ST MC SDK 5.x WB应用指南和固件详解

STM32电动机控制应用系列讲座之四

life.augmented

目录

- ST MC SDK5.x WB应用指南
  - 软件工具的下载和安装
  - ST MC Workbench及关键配置参数
  - MC Project 的生成，编译和下载
  - 电机控制及监控
- ST MC SDK5.x固件详解
  - 程序架构
  - 组件
  - 例程代码讲解
  - 开发实战
    - 如何向例程中添加外设和自己的代码
    - Step-by-Step添加一段闪灯代码

life.augmented

ST MC SDK5.x WB应用指南

- ST MC SDK5.x WB应用指南
  - 软件工具的下载和安装
  - ST MC Workbench及关键配置参数
  - MC Project 的生成，编译和下载
  - 电机控制及监控
- ST MC SDK5.x固件详解
  - 程序架构
  - 组件
  - 例程代码讲解
  - 开发实战
    - 如何向例程中添加外设和自己的代码
    - Step-by-Step添加一段闪灯代码

life.augmented

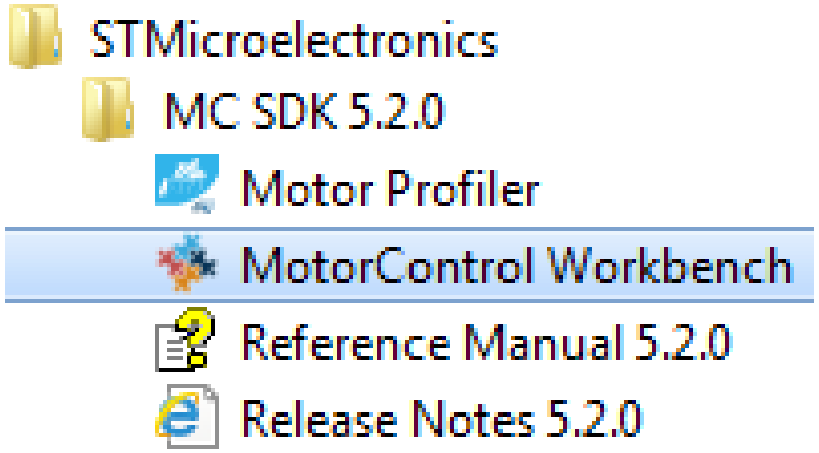
请预先安装下列PC软件工具:

- X-CUBE-MCSDK 或 X-CUBE-MCSDK-FUL
- STM32CubeMX (v4.25.1/v4.26.1)及固件库
- ST-LINK/V2 (v4.2.0)
- IDE:
  - IAR Embedded Workbench for Arm (v7.80.4/v8.20.2)
  - µVision® IDE for Arm® (Keil® MDK) v5.25
  - Atollic TrueSTUDIO for STM32 version 9.0.0

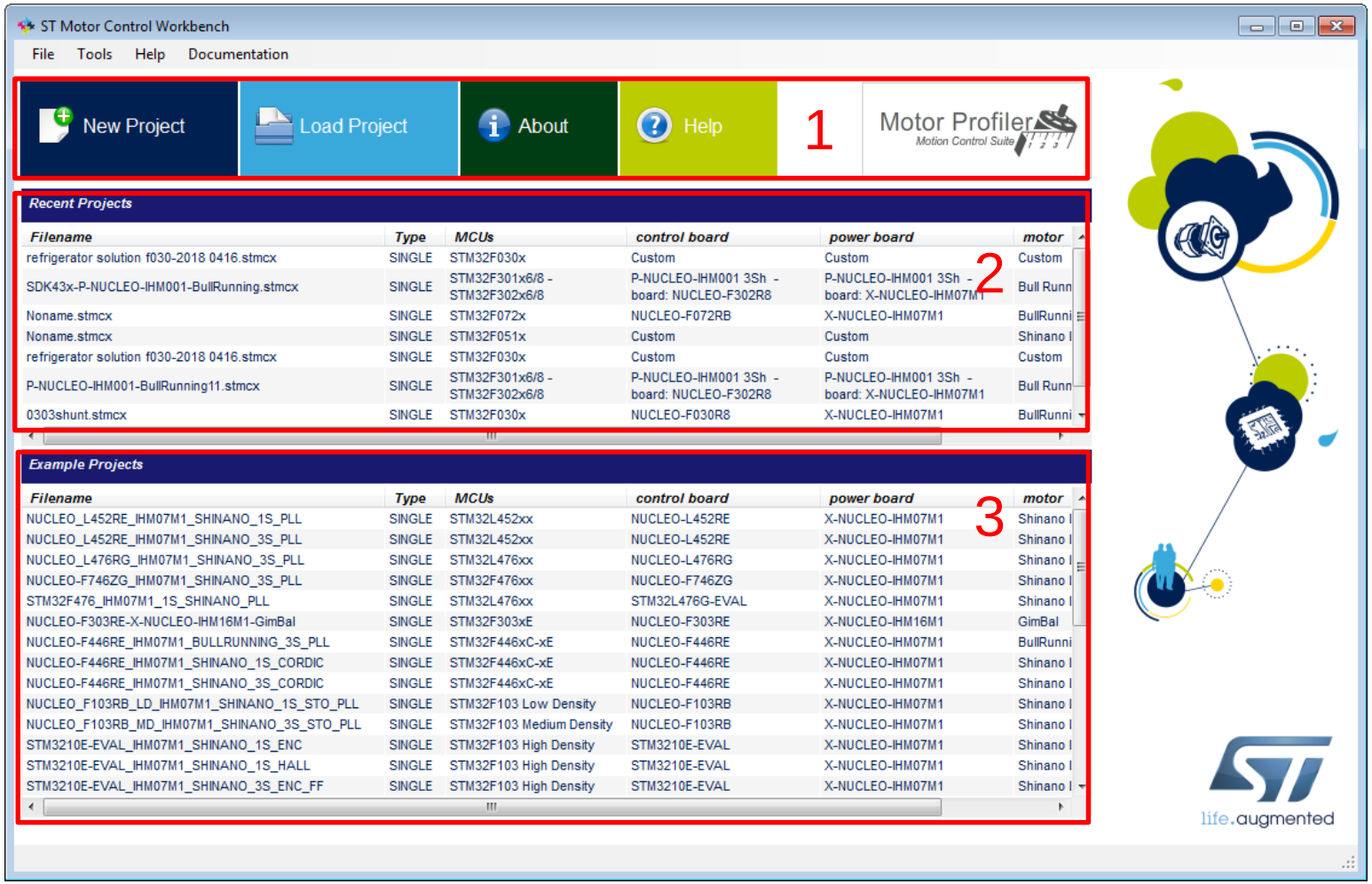
life.augmented

下列方式可启动ST MC Workbench软件工具：

- 单击其图标
- 从安装文件夹路径直接启动



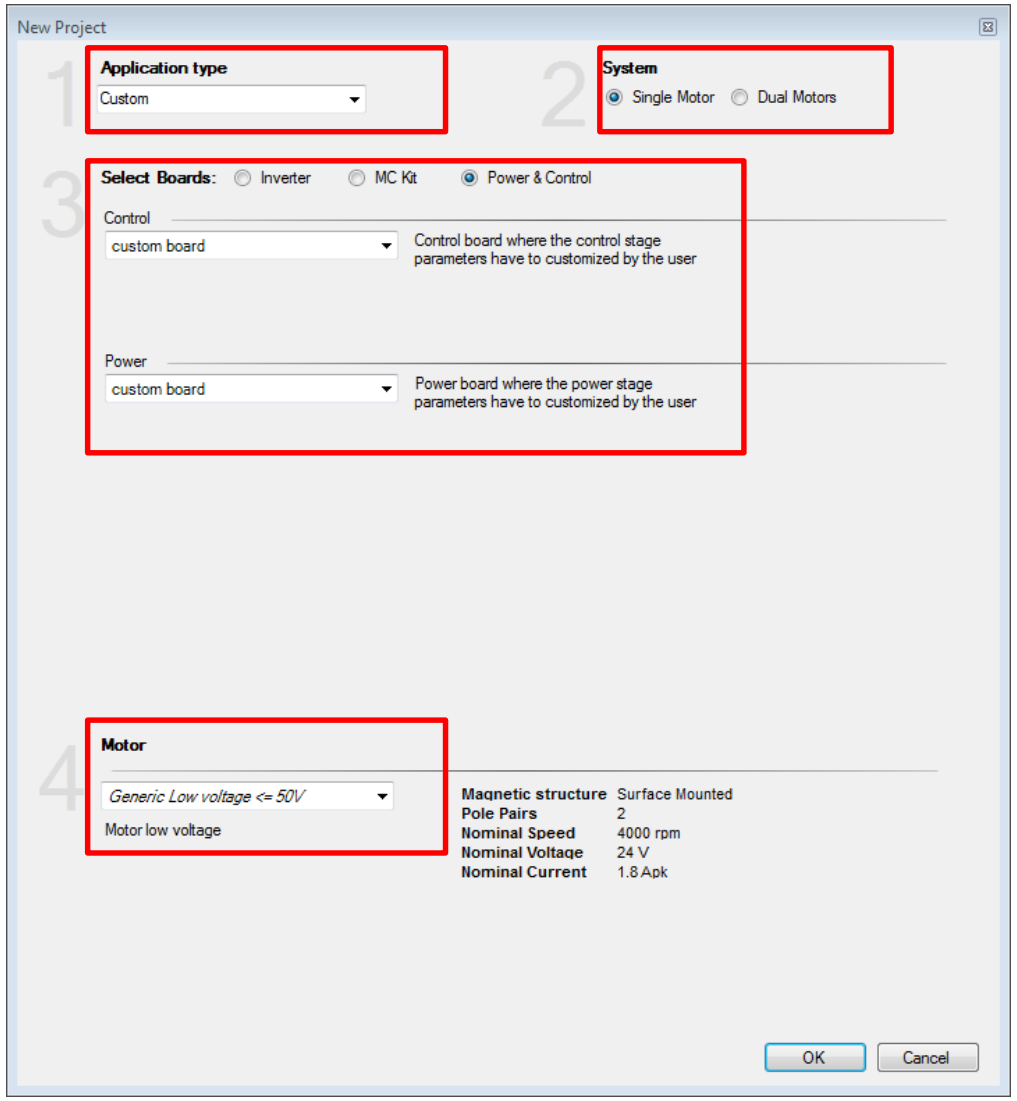
life.augmented

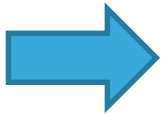


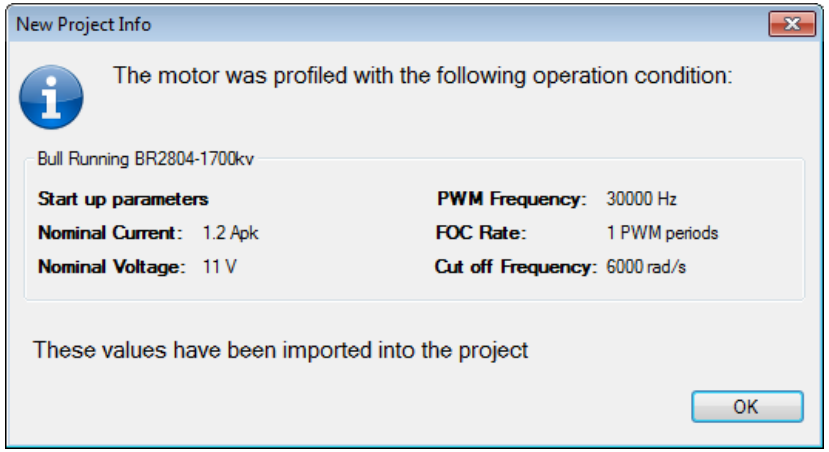
1. 用户按钮区用于创建新项目，加载已有项目或启动ST电机参数测量工具。
2. 最近的项目区用于加载近期的项目。
3. 例程区用于加载项目示例。

life.augmented

创建新工程

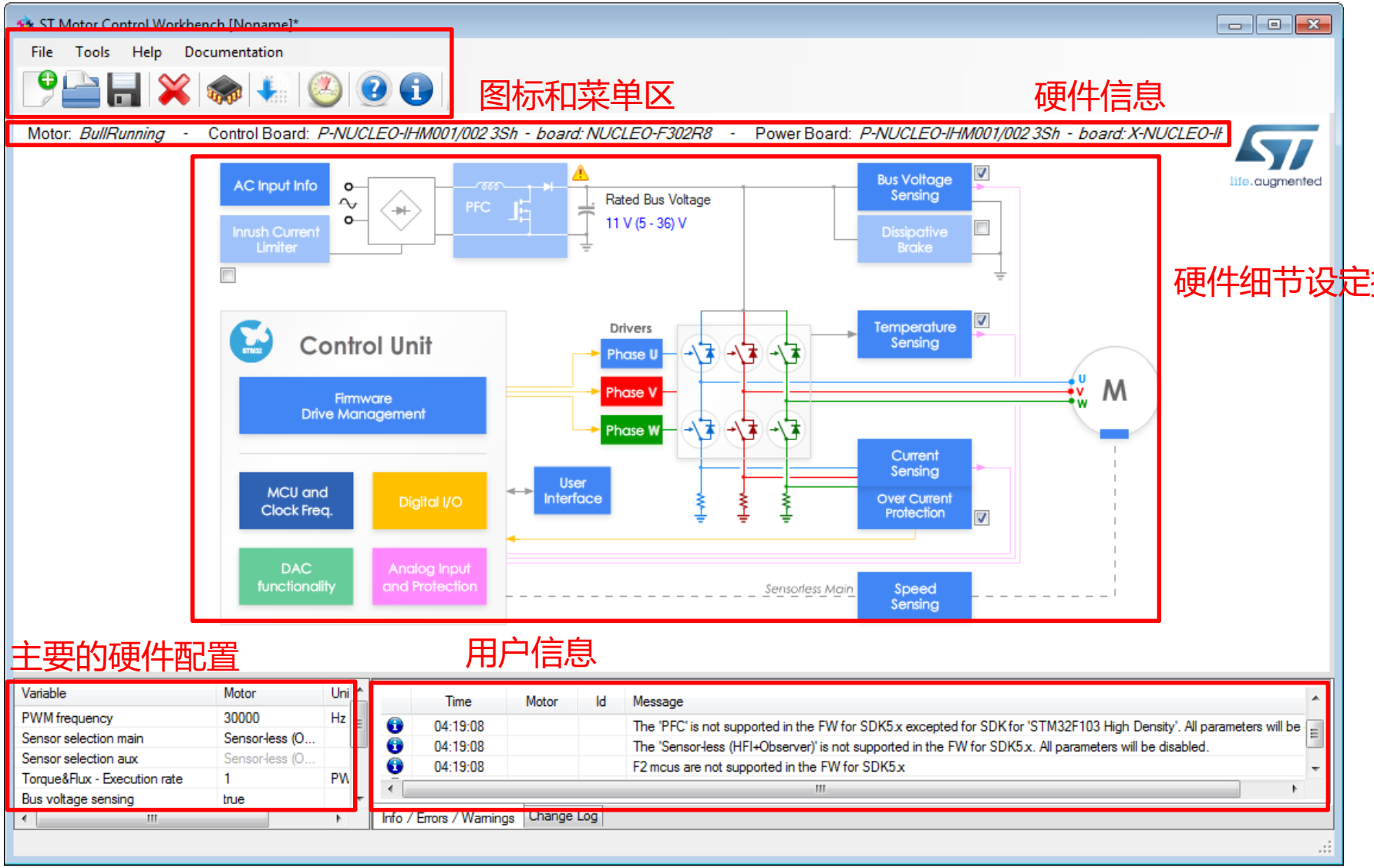




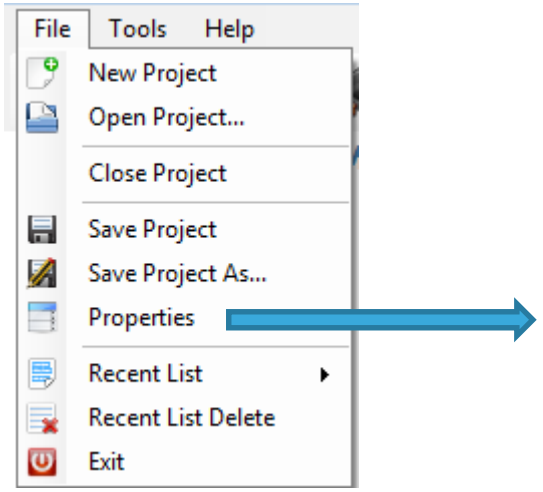


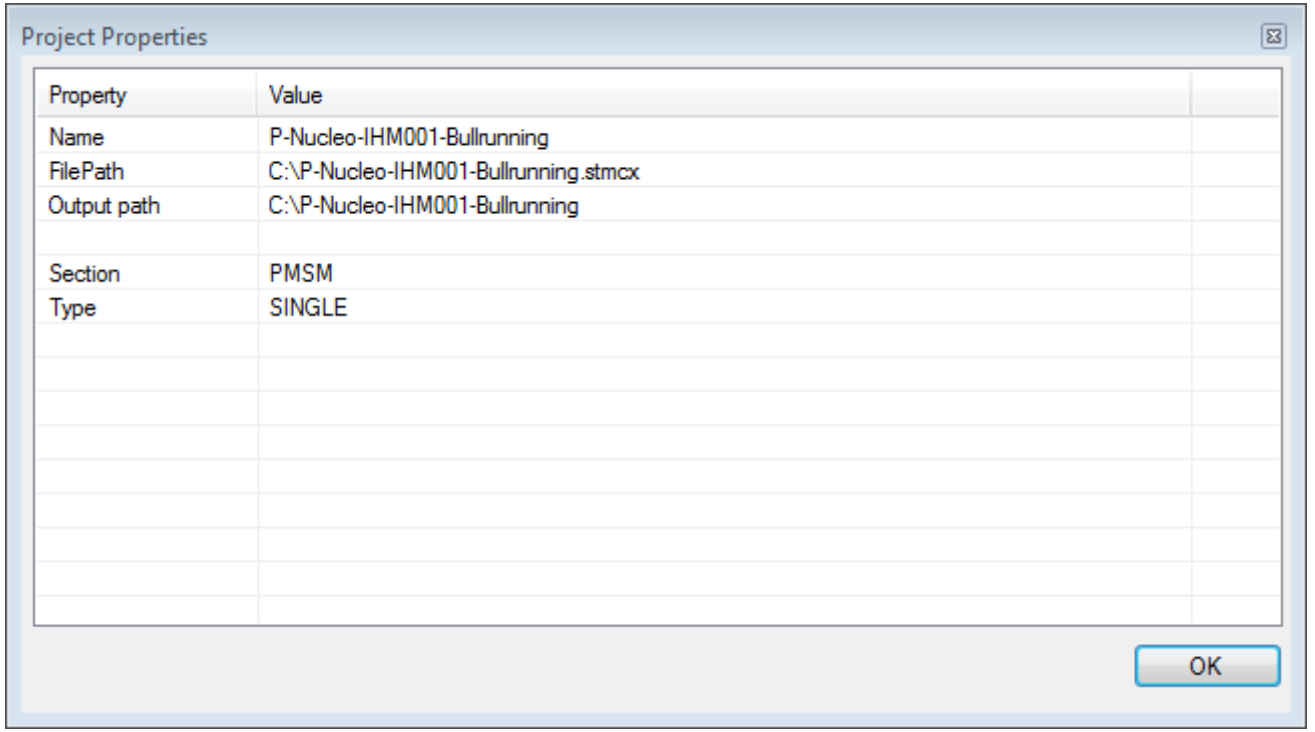
life.augmented

硬件配置窗口



Workbench菜单 (1)

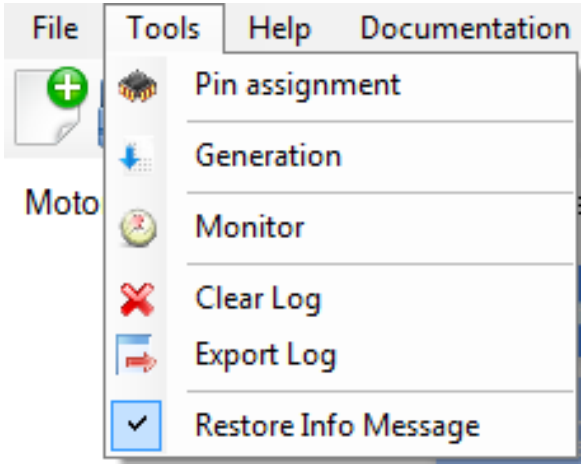


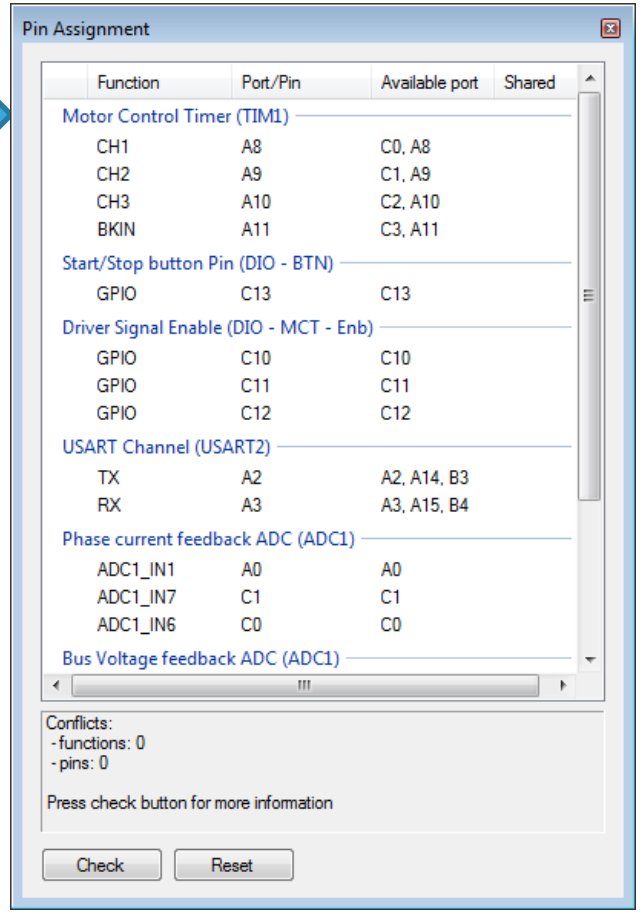


- ✓ New Project: 创建一个新项目
- ✓ Open Project…: 打开一个现有的项目
- ✓ Close Project: 关闭现在的项目
- ✓ Save Project: 使用相同文件名保存打开的项目
- ✓ Save Project As…: 将打开的项目保存为指定的文件名
- ✓ Properties: 查看项目属性
- ✓ Recent List: 从最近打开过的项目列表中加载一个现有项目
- ✓ Recent List Delete: 删除最近的项目列表
- ✓ Exit: 从硬件配置窗口退出

life.augmented

Workbench菜单 (2)

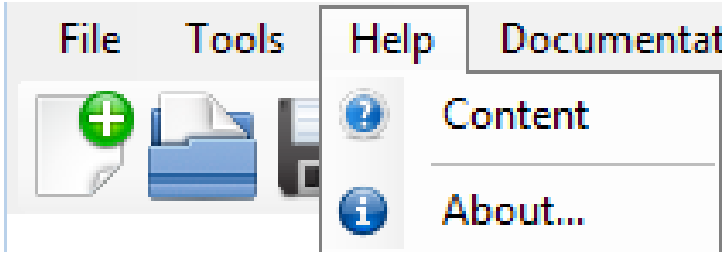


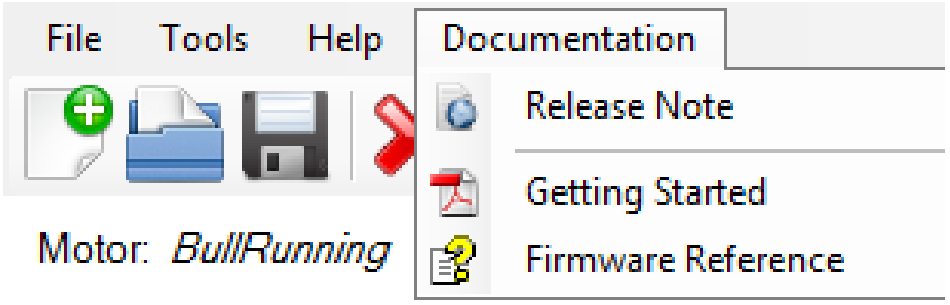


- ✓ Pin assignment:检查MCU的引脚分配和剩余的可用引脚
- ✓ Generation:根据所选的IDE，生成MC应用工程文件
- ✓ Monitor:监控并转动电机
- ✓ Clear Log:清除用户信息表
- ✓ Export Log: 将用户信息表以文本格式导出到日志文件
- ✓ Restore Info Message: 需要时显示用户信息表

life.augmented

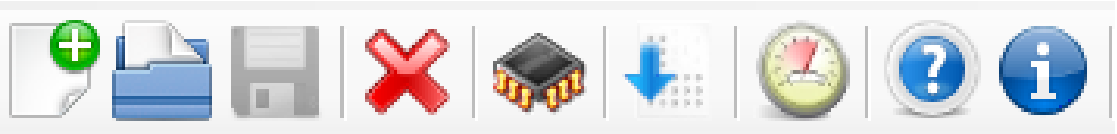
Workbench菜单 (3)





