the `TextLabel` as soon as the `setup()` function returns. This means it cannot be rendered to the screen, because it has to be present in memory when screen repaints occur.

The correct approach here is to create a class (say *MyClass*), and let the *TextLabel* be a member of that class. We then create an object of the class in the global context. (Outside the `setup()` and `loop()` functions.) But all this will be out of scope with this tutorial, so we will do it the ugly way. Just know that having many global context objects, is a bad thing programmatically.

Adding the TextLabel

The complete code added to the project global context and in the `setup()` function:

```
#include <mono.h> // 1

mono::ui::TextLabelView textLbl; // 2

void setup() {

    textLbl = mono::ui::TextLabelView(mono::geo::Rect(30, 73, 176, 20), "Hi, I'm Mono");
    // 3
    textLbl.show(); // 4
}

void loop() {
}
```

I have numbered the interesting source code lines, let go through them one by one:

1. We include the Mono Framework, to have access to Mono's API.
2. Here we define the global *TextLabel* object called `textLbl`. Because it is global it will stick around and not be deallocated.
3. Construct a *TextLabelView* with a rectangle object (*Rect*), and give the position (30, 73) and dimension (176, 20). In the constructor's second parameters we set the text content on the *TextLabel*. This is the text that will be displayed on the screen.
4. We tell the *TextLabel* to render itself on the screen. All UI widgets are hidden by default. You must call `show()` to render them.

Now you can press the compile button () and see the code compile. If you have Mono connected you can upload the application by pressing the button.

Notice that we did not need to put any code inside the `loop()` function.

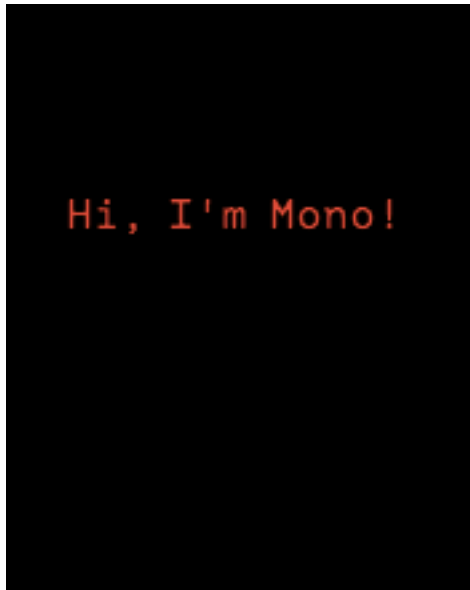
Enhancing the look and feel

To make our Hello World exactly like the *The obligatory Hello World project* guide, we need to add some refinements to it. We need to center the text on the screen and to color it a fancy red color. But that's easy, just two calls added to the `setup()` function:

```
textLbl.setAlignment(mono::ui::TextLabelView::ALIGN_CENTER);

textLbl.setTextColor(mono::display::AlizarinColor);
```

Now, if you build and run the application the text will be a fancy color and centered on the screen:



C++ from Arduino IDE

Even though Arduino is actually coded in C++, the C++ features are not utilized (that much). Some programmers might think they code in C, and not C++ - which is to some extent correct. Still, Arduino IDE uses the C++ compiler.

In Mono's SDK we use the C++ feature set more extensively. This means that C developers might see unfamiliar *keywords* or syntax constructs, like `new` and `::`.

If you are not familiar with C++, I urge you to read our *C programmer's guide to C++*. This guide is a brief tour of C++ explained for C developers.

Further reading

Now you know how to build mono applications from the Arduino IDE. You might want to dive into the native API and build system, or reading one of the in-depth articles:

- *C programmer's guide to C++* : How C++ differs from good old C
- *Install the native framework* : Install the native mono build system
- *Architectural Overview* : About application lifecycles and structure.

Happy hacking.

Common Misconceptions & Worst Practice

To clear out, what we imagine will be common mistakes, let's go through some scenarios that you should avoid - at least!

Who should read this?

Mono API and application structure might be new to you, if you previously programmed only for Arduino or similar embedded devices. We are aware of our framework might be quite unfamiliar to bare metal developers, who expect to have full access and control, from `main()` to `return 0`.

Mono Framework is advanced and its good performance depends on you, following the best practice guide lines. Read this, and you can avoid the most basic mistakes that degrade Mono's functionality.

No `while(1)`'s

First, never ever create your own run loop! Never do this:

```
void AppController::monoWakeFromReset ()
{
    // do one time setups ...

    // now lets do repetitive tasks, that I want to control myself
    while(1)
    {
        //check status of button

        // check something else

        // maybe update the screen

        // increment some counter
    }
    // we will never return here
}
```

Try to do this, and you will find Mono completely unresponsive. The USB port will not work, the programmer (monoprog) will not work, along with a whole bunch of other stuff.

Like other applications in modern platforms, Mono applications uses an internal run loop. If you create your own, the internal run loop will not progress. All features that depend on the run loop will not work. Timers will not run, display system will not work, and worst `monoprog` cannot reset mono, to upload a new app.

If you want to do repetitive tasks, that should run always (like `while(1)`), you should instead utilize the run loop. You can inject jobs into the run loop by implementing an interface called *IRunLoopTask*. This will allow you to define a method that gets called on each run loop iteration. That's how you do it. We shall not go into more details here, but just refer to the tutorial [Using the Run Loop](#)

No busy waits

Many API's (including Mono's), allows you to do busy waits like this:

```
// do something
wait_ms(500); // wait here for 0.5 secs

// do something else
```

It is really convenient to make code like this, but it is bad for performance! For half a second you just halt the CPU - it does nothing. The application run loop is halted, so all background tasks must wait as well. The CPU runs at 66 Mhz, imagine all code it could have executed in that half second!

Instead of halting the CPU like this, you should use callbacks to allow the CPU to do other stuff, while you wait:

```
// do someting
mono::Timer::callOnce<MyClass>(500, this, &MyClass::doSomethingElse); // do
↪ something else in 0.5 secs
```

By using the *Timer* class, and encapsulating the “do something else” functionality in a method - you free the CPU to do useful stuff while you wait. To learn more about callbacks see the tutorial: [Callbacks in C++](#).

Extensive use of new or malloc

The C++ new operator uses the *stdlib* function *malloc* to allocate memory on the heap. And it is very easy and convenient to use the heap:

```
// object allocation on the heap - because Qt and Obj-C Cocoa uses this scheme!
mono::geo::Rect *bounds = new mono::geo::Rect(0,0,100,20);
mono::ui::TextLabelView *txtLbl = new mono::ui::TextLabelview(*bounds, "I'm on the_
↪heap!");

//pass the pointer around
return txtLbl;
```

What happened to the *bounds* pointer, that had a reference to a *Rect* object? Nothing happened, the object is still on the heap and we just lost the reference to it. Our application is leaking memory. And that is one issue with using the heap. We do not have a *Garbage Collector*, so you must be careful to always free your objects on the heap.

And it gets worse, the heap on Mono PSoC5 MCU is not big - it is just 16 Kb. You might run out of heap quicker than you expect. At that point *malloc* will start providing you with *NULL* pointers.

Use heap for Asynchronous tasks

There are some cases where you must use the heap, for example this will not work:

```
void getTemp()
{
    // say we got this variable from the temperature sensor
    int celcius = 22;

    char tempStr[100]; // make a local buffer variable to hold our text

    // format a string of max 100 chars, that shows the temperature
    snprintf(tempStr, 100, "the temperature is: %i C", celcius);

    renderOnDisplayAsync(tempStr);
}
```

Here we have an integer and want to present its value nicely wrapped in a string. It is a pretty common thing to do in applications. The issue here is that display rendering is asynchronous. The call to *renderOnDisplayAsync* will just put our request in a queue, and then return. This means our buffer is removed (deallocated) as soon as the *getTemp()* returns, because it is on the stack.

Then, when its time to render the display there is no longer a *tempStr* around. We could make the string buffer object global, but that will take up memory - especially when we do not need the string.

In this case you should the heap! And luckily we made a *String* class that does this for you. It store its content on the heap, and keeps track of references to the content. As soon as you discard the last reference to the content, it is automatically freed - no leaks!

The code from above becomes:

```
int celcius = 22; // from the temp. sensor
```

```
// lets use mono's string class to keep track of our text
mono::String tempStr = mono::String::Format("the temperature is: %i C", celcius);

renderOnDisplayAsync(tempStr);
```

That's it. Always use Mono's *String* class when handling text strings. It is lightweight, uses data de-duplication and do not leak.

(The method `renderOnDisplayAsync` is not a Mono Framework method, it is just for demonstration.)

Avoid using the Global Context

If you write code that defines variables in the global context, you might encounter strange behaviours. Avoid code like this:

```
// I really need this timer in reach of all my code
mono::Timer importantTimer;

// some object I need available from everywhere
SomeClass myGlobalObject;

class AppController : public mono::IApplication
{
    // ...
};
```

If you use Mono classes inside `SomeClass` or reference `myGlobalTimer` from it, when you will likely run into problems! The reason is Mono's initialization scheme. A Mono application's start procedure is quite advanced, because many things must be setup and ready. Some hardware components depend on other components, and so on.

When you define global variables (that are classes) they are put into C++'s *global initializer lists*. This means they are defined *before* `monoWakeFromReset()` is executed. You can not expect peripherals to work before `monoWakeFromReset` has been called. When it is called, the system and all its features is ready. If you interact with Mono classes in code you execute before, it is not guaranteed to work properly.

If you would like to know more about the startup procedures of mono applications and how application code actually loads on the CPU, see the [Boot and Startup procedures in-depth](#) article.

Direct H/W Interrupts

If you are an experienced embedded developer, you know interrupts and what the requirements to their ISR's are. If you are thinking more like: "What is ISR's?" Or, "ISR's they relate to IRQ's right?" - then read on because you might make mistakes when using interrupts.

First, let's see some code that only noobs would write:

```
// H/W calls this function on pin state change, for example
void interruptServiceRoutine()
{
    flag = true;
    counter += 1;

    //debounce?
    wait_ms(200);
}
```


With the `wait_ms()` call, this interrupt handler (or ISR) will always take 200 ms to complete. Which is bad. A rule of thumb is that ISR's should be fast. You should avoid doing any real work inside them, least of all do busy waits.

Mono Framework is build on top of mbed, that provides classes for H/W Timer interrupts and input triggered interrupts. But because you should never do real work inside interrupt handlers, you normally just set a flag and then check that flag every run loop iteration.

We have includes classes that does all this for you. We call them *Queued Interrupts*, and we have an in-depth article about the topic: Queued callbacks and interrupts. There are the *QueuedInterrupt* class, that trigger a queued (run loop based) interrupt handler when an input pin changes state. And the *Timer* class, that provides a queued version of hardware timer interrupts.

Caution: We strongly encourage you to use the queued versions of timers and interrupts, since you avoid all the issues related to real H/W interrupts like: reentrancy, race-conditions, volatile variable, dead-locks and more.

Monokiosk

Using Monokiosk

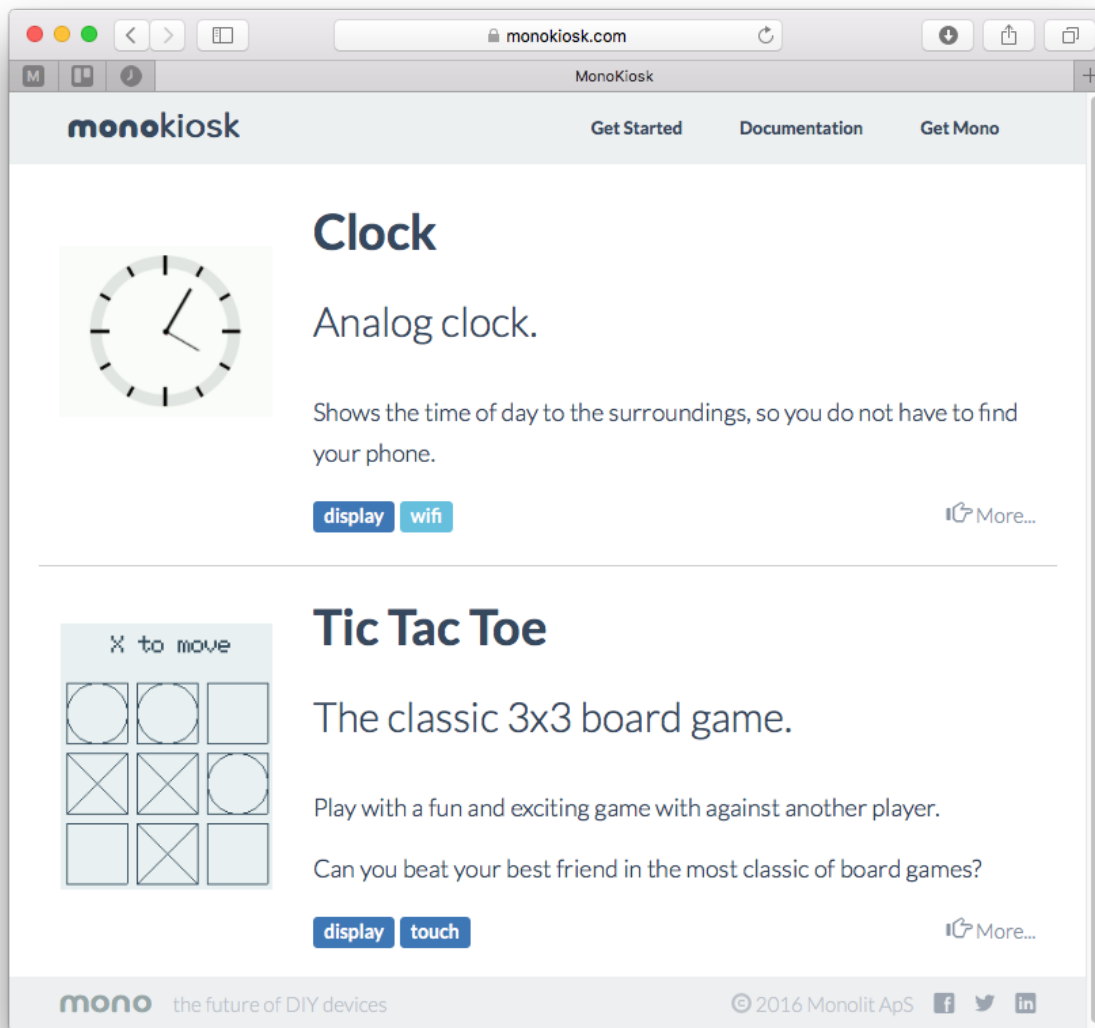
If you want to install an existing app from Monokiosk, on to your Mono device - this guide is for you!

In this guide we will show you how to download and install a pre-built application on Mono. Monokiosk is the *app store* where you can browse and download mono applications, built by makers from around the world.

Note: We plan to add more applications to the kiosk ourselves, and hope that our community will submit their own creations to the site.

Visit monokiosk.com

First order of business, direct your favorite browser to monokiosk.com, and you will see this page like this:



Currently there are a simple *Clock* app and a *Tic Tac Toe* app. But before you frantically click on one of these crazy mind blowing apps, you first need to install our *MonoMake* tool to install the application onto Mono.

MonoMake is an application that *programs* the application on Mono. Such a *programmer* transfers application files to mono, using the USB port. You need this tool to get the application from your computer to your Mono device. When you install applications from Monokiosk, the overall procedure is:

1. Connect Mono to your computer using an USB cable
2. Make sure Mono is *on*
3. Click the *Install* button in MonoKiosk, for the app you wish to install.

However, **first** you must download *MonoMake* itself, so click on the button that suits your system:

Choose the option that fits you or your OS. The downloads are installers that will install *monoprog* on your system.

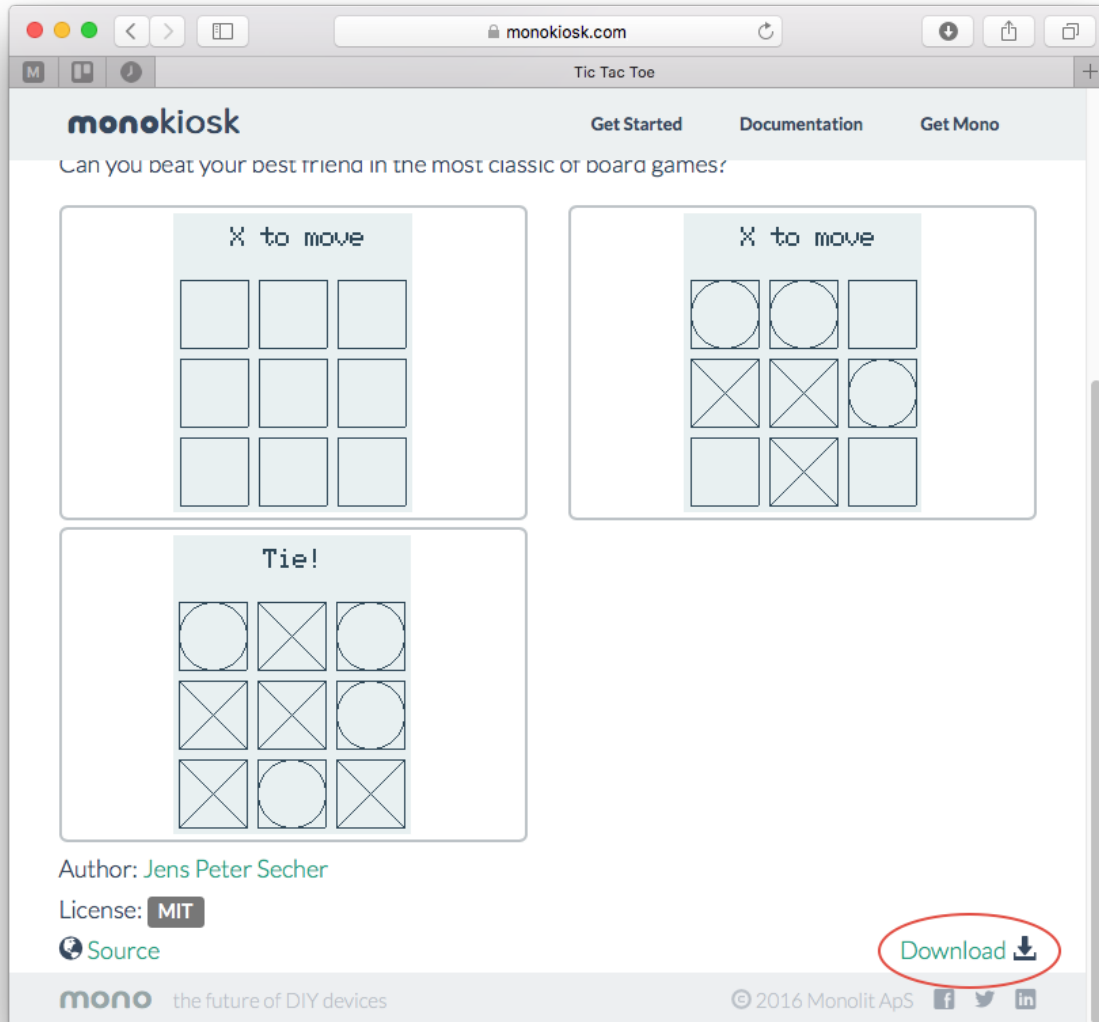
Note: This software you have installed is the complete toolset for developing OpenMono applications. However, you

don't need to use that. Here, we are just using the tool *MonoMake*.

Install Tic Tac Toe

After you have download and installed the OpenMono SDK on your system, let's continue.

Go to the frontpage and click on the *Tic Tac Toe* app. You will now see this page, where you will see the *install button* at the bottom right:

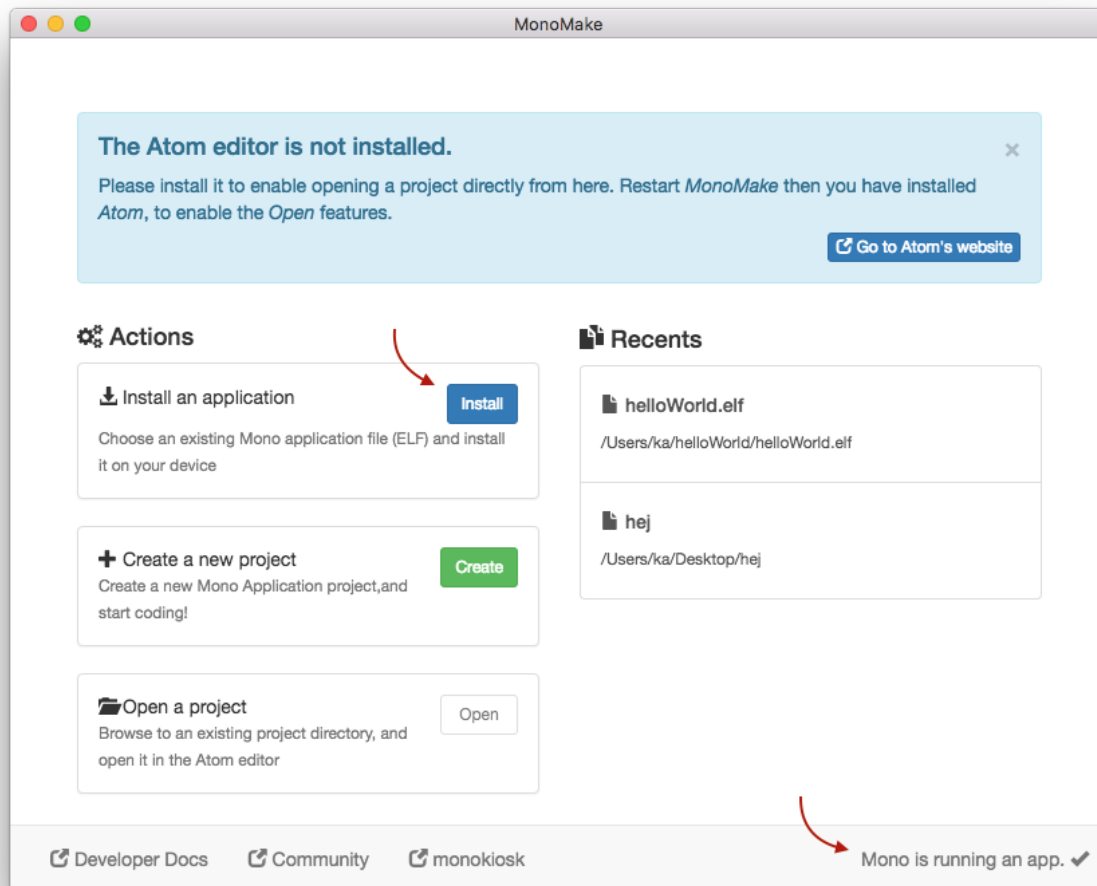


When you click the link, the *install* button, the *MonoMake* application will launch. *MonoMake* will automatically download the chosen application binary and install it on to Mono.

Now the application's binary code is loaded to Mono's internal flash memory. If everything goes well Mono will wake up and display the *Tic Tac Toe* app. Next, you can find a friend to play Tic Tac Toe with, you can install the other app or you could consider creating your own!

Manual installs

If, for some reason *MonoMake* does not launch when you click the *install* button - there are other options. Scroll down and click the *Download* button instead. This downloads the application binary file (*.elf*) to your computer. When the file is downloaded, launch *MonoMake* yourself.



Make sure that *Monomake* can see your connected (and *running*) Mono device. This is indicated at the lower right corner. Now, click the blue *Install* button left of the center. A standard *Open file* dialog box will open. Navigate to the just downloaded application *.elf* file, and open it.

MonoMake installs the application onto Mono.

Tutorials

Essentials

Resetting Mono

Like most Wifi routers and alike, Mono has a reset switch hidden inside a cavity.

If you have gotten stuck and need to force reboot Mono, this guide will help you in resetting Mono. If you have made a coding mistake that might have caused Mono to freeze - then we shall later look at how force Mono into bootloader mode.

Hardware Reset

If you just need to trigger a hardware reset, follow these steps:



1. Find a small tool like a small pen, steel wire or paper clip
2. Insert the tool into the *reset cavity*, as displayed in the picture above. *Be aware not to insert it into the buzzer opening.*
3. Push down gently to toggle the reset switch, and lift up the tool.

Mono will reset. It will load the bootloader that waits for 1 sec, before loading the application programmed in memory.

Force load Bootloader

If you need to install an app from Monokiosk or likewise, it might be nice to force Mono to stay in bootloader - and not load the programmed application. You can do this by pressing the User button, when releasing the reset switch. Then Mono will stay in Bootloader and not load any application. You will be able to upload a new app to it with monoprogram.

To force Mono to stay in bootloader:

1. Press and hold the User button
2. Gently press and release the reset switch
3. Release the User button

The *stay in bootloader* mode is only triggered by the pressed User button, then awaking from reset. There are no timeouts. To exit from bootloader, you must do an ordinary hardware reset.

Caution: Do not leave Mono in bootloader mode, since this will drain the battery. If you are in doubt, just do an extra normal reset.

Monoprog can detect the Bootloader

If you have connected Mono to your computer, you can use the Monoprog-tool to detect if Mono is in bootloader. Open a console / terminal and type:

```
$ monoprog -d
```

Monoprog will tell you if it could detect Mono. If it can, it is in bootloader!

Software Resets

You can programmatically trigger a reset from code! What happens is the CPU will reset itself if you explicitly tell it to do so. (That is, writing to a specific register.) In Mono Framework there are 3 functions you can use to trigger a reset:

- *Ordinary Reset*, where bootloader runs for 1 sec.
- *Reset To Application*, where bootloader is skipped.
- *Reset To Bootloader*, where Mono stays in bootloader.

The 3 functions are static (or class methods) on `IApplcationContext`, and can be used like this:

```
// Ordinary Reset
mono::IApplcationContext::SoftwareReset();

// Reset to Application
mono::IApplcationContext::SoftwareResetToApplication();

// Reset to Bootloader
mono::IApplcationContext::SoftwareResetToBootloader();
```

Note that these functions will never return, since they cause the CPU to reset. So any code beneath the reset functions, will get be reached, just take up memory!

Using Mono's Serial port

Let us examine how to use Mono's built in USB Serial port, and how to monitor it from your computer

By default when you plug in Mono's USB cable to a computer, Mono will appear as a USB CDC device. If you run Windows you have to install a driver, please goto [this section](#) to see how.

Get a Serial Terminal

First you need a serial terminal on your computer. Back in the old Windows XP days there was *Hyper Terminal*, but I guess it got retired at some point. So both Mac/Linux and Windows folks need to go fetch a serial terminal application from the internet.

Windows Serial apps:

Mac / Linux Serial apps

- [CoolTerm](#)
- [Minicom](#) (What we are using!)
- [ZTerm](#) (Mac only)
- [SerialTools](#) (Mac only)

We are very happy with *minicom*, since it has a feature that makes it really great with Mono. More about that later! Unfortunately *minicom* does not run on Windows, so we are considering making our own serial terminal client for Windows - that is similar to *minicom*.

If you use Linux / Mac, you should properly install *minicom* now. But you do not have to, you can also use any of the other choices.

Installing *minicom*

To install *minicom* on a Debian based Linux you should just use *aptitude* or *apt-get*:

```
$ sudo aptitude install minicom
```

On Mac you need the package manager called [Homebrew](#). If you don't have it, go get it from their homepage. When you are ready type:

```
$ brew install minicom
```

Sending data from Mono

Transferring text or binary data from Mono is really easy. In fact we have standard I/O from the standard C library available! To write some text to the serial port just do:

```
printf("Hello Serial port!!\t\n");
```

Notice that we ended the line with `\t\n` and not only `\n`. That is because the serial terminal standard is quite old, therefore many serial terminals expects both a *carriage return* and a *line feed* character.

To capture output from Mono in a terminal, we need to continuously output text. Therefore we need to call the print function periodically.

In *app_controller.h*:

```
class AppController : mono::IApplication
{
public:

    //Add a timer object to our appController
    mono::Timer timer;
```

```
// just a counter variable - for fun
int counter;

// class constructor
AppController();

// add this method to print to the serial port
void outputSomething();

// ...
```

Then in *app_controller.cpp*:

```
AppController::AppController()
{
    // in the constructor we setup the timer to fire periodically (every half second)
    timer.setInterval(500);

    // we tell it which function to call when it fires
    timer.setCallback<AppController>(this, &AppController::outputSomething);

    // set the counter to zero
    counter = 0;

    //start the timer
    timer.Start();
}

void AppController::outputSomething()
{
    printf("I can count to: %i",counter++);
}
```

Compile and upload the app to Mono.

Note: We are following best practice here. We could also have created a loop and a `wait_ms()` and `printf()` statement inside. But that would have broken serial port I/O!

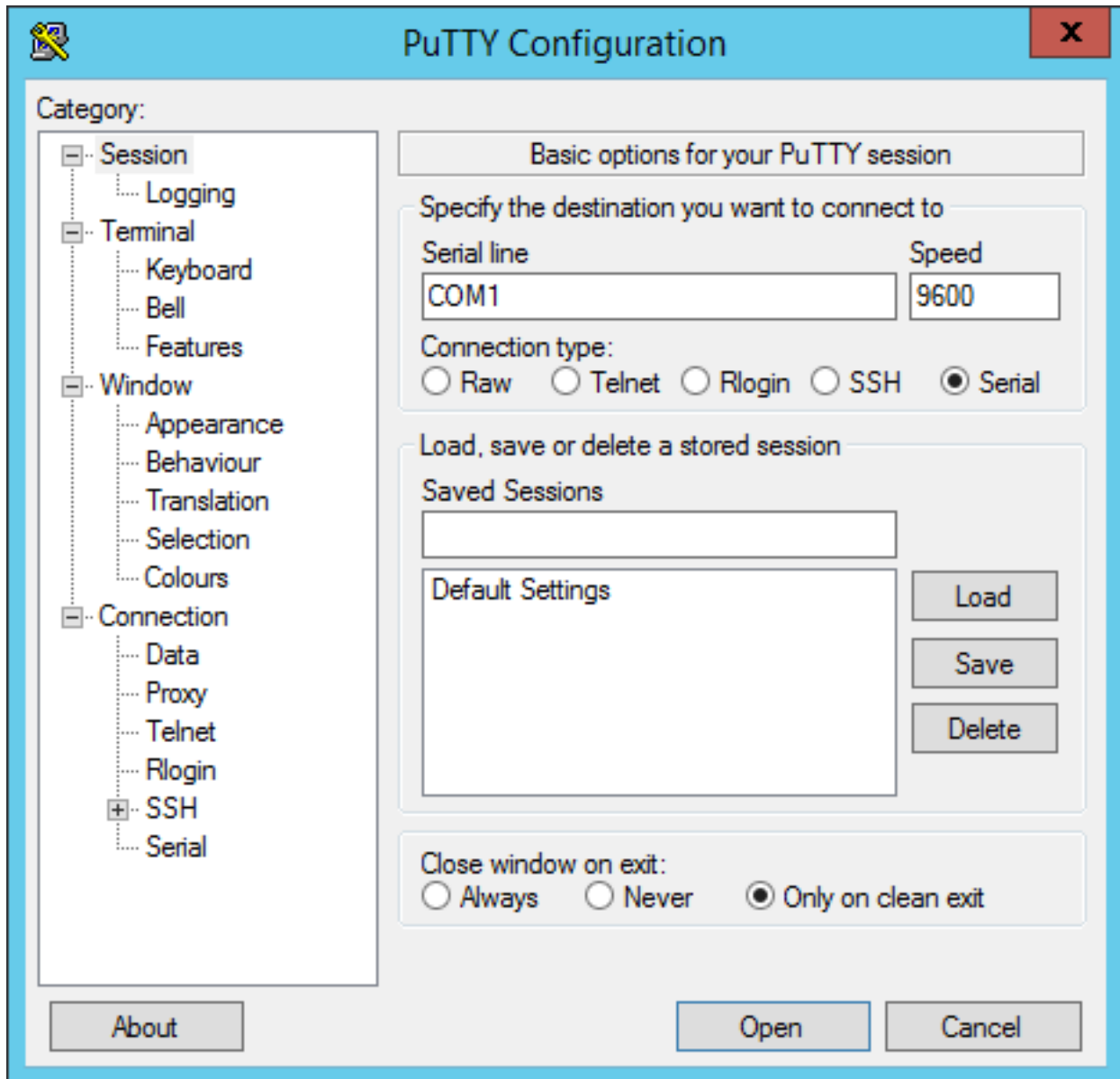
Connecting to Mono

When Mono is plugged in to the USB port, you should see a serial device on your computer. In Windows a *COM* port should be present in *Device manager*. On Linux / Mac there should exist a I/O device in the folder `/dev`. On Mac it would be named something like `/dev/cu.usbmodem1246`. On Linux the name could be `/dev/ttyACM0`.

If you use *minicom* you connect to the serial port with the `-D` flag, like this:

```
$ minicom -D /dev/cu.usbmodem1246
```

With PuTTY on Windows you should check the *COM* port number in *Device Manager* and type the this number in the *Serial line* text box:



Now you should be connected to the Mono serial output:

```
I can count to: 3
I can count to: 4
I can count to: 5
I can count to: 6
```

Because Mono does not wait for you to open the serial port, you might lose some output. That is why you properly will not see *I can count to 0* and *I can count to 1*. At some point we might change this behaviour and add a larger output buffer on Mono.

Note: You can also read from the serial port using the standard `getc stdio` function. Avoid using `scanf` since this will block until the formatted input has been read from the serial port.

Reconnects and Resets

You will soon discover that every time Mono resets, when uploading new apps or likewise, the serial port disappears. If you are not using *minicom*, you will have to manually connect to the port again. That is why we prefer to use *minicom*, because it automatically connects again when the port re-appears.

If you are not using *minicom* you will get pretty tired of reconnecting over and over again. At some point you might even consider helping us out with a .NET based serial tool to resolve the issue :-)

Why does the serial port disappear?

Unlike Arduino, Mono does not have a dedicated serial port to USB chip. We use the CPU's built-in USB component to create the serial port. This means that when the CPU is reset, the USB port is reset. That efficiently ends the serial connection. There is no way around this issue, expect using a dedicated USB chip.

Reset over the USB

Like Arduino we also use the Serial connection to trigger resets. Before a new application can be uploaded to Mono, we have to put it into bootloader. This is done by a reset, just like Arduino does. We use the serial port's DTR line to trigger a reset. Mono continuously monitors the DTR line, ready to trigger a software reset on itself.

If you do not follow the coding best practice convention and do something ugly, like this:

```
while(1)
{
    // I just wanna stay here for the rest of time ...
}
```

You have effectively cut off the possibility of resetting using the DTR, to trigger a software reset.

Serial Port Windows Driver

Windows do not support mapping USB CDC devices to Serial ports (COM devices) out of the box. It needs an `.inf` file to tell it to do so. You can download an installer for this [INF file here](#), but it should have been installed automatically. The driver is included in both the [Monokiosk based installer](#) and the [SDK toolchain](#).

Sleep Mode

In this tutorial we will quickly demonstrate how to put Mono into sleep mode.

Mono has no physical on/off switch, so you cannot cut the power from the system. This means you *must always* provide a way for Mono to goto to sleep. Sleep mode is the closest we come to being powered off. Mono's power consumption in sleep mode is around 50 μ A (micro amperes), which is really close to no consumption at all.

Default behaviour

Because it is crucial to be able to turn off Mono (goto sleep mode), we provide this functionality by default. When you create a new project with:

```
$ monomake project MyNewProject
```

The SDK predefines the behaviour of the *User button*, to toggle sleep mode. This is important because controlling on/off functionality, is not what is first on your mind when developing new mono apps. So you don't have to consider it too much, unless to wish to use the *User button* for something else.

Sleep and USB

In our *v1.1* release of our SDK, we enabled sleep mode while connected to the USB. This means that triggering sleep will power-down Mono's USB port. Therefore our computer will loose connection to Mono if it goes to sleep.

When you wake up Mono, it will be enumerated once again.

Sleep and external power

Mono has the ability to provide power for components mounted either on the *Mono Shield Adaptor* or attached through the *3.5mm jack connector*. By default Mono provides 3.3V on this external power rail (called VAUX). To safe battery life, the VAUX power is turned off in sleep mode. This is the default behaviour, but you can change it if you need to.

In the *3.5mm jack connector* the power on J_TIP in sleep mode depends on the USB state. If USB is connected the voltage on J_TIP is 1.7V, because of leakage currents from VBUS. With no USB attached, the J_TIP voltage is 0.0V in sleep mode.

Warning: We have introduced this behaviour in *Release 1.4*. Before this version the VAUX line was not limited to 3.3V! Especially in sleep mode, the voltage rises to the battery's current voltage level. In *Release 1.4* we fixed this issue, by turning off auxillary power in sleep mode.

You can consult the [schematics](#) to see how the power system is configured.

Triggering sleep mode

Say we have an application that utilizes the *User button* to do something else, than toggling sleep mode. Now we need another way of going to sleep, so lets create a button on the touch screen to toggle sleep. First in our *app_controller.h* we add the *ButtonView* object to the class:

```
class AppController : public mono::IApplication {

    // we add our button view here
    mono::ui::ButtonView sleepBtn;

    // ... rest of appcontroller.h ...
```

In the implementation file (*app_controller.cpp*) we initialize the button, setting its position and dimensions on the screen. We also define the text label on the button:

```
AppController::AppController() :

    //first we call the button constructor with a Rect object and a string
    sleepBtn(Rect(10,175,150,40), "Enter Sleep"),

    // here comes the project template code...
    // Call the TextLabel's constructor, with a Rect and a static text
    helloLabel(Rect(0,100,176,20), "Hi, I'm Mono!")
{
```

Now, we need to tie a function to the button's click handler. That means when the button is clicked, it automatically triggers a function to be called. We can call the standard static function for going to sleep, that is defined in the global `IAApplicationContext` object. The function is:

```
mono::IAApplicationContext::EnterSleepMode();
```

Normally, we could just add this function directly to the button's callback handler, but in this particular case it is not possible! The callback handler always expects a member function, not a static class function like `EnterSleepMode`. So we need to define a member function on our `AppController` and wrap the call inside that.

Inside `app_controller.h` add:

```
public:

    // The default constructor
    AppController();

    // we add our sleep method here:
    void gotoSleep();

    // Called automatically by Mono on device reset
    void monoWakeFromReset();
```

Then in the implementation file (`app_controller.cpp`), we define the body of the function to:

```
void AppController::gotoSleep()
{
    mono::IAApplicationContext::EnterSleepMode();
}
```

Lastly, we tell the `sleepBtn` object to call our function, when it gets clicked - we do this from `AppController`'s constructor:

```
// set another text color
helloLabel.setTextColor(display::TurquoiseColor);

// tell the button to call our gotoSleep function
sleepBtn.setClickCallback<AppController>(this, &AppController::gotoSleep);

// tell the button to show itself on the screen
sleepBtn.show();
```

Okay, go compile and install the app on Mono - and you should see this on the screen:



Try to press the button and you will see Mono goto sleep and turning off the display. In this example you wake Mono again just by pressing the *User button*.

Danger: Be aware that if you overwrite the *User Button* functionality, you are responsible for ensuring that Mono has a wake up source. A wake source is always a physical input pin interrupt. In most cases you should use the User button.

In another tutorial we shall see how you overwrite the *User button* functionality.

Complete sample code

For reference, here is the complete sample code of the tutorial:

app_controller.h:

```
#ifndef app_controller_h
#define app_controller_h

// Include the Mono Framework
#include <mono.h>

// Import the mono and mono::ui namespaces into the context
// to avoid writing long type names, like mono::ui::TextLabel
using namespace mono;
using namespace mono::ui;

// The App main controller object.
// This template app will show a "hello" text in the screen
class ApplicationController : public mono::IApplication {

    // we add our button view here
    mono::ui::ButtonView sleepBtn;

    // This is the text label object that will displayed
    TextLabelView helloLabel;
```

```
public:

    // The default constructor
    AppController();

    // we add our sleep method here:
    void gotoSleep();

    // Called automatically by Mono on device reset
    void monoWakeFromReset();

    // Called automatically by Mono just before it enters sleep mode
    void monoWillGotoSleep();

    // Called automatically by Mono right after after it wakes from sleep
    void monoWakeFromSleep();

};

#endif /* app_controller_h */
```

app_controller.cpp:

```
#include "app_controller.h"

using namespace mono::geo;

// Contructor
// initializes the label object with position and text content
// You should init data here, since I/O is not setup yet.
AppController::AppController() :

    //first we call the button constructor with a Rect object and a string
    sleepBtn(Rect(10,175,150,40), "Enter Sleep"),

    // Call the TextLabel's constructor, with a Rect and a static text
    helloLabel(Rect(0,100,176,20), "Hi, I'm Mono!")
{

    // the label is the full width of screen, set it to be center aligned
    helloLabel.setAlignment(TextLabelView::ALIGN_CENTER);

    // set another text color
    helloLabel.setTextColor(display::TurquoiseColor);

    // tell the button to call our gotoSleep function
    sleepBtn.setClickCallback<AppController>(&this, &AppController::gotoSleep);

    // tell the button to show itself on the screen
    sleepBtn.show();
}

void AppController::gotoSleep()
{
    mono::IApplicationContext::EnterSleepMode();
}
```

```
void AppController::monoWakeFromReset()
{
    // At this point after reset we can safely expect all peripherals and
    // I/O to be setup & ready.

    // tell the label to show itself on the screen
    helloLabel.show();
}

void AppController::monoWillGotoSleep()
{
    // Do any clean up here, before system goes to sleep and power
    // off peripherals.
}

void AppController::monoWakeFromSleep()
{
    // Due to a software bug in the wake-up routines, we need to reset here!
    // If not, Mono will go into an infinite loop!
    mono::IApplicationContext::SoftwareResetToApplication();
    // We never reach this point in the code, CPU has reset!

    // (Normally) after sleep, the screen memory has been cleared - tell the label to
    // draw itself again
    helloLabel.scheduleRepaint();
}
```

Coding

Using Wifi

Let us walk through the steps required to connect Mono to a Wifi access point, and download the content of a web page

The Goal

We shall create a small mono application that connects to a Wifi access point and downloads a website. To achieve this, we need to accomplish the following steps:

1. Initialize the Wifi module
2. Connect to an access point, using hardcoded credentials
3. Create a HTTP Get request to a URL and display the response

Setting up the project

First order of business is to create a new mono application project. I assume you already have installed the *OpenMono SDK*.

Open a terminal (or command prompt) and fire up this command:

```
$ monomake project --bare wifi_tutorial
```

`monomake` will now create an application project template for us. Open the two source files (*app_controller.h* and *app_controller.cpp*) in your favorite code editor. In the header file (*.h*) we need to include, to import the wireless module definitions:

```
#include <mono.h>
#include <io/wifi.h>

using namespace mono;
using namespace mono::ui;
```

Also, in the header file we need to add member variables for the module to the *AppController* class definition.

Note: The class *HttpClient* is a quick’n’dirty implementation, and is likely to be phased out to future releases of Mono Framework.

Therefore we extend the existing *AppController* with the class members:

```
class AppController : public mono::IApplication {

    // The wifi hardware class
    io::Wifi wifi;

    // The http client
    network::HttpClient client;

    // a console view to display html data
    ConsoleView<176, 220> console;

public:

    AppController();

    // ...
};
```

Now, we have imported the objects we are going to need, the next step is to initialize them properly.

Initializing the Wifi and connecting

We need to supply our Wifi’s *SSID* (access point’s name) and passphrase to the `Wifi` object. These are passed in the *constructor*:

```
// You should init data here, since I/O is not setup yet.
AppController::AppController() :
    wifi("MY_SSID", "MY_PASSPHRASE")
{
    // ...
}
```

Next we want to connect to our access point, using the credential we just provided. So far the hardware `Wifi` module is not connected or initialized. This happens when we call the method `connect` on `Wifi`. We cannot do that from

the constructor, but only from the method `monoWakeFromReset()`:

```
void AppController::monoWakeFromReset()
{
    //show our console view
    console.show();

    //connect the wifi module
    wifi.connect();

    //tell the world what we are doing
    console.WriteLine("Connecting...");
}
```

The module will try to connect to the given access point, and expect to get a DHCP configured IP address.

The constructor on `Wifi` actually had a third optional parameter that can define the security setting. The default value is WPA/WPA2 Personal. Other supported options are: *No security*, *WEP* and *Enterprise WPA/WPA2*.

Caution: Almost all calls to the `Wifi` module are *asynchronous*. This means they add commands to a queue. The function call returns immediately and the commands will be processed by the applications run-loop. So when the method returns, the network is not connected and ready yet.

Because the connect process is running in the background, we would like to be notified when the network is actually ready. Therefore, we need to setup a callback method. To do that we add a new method to our `AppController` class. We add the method definition in the header file:

```
class AppController : public mono::IApplication
{
    // ...

public:
    void networkReadyHandler();

    // ...
}
```

Next, we add the method body in the implementation (`.cpp`) file:

```
void AppController::networkReadyHandler()
{
    console.WriteLine("Network ready");
}
```

Now, we need to tell the `wifi` object to call our method, when the network is connected. We append this line to `monoWakeFromReset()`:

```
wifi.setConnectedCallback(this, &AppController::networkReadyHandler);
```

This sets up the callback function, such that the module will call the `networkReadyHandler()` method, on our `AppController` instance.

If you feel for it, tryout the code we have written so far. If you monitor the serial port, you should see the `Wifi` module emitting debug information. Hopefully you should see the *Network Ready* text in screen after ~20 secs. If not, consult the serial terminal for any clue to what went wrong.

Caution: We have not set any error handler callback function. You should always do that, because failing to connect to an access point is a pretty common scenario. See the API reference for Wifi.

Download a website

Now that we have connected to an access point with DHCP, I take the freedom to assume that your Mono now has internet access! So let's go ahead and download: this webpage!

To download a website means doing a HTTP GET request from a HTTP client, and here our `HttpClient` class member from earlier, comes into action.

Like the process of connecting to an access point was asynchronous, (happening in the background), the process of downloading websites is asynchronous. That means we are going to need another callback function, so let's define another method on *AppController.h*:

```
// ...

public:

    void networkReadyHandler();

    void httpHandleData(const network::HttpClient::HttpResponseData &data);

// ...
```

Notice the ampersand (&) symbol that defines the `data` parameter as a reference.

Note: If you want to know more about references in C++, I recommend our article: [The C programmers guide to C++](#).

In the implementation file (*app_controller.cpp*) we add the function body:

```
void AppController::httpHandleData(const network::HttpClient::HttpResponseData &
↪data)
{
    console.WriteLine(data.bodyChunk);

    if (data.Finished)
    {
        console.WriteLine("All Downloaded");
    }
}
```

`HttpClient` will return the HTML content in multiple calls, and you use the `Finished` member to see when all data has arrived. Here we just append the HTML chunk to the console, so it is not too pretty to look at. When the response has been downloaded, we append the text *All Downloaded*.

Now, we are ready to setup the http client and fetch the webpage. We can use `HttpClient` *only after* the network is ready. So in the implementation file, add this to `networkReadyHandler()`:

```
void AppController::networkReadyHandler()
{
    console.WriteLine("Network ready");
}
```

```

        //fetch a webpage
        client = mono::network::HttpClient("http://developer.openmono.com/en/latest/
↪");

        //now the client will be fetching the web page
        // let setup the data callback
        client.setDataReadyCallback(this, &AppController::httpHandleData);
    }

```

Go ahead and build the app. Upload to Mono and see the HTML content scroll across the display.

Using file I/O

In this tutorial we will see how to read and write files on an SD card.

Mono has a Micro SD Card slot and currently supports communicating with an SD card using an [SPI](#) interface. We have included both SD card I/O and FAT32 file system I/O in our framework. As soon as you have setup the SD card I/O you can use the familiar `stdio.h` file I/O.

Get an SD Card

First order of business is for you to obtain a MicroSD Card and format it in good ol' FAT32. Then insert the card into Mono's slot and fire up a new project:

```

$ monomake project fileIO --bare
Creating new bare mono project: fileIO...
* fileIO/app_controller.h
* fileIO/app_controller.cpp
Writing Makefile: fileIO/Makefile...
Atom Project Settings: Writing Auto complete includes...

```

Note: Notice that we use the switch `--bare` when we created the project. This option strips all comments from the template code. This way you have a less verbose starting point.

Now `cd` into the `fileIO` directory and open the code files in your favourite code editor.

Initializing the SD Card

Open `app_controller.h` and add these lines:

```

#include <mono.h>

// import the SD card and FS definitions
#include <io/file_system.h>
#include <stdio.h>

class AppController : public mono::IApplication {
public:

    mono::io::FileSystem fs; // create an instance of the FS I/O

```

```
AppController();
```

Here we include the definitions for the both the SD card I/O and the file I/O. Next, we need to construct the `fs` object in `AppController`'s constructor. Go to `app_controller.cpp`:

```
AppController::AppController() :  
    fs("sd")  
{  
}
```

Here we initialize the file system and provide the library for communicating with the SD Card. The parameter `"sd"` is the mount point. This means the SD Card is mounted at `/sd`.

Writing to a file

Let us write a file in the SD card. We use the standard C library functions `fopen` and `fwrite`, that you might know - if you ever coded in C.

So, to write some data to a file we insert the following code in the `monoWakeFromReset` method:

```
void AppController::monoWakeFromReset()  
{  
    FILE *file = fopen("/sd/new_file.txt", "w");  
  
    if (file == 0) {  
        printf("Could not open write file!\r\n");  
        return;  
    }  
    else {  
        const char *str = "Hello file system!\nRemember coding in C?";  
        int written = fwrite(str, 1, strlen(str), file);  
        printf("Wrote %d bytes\r\n", written);  
        fclose(file);  
    }  
}
```

Here we open/create a file on the SD card called `new_file.txt`. The `fopen` function returns a file descriptor (`FILE*`) that is 0 if an error occurs.

If `file` is not 0 we write some text to it and finally close (`fclose`) the file to flush the written data to the disk. You should always close files when you are done writing to them. If you don't, you risk losing your written data.

Reading from a file

So we just written to a file, now let us read what we just wrote. Append the following to the `monoWakeFromReset` method:

```
FILE *rFile = fopen("/sd/new_file.txt", "r");  
if (rFile == 0) {  
    printf("Could not open read file!\r\n");  
    return;  
}  
  
char buffer[100];
```

```
memset(buffer, 0, 100);
int read = fread(buffer, 1, 100, rFile);
printf("Read %d bytes from file\r\n", read);
printf("%s\r\n", buffer);
fclose(rFile);
```

Here we first open the file we previously written. Then, we create a byte buffer to hold the data we read from the file. Because the initial content of `buffer` is nondeterministic, we zero its contents with the `memset` call.

Note: We do this because `printf` needs a *string terminator*. A string terminator is a 0 character. Upon accounting a 0 `printf` will know that the end of the string has been reached.

The `fread` function reads the data from the file. It reads the first 100 bytes or until `EOF` is reached. Then we just print the contents of `buffer`, which is the content of the file.

Standard C Library

As mentioned earlier, you have access to the file I/O of `stdlib`. This means you can use the familiar `stdlib` file I/O API's.

These include:

- `fprintf`
- `fscanf`
- `fseek`
- `ftell`
- `fflush`
- `fgetc`
- `fputc`
- etc.

When you for example read or write from the serial port (using `printf`), you in fact just use the `stdout` and `stdin` global pointers. (`stderr` just maps to `stdout`.)

See the API for the `stdlib` file I/O here: www.cplusplus.com/reference/cstdio

Custom Views

In this tutorial we shall create our own custom view, that utilizes the drawing commands to display content on the screen.

Our software framework contains a selection of common *View* classes. These include `TextLabels`, `Buttons` and things like progress and state indicators. But you also have the ability to create your own views, to display custom content. Before we begin it is useful to know a little bit about views. Also, you should see the *Display System Architecture* article.

About Views

We have stolen the concept of a *View* from almost all other existing GUI frameworks. A *view* is a rectangular frame where you can draw primitives inside. All views define a *width* and *height*, along with a *x,y* position. These properties position all views on Mono's display - with respect to the display coordinates.

As you can see on the figure, the display coordinates have a *origo* at the top left corner of the screen. The positive *Y* direction is downward. In contrast to modern GUI frameworks Mono *views* cannot be nested and does not define their own internal coordinate system. All coordinates given to the drawing command are in display coordinates. It is your hassle to correct for the view's *x,y* offsets.

Views can be Invisible and Dirty

All views has a visibility state. This state is used by the display system to know if it can render views to the screen. Only visible views are painted. By convention all views must be created in the *invisible* state. Further, views can also be *dirty*. This means that the view has changes that need to be rendered on the screen. Only dirty views are rendered. You can mark a view as dirty by calling the method: `scheduleRepaint()`.

Painting views

When you create your own views, you must subclass the `View` class and you are required to overwrite 1 method: `repaint()`. This is the method that paints the view on the screen. The method is automatically called by the display system - you should never call it manually!

All views share a common `DisplayPainter` object that can draw primitives. You should draw primitives only from inside the `repaint()` method. If you draw primitives from outside the method, you might see artifacts on the screen.

The Goal

In this tutorial I will show you how to create a custom view that displays two crossing lines and a circle. To accomplish this we use the `DisplayPainter` routines `drawLine()` and `drawCircle()`, along with other routines to make our view robust. This means it does not make any assumptions about the screen state.

We want the cross and circle to fill the views entire *view rectangle*, so we use the view dimensions as parameters for the drawing functions.

Creating the project

First go ahead and create a new project:

```
$ monomake project customView --bare
```

Note: We use the `--bare` option to create an empty project without any example code.

It is good practice to create our custom view in separate files. Create two new files:

- `custom_view.h`
- `custom_view.cpp`

In the header file we define our new custom view class:

```
class CustomView : public mono::ui::View {
public:

    CustomView(const mono::geo::Rect &rect);

    void repaint();
};
```

We overwrite only the constructor and the repaint function. The default constructor for *views* takes a *Rect* that defines the views position and dimensions.

In the implementation file (*custom_view.cpp*) we add the implementation of the two methods:

```
// Constructor
CustomView::CustomView(const mono::geo::Rect &rect) : mono::ui::View(rect)
{
}

// and the repaint function
void CustomView::repaint()
{
    // enable anti-aliased line painting (slower but prettier)
    this->painter.useAntialiasedDrawing();

    // setup the view background color to pitch black
    painter.setBackgroundColor(mono::display::BlackColor);

    //draw the black background: a filled rect
    painter.drawFillRect(this->ViewRect(), true);

    // set the foreground color to a deep blue color
    painter.setForegroundColor(mono::display::MidnightBlueColor);

    // draw an outline rect around the view rectangle
    painter.drawRect(this->ViewRect());

    //set a new foreground color to red
    painter.setForegroundColor(mono::display::RedColor);

    //draw the first line in the cross
    painter.drawLine(this->ViewRect().UpperLeft(), this->ViewRect().LowerRight());

    //draw the second line in the cross
    painter.drawLine(this->ViewRect().LowerLeft(), this->ViewRect().UpperRight());

    //now we will draw the circle, with a radius that is the either width
    // or height - which ever is smallest.
    int radius;
    if (this->ViewRect().Width() < this->ViewRect().Height())
        radius = this->ViewRect().Width()/2;
    else
        radius = this->ViewRect().Height()/2;

    // create a circle object with center inside the views rectangle
    mono::geo::Circle c(this->ViewRect().Center(), radius - 1);
```

```
//set the foreground color
painter.setForegroundColor(mono::display::WhiteColor);

//draw the circle
painter.drawCircle(c);

// disable anti-aliasing to make drawing fast for any other view
painter.useAntialiasedDrawing(false);
}
```

Now, this code snippet is a mouthful. Let me break it down to pieces:

The constructor

Our constructor simply forwards the provided `Rect` object to the parent (`View`) constructor. The parent constructor will take care of initializing the our view's properties. In our implementation we simply call the parent constructor and leave the method's body empty.

