IOT: EMBEDDED APPLICATIONS PROTOTYPING WITH ESP8266 AND ARDUINO FRAMEWORK

SHOWCASE AT BERN SILICON VALLEY IEVS HACKLAB GROUP

AGENDA

- How it started
- Timer – the Enabler for complex Projects
- Debug Features needed
- Several Projects with common Functions – Skeleton Application

HOW IT STARTED

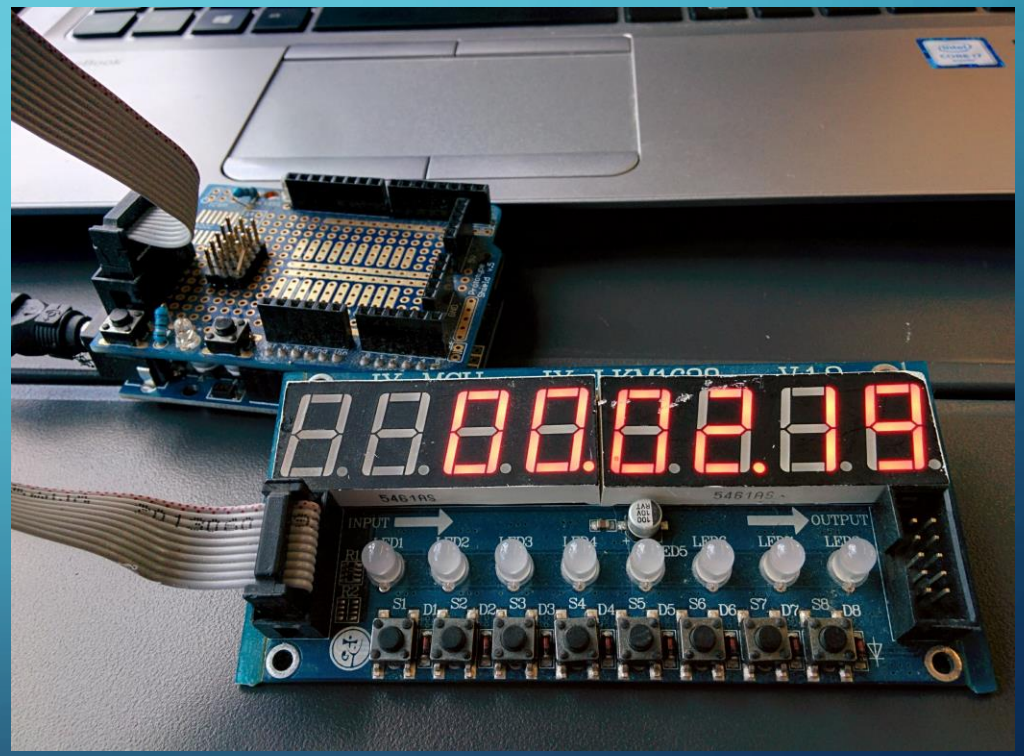
TIMER – THE ENABLER FOR COMPLEX PROJECTS

DEBUG FEATURES NEEDED

SEVERAL PROJECTS WITH COMMON FUNCTIONS – SKELETON APPLICATION

HISTORY

COUNTDOWN WATCH



CREATE A TIMING USING THE ARDUINO DELAY() FUNCTION

- ✓ Simple to use
- ✓ Straight forward
- ❖ nothing else can be executed in the meanwhile

```
// the setup function runs once when you press reset or power the board
void setup()
{
  // initialize digital pin LED_BUILTIN as an output.
  pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop()
{
  digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);                       // wait for a second
  digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);                       // wait for a second
}
```

CREATE A TIMING W/O USING ARDUINO DELAY()