life.augmented

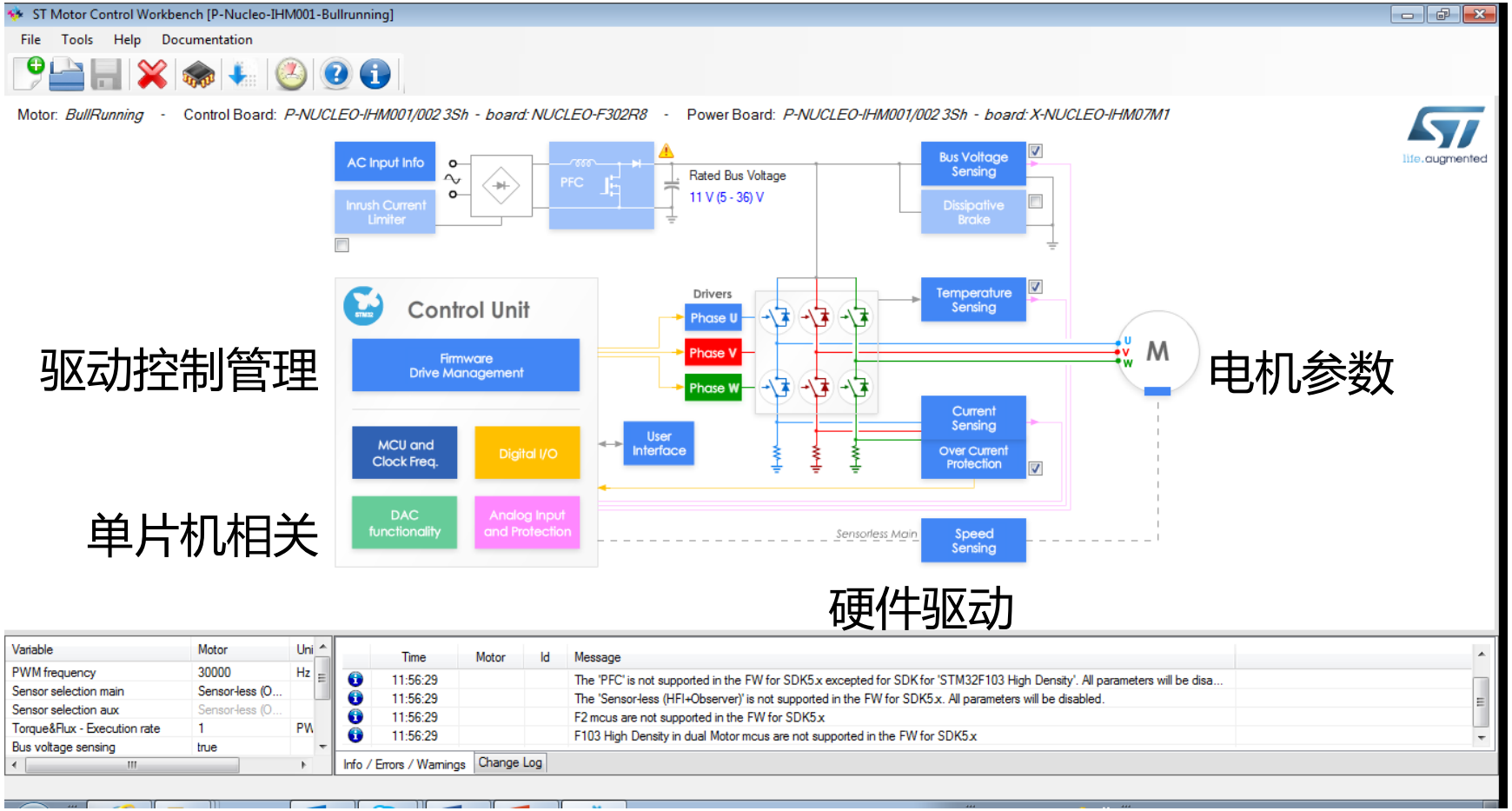
工具栏图标



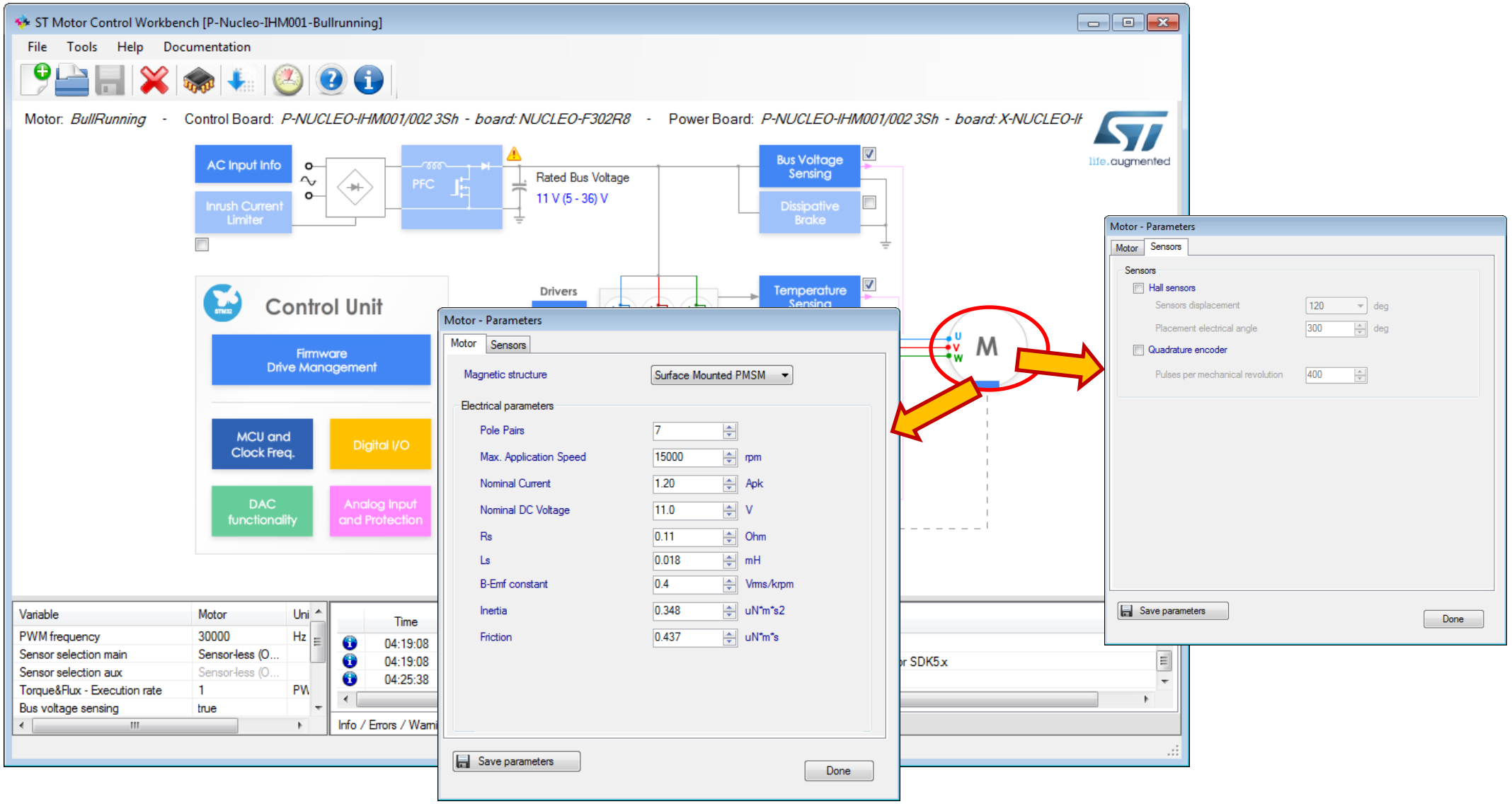
- ✓ New：创建一个新项目
- ✓ Load：打开一个现有的项目
- ✓ Save：使用相同文件名保存打开的项目
- ✓ Clear Log：清除用户信息表
- ✓ Pin assignment： 检查MCU的引脚分配和剩余的可用引脚
- ✓ Generation: 根据所选的IDE，生成MC应用工程文件.
- ✓ Click to open monitor: 监控并转动电机
- ✓ Help: 提供在线帮助文件的访问接口
- ✓ About: 显示该应用的版本号

life.augmented

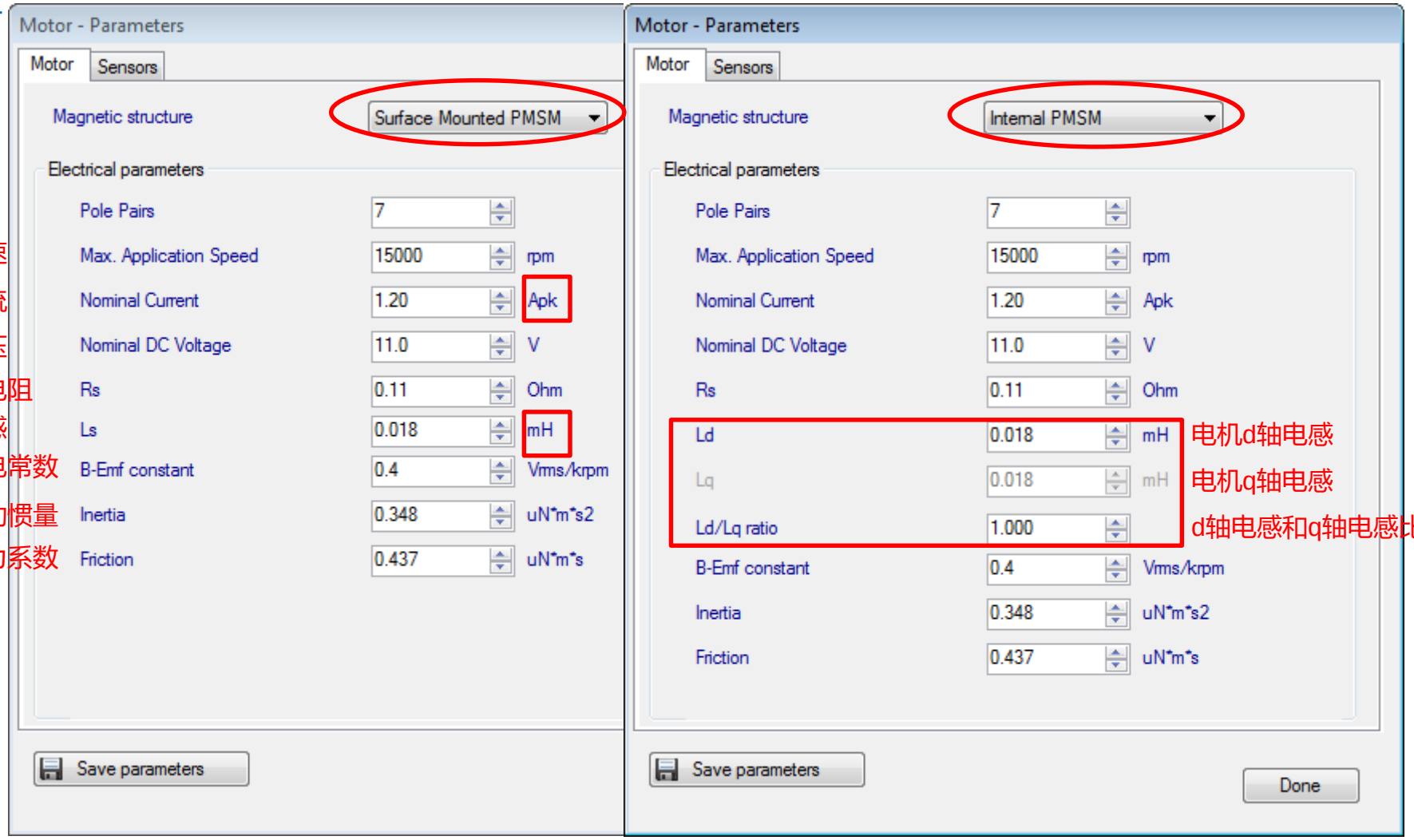
根据客户需求快速设置电机库



电机参数



➢ 马达参数配置



极对数

最大转速

最大电流

额定电压

电机相电阻

电机电感

电机发电常数

电机转动惯量

电机阻力系数

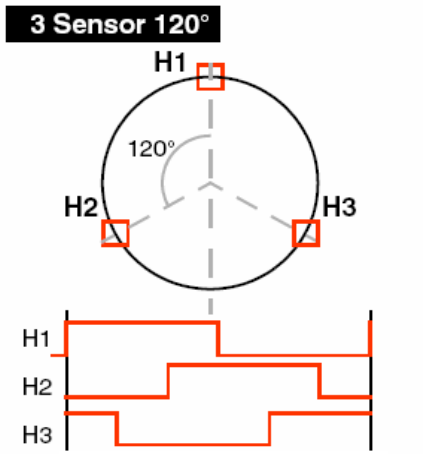
电机d轴电感

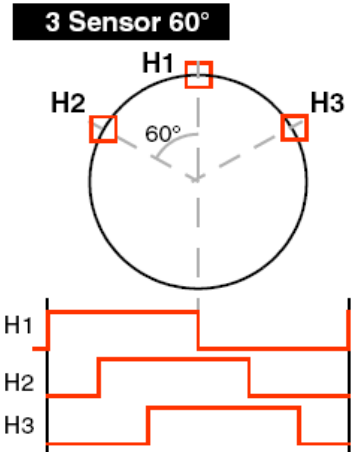
电机q轴电感

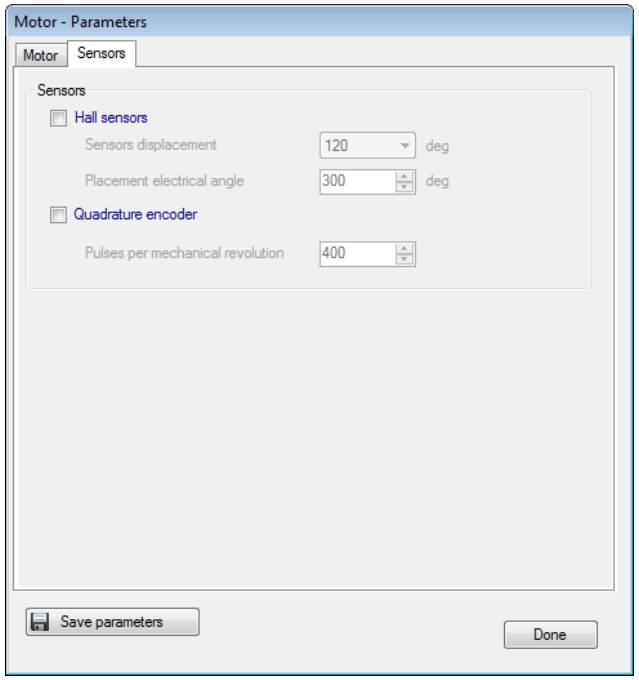
d轴电感和q轴电感比率

life.augmented

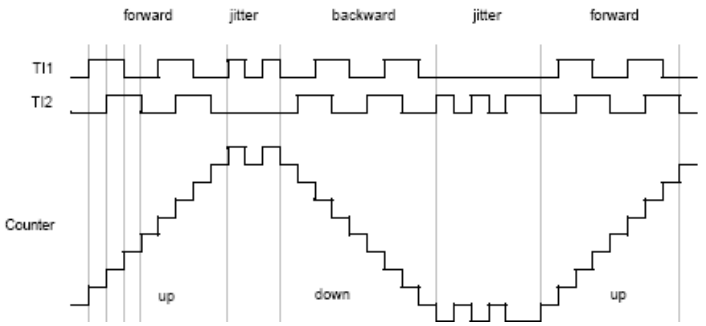
- Hall同步电角度：
  - 电机Hall A的上升沿到电机A相反电动势最高点的延迟角度。
  - 默认电机A相的反电动势最高点作为电角度的0度；