The repainting

This is here we actually paint the view. As mentioned earlier, all views share a global `DisplayPainter` object. This object holds a set of properties like paint brush colors and anti-aliasing settings. Therefore, to be sure about what colors you are painting, you should always set the colors in your `repaint()` method.

We start by enabling anti-aliased drawing of lines. This slows down the painting process a bit, but the lines looks much smoother. Next, we set the *painter's* background color to black. With the black color set, we draw a filled rectangle clearing the entire area of the view. This is important because there might be some old graphics present on the screen.

To highlight the views boundary rectangle, we draw an outlined rectangle with the dimension of the view's own rectangle. You can say we give the view a border.

Next, we begin to draw the crossing lines - one at the time. The `drawLine` routine takes the begin and end points of the line. We use a set of convenience methods on the `Rect` class to get the positions of the *view rectangle's* corners.

The `Circle` class defines circles by a center and a radius. We can get the *view rectangle's* center using another convenient method on the `Rect` object. But we need to do a little bit of calculations to get the radius. We use the smallest of the width and height, to keep the circle within the view boundaries. (And subtract 1 to not overwrite the border.)

Lastly, we disable the anti-aliased drawing. To leave the `DisplayPainter` as we found it.

The Result

Now, we must add our *CustomView* to our application. This means we must use it from the *AppController*. Therefore, include the header file in *custom_view.h*, and add our *CustomView* as a member variable:

```
#include "custom_view.h"

class AppController : public mono::IApplication {
public:

    CustomView cv;
```



```
AppController();
//...
```

Finally, we need to initialize our view correctly in *AppController*'s constructor and set the views visibility state by calling the `show()` method.

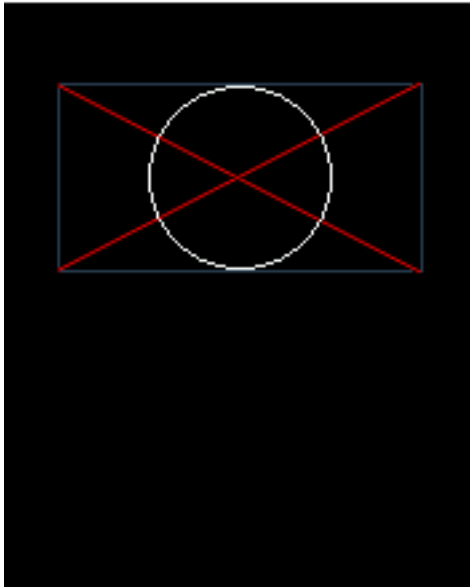
In *app_controller.cpp* add:

```
AppController::AppController() :
    cv(mono::geo::Rect(20,30,136,70))
{
}

void AppController::monoWakeFromReset()
{
    cv.show();
}
```

Notice that we set the views position and dimensions in the constructor, by defining a `Rect` `((20, 30, 136, 70))`.

Do a `make install` and our custom should look like this on your Mono:



Because we use the views own dimensions to draw the primitives, the view will paint correctly for all dimensions and positions.

Responding to Touch

In this tutorial we shall see how you can easily respond to touch input. First we just accept any touch input, and later we see how you combine touch and display drawing.

Mono's touch system is implemented as a responder chain. That means touch input are events traveling down a chain of *responders*, until one decides to act on the event. A *responder* is simply an object that can handle touch events. Responders are not required to act upon a touch event, they can ignore them if they wish. All responders must subclass from the `TouchResponder` class.

Note: In this tutorial I assume you are somewhat familiar with C++. If you are not, you might wonder more about

syntax than about what is really happening. I would suggest you read our article [The C programmers guide to C++ first](#).

A simple Touch Response

A simple touch responder could be an object handling display dimming. To save battery you might want the screen to dim, after some time of inactivity. Then, if you touch the display the screen should light up again. In this case the touch position is irrelevant.

Let us create a class we call `AutoDimmer` in a file called *auto_dimmer.h*:

```
class AutoDimmer : public mono::TouchResponder {
public:

    mono::Timer dimTimer;

    AutoDimmer();

    void respondTouchBegin(mono::TouchEvent &event);

    void dim();
};
```

Our class inherits from `TouchResponder` and it overwrites the method `respondTouchBegin`. This method gets called on every touch event, regardless on the touch inputs position. To dim the screen after some time, we use a `Timer` object, that calls the `dim()` method when it times out. Lastly, we create a constructor to setup the timer and insert our class into the touch responder chain.

Create an implementation file for our class, called *auto_dimmer.cpp*:

```
AutoDimmer::AutoDimmer() : mono::TouchResponder()
{
    //dim after 3 secs
    dimTimer.setInterval(3000);

    // setup timer callback
    dimTimer.setCallback<AutoDimmer>(this, &AutoDimmer::dim);

    //start dim timer
    dimTimer.start();
}

void AutoDimmer::respondTouchBegin(mono::TouchEvent &event)
{
    //stop timer
    dimTimer.stop();

    //undim screen
    mono::display::IDisplayController *ctrl = mono::IApplicationContext::Instance->
    ↳DisplayController;
    ctrl->setBrightness(255);

    //restart dim timer
    dimTimer.start();

    //make sure event handling continues
```

```

    event.handled = false;
}

void AutoDimmer::dim()
{
    dimTimer.stop();

    //dim screen
    mono::display::IDisplayController *ctrl = mono::IApplicationContext::Instance->
↳DisplayController;
    ctrl->setBrightness(50);
}

```

The implementation file has 3 methods:

Constructor

We call the parent class' (TouchResponder) constructor, that will insert our class' instance into the touch system's responder chain.

In the constructor body we setup the timer that will dim the screen. We give it a timeout interval of 3 seconds and set its callback method to the `dim()` method. Last, we start the timer so the display will dim within 3 seconds.

Respond to Touch begin

The `RespondTouchBegin` method is an overloaded method from the parent class. When touch events arrive in the responder chain, this method will automatically be called. So within this method we know that a touch has occurred, so we undim the screen and restart the timer. To undim the screen we obtain a pointer to the `DisplayController` from the *application context*. The `DisplayController` has methods to change the brightness of the screen.

We restart the timer so it will dim the screen within 3 seconds again, if no other touch events occurs.

Dim the screen

Our last method simply stops the timer and dims the screen. It also uses the *application context* to obtain a pointer to the `DisplayController`. We set the dimmed display brightness to 50 out of 255.

Run the example

To use `AutoDimmer` we must add it as a member variable on an *AppController*. Create a mono project and add our *auto_dimmer.** files to its project directory. Now add `AutoDimmer` as a member variable on `AppController`:

```

#include "auto_dimmer.h"

class AppController : public mono::IApplication {
public:

    AutoDimmer dimmer;

    AppController();
    // ...

```

Thats it. Go ahead and `make install` the project. You should see that the display brightness changes after 3 secs. And if you touch the display, the brightness should change back to the maximum level.

Caveats

There is a small caveat in our implementation. Our `AutoDimmer` object might not be the first responder in the chain. Touch events might get handled by an earlier responder in the chain. This can mean we might miss touch events.

To fix this we need to make sure our `AutoDimmer` is the *first responder*, meaning it sits first in the responder chain. Responders are added to the chain as first come, first serve. The simplest approach is to make sure `AutoDimmer` is the first responder to have its constructor called.

Views that respond to touch

Now that we know how to handle touch input, let us next up the game a bit. Let us take an existing `View` class, that does not respond to touch, and make it respond to touches.

Let us take the `ProgressBarView` and make it respond to touch input. We want to set its progress position to the *x* coordinate of the touch input. This effectively creates a crude *slider*.

The *ResponderView*

We are going to use the class `ResponderView`. This is a subclass of the class `TouchResponder`, that we used previously. `ResponderView` is a `View` that can handle touch. Unlike `TouchResponder` the `ResponderView` only responds to touches that occur inside its rectangular boundary.

Hint: All `Views` have a *view rect*, the rectangular area they occupy on the screen. This is the active touch area for `ResponderViews`, all touches outside the *view rect* are ignored, and passed on in the responder chain.

The `ResponderView` requires you to overload 3 methods, that differ from the ones from `TouchResponder`. These are:

- `void touchBegin(TouchEvent &)`
- `void touchMove(TouchEvent &)`
- `void touchEnd(TouchEvent &)`

These methods are triggered only if the touch are inside the *view rect*. Further, the `TouchEvent` reference they provide are converted to display coordinates for you. This means these are in pixels, and not raw touch values.

If you wish or need to, you can overload the original responder methods from `TouchResponder`. These will be triggered on all touches.

The simple slider

To realize our simple slider UI component, we create our own class: `Slider`, that will inherit from both `ProgressBarView` and `ResponderView`.

Caution: In C++ a class can inherit from multiple parent classes. In this case we inherit from two classes that themselves inherit from the same parent. This causes ambiguity cases, known as *the diamond problem*. However, for the sake of this tutorial we use this multi-inheritance approach anyway.

Create a set of new files called: *slider.h* and *slider.cpp* and add them to a new application project.

Open the *slider.h* file at paste this:

```
#include <mono.h>

class Slider : // inheritance
    public mono::ui::ProgressBarView,
    public mono::ui::ResponderView
{
public:

    Slider(const mono::geo::Rect &rct);

    // ResponderView overloads
    void touchBegin(mono::TouchEvent &event);
    void touchEnd(mono::TouchEvent &event);
    void touchMove(mono::TouchEvent &event);

    // Ambiguous View overloads
    void show();
    void hide();
    void repaint();
};
```

Our Slider class inherits the drawing from *ProgressBarView* and the touch listening from *ResponderView*. Both parents define the common View methods: *repaint()*, *show()* and *hide()*. Therefore, to avoid any ambiguity we overload these explicitly.

Next, insert this into the *slider.cpp* implementation file:

```
#include "Slider.h"

Slider::Slider(const Rect &rct) :
    mono::ui::ProgressBarView(rct),
    mono::ui::ResponderView(rct)
{
    this->setMaximum(this->ProgressBarView::viewRect.Width());
    this->setMinimum(0);
}

void Slider::touchBegin(TouchEvent &event)
{
    int relative = event.Position.X() - this->ProgressBarView::viewRect.X();

    this->setValue(relative);
    this->ProgressBarView::scheduleRepaint();
}

void Slider::touchEnd(TouchEvent &) { }

void Slider::touchMove(TouchEvent &event)
{
    int relative = event.Position.X() - this->ProgressBarView::viewRect.X();

    this->setValue(relative);
    this->ProgressBarView::scheduleRepaint();
}
```

```
void Slider::show()
{
    this->ProgressBarView::show();
    this->ResponderView::show();
}
void Slider::hide()
{
    this->ProgressBarView::hide();
    this->ResponderView::hide();
}
void Slider::repaint()
{
    this->mono::ui::ProgressBarView::repaint();
}
```

For clarity I will go through the code in sections.

Constructor

```
Slider::Slider(const Rect &rct) :
    mono::ui::ProgressBarView(rct),
    mono::ui::ResponderView(rct)
{
    this->setMaximum(this->ProgressBarView::viewRect.Width());
    this->setMinimum(0);
}
```

First the constructor, where we call both parent constructors with the *view rectangle* provided. In the body we set the maximum and minimum values for the progress bar.

We set the maximum value to equal the width of the view. Thereby we avoid any need for a conversion from touch position into a progress percentage.

Touch begin & touch move

```
int relative = event.Position.X() - this->ProgressBarView::viewRect.X();

this->setValue(relative);
this->ProgressBarView::scheduleRepaint();
```

Here we calculate the *X* position of the touch, with respect to the upper left corner of the *view rect*. This is done by subtracting the *X* offset of the *view rect*. Then we simply set the progressbar value to the relative position.

Finally we explicitly call `scheduleRepaint()` on the *ProgressBarView* parent class.

The content of the `TouchMove` method is identical.

Show, hide & repaint

These 3 methods are overloaded to explicitly call the matching method on both parent classes. In both `show` and `hide` we call both parents, to ensure we are listening for touch and will be painted on the screen.

The `repaint` method is an exception. Here we need only to call the `ProgressBar`'s `repaint` method, because the other parent does not do any painting.

Result

To connect the pieces we need to add our new slider to the `app_controller.h`:

```
#include "slider.h"

class AppController : public mono::IApplication {
public:

    Slider slider;

    ...
}
```

Also, we need to call `show()` on our slider object. In `app_controller.cpp` insert this:

```
AppController::AppController() :
    slider(mono::geo::Rect(10, 100, 156, 35))
{
    slider.show();
}
```

Now go ahead and make `install`, and you should be able to change the progress bars position by touch input.

Graph your heart beat

In this tutorial we shall go through the steps of writing a very crude heart beat sensor. You will need some sensing hardware, either the one from pulsesensor.com or you can roll your own, if you are a hardware wiz.

The final app will render the input signal from the sensor as a graph on Mono's display. The heart beat will appear as spikes on the graph.

In this video example our hardware engineer Lasse has build his own version of the sensor and software, based on the schematics from pulsesensor.com

We will focus on the software and assume you have the sensor from pulsesensor.com. We will be sampling the ADC at fixed intervals and update the display. Further, if you wish to detect the frequency of the spikes, I will link to a nice article about the topic.

For starters, let us talk about the hardware setup.

Wiring

Regardless of using pulsesensor.com or your own custom, there are a minimum of 3 connections needed:

Signal	Color	3.5mm Jack	Mono Pin
Power (+)	Red	Tip	<i>J_TIP</i>
Gnd (-)	Black	Sleeve (Base)	
Analog output	Purple	Ring 1 (Outer most)	<i>J_RING1</i>

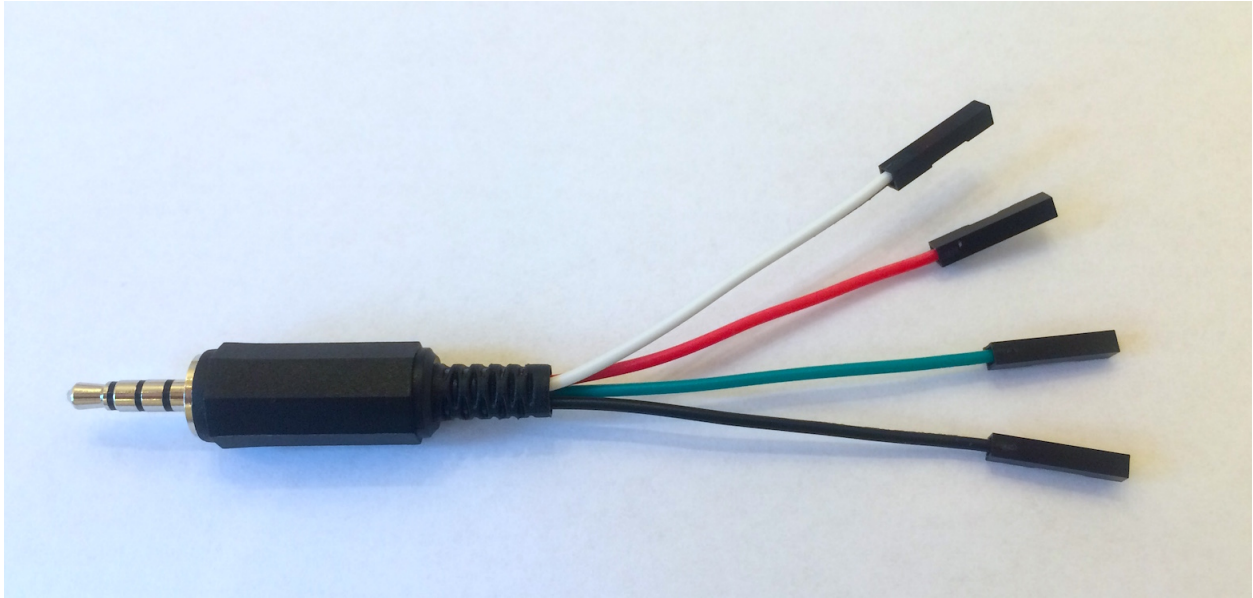
The colors are in respect to pulsesensor.com's pin connections

The *analog output* is the sensors signal that we need to feed to the Mono's ADC. The 5V power source we can get from the USB connection. As long as we keep the USB connected, we do not need to use Mono's internal boost converter.

Step 1: Jack connector

You need to have a 3 or 4 pole 3.5mm jack connector. If you have one in your stash, use that. If not, you need to go acquire one from the web or your favourite electronics dealer.

Then, solder the 3 wires from the pulse sensor to the connector, as layed out in the table above. (If you use a 4 pole jack, leave *Ring 2* unconnected.)



For ease I have created a jack connector like this, that is essentially a breakout board for a 3.5mm jack connector. You can do the same, if you do not wish to solder directly on your *pulsesensor*.

Software

When the hardware in place, we are ready to code! We need to do a few steps in software to achieve our goal:

1. Enable voltage supply output in the jack tip pin
2. Sample the ADC at fixed intervals
3. Lowpass filter the ADC output
4. Draw a graph of the filtered signal.

Step 2: New project

Note: I have taken the liberty to assume that you already have installed Mono SDK. If you have not, please follow the getting started tutorial, and return here again thereafter.

First, we must create a new *mono application project*. Fire up a terminal (or *Command prompt* on Windows) and:


```
$ monomake project --bare pulsesensor
Creating new bare mono project: pulsesensor...
* pulsesensor/app_controller.h
* pulsesensor/app_controller.cpp
Writing Makefile: pulsesensor/Makefile...
Atom Project Settings: Writing Auto complete includes...
```

Notice the `--bare` option. This creates an empty project without example code. Step into the newly created directory:

```
$ cd pulsesensor
```

Now, you should spin up your favorite editor or IDE. Since the *monomake* tool automatically adds a code completion settings file for Atom's *autocomplete-clang* plugin, I will use *Atom* in the tutorial:

```
$ atom .
```

This command opens the project in Atom.

Step 3: Drawing a Graph

Now that we have an empty project, we need a C++ class to draw the graph on the display. In Mono, all software classes that draw to the screen must inherit from the *View* class: `mono::ui::View`.

Create a new file called `graph.h`. This file should implement our `GraphView` class, that will draw the graph on the display. Then, paste this into the file:

```
#include <mono.h>

class GraphView : public mono::ui::View
{
public:
    int values[176];
    int dataCursor;
    int dispCursor;

    GraphView(uint16_t yOffset, uint16_t height) :
        View(mono::geo::Rect(0,yOffset,176,height))
    {
        dispCursor = 0;
        dataCursor = 0;
    }

    void append(int value)
    {
        static const int maxX = 0x8000;
        static const int minX = 0x4000;

        if (value < minX)
            value = minX;
        else if (value > maxX+minX)
            value = maxX+minX;

        int scaled = (value * viewRect.Height() / maxX);
        int graphOffset = minX * viewRect.Height() / maxX;
        values[dataCursor++] = scaled - graphOffset;
```

```
        if (dataCursor >= 176)
            dataCursor = 0;
    }

    void repaint()
    {
        painter.setForegroundColor(mono::display::RedColor);
        painter.setBackgroundColor(StandardBackgroundColor);

        if (dispCursor < dataCursor)
        {
            drawDiff(dispCursor, dataCursor);
            dispCursor = dataCursor;
        }
        else
        {
            drawDiff(dispCursor, viewRect.Width());
            dispCursor = 0;
        }

        painter.setForegroundColor(mono::display::GreenColor);
        painter.drawVLine(dispCursor, viewRect.Y(), viewRect.Y2());
    }

    void drawDiff(int begin, int end)
    {
        for(int x = begin; x<end; x += 2)
        {
            painter.drawFillRect(x, viewRect.Y(), 2, viewRect.Height(), true);
            painter.drawFillRect(x, viewRect.Y()+values[x], 2, 2);
        }
    }
};
```

Let us go through the code, step by step.

Declaration

```
class GraphView : public mono::ui::View
{
public:
    int values[176];
    int dataCursor;
    int dispCursor;
```

First we declare the class `GraphView` and define it inherits from `View`. We define 3 member variables, an array of sample values and cursors that points to the latest sample. Both the latest added sample to the values array, and the latest values painted on the display.

Constructor

```

GraphView(uint16_t yOffset, uint16_t height) :
    View(mono::geo::Rect(0,yOffset,176,height))
{
    dispCursor = 0;
    dataCursor = 0;
}

```

The class' constructor takes 2 arguments, an *Y* offset and a height. The graph will always take up the full width of the screen, which is 176 pixels. The *Y* offset defines where the view is positioned with respect to top of the screen.

The body of the constructor sets the 2 cursor variables to zero. This is important in an embedded environment, since you cannot know the initial value of a variable.

Append method

This method adds a new value to the graph and increments the cursor. First the new value is cropped to the *max* and *min* boundaries defined by the static variables.

```

void append(int value)
{
    static const int maxX = 0x8000;
    static const int minX = 0x4000;

    if (value < minX)
        value = minX;
    else if (value > maxX+minX)
        value = maxX+minX;
}

```

Next, we scale the value to fit the height of the graph view itself. Before adding the new value to the sample array, we subtract a static offset.

```

int scaled = (value * viewRect.Height() / maxX);
int graphOffset = minX * viewRect.Height() / maxX;

```

Attention: In Mono's *display coordinate system* the *Y* axis is flipped. This means positive *Y* direction is downward.

The static offset (*graphOffset*) ensures that that center of the graph is the center of the value boundaries. The input span is trimmed from 0 to 0xFFFF, to 0x4000 to 0x8000.

Hint: I have chosen these numbers to ease the CPU's integer division. When dividing by exponents of 2, the compiler can substitute the division with a shift operation.

The scaled value is saved in the values array, and the *dataCursor* is incremented. If the cursor exceeds the width of the array, it is reset to 0.

Repaint

The `repaint()` method is part of the *View* interface, and is automatically called by the display system. Inside this method we implement the actual drawing of the graph.

All *Views* share a `painter` object that can draw shapes on the screen. First, we set the painter's color palette, a foreground and a background.

```
void repaint()
{
    painter.setForegroundColor(mono::display::RedColor);
    painter.setBackgroundColor(StandardBackgroundColor);
}
```

Next, if the data cursor is larger than the display cursor, we call the `drawDiff` method for the difference. Then the `dispCursor` is set equal to the data cursor.

```
if (dispCursor < dataCursor)
{
    drawDiff(dispCursor, dataCursor);
    dispCursor = dataCursor;
}
```

The `else` branch of the *if statement* is taken when the `dataCursor` has looped or reset. In this case we draw the data from display cursor position up to the end of the array.

```
else
{
    drawDiff(dispCursor, viewRect.Width());
    dispCursor = 0;
}
```

The last task is to draw a green sweeping cursor, that indicate the end of the buffer. We set the foreground color palette to green, and draw a vertical line at the position of the display cursor.

```
painter.setForegroundColor(mono::display::GreenColor);
painter.drawLine(dispCursor, viewRect.Y(), viewRect.Y2());
```

drawDiff

This is where the graph drawing occurs. The method receives two parameters: the beginning and the end of the array values to draw.

We loop through the values and at each value we paint a 2 pixel wide vertical line, using the background color. Then, we draw a filled 2x2 pixels rectangle, at the `x` value position.

```
for(int x = begin; x<end; x += 2)
{
    painter.fillRect(x, viewRect.Y(), 2, viewRect.Height(), true);
    painter.fillRect(x, viewRect.Y()+values[x], 2, 2);
}
```

This is the complete graph class, that can draw an array of values as a graph on the screen.

Note: You might wonder where the *.cpp* implementation file for this class is. You can in fact implement function bodies in header files. However, this is far from the best practice. My only excuse to do it here, is that I am lazy and I

try to keep this tutorial as short as possible.

Step 4: App Controller

Now we will turn attention to the *app_controller.h* and *.cpp* files. This object will control power and ADC sampling, lowpass filter and utilize our graph view.

Right now the class is just a stub of empty methods. Start by going to the *app_controller.h* and we will add some member variables. Edit (or copy-paste) this into your copy, to make it look like this:

```
#ifndef app_controller_h
#define app_controller_h

#include <mono.h>
#include <mbed.h>
#include "graph.h"

class AppController : public mono::IApplication {
public:

    mbed::AnalogIn adc;           // Analog input
    mbed::Ticker adcTicker;       // interrupt based timer
    GraphView graph;             // Our graph view

    uint16_t filter[8];          // Low pass filter memory
    int filterPos;                // Filters oldest sample pointer

    AppController();              // default constructor

    void sampleAdc();             // sample ADC & update display

    void monoWakeFromReset();
    void monoWillGotoSleep();
    void monoWakeFromSleep();
};

#endif /* app_controller_h */
```

Basically we have added member variables to handle the analog pin input, a timer (called a *Ticker* in *mbed*) to sample at regular intervals and a moving average lowpass filter.

We also added the method `sampleAdc()` that should be called by the *Ticker* when the analog input must be sampled.

AppController implementation

Now, go to the *app_controller.cpp* file and overwrite its contents with this:

```
#include "app_controller.h"

AppController::AppController() :
    adc(J_RING2),
    graph(50, 220-50)
{
    adcTicker.attach_us(this, &AppController::sampleAdc, 10000);
}
```

```
graph.show();

filterPos = 0;
memset(filter, 0, 8);
}

void AppController::sampleAdc()
{
    filter[filterPos++] = adc.read_u16();

    if (filterPos >= 8)
        filterPos = 0;

    int sum = 0;
    for (int i=0;i<8;i++)
        sum += filter[i];

    graph.append( sum / 8 );

    graph.scheduleRepaint();
}

void AppController::monoWakeFromReset()
{
    // enable J_TIP power
    mbed::DigitalOut jpi(JPI_nEN, 1);
    mbed::DigitalOut jpo(JPO_nEN, 0);
}

void AppController::monoWillGotoSleep()
{
    mbed::DigitalOut jpi(JPI_nEN, 1);
    mbed::DigitalOut jpo(JPO_nEN, 1);

    adcTicker.detach();
}

void AppController::monoWakeFromSleep()
{
    mbed::DigitalOut jpi(JPI_nEN, 1);
    mbed::DigitalOut jpo(JPO_nEN, 0);

    adcTicker.attach_us(this, &AppController::sampleAdc, 10000);

    graph.scheduleRepaint();
}
```

Again, I will chop up the code and explain the functions and their job bit by bit.

Constructor

When our `AppController` class is constructed by the system, it needs to setup two of its member objects: `mbed::AnalogIn` and our own `GraphView`. These have designated constructors that we need to call explicitly, and we do that in the constructor right before the actual function body:

```
AppController::AppController() :
    adc(J_RING1),
    graph(50, 220-50)
{
    ...
}
```

Here we define we want the ADC to sample the J_RING1 pin. The GraphView is set to occupy the lower 3/4 of the screen. This is defined by the *Y* offset of 50 pixels and the height of 170 pixels.

```
adcTicker.attach_us(this, &AppController::sampleAdc, 10000);

graph.show();

filterPos = 0;
memset(filter, 0, 8);
```

In the constructor body we first install the callback function for the `Ticker`. It will call `sampleAdc` every 10 ms. This gives us a samplerate of 100 Hz.

We tell our graph to be visible, meaning that the display system will bother to paint it.

Lastly, the running average filter is initialized. We zero all values and set the pointer to 0.

Sampling the input

The `sampleAdc` method is called by the `Ticker` from inside the MCU timer interrupt. This means that this method will run inside a interrupt context, requiring it to be fast.

```
filter[filterPos++] = adc.read_u16();

if (filterPos >= 8)
    filterPos = 0;
```

The input pin is sampled, and the result is returned as an unsigned 16-bit integer. This sample value is stored in the `filter` array and the *filter position* is incremented.

When the position reaches the end of the array, it is reset.

```
int sum = 0;
for (int i=0; i<8; i++)
    sum += filter[i];

graph.append( sum / 8 );
```

Before inserting the new value into the graph, we lowpass filter it, by calculating the average value of the entire filter array. This average is then appended to the graph.

Note: This filtering should remove any 50 Hz or 60 Hz noise. If it does not, you might need to tweak it a little bit. (Changing the array length.)

```
graph.scheduleRepaint();
```

The final line tells the graph that it must repaint itself, at the next *vsync* event. Note that we do not repaint the graph view from here, it is only scheduled for repaint at a later point in time.

Power event methods

The next 3 methods are the standard power event methods, handling reset, sleep and wake routines. Here we need to setup the power and shut it down again upon sleep.

```
void AppController::monoWakeFromReset()
{
    // enable J_TIP power
    mbed::DigitalOut jpi(JPI_nEN, 1);
    mbed::DigitalOut jpo(JPO_nEN, 0);
}
```

We use mbed's I/O interfaces to set the values of the power control switches on Mono. Specifically we set the JPI_nEN and the JPO_nEN pins, to enable power output on the jack connector's *TIP* pin. By asserting JPO_nEN we connect the VAUX power line to J_TIP.

Just to make our intension clear, we also de-assert JPI_nEN, to explicitly state that there is no short circuit from VAUX to VBUS. (Refer to the [schematics](#) for Mono.)

```
void AppController::monoWillGotoSleep()
{
    mbed::DigitalOut jpi(JPI_nEN, 1);
    mbed::DigitalOut jpo(JPO_nEN, 1);

    adcTicker.detach();
}
```

In the go-to-sleep routine we cut the power to J_TIP and disable the Ticker. If we do not disable the ticker, we risk having the timer fire when the RTC system runs in sleep mode.

```
void AppController::monoWakeFromSleep()
{
    mbed::DigitalOut jpi(JPI_nEN, 1);
    mbed::DigitalOut jpo(JPO_nEN, 0);

    adcTicker.attach_us(this, &AppController::sampleAdc, 10000);

    graph.scheduleRepaint();
}
```

Upon wake-up we re-enable power output on J_TIP and activate the Ticker again. Because the display has been powered off, we also need to repaint its content.

Step 5: Result

Now, is the moment of truth where you should compile the app. Go to the terminal and run:

```
$ make
```

Hopefully it compiles and you should see something like this:

```
rm -f pulsesensor.elf mono_project.map
rm -f -r build
creating build directory
Compiling C++: app_controller.cpp
Compiling C++: Default main function
Linking pulsesensor.elf
```


Install the app on your Mono by running:

```
$ make install
```

If everything goes as we expect, you should see the signal stabilizing around the middle of the graph view. This is the DC offset removal filter in the pulsesensor that is settling.

Place your finger or earlobe on the sensor, sit still and look for the spikes, after the DC filter has settled again.

This is what you should see on Mono's display. Depending on the sensor environment, spikes might be more or less visible.

Further reading

If you wish to add a BPM counter, you need to detect the spikes in software. To do that I will recommend the very detailed articles on pulsesensor.com:

Triggering IFTTT Webhooks

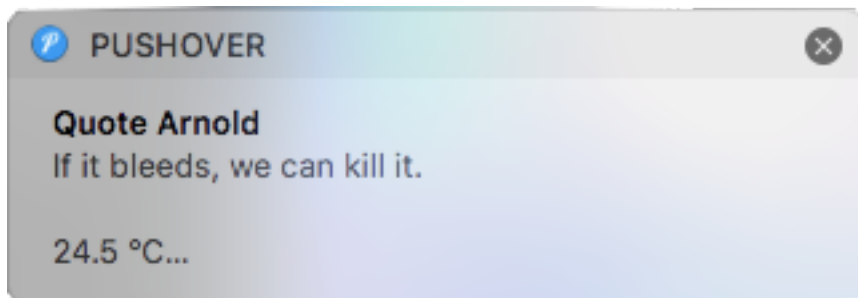
In this example we shall see how to use Mono to trigger a webhook on IFTTT. To demonstrate this, we create an application *Quote Arnold* using Arduino SDK. This app will send random Schwarzenegger movie quotes to IFTTT.

Who should read this?

First I assume you are aware of IFTTT (If This Then That), and know how to setup *Applets* on their platform. You should also be familiar with Arduino IDE, and have installed the [OpenMono board package](#) using the *Board Manager*. It is (though not required), preferred if you know a good share of classic Arnold Schwarzenegger movies.

Quote Arnold

Our goal is to create a Mono application that sends randomized Arnold Schwarzenegger movie quotes to IFTTT. On IFTTT we can forward them to the IFTTT app using push notification. Or you can choose to do something else with them. In my IFTTT applet I have chosen to forward the message to [Pushover](#), so I can receive desktop push notifications:

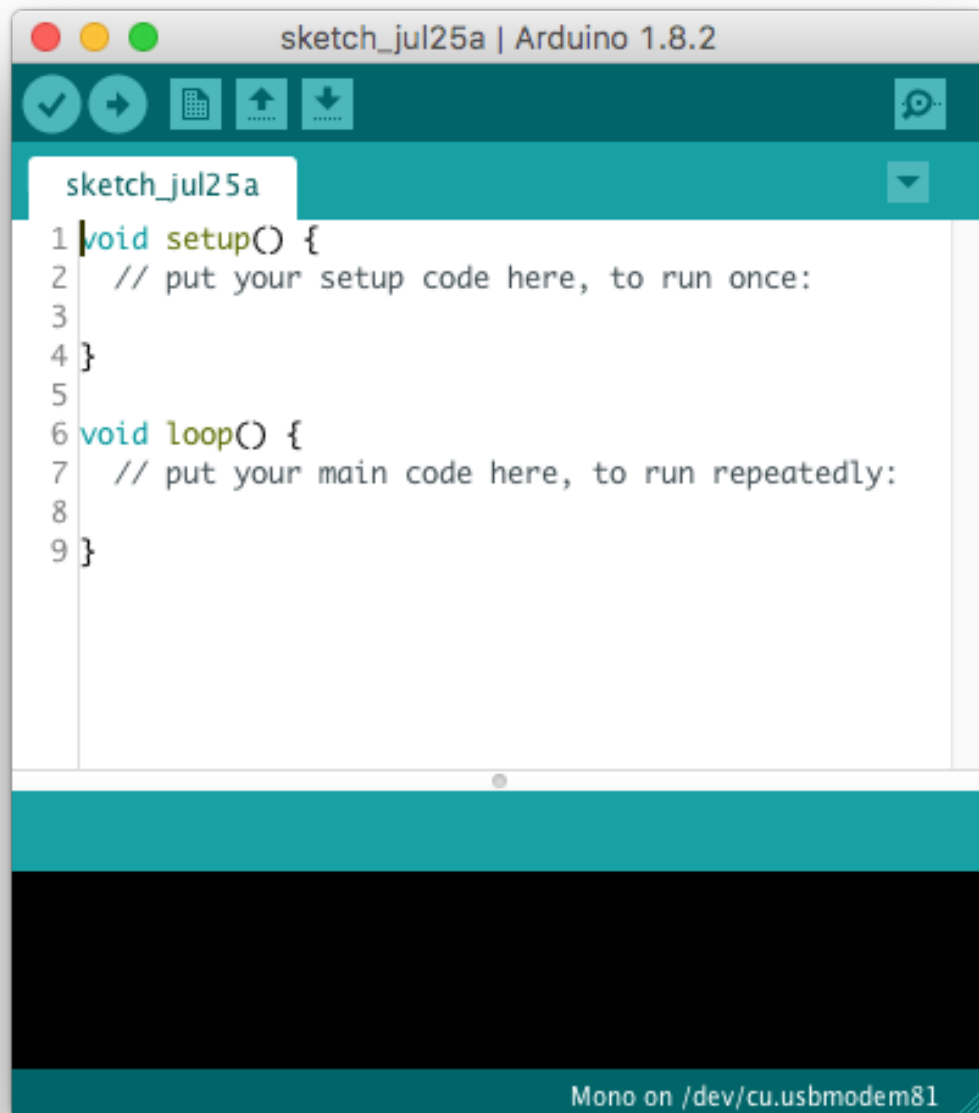


Oh - and by the way, just because we can, we will also send the current temperature and battery level, along with the quote.

Setup

We will use Arduino IDE to implement our application. Of course we could also use a standard OpenMono SDK project, but I have chosen to demonstrate the use of Arduino IDE with OpenMono here.

To begin with, go ahead and open up Arduino IDE and create a new sketch.



Because Arduino only defines two C functions (`setup` & `loop`), all our resources must be declared in the global context. This means to must declare all *ButtonViews*, *HttpClient*s and alike in the global context - that is outside the functions.

Also, because our application is driven by user input (UI button pushes), we will not use the `loop` function at all!

Adding the push button

For starters we need to add the button that will trigger a quote being sent via a *Http request* to IFTTT.

First, we include *OpenMono* classes and declare two pointers to *ButtonView* and *TextLabelView* instances:

```
#include <mono.h>           // include mono library

using namespace mono::ui;   // Add mono namespace
using namespace mono::geo;

ButtonView *button;        // create a ButtonView pointer
TextLabelView *statusLbl;  // a textlabel pointer
```

Inside the `setup` function will create (*construct*) the *ButtonView* and *TextLabelView* objects and position them on the screen:

```
void setup() {
    // Button
    button = new ButtonView(Rect(20, 80, 176 - 40, 65), "Quote Arnold");
    button->setClickCallback(&handleButton);
    button->show();

    // Text status label
    statusLbl = new TextLabelView(Rect(10,180,156,35), "Not connected");
    statusLbl->setAlignment(TextLabelView::ALIGN_CENTER);
    statusLbl->show();
}
```

We use the new syntax since we allocate the objects on the *stack*. This was why we created `button` and `statusLbl` as a pointer types.

Note: The C++ new operator uses *standard lib*'s `malloc` function. This means the actual objects are first created inside the `setup()` function. This is needed, because we need to the system to be ready when the object are constructed. We cannot create them in global space.

We set the button click handler to a function called `handleButton`, which we will define in a moment. Lastly we tell the button to `show()` itself.

We also set the text alignment and content of the *TextLabelView*, before showing it.

Let's add a function for `handleButton` and then try out our code:

```
void handleButton()
{
    // we will add content later
}
```

Go ahead and compile and *upload* the code, and you should see a button and a text label on Mono's display. On my Mono it looks like this:



Starting Wifi

Before we can send any data to IFTTT, we need to connect to Wifi. Luckily since *SDK 1.7.3* we have a *Wifi* class, that handles wifi setup and initialization. Let's add that to our sketch, also as a pointer that we initialize in `setup()`:

```
// button and text is declared here also
mono::io::Wifi *wifi;

void setup() {
    // Initialization of button and text left out here

    wifi = new mono::io::Wifi(YOUR_SSID, YOUR_PASSPHRASE);
    wifi->setConnectedCallback(&networkReady);
    wifi->setConnectErrorCallback(&networkError);
}
```

Notice that our *Wifi* object takes two callback functions, for handling the *wifi connected* and *connect error* events. Let's declare these two functions and let them change the `statusLbl` to reflect the new state:

```
void networkReady()
{
    printf("network ready!\r\n");
    statusLbl->setText("Connected");
}

void networkError()
{
    statusLbl->setText("Connect Error");
}
```

Now, we need to call `connect` on *Wifi* when the button is pushed, therefore we add this line inside the

handleButton function:

```
void handleButton ()
{
    wifi->connect ();
    statusLbl->setText ("Connecting...");
}
```

Now, compile and upload the app. Push the button and if everything is good, you should see “*Connected*” appear. If you encounter errors, a good help is to open Arduino’s *Serial Monitor*. The Wifi system is quite verbose on the serial port.

Pushing to IFTTT

When we have network access, we are ready to do a *HTTP Post* call to IFTTT. On IFTTT you need to setup a custom *Applet* using their *new applet* site.

The applet’s *if this* part must be a *Webhook*. This is the component that receives the data from Mono. You can use whatever service you like as the *then that* part. I used the *Pushover* service, to forward the quote as a *Push Notification*. This means I will get a push notification whenever the Webhook is called.

To get the URL of the webhook, is tricky. Open [this page](#) and click on *Documentation*. The URL should be on the form:

```
https://maker.ifttt.com/trigger/{YOUR_EVENT_NAME}/with/key/{YOUR_API_KEY}
```

When we insert this URL into our mono app, we must remove the *s* in *https*. Mono does not yet support HTTPS out-of-the-box.

Note: Lucky for us IFTTT exposes their API on plain old *http*. This means our service is unsecure, however it serves our example. Mono’s Wifi hardware supports using *https*, however you need to use the low level Redpine API to access this feature in SDK 1.7.3.

The Http Post client class

Now, let us add a global `HttpPostClient` object next to our existing global pointers. This time though, we will not be using a pointer - but a real object:

```
// previous declarations left out
mono::io::Wifi *wifi;
mono::network::HttpPostClient client;
float temp = 0.0;
int lastBatteryLevel = 100;
```

I have also added global variables for caching the current temperature and battery level. Their usage will become clear in a moment.

We can begin to use the object as soon as the network is ready. That means we must use it only after the `networkReady()` function has fired. Therefore we create a new function called `beginPostRequest()`:

```
void beginPostRequest ()
{
    if (!wifi->isConnected()) // bail if network not ready
    {
```

```

    statusLbl->setText("not yet ready!");
    return;
}

client = mono::network::HttpPostClient(
    "http://maker.ifttt.com/trigger/YOUR_EVENT_NAME/with/key/YOUR_API_KEY",
    "Content-Type: application/json\r\n");
client.setBodyDataCallback(&httpData);
client.setBodyLengthCallback(&httpLength);

temp = mono::IApplicationContext::Instance->Temperature->ReadMilliCelcius() / 1000.
↪0;
lastBatteryLevel = mono::power::MonoBattery::ReadPercentage();

client.post();
}

```

This function checks if the network is ready and bails if it is not. Then we create a new `HttpPostClient` object and provide the URL it should call. Notice that we also provide an optional second argument, that is an extra *Http header*. This header defines the *content type* of the *POST* body data.

For the actual body data and the *Content-Length* header we provide 2 callback functions: `httpLength()` and `httpBody()`. The first will return the actual byte length of the body payload data. The second will write the data into a buffer, provided by the system.

Before we send off the request (`post()`) we cache the current temperature and battery level. We want the temperature as a floating point so we can extract its decimal values later.

Now, let's implement the callback functions:

```

uint16_t httpLength()
{
    // get the fraction part of floating point
    int fracPart = 10*(temp - (int)temp);
    return snprintf(0,0, "{ \"value1\": \"%s\", \"value2\": %i.%i, \"value3\": %i}",
↪"Arnold quote here", (int)temp, fracPart, lastBatteryLevel);
}

void httpData(char *data)
{
    // get the fraction part of floating point
    int fracPart = 10*(temp - (int)temp);
    snprintf(data,httpLength()+1, "{ \"value1\": \"%s\", \"value2\": %i.%i, \"value3\":
↪%i}", "Arnold quote here", (int)temp, fracPart, lastBatteryLevel);
}

```

Caution: In Mono's embedded environment we do not have access to `printf`'s floating point conversion. This conversion takes up so much memory, that it is left out by default. This means the format specifier `%f` does not work. Therefore, we must extract the floating point decimals manually.

These two functions do basically the same thing. However, the first (`httpLength()`) returns the payload data length. We use *stdio*'s `snprintf` to calculate the byte-length of the concatenated formatted string.

The system allocates the buffer needed to hold the payload data, and appends room for a string terminator character. This means `httpData()` can write directly into this buffer.

Caution: Please be aware that `sprintf` arguments are a bit quirky. The *max length* argument, that specifies the total size of the data buffer, must include the string terminator. However the returned length of the total string, behaves like `strlen` - not including the terminator.

Let us not forget to call `beginPostRequest()`. There are two scenarios where we can trigger the call: when the network is *not ready* or when *it is ready*.

If the network is not ready, we should just call the HTTP call from inside the `networkReady()` function:

```
void networkReady()
{
    printf("network ready!\n");
    statusLbl->setText("Connected");

    beginPostRequest();
}
```

The other scenario happens if we press the button multiple times. If network is already available, then we should just call `beginPostRequest()` right away, directly from the button handler function:

```
void handleButton()
{
    if (wifi->isConnected())
    {
        beginPostRequest();
        return;
    }

    wifi->connect();
    statusLbl->setText("Connecting...");
}
```

That's it! Try it out. Remember to observe the *Serial Monitor* to see if there are any error messages.

Tip: We have not created a callback function that handles the HTTP response from IFTTT. Therefore you will not see any error messages. If you wish to print the HTTP response, then you should setup the *dataReady* callback. (See the tutorial *Using Wifi*.)

Random quotes from Arnold

Our basic functionality is done, however we don't quote Arnold just yet. Let us now implement a list of great movie quotes and select between them randomly. We must also seed our random generator, such that we don't end up with a deterministic quote sequence.

First we declare a global array of strings, this is our list of quotes:

```
const char *quotes[] = {
    "I'll be back.",
    "You are one ugly motherfucker.",
    "Hasta la vista, baby.",
    "Remember, Sully, when I promised to kill you last?",

    "Fuck you, asshole.",
    "Consider this a divorce!",
```

```
"Get to the chopper!!",
"Honey, you shouldn't drink and bake.",

"Come with me if you want to live.",
"If it bleeds, we can kill it."
};
int curQuote = 0;
```

Here are 10 quotes. These are some of my favorites. Add or replace them with your favorites, maybe you find those from Arnold's Conan area better? (BTW, [this is a great source](#) of Arnold quotes.)

We also declare an `int` that defines the currently selected quote. When the button is clicked we must select a random value for `curQuote`, between 0 and 9. Therefore add this to the beginning of `handleButton()`:

```
void handleButton()
{
    curQuote = rand() % 10;

    if (wifi->isConnected())
        // rest of function is left out
}
```

Now we must replace the static string in the two HTTP body data callbacks: `httpLength()` and `httpData()`. Therefore replace "Arnold quote here" with `quotes[curQuote]` in both functions. For `httpData()` the `snprintf`-line should look like this:

```
snprintf(data, httpLength()+1, "{ \"value1\": \"%s\", \"value2\": %i.%i, \"value3\": 
↪%i}", quotes[curQuote], (int)temp, fracPart, lastBatteryLevel);
```

Now the app sends a random sequence of quotes to IFTTT. However, we need to seed the `rand()` function, before it is really random. In `setup()` we use the temperature as seed. Append this line to `setup()`:

```
srand(mono::IApplicationContext::Instance->Temperature->ReadMilliCelcius());
```

Now the quote sent to IFTTT is unpredictable.

The sugar on top

By now, we have the basic functionality in place. The app sends quotes to IFTTT, so let's add some extra functionality to make it appear more complete.

By default you control Mono's sleep and wake cycles, with the push button. However, it is a good idea to add an auto sleep function. This will prevent the battery from being drained, if Mono is left unattended.

The SDK provides this functionality with the class `PowerSaver`. This class will dim the screen after a period of inactivity and then, after more inactivity, put Mono in sleep mode.

Let's add this to our app. First we declare a global pointer to the object instance, just as we did with the `Wifi` object:

```
mono::PowerSaver *saver;
```

Then, insert this at the beginning of `setup()`. It is important that it comes before initialization of any UI class.

```
saver = new PowerSaver(10000, 60000);
```

This initializes the `PowerSaver` object with a configuration to dim the display after 10 seconds and then sleep after a further 60 seconds of inactivity.

We are done. Our final app sends random quotes to IFTTT using HTTP Post calls, and includes the current temperature and battery level in the call as well.

If the app is left unattended it will automatically sleep Mono, to preserve battery.

Source code

You can download the complete source code [here](#).

To compile the code you must replace or define 4 placeholders:

- YOUR_SSID: Your Wifi's name
- YOUR_PASSPHRASE: Your Wifi's password
- YOUR_EVENT_NAME: The eventname as defined in the Webhook applet on IFTTT
- YOUR_API_KEY: Your IFTTT *Api Key* for the Webhook.

Humidity app

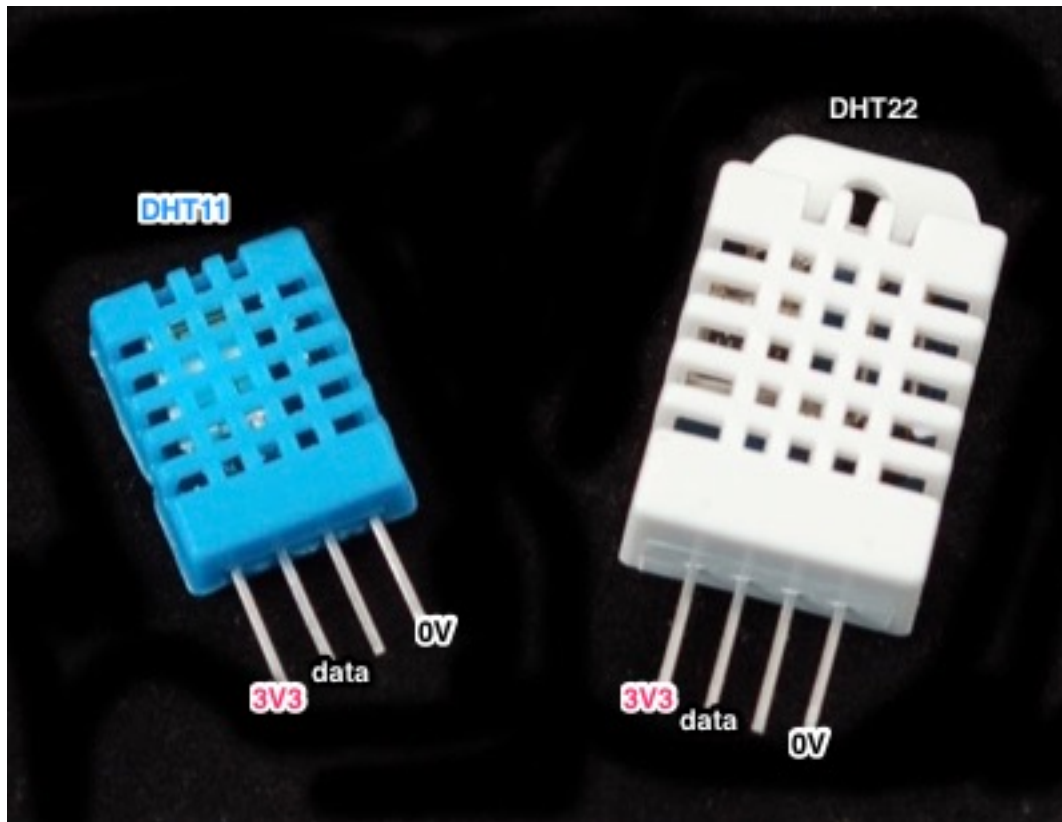
Humidity hardware setup

The purpose of this tutorial is to build a humidity app.



Sensor

To get humidity readings into my Mono, I will need a humidity sensor. For this app I will use the relatively low cost sensors [DHT11](#) and [DHT22](#). Their underlying hardware communication protocol is the same, but the interpretation of the readings differ slightly (DHT22 has better resolution).



Connecting the sensor to Mono

The sensor uses a single wire to transmit data, and it must get power through two additional wires (3.3V and 0V).

So I need three wires in total from Mono to the sensor. Mono's mini-jack accommodates a total of four wires, so I will use a mini-jack connector and solder a set of wires to it. For this particular application, I could use a regular three-wire mini-jack, but the mini-jack connector I have has four connections, so I will solder all four wires and reserve the fourth wire for future experiments.



Here I have put a red wire on the tip, a white wire on ring 1 (the one next to the tip), a black wire on the sleeve. The green wire is connected to ring 2, but it is not used in the app.

With that taken care of, I can connect the sensor to my Mono and start pulling out data from the sensor.

Data communication

To sanity check the connection, I will make the simplest possible app that can request a reading from the sensor, and then view the result on an oscilloscope. **You do not need to do this**, of course, but I will need to do that to show you what the sensor communication looks like.

An application to get the sensor talking must put 3.3V on the tip (red wire), and then alternate the data line (white wire) between 3.3V and 0V to tell the sensor that it needs a reading. The sleeve (black wire) is by default set to 0V, so nothing needs to be setup there.

More specifically, the data line must be configured to be an output pin with `pullup`. To request a reading from the sensor, the data line needs to be pulled down to 0V for 18ms, and then set back to 1. After that, the sensor will start talking.

The following program makes such a request every 3 seconds.

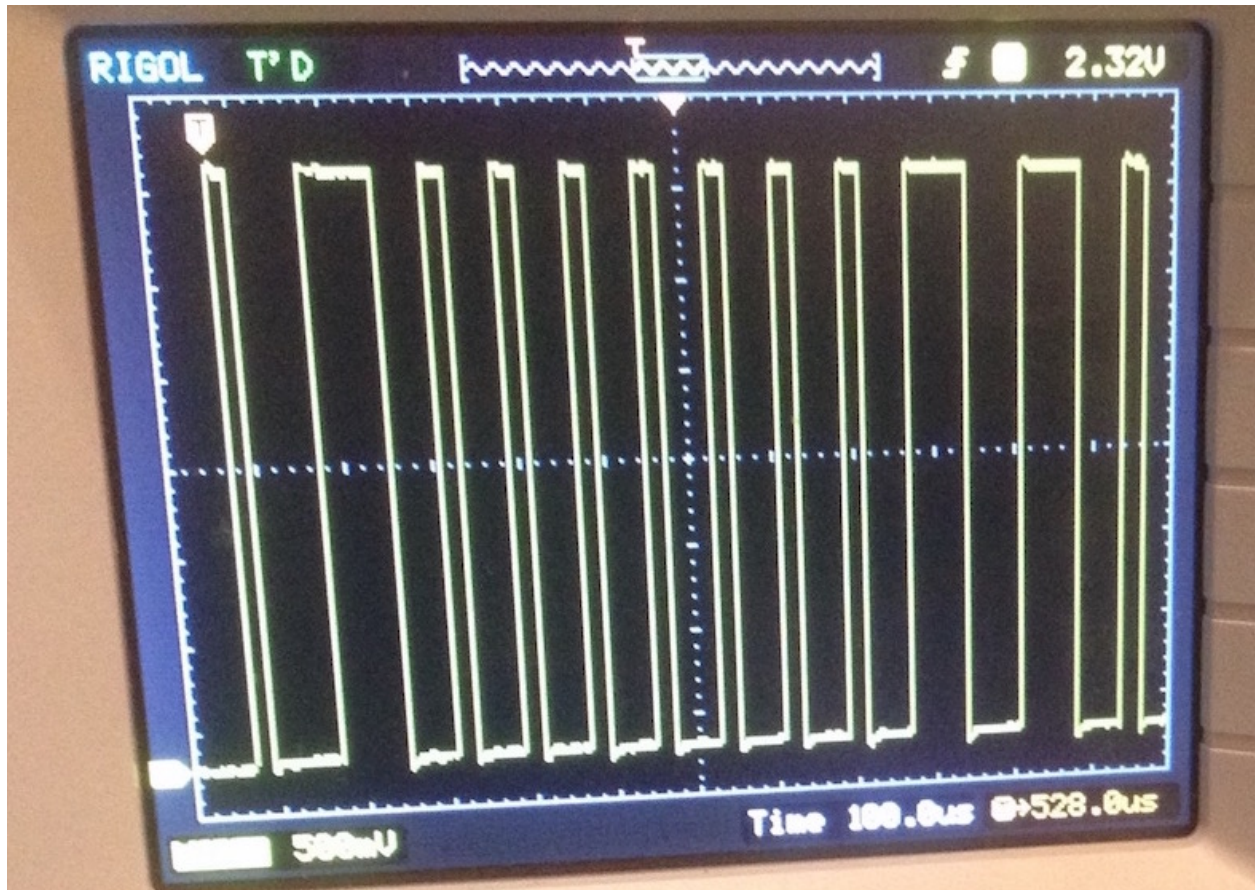
```
#include <mono.h>
#include <mbed.h>

class AppController
:
public mono::IApplication
{
    mono::Timer measure;
    mbed::Ticker ticker;
    mono::io::DigitalOut out;
public:
    AppController()
    :
        measure(3*1000),
        out(J_RING1,1,PullUp),
    {
        measure.setCallback<AppController>(this,&AppController::requestSensorReading);
    }
    void monoWakeFromReset ()
    {
        put3V3onTip();
        measure.Start();
    }
    void monoWillGotoSleep ()
    {
        turnOffTip();
    }
    void monoWakeFromSleep () {}
    void put3V3onTip ()
    {
        DigitalOut(VAUX_EN,1);
        DigitalOut(VAUX_SEL,1);
        DigitalOut(JPO_nEN,0);
    }
    void turnOffTip ()
    {
        DigitalOut(JPO_nEN,1);
    }
    void requestSensorReading ()
    {
        out = 0;
        ticker.attach_us(this,&AppController::IRQ_letGoOfWire,18*1000);
    }
    void IRQ_letGoOfWire ()
```

```
{  
    out = 1;  
}  
};
```

Side note: I use the `IRQ_` prefix on functions that are invoked by interrupts to remind myself that such functions should not do any heavy lifting.

