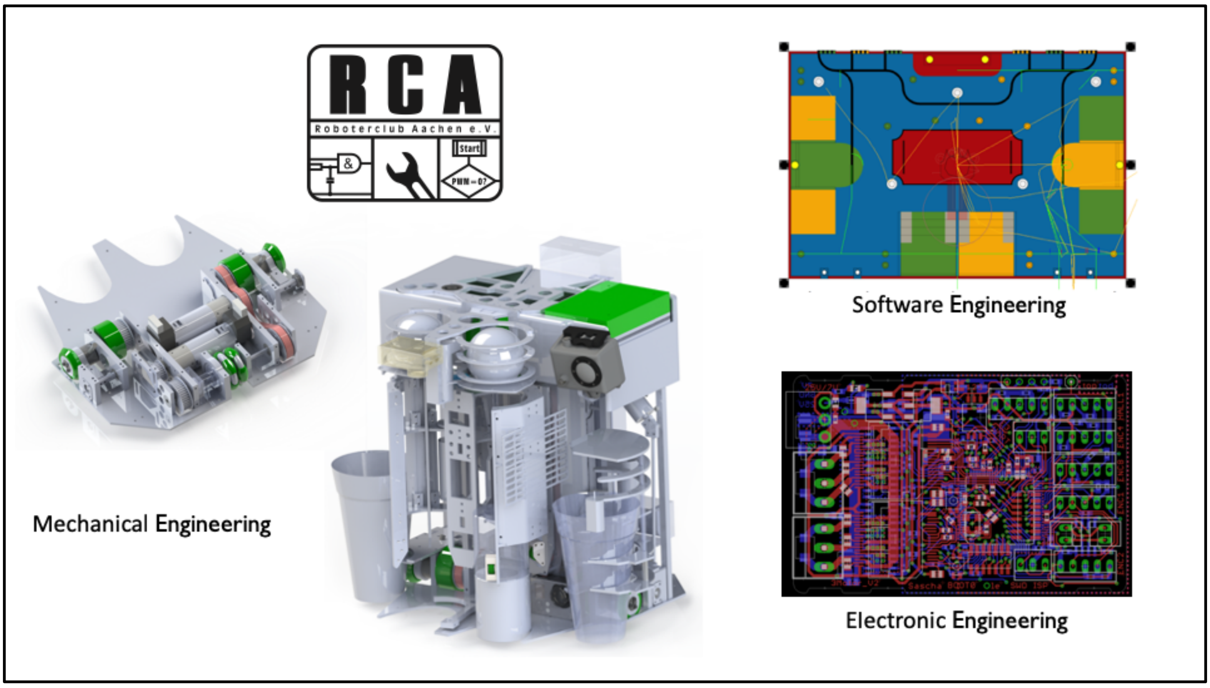
Modular Code Generation with lbuild in Python 3

Niklas Hauser

emBO++ 2019

RWTH AACHEN UNIVERSITY

I study something with computers.



But I also build autonomous robots in my spare time.

We use a lot of different microcontrollers, so we needed to port our HAL a lot, and then automated this process a little.
We call our HAL modm, and it supports 1000 AVR and STM32 targets.

What wrong with the C preprocessor?

```
#if TARGET_FAMILY == "f1"
    data = USART3->DR;
#else
    data = USART3->RDR;
#endif
```

You're all already using a code generator, it's called the C preprocessor.
It works pretty well FOR LIMITED USE-CASES.

Difficult to get non-language related data into the CPP, and only supports *very restrictive* operations on it.