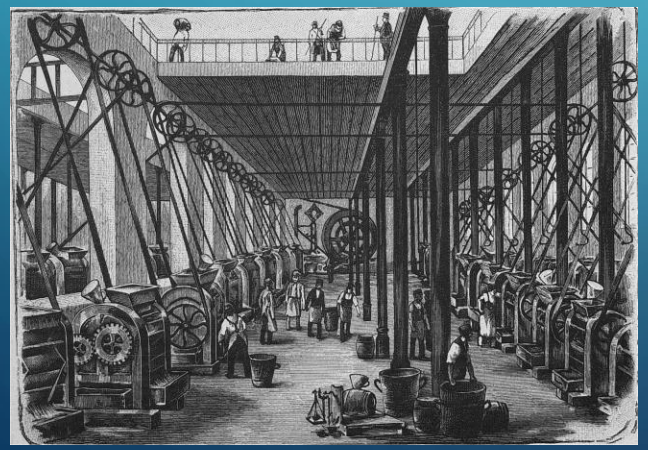
- ✓ Add multiple `if (currentMillis - previousMillis >= interval) { }` blocks for any task having a different interval
- ❖ Brings a lot of Spaghetti code into the loop() function
- ❖ Result: huge loop() function

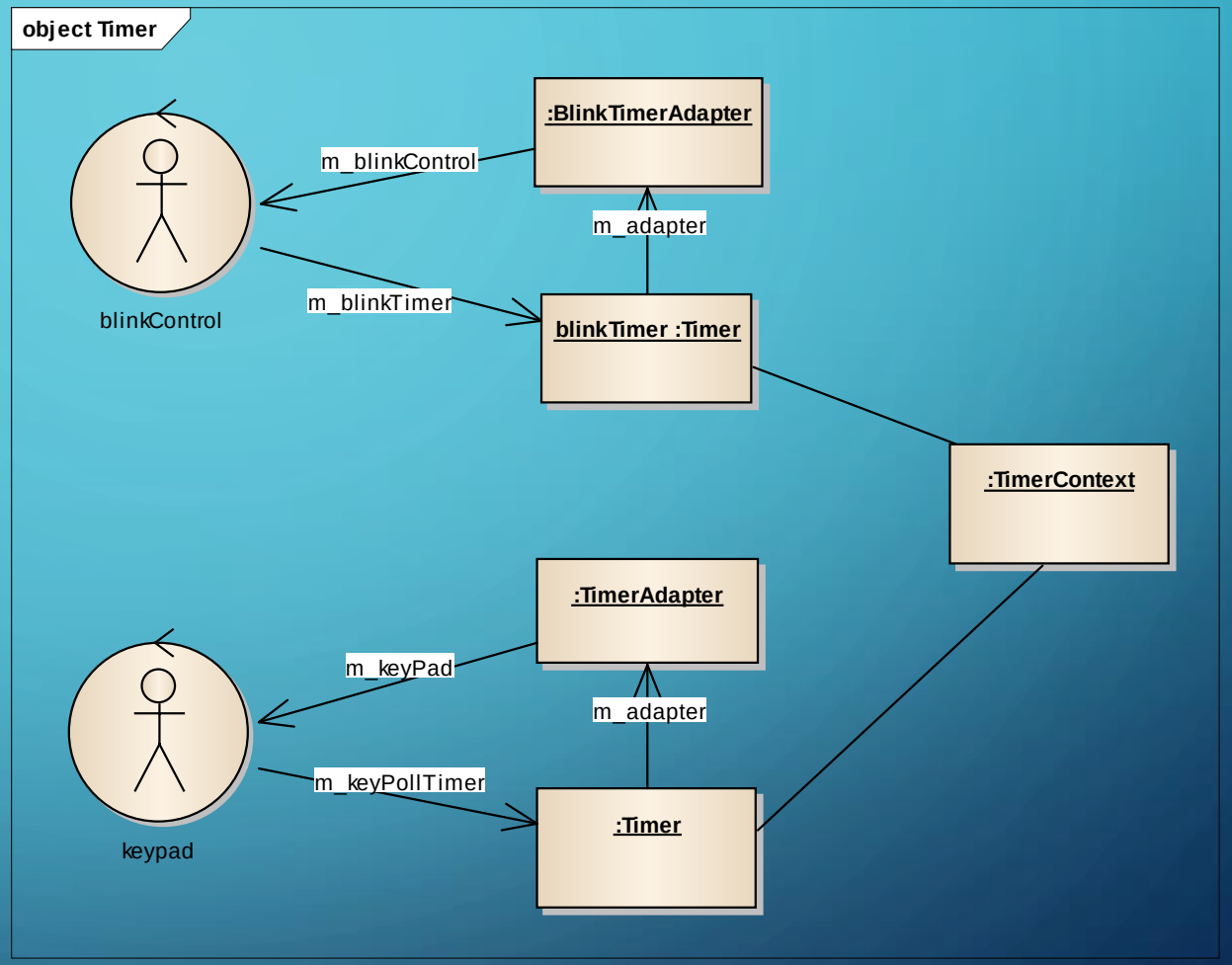
```ino
// constants won't change. Used here to set a pin number :
const int ledPin =  LED_BUILTIN;// the number of the LED pin

// Variables will change :
int ledState = LOW;             // ledState used to set the LED

// Generally, you should use "unsigned long" for variables that hold time
// The value will quickly become too large for an int to store
unsigned long previousMillis = 0;        // will store last time LED was updated

// constants won't change :
const long interval = 1000;           // interval at which to blink (milliseconds)

void setup() {
  // set the digital pin as output:
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // here is where you'd put code that needs to be running all the time.

  // check to see if it's time to blink the LED; that is, if the
  // difference between the current time and last time you blinked
  // the LED is bigger than the interval at which you want to
  // blink the LED.
  unsigned long currentMillis = millis();

  if (currentMillis - previousMillis >= interval) {
    // save the last time you blinked the LED
    previousMillis = currentMillis;

    // if the LED is off turn it on and vice-versa:
    if (ledState == LOW) {
      ledState = HIGH;
    } else {
      ledState = LOW;
    }

    // set the LED with the ledState of the variable:
    digitalWrite(ledPin, ledState);
  }
}
```

