Mono Developer Documentation Documentation

Release alpha

Monolit ApS

July 15, 2016

Contents

Hi, and welcome to our developer documentation site. We imagine you want to create your own mono applications? Well, you can skip all the talking and just start off with: Getting started installing the framework, and you are already half way.

CHAPTER 1

What to find here?

Here on our developer site, we will 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.

Right now we are in the process of writing the documentation, so you might encounter broken links, typos, bad gramma and a lot of misspellings.

We prioritize to get as much text published fast, instead of keeping our cards close. We hope you like our desicion. Anyway - should you encounter anything you would like to correct - see: *Contributing*

CHAPTER 2

Content

2.1 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.

2.1.1 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%
```

2.1.2 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 expexct you have installed the Mono tool chain, as decribed 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 is a class and inheritance!" But read on anyways, but I will recommend to read our Object oriented guide to C++.

Create a new project

Mono comes with a tool called `monomake`, that does once - 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 sill 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 at 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 interresting 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 contructor for our `AppController` class, to allow us to do custom initialization. You will see later why.
6. This is a required overwrite fro 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 contructor 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 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.

2.1.3 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 pixel 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 in 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.

2.1.4 Mono for Arduino Hackers

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

Caution: Our Arduino Board package extension is a pre-release. There are still issue with the Serial port and uploading sketches. Some API still needs implementation. If you have issues, please post on community.openmono.com.

Prerequisites

First I expect 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 independed. 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

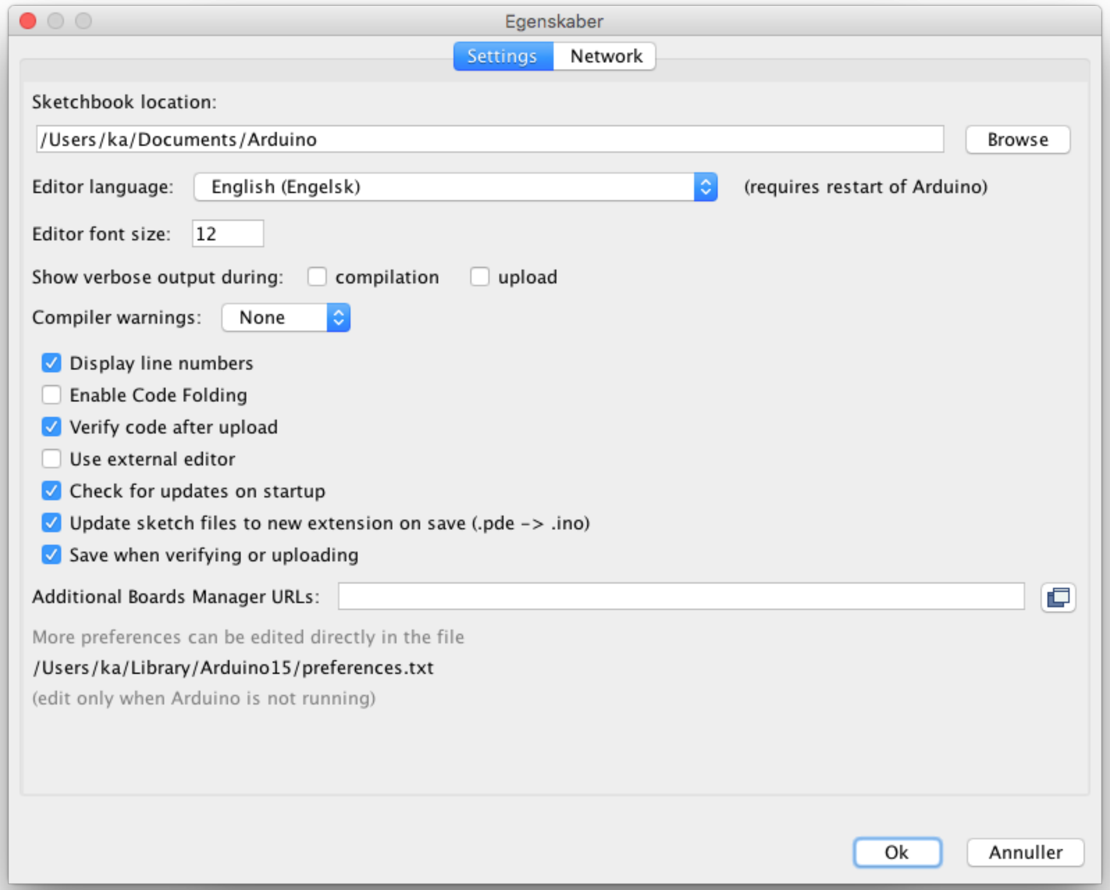
The Arduino IDE has a plugin system, where you can add support for third-party boards. We use such a plugin, that adds Mono as a target board. To install the plugin we use the *Board Manager*, that can install new target boards.

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 Ardiono IDE. Below is a screenshot of the window:



Sketchbook location: /Users/ka/Documents/Arduino

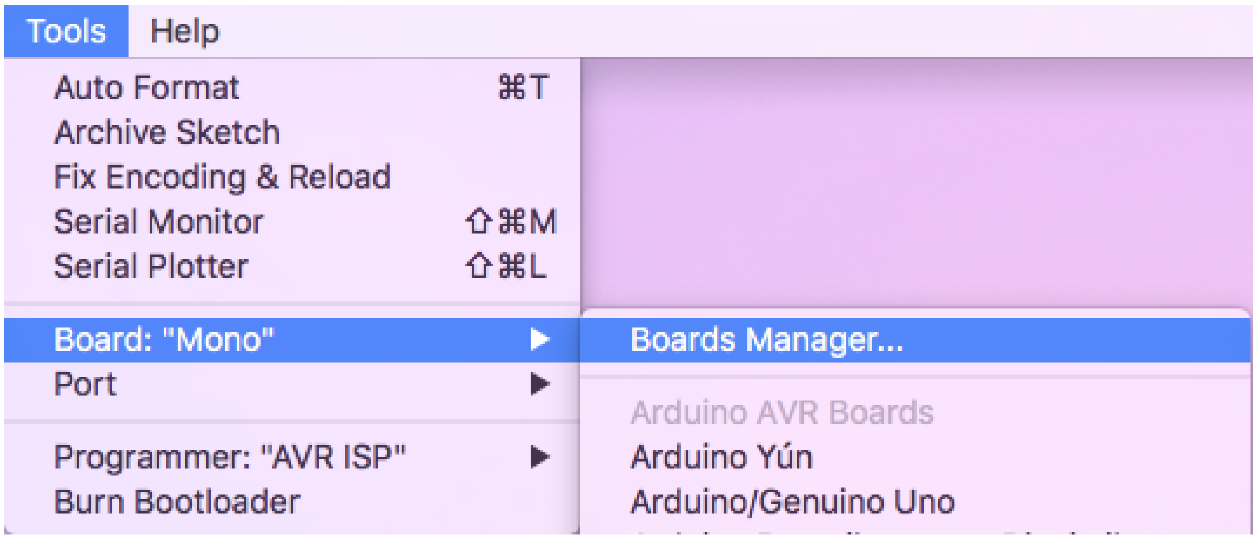
Editor language: English (Engelsk) (requires restart of Arduino)
Editor font size: 12
Show verbose output during: ☐ compilation ☐ upload
Compiler warnings: None
☑ Display line numbers
☐ Enable Code Folding
☑ Verify code after upload
☐ Use external editor
☑ Check for updates on startup
☑ Update sketch files to new extension on save (.pde -> .ino)
☑ Save when verifying or uploading
Additional Boards Manager URLs:
More preferences can be edited directly in the file
/Users/ka/Library/Arduino15/preferences.txt
(edit only when Arduino is not running)

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

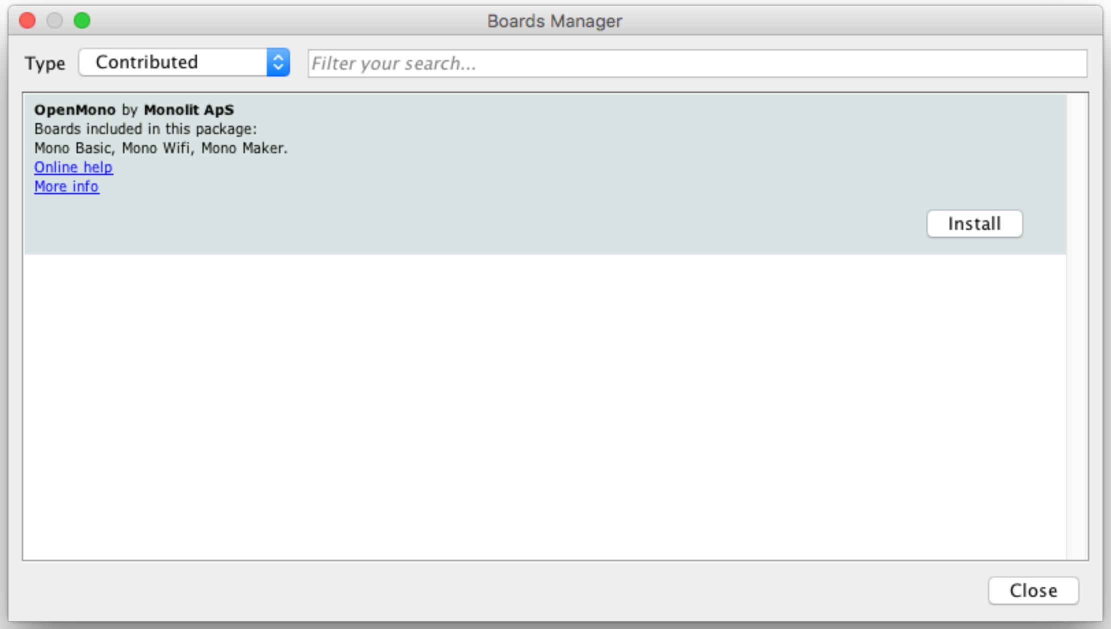
```
https://github.com/getopenmono/arduino_comp/releases/download/current/package_openmono_index.json
```

And press *OK*.

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



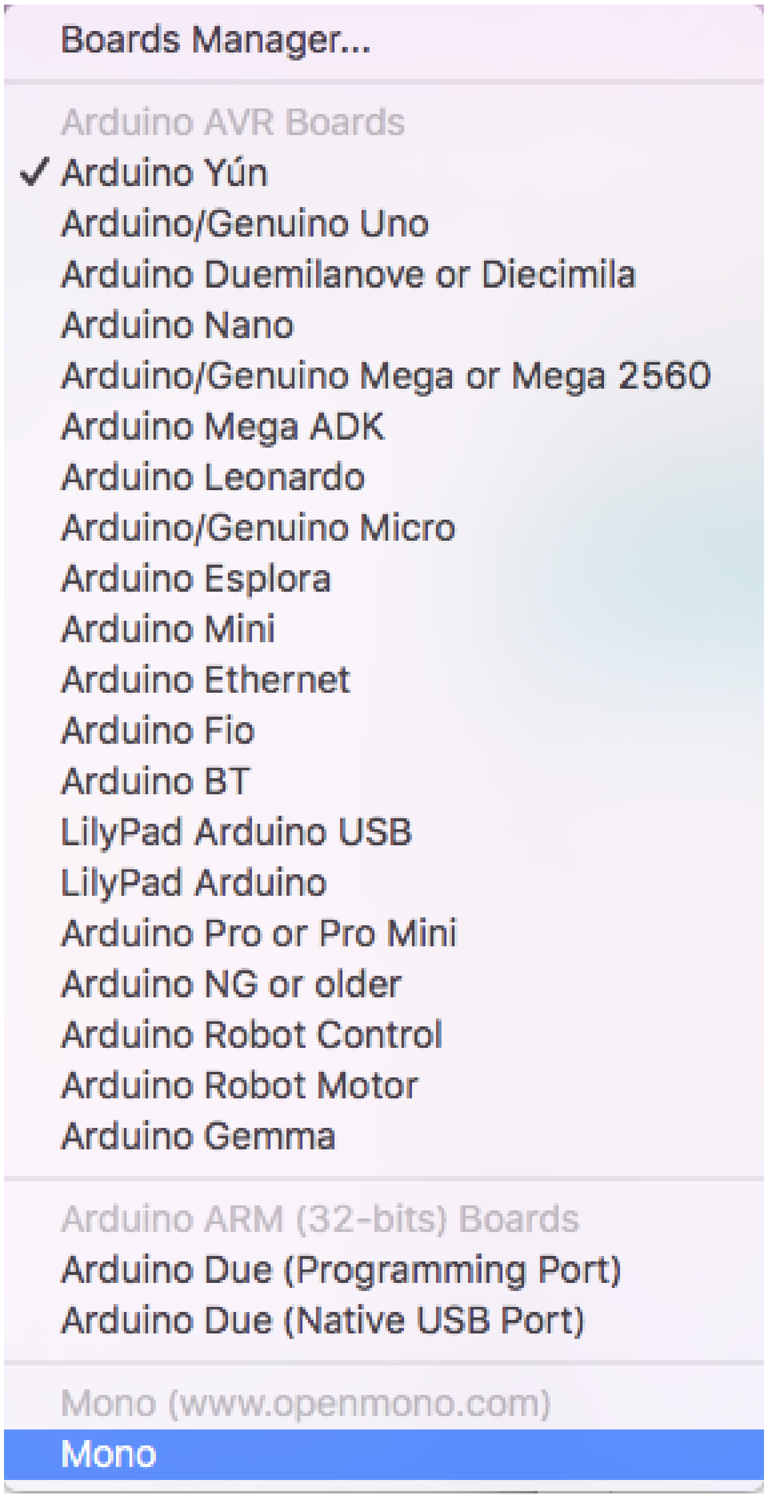
The *Board Manager* appears, and query the source URLs for new board types. It will discover a new board type OpenMono. Select the type *Contributed*, in top left corner:



OpenMono by **Monolit ApS**
Boards included in this package:
Mono Basic, Mono Wifi, Mono Maker.
Online help
More info

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*:

raw.githubusercontent.com/getopenmono/mono_psoc5_library/master/Generated_Source/PSoC5/USBUART_cdc.inf

When you connect Mono the *New hardware found...* wizard should appear, and you must direct this wiazrd to search for a driver in the directory where you downloaded the `.inf` file.

Note: Windows will yell at you because the driver is not signed. Just hold your breath and choose *continue anyway*, to install the driver. We promise it is legit :)

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. We dim the TFT LED backlight, so it will use minimal power.

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 displat 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 have 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 than create an object of the class in te 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(mono::geo::Rect(0,20,176,20),"Hi, I'm Mono"); // 2

    void setup() {
        // put your setup code here, to run once:
        textLbl.setTextColor(mono::display::WhiteColor);    // 3
        textLbl.show();    // 4
    }
```