Just use Python

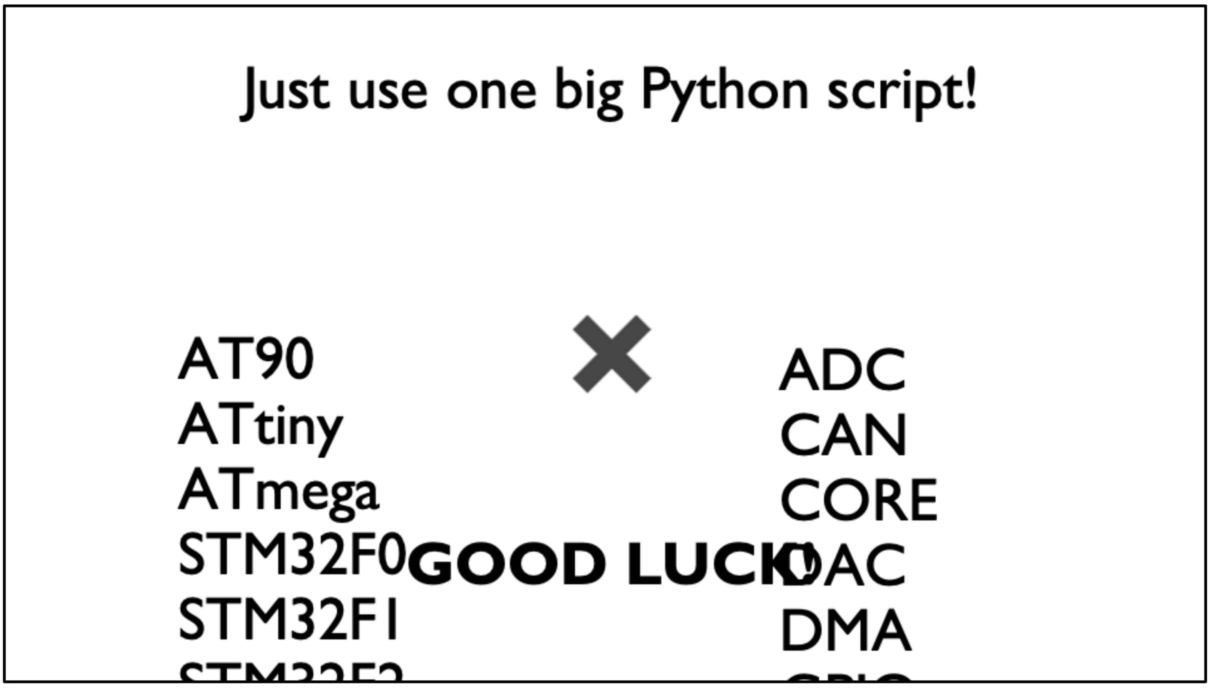
```
%% if target.family == "f1"
    data = USART3->DR;
%% else
    data = USART3->RDR;
%% endif

     stm32 f1 03 r b t
```

Let's replace the CPP with the Jinja template engine called from Python.
The immediate difference isn't that much, but you can access any Python object in Jinja, so we've replace the macro TARGET_FAMILY with an OBJECT that gives us access to the target identifier in a structured way.

Just use Python already

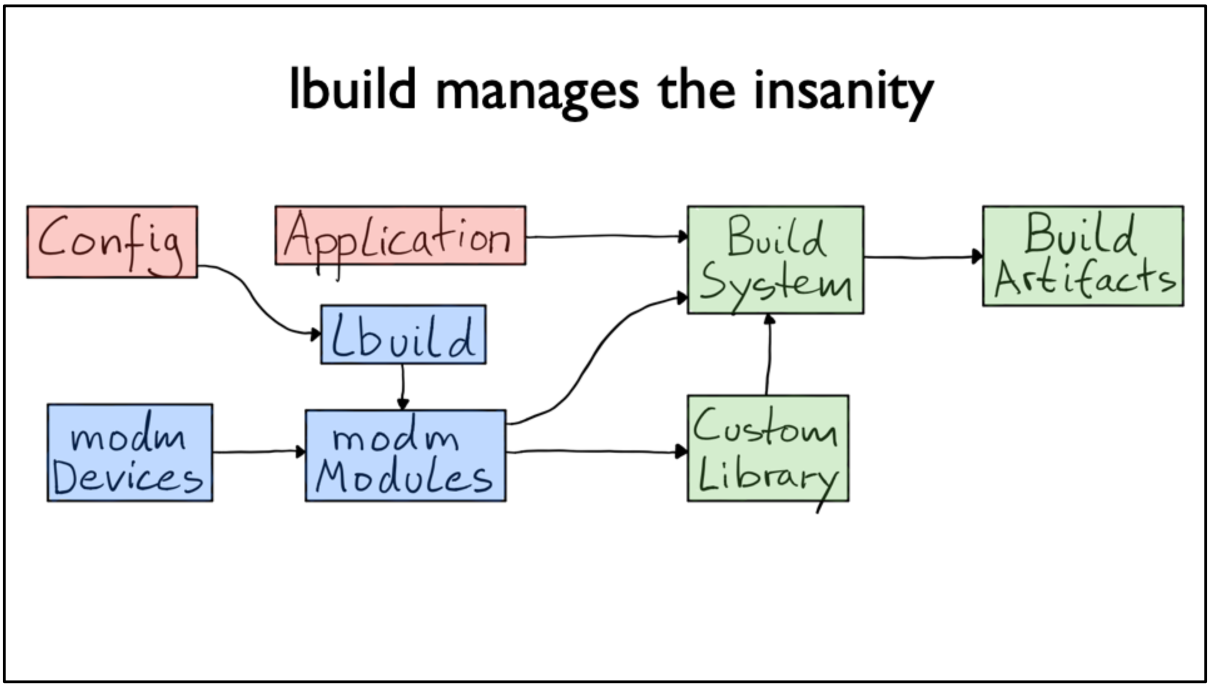
```
uint32_t vector_table[] = {
    __stack_top,
    Reset_Handler,

%% for irq in irqs
    {{ irq }}_IRQHandler,
%% endfor
};
```