https://www.arduino.cc/en/Tutorial/BlinkWithoutDelay

VISION: ENCAPSULATE THE TIMING ASPECT

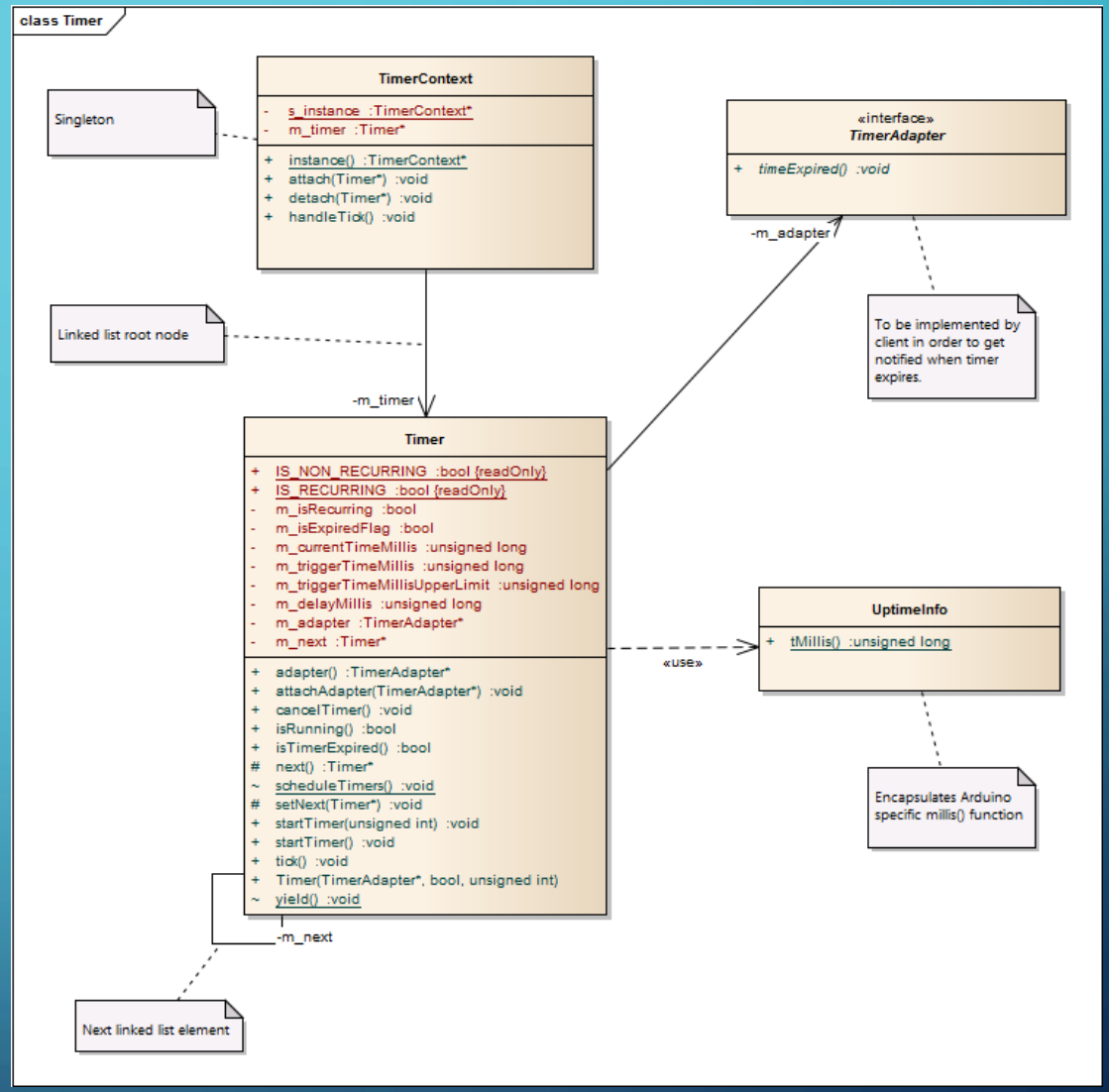
- I'd like to create different independent components
- Each of these components shall be responsible for its own timing aspects
- Similar to a power transmission system the TimerContext drives all Timer objects evaluating their due times autonomously