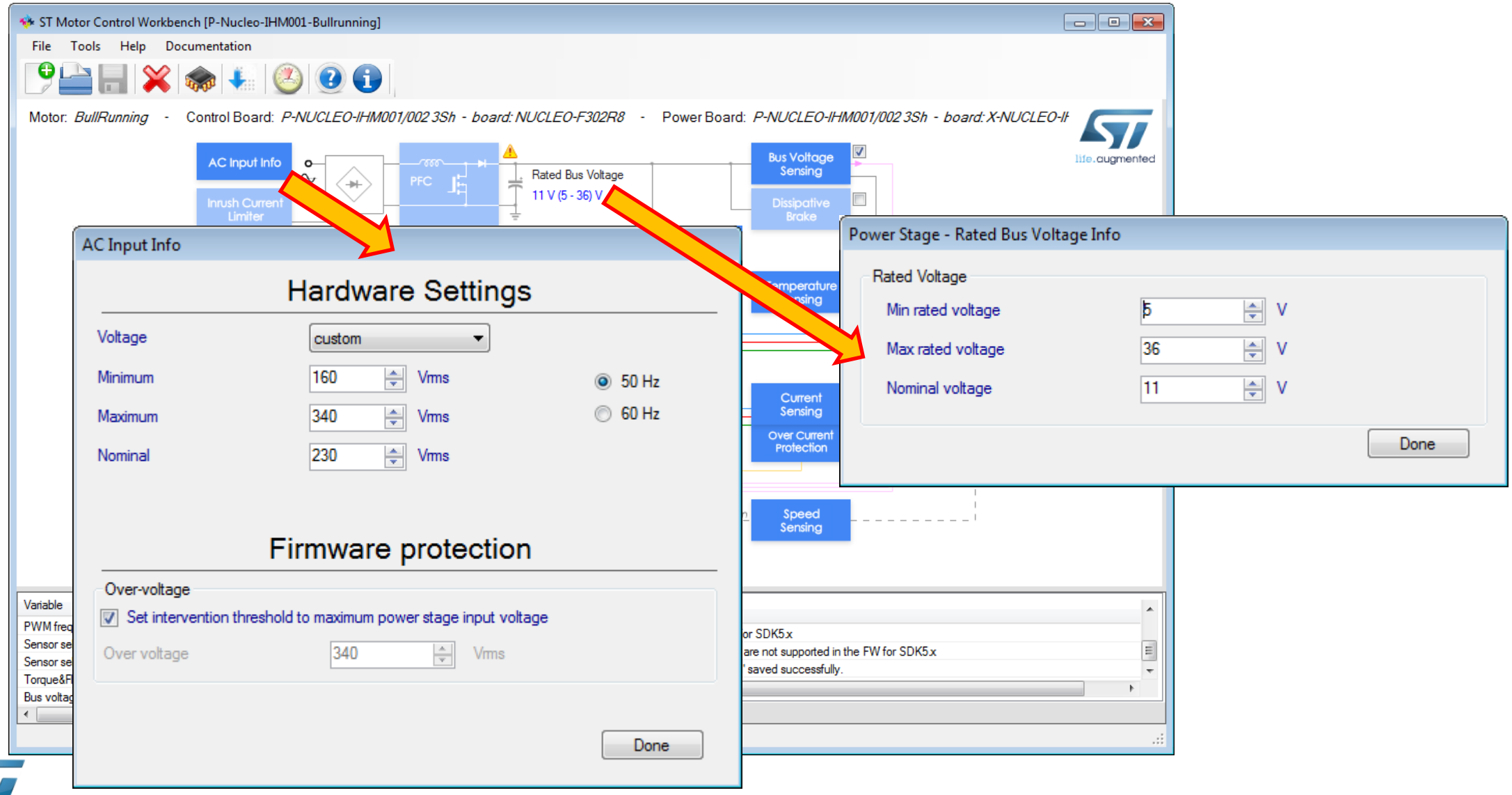




- 编码器参数
  - Resolution：分辨率，单位：Pulse/Round
  - 旋转一圈的编码器脉冲个数

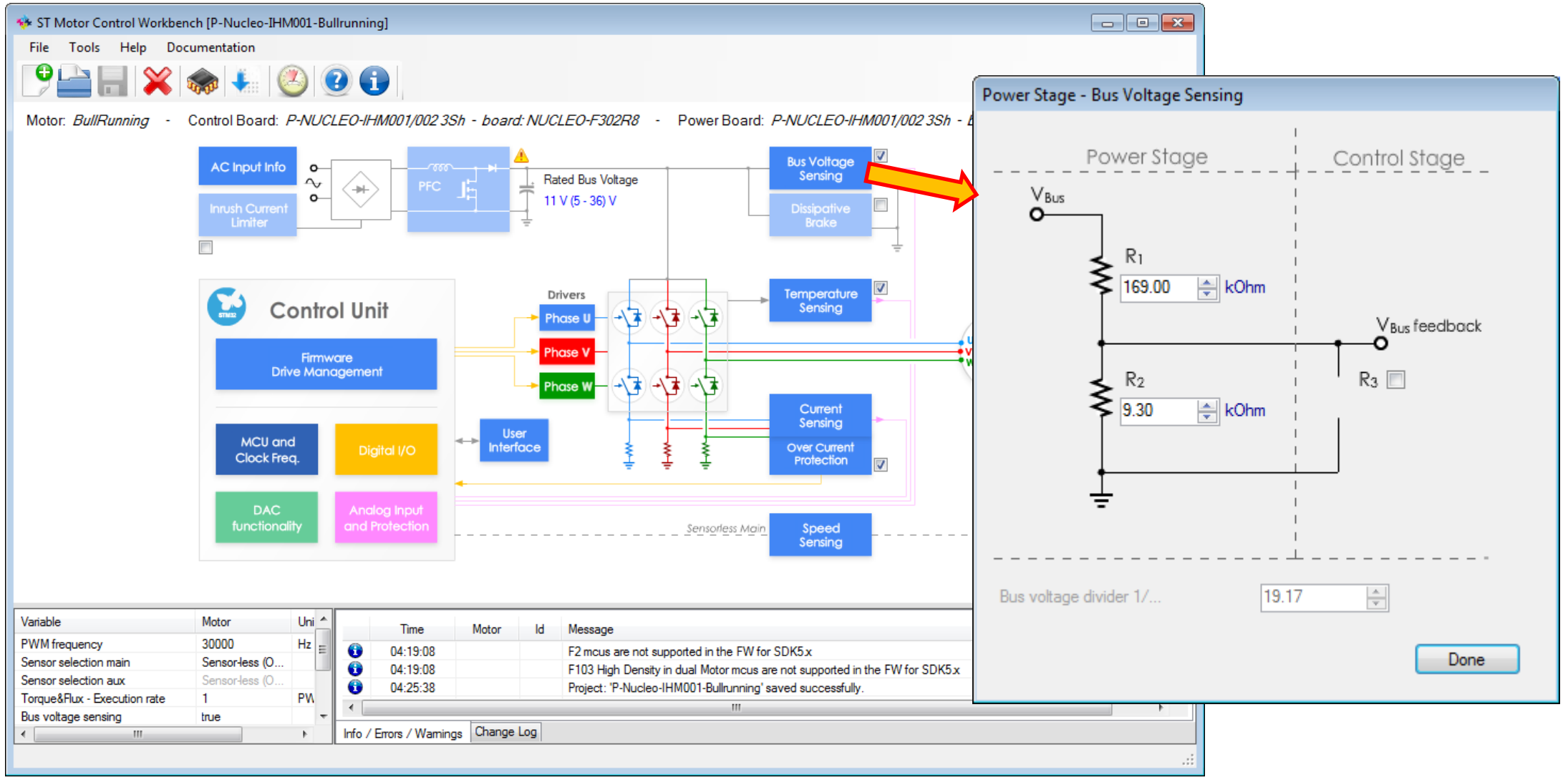


电源输入信息



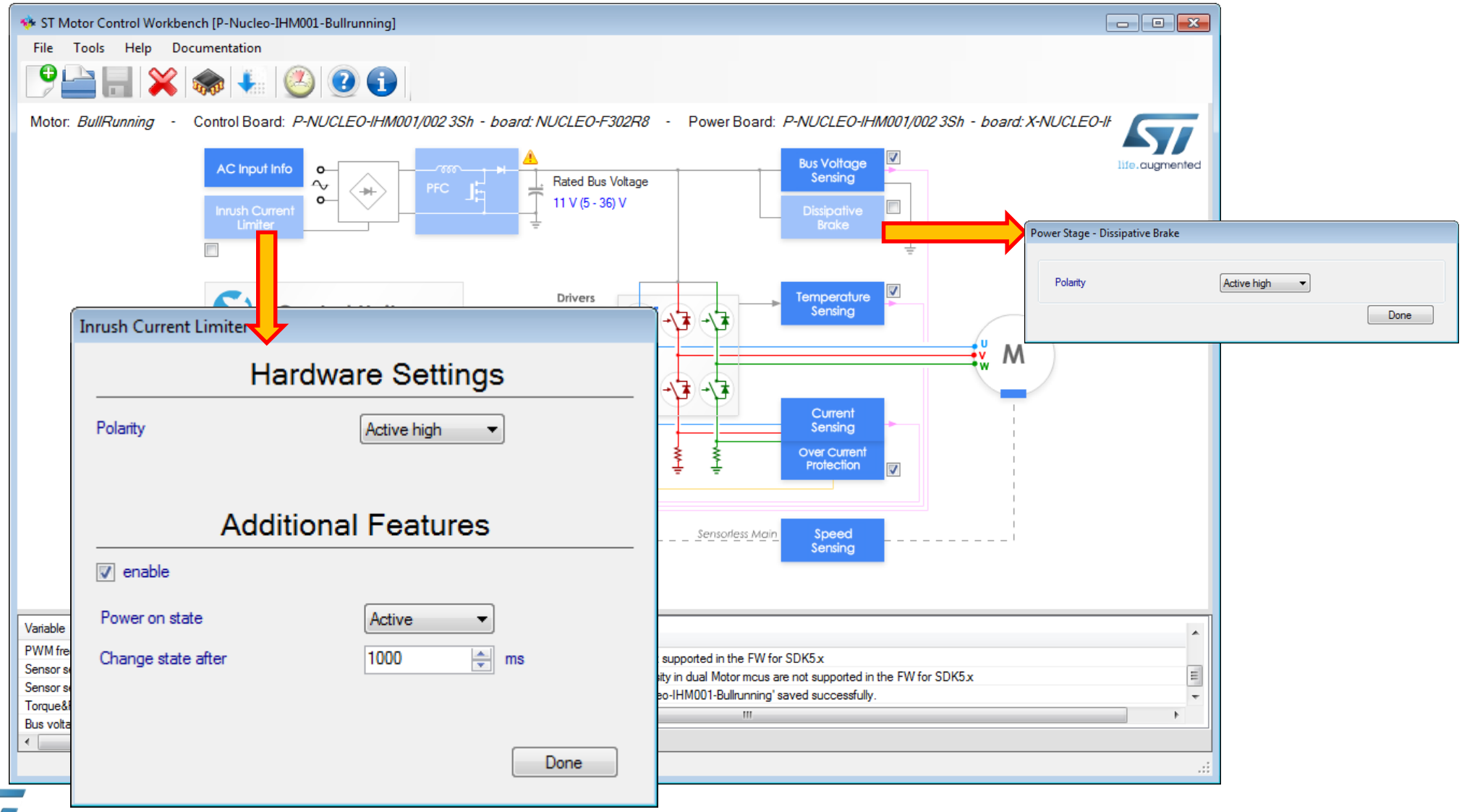
life.augmented

直流母线电压采样拓扑

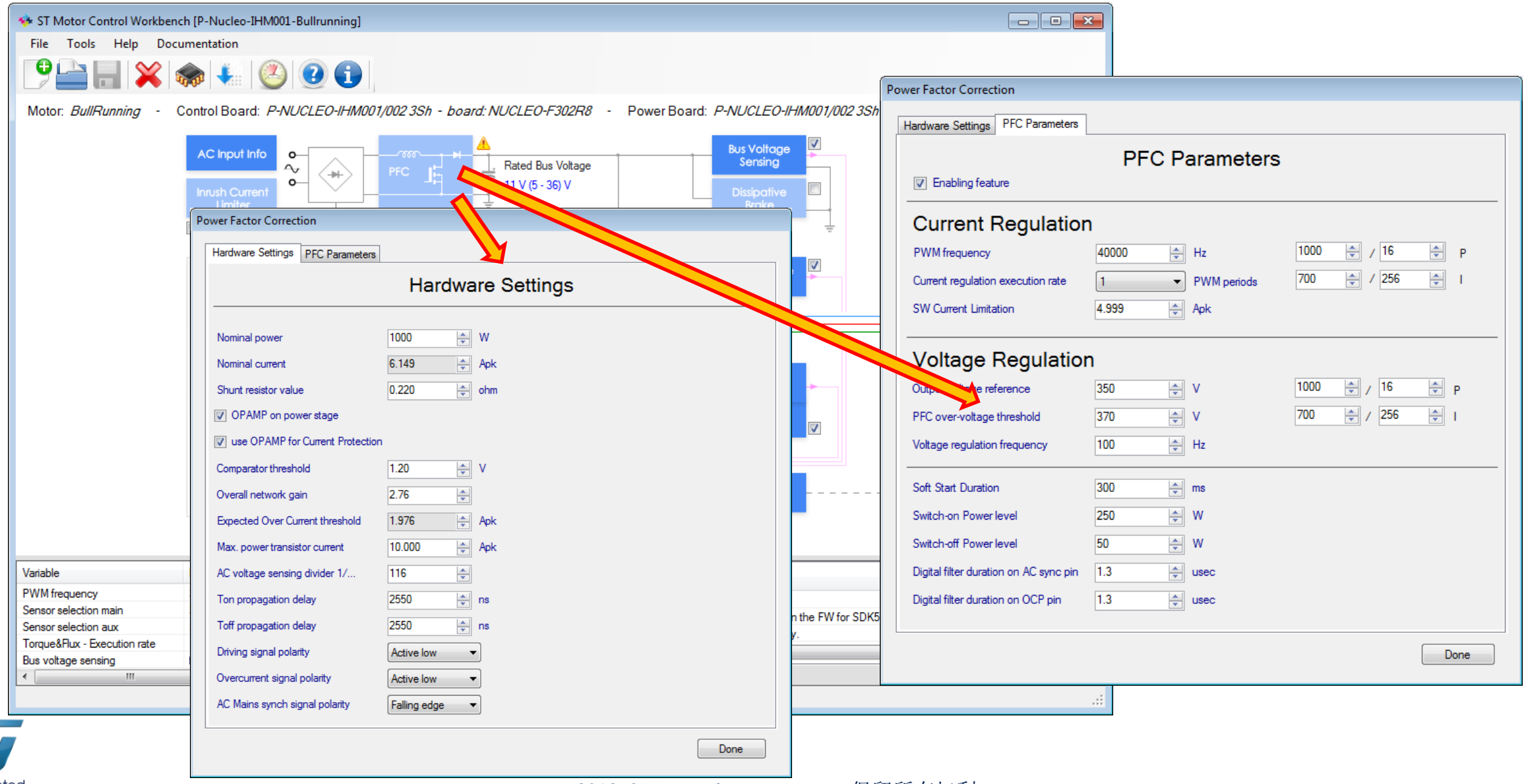


life.augmented

浪涌电流限制器和耗散制动

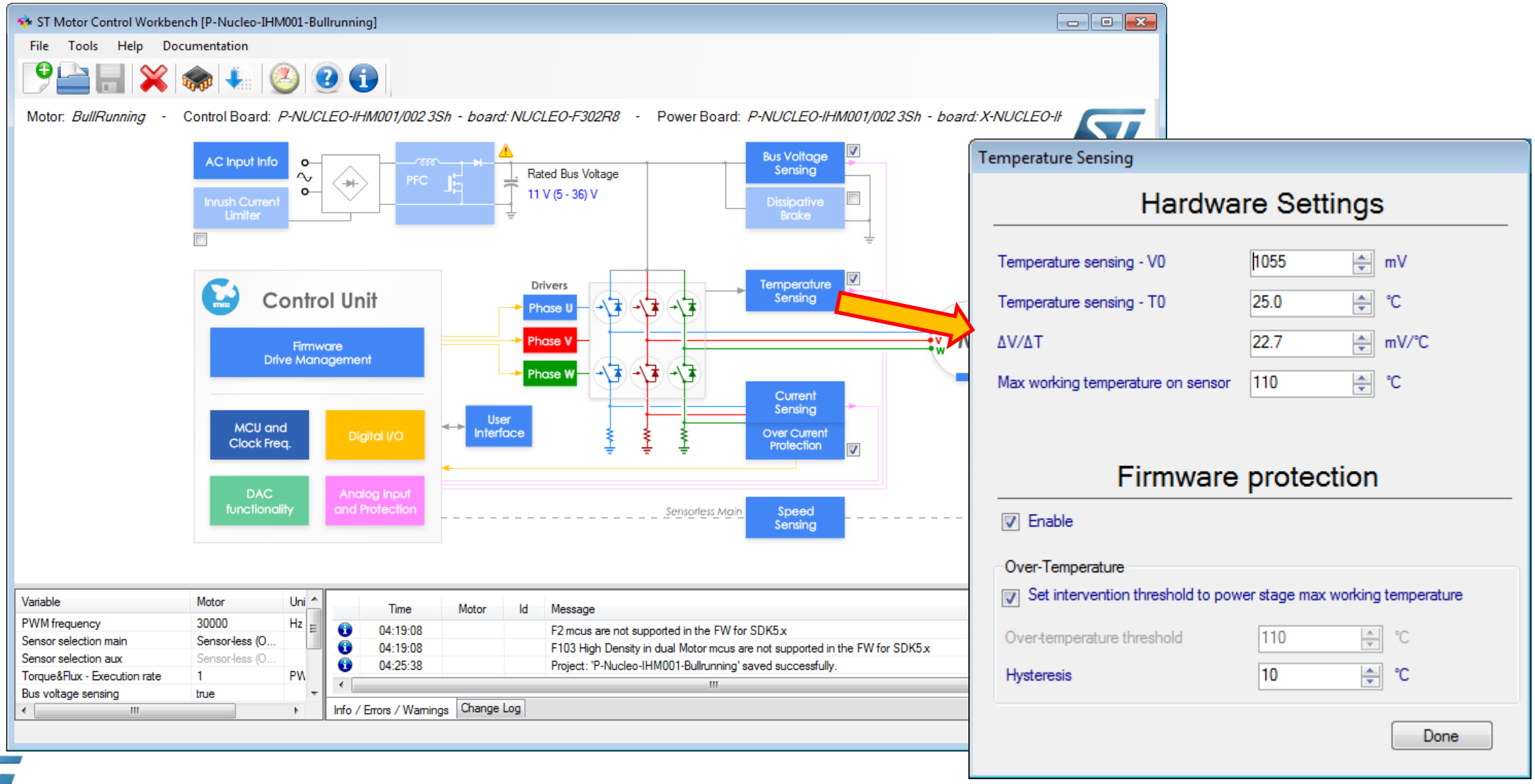


功率因数校正

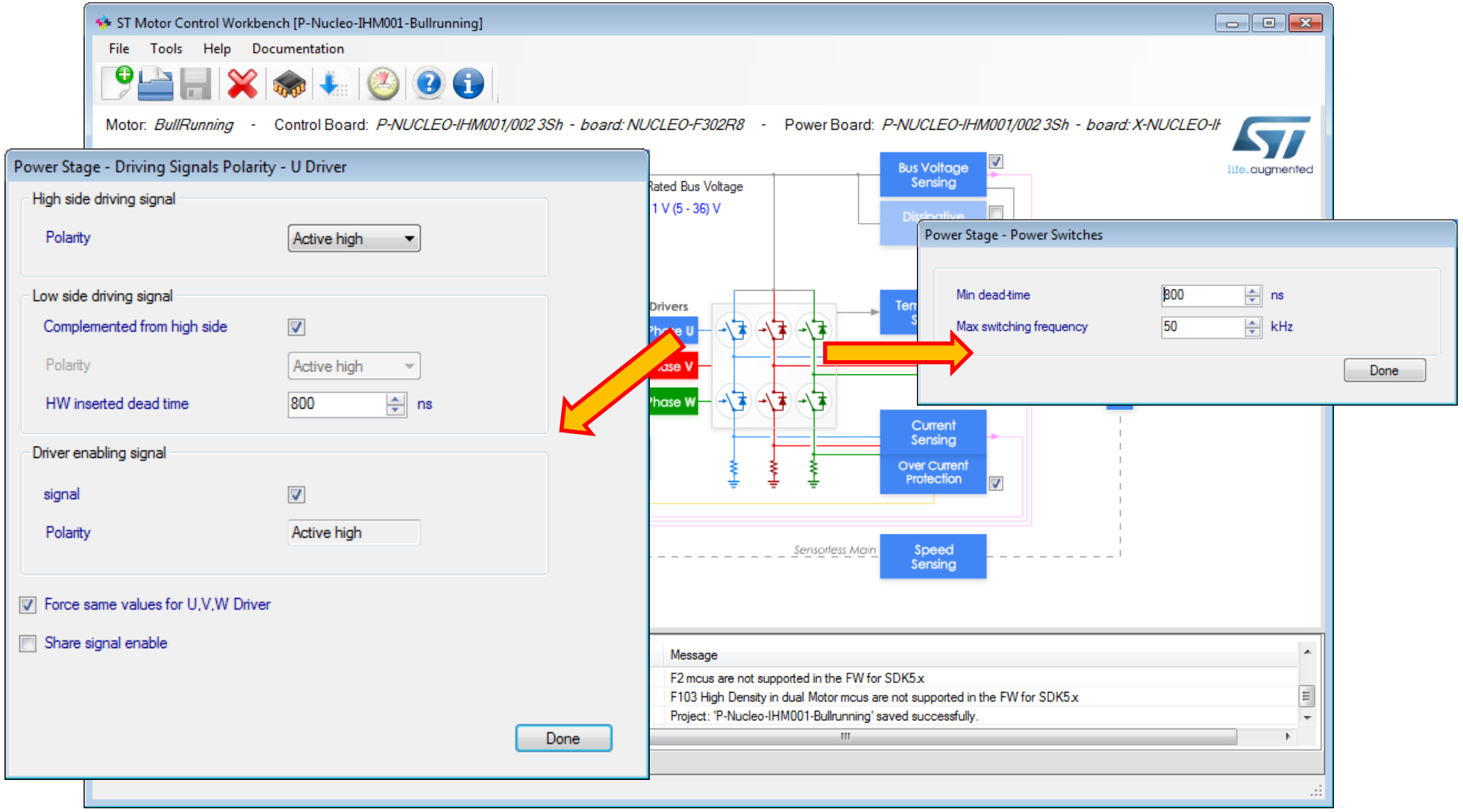


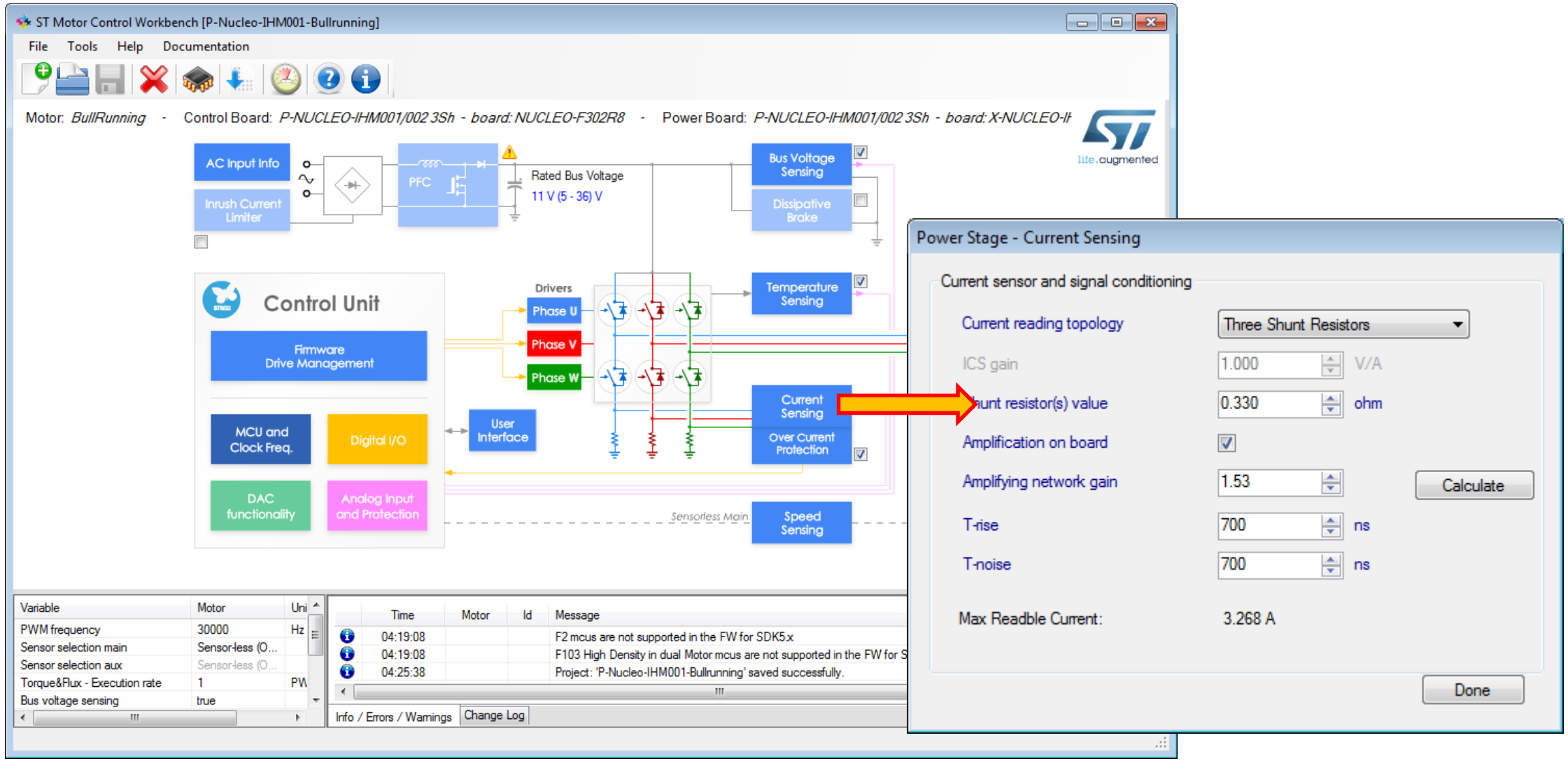
life.augmented

温度检测



life.augmented

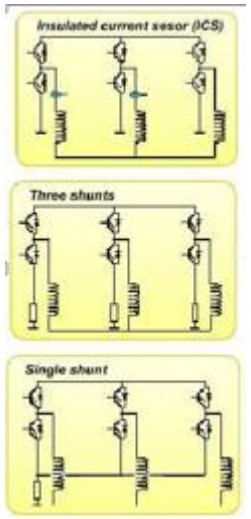


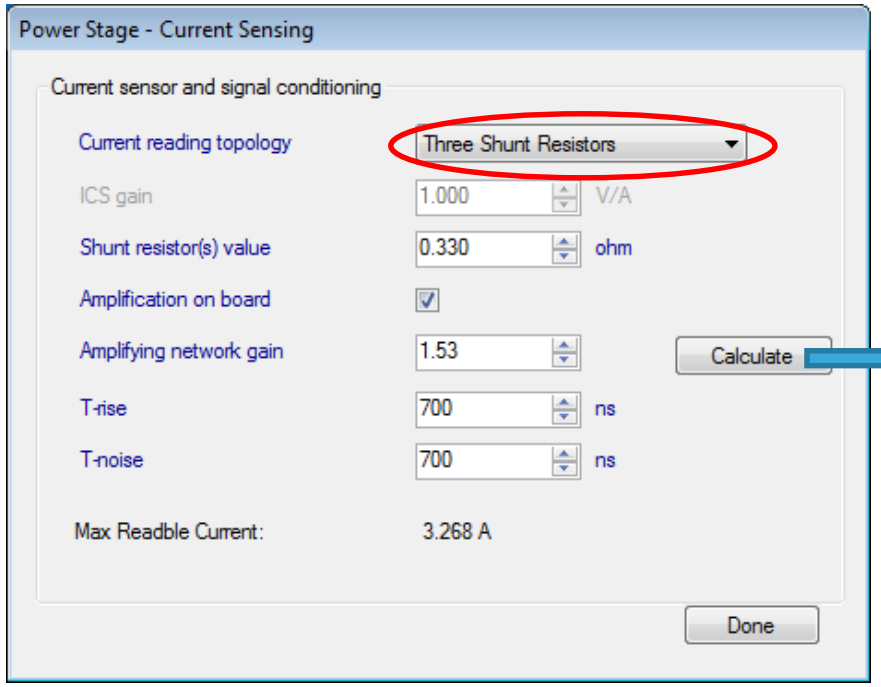


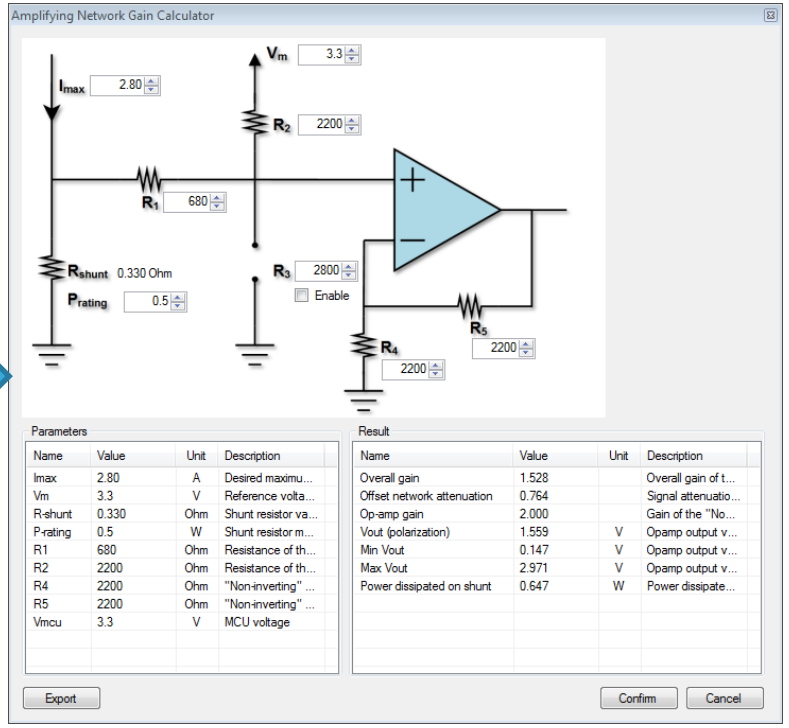
电流采样增益计算器

电流采样拓扑结构选择：

$$(1+R5/R4)\cdot\frac{(R2\,//\,R3)}{((R2\,//\,R3)+R1)}$$

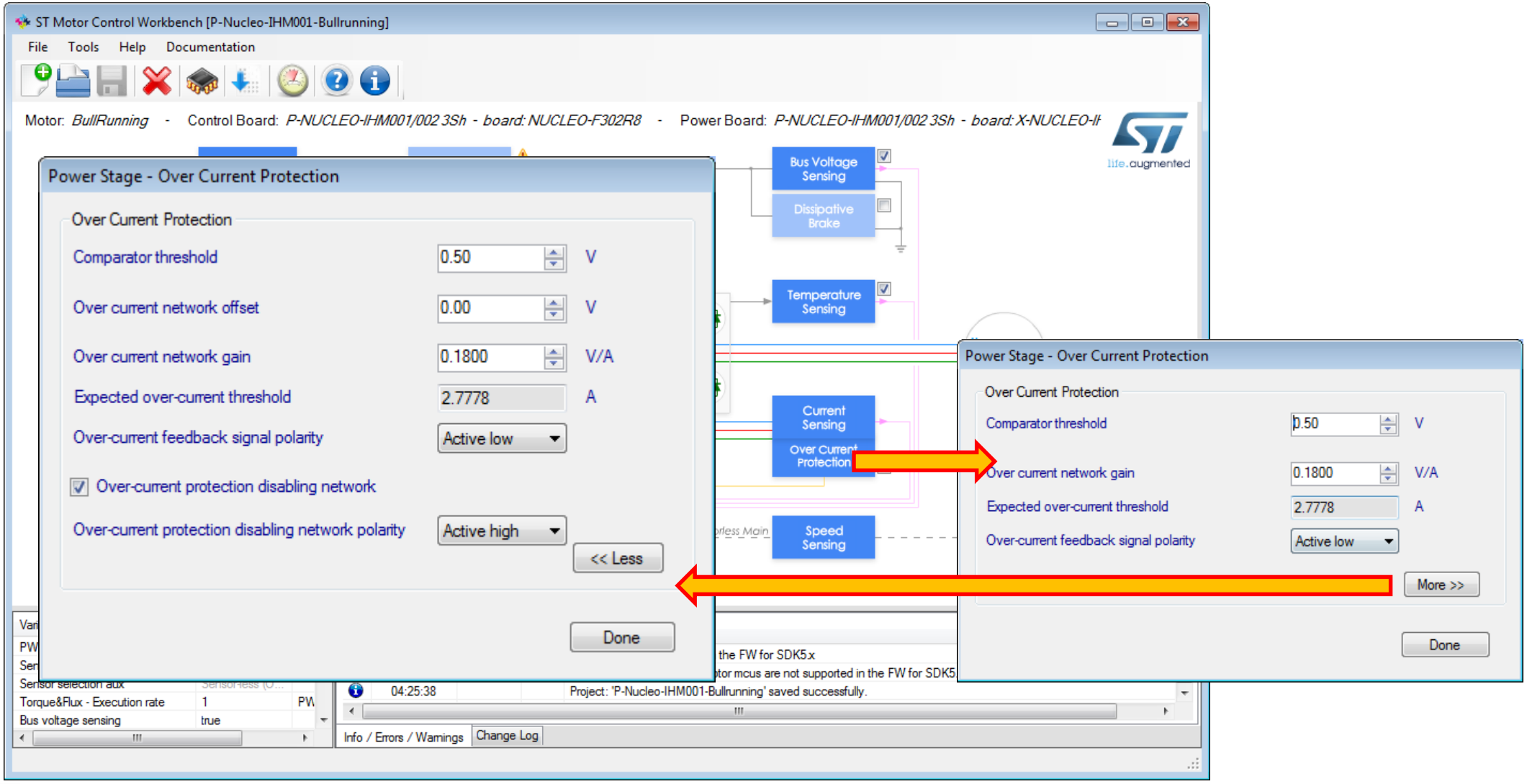


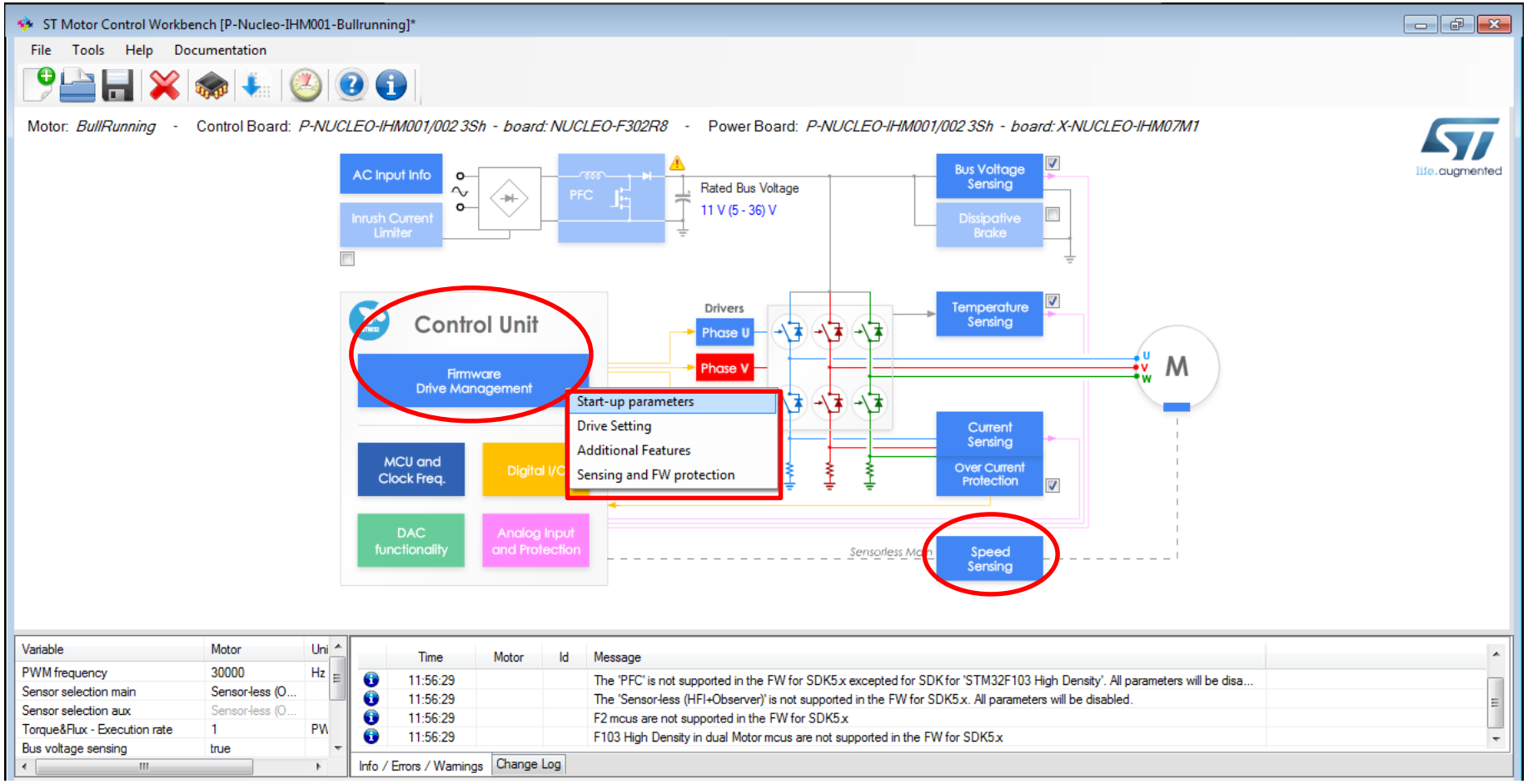




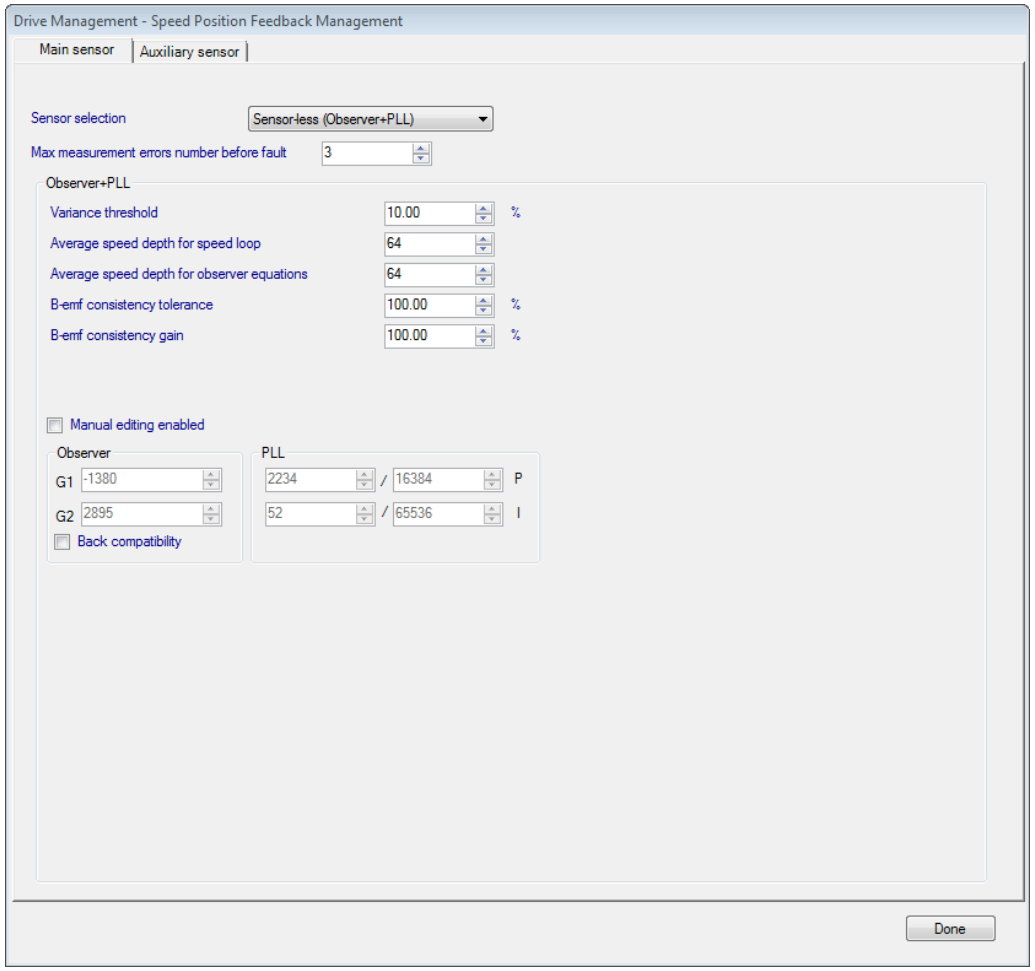
life.augmented

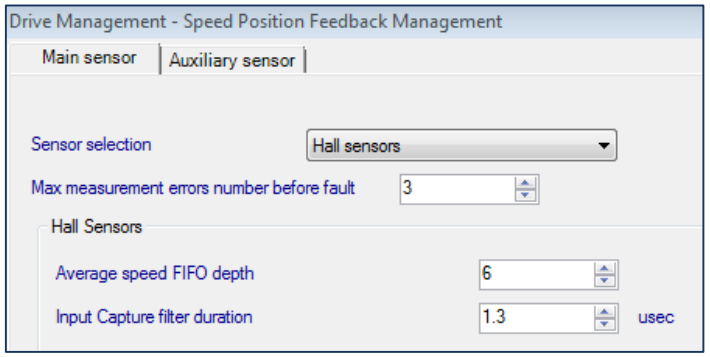
过流保护

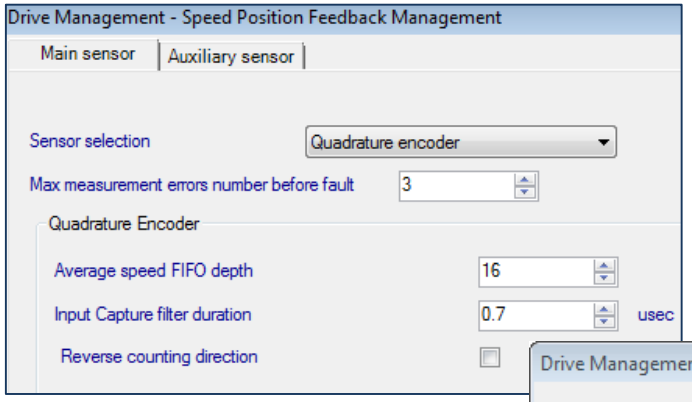


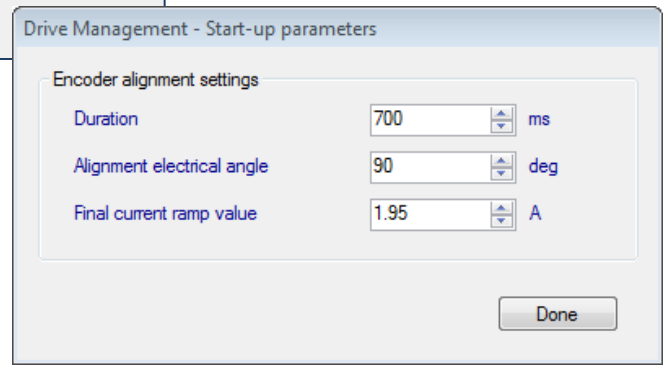


主传感器(1)