For example you can pass a list of interrupt vectors into your template and format your interrupt vector table using this information.



Once doesn’t simply generate a HAL for 1000 AVR and STM32 targets.

This wouldn’t be manageable as a huge Python script, so we broke it down into smaller modules and thus lbuild was born.



Thus lbuild helps us build our library, it's a library builder, hence lbuild.

modm uses the data in modm-devices to generate a custom library and build system for your target.

```
 $ lbuild -r repo.lb discover
Parser(lbuild)
└── Repository(modm @ .)   modm: a barebone embedded library generator
    ├── EnumerationOption(target) = REQUIRED in [at90can128, at90can32, at90can
    ├── Configuration(modm:al-avreb-can)   AL-AVREB_CAN Board
    ├── Configuration(modm:arduino-uno)   Arduino UNO
    ├── Configuration(modm:black-pill)   Black Pill
    ├── Configuration(modm:blue-pill)   Blue Pill
    ├── Configuration(modm:disco-f051r8)   STM32F0DISCOVERY
    ├── Configuration(modm:disco-f072rb)   STM32F072DISCOVERY
    ├── Configuration(modm:disco-f100rb)   STM32VLDISCOVERY
    ├── Configuration(modm:disco-f303vc)   STM32F3DISCOVERY
    ├── Configuration(modm:disco-f407vg)   STM32F4DISCOVERY
    ├── Configuration(modm:disco-f429zi)   STM32F429IDISCOVERY
    ├── Configuration(modm:disco-f469ni)   STM32F469IDISCOVERY
    ├── Configuration(modm:disco-f746ng)   STM32F7DISCOVERY
    ├── Configuration(modm:disco-f769ni)   STM32F769IDISCOVERY
    ├── Configuration(modm:disco-l476vg)   STM32L476DISCOVERY
    ├── Configuration(modm:nucleo-f031k6)   NUCLEO-F031K6
    ├── Configuration(modm:nucleo-f042k6)   NUCLEO-F042K6
    ├── Configuration(modm:nucleo-f103rb)   NUCLEO-F103RB
    ├── Configuration(modm:nucleo-f303k8)   NUCLEO-F303K8
```

So how does it look like?