SOLUTION: TIMER

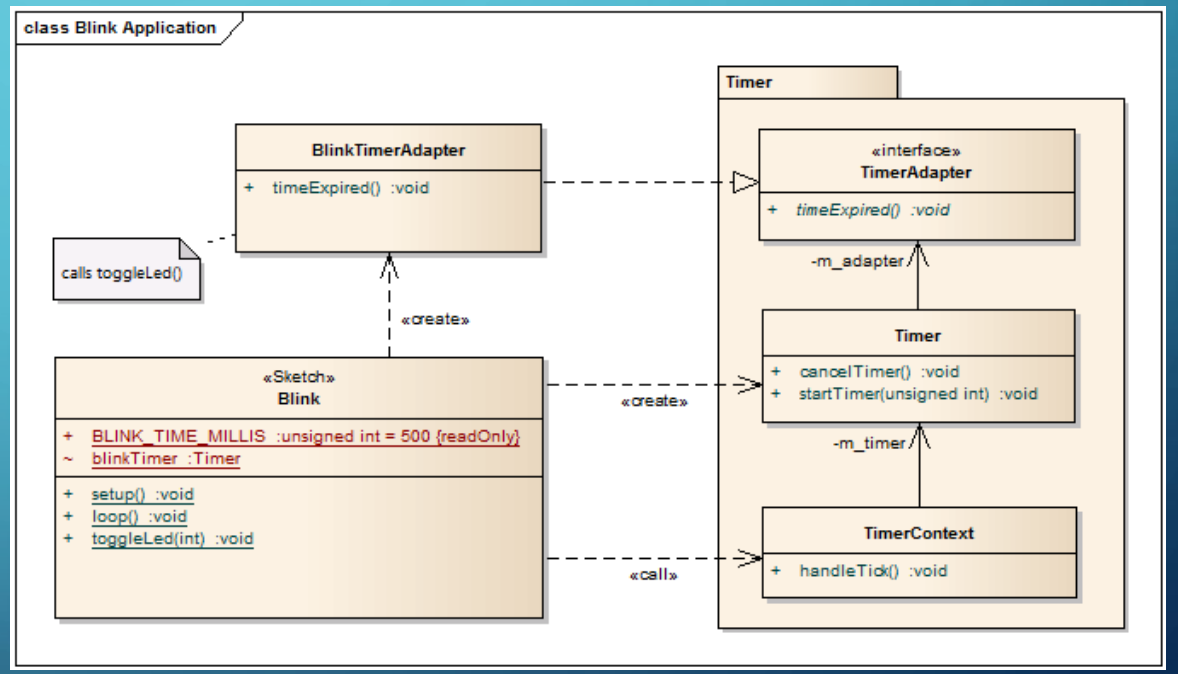
- **Timer**
  - On creation
    - Have to receive a TimerAdapter implementation as an injected object
    - attaches itself to TimerContext
  - Multiple objects are organized in a single linked list
- **TimerContext**
  - Is a Sigleton, instance always available from everywhere
  - The «Transmission Wheel» is the handleTick() method which shall be called by the Arduino loop() function
  - handleTick() then calls tick() on any attached Timer object



https://github.com/dniklaus/arduino-utils-timer

EXAMPLE: BLINK WITH TIMER

```ino
#include "Timer.h"

void toggleLed(int ledPin)
{
  bool isLedOn = digitalRead(ledPin);
  digitalWrite(ledPin, !isLedOn);
}

const unsigned int  BLINK_TIME_MILLIS = 200;
class BlinkTimerAdapter : public TimerAdapter
{
public:
  void timeExpired()
  {
    toggleLed(LED_BUILTIN);
  }
};

//The setup function is called once at startup of the sketch
void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
  new Timer(new BlinkTimerAdapter(), Timer::IS_RECURRING, BLINK_TIME_MILLIS);
}

// The loop function is called in an endless loop
void loop()
{
  yield();
}
```