When I connect Mono to the sensor, and hook up an oscilloscope to the data and ground wires, then I get the following picture of the communication when I run the app.



To the left you can see a tiny bit of the end of the 18ms period, ending in a rising edge (the transition from 0V to 3.3V) marked by **T** (the trigger point). From there on, the sensor takes over and starts alternating the data line between 3.3V and 0V.

The first 3.3V period is just a handshake, and after that the length of each 3.3V period determines whether data sent from the sensor is a logical 1 or a logical 0. For the screenshot above, the visible part of data is 0000000110.

What exactly does that mean? Well, I will tell you in the next part.

Humidity app

In the *first part* of this Humidity app tutorial, I showed how to connect a humidity sensor to Mono. Now, I will show how to get and display humidity and temperature readings.

Displaying readings

The humidity sensor measures both humidity and temperature, and I want these readings shown in a nice big font and funky colours.

```
#include <mono.h>
#include <ptmono30.h>
using mono::geo::Rect;
using mono::ui::TextLabelView;

class AppController
:
    public mono::IApplication
{
    TextLabelView humidityLabel;
    TextLabelView humidityValueLabel;
    TextLabelView temperatureLabel;
    TextLabelView temperatureValueLabel;
public:
    AppController()
    :
        humidityLabel(Rect(0,10,176,20),"humidity"),
        humidityValueLabel(Rect(0,30,176,42),"--.--"),
        temperatureLabel(Rect(0,80,176,20),"temperature"),
        temperatureValueLabel(Rect(0,100,176,42),"--.--")
    {
    }
    void monoWakeFromReset ()
    {
        humidityLabel.setAlignment(TextLabelView::ALIGN_CENTER);
        humidityLabel.setTextColor(TurquoiseColor);
        humidityValueLabel.setAlignment(TextLabelView::ALIGN_CENTER);
        humidityValueLabel.setFont(PT_Mono_30);
        humidityValueLabel.setTextColor(AlizarinColor);
        temperatureLabel.setAlignment(TextLabelView::ALIGN_CENTER);
        temperatureLabel.setTextColor(TurquoiseColor);
        temperatureLabel.setAlignment(TextLabelView::ALIGN_CENTER);
        temperatureValueLabel.setAlignment(TextLabelView::ALIGN_CENTER);
        temperatureValueLabel.setFont(PT_Mono_30);
        temperatureValueLabel.setTextColor(AlizarinColor);
        humidityLabel.show();
        humidityValueLabel.show();
        temperatureLabel.show();
        temperatureValueLabel.show();
    }
    void monoWillGotoSleep () {}
    void monoWakeFromSleep () {}
};
```

Getting data from the sensor

From the *first part* of this tutorial, you know how to start a reading from the sensor, but it gets somewhat more complicated to capture and interpret the data from the sensor.

The data from the sensor is a series of bits, where each bit value is determined by the length of each wave. So I can make my app to trigger on the start of each new wave and then record the time that has passed since the the last wave

started. The triggering can be done by attaching an interrupt handler to the data wire, which is done by using the `InterruptIn` class from the [mbed library](#).

Compared to the *first version*, I now have an array `bits` and an index `bitIndex` into this array so that I can collect the bits I read from the sensor. The `requestSensorReading` function now resets `bitIndex` before requesting a new reading, and `IRQ_letGoOfWireAndListen` sets up the function `IRQ_falling` to get called every time there is a *falling edge* on the data line from the sensor:

```
#include <mono.h>
#include <mbed.h>
using mono::io::DigitalOut;

#define LEADBITS 3
#define TOTALBITS LEADBITS+5*8

class AppController
:
public mono::IApplication
{
    mono::Timer measure;
    mbed::Ticker ticker;
    mbed::InterruptIn in;
    DigitalOut out;
    uint8_t bits [TOTALBITS];
    size_t bitIndex;
    uint32_t usLastTimeStamp;
public:
    AppController()
    :
        measure(3*1000),
        /// It is important that InterruptIn in initialised...
        in(J_RING1),
        /// ...before DigitalOut because they use the same pin, and the initialisation
        /// sets the pin mode, which must be pull-up.
        out(J_RING1,1,PullUp)
    {
        measure.setCallback<AppController>(&this,&AppController::requestSensorReading);
    }
    void monoWakeFromReset ()
    {
        put3V3onTip();
        measure.Start();
    }
    void monoWillGotoSleep ()
    {
        turnOffTip();
    }
    void monoWakeFromSleep () {}
    void put3V3onTip ()
    {
        DigitalOut(VAUX_EN,1);
        DigitalOut(VAUX_SEL,1);
        DigitalOut(JPO_nEN,0);
    }
    void turnOffTip ()
    {
        DigitalOut(JPO_nEN,1);
    }
    void requestSensorReading ()
```



```

{
    bitIndex = 0;
    out = 0;
    ticker.attach_us(this, &AppController::IRQ_letGoOfWireAndListen, 18*1000);
}
void IRQ_letGoOfWireAndListen ()
{
    out = 1;
    usLastTimeStamp = us_ticker_read();
    in.fall(this, &AppController::IRQ_falling);
}
void IRQ_falling ()
{
    uint32_t usNow = us_ticker_read();
    uint32_t usInterval = usNow - usLastTimeStamp;
    usLastTimeStamp = usNow;
    uint8_t bit = (usInterval < 100) ? 0 : 1;
    bits[bitIndex] = bit;
    ++bitIndex;
    if (bitIndex >= TOTALBITS)
    {
        in.disable_irq();
        // TODO:
        //async(this, &AppController::collectReadings);
    }
}
};

```

The `IRQ_falling` function calculates the time difference between the last falling edge on the data from the sensor, and if that interval is less than 100 μ s, then the received bit is a 0; otherwise it is a 1. When enough bits have been received, the interrupt is turned off so that I will stop receiving calls to `IRQ_falling`.

I use the `IRQ_` prefix on functions that are invoked by interrupts to remind myself that such functions should not do any heavy lifting. That is also why the (to be done) processing of the received bits is wrapped in an `async` call.

Interpreting the data from the sensor

Up until now, it has made no difference whether I was using a DHT11 or DHT22 sensor. But now I want to implement the `collectReadings` function to interpret the bits I get back from the sensor, and then the type of sensor matters.

I will start with the DHT11 sensor, which only gives me the integral part of the humidity and temperature value. So I need to go through the array of bits, skip the initial handshakes, dig out the humidity, dig out the temperature, and finally update the display with the new values:

```

// DHT11
void collectReadings ()
{
    uint16_t humidity = 0;
    for (size_t i = LEADBITS; i < LEADBITS + 8; ++i)
    {
        size_t index = 7 - (i - LEADBITS);
        if (1 == bits[i])
            humidity |= (1 << index);
    }
    uint16_t temperature = 0;
    for (size_t i = LEADBITS + 16; i < LEADBITS + 24; ++i)
    {

```

```
        size_t index = 7 - (i - LEADBITS - 16);
        if (1 == bits[i])
            temperature |= (1 << index);
    }
    humidityValueLabel.setText (String::Format ("%d%%", humidity) ());
    humidityValueLabel.scheduleRepaint ();
    temperatureValueLabel.setText (String::Format ("%dC", temperature) ());
    temperatureValueLabel.scheduleRepaint ();
}
```

For the DHT22 sensor, the values have one decimal of resolution. So I need to do a little bit more manipulation to display the reading, because the Mono framework do not support formatting of floating point:

```
// DHT22
void collectReadings ()
{
    uint16_t humidityX10 = 0;
    for (size_t i = LEAD; i < LEAD + 16; ++i)
    {
        size_t index = 15 - (i - LEAD);
        if (1 == bits[i])
            humidityX10 |= (1 << index);
    }
    int humiWhole = humidityX10 / 10;
    int humiDecimals = humidityX10 - humiWhole*10;
    uint16_t temperatureX10 = 0;
    for (size_t i = LEAD + 16; i < LEAD + 32; ++i)
    {
        size_t index = 15 - (i - LEAD - 16);
        if (1 == bits[i])
            temperatureX10 |= (1 << index);
    }
    int tempWhole = temperatureX10 / 10;
    int tempDecimals = temperatureX10 - tempWhole*10;
    humidityValueLabel.setText (String::Format ("%d.%0d%%", humiWhole,
↪humiDecimals) ());
    humidityValueLabel.scheduleRepaint ();
    temperatureValueLabel.setText (String::Format ("%d.%0dC", tempWhole,
↪tempDecimals) ());
    temperatureValueLabel.scheduleRepaint ();
}
```

What is still missing is detecting negative temperatures, unit conversion and [auto sleep](#), but I will leave that as an exercise. Of course, you *could* cheat and look at the full app in [MonoKiosk](#).

Quick Examples

Incrementing a variable on Mono

This is a brief example of how to show a incrementing number on Mono's display, using the Arduino IDE.

When Mono (and some Arduino models) runs a program there is more going on than what you can see in the `setup()` and `loop()`. Every time the *loop* is starting over, Mono will do some housekeeping in between. This include such tasks as updating the screen and servicing the serial port.

This means if you use `wait / sleep` functions or do long running ntensive tasks, Mono will not have time to update the screen or listening on the serial port. This will also affect Monos ability to receive a reset request, which is

important every time you are uploading a new sketch or app.

Hint: If you find your Mono ended up in a `while (1)` loop or something similar, see our brief tutorial on [Resetting Mono](resetting_mono.md).

To periodically increment a variable, and avoid doing *wait* or *sleep* calls, the following example uses an alternative approach to the *wait* function. To slow down the counting, we use a variable to count loop iterations and an `if` statement to detect when it reaches 1000 and then increment the counter and update the label on the screen.

Caution: When using this method the timing will be highly dependent on what mono is doing for housekeeping.

```

    /**
     *
     * This is a small example of how to show a counting variable on mono's screen.
     *
     * Instead of using a delay function to slow down the counting, I here use a
    ↪variable to count loop iterations
     * and an if() to detect when it reaches 1000 and then increment the counter and
    ↪update the label on the screen.
     *
     */
    ***/
#include <mono.h>

mono::ui::TextLabelView textLbl;

int loopItererations;
int counter;

void setup()
{
    textLbl = mono::ui::TextLabelView(mono::geo::Rect(0,20,176,20),"Hi, I'm Mono");
    textLbl.setTextColor(mono::display::WhiteColor);
    textLbl.show();
}

void loop()
{
    loopItererations++;

    if( loopItererations >= 1000 )
    {
        loopItererations = 0;
        counter++;
        textLbl.setText(mono::String::Format("count: %i", counter));
    }
}

```

Adding a Button to the Screen

In this quick tutorial we shall see how to add a set of push buttons to the screen.

The SDK comes this standard classes for screen drawing and listening for touch input. One of these classes are `ButtonView`. `ButtonView` display a simple push button and accepts touch input.

Reacting to clicks

Let us go create a new Mono project, fire up your terminal and:

```
$ monomake project buttonExample
```

To create a button on the screen we first add a `ButtonView` object to `AppController`. Insert this into `app_controller.h`:

```
class AppController : public mono::IApplication {

    // This is the text label object that will displayed
    TextLabelView helloLabel;

    // We add this: our button object
    ButtonView btn;

public:

    // The default constructor
    AppController();

    // We also add this callback function for button clicks
    void buttonClick();
```

We added a member object for the button itself and a member method for its callback. This callback is a function that is called, then the button is clicked.

Now, in the implementation file (`app_controller.cpp`), we add the button the constructor initializer list:

```
AppController::AppController() :

    // Call the TextLabel's constructor, with a Rect and a static text
    helloLabel(Rect(0,100,176,20), "Hi, I'm Mono!"),

    // Here we initialize the button
    btn(Rect(20, 175, 136, 40), "Click me!")
{
```

The button's constructor takes 2 arguments: *position and dimension* rectangle and its *text label*. The first argument is a `Rect` object, it defines the rectangle where the Button lives. This means it will draw itself in the rectangle and listen for touch input in this rectangle:

The second argument is the text label that is displayed inside the button. In this example it is just the text *Click me!*

To trigger a response when we click the button, we need to implement the function body for the `buttonClick` method. In `app_controller.cpp` add this method:

```
void AppController::buttonClick()
{
    helloLabel.setText("Button clicked!");
}
```

This method changes the content of the project templates existing `helloLabel` to a new text. Lastly, we connect the button click handler to call our function. From inside the `monoWakeFromReset` method, we append:

```
// tell the label to show itself on the screen
helloLabel.show();
```

```
// set the callback for the button click handler
btn.setClickCallback<AppController>(this, &AppController::buttonClick);
// set the button to be shown
btn.show();
```

That's it! Run `make install` and see the example run on Mono:



Periodically call a Function

*In this quick example we will see how to use a **Timer** to repetitively call a function*

A big part of developing apps is do tasks at regular intervals. Mono employs a timer architecture that allows you to schedule single or recurring function calls, ahead of time. The semantics are “*call this function 3 minutes from now*”, or “*call this method every 45 second*”.

Timers

The timer system on Mono is very powerful, you can schedule as many timers as of like! (Okay, you are limited by the amount of RAM). The timers are instances of the class `Timer` and they are built upon the `bed::Ticker` class. This architecture leverages the versatility of the *mbed* timers and adds thread safety from Mono’s own `Timer` class.

You can schedule a method to be called 5 minutes from now, by a single line of code:

```
mono::Timer::callOnce<MyClass>(5*60*1000, this, &MyClass::MyCallbackMethod);
```

This will create a timer instance on the heap, and it will deallocate itself after it has fired. Because we use C++ methods, and not C functions as callbacks, you must provide the `this` pointer and the type definition of the context. (`MyClass` in the example above.) The last parameter is the pointer to the actual method on the class. This makes the call a bit more verbose, compared to C function pointers, but being able define callback methods in C++ is extremely powerful.

Note: In recent versions of C++ (C++11 and C++14), lambda functions has been added. These achieve the same goal with a cleaner syntax. However, we cannot use C++11 or 14 on Mono, the runtime is simply too large!

Call a function every second

Now, let us see how to repeatedly call a function every second. First, we create a new project from the console / terminal:

```
$ monomake project timerExample
```

Open the `app_controller.h` file and add a `Timer` as a member on the `AppController` class, and define the method we want to be called:

```
class AppController : public mono::IApplication {

    // This is the text label object that will displayed
    TextLabelView helloLabel;

    // this is our timer object
    Timer timer;

public:

    //this is our method we want the timer to call
    void timerFire();
```

Because we want to repetitively call a function, we need the timer to stick around and not get deallocated. Therefore, it is declared as a member variable on `AppController`. In the implementation file (`app_controller.cpp`) we need to initialize it, in the constructors initialization list:

```
AppController::AppController() :

    // Call the TextLabel's constructor, with a Rect and a static text
    helloLabel(Rect(0,100,176,20), "Hi, I'm Mono!"),
    // set our timers interval to 1000 ms
    timer(1000)
{
```

Let us add the body of the `timerFire` method to the implementation file, also:

```
void AppController::timerFire()
{
    printf("Timer did fire!\t\n");
}
```

Lastly, we tie the timer callback handler to our method. This is done from inside the `monoWakeFromReset` method:

```
void AppController::monoWakeFromReset()
{
    // tell the label to show itself on the screen
    helloLabel.show();

    // set the timers callback handler
    timer.setCallback<AppController>(this, &AppController::timerFire);
```

```
// start the timer
timer.start();
}
```

All right, go to the console and run `make install` and our app should compile and upload to mono. Open a serial terminal and you should see:

```
Timer did fire!
Timer did fire!
Timer did fire!
Timer did fire!
```

Arriving with 1 second intervals.

Timing the UI

Now, let us step it up a bit. We want to toggle a UI element with our timer function. The SDK includes a class called `StatusIndicatorView`, it mimics a LED that just is *on* or *off*. Lets add it as a member on our `AppController`:

```
class AppController : public mono::IApplication {

    // This is the text label object that will displayed
    TextLabelView helloLabel;

    Timer timer;

    StatusIndicatorView stView;
```

We also need to initialize with position and dimension in the initializer list:

```
AppController::AppController() :

    // Call the TextLabel's constructor, with a Rect and a static text
    helloLabel(Rect(0,100,176,20), "Hi, I'm Mono!"),
    // set our timers interval to 1000 ms
    timer(1000),

    stView(Rect(75,55,25,25))
{
```

Then, in the `monoWakeFromReset` method we must set its **visibility state* **to shown*:

```
// tell the label to show itself on the screen
helloLabel.show();

// set the timers callback handler
timer.setCallback<AppController>(this, &AppController::timerFire);
// start the timer
timer.start();

stView.show();
```

Last we insert code to toggle its state in the `timerFire` method:

```
void AppController::timerFire()
{
```

```
printf("Timer did fire!\t\n");

stView.setState(!stView.State());
}
```

Go compile and run the modified code. You should see this on your mono:



Sample code

Here is the full source code for reference:

app_controller.h:

```
#ifndef app_controller_h
#define app_controller_h

// Include the Mono Framework
#include <mono.h>

// Import the mono and mono::ui namespaces into the context
// to avoid writing long type names, like mono::ui::TextLabel
using namespace mono;
using namespace mono::ui;

// The App main controller object.
// This template app will show a "hello" text in the screen
class ApplicationController : public mono::IApplication {

    // This is the text label object that will displayed
    TextLabelView helloLabel;

    Timer timer;

    StatusIndicatorView stView;

public:
```

```

    //this is our method we want the timer to call
    void timerFire();

    // The default constructor
    AppController();

    // Called automaticlly by Mono on device reset
    void monoWakeFromReset();

    // Called automatically by Mono just before it enters sleep mode
    void monoWillGotoSleep();

    // Called automatically by Mono right after after it wakes from sleep
    void monoWakeFromSleep();

};

#endif /* app_controller_h */

```

app_controller.cpp:

```

#include "app_controller.h"

using namespace mono::geo;

// Contructor
// initializes the label object with position and text content
// You should init data here, since I/O is not setup yet.
AppController::AppController() :

    // Call the TextLabel's constructor, with a Rect and a static text
    helloLabel(Rect(0,100,176,20), "Hi, I'm Mono!"),
    // set our timers interval to 1000 ms
    timer(1000),

    stView(Rect(88,55,25,25))
{
    // the label is the full width of screen, set it to be center aligned
    helloLabel.setAlignment(TextLabelView::ALIGN_CENTER);

    // set another text color
    helloLabel.setTextColor(display::TurquoiseColor);
}

void AppController::timerFire()
{
    printf("Timer did fire!\t\n");

    stView.setState(!stView.State());
}

void AppController::monoWakeFromReset()
{
    // At this point after reset we can safely expect all peripherals and
    // I/O to be setup & ready.

```

```
// tell the label to show itself on the screen
helloLabel.show();

// set the timers callback handler
timer.setCallback<AppController>(this, &AppController::timerFire);
// start the timer
timer.start();

stView.show();
}

void AppController::monoWillGotoSleep()
{
    // Do any clean up here, before system goes to sleep and power
    // off peripherals.
}

void AppController::monoWakeFromSleep()
{
    // Due to a software bug in the wake-up routines, we need to reset here!
    // If not, Mono will go into an infinite loop!
    mono::IApplicationContext::SoftwareResetToApplication();
    // We never reach this point in the code, CPU has reset!

    // (Normally) after sleep, the screen memory has been cleared - tell the label to
    // draw itself again
    helloLabel.scheduleRepaint();
}
```

Measuring the Temperature

This quick tutorial will demonstrate how you measure the temperature, by using the standard temperature sensor API

Mono has a built-in thermometer that is situated on the PCB under the SD Card connector. We have a standard API for getting the temperature in degrees Celcius. If you wish to get convert to Fahrenheit, use this formula: $(^{\circ}\text{C} \cdot 1.8 + 32)$

Example

Let us try to fetch the temperature! The Mono SDK uses a standard interface for getting the temperature, that abstracts away the hardware. The interface contains of only two functions:

- `Read()` To get the temperature in celcius (as an integer)
- `ReadMilliCelcius()` To get the temperature in an integer that is 1000th of a celcius. $(1^{\circ}\text{C} = 1000^{\circ}\text{mC})$

You acquire a reference (a pointer) to the interface through the global `IApplicationContext` variable:

```
sensor::ITemperature *temp = IApplicationContext::Instance->Temperature;
int mcel = temp->ReadMilliCelcius();
```

Now the variable `mcel` hold the temperature in millicelcius. Divide by 1000 to get the value in celcius. You can easily print the temperature on the serial port by using `printf`:


```
printf("%d.%d", mcel/1000, mcel%1000);
```

Caution: Please observe that we cannot use the `%f` format specifier in the `printf` function! To make the application binary smaller, it is not linked with `printf`'s floating point formatting. If you wish to add floating point I/O, then you should add `-u _printf_float` in the linker command!

That is it. You can now read the temperature! Go hack or keep reading, for a little more on temperature measurements.

Temperature measuring Caveats

Measuring the temperature seems like a simple operation, but you should know that it is actually quite difficult to get it right. First for all, unless you really invest money and time in advanced equipment and in calibrating this equipment, then you will not get a precise measurement. But then, what is a precise measurement?

First let us visit the terms: *absolute* and *relative* measurements. An absolute temperature measurement is a temperature measured against a fixed global reference. At the summer the sea temperature at the beaches reach (25°C) or (77°F) . This is an absolute measurement. In contrast if I say: The sea temperature has risen by (2°C) or $(3,5^{\circ}\text{F})$, rise in temperature is a relative measurement.

When measuring temperature you should know that absolute measurements are hard, and relative measurements are easy in comparison. Normal household thermometers *do not* achieve a precision below (1°C) or (1.8°F) , in absolute measurements. But their relative precision can be far better - like (0.1°C) or (0.18°F) .

Mono's built-in thermometer share the same characteristics. However, be aware that the thermometer is mounted on the PCB which get heated by the electronics inside the device. You are measuring the temperature of the PCB - not the air temperature. To overcome this you can put mono in sleep mode for some time, and then wake up and measure the temperature. When Mono is in sleep, the PCB will (over time) get the same temperature as the air around it.

Using the Buzzer

In this quick tutorial I will demonstrate how to make Mono buzz, using the simple API in the SDK

Mono has a built-in buzz speaker. The speaker is directly connected to a GPIO on Mono's MCU. We have configured the MCU such that the GPIO pin is used by a hardware PWM. This means the speaker is driven by a square pulse signal generated by the MCU hardware. The software only has to turn the PWM on and off.

Simple Example

Let us begin with a code snippet that beeps for 0.5 sec:

```
void AppController::monoWakeFromReset() {

    // Get a pointer to the buzzer object
    mono::sensor::IBuzzer *buzzer = mono::IApplicationContext::Instance->Buzzer;

    // make a beep for 0.5 sec
    buzzer->buzzAsync(500);
}
```

First we get a pointer to the current buzzer object that has been created by the global `ApplicationContext` object. All buzz-speaker objects must implement the `IBuzzer` interface, that defines methods to emit buzzing sounds.

Then we use the method `buzzAsync` that turns on the speaker. The important thing here is to note that the buzzing is asynchronous. The signal sent to the speaker is hardware generated, so the software does not need to do anything. When `buzzAsync` returns, the buzzing is still going on - and will it do so for the next 0.5 sec.

Multiple Beeps

If you want to make, say 3 beeps in a row, you need to use callbacks. This is due to the asynchronous behaviour of the `IBuzzer` interface. Luckily there is a similar method called: `buzzAsync(void (*callback)(void))`. This method takes a callback function, that gets called when the buzzing has ended. We can use this function to chain the buzzing, thereby making multiple beeps.

To do this, we use the built-in `Timer` class to control the delay between beeps.

Note: This is the best practice approach. The lazy ones might choose to use several `wait_ms()` calls. But this approach will stall the CPU, making it unresponsive.

In our `AppController` we declare an integer count variable. We also add a callback function for the buzz (`buzzEnded()`) and one for the delay (`pauseEnded()`).

wakeFromReset:

```
void AppController::monoWakeFromReset() {  
  
    // init the count to 0  
    count = 0;  
  
    mono::sensor::IBuzzer *buzzer = mono::IApplicationContext::Instance->Buzzer;  
  
    // begin the buzzing (for 0.5 sec)  
    buzzer->buzzAsync<AppController>(500, this, &AppController::buzzEnded);  
}
```

buzzEnded:

```
void AppController::buzzEnded() {  
  
    // increment the buzz beep count  
    count++;  
  
    // If we have buzzed less the 3 times, start a delay timer  
    if (count < 3)  
        mono::Timer::callOnce<AppController>(500, this, &AppController::buzzEnded);  
}
```

pauseEnded:

```
void AppController::buzzEnded() {  
  
    //the delay timed out - begin buzzing again  
  
    // get the buzzer pointer  
    mono::sensor::IBuzzer *buzzer = mono::IApplicationContext::Instance->Buzzer;
```

```
// begin buzzing again (for 0.5 sec)
buzzer->buzzAsync<AppController>(500, this, &AppController::buzzEnded);
}
```

Now, since we use Timers and async buzzing, Mono will stay responsive during both the buzzing and the pauses in between. This means it will keep sampling the touch input, updating display and running other timers in the background.

Killing a buzz

Say you might have called `buzzAsync(60000)`, which is a one minute long buzz tone. After some seconds you regret the decision and wish cancel the buzzing. To do that you use the `buzzKill()` method! Calling this method will immediate stop any buzzing started earlier.

Note: If you have installed a callback at the end of your buzz - the callback will still get called.

In-Depth Articles

In this chapter we shall take a dive into the structure and workings of *Mono Framework*. The purpose of each article is to give you an understanding of how mono internals works. You do not have to read these articles, but knowledge of the framework design and workings will definately help you. Especially if (when) you bump into problems.

Architectural Overview

In this article we take a quick tour of the complete framework, its classes and features. We will visit the most important topics and requirements for developing Mono applications.

Who should read this?

If you wish to understand the concepts and thoughts behind the frameworks structure, this article is for you. Or if you should choose to read only one in-depth article, it should definitely be this one!

Warning: This is a *draft article*, that is *work in progress*. It still needs some work, therefore you might stumble upon missing words, typos and unclear passages.

Overview

Mono Framework is a collection of C++ classes, all build on top of the `mbed library` created by ARM. The complete Mono software stack consists of 3 levels of abstractions, which are separate libraries:

1. **Mono layer** (C++): All high-level classes
2. **mbed layer** (C++/C): Hardware level I/O and functions (including most of stdlib)
3. **Cypress layer** (C): Hardware dependent code, generated by PSoC Creator

In this article we focus mainly on the *Mono layer*. Mono is an open system, so you have access to the underlying layers from your application. However, you should use only layer 3 (and some of mbed), if you really can not avoid it. Using these layers might break compatibility with future hardware releases and the *to-be-released* simulator.

API Overview

Below is a diagram of the features provided by *Mono Framework*. These are the high-level interfaces that makes it fast and easy, for you to take advantage of all Mono's features.

As you can see in the diagram, the features can be grouped by their function. Some framework classes are generic, like the *String* class. Other serves a specific purpose, like providing the accelerometer interface (*IAccelerometer*).

Core Concepts

Since there is no operating system, your application will run on *bare metal*, meaning it interfaces the hardware directly. On a versatile platform, such as Mono, it means that your application must deal with some critical events and concepts. In the next sections we shall take a look at the key functions and requirements of all applications targeting the OpenMono platform.

Application lifecycle

The application lifecycle is the time from execution of the first instruction to the last. In conventional environments this is from `main()` gets called, until it returns:

```
// This is a normal familiar C++ application main function:
int main(char *argv[], int argc)
{
    // Application lifecycle begins

    // do stuff

    // Application lifecycle ends
    return 0;
}
```

This is the case when you are inside an operating system. Mono is an embedded platform, so here the lifecycle is quite different and determined by the power *on* and power *off* events.

When the CPU powers up, it will immediately start executing your application. And it will not stop before you cut the CPU's power source - literally! There is no `return 0` that stops your application.

Mono is Always on

Mono's hardware is always powered, because there is just no power switch! You can not simply cut the power to the CPU, when you want to turn off Mono. The "turn off" feature needs to be the software that throttles down the CPU and puts all peripherals into a low power state. We call this state: *sleep mode*.

Mono Framework helps you with handling *sleep mode*. By default Mono's side-button will toggle sleep mode. It will put Mono to sleep, and wake Mono up again if pressed during sleep. You do not need to do anything to support sleep mode, it is provided to you by the framework. Only if you need to make use of the side-button for your own purpose, you must provide a way of going to sleep. This is done by calling the `IApplicationContext::EnterSleepMode` method:

```
// put mono into sleep mode:
mono::IApplicationContext::EnterSleepMode(); // execution halts here until wake-up

// only after wake-up will EnterSleepMode return
printf("Mono has awoken!");
```

Because power is never cut from the CPU, it rarely resets. The application state is preserved across sleep cycles, meaning that your application lifespan will be long. Even significantly longer when compared to desktop applications. The long lifespan makes the application more vulnerable to errors, such as memory leaks, corrupting memory or stack overflows. The point is: it is a tough job to be an embedded developer .

Power On Reset

The term *Power On Reset* or *POR* means the initial reset that occurs when the CPU powers on. This is when the power supply is first asserted or the physical H/W reset line is de-asserted. On Mono a POR is the same as a hardware reset.

A POR can be triggered in a number of different ways:

- Pressing Mono's reset button
- If Mono's battery is drained, the power control software will halt the CPU's. When Mono is charged, the system will wake the CPU and a software triggered POR happens.
- Uploading a new application to Mono, using `monoprog`.
- Your application can trigger a *SoftwareReset*, that results in a POR.

Every Mono application is required to handle the POR event. It is here your application must setup all needed peripherals, such as temperature sensor or SD Card file I/O. If you use any *UI Widgets*, you need to initialize them on POR as well.

Later in the *Required virtual methods* section, we shall see how you handle the POR event.

Sleep and Wake-up

When Mono goes to *sleep mode* it turns off all peripherals to minimize power consumption.

You have the option to handle the *go to sleep* and *wake from sleep* events, as we shall see in the section about the *The AppController*. We imagine you might need to do some book-keeping or I/O flushing before entering sleep. Likewise, you may need some setup after waking from sleep. If you use the display, you *will* need to take repaint actions when waking from sleep.

However, if you are lazy could could just trigger a *SoftwareReset* upon *wake from sleep*, but you would loose any state that is not serialized.

The run loop

Like most modern application runtimes, Mono has an internal *run loop* (also called an *event loop*). The loop handles periodic tasks, like sampling the touch system, updating the display, processing Timers and handling any other asynchronous task. You can inject your own tasks into the run loop, and there by achieve the Arduino-like `loop()` functionality.

The run loop is started right after your POR handler returns, and runs for the entire length of the application lifecycle.

Callback functions

Because we have a run loop we can make tasks asynchronous. This does not mean your code will run concurrently, it means that you can put tasks in the background. You do not need to think about race-conditions and other rough topics related to parallelism.

You use callback functions to handle events that arrive, and require your action. For example you can schedule a function call in the future. The *Timer* class can schedule a function getting called 5 secs from now:

```
mono::Timer::callOnce<MyClass>(5000, this, &MyClass::futureFunction);
```

Notice the syntax here. We use C++ templates and function pointers. Reason is the complexity of context and function pointers in C++. In C you create function pointers with ease:

```
void MyFunction() {}

mono::Timer::callOnce(5000, &MyFunction);
```

C functions has no context (do not belong to a class), and can be identified by a pointer. Functions (*methods* to be precise) in C++ exists as attributes on object instances. When we use these as callback handlers, we need to define 3 parameters:

1. The type of class where the *method* is defined
2. Provide a pointer to an instance of the class (the *object*)
3. Provide the actual function (*method*) pointer

Note: That we can have callback methods in old C++98 is a kind of hack. In more modern C++ version, *lambda functions* achieve the same - but less verbose. Unfortunately Mono do not have enough memory to contain the runtime libraries for either C++11 or C++14.

Timers

- Timers trigger a periodic event handler callback
- Real-Time apps might update its state/content on a regular interval
- Timers can also be used to call a function at some point in the future (as soon as possible).

Queued interrupts

- in embedded environment interrupts are hardware triggers, that call a C function (the ISR)
- the ISR should be fast and return very quickly - a lot of concurrency issues arise when using ISR.
- mono uses Queued interrupt, where the ISR is handled in the run loop.
- no concurrency issues
- you can longer lived ISR's
- they can debounce your hardware input signals, to create more robust handling of button or switches

The AppController

All application must have a app controller - this is there entry point

Required virtual methods

Application Entry Point & Startup

1. static inits
2. main func
3. app ctrl POR method
4. run loop

The Bootloader

Crashes and Exceptions

Best Praticce

some do and dont's

Further reading

in depth articles:

- Boot and Startup procedures
- Queued callbacks and interrupts
- [[Display System Architecture|display_system_architecture]]
- Touch System Architecture
- Wifi & networking
- Power Management Overview
- Memory Management: Stack vs heap objects?
- Coding C++ for bare metal
- The Build System

Display System Architecture

Mono display system is makes it easy and fast to create graphical user interfaces (GUIs). You can take advantage of the many high-level classes, that display controls or text in the screen.

Who should read this?

In this article we will take an in-depth look at Mono's display system. You should read this if you wish to create your own *User Interface* elements or if you experience issues related to displaying graphics. Also, if you simply would like to know more about how the software works under the hood. I presume you already are familiar with other GUI system programming, like iOS or Qt.

Overview

The Mono framework implements a display stack that closely assembles the first computer GUI systems, like the first Mac OS or Atari TOS. It is a single display buffer that your application paints in. The buffer is placed in the display chip, and is therefore not a part of the MCU systems internal RAM. This means writing (and reading) to the display buffer is expensive operations, and should only be done in an efficient way.

To paint on the display the view stack has 3 distinct layers:

1. **The Display controller:** An object that communicates with the hardware display chip, and can read and write to the display buffer. The display controller objects can write pixels in an file I/O like manner. It does not have any notion of text glyphs or graphical shapes.
2. **The Display Painter:** The painter object can translate geometrical shapes into pixels. It utilizes the *Display Controller* as a pixels drawing target. The painter can also draw individual text glyphs, and works with colors too.
3. **The Views:** A view is an abstract class that represents a User Interface element, like a button. It uses the *Display Painter* to composite a complete element from a series of shape painting routines. Some views also works with touch input.

We shall only visit the last layer (Views) in this tutorial.

The Views

All UI element classes inherit from the `View` class. The view class defines the properties and behaviors shared by all UI elements. The mono framework comes with a set of predefined UI views that comprises different UI elements. They all inherit from the `View` class, as seen on the figure below:

If you need learn about the specific UI classes can their usage, please see the reference documentation or the **Drawing UI Elements** tutorial.

As all classes inherit from the parent `View` class, they all define these central properties:

- **The View Rect:** A rectangle that defines the boundaries of the view. This is the views width and height, but also its X,Y position on the display.
- **Standard Colors:** All views share a palette of standard/default colors for borders, text, backgrounds and highlights. Changing one of these will affect all view subclasses.
- **Dirty state:** Views can be *dirty*, meaning that they need to be repainted on the screen. You might change the content of a `TextLabelView`, and the view will need to be repainted - therefore it is *dirty*. When the view has been repainted, the dirty state is cleared.
- **Repainting:** All `View` subclasses must define the protected method `repaint()`. Dirty views are scheduled for repaint by the display system, meaning that the `repaint()` method is automatically called to actually draw the view. If you create your own custom views, all your shape painting *must* happen inside the `repaint()` routine.
- **Visibility state:** Views can be visible or invisible. When first created, a view is always invisible. This means it will not be scheduled for repaints at all. To make a view appear on the display, you must first call the `show()` method. This will set its state to *visible*.

Since all views share a single global display buffer, you can (by mistake or on purpose) position one view overlapping another. The display system does not have any notion of a Z-axis. To the top-most view will be the one that gets its `repaint()` method called last. The display system keeps dirty views in a queue, so they are repainted in a FIFO style manner.

When you create your own views, it is your responsibility to respect the views boundaries. Say, a view with the dimensions 100x100, must not draw any shapes outside its 100x100 rectangle. Shape drawing inside the `repaint()` method is not automatically clipped to the views bounding rectangle. It is perfectly possible to create a view, that completely ignores its bounding rectangle.

In contrast to many modern GUI systems, mono views cannot contain nested views. However, this does not mean a view cannot contain another. It just has to manually manage it.

Display Coordinate System

All views and painted shapes exists in the painter's coordinate system. This coordinate system is cartesian with origin in the top left corner. The positive Y direction is downward, while positive X is left to right. The coordinates are in pixels, meaning they are integers.

An example of the used coordinate system is seen in the figure above. Notice how the pixel's coordinate references the upper left corner of the pixel area - not the center.

Because views cannot be nested, we use only one global coordinate system. It is called the absolute coordinate system, and all shapes and views are painted relative to that. This means that if you position views with the offset coordinate `\(20,20\)`, you must offset all shape painting with `\(20,20\)`.

Rotations

Mono includes an accelerometer, that enables you to detect orientation changes. You can create an application that layout its content differently in landscape and portrait modes.

Unfortunately, at this time, I have yet to implement an automatic coordinate system rotation, that uses the accelerometer. I plan to augment the `Display Painter` class with the ability to rotate the coordinate system, to account for mono physical orientation. This will mean the coordinate system's origin will always be the upper left corner relative to gravity, and independent on the physical orientation.

Pixel Blitting

The *display painter* class takes commands like `drawRect(x,y,w,h)`, that paints an outlined rectangle on the display. It handles conversion from geometric shape parameters, into a series of pixels. These pixels are written to the display through the *Display Controller* object.

The pixel color is determined by the state of the painter. The painter has foreground and background color, that can be set before the calls to shape drawing routines. Shapes are painted using the foreground color by default, but you can explicitly tell the routines to use the background color instead.

The text glyphs drawing routine uses both the foreground and background colors, to draw the text glyphs against the background color.

Bitmaps & Colors

The display painter cannot take pixels as input. If you need to draw raw pixels or bitmaps from a file or memory, you need to interface the Display Controller directly. The display controller has a cursor that points to a location on

the display. When you write a pixel, the cursor increments. The incrementation is from left to right, and downward. (Normal western reading direction.)

Basically you need only to use 2 methods: `write(color)` and `setCursor(x,y)`. You can see how if you take a look at the source code for the class `ImageView`. It blit pixels using the *display controller*, from within its `repaint()` method.

If you plan to use bitmaps, keep in mind that Mono's memory is very limited. Therefore I will encourage you *not* to use large in-memory pixel buffers. Instead use the SD Card file I/O, as done by the `ImageView` class.

When you write raw pixels, you must use the correct pixel color format. For mono this is **16 bit, 5-6-5 RGB colors**. Note that Mono's CPU architecture is little endian, and the display uses big endian. If you define a color like this:

```
uint16_t color = 0x07E0; // I think this might be a green color?
```

The color will be interpreted by the display as: `0xE007`. For convenience you should use the `Color` class, that has a constructor that takes RGB components as separate values.

V-Sync and refreshes

The display hardware periodically refreshes the LCD. If you change the display buffer during a refresh, you will see weird artifacts. Especially animations are prone to such artifacts.

To counter this mono uses *tearing effect* interrupts. This interrupt works like the v-sync signal on RGB interfaced displays. It occurs the moment after a display refresh. After the interrupt there is a time window, where the display buffer can be changed, before the display refreshes again.

Modern systems uses a technique called double buffering, where two separate display buffers exists. This means that one can be modified while the other is shown. When all changes has been written, the buffer that is displayed are changed to the other one. This technique makes it possible is to (slowly) write a lot of changes to the display, and have them appear instantly.

Unfortunately we do not have this facility in Mono. There is only one display buffer. This means all drawing must have finished, by the time the display is refreshed again. To not exceed the time window between display refreshes, all painting routines must be very efficient and optimized. If you create your own view subclasses, keep in mind that your drawing must be highly efficient. It is best only to paint changes, and not the entire view again.

The display system automatically handle this *tearing effect* timing, and skips repainting, should the CPU be too busy at the moment of the interrupt.

The C programmers guide to C++

In this article you will learn to use C++, if you previously have coded in Arduino or C. We will go through to fundamental C++ constructs and concepts.

Who should read this?

I assume that you are familiar with basic C, nothing fancy - just plain `main()` with function calling and *maybe - just maybe*, some pointer juggling. Also, if you have coded Arduino and are used to the concepts there - this article is for you!

Read on, and we shall take a tour of the basic features and headaches of C++.

Overview

Allow me to start off with a ultra brief history lesson in C++. Back in the days it started its life as an extension to C. This means you still have access to all of C from C++. Originally C++ was named *C with Classes*, later the name was changed to C++. The name comes from the *increment by one* operator in C: ++. The symbolism is to indicate that C++ is C incremented or enhanced.

C++ introduces a set of new language constructs on top of C, and also tightens some *type casting* rules that are looser in C.

In this article we will examine the following C++ features:

1. *Classes*
2. *Inheritance*
3. *Constructors*
4. *Namespaces*
5. *References*
6. *The rule of 3*

Let's dive in.

Classes

If you ever heard of *object oriented programming*, you might have heard about classes. I believe the idea behind the class concept, is best explained by an example.

Let's say we want to programmatically represent a rectangle (the geometric shape, that is). Our rectangle will have the following properties:

- X offset (X coordinate of upper left corner)
- Y offset (Y coordinate of upper left corner)
- Width
- Height

In C code you should normally create a `struct` type that represents the collection of properties like this:

```
struct Rect {  
    int x;  
    int y;  
    int width;  
    int height;  
};
```

If we want to create a `Rect` variable in C, we now do:

```
struct Rect windowFrame;  
windowFrame.x = 10;  
windowFrame.y = 10;  
windowFrame.width = 500;  
windowFrame.height = 500;
```

A `struct` in C provides a great way of grouping properties that are related. In C++ we achieve the same with the `class` keyword:

```
class Rect {
public:
    int x;
    int y;
    int width;
    int height;
};
```

Note that apart from the word change from `struct` to `class`, the only difference is the `public` keyword. Do not mind about this know, we shall get back to it.

Now in C++ we have declared the class `Rect`, and we can use it like this:

```
Rect winFrm;
winFrm.x = 10;
winFrm.y = 10;
winFrm.width = 500;
winFrm.height = 500;
```

Notice we just declare the type `Rect`, no need for the extra keyword `struct`, like in C.

What we have in fact created now are an *instance* of our class `Rect`. An instance is also called an *object*.

Adding functions to classes

Let's say we want a function to calculate the area of our rectangle. Simple in C:

```
int calcArea(struct Rect rct)
{
    return rct.width * rct.height;
}
```

If I were to write the same function in C++, I could then use the class `Rect` and not the `struct`. To rewrite the function to handle the `Rect`, I need only to remove the `struct` keyword and the function would work with C++ class types. However, the concept of *object oriented programming* teaches us to do something else.

We should group functionality and data. That means our `Rect` class should itself know how to calculate its own area. Just like the `Rect` has width and height properties, it should have an area property.

We *could* define an extra variable in the class, like this:

```
class Rect {
public:
    int x;
    int y;
    int width;
    int height;
    int area; // a new area variable
};
```

This would be highly error prone though. Since we have to remember to update this variable everytime we change width or height. Let us instead define area as a function that exists on `Rect`. The complete class definition will look like this:

```
class Rect
{
public:
```

```

int x, y;
int width, height;

int area()
{
    return width * height;
}
};

```

Now our *Rect* class consists of the 4 variables and a function called `area()`, that returns an `int`. A function that is defined on class like this, is called a *method*.

We can use the *method* like this:

```

Rect winFrm;
winFrm.x = 10;
winFrm.y = 10;
winFrm.width = 500;
winFrm.height = 500;

int area = winFrm.area();

```

This is the idea of object oriented coding - where data and related functionality are grouped together.

Access levels

As promised earlier let us talk briefly about the `public` keyword. C++ lets you protect variables and methods on your classes using 3 keyword: `public`, `protected` and `private`.

So far we only seen *public* in use, because it allows us to access variables and methods from outside the class. However, we can use the other keywords to mark variables or methods as inaccessible from outside the class. Take an example like this:

```

class CreditCard
{
private:
    const char *cardNumber;

protected:
    const char *cardType;
    const char * cardholderName;
    int expirationMonth;
    int expirationYear;

public:
    const char *cardAlias;

    int getAmount();
};

```

In this example we created the class *CreditCard*, that defines a persons credit card with all the data normally present on a credit card.

Some of the variables are sensitive and we don't want developers to carelessly access them. Therefore we can use access protection levels to block access to these from code outside the class itself.

Private members

The variable `cardNumber` is marked as *private*. This means it is visible only from inside the *CreditCard* class itself. No outside code or class can reference it. Not even a *subclass* of *CreditCard* have access to it. (We will get to *subclasses* in the next section.)

You should use *private* properties only when you actively want to block future access to variables or methods. Don't use it if you just can't see any reason not to. The paradigm should be to actively argue that future developers shall never access this variable or method.

Unfortunately in C++ this is the default access level. If you do not mark your members with *public* or *protected*, they become *private* by default.

Protected members

All *protected* variables and methods are inaccessible from outside the class, just like private variables. However, subclasses can access *protected* variables and methods.

If you have a variable or method that should not be accessible from outside code, you should mark it as *protected*.

Public members

Public variables and methods are accessible both from outside the class and from subclasses. When a method is *public*, we can call it from outside the class, as we saw done with the `area()` method.

Inheritance

Inheritance is classes standing on the shoulders of each other. If classes are one leg of object oriented programming, inheritance are the other.

Let us continue the rectangle example. I have heard about an exciting new trend called 3D graphics! I really want my *Rect* shape to support this extra dimension. At the same time I already use my existing class *Rect* many places in my code, so I cannot modify it.

My first thought is to just reimplement the class as a 3D shape. Unfortunately my code is open source and I do not want to loose any *street cred* in the community, by not following the *DRY* (Don't Repeat Yourself) paradigm.

It is now inheritance comes to the rescue. We can use it to create a new class *Rect3D* that builds upon the existing *Rect* class, reusing the code and extending the functionality:

```
class Rect3D : public Rect
{
public:
    int z;
    int depth;

    int volume()
    {
        return width * height * depth;
    }
}
```

See the colon at the first line? It defines that our *Rect3D* class inherits from *Rect*. We say that *Rect3D* is a *subclass* of *Rect* and that *Rect* is the *parent class* of *Rect3D*.

The magic here is the variables `x`, `y`, `width` and `height` now exists on `Rect3D` through inheritance. The same is true for the method `area()`.

Inheritance takes all methods and variables (also called properties) and makes them present on subclasses.

The `public` keyword instructs that *public* members on the parent class are still *public* on the subclass.

Unfortunately in C++ the default inheritance access level is *private*. This means you almost always need to declare the inheritance as *public*.

Let's try our new 3D class:

```
Rect3D cube;
cube.x = 10;
cube.y = 10;
cube.z = 10;

cube.width = 75;
cube.height = 75;
cube.depth = 75;

int area = cube.area(); // gives 5625
int volume = cube.volume(); // gives 421875
```

Now we have two classes, where one (`Rect3D`) inherits from the other (`Rect`). Our code is kept *DRY* and existing code that uses `Rect` is not affected by the existence of `Rect3D`.

Overloading methods

Luckily for us, we denote area and volume different. This means that the method `volume()` in `Rect3D`, does not conflict with the existing method `area()`. However, we are not always that lucky.

Say we had added a method that calculated the surface area of the shape. In the two dimensional *Rect* the surface and area are the same, so a `surface()` method is trivial:

```
int surface()
{
    return area();
}
```

Our method simply calls the existing `area()` method and returns its result. But now *Rect3D* inherits this behaviour - which is incorrect in three dimensions.

To get the surface area of a cube we must calculate the area of each side, and sum for all sides. We use *method overloading* to re-declare the same method on `Rect3D`:

```
int surface()
{
    return width * height * 2
        + width * depth * 2
        + height * depth * 2;
}
```

Now both classes declare a method with the same name and arguments. The effect is `Rect3D`'s `surface()` method replaces the method on its *parent class*.

A complete example of the code is:

```
class Rect {
public:
    int width, height;

    int area()
    {
        return width*height;
    }

    int surface()
    {
        return area();
    }
};

class Rect3D : public Rect
{
public:
    int depth;

    int surface()
    {
        return width * height * 2
            + width * depth * 2
            + height * depth * 2;
    }
};
```

Multiple inheritance

Classes in C++ can, as in nature, inherit from more than one parent class. This is called multiple inheritance. Let us exemplify this by breaking up our *Rect* into two classes: *Point* and *Size*:

```
class Point
{
public:
    int x, y;
};

class Size
{
public:
    int width, height;

    int area()
    {
        return width * height;
    }
};
```

If we now combine *Point* and *Size*, we get all the properties needed to represent a *Rect*. Using multiple inheritance we can create a new *Rect* class that build upon both *Point* and *Size*:

```
class Rect : public Point, public Size
{

```



```
};
```

Our new *Rect* class does not define anything on its own, it simply stands on the shoulders of both *Point* and *Size*.

Inbreeding

When using multiple inheritance you should be aware of what is called the *diamond problem*. This occurs when your class inherits from two classes with a common ancestor.

In our geometry example, we could introduce this diamond problem by letting both *Point* and *Size* inherit from a common parent class, say one called: *Shape*.

In C++ there are ways around this issue called *virtual inheritance*, it is an advanced topic though. In this article we will not go into detail about this - you should just know that the problem is solvable.

Constructors

A *constructor* is a special method on a class that gets called automatically when the class is created. Constructors often initialize default values of member variables.

When we develop for embedded systems, we cannot assume variable values are initialized to 0, upon creation. For this reason we want to explicitly set all variables of our *Rect* class to 0:

```
class Rect
{
public:
    int x, y;
    int width, height;

    Rect ()
    {
        x = y = 0;
        width = height = 0;
    }

    int area()
    {
        return width*height;
    }
};
```

Notice the special *constructor* method *Rect ()* has no return type - not even *void*! Now we have created a constructor that sets all member variables to zero, so we ensure they are not random when we create an instance of *Rect*:

```
Rect bounds;
int size = bounds.area(); // gives 0
```

Our constructor is executed upon creation of the *bounds* variable.

When a constructor takes no arguments, as ours, it is called the *default constructor*.

Non-default Constructors

We can declare multiple constructors for our class in C++. Constructors can take parameters, just as functions can.

Let us add another constructor to *Rect* that takes all the member variables as parameters:

```
class Rect
{
public:
    int x, y, width, height;

    // The default constructor
    Rect()
    {
        x = y = width = height = 0;
    }

    // Our special constructor
    Rect(int _x, int _y, int _w, int _h)
    {
        x = _x;
        y = _y;
        width = _w;
        height = _h;
    }

    int area() { return width*height; }
};
```

Now we have a special constructor that initializes a *Rect* object with a provided set of values. Such constructors are very convenient, and make our code less verbose:

```
Rect bounds; // default constructor inits to 0 here
bounds.x = 10;
bounds.y = 10;
bounds.width = 75;
bounds.height = 75;

// now the same can be achieved with a single line
Rect frame(10,10, 75, 75);
```

When you call a special constructor like *Rect(10,10,75,75)* the *default constructor* is *not* executed! In C++ only one constructor can be executed, they can not be daisy chained.

Namespaces

When developing your application you might choose class names that already exists in the system. Say you create a class called *String*, changes are that this name is taken by the system's own *String class*. Indeed this is the case in OpenMono SDK.

To avoid name collisions for common classes such as *Array*, *String*, *Buffer*, *File*, etc. C++ has a feature called *namespaces*.

A *namespace* is a grouping of names, inside a named container. All OpenMono classes provided by the SDK is defined inside a *namespace* called *mono*. You can use double colons to reference classes inside namespaces:

```
mono::String str1;
mono::io::Wifi wifi;
mono::ui::TextLabelView txtLbl;
```

Here we define instances (using the *default constructor*) that are declared inside the namespace *mono* and the sub-namespaces: *io* and *ui*.

Declaring namespaces

So far in this guide, we have only seen classes declared in global space. That is outside any namespace. Say, we want to group all our geometric classes in a namespace called `geo`.

Then `Rect` would be declared as such:

```
namespace geo {
    class Rect
    {
    public:
        int x,y,width,height;

        // ...
    };
}
```

Now, any code inside the `namespace geo { ... }` curly braces can reference the class `Rect` by its name. However, any code outside the namespace must define the namespace as well as the class name: `geo::Rect`.

Namespaces can contains other namespaces. We can create a new namespace inside `geo` called `threeD`. Then, we can rename `Rect3D` to `Rect` and declare it inside the `threeD` namespace:

```
namespace geo {
    namespace threeD {
        class Rect : public geo::Rect
        {
            int z, depth;

            // ...
        };
    }
}
```

The `using` directive

If you are outside a namespace (like `geo`) and often find yourself referencing `geo::Rect`, there is a short cut. C++ offers a `using` directive much like C# does.

The `using` directive imports a namespace into the current context:

```
using namespace geo;

Rect frame;
```

Now you do not have to write `geo::Rect`, just `Rect` - since `geo` has become implicit.

If you look through OpenMono SDK's source code, you will often see these `using` statement at the beginning of header files:

```
using namespace mono;
using namespace mono::ui;
using namespace mono::geo;
```

This reduces the verbosity of the code, by allowing referencing classes without namespace prefixes.

Using a single class

Another less invasive option is to import only a specific class into the current context - not a complete namespace. If you now you are only going to need the `geo::Rect` class and not any other class defined in `geo`, you can:

```
using geo::Rect;

Rect frame;
```

This imports only the *Rect* class. This allows you to keep your context clean.

Tip: On a side note, remember that importing namespaces has no effect on performance. C++ is a compiled language, and namespaces does not exist in binary. You can declare and import as many namespaces as you like - the compiled result is not affected on performance.

References

C++ introduces an alternative to C pointers, called references. If you know C pointers, you are familiar with the `*` syntax. If you don't, just know that in C you can provide a copy of a variable or a pointer to the variable.

In C you denote pointer types with an asterisk (`*`). C++ introduces references denoted by an ampersand (`&`), which are somewhat like pointers.

A reference in C++ is constant pointer to another object. This means a reference cannot be re-assigned. It is assigned upon creation, and cannot be changed later:

```
Rect frame = Rect(0,0,25,25);
Rect& copy = frame;
Rect frm2;
copy = frm2; // AArgh, compiler error here!!
```

The `copy` variable is a reference to `frame` - always. In contrast to pointers in C, you do not have to take the address of an variable to assign the reference. C++ handles this behind the scenes.

Reference in functions

A great place to utilize references in C++ is when defining parameters to functions or methods. Let us declare a new method on `Rect` that check if a `Point` is inside the *rectangles* interior. This method can take a reference to such a point, no reason to copy data back and forth - just pass a reference:

```
class Rect
{
public:
    // rest of decleration left out

    bool contains(const Point &pnt)
    {
        if (    pnt.x > x && pnt.x <= (x + width)
            && pnt.y > y && pnt.y <= (y +height))
            return true;
        else
            return false;
    }
}
```

Our method takes a reference to a *Point* class, as denoted by the ampersand (&). Also, we have declared the reference as `const`. This means we will not modify the `pnt` object.

If we left out the `const` keyword, we are allowed to make changes to `pnt`. By declaring it `const` we are restraining ourselves from being able to modify `pnt`. This help the C++ compiler create more efficient code.