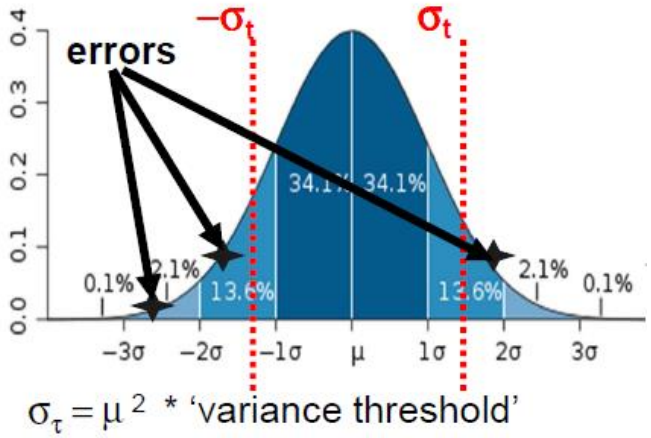








life.augmented

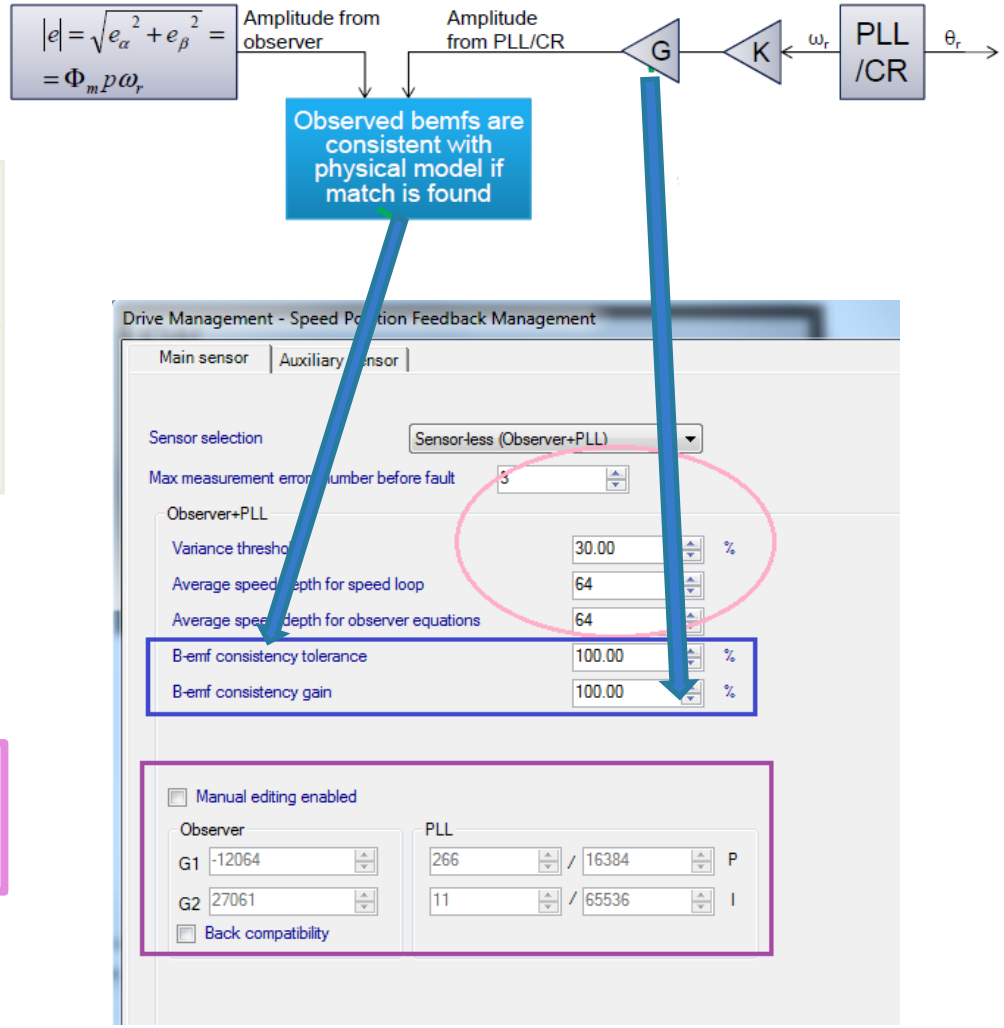


Speed Average value

$$\mu = \frac{\sum_{i=1}^{64} \omega_i}{64}$$

Speed Variance

$$\sigma^2 = \frac{\sum_{i=1}^{64} (\omega_i - \mu)^2}{64}$$

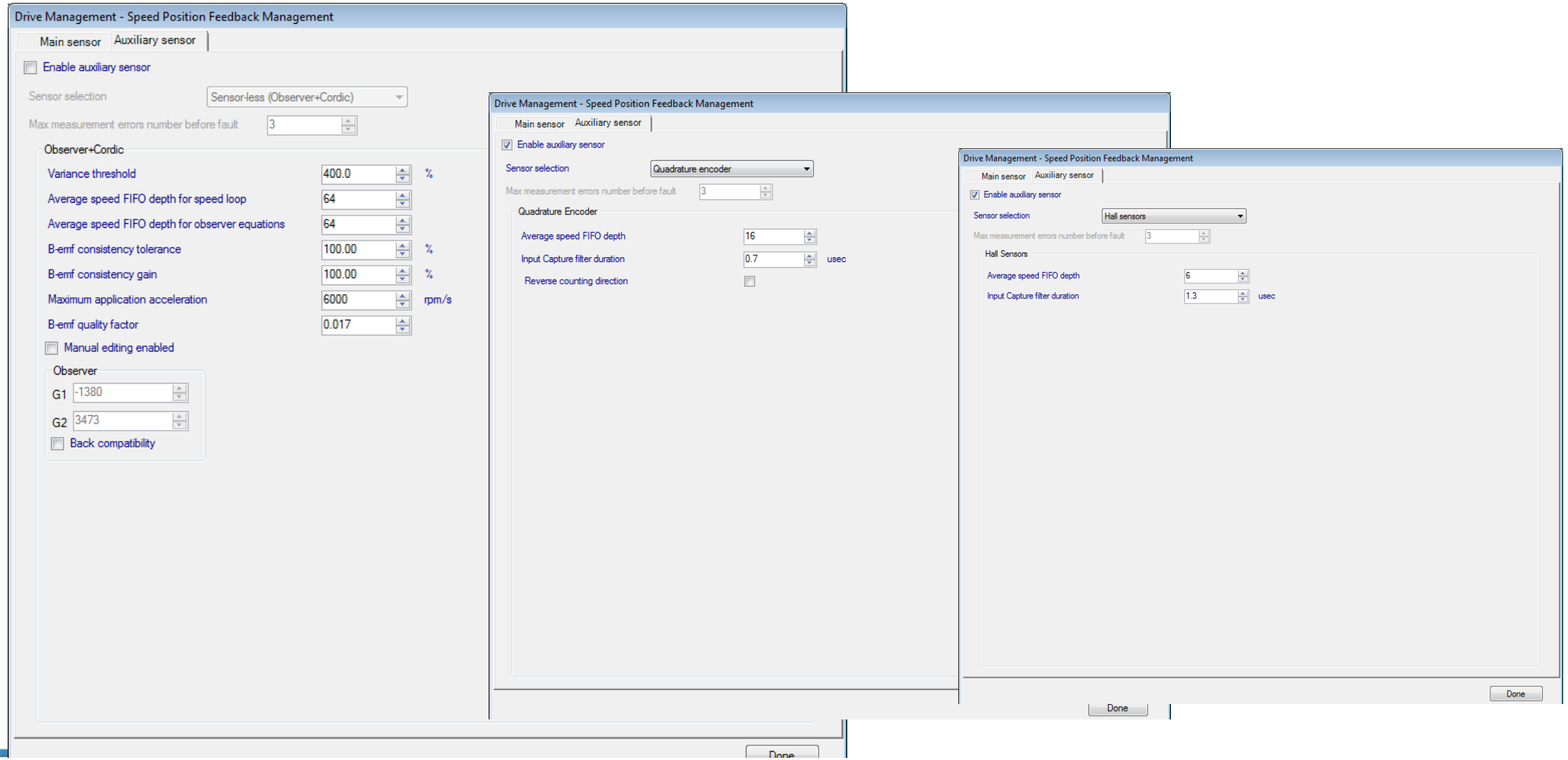


- BEMF收敛：
  - 一般用于堵转判断
  - 设定为100%时，忽略此判断

根据参数计算得到的系数，一般情况下无需修改，保持为默认数据

life.augmented

辅助传感器



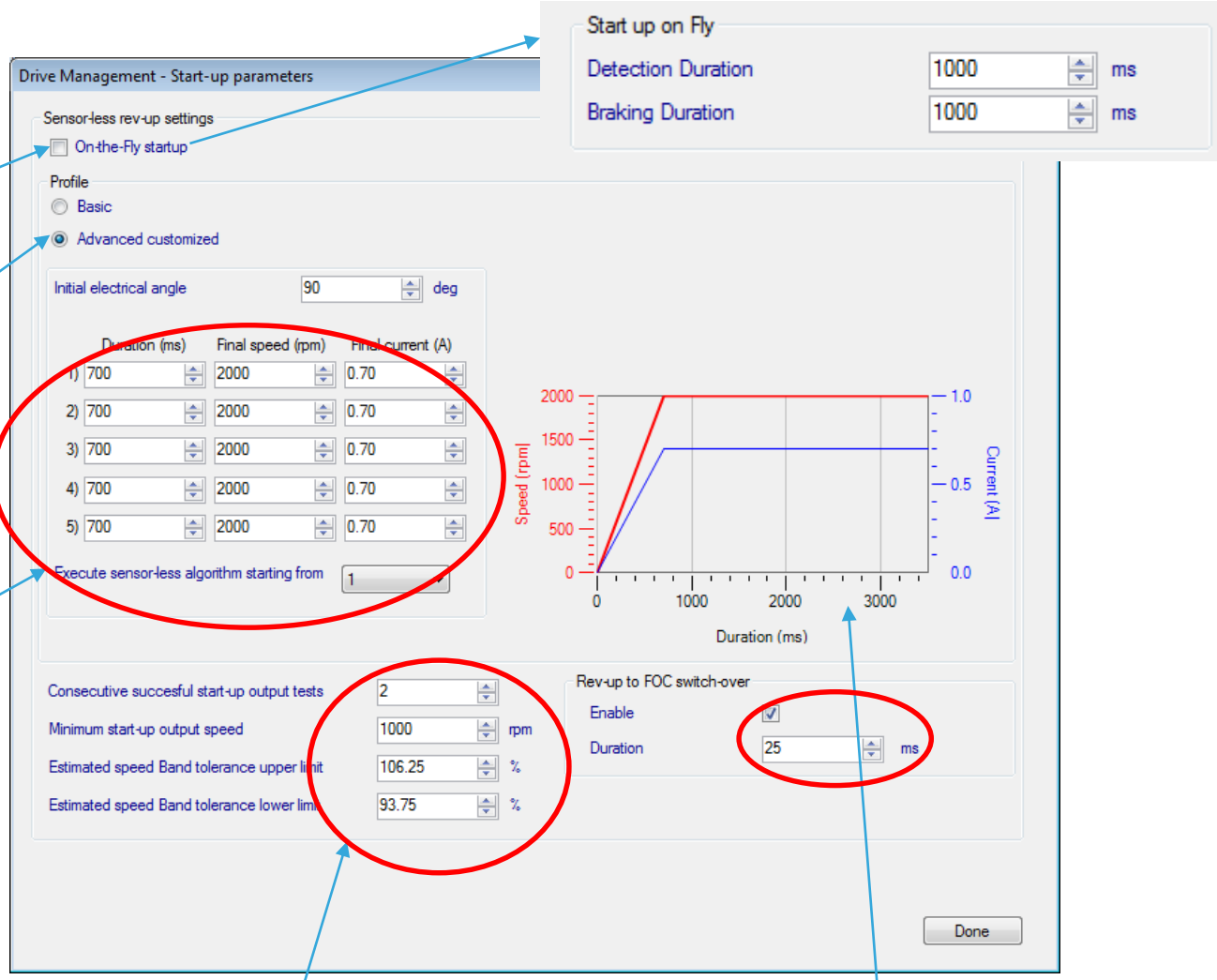
life.augmented

- 配置无传感器开环启动参数
  - 有传感器不用配置

顺逆风启动检测使能

折线或曲线启动选择

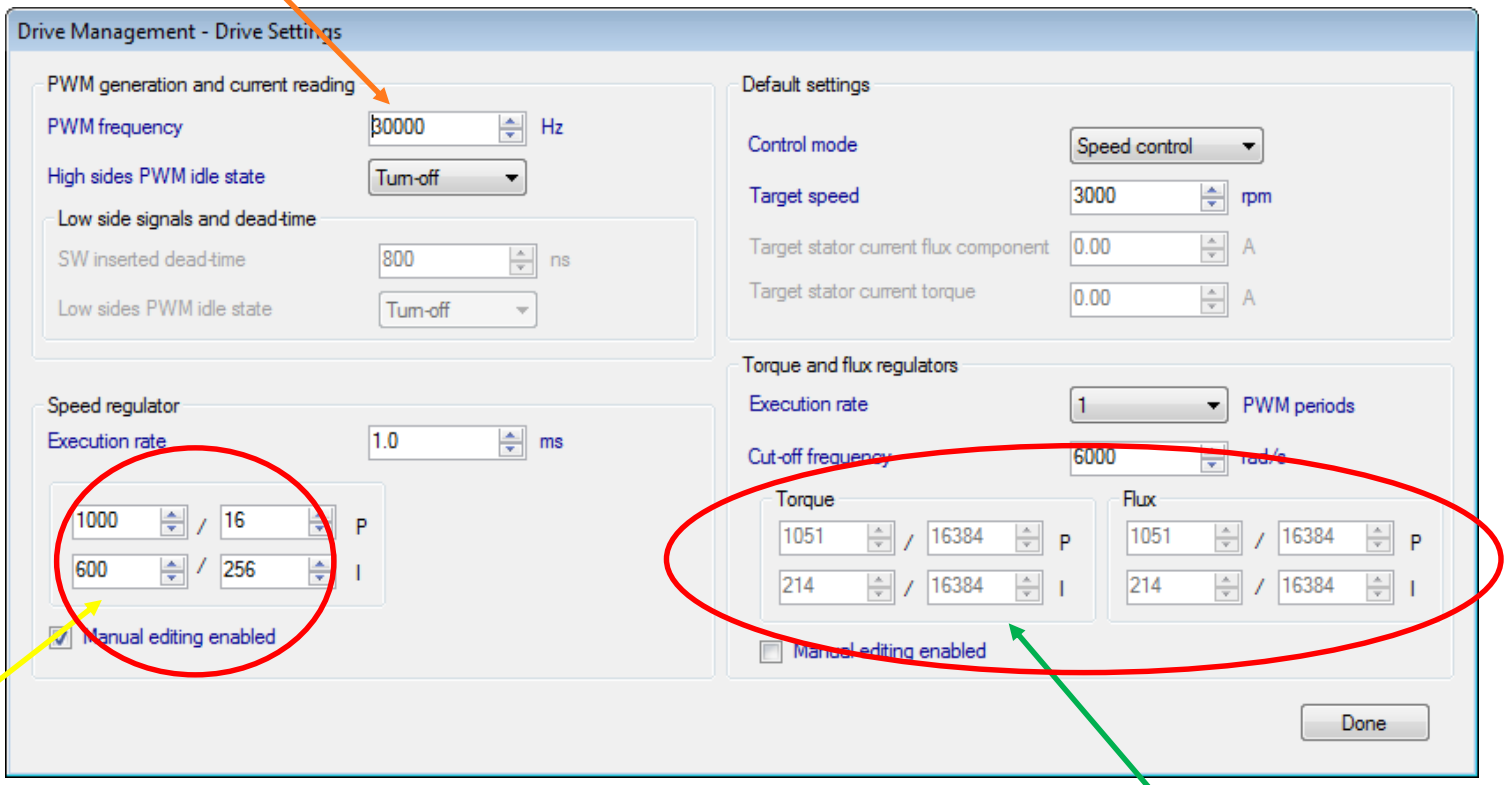
启动曲线配置，包含速度，电流，时间



切入闭环条件判定

启动曲线

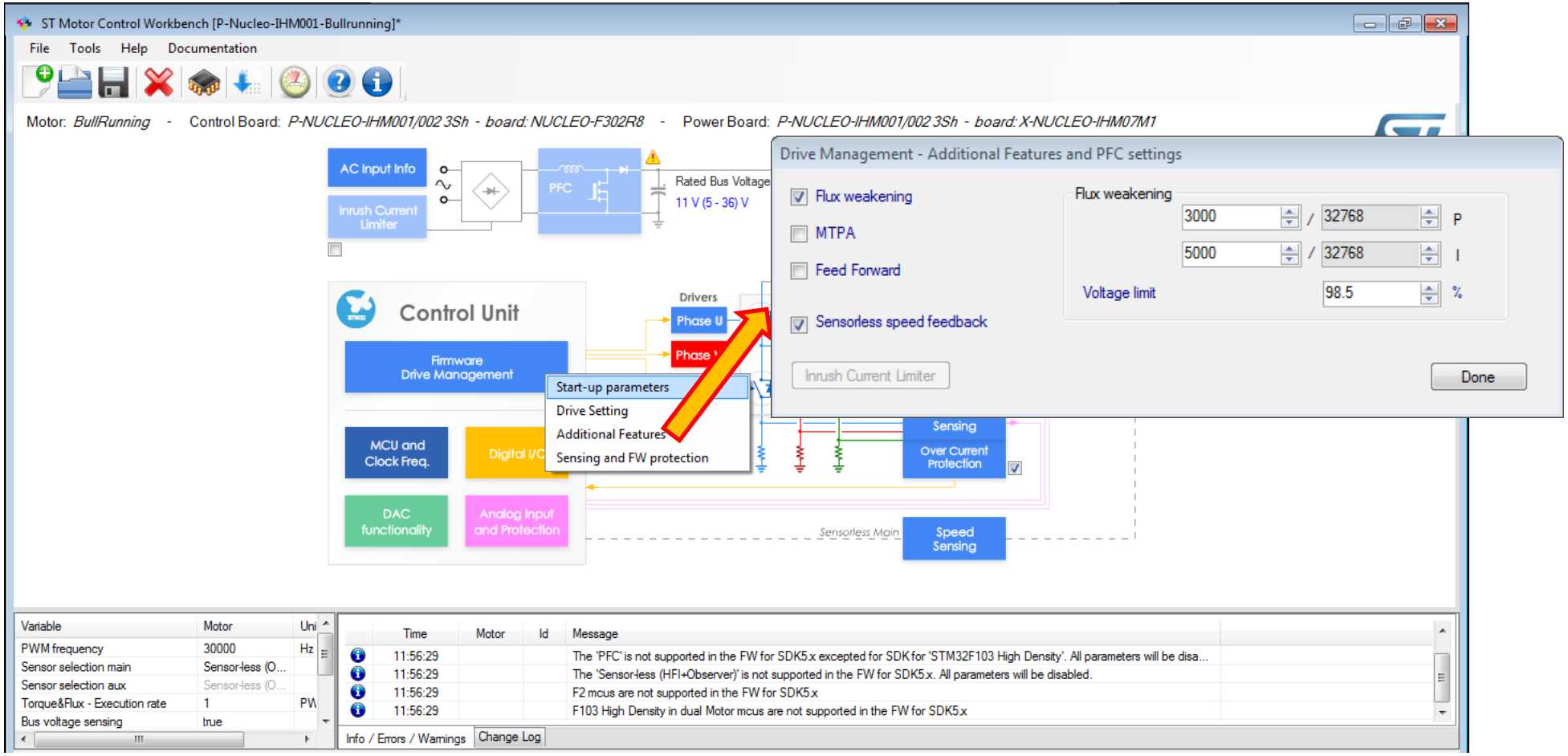
PWM载波频率



速度环Kp & Ki

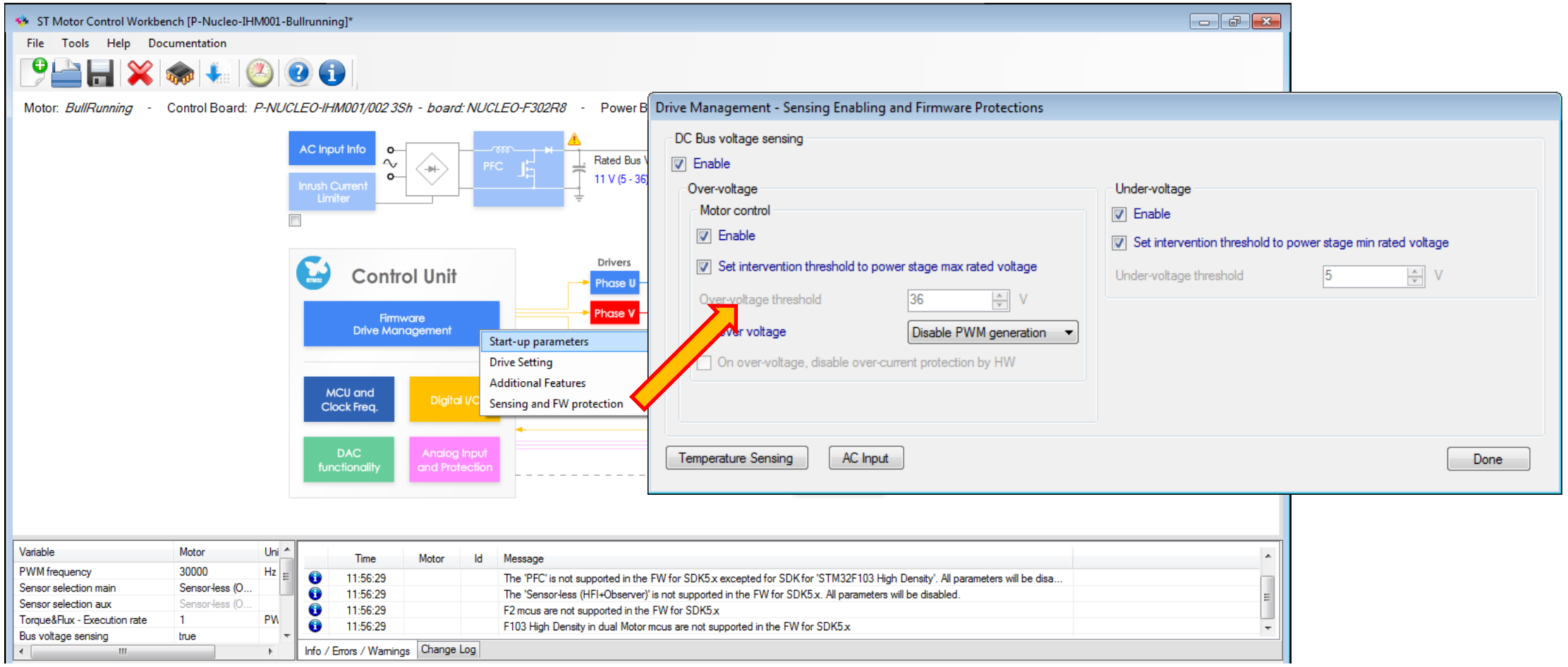
电流环Kp & Ki

life.augmented



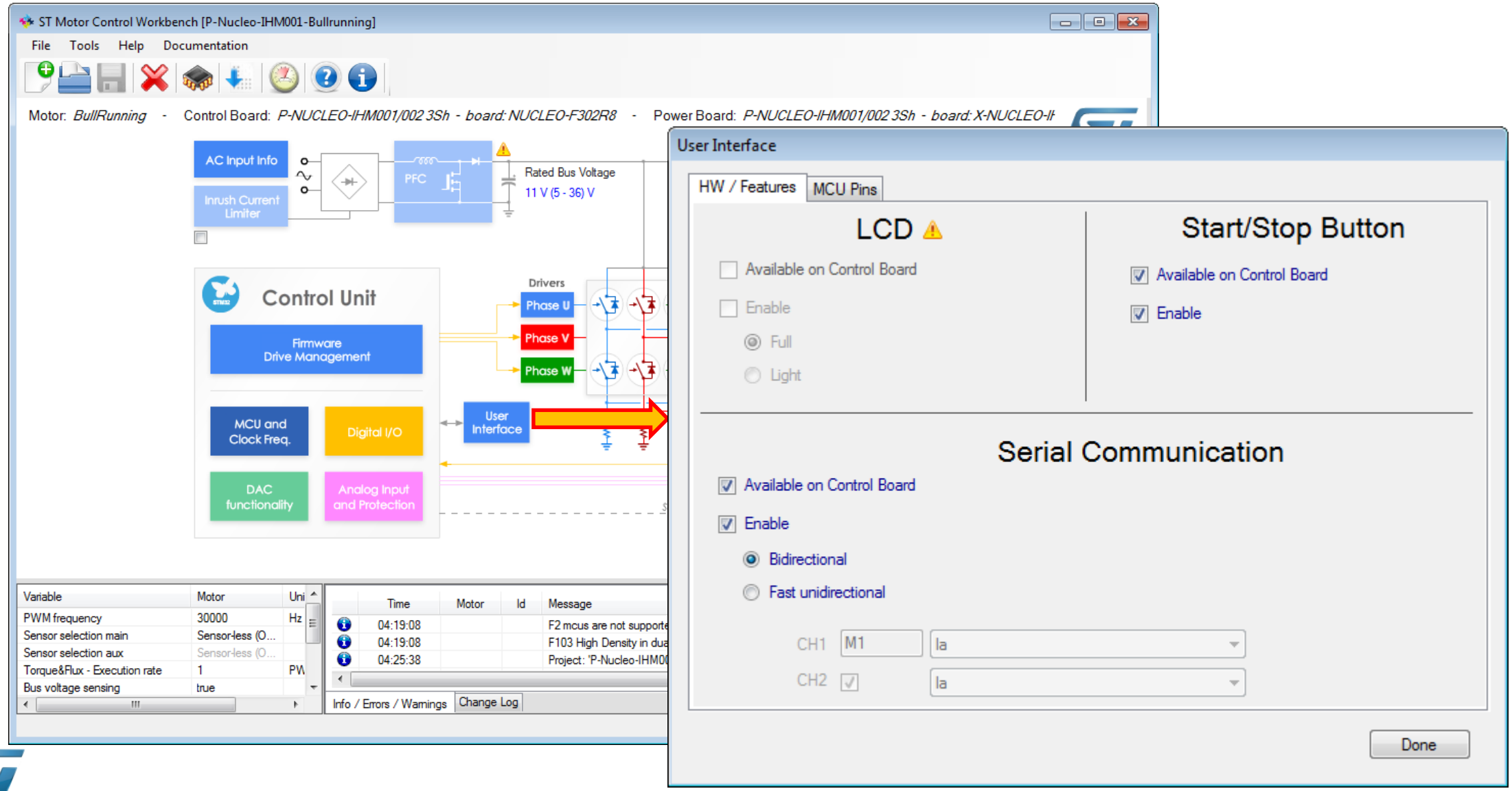