lbuild operates on repositories and modules, here we can see the discovered modm repository.
You can very prominently see the modm:target option, which is REQUIRED to see the repository.
Below are some predefined library configurations that you can inherit for well-known development boards.

```
 $ lbuild -r repo.lb discover modm:target
>> modm:target  [EnumerationOption]

Meta-HAL target device

Value: REQUIRED
Inputs: [at90can128, at90can32, at90can64, at90pwm1, at90pwm161, at90pwm2,
         at90pwm216, at90pwm3, at90pwm316, at90pwm81, at90usb1286, at90usb1287,
         at90usb162, at90usb646, at90usb647, at90usb82, atmega128, atmega1280,
         atmega1281, atmega1284, atmega1284p, atmega1284rfr2, atmega128a,
         atmega128rfa1, atmega128rfr2, atmega16, atmega162, atmega164a,
         atmega164p, atmega164pa, atmega165a, atmega165p, atmega165pa,
         atmega168, atmega168a, atmega168p, atmega168pa, atmega168pb,
         atmega169a, atmega169p, atmega169pa, atmega16a, atmega16hva,
         atmega16hvb, atmega16hvbrevb, atmega16m1, atmega16u2, atmega16u4,
         atmega2560, atmega2561, atmega2564rfr2, atmega256rfr2, atmega32,
         atmega324a, atmega324p, atmega324pa, atmega324pb, atmega325,
         atmega3250, atmega3250a, atmega3250p, atmega3250pa, atmega325a,
         atmega325p, atmega325pa, atmega328, atmega328p, atmega328pb,
         atmega329, atmega3290, atmega3290a, atmega3290p, atmega3290pa,
         atmega329a, atmega329p, atmega329pa, atmega32a, atmega32c1,
         atmega32hvb, atmega32hvbrevb, atmega32m1, atmega32u2, atmega32u4,
```

Let's discover the modm:target option: it's an enumeration option with a huge list of targets. These are all the microcontrollers that modm can generate a HAL for.

I'm going to choose the STM32F469 target

```
 $ lbuild -r repo.lb -D :target=stm32f469nih discover -t :platform
Module(modm:platform)   Platform HAL
├── Module(modm:platform:1-wire.bitbang)   Software 1-Wire
├── Module(modm:platform:adc)   Analog-to-Digital Converter (ADC)
│   ├── Module(modm:platform:adc:1)   Instance 1
│   ├── Module(modm:platform:adc:2)   Instance 2
│   └── Module(modm:platform:adc:3)   Instance 3
├── Module(modm:platform:can)   Controller Area Network (CAN)
│   ├── Module(modm:platform:can:1)   Instance 1
│   │   ├── NumericOption(buffer.rx) = 32 in [1 .. 32 .. 65534]
│   │   └── NumericOption(buffer.tx) = 32 in [1 .. 32 .. 65534]
│   └── Module(modm:platform:can:2)   Instance 2
│       ├── NumericOption(buffer.rx) = 32 in [1 .. 32 .. 65534]
│       └── NumericOption(buffer.tx) = 32 in [1 .. 32 .. 65534]
├── Module(modm:platform:can.common)   CAN Common
├── Module(modm:platform:clock)   System Clock
├── Module(modm:platform:core)   STM32 core module
├── Module(modm:platform:cortex-m)   ARM Cortex-M Core
│   ├── EnumerationOption(allocator) = newlib in [block, newlib, tlsf]   Dy ...
│   ├── NumericOption(linkerscript.flash_offset) = 0 in [0 ... 2097152]   A ...
│   ├── NumericOption(main_stack_size) = 3040 in [256 .. 3040 .. 65536]   M ...
│   ├── BooleanOption(stack_execution_guard) = False in [True, False]   Fil ...
```

When I run discover again with this repository option, we can then finally see the modules.

Here you see a selection of the microcontrollers peripherals, as hierarchical modules. We have very fine-grained modules, and split up each instance of each peripheral into its own module.

This reduces the amount of code that falls out of modm at the end, which can lead to *very small* binaries.

Note the allocator option in the platform:cortex-m module. It allows us to change the entire heap allocator with an option.

```xml
<library>
  <repositories>
    <repository>modm/repo.lb</repository>
  </repositories>
  <options>
    <option name="modm:target">stm32f469nih</option>
    <option name="modm:platform:uart:3:buffer.tx">2048</option>
    <option name="modm:platform:cortex-m:allocator">tlsf</option>
  </options>
  <modules>
    <module>modm:platform:core</module>
    <module>modm:platform:gpio</module>
    <module>modm:platform:rcc</module>
    <module>modm:platform:uart:3</module>
    <module>modm:platform:timer:1</module>
    <module>modm:ui:animation</module>
    <module>modm:ui:led</module>
  </modules>
```

This is how you specify the options and modules you want to use.
It’s a XML config file, it’s pretty straight-forward.

```python
def init(module):
    module.name = ":ui:led"
    module.description = FileReader("module.md")
```

How does a module work?