The rule of 3

I shall briefly touch the *Rule of Three* concept, though it is beyond the scope of this article.

When you assign objects in C++ its contents is automatically copied to the destination variable:

```
Rect rct1(5,5,10,10); // special constructor
Rect rct2; // default constructor
rct2 = rct1; // assignment, rct1 is copied to rct2
```

All of *Rect* member variables are automatically copied by C++. This is fine 90% of the time, but there are times when you need or want special behaviour. Often in these cases a advanced behaviour is needed, for example to implement *reference counting* or similar.

As an example here, we just want to modify our *Rect* class to print to the console everytime it is copied.

To achieve this, we must overwrite two implicit defined methods in C++. These are the *copy constructor* and the *assignment operator*.

The Copy Constructor

The *copy constructor* is a special constructor that takes an instance of an object and initializes itself as a copy. C++ calls the *copy constructor* when creating a new variable from an existing one. These are common examples:

```
Rect frame(0,0,100,100); // special constructor
Rect frame2 = frame; // copy constructor
someFunction(frame); // copy constructor again
```

When we create a new instance by assigning an existing object the *copy constructor* is used. Further, if we have a function or method and takes a class type as parameter, the function is provided with a fresh copy of the object by the *copy constructor*.

To create your own copy constructor you define it like this:

```
class Rect
{
public:
    // copy constructor
    Rect(const Rect &other)
    {
        x = other.x;
        y = other.y;
        width = other.width;
        height = other.height;

        printf("Hello from copy constructor");
    }
};
```

We left out the other constructors, and members in this example. The *copy constructor* is a constructor method that takes a `const` reference to another instance of its class.

In the *Rect* class we copy all variables (the default behaviour of C++, if we had not defined any *copy constructor*) and prints a line to the console.

This serves to demonstrate that you can define exactly what it means to assign your class to a new variable. You can make your new object a shallow copy of the original or change some shared state.

The Assignment Operator

There is still the other case: *assignment operator*. It is where the default *assignment operator* is used. The default *assignment operator* occurs when:

```
Rect view(10,10,100,100); // convenient constructor
Rect bounds; // default constructor
bounds = view; // assignment operator
```

Here we create a instance with some rectangle `view`, and a zeroed instance `bounds`. If we want the same behaviour as with the *copy constructor*, we need to declare the *assignment operator* on *Rect*:

```
class Rect
{
public:
    // rest of class content left out

    // assignment operator
    Rect& operator=(const Rect &rhs)
    {
        x = rhs.x;
        y = rhs.y;
        width = rhs.width;
        height = rhs.height;

        printf("Hello from assignment operator");
        return *this;
    }
};
```

Just like the *copy constructor*, the assignment operator takes a `const` reference to the object that need to be assigned (copied). But its assignment must also return a reference of itself, as defined by `Rect&`. This is also why we have to include the `return *this` statement. In C++ this is a pointer to the instance of the class - the object itself.

A C pointer juggling champ, will recognize that we dereference the pointer by adding the asterisk (*) in front.

Just as is the case with the *copy constructor*, we can now define the assignment behavior of *Rect*. Here (again), it is illustrated by printing to the console upon assignment.

The Destructor

This is the last part of the *Rule of Three* in C++.

The *destructor* is the inverse of the constructor - it is called when an object dies or rather - is deallocated. Objects get deallocated when they go out of scop. As is the case when a function or method returns.

To follow our previous examples we want the *destructor* to just print to the console.

```

class Rect
{
public:
    // rest of class content is left out

    //the destructor
    ~Rect ()
    {
        printf("Hello from the de-structor");
    }
};

```

The *destructor* is defined as the class' name with a tilde (~) in front. A destructor takes no arguments.

Now this is the rule of three. Defining the:

- *copy constructor* : `Class (const Class &other)`
- *assignment operator* : `Class& operator=(const CClass &rhs)`
- *destructor* : `~Class ()`

When you create your own C++ classes, think about these three. Mostly you don't have to implement them, but in some cases you will.

Further reading

C++ is an admittedly an avanced and tough language. I still bang my head against the wall sometimes. More modern languages like Java and C# are easier to use, but in an embedded environment we don't have the luxury of a large runtime. So for now, we are left with C++.

There are still a ton of topics that I did not cover here. I'd like to encourage you to read about these more advanced topics:

Bare Metal C++: A Practical Guide

*If you what to be an embedded coding champ, you should really read Alex Robenko's book: **Practical Guide to Bare Metal C++ (and I mean: really!)***

Alex' book goes through very interesting topics of getting C++ runnning on embedded devices. It covers important shortcomings and advantages of C++ in an embedded environment.

If you know C++ you might want to use *Exceptions* and *RTTI* features, before you do: Read the book! In contrast, if you do not know C++ you might (will) make mistakes that can take hours to recover from. Again: Read the book!

Here is a short list of most interesting chapters of the book:

- Dynamic Memory Allocation
- Exceptions
- RTTI
- Removing Standard library
- Static objects
- Abstract classes
- Templates

- Event loops

As a Mono developer you will face most of these topics.

Read the Book

- Queued callbacks and interrupts
- Touch System Architecture
- Wifi & networking
- Boot and Startup procedures
- Power Management Overview
- Memory Management: Stack vs heap objects?
- The Build System

Schematics

This is the schematics for our hardware. These are meant to help you develop our own extensions to Mono, go create!

If you need more specific schematics than what is shown here, please consider posting a request on our [community](#).

Mono (Maker + Basic)



If you have a Mono Basic, it is the same as Maker, just without the Wifi / Bluetooth component mounted.

Mono Shield Adapter



MonoKiosk

Mono apps are distributed through the [Kiosk](#), and you can get your app into the Kiosk by following the recipe below.

GitHub

If your source code is hosted on [GitHub](#), you will need to make a [GitHub release](#) and attach three types of files to the release, namely

- The app description.
- A set of screenshots.

- The binary app itself.

App description

The release must contain a file named `app.json` that contains the metadata about your app, for example

```
{
  "id": "com.openmono.tictactoe"
  , "name": "Tic Tac Toe"
  , "author": "Jens Peter Secher"
  , "authorwebsite": "http://developer.openmono.com"
  , "license": "MIT"
  , "headline": "The classic 3x3 board game."
  , "description":
    [ "Play with a fun and exciting game with against another player."
    , "Can you beat your best friend in the most classic of board games?"
    ]
  , "binary": "ttt.elf"
  , "sourceurl": "https://github.com/getopenmono/ttt"
  , "required": ["display", "touch"]
  , "optional": []
  , "screenshots":
    [ "tic-tac-toe-part1.png"
    , "tic-tac-toe-part2.png"
    , "tic-tac-toe-part3.png"
    ]
  , "cover": "tic-tac-toe-part2.png"
  , "kioskapi": 1
}
```

As you can see, `app.json` refers to three distinct images (`tic-tac-toe-part1.png`, `tic-tac-toe-part2.png`, `tic-tac-toe-part3.png`) to be used on the app's page in MonoKiosk, so these three files must also be attached to the GitHub release. The metadata also refers to the app itself (`ttt.elf`), the result of you building the application, so that file must also be attached to the release.

The format of the metadata needs to be very strict, because it is used to automatically create an entry for your app in MonoKiosk. The metadata must be in **JSON** format, and the file must be named `app.json`. In the following, we will describe the format in detail.

id

The `id` must be unique within the Kiosk, so you should use **reverse domain name notation** like `uk.homebrewers.brewcenter`.

name

The name of the app as it should appear to people browsing the Kiosk.

author

Your name or Organisation, as it should appear to people browsing the Kiosk.

authorwebsite

An *optional* URL to your (organisation's) website.

license

How other people can use your app and the source code. We acknowledges the following licenses:

If you feel that you need another license supported, take it up in the [community](#).

headline

Your headline that accompanies the app on the Kiosk.

description

A list of paragraphs that give other people a detailed desription of the app, such as why they would need it and what it does.

binary

The name of the [ELF](#) file which has been produced by your compiler, and which you have attached to the release.

sourceurl

An URL to the source code of the app.

required

A list of hardware that must be present in a particular Mono to run the app. The acknowledged hardware is as follows.

- accelerometer
- buzzer
- clock
- display
- jack
- temperature
- touch
- wifi
- bluetooth

optional

A list of Mono hardware that the app will make use of if present. The acknowledged hardware is the same as for the required list.

screenshots

A list of images that will be presented in the Kiosk alongside the app description.

All images must be either 176x220 or 220x176 pixels, and they must be attached to the release.

cover

One of the screenshots that you want as cover for app in the Kiosk.

kioskapi

The format of the metadata. The format described here is version 1.

How to get your app included

When you have created a new (version) of your app, you can contact us at `kiosk@openmono.com` with the URL of your release (eg. `https://api.github.com/repos/getopenmono/ttt/releases/tags/v0.1.0`), and we will do a sanity check of the app and add to the official list used by the Kiosk.

For GitHub, the url for a release is `https://api.github.com/repos/:owner/:repo/releases/tags/:tag`

Datasheets

If you need to dive a little deeper into the inner workings of Mono, we have collected the datasheets for the components in Mono

You might need to consult specific datasheets for the components in Mono, if you are debugging or just need some advanced features not provided by the API.

Accelerometer

Mono's accelerometer is a *MMA8652FC* chip by Freescale. It is connected to Mono's I2C bus.

The accelerometer is handled by the `MonoAccelerometer` class in the software framework. If you need specific features, or just wish to play with the component directly, you should consult the datasheet.

MCU

Mono's Micro Controller Unit (MCU) is a Cypress PSoC5LP, that is an Arm Cortex-M3 CPU. You can use all its registers and functions for your application, the SDK includes headers for all pins and registers. (You must explicitly include the `project.h` file.)

The MCU model we use has 64 Kb SRAM, 256 Kb Flash RAM and runs at 66 Mhz.

The software framework encapsulates most MCU features in the *mbed* layer, such as GPIO, interrupts and timers. Power modes is also controlled by the registers in the MCU and utilized by the `PowerManagement` class.

Display Chip

The display is driven by an *ILITEK 9225G* chip. On mono we have hardwired the interface to 16 bit 5-6-5 color space and the data transfer to be 9-bit dedicated SPI, where the 9th bit selects data/command registers. (This should make sense, when you study the datasheet.)

In the framework the display controller class `ILI9225G` utilizes the communication and pixel blitting to the display chip.

Wireless

Mono uses the Redpine Wifi chip to achieve wireless communication. (The same chip includes Bluetooth for the Maker model, also.) The chip is connected via a dedicated SPI interface, and has a interrupt line connected as well.

The communication interface is quite advanced, including many data package layers. You can find our implementation of the communication in the `ModuleSPICommunication` class. This class utilizes the SPI communication from and to the module, it does not know anything about the semantics of the commands sent.

Temperature Sensor

The temperature sensor is an Amtel *AT30TS74* chip, connected via the internal I2C bus.

The temperature interface is used in the `AT30TS74Temperature` class.

API Reference

Common Utilities

DateTime

class A Date and time representation in the Gregorian calendar. This class represents a point in time, defined in the gregorian calendar. Such a timestamp are given in year, month, day of month, hours since midnight, minutes and seconds. This class also defined if the timestamp is in UTC / GMT or a defined local time zone. The class handles leap years and the varying length of months. You can add seconds, minutes, hours and days to a *DateTime* object and get the result as a new *DateTime* object. When you create *DateTime* objects they are created in the local time zone by default. The local time zone is defined as a offset in hours relative to UTC. There is no notion of IANA Time Zone names of alike - just an offset to the UTC time. There are two convenient method to print *DateTime* as readable strings. The *toString* method print a human readable (MySQL compatible) timestamp. The other *toISO8601* returns a string formatted in the ISO 8601 standard format used in JSON objects. When printing *DateTime* objects, they are returned in the time zone that they are created in. *System Wall Clock* This class also has a global *DateTime* object reserved for use by a RTC feature. A subsystem manages the RTC and increments the global system *DateTime* object. You can get the current *DateTime* time by using the static method *now* To set the system clock use the static method *setSystemClock* **Public Types**

enum type `mono::DateTime::TimeTypes`

DateTime timestamps can be one of three types

Values:

The *DateTime* is specified in local time zone

The *DateTime* is specified in UTC / GMT time zone

The *DateTime* do not have a specified time zone

Public Functions

`DateTime::DateTime()`

Construct an empty / invalid *DateTime* object.

`DateTime::DateTime(const time_t t, bool localTimeZone)`

Create a *DateTime* from the a libc simple time.

This constructor takes a standard libc time variable and create a calendar *DateTime* object from that. It uses the systems timezone, and libc systems time stamp to calendar conversion.

You can set the optional argument `localTimeZone` to `false` to force the *DateTime* to be created in UTC time, instead of system time.

Times is converted to a date in the Gregorian calendar.

Parameters

- `t` - The libc simple time stamp
- `localTimeZone` - Optional: Use the systems timezone in conversion

`DateTime::DateTime(const tm *brokentime, bool localTimeZone)`

Create a *DateTime* from the a libc broken-down time struct.

This constructor takes a standard libc broken-downtime variable and creates a calendar *DateTime* object from that. It uses the timezone present in libc structure.

You can set the optional argument `localTimeZone` to `false` to force the *DateTime* to be created in UTC time, instead of system time.

Parameters

- `brokentime` - The libc simple broken-down calendar time
- `localTimeZone` - Optional: Use the systems timezone in conversion

`DateTime::DateTime(uint16_t years, uint8_t months, uint8_t days, uint8_t hours, uint8_t minutes, uint8_t seconds, TimeTypes zone)`

Construct a *DateTime* object with a given date and time.

Parameters

- `years` - The Year component of the date, for example 2016
- `months` - The month component of the date from 1 to 12, May is 5
- `days` - The day component of the date, 1-indexed, from 1 to 31
- `hours` - Optional: The hour component of the timestamp, range is 0 to 23
- `minutes` - The minute component of the timestamp, range is 0 to 59
- `seconds` - The seconds component of the timestamp, range is 0 to 59
- `zone` - The timezone where this *DateTime* define its time, default is the local timezone

`String DateTime::toString()`
constReturn the *DateTime* object as a huamn readable string.

Return

A mono string on the format: yyyy-MM-dd hh:mm:ss

`String DateTime::toISO8601()`
constReturn an ISO8601 formatted timestamp as a string.

This returned string is on the format: yyyy-MM-ddTHH:mm:ss+tt:00 if not UTC or yyyy-MM-ddTHH:mm:ssZ

`String DateTime::toRFC1123()`
constGet the *DateTime* as an RFC1123 compatible date/time string.

This is the standard used by the HTTP standard and therefore used in HTTP headers like *Date* and *Last-Modified*. Use this method to format a *DateTime* to this representation.

`String DateTime::toTimeString()`
constReturn only a time string from the *DateTime*.

The format is: HH:mm:ss

`String DateTime::toDateString()`
constReturn only a date string from the *DateTime*.

The format is: yyyy-MM-dd

`uint32_t DateTime::toJulianDayNumber()`
constReturn Julian day number of the *DateTime*.

`uint32_t DateTime::toUnixTime()`
constReturn Unix time of the *DateTime*.

`struct tm DateTime::toBrokenDownUnixTime()`
constReturn a libc broken-down time/date compnents.

This uses the default timezone of the local systems time zone

`time_t DateTime::toLibcUnixTime()`
constReturns the libc style timestamp (simple time)

`bool DateTime::isValid()`
constReturn `true` if the *DateTime* is valid.

Invalid date object is conctructed by the default constructor

`DateTime DateTime::toUtcTime()`
constConvert this *DateTime* to UTC time.

`uint8_t DateTime::Hours()`
constGet the hour component, from 0 to 23.

`uint8_t DateTime::Minutes()`
constGet the Minute component, from 0 to 59.

`uint8_t DateTime::Seconds()`
constGet the Seconds component, from 0 to 59.

`uint8_t DateTime::Days()`

constGet the day of month component, first day of month is 1.

`uint8_t DateTime::Month()`

constGet the month component, January is 1, December 12.

`uint16_t DateTime::Year()`

constGet the year component.

`DateTime DateTime::addSeconds (int seconds)`

constReturn a new object with a number of seconds added.

This method increments the timestamp for the given second interval

Return

The new *DateTime* object with seconds added

Parameters

- `seconds` - The seconds to add

`DateTime DateTime::addMinutes (int minutes)`

constReturn a new object with a number of minutes added.

This method increments the timestamp for the given minute interval

Return

The new *DateTime* object with minutes added

Parameters

- `minutes` - The minutes to add

`DateTime DateTime::addHours (int hours)`

constReturn a new object with a number of hours added.

This method increments the timestamp for the given hour interval

Return

The new *DateTime* object with hours added

Parameters

- `hours` - The hours to add

`DateTime DateTime::addDays (int days)`

constReturn a new object with a number of days added.

This method increments the timestamp for the given day interval

Return

The new *DateTime* object with days added

Parameters

- `days` - The days to add

`DateTime DateTime::addTime (const time_t *t)`

constReturn a new object with the Unix simple time component added.

The provided pointer to a unix time is added to the existing *DateTime* and the result is returned as a new object.

Return

The new *DateTime* object with the time interval added

Parameters

- `t` - The unix simple time to add

String *DateTime* : **toString** (const char **format*)

constFormat the *DateTime* to a string, using *libc* `strftime` function.

Use this method to get a custom representation of the date/time as text. You control the format of the output using the string at format. It can contain two kinds of specifications: text to be copied literally into the formatted string, and time conversion specifications. Time conversion specifications are two- and three-character sequences beginning with “ (use ‘%’ to include a percent sign in the output). Each defined conversion specification selects only the specified field(s) of calendar time data from *timp, and converts it to a string in one of the following ways:

a The abbreviated weekday name according to the current locale. [tm_wday]

A The full weekday name according to the current locale. In the default “C” locale, one of ‘Sunday’, ‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’, ‘Saturday’. [tm_wday]

b The abbreviated month name according to the current locale. [tm_mon]

B The full month name according to the current locale. In the default “C” locale, one of ‘January’, ‘February’, ‘March’, ‘April’, ‘May’, ‘June’, ‘July’, ‘August’, ‘September’, ‘October’, ‘November’, ‘December’. [tm_mon]

c The preferred date and time representation for the current locale. [tm_sec, tm_min, tm_hour, tm_mday, tm_mon, tm_year, tm_wday]

C The century, that is, the year divided by 100 then truncated. For 4-digit years, the result is zero-padded and exactly two characters; but for other years, there may a negative sign or more digits. In this way, ‘Cy’ is equivalent to ‘Y’. [tm_year]

d The day of the month, formatted with two digits (from ‘01’ to ‘31’). [tm_mday]

D A string representing the date, in the form “'%m/%d/%y'”. [tm_mday, tm_mon, tm_year]

e The day of the month, formatted with leading space if single digit (from ‘1’ to ‘31’). [tm_mday]

Ex In some locales, the E modifier selects alternative representations of certain modifiers x. In newlib, it is ignored, and treated as x.

F A string representing the ISO 8601:2000 date format, in the form “'%Y-%m-%d'”. [tm_mday, tm_mon, tm_year]

g The last two digits of the week-based year, see specifier G (from ‘00’ to ‘99’). [tm_year, tm_wday, tm_yday]

G The week-based year. In the ISO 8601:2000 calendar, week 1 of the year includes January 4th, and begin on Mondays. Therefore, if January 1st, 2nd, or 3rd falls on a Sunday, that day and earlier belong to the last week of the previous year; and if December 29th, 30th, or 31st falls on Monday, that day and later belong to week 1 of the next year. For consistency with Y, it always has at least four characters. Example: “%G” for Saturday 2nd January 1999 gives “1998”, and for Tuesday 30th December 1997 gives “1998”. [tm_year, tm_wday, tm_yday]

h Synonym for “%b”. [tm_mon]

H The hour (on a 24-hour clock), formatted with two digits (from ‘00’ to ‘23’). [tm_hour]

I The hour (on a 12-hour clock), formatted with two digits (from ‘01’ to ‘12’). [tm_hour]

- j The count of days in the year, formatted with three digits (from '001' to '366'). [tm_yday]
- k The hour (on a 24-hour clock), formatted with leading space if single digit (from '0' to '23'). Non-POSIX extension (c.p. I). [tm_hour]
- l The hour (on a 12-hour clock), formatted with leading space if single digit (from '1' to '12'). Non-POSIX extension (c.p. H). [tm_hour]
- m The month number, formatted with two digits (from '01' to '12'). [tm_mon]
- M The minute, formatted with two digits (from '00' to '59'). [tm_min]
- n A newline character (' ').
- Ox In some locales, the O modifier selects alternative digit characters for certain modifiers x. In newlib, it is ignored, and treated as x.
- p Either 'AM' or 'PM' as appropriate, or the corresponding strings for the current locale. [tm_hour]
- P Same as 'p', but in lowercase. This is a GNU extension. [tm_hour]
- r Replaced by the time in a.m. and p.m. notation. In the "C" locale this is equivalent to "%I:%M:%S %p". In locales which don't define a.m./p.m. notations, the result is an empty string. [tm_sec, tm_min, tm_hour]
- R The 24-hour time, to the minute. Equivalent to "%H:%M". [tm_min, tm_hour]
- S The second, formatted with two digits (from '00' to '60'). The value 60 accounts for the occasional leap second. [tm_sec]
- t A tab character (' ').
- T The 24-hour time, to the second. Equivalent to "%H:%M:%S". [tm_sec, tm_min, tm_hour]
- u The weekday as a number, 1-based from Monday (from '1' to '7'). [tm_wday]
- U The week number, where weeks start on Sunday, week 1 contains the first Sunday in a year, and earlier days are in week 0. Formatted with two digits (from '00' to '53'). See also W. [tm_wday, tm_yday]
- V The week number, where weeks start on Monday, week 1 contains January 4th, and earlier days are in the previous year. Formatted with two digits (from '01' to '53'). See also G. [tm_year, tm_wday, tm_yday]
- w The weekday as a number, 0-based from Sunday (from '0' to '6'). [tm_wday]
- W The week number, where weeks start on Monday, week 1 contains the first Monday in a year, and earlier days are in week 0. Formatted with two digits (from '00' to '53'). [tm_wday, tm_yday]
- x Replaced by the preferred date representation in the current locale. In the "C" locale this is equivalent to "%m/%d/%y". [tm_mon, tm_mday, tm_year]
- X Replaced by the preferred time representation in the current locale. In the "C" locale this is equivalent to "%H:%M:%S". [tm_sec, tm_min, tm_hour]
- y The last two digits of the year (from '00' to '99'). tm_year
- Y The full year, equivalent to Cy. It will always have at least four characters, but may have more. The year is accurate even when tm_year added to the offset of 1900 overflows an int. [tm_year]
- z The offset from UTC. The format consists of a sign (negative is west of Greenwich), two characters for hour, then two characters for minutes (-hhmm or +hhmm). If tm_isdst is negative, the offset is unknown and no output is generated; if it is zero, the offset is the standard offset for the current time zone; and if it is positive, the offset is the daylight savings offset for the current timezone. The offset is determined from the TZ environment variable, as if by calling tzset(). [tm_isdst]
- Z The time zone name. If tm_isdst is negative, no output is generated. Otherwise, the time zone name is based on the TZ environment variable, as if by calling tzset(). [tm_isdst]

%% A single character, ‘.’.

Public Static Functions

static *DateTime* **mono::DateTime::maxValue()**
Get the maximum possible *DateTime* value (far in the future)

static *DateTime* **mono::DateTime::minValue()**
Get the lowest possible *DateTime* value (the distant past)

bool *DateTime::isLeapYear* (uint16_t year)
Check is a year is a leap year.

DateTime **DateTime::fromISO8601** (String date)
Parse a subset of ISO 8601 compatible date time representations.

This static method takes a ISO 8601 formatted string, and creates a *DateTime* object from that. This method only parses a subset of the possible date representations allowed in ISO 8601. Specifically it can handle dates in these format:

- yyyy-MM-ddTHH:mm:ssZ
- yyyy-MM-ddTHH:mm:ss+01:00 or other time zones
- yyyy-MM-dd HH:mm:ssZ
- yyyy-MM-dd HH:mm:ss

Return

The parsed *DateTime* object, that might be valid or invalid

void *DateTime::setSystemDateTime* (*DateTime* dt)
Set a new system *DateTime*.

DateTime **DateTime::now** ()
Get the current *DateTime* from the system RTC clock.

void *DateTime::incrementSystemClock* ()
Internal method used by the RTC system to increment the system *DateTime*. You should not call this manually.

Public Static Attributes

int *DateTime::LocalTimeZoneHourOffset*
The systems current *TimeZone* setting.
The timezone is just an hour-offset from the UTC / GMT time

GenericQueue

template <typename *Item*>
class A templated *Queue*, where template defines the queue element type.
This class is identical to *Queue*, but it uses templating to preserve type information.

This is a basic single linked FIFO queue structure. All items in the queue *must* implement the `QueueItem` interface.

Item data types

Items added to the queue *must* inherit from the interface `QueueItem`. This is required because this interface defined the queues *next pointer*.

Should you try to construct a `GenericQueue` with non-`QueueItem` based template, you will get a compiler warning.

Complexity

As with standard queue data types, enqueueing and dequeueing items are constant time ($O(1)$). However removing an element inside the queue is linear ($O(n)$), same for getting the current length or size of the queue.

See

`QueueItem`

PowerSaver

class Auto dim the display and sleep Mono after a given period of inactivity. This class will automatically dim the display and enter sleep mode, after a period of user inactivity. The uses 3 states: Normal mode (full brightness on display) After a period of no user touch input (default 10 secs.), display dims to 11% After yet a period (default 10 secs.) of no touch input, sleep mode is entered.

Any touch input will reset the state to *1*.

You should add this object to your `AppController` class, to enable automatic power saving.

Multiple PowerSavers

If you want to use multiple instances of this class, remember to *deactivate* the instances that should be inactive. Having two active instances at one time, can be confusing to the users.

Catch-all Touches

TO capture all touches, any instane should be the first to respond to touch inputs. The touch system has a responder chain of objects that handles touch input. Any of these objects might choose to break the chain and allow no further processing of a given touch event.

Therefore the `PowerSaver` must be the first (or one of) to handle touch events. You accomplish this by initializing the `PowerSaver` object as the first member object of your `AppController`. Before any other touch related classes.

Public Functions

PowerSaver::PowerSaver (int *dimTimeoutMs*, int *sleepTimeoutMs*, int *dimBrightness*, int *full-Brightness*)

Construct a auto dimmer and sleep timer.

Parameters

- *dimTimeoutMs* - The delay before display is dimmed (in milliseconds)
- *sleepTimeoutMs* - The delay before sleep mode is entered (in milliseconds)
- *dimBrightness* - The brightness level when display is dimmed (0 to 255)
- *fullBrightness* - The full brightness level (normal mode) (0 to 255)

void `PowerSaver::startDimTimer()`

Starts the timer that will dim the display after the chosen timeout.

This call stops any running sleep timer. After the dim timer fires, the sleep timer is automatically started.

void `PowerSaver::startSleepTimer()`

Starts the sleep timer, that will sleep Mono after the chosen timeout.

This will stop a running *dim timer*, and trigger sleep mode on timeout.

void `PowerSaver::dim()`

Immediately dim the display and then start the sleep timer.

This will (asynchronously) dim the display to the chosen brightness level. When the display has been dimmed, the sleep timer is started.

void `PowerSaver::undim()`

Immediately undim the display to full brightness.

This will stop the sleep timer and re-start the dim timer.

void `PowerSaver::deactivate()`

Disable the *PowerSaver* all together.

No dimming or sleep is triggered. You must call *startDimTimer* to re-enabled the *PowerSaver*.

Queue

class A pointer based FIFO style *Queue*. ****Note:** You should avoid using this *Queue* class, and consider its template based counter part: *GenericQueue* ****** This is a basic single linked FIFO queue structure. All items in the queue *must* implement the *QueueItem* interface. In theory you can add different types into the queue, as long as they all inherit *QueueItem*. However, you should *not* do this. Mixing different types in the queue, removes type information - such that the objects original type can not be restored later. Best practice is therefore to use only one type in the queue. To help with this you should really use the *GenericQueue* class. This class uses C++ templating to keep type information and you do not need to manual type casting. *GenericQueue*

Public Functions

~~See~~ `Queue::Queue()`

Construct an empty queue.

void `mono::Queue::enqueue(IQueueItem *item)`

Add a new element to the back of the queue Insert a pointer to an element on the back of the queue.

IQueueItem *`mono::Queue::dequeue()`

Returns and removes the oldest element in the queue.

IQueueItem *`mono::Queue::peek()`

Return the oldest element in the queue, without removing it.

IQueueItem *`mono::Queue::next(IQueueItem *item)`

Get the next element in the queue, after the one you provide.

NOTE: There is no check if the item belongs in the parent queue at all!

Return

The next element in the queue, after the item you provided.

Parameters

- `item` - A pointer to an item in the queue

`bool mono::Queue::exists (IQueueItem *item)`

Check that an object already exists in the queue. Because of the stack based nature of this queue, all objects can only exist one replace in the queue. You cannot add the same object to two different positions in the queue.

Parameters

- `item` - The element to search for in the queue

`uint16_t Queue::Length ()`

Return the length of the queue.

The length is the number of item currently present in the queue.

This method runs in O(n) (linear time)

IQueueItem

class An interface for object that can be put into a *Queue*. This interface defines the *next queue item* pointer on the sub-classes. This pointer is used by the *Queue* class to realize the queue data structure. Any object you wish to insert into a queue must inherit from this interface. *GenericQueue*

Regex

class This class is a C++ wrapper around the C library called SLRE (Super Lightweight Regular Expressions)
Example

See `Regex::Capure caps[3];`
`Regex reg("(..) (..) (..)");`
`bool success = reg.Match("test my regex", caps, 3);`
`if (success) {`
`String cap1 = reg.Value(caps[0]);`
`}`

Pattern syntax

- `(?i)` Must be at the beginning of the regex. Makes match case-insensitive
- `^` Match beginning of a buffer
- `$` Match end of a buffer
- `()` Grouping and substring capturing
- `\s` Match whitespace
- `\S` Match non-whitespace
- `\d` Match decimal digit
- `\n` Match new line character
- `\r` Match line feed character
- `\f` Match form feed character
- `\v` Match vertical tab character

- `\t` Match horizontal tab character
- `\b` Match backspace character
- `+` Match one or more times (greedy)
- `+`? Match one or more times (non-greedy)
- `*` Match zero or more times (greedy)
- `*`? Match zero or more times (non-greedy)
- `?` Match zero or once (non-greedy)
- `x|y` Match x or y (alternation operator)
- `\meta` Match one of the meta character: `^$().[*+?\\`
- `\xHH` Match byte with hex value 0xHH, e.g.
- `[. . .]` Match any character from set. Ranges like `[a-z]` are supported
- `[^ . . .]` Match any character but ones from set

<https://github.com/cesanta/slr>

Public Types

typedef *Regex* Match capture object holding the first match capture*Regex*

Public Functions

Regex : **Regex** (String *pattern*)

Create a regular expression object from a pattern string

bool **Regex** : **IsMatch** (String *matchStr*)

Test if a string matches the regex pattern

Return

true on match, false otherwise

bool **Regex** : **Match** (String *matchStr*, Capture **captureArray*, uint32_t *capArraySize*)

Get a the first capture group match from a string

The *Regex* class does not allocate any capure objects, so you must supply all needed objects for captures.

Return

true on match, false otherwise

Parameters

- *matchStr* - The string to match against the regex pattern
- *captureArray* - A pointer to a array of Capture objects
- *capArraySize* - The size of the provided capture array

String **Regex** : **Value** (Capture &*cap*)

Return the string value from a match capture object

String

class High level string class. The mono framework has it own string class, that either reside on the HEAP or inside the read-only data segment (`.rodata`). We use this string class to pass string data to async routines like the View 's `scheduleRepaint` method. Because views might be repainted at any point in time, we cannot have view data reside on the stack. This string class hold its data on the HEAP, but behaves as it would reside on the stack. This string class takes care of all alloc and dealloc of memory. It is a referenced based string class. You should not pass pointers of C++ references to this class, but instead normal assignment or pass the full class to functions. The efficient copy / assignment operator methods on the class ensure only data references are passed, behind the scenes. For example:

```
String str = String::Format("Hello World, number: %i", 1);
String str2 = str;
String str3 = str2;
```

In the code only 1 copy of the string data is present in memory. And only references are passed to the objects `str2` and `str3`. Only as the last object is deallocated is the data disposed from the HEAP.

These features makes the class very lightweight and safe to pass around functions and objects.

Public Functions

`String::String()`

Construct an empty invalid string.

`String::String (uint32_t preAllocBytes)`

Construct an empty string with a pre-allocated size.

Use this constructor to created a buffer-like string object.

Parameters

- `preAllocBytes` - The number of bytes to allocate in the string object

`String::String (char *str, uint32_t length)`

Construct a string from an existing C string, with a fixed length.

The string data is copied to the mono string, such that *String* has it own copy of the string data.

Parameters

- `str` - Pointer to the original C string
- `length` - The length of the provided string, without NULL terminator

`String::String (char *str)`

Construct a mono *String* from an existing NULL terminated C string.

The length of the source string is determined by the Clib function `strlen`.

Parameters

- `str` - A pointer to the C style string

`String::String (const char *str)`

Construct a mono *String* from an existing NULL terminated C string.

The length of the source string is determined by the Clib function `strlen`.

Parameters

- `str` - A pointer to the C style string

`String::String (const String &str)`

Construct a mono *String* from an existing mono string.

The length of the source string is determined by the Clib function `strlen`.

Parameters

- `str` - A reference to the mono string

`uint32_t String::Length ()`

constReturn the length of the string.

The length of the actual string is returned, not counting the NULL terminator.

This method uses `strlen` which does not support variable byte-length character encoding. (That means UTF8, 16 and alike.)

`char String::operator [] (uint32_t pos)`

constAccess each character (or byte) of the string in an array like fashion.

`char *String::operator () ()`

constShort-hand for *CString*.

See

CString

`char *String::CString ()`

constGet a pointer to the raw C style string.

Use this method if need to print a string using `printf` or alike.

To obtain a pointer to the raw string data.

Public Static Functions

`String String::Format (const char *format, ...)`

C lib. style format string.

Construct a mono string using the *string format* syntax.

Return

String The resulting string, interpolated with provided parameters

Parameters

- `format` - The formatted string with placeholders

ScheduledTask

class Schedule function a call at a point in the future using the RTC. A *ScheduledTask* is a wrapper around a function call that is scheduled some time in the future. You create a task by defining a function to be called, and a *DateTime* on when it should be called. A *ScheduledTask* differs from *Timer* by not being repetitive by nature. Also, the time resolution on is seconds, not milliseconds as on *Timers*. An advantage of scheduled tasks is they can execute in sleep mode. They have the ability to wake Mono from sleep, either completely or just to execute

simple house keeping code inside the sleep mode state! Scheduled tasks rely on the RTC system to execute. They use the global system *DateTime* object to execute when they are due. A *ScheduledTask* executes once. If you wish to re-schedule the task at a new point in the future, use the method *reschedule*. *Execution in Sleep Mode* You can opt-in to allow your task to run in sleep mode. This means Mono's CPU wake up and executes your task's callback function. You enabled sleep mode execution by:

```
ScheduledTask task(DateTime::now().addMinutes(30));
task.setRunInSleep(true);
task.setTask<MyClass>(this, &MyClass::myCallback);
```

The function is executed without exiting the frameworks sleep state. This means the auxillary power is not enabled. (Display, Wireless chip and sensors are off.) Inside you callback function you can do small lightweight tasks, and when your function returns Mono re-enter the sleep state.

Wake-up from a task

If you want to wake Mono from sleep, you must indicate to the IPowerManager that sleep mode should be exited. This is done by setting the flag:

```
IApplicationContext::Instance->PowerManager->__shouldWakeUp = true;
```

This will trigger a full wake-up, just as toggling the user button does.

RTC must run

Scheduled tasks *will not* work if the RTC system is not started. This system is automatically started on reset, so the system is enabled by default.

Just like *Timer* you are not guaranteed to have you function executed at the exact moment defined by the *DateTime* provided. The guarantee is that your function will not be executed *before* the provided time stamp.

Public Functions

ScheduledTask::ScheduledTask()

Construct an empty (non-executable) Task.

You must provide a callback function and reschedule the task using the accessors in subsequent calls.

ScheduledTask::ScheduledTask(const DateTime &scheduledTime)

Construct a task that is due at a provided time stamp.

Create a *ScheduledTask* that is due on the given *DateTime*, which must be in the future. You must also provide the callback function using the *setTask* method.

Parameters

- *scheduledTime* - The point in the future where the task should run

void ScheduledTask::reschedule(const DateTime &newTime)

Reschedules the task at a new point in the future.

Tasks only executes once. To enable re-occurring tasks, use this method to schedule the task again.

Parameters

- *newTime* - The new point in the future where the task is run again

bool ScheduledTask::willRunInSleep()

constGet is this task will execute in sleep mode.

`void ScheduledTask::setRunInSleep (bool run)`

Allow this task to run in sleep mode.

If you set this to `true` the task will execute in sleep mode. Note that it does not abort sleep (wake up), it execute inside the sleep mode.

Task that do not run in sleep are executed ASAP after wake-up.

Parameters

- `run` - `true` if the task can run in sleep, `false` otherwise

`template <typename Class>`

`void mono::ScheduledTask::setTask (Class * context, void (Class::*) (void) memptr)`

Set the tasks callback function.

Here you provide the function that is called when the task is scheduled

Parameters

- `context` - The `this` pointer or object instance
- `memptr` - A pointer to a member method on a class

Public Static Functions

`void ScheduledTask::processScheduledTasks (bool isSleeping)`

System routine of run any due scheduled task.

the RTC system automatically calls this method, you should not call it yourself.

`bool ScheduledTask::pendingScheduledTasks (bool inSleep)`

Returns `true` if there are scheduling tasks pending for processing.

This means a running task has timed out and are ready to have its handler called. tasks with no callback handler are not regarded as pending.

Parameters

- `inSleep` - If this static method is called from inside sleep mode, set to `true`

Protected Functions

`void ScheduledTask::runTask (bool inSleep)`

Execute the tasks callback function

`bool ScheduledTask::isDue ()`

constReturn `true` if the tasks time stamp has been reached

Protected Static Attributes

`GenericQueue<ScheduledTask> ScheduledTask::queue`

The global singleton queue of pending tasks

Timer

class A queued *Timer* class, recurring or single shot. A timer can call a function at regular intervals or after a defined delay. You can use the timer to do periodic tasks, like house-keeping functions or display updates. *Queued callback* The timer uses the Application Run Loop to schedule the callback handler function. This means your callback are not executed inside a hardware interrupt context. This is very convenient since you can do any kind of heavy lifting in your callback handler, and your code is not pre-empted. *Precision* You should note that the timer are not guaranteed to be precisely accurate, it might fire later than your defined interval (or delay). The timer will not fire before your defined interval though. If you use any blocking `wait` statements in your code, you might contribute to loss in precision for timers. If you want precise hardware timer interrupts consider the `mbed Ticker` class, but you should be aware of the hazards when using hardware interrupts. *Example* Create a reoccurring timer that fires each second:

```
Timer timr(1000);
timr.setCallback<MyClass>(this, &MyClass::callback);
timr.start();
```

The member function `callback` will now be called every second. If you want to use a single shot callback with a delay, *Timer* has a convenience static function:

```
Timer delay = Timer::callOnce<MyClass>(100, this, &MyClass::callback);
```

Now `delay` is a running timer that calls `callback` only one time. *Note* that the timer object (`delay`) should not be deallocated. Deallocating the object will cause the timer to shut down.

Time slices

Say you set an interval of 1000 ms, and your callback takes 300 ms to execute. Then timer will delay for 700 ms and not 1000 ms. It is up to you to ensure your callback do not take longer to execute, than the timer interval.

Public Functions

Timer::Timer()

Construct an empty (zero-timeout) re-occurring timer.

After calling this constructor, you must set the time out and callback function. Then start the timer.

Timer::Timer (uint32_t *intervalOrTimeoutMs*, bool *snglShot*)

Create a new timer with with an interval or timeout time.

All newly created timers are stopped as default. You must also attach callback handler to the timer, before it can start.

Parameters

- `intervalOrTimeoutMs` - The timers time interval before it fires, in milliseconds
- `snglShot` - Set this to `true` if the timer should only fire once. Default `false`

void `mono::Timer::start()`

Start the timer and put into *running* state.

Note: You must set a callback handler, before starting the timer.

void `Timer::Start()`

void `mono::Timer::stop()`

Stop the timer, any pending callback will not be executed.

void `Timer::Stop()`

`bool Timer::SingleShot ()`
constSee if the timer is single shot.

`bool Timer::Running ()`
constSee if the timer is currently running

`void Timer::setInterval (uint32_t newIntervalMs)`
Set a new timer interval.

Parameters

- `newIntervalMs` - The timer interval in milliseconds

`template <typename Owner>`

`void mono::Timer::setCallback (Owner * obj, void(Owner::*) (void) memPtr)`
Sets a C++ callback member function to the timer.

Parameters

- `obj` - A pointer to the callback member function context (the `this` pointer)
- `memPtr` - A pointer to the member function, that is the callback

`void mono::Timer::setCallback (void (*cFunction)) void`
Sets a callback handler C function to the timer.

Parameters

- `cFunction` - A pointer to the C function, that is the callback

Public Static Functions

`template <typename Owner>`

`static Timer* mono::Timer::callOnce (uint32_t delayMs, Owner * obj, void(Owner::*) (void`
Create a single shot timer with a delay and callback function.

The timer object is created on the HEAP, which allows it to exists across stack frame contexts. You can safely create a `callOnce (. . .)` timer, and return from a function. Even if you do not have a reference to the timer object, it will still run and fire. The timer deallocates itself after it has fired. It cannot be reused.

Return

A pointer to the single shot timer

Parameters

- `delayMs` - Delay time before the timer fires, in milliseconds.
- `obj` - A pointer to the callbacks function member context (the `this` pointer)
- `memPtr` - A pointer to the callback member function

`static Timer *mono::Timer::callOnce (uint32_t delayMs, void (*memPtr)) void`
Create a single shot timer with a delay and callback function.

The timer object is created on the HEAP, which allows it to exists across stack frame contexts. You can safely create a `callOnce (. . .)` timer, and return from a function. Even if you do not have a reference

to the timer object, it will still run and fire. The timer deallocates itself after it has fired. It cannot be reused.

Return

A pointer to the single shot timer

Parameters

- `delayMs` - Delay time before the timer fires, in milliseconds.
- `memPtr` - A pointer to the callback C function

Protected Functions

virtual This is the method that gets called by the run loop. *NOTE* that this is not an interrupt function, you can do stuff that take some time.

Core Classes

AppRunLoop

class This is the event run-loop for all mono applications. This class is instantiated and used inside the *ApplicationContext* interface. You should not interact with this class directly. The run loop handles non-critical periodically tasks. Classes can install tasks in the run-loop. Such classes are usually repetitive timers or lazy interrupt handlers. Some standard system tasks are handled statically inside the loop, like the USB serial reads.

Public Functions

`void Timer::taskHandler()`
Start executing the run loop.

`void AppRunLoop::`

`void AppRunLoop::checkUsbDtr()`
Do a single check of the DTR on the virtual UART.

`bool AppRunLoop::addDynamicTask(IRunLoopTask *task)`
Add a task to the dynamic task queue. This task is repeated over and over, until it reports that its should not be scheduled.

The task is added to a linked list, runtime is *n*.

Return

Always true at this point

`bool AppRunLoop::removeDynamicTask(IRunLoopTask *task)`
Remove a task from the dynamic task queue. This will search the queue for the pointer provided, and remove it.

Return

`true` if the object was found and removed, `false` otherwise.

Parameters

- `task` - A pointer to the object, that should be removed

`void AppRunLoop::setResetOnUserButton (bool roub)`

Sets the *Reset on User Button* mode.

If `true` the run loop will check the user button, and if pressed it will trigger a software reset.

Parameters

- `roub` - `true` will reset on user button, `false` is normal functionality.

`void AppRunLoop::quit ()`

Terminate the run loop. Application events and more will stop working

You should use this, if you use your own embedded run loops.

Public Members

`bool mono::AppRunLoop::resetOnDTR`

As default behaviour the run loop will force a reset on high-to-low transition on the serial ports DTR (Data Terminal Ready) line.

This property controls this feature, setting it to `true` will enable software reset via the serial connection. This means the *monoprog* programmer can reset the device and connect to the bootloader.

Setting this to `false` means *monoprog* cannot automatically reset into the bootloader, you must press the reset button yourself.

`uint32_t mono::AppRunLoop::TouchSystemTime`

The CPU time used on processing touch input. This includes:

- ADC sampling (approx 16 samples)
- Touch value evaluation, and possible conversion into events
- Traversing the responder chain
- Handling `TouchBegin`, `TouchEnd` & `TouchMove`, and any function they call

This time includes the execution of your code if you have any button handlers or touch based event callbacks.

`uint32_t mono::AppRunLoop::DynamicTaskQueueTime`

The CPU time used on processing the dynamic task queue The time spend here include all queued tasks and callbacks. these could be:

- Timer* callback
- Any *QueueInterrupt* you might have in use
- All display painting routines (repainting of views subclasses)
- Any custom active *IRunLoopTask* you might use

Nearly all callbacks are executed with origin inside the dynamic task queue. Expect that the majority of your code are executed here.

Protected Functions

bool runLoopActive mono::AppRunLoop::__DEPRECATED("Will be removed in future releases"

As long as this is `true` the standard run loop will run

If set to `false`, the run loop will exit and `main()` will return, which you should absolutely **not** do!.

void AppRunLoop::processDynamicTaskQueue ()

Execute all tasks in the dynamic task queue

void AppRunLoop::removeTaskInQueue (IRunLoopTask *task)

Internal method to sow together neighbours in the linked list

void AppRunLoop::process ()

Process a single iteration of the run loop

void AppRunLoop::checkUsbUartState ()

read the UART DTR state if possible

Protected Attributes

bool mono::AppRunLoop::lastDtrValue

The last seen serial DTR value. Reset can only happen in transitions.

bool mono::AppRunLoop::resetOnUserButton

Set to `true` if you want the run loop to call software reset when pressing the user button. Initial value is `false`

IRunLoopTask ***mono::AppRunLoop::taskQueueHead**

A pointer to the head task of the dynamic task queue. If no task are in the queue, this is `NULL`

IApplication

class Entry point for all mono applications, abstract interface. Every mono application must implement this interface. This is the starting point of your application code, you must call it after the runtime initialization. You do this from inside the `main()` function. Your main function should look like this:

```
int main()
{
    // Construct you IApplication subclass
    MyIApplicationSubclass appCtrl;

    // Tell the IApplicationContext of your existence
    IApplicationContext::Instance->setMonoApplication(&appCtrl);

    // Start the run loop... - and never come back! (Gollum!, Gollum!)
    return appCtrl.enterRunLoop();
}
```

Your mono applications entry point must be your own subclass of *IApplication*. And you *must* initialize it inside (not outside) the `main()` function. This is strictly necessary, because the *IApplicationContext* must be ready when the *IApplication* is executed.

Also you must call the *enterRunLoop* method from main, to enter the event loop and prevent `main()` from returning.

Public Functions

`mono::IApplication::IApplication()`

Construct the global Application class.

Constructor for the global Application Controller. See [IApplication](#) for a description on when to call this constructor.

virtual = 0 Called when mono boot after having been power off or after a reset This method is only called once, you should use it to do initial data and object setup. When this method returns mono will enter in an event loop, so use this method to setup event listeners for your code. Do not call this method yourself, it is intended only to be called by the mono framework runtime.

virtual = 0 The runtime library calls this function when the MCU will go into standby or sleep mode. Use this method to disconnect from networks or last-minute clean ups. When you return from this method the system will go to sleep, and at wakeup the [monoWakeFromSleep\(\)](#) method will be called automatically. Do not call this method yourself, it is intended only to be called by the mono framework runtime.

virtual = 0 Called when mono comes out of a standby or sleep state, where the MCU instruction execution has been paused. Use this method to reestablish I/O connections and refresh data objects. You should not call this method yourself, it is intended only to be called by the mono framework runtime.

`void mono::IApplication::monoWakeFromReset()`

`void mono::IApplication::monoWakeFromReset()`

Start the mono application run loop.

Start the main run loop for your mono application. This method calls the global [IApplicationContext](#) run loop.

The last line in the main.cpp file must be a call to this function:

```
int main()
{
    MyIApplicationSubclass appCtrl;

    // Some app ctrl setup code here perhaps?

    return appCtrl.enterRunLoop();
}
```

Return

The run loop never returns, the return type is only for conformaty.

IApplicationContext

class The Application context class is a singleton class that is automatically instanciated by the framework. You should not need to interact with it directly. It is allocated on the stack, with its member objects. The application context controls the application event loop at hardware event inputs. It is essential for communicating with Serial-USB and the display. Depending on the execution context (hardware mono device or simulator), different subclasses of this interface it used. This interface is provided to give your application code a pointer to the concrete implementation of the application context. Regardless of running on a simulator or the actual device.

Public Functions

virtual = 0 Start the application run loop. This method starts the global run/event loop for the mono application. The method never returns, so a call to this function should be the last line in your `main()` function. The event loop automatically schedules the sub system, such as the network, inputs and the display.

virtual = 0 Sets a pointer to the mono application object

Public Members

`int mono::IApplicationContext::exec()`

`void mono::IAppli`

A pointer the power management system.

Pointer to the global power management object, that controls power related events and functions. Use this pointer to go into sleep mode' or get the current battery voltage level.

AppRunLoop *mono::IApplicationContext::RunLoop

A reference to the main run loop of the application. This pointer must be instanciated be subclasses

`display::IDisplayController` *mono::IApplicationContext::DisplayController

Pointer to the display interface controller object. The object itself should be initialized differntly depending on the ApplicationContext

ITouchSystem *mono::IApplicationContext::TouchSystem

Pointer to the touch system controller object.

The touch system handles touch input from the display or other input device. It must be initialized by an ApplicationContext implementation.

The touch system is the source of *TouchEvent* and delegate these to the *TouchResponder* classes. It is the *ITouchSystem* holds the current touch calibration. To re-calibrate the touch system, you can use this reference.

See

ITouchSystem

QueueInterrupt *mono::IApplicationContext::UserButton

The User Button queued interrupt handler.

Here you add your application handler function for mono user button. To handle button presses you can set a callback function for the button push.

The callback function is handled in the *AppRunLoop*, see the *QueueInterrupt* documentation for more information.

Note that the default initialized callback handler will toggle sleep mode. This means that if you do not set your own handler, the user button will put mono into sleep mode. The default callback is set on the `.fall(...)` handler.

Example for replacing the user button handler, with a reset handler:

```
// the button callback function
void MyApp::handlerMethod()
{
    IApplicationContext::SoftwareReset();
}

// on reset install our own button handler callback
void MyApp::monoWakeFromReset()
{
    IApplicationContext::Instance->UserButton->fall<MyApp>(this, &
↪MyApp::handlerMethod);
}
```

sensor::*ITemperature* *mono::IApplicationContext::**Temperature**

A pointer to the Temperature sensor, if present in hardware.

This is an automatically initialized pointer to the temperature object, that is automatically created by the framework.

sensor::*IAccelerometer* *mono::IApplicationContext::**Accelerometer**

A pointer to the Accelerometer, if present in hardware.

This is an automatically initialized pointer to the accelerometer object, that is automatically created by the framework.

sensor::*IBuzzer* *mono::IApplicationContext::**Buzzer**

A pointer to the buzzer, if present in hardware.

This is an automatically initialized pointer to the buzzer object, that is automatically created by the framework.

IRTCSystem *mono::IApplicationContext::**RTC**

A Pointer to the current RTC interface, if such exists.

Mono has a RTC clock, that can control a system date time clock, that is accessed by the *DateTime* class

You can start or stop the RTC using this interface. Note that this pointer might be NULL

Public Static Functions

static void mono::IApplicationContext::**EnterSleepMode**()

The mono application controller should call this to give the Application Context a reference to itself.

This will ensure the Application Controllers methods gets called. Call this method to make mono goto sleep.

In sleep mode the CPU does not excute instruction and powers down into a low power state. The power system will turn off dynamically powered peripherals.

NOTE: Before you call this method make sure that you configured a way to go out of sleep.

static void mono::IApplicationContext::**ResetOnUserButton**()

Enable *Reset On User Button* mode, where user button resets mono.

If your application encounters unmet dependencies (missing SD Card) or gracefully handles any runtime errors, you can call this method. When called, the run loop will reset mono if the user button (USER_SW) is activated.

This method allows you to reset mono using the user button, instead of the reset button.

static void mono::IApplicationContext::**SleepForMs**(uint32_t ms)

Enter MCU sleep mode for a short time only. Sets a wake-up timer us the preferred interval, and calls the *EnterSleepMode* method.

Parameters

- ms - The number of milli-second to sleep

static void mono::IApplicationContext::**SoftwareReset**()

Trigger a software reset of Mono's MCU.

Calls the MCU's reset exception, which will reset the system. When reset the bootloader will run again, before entering the application.

static void mono::IApplicationContext::SoftwareResetToApplication()

Trigger a software reset of MOno's MCU, that does not load the bootloader.

Use this to do a fast reset of the MCU.

static void mono::IApplicationContext::SoftwareResetToBootloader()

Trigger a software reset, and stay in bootloader.

Calls the MCU reset exception, which resets the system. This method sets bootloader parameters to stay in bootloader mode.

CAUTION: To get out of bootloader mode you must do a hard reset (by the reset button) or program mono using monoprog.

Public Static Attributes

IApplicationContext *IApplicationContext::Instance

Get a pointer to the global application context

Protected Functions

virtual = 0Subclasses should override this method to make the system goto sleep

virtual = 0Subclasses should override this to enable sleep mode for a specific interval only.

virtual = 0Subclasses must implement this to enable the "Reset On User Button" behaviour. See [ResetOnUserButton](#)

virtual = 0Subclasses must implement this method to enable software resets. See [SoftwareReset](#)

virtual = 0Subclasses must implement this to enable software reset to application See [SoftwareResetToApplication](#)

virtual = 0Subclasses must implement this method to allow *reset to bootloader* See [SoftwareResetToBootloader](#)

void mono::IApplicationContext::enterSleepMode()

void mono::IAppli

Protected constructor that must be called by the sub class. It sets up needed pointers for the required subsystems. This ensure the pointers are available when class members' constructors are executed.

If this constructor did not setup the pointers, the PowerManagement constructor would see the [Instance](#) global equal null.

IRTCSystem

class Abstract interface for controlling a MCU specific RTC. This interface defines methods for setting up the RTC component, and starting and stopping the RTC interrupt. The *DateTime* class contains a static *DateTime* object that represents the system time. It is this RTC interface's task to maintain a valid system timestamp by incrementing the static system *DateTime* object. **Public Functions**

virtual = 0Setup the RTC system with all register setting and interrupts needed. This should not start the RTC, just initialize it - making it ready for calling *startRtc()*. Any subclass must implement this function - it is automatically called by the *IApplicationContext* class.

virtual = 0Start the built-in RTC, effectively starting the system date/ time clock.

virtual = 0Stop the built-in RTC, effectively pausing the system date/ time clock.

Public Static Attributes

void mono::IRTCSystem::setupRtcSystem()

The RTC should set this when it is time to process scheduled tasks.

The run loop and the sleep loop should check this variable to know when to process all scheduled tasks. You must set this variable in your RTC interrupt function.

void mono::IRTCSystem::

IRunLoopTask

class This interface defines tasks or functions that can be inserted into the ApplicationRunLoop. The interface defines a method that implements the actual logic. Also, the interface defines the pointers *previousTask* and *nextTask*. These define the previous and next task to be run, in the run loops task queue. To avoid dynamic memory allocation of linked lists and queues in the run loop, the run loop handler functions, are themselves items in a linked list. All classes that want to use the run loop, must inherit this interface. *NOTE* that tasks in the run loop do not have any constraints on how often or how rare they are executed. If you need a function called at fixed intervals, use a Ticker or timer. **Protected Functions**

virtual = 0This is the method that gets called by the run loop. *NOTE* that this is not an interrupt function, you can do stuff that take some time.