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.
    - In *TextLabelView*'s contructor we create a rectangle object (Rect), and give the position $(0, 50)$ and dimension $(176, 20)$.
    - In the constructors second parameters we set the text content on the *TextLabel*. This is the text that will be displayed on the screen.
3. Because the screen on the Arduino template app is black, we need to tell the label to use a *White* text color.
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 compiles. If you have Mono connected you can upload the application by pressing the button.

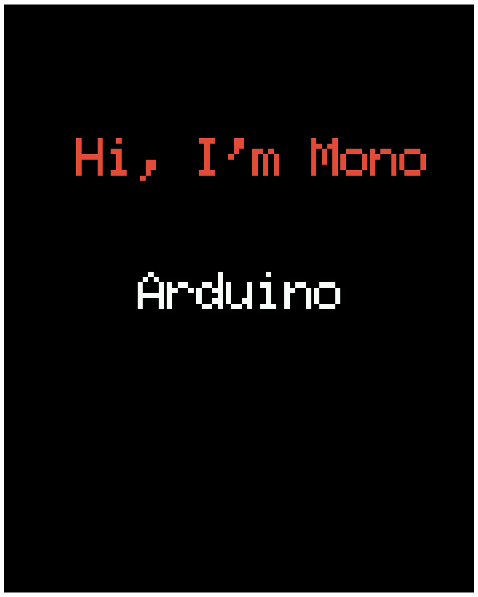
Notice that we did not need to put any code inside the `loop()` function.

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 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:



A quick note on namespaces

If find yourself wondering about what this is: `mono::ui::`, then read on. Everybody else - you can skip this section.

C++ uses namespaces to encapsulate class names, minimizing the risk of conflicting names. If you for example define a class called `Stream`, you like likely get a compiler error. This is because Arduino already have a class called `Stream` - the name is already taken.

To avoid this situation we defined all Mono classes inside a *namespace*, meaning that we exists inside a enclosed context:

```
namespace mono {
    class String;
    class Stream;
    class WhatEver;
}
```

You access classes inside a namespace by prepending the namespace to the class name like this: `mono::String`. Namespaces can be nested like this:

```
namespace mono {
    namespace display {
        class Color;
    }

    namespace ui {
        class TextLabelView;
    }
}
```

Importing into global context

You can import a namespace into the global context, to avoid prepending all mono classes with `mono::`, by the `using` keyword:

```
using namespace mono;
```

Now, instead of accessing classe by `mono::String`, you just write `String`, the `mono::` has become implcit. You can import multiple namespaces into the global context by:

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

Now you can implicit access classes from 3 different namespaces!

Note: The classes are not imported or loaded into the global context (like in Python), C++ does not work that way. It is just a short hand convenience feature, to make the code less verbose.

Further reading

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

- Install the native framework : Install the native mono build system
- Architectural Overview : About application lifecycles and structure.

Happy hacking.

