


# Practical Legato

A quick start guide to Legato development



# Interfacing the Legato Way

Use the mangOH Red and Renfell GPIO  
Lab IoT card to build a complex blinky using  
timers and GPIO



## Interfacing, the Legato Way

**Legato is a framework that provides many APIs to ease hardware interfacing**

- **Access to Modem, GNSS, GPIO and system components all provided via APIs which provide fine grained control over hardware**
- **Many support libraries also provided – file directory operations, mutexes, timers, list handlers**
- **Worth learning the Legato APIs and libraries as they wrap up some things that are difficult or complex to in ‘pure’ linux.**
- **Legato framework is Event Driven, so most APIs follow a ‘register handler’, ‘wait for event’ style of programming**

# Timers

- **Legato has native support for timers**
- **A Timer can be one shot or repeat 'n' times or free-running**
- **Limit of 32 timers per application**
- **Event driven - Use a handler function called when timer times out**
- **Can be started, stopped, restarted or deleted**
- **Period range from mS to hours (or longer)**
- **Individual timers referenced by 'handle'**

# Sample Timer Configuration

## Timer setup

- **Timer Handle variable**
- **Create new timer**
- **Add the event handler**
- **Set the interval**
- **Configure repeat**
- **Start the timer**
- **Event Handler Definition**
- **Do something when the timer fires**

```
le_timer_Ref_t BlinkTimer = NULL;

BlinkTimer = le_timer_Create("BlinkTimer");

le_timer_SetHandler ( BlinkTimer,
                      BlinkTimerHandler );

le_timer_SetMsInterval( Gpio1BlinkTimer, 500 );

le_timer_SetRepeat( Gpio1BlinkTimer, 0);

le_timer_Start( Gpio1BlinkTimer );

void BlinkTimerHandler(le_timer_Ref_t pTimerRef )
{
    static uint8_t state = 0;

    if (state) { state = 0; }
    else      { state = 1; }

    LE_INFO( "state= [%s]", ((state)?"HI":"LO") );

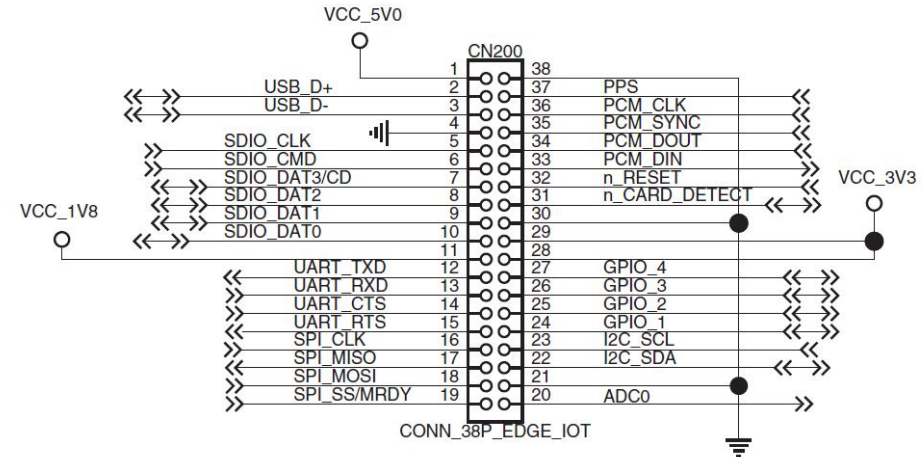
    return;
}
```