Protected Attributes

void mono::IRunLoopTask::taskHandler()

A pointer to the previous task in the run loop The task is the first in queue, this is NULL

IRunLoopTask *mono::

IRunLoopTask *mono::IRunLoopTask::nextTask

A pointer to the next task to be run, after this one. If this task is the last in queue, this is NULL

bool mono::IRunLoopTask::singleShot

Tasks are expected to be repetitive. They are scheduled over and over again. Set this property to true and the task will not be scheduled again, when handled.

ITouchSystem

class Interface for the Touch sub-system **Public Functions**

virtual = 0Initialize the touch system controller. *NOTE*: This method is called by the current *IApplicationContext* automatically upon reset. You should call this manually.

virtual = 0Sample the touch inputs and dispatch events if needed. This *AppRunLoop* calls this method automatically on each cycle. If you wish to limit the sample frequency, your subclass must manually abort some calls to this method. Note: Inactive touch systems must ignores calls to this method.

virtual = 0Sub-classes must convert corrdinates to screen pixels (X axis)

virtual = 0Sub-classes must convert corrdinates to screen pixels (Y axis)

virtual = 0Get the touch systems current calibration config.

virtual = 0Set a new calibration config on the touch system.

virtual Activate (enables) the touch system.

virtual Deactivates (disables) the touch system.

virtual constRetuern `true` if touch system is active.

Protected Functions

void `mono::ITouchSystem::init()`

Run the system system handlers of the begin touch event.

NOTE: You should not call this method manually. It is used by the sub- class of this interface.

Parameters

- `pos` - The begin event position on the screen

void `mono::ITouchSystem::runTouchMove` (`geo::Point &pos`)

Run the system system handlers of the move touch event.

NOTE: You should not call this method manually. It is used by the sub- class of this interface.

Parameters

- `pos` - The move event position on the screen

void `mono::ITouchSystem::runTouchEnd` (`geo::Point &pos`)

Run the system system handlers of the end touch event.

NOTE: You should not call this method manually. It is used by the sub- class of this interface.

Parameters

- `pos` - The end event position on the screen

ManagedPointer

template <typename ContentClass>

class Pointer to a heap object, that keeps track of memory references.

The managed pointer is an object designed to live on the stack, but point to memory cobntent that live on the heap. The *ManagedPointer* keeps track of memory references, such that the it can be shared across multiple objects in your code.

It maintains an internal reference count, to keep track of how many objects holds a reference to itself. If the count reaches zero, then the content is deallocated.

With *ManagedPointer* you can prevent memory leaks, by ensuring un-references memory gets freed.

Public Functions

`mono::ManagedPointer::ManagedPointer()`

Create an empty pointer

`void mono::ManagedPointer::Surrender()`

Give up this ManagedPointers references to the content object, by setting its pointer to NULL.

This means that if the Reference count is 1, this pointer is the only existing, and it will not dealloc the content memory upon deletion of the *ManagedPointer*.

We Reference count is > 1, then other ManagedPointers might dealloc the content memory.

TouchEvent

class

TouchResponder

class The TouchReponder handles incoming touch events. The *TouchResponder* is an interface that classes and inherit from to receive touch input events. This class also defined global static method used by Mono's hardware dependend touch system. These static methods receives the touch events and delegates them to all objects in the responder chain. You can make any object a receiver of touch events if you inherit from this interface. You need to override 3 methods: RespondTouchBegin RespondTouchMove RespondTouchEnd

These methods are called on any subclass when touch input events are received. *Note* that your subclass will receive all incoming events not handled by other responders.

If you want to make touch enabled graphical elements, you should use the interface ResponderView. This class is the parent class for all touch enabled views.

See •

ResponderView

See

ITouchSystem

Public Functions

`TouchResponder::TouchResponder()`

Create a new responder object that receives touch input.

Upon creation, this object is automatically inserted into the responder chain, to receive touch input events.

`void mono::TouchResponder::activate()`

Add this responder to the responder chain

`void mono::TouchResponder::deactivate()`

Remove this responder from the responder chain

Public Static Functions

`TouchResponder *TouchResponder::FirstResponder()`

Get the top of the responder queue, the first to handle touch.

If there are no objects repnding to touch, this will return `NULL`

UI Widgets

AbstractButtonView

class **Public Functions**

template <typename Owner>

`void mono::ui::AbstractButtonView::setClickCallback(Owner * obj, void(Owner::*)(void))`

Attach a member function as the button click handler.

Provide the callback member function you ewant to be called when the button is clicked.

NOTE: There can only be one callback function

Parameters

- `obj` - A pointer to the object where the callback method exists
- `memPtr` - A pointer to the callback method itself

`void mono::ui::AbstractButtonView::setClickCallback(void (*memPtr)) void`

Attach a C function pointer as the button click handler.

Provide a pointer to the callback C function, you ewant to be called when the button is clicked.

NOTE: There can only be one callback function.

Parameters

- `memPtr` - A pointer to the C function callback

Protected Functions

`AbstractButtonView::AbstractButtonView()`

Cannot construct abstract buttons.

`AbstractButtonView::AbstractButtonView(geo::Rect rct)`

Construct a button with a defined viewRect.

BackgroundView

class A full screen solid colored background. Use this view to paint the background any color you like. This view is by default the same size as Mono's display. It can be used a solid colored background for your GUI. To save memory you should only have one background view per app. *Painting on top of the background* To ensure that the background is *behind* all other UI widgets on the screen, it has to be rendered first. Then all other views will apeear on top of the background. To achieve this, it is important to keep track of your repaint order. All paint calls must begin with a *scheduleRepaint* to the background view. *Example*

```
// Construct the view (should alwaws be a class member)
```

```
mono::ui::BackgroundView bg;

// set an exiting new background color
bg.setBackgroundColor(mono::display::RedColor);

//show the background
bg.show();

// show all my other views, after the call to bg.show()
```

Public Functions

`BackgroundView::BackgroundView` (display::Color *color*)

Construct a Background view on the entire screen.

This constructor takes a optional color to use for the background. If no argument is provided the color default to the `View::StandardBackgroundColor`

Also be default the view dimension is the entire display, meaning it has a bounding rect (`vireRect`) that is (0,0,176,220).

Note: Remember to all *show* to make the view visible.

Parameters

- `color` - An optional background color

`BackgroundView::BackgroundView` (geo::Rect **const** &*rect*, display::Color *color*)

Construct a *BackgroundView* with a specific position and size.

If you need at solid color (as background) in a confined space on the screen, then you can use this constructor.

Parameters

- `rect` - The position and size of the view
- `color` - An optional color, default is `StandardBackgroundColor`

`void BackgroundView::setBackgroundColor` (display::Color *color*)

Sets a new background color on the view.

Set a new background color on the view. If the view is shared between multiple UI scenes, you can use this method to change the background color.

When changing the background color the view is not automatically repainted. You must call *scheduleRepaint* yourself.


`mono::display::Color BackgroundView::Color` ()

constGets the current background color.

virtual Repaint the view content, using the *View::painter*. Re-paint the view content. This method should be called when the view content has changed. You can call this method directly, but it might cause graphics artifacts because the display is not double buffered. Instead you should schedule a repaint by calling the *scheduleRepaint()* method. This method will schedule the repaint, right after the next display update. The display system will not schedule any repaints automatically. The view does not contain any state information, so you or other classes utilizing view must schedule repaints. In subclasses of *View*, this method *must* be overwritten.

ButtonView

class A Push Button UI Widget. This is a common state-less push button. It is basically a bordered text label. This button reacts to touch input (pushes) and can call a function when it is pushed. You provide the button with a callback function, that gets called when the button is pushed. A valid button push is a touch that begins *and ends* within the button boundaries. If a touch begins inside the buttons boudaries, but ends outside - the button click callback is not triggered. You define the button dimensions by the Rect you provide in the constructor. Note that the resistive touch panel is not that precise, you should not create buttons smaller than 40x35 pixels. Also note that buttons do not automatically scale, when you set their text content. *Example*

```
void BackgroundView::repaint() // 
    ↪ Create the button (should normally be defined as a class member)
    mono::ui::ButtonView btn(mono::geo::Rect(10,10,100,35), "Push Here")

    // Setup a click handler
    btn.setClickCallback<MyClass>(this, &MyClass::MyClickHandler);

    // show the button on the screen
    btn.show();
```

Public Functions

ButtonView::ButtonView()

Construct an empty button.

The button will have zero dimensions and no text.

ButtonView::ButtonView (geo::Rect *rect*, String *text*)

Construct a button with dimensions and text.

Creates a button with the provided dimensions and text to display. You still need to setup a callback and call *show*.

Parameters

- *rect* - The view rect, where the button is displayed
- *text* - The text that is showed on the button

ButtonView::ButtonView (geo::Rect *rect*, const char **text*)

Construct a button with dimensions and text.

Creates a button with the provided dimensions and text to display. You still need to setup a callback and call *show*.

Parameters

- *rect* - The view rect, where the button is displayed
- *text* - The text that is showed on the button

void ButtonView::setText (String *txt*)

Set the text content.

Sets the text that is displayed on the button. Note that the width and height of the button is not changed. You must change the buttons *viewRect* if your text is larger than the buttons dimensions.

When you set new text content, the button is automatically repainted

Parameters

- `txt` - The new text content

void `mono::ui::ButtonView::setFont` (MonoFont `const &newFont`)

Change the button fontface (font family and size)

You can change the buttons font to use a larger (or smaller) font.

void `mono::ui::ButtonView::setFont` (GFXfont `const &newFont`)

Change the buttons fontface (font family and size)

You can change the buttons font to use a larger (or smaller) font.

Parameters

- `newFont` - The font face to render the text content with

void `ButtonView::setBorder` (Color `c`)

Sets the border color.

This method will not schedule repaint! You must *scheduleRepaint* manually.

Parameters

- `c` - The new border color

void `ButtonView::setText` (Color `c`)

Sets the text color.

This method will not schedule repaint! You must *scheduleRepaint* manually.

Parameters

- `c` - The new text color

void `ButtonView::setHighlight` (Color `c`)

Sets the highlight color (border & text)

The highlight color is the color used to represent a button that is currently being pushed. This means that a touch event has started inside its boundaries. The highlight color is applied to both border and text.

This method will not schedule repaint! You must *scheduleRepaint* manually.

Parameters

- `c` - The new highlight color

void `ButtonView::setBackground` (Color `c`)

Sets the background color.

The background color of the fill color inside the button bounding rectangle.

This method will not schedule repaint! You must *scheduleRepaint* manually.

Parameters

- `c` - The new border and text color

virtual Set the view's position and size, by providing a rectangle object. *Note* that this method does not repaint the view, you must do that explicitly.

void ButtonView::setRect (geo::Rect rect)

Parameters

- rect - The view rectangle, containing size and position

const TextLabelView &ButtonView::TextLabel ()

constGet a reference to the internal TextLabel object.

You get a const reference to the button internal *TextLabelView*

template <typename Owner>

void mono::ui::ButtonView::setClickCallback (Owner * obj, void(Owner::*)(void) memPtr)

Attach a member function as the button click handler.

Provide the callback member function you ewant to be called when the button is clicked.

NOTE: There can only be one callback function

Parameters

- obj - A pointer to the object where the callback method exists
- memPtr - A pointer to the callback method itself

void mono::ui::ButtonView::setClickCallback (void (*memPtr)) void

Attach a C function pointeras the button click handler.

Provide a pointer to the callback C function, you ewant to be called when the button is clicked.

NOTE: There can only be one callback function.

Parameters

- memPtr - A pointer to the C function callback

virtual Painters.

ConsoleView

void ButtonView::repaint ()

class

template <uint16_t W, uint16_t H>

Public Functions

mono::ui::ConsoleView::ConsoleView (geo::Point pos)

Construct a new *ConsoleView*, for viewing console output on the screen.

void mono::ui::ConsoleView::WriteLine (String txt)

Write a string to the console, and append a new line.

Parameters

- txt - The string to write to the console

virtual Repaint the view content, using the [View::painter](#). Re-paint the view content. This method should be called when the view content has changed. You can call this method directly, but it might cause graphics artifacts because the display is not double buffered. Instead you should schedule a repaint by calling the [scheduleRepaint\(\)](#) method. This method will schedule the repaint, right after the next display update. The display system will not schedule any repaints automatically. The view does not contain any state information, so you or other classes utilizing view must schedule repaints. In subclasses of [View](#), this method *must* be overwritten.

```
void mono::ui::ConsoleView::repaint ()
```

Get the width of a line in characters.

<# description #>

Return

Number of characters in one line

```
int mono::ui::ConsoleView::consoleLines ()
```

Get the number lines in the console.

<# description #>

Return

Numeber of text lines on the console view.

Protected Attributes

```
TextBuffer<(W-4)/5, (H-4)/9> mono::ui::ConsoleView::textBuffer
```

Text buffer that hold the visible text in the console. When the console scrolls, the text in the buffer is overwritten.

```
bool mono::ui::ConsoleView::scrolls
```

Becomes true when the console text has reached the bottom line of its view rectangle. And all text appending from now on, causes the console to scroll.

GraphView

class Visualizes a data series as a graph, based on a associated data source. This class can visualize an array of samples on a graph. You provide the data by subclasseing the [IGraphViewDataSource](#) to deliver to data to display. This class only display the data, it does not hold or buffer it. In this sense the [GraphView](#) contains no state. *Example* To demonstrate a simple example of using the [GraphView](#) we must also create a data source. For an associated data source, let us wrap a simple C array as a [IGraphViewDataSource](#) subclass:

```
// Subclass the IGraphViewDataSource interface
class DataSource : public IGraphViewDataSource
{
private:

    // Use an internal array as data store
    uint8_t data[100];

public:

    // Override the method that provide data samples
```

```

    int DataPoint(int index) { return data[index]; }

    // Override the method that return the total length of the data set
    int BufferLenght() { return 100; }

    // Override the method that return the valid value range of the data
    // samples.
    int MaxSampleValueSpan() { return 256; }
};

```

The class `DataSource` is just an array with a length of 100. Note that we only store 8-bit data samples (`uint_t`), therefore the valid data range is 256. The [GraphView](#) expects the data values to be signed, meaning the valid range is from -127 to +127.

We have not provided any methods for putting data into the data source, but we will skip that for this example.

Now, we can create a [GraphView](#) that displays data from the array:

```

// Crate the data source object
DataSource ds;

// The view rectangle, where the graph is displayed
mono::geo::Rect vRect(0,0,150,100);

// create the graph view, providing the display rect and data
mono::ui::GraphView graph(vRect, ds);

//tell the graph view to be visible
graph.show();

```

Update Cursor

If you [IGraphViewDataSource](#) subclass overrides the method: `NewestSampleIndex()`, the [GraphView](#) can show an update cursor. The cursor is a vertical line drawn next to the latest or newest sample. The cursor is hidden by default, but you can activate it by overriding the data source method and calling `setCursorActive`.

Note that if you time span across the x-axis is less than 100 ms, the cursor might be annoying. I would recommend only using the cursor when your graph updates slowly.

Scaling and Zooming

the graoh view will automatically scale down the data samples to be display inside its graph area. It displays the complete data set from the data source, and does not support displaying ranges of the data set.

If you wish to apply zooming (either on x or Y axis), you must do that by scaling transforming the data in the data source. You can use an intermediate data source object, that scales the data samples, before sending them to the view.

See

[IGraphViewDataSource](#)

Public Functions

`GraphView::GraphView()`

Construct a [GraphView](#) with no *viewRect* and data source.

This contructor creates a [GraphView](#) object that needs a *viewRect* and source to be set manually.

You cannot display view objects that have a null-*viewRect*

See

`setDataSource`

See

`setRect`

`GraphView::GraphView (geo::Rect rect)`

Construct a *GraphView* with a defined *viewRect*.

This constructor takes a *viewRect*, that defines the space where graph is displayed.

****No data will be displayed before you set a valid source****

See

`setDataSource`

Parameters

- `rect` - The *viewRect* where the graph is displayed

`GraphView::GraphView (geo::Rect rect, const IGraphViewDataSource &dSource)`

Construct a *GraphView* with `viewRect` and a data source.

Parameters

- -

Protected Functions

virtual Repaint the view content, using the *View::painter*. Re-paint the view content. This method should be called when the view content has changed. You can call this method directly, but it might cause graphics artifacts because the display is not double buffered. Instead you should schedule a repaint by calling the *scheduleRepaint()* method. This method will schedule the repaint, right after the next display update. The display system will not schedule any repaints automatically. The view does not contain any state information, so you or other classes utilizing view must schedule repaints. In subclasses of *View*, this method *must* be overwritten.

IGraphViewDataSource

class Data provider interface for the *GraphView* class. This interface defines the *DataSource* for *GraphView* objects. The graph view queries an associated data source object for the data to display. All *GraphView* objects must have a data source object that hold the data to be displayed. The data source interface exposes an array-like scheme for providing data to the *GraphView*. This interface forces you to expose your data sample as an array, where all samples has an index. The index starts at index 0 and has variable length. You must subclass this interface and override at least 3 virtual methods. These 3 methods are: *DataPoint(int)* : Return individual data samples *BufferLength()* : Return the full length of the data source *MaxSampleValueSpan()* : Return the full range of the data sample values

By providing the methods for retrieving data, getting the total length of the data buffer and defining the valid value range of the data samples.

If your subclass is representing a casual signal buffer, where samples are continuesly written, you might override the method:

```
void GraphView::repaint ()
    NewestSampleIndex()
```

This method should return the index to the newest sample in the buffer. This enables the *GraphView* to display a scrolling cursor, that moves as the buffer data gets updates.

Note: You are in charge of notifying the associated *GraphView*'s when the data source content changes.

See

GraphView

Public Functions

virtual = 0Override this to return data point samples to the view. Provides a sample from the data source. You must override this method to return samples at any given index.

```
int mono::ui::IGraphViewDataSource::DataPoint (int index) Return
```

the sample value at the given index

Parameters

- *index* - The 0-indexed sample position, that should be returned

virtual = 0Override this to return the data sample buffer length to the view. Returns the length / the total number of data samples in the data source. You must override this method to return the total number of data sample in your data source implementation.

```
int mono::ui::IGraphViewDataSource::BufferLength () Return
```

the size / length of the data source

virtual = 0The value span for the data samples. used to map the buffer samples to the screen (view height)

virtual Return the position / index of the newest sample in the buffer. Override this method to get a scrolling pointer on the *GraphView*, a pointer that is drawn at this index position.

```
int mono::ui::IGraphViewDataSource::MaxSampleValueSpan () int mono::ui::IGra
```

Position of newest sample

IconView

class A view that display a bitmap icon. This UI widget display the bitmap contents of the Mono's icon format. To display the icon you just provide a reference to the global variable for the icon, and a position offset. *Mono Icons* The icon format is a bitmasps of color blending values. This means every pixel value represents a blend between to colors, the fore- and background. You set these colors specifically or uses the default system values. The technique allows icon to be semi-transparent over a background. The default values for foreground color are the *View* static variable *StandardBorderColor*. SImialr the default background is *StandardBackgroundColor* *System icons* The framework contains a number of default icons each in two sizes: 16x16 and 24x24 pixels. These are distributed as part of the official SDK releases. These icons include: *alarmBell16*: defined in *alarm-bell-16.h* *alarmBell24*: defined in *alarm-bell-24.h* *alarmClock16*: defined in *alarm-clock-16.h* *alarmClock24*: defined in *alarm-clock-24.h* *clock16*: defined in *clock-16.h* *clock24*: defined in *clock-24.h* *cloudComputing16*: defined in *cloud-computing-16.h* *cloudComputing24*: defined in *cloud-computing-24.h* *mute16*:

defined in *mute-16.h* mute24: defined in *mute-24.h* pauseSymbol16: defined in *pause-symbol-16.h* pauseSymbol24: defined in *pause-symbol-24.h* playButton16: defined in *play-button-16.h* playButton24: defined in *play-button-24.h* reload16: defined in *reload-16.h* reload24: defined in *reload-24.h* settings16: defined in *settings-16.h* settings24: defined in *settings-24.h* speaker16: defined in *speaker-16.h* speaker24: defined in *speaker-24.h* thermometer16: defined in *thermometer-16.h* thermometer24: defined in *thermometer-24.h* upload16: defined in *upload-16.h* upload24: defined in *upload-24.h* wifiMediumSignalSymbol16: defined in *wifi-medium-signal-symbol-16.h* wifiMediumSignalSymbol24: defined in *wifi-medium-signal-symbol-24.h*

To see the full list of available system icons, you should see the source code folder: `/include/display/icons` or on [Github](#): `/resources/icons`

Example

Setup an *IconView* with the speaker 16x16 icon:

```
•#include <speaker-16.h>

IconView icn(geo::Point(20,20), speaker);
icn.show();
```

Creating your own icons

The tool we use to create icon objects (simple header files) are available on GitHub and converts bitmap images (like JPG and PNG) into icons files.

[img2icon on Github](#)

See

mono::display::MonoIcon

Public Functions

IconView::IconView()

Construct an empty icon view, with no icon and zero dimensions.

IconView::IconView(const geo::Point &position, const MonoIcon &icon)

Construct an icon view with an icon and position offset.

You provide the offset position (not the Rect) to the view and the global variable for the icon object.

The views rect (its dimensions) is calculated from the icon you provide.

Remember: to include the header file for the icon you wish to use

Parameters

- `position` - The offset where the view is positioned
- `icon` - The reference to the global icon object

void IconView::setForeground(Color c)

Set the foreground color for the icon.

Parameters

- `c` - The color to use as foreground

void `IconView::setBackground` (Color *c*)
Set the background blending color for the icon.

Parameters

- *c* - The color to use as background

void `IconView::setIcon` (const `MonoIcon` **icon*)
Replace the existing icon with a new one.

Parameters

- *ocon* - replacement


Color `IconView::Foreground` ()
constGet the current foreground color.

Color `IconView::Background` ()
constGet the current background blening color.

virtual Repaint the view content, using the [View::painter](#). Re-paint the view content. This method should be called when the view content has changed. You can call this method directly, but it might cause graphics artifacts because the display is not double buffered. Instead you should schedule a repaint by calling the [scheduleRepaint\(\)](#) method. This method will schedule the repaint, right after the next display update. The display system will not schedule any repaints automatically. The view does not contain any state information, so you or other classes utilizing view must schedule repaints. In subclasses of [View](#), this method *must* be overwritten.

ImageView

class Displays a bitmap image on the display. The [ImageView](#) can render a bitmap image on the display. It needs a image data source, that delivers the actual bitmap. (See [BMPIImage](#)) You provide the image data and a bounding rect where the [ImageView](#) is painted. If you wish to use the class [BMPIImage](#) as a image source, you must initialize the SD Card Filesystem first! ## Example

```
void IconView::repaint () // 
    ↪init the SD card before accessing the file system

// Open and prepare the BMP file to display
mono::media::BMPIImage img("/sd/my_pic.bmp");

// create the view and provide the image source
mono::ui::ImageView imgView(&img);

// tell the image to be showed
imgView.show();
```

It is your responsibility to make sure the source image data object is staying around, and do not get deallocated. Preferreably you should make both the image source and the view class members.

Cropping

The image view can crop the source image, thereby only showing a selected portion. The default crop is defined by the views bounding rect. Naturally images larger than the view's rect will be cropped with repect to the upper left corner.

The default cropping Rect is therefore `*(0, 0, imageWidth, imageHeight)*`

Public Functions

`ImageView::ImageView()`

Construct an empty image view, where no image is displayed To display an image you need to call [setImage](#) later.

`ImageView::ImageView (media::Image *img)`

Construct an UI image from an image file

At the moment only BMP images are supported! *Remember to initialize the mbed class object before calling this constructor!*

The image [viewRect](#) is set to the full screen size. Use [setRect](#) to adjust size and position.

Parameters

- `img` - The image data source to show

`void ImageView::setImage (media::Image *img)`

Set a new image source object *Note: This method also resets the current cropping rectangle!*

Parameters

- `img` - A pointer to the new image to display

`void ImageView::setCrop (geo::Rect crp)`

Set image cropping.

Define which portion of the image should be displayed inside the view's own bounding rectangle. By default as much of the original image as possible will be shown.

By defining a cropping rectangle you can define an offset and size to display from the source image.

The source image has the same coordinate system as the display. That is (0,0) is the upper left corner.

Parameters

- `crp` - A cropping rectangle

`const mono::geo::Rect &ImageView::Crop()`

Get the current cropping rectangle Get the current used cropping rectangle for the source image.

virtual Repaint the view content, using the [View::painter](#). Re-paint the view content. This method should be called when the view content has changed. You can call this method directly, but it might cause graphics artifacts because the display is not double buffered. Instead you should schedule a repaint by calling the [scheduleRepaint\(\)](#) method. This method will schedule the repaint, right after the next display update. The display system will not schedule any repaints automatically. The view does not contain any state information, so you or other classes utilizing view must schedule repaints. In subclasses of [View](#), this method *must* be overwritten.

Protected Attributes

`void ImageView::repaint()`

A pointer to the Image object that is to be displayed in the screen.

`media::Image *mono:`

`geo::Rect mono::ui::ImageView::crop`

Image crop rectangle. Setting this rect will crop the source image (non destructive).


OnOffButtonView

class Public Functions

virtual Repaint the view content, using the [View::painter](#). Re-paint the view content. This method should be called when the view content has changed. You can call this method directly, but it might cause graphics artifacts because the display is not double buffered. Instead you should schedule a repaint by calling the [scheduleRepaint\(\)](#) method. This method will schedule the repaint, right after the next display update. The display system will not schedule any repaints automatically. The view does not contain any state information, so you or other classes utilizing view must schedule repaints. In subclasses of [View](#), this method *must* be overwritten.

ProgressBarView

class A UI widget displaying a common progress bar. This progressbar is simply a rectangle with a line (progress indicator) inside. The indicator value is (by default) a value between 0 and 100, but you can set your own minimum and maximum points. The progress value's minimum and maximum is automatically calculated into the correct pixel value, scaled to the pixel width of the progressbar. *Example*

```
void OnOffButtonView::repaint() // 
    → Create the progressbar object
    mono::ui::ProgressBarView prgs(mono::geo::Rect(10,10,156,100));

    // set the progressbars indicator value to 50%
    prgs.setValue(50);

    // display it
    prgs.show();
    A common mistake
```

Be aware that the progressbar is painted asynchronously. This means you cannot increment its value inside a for-loop or alike.

This code will not work:

```
for (int i=0; i<100; i++) {
    prgs.setValue(i);
}
```

Even if the loop ran veery slow, you will not see a moving progress indicator. The reason is that the view is only painted in the run-loop, so no screen updates can happen from inside the for-loop!

You should use a continous timer, with a callback that increments the progress indicator:

```
void updateProgress() {
    prgs.setValue(i++);
}

mono::Timer tim(500);
tim.setCallback(&updateProgress);
```

This code inject a continous task to the run loop that can increment the progressbar. This is the correct way to animate a progressbar.

Public Functions

```
ProgressBarView::ProgressBarView()
    Create a ProgressBar with a zero view rect (0,0,0,0)
```

`ProgressBarView::ProgressBarView (geo::Rect rect)`

Create a *ProgressBarView* with values from 0 to 100.

Create a new *ProgressBarView*, with a defined view rect and default progress value span: 0 to 100, with a current value of 0.

Parameters

- `rect` - The view rect where the ProgressBar is painted

`void ProgressBarView::setValue (int newValue)`

Set a new progress value.

Set a new current value for the progress bar. The value is truncated to the existing value span (min and max values).

Changes to the value will trigger the *value changed callback* and cause the view to schedule itself for repaint.

Parameters

- `newValue` - The new value

`void ProgressBarView::setMaximum (int max)`

Define a new minimum value for the progress indicator.

SETTERS.

`void ProgressBarView::setMinimum (int min)`

define a new maximum value for the progress indicator

`template <typename Owner>`

`void mono::ui::ProgressBarView::setValueChangedCallback (Owner * cnxt, void (Owner::*) (v`

Set a progress value change callback function.

Get notification callback everytime this progressbar changes its value

Parameters

- `cnxt` - A pointer to the callback member context (the `this` pointer)
- `memPtr` - A pointer to the callback member function

`int ProgressBarView::Minimum ()`

constGETTERS.

`int ProgressBarView::Maximum ()`

constGETTERS.

Protected Functions

`void ProgressBarView::init ()`

convenience initializer

virtual MISC.

ResponderView

class Public Functions

virtual Shows (repaints) the view and insert into the responder chain.

void ProgressBarView::repaint ()

void ResponderView

View::show

virtual hides the view, and remove from the responder chain

void ResponderView::hide ()

See

View::hide

Protected Functions

virtual Internal touch handler, you should not overwrite this

virtual Internal touch handler, you should not overwrite this

virtual Internal touch handler, you should not overwrite this

SceneController

class A logical grouping of views, that together comprise a scene. A Scene is a group of views that together represent a UI. The scene enables you to *hide*, *show* and repaint all the views at once. You should use the scene to represent a complete UI that takes up a full screen. Your application can have multiple scenes, that you can switch between to navigate in your app. *Adding views to a scene* When you create a scene (constructing the object instance), you need to add the views to it. You do this assigning the relevant view by calling the *addView* method, for every view. When a view is added to a scene, the scene are now in charge of controlling the views state. It is the scene that handles repaints and visibility states. Like views, scenes are constructed in *hidden* state. *Changing scenes* A scene is shown on the display by calling it *show* method. This triggers the callback set be *setShowCallback*, and enables you to provide your scene's views with the data to be displayed. When you wish to hide or navigate to another view, you must call *requestDismiss*. This will trigger the callback set by *setDismissCallback*, that must do the actual *hide* call. Also, this method must call *show* on the new scene to display. *Scene background color* Scenes have a background color, that is painted before any of the views in the scene. By default the background color is *View::StandardBackgroundColor* , but you can change that with the *setBackground* method. *Scene dimensions* A scene has display bounaries, just like views. However, these are not enforced by the scene itself. They exist to allow you to enforce restrictions on dimensions. **Public Functions**

void ResponderView::respondTouchBegin (TouchEvent &event)

void ResponderView

Construct a scene that takes up the entire display.

SceneController::SceneController (const geo::Rect &rect)

Construct a scene with specific position and dimensions.

Parameters

- rect - The scenes visible Rect

virtual Add a view to the scene.

void SceneController::addView (const IViewALike &child)

Parameters

- `child` - The view to add to the scene
- virtual Remove a view from the scene.

`void SceneController::removeView (const IViewALike &child)`

Parameters

- `child` - The view to remove from the scene

`void SceneController::requestDismiss ()`
Call the scene dismiss callback handler.

See

[*setDismissCallback*](#)

`mono::display::Color SceneController::BackgroundColor ()`
constGet the scenes background color.

`void SceneController::setBackground (display::Color color)`
Set the scenes background color.

virtual constGet the scene's visibility state.

virtual Show the scene. This will trigger the [*setShowCallback*](#) and call `show` on all the scene's views.

virtual Hide the scene. This will call the scene's hide handler as set by [*setHideCallback*](#) and then call `hide` on all the scene's views. **Note:** When you need to change between scenes, you should use [*requestDismiss*](#) - not `hide`

virtual Schedule repaint of the scene's views.

virtual Sets a new Rect for the scene's.

virtual constGet the scene's view rect area and offset.

virtual Get the scene position (upper left corner)

virtual Get the scene's dimension.

`bool SceneController::Visible ()`

`void SceneControl`

`void mono::ui::SceneController::setShowCallback (Context * cnxt, void (Context::*) (const`
Sets the callback handler for the `show` event.

You can provide a callback to be called when the scene is about to be shown. Use this callback function to do any pre-setup of the scenes internal views.

Parameters

- `cnxt` - The callback context, normally the `this` pointer
- `memptr` - The callback method on the context class.

`template <typename Context>`

`void mono::ui::SceneController::setHideCallback (Context * cnxt, void (Context::*) (const`
Sets the callback handler for the `hide` event.

You can provide a callback to be called when the scene is about to be hidden. Use this callback function to do any post-teardown of the scenes internal views.

Parameters

- `cnxt` - The callback context, normally the `this` pointer
- `memptr` - The callback method on the context class

```
template <typename Context>
```

```
void mono::ui::SceneController::setDismissCallback(Context * cnxt, void(Context::*) (vo
```

Sets the callback handler for the `dismiss` event.

You can provide a callback to be called when the scene is about to be dismissed. You can use this callback to hide this scene and setup the new view to be displayed.

Parameters

- `cnxt` - The callback context, normally the `this` pointer
- `memptr` - The callback method on the context class

StatusIndicatorView

class `Indicate` a boolean status, true/false, on/off or red/green. The status indicator displays a circular LED like widget, that is red and green by default. (Green is true, red is false) You use the method [setState](#) to change the current state of the indicator. This you change the state, the view automatically repaints itself on the screen.

Example

```
// Initialize the view (you should make it a class member)
mono::ui::StatusIndicatorView indicate(mono::geo::Rect(10,10,20,20), ↵
↪false);

// Change the default off color from red to to white
indicator.setOnStateColor(mono::display::WhiteColor);

// set the state to be On
indicator.setState(true);

// make the view visible
indicator.show();
```

Public Functions

```
StatusIndicatorView::StatusIndicatorView()
```

construct a `StatusIndicator` with default parameters

this view will be initialized in an empty *viewRect*, that is (0,0,0,0)

```
StatusIndicatorView::StatusIndicatorView(geo::Rect vRct)
```

Construct a `Indicator` and provide a [View](#) rect.

This create a view rect the exists in the provided Rect and has the default state `false`

Note: You need to call [show](#) before the view can be rendered

Parameters

- `vRct` - The bounding view rectangle, where the indicator is displayed

```
StatusIndicatorView::StatusIndicatorView(geo::Rect vRct, bool status)
```

Construct a `Indicator` and provide a [View](#) rect and a initial state.

This create a view rect the exists in the provided Rect

Note: You need to call [show](#) before the view can be rendered

Parameters

- `vRect` - The bounding view rectangle, where the indicator is displayed
- `status` - The initial state of the indicator

`void StatusIndicatorView::setOnStateColor` (`display::Color col`)

Sets the *on* state color.

If you change the color, you must call [*scheduleRepaint*](#) to make the change have effect.

Parameters

- `col` - The color of the indicator when it is *on*

`void StatusIndicatorView::setOffStateColor` (`display::Color col`)

Sets the *off* state color.

MARK: SETTERS.

If you change the color, you must call [*scheduleRepaint*](#) to make the change have effect.

Parameters

- `col` - The color of the indicator when it is *off*

`void StatusIndicatorView::setState` (`bool newState`)

Sets a new on/off state.

Note that the view repaints itself if the new state differs from the old.

Parameters

- `newState` - The value of the new state (`true` is *on*)

`bool StatusIndicatorView::State` ()

constGets the state of the indicator.

MARK: GETTERS.

Return

`true` if the state is *on*, `false` otherwise.

Protected Functions

`void StatusIndicatorView::initializer` ()

MARK: Constructors.

virtual Repaint the view content, using the [*View::painter*](#). Re-paint the view content. This method should be called when the view content has changed. You can call this method directly, but it might cause graphics artifacts because the display is not double buffered. Instead you should schedule a repaint by calling the [*scheduleRepaint\(\)*](#) method. This method will schedule the repaint, right after the next display update. The display system will not schedule any repaints automatically. The view does not contain any state information, so you or other classes utilizing view must schedule repaints. In subclasses of [*View*](#), this method *must* be overwritten.

TextLabelView

class A Text Label displays text strings on the display. Use this UI view whenever you need to display text on the screen. A text label renders text strings on the display. As all views the label lives inside a defined rectangle (*viewRect*), where the text is rendered. If the rectangle is smaller than the length of the text content, the content will be cropped. If the rectangle is larger, then you can align the text inside the rectangle (left, center or right). *Example* You can mix and match mono strings with standard C strings when constructing TextLabels. **Create a label using a C string:**

```
void StatusIndicatorView::repaint()
```

TextLabelView

```
    ↳ lbl("This is a constant string");
```

Also you can use C strings allocated on the stack:

```
char text[4] = {'m', 'o', 'n', 'o'};
```

```
TextLabelView lbl(text);
```

Above the TextLabel will take a copy of the input string, to ensure it can be accessed asynchronously.

Content

The text view contains its content (a *String* object), and therefore has a state. You get and set the content to update the rendered text on the display. When you set new text content the label automatically re-renders itself. (By calling *scheduleRepaint*)

Because the view is rendered asynchronously, its text content must be allocated on the heap. Therefore it uses the *String* as text storage. You can provide it with C strings, but these must be allocated inside the .rodata segment of your binary. (Meaning they are `static const`.)

Text Format

Currently there are only one font type. But the text color and font can be changed. You change these for parameters:

- Text font (including size)
- Text color
- Text background color (the color behind the characters)

Getting text dimensions

To help you layout your views, you can query the TextLabel of the current width and height of its contents. The methods *TextPixelWidth* and *TextPixelHeight*, will return the text's dimensions in pixels

- regardless of view rectangle. Also, you can use these methods before the view has been rendered.

Public Types

enum type `mono::ui::TextLabelView::TextAlignment`

Three ways of justifying text inside the TextLabel.

Values:

- Align text to the left
- Align text in the center
- Align text to the right

enum type `mono::ui::TextLabelView::VerticalTextAlignment`

The Vertical justification of the text, inside the label's Rect.

Values:

- Align the text at the top of the label

Align the text at in the middle of the label

Align the text at the bottom of the label

Public Functions

`mono::ui::TextLabelView::TextLabelView (String txt)`

Construct a text label with defined content, but no dimensions.

Before you can render the label you still need to set the view dimensions. This constructor take the *String* object as defined in the mono framework.

Parameters

- `txt` - The labels text content (as a mono lightweight string)

`TextLabelView::TextLabelView (const char *txt)`

Construct a text label with defined content, but no dimensions.

Before you can render the label you still need to set the view dimensions. This constructor takes a `static const C` string pointer that must *not* exist on the stack! (It must live inside the `.rodata` segment.

Parameters

- `txt` - A pointer to the static `const C` string (`.rodata` based)

`TextLabelView::TextLabelView (geo::Rect rct, String txt)`

Construct a label in a defined rectangle and with a string.

You provide the position and size of the label, along with its text content. You can call this constructor using a mono type string or a stack based C string - and it is automatically converted to a mono string:

```
int celcius = 22;
```

```
// char array (string) on the stack
char strArray[50];
```

```
// format the string content
sprintf(strArray,"%i celcius", celcius);
```

```
// construct the label with our stack based string
TextLabelView lbl(geo::Rect(0,0,100,100), strArray);
```

`TextLabelView::TextLabelView (geo::Rect rct, const char *txt)`

Construct a label in a defined rectangle and with a string.

You provide the position and size of the label, along with its text content. You can call this constructor using `static const C` string:

```
// construct the label with our stack based string
TextLabelView lbl(geo::Rect(0,0,100,100), "I am a .rodata string!");
```

`uint8_t TextLabelView::TextSize()`

`const`The text size will be phased out in coming releases. You control text by changing the font.

`mono::display::Color TextLabelView::TextColor()`

`const`Get the current color of the text.

`TextLabelView::TextAlignment` `TextLabelView::Alignment` ()
constGet the current horizontal text alignment.

`TextLabelView::VerticalTextAlignment` `TextLabelView::VerticalAlignment` ()
constGet the current vertical text alignment.

`uint16_t` `TextLabelView::TextPixelWidth` ()
constGet the width of the current text dimension.

`uint16_t` `TextLabelView::TextPixelHeight` ()
constGet the height of the current text dimension.

`const` `MonoFont` *`TextLabelView::Font` ()
constIf not NULL, then returns the current selected `MonoFont`.

`const` `GFXfont` *`mono::ui::TextLabelView::GfxFont` ()
constIf not NULL, then returns the current selected `GFXfont`.

`mono::geo::Rect` `TextLabelView::TextDimension` ()
constReturns the dimensions (*Size* and offset Point) of the text.

`void` `TextLabelView::setTextSize` (`uint8_t newSize`)
We will phase out this attribute in the coming releases. To change the font size you should rely on the font face.

If you set this to 1 the old font (very bulky) font will be used. Any other value will load the new default font.

`void` `TextLabelView::setTextColor` (`display::Color col`)

`void` `TextLabelView::setBackgroundColor` (`display::Color col`)

`void` `mono::ui::TextLabelView::setText` (`display::Color col`)
Set the text color.

`void` `mono::ui::TextLabelView::setBackground` (`display::Color col`)
Set the color behind the text.

`void` `TextLabelView::setAlignment` (`TextAlignment align`)
Controls text justification: center, right, left.

`void` `TextLabelView::setAlignment` (`VerticalTextAlignment vAlign`)
Set the texts vertical alignment: top, middle or bottom.

`void` `mono::ui::TextLabelView::setText` (`const char *text`)
Change the text content of the Text label, and schedules repaint.

This method updates the text that is rendered by the textlabel. It automatically schedules an incremental (fast) repaint.

Parameters

- `text` - The C string text to render

`void` `mono::ui::TextLabelView::setText` (*String text*)
Change the text content of the Text label, and schedules repaint.

This method updates the text that is rendered by the textlabel. It automatically schedules an incremental (fast) repaint.

Parameters

- `text` - The Mono string text to render

```
void mono::ui::TextLabelView::setText (const char *txt, bool resizeViewWidth)
```

```
void mono::ui::TextLabelView::setText (String text, bool resizeViewWidth)
```

```
void mono::ui::TextLabelView::setFont (MonoFont const &newFont)
```

Set a new font face on the label.

You can pass any MonoFont to the label to change its appearance. Fonts are header files that you must include yourself. Each header file defines a font in a specific size.

The header file defines a global `const` variable that you pass to to this method.

Parameters

- `newFont` - The mono-spaced to use with the textlabel

```
void mono::ui::TextLabelView::setFont (GFXfont const &font)
```

Set a new font face on the label.

You can pass any Adafruit GfxFont to the label to change its appearance. Fonts are header files that you must include yourself. Each header file defines a font in a specific size.

The header file defines a global `const` variable that you pass to to this method.

```
mono::String TextLabelView::Text ()
```

constGets the content of the text label.

```
void mono::ui::TextLabelView::scheduleFastRepaint ()
```

Repaints the view, using incremental repaints if possible.

This method might be faster than *scheduleRepaint*, since this repaint allows the text to be repainted incrementally. This means fast repaint of counters or fade animations.

If you experience rendering errors, you should use the normal *scheduleRepaint* method.

virtual Schedule this view for repaint at next display refresh. This method add the view to the display systems re-paint queue. The queue is executed right after a display refresh. This helps prevent graphical artifacts, when running on a single display buffer system. Because views have no state information, they do not know when to repaint themselves. You, or classes using views, must call this repaint method when the view is ready to be repainted.

virtual Repaint the view content, using the *View::painter*. Re-paint the view content. This method should be called when the view content has changed. You can call this method directly, but it might cause graphics artifacts because the display is not double buffered. Instead you should schedule a repaint by calling the *scheduleRepaint()* method. This method will schedule the repaint, right after the next display update. The display system will not schedule any repaints automatically. The view does not contain any state information, so you or other classes utilizing view must schedule repaints. In subclasses of *View*, this method *must* be overwritten.

Public Members

```
void TextLabelView::scheduleRepaint ()
```

This indicate if the next repaint should only repaint differences

```
void TextLabelView
```

```
void TextLabelView::setBackground
```

Set the color behind the text

Public Static Functions

static const MonoFont* StandardTextFont `mono::ui::TextLabelView::__DEPRECATED("use the`
This is the default font for all *TextLabelView*'s.

This points to the default Textlabel font. You can overwrite this in your own code to change the default appearance of all TextLabels.

You can also overwrite it to use a less memory expensive (lower quality) font face.

Public Static Attributes

const GFXfont* `mono::ui::TextLabelView::StandardGfxFont`

This is the default font for all *TextLabelView*'s.

This points to the default Textlabel font. You can overwrite this in your own code to change the default appearance of all TextLabels.

You can also overwrite it to use a less memory expensive (lower quality) font face.

Protected Functions

bool `TextLabelView::isTextMultiline()`
constCheck if the current text has newline characters.

This runs O(n)

Return

`true` is newlines are found

TouchCalibrateView

class Touch Calibration *View*, that calibrates the touch system. This calibration is stored in Mono persistent settings storage ISettings. If there already exists a calibration, then this calibration will be loaded and the `setCalibrationDoneCallback` handler will be called immediately after the call to *show*. **Public Functions**

`TouchCalibrateView::TouchCalibrateView()`
Construct a CalibrationView that takes the whole display.

`TouchCalibrateView::TouchCalibrateView (geo::Rect viewRct)`
Construct a CalibrationView inside a given rectangle.

Parameters

- `viewRct` - The rectangle on the display, where the view the shown

void `TouchCalibrateView::StartNewCalibration()`
Trigger a new calibration of the touch.

This will force at new calibration to be done. The Calibration will take over the touch input and screen and run its calibration routine.

virtual Shows (repaints) the view and insert into the responder chain.

`void TouchCalibrateView::show()`

See

View::show

virtual Internal touch handler, you should not overwrite this

Protected Functions

virtual Repaint the view content, using the *View::painter*. Re-paint the view content. This method should be called when the view content has changed. You can call this method directly, but it might cause graphics artifacts because the display is not double buffered. Instead you should schedule a repaint by calling the *scheduleRepaint()* method. This method will schedule the repaint, right after the next display update. The display system will not schedule any repaints automatically. The view does not contain any state information, so you or other classes utilizing view must schedule repaints. In subclasses of *View*, this method *must* be overwritten.

View

class Abstract interface for all UI Views, parent class for all views. Abstract *View* class/interface. All UI view/widgets that paint to the screen must inherit from this class. Views handle repaint queues, touch input and painting to the display buffer automatically. All views have a width and height, along with an absolute x,y coordinate that defines the upper left corner of the view rectangle. Views must not contain any state. They only draw data to the display. Therefore views might contain or have references to objects holding the actual state information. Some simple views, like *TextLabelView*, are exceptions to this rule, since it is highly convenient to let them hold some state. (Like text content.) Something on dependence of AppContext and Appctrl design pattern*ResponderView*

Public Types

`void TouchCalibrateView::respondTouchBegin(TouchEvent &event)`

`void TouchCalibrat`

Define the 4 different orientations of the display. The display controller applies the orientation transformation to real display. For the UI Views the coordinate system remains the same, it just changes width and height. The origin is always the top left corner (defined relative to gravity), no matter the physical orientation of mono's display.

Values:

- = 0 Expected standard orientation of mono, where the thick edge is at the bottom
- = 1 Upside-down of *PORTRAIT*, where the thick edge is at the top
- = 2 *PORTRAIT* rotated 90 degrees clockwise
- = 3 *PORTRAIT* rotated 90 degrees counter-clockwise

Public Functions

`View::View()`

Construct an empty view, you should not do this! You should not use *View* directly, subclass it instead.

`View::View(geo::Rect rect)`

Construct a view with dimensions, you should not do this! You should not use *View* directly, subclass it instead.

virtual Change the view's position on the screens coordinate system. Changes the view's position on the screen. Note that changing the position does not implicitly redraw the view. This means you will need to update the screen the affected areas to make the change visible.

void View::setPosition (geo::Point *pos*)

Parameters

- *pos* - The new position of the view

virtual Change the size (width, height) of the view. Changes the view's dimensions. The effect of size changes might depend on the specific view subclass. Some views might use their size to calculate their internal layout - others might only support fixed size. Note that changing the size here does not redraw the view. The screen needs to be redrawn to make the size change visible.

void View::setSize (geo::Size *siz*)

Parameters

- *siz* - The new size of the view

virtual Set the view's position and size, by providing a rectangle object. *Note* that this method does not repaint the view, you must do that explicitly.

void View::setRect (geo::Rect *rect*)

Parameters

- *rect* - The view rectangle, containing size and position

virtual Get the current position of the view's upper left corner.

mono::geo::Point &View::Position ()

Return

A reference to the current position

virtual Get the view's current size rectangle.

mono::geo::Size &View::Size ()

Return

A reference to the view's size rectangle

virtual const Get the views *view rect* This method returns a reference to the views current view rect.

virtual Schedule this view for repaint at next display refresh. This method add the view to the display systems re-paint queue. The queue is executed right after a display refresh. This helps prevent graphical artifacts, when running on a single display buffer system. Because views have no state information, they do not know when to repaint themselves. You, or classes using views, must call this repaint method when the view is ready to be repainted.

virtual const Returns the view's visibility. Get the view visible state. Non-visible view are ignored by the method *scheduleRepaint*. You change the visibility state by using the methods *show* and *hide*

const mono::geo::Rect &View::ViewRect ()

void View::scheduleRepaint ()

true if the view can/should be painted on the screen, false otherwise.

See

show

See

hide

virtual Set the view to visible, and paint it. Change the view's visibility state to visible. This means it can be scheduled for repaint by *scheduleRepaint*. This method automatically schedules the view for repaint.

void View::show()

See

hide

See

Visible

virtual Set the view to be invisible. Change the view's state to invisible. This method will remove the view from the *dirtyQueue*, if it has already been scheduled for repaint. Any calls to *scheduleRepaint* will be ignored, until the view is set visible again.

void View::hide()

See

show

See

Visible

Public Static Functions

uint16_t View::DisplayWidth()

Returns the horizontal (X-axis) width of the display canvas, in pixels. The width is always defined as perpendicular to gravity

uint16_t View::DisplayHeight()

Returns the vertical (Y-axis) height of the display canvas, in pixels. The height axis is meant to be parallel to the gravitational axis.

View::Orientation View::DisplayOrientation()

Returns the current physical display orientation of the display The orientation is controlled by the IDisplayController

Public Static Attributes

uint32_t View::RepaintScheduledViewsTime

The CPU time used to repaint the latest set of dirty views. This measure includes both the painting algorithms and the transfer time used to communicate with the display hardware.

Protected Functions

void View::callRepaintScheduledViews()

A member method to call the static method *repaintScheduledViews*.

See

repaintScheduledViews

virtual = 0Repaint the view content, using the *View::painter*. Re-paint the view content. This method should be called when the view content has changed. You can call this method directly, but it might cause graphics artifacts because the display is not double buffered. Instead you should schedule a repaint by calling the *scheduleRepaint()* method. This method will schedule the repaint, right after the next display update. The display system will not schedule any repaints automatically. The view does not contain any

state information, so you or other classes utilizing view must schedule repaints. In subclasses of [View](#), this method *must* be overwritten.

Protected Attributes

void `mono::ui::View::repaint()` geo::Rect mono::ui::

The rect defines the position and size of this view on the screen. This defines where the view rectangles upper left corner is situated, and the width and height in pixels.

bool `mono::ui::View::isDirty`

Indicate is this view should be repainted on next display refresh.

bool `mono::ui::View::visible`

Views can be visible or non-visible (hidden). When a view is *not* visible [scheduleRepaint](#) will ignore requests.

You should use the methods [show](#) and [hide](#) to toggle visibility.

See

[show](#)

See

[hide](#)

See

[Visible](#)

Protected Static Functions

void `View::repaintScheduledViews()`

This class method will run through the scheduled re-paints queue and call the [repaint](#) method on all of them.

This method is called automatically by the display system, you do not need to call it yourself.

Protected Static Attributes

`mono::display::DisplayPainter` `View::painter`

Global [View](#) painter object. Once the first [View](#) is included in the code, this painter is initialized on the stack. Every view object uses this painter to paint itself.

The painter is initialized with the display controller of the application context. If you want views to draw themselves on another display, you must subclass or change the current display controller of the `mono::ApplicationContext` object.

`mono::GenericQueue<View>` `View::dirtyQueue`

The global re-paint queue.

When you call the [scheduleRepaint](#) method, your views are added to the re-paint queue.

IO

DigitalOut

class Mono variant of `mbed::DigitalOut` class, adding `setMode()` method. This class is an extension of *mbed's* `DigitalOut` class, adding an additional constructor and public method. The additional function is the possibility to change the pin drive mode, such that it does not need to be strong drive (CMOS). This class allows you to set the mode to `PullUp` / `PullDown` / `OpenDrain`. To see how you use the `DigitalOut` class, please refer to the [mbed documentation page](#) **Public Functions**

`DigitalOut::DigitalOut` (`PinName pin`, `int value`, `PinMode mode`)

Construct a output pin with a specific output mode, other than the default strong drive (CMOS).

WARNING: The mode is set *AFTER* the pin value. You should account for this possible hazard, by setting a sound initial value.

Parameters

- `pin` - The pin identifier
- `value` - the initial value of the pin.
- `mode` - the initial drive mode of the pin

`void DigitalOut::setMode` (`PinMode mode`)

Set the pins drive mode.

Use this method to enable resistive pull up/down or open drain modes.

Parameters

- `mode` - The pin drive mode to use

File

class A cloolection of static methods relating to *File* I/O. This class contains a convenient handful of static methods to do common file operations. These methods abstract away the crude API's of *libc's* `stdio.h` functions. **Public Static Functions**

`bool File::exists` (`String path`)

Check if a file exists.

Determine if a file exists at a given path.

Return

true if file exists, false otherwise

Parameters

- `path` - The file's path (relative or absolute)

`size_t File::size` (`String path`)

Return a file's size in bytes.

This is a short hand method, that uses `stdlib` routines to get a file's size. You provide the path to the file.

If the size cannot be determined (file does not exists), this method will return 0.

Return

The file size in bytes or 0 on error (or empty file)

Parameters

- `path` - The relative or absolute path to the file

`mono::String File::readFirstLine (String path, int maxLength, char endLineDelimiter)`

Read and return the first line in a text file.

Read the first line of text in the file, defined by the given path. **It is very important the file is a text file!** This method will (by default) search for the first occurrence of a newline character. All bytes before the newline is returned. (This means the newline character itself is not part of the returned.)

MEMORY USAGE: Be aware that the returned string is contained in memory. Be cautious not to exhaust all RAM, by loading in a lot of text. Limit the maximum memory used by setting the optional `maxLength` argument.

If the file does not exist, an empty string is returned.

Return

A string with the first line of text

Parameters

- `path` - The path to the text file
- `maxLength` - **OPTIONAL:** Set to a positive integer, to load only that many bytes.
- `endLineDelimiter` - **OPTIONAL:** You can provide a custom stop delimiter, if you so not want .

`bool File::writeString (String text, String path)`

Write a string to a file (destructive)

You provide a text string, that is written to a provided file path. Any exists file at the path gets overwritten. The complete content of the string is written.

Return

`true` upon success, `false` otherwise

Parameters

- `text` - The string data to write to a file
- `path` - The file's path

`bool File::appendString (String text, String path)`

Append a text string to the end of a file.

Write a text string to a file (appending), without overwriting any text in the file. Only characters present in the string are written to the file. This method does not automatically insert newlines. If you want to append a line of text see [appendLine](#)

If the file does not exist, it is created and the text is written to the beginning of the file.

Return

`true` on success. `false` otherwise.

Parameters

- `text` - The text string to append to the file
- `path` - The path to the file that is appended

`bool File::appendLine` (String *text*, String *path*, **const** char **lineDelimiter*)

Append a line of text to the end of a file.

Append a text line (string and a newline character), to the end of the defined file. By default the newline sequence is `\n`, but you can overwrite this behavior using the third optional argument.

This method ensure that a subsequent string written to the file, will begin on a new line. If the provided text string already ends with a newline sequence, the outline will be 2 lines skipped.

If the destination file does not exist, it is created.