2.1.5 Common Misconceptions & Worst Practice

To clear out, what we imagine will be common mistakes, let's go through some senarios 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 covenient 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.
```

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 periphrals 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 exerienced 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.

2.1.6 Monokiosk

Using Monokiosk

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

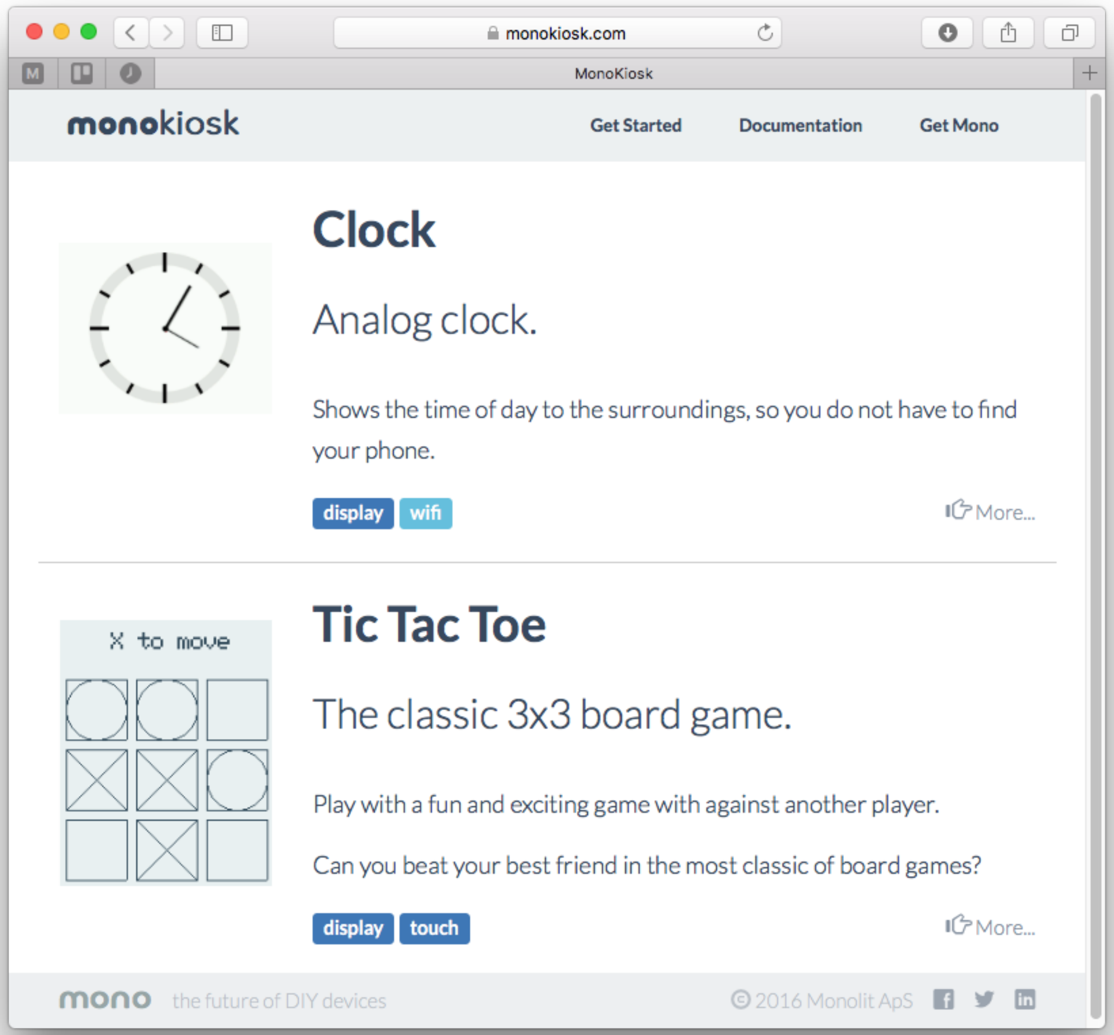
In this guide we will show you how to download and install a pre-built application on Mono. Monokiosk is the *app store* where you can browse and download mono applications, built by makers from around the world.

Note: In this early phase the number of applications on monokiosk is quite limited. We plan to add more applications to the kiosk ourselves, and hope that our community will submit their own creations to the site.

But until then, let us focus on the few apps we have in the kiosk right now. In this guide we shall install the *Tic Tac Toe* sample app.

Visit monokiosk.com

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



Clock

Analog clock.

Shows the time of day to the surroundings, so you do not have to find your phone.

Tic Tac Toe

The classic 3x3 board game.

Play with a fun and exciting game with against another player.

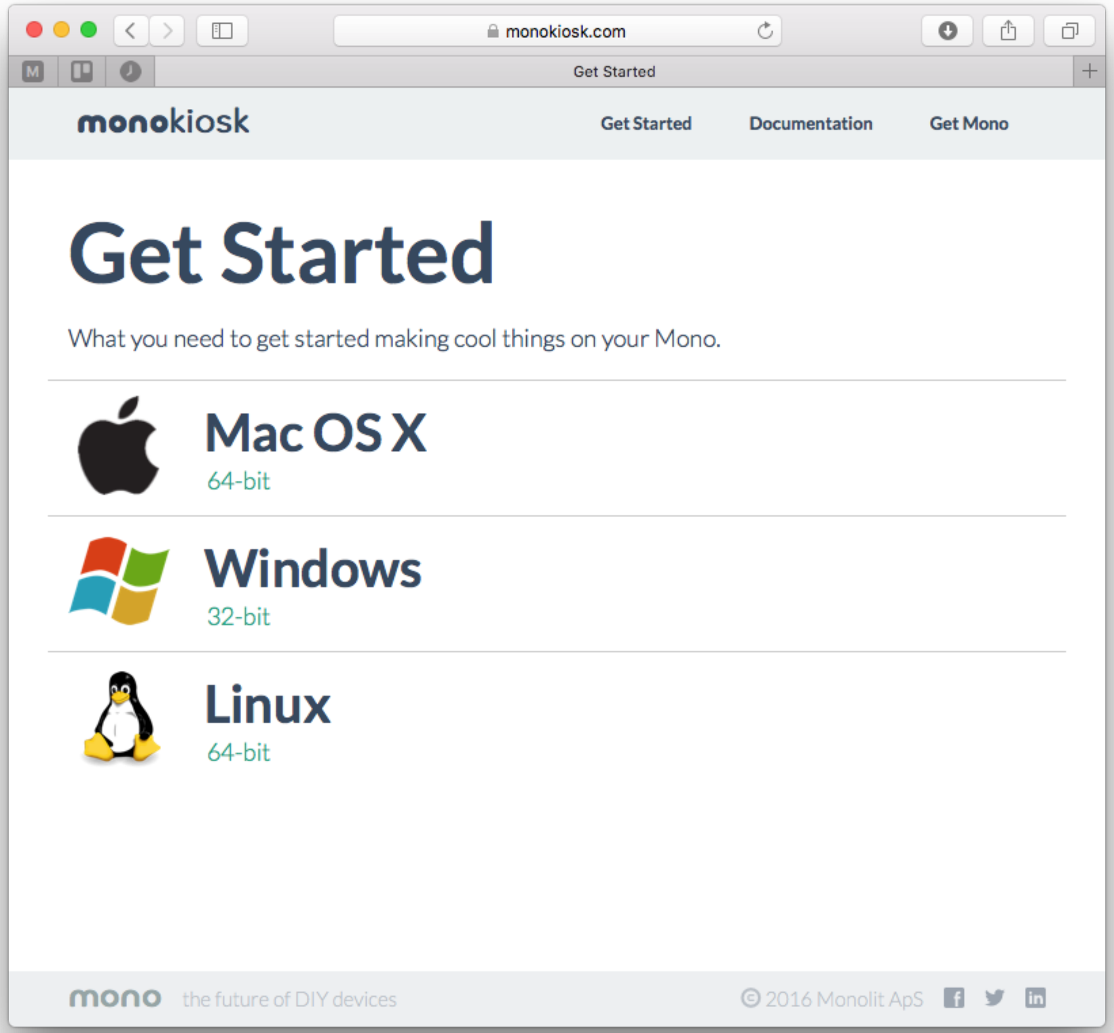
Can you beat your best friend in the most classic of board games?

Currently there are a simple *Clock* app and and a *Tic Tac Toe* app. But before you franticly click on one of these crazy mind blowing apps, you first need to install a tool called **monoprog**.

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

1. Download the application from monokiosk. (An application is a file with the extension: *.elf*)
2. Use *monoprog* to transfer the downloaded *.elf* file to Mono.
3. There is no step 3!

First you must download monoprog itself, so click on *Get Started* in the menu bar. Nowyou will see this page:



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

Windows 8 and 10 Users:

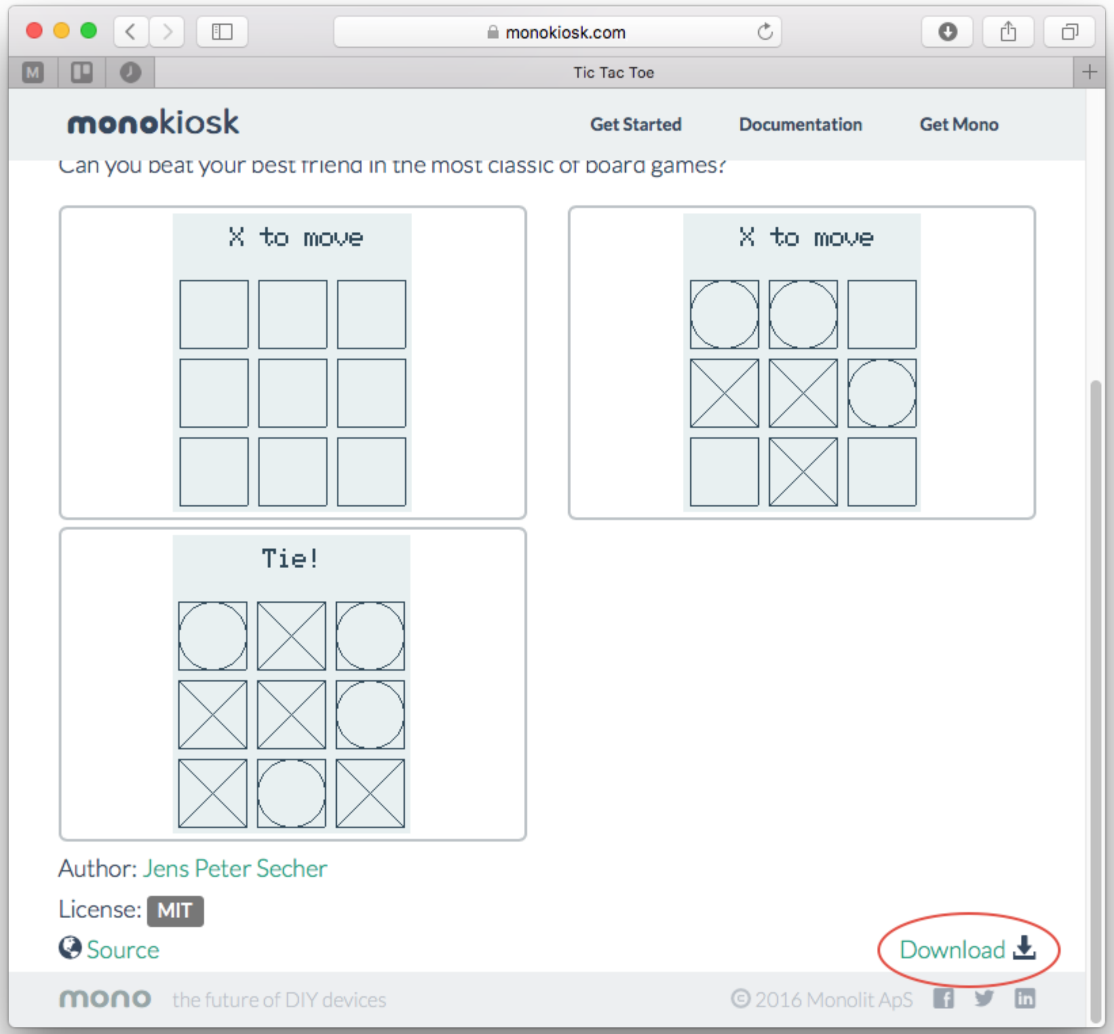
We are working on a signed driver package to be a part of the installer, but until then Window 8 and 10 users must disable the driver signing requirement. If the device driver is not installed, Windows will not detect Mono's serial port as a COM device. See Sparkfuns tutorial on How to Disable Signing Checks

Linux Users:

We have compiled a debian package for you. You need to use the *dpkg* tool to install the package, and then run *apt-get install* to install any dependencies. Non-debian users: You are skilled enough to compile monoprog from source.

Download Tic Tac Toe

Now, go back to the frontpage and click on the *Tic Tac Toe* app. You will now see this page, where you need to scroll down to the download link at the bottom right:



When you click the link, the file `ttt.elf` will be downloaded to your computer. This file is the application binary file and it is compatible with all present Mono models. Save the file in a folder, that you can easily reach from a terminal. Yes, now we must use terminal or command prompt.

Transfer the app to Mono

Attention: Since this is the first version of *monoprog*, it is a command line application. We have plan to wrap it inside a nice GUI - but for now bear with us.

Open a terminal window:

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

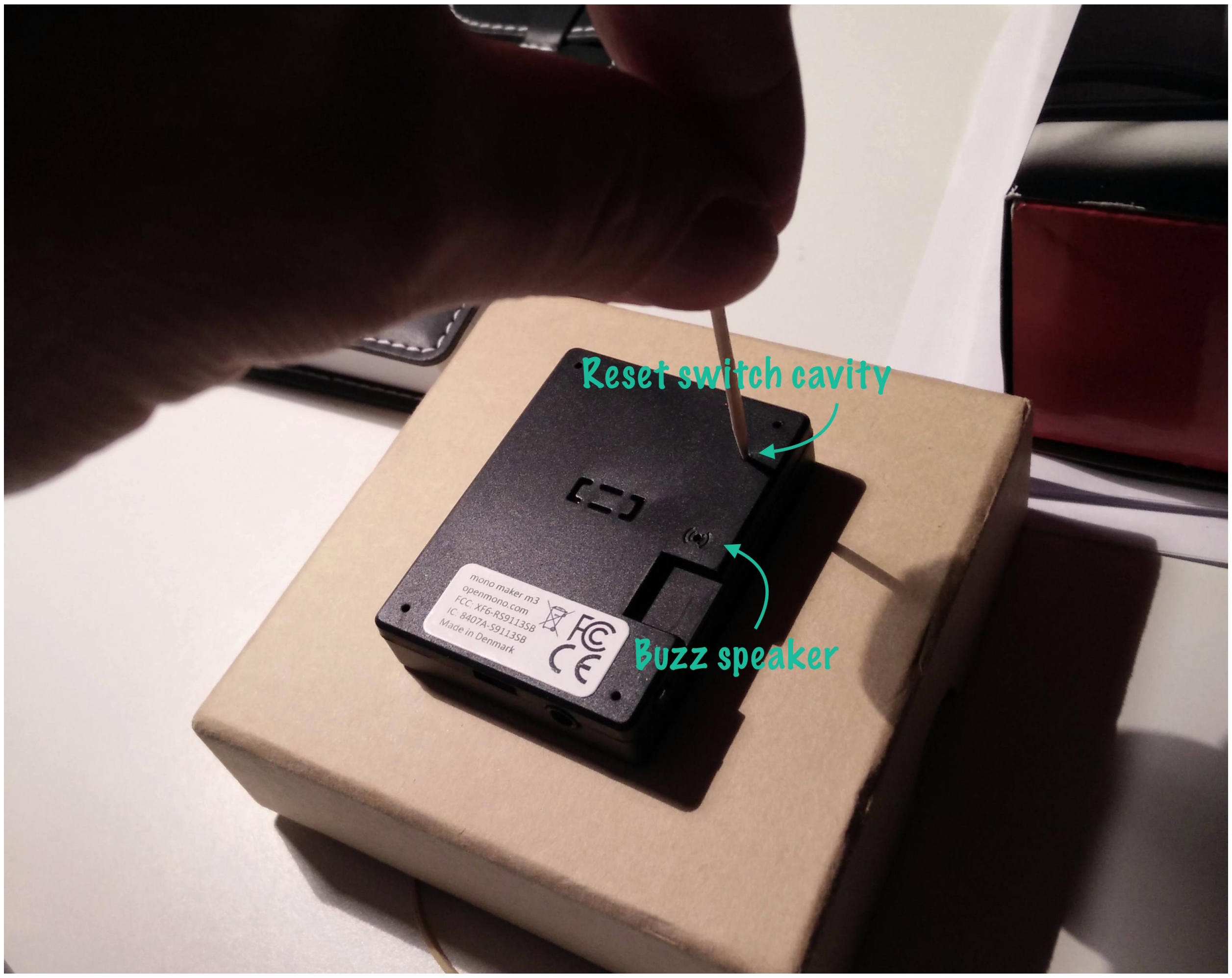
Type type this in the console, to verify monoprog is installed:

```
$ monoprog
Usage: monoprog [options]
Bootloadable Programmer for Mono board.