life.augmented

欠压过压检测及保护

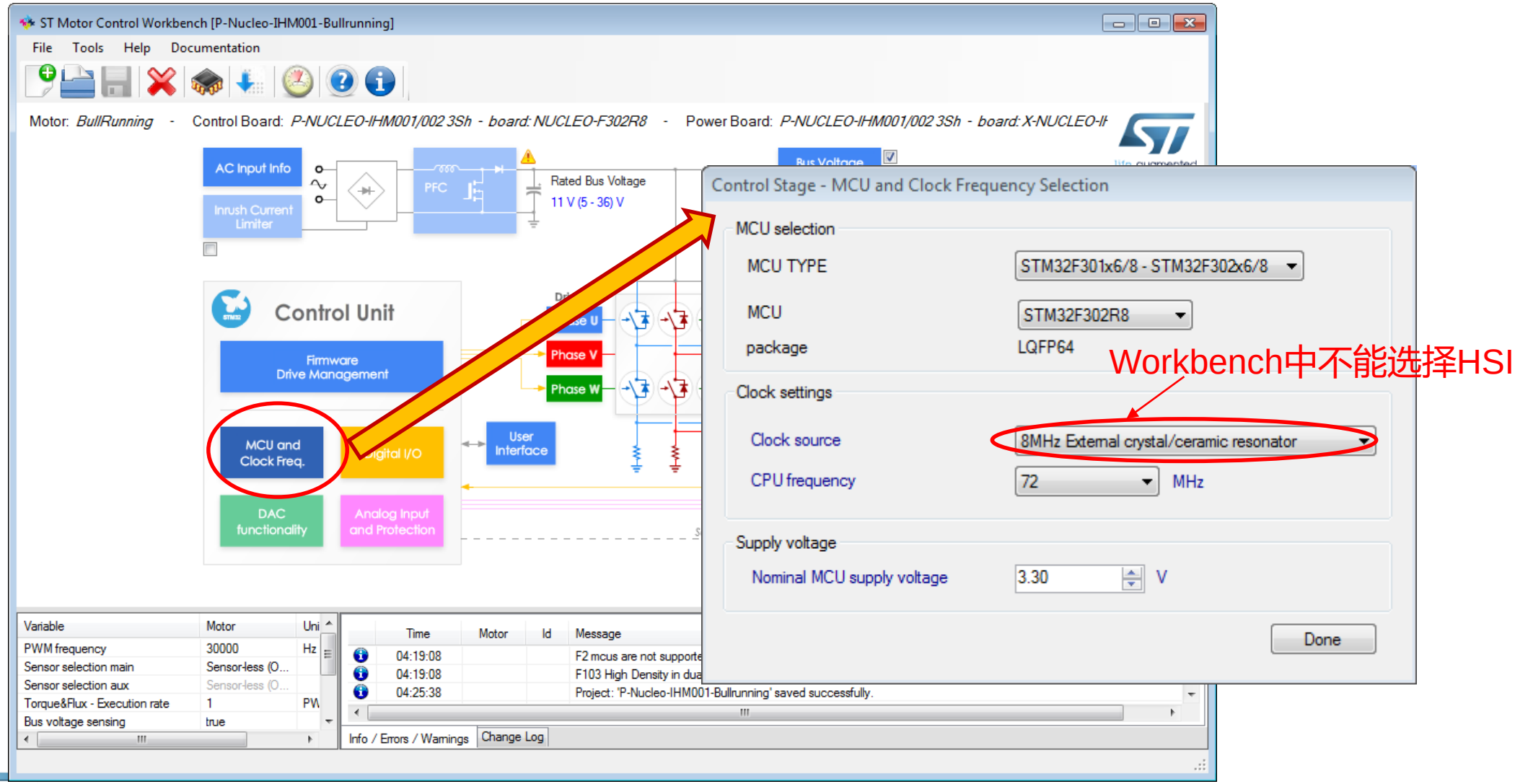


life.augmented

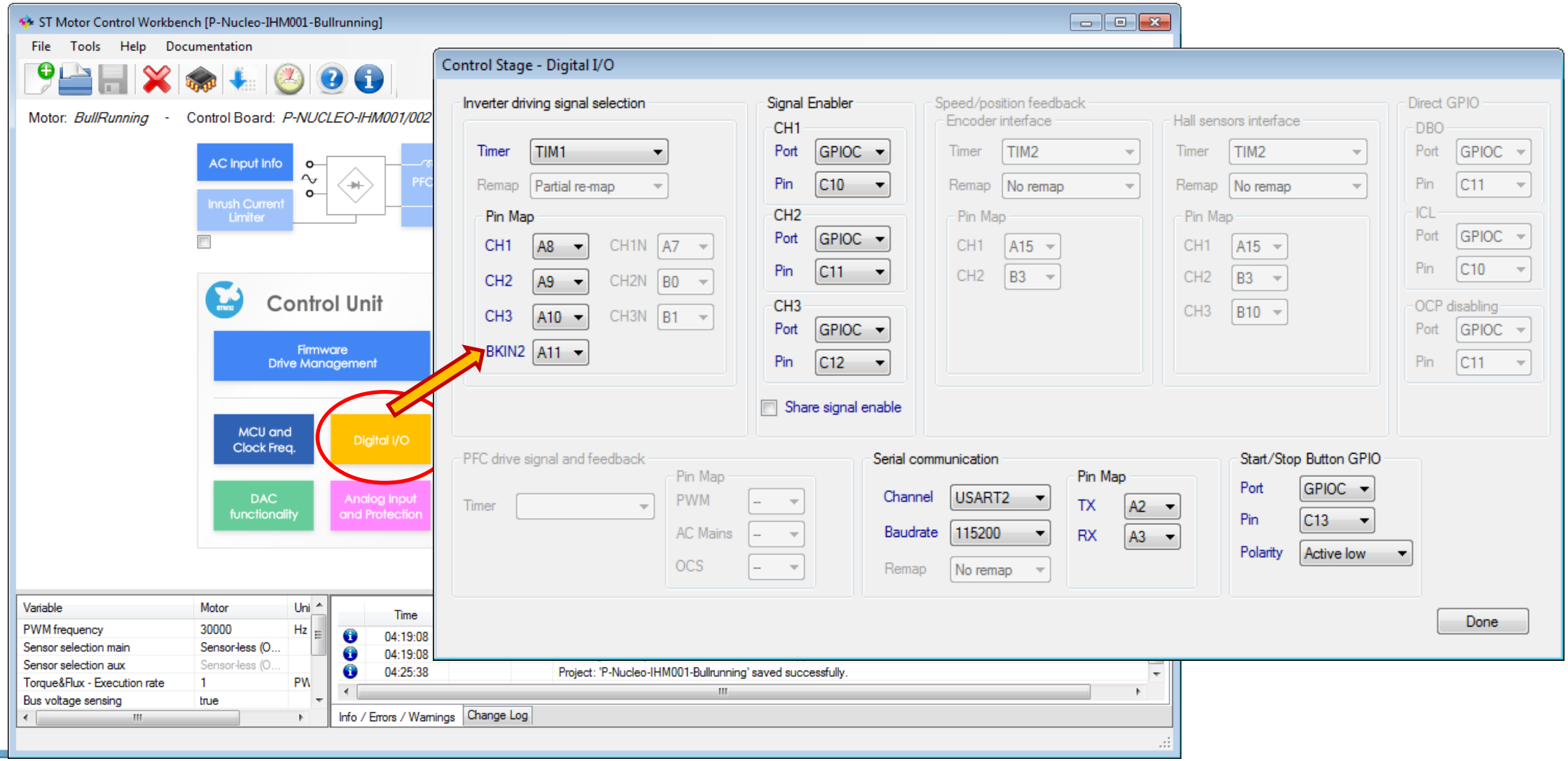
用户接口



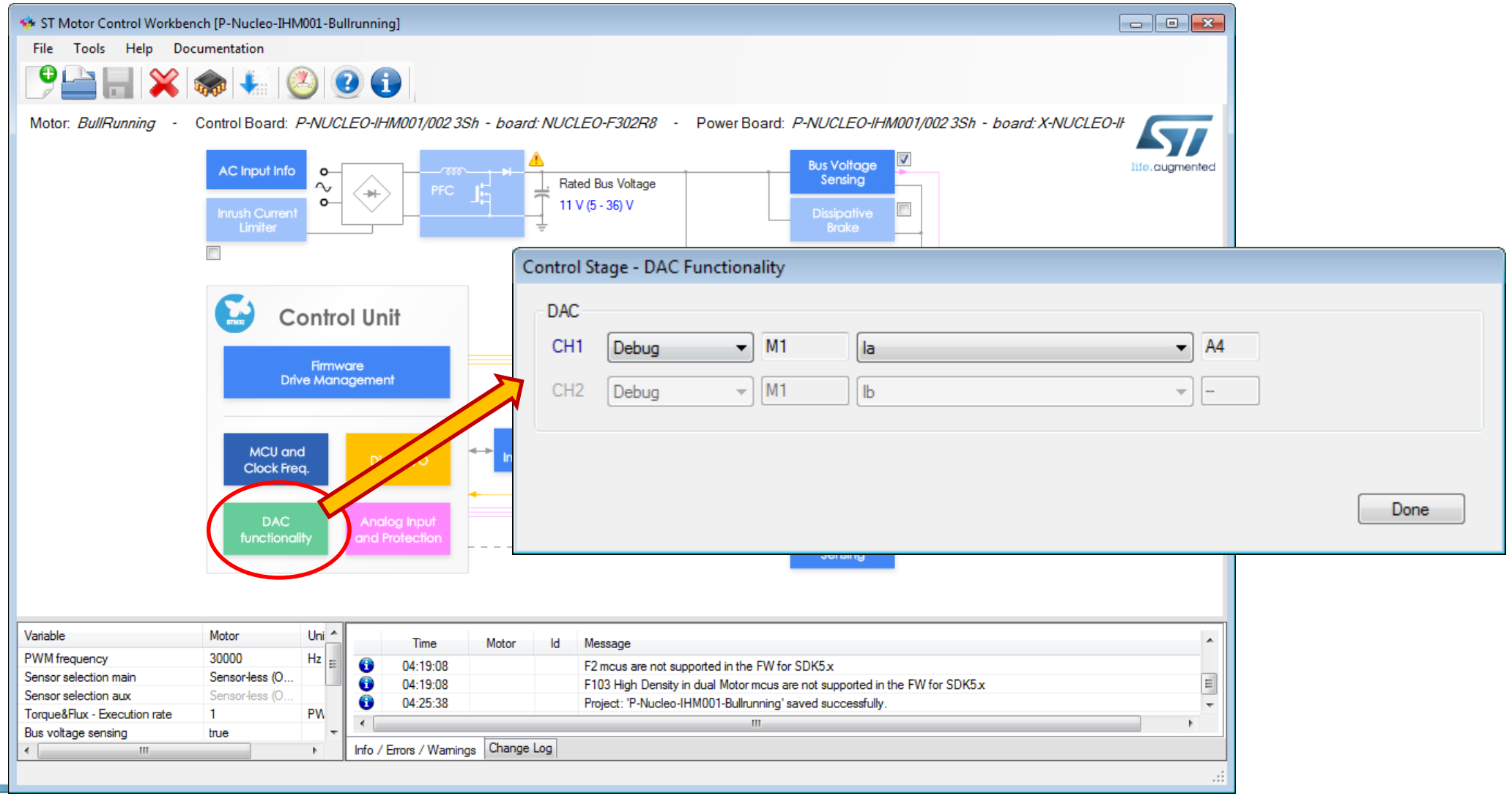
life.augmented



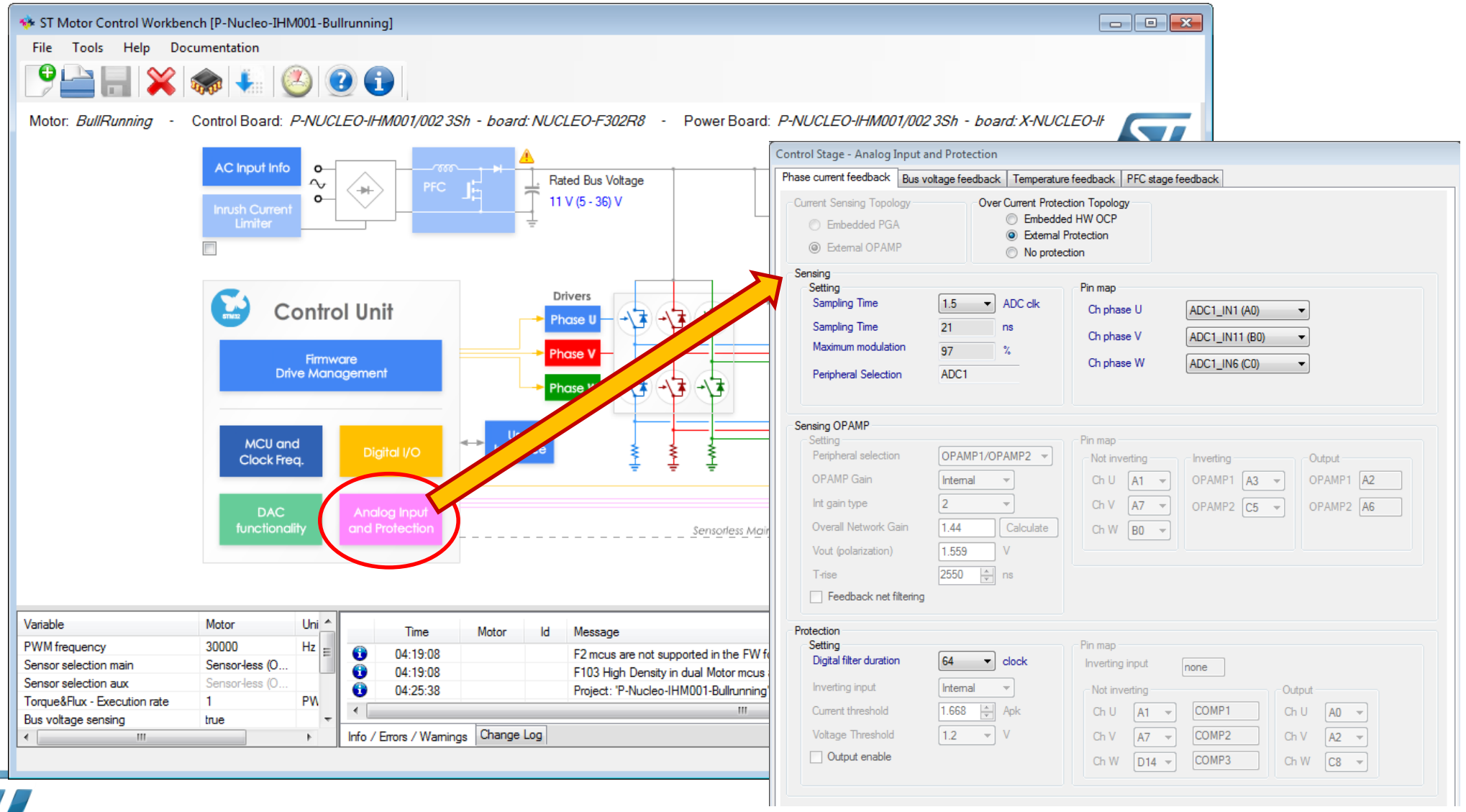
life.augmented



life.augmented

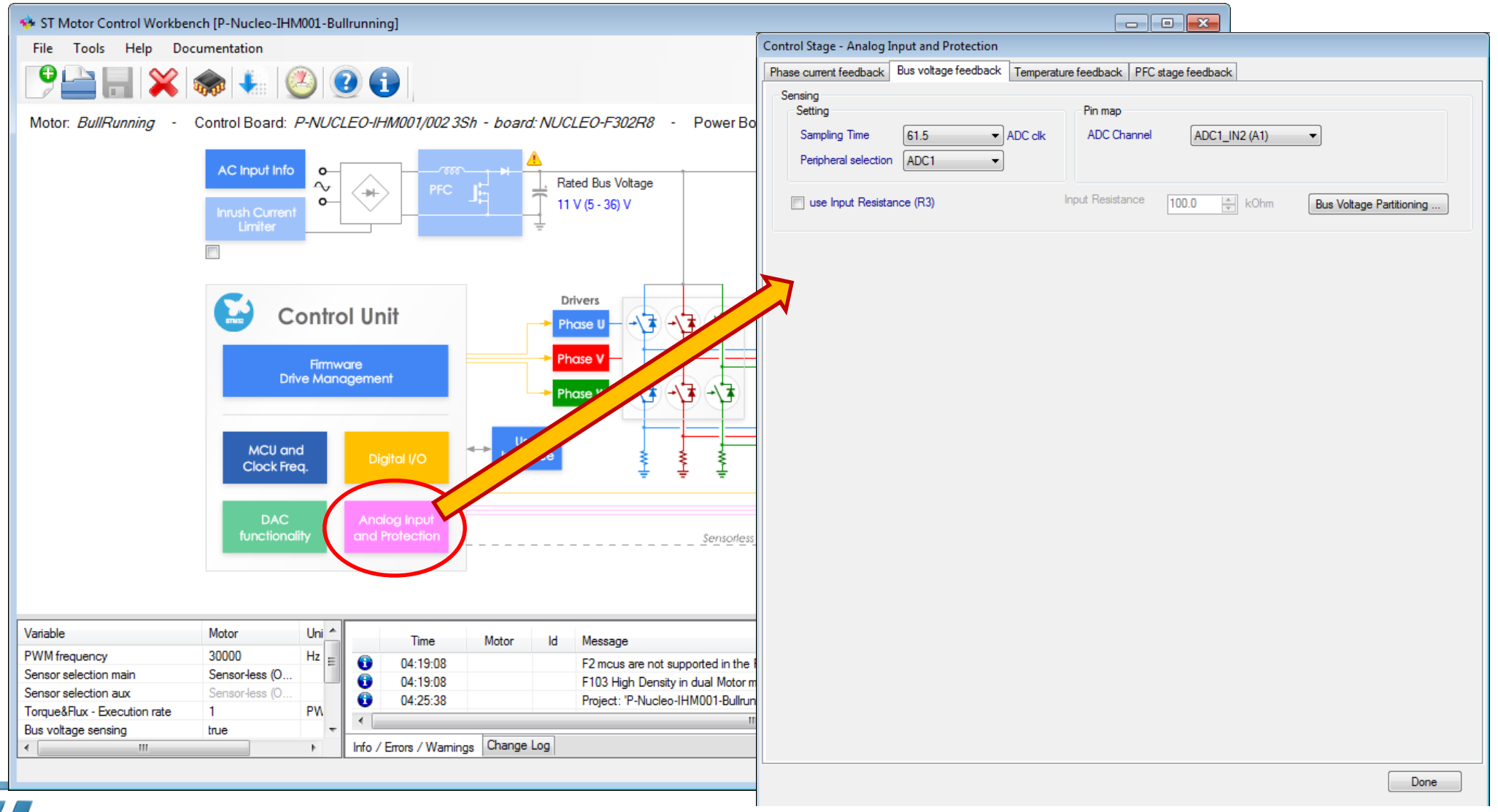


模拟输入和保护(1)



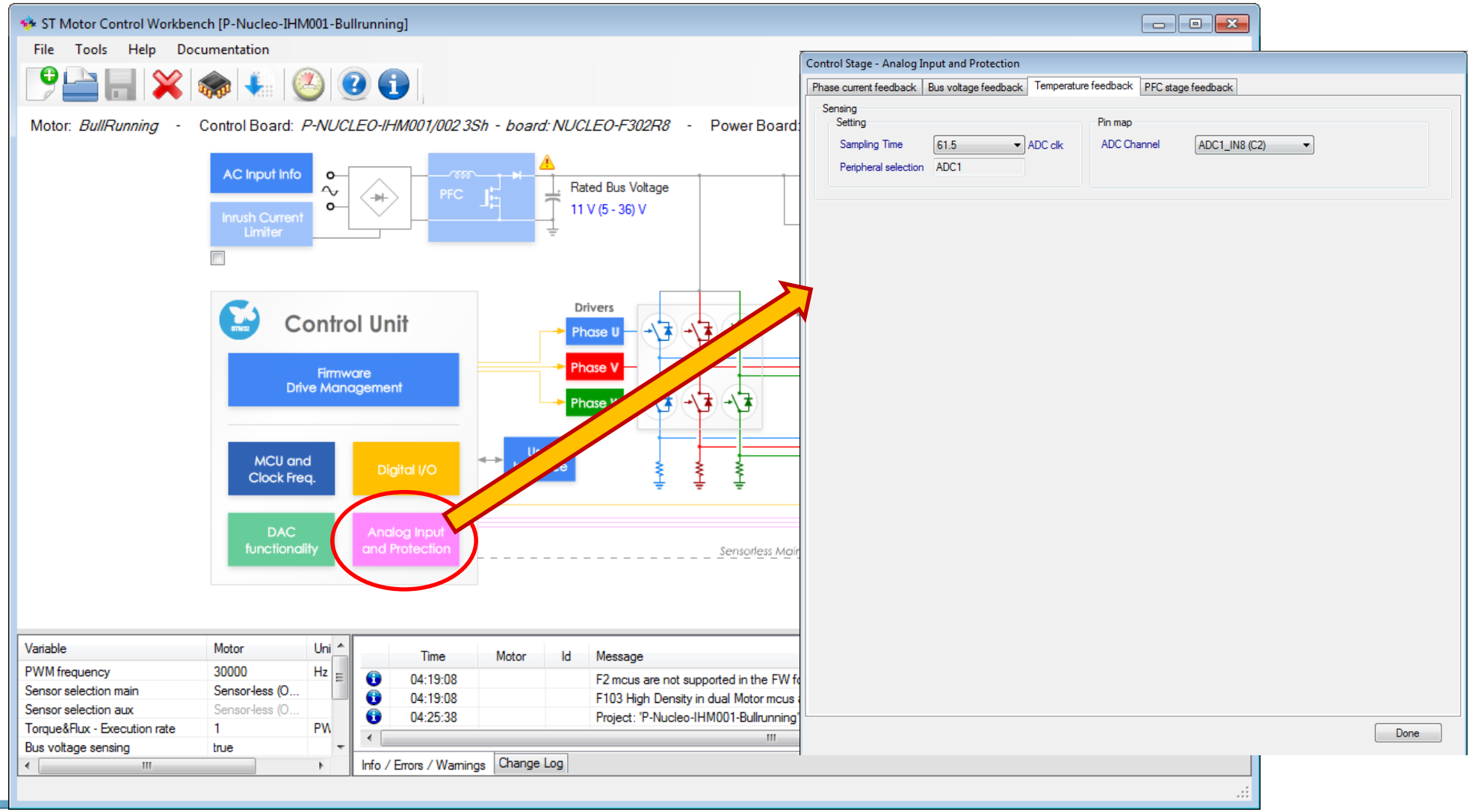
life.augmented

模拟输入和保护(2)



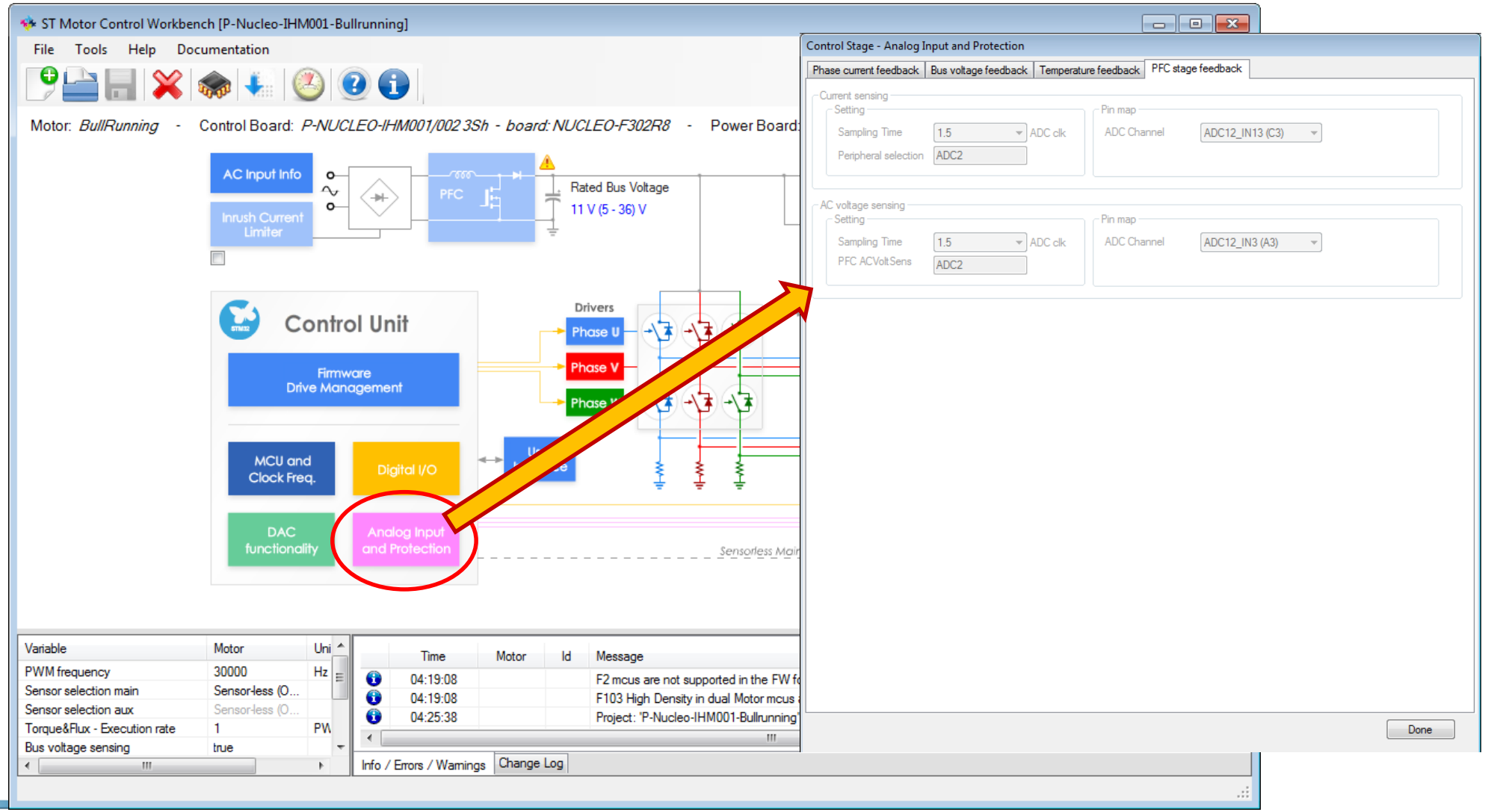
life.augmented

模拟输入和保护(3)



life.augmented

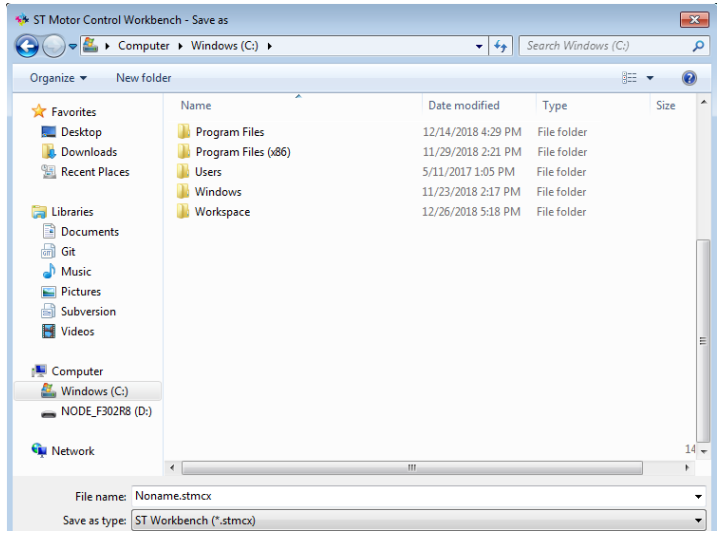
模拟输入和保护(4)



life.augmented

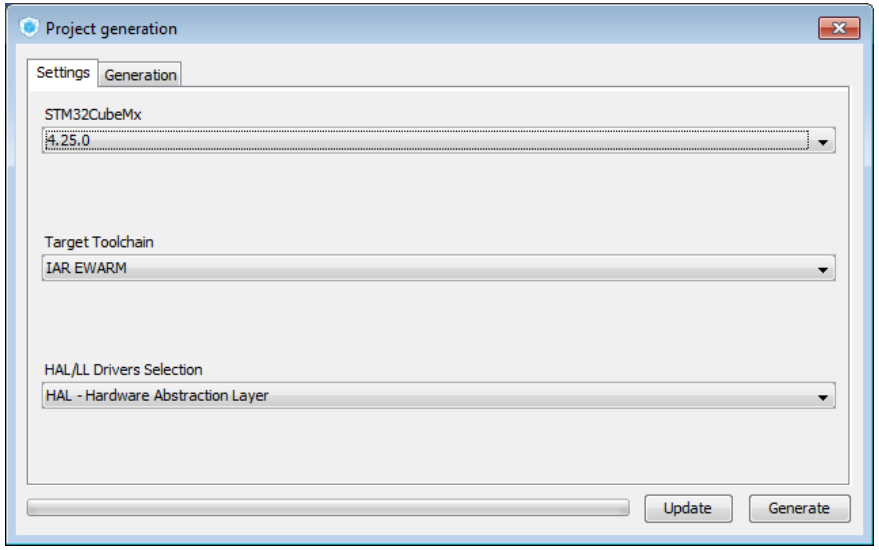
所有的参数配置完成后，点击生成图标，可根据所选的IDE生成MC应用工程：

- 用户不再需要将MC Workbench输出文件拷贝到其工程下。
- STM32CubeMX作为接口被MC Workbench后台调用，来生成选择的IDE项目工作框架。