Return

`true` on success, `false` otherwise

Parameters

- `text` - The text string to write to the file
- `path` - The file destination path
- `lineDelimiter` - Optional: Define a custom new line sequence

FileSystem

class Mono specific version of mbed's `SDFileSystem` class. Access the filesystem on an SD Card using standard C lib functions: `fopen` `fread` `fwrite` `fclose`

You should include *stdio.h* and use I/O functions as in normal C on a OS.

Reset and wake

You should atke care not to write to the SD card immediately after a reset or wake up. SD cards might need to time from power-on to they react on SD initialization sequence. Therefore you should at least wait some milliseconds to allow the SD to be powered, before you start to write data.

Example

You should this class as a member of your global *AppController* class, such that it is first initialized when your *AppController* is conctructed.

Add as a variable on *AppController*:

```
•class AppController : public mono::IApplication
{
    public:
        mono::io::FileSystem fs;
}
```

Then call the designated initializor oin your *AppController*'s constructor:

```
AppController::AppController() :
    fs("sd")
{
    // ...
}
```

Now the SD card is mounted at: /sd

This class adds support for sleep mode to *mbed* FS functionality.

Public Functions

`FileSystem::FileSystem()`

Construct an invalid file system driver

`FileSystem::FileSystem(const char *mountPoint)`

Initialize the *File* system for a attached SD Card.

virtual Called right before the MCU goes into sleep mode. You can override this method to get sleep notifications. Before the CPU stop executing intructions and goes into low power sleep mode, this method gets called. Use this method to prepare your data or MCU components for sleep. Help preserve battery by power down any peripheral, that is not needed during sleep. This method can be called many times during your apps life cycle. (That is between resets.) After each sleep period, when the MCU wakes the *onSystemWakeFromSleep* is guaranteed to be called.

virtual Override to get notified when system wakes from sleep. You can override this method to get wake-up notifications. When the CPU starts executing intructions and the power system has powered up all peripherals - this method gets called. Use this method to setup your app to resume after sleep mode. This method can be called many times during your apps life cycle. (That is between resets.)

FilteredAnalogIn

`void FileSystem::onSystemEnterSleep()`

`void FileSystem::onSy`

class A low pass filtered analog input pin.

This is a class built upon the standard *mbed::AnalogIn* class, that include running average low pass filtering.

Filtered analog inputs

In almost all cases you should low pass filter an incoming analog signal, to remove high frequency noise. This class is a convenient way to do that. It uses the *RunningAverageFilter* to filter all incoming samples.

Just like the *RunningAverageFilter*, this class is templated with the length of the low pass filter.

Example

Here we create 32 sample long filter, attached to the 3.5mm jack connector input pin:

```
FilteredAnalogIn<32> input(J_RING1);
uint16_t filteredSample = input.read_u16();
```

See

RunningAverageFilter

Public Functions

`mono::io::FilteredAnalogIn::FilteredAnalogIn(PinName pin)`

Create an AnalogIn, connected to the specified pin.

Parameters

- *pin* - AnalogIn pin to connect to

```
float mono::io::FilteredAnalogIn::read()
```

Read the input voltage, represented as a float in the range [0.0, 1.0].

The value is filtered through a running average filter Complexity: O(1)

Return

A floating-point value representing the current input voltage, measured as a percentage

```
unsigned short mono::io::FilteredAnalogIn::read_u16()
```

Read the input voltage, represented as an unsigned short in the range [0x0, 0xFFFF].

The value is filtered through a running average filter Complexity: O(1)

Return

16-bit unsigned short representing the current input voltage, normalised to a 16-bit value

HysteresisTrigger

class Hysteresis or Schmitt trigger function class. This class implements a simple hysteresis check that allows you to continuously provide samples, that is checked against lower and upper boundaries. The class will call your provided callback methods whenever the hysteresis trigger changes from one boundary to another. A hysteresis trigger is a sound way of change program states, on base of an analog signal value. This could be a light sensor or battery voltage monitoring. A hysteresis loop will trigger the state changes on the base signal value changes. *Hysteresis* This is a mechanism that defines an upper and lower threshold in signal values, say 1000 and 100 for example. Whenever the input signal rises above the upper threshold (1000 in this example), the hysteresis triggers a state change to an *upper* state. Low the signal value must fall below the lower threshold (100) to trigger an equivalent *lower state* change. This process repeats, now the signal must exceed the upper threshold again to trigger the *upper state* change. Read more on hysteresis triggers here: https://en.wikipedia.org/wiki/Hysteresis#Control_systems *Example* Here we setup a hysteresis trigger object, that triggers the callbacks then a signal values exceed or fall under certain thresholds:

```
HysteresisTrigger sensorTrigger(10000, 3000);

sensorTrigger.setUpperTriggerCallback(this, &MyApp::clearErrorState);
sensorTrigger.setLowerTriggerCallback(this, &MyApp::setErrorState);

// call this in a loop
uint16_t value = getSampleValue(); // get some signal value
sensorTrigger.check(value);
```

See

FilteredAnalogIn

Public Functions

```
mono::io::HysteresisTrigger::HysteresisTrigger(int upper, int lower, TriggerTypes
InitialState)
```

Construct a new hysteresis object with specific boundaries.

Setup a hysteresis object with defined upper and lower threshold values.

Parameters

- upper - The upper value (ceil) that triggers a state change

- `lower` - The lower value (floor) that triggers a state change
- `InitialState` - Optional: The start state of the hysteresis object

`bool mono::io::HysteresisTrigger::check (int value)`

Provide a sample value to the hysteresis, that might trigger a state change.

Pass sample values of the signal source to this method, and it will be compared against the upper and lower threshold of the hysteresis.

If the value exceeds a threshold one of the state trigger handlers is called.

Return

`true` if an state change was triggered, `false` otherwise

Parameters

- `value` - The signal sample to check against the threshold

`TriggerTypes mono::io::HysteresisTrigger::NextTriggerType ()`

Return the current next state of the hysteresis loop.

This means if the signal was previously above the *upper threshold* the next state change will be from *upper* to *lower*, eg. `TRIG_LOWER_NEXT`

`void mono::io::HysteresisTrigger::setNextTrigger (TriggerTypes next)`

Set the hysteresis next state trigger.

You can force the hysteresis loop into a given state, by providing its *next state* variable here.

Parameters

- `next` - The next state change that should be triggered

`template <typename Context>`

`void mono::io::HysteresisTrigger::setUpperTriggerCallback (Context * cnxt, void (Context`

Set the upper threshold callback.

Provide a method that gets called whenever the hysteresis loop triggers for value above the *upper threshold*

Parameters

- `cnxt` - The context of the method to call, usually it is the `this` pointer.
- `memptr` - Function pointer to the method on the context class.

`template <typename Context>`

`void mono::io::HysteresisTrigger::setLowerTriggerCallback (Context * cnxt, void (Context`

Set the lower threshold callback.

Provide a method that gets called whenever the hysteresis loop triggers for value above the *lower threshold*

Parameters

- `cnxt` - The context of the method to call, usually it is the `this` pointer.
- `memptr` - Function pointer to the method on the context class.

OneWire

class A one-wire communication line with a sensor that has a data line with pull-up. The sensor is expected to be activated by pulling down the wire and the releasing it. The sensor is then expected to send data, distinguishing between zero bits and one bits by two different period lengths of logic-high on the wire. Jens Peter Secher

Public Functions

OneWire::OneWire (PinName *wire*, uint32_t *usPeriodLow*, uint32_t *usPeriodHiZero*, uint32_t *usPeriodHiOne*, uint32_t *usFullReadTimeOut*)

Setup a one-wire communication line with a sensor that has a data line with pull-up. The sensor is expected to be activated by pulling down the wire and the releasing it. The sensor is then expected to send data, distinguishing between zero bits and one bits by two different period lengths of logic-high on the wire.

...*DigitalOut* because they use the same pin, and the initialisation sets the pin mode.

Parameters

- *wire* - Pin to communication through.
- *usPeriodLow* - Expected logic-low length in μ s common to all bits.
- *usPeriodHiZero* - Expected logic-high length of a zero bit in μ s.
- *usPeriodHiOne* - Expected logic-high length of a one bit in μ s.
- *usFullReadTimeOut* - Timeout in μ s for a complete read.

template <typename Class>

void **mono::io::OneWire::send**(uint32_t *usInitialPeriodLow*, size_t *skipBits*, uint8_t * *buffer*, size_t *length*, void (**obj*)(void), void (**member*)(void))

Tell the sensor to start sending data. Reading from the sensor stops when the buffer is filled up or when timeout is reached, and then the buffer will contain whatever bits that have been read so far.

Parameters

- *usInitialPeriodLow* - Period in μ s that wire will be pulled low to initiate sensor.
- *skipBits* - Number of initial handshake bits to ignore.
- *buffer* - Buffer where the received bytes will be stored.
- *length* - Buffer length in bytes.
- *obj* - Object that will receive notification when read is done.
- *member* - Object member that will receive notification when read is done.

QueueInterrupt

class An queued input pin interrupt function callback handler This class represents an input pin on mono, and provides up to 3 different callback handler functions. You can installed callback function for rising, falling or both edges. *Queued interrupts* In Mono framework a queued interrupt is handled inside the normal execution context, and not the hardware interrupt routine. In embedded programming it is good practice not to do any real work, inside the hardware interrupt routine. Instead the best practice method is to set a signal flag, and handled the event in a run loop. *QueueInterrupt* does this for you. The *rise*, *fall* and *change* callback are all executed by the default mono run loop (*AppRunLoop*) You can safely do heavy calculations or use slow I/O in the callback routines you assign to QueueInterrupt! *Latency* The run loop might handle the interrupt callback some time after it occur, if it is busy doing other stuff. Therefore you cannot expect to have your callback executed the instant the interrupt fires. (If you need that use DirectInterrupt) *QueueInterrupt* holds the latest interrupt trigger timestamp, to help you determine the latency between the actual interrupt and you callback. Also, many interrupt triggering signal edges might occur, before the run loop executes you handler. The timestamp only shows the latest one. **Public Functions**

`QueueInterrupt : : QueueInterrupt (PinName inputPinName, PinMode mode)`

Assign a queued inetrrupt handler to a physical pin

Parameters

- `inputPinName` - The actual pin to listen on (must be PORT0 - PORT15)
- `mode` - OPTIONAL: The pin mode, default is Hi-Impedance input.

`void QueueInterrupt : : DeactivateUntilHandled (bool deactive)`

Set this property to `true`, to turn off incoming interrupts while waiting for the run loop to finish process a pending interrupt.

If you want to do heavy calculations or loading in your interrupt function, you might want to not queue up new interrupts while you process a previous one.

Parameters

- `deactive` - OPTIONAL: Set this to false, to *not* disable interrupts while processing. Default is `true`

`bool QueueInterrupt : : IsInterruptsWhilePendingActive ()`

constGet the state of the *DeactivateUntilHandled* property. If `true` the hardware interrupt is deactivated until the handler has run. If `false` (the default when constructing the object), all interrupt are intercepted, and will be handled. This means the handler can be executed two times in row.

Return

`true` if incomming interrupt are displaed, until previous is handled.

`void QueueInterrupt : : setDebounce (bool active)`

Enable/Disable interrupt de-bounce.

Switches state change might cause multiple interrupts to fire, or electrostatic discharges might cause nano seconds changes to I/O lines. The debounce ensures the interrupt will only be triggered, on sane button presses.

`void QueueInterrupt : : setDebounceTimeout (int timeUs)`

Change the timeout for the debounce mechanism.

Parameters

- `timeUs` - The time from interrupt to the signal is considered stable, in micro-seconds

`void mono::QueueInterrupt::rise (void (*fptr)) void`
Attach a function to call when a rising edge occurs on the input

Parameters

- `fptr` - A pointer to a void function, or 0 to set as none

`template <typename T>`

`void mono::QueueInterrupt::rise (T * tptr, void(T::*)(void) mptr)`
Attach a member function to call when a rising edge occurs on the input

Parameters

- `tptr` - pointer to the object to call the member function on
- `mptr` - pointer to the member function to be called

`void mono::QueueInterrupt::fall (void (*fptr)) void`
Attach a function to call when a falling edge occurs on the input

Parameters

- `fptr` - A pointer to a void function, or 0 to set as none

`template <typename T>`

`void mono::QueueInterrupt::fall (T * tptr, void(T::*)(void) mptr)`
Attach a member function to call when a falling edge occurs on the input

Parameters

- `tptr` - pointer to the object to call the member function on
- `mptr` - pointer to the member function to be called

`uint32_t QueueInterrupt::FallTimeStamp ()`

On fall interrupts, this is the μ Sec. ticker timestamp for the falling edge inetrrupt. You can use this to calculate the time passed from the interrupt ocured, to the time you process it in the application run loop.

Return

The ticker time of the falling edge in micro seconds

`uint32_t QueueInterrupt::RiseTimeStamp ()`

On rise interrupts, this is the μ Sec. ticker timestamp for the rising edge inetrrupt. You can use this to calculate the time passed from the interrupt ocured, to the time you process it in the application run loop.

Return

The ticker time of the rising edge in micro seconds

`bool QueueInterrupt::Snapshot ()`

The pin value at the moment the H/W interrupt triggered The callback might be executed some time after the actual inetrrupt ocured. THis method return the pin state at the moment of the interrupt.

Return

The pin state, at the time of the interrupt

```
bool QueueInterrupt::willInterruptSleep ()
    constInterrupt will abort sleep mode to run handler.
```

If interrupt fires during sleep mode, Mono will wake up to service the handler in the run loop

```
void QueueInterrupt::setInterruptsSleep (bool wake)
    Set if this interrupt will wake Mono from sleep.
```

Set to `true` to wake from sleep and continue the run loop upon interrupts.

Parameters

- `wake` - `true` to enabled wake-from-sleep

```
template <typename T>
```

```
void mono::QueueInterrupt::change (T * tptr, void(T::*) (void) mptr)
    Attach a member function to call when a rising or falling edge occurs on the input
```

Parameters

- `tptr` - pointer to the object to call the member function on
- `mptr` - pointer to the member function to be called

Protected Functions

virtual This is the method that gets called by the run loop. *NOTE* that this is not an interrupt function, you can do stuff that take some time.

RunningAverageFilter

```
void QueueInterrupt::taskHandler ()
    class A basic running average low pass filter.
```

```
template <uint16_t Length>
```

This is a standard moving window / moving average low pass filter, to remove noise from analog input signals. You append raw samples to the filter, and read the filtered output by *value* method.

Filter sample values are always unsigned 16-bit integers, since this is the return type of `mbed::AnalogIn`

Moving Average Low pass filtering

This digital filter type is quite common, and works well in most cases. You control the filter amount by varying the length of the filter. Longer filters does more aggressive low pass filtering. A filter with the length of 1, does no filtering at all.

The length of a filter object is determined by C++ templating and are fixed for the life-time of the filter.

Appending the first sample

If you create a filter with the length of 100 samples, then the filter will need an initialization period of 100 samples before its output is correctly filtered. To account for this issue you can add the first sample using the *clear* method, providing it the value of the first value.

This will in most cases help you to not see filter values rise slowly until it stabilizes after 100 samples.

Complexity

Filter operations are normally $O(1)$, that is constant time. You can append new samples and read filter output in $O(1)$. However, clearing the filter or calculating the variance of the samples is $O(n)$.

To get even faster output calculations use a filter length that is a multiple of 2, like 8, 16, 32, 64 etc. This reduces the CPU's integer division to simple shift operations.

Example

```
med::AnalogIn adc(J_RING1);
RunningAverageFilter<128> filter(adc.read_u16());

filter.append(adc.read_u16());
uint16_t lpValue = filter.value();
```

See

FilteredAnalogIn

Public Functions

`mono::io::RunningAverageFilter::RunningAverageFilter (uint16_t initialValue)`
Construct a filter with an optional initial value.

Parameters

- `initialValue` - Provide an initial filter value, for quick stabilization

`uint16_t mono::io::RunningAverageFilter::append (uint16_t val)`
Add a sample to the filter.

`void mono::io::RunningAverageFilter::clear (uint16_t initialValue)`
Reset the filter setting all samples to a new value.

Use this method to reset the filter, perhaps after a period of inactivity in the sampling of your signal source.

`uint16_t mono::io::RunningAverageFilter::value ()`
constGets the current low pass filtered value.

`uint16_t mono::io::RunningAverageFilter::sum ()`
constGet the sum of all samples in the filter.

`uint16_t mono::io::RunningAverageFilter::variance ()`
constGet the variance of the filter samples.

`uint16_t mono::io::RunningAverageFilter::operator[] (uint16_t indx)`
constGet a specific sample from the filters array of samples.

`uint16_t mono::io::RunningAverageFilter::length ()`
constGet the length of the filter.

Serial

class Power / USB aware serial port class, that uses only the USB UART. This is the implementation of communication via USBUART. It builds upon mbed's `mbed::Serial` class. Because mbed is not aware of changes power in power and charging states, this class acts as a wrapper around the `mbed::Serial` class. *Restrictions* This *Serial* class will communicate with the USBUART only when the USB power is connected. All data consumed by the class, then the USB power is absent, is ignored. To see how you use the *Serial* port, please refer to the [mbed documentation](#) **Public Functions**

`Serial::Serial ()`
Create a new instance of the USB UART port.

This is the only available constructor, because the RX and TX lines are hardwired. Ideally you should not initialize your own USB serial port, but use the systems default I/O streams.

```
bool Serial::DTR()
```

See the status for the Data Terminal Ready signal.

Return

true if DTR is set

```
bool Serial::IsReady()
```

Test if the USB is connected (powered) and mounted CDC device is a CDC device by the host.

Before any data can be sent via the USB *Serial* link, the host (computer) must have emunerated Mono as a USB CDC device. Also, to ensure the USB cable is connected, the USB power is monitored. These two conditions must be met, before this method returns true.

Return

true if USB power is on and USB is enumerated on host

IWifi

class Interface for Wifi setup methods. Subclasses that implement this interface will control Wifi hardware and encapsulate connection processes inside this convenient interface. You need only to call the *connect* method and wait the *connected* callback event. The implementation of this interface must take care of associating with a Wifi Access Point and getting an dynamic IP address from AP's DHCP. *Sleep mode* When mono enters sleep, the wifi will loose power. Upon wake-up this interface will notify you with a *disconnected* callback. *Example* You should have only one *IWifi* based object in your application. Therefore add it as a member of your *AppController* class:

```
class AppController : public mono::IApplication
{
    public:
        mono::io::Wifi wifi;
}
```

Then, when you need network access the first time, just call:

```
AppController::monoWakeFromReset()
{
    bool connecting = wifi.connect();

    if (!connecting)
        printf("Error, can not connect");
}
```

Public Types

enum type `mono::io::IWifi::NetworkEvents`

The Wifi connection status event types.

Values:

- When Wifi is connected and ready
- Wifi failed to connect to access point
- Wifi was disconnected

Public Functions

`virtual = 0` Connect to the Wifi network. Use the parameters provided in the constructor to connect to the configured access point. This method will also obtain an IP address from the AP's DHCP server. You can call this method multiple times, even if you are already connected. The method will simply ignore attempt to connect, when already connected.

`bool mono::io::IWifi::connect ()`

Return

`true` is configuration is valid and Wifi hardware is OK.

`virtual const = 0` Return `true` Wifi is connected. This will return `true` only if a connection is currently present. This means it will return `false` even if the Wifi hardware is currently in process of connecting to an access point or obtaining an IP address.

`bool mono::io::IWifi::isConnected ()`

`void mono::io::IWifi::setConnectedCallback (void (Context::*) (void) memptr)`

Set a callback for the *connected* event.

Set a callback function to get notified when network access is established.

Parameters

- `cfunc` - A pointer to the C function that handles the event

`template <typename Context>`

`void mono::io::IWifi::setConnectedCallback (Context * cnxt, void (Context::*) (void) memptr)`

Set a callback for the *connected* event.

Set a callback function to get notified when network access is established.

Parameters

- `cnxt` - A pointer to the object that handles the event
- `memptr` - A pointer to the method on the class that handles the event

`void mono::io::IWifi::setConnectErrorCallback (void (*cfunc)) void`

Set a callback for the *connect failed* event.

Set a callback function to get notified when network access could not be established, and the Wifi hardware has given up.

Parameters

- `cfunc` - A pointer to the C function that handles the event

`template <typename Context>`

`void mono::io::IWifi::setConnectErrorCallback (Context * cnxt, void (Context::*) (void) memptr)`

Set a callback for the *connect failed* event.

Set a callback function to get notified when network access could not be established, and the Wifi hardware has given up.

Parameters

- `cnxt` - A pointer to the object that handles the event
- `memptr` - A pointer to the method on the class that handles the event

`template <typename Context>`

void mono::io::IWifi::setDisconnectedCallback (Context * cnxt, void (Context::*)(void) m
Set a callback for the *disconnected* event.

Set a callback function to get notified when network access is terminated.

this event will always fire after mono comes out of sleep, if network was established when it went to sleep.

Parameters

- *cnxt* - A pointer to the object that handles the event
- *memptr* - A pointer to the method on the class that handles the event

void mono::io::IWifi::setDisconnectedCallback (void (**cfunc*)) void
Set a callback for the *disconnected* event.

Set a callback function to get notified when network access is terminated.

this event will always fire after mono comes out of sleep, if network was established when it went to sleep.

Parameters

- *cfunc* - A pointer to the C function that handles the event

void mono::io::IWifi::setEventCallback (void (**cfunc*)) *NetworkEvents*
Set a callback for the all events.

Set a callback function to get notified when network access state is changed.

Parameters

- *cfunc* - A pointer to the C function that handles the event

template <typename Context>
void mono::io::IWifi::setEventCallback (Context * cnxt, void (Context::*)(*NetworkEvents*
Set a callback for the all events.

Set a callback function to get notified when network access state is changed.

Parameters

- *cnxt* - A pointer to the object that handles the event
- *memptr* - A pointer to the method on the class that handles the event

Network

DnsResolver

class Resolve a domain name to a IPv4 address. The *DnsResolver* class converts a domain name to an IP Address (A record) **Public Functions**

DnsResolver::DnsResolver ()
Create an empty dns resolve request

DnsResolver::DnsResolver (String *aDomain*)
Create a DNS resolve request, and fetch the IP address right away.

Note The request runs asynchronously, make sure to install the completion callback right after construction.

See

setCompletionCallback

mono::String DnsResolver::IpAddress()

constGet the resolved IP address.

Get the resolved IP address. **Note** This is first valid after the request is completed.

Return

A text representation of the resolved IP address, or an empty string.

See

IsCompleted

mono::String DnsResolver::DomainName()

constGets the Dns request domain name, that is-to-be or has been resolved.

Return

The domain name

HttpClient

class A HTTP GET Request Client. This class implements a HTTP GET Request, and provide you with a notification, when the response arrives. You provide an URL and one or more callback functions, for getting the response data. *Example*

```
HttpClient client("http://www.myip.com");
```

```
client.setDataReadyCallback<MyClass>(this, &MyClass::httpData);
```

Now your method `httpData` is called when response data arrives. Note that the callback can be called multiple times. A flag in the returned data object indicates, when this was the last data to be received.

DNS resolution is done automatically. You can also change the HTTP port to something other than 80 by providing an alternative port via the URL: . Likewise, you can use raw IPv4 addresses: .

Headers

You can provide additional headers for the request by using the constructors optional second argument. Just remember to end each header line with a . Like this: `Accept: text/html\r\n`

Error handling

This client class handles error that arise from:

- DNS lookup failed (no such domain name)
- No server response (host does not respond)

This class does not handle errors based on HTTP status codes or content types.

If you wish to handle errors you should setup a callback for the *setErrorCallback*

Public Functions

`HttpClient::HttpClient()`

Construct an empty invalid client object.

`HttpClient::HttpClient (String anUrl, String headers)`

Construct a new HTTP request to a given URL.

To receive the response, also setup the callback *setDataReadyCallback*

Parameters

- `anUrl` - The URL string
- `headers` - Any additional headers to add

`template <typename Owner>`

`void mono::network::HttpClient::setDataReadyCallback (Owner * cnxt, void (Owner::*) (const`

Provide your callback function to handle response data.

Parameters

- `cnxt` - A pointer to the callback member function context (the `this` pointer)
- `memPtr` - A pointer to the member function, that is the callback

`void mono::network::HttpClient::setDataReadyCallback (void (*cfunc)) const HttpRe-`
sponseData&

Provide your callback function to handle response data.

Parameters

- `cfunc` - A pointer to the function that handles the data response

Protected Functions

`void HttpClient::dnsResolutionError (INetworkRequest::ErrorEvent *evnt)`

Error callback for dns resolver

virtual dns resolver completion event handler

`void HttpClient::dnsComplete (INetworkRequest::CompletionEvent *evnt)`

`void HttpClient::I`

when the HTTP GET frame completes, check for error

class Class to represent HTTP response data. This class defines the response data chunk return by an HTTP request. It defines only public accessible properties, that indicate the response state and the raw data. **Public Members**

HttpClient *`mono::network::HttpClient::HttpresponseData::Context`

Pointer to the originating *HttpClient* object.

String `mono::network::HttpClient::HttpresponseData::bodyChunk`

The raw HTTP response chunk. More chunk might arrive later.

`bool mono::network::HttpClient::HttpresponseData::Finished`

true if this response chunk is the final.

HttpPostClient

class HTTP POST Request Client. This class can send a HTTP POST request with body data. You provide callbacks to provide the body length (in bytes) and content. This is to avoid caching the content, and save memory. Before this class works you must setup a working wifi connection. *Example*

```
HttpPostClient client("http://www.postrequest.org/");
client.setBodyLengthCallback<MyClass>(this, &MyClass::getDataLength);
client.setBodyDataCallback<MyClass>(this, &MyClass::getData);
client.post();
```

This will post the request with no default `Content-Type` header. You can provide your own `Content-Type` by using the second (optional) argument in the constructor. If you do, remember to insert a `\r\n` at the end of the header string.

Trailing white space

Due to a bug in the Redpine module protocol, the post body risk getting trailing whitespace characters (spaces: 0x20). There can up to 3 trailing spaces: . The reason is the redpine frame must end on a 4-byte byte boundry. Instead of appending NULL characters, we append spaces. Most HTTP servers does not respond well to NULL character being part of a text body.

This is of course an issue when sending binary data via POST requests. In these cases you should (at this point) avoid binary formats. (Maybe use base64 encoding.)

Public Functions

`HttpPostClient::HttpPostClient()`

Construct an empty invalid HTTP client.

`HttpPostClient::HttpPostClient(String anUrl, String headers)`

Construct a client with a target URL.

Headers like `Host` and `Content-Length` are created automatically. The DNS resolution of the host-name is also done automatically.

The actual POST request is not execute before you call *post*. However, DNS resolution is started immediately.

Parameters

- `anUrl` - THE URL to call
- `headers` - Optional extra headers to add to the request

template <typename Class>

void `mono::network::HttpPostClient::setBodyLengthCallback(Class * context, uint16_t (C1`

Callback for providing the length of the HTTP POST body.

You should provide a efficient method of getting the body length, since this callback is used multiple times under the request execution.

NOTE: The length returned *must* be the real HTTP *Content-length*. This means any C string terminator should *not* be included.

Parameters

- `context` - pointer to the object to call the member function on
- `method` - pointer to the member function to be called

```
void mono::network::HttpPostClient::setBodyLengthCallback (uint16_t  
                                                         (*cfunc)) void
```

Callback for providing the length of the HTTP POST body.

You should provide a efficient method of getting the body length, since this callback is used multiple times under the request execution.

NOTE: The length returned *must* be the real HTTP *Content-length*. This means any C string terminator should *not* be included.

Parameters

- cfunc - Pointer to the function to call for request data length

```
template <typename Class>
```

```
void mono::network::HttpPostClient::setBodyDataCallback (Class * context, void (Class::*)  
                                                         Callback for providing the body content of the HTTP POST.
```

The internals of the request will ensure the provided `char*` is large enough to hold the size of the HTTP body, including the the string terminator character.

The buffer pointer provided, points to a buffer that is the length you provided from [setBodyLengthCallback](#) plus a terminator character.

Parameters

- context - pointer to the object to call the member function on
- method - pointer to the member function to be called

```
void mono::network::HttpPostClient::setBodyDataCallback (void (*cfunc)) char *  
                                                         Callback for providing the body content of the HTTP POST.
```

The internals of the request will ensure the provided `char*` is large enough to hold the size of the HTTP body, including the the string terminator character.

The buffer pointer provided, points to a buffer that is the length you provided from [setBodyLengthCallback](#) plus a terminator character.

Parameters

- cfunc - Pointer to the function to call for request body data

```
void HttpPostClient::post ()
```

Execute the POST request.

Commit the request sending it to the server.

Protected Functions

virtual dns resolver completion event handler

INetworkRequest

class Interface class that defines a generic network request. Network request have a notion of 3 states: setup in progress completed error

Public Functions

void `HttpPostClient::dnsComplete` (`INetworkRequest::CompletionEvent *evnt`)

`INetworkRequest::State`

constGet the current state of the underlying network request.

PUBLIC METHODS.

Return

The current state of the request

bool `INetworkRequest::IsCompleted` ()

constCheck to see if the underlying network request has finished and are in the `COMPLETE_STATE`

Return

true if the request is completed

bool `INetworkRequest::HasFailed` ()

constCheck to see if the underlying network has failed, and are in `ERROR_STATE`

Return

true if the request has failed

template <typename Owner>

void `mono::network::INetworkRequest::setCompletionCallback` (`Owner * cnxt`, void(`Owner::*`)

Set the completion handler callback function.

To receive a notification callback when the underlying network request finishes successfully, use this method to install a callback function.

Parameters

- `cnxt` - A pointer to the callback context (the `this` pointer)
- `memPtr` - A pointer to the context class' member function

void `mono::network::INetworkRequest::setCompletionCallback` (void

(**cfunc*) `CompletionEvent`
*

Set the completion handler callback function.

To receive a notification callback when the underlying network request finishes successfully, use this method to install a callback function.

Parameters

- `cfunc` - A pointer to the function handling completion

template <typename Owner>

void `mono::network::INetworkRequest::setErrorCallback` (`Owner * cnxt`, void(`Owner::*`) (`Er`

Set an error handler callback function.

To receive a notification callback if this request fails, set a callback function here.

Parameters

- `cnxt` - A pointer to the callback context (the `this` pointer)

- `memPtr` - A pointer to the context class' member function

```
void mono::network::INetworkRequest::setErrorCallback (void (*cfunc)) ErrorEvent
*
```

Set an error handler callback function.

To receive a notification callback if this request fails, set a callback function here.

Parameters

- `cfunc` - A pointer to the function handling errors

```
template <typename Owner>
```

```
void mono::network::INetworkRequest::setStateChangeEventCallback (Owner * cnxt, void (Ow
Set a state change observer callback.
```

To receive notifications of state changes to the underlying network request, install a callback function with this method.

Parameters

- `cnxt` - A pointer to the callback context (the `this` pointer)
- `memPtr` - A pointer to the context class' member function

```
void mono::network::INetworkRequest::setStateChangeEventCallback (void
(*cfunc)) StateChangeEvent
*
```

Set a state change observer callback.

To receive notifications of state changes to the underlying network request, install a callback function with this method.

Parameters

- `cfunc` - A pointer to the function handling request state changes

Protected Functions

```
void INetworkRequest::setState (States newState)
set new states and trigger stateCHange callback if state changed
```

```
void INetworkRequest::triggerCompletionHandler ()
calls the completion handler
```

```
void INetworkRequest::triggerDirectErrorHandler ()
call the error handler, directly
```

```
void INetworkRequest::triggerQueuedErrorHandler ()
call the error callback, from the run queue
```

```
mono::network::INetworkRequest::INetworkRequest ()
A new network request in SETUP state
```

Protected Attributes

`mbed::FunctionPointerArg1<void, CompletionEvent *> mono::network::INetworkRequest::completionHandler`
Handler called upon successful completion of request.

`mbed::FunctionPointerArg1<void, ErrorEvent *> mono::network::INetworkRequest::errorHandler`
Handler called upon any error (transition into `ERROR_STATE`)

`mbed::FunctionPointerArg1<void, StateChangeEvent *> mono::network::INetworkRequest::stateChangeHandler`
handler called whenever the states changes

States `mono::network::INetworkRequest::state`
the current state of the request

int `mono::network::INetworkRequest::lastErrorCode`
holds the latest error code, 0 = no error

Timer `*mono::network::INetworkRequest::cbTimer`
queued callback timer object

Drawing

Color

class A RGB color representation, generic for display devices. This class implements a 16-bit RGB 5-6-5 color model. It support methods for calculating color blendings and more. The class definition also define a set of global constant predefined colors, like white, red, green, blue and black. Further, it includes a set of the FlatUI colors that Mono uses: Clouds WetAsphalt Concrete Silver Asbestos BelizeHole MidnightBlue Alizarin Turquoise Emerald

Public Functions

`Color::Color()`
Construct a black color.

`Color::Color(const int col)`
Construct a color from an existing 5-6-5 encoded value.
This constructor takes a an integer an casts it to a 5-6-5 RGB color, from from the integers LSB's.

Parameters

- `col` - The 5-6-5 RGB color value

`Color::Color(uint8_t R, uint8_t G, uint8_t B)`
Construct a color from individual RGB components.
You provide 8-bit RGB components to create the color.

Parameters

- `R` - The red component, 0 to 255
- `G` - The red component, 0 to 255
- `B` - The red component, 0 to 255

`uint8_t Color::Red()`
constReturn 8-bit red color component

`uint8_t Color::Green()`
constReturn 8-bit green color component

`uint8_t Color::Blue()`
constReturn 8-bit blue color component

`Color Color::scale (uint8_t factor)`
constMisc.
Multiply each RGB channel by a factor from 0-255

Return

the scaled color

`Color Color::blendMultiply (Color other)`
constReturn the product of two colors

Return

the multiply blended color

`Color Color::blendAdditive (Color other)`
constAdd this color with another

`Color Color::invert ()`
constReturn the inverse

`Color Color::alphaBlend (uint8_t intensity, Color const &other)`
constBlend this color with another using a provided intensity.

Alpha blending is done when you need to combine two colors, say you wish to realize a gradient. If you blend black and white the intensity will define all the grayscales in between.

Use this method to realize an alpha channel that controls how transparent a certain color is. The alpha channel controls the intensity, and you must provide the color that lies *behind* the alpha blended color.

Return

The new blended color

Parameters

- `intensity` - The intensity of this color, 0 to 255.
- `other` - The color to blend with

`mono::String Color::toString ()`
constGet a human readable string representatio of the color.

Returns a string of the form: (RR, GG, BB)

Return

a color string

DisplayPainter

class The *DisplayPainter* draws shapes on a display, using the DisplayController interface. You should use this class to draw shapes on the screen, and use it from inside view only. The standard view class has a reference to an instance of this class. The coordinate system used by the painter is the same as used by the display interface. This means all shape coords are relative to the display origo at the top left corner for the screen, when screen is in portrait mode. A painter keeps track of an active foreground and background color. You can set new colors at any time, and the succeeding draw calls will paint in that color. The standard draw color is the active foreground color, but some draw routines might use the background color also. An example is the font drawing routines. Like colors, the painter object keeps a active line width and textsize. When drawing shapes based on lines *drawLine*, drawPolygon, *drawRect* and drawEllipse, the line width used is the currently active line width property of the painter object. When painting text characters the character size is dependent on the textsize property. Text painting is not affected by the current line width. **Public Functions**

`DisplayPainter::DisplayPainter (IDisplayController *displayController, bool assignRefresh-Handler)`

Construct a new painter object that are attached to a display. A painter object is automatically initialized by the view/UI system and shared among the view classes.

In most cases you should not have to initialize your own display painter.

Parameters

- `displayController` - A pointer to the display controller of the active display
- `Take` - ownership of the DisplayControllers refresh callback handler

template <typename Owner>

void `mono::display::DisplayPainter::setRefreshCallback (Owner * obj, void (Owner::*) (void))`
Set/Overwrite the display tearing effect / refresh callback.

Set the Painters display refresh callback handler. The display refreshes the screen at a regular interval. To avoid graphical artifacts, you should restrict your paint calls to right after this callback gets triggered.

The default View painter already has a callback installed, that triggers the View's re-paint queue. If you create you own painter object you can safely overwrite this callback.

Parameters

- `obj` - The `this` pointer for the object who should have its member function called
- `memPtr` - A pointer to the class' member function.

void `DisplayPainter::setForegroundColor (Color color)`
Set the painters foreground pencil color.

Parameters

- `color` - The new foreground color

void `DisplayPainter::setBackgroundColors (Color color)`
Sets the painters background pencil color.

Parameters

- `color` - the new background color

`Color DisplayPainter::ForegroundColor()`
constGets the painters current foreground pencil color.

Return

The current foreground color

`Color DisplayPainter::BackgroundColor()`
constGets the painters current background pencil color.

Return

The current foreground color

`void DisplayPainter::useAntialiasedDrawing (bool enable)`
Turn on/off anti-aliased line drawing.

You can enable or disable anti-aliased drawing if you need nicer graphics or faster rendering. Anti-aliasing smoothes lines edges, that can otherwise appear jagged.

Parameters

- `enable` - Optional: Switch to turn on/off anti-aliasing. Deafult is enabled.

`bool DisplayPainter::IsAntialiasedDrawing()`
Returns `true` if anti-aliased drawing is enabled.

See

[*useAntialiasedDrawing*](#)

Return

`true` if enabled, `false` otherwise.

`uint16_t DisplayPainter::CanvasWidth()`
constGet the canvas width in pixels. This is the display display width as well.

Return

The canvas/display width in pixels

`uint16_t DisplayPainter::CanvasHeight()`
constGet the canvas height in pixels. This is the display display height as well.

Return

The canvas/display height in pixels

`IDisplayController *DisplayPainter::DisplayController()`
constGet a pointer to the painters current display controller You can use this method to obtain the display controller interface if you need to blit pixels directly to the display.

Return

A pointer to an object implementing the [*IDisplayController*](#) interface

`void DisplayPainter::drawPixel (uint16_t x, uint16_t y, bool background)`

Draw a single pixel on a specific position on the display.

The pixel will be the active foreground color, unless you set the third parameter to true.

Parameters

- `x` - The X-coordinate
- `y` - The Y coordinate
- `background` - Optional: Set to true to paint with active background color.

`void DisplayPainter::drawPixel (uint16_t x, uint16_t y, uint8_t intensity, bool background)`

Draw a pixel and blend it with the background/foreground color.

Use this method to draw transparent pixels. You define an intensity of the pixel, that corresponds to its Alpha value or opacity. 0 is totally transparent and 255 is fully opaque.

Only the foreground is influenced by the alpha value, the background is always treated as fully opaque.

Parameters

- `x` - The pixels x coordinate
- `y` - The pixels y coordinate
- `intensity` - The alpha value, 0 to 255 where 0 is transparent.
- `Optionaluse` - the background color as foreground and vice versa

`void DisplayPainter::drawFillRect (uint16_t x, uint16_t y, uint16_t width, uint16_t height, bool background)`

Draw a filled rectangle.

Paints a filled rectangle in the active foreground color. Coordinates defining the point of the rectangles upper left corner and are given in screen coordinates. (Absolute coordinates)

Parameters

- `x` - X coordinate of upper left corner, in screen coordinates.
- `y` - Y coordinate of upper left corner, in screen coordinates.
- `width` - The width of the rectangle
- `height` - The height of the rectangle
- `background` - Optional: Set to true to paint in active background color

`void DisplayPainter::drawLine (uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, bool background)`

Draw a straight line between two points.

Draw a line on the display. The line is defined by its two end points. End point coordinates are in absolute screen coordinates.

The line is *not* anti-aliased.

Based on Bresenham's algorithm. But If the line you intend to draw is horizontal or vertical, this method will use more efficient routines specialized for these cases.

Parameters

- `x1` - The X coordinate of the lines first endpoint
- `y1` - The Y coordinate of the lines first endpoint
- `x2` - The X coordinate of the lines second endpoint
- `y2` - The Y coordinate of the lines second endpoint
- `background` - Optional: Set this to `true` to paint in active background color

`void DisplayPainter::drawRect` (`uint16_t x`, `uint16_t y`, `uint16_t width`, `uint16_t height`, `bool background`)

Draw a outlined rectangle.

Draw an outlined rectangle with the current line width and the active color.

Parameters

- `x` - Top left corner X coordinate
- `x` - Top left corner X coordinate
- `width` - The rectangles width
- `height` - The rectangles height
- `background` - Optional: Set this to `true` to paint in active background color

`void DisplayPainter::drawChar` (`uint16_t x`, `uint16_t y`, `char character`)

Draw a single character on the display.

Paint a single ASCII character on the display. Characters are always painted in the foreground color, with the background color as background fill.

The character is painted in the currently selected text size. The text is a monospaced font, with a minimum size of (5,3) per character. The origo of a character is the upper left corner of the (5,3) rectangle.

If you want write text on the screen, you should use the `TextLabel` view, or the `Console` view.

Parameters

- `x` - The X coordinate of the characters upper left corner
- `y` - The Y coordinate of the characters upper left corner
- `character` - The text character to draw

`void DisplayPainter::drawVLine` (`uint16_t x`, `uint16_t y1`, `uint16_t y2`, `bool background`)

Helper function to draw a vertical line very fast. This method uses much less communication with the display.

*This method is automatically called by *drawLine**

Parameters

- `x` - The lines X coordinate (same for both end points)
- `y1` - The first end points Y coordinate
- `y2` - The second end points Y coordinate
- `background` - Optional: Set this to `true` to paint in active background color

void DisplayPainter::drawHLine (uint16_t x1, uint16_t x2, uint16_t y, bool background)

Helper function to draw a horizontal line very fast. This method uses much less communication with the display.

*This method is automatically called by *drawLine**

Parameters

- x1 - The first end points X coordinate
- x2 - The second end points X coordinate
- y - The lines Y coordinate (same for both end points)
- background - Optional: Set this to true to paint in active background color

void DisplayPainter::drawCircle (uint16_t x0, uint16_t y0, uint16_t r, bool background)

Paint an outlined circle.

Protected Functions

void DisplayPainter::swap (uint16_t &a, uint16_t &b)

Inline swap of two numbers.

Protected Attributes

mbed::FunctionPointer mono::display::DisplayPainter::displayRefreshHandler

Handler for the DisplayControllers action queue, that gets triggered when the display refreshes.

This handler is normally used by the first View that gets constructed, to enable the re-paint queue.

IDisplayController

class Abstract Interface for display controllers like ILITEK chip etc. This interface a simple set of function that display interface must implement. Mono display system build upon this interface, and all drawing is done through these functions. You must override all the defined methods in this interface. The interface does not define or depend on a communication protocol, like parrallel or SPI. Kristoffer Andersen

Public Functions

Author mono::display::IDisplayController::IDisplayController (int, int)

Setup the display controller object, init variables and the screen size. The width is the horizontal measure, when mono is held in portrait mode.

This constructor should not transfer any data or initialize the display. You must do that in the *init* method.

Return

The display controller instance.

Parameters

- width - The display width (horizontal) in pixels
- height - The display height (vertical) in pixels

`virtual = 0`Initializes the hardware display controller. Initializing the hardware display controller means setting up the power, gamma control and sync options. The display should be ready to draw graphics when this method returns. It is not necessary to do any draw commands from this method, other classes will take care of clearing the screen to black, etc.

`virtual = 0`Changes the current active window to a new rectangle. The display controller must support windowing in software or in hardware. The window is the active painting area of the screen, where you can paint. Upon drawing a single pixel the display cursor increments inside the window. This means you can sometime skip calls to the [setCursor](#) method.

```
void mono::display::IDisplayController::init()
```

```
void mono::display
```

- `x` - X-coordinate of the new windows upper left corner
- `y` - Y-coordinate of the new windows upper left corner
- `width` - The window width
- `height` - The window height

```
void mono::display::IDisplayController::setRefreshHandler (mbed::FunctionPointer
                                                         *handler)
```

Set the callback for display refreshes.

Set the callback function, that is called whenever the display has just repainted itself. This means it is time to repaint any dirty views, that needs to be updated.

IMPORTANT: You should re-paint graphics from within this callback, since it might run inside a hardware interrupt. It is better to just schedule the repaint from here.

Parameters

- `obj` - The owner object of the callback method (the `this` context)
- `memPtr` - A pointer to the owner objects callback member function

`virtual = 0`Set the drawing cursor to a new absolute position. Sets the current drawing cursor to a new position (must be within the currently active window). When setting the cursor you must use absolute coordinates (screen coordinates), not coordinates inside the active window.

```
void mono::display::IDisplayController::setCursor (int x, int y)
```

Parameters

- `x` - The new X position (screen coordinates)
- `y` - The new X position (screen coordinates)

`virtual = 0`Draw a pixel with the given color, at cursor position. Write a pixel color to the display, at the cursor position. This method will automatically increment the cursor position. If the increment happens automatically in hardware, the controller implementation must keep its own cursor in sync.

```
void mono::display::IDisplayController::write (Color pixelColor)
```

Parameters

- `pixelColor` - The 16-bit 5-6-5 RGB color to draw

`virtual = 0`Set the display backlight brightness. Higher values means more brightness. The display controller implementation might use a PWM to switch backlight LED's.

```
void mono::display::IDisplayController::setBrightness (uint8_t value)
```

Parameters

- `value` - The brightness 0: off, 255: max brightness

virtual Set the display backlight brightness. Higher values means more brightness. The display controller implementation might use a PWM to switch backlight LED's.

`void mono::display::IDisplayController::setBrightnessPercent (float percent)` **Parameters**

- percent - The brightness percentage, 0.00: off, 1.00: max brightness

virtual const = 0 Gets the current LES backlight brightness The display controller implementation might use a PWM to dim the display, this method returns the PWM duty cycle.

`uint8_t mono::display::IDisplayController::Brightness ()` **Return**

The current brightness level in 8-bit format: 0: off, 255: max brightness

Public Members

`uint32_t mono::display::IDisplayController::LastTearningEffectTime`

The last tearing effect interrupt time (`us_ticker_read()`)

To calculate the time since the last tearing effect interrupt (display refresh), you can use this member variable. On each interrupt this value is updated.

If too much time has passed between the interrupt occurred and you handle the painting, you might want to skip the painting. This is to avoid artifacts, when drawing on a refreshing display.

Any implementation of the *IDisplayController* must update this value in its tearing effect interrupt handler.

TextRender

class Text Render class to paint Text paragraphs on a DisplayController. This is a Text glyph and paragraph render. It uses a bitmap based fonts and typesets the text to provide. You need to provide the text render with either a *DisplayPainter* or *IDisplayController* that serves as a target for the text rendering. The *TextRender* does not include any Font definitions. When you render your text, you need to provide a pointer to wither the MonoFont structure, or a Adafruit compatible GFXfont structure. One these is used as the rendered font. *Monospaced MonoFonts* Mono has its own monospace-only font format MonoFont This renderer has a palette like *DisplayPainter*, and uses it the blend the semi-transparent pixels in the font anti-aliasing. The font bitmap defines pixel intensities, that are the foreground opacity. The MonoFont defines the text size and the anti-aliasing quality. Some fonts has 2 bit pixels, others have 4 bit pixels. *Adafruit GFXfonts TextRender* can also work with Adafruits proportionally spaced fonts, that include many different styles: Sans, serif, plain, italic, bold and combinations of these. This format is still bitmap based, but it does not used semi-transparent pixels to achieve anti-aliasing. The bitmaps are one bit per pixel. Because coherent glyphs might overlap each others bounding rects, this format does not draw text backgrounds. Since this will overwrite overlapping areas of text glyphs. *Line Layouts* The text rendering has two modes: line layout or not. Line layout means a text line always has a fixed height: the line separation height. This is critically when rendering multiline text layouts. But single line text will appear off-centered in a vertically centered box. The reason is the line height is much higher than the visual center of the text glyphs. If you need to render a single line of text at the center of a box, you want to not use line layout mode. In this mode, text dimension height are the distance from the texts highest to lowest point. **Public Types**

enum type `mono::display::TextRender::HorizontalAlignment`

The horizontal text alignment, when rendering inside rects.

Values:

Left justify text inside containing Rect

Justify at center of containing Rect
 Right justify text inside containing Rect

enum type `mono::display::TextRender::VerticalAlignmentType`

The horizontal text alignment, when rendering inside rects.

Values:

Place text at the top of containing Rect
 Place text at the vertical center of containing Rect
 Place text at the bottom of containing Rect

typedef Function interface that for character drawing routines

Public Functions

`TextRender::TextRender (IDisplayController *displayCtrl)`

Construct a *TextRender* that renders to a DisplayController Text Colors default to View::Standard colors.

Parameters

- `displayCtrl` - A pointer to the display controller that is the render target

`TextRender::TextRender (IDisplayController *displayCtrl, Color foreground, Color background)`

Construct a *TextRender* that renders to a DisplayController You provide explicit text colors.

Parameters

- `displayCtrl` - A pointer to the display controller that is the render target
- `foreground` - The text color
- `background` - the background color

`TextRender::TextRender (const DisplayPainter &painter)`

Construct a *TextRender* that renders to the DisplayController provided by a *DisplayPainter*. The painter current pencil colors are used for the text color.

Parameters

- `painter` - The display painter to copy DisplayController and color palette from

template <typename Context>

void `mono::display::TextRender::layoutInRect (geo::Rect rect, const String text, const`

Layout the string in a rect, using af callback to handle characters.

This method will parse a text string and trigger a callback for each character, providing the position and allowed dimension of each character.

Parameters

- `rect` - The rectangle to render in
- `text` - The text string to render
- `fontFace` - A pointer the Adafruit GFX font to use
- `self` - A pointer to the callback method context object
- `memptr` - A pointer to the callback method

- `lineLayout` - Default: `true`, Render text as a multiline layout

```
void mono::display::TextRender::drawInRect (geo::Rect rect, String text, const MonoFont  
                                         &fontFace)
```

Renders a text string in a provided Rectangle.

This method paints / renders the text in bounding rectangle. The text is always rendered with origin in the rectangles top left corner. If the provided Rect is not large enough, the text is clipped!

Parameters

- `rect` - The rectangle to render in
- `text` - The text string to render
- `fontFace` - A pointer the fontface to use

```
geo::Size mono::display::TextRender::renderDimension (String text, const MonoFont  
                                                    &fontFace)
```

Return the resulting dimension / size of some rendered text.

The final width and height of a rendered text, with the defined font face.

Parameters

- `text` - The text to calculate the dimensions of
- `fontFace` - The font to use

```
void mono::display::TextRender::drawChar (const geo::Point &position, const GFXfont  
                                         &font, const GFXglyph *gfxGlyph, geo::Rect  
                                         const &boundingRect, const int lineHeight)
```

Render a single Adafruit GfxFont character.

Parameters

- `position` - The point where the glyph is rendered
- `font` - The Adafruit GfxFont to use
- `gfxGlyph` - The specific glyph to render
- `boundingRect` - The Rect that limits the render canvas (no paints beyond this Rect)
- `lineHeight` - The glyph height offset, to align glyphs on the same baseline

```
void mono::display::TextRender::drawInRect (const geo::Rect &rect, String text, const  
                                         GFXfont &fontFace, bool lineLayout)
```

Renders a text string in a provided Rectangle.

This method paints / renders the text in bounding rectangle. The text is always rendered with origin in the rectangles top left corner. If the provided Rect is not large enough, the text is clipped!

Parameters

- `rect` - The rectangle to render in
- `text` - The text string to render
- `fontFace` - A pointer the Adafruit GFX font to use
- `lineLayout` - Default: `true`, Render text as a multiline layout


```
geo::Size mono::display::TextRender::renderDimension(String text, const GFXfont
&fontFace, bool lineLayout)
```

Return the resulting dimension / size of some rendered text.

The final width and height of a rendered text, with the defined font face.

Parameters

- `text` - The text to calculate the dimensions of
- `fontFace` - The font to use
- `lineLayout` - Default: `true`, use line layout or not

```
mono::geo::Rect TextRender::renderInRect (const geo::Rect &rect, String text, const GFXfont
&fontFace, bool lineLayout)
```

Return the calculated offset of the text in the drawing Rect.

The offset is returned in absolute coords, *not* the relative coords inside the drawing rect. The offset matters for text that is aligned other than left and top.

Parameters

- `rect` - The drawing Rect, that the text should be rendered inside
- `text` - The text
- `fontFace` - The GFXfont to use
- `lineLayout` - Default: `true`, sets if `lineLayout` is used

```
void TextRender::setForeground (Color fg)
```

Set the text color.

```
void TextRender::setBackground (Color bg)
```

Set the background color. Transparent pixels will be blended with this color.

```
void TextRender::setAlignment (HorizontalAlignment align)
```

Sets the texts justification within the drawing rect.

Parameters

- `align` - The alignment

```
void TextRender::setAlignment (VerticalAlignmentType vAlign)
```

Set the texts vertical alignment within the drawing rect.

Parameters

- `vAlign` - The vertical alignment

Color TextRender::Foreground()

constGet the current text color.

Color TextRender::Background()

constGet the current text background color.

TextRenderer::HorizontalAlignment TextRenderer::Alignment()

constGet the horizontal text alignment.

`TextRender::VerticalAlignmentType TextRender::VerticalAlignment ()`
constGet the vertical text alignment.

Protected Functions

`void mono::display::TextRender::drawChar (geo::Point position, char character, const MonoFont &font, geo::Rect const &boundingRect)`

Render a single MonoFont character.

`int TextRender::calcUnderBaseline (String text, const GFXfont &font)`
Calculated the maximum offset under the text baseline.

This method returns the maximum offset under the baseline for some text. Some characters (glyphs) has parts under the baseline, examples are lower-case *g*, *p*, *q* and *j*.

To align all glyphs in a text line, the character with the largest undershoot is found and returned here.

Return

The maximum offset below the text baseline

Parameters

- `text` - The text content to process font The font face to use

`void TextRender::writePixel (uint8_t intensity, bool bg)`
Blend and emit a single pixel to the DisplayController.

`uint32_t TextRender::remainingTextlineWidth (const GFXfont &font, const char *text)`
Returns the pixel width of the text provided.

This method is used in multiline text rendering, to get the width of a stand-alone line inside a multiline text blob. Unlike `renderDimensions` that gets the width of the longest line, this method get the length of the current line.

Return

The pixel width of the line

Parameters

- `font` - The font face to use
- `The` - C string text to get the current line from (until newline or end)

Monolcon

struct The icon type object for the Mono's icon system. This struct defines an icon bitmap. The bitmap is a row-first array of pixel values, that represents blending between two colors.[*mono::ui::IconView*](#)

Public Members

`uint16_t mono::display::MonoIcon::width`
The icons width in pixels

`uint16_t mono::display::MonoIcon::height`
The icons height in pixels

`uint8_t *mono::display::MonoIcon::bitmap`
The raw icon pixel data

Geometry

Circle

class

Point

class Representation of a point in a 2D cartesian coordinate system. The point has no width or height, only an x-coordinate and a y-coordinate. This class defines the coordinates, as well as methods to manipulate the point. Also functions for geometrical calculus is present.

Rect

class A Rectangle in a Cartesian coordinate system, having a size and position. This class defines a geometric rectangle. It exists in a std. cartesian coordinate system. It is defined by its upper left corner (X,Y), and a width and height. The rectangle cannot be rotated, its sides a parallel to the coordinate system axis. It defines helper methods to calculate different positions and properties. You can also extract interpolation information.*Size Point*

Public Functions

~~Size~~ `Rect::Rect (int x, int y, int width, int height)`
Construct a rectangle from position coordinates and size.

`Rect::Rect (const Point &p, const Size &s)`
Construct a rectangle from *Point* and *Size* objects.

`Rect::Rect ()`
Construct an empty rectangle having position (0,0) and size (0,0)

`Point Rect::UpperLeft ()`
constRetrn the position of the upper left corner.
This method is the same as casting the *Rect* to a *Point*

Return

The point of the upper left corner

`Point Rect::LowerRight ()`
constRetrn the position of the lower right corner.