Options:
  -h, --help              Displays this help.
  -V, --version           Displays version information.
  --license               Displays licenses of software components.
  -d, --detect            Detects whether a Mono is connected via USB.
  --mock <type>           Simulates device to be in <type>.
  -p, --program <app>     Transfers <app> to Mono.
  -v, --verbose <level>   Set output verbosity level (default is 1).
  -q, --quiet, --silent   Set output verbosity level to 0.
```

If you see a message similar to this one, then everything is awesome! If your console brags about unknown command or application, then please run the installer again.

Connect Mono to your computer using a standard microUSB cable.

Before monoprog can transfer the application, your Mono must be in bootloader mode. Mono is in bootloader to 1 sec after reset, but you can force it to stay in bootloader:



1. Find a small piece of metal wire to insert into the reset switch hole, on Mono's back. (You can also use a paper clip.)
2. Then push the wire into the reset switch cavity, and hold down the user button on Mono's side. (***Note**: Do not push the wire into the buzzer opening.*)
3. Release the reset switch first, then release the user button.

Mono is now in bootloader and will stay in bootloader. Do a standard reset to bring Mono out of bootloader.

Danger: Do *not* leave Mono in bootloader! This will drain the battery until power is cut by the under voltage protection circuit. Even with such a circuit, the battery will still suffer from the discharging.

Next, navigate to the directory where you placed the file `ttt.elf`. Then write this *monoprog* command:

```
$ monoprog -d
```

Monoprog will now try to find any connected Mono devices. If your device is found it returns: *Mono device detected*.

If *monoprog* does not detect any connected Mono device, please check that you see Mono in *Device Manager*, it identifies itself as *PSoC3 Bootloader* HID device. Windows must say that it is *Ready to use*.

To transfer (program) the app to Mono write:

```
$ monoprog -p ttt.elf
```

Now, *monoprog* programs the application binary code 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 download the other app or you could consider creating your own!

2.2 Tutorials

2.2.1 Essentials

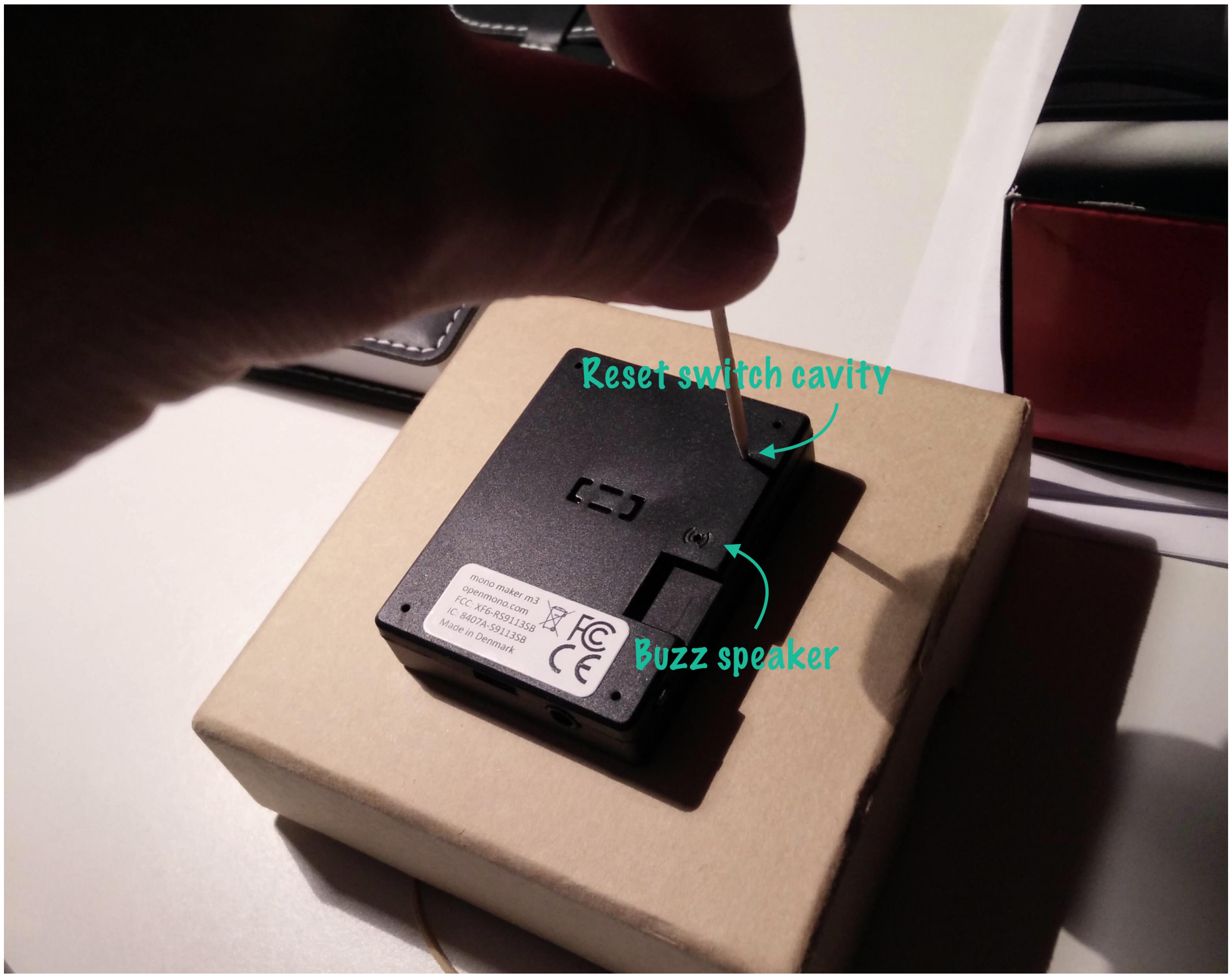
Resetting Mono

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 the 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 monoprog.

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 programmaticly trigger a reset from code! What happens is the CPU will reset itself if you explicitly tell it to do so. (That it, 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 `IApplicationContext`, and can be used like this:

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

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

// Reset to Bootloader
mono::IApplicationContext::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, therfore many serial terminals expects both a *carriage return* and a *line feed* character.

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

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 contructor
    AppController();

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


    // ...
```

Then in *app_controller.cpp*:

```
AppController::AppController()
{
    // in the contructor we setup the timer to fire periodicly (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!

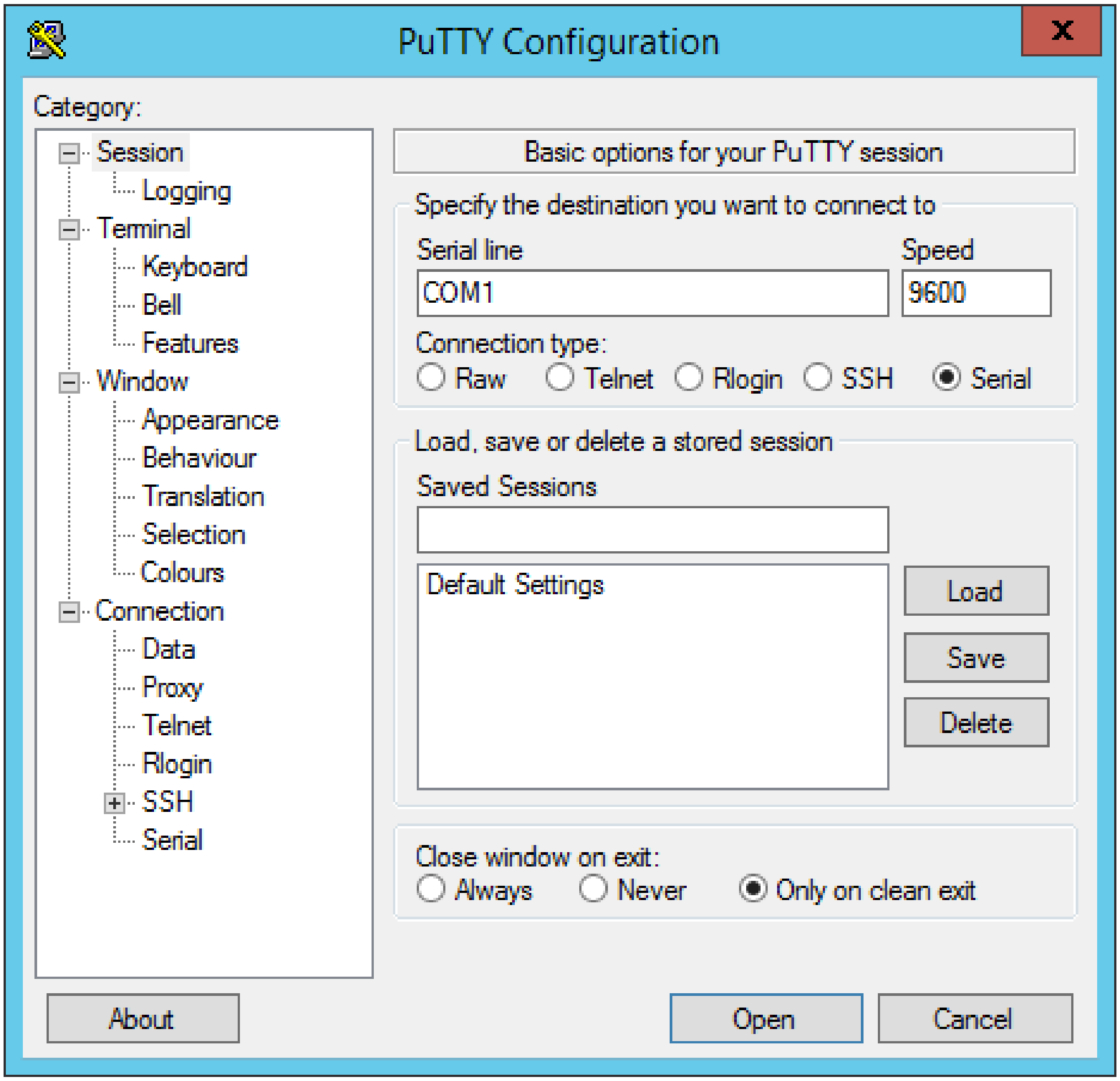
Connecting to Mono

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:



Specify the destination you want to connect to

Serial line: COM1

Speed: 9600

Connection type: Raw Telnet Rlogin SSH [✓ Serial]

Close window on exit: Always Never [✓ Only on clean exit]

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 loose 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.

Attention: Currently the serial API in the Mono's SDK support only writing. This means you can only transfer data from Mono to your serial terminal. On the next SDK release we will implement receiving data as well.

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 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.

Attention: Windows 8 and Windows 10 users will not have the driver installed automatically! These versions of Windows allows only signed drivers to be installed. We have yet to obtain our code signing certificate. In the mean time you can disable this check, by following this guide.

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 µA (micro amperes), which is really close to no consumption at all.

Default behaviour

Because it is crusial 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 dont't have to consider it too much, unless to wish to use the *User button* for something else.

Sleep and USB

The current implementation of the USB stack on Mono will wake Mono from sleep. This means that Mono cannot stay in sleep mode if it is connected to a computer. Therefore, to test sleep mode you need to unplug it from the USB Host, before you trigger sleep mode.

If you plug in Mono's USB while it is in sleep, it will wake up when the computer (USB Host) enumerates it. If you plug it in to a power-only USB tough, it will not wake up.

Bottom line: **Unplug the USB connection if you wish to use sleep mode!**

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 `IApplicationContext` object. The function is:

```
mono::IApplicationContext::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 automaticlly by Mono on device reset
    void monoWakeFromReset();
```

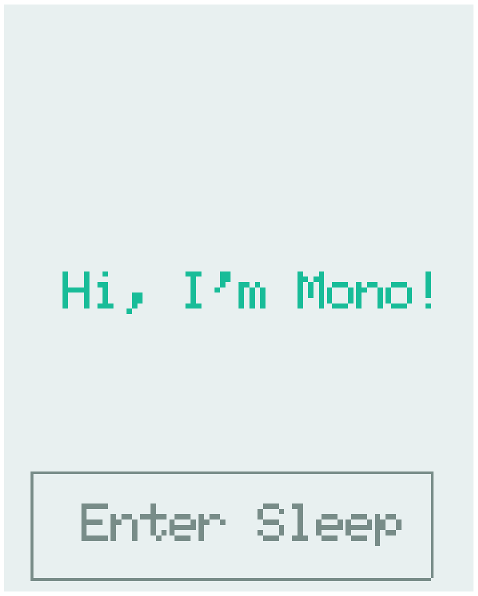
Then in the implementation file (*app_controller.cpp*), we define the body of the function to:

```
void AppController::gotoSleep()
{
    mono::IApplicationContext::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 AppController : 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 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() :

    //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();
}
```

2.2.2 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

Note: The network abstraction layer in *Mono Framework* is not implemented at this point. We will use the hardware layers directly, so this tutorial will be simpler in the future.

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 SPI communication to te Wifi module
2. Initialize the Wifi module
3. Connect to an access point, using either hardcoded credentials or read the credentials from SD card.
4. Using DHCP to get an IP address from the access point
5. 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 developer tool chain.

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

```
$ monomake project 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 add 2 includes, to import the wireless module definitions:

```
    #include <mono.h>
    #include <wireless/module_communication.h>
    #include <wireless/redpine_module.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. Two for the SPI communication and one the HTTP client class.

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 class members with:

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

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

    // The hardware SPI port
    mbed::SPI spi;
    // The spi based communication interface for the module
    redpine::ModuleSPICommunication spiComm;

    // The http client object variable
    network::HttpClient client;

public:

    AppController();

    // ...
}
```

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

Initializing the Communication channel

The wifi module is connected to Mono's MCU by a dedicated SPI. In the initial release of Mono Framework there is no abstraction layer for the Wifi module, so we have to initialize this SPI explicitly.

First we need to add the raw *mbed* SPI object to the *AppController*'s constructor list. Therefore we add two new lines next to the existing initialization of *helloLabel* object:

```
// You should init data here, since I/O is not setup yet.
AppController::AppController() :
    helloLabel(Rect(0,100,176,20), "Hi, I'm Mono!"),
    spi(RP_SPI_MOSI, RP_SPI_MISO, RP_SPI_CLK),
    spiComm(spi, NC, RP_nRESET, RP_INTERRUPT)
{
// ...
```

So what is happening here? We are setting up two objects: the basic SPI port and a SPI based communication channel to the module (`spiComm`). The module uses a few additional hardware signals, like the reset and interrupt signals. Now we have initialized the communication to the module, so we are ready to send commands to it!

The first thing we wanna do is tell the module to boot up and begin listening for commands. But we can not do that from the constructor, because the module might not be powered yet. We need to initialize it from the `monoWakeFromReset()` method:

```
void AppController::monoWakeFromReset()
{
    //initialize the wifi module
    redpine::Module::initialize(&spiComm);
```

Now the module will boot, so next we will tell it to connect to an access point.

Connecting to an Access Point

Let us begin with a hardcoded SSID and passphrase. (Still from inside the `monoWakeFromReset()` method.) Add this code line:

```
redpine::Module::setupWifiOnly("MY_SSID", "MY_PASSPHRASE");
```

Now the module will try to connect to the given access point, and expect to get a DHCP configured IP address. The `setupWifiOnly` function has a third parameter that defines 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 connecting 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 : mono::IApplication
{
// ...

public:

    void networkReadyHandler();

// ...
```

Next, we add the method body in the implementation file:

```
void AppController::networkReadyHandler()
{
    helloLabel.setText("Network Ready");
}
```

Notice that we use the existing `helloLabel` to display the network state on the screen.

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

```
redpine::Module::setNetworkReadyCallback<AppController>(this, &AppController::networkReadyHandle
```

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

Tip: Callback functions are an important part of using the network on Mono. If you wish to familiarize yourself with the concept, please see the in-depth article: *Queued callbacks and interrupts*

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.

Download a website

Now that we have connected to an access point with DHCP, I take the freedom to assume that Mono now has internet access! So lets 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 asynchrounous, (happening in the background), the process of downloading websites is asynchrounous. That means we are going to need another callback function, so lets define another method on *AppController.h*:

```
// ...

public:

    void networkReadyHandler();

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

// ...
```

Notice the ampersand (`&`) symbol that define the `data` parameter as a reference. In the implementation file we add the function body:

```
void AppController::httpHandleData(const network::HttpClient::HttpResponseData &data)
{
    helloLabel.setText(data.bodyChunk);

    if (data.Finished)
    {
        helloLabel.setText("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 set the label content to the HTML chunk, so it is not so pretty to look at. When the response has been downloaded, we set the text label to display *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()
{
    helloLabel.setText("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<AppController>(this, &AppController::httpHandleData);
}
```

Quick Examples

Counting variable on mono's screen.

This is a small example of how to show a counting variable on mono's screen.

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.

When mono (and some Arduino's) runs a program there is more going on than what you can see in the setup() and main() loop. Every time the main loop is starting over, mono will do some housekeeping. This includes tasks as updating the screen and servicing the serial port. This means that if you use wait functions or do long intensive tasks in the main loop, mono will never have time for updating the screen or listening to the serial port. This will also affect monos ability to receive a reset announcement, which is important every time you are uploading a new sketch.

If you are running into this you can always put mono into bootloader manually

1. press and hold down the user button on the side.
2. press and release the reset switch with a clips.
3. release the user button.

To avoid doing this every time the following example uses an alternative to the wait function. To slow down the counting, we 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.

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

For the time being the housekeeping is not optimized, we will work on this in near future. This means that the timing in your program will change when we update the framework. We are working on making a tutorial that shows how to make time-critical applications.

```
/***
 *
 * 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 lo
 * and an if() to detect when it reaches 1000 and then increment the counter and update the labe
 *
 ***/
#include <mono.h>

mono::ui::TextLabelView textLbl(mono::geo::Rect(0,20,176,20),"Hi, I'm Mono");

int loopItererations;
int counter;

void setup()
{
  textLbl.setTextColor(mono::display::WhiteColor);
  textLbl.show();

  // to prevent the framework from dimming the light
  CY_SET_REG8( CYREG_PRT5_BYP, 0 );                             // attention: this will affect a
  CyPins_SetPinDriveMode( CYREG_PRT5_PC1, CY_PINS_DM_STRONG );  // set drivemode to strong for TF
  CyPins_SetPin( CYREG_PRT5_PC1 );                              // set pin high for TFT LED back
```

```
    }

    void loop()
    {
      loopItererations++;

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

Attention: This example uses a hack to prevent dimmer of the display. This is only a temporary solution, and is not recommended. You should replace any use of the code when we release a best-practice method.

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 contructor 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!*

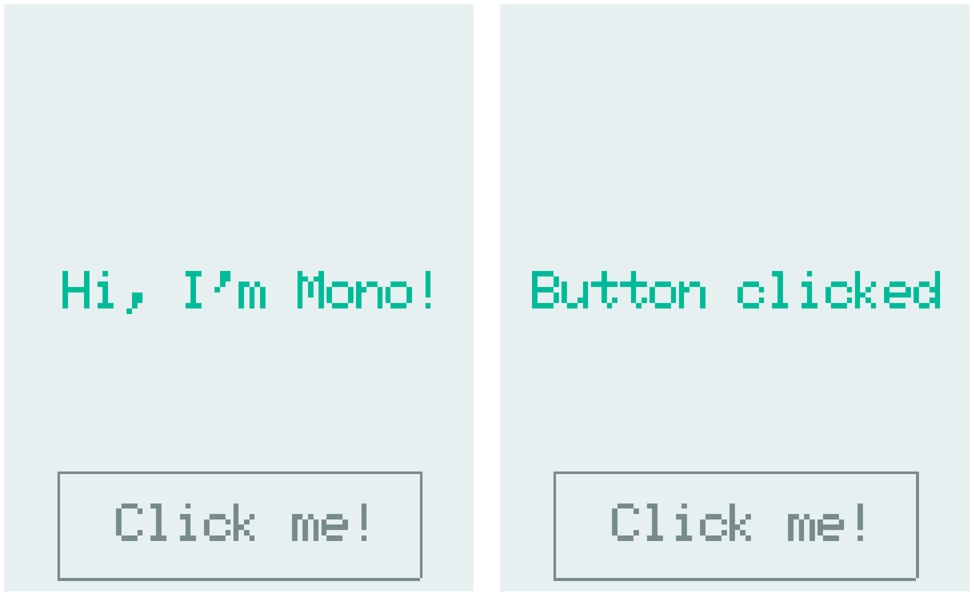
To trigger a response when we click the button, we need to implement the function body for the `buttonClick` method. In *app_controller.cpp* add this method:

```
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 out 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*:

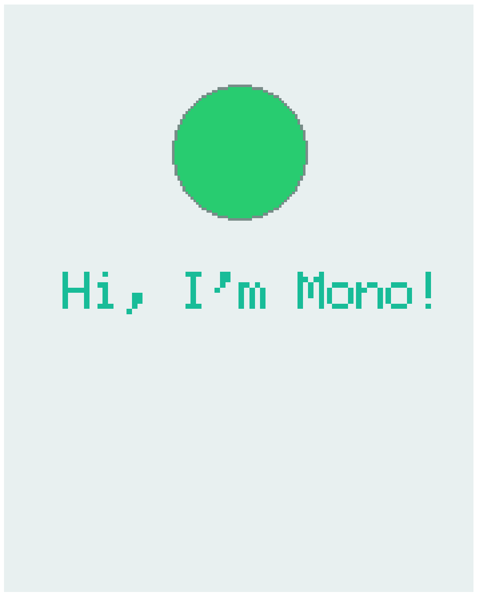
```
// tell the label to show itself on the screen
helloLabel.show();

// 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 AppController : 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 automaticly 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: \(^{\circ}F = ^{\circ}C \cdot 1.8 + 32 \)

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 lets visit the terms: absolute and relative measurements. An absolute temperature measurement is a temperature measured againts a fixed global reference. At the summer the sea temperature at the beaches reach \(25 ^{\circ}C\) or \(77 ^{\circ}F\). This is an absolute measurement. In contrast if I say: The sea temperature has rised by \(2 ^{\circ}C\) or \(3,5 ^{\circ}F\), rise in temperature is a reltive 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 \(1 ^{\circ}C\) or \(1.8 ^{\circ}F\), in absolute measurements. But their relative precision can be far better - like \(0.1 ^{\circ}C\) or \(0.18 ^{\circ}F\).

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 electronic 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.

Example Well, enough of that! 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. \(1 ^{\circ}C = 1000 ^{\circ}mC\)

2.3 In-Depth Articles

In this chapter we shall take a dive into the structure and workings of *Mono Framekwork*. 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.

2.3.1 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 what 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 you 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 awaken!");
```

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 reset button
- If Mono's battery is completely drained, the power control subsystem will cut the CPU's supply. When Mono is charged, the power to the CPU will be restored and a 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 We expect the POR events to be relatively rare, compared the number of sleep and wake up events. When Mono goes to sleep it turns off all peripherals to minimize power consumption.

Attention: Due to a software bug in the current release (1.0) of Mono Framework, Mono cannot wake up gracefully from sleep. It gets trapped in an infinite loop. Therefore we have added a reset to the default wake-up routine. So as of release 1.0, Mono will reset at wake-up.

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 clean-ups 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, it 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. 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() {}

setCallback(&MyFunction);
```

- we have callback functions for C++
- beside C function pointers, you can use C++ member pointers (type info is preserved!)