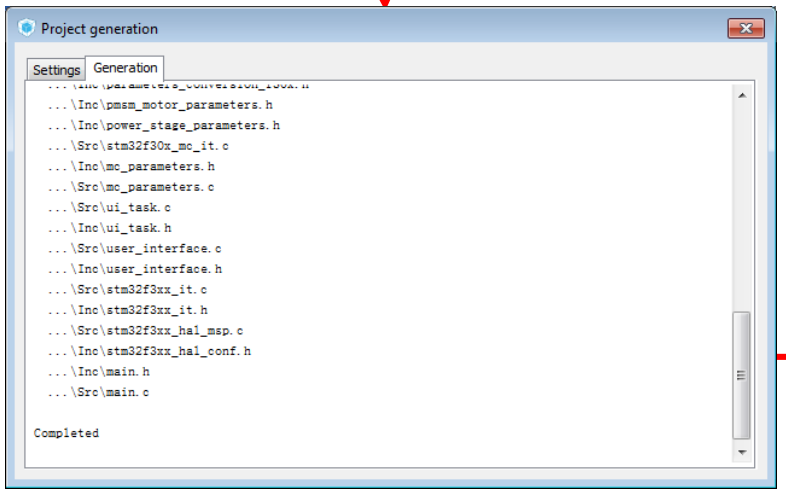
如果当前工程尚未保存，会显示一个文件管理器窗口，询问是否将当前工程设置保存为一个新项目。

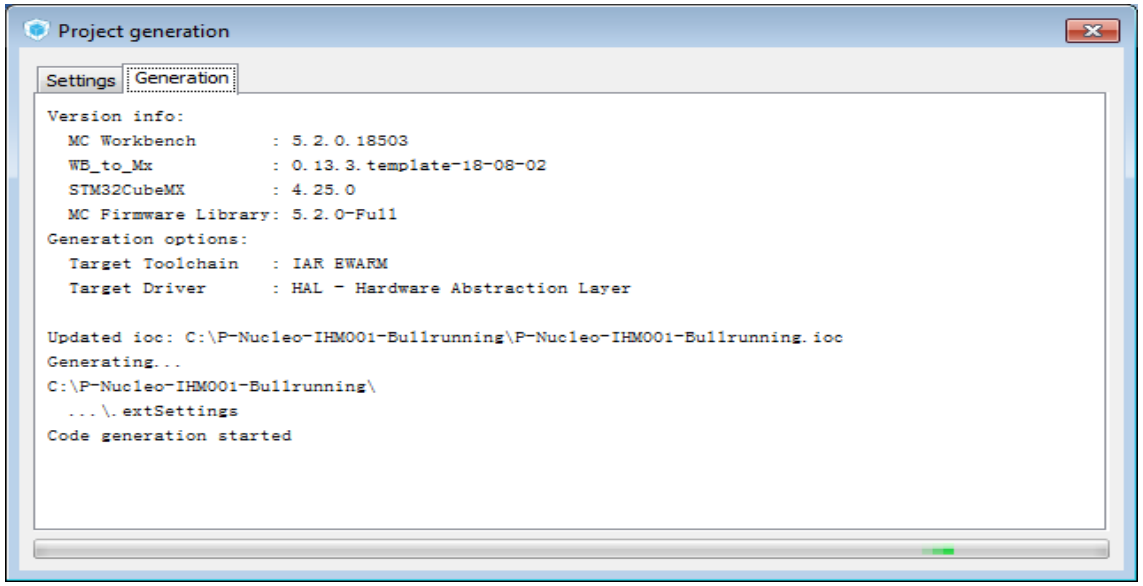
如果当前工程已保存过，会弹出右侧的窗口，进行CubeMX版本，IDE的选择和HAL/LL的选择。

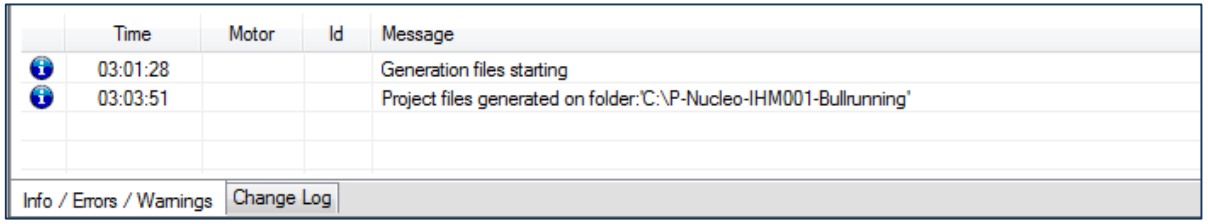


life.augmented

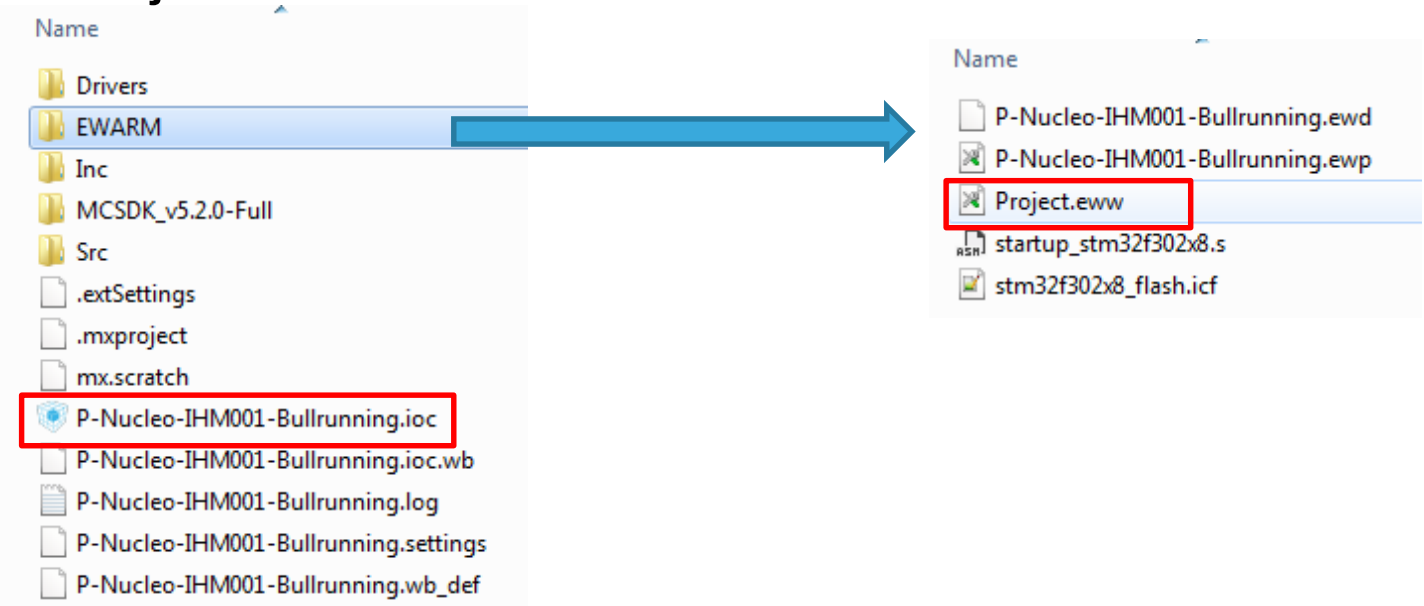
开始生成时，会显示一个进程窗口，告知用户脚本在执行，完成后的提示窗口如下图所示，可关闭该窗口。用户信息表也随之更新。







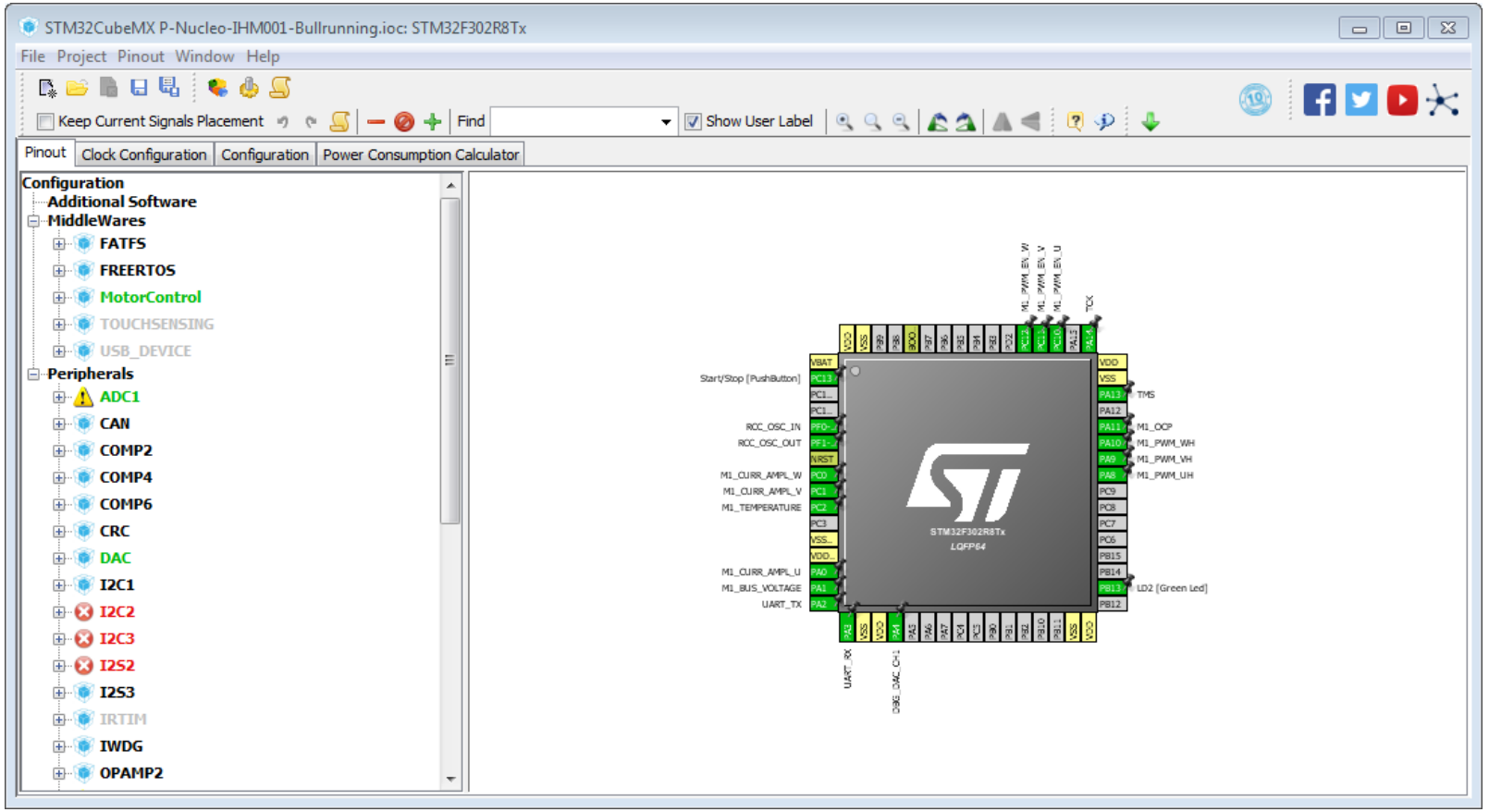
生成的IAR工程文件保存在如下路径：
$ProjectFolder\EWARM



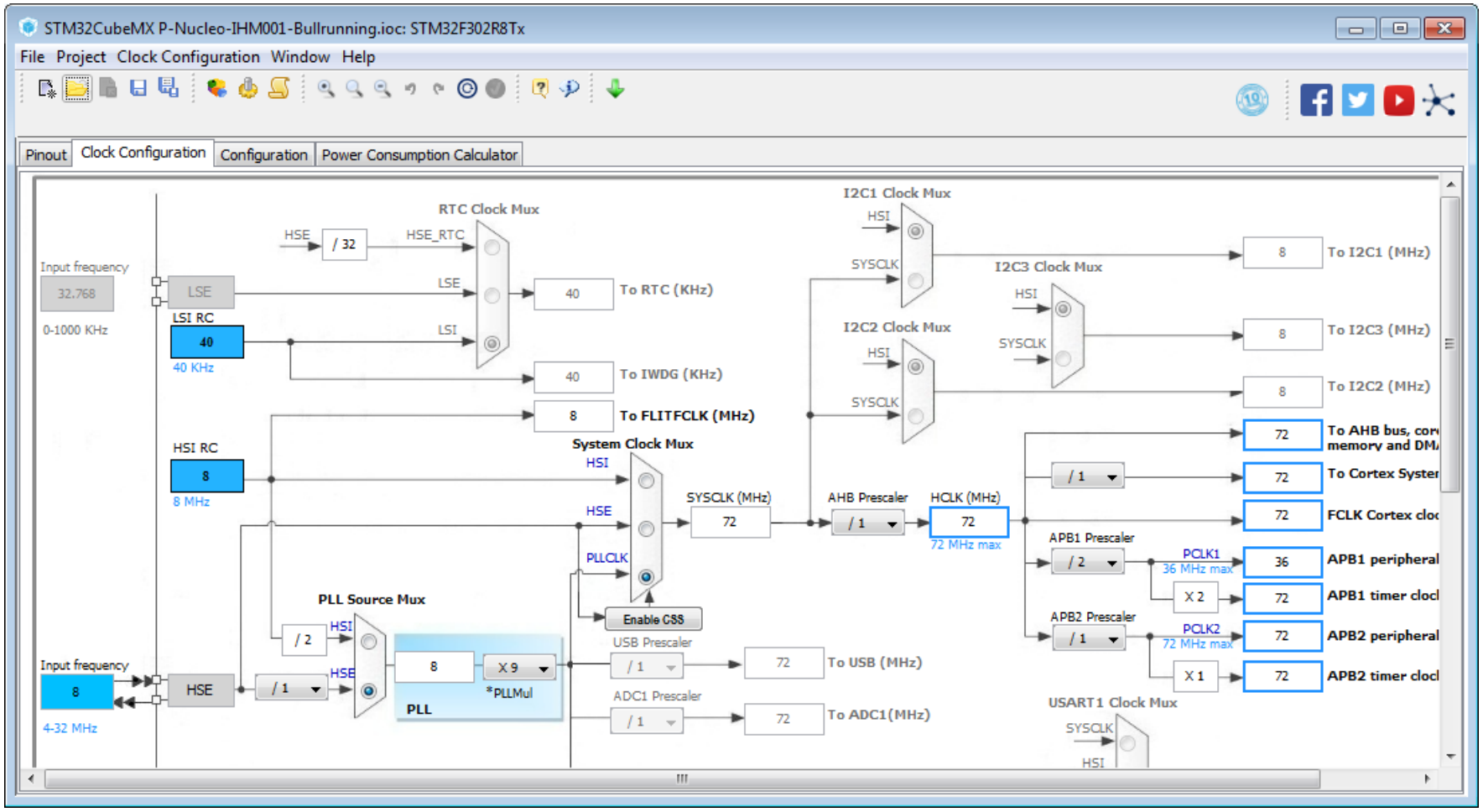
如果不需要配置其他外设，可直接打开.eww文件。
如果需要配置其他外设，请用CubeMX打开.ioc文件进行添加。

life.augmented

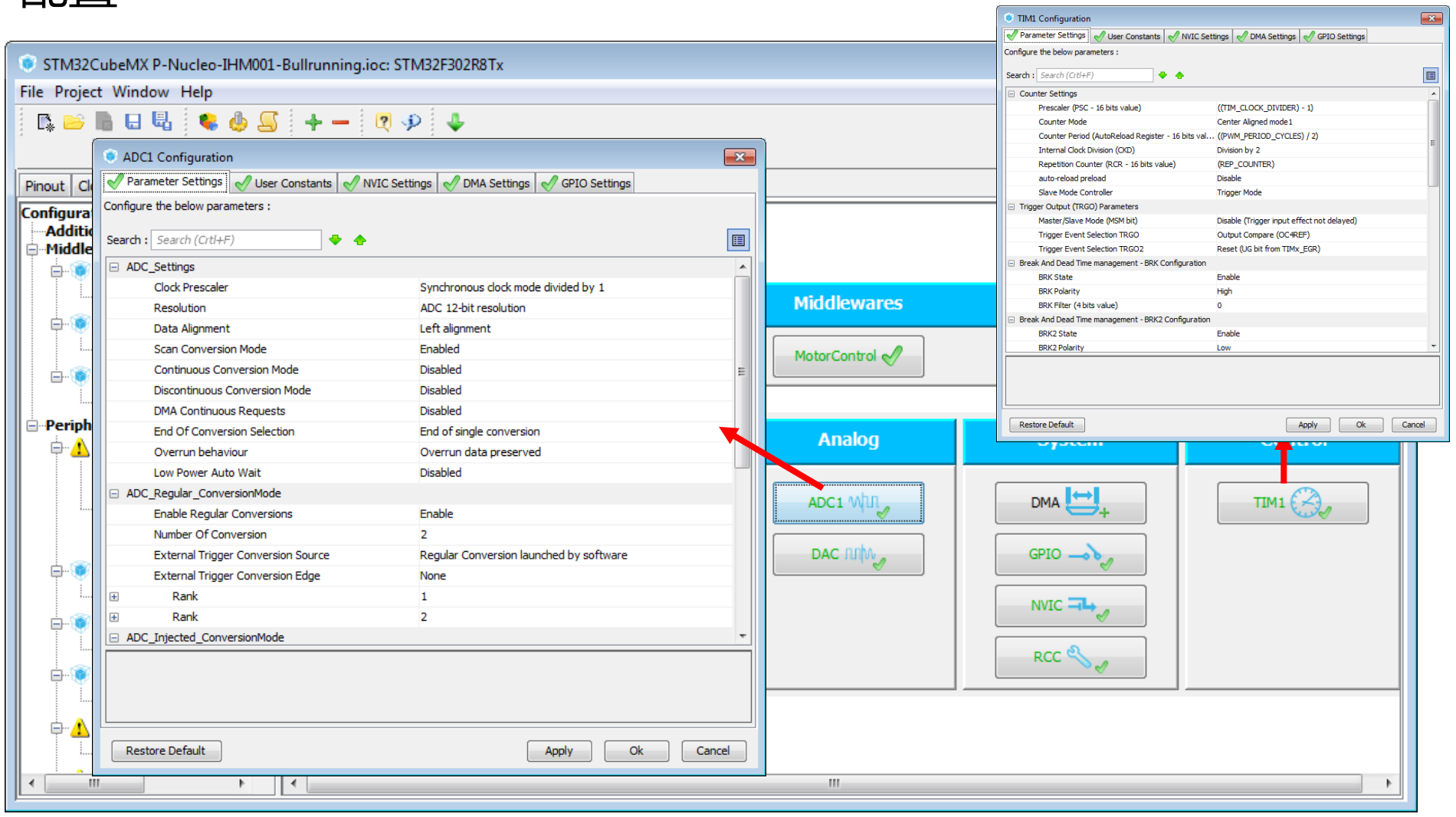
打开.ioc文件，STM32CubeMX将被打开。
图形化的引脚分配配置



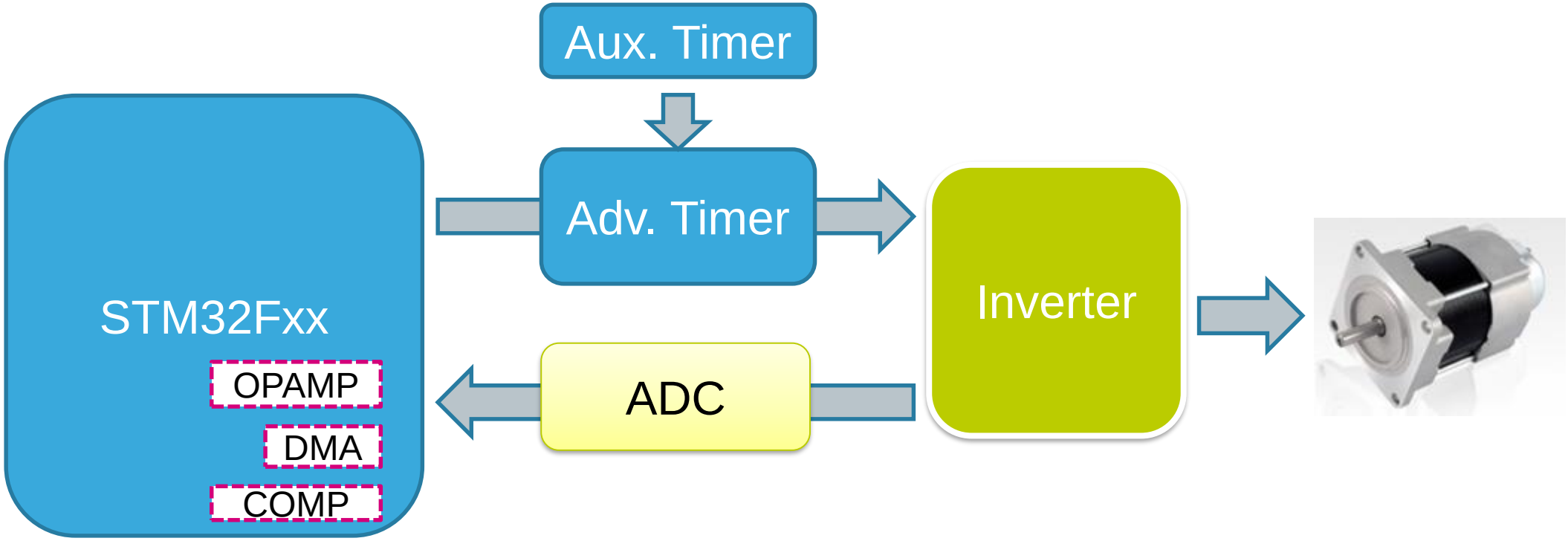
时钟树结构



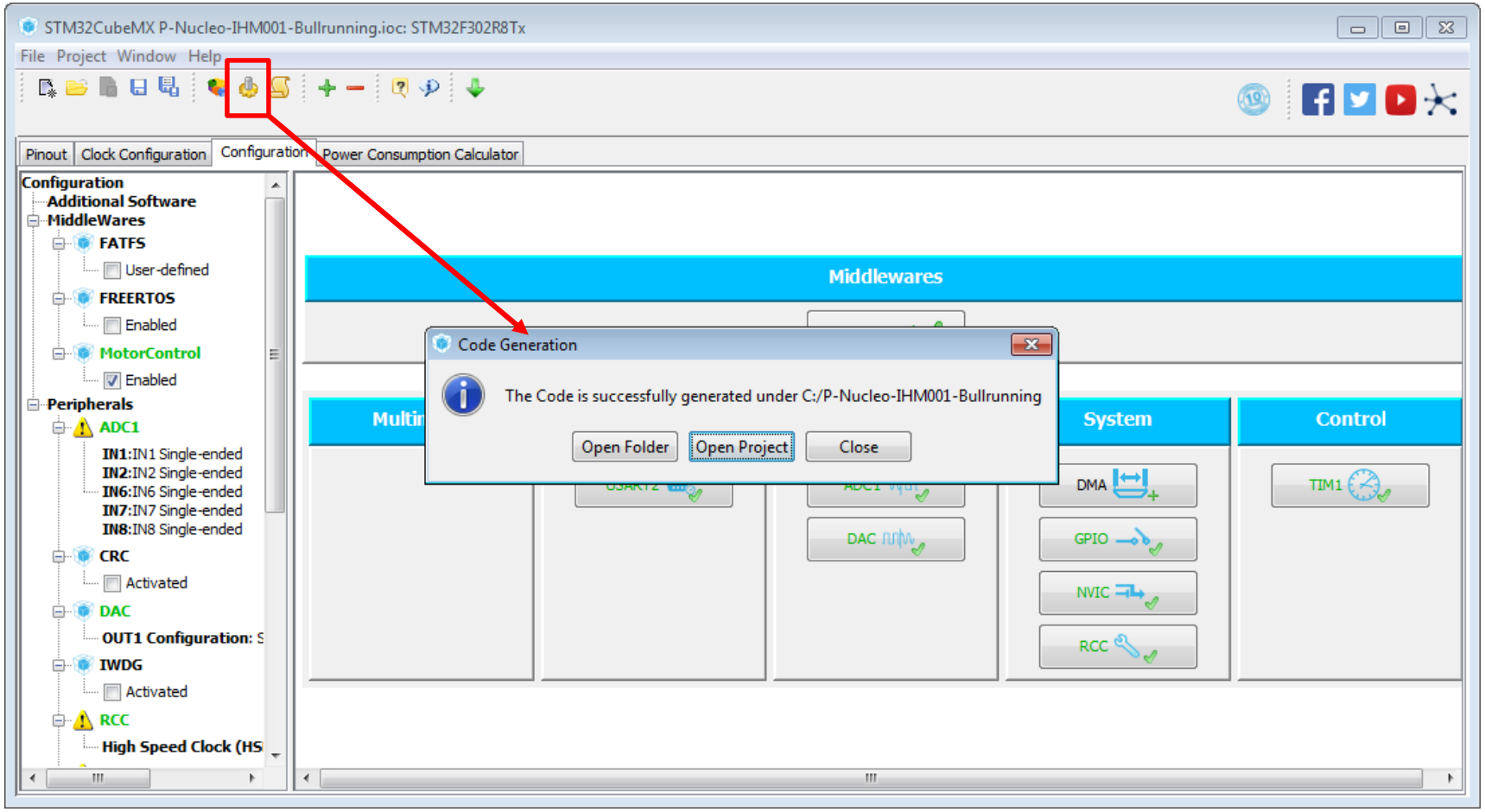
配置



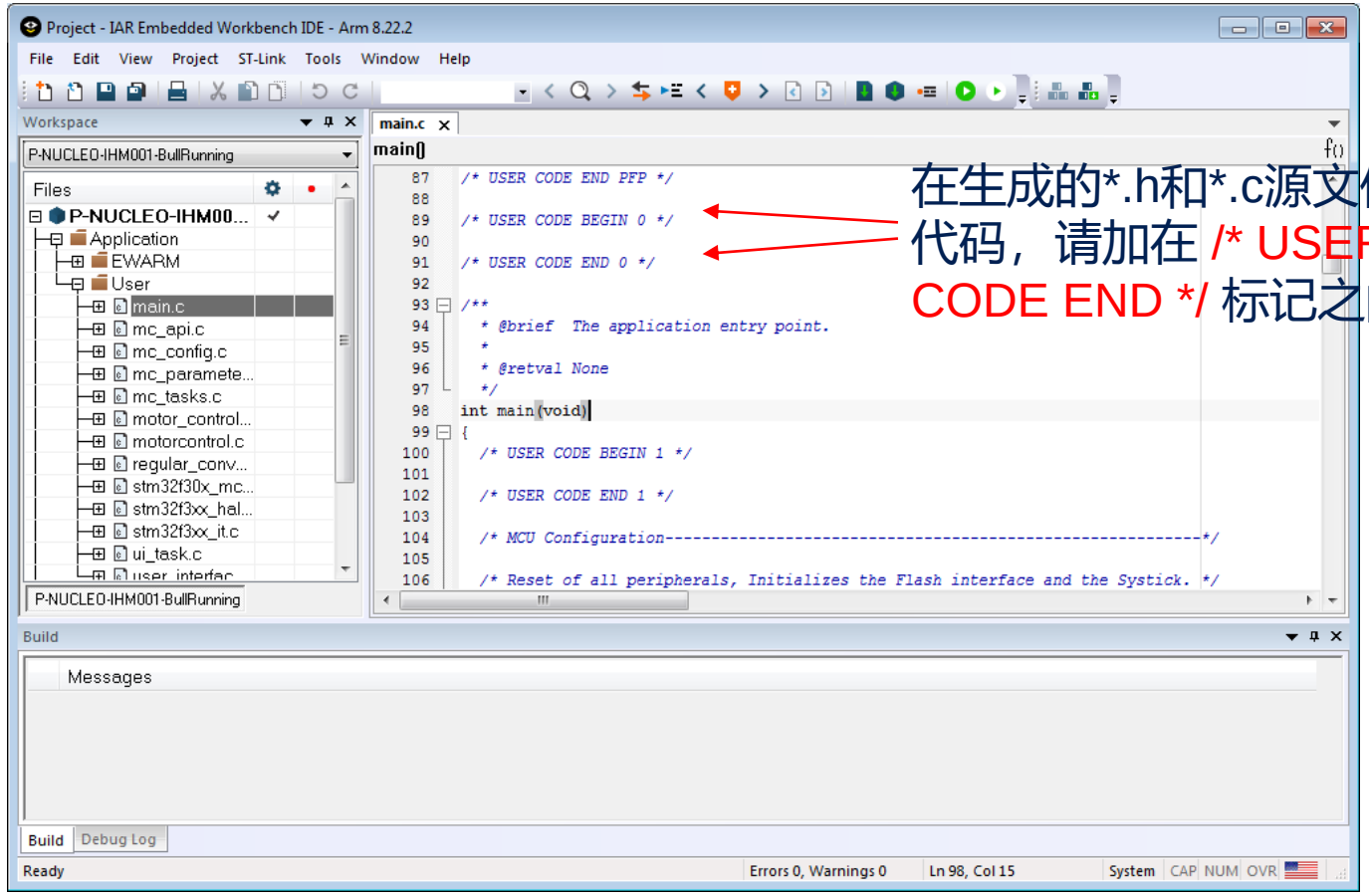
在CubeMX中请不要修改电机控制相关外设的设定!