Return

The point of the lower right corner

`Point Rect::UpperRight ()`
constRetrn the position of the upper right corner.

Return

The point of the upper right corner

`Point Rect::LowerLeft()`
constReturn the position of the lower left corner.

Return

The point of the lower left corner

`class Point Rect::Center()`
constReturn the position of the Rectangles center.

`void Rect::setPoint(class Point p)`
Move (translate) the rectangle by its upper left corner.

`void Rect::setSize(class Size s)`
Set a new size (width/height) of the rectangle.

`bool Rect::contains(class Point &p)`
constCheck whether or not a *Point* is inside this rectangle.

`Rect Rect::crop(Rect const &other)`
constReturn this *Rect* cropped by the boundaries of another rect

`mono::String Rect::ToString()`
constReturn a string representation of the ractangle.

Return

A string of the form `Rect (x, y, w, h)`

Size

class Class to represent a rectangular size, meaning something that has a width and a height. A size class does not have any position in space, it only defines dimensions. **Public Functions**

`Size::Size()`
Construct a NULL size. A size that has no width or height.

`Size::Size(int w, int h)`
Construct a size with defined dimenstions

`Size::Size(const Size &s)`
Construct a size based on another size

Sensors**IAccelerometer**

class Abstract inteface for interacting with the accelerometer. **Public Functions**

virtual Start the accelerometer. Before you can sample any acceleration, you must start the accelerometer. When the accelerometer is running its power consumption will likely increase. Remember to *Stop* the accelerometer, when you are done sampling the acceleration.

virtual Stops the accelerometer. A stopped accelerometer can not sample acceleration. [Start](#) the accelerometer before you sample any axis.

virtual = 0 Return the current Start/Stop state of the accelerometer.

```
void mono::sensor::IAccelerometer::Start ()
```

```
void mono::sensor::IAccelerometer::Start ()
```

true only if the accelerometer is started and sampling data

virtual = 0<# brief desc #>

```
int16_t mono::sensor::IAccelerometer::rawXAxis (bool monoOrientation)
```

Return

<# return desc #>

virtual = 0<# brief desc #>

```
int16_t mono::sensor::IAccelerometer::rawYAxis (bool monoOrientation)
```

Return

<# return desc #>

virtual = 0<# brief desc #>

```
int16_t mono::sensor::IAccelerometer::rawZAxis (bool monoOrientation)
```

Return

<# return desc #>

IBuzzer

class Generic Buzzer interface. This interface defines a generic API for buzzers used in the framework. You should not construct any subclass of this interface yourself. The system automatically creates a buzzer object for you, that you can obtain through the IApplicationContext: **Example**

```
mono::sensor::IBuzzer *buzz = mono::IApplicationContext::Instance->Buzzer;
```

To make a short buzz sound do:

```
mono::IApplicationContext::Instance->Buzzer->buzzAsync(100);
```

Public Functions

virtual = 0 Buzz for a given period of time, then stop. Sets the buzzer to emit a buzz for a defined number of milliseconds. Then stop. This method is asynchronous, so it returns immediately. It relies on interrupts to mute the buzzer later in time. You should not call it multiple times in a row, since it behaves asynchronously. Instead use [Timer](#) to schedule multiple beeps.

```
void mono::sensor::IBuzzer::buzzAsync (uint32_t timeMs)
```

Parameters

- timeMs - The time window where the buzzer buzzes, in milliseconds

virtual = 0 Stop any running buzz. Use this method to cancel a buzz immediately. This method will not have any impact on callback functions. They will still be called, when the buzz was suppose to end.

```
void mono::sensor::IBuzzer::buzzKill ()
```

```
template <typename C> void mono::sensor::IBuzzer::buzzKill ()
```

```
void mono::sensor::IBuzzer::buzzAsync (uint32_t timeMs, Object * self, void (Object::*) (void))
```

Buzz for some time, and when done call a C++ member function.

Sets the buzzer to emit a buzz for a defined number of milliseconds. Then stop. This method is asynchronous, so it return immediately. It relies on the run loop to mute the buzzer later in time. You also provide a callback function, that gets called when the buzz is finished.

You should not call it multiple times in a row, since it behaves asynchronously. Instead you should use the callback function to make a new beep.

Example This will buzz for 100 ms, then call `buzzDone`.

```
buzzAsync<AppController>(100, this, &AppController::buzzDone);
```

Parameters

- `timeMs` - The time window where the buzzer buzzes, in milliseconds
- `self` - The `this` pointer for the member function
- `member` - A pointer to the member function to call

```
void mono::sensor::IBuzzer::buzzAsync (uint32_t timeMs, void (*function)) void
```

Buzz for some time, and when done call a C function.

Sets the buzzer to emit a buzz for a defined number of milliseconds. Then stop. This method is asynchronous, so it return immediately. It relies on the run loop to mute the buzzer later in time. You also provide a callback function, that gets called when the buzz is finished.

You should not call it multiple times in a row, since it behaves asynchronously. Instead you should use the callback function to make a new beep.

Example This will buzz for 100 ms, then call global C function `buzzDone`.

```
buzzAsync(100, &buzzDone);
```

Parameters

- `timeMs` - The time window where the buzzer buzzes, in milliseconds
- `function` - A pointer to the global C function to call

Protected Attributes

```
mbed::FunctionPointer mono::sensor::IBuzzer::timeoutHandler
```

A Handler to call when buzzing finished.

This member is setup when you provide a callback to the `buzzAsync` methods. Your subclass must call this when buzzing ends.

ITemperature

class Abstract Interface for temperature sensors. This interface creates a hardware-independent abstraction layer for interacting with temperature sensors. Any hardware temperature sensor in Mono must subclass this interface. In Mono Framework these is only initialized one global temperature sensor object. To obtain a reference to the temperature sensor, use the `IApplicationContext` object:

```
ITemperature *temp = mono::IApplicationContext::Instance->Temperature;
```

This object is automatically initialized by the `IApplicationContext` and the current `ITemperature` subclass. IT is the `IApplicationContext`'s job to initialize the temperature sensor.

Public Functions

virtual = 0 Reads the current temperature from the temperature sensor

```
int mono::sensor::ITemperature::Read()
```

Return

the temperature in Celcius

virtual Reads the temperature in fixed point milli-Celcius. To get a higher precision output, this method will return milliCelcius such that: 22,5 Celcius == 22500 mCelcius

```
int mono::sensor::ITemperature::ReadMilliCelcius()
```

Return

The temperature in mCelcius

Power

MonoBattery

class Static class to get Mono's battery voltage or percentage. Markus Laire

Public Static Functions

```
uint8_t MonoBattery::CalculatePercentage (uint16_t voltage, const uint16_t lookupTable[100])
```

Calculate percentage of battery remaining.

Percentage is calculated using a simple lookup-table of 100 voltage values, in millivolts:

- if voltage < lookupTable[0], return 0
- else if voltage < lookupTable[1], return 1
- ...
- else if voltage < lookupTable[99], return 99
- else return 100

Return

Percentage of battery remaining

Parameters

- voltage - Battery voltage as returned by ReadMilliVolts
- lookupTable - Custom lookup-table to use

```
uint16_t MonoBattery::ReadMilliVolts()
```

Read the battery voltage in millivolts.

To get higher accuracy battery voltage is sampled multiple times and average of these is returned.

Shortly after Mono has been reset, battery voltage can't be read. In this case 0 is returned.

Return

The battery voltage in mV

```
uint8_t MonoBattery::ReadPercentage (const uint16_t lookupTable[100])
```

Read the battery voltage and return percentage remaining.

See CalculatePercentage for details of how percentage is calculated.

Return

Percentage of battery remaining

Parameters

- `lookupTable` - Custom lookup-table to use

IPowerAware

class The Power Aware Abstract interface, for classes that handle power events. Classes that implements this interface can receive power related events in from of handler methods. This interface define the 3 handler methods: `onSystemPowerOnReset` : Called reset or first power on `onSystemEnterSleep` : Called just before system enter sleep mode `onSystemWakeFromSleep` : Called right after the system has woken from sleep `onSystemBatteryLow` : Called when battery is critically low

Inheriting this interface is not enough to active the functionality. You must remember to add your object instance to the *IPowerManagement* instance, that can be obtained from the static *IApplicationContext* class.

Public Functions

virtual Called when the system powers on after a reset. You can override this method. It gets called right after the system power on for the first time, or after a reset condition. Use this method to setup / initialize components and data structures. It is only called once for each system reset.

virtual Called right before the MCU goes into sleep mode. You can override this method to get sleep notifications. Before the CPU stop executing instructions and goes into low power sleep mode, this method gets called. Use this method to prepare your data or MCU components for sleep. Help preserve battery by power down any peripheral, that is not needed during sleep. This method can be called many times during your apps life cycle. (That is between resets.) After each sleep period, when the MCU wakes the *onSystemWakeFromSleep* is guaranteed to be called.

virtual Override to get notified when system wakes from sleep. You can override this method to get wake-up notifications. When the CPU starts executing instructions and the power system has powered up all peripherals - this method gets called. Use this method to setup your app to resume after sleep mode. This method can be called many times during your apps life cycle. (That is between resets.)

virtual Override to get notified when the battery voltage reaches a critical level. You can override this method to get *battery low* notifications. When this methods gets called, you have some time to finish critical tasks. That might writing state to file system or transfer data over the network. Depending on the health of the battery, the time between this notification and the actual system enforced power off, might vary. In contrast to the other *power aware* methods this is only called once in the application life cycle. After the enforced power off, when the battery is charged, the system will automatically reset.

Public Members

```
void mono::power::IPowerAware::onSystemPowerOnReset () void mono::power::IPowerAware::onSystemEnterSleep ()
Next pointer in the power awareness object queue. The IPowerManagement uses this to traverse all power aware objects.
```

```
IPowerAware *mono::power::IPowerAware::_pwrwr_previousPowerAware
Previous pointer in the power awareness queue.
```


IPowerManagement

class Generic abstract interface for the power management system. A PowerManagement implementation class handles power related events and sets up the system. The ApplicationContext object initializes an instance of this class automatically. Use can find a pointer to the PowerManagement object in the static *ApplicationContext* class. Depending on the system (mono device or simulator), the concrete sub-class implementation of this interface varies. The active *ApplicationContext* initializes this class and calls its POR initialization functions. Implementations of this class then calls and initializes any nessasary related classes, like the power supply IC sub-system (*IPowerSubSystem*). This interface defines queues of objects that implement the *IPowerAware* interface. This interface lets classes handle critical power events like: **Power-On-Reset** (POR): Called when the chip powers up after a reset **Enter Sleep Mode**: Called before system goes in to low power sleep mode **Awake From Sleep**: Called when nornal mode is restored after sleep

Power Awareness

Classes that handle components like I2C, Display, etc can use the PowerAware interface to receive these type of events. Its the PowerManagement object task to call these *IPowerAware* objects.

The interface defines a protected member object *powerAwarenessQueue* that is a pointer to the first object in the queue. The Power Awareness *Queue* is a list of objects that implment the IpowerAware interface and have added themselves to the queue by calling *AppendToPowerAwareQueue*

Objects in the queue receive power aware notifications on event like enter sleep mode, wake from sleep and battery low. You can add your own objects to the queue to make them “power aware” or you remove the system components that gets added by default. (But why would you do that?)

Public Functions

virtual = 0Send Mono to sleep mode, and stop CPU execution. In sleep mode the CPU does not ex-cute instruction and powers down into a low power state. The power system will turn off dynamically powered peripherals. Any power aware objects (*IPowerAware*), that has registered itself in the power-AwarenessQueue must have its onSystemEnterSleep method called. *NOTE: Before you call this method, make sure you have configured a way to go out of sleep.*

virtual Add a *IPowerAware* object to the awareness queue By added object to the Power Awareness *Queue* they receive callbacks on power related events, such as reset and sleep.

```
void mono::power::IPowerManagement::EnterSleep() void mono::power::
```

- object - A pointer to the object that is power aware

virtual Remove an object from the Power Awareness *Queue*. Searches the Power Awareness *Queue* for the object and removes it, if it is found. This object will no longer receive power related notifications.

```
bool mono::power::IPowerManagement::RemoveFromPowerAwareQueue (IPowerAware *object) Return
```

true if object was removed, false if the object was not in the queue

Parameters

- object - A pointer to the object that should be removed from the queue

Public Members

IPowerSubSystem *mono::power::IPowerManagement::PowerSystem

A pointer to the initialized power sub-system. This is initialize automatically and depends on compiled

environment. The power system to used to control power supply to peripherals and to give interrupt on power related events.

WARNING: Use this class with extreme caution! Wrong power settings can fry the MCU and other peripherals!

Public Static Attributes

volatile bool `IPowerManagement::__shouldWakeUp`

This variable must be `true` before sleep mode is aborted.

EnterSleep() must set this to `false`, you should set it to `true` to abort sleep mode and re-enter the run loop execution. *EnterSleep()* will not *return* before this is set to `true`.

****Note:** The class `QueuedInterrupt` will set this automatically!**

volatile bool `IPowerManagement::__busySleep`

Global flag to indicate to not halt CPU during sleep mode.

Some processes might require hardware to run async under what is normally sleep mode, where hardware peripherals are halted.

Set this flag to `true` to keep the the CPU awake inside the sleep mode loop. This is especially useful when handled debouncing of edge triggered interrupts.

Note: You should keep this flag `false` to ensure power presevation

Protected Functions

virtual Call all the power aware objects right after Power-On-Reset The singleton power management object must call this method on reset

Protected Attributes

void `mono::power::IPowerManagement::processResetAwarenessQueue()`

IPowerAware *`mono::`

A pointer to the top object in the *Power Awareness Queue*

The Power Awareness queue is realized by having the power object themselves hold references to the next and previous object in the queue. This eliminates the need for dynamic memory allocation a runtime.

The *IPowerAware* interface defines the *next* and *previous* pointers for the object in the linked list. This class only holds a reference to the first object in the queue.

IPowerSubSystem

class Abstract interface for the power sub-system. It defines 3 basic methods related to reset, enter sleep and exit sleep modes. This interface is sub-classed by implementations of the different power supply IC's on mono or an emulator. Subclasses of this interface should only conduct routines related to a power sub-system - not to any CPU specific operations! This means setting up voltage levels and enabling power fencing to peripherals. This power supply sub-system interface also defines callbacks that are called then the battery events occur. These are: Battery low warning

You can listen to these events by supplying a callback handler function

Power to MCU internal modules are controller by the abstract interface for power management *IPowerManagement*

Public Types

enum type `mono::power::IPowerSubSystem::ChargeState`

Battery charging states

See

ChargeStatus

Values:

- Chip does not support charging or dont disclose it
- Charging has just begun (pre-condition)
- Fast Charging in constant current mode
- Slower charging, in Constant Voltage mode
- Charge ended of cycle, battery is full
- No battery attached or wrong battery voltage levels

Public Functions

`virtual = 0` Called by the application as first thing after power-on or system reset. The function must set up the default power configuration of the system, peripherals, voltages etc.

`virtual = 0` Called before the system enter a sleep mode, where the CPU is not excuting instructions. To enable the lowest possible power consumption subclasses can turn off selected periphrrals here.

`virtual = 0` Called after the system has woken from a sleep mode. This is only called after an call to *onSystemEnterSleep* has occured. Use this method to turn on any disabled peripheral.

`virtual` Return the current status of the Power Fence. The power fence cuts power to specific peripherals. Each peripheral driver should know whether or not it is behind the fence.

`void mono::power::IPowerSubSystem::onSystemPowerOnReset ()`

`void mono::power:`

`true` if the power fence is active (power is not present), `false` if power is ON.

`virtual` Turn on/off the power fence. Some peripherals are behind a power fence, that can cut their power. You can control this power, and remove their supply upon going to sleep mode, to safe battery.

`void mono::power::IPowerSubSystem::setPowerFence (bool active)`

Parameters

- `active` - `true` will cut the power, `false` will power the peripherals

`virtual` Get the current charge status for the attached battery. The Subsystem implementation might be able to monitor the current charging state of the battery. If no battery exists the state will be `SUSPENDED`. If the implementation does not support charge states this method will always return *UNKNOWN*. The different states is explained by the *ChargeState* enum.

ChargeState `mono::power::IPowerSubSystem::ChargeStatus ()`

Return

The current charge state integer

See

ChargeState

`virtual` Get the USB charging state (True if charging now) This methods default implementation uses the *ChargeStatus* method to check the `CHARGE_*` enum and `true` if it is not `SUSPENDED` or `UNKNOWN`. PowerSubsystem subclasses might override this method do their own checks.

virtual = 0Return `true` is the battery voltage is OK, `false` is empty. This method query the system power state, to see if the battery is OK. **In case this return , the system should enter low-power sleep immediately!**

Public Members

`bool mono::power::IPowerSubSystem::IsUSBCharging()` `bool mono::power:`
Function handler that must be called when the PowerSystem detect low battery

`mbed::FunctionPointer mono::power::IPowerSubSystem::BatteryEmptyHandler`
Function handler that must be called when battery is empty

Redpine Module I/O

ManagementFrame

class Public Functions

`ManagementFrame::ManagementFrame()`
Construct an empty (uninitialized) management frame This is used when you need to allocate memory on the stack for a frame, and pass it to another function.

`ManagementFrame::ManagementFrame (mgmtFrameRaw *rawFrame)`
Construct a management frame from reponse data
The constructed object will not reference the raw data in any way.

Parameters

- `rawFrame` - A pointer to the raw frame data structure

`ManagementFrame::ManagementFrame (RxTxCommandIds commandId)`
Construct a outgoing management frame, having the TX_FRAME direction parameter set. No other frame parameters are initialized

Parameters

- `commandId` - The TX command id

virtual If this frame is of type TX_FRAME this method will sent it to the module. When the command frame is sent, the method will wait for a response and then check the response against the request. If the property *responsePayload* is set to `true`, the method *responsePayloadHandler* is called automatically. If the command takes multiple responses, like TCP data receive on sockets, you should see the *lastResponseParsed* property. When the method returns, the frame response (with payload data) is expected to be present.

`bool ManagementFrame::commit()` **Return**

`true` on success, `false` otherwise
virtual Send a TX_FRAME to the module asynchronous. Same as *commit*, but but asynchronous and return immediately. You should set the completion callback handler (*setCompletionCallback*) before calling this method.

void ManagementFrame::commitAsync ()

See

commit

See

setCompletionCallback

virtual Abort the execution (commit) of the frame. If the frame is pending, it is aborted and removed from the to-be-sent request queue. If the frame has already been sent to the module, the abort is ignored. Still the completion callback handler if removed, to avoid calling freed objects.

virtual Internal method used by *commitAsync* method to send the frame to the module, inside the async function handler. You should not call this directly.

virtual Internal method to trigger the completion handler callback - if any This method should only be used by the *Module* member method `moduleEventHandler`.

virtual Gets the frames raw data format for transfer via the communication channel. This format is only the frame itself, not any data payload

void ManagementFrame::abort ()

bool ManagementFr

raw data struct

Parameters

- data - A pointer to the raw frame structure were the data is written to.

virtual Get the byte length of the data payload for this management frame The payload length varies for every subclass.

int ManagementFrame::payloadLength ()

Return

payload data byte length

virtual Write the payload data for the frame into the provided buffer. The payload data is dependent on the specific management frame subclass. If the managment frame subclass does not have any payload data, this method not do anything. The target buffer will be left untouched. Before you call this method, you should check if payload data exists using the *payloadLength* method. The data returned to ready to be transferred to the module, it is 4-byte aligned as required. It is you responsibility to ensure the provided data buffer is large enough to contain the needed bytes.

void ManagementFrame::dataPayload (uint8_t *dataBuffer)

Parameters

- dataBuffer - A pointer to the target buffer, where the payload data is written.

virtual When frames need to parse or react to response payloads, there are received after a call to *commit*. You must overload this method in your subclass and handle the parsing of the response payload. This parsing should set object properties that can later be accessed by outside objects. This mehod is called internally by the *commit* method, only if property *responsePayload* is set to `true`.

void ManagementFrame::responsePayloadHandler (uint8_t *payloadBuffer)

Parameters

- payloadBuffer - A pointer to the raw payload data buffer

template <typename Owner>

void mono::redpine::ManagementFrame::setCompletionCallback (Owner * obj, void (Owner::*)

Set the frame completion callback handler.

The member function you provide is called when the frame is successfully committed. This means it has been sent to the module, and a response has been received.

The callback function must accept an input parameter:

```
void functionName (ManagementFrame::FrameCompletionData *);
```

Parameters

- `obj` - The member functions context pointer (the `this` pointer)
- `memPtr` - A pointer to the member function on the class

Public Members

`uint16_t mono::redpine::ManagementFrame::status`
Management response status

`bool mono::redpine::ManagementFrame::responsePayload`

The length of this frames payload data, differs for every subclass. Frames can handle responses from the module themselves This `bool` indicates that a frame subclass can handle a response frame and data payload.

Normally the (*commit*) method handles and parses a frames response, but when there is a payload, it needs a *response payload handler*

This you set this property to `true` in your subclass, when you must overload the method (*responsePayloadHandler*).

The default value for this property is `false`

`bool mono::redpine::ManagementFrame::lastResponseParsed`

As long as this property is `false` the frame will continue to process responses.

Some commands gets multiple responses from the module. Say you use the HTTP GET command or a socket receive command, the received data arrive in chunks of different lengths.

Set this property to `false` and the *commit* method will know, it should parse multiple responses for the command.

If you set this property to `false`, the method *responsePayloadHandler* will be called multiple times.

Finally, when you know the module will not sent any further response frames, you must indicate this by setting this property to `true`

If a command frame takes only one response, your subclass can ignore this property.

`true`

`bool mono::redpine::ManagementFrame::autoReleaseWhenParsed`

The module handle class (*Module*) should dealloc this object when it has parsed the last chunk of response data.

`mbed::FunctionPointerArg1<void, FrameCompletionData *> mono::redpine::ManagementFrame::completionHandler`

The handler called when *commit* finishes.

To handle frame request and responses asynchronously, this handler is triggered when the frame is committed.

`void *mono::redpine::ManagementFrame::handlerContextObject`

A reference to the completion handler context object

struct The datastructure provided by the Management Frames async completion handler. You should use the member variable `Context` to reference any frame response data. You should expect the frame to be deallocated right after you callback handler functions returns. **Note:** You type safety you can check the frames `CommandId` before downcasting the `Context` pointer to a specific frame subclass. **Public Members**

`bool mono::redpine::ManagementFrame::FrameCompletionData::Success`
See if the request was successfull

ManagementFrame `*mono::redpine::ManagementFrame::FrameCompletionData::Context`
A pointer to the request frame object

Module

class Redpine Wireless module Class. The module class controls the physical module, sending and receiving data from the device. It utilizes a communication interface (*ModuleCommunication*) to send and receive data. This makes the class independent of the communication protocol. Because there is only one physical module, this class is a singleton object. You access it by the static *Instance* method. To setup the Redpine module you must call the static global methods: ***Initialize***: Initializes the communication interface ***setupWifiOnly***: Join a wifi network with security and setup DHCP

Only networking classes should access the module directly. Tcp socket, DNS resolution etc, should be handled by dedicated classes, that uses this class.

Public Types

enum type `mono::redpine::Module::ModuleCoExistenceModes`

The module can operate in different modes, that controls the different wireless capabilities.

Values:

- Only enabled Wifi, with embedded TCP/IP
- Only Bluetooth Classic (2.0) is enabled
- Only Bluetooth LE is enabled
- Coex-mode with wifi and BT classic
- Coex-mode with wifi and BLE

enum type `mono::redpine::Module::WifiSecurityModes`

List of Wifi security modes. Use these when you connect to an access point.

The last option `SEC_WPA_WPA2` is a mixed mode, that gives priority to WPA2.

Values:

- No security

Either in WPA/WPA2 mode (Mixed mode)

enum type `mono::redpine::Module::BootloaderMessageCodes`

Command messages that can be written to the Bootloaders host interface register.

Values:

= 0xAB00

= 0x0031

= 0x0037

enum type `mono::redpine::Module::BootloaderRegisters`

Memory addresses of the registers related the Bootloader

Values:

= 0x4105003C

= 0x41050034

Public Functions

`void` `Module::moduleEventHandler()`

Callback function installed into the CommunicationInterface interrupt callback listener.

Public Members

`mbed::FunctionPointerArg1<bool, ManagementFrame *>` `*mono::redpine::Module::asyncManagementFrameHan`

The default handler for async incoming management frames.

Such frames are power, socket connect and disconnect events.

Public Static Functions

`Module *``Module::Instance()`

MARK: STATIC PUBLIC METHODS.

Obtain a reference to the singleton module object

`bool` `Module::initialize` (`ModuleCommunication *``commInterface`)

Sends the initalize command and waits for Card Ready from the module and set the communication interface for this object. After this method is called the module is ready to receive the OperMode command etc.

Parameters

- `commInterface` - The module communication interface to be used

`bool` `Module::setupWifiOnly` (`String ssid`, `String passphrase`, `WifiSecurityModes secMode`)

Initializes the module in Eifi only mode with enabled TCP/IP stack We expect this mode to be the default mode, when only using Wifi

The method will set the module in Wifi client mode with the 2.4 GHz band antenna.

Parameters

- `ssid` - The SSID name of access point to connect to

- `passphrase` - The passphrase of the access point to join
- `secMode` - The access points security setting, default is WPA/WPA2

`bool Module::IsNetworkReady ()`

See if the network is ready and running.

Check to see if the module has a ready network stack with initialized IP configuration. A ready network stack is ready to fetch or send data to and from the internet.

Return

`true` if the network is ready, `false` otherwise

Protected Functions

`Module::Module ()`

Protected class constructor Can only be called by the *Instance()* method.

`void Module::handleSleepWakeUp ()`

Handle the modules periodic wake ups Puts the module to sleep again, if no input is to be sent.

`void Module::onNetworkReady (ManagementFrame::FrameCompletionData *data)`

Callback for the DHCP Mgmt frame response, that indicate the network is ready

virtual Called when the system powers on after a reset. You can override this method. It gets called right after the system power on for the first time, or after a reset condition. Use this method to setup / initialize components and data structures. It is only called once for each system reset.

virtual Called right before the MCU goes into sleep mode. You can override this method to get sleep notifications. Before the CPU stop executing instructions and goes into low power sleep mode, this method gets called. Use this method to prepare your data or MCU components for sleep. Help preserve battery by power down any peripheral, that is not needed during sleep. This method can be called many times during your apps life cycle. (That is between resets.) After each sleep period, when the MCU wakes the *onSystemWakeFromSleep* is guaranteed to be called.

virtual Override to get notified when system wakes from sleep. You can override this method to get wake-up notifications. When the CPU starts executing instructions and the power system has powered up all peripherals - this method gets called. Use this method to setup your app to resume after sleep mode. This method can be called many times during your apps life cycle. (That is between resets.)

virtual Override to get notified when the battery voltage reaches a critical level. You can override this method to get *battery low* notifications. When this methods gets called, you have some time to finish critical tasks. That might writing state to file system or transfer data over the network. Depending on the health of the battery, the time between this notification and the actual system enforced power off, might vary. In contrast to the other *power aware* methods this is only called once in the application life cycle. After the enforced power off, when the battery is charged, the system will automatically reset.

`void Module::onSystemPowerOnReset ()`

Handles an incoming data frame (expect that it contain socket data)

`void Module::onSy`

`bool Module::initAsyncFrame (const DataReceiveBuffer &buffer, ManagementFrame **frame)`

Initiazes (constructs) the correct subclass of an async mgmt frame

`bool Module::discardIfNeeded (ManagementFrame *frame)`

MARK: PRIVATE METHODS.

Protected Attributes

`bool mono::redpine::Module::communicationInitialized`

True is the module communication interface is initialized

`bool mono::redpine::Module::networkInitialized`

True if the module has its network stack ready

ModuleCommunication *`mono::redpine::Module::comIntf`

The communication interface used by the module

ModuleCoExistenceModes `mono::redpine::Module::OperatingMode`

This holds the module currently initialized operation mode

`ModulePowerState mono::redpine::Module::CurrentPowerState`

The current state of the module, is it awake or sleeping

GenericQueue<ManagementFrame> `mono::redpine::Module::requestFrameQueue`

A queue over the pending requests to-be-send to the module

GenericQueue<ManagementFrame> `mono::redpine::Module::responseFrameQueue`

A queue over the pending requests sent / pending to the module

`mbed::FunctionPointer mono::redpine::Module::networkReadyHandler`

User can install a network ready event callback in this handler

Protected Static Attributes

`Module Module::moduleSingleton`

The only instantiation of the module class

Construct the module

class List of found AP's, as retrieved by the scanNetworks method. TODO: This class should become generic across H/W!scanNetworks

Public Members

~~Static~~ `uint32_t mono::redpine::Module::ScannedNetworks::Count`

Number of found networks

`Network *mono::redpine::Module::ScannedNetworks::networks`

List of results

struct Object to configure static IP **Public Members**

`uint8_t mono::redpine::Module::StaticIPParams::ipAddress[4]`

IPv4 client address

`uint8_t mono::redpine::Module::StaticIPParams::netmask[4]`

IPv4 netmask

`uint8_t mono::redpine::Module::StaticIPParams::gateway[4]`

IPv4 Gateway

ModuleCommunication

class Abstract class that defines redpine module communication through a hardware interface. Subclasses of this should utilize UART, USB or SPI communication. **Public Functions**

virtual = 0Send a interface initialization command to the module.

virtual = 0Trigger the reset line to the module. A reset will put the module into bootloader, where a firmware image can be selected. Before calling *initializeInterface*, you must reset the module. The reset command has to wait while the module resets, this means the function will block for approx 3 ms.

virtual Test the data inside a buffer to see if it is a management frame

```
bool mono::redpine::ModuleCommunication::initializeInterface() void mono::redpine::ModuleCommunication::reset()
```

- *buffer* - The buffer to test

virtual Test if a buffer is a data Frame

virtual = 0Polls the module for waiting input data. This can be frame responses or any other data, waiting to be read by us. The method uses the current communication interface method to wait for pending input data. This function blocks, until data is ready. This function can be used it interrupts are not applicable

```
bool ModuleCommunication::bufferIsDataFrame(DataReceiveBuffer &buffer) bool mono::redpine::ModuleCommunication::bufferIsDataFrame(DataReceiveBuffer &buffer)
```

true if there is data to read, false otherwise

virtual = 0Return true if interrupt is active. The module will keep the interrupt pin high until no more input is present.

virtual = 0Read the frame header (the first 16 bytes) Use this to probe what kind of frame is coming from the module and read the frame payload later using the dedicated methods.

```
bool mono::redpine::ModuleCommunication::interruptActive() bool mono::redpine::ModuleCommunication::interruptActive()
```

True on read success, false otherwise

Parameters

- *rawFrame* - A pointer to the pre-allocated memory to hold the header

virtual = 0Read the first available frame from the modules input queue This function should be called when you are sure there is data pending

```
bool mono::redpine::ModuleCommunication::readManagementFrame(DataReceiveBuffer &buffer, ManagementFrame &frame) Return bool mono::redpine::ModuleCommunication::readManagementFrame(DataReceiveBuffer &buffer, ManagementFrame &frame)
```

True on success, false otherwise

Parameters

- *frame* - A reference to management frame placeholder object

virtual = 0Read a pending frame from the module, and interpret it as a response to an earlier frame. The response frame (the one that is read) and the provided frame (the request), are compared by commandId and the response status is validated. This there is any response payload data, this data is passed to the request frames *ManagementFrame::responsePayloadHandler* function. When this method returns the request frame object is converted to an response frame (RX_FRAME) with status and any payload data.

```
bool mono::redpine::ModuleCommunication::readManagementFrameResponse (DataReceiveBuffer  
                                &buffer, Management-Frame  
                                &request)
```

true on success, false otherwise.

Parameters

- request - A reference to the request frame, that is awaiting a response
- virtual = 0 Read a pending frame a Data frame. Data frame arrive out-of-order with anything else. Also, we expect that they deliver data to any open socket. This method read the data from the module and call the DataPayloadHandler function provided. This function then takes care of the actual data payload!

```
bool mono::redpine::ModuleCommunication::readDataFrame (DataReceiveBuffer      Return  
                                &buffer, DataPayload-  
                                Handler &payloadHan-  
                                dler)
```

true on succs, false otherwise

Parameters

- payloadHandler - A reference the data payload callback handler
- virtual = 0 Internal function to read from a memory address. This is used when communicating with the Redpine Modules Bootloader.

```
uint16_t mono::redpine::ModuleCommunication::readMemory (uint32_t      memoryAd- Return  
                                                         dress)
```

The 16-bit content of that address

Parameters

- memoryAddress - The address position to read from
- virtual = 0 Method to write to the module memory. This can be used when communicating with the bootloader of the module.

```
void mono::redpine::ModuleCommunication::writeMemory (uint32_t      memoryAddress, Parameters  
                                                         uint16_t value)
```

- memoryAddress - The address to write to
 - value - The 16-bit value to write at the address
- virtual = 0 Send a frame to the module

```
bool mono::redpine::ModuleCommunication::writeFrame (ManagementFrame *frame) Return
```

true on success, false otherwise

Parameters

- frame - A pointer to the frame to send

virtual = 0Sends a frame's payload data to the module. This payload data must be sent after the command / management frame has been sent to the module. This format of the data is determined by which command that was sent by the Command/Management frame. The RS9113 Software PRM is not clear on this data frame matter. But this code is from reverse engineering the example code by RSI.

```
bool mono::redpine::ModuleCommunication::writePayloadData(const      uint8_t      Return
                                                           *data,      uint16_t
                                                           byteLength,  bool
                                                           force4ByteMultiple)
```

true upon success, false otherwise.

Parameters

- data - A pointer to the raw data to write to the module
- byteLength - The length of the data in bytes
- force4ByteMultiple - Optional: Set to false, to bot enforce payload to be a 4-byte multiple

Public Members

```
uint16_t mono::redpine::ModuleCommunication::InterfaceVersion
```

Defines the communication protocol version to use. Redpine change the way FrameDescriptor headers return frame length in bootloader version 1.6. Version 1.5 uses a slightly different procedure.

The *Module* should write the communication protocol version here. So far we have only seen the values:

- Bootloader 1.5: 0xAB15
- Bootloader 1.6: 0xAB16

```
mbed::FunctionPointer mono::redpine::ModuleCommunication::interruptCallback
```

Interrupt callback function, called by the communication interface. The module provides the callback function, that gets called when ever the module triggers an interrupt.

struct Method that must be implemented, that will be called by the run loop It should only be scheduled by an hardware interrupt handler, and remove it self again from the run loop, after it has run. Structure to describe the data payload pointer and length for Data frame payloads

ModuleFrame

class A generic frame that is used to communicate which the module Subclasses of this will represent data or management frames. **Public Types**

```
enum type mono::redpine::ModuleFrame::FrameDirection
```

List of frame direction, sent og received

Values:

- The frame is a request to the module
- The frame is a response from module

```
enum type mono::redpine::ModuleFrame::RxTxCommandIds
```

List of request commands id's

Values:

- = 0x00Frame is data frame
- = 0x10
- = 0x11
- = 0x12
- = 0x13
- = 0x14
- = 0x15
- = 0x16
- = 0x17
- = 0x18
- = 0x19
- = 0x1B
- = 0x1C
- = 0x1D
- = 0x20
- = 0x21
- = 0x22
- = 0x23
- = 0x24
- = 0x25
- = 0x26
- = 0x29
- = 0x3A
- = 0x40
- = 0x41
- = 0x42
- = 0x43
- = 0x44DNS Resolution lookup command
- = 0x49Query the modules firmware version
- = 0x51HTTP Client, GET request
- = 0x52HTTP Client, POST request
- = 0xCD
- = 0xDE

enum type `mono::redpine::ModuleFrame::RxCommandIds`

List of response command Id's

Values:

- = 0x30Async connection accept request from remote wfd device
- = 0x61Async TCP Socket Connection Established
- = 0x62Async Socket Remote Terminate
- = 0x89Card Ready command, when module is initialized and ready

Public Functions

`ModuleFrame::ModuleFrame()`

If this object sits in a queue, this is the pointer the next in the queue Construct an empty frame with no properties set. The frame will not represent any data or any command

virtual Dealloc the frame if its subclass contains resources that should be removed gracefully.

Public Members

`ModuleFrame::~~ModuleFrame()`

Indicate if the frame is request or response

FrameDirection mono:

`uint16_t mono::redpine::ModuleFrame::length`

This is the payload data length

`uint8_t mono::redpine::ModuleFrame::commandId`

Either a RxTxCommandId or a TxCommand id

Public Static Attributes

`const uint8_t mono::redpine::ModuleFrame::size`

The size of frame in raw format

ModuleSPICommunication

class SPI based communication subclass **Public Types**

enum type `mono::redpine::ModuleSPICommunication::SpiRegisters`

List of available SPI Registers on RS9113

Values:

SPI Interrupt occurred register

Public Functions

`ModuleSPICommunication::ModuleSPICommunication()`

Initialize invalid SPI communication

`ModuleSPICommunication::ModuleSPICommunication` (mbed::SPI &*spi*, PinName *chipSelect*, PinName *resetPin*, PinName *interruptPin*)

Create a communication class, and assign a SPI hardware interface

Parameters

- *spi* - The initialized and ready SPI hardware module
- *chipSelect* - The SPI CS pin, active low
- *resetPin* - The GPIO pin connected to the module reset line (active low)
- *interruptPin* - The pin where the modules interrupt signal is connected

virtual Trigger the reset line to the module. A reset will put the module into bootloader, where a firmware image can be selected. Before calling *initializeInterface*, you must reset the module. The reset command has to wait while the module resets, this means the function will block for approx 3 ms.

virtual Send a interface initialization command to the module.

`void ModuleSPICommunication::resetModule()` bool ModuleSPIComr
Internal function to read content of a SPI register SPI registers are only available to the modules SPI interface

Return

the content of the 8-bit register

Parameters

- `reg` - The register to read

virtual Internal function to read from a memory address. This is used when communicating with the Redpine Modules Bootloader.

`uint16 ModuleSPICommunication::readMemory(uint32_t memoryAddress)` Return

The 16-bit content of that address

Parameters

- `memoryAddress` - The address position to read from

virtual Method to write to the module memory. This can be used when communicating with the bootloader of the module.

`void ModuleSPICommunication::writeMemory(uint32_t memoryAddress, uint16_t value)` Parameters

- `memoryAddress` - The address to write to
- `value` - The 16-bit value to write at the address

virtual Polls the module for waiting input data. This can be frame responses or any other data, waiting to be read by us. The method uses the current communication interface method to wait for pending input data. This function blocks, until data is ready. This function can be used it interrupts are not applicable

`bool ModuleSPICommunication::pollInputQueue()` Return

true if there is data to read, false otherwise

virtual Return true if interrupt is active. The module will keep the interrupt pin high until no more input is present.

virtual Read the frame header (the first 16 bytes) Use this to probe what kind of frame is coming from the module and read the frame payload later using the dedicated methods.

`bool ModuleSPICommunication::interruptActive()` bool ModuleSPIComr

True on read success, false otherwise

Parameters

- `rawFrame` - A pointer to the pre-allocated memory to hold the header

virtual Read the first available frame from the modules input queue This function should be called when you are sure there is data pending

```
bool ModuleSPICommunication::readManagementFrame (DataReceiveBuffer &buffer, Man-      Return
agementFrame &frame)
```

True on success, false otherwise

Parameters

- frame - A reference to management frame placeholder object

virtual Read a pending frame from the module, and interpret it as a response to an earlier frame. The response frame (the one that is read) and the provided frame (the request), are compared by commandId and the response status is validated. This there is any response payload data, this data is passed to the request frames *ManagementFrame::responsePayloadHandler* function. When this method returns the request frame object is converted to an response frame (RX_FRAME) with status and any payload data.

```
bool ModuleSPICommunication::readManagementFrameResponse (DataReceiveBuffer      Return
&buffer,      Manage-
mentFrame      &re-
quest)
```

true on success, false otherwise.

Parameters

- request - A reference to the request frame, that is awaiting a response

virtual Read a pending frame a Data frame. Data frame arrive out-of-order with anything else. Also, we expect that they deliver data to any open socket. This method read the data from the module and call the DataPayloadHandler function provided. This function then takes care of the actual data payload!

```
bool ModuleSPICommunication::readDataFrame (DataReceiveBuffer &buffer, DataPayload-      Return
Handler &payloadHandler)
```

true on succs, false otherwise

Parameters

- payloadHandler - A reference the data payload callback handler

virtual Send a frame to the module

```
bool ModuleSPICommunication::writeFrame (ManagementFrame *frame)      Return
```

true on success, false otherwise

Parameters

- frame - A pointer to the frame to send

virtual Sends a frame's payload data to the module. This payload data must be sent after the command / management frame has been sent to the module. This format of the data is determined by which command that was sent by the Command/Management frame. The RS9113 Software PRM is not clear on this data frame matter. But this code is from reverse engineering the example code by RSI.

```
bool ModuleSPICommunication::writePayloadData (const uint8_t *data, uint16_t byte-      Return
Length, bool force4ByteMultiple)
```

true upon success, false otherwise.

Parameters

- `data` - A pointer to the raw data to write to the module
- `byteLength` - The length of the data in bytes
- `force4ByteMultiple` - Optional: Set to false, to not enforce payload to be a 4-byte multiple

virtual Called when the system powers on after a reset. You can override this method. It gets called right after the system power on for the first time, or after a reset condition. Use this method to setup / initialize components and data structures. It is only called once for each system reset.

virtual Called right before the MCU goes into sleep mode. You can override this method to get sleep notifications. Before the CPU stop executing instructions and goes into low power sleep mode, this method gets called. Use this method to prepare your data or MCU components for sleep. Help preserve battery by power down any peripheral, that is not needed during sleep. This method can be called many times during your apps life cycle. (That is between resets.) After each sleep period, when the MCU wakes the [*onSystemWakeFromSleep*](#) is guaranteed to be called.

virtual Override to get notified when system wakes from sleep. You can override this method to get wake-up notifications. When the CPU starts executing instructions and the power system has powered up all peripherals - this method gets called. Use this method to setup your app to resume after sleep mode. This method can be called many times during your apps life cycle. (That is between resets.)

virtual Override to get notified when the battery voltage reaches a critical level. You can override this method to get *battery low* notifications. When this methods gets called, you have some time to finish critical tasks. That might writing state to file system or transfer data over the network. Depending on the health of the battery, the time between this notification and the actual system enforced power off, might vary. In contrast to the other *power aware* methods this is only called once in the application life cycle. After the enforced power off, when the battery is charged, the system will automatically reset.

Protected Functions

```
void ModuleSPICommunication::onSystemPowerOnReset ()
```

```
void ModuleSPICommunication::onSystemPowerOnReset ()
```

Auxillary function to transfer C1 and C2 commands

Return

The Redpine command status code (0x58 is success)

Parameters

- `c1` - The C1 command byte, the SPI response here is ignored
- `c2` - The C2 command byte, the response for this byte is the cmd status

```
CommandStatus ModuleSPICommunication::sendC1C2 (CommandC1 c1, CommandC2 c2)
```

Auxillary function to transfer C1 and C2 commands

Return

The Redpine command status code (0x58 is success)

Parameters

- `c1` - The C1 command object, the SPI response here is ignored
- `c2` - The C2 command object, the response for this byte is the cmd status

`bool ModuleSPICommunication::waitForStartToken (bool thirtyTwoBitMode)`

Auxillary function to poll module for a start token Transfers zeros to the module until some other than zeroes are received

Return

True is module send a START_TOKEN command, false on timeout or unknown response.

Parameters

- `thirtyTwoBitMode` - Defaults to false, set TRUE for 32-bit mode SPI

`bool ModuleSPICommunication::readFrameDescriptorHeader (frameDescriptorHeader
*buffer)`

Reads the 4 bytes of a data og management frame header.

The module sends a 4 byte header before sending the 16-byte frame descriptor. Used to query a pending frame for its length.

Return

true is success, false otherwise

Parameters

- `buffer` - A pointer to where the length is stored

`bool ModuleSPICommunication::readFrameBody (frameDescriptorHeader &frameHeader,
SPIReceiveDataBuffer &buffer)`

Read a pending frame, based on a frame descriptor header (describing the length), read the real frame - data og mgmt.

Return

true on success, false otherwise

Parameters

- `frameHeader` - Reference to the header for the pending frame
- `buffer` - A reference to the buffer where the frame is to be stored

`int ModuleSPICommunication::spiWrite (const uint8_t *data, int byteLength, bool thirtyTwoBitFormat)`

Convenience function to write data to mbed SPI The method only return the last received input from SPI, all other read bytes are discarded.

Use this function if you need to write multiple bytes at once. The method can transfer the data as 32-bit values if you choose.

Return

the last read value on the SPI bus

Parameters

- `data` - A ponter to the data buffer, that will be written
- `byteLength` - The length of the data to write, in bytes
- `thirtyTwoBitFormat` - Set this to true to use 32-bit mode, default is false

```
void ModuleSPICommunication::setChipSelect (bool active)
```

Sets the SPI chip select for the module. This must be called before all SPI write or reads. This method automatically handles the value of the chip select pin.

Parameters

- `active` - Set this to `true` to activate the chip select, `false` otherwise.

mbed API

If you need to interact with the *GPIO*, *Hardware interrupts*, *SPI*, *I2C*, etc. you should use the *mbed* layer in the SDK. Take a look at the documentation for mbed:

[mbed documentation](#)

Release Notes

An overview of what features are introduced in the different releases.

Release 1.7

June 23rd, 2017

This is the 2017 summer release of Mono Framework. It includes major new features along with bug fixes.

In our effort to streamline the API and make embedded development on Mono easy and fast, we have introduced new C++ classes and augmented existing ones.

In this release note we shall briefly mention all new features in 1.7 and discuss the most important in-depth.

Update 1.7.3

July 18th, 2017

A major change in this release is the `TouchResponder` and `ResponderView` interfaces. The touch handler method are now not capitalized. However, legacy code will continue to work. You should still transition your code to use the lower case variants of the touch handler methods.

- Added a Wifi initialization class, to ease use of Wifi. [Feature](#)
- Fixed typo in `GraphView` class, `BufferLenght` is now: `BufferLength`
- Added C function pointer callbacks for Redpine module. [Improvement](#)
- Better Arduino API support in Module communication class, added parametersless constructor. [\[Improvement\]\[https://github.com/getopenmono/mono_framework/issues/19\]](https://github.com/getopenmono/mono_framework/issues/19)
- Added C function pointer callbacks for network classes. [Improvement](#)
- `HttpPost` does not append dummy bytes. [Fix](#)
- `PowerSaver` class can be assigned. [Fix](#)
- `TouchResponder` uses more non-capitalized syntax. [Improvement](#)
- Added mono owned SD `FileSystem` class, supporting sleep and wake. [Feature](#)
- More deprecated upper-cased method names on: `Queue`, `Power Aware interface`, `TouchResponder` and more.

- Added more documentation to core classes: *String*, *Queue* and *Power Aware*
- *GenericQueue* will not compile if you use an queue item not inheriting from `IQueueItem`.

Update 1.7.1

June 28th, 2017

This minor update contains two main changes:

- Fix: The power to the RTC is no longer turned off in sleep
- Feature: Added a `setIcon` method to `IconView` as needed by the *Chores* app

Original 1.7 Release

List of new features and fixes

New features

This is a major release with many new features, including:

- **Icon system:** A set of predefined icons with symbols like: wifi, temperature, mute, play, pause, etc.
- **Scenes:** A class for grouping UI elements into a *scene*. Then you can navigate between scenes.
- **Battery level** Class for getting the remaining battery power
- **Analog low-pass filter and filtered inputs** Classes for low-pass filtering analog inputs
- **Hysteresis triggers** Class that implements a Schmitt-trigger, for detecting analog values has exceeded a threshold.
- **URL Encoding** A new class can URL encode strings, for use with HTTP GET query parameters
- **Time conversions** The `DateTime` class now builds on top of the standard *libc* time APIs
- **RingBuffer** Added *mbed's* circular buffer class.
- **DHT One wire protocol** We added a class that implements the DHT one wire protocol interface
- **Pin change interrupts** Our interrupt classes now support pin change events. (Rise + Fall events)

Fixes

This release also fixes a number of bugs, these include:

- Increased the buffer size for *Redpine module's* receive data buffer
- `PowerSaver` class is more robust when event fire in dimming animation
- Alignment issues in `OnOffButtonView` graphics
- Buzzer API now uses CPU interrupts to end buzzing
- *Major touch input* fix. Y six was ignored until now. Touch work quite precise now.
- Fixed a graphics bug in incremental repaints of `TextLabelview`
- Touch system does no longer sample input when there is no touch responders
- Fixed bug in queue system, that caused the task queue to become unstable

- RTC wakes does not trigger I/O ports initialization
- Fixed issue that caused `ScheduledTask`'s to wake mono each second

Deprecated methods

With the new release we are deprecating a couple of redundant methods. Mostly methods with unnecessary verbose names. If you use a deprecated method, you will get a compiler warning.

You should port your code to not use the deprecated methods, since they will be removed in future releases.

Features in-depth

Let us take a moment to examine some of the most important features and discuss why they are important.

Icons

We repeatedly found ourselves in need of having icon-decorated push buttons. Also, we again and again needed to display some status info that would be nice to convey in form of small icons.

Therefore we have introduced a monochrome icon bitmap-format, that can be stored in flash memory and displayed in a few lines of code. Further, we added a set of commonly used icons to the framework. These include icons for speaker, wifi, battery, play, pause etc.

You can create your own icons from bitmap images using our free tool [img2icon](#)

Example

To use icons are incredibly easy, since we added a `IconView` class to display them on the display:

```
#include <icons/speaker-16.h>

IconView icn(geo::Point(20,20), speaker16);
icn.show();
```

You can also change the coors of icons by using the `setForegroundand` `setBackground` methods on `IconView`.

Scenes

We found we needed a way to switch between different view in applications. Say, the [Alarm Clock](#) app on MonoKiosk needs to have 3 different scenes to show:

- Normal alarm clock time display (Main Scene)
- Set the time configuration scene (set time scene)
- Set the alarm time scene (set alarm scene)

Each scene takes up the entire display, and we needed to have an easy way to switch between these different scenes.

Inspired by what iOS and Android uses (*Segues* and *Intents*) we have created a class called `SceneController`. This represents a scene of contents. A scene is logical container of `View`-based objects. You add *views* to a scene. The scene now controls the `show` and `hide` methods for all added views.

This means to can **show** and *hide* all the views in the scene at once, by calling the equivalent methods on the *scene* object. Further, scene implement an interface for implementing transitions between different scenes. This is a enforced structure that makes it easy to to setup and teardown, related to scene changes.

Example

To use a scene you simply instantiate it and then add views to it.

```
#include <icons/speaker-16.h>

using mono:ui;
using mono::geo;

SceneController scn;

TextLabelView helloLabel(Rect(0,100,176,20), "Hi, I'm Mono!");
IconView icn(Point(50,100), speaker16);

scn.addView(helloLabel);
scn.addView(icn);

scn.show();
```

RTC Time integrated with C library functions

Since *Release 1.4.2* we had the RTC time and date tracking enabled. However, the RTC system was integrated only with the highlevel `DateTime` class. Now we have hooked it into the lower level C API's provided by Std. Libc. This means you can use functions like `time` and `localtime`. These also work in conjunction with `DateTime`.

Example

```
time_t now = time();
DateTime dtNow(now);
printf("Time and date: %s\\r\\n", dtNow.toISO8601()());
```

Critical bug fix

We have also fixed 2 major bugs relating to power consumption in sleep. Due to a bug in the `ScheduledTask` class, Mono consumed too much power in sleep mode. Further, the entire I/O (GPIO) system was initialized upon the RTC seconds increments, while in sleep. This introduced even a power consumption overhead.

In 1.7 we have fixed these issues, and achieved low power consumption in sleep mode, as we originally aimed for: 35 μ A in sleep mode.

Download

Download OpenMono SDK for your platform [here](#)

Release 1.6

February 10th, 2017

We are happy to announce release 1.6 of the Mono SDK. This release focuses on improvement in the software library, introducing new font and timer systems.

New features

Here is a short intro to the the new features of Release 1.6. Later we will discuss some of them in detail, also we strive to write tutorials on the all the new features.

New proportionally spaced fonts

We have taken the font format from [Adafruits Gfx](#) library, and made support for it in the text rendering system. This means the `TextRender` class now support the Adafruit *GFXfont* format. The class `TextLabelView` can use the new fonts as well. In fact, the default font face is changed to a beautiful proportionally spaced sans serif type.

We have included all the available font faces from Adafruits library. These include Italic and bold faces of Serif and Sans Serif fonts.

The old mono spaced fonts are still available, but you have to explicitly set them in your code.

Wake Mono on timers

For a long time we wanted this feature: To sleep mono and then wake up at a specific point in the future. With our new `ScheduledTask` class, this is possible! A `ScheduledTask` is a object that calls a function at a given point in time. You provide it with a function pointer and a `DateTime`. This function will then be called at that time, also if Mono are in sleep at that point. (This is opt-in.)

Using `ScheduledTask` we can create a temperature logger, that sleeps and automatically wakes up periodically to measure the temperature.

Analog API (mbed)

In this release we finally had the time to implement `mbed::AnalogIn` functionality. This means you can easily use the ADC to measure the voltage on the input pins. Almost all pins on our Cypress PSoC5 MCU can be routed to the ADC, exceptions are the USB pins and the SIO (*Special I/O*) pins. One of these SIO pins are the `J_TIP` pin.

An example of how you read the voltage level on the 3.5mm jack connector's `J_RING1` pin is:

```
mbed::AnalogIn ring1(J_RING1);  
float normalizedValue = ring1;  
uint16_t rawValue = ring1.read_u16();
```

Thats it! The system take care of setting the right pins mode, and setup the routing of the analog interconnect.

Power saver

We found ourself keep writing code to auto-sleep mono. Just like smartphone will automatically dim the display and eventually go into standby, to spare the battery. For Mono we wanted it to be opt-in, so you always start a new application project, with a clean slate.

Therefore we have introduced the class `PowerSaver` in 1.6. This class dims the display after a period of inactivity. After even further inactivity, it automatically trigger sleep mode.

Inactivity is no incoming touch events. You can manually ping or momentarily deactivate the class to keep mono awake. But by default it will trigger sleep mode, if nothing happens to 20 secs.

You should add the `PowerSaver` as a member variable to your `AppController` class, to enable its functionality.

More new features

- `TextLabelView` does better text alignment (horizontal and vertical)
- Multiline text in `TextLabelView`
- Callback for Wifi join error handler
- `DateTime` now support negative numbers in `addTime()` methods
- `HttpPostClient` tries to fix Redpine bug, by appending whitespace to body data.

Download

Release package hashes

Upon request we have added *sha256* (*sha1* for Windows SDK) hash files for each installer package. In this way you can validate you copy of the installer package against our copy, by comparing hash values.

[Go to the release packages](#)

Release 1.5

January 23rd, 2017

In this 1.5 release we have focused on tooling, there is no feature additions to the library code. We have added a brand new graphical user interface to manage creation of mono application projects.

Introducing Monomake UI application

We have added a UI for our standard console based *monomake* tool. The UI make the task of creating new projects and installing existing applications on Mono easy and friendly. We did this to help people who are not familiar with the terminal, and just want to install an application on Mono.

However, the best of all is this new Monomake UI enables us to achieve one-click application installs through MonoKiosk. Our new tool registers the URL scheme: `openmono://` and associate with ELF files.

Integration with Atom

We know the choice of editor is kind of a religion. Therefore we won't force a specific editor on you. However, should linger undecided for an editor, GitHub have made a fine cross-platform editor called `Atom`. If you have Atom installed on your system, our new Monomake UI takes advantage of it right away. New project can be opened in Atom and you can open recent projects in Atom.