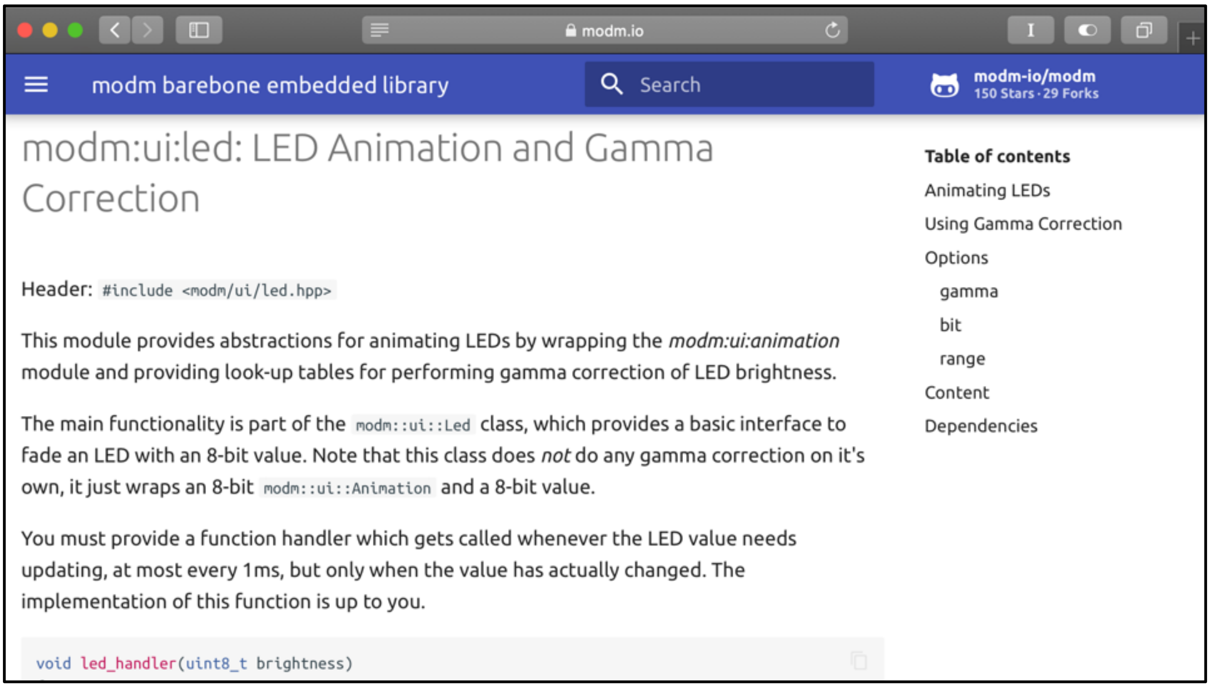
Add a bunch of python files to your repository with three functions: init, prepare, build.

Here the module name and module description is set.