HOW IT STARTED

TIMER – THE ENABLER FOR COMPLEX PROJECTS

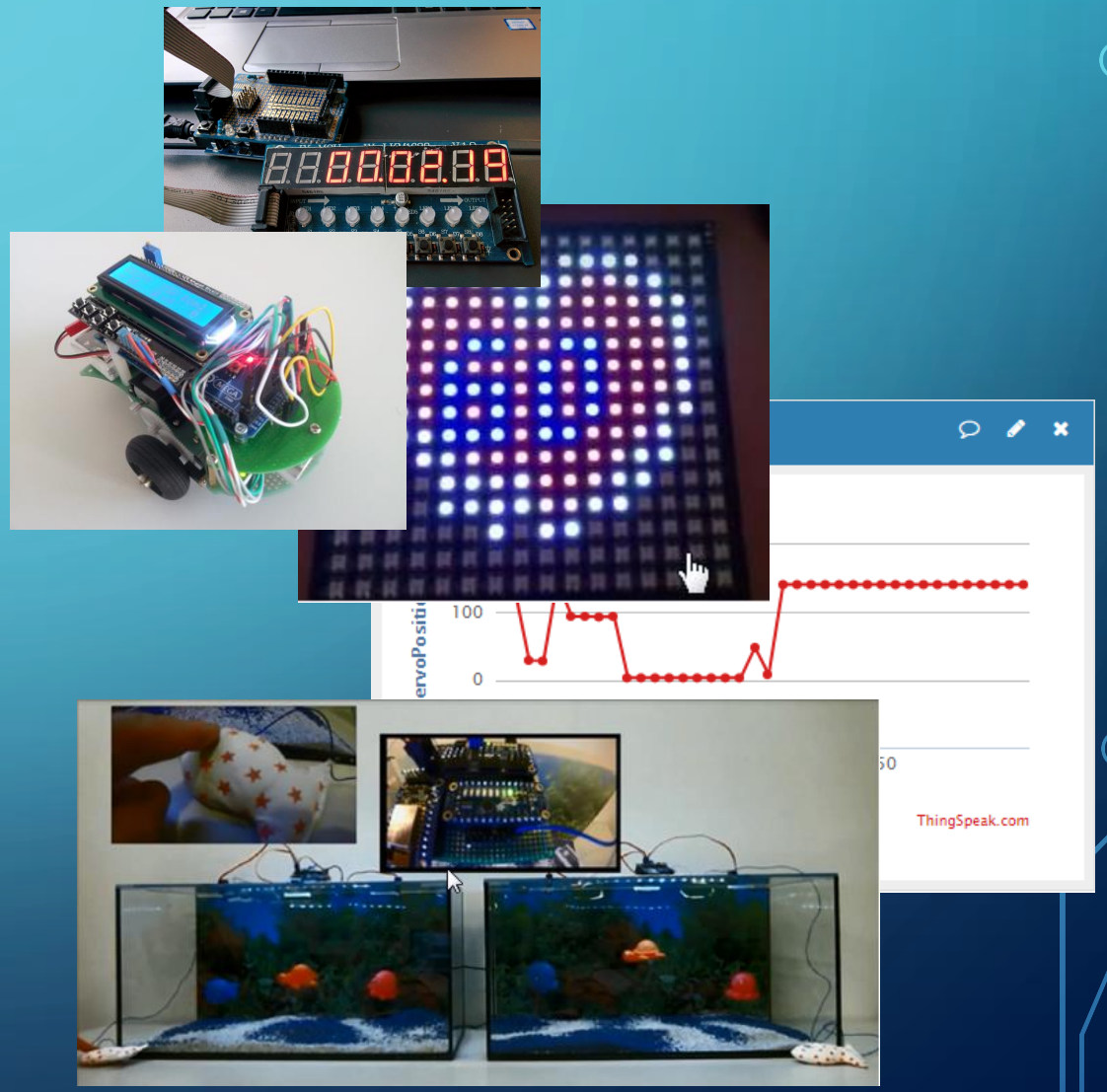
DEBUG FEATURES NEEDED

SEVERAL PROJECTS WITH COMMON FUNCTIONS – SKELETON APPLICATION

EASY PROTOTYPING BASED ON THE ARDUINO FRAMEWORK

TIMER – ENABLER FOR MANY COMPLEX PROJECTS

- Countdown Watch
- Lintilla Robot (WiFi, RestAPI)
- Memphis: Heart rate monitor with animated T-Shirt (HBR sent to ThingSpeak over WiFi)
- IoF - Internet of Fish (connected devices, WiFi, MQTT)