Should you choose to use Atom, we strongly recommend installing the `autocomplete-clang` package. This package enables as-you-type auto complete suggestions. It also display the inline documentation present in Mono Framework. (Windows users will need to install the `clang compiler`.)

Bridging the gap to our Arduino plugin

We have to admit, have we unintentionally stalled the Arduino compatibility plug-in for the Arduino IDE. In this release we right that by bringing the 1.5 library to Arduino.

Release 1.4

November 21th, 2016

We corrected some critical power issues in this release and bugs in the *TextLabel* object.

Additions

- We added a HTTP OST class called: *HttpPostClient*, that work like the existing *HttpClient*. The *PostClient* uses 2 callbacks where you must provide POST body length and data.
- (1.4.2 only) Real Time Clock enabled. Mono now has a running system clock, that can be accessed through the `DateTime` class.
- (1.4.3 only) Build system always `clean` before building - since we have experienced errors in the dependency system

Bug Fixes

- Fixed issue that caused the voltage on VAUX to changed when Mono went to sleep mode. Now (by default) VAUX is 3.3V and 0V in sleep.
- Fixed issue that caused J_TIP to change voltage levels in wake and sleep modes. We controlled this issue by limiting the voltage changes. J_TIP is 0V by default when Mono is running and in sleep. But when the USB is connected leakage currents will raise the voltage on J_TIP to 1.7V.
- Fixed repaint error in *TextLabelView*, caused `setText` to fail.
- Fixed issue in *TextRender*, that might cause the render to skip glyphs

Release 1.3

October 18th, 2016

We have discovered some annoying bugs in version 1.2 and therefore we have released version 1.3 that resolves many issues.

SDK 1.3 adds no new features, it is a purely bug fix release.

Fixes Serial port reset bug

Many users have experienced that the `make install` did not work properly. They always needed to do a *force bootloader* reset manually, before being able to upload new apps to mono.

The issue was that the serial DTR signal did not trigger a reset on Mono. This has been resolved in 1.3, and `make install` works like it should.

I2C functionality restored

A linking error in 1.2 broke the I2C communication. This meant the temperature and accelerometer could not be used. This is fixed in 1.3

No more spontaneous wake ups

An issue with false-positive interrupts from the Wireless and Display chips while in sleep is resolved. Now sleep mode disables interrupts from these components, this means they cannot wake Mono.

When in sleep mode, the power supply (3V3 net) for all peripherals is turned off - to preserve battery. But capacity on the power net means that the voltage stays high and slowly falls towards zero. This triggered random behaviour in the IC's causing flicker on the interrupt signals.

If you had issues with Mono waking up approximately 8-10 secs after going to sleep, the fix will resolve this issue.

HttpClient parsed port number incorrectly

The feature added in 1.2 allowed the class HttpClient to handle URL's like: `http://mydomain.com:8080/hello`. But a bug in the overloaded assignment operator, caused the port number to be lost during assignment. This effectively broke support for port numbers.

This issue have been resolved in 1.3

Structural change in libraries

The SDK has until now consisted to code from 4 distinct repositories:

- `mono_psoc5_library`
- `mbedComp`
- `mbed`
- `mono_framework`

Starting with 1.3 we have merged all these into *mono_framework*, since the other 3 were used only by *mono_framework*. This simplifies the building process of the library, hopefully reducing the number of bugs.

On a side note, @jp is working to setup *continous integration* tests for the repositories.

Download

- [macOS](#)
- [Windows](#)
- [Linux](#) (Debian based distorts)

Release 1.2

September 12th, 2016

We have a new version of the framework out, with mostly bug fixes

HttpClient supports customs TCP ports

The interim class for doing HTTP get requests now has the ability to use other ports than 80. You simply insert a custom port in the passed URL - like: `http://192.168.1.5:3000/api/hello`

The request will then be directed to port 3000.

New minimal project template

When you create new projects with `monomake project myNewProject`, they are pretty verbose and cluttered with comments. These serve as a help to new developers, but they become irrelevant when you get to know the API.

`monomake` now has a `--bare` switch when creating new projects, that will use a minimalistic template without any comments or example code. You use it like this:

```
$ monomake project --bare my_new_project
```

New 30 pt font added

The API now includes a larger font, that you can optionally use in `TextLabelView`'s:

```
#include <ptmono30.h>
mono::ui::TextLabelView lbl(Rect(...), "Hello");
lbl.setFont(Pt_Mono_30);
lbl.show();
```

Since the font are bitmaps, they use significant amounts of memory. Therefore the new font is only included in your application if you actually use it. That is, use must define it with the `#include` directive, as shown above.

See the API for changing the font `TextLabelView.setFont(...)`

Shorthand function for asynchronous function calls

To make dispatching function calls to background handling easier, we added a new global function called: `async` in the `mono` namespace. The function injects your function into the run loop, such that it will be handled at first coming opportunity. The `async` call is really just a shorthand for a `Timer` that has a timeout of 0.

Example with C++ method:

```
mono::async<MyClass>(this, &MyClass::handleLater);
```

Example with C function:

```
mono::async(&myFunction);
```

Bug fixes and other improvements

- Optimized text glyph rendering in `TextLabelView`
- SD Card SPI clock speed increased to 8.25 MHz
- Fixed cropping bug in `ImageView`
- Fixed bug in `BMPImage` that caused a crash then invoking *copy constructor*.
- Fixed bug in the `Queue` class, that could caused Mono to freeze

- Fixed wrong premise on `String`'s memory management.
- Fixed issue that caused `make clean` to not remove all object files
- Fixed `make clean` such that it works on Windows Command Prompt - and only PowerShell.

Download

Goto our Documentation page to download the new SDK version:

<http://developer.openmono.com/en/latest/getting-started/install.html>

Release 1.1

July 20th, 2016

Finally, after much work we have a new release of Mono's software library and tool chain.

The new v1.1 release is available from our [Developer site](#). Let me go through the biggest improvements and additions:

New default system font

The existing font was pretty old fashioned to put it nicely. We have a new font that takes up a little more room, but it is much better. We have created a bitmap font system, such that we can add more fonts later. If you have used `TextLabelView` (with text size 2), you will instantly take advantage of the new font.

Wake-up works (No more resets)

We have fixed the issue that required us to do a reset immediately after wake up. Now mono resumes normal operation after a wake up. Therefore you will not see the `SoftwareResetToApplication` call in *AppController* in your new projects.

If you wish your existing apps to take advantage of this, simply remove the call to `SoftwareResetToApplication`.

Sleep while USB is connected

Further, Mono is now able to sleep when connected to the computer. Previously the USB woke up mono when connected. Now you can shut off Mono while the USB is connected.

API for the Buzzer

In other news we have added a generic API for the buzz speaker. It is very simple: you define for how long the buzz sound should last. We plan to add frequency tuning in later releases.

Signed installer on Windows

We have received a code signing certificate, and used it to sign the installer and the USB Serial Port driver. This means that Windows *should not* yell that much about dangerous, possibly harmful binaries. We still need a certificate for the Mac installer, so Mac users - bear with us.

New default colors

The framework defines a set of standard colors for common UI elements, such as: text, borders, highlights and background. Some of these are now changed. Specifically we changed the background color to pitch black, to increment the contrast.

Type completion in Atom

Should you choose to use GitHub's excellent [Atom](#) code editor, you can now get type completion with documentation. This works much like *Intellisense* in Visual Studio or auto completion in Xcode.

To enable type completion in Atom you need to install [clang](#) on [Windows](#) and [Linux](#) systems. Then, you must add the [AutoComplete-Clang](#) plugin to Atom. Create a new project with `monomake` and open the project in Atom. Voila!

Bug fixes and more

- More accessors in UI Classes (*ButtonView*, *TextLabelView*)
- Default screen brightness is now 100%, instead of 50%
- Less verbose on serial port
- Added PinNames for coming Mono Schematics
- Fixed issue that caused USB not to work after sleep
- *QueueInterrupt* is now setting the correct `snapshot` value
- Fixed issue when color blending 100% opaque or transparent pixels
- Fixed issue that left out a single dot when drawing outlined rectangles

Documentation

On [developer.openmono.com](#) you can now choose between the *latest* documentation (v1.1) and the older *v1.0* version. By default we will show you the *latest* documentation version.

Here on our developer site, we collect all learning resources about mono. You will find initial getting started guides, followed by tutorials on specific topics, in-depth articles and last (but not least) the API reference docs.

We prioritize to get as much text published fast, instead of keeping our cards close. We hope you like our decision. Anyway - should you encounter anything you would like to correct - see: [Contributing](#) ->

Go straight to our download page: [Download SDKs](#) ->

Implementation status

As with this site, the software is also being finished right now. The *current status* page include a list of all the planned features of the framework - and their implementation status:

- **Current Status**

Contribute

The source to this documentation is available publicly on our GitHub. Please just fork it, correct all our *bad english* - and submit a pull request. We would just loooove that!

You can contribute to this documentation through the [GitHub repository](#). Note that everything you contribute will be free for anyone to use because it falls under the site license.

What is Mono anyway?

Haven't you heard the word? Mono is an embedded hardware platform that brings all the goods from [Arduino](#) and [mbed](#) to a whole new level! No fiddling with voltage levels, open-drain vs pull-up configurations. We take care of all this low-level stuff for you. You just focus on building your application, taking advantage of all the build-in hardware functionalities - like:

- Arm Cortex-M3 MCU
- Touch display
- Battery
- Wifi
- Bluetooth
- Accelerometer
- Temperature Sensor
- SD Card
- General Purpose 3.5mm jack connector

On this developer documentation site, you learn how to use all these features through our high-level API.

A

AbstractButtonView::AbstractButtonView (C++ function), 147
AppRunLoop::addDynamicTask (C++ function), 137
AppRunLoop::CheckUsbDtr (C++ function), 137
AppRunLoop::checkUsbUartState (C++ function), 139
AppRunLoop::exec (C++ function), 137
AppRunLoop::process (C++ function), 139
AppRunLoop::processDynamicTaskQueue (C++ function), 139
AppRunLoop::quit (C++ function), 138
AppRunLoop::removeDynamicTask (C++ function), 137
AppRunLoop::removeTaskInQueue (C++ function), 139
AppRunLoop::setResetOnUserButton (C++ function), 137

B

BackgroundView::BackgroundView (C++ function), 148
BackgroundView::Color (C++ function), 148
BackgroundView::repaint (C++ function), 148
BackgroundView::setBackgroundColor (C++ function), 148
ButtonView::ButtonView (C++ function), 149
ButtonView::repaint (C++ function), 151
ButtonView::setBackground (C++ function), 150
ButtonView::setBorder (C++ function), 150
ButtonView::setHighlight (C++ function), 150
ButtonView::setRect (C++ function), 151
ButtonView::setText (C++ function), 149, 150
ButtonView::TextLabel (C++ function), 151

C

Color::alphaBlend (C++ function), 195
Color::blendAdditive (C++ function), 195
Color::blendMultiply (C++ function), 195
Color::Blue (C++ function), 195
Color::Color (C++ function), 194
Color::Green (C++ function), 195
Color::invert (C++ function), 195

Color::Red (C++ function), 194
Color::scale (C++ function), 195
Color::toString (C++ function), 195

D

DateTime::addDays (C++ function), 123
DateTime::addHours (C++ function), 123
DateTime::addMinutes (C++ function), 123
DateTime::addSeconds (C++ function), 123
DateTime::addTime (C++ function), 123
DateTime::DateTime (C++ function), 121
DateTime::Days (C++ function), 122
DateTime::fromISO8601 (C++ function), 126
DateTime::Hours (C++ function), 122
DateTime::incrementSystemClock (C++ function), 126
DateTime::isLeapYear (C++ function), 126
DateTime::isValid (C++ function), 122
DateTime::LocalTimeZoneHourOffset (C++ member), 126
DateTime::Minutes (C++ function), 122
DateTime::Month (C++ function), 123
DateTime::now (C++ function), 126
DateTime::Seconds (C++ function), 122
DateTime::setSystemDateTime (C++ function), 126
DateTime::toBrokenDownUnixTime (C++ function), 122
DateTime::toDateString (C++ function), 122
DateTime::toISO8601 (C++ function), 122
DateTime::toJulianDayNumber (C++ function), 122
DateTime::toLibcUnixTime (C++ function), 122
DateTime::toRFC1123 (C++ function), 122
DateTime::toString (C++ function), 122, 124
DateTime::toTimeString (C++ function), 122
DateTime::toUnixTime (C++ function), 122
DateTime::toUtcTime (C++ function), 122
DateTime::Year (C++ function), 123
DigitalOut::DigitalOut (C++ function), 174
DigitalOut::setMode (C++ function), 174
DisplayPainter::BackgroundColor (C++ function), 197
DisplayPainter::CanvasHeight (C++ function), 197
DisplayPainter::CanvasWidth (C++ function), 197

DisplayPainter::DisplayController (C++ function), [197](#)
DisplayPainter::DisplayPainter (C++ function), [196](#)
DisplayPainter::drawChar (C++ function), [199](#)
DisplayPainter::drawCircle (C++ function), [200](#)
DisplayPainter::drawFillRect (C++ function), [198](#)
DisplayPainter::drawHLine (C++ function), [199](#)
DisplayPainter::drawLine (C++ function), [198](#)
DisplayPainter::drawPixel (C++ function), [197](#), [198](#)
DisplayPainter::drawRect (C++ function), [199](#)
DisplayPainter::drawVLine (C++ function), [199](#)
DisplayPainter::ForegroundColor (C++ function), [196](#)
DisplayPainter::IsAntialiasedDrawing (C++ function), [197](#)
DisplayPainter::setBackgroundColor (C++ function), [196](#)
DisplayPainter::setForegroundColor (C++ function), [196](#)
DisplayPainter::swap (C++ function), [200](#)
DisplayPainter::useAntialiasedDrawing (C++ function), [197](#)
DnsResolver::DnsResolver (C++ function), [187](#)
DnsResolver::DomainName (C++ function), [188](#)
DnsResolver::IpAddress (C++ function), [188](#)

F

File::appendLine (C++ function), [176](#)
File::appendString (C++ function), [175](#)
File::exists (C++ function), [174](#)
File::readFirstLine (C++ function), [175](#)
File::size (C++ function), [174](#)
File::writeString (C++ function), [175](#)
FileSystem::FileSystem (C++ function), [177](#)
FileSystem::onSystemEnterSleep (C++ function), [177](#)
FileSystem::onSystemWakeFromSleep (C++ function), [177](#)

G

GraphView::GraphView (C++ function), [153](#), [154](#)
GraphView::repaint (C++ function), [154](#)

H

HttpClient::dnsComplete (C++ function), [189](#)
HttpClient::dnsResolutionError (C++ function), [189](#)
HttpClient::HttpClient (C++ function), [189](#)
HttpClient::httpCompletion (C++ function), [189](#)
HttpPostClient::dnsComplete (C++ function), [191](#)
HttpPostClient::HttpPostClient (C++ function), [190](#)
HttpPostClient::post (C++ function), [191](#)

I

IApplicationContext::Instance (C++ member), [143](#)
IconView::Background (C++ function), [157](#)
IconView::Foreground (C++ function), [157](#)
IconView::IconView (C++ function), [156](#)
IconView::repaint (C++ function), [157](#)
IconView::setBackground (C++ function), [156](#)

IconView::setForeground (C++ function), [156](#)
IconView::setIcon (C++ function), [157](#)
ImageView::Crop (C++ function), [158](#)
ImageView::ImageView (C++ function), [158](#)
ImageView::repaint (C++ function), [158](#)
ImageView::setCrop (C++ function), [158](#)
ImageView::setImage (C++ function), [158](#)
INetworkRequest::HasFailed (C++ function), [192](#)
INetworkRequest::IsCompleted (C++ function), [192](#)
INetworkRequest::setState (C++ function), [193](#)
INetworkRequest::State (C++ function), [192](#)
INetworkRequest::triggerCompletionHandler (C++ function), [193](#)
INetworkRequest::triggerDirectErrorHandler (C++ function), [193](#)
INetworkRequest::triggerQueuedErrorHandler (C++ function), [193](#)
IPowerManagement::__busySleep (C++ member), [214](#)
IPowerManagement::__shouldWakeUp (C++ member), [214](#)
IRTCSystem::__shouldProcessScheduledTasks (C++ member), [144](#)

M

ManagementFrame::abort (C++ function), [217](#)
ManagementFrame::commit (C++ function), [216](#)
ManagementFrame::commitAsync (C++ function), [216](#)
ManagementFrame::dataPayload (C++ function), [217](#)
ManagementFrame::ManagementFrame (C++ function), [216](#)
ManagementFrame::payloadLength (C++ function), [217](#)
ManagementFrame::rawFrameFormat (C++ function), [217](#)
ManagementFrame::responsePayloadHandler (C++ function), [217](#)
ManagementFrame::triggerCompletionHandler (C++ function), [217](#)
ManagementFrame::writeFrame (C++ function), [217](#)
Module::discardIfNeeded (C++ function), [221](#)
Module::handleDataPayload (C++ function), [221](#)
Module::handleSleepWakeUp (C++ function), [221](#)
Module::initAsyncFrame (C++ function), [221](#)
Module::initialize (C++ function), [220](#)
Module::Instance (C++ function), [220](#)
Module::IsNetworkReady (C++ function), [221](#)
Module::Module (C++ function), [221](#)
Module::moduleEventHandler (C++ function), [220](#)
Module::moduleSingleton (C++ member), [222](#)
Module::onNetworkReady (C++ function), [221](#)
Module::onSystemBatteryLow (C++ function), [221](#)
Module::onSystemEnterSleep (C++ function), [221](#)
Module::onSystemPowerOnReset (C++ function), [221](#)
Module::onSystemWakeFromSleep (C++ function), [221](#)
Module::setupWifiOnly (C++ function), [220](#)

- ModuleCommunication::bufferIsDataFrame (C++ function), 223
- ModuleCommunication::bufferIsMgmtFrame (C++ function), 223
- ModuleFrame::~~ModuleFrame (C++ function), 227
- ModuleFrame::ModuleFrame (C++ function), 227
- ModuleSPICommunication::initializeInterface (C++ function), 228
- ModuleSPICommunication::interruptActive (C++ function), 228
- ModuleSPICommunication::ModuleSPICommunication (C++ function), 227
- ModuleSPICommunication::onSystemBatteryLow (C++ function), 230
- ModuleSPICommunication::onSystemEnterSleep (C++ function), 230
- ModuleSPICommunication::onSystemPowerOnReset (C++ function), 230
- ModuleSPICommunication::onSystemWakeFromSleep (C++ function), 230
- ModuleSPICommunication::pollInputQueue (C++ function), 228
- ModuleSPICommunication::readDataFrame (C++ function), 229
- ModuleSPICommunication::readFrame (C++ function), 228
- ModuleSPICommunication::readFrameBody (C++ function), 231
- ModuleSPICommunication::readFrameDescriptorHeader (C++ function), 231
- ModuleSPICommunication::readManagementFrame (C++ function), 229
- ModuleSPICommunication::readManagementFrameResponse (C++ function), 229
- ModuleSPICommunication::readMemory (C++ function), 228
- ModuleSPICommunication::readRegister (C++ function), 228
- ModuleSPICommunication::resetModule (C++ function), 228
- ModuleSPICommunication::sendC1C2 (C++ function), 230
- ModuleSPICommunication::setChipSelect (C++ function), 231
- ModuleSPICommunication::spiWrite (C++ function), 231
- ModuleSPICommunication::waitForStartToken (C++ function), 230
- ModuleSPICommunication::writeFrame (C++ function), 229
- ModuleSPICommunication::writeMemory (C++ function), 228
- ModuleSPICommunication::writePayloadData (C++ function), 229
- mono::AppRunLoop (C++ class), 137
- mono::AppRunLoop::DynamicTaskQueueTime (C++ member), 138
- mono::AppRunLoop::lastDtrValue (C++ member), 139
- mono::AppRunLoop::resetOnDTR (C++ member), 138
- mono::AppRunLoop::resetOnUserButton (C++ member), 139
- mono::AppRunLoop::taskQueueHead (C++ member), 139
- mono::AppRunLoop::TouchSystemTime (C++ member), 138
- mono::DateTime (C++ class), 120
- mono::DateTime::LOCAL_TIME_ZONE (C++ class), 120
- mono::DateTime::maxValue (C++ function), 126
- mono::DateTime::minValue (C++ function), 126
- mono::DateTime::TimeTypes (C++ type), 120
- mono::DateTime::UNKNOWN_TIME_ZONE (C++ class), 121
- mono::DateTime::UTC_TIME_ZONE (C++ class), 121
- mono::display::Color (C++ class), 194
- mono::display::DisplayPainter (C++ class), 196
- mono::display::DisplayPainter::displayRefreshHandler (C++ member), 200
- mono::display::IDisplayController (C++ class), 200
- mono::display::IDisplayController::Brightness (C++ function), 202
- mono::display::IDisplayController::IDisplayController (C++ function), 200
- mono::display::IDisplayController::init (C++ function), 201
- mono::display::IDisplayController::LastTearingEffectTime (C++ member), 202
- mono::display::IDisplayController::setBrightness (C++ function), 201
- mono::display::IDisplayController::setBrightnessPercent (C++ function), 202
- mono::display::IDisplayController::setCursor (C++ function), 201
- mono::display::IDisplayController::setRefreshHandler (C++ function), 201
- mono::display::IDisplayController::setWindow (C++ function), 201
- mono::display::IDisplayController::write (C++ function), 201
- mono::display::MonoIcon (C++ class), 206
- mono::display::MonoIcon::bitmap (C++ member), 207
- mono::display::MonoIcon::height (C++ member), 206
- mono::display::MonoIcon::width (C++ member), 206
- mono::display::TextRender (C++ class), 202
- mono::display::TextRender::ALIGN_BOTTOM (C++ class), 203
- mono::display::TextRender::ALIGN_CENTER (C++ class), 203

mono::display::TextRender::ALIGN_LEFT (C++ class), 202

mono::display::TextRender::ALIGN_MIDDLE (C++ class), 203

mono::display::TextRender::ALIGN_RIGHT (C++ class), 203

mono::display::TextRender::ALIGN_TOP (C++ class), 203

mono::display::TextRender::charDrawFunction (C++ type), 203

mono::display::TextRender::drawChar (C++ function), 204, 206

mono::display::TextRender::drawInRect (C++ function), 204

mono::display::TextRender::HorizontalAlignment (C++ type), 202

mono::display::TextRender::renderDimension (C++ function), 204, 205

mono::display::TextRender::VerticalAlignmentType (C++ type), 203

mono::GenericQueue (C++ class), 126

mono::geo::Circle (C++ class), 207

mono::geo::Point (C++ class), 207

mono::geo::Rect (C++ class), 207

mono::geo::Size (C++ class), 208

mono::IApplication (C++ class), 139

mono::IApplication::enterRunLoop (C++ function), 140

mono::IApplication::IApplication (C++ function), 140

mono::IApplication::monoWakeFromReset (C++ function), 140

mono::IApplication::monoWakeFromSleep (C++ function), 140

mono::IApplication::monoWillGotoSleep (C++ function), 140

mono::IApplicationContext (C++ class), 140

mono::IApplicationContext::_softwareReset (C++ function), 143

mono::IApplicationContext::_softwareResetToApplication (C++ function), 143

mono::IApplicationContext::_softwareResetToBootloader (C++ function), 143

mono::IApplicationContext::Accelerometer (C++ member), 142

mono::IApplicationContext::Buzzer (C++ member), 142

mono::IApplicationContext::DisplayController (C++ member), 141

mono::IApplicationContext::EnterSleepMode (C++ function), 142

mono::IApplicationContext::enterSleepMode (C++ function), 143

mono::IApplicationContext::exec (C++ function), 140

mono::IApplicationContext::IApplicationContext (C++ function), 143

mono::IApplicationContext::PowerManager (C++ member), 141

mono::IApplicationContext::ResetOnUserButton (C++ function), 142

mono::IApplicationContext::resetOnUserButton (C++ function), 143

mono::IApplicationContext::RTC (C++ member), 142

mono::IApplicationContext::RunLoop (C++ member), 141

mono::IApplicationContext::setMonoApplication (C++ function), 140

mono::IApplicationContext::SleepForMs (C++ function), 142

mono::IApplicationContext::sleepForMs (C++ function), 143

mono::IApplicationContext::SoftwareReset (C++ function), 142

mono::IApplicationContext::SoftwareResetToApplication (C++ function), 142

mono::IApplicationContext::SoftwareResetToBootloader (C++ function), 143

mono::IApplicationContext::Temperature (C++ member), 141

mono::IApplicationContext::TouchSystem (C++ member), 141

mono::IApplicationContext::UserButton (C++ member), 141

mono::io::DigitalOut (C++ class), 174

mono::io::File (C++ class), 174

mono::io::FileSystem (C++ class), 176

mono::io::FilteredAnalogIn (C++ class), 177

mono::io::FilteredAnalogIn::FilteredAnalogIn (C++ function), 177

mono::io::FilteredAnalogIn::read (C++ function), 177

mono::io::FilteredAnalogIn::read_u16 (C++ function), 178

mono::io::HysteresisTrigger (C++ class), 178

mono::io::HysteresisTrigger::check (C++ function), 179

mono::io::HysteresisTrigger::HysteresisTrigger (C++ function), 178

mono::io::HysteresisTrigger::NextTriggerType (C++ function), 179

mono::io::HysteresisTrigger::setNextTrigger (C++ function), 179

mono::io::IWifi (C++ class), 185

mono::io::IWifi::connect (C++ function), 186

mono::io::IWifi::CONNECT_ERROR (C++ class), 185

mono::io::IWifi::CONNECTED (C++ class), 185

mono::io::IWifi::DISCONNECTED (C++ class), 185

mono::io::IWifi::isConnected (C++ function), 186

mono::io::IWifi::NetworkEvents (C++ type), 185

mono::io::IWifi::setConnectedCallback (C++ function), 186

mono::io::IWifi::setConnectErrorCallback (C++ function), 186

- mono::io::IWifi::setDisconnectedCallback (C++ function), 187
- mono::io::IWifi::setEventCallback (C++ function), 187
- mono::io::OneWire (C++ class), 180
- mono::io::RunningAverageFilter (C++ class), 183
- mono::io::RunningAverageFilter::append (C++ function), 184
- mono::io::RunningAverageFilter::clear (C++ function), 184
- mono::io::RunningAverageFilter::length (C++ function), 184
- mono::io::RunningAverageFilter::operator[] (C++ function), 184
- mono::io::RunningAverageFilter::RunningAverageFilter (C++ function), 184
- mono::io::RunningAverageFilter::sum (C++ function), 184
- mono::io::RunningAverageFilter::value (C++ function), 184
- mono::io::RunningAverageFilter::variance (C++ function), 184
- mono::io::Serial (C++ class), 184
- mono::IQueueItem (C++ class), 129
- mono::IRTCSystem (C++ class), 144
- mono::IRTCSystem::setupRtcSystem (C++ function), 144
- mono::IRTCSystem::startRtc (C++ function), 144
- mono::IRTCSystem::stopRtc (C++ function), 144
- mono::IRunLoopTask (C++ class), 144
- mono::IRunLoopTask::nextTask (C++ member), 144
- mono::IRunLoopTask::previousTask (C++ member), 144
- mono::IRunLoopTask::singleShot (C++ member), 144
- mono::IRunLoopTask::taskHandler (C++ function), 144
- mono::ITouchSystem (C++ class), 144
- mono::ITouchSystem::activate (C++ function), 145
- mono::ITouchSystem::CurrentCalibration (C++ function), 145
- mono::ITouchSystem::deactivate (C++ function), 145
- mono::ITouchSystem::init (C++ function), 144
- mono::ITouchSystem::isActive (C++ function), 145
- mono::ITouchSystem::processTouchInput (C++ function), 145
- mono::ITouchSystem::runTouchBegin (C++ function), 145
- mono::ITouchSystem::runTouchEnd (C++ function), 145
- mono::ITouchSystem::runTouchMove (C++ function), 145
- mono::ITouchSystem::setCalibration (C++ function), 145
- mono::ITouchSystem::toScreenCoordsX (C++ function), 145
- mono::ITouchSystem::toScreenCoordsY (C++ function), 145
- mono::ManagedPointer (C++ class), 145
- mono::ManagedPointer::ManagedPointer (C++ function), 146
- mono::ManagedPointer::Surrender (C++ function), 146
- mono::network::DnsResolver (C++ class), 187
- mono::network::HttpClient (C++ class), 188
- mono::network::HttpClient::HttpResponseBodyData (C++ class), 189
- mono::network::HttpClient::HttpResponseBodyData::bodyChunk (C++ member), 189
- mono::network::HttpClient::HttpResponseBodyData::Context (C++ member), 189
- mono::network::HttpClient::HttpResponseBodyData::Finished (C++ member), 189
- mono::network::HttpClient::setDataReadyCallback (C++ function), 189
- mono::network::HttpPostClient (C++ class), 190
- mono::network::HttpPostClient::setBodyDataCallback (C++ function), 191
- mono::network::HttpPostClient::setBodyLengthCallback (C++ function), 190
- mono::network::INetworkRequest (C++ class), 191
- mono::network::INetworkRequest::cbTimer (C++ member), 194
- mono::network::INetworkRequest::completionHandler (C++ member), 194
- mono::network::INetworkRequest::errorHandler (C++ member), 194
- mono::network::INetworkRequest::INetworkRequest (C++ function), 193
- mono::network::INetworkRequest::lastErrorCode (C++ member), 194
- mono::network::INetworkRequest::setCompletionCallback (C++ function), 192
- mono::network::INetworkRequest::setErrorCallback (C++ function), 193
- mono::network::INetworkRequest::setStateChangeEventCallback (C++ function), 193
- mono::network::INetworkRequest::state (C++ member), 194
- mono::network::INetworkRequest::stateChangeHandler (C++ member), 194
- mono::power::IPowerAware (C++ class), 212
- mono::power::IPowerAware::_pwwrwr_nextPowerAware (C++ member), 212
- mono::power::IPowerAware::_pwwrwr_previousPowerAware (C++ member), 212
- mono::power::IPowerAware::onSystemBatteryLow (C++ function), 212
- mono::power::IPowerAware::onSystemEnterSleep (C++ function), 212
- mono::power::IPowerAware::onSystemPowerOnReset (C++ function), 212
- mono::power::IPowerAware::onSystemWakeFromSleep (C++ function), 212

mono::power::IPowerManagement (C++ class), 213
 mono::power::IPowerManagement::AppendToPowerAwareQueue (C++ function), 213
 mono::power::IPowerManagement::EnterSleep (C++ function), 213
 mono::power::IPowerManagement::powerAwareQueue (C++ member), 214
 mono::power::IPowerManagement::PowerSystem (C++ member), 213
 mono::power::IPowerManagement::processResetAwareQueue (C++ function), 214
 mono::power::IPowerManagement::RemoveFromPowerAwareQueue (C++ function), 213
 mono::power::IPowerSubSystem (C++ class), 214
 mono::power::IPowerSubSystem::BatteryEmptyHandler (C++ member), 216
 mono::power::IPowerSubSystem::BatteryLowHandler (C++ member), 216
 mono::power::IPowerSubSystem::CHARGE_ENDED (C++ class), 215
 mono::power::IPowerSubSystem::CHARGE_FAST (C++ class), 215
 mono::power::IPowerSubSystem::CHARGE_PRECONDITION (C++ class), 215
 mono::power::IPowerSubSystem::CHARGE_SLOW (C++ class), 215
 mono::power::IPowerSubSystem::CHARGE_SUSPENDED (C++ class), 215
 mono::power::IPowerSubSystem::ChargeState (C++ type), 215
 mono::power::IPowerSubSystem::ChargeStatus (C++ function), 215
 mono::power::IPowerSubSystem::IsPowerFenced (C++ function), 215
 mono::power::IPowerSubSystem::IsPowerOk (C++ function), 216
 mono::power::IPowerSubSystem::IsUSBCharging (C++ function), 215
 mono::power::IPowerSubSystem::onSystemEnterSleep (C++ function), 215
 mono::power::IPowerSubSystem::onSystemPowerOnReset (C++ function), 215
 mono::power::IPowerSubSystem::onSystemWakeFromSleep (C++ function), 215
 mono::power::IPowerSubSystem::setPowerFence (C++ function), 215
 mono::power::IPowerSubSystem::UNKNOWN (C++ class), 215
 mono::power::MonoBattery (C++ class), 211
 mono::PowerSaver (C++ class), 127
 mono::Queue (C++ class), 128
 mono::Queue::dequeue (C++ function), 128
 mono::Queue::enqueue (C++ function), 128
 mono::Queue::exists (C++ function), 129
 mono::Queue::next (C++ function), 128
 mono::Queue::peek (C++ function), 128
 mono::QueueInterrupt (C++ class), 181
 mono::QueueInterrupt::fall (C++ function), 182
 mono::QueueInterrupt::rise (C++ function), 181
 mono::redpine::ManagementFrame (C++ class), 216
 mono::redpine::ManagementFrame::autoReleaseWhenParsed (C++ member), 218
 mono::redpine::ManagementFrame::completionHandler (C++ member), 218
 mono::redpine::ManagementFrame::FrameCompletionData (C++ class), 219
 mono::redpine::ManagementFrame::FrameCompletionData::Context (C++ member), 219
 mono::redpine::ManagementFrame::FrameCompletionData::Success (C++ member), 219
 mono::redpine::ManagementFrame::handlerContextObject (C++ member), 218
 mono::redpine::ManagementFrame::lastResponseParsed (C++ member), 218
 mono::redpine::ManagementFrame::responsePayload (C++ member), 218
 mono::redpine::ManagementFrame::status (C++ member), 218
 mono::redpine::Module (C++ class), 219
 mono::redpine::Module::asyncManagementFrameHandler (C++ member), 220
 mono::redpine::Module::BootloaderMessageCodes (C++ type), 219
 mono::redpine::Module::BootloaderRegisters (C++ type), 220
 mono::redpine::Module::comIntf (C++ member), 222
 mono::redpine::Module::communicationInitialized (C++ member), 222
 mono::redpine::Module::CurrentPowerState (C++ member), 222
 mono::redpine::Module::HOST_INTERACT_REG_VALID (C++ class), 220
 mono::redpine::Module::HOST_INTF_REG_IN (C++ class), 220
 mono::redpine::Module::HOST_INTF_REG_OUT (C++ class), 220
 mono::redpine::Module::MODE_BLE (C++ class), 219
 mono::redpine::Module::MODE_BT_CLASSIC (C++ class), 219
 mono::redpine::Module::MODE_WIFI_BLE (C++ class), 219
 mono::redpine::Module::MODE_WIFI_BT_CLASSIC (C++ class), 219
 mono::redpine::Module::MODE_WIFI_ONLY (C++ class), 219
 mono::redpine::Module::ModuleCoExistenceModes (C++ type), 219
 mono::redpine::Module::networkInitialized (C++ mem-

- ber), 222
- mono::redpine::Module::networkReadyHandler (C++ member), 222
- mono::redpine::Module::OperatingMode (C++ member), 222
- mono::redpine::Module::requestFrameQueue (C++ member), 222
- mono::redpine::Module::responseFrameQueue (C++ member), 222
- mono::redpine::Module::RSI_ENABLE_BOOT_BYPASS (C++ class), 220
- mono::redpine::Module::RSI_LOAD_IMAGE_I_FW (C++ class), 220
- mono::redpine::Module::ScannedNetworks (C++ class), 222
- mono::redpine::Module::ScannedNetworks::Count (C++ member), 222
- mono::redpine::Module::ScannedNetworks::networks (C++ member), 222
- mono::redpine::Module::SEC_ENTERPRISE_WPA (C++ class), 219
- mono::redpine::Module::SEC_ENTERPRISE_WPA2 (C++ class), 219
- mono::redpine::Module::SEC_OPEN_NETWORK (C++ class), 219
- mono::redpine::Module::SEC_WEP (C++ class), 219
- mono::redpine::Module::SEC_WPA (C++ class), 219
- mono::redpine::Module::SEC_WPA2 (C++ class), 219
- mono::redpine::Module::SEC_WPA_WPA2 (C++ class), 219
- mono::redpine::Module::StaticIPParams (C++ class), 222
- mono::redpine::Module::StaticIPParams::gateway (C++ member), 222
- mono::redpine::Module::StaticIPParams::ipAddress (C++ member), 222
- mono::redpine::Module::StaticIPParams::netmask (C++ member), 222
- mono::redpine::Module::WifiSecurityModes (C++ type), 219
- mono::redpine::ModuleCommunication (C++ class), 223
- mono::redpine::ModuleCommunication::DataPayload (C++ class), 225
- mono::redpine::ModuleCommunication::initializeInterface (C++ function), 223
- mono::redpine::ModuleCommunication::InterfaceVersion (C++ member), 225
- mono::redpine::ModuleCommunication::interruptActive (C++ function), 223
- mono::redpine::ModuleCommunication::interruptCallback (C++ member), 225
- mono::redpine::ModuleCommunication::pollInputQueue (C++ function), 223
- mono::redpine::ModuleCommunication::readDataFrame (C++ function), 224
- mono::redpine::ModuleCommunication::readFrame (C++ function), 223
- mono::redpine::ModuleCommunication::readManagementFrame (C++ function), 223
- mono::redpine::ModuleCommunication::readManagementFrameResponse (C++ function), 223
- mono::redpine::ModuleCommunication::readMemory (C++ function), 224
- mono::redpine::ModuleCommunication::resetModule (C++ function), 223
- mono::redpine::ModuleCommunication::writeFrame (C++ function), 224
- mono::redpine::ModuleCommunication::writeMemory (C++ function), 224
- mono::redpine::ModuleCommunication::writePayloadData (C++ function), 225
- mono::redpine::ModuleFrame (C++ class), 225
- mono::redpine::ModuleFrame::AntennaSelect (C++ class), 226
- mono::redpine::ModuleFrame::APConfig (C++ class), 226
- mono::redpine::ModuleFrame::AsyncConnAcceptReq (C++ class), 226
- mono::redpine::ModuleFrame::AsyncSckTerminated (C++ class), 226
- mono::redpine::ModuleFrame::AsyncTcpConnect (C++ class), 226
- mono::redpine::ModuleFrame::Band (C++ class), 226
- mono::redpine::ModuleFrame::CardReady (C++ class), 226
- mono::redpine::ModuleFrame::commandId (C++ member), 227
- mono::redpine::ModuleFrame::ConfigEnable (C++ class), 226
- mono::redpine::ModuleFrame::ConfigGet (C++ class), 226
- mono::redpine::ModuleFrame::ConfigSave (C++ class), 226
- mono::redpine::ModuleFrame::DebugPrintUART2 (C++ class), 226
- mono::redpine::ModuleFrame::direction (C++ member), 227
- mono::redpine::ModuleFrame::Disconnect (C++ class), 226
- mono::redpine::ModuleFrame::DnsResolution (C++ class), 226
- mono::redpine::ModuleFrame::FrameDirection (C++ type), 225
- mono::redpine::ModuleFrame::HttpGet (C++ class), 226
- mono::redpine::ModuleFrame::HttpPost (C++ class), 226
- mono::redpine::ModuleFrame::Init (C++ class), 226
- mono::redpine::ModuleFrame::Join (C++ class), 226
- mono::redpine::ModuleFrame::length (C++ member), 227

mono::redpine::ModuleFrame::MulticastAddrFilter (C++ class), 226

mono::redpine::ModuleFrame::PingCommand (C++ class), 226

mono::redpine::ModuleFrame::PowerSaveACK (C++ class), 226

mono::redpine::ModuleFrame::PowerSaveMode (C++ class), 226

mono::redpine::ModuleFrame::QueryFirmware (C++ class), 226

mono::redpine::ModuleFrame::QueryNetworkParams (C++ class), 226

mono::redpine::ModuleFrame::RSSIQuery (C++ class), 226

mono::redpine::ModuleFrame::RX_FRAME (C++ class), 225

mono::redpine::ModuleFrame::RxCommandIds (C++ type), 226

mono::redpine::ModuleFrame::RxTxCommandIds (C++ type), 225

mono::redpine::ModuleFrame::Scan (C++ class), 226

mono::redpine::ModuleFrame::SendData (C++ class), 225

mono::redpine::ModuleFrame::SetIPParameters (C++ class), 226

mono::redpine::ModuleFrame::SetMacAddress (C++ class), 226

mono::redpine::ModuleFrame::SetOperatingMode (C++ class), 226

mono::redpine::ModuleFrame::SetRegion (C++ class), 226

mono::redpine::ModuleFrame::SetWEPKeys (C++ class), 226

mono::redpine::ModuleFrame::size (C++ member), 227

mono::redpine::ModuleFrame::SleepTimer (C++ class), 226

mono::redpine::ModuleFrame::SocketClose (C++ class), 226

mono::redpine::ModuleFrame::SocketCreate (C++ class), 226

mono::redpine::ModuleFrame::SoftReset (C++ class), 226

mono::redpine::ModuleFrame::TX_FRAME (C++ class), 225

mono::redpine::ModuleFrame::UserStoreConfig (C++ class), 226

mono::redpine::ModuleFrame::WakeFromSleep (C++ class), 226

mono::redpine::ModuleSPICommunication (C++ class), 227

mono::redpine::ModuleSPICommunication::SPI_HOST_IN (C++ class), 227

mono::redpine::ModuleSPICommunication::SpiRegisters (C++ type), 227

mono::Regex (C++ class), 129

mono::Regex::Capture (C++ type), 130

mono::ScheduledTask (C++ class), 132

mono::sensor::IAccelerometer (C++ class), 208

mono::sensor::IAccelerometer::IsActive (C++ function), 209

mono::sensor::IAccelerometer::rawXAxis (C++ function), 209

mono::sensor::IAccelerometer::rawYAxis (C++ function), 209

mono::sensor::IAccelerometer::rawZAxis (C++ function), 209

mono::sensor::IAccelerometer::Start (C++ function), 208

mono::sensor::IAccelerometer::Stop (C++ function), 209

mono::sensor::IBuzzer (C++ class), 209

mono::sensor::IBuzzer::buzzAsync (C++ function), 209, 210

mono::sensor::IBuzzer::buzzKill (C++ function), 209

mono::sensor::IBuzzer::timeoutHandler (C++ member), 210

mono::sensor::ITemperature (C++ class), 210

mono::sensor::ITemperature::Read (C++ function), 210

mono::sensor::ITemperature::ReadMilliCelcius (C++ function), 211

mono::String (C++ class), 131

mono::Timer (C++ class), 135

mono::Timer::callOnce (C++ function), 136

mono::Timer::setCallback (C++ function), 136

mono::Timer::start (C++ function), 135

mono::Timer::stop (C++ function), 135

mono::TouchEvent (C++ class), 146

mono::TouchResponder (C++ class), 146

mono::TouchResponder::activate (C++ function), 146

mono::TouchResponder::deactivate (C++ function), 146

mono::ui::AbstractButtonView (C++ class), 147

mono::ui::AbstractButtonView::setClickCallback (C++ function), 147

mono::ui::BackgroundView (C++ class), 147

mono::ui::ButtonView (C++ class), 149

mono::ui::ButtonView::setClickCallback (C++ function), 151

mono::ui::ButtonView::setFont (C++ function), 150

mono::ui::ConsoleView (C++ class), 151

mono::ui::ConsoleView::consoleLines (C++ function), 152

mono::ui::ConsoleView::ConsoleView (C++ function), 151

mono::ui::ConsoleView::lineLength (C++ function), 152

mono::ui::ConsoleView::repaint (C++ function), 152

mono::ui::ConsoleView::scrolls (C++ member), 152

mono::ui::ConsoleView::textBuffer (C++ member), 152

mono::ui::ConsoleView::WriteLine (C++ function), 151

mono::ui::GraphView (C++ class), 152

mono::ui::IconView (C++ class), 155

- mono::ui::IGraphViewDataSource (C++ class), 154
 - mono::ui::IGraphViewDataSource::BufferLength (C++ function), 155
 - mono::ui::IGraphViewDataSource::DataPoint (C++ function), 155
 - mono::ui::IGraphViewDataSource::MaxSampleValueSpan (C++ function), 155
 - mono::ui::IGraphViewDataSource::NewestSampleIndex (C++ function), 155
 - mono::ui::ImageView (C++ class), 157
 - mono::ui::ImageView::crop (C++ member), 158
 - mono::ui::ImageView::image (C++ member), 158
 - mono::ui::OnOffButtonView (C++ class), 159
 - mono::ui::ProgressBarView (C++ class), 159
 - mono::ui::ResponderView (C++ class), 161
 - mono::ui::SceneController (C++ class), 161
 - mono::ui::StatusIndicatorView (C++ class), 163
 - mono::ui::TextLabelView (C++ class), 165
 - mono::ui::TextLabelView::ALIGN_BOTTOM (C++ class), 166
 - mono::ui::TextLabelView::ALIGN_CENTER (C++ class), 165
 - mono::ui::TextLabelView::ALIGN_LEFT (C++ class), 165
 - mono::ui::TextLabelView::ALIGN_MIDDLE (C++ class), 166
 - mono::ui::TextLabelView::ALIGN_RIGHT (C++ class), 165
 - mono::ui::TextLabelView::ALIGN_TOP (C++ class), 165
 - mono::ui::TextLabelView::GfxFont (C++ function), 167
 - mono::ui::TextLabelView::incrementalRepaint (C++ member), 168
 - mono::ui::TextLabelView::scheduleFastRepaint (C++ function), 168
 - mono::ui::TextLabelView::setBackground (C++ function), 167
 - mono::ui::TextLabelView::setFont (C++ function), 168
 - mono::ui::TextLabelView::setText (C++ function), 167, 168
 - mono::ui::TextLabelView::StandardGfxFont (C++ member), 169
 - mono::ui::TextLabelView::TextAlignment (C++ type), 165
 - mono::ui::TextLabelView::TextLabelView (C++ function), 166
 - mono::ui::TextLabelView::VerticalTextAlignment (C++ type), 165
 - mono::ui::TouchCalibrateView (C++ class), 169
 - mono::ui::View (C++ class), 170
 - mono::ui::View::isDirty (C++ member), 173
 - mono::ui::View::LANDSCAPE_LEFT (C++ class), 170
 - mono::ui::View::LANDSCAPE_RIGHT (C++ class), 170
 - mono::ui::View::Orientation (C++ type), 170
 - mono::ui::View::PORTRAIT (C++ class), 170
 - mono::ui::View::PORTRAIT_BOTTOMUP (C++ class), 170
 - mono::ui::View::repaint (C++ function), 172
 - mono::ui::View::viewRect (C++ member), 173
 - mono::ui::View::visible (C++ member), 173
 - MonoBattery::CalculatePercentage (C++ function), 211
 - MonoBattery::ReadMilliVolts (C++ function), 211
 - MonoBattery::ReadPercentage (C++ function), 211
- ## O
- OneWire::OneWire (C++ function), 180
 - OnOffButtonView::repaint (C++ function), 159
- ## P
- PowerSaver::deactivate (C++ function), 128
 - PowerSaver::dim (C++ function), 128
 - PowerSaver::PowerSaver (C++ function), 127
 - PowerSaver::startDimTimer (C++ function), 127
 - PowerSaver::startSleepTimer (C++ function), 128
 - PowerSaver::undim (C++ function), 128
 - ProgressBarView::init (C++ function), 160
 - ProgressBarView::Maximum (C++ function), 160
 - ProgressBarView::Minimum (C++ function), 160
 - ProgressBarView::ProgressBarView (C++ function), 159, 160
 - ProgressBarView::repaint (C++ function), 160
 - ProgressBarView::setMaximum (C++ function), 160
 - ProgressBarView::setMinimum (C++ function), 160
 - ProgressBarView::setValue (C++ function), 160
- ## Q
- Queue::Length (C++ function), 129
 - Queue::Queue (C++ function), 128
 - QueueInterrupt::DeactivateUntilHandled (C++ function), 181
 - QueueInterrupt::FallTimeStamp (C++ function), 182
 - QueueInterrupt::IsInterruptsWhilePendingActive (C++ function), 181
 - QueueInterrupt::QueueInterrupt (C++ function), 181
 - QueueInterrupt::RiseTimeStamp (C++ function), 182
 - QueueInterrupt::setDebounceTimeout (C++ function), 181
 - QueueInterrupt::setDebouncing (C++ function), 181
 - QueueInterrupt::setInterruptsSleep (C++ function), 183
 - QueueInterrupt::Snapshot (C++ function), 182
 - QueueInterrupt::taskHandler (C++ function), 183
 - QueueInterrupt::willInterruptSleep (C++ function), 182
- ## R
- Rect::Center (C++ function), 208
 - Rect::contains (C++ function), 208
 - Rect::crop (C++ function), 208

[Rect::LowerLeft \(C++ function\), 208](#)
[Rect::LowerRight \(C++ function\), 207](#)
[Rect::Rect \(C++ function\), 207](#)
[Rect::setPoint \(C++ function\), 208](#)
[Rect::setSize \(C++ function\), 208](#)
[Rect::ToString \(C++ function\), 208](#)
[Rect::UpperLeft \(C++ function\), 207](#)
[Rect::UpperRight \(C++ function\), 207](#)
[Regex::IsMatch \(C++ function\), 130](#)
[Regex::Match \(C++ function\), 130](#)
[Regex::Regex \(C++ function\), 130](#)
[Regex::Value \(C++ function\), 130](#)
[ResponderView::hide \(C++ function\), 161](#)
[ResponderView::respondTouchBegin \(C++ function\), 161](#)
[ResponderView::respondTouchEnd \(C++ function\), 161](#)
[ResponderView::respondTouchMove \(C++ function\), 161](#)
[ResponderView::show \(C++ function\), 161](#)

S

[SceneController::addView \(C++ function\), 161](#)
[SceneController::BackgroundColor \(C++ function\), 162](#)
[SceneController::hide \(C++ function\), 162](#)
[SceneController::Position \(C++ function\), 162](#)
[SceneController::removeView \(C++ function\), 162](#)
[SceneController::requestDismiss \(C++ function\), 162](#)
[SceneController::SceneController \(C++ function\), 161](#)
[SceneController::scheduleRepaint \(C++ function\), 162](#)
[SceneController::setBackground \(C++ function\), 162](#)
[SceneController::setRect \(C++ function\), 162](#)
[SceneController::show \(C++ function\), 162](#)
[SceneController::Size \(C++ function\), 162](#)
[SceneController::ViewRect \(C++ function\), 162](#)
[SceneController::Visible \(C++ function\), 162](#)
[ScheduledTask::isDue \(C++ function\), 134](#)
[ScheduledTask::pendingScheduledTasks \(C++ function\), 134](#)
[ScheduledTask::processScheduledTasks \(C++ function\), 134](#)
[ScheduledTask::queue \(C++ member\), 134](#)
[ScheduledTask::reschedule \(C++ function\), 133](#)
[ScheduledTask::runTask \(C++ function\), 134](#)
[ScheduledTask::ScheduledTask \(C++ function\), 133](#)
[ScheduledTask::setRunInSleep \(C++ function\), 133](#)
[ScheduledTask::willRunInSleep \(C++ function\), 133](#)
[Serial::DTR \(C++ function\), 185](#)
[Serial::IsReady \(C++ function\), 185](#)
[Serial::Serial \(C++ function\), 184](#)
[Size::Size \(C++ function\), 208](#)
[StatusIndicatorView::initializer \(C++ function\), 164](#)
[StatusIndicatorView::repaint \(C++ function\), 164](#)
[StatusIndicatorView::setOffStateColor \(C++ function\), 164](#)

[StatusIndicatorView::setOnStateColor \(C++ function\), 164](#)
[StatusIndicatorView::setState \(C++ function\), 164](#)
[StatusIndicatorView::State \(C++ function\), 164](#)
[StatusIndicatorView::StatusIndicatorView \(C++ function\), 163](#)
[String::CString \(C++ function\), 132](#)
[String::Format \(C++ function\), 132](#)
[String::Length \(C++ function\), 132](#)
[String::operator\(\) \(C++ function\), 132](#)
[String::operator\[\] \(C++ function\), 132](#)
[String::String \(C++ function\), 131, 132](#)

T

[TextLabelView::Alignment \(C++ function\), 166](#)
[TextLabelView::Font \(C++ function\), 167](#)
[TextLabelView::isTextMultiline \(C++ function\), 169](#)
[TextLabelView::repaint \(C++ function\), 168](#)
[TextLabelView::scheduleRepaint \(C++ function\), 168](#)
[TextLabelView::setAlignment \(C++ function\), 167](#)
[TextLabelView::setBackground \(C++ member\), 168](#)
[TextLabelView::setBackgroundColor \(C++ function\), 167](#)
[TextLabelView::setTextColor \(C++ function\), 167](#)
[TextLabelView::setTextSize \(C++ function\), 167](#)
[TextLabelView::Text \(C++ function\), 168](#)
[TextLabelView::TextColor \(C++ function\), 166](#)
[TextLabelView::TextDimension \(C++ function\), 167](#)
[TextLabelView::TextLabelView \(C++ function\), 166](#)
[TextLabelView::TextPixelHeight \(C++ function\), 167](#)
[TextLabelView::TextPixelWidth \(C++ function\), 167](#)
[TextLabelView::TextSize \(C++ function\), 166](#)
[TextLabelView::VerticalAlignment \(C++ function\), 167](#)
[TextRender::Alignment \(C++ function\), 205](#)
[TextRender::Background \(C++ function\), 205](#)
[TextRender::calcUnderBaseline \(C++ function\), 206](#)
[TextRender::Foreground \(C++ function\), 205](#)
[TextRender::remainingTextlineWidth \(C++ function\), 206](#)
[TextRender::renderInRect \(C++ function\), 205](#)
[TextRender::setAlignment \(C++ function\), 205](#)
[TextRender::setBackground \(C++ function\), 205](#)
[TextRender::setForeground \(C++ function\), 205](#)
[TextRender::TextRender \(C++ function\), 203](#)
[TextRender::VerticalAlignment \(C++ function\), 205](#)
[TextRender::writePixel \(C++ function\), 206](#)
[Timer::Running \(C++ function\), 136](#)
[Timer::setInterval \(C++ function\), 136](#)
[Timer::SingleShot \(C++ function\), 135](#)
[Timer::Start \(C++ function\), 135](#)
[Timer::Stop \(C++ function\), 135](#)
[Timer::taskHandler \(C++ function\), 137](#)
[Timer::Timer \(C++ function\), 135](#)
[TouchCalibrateView::repaint \(C++ function\), 170](#)

TouchCalibrateView::respondTouchBegin (C++ function), [170](#)
TouchCalibrateView::show (C++ function), [170](#)
TouchCalibrateView::StartNewCalibration (C++ function), [169](#)
TouchCalibrateView::TouchCalibrateView (C++ function), [169](#)
TouchResponder::FirstResponder (C++ function), [147](#)
TouchResponder::TouchResponder (C++ function), [146](#)

V

View::callRepaintScheduledViews (C++ function), [172](#)
View::dirtyQueue (C++ member), [173](#)
View::DisplayHeight (C++ function), [172](#)
View::DisplayOrientation (C++ function), [172](#)
View::DisplayWidth (C++ function), [172](#)
View::hide (C++ function), [172](#)
View::painter (C++ member), [173](#)
View::Position (C++ function), [171](#)
View::repaintScheduledViews (C++ function), [173](#)
View::RepaintScheduledViewsTime (C++ member), [172](#)
View::scheduleRepaint (C++ function), [171](#)
View::setPosition (C++ function), [171](#)
View::setRect (C++ function), [171](#)
View::setSize (C++ function), [171](#)
View::show (C++ function), [171](#)
View::Size (C++ function), [171](#)
View::View (C++ function), [170](#)
View::ViewRect (C++ function), [171](#)
View::Visible (C++ function), [171](#)