代码生成:



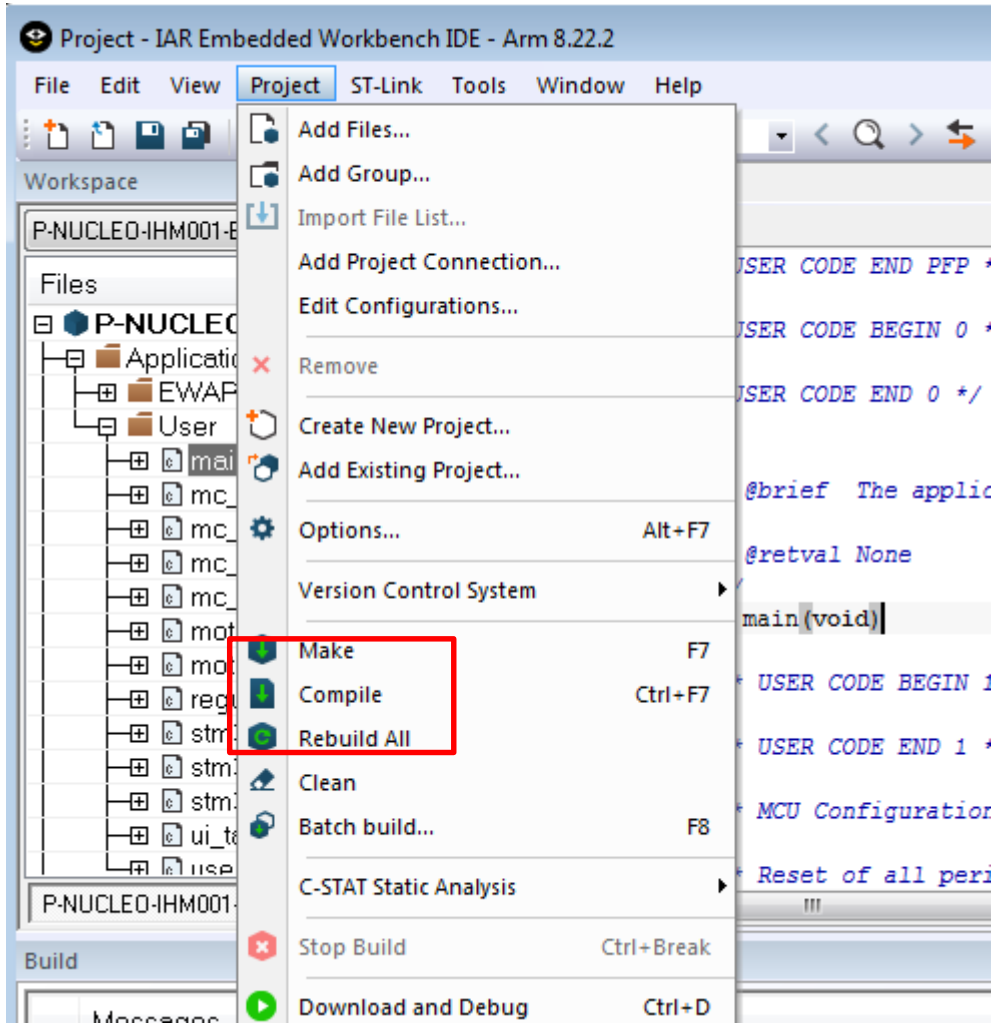
IAR Embedded Workbench for Arm (v8.xx)

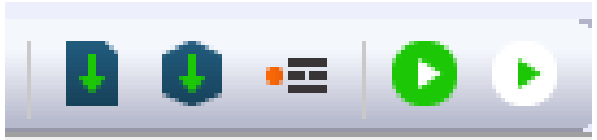


在生成的*.h和*.c源文件中，用户如果添加自己的应用代码，请加在 /* USER CODE BEGIN */和/* USER CODE END */ 标记之间。

在IAR Embedded Workbench for Arm v8.xx中，C编译器的优化选择请设定为 ***Speed.***

life.augmented





Compile

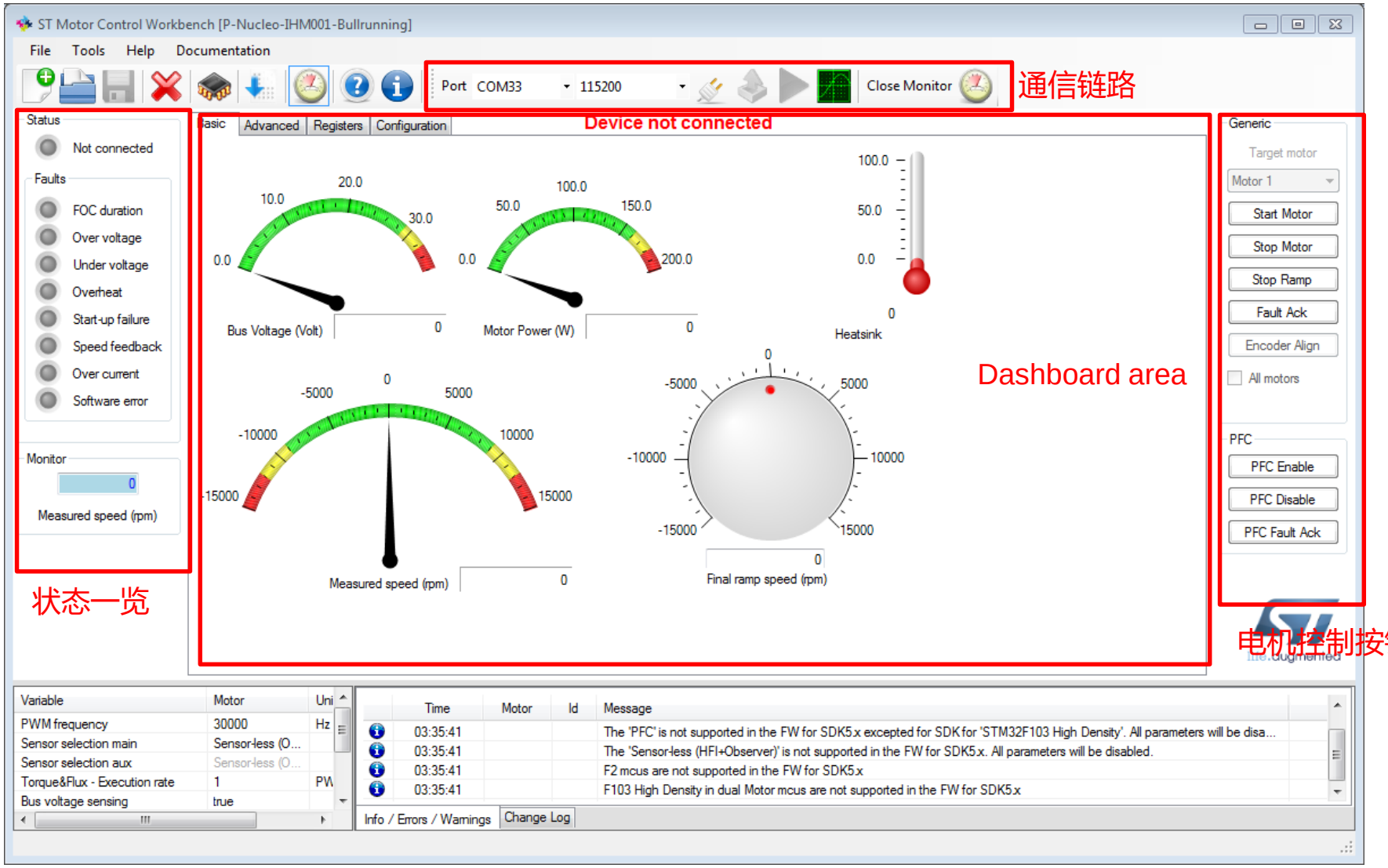
Make

Toggle Breakpoint

Download and Debug

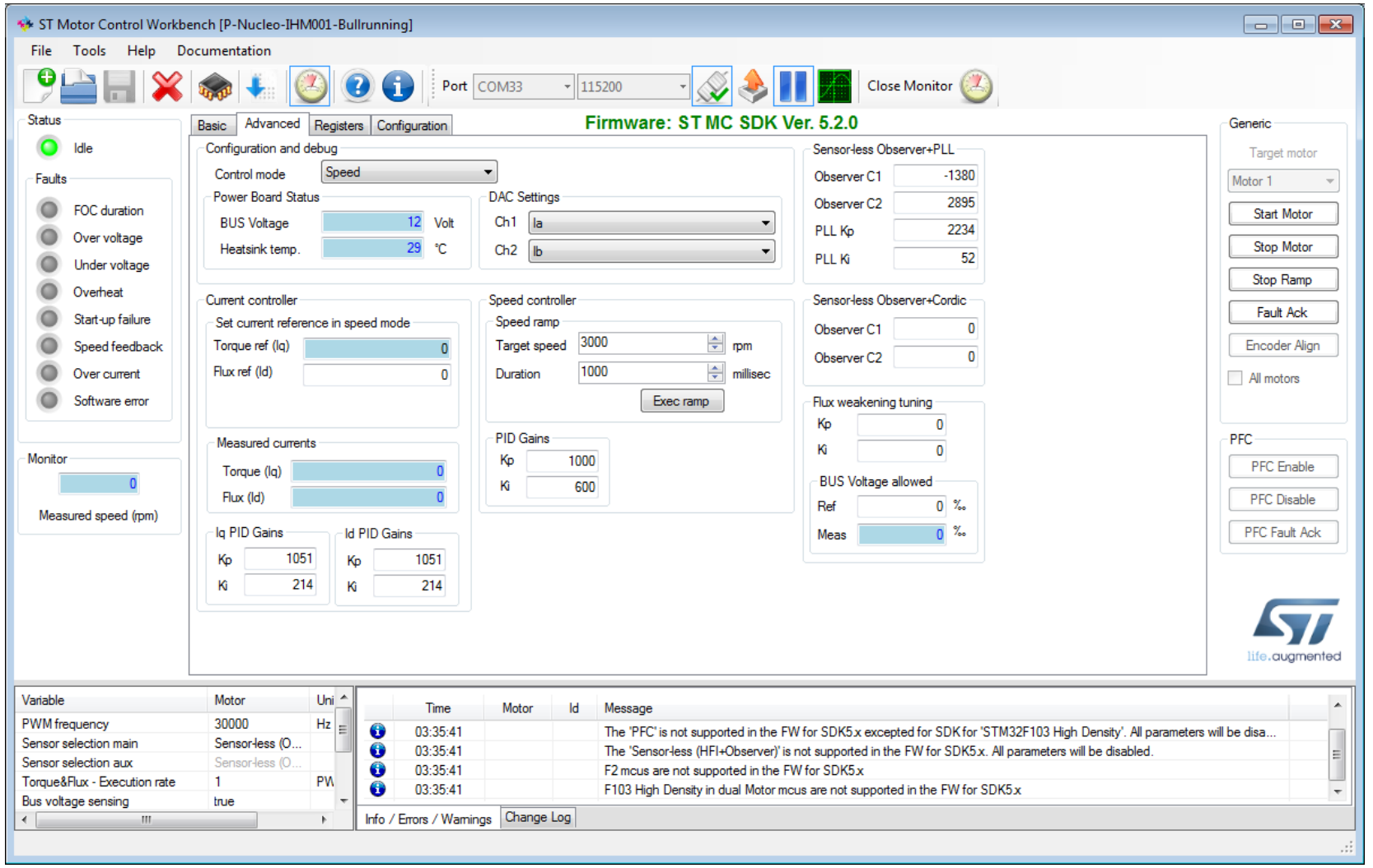
Debug without Downloading

电机控制和监控(1)



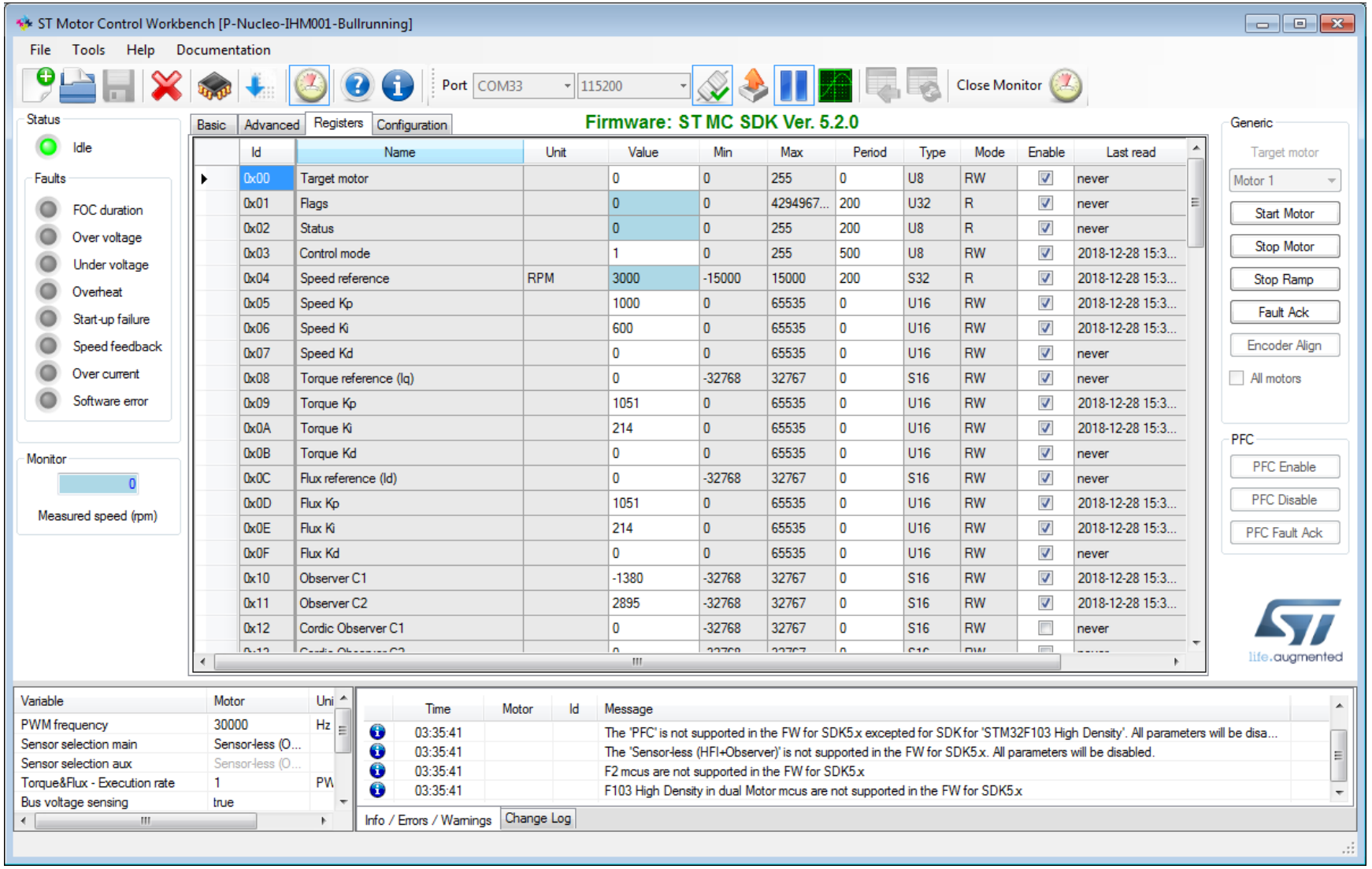
life.augmented

电机控制和监控(2)

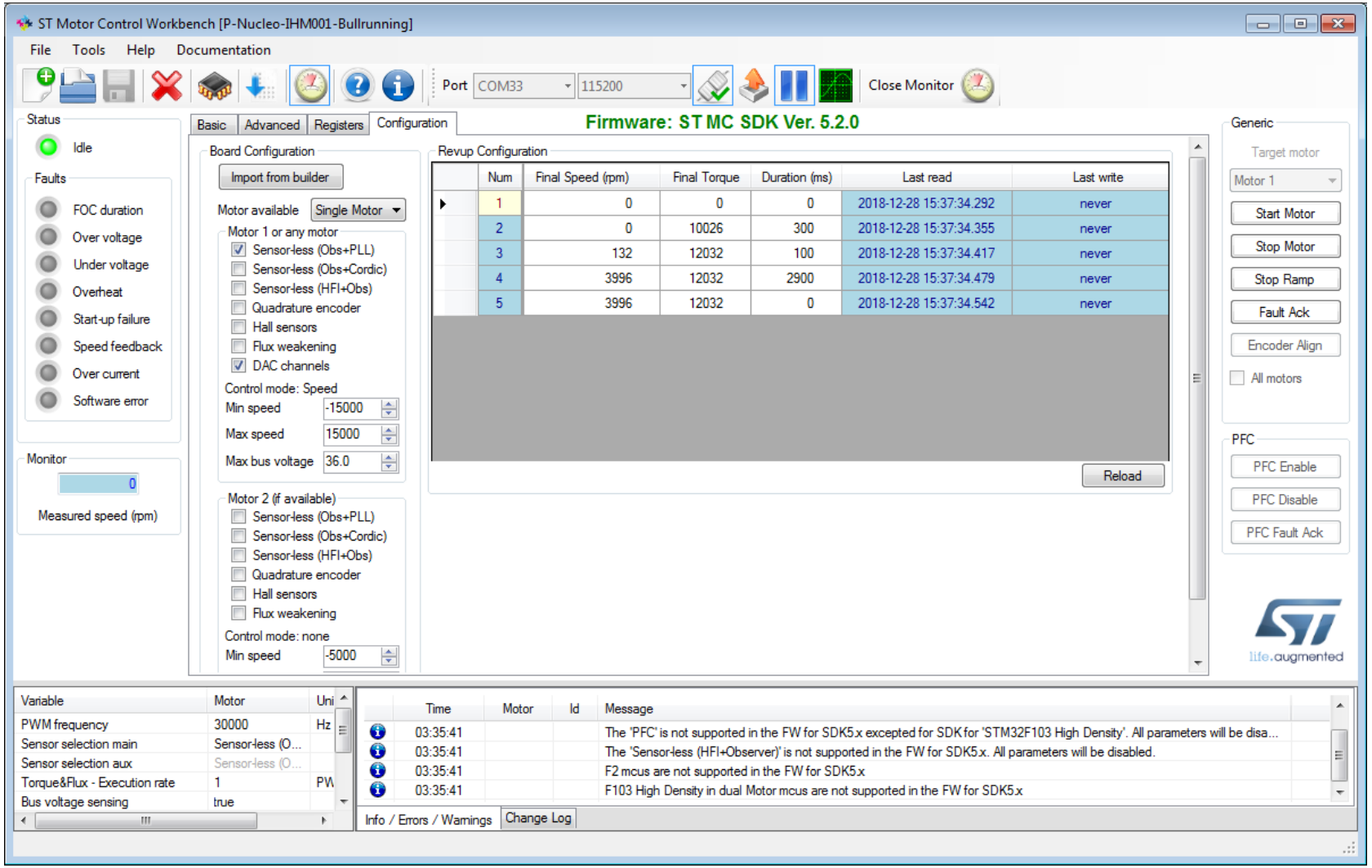


life.augmented

电机控制和监控(3)



life.augmented

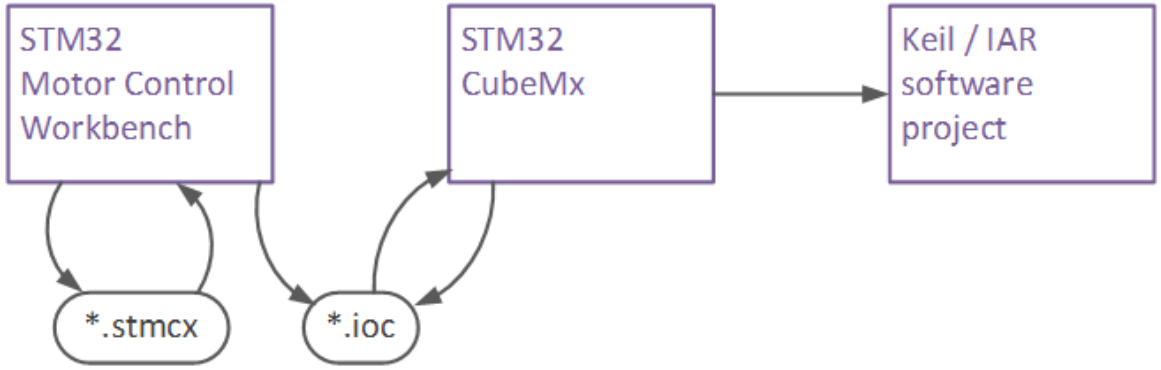


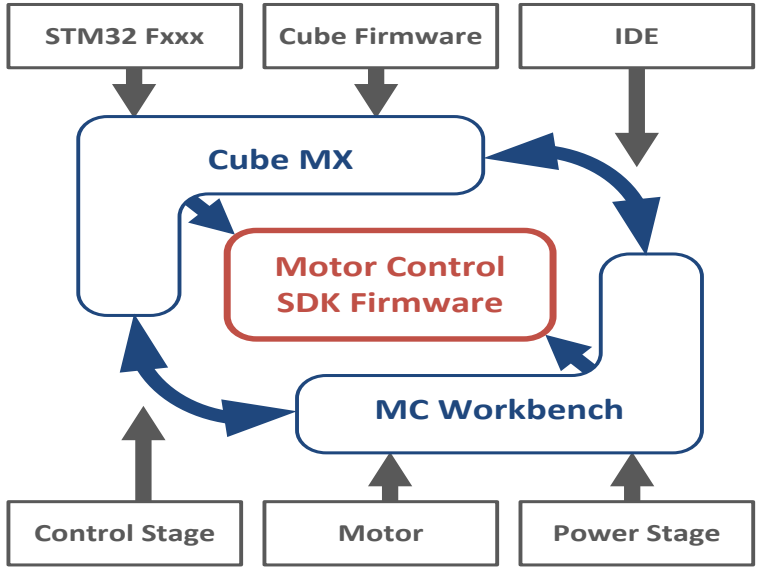
ST MC SDK5.x固件详解

- ST MC SDK5.x WB应用指南
  - 软件工具的下载和安装
  - ST MC Workbench及关键配置参数
  - MC Project 的生成，编译和下载
  - 电机控制及监控
- ST MC SDK5.x固件详解
  - 程序架构
  - 组件
  - 例程代码讲解
  - 开发实战
    - 如何向例程中添加外设和自己的代码
    - Step-by-Step添加一段闪灯代码

life.augmented

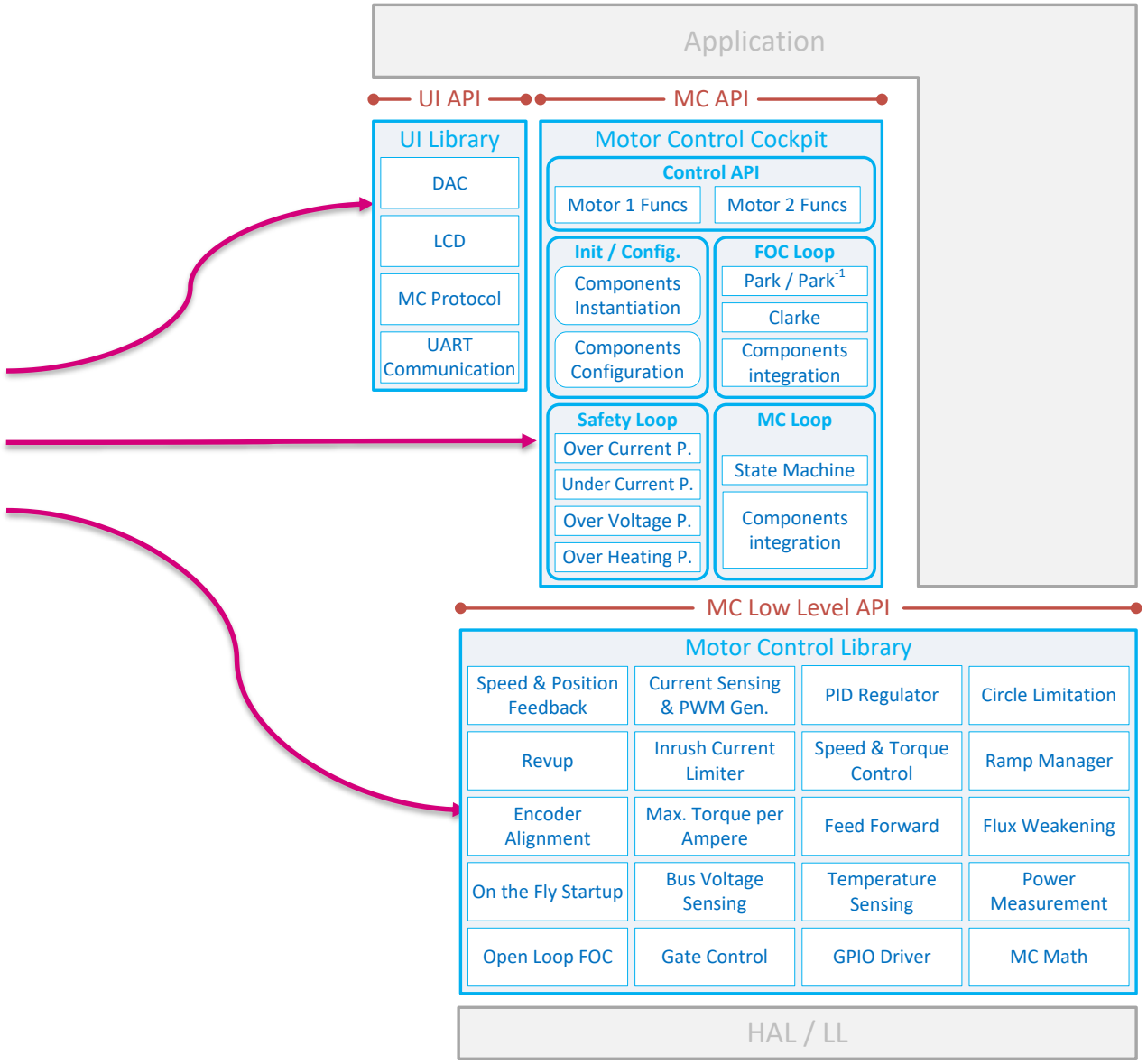
代码生成流程





整个软件库由三部分组成

- **用户界面库** UI Library
- **电机驾驶舱** Motor Control Cockpit
- **电机控制库** Motor Control Library