HOW IT STARTED

TIMER – THE ENABLER FOR COMPLEX PROJECTS

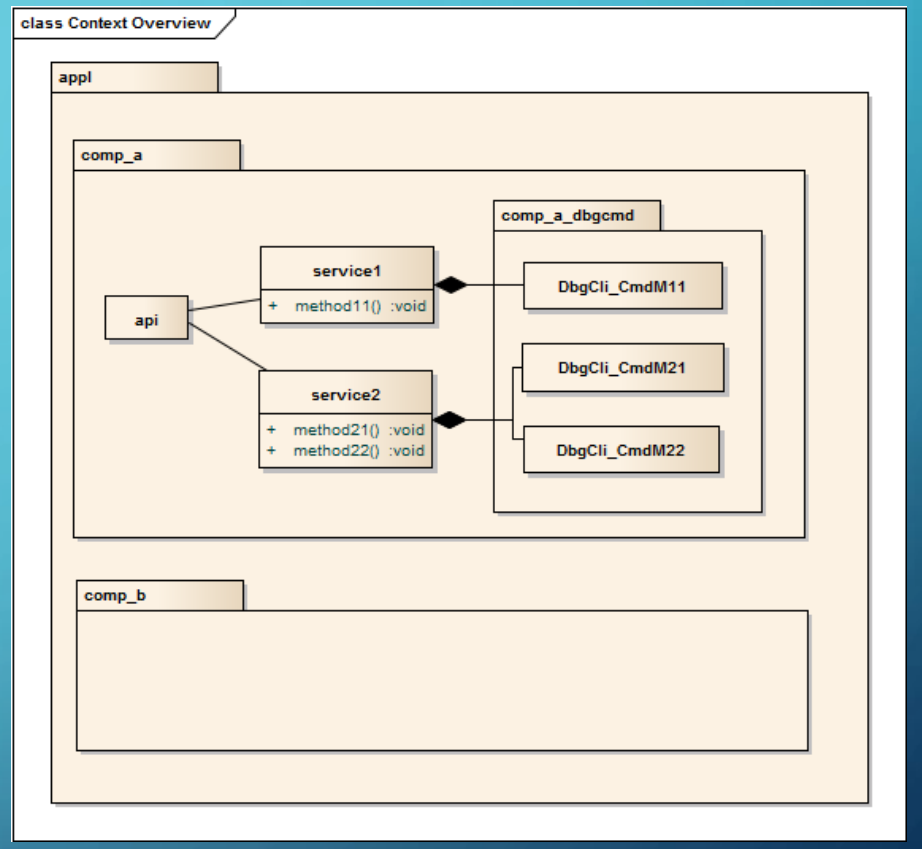
DEBUG FEATURES NEEDED

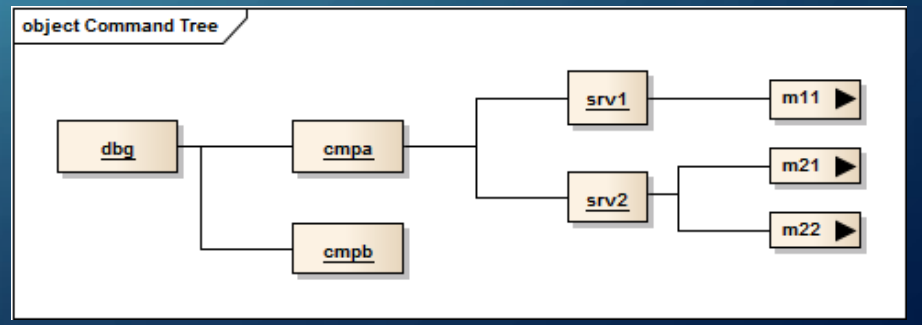
SEVERAL PROJECTS WITH COMMON FUNCTIONS – SKELETON APPLICATION

DEBUG CLI & TRACE COMPONENTS

DEBUG FEATURES – DEBUG CLI

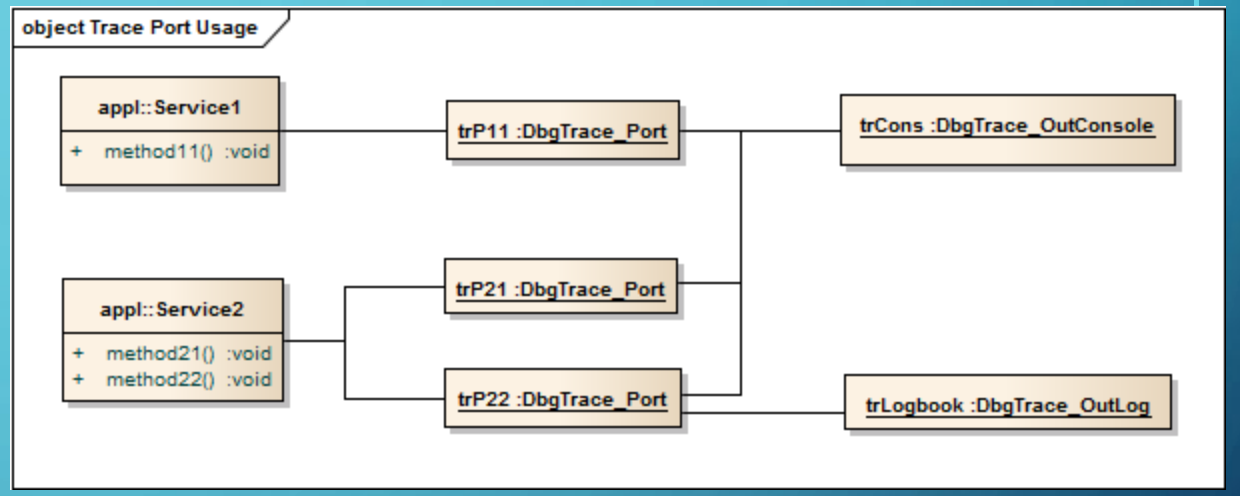
- Functional Features
  - execute commands entered at a serial console
  - run specific methods and functions of components within an embedded application
  - enable to perform automated module integration tests
- Design Aspects
  - represented as classes that can be attached to any component of an embedded application
  - Build a tree of command tokens





DEBUG FEATURES – DEBUG TRACE

- Functional Features
  - print out debug and trace log messages trough trace port objects
  - support different output channels, such as serial console interface or to a log file stored on a compact flash card
  - trace messages can be filtered according to adjustable trace log levels
- Design Aspects
  - Debug Trace Ports are represented as objects that can be attached to any component of an embedded application
  - any trace port instance brings the ability to control the filter level using the Debug CLI




```
typedef enum
{
  none      =  -1,
  emergency =   0,
  alert     =   1,
  critical  =   2,
  error     =   3,
  warning   =   4,
  notice    =   5,
  info      =   6,
  debug     =   7
} DbgTraceLevel;
```

HOW IT STARTED

TIMER – THE ENABLER FOR COMPLEX PROJECTS

DEBUG FEATURES NEEDED

SEVERAL PROJECTS WITH COMMON FUNCTIONS – SKELETON APPLICATION

PLATFORMIO.ORG PROJECT WITH ALL THE LIBRARIES, BIOLERPLATE SETUP CODE AND INTERACTIVE FEATURES

IOT SKELETON APPLICATION