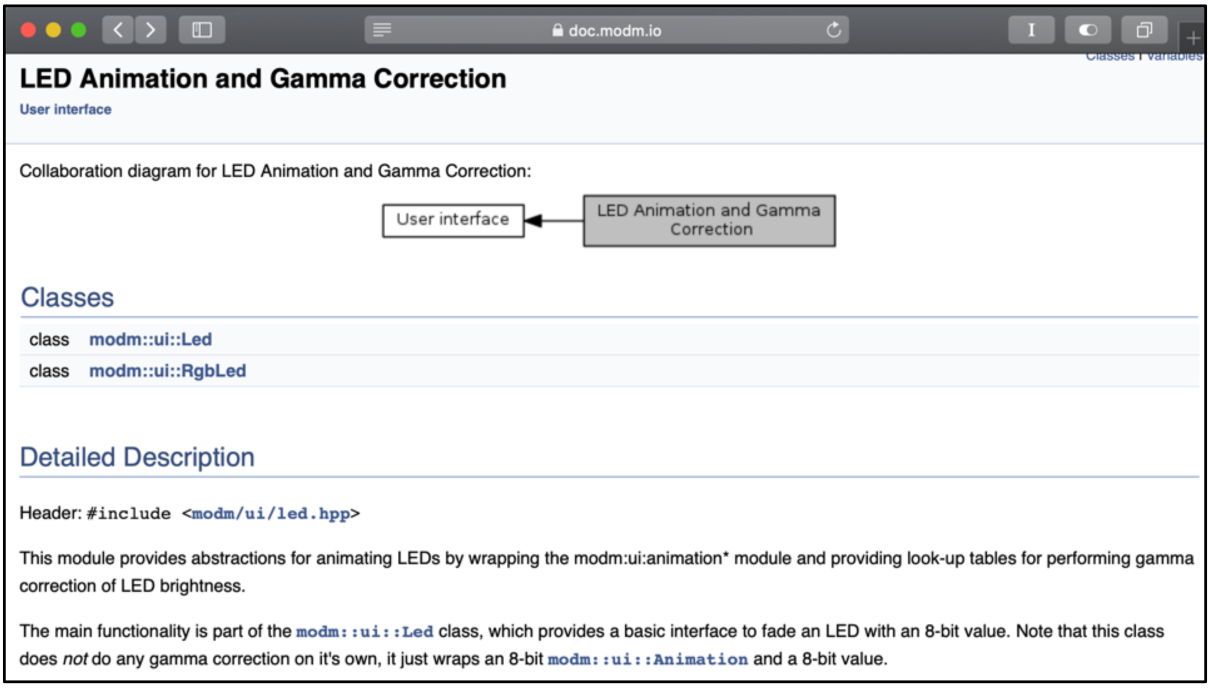
```
 $ lbuild discover :ui:led
>> modm:ui:led  [Module]

# LED Animation and Gamma Correction

Header: `#include <modm/ui/led.hpp>`

This module provides abstractions for animating LEDs by wrapping the
*modm:ui:animation* module and providing look-up tables for performing
gamma correction of LED brightness.

The main functionality is part of the `modm::ui::Led` class, which provides
a basic interface to fade an LED with an 8-bit value.
Note that this class does *not* do any gamma correction on it's own, it just
wraps an 8-bit `modm::ui::Animation` and a 8-bit value.

You must provide a function handler which gets called whenever the LED value
needs updating, at most every 1ms, but only when the value has actually changed
.
The implementation of this function is up to you.
```

This then allows lbuild to build a module tree and display the module description.

Here on the command line.



But we can also use this information to place the module description on our website.


doc.modm.io

Classes | Variables

LED Animation and Gamma Correction

User interface

Collaboration diagram for LED Animation and Gamma Correction:

User interface ← LED Animation and Gamma Correction

Classes

class	modm::ui::Led
class	modm::ui::RgbLed

Detailed Description

Header: `#include <modm/ui/led.hpp>`

This module provides abstractions for animating LEDs by wrapping the modm:ui:animation* module and providing look-up tables for performing gamma correction of LED brightness.

The main functionality is part of the `modm::ui::Led` class, which provides a basic interface to fade an LED with an 8-bit value. Note that this class does *not* do any gamma correction on it's own, it just wraps an 8-bit `modm::ui::Animation` and a 8-bit value.

And also automatically generate Doxygen groups with this description, so everything is always in sync.

```
 $ lbuild search animation
Parser(lbuild)
╰── Repository(modm @ ../../..)   modm: a barebone embedded library generator
    ╰── Module(modm:ui)   User interface
        ├── Module(modm:ui:animation)   Animators
        ╰── Module(modm:ui:led)   LED Animation and Gamma Correction


>> modm:ui  [Module]

  4  Contains code for Graphics, Buttons, LEDs, Animations, Menu Structures.


>> modm:ui:animation  [Module]


>> modm:ui:led  [Module]

  1  # LED Animation and Gamma Correction
  6  *modm:ui:animation* module and providing look-up tables for performing
 12  wraps an 8-bit `modm::ui::Animation` and a 8-bit value.
 38  Depending on how smooth you require your animation to be, you may call the
```

You can then also search these descriptions which is useful when you have a lot of modules.

```python
def init(module):
    module.name = ":platform:cortex-m"
    module.description = FileReader("module.md")

def prepare(module, options):
    module.depends(":architecture:heap")
    module.add_option(
        EnumerationOption(name="allocator", default="newlib",
            enumeration=["newlib", "block", "tlsf"],
            description=FileReader("option/allocator.md")))
    return options[":target"].has_driver("core:cortex-m*")
```

The prepare step allows you to add dependencies on other modules.
Here the :platform:cortex-m module depends on the heap interface.