Events

- most apps a events based, they change state on events
- events are touch input, power/sleep triggers or button interrupts
- events are handled in callback functions and member method overrides
- a todo list app changes state on touch events, between events it does nothing

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 Pratice

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

2.3.2 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 in 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.

2.3.3 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

2.4 MonoKiosk

Mono apps are distributed through the Kiosk, and you can get your app into the Kiosk by following the recipe below.

2.4.1 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 forum.

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 pixes, 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.

2.4.2 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

2.5 API Reference

2.5.1 Core Classes

IApplication

class `mono::`**`IApplication`**

Entry point for all mono applications, abstract interface.

Every mono application must implement this interface. This is the starting point of the your application code, you must call it after the runtime initalization.

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 existance
    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* initalize it inside (not outside) the `main()` function. This is strictly nessesary, 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

`IApplication``()`

Construct the global Application class.

Constructor for the global Application Controller. See *IApplication* for a describtion on when to call this constructor.

virtual void **`monoWakeFromReset`**`()` = 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 inital 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 void **`monoWillGotoSleep`**`()` = 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 goto sleep, and at wakeup the *monoWakeFromSleep()* method will be called automatically.

Do not call this method yourself, it is ontended only to be called by the mono framework runtime.

virtual void **`monoWakeFromSleep`**`()` = 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 your self, it is intended only to be called by the mono framework runtime.

int **`enterRunLoop`**`()`

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 comformaty.

IApplicationContext

class `mono::`**`IApplicationContext`**

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.

Subclassed by mono::ApplicationContext

Public Functions

virtual int **`exec`**`()` = 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 void **`setMonoApplication`**`(`mono::*IApplication* **app*`)` = 0

Sets a pointer to the mono application object

Public Members

power::IPowerManagement ***`PowerManager`**

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 ***`RunLoop`**

A reference to the main run loop of the application. This pointer must be instanciated be subclasses

display::*IDisplayController* ***`DisplayController`**

Pointer to the display interface controller object. The object itself should be initialized differntly depending on the ApplicationContext

ITouchSystem ***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 ***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::handlerMet
}
```

sensor::*ITemperature* ***Temperature**

A pointer to the Temperatrure sensor, if present in hardware.

This is an automatically initialized pointer to the temperature object, that is automatically created by the framework.

sensor::*IAccelerometer* ***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 ***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.

Public Static Functions

static void **`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 **`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 **`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 **`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 **`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 **`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 ***`Instance`**

Get a pointer to the global application context

Protected Functions

virtual void **`enterSleepMode`**`()` = 0

Subclasses should overirde this method to make the sysetm goto sleep

virtual void **sleepForMs**(uint32_t *ms*) = 0
: Subclasses should override this to enable sleep mode for a specefic interval only.

virtual void **resetOnUserButton**() = 0
: Subclasses must implement this to enable the "Reset On User Button" behaviour. See *ResetOnUserButton*

virtual void **_softwareReset**() = 0
: Subclasses must implement this method to enable software resets. See *SoftwareReset*

virtual void **_softwareResetToApplication**() = 0
: Subclasses must implement this to enable software reset to application See *SoftwareResetToApplication*

virtual void **_softwareResetToBootloader**() = 0
: Subclasses must implement this method to allow *reset to bootloader* See *SoftwareResetToBootloader*

IApplicationContext(power::IPowerManagement **pwr*, *AppRunLoop* **runLp*, display::*IDisplayController* **dispCtrl*, *ITouchSystem* **tchSys*, *QueueInterrupt* **userBtn*, sensor::*ITemperature* **temp* = 0, sensor::*IAccelerometer* **accel* = 0, sensor::IBuzzer **buzzer* = 0)
: 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 contructor did not setup the pointers, the PowerManagement constructor would see the *Instance* global equal `null`.

AppRunLoop

class `mono::`**AppRunLoop**
: This is the event run-loop for all mono applications. This class is instantiated and used inside the *IApplicationContext* interface. You should not interact with this class directly.

  The run loop handles non-critical periodicly 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 staticly inside the loop, like the USB serial reads.

Public Functions

void **exec**()
: Start executing the run loop.

void **CheckUsbDtr**()
: Do a single check of the DTR on the virtual UART.

bool **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 **removeDynamicTask**(IRunLoopTask **task*)
: Remove a task from the dynamic task queue. This will search the queue for he 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 **`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 **`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 **`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 **`TouchSystemTime`**

The CPU time used on proccessing touch input. This includes:

- ADC sampling (approx 16 samples)
- Touch value evaluation, and possible convertion 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 **`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

void **`processDynamicTaskQueue`**()

Execute all tasks in the dynamic task queue

void **removeTaskInQueue**(IRunLoopTask **task*)
Internal method to sow together neightbourghs in the linked list

void **process**()
Process a single iteratio of the run loop

void **checkUsbUartState**()
read the UART DTR state if possible

Protected Attributes

bool **runLoopActive**
As long as this i `true` the stadard run loop will run

If set to `false`, the run loop will exit, and mono might will enter a low power state. TODO: power safe modes and run loops?

bool **lastDtrValue**
The last seen serial DTR value. Reset can only happen in transitions.

bool **resetOnUserButton**
Set to `true` if you want the run loop to call software reset when pressing the user button. Initial value is `false`

IRunLoopTask ***taskQueueHead**
A pointer to the head task of the dynamic task queue. If no task are in the queue, this is NULL

IRunLoopTask

class `mono::`**IRunLoopTask**
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 that want to use the run loop, must inherit this interface.

NOTE that tasks in the run loop do not have any contraints on how often or how rare they are executed. If you need a function called at fixes intervals, use a Ticker or timer.

Subclassed by mono::display::ILI9225G, mono::power::MonoPowerManagement, *mono::QueueInterrupt*, *mono::Timer*, mono::ui::Animator

Protected Functions

virtual void **taskHandler**() = 0
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.

Protected Attributes

IRunLoopTask ***previousTask**
A pointer to the previous task in the run loop The the task is the first in queue, this is NULL

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 **singleShot**
Tasks are expected to be repetative. They are scheduled over and over again. Set this property to `true` and the task will not scheduled again, when handled.

String

class `mono::`**`String`**
High level string class, that is allocated on the HEAP or `rodata`

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.

If you create a string from some static C text string (of type `const char*`), the compiler will put the data in the read-only data segment (`.rodata`). The string object would not alloc anything on the HEAP, but just reference the static string in the data segment.

These features makes the class very lightweight and safe to pass around functions and objects.

Timer

class `mono::`**`Timer`**
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 your can do any kind of heavy lifting in your callback handler, and your code is not pre-empted.

Presicion

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 reocurring 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.

Inherits from *mono::IRunLoopTask*

Public Functions

`Timer`()

Contruct an empty (zero-timeout) re-occurring timer.

After calling this contructor, you must set the time out and callback function. Then start the timer.

`Timer`(uint32_t *intervalOrTimeoutMs*, bool *snglShot* = false)

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 **`Start`**()

Start the timer and put into *running* state.

Note: You must set a callback handler, before starting the timer.

void **`Stop`**()

Stop the timer, any pending callback will not be executed.

bool **`SingleShot`**()

See if the timer is single shot.

bool **`Running`**()

See if the timer is currently running

void **`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 **`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* ***`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 void **`taskHandler`**`()`

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.

DateTime

class `mono::`**`DateTime`**

A *DateTime* representation.

Public Functions

`DateTime``()`

Construct an empty / invalid *DateTime* object.

`DateTime``(`uint16_t *years*, uint8_t *months*, uint8_t *days*, uint8_t *hours* = 0, uint8_t *minutes* = 0, uint8_t *seconds* = 0`)`

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

String **`ToString`**`()` const

Return the *DateTime* object as a huamn readable string.

Return A mono string on the format: yyyy-MM-dd hh:mm:ss

ISettings

Warning: doxygenclass: Cannot find class "mono::ISettings" in doxygen xml output for project "monoapi" from directory: xml

QueueInterrupt

class `mono::`**`QueueInterrupt`**

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. THere-fore 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.

Inherits from InterruptIn, *mono::IRunLoopTask*

Public Functions

`QueueInterrupt`(PinName *inputPinName* = NC, PinMode *mode* = PullNone)

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 **`DeactivateUntilHandled`**(bool *deactive* = true)

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

- `OPTIONAL` - Set this to false, to *not* disable interrupts while processing. Default is `true`

bool **`IsInterruptsWhilePendingActive`**() const

Get 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 **`setDebouncing`**(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 **`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 **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 **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 **FallTimeStamp** ()

On fall interrupts, this is the µSec. ticker timestamp for the falling edge inetrrupt. You can use this to calculate the time passed from the interrupt occured, to the time you process it in the application run loop.

Return The ticker time of the falling edge in micro seconds

uint32_t **RiseTimeStamp** ()

On rise interrupts, this is the µSec. ticker timestamp for the rising edge inetrrupt. You can use this to calculate the time passed from the interrupt occured, to the time you process it in the application run loop.

Return The ticker time of the rising edge in micro seconds

bool **Snapshot** ()

The pin value at the moment the H/W interrupt triggered The callback might be executed some time after the actual inetrrupt occured. THis method return the pin state at the moment of the interrupt.

Return The pin state, at the time of the interrupt

template <typename T>
void mono::QueueInterrupt::change(T * tptr, void(T::*)(void) mptr)

Attach a function to call when a rising or falling edge occurs on the input

Parameters

- `fptr` - A pointer to a void function, or 0 to set as noneAttach a member function to call when a rising or falling edge occurs on the input

- `tptr` - pointer to the object to call the member function on
- `mptr` - pointer to the member function to be called

Protected Functions

virtual void **`taskHandler`**`()`

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.

TouchResponder

class `mono::`**`TouchResponder`**

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

ITouchSystem

Inherits from mono::IQueueItem

Subclassed by *mono::ui::ResponderView*

Public Functions

`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 **`Activate`**`()`

Add this responder to the responder chain

void **`Deactivate`**`()`

Remove this responder from the responder chain

TouchEvent

class mono::**TouchEvent**

Public Members

TouchEvent ***TouchBeginEvent**

If

ITouchSystem

class mono::**ITouchSystem**

Interface for the Touch sub-system

Subclassed by mono::MonoTouchSystem

Public Functions

virtual void **init**() = 0

Initialize the touch system controller.

<# description #>

virtual void **processTouchInput**() = 0

<# brief desc #>

<# description #>

Protected Functions

void **runTouchBegin**(geo::*Point* &*pos*)

<# brief desc #>

<# description #>

Parameters

- - param desc #>

void **runTouchMove**(geo::*Point* &*pos*)

<# brief desc #>

<# description #>

Parameters

- - param desc #>

void **runTouchEnd**(geo::*Point* &*pos*)

<# brief desc #>

<# description #>

Parameters

- - param desc #>

Regex

class `mono::`**`Regex`**

This class is a C++ wrapper around the C library called SLRE (Super Lightweight Regular Expressions)

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/slre

Public Types

typedef slre_cap **`Capture`**

Regex Match capture object holding the first match capture

See *Regex*

Public Functions

`Regex`(String *pattern*)

Create a regular expression object from a pattern string

bool **`IsMatch`**(String *matchStr*)

Test if a string matches the regex pattern

Return `true` on match, `false` otherwise

bool **`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.