- builds up an Arduino Framework based IoT application skeleton
- contains several components helping with debugging and integrating embedded applications on ESP8266 based controller modules
- is based on the PlatformIO.org toolchain and thus provide more flexibility to the professional developer (IDE selectable according to personal preferences)
- will accelerate the development of new applications for devices

https://github.com/ERNICommunity/wiring-iot-skeleton

IOT SKELETON APPLICATION - COMPONENTS

- ***Timer***: configurable recurring or non-recurring timer to schedule events. This component enables to improve your application's architecture by encapsulating the timing functionality into your components and thus make them active
- ***Ramutils***: helps to determine the RAM that is currently available
- ***DbgTrace***: debug trace log environment with mutable log levels during run time
- ***DbgCLI***: interactive console environment with command tree that can be built up decentralized (from any different location in your application code and within any component)
- ***App-Dbg***: boilerplate code setting up all the debug environment such as CLI and Tracing and RAM info printer
- ***MqttClient***: Mqtt Client wrapping around the 3rd party ***PubSubClient*** library, monitoring the LAN and the connection to the MQTT broker, able to automatically re-connect on connection loss, providing auto publish for selectable topics and auto subscribe for all registered topic subscriptions on re-connection, supports multiple subscriptions also with wildcards in the topic path

THANK YOU!

Dieter Niklaus

Embedded SW Engineer

ERNI Consulting AG, Bern

dieter.niklaus@gmx.net

https://github.com/dniklaus