And you can add module options: here choose between three different allocators.

And finally, the modules can access the repository target, here we're only enabling this module if the target has a Cortex-M core.

```python
def build(env):
    env.substitutions = {"vectors": vectors}
    env.outbasepath = "modm/platform/core"
    env.template("vectors.c")

    allocator = env["allocator"]
    if allocator == "newlib":
        env.template("heap_newlib.c.in")
    elif allocator == "tlsf":
        env.template("heap_tlsf.c.in")
    elif allocator == "block":
        env.template("heap_block_allocator.cpp.in")
```

The user will then make their decision which modules to build, and with which options.

The build step then finally allows you to generate code: here we generate the vector table and the actual heap implementation.

This is nice because you can choose a completely different template FILE, so you don’t have to cram everything into one template.

```
FunctionPointer vectorsRom[] =
{
    (FunctionPointer)__main_stack_top,      // -16: stack pointer
    Reset_Handler,                          // -15: code entry point
    NMI_Handler,                            // -14: Non Maskable Interrupt handler
    HardFault_Handler,                      // -13: hard fault handler
    // ...
    SysTick_Handler,                        //  -1
    WWDG_IRQHandler,                        //   0
    PVD_IRQHandler,                         //   1
    TAMP_STAMP_IRQHandler,                  //   2
    RTC_WKUP_IRQHandler,                    //   3
    FLASH_IRQHandler,                       //   4
    RCC_IRQHandler,                         //   5
    EXTI0_IRQHandler,                       //   6
    EXTI1_IRQHandler,                       //   7
    EXTI2_IRQHandler,                       //   8
    EXTI3_IRQHandler,                       //   9
    EXTI4_IRQHandler,                       //  10
    DMA1_Stream0_IRQHandler,                //  11
    DMA1_Stream1_IRQHandler,                //  12
    DMA1_Stream2_IRQHandler,                //  13
    DMA1_Stream3_IRQHandler,                //  14
    DMA1_Stream4_IRQHandler,                //  15
    DMA1_Stream5_IRQHandler,                //  16
```

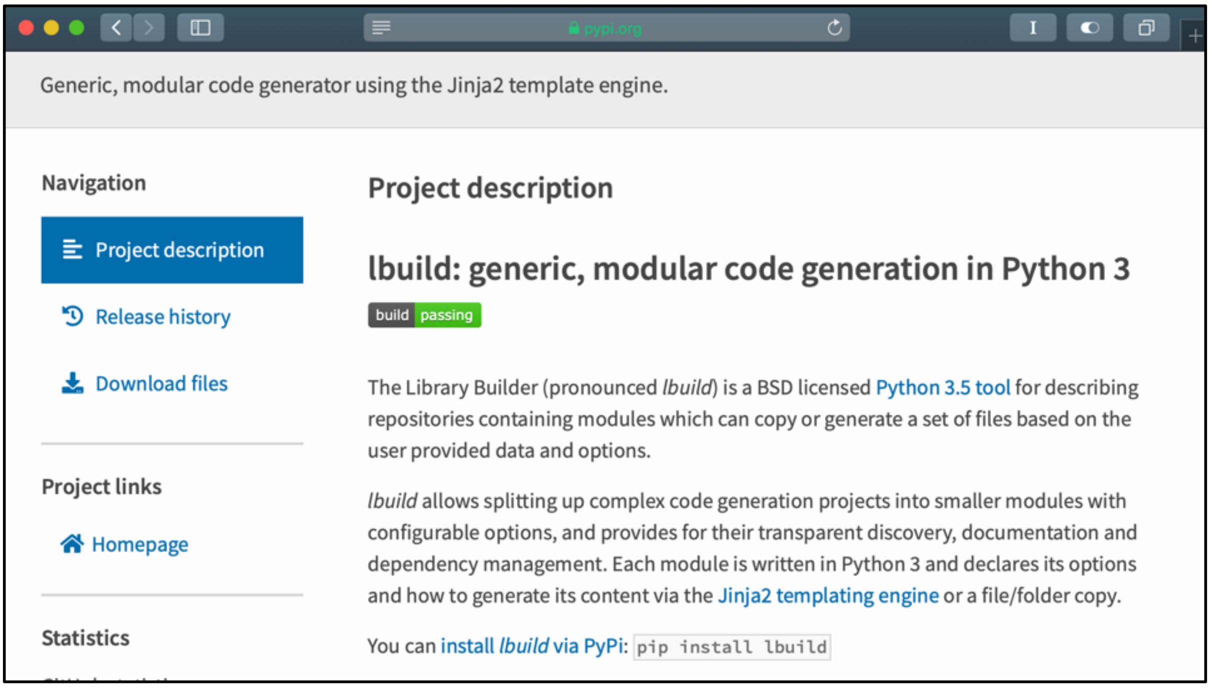
After you build, this is what the vector table looks like for our F469 target.