# IoT connector

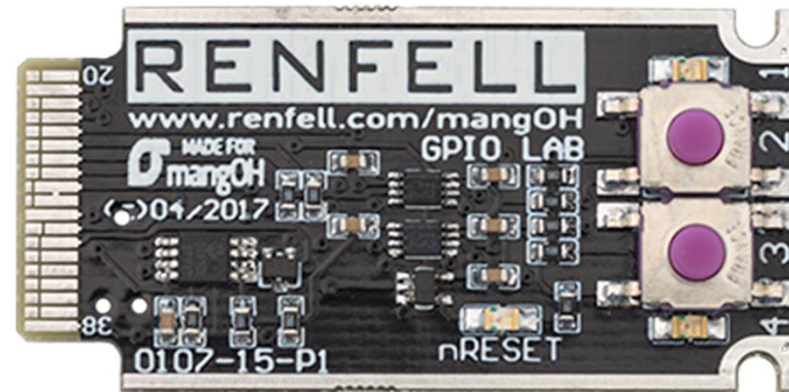
Common hardware interface between mangOH and FX30.

Useful I/O:

- GPIO
- SPI
- I2C
- UART
- USB
- others



All I/O is at 1V8 so interface hardware is required to talk to real world.



# GPIO on IoT Connector

- Each GPIO can be configured as an Input or an Output
- Inputs can have internal pull-up OR pull-down resistors enabled, or neither
- Inputs can have an event handler attached to react to input changes
- Outputs can be push-pull (drive to +V or GND) or open drain (drive to GND only) or Tri-State (output state ignored – used when sharing GPIO with other active devices)
- Outputs have polarity active HI (+V when on and 0 when off) or active LO (0V when on and +V when off)
- GPIO I/O is managed by the Legato GPIO service – the user doesn't directly have to manipulate hardware registers

# GPIO Input events

- Can add an 'Input Event Change Handler' to be activated when the input changes state (interrupt)
- Event can be triggered on Rising Edge, Falling edge or both edges
- Input change works even if underlying GPIO hardware doesn't support interrupt on change
- If the GPIO doesn't support interrupt the pin will be scanned by a timer every n mS and the interrupt 'emulated' by the GPIO service
- Input Event Change Handler should be short and as minimal as possible.



# Linking Components to Services

- **A component can publish functionality for other components to use – so it will become a server**
- **Most system interfaces are provided via Services (GPIO, GPS positioning etc)**
- **A component attaches to a service by listing the service as a ‘requirement’ in the requires: stanza in Component .cdef**
- **An application links the component to the service by adding bindings in the application.adeb file**

## Component.cdef requires: api:

- External services are listed as APIs in the `requires: stanza` in `Component.cdef`
- APIs can be renamed to make their usage clearer in your code

```
requires:
{
  api:
  {
    le_cfg.api
  }
}

requires:
{
  api:
  {
    IoT_RESET = le_gpio.api
  }
}
```

# Application.adev bindings:

- The APIs required by a component are bound to the appropriate service in the .adev bindings: stanza
- APIs are bound on a process.component.api basis
- If there is more than one process then there may be multiple entries for the same target service

```
bindings:
{
}

bindings:
{
  proc.comp.le_avc -> avcService.le_avc
}

bindings:
{
  proc1.comp1.le_avc -> avcService.le_avc
  proc2.comp2.le_avc -> avcService.le_avc
}
```

# GPIO bindings

- Many GPIOs use the same `le_gpio.api` API
- Each GPIO must be explicitly named in `Component.cdef`
- Each renamed GPIO must be mapped to the required hardware pin in the `adef bindings: stanza`

## **IMPORTANT:**

**renaming an API will ALSO rename all the functions and constants available in the API**

## Example: A complex blinky

- Using Renfell GPIO Lab card – 2 digital out indicators, 2 digital in switches (one pulled high, one pulled low)
- Use a repeating timer to flash GPIO1 at 500 ms on, 500mS off
- Use a timer to flash GPIO4 at value stored in config flash
- Use an input event on GPIO2 and GPIO3 input to catch button press events
- GPIO2 pressed, increase the flash rate of GPIO4
- GPIO3 pressed, decrease the flash rate of GPIO4
- Save GPIO4 flash rate into config flash when the application is stopped or terminated

# Demonstration

Demonstration