```
Regex::Capure caps[3];
Regex reg(``(..) (..) (..)'');
bool success = reg.Match(``test my regex'', caps, 3);
```

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 **`Value`** (Capture &*cap*)
  Return the string value from a match capture object

Queue

class `mono::`**`Queue`**
  A pointer based FIFO style *Queue*.

  See *GenericQueue*

  Subclassed by *mono::GenericQueue< mono::redpine::ManagementFrame >*, *mono::GenericQueue< mono::TouchResponder >*, *mono::GenericQueue< mono::ui::View >*, *mono::GenericQueue< Item >*

Public Functions

void **`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 ***`Dequeue`** ()
  Returns and removes the oldest element in the queue.

IQueueItem ***`Peek`** ()
  Return the oldest element in the queue, without removing it.

IQueueItem ***`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 **`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

GenericQueue

template <typename *Item*>
class `mono::`**`GenericQueue`**
  A templated *Queue*, where template defines the queue element type.

  This class is identical to *Queue*, but it uses templating to preserve type information.

  See *Queue*

  Inherits from *mono::Queue*

2.5.2 UI Widgets

View

class `mono::ui::`**`View`**

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 inputs and painting to the display buffer automatically.

All views has a width and height, along with an absolute x,y coordinate. The coordinates define the upper left corner of the view rectangle.

As a design pattern, we chose that Views must not contain any state. They only draw data to the display. Therefore view might contain or have references to objects holding the actual state information.

The 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

See *ResponderView*

Inherits from mono::IQueueItem

Subclassed by mono::ui::BackgroundView, *mono::ui::ConsoleView< W, H >*, *mono::ui::GraphView*, *mono::ui::ImageView*, *mono::ui::ProgressBarView*, *mono::ui::ResponderView*, *mono::ui::StatusIndicatorView*, *mono::ui::TextLabelView*

Public Types

enum `Orientation`

Define the 4 dirrerent orientations the of display. The display controller apply the orientation transformation to real display. For the UI Views the coordinate system remains the same, it just changes width and height. The orico is always the top left corner (defined relative to gravity), no matter the physical orientation of mono's display.

Values:

`PORTRAIT` = 0

Expected standard orientation of mono, where the thick edge is at the bottom

`PORTRAIT_BOTTOMUP` =1

Upside-down of *PORTRAIT*, where the thick edge is at the top

`LANDSCAPE_RIGHT` = 2

PORTRAIT rotated 90 degrees clock-wise

`LANDSCAPE_LEFT` = 3

PORTRAIT rotated 90 degrees counter clock-wise

Public Functions

`View``()`

Construct an empty view, you should not do this! You should not use *View* directly, subclass it instead.

`View`(geo::Rect *rect*)

Construct a view with dimensions, you should not do this! You should not use *View* directly, subclass it instead.

virtual void **`setPosition`**(geo::Point *pos*)

Change the views position on the screens coordinate system.

Changes the views 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.

Parameters

- `pos` - The new position of the view

virtual void **`setSize`**(geo::Size *siz*)

Change the size (width, height) of the view.

Changes the views dimensions. The affect of size changes might depend on the specefic view subclass. Some views might use their size to calculate their internal layout - other might only support fixed sizes.

Note that changing the size here does not redraw the view. The screen needs to be redrawn the affected places, to make the size change visible.

Parameters

- `siz` - The new size of the view

virtual void **`setRect`**(geo::Rect *rect*)

Set the views position and size, by providing a rectangle object.

Note that this method does not repaint the view, you must do that explicitly.

Parameters

- `rect` - The view rectangle, containing size and position

virtual mono::geo::*Point* &**`Position`**()

Get the current position of the views upper left corner.

Return A reference to the current position

virtual mono::geo::*Size* &**`Size`**()

Get the views current size rectangle.

Return A reference to the views size rectangle

void **`scheduleRepaint`**()

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 view has no state information, they do not know when to repaint themselves. You or classes using views, must manually call this repaint method when its time to repaint the view.

virtual bool **`Visible`**() const

Returns the views 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*

Return `true` if the view can/should be painted on the screen, `false` otherwise.

See *show*

hide

virtual void **`show`**`()`

Set the view to visible, and paint it.

Change the views visibility state to visible. This means it can be scheduled for repaint by *scheduleRepaint*. This method automatically schedules the view for repaint.

See *hide*

Visible

virtual void **`hide`**`()`

Set the view to be invisible.

Change the views 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.

See *show*

Visible

Public Static Functions

uint16_t **`DisplayWidth`**`()`

Returns the horizontal (X-axis) width of the display canvas, in pixels. The width is always defined as perpendicular to gravity

uint16_t **`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 **`DisplayOrientation`**`()`

Returns the current physical display orientation of the display The orientation is controlled by the IDisplayController

Public Static Attributes

uint32_t **`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 comminicate with the disdplay hardware.

Protected Functions

void **`callRepaintScheduledViews`**`()`

A member method to call the static method *repaintScheduledViews*.

See *repaintScheduledViews*

virtual void **repaint**() = 0

Repaint the view content, using the *View::painter*.

Re-paint the view content. This method should be called then 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

geo::*Rect* **viewRect**

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 **isDirty**

Indicate is this view should be repainted on next display refresh.

bool **visible**

Views can be visible of non-visisble (hidden). When a view is *not* visible *scheduleRepaint* will ignore requests.

You should use the methods *show* and *hide* is toggle visibility.

See *show*

hide

Visible

Protected Static Functions

void **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 be the display system, you do not need to call it yourself.

Protected Static Attributes

mono::display::*DisplayPainter* **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 themselv on another display, you must subclass or change the current display controller of the mono::ApplicationContext object.

mono::*GenericQueue*<*View*> **dirtyQueue**

The global re-paint queue.

When you call the *scheduleRepaint* method, your views is added to the re-paint queue.

TextLabelView

class `mono::ui::`**`TextLabelView`**

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 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:
`TextLabelView lbl(``This is a contant 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 it 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 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 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 texts dimensions in pixels

- regardless of view rectangle. Also, you can use these methods before the view has been rendered.

Inherits from *mono::ui::View*

Public Types

enum `TextAlignment`

Three ways of justifing text inside the TextLabel.

Values:

`ALIGN_LEFT`

Align text to the left

ALIGN_CENTER
Align text in the center

ALIGN_RIGHT
Align text to the right

Public Functions

TextLabelView()
Construct an empty TextLabel, with no dimensions and content.

An empty view has view dimensions (0,0,0,0) and the empty string as its content. You have setup its dimensions and its content before it can be rendered on the display.

See *setRect*

setText

TextLabelView(*String* *txt* = String ())
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(**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(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(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 **`TextSize`**() const
MARK: Getters.

The text size will be phased out in coming releases. You control text by changing the font.

void **`setTextSize`**(uint8_t *newSize*)
MARK: Setters.

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 **`setTextColor`**(display::Color *col*)

void **`setBackgroundColor`**(display::Color *col*)

void **`setText`**(display::Color *col*)
Set the text color

void **`setBackground`**(display::Color *col*)
Set the color behind the text

void **`setAlignment`**(TextAlignment *align*)
Controls text justification: center, right, left

void **`setFont`**(MonoFont **const** &*newFont*)
Set a new font face on the label.

You can pass any MonoFont to the label to change its appearence. Fonts are header files that you must include youself. Each header file defines a font in a specific size.

The header file defines a gloabl `const` variable that you pass to to this method.

virtual void **`repaint`**()
MARK: Aux Functions.

Public Static Attributes

const MonoFont ***`StandardTextFont`**
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 appearence of all TextLabels.

You can also overwrite it to use a less memory expensive (lower quality) font face.

ButtonView

class `mono::ui::`**`ButtonView`**
Inherits from *mono::ui::ResponderView*

Public Functions

void **setText** (String *txt*)
MARK: Accessors.

void **setBorder** (Color *c*)
Sets the border and text color This method will not schedule repaint!

void **setHighlight** (Color *c*)
Sets the active / pressed color (border & text) This method will not schedule repaint!

void **setBackground** (Color *c*)
Sets the background color This method will not schedule repaint!

const TextLabelView &**TextLabel** () const
Get a reference to the internal TextLabel object.

template <**typename** Owner>
void mono::ui::ButtonView::setClickCallback(Owner * obj, void(Owner::*)(void) memPtr)
MARK: Callbacks.

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

void **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.

virtual void **repaint** ()
Painters.

Protected Functions

virtual void **TouchBegin** (TouchEvent &*event*)
MARK: Touch Handlers.

void **initButton** ()
MARK: Conructors.

ProgressBarView

class mono::ui::**ProgressBarView**
Inherits from *mono::ui::View*

Public Functions

ProgressBarView ()
Create a ProgressBar with a zero view rect (0,0,0,0)

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 **`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 **`setMaximum`**(int *max*)

SETTERS.

template <typename Owner>
`void mono::ui::ProgressBarView::setValueChangedCallback(Owner * cnxt, void(Owner::*)(vo`

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 **`Minimum`**() const

GETTERS.

int **`Maximum`**() const

GETTERS.

Protected Functions

void **`init`**()

convenience initializer

virtual void **`repaint`**()

MISC.

StatusIndicatorView

class `mono::ui::`**`StatusIndicatorView`**

Indicate a boolean status, true/false, on/off or red/green.

The status indicator displays a circular LED like widget, that is red or green by default.

Inherits from *mono::ui::View*

Public Functions

void **setOffStateColor** (display::Color *col*)
MARK: SETTERS.

bool **State** () const
MARK: GETTERS.

Protected Functions

void **initializer** ()
MARK: Constructors.

virtual void **repaint** ()
MARK: Auxilliaries.

ConsoleView

template <uint16_t *W*, uint16_t *H*>
class mono::ui::**ConsoleView**
Inherits from *mono::ui::View*

Public Functions

ConsoleView (geo::*Point* *pos*)
Construct a new *ConsoleView*, for viewing console output on the screen.

void **WriteLine** (*String* *txt*)
Write a string to the console, and append a new line.

Parameters

- `txt` - The string to write to the console

virtual void **repaint** ()
Repaint the view content, using the *View::painter*.

Re-paint the view content. This method should be called then 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 **setCursor** (geo::*Point* *pos*)
MARK: Auxilliary methods.

int **lineLength** ()
Get the width of a line in characters.

<# description #>

Return Number of characters in one line int **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> **textBuffer**

Text buffer that hold the visible text in the console. When the console scrolls, the text in the buffer is overwritten.

bool **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.

ImageView

class mono::ui::**ImageView**

<# brief desc #>

Inherits from *mono::ui::View*

Public Functions

ImageView()

Construct an empty image view, where no image is displayed To display an image you need to call *setImage* later.

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 consructor!*

void **setImage**(media::Image **img*)

Set a new image source object *Note: This method also resets the current cropping rectangle!*

virtual void **repaint**()

Repaint the view content, using the *View::painter*.

Re-paint the view content. This method should be called then 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

media::Image ***image**

A pointer to the Image object that is to be displayed in the screen.

geo::*Rect* **crop**
  Image crop rectangle. Setting this rect will crop the source image (non destructive).

GraphView

class mono::ui::**GraphView**
  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 *IGraphView-DataSource* 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 overiding 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*

Inherits from *mono::ui::View*

Public Functions

`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

setRect

`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``(`geo::Rect *rect*, **const** IGraphViewDataSource &*dSource*`)`

Construct a *GraphView* with `viewRect` and a data source.

Parameters

- `@param` -