The names and positions come from modm-devices, not from lbuild.

```
 $ ls modm/src/modm/platform/gpio
base.hpp             gpio_C14.hpp         gpio_F0.hpp          gpio_H4.hpp
connector.hpp        gpio_C15.hpp         gpio_F1.hpp          gpio_H5.hpp
connector_detail.hpp gpio_C2.hpp          gpio_F10.hpp         gpio_H6.hpp
enable.cpp           gpio_C3.hpp          gpio_F11.hpp         gpio_H7.hpp
gpio_A0.hpp          gpio_C4.hpp          gpio_F12.hpp         gpio_H8.hpp
gpio_A1.hpp          gpio_C5.hpp          gpio_F13.hpp         gpio_H9.hpp
gpio_A10.hpp         gpio_C6.hpp          gpio_F14.hpp         gpio_I0.hpp
gpio_A11.hpp         gpio_C7.hpp          gpio_F15.hpp         gpio_I1.hpp
gpio_A12.hpp         gpio_C8.hpp          gpio_F2.hpp          gpio_I10.hpp
gpio_A13.hpp         gpio_C9.hpp          gpio_F3.hpp          gpio_I11.hpp
gpio_A14.hpp         gpio_D0.hpp          gpio_F4.hpp          gpio_I12.hpp
gpio_A15.hpp         gpio_D1.hpp          gpio_F5.hpp          gpio_I13.hpp
gpio_A2.hpp          gpio_D10.hpp         gpio_F6.hpp          gpio_I14.hpp
gpio_A3.hpp          gpio_D11.hpp         gpio_F7.hpp          gpio_I15.hpp
gpio_A4.hpp          gpio_D12.hpp         gpio_F8.hpp          gpio_I2.hpp
gpio_A5.hpp          gpio_D13.hpp         gpio_F9.hpp          gpio_I3.hpp
gpio_A6.hpp          gpio_D14.hpp         gpio_G0.hpp          gpio_I4.hpp
gpio_A7.hpp          gpio_D15.hpp         gpio_G1.hpp          gpio_I5.hpp
gpio_A8.hpp          gpio_D2.hpp          gpio_G10.hpp         gpio_I6.hpp
gpio_A9.hpp          gpio_D3.hpp          gpio_G11.hpp         gpio_I7.hpp
gpio_B0.hpp          gpio_D4.hpp          gpio_G12.hpp         gpio_I8.hpp
```

And here are all the files generated for the specific pins of your target.

Again this data comes from modm-devices, not from lbuild.



lbuild is available via Pip. It’s a Python 3.5 tool.

Just pip install lbuild.

Then open a bunch of issues on GitHub.

lbuild is part of the modm project

git clone --recursive
https://github.com/modm-io/modm.git

cd modm/examples/blue_pill/blink

lbuild discover

If you want to play around with lbuild:

Clone the modm repository (recursively), go into *any* example, and type lbuild discover.

Thank you for listening!

Niklas Hauser

salkinium.com
blog.salkinium.com
twitter.com/salkinium
github.com/salkinium

Want to know more about modm-devices, have a look at my blog.