Protected Functions

virtual void **`repaint`**`()`

Repaint the view content, using the *View::painter*.

Re-paint the view content. This method should be called then 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 `mono::ui::`**`IGraphViewDataSource`**

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 retreiving 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:

- *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 int **`DataPoint`**`(`int *index*`) = 0`

Override 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.

Return the sample value at the given index

Parameters

- `index` - The 0-indexed sample position, that should be returned

virtual int **`BufferLenght`**`()` = 0

Override 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.

Return the size / length of the data source

virtual int **`MaxSampleValueSpan`**`()` = 0

The value span for the data samples. üsed to map the buffer samples to the screen (view height)

virtual int **`NewestSampleIndex`**`()`

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.

Return Position of newest sample

TouchCalibrator

Warning: doxygenclass: Cannot find class "mono::ui::TouchCalibrator" in doxygen xml output for project "monoapi" from directory: xml

ResponderView

class `mono::ui::`**`ResponderView`**

Inherits from *mono::ui::View*, *mono::TouchResponder*

Subclassed by *mono::ui::ButtonView*, mono::ui::TouchCalibrateView

Public Functions

virtual void **`show`**`()`

Shows (repaints) the view and insert into the responder chain.

See *View::show*

virtual void **`hide`**`()`

hides the view, and remove from the responder chain

See *View::hide*

Protected Functions

virtual void **RespondTouchBegin** (TouchEvent &*event*)
Internal touch handler, you should not overwrite this

virtual void **RespondTouchMove** (TouchEvent &*event*)
Internal touch handler, you should not overwrite this

virtual void **RespondTouchEnd** (TouchEvent &*event*)
Internal touch handler, you should not overwrite this

2.5.3 Drawing

DisplayPainter

class `mono::display::`**`DisplayPainter`**
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 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 dependend on the textsize property. Text painting is not affected by the current line width.

Public Functions

`DisplayPainter` (IDisplayController **displayController*)
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

template <typename Owner>
`void mono::display::DisplayPainter::setRefreshCallback(Owner * obj, void(Owner::*)(voi`
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 shoould have its member function called
- `memPtr` - A pointer the the class' member function.

void **`setForegroundColor`**(Color *color*)

MARK: *Color* Accessors.

Set the painters foreground pencil color

Parameters

- `color` - The new foreground color

void **`setBackgroundColor`**(Color *color*)

Sets the painters background pencil color.

Parameters

- `color` - the new background color

Color **`ForegroundColor`**() const

Gets the painters current foreground pencil color.

Return The current foreground color

Color **`BackgroundColor`**() const

Gets the painters current background pencil color.

Return The current foreground color

void **`useAntialiasedDrawing`**(bool *enable* = true)

Turn on/off anti-aliased line drawing.

MARK: Anti-aliasing Accessors.

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 **`IsAntialiasedDrawing`**()

Returns `true` if anti-aliased drawing is enabled.

See *useAntialiasedDrawing*

Return `true` if enabled, `false` otherwise.

uint8_t **`LineWidth`**() const

MARK: Pencil Property Accessors.

uint16_t **`CanvasWidth`**() const

MARK: Painting Canvas Info Accessors.

Get the canvas width in pixels. This is the display display width as well.

Return The canvas/display width in pixels uint16_t **`CanvasHeight`**`()` const

Get the canvas height in pixels. This is the display display height as well.

Return The canvas/display height in pixels

IDisplayController ***`DisplayController`**`()` const

Get 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 **`drawPixel`**`(`uint16_t *x*, uint16_t *y*, bool *background* = false`)`

MARK: Drawing methods.

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 **`drawPixel`**`(`uint16_t *x*, uint16_t *y*, uint8_t *intensity*, bool *background* = false`)`

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 corrosponds 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 puxels y coordinate
- `intensity` - The alpha value, 0 to 255 where 0 is transparent.
- `Optionaluse` - the background color as foreground and vice versa

void **`drawFillRect`**`(`uint16_t *x*, uint16_t *y*, uint16_t *width*, uint16_t *height*, bool *background* = false`)`

Draw a filled rectangle.

Paints a filled rectangle in the actuive foreground color. Coordinates a 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 **drawLine** (uint16_t *x1*, uint16_t *y1*, uint16_t *x2*, uint16_t *y2*, bool *background* = false)

Draw a straight line between two points.

MARK: Lines.

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 of vertical, this method will usemore 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 **drawRect** (uint16_t *x*, uint16_t *y*, uint16_t *width*, uint16_t *height*, bool *background* = false)

Draw a outlined rectangle.

MARK: Rects.

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 **drawChar** (uint16_t *x*, uint16_t *y*, char *character*)

Draw a single character on the display.

MARK: Simple Characters.

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, width 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 **drawVLine** (uint16_t *x*, uint16_t *y1*, uint16_t *y2*, bool *background* = false)

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 **drawHLine** (uint16_t *x1*, uint16_t *x2*, uint16_t *y*, bool *background* = false)

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 **drawCircle** (uint16_t *x0*, uint16_t *y0*, uint16_t *r*, bool *background* = false)

Paint an outlined circle.

MARK: Circles.

Protected Functions

void **swap** (uint16_t &*a*, uint16_t &*b*)

MARK: Simple Helper Methods.

Inline swap of two numbers.

Protected Attributes

mbed::FunctionPointer **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 contructed, to enable the re-paint queue.

Color

class `mono::display::`**`Color`**

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()`
MARK: Constructors.

MARK: Public contructors.

uint8_t **`Red`**`()` const
MARK: Getters.

Return 8-bit red color component

uint8_t **`Green`**`()` const
Return 8-bit green color component

uint8_t **`Blue`**`()` const
Return 8-bit blue color component

Color **`scale`**(uint8_t *factor*) const
Misc.

Multiply each RGB channel by a factor from 0-255

Return the scaled color

Color **`blendMultiply`**(*Color other*) const
Return the product of two colors

Return the multiply blended color

Color **`blendAdditive`**(*Color other*) const
Add this color with another

Color **`invert`**`()` const
Return the inverse

uint8_t ***`BytePointer`**`()`
MARK: Misc.

uint16_t **`operator=`**(*Color col*)
MARK: Operator overloads.

IDisplayController

class `mono::display::`**`IDisplayController`**

Abstract Interface for display controllers like ILITEK chip etc.

This interface a simple set of function that display interface must implement.

Mono dislpay 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.

Author Kristoffer Andersen

Subclassed by mono::display::HX8340, mono::display::ILI9225G

Public Functions

`IDisplayController`(int *width*, int *height*)

Setup the display controller object, init vaiables 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 void **`init`**() = 0

Initializes the hardware display controller.

Initalizing 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 nessasary to do any draw commands from this method, other classes will take care of clearing the screen to black, etc.

virtual void **`setWindow`**(int *x*, int *y*, int *width*, int *height*) = 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 acrtive painting area of the screen, where you can paint. Upon drawing a single pixel the dispay cursor increments inside the window. This means you can sometime skip calls to the *setCursor* method.

Parameters

- `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 **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 void **setCursor**(int *x*, int *y*) = 0

Set the drawing cursor to a new absolute position.

Sets the current drawing cursor to a new position (must be within the cuurently active window).

When setting the cursor you must use absolute coordinates (screen coordinates), not coordinates inside the active window.

Parameters

- `x` - The new X position (screen coordinates)
- `y` - The new X position (screen coordinates)

virtual void **write**(Color *pixelColor*) = 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.

Parameters

- `pixelColor` - The 16-bit 5-6-5 RGB color to draw

virtual void **setBrightness**(uint8_t *value*) = 0

Set the display backlight brightness. Higher values means more brightness. The display controller implementation might use a PWM to switch backlight LED's.

Parameters

- `value` - The brightness 0: off, 255: max brightness

virtual void **setBrightnessPercent**(float *percent*)

Set the display backlight brightness. Higher values means more brightness. The display controller implementation might use a PWM to switch backlight LED's.

Parameters

- `percent` - The brightness percentage, 0.00: off, 1.00: max brightness virtual uint8_t **Brightness**() 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.

Return The current brightness level in 8-bit format: 0: off, 255: max brightness

Public Members

uint32_t **LastTearningEffectTime**

The last tearning effect interrupt time (`us_ticker_read()`)

To calculate the time since the last tearning 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 occured 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 tearning effect interrupt handler.

2.5.4 Geometry

Point

class `mono::geo::`**Point**

Class representing a point in a 2D cartisian coordinate system The point has no width or height, omly an x-coordinate and a y-coordinate

This class defines the coordinates, as well as method to manipulate the point. Also functions for geometrical calculus is present.

Subclassed by *mono::geo::Circle*, *mono::geo::Rect*

Size

class `mono::geo::`**Size**

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.

Subclassed by *mono::geo::Rect*

Public Functions

Size()

Construct a NULL size. A size that has no width or height.

Size(int *w*, int *h*)

Construct a size with defined dimenstions

Size(**const** *Size* &*s*)

Construct a size based on another size

Rect

class `mono::geo::`**`Rect`**

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.

See *Size*

Point

Inherits from *mono::geo::Point*, *mono::geo::Size*

Public Functions

`Rect`(int *x*, int *y*, int *width*, int *height*)

Construct a rectangle from position coordinates and size.

`Rect`(Point &*p*, Size &*s*)

Construct a rectangle from *Point* and *Size* objects.

`Rect`()

COnstruct an empty rectangle having position (0,0) and size (0,0)

Point **`UpperLeft`**() const

Reutrn 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 **`LowerRight`**() const

Reutrn the position of the lower right corner.

Return The point of the lower right corner

Point **`UpperRight`**() const

Reutrn the position of the upper right corner.

Return The point of the upper right corner

Point **`LowerLeft`**() const

Reutrn the position of the lower left corner.

Return The point of the lower left corner

class Point **`Center`**() const

Return the position of the Rectangles center.

void **`setPoint`**(**class** Point *p*)

Move (translate) the rectangle by its upper left corner.

void **`setSize`**`(`**class** Size *s*`)`
: Set a new size (width/height) of the rectangle.

bool **`contains`**`(`**class** Point &*p*`)` const
: Check whether or not a *Point* is inside this rectangle.

Rect **`crop`**`(`*Rect* **const** &*other*`)` const
: Return this *Rect* cropped by the baundaries of another rect

mono::*String* **`ToString`**`()` const
: Return a string representation of the ractangle.

  Return A string of the form `Rect( x, y, w, h )`

Circle

class `mono::geo::`**`Circle`**
Inherits from *mono::geo::Point*

Public Functions

`Circle``()`
: MARK: Contructors.

uint32_t **`Radius`**`()` const
: MARK: GETTERS.

2.5.5 Sensors

IAccelerometer

class `mono::sensor::`**`IAccelerometer`**
Abstract inteface for interacting with the accelerometer.

Subclassed by mono::sensor::MMAAccelerometer

Public Functions

virtual void **`Start`**`()` = 0
: 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 void **`Stop`**`()` = 0
: Stops the accelerometer.

  A stopped accelerometer can not sample acceleration. *Start* the accelerometer before you sample any axis.

virtual bool **`IsActive`**`()` = 0
: Return the current Start/Stop state of the accelerometer.

  Return `true` only if the accelerometer is started and sampling data

virtual int16_t **rawXAxis**() = 0
<# brief desc #>

Return <# return desc #>

ITemperature

class `mono::sensor::`**`ITemperature`**
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.

Subclassed by mono::sensor::AT30TS74Temperature, mono::sensor::PCT2075Temperature

Public Functions

virtual int **Read**() = 0
Reads the current temperature from the temperature sensor

Return the temperature in Celcius

virtual int **ReadMilliCelcius**()
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

Return The temperature in mCelcius

IPowerManagement

Warning: doxygenclass: Cannot find class "mono::power::IPowerManagement :project: monoapi" in doxygen xml output for project "monoapi" from directory: xml

2.6 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`**

2.7 Contribute

The source to this documentation is available publicly on our GitHub. Please just fork it, correct all our *bad engrish* - 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`.

2.8 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.