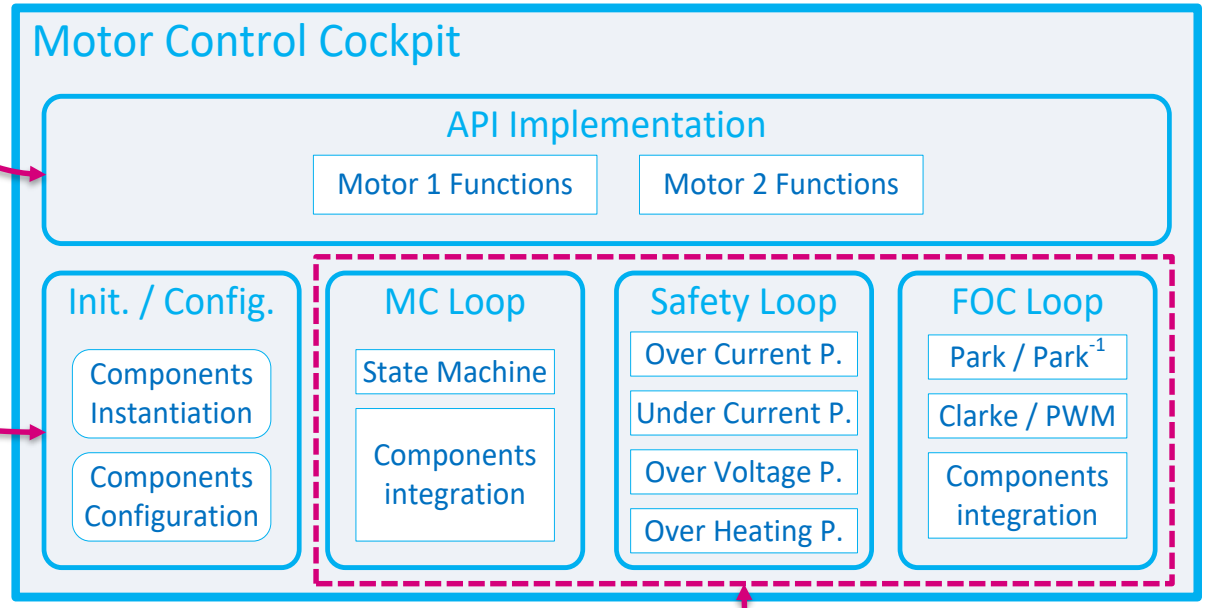


life.augmented

电机驾驶舱 Motor Control Cockpit 由三部分构成

电机控制接口通过 MC API来实现

电机控制配置实例化并配置所有需要的组件。

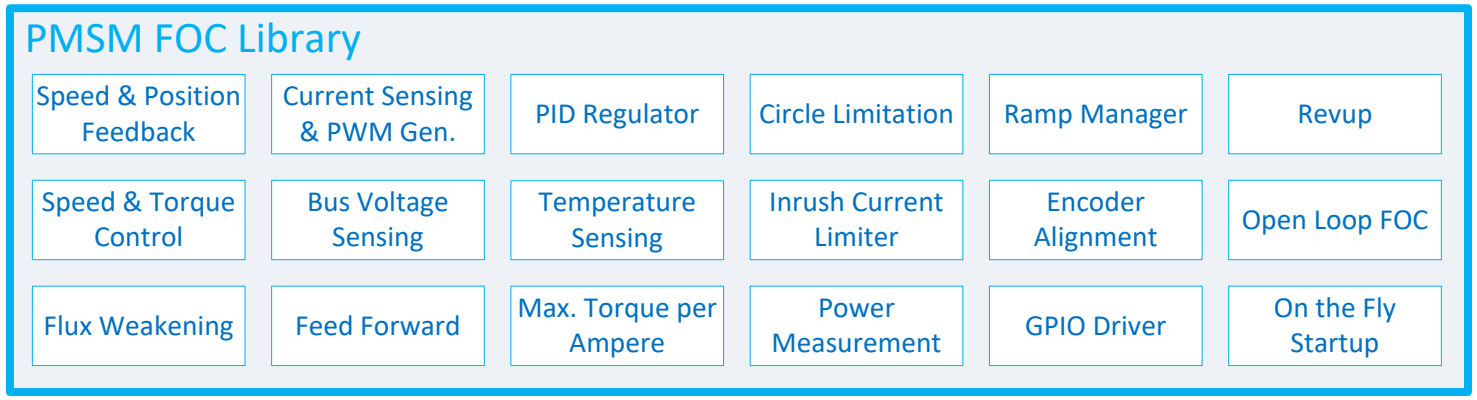


电机控制动态实现对电机动的动态控制：

- FOC控制环路(高频任务)
- 电机控制环路(中频任务)
- 安全控制环路 (安全任务)

life.augmented

电机控制库



是诸多组件的集合。每一个组件实现电机控制的一个功能例如，速度和位置检测, 电流检测, PID算法等等…

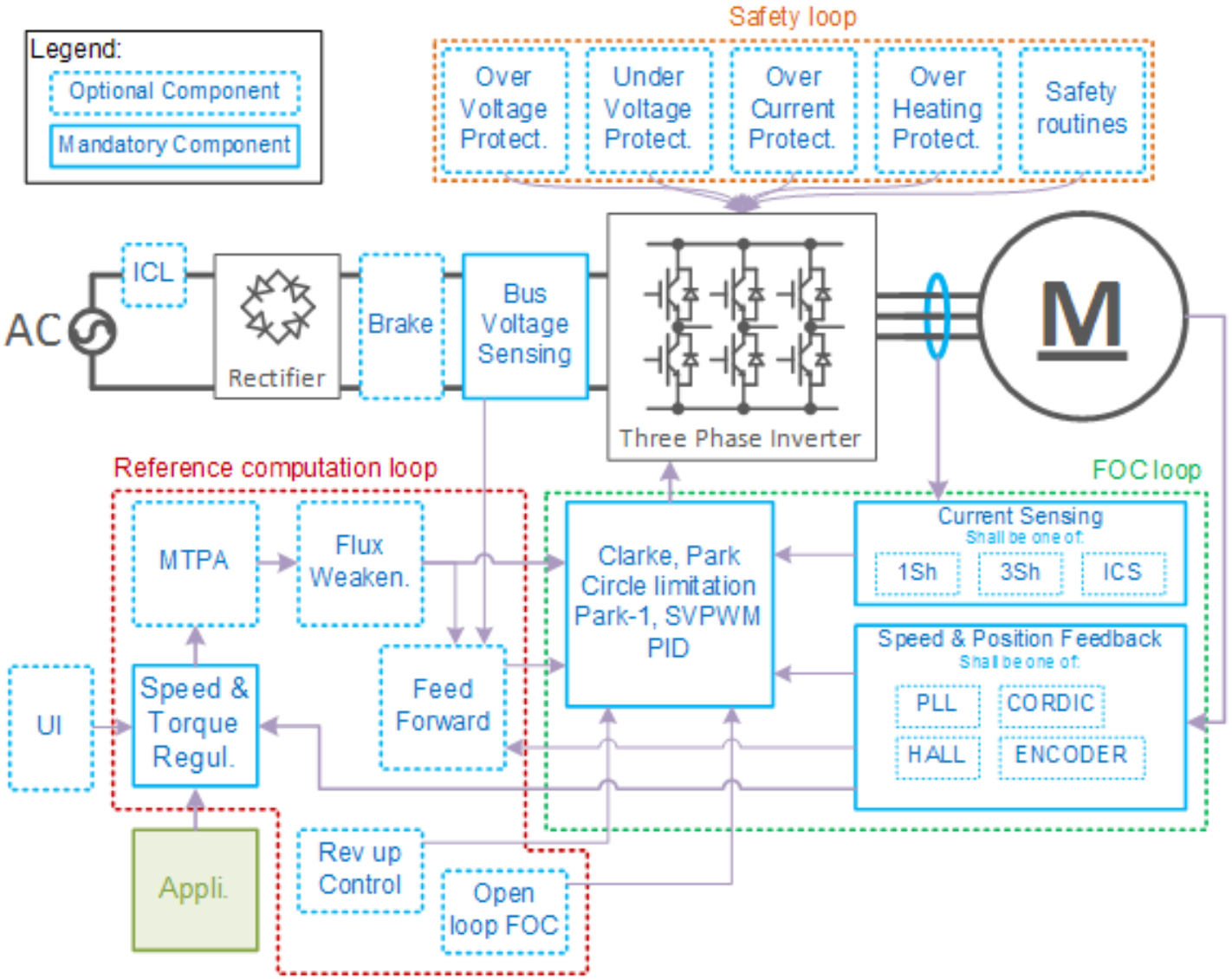
life.augmented

下列组件均以库的形式提供。用户无法看到源码，但可以根据提供的头文件从库里调用相应的函数。

组件名称	描述
feed_forward_ctrl.c	前馈控制
flux_weakening_ctrl.c	弱磁控制
max_torque_per_ampere.c	最大转矩控制
sto_cordic_speed_pos_fdbk.c	速度和位置反馈cordic
sto_pll_speed_pos_fdbk.c	速度和位置反馈PLL
revup_ctrl.c	启动控制

用户界面库

用户界面库包含负责通讯的组件。电机控制代码通过这些组件控制串口和 DAC 与外界通讯。通过这个库我们可以连接 MCU 和 Workbench 。在 Workbench 中实现对电机运行状态的监控。

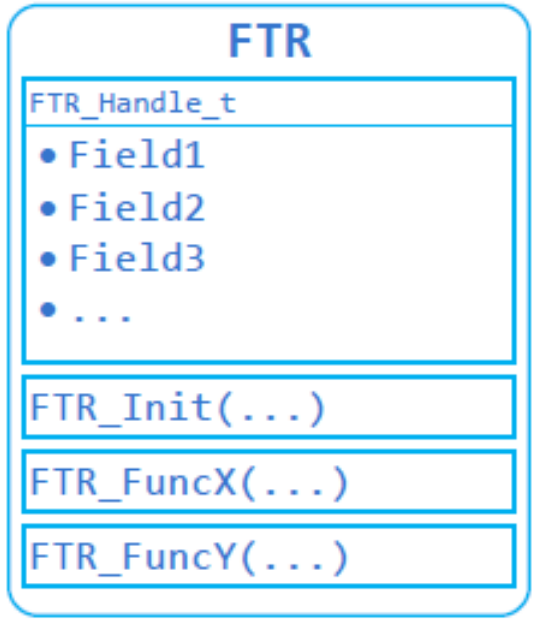


- FOC 控制环路 loop
- 电机控制环路 loop
- 安全控制环路

life.augmented

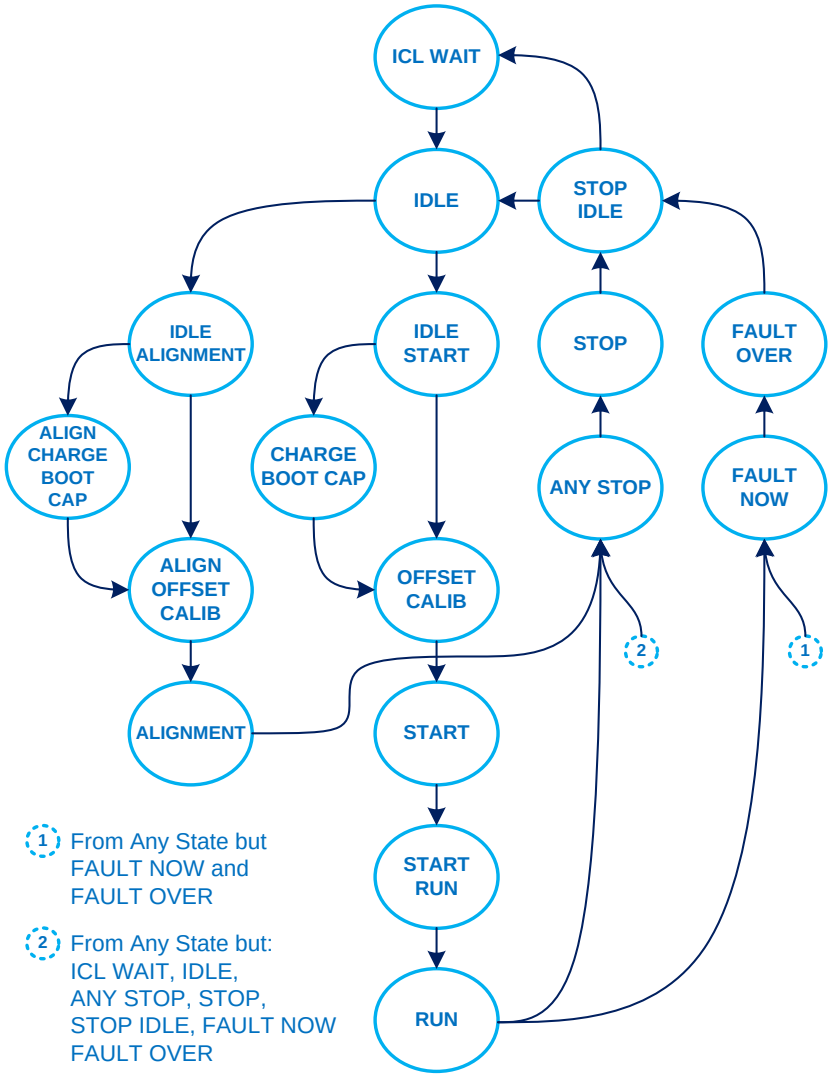
电机控制库是由我们称之为**组件**(component) 的单元构成的。组件是一个自给自足的软件单元，他包含两个部分：

- 包含一个能满足此组件功能的数据的结构体。
- 一系列的函数，这些函数操作组件的结构体变量以实现组件的功能。



组件通常包含一个 **.c** 文件和一个 **.h** 文件。结构体以 **组件名缩写 + _Handle_t** 命名，函数以 **组件名缩写 + 函数功能** 命名

life.augmented



电机控制状态机

- 程序用状态机来控制电机状态。
- 电机在如图的状态之间来回切换。
- SDK 5.x 中用 **state_machine** 组件来实现状态机的功能。

life.augmented

- state_machine 这个组件包含两个文件 state_machine.c 和 state_machine.h。
- 结构体以 STM_Handle_t 命名。
- 初始化函数以 STM_Init 命名。函数的参数是指向结构体变量 STM_Handle_t 的指针 pHandle。

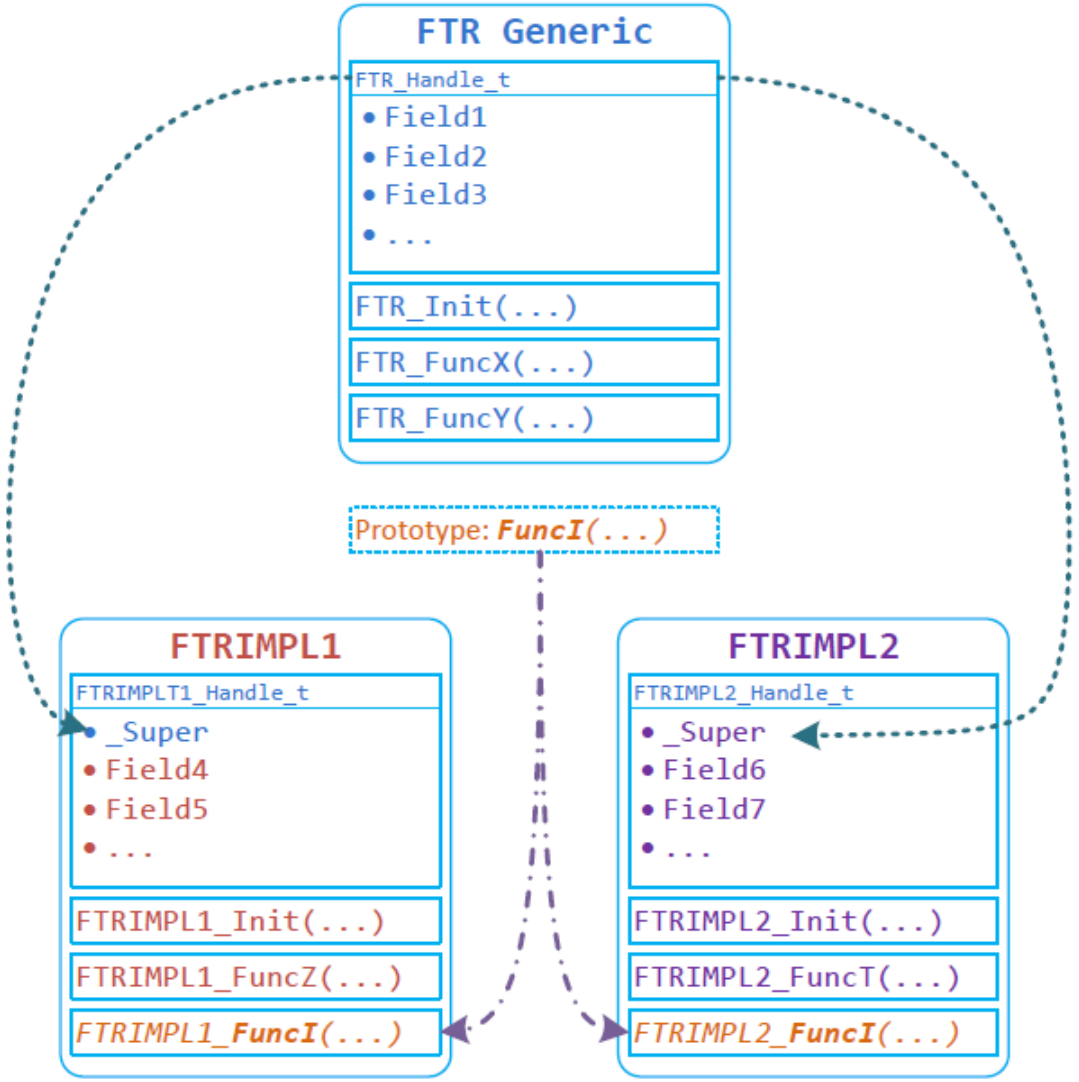
```
typedef struct
{
  State_t   bState;

  uint16_t  hFaultNow;

  uint16_t  hFaultOccurred;


} STM_Handle_t;
```

结构体

```
void STM_Init( STM_Handle_t * pHandle );


bool STM_NextState( STM_Handle_t * pHandle, State_t bState );

State_t STM_FaultProcessing( STM_Handle_t * pHandle, uint16_t hSetErrors, uint16_t
                             hResetErrors );


State_t STM_GetState( STM_Handle_t * pHandle );


bool STM_FaultAcknowledged( STM_Handle_t * pHandle );


uint32_t STM_GetFaultState( STM_Handle_t * pHandle );
```

函数

life.augmented



- 同一个功能有很多不同的具体实现方式，这些具体实现方式既包含共性的部分又有各自的个性。
- SDK 5.x 用**通用组件**来定义共性的部分，用**执行组件**来定义具体实现。
  - 通用组件包含公共的数据、函数和函数原型。
  - 执行组件包含共性加个性的部分。

- 图中的 **FTR** 是实现某个功能的通用组件。**FRTIMPL1** 和 **FRTIMPL2** 是这个功能的执行组件。
- 这两个执行组件的结构体中都有一个成员叫 **_Super** 。执行组件通过成员 _Super 实现对 FTR 所定义的公共资源的复用。
- 通用组件 FTR 中定义一系列的函数原型如 **FuncI(...)**，并在其结构体中定义了这个函数原型的函数指针。
- FRTIMPL1 和 FRTIMPL2 分别定义函数 **FTRIMPL1_ FuncI(...)** 和 **FTRIMPL2_ FuncI(...)** 来具体实现通用组件定义的函数原型。

life.augmented

下面以 SDK 5.x 自带的例程代码 **P-NUCLEO-IHM001-BullRunning** 中的 PWM 和电流采样功能为例，进一步说明通用组件及其执行组件。

- PWM 和电流采样功能可以采用单电阻，3电阻或者电流传感器来实现
- SDK 5.x 定义了通用组件 **PWMC** 来实现这个功能的通用部分。
- 通用组件 PWMC 有很多执行组件。我们以其中的 **R3_1_F30X** 和 **R1F30X** 这两个执行组件为例。

通用组件 **PWMC** 中定义的结构体 **PWMC_Handle_t** 。这个结构体的成员中不仅有数据变量，还有函数指针。如下图中的 **pFctGetPhaseCurrents**

```
typedef struct PWMC_Handle PWMC_Handle_t;

struct PWMC_Handle
{
  /** @{ */
  PWMC_IrqHandler_Cb_t pFctIrqHandler;                  /**< pointer on the interrupt handling function. */
  PWMC_GetPhaseCurr_Cb_t
  pFctGetPhaseCurrents;          /**< pointer on the function the component instance uses to retrieve pahse currents */
  PWMC_Generic_Cb_t
  pFctSwitchOffPwm;                   /**< pointer on the function the component instance uses to switch PWM off */
```

通用组件定义了这个函数指针的函数原型 **PWMC_GetPhaseCurr_Cb_t** 。

```
typedef void ( *PWMC_GetPhaseCurr_Cb_t )( PWMC_Handle_t * pHandle, Curr_Components * pStator_Currents );
```

life.augmented

```
typedef struct
{
  PWMC_Handle_t _Super;
  uint32_t wPhaseAOffset;
  uint32_t wPhaseBOffset;
  uint32_t wPhaseCOffset;
  uint16_t Half_PWMPeriod;
  uint16_t hRegConv;
    .
    .
    .
    .
    .
    .
    .
    .
    .
  volatile uint8_t  bIndex;
  uint16_t ADC_ExternalTriggerInjected;
  /*!< Trigger selection for ADC peripher
  bool OverCurrentFlag;      /*!< This flag
  bool OverVoltageFlag;      /*!< This flag
  bool BrakeActionLock;      /*!< This flag
                                interrupt
  R3_1_F30XParams_t const * pParams_str;

} PWMC_R3_1_F3_Handle_t;
```

```
typedef struct
{
  PWMC_Handle_t _Super;
  uint16_t hDmaBuff[2];
  uint16_t hCntSmp1;
  uint16_t hCntSmp2;
  uint8_t sampCur1;
  uint8_t sampCur2;
    .
    .
    .
    .
    .
    .
    .
    .
    .

  DMA_Channel_TypeDef * PreloadDMAy_Chx; /*
  DMA_Channel_TypeDef * DistortionDMAy_Chx;
  bool OverCurrentFlag;      /*!< This flag
  bool OverVoltageFlag;      /*!< This flag
  bool BrakeActionLock;      /*!< This flag

  R1_F30XParams_t const * pParams_str;

} PWMC_R1_F3_Handle_t;
```

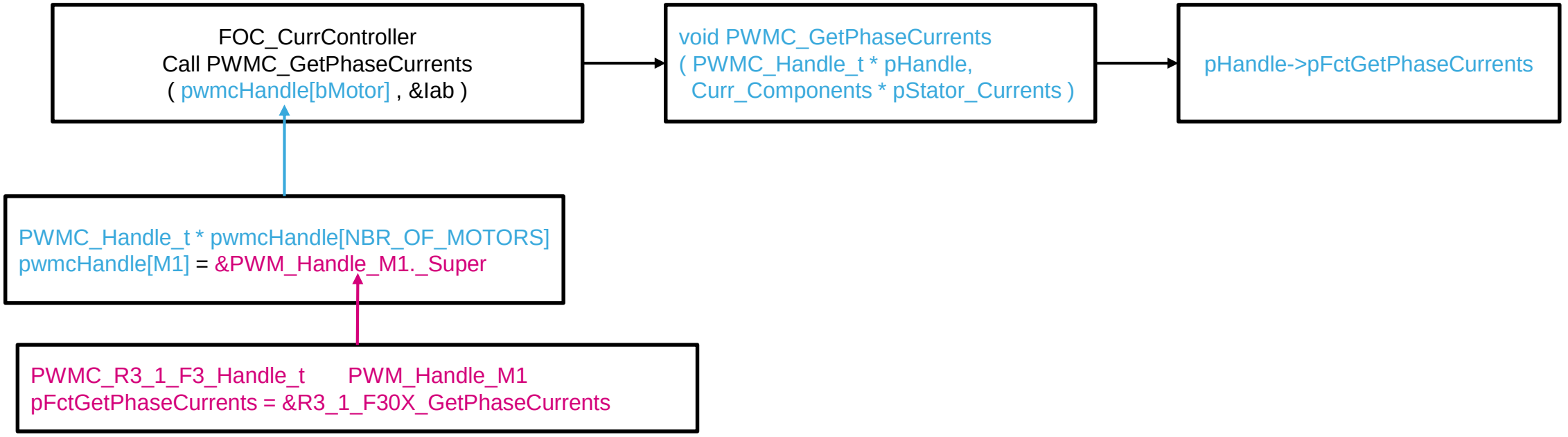
- 组件 **R3_1_F30X** 和 **R1F30X** 的结构体中都有一个成员**_Super**。通过 _Super 的类型是通用组件 **PWMC** 定义的结构体 **PWMC_Handle_t** 。
- R3_1_F30X 和 R1F30X 的实例会把 _Super 中的 **pFctGetPhaseCurrents** 指向各自的实现函数.

```
/* Change back function to be executed in ADCx_ISR */
pHandle->_Super.pFctGetPhaseCurrents = &R3_1_F30X_GetPhaseCurrents;

/* Change back function to be executed in ADCx_ISR */
pHandle->_Super.pFctGetPhaseCurrents = &R1F30X_GetPhaseCurrents;
```

- R3_1_F30X 和 R1F30X 除了通用组件中公有的部分还各自定义了自己独有的数据。

life.augmented

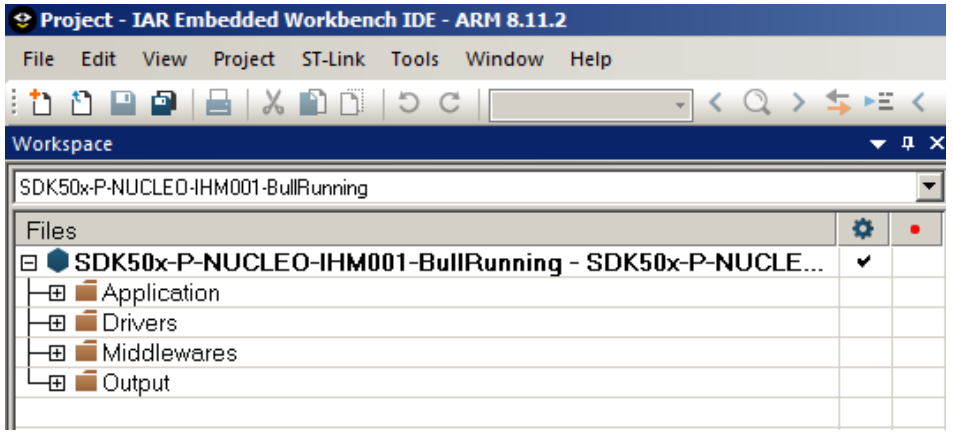
以读取电流值这个功能为例，说明例程代码 P-NUCLEO-IHM001-BullRunning 如何使用通用组件 **PWMC** 和执行组件 **R3_1_F30X**



下列组件是SDK5.x中通用组件的结构体及其执行组件结构体

通用组件结构体	执行组件结构体	执行组件结构体	执行组件结构体
BusVoltageSensor_Handle_t	RDivider_Handle_t	VirtualBusVoltageSensor_Handle_t	
SpeednPosFdbk_Handle_t	ENCODER_Handle_t	HALL_Handle_t	STO_Handle_t
	STO_CR_Handle_t	VirtualSpeedSensor_Handle_t	STO_PLL_Handle_t
UI_Handle_t	DAC_UI_Handle_t	MCP_Handle_t	UDFastCom_Handle_t
FCP_Handle_t	UFCP_Handle_t		
PWMC_Handle_t	PWMC_ICS_F3_Handle_t	PWMC_ICS_F4_Handle_t	PWMC_ICS_F7_Handle_t
	PWMC_ICS_HD2_Handle_t	PWMC_ICS_L4_Handle_t	PWMC_ICS_LM1_Handle_t
	PWMC_R1_F0_Handle_t	PWMC_R1_F3_Handle_t	PWMC_R1_F4_Handle_t
	PWMC_R1_F7_Handle_t	PWMC_R1_HD2_Handle_t	PWMC_R1_L4_Handle_t
	PWMC_R1_VL1_Handle_t	PWMC_R3_1_F3_Handle_t	PWMC_R3_1_F4_Handle_t
	PWMC_R3_1_F7_Handle_t	PWMC_R3_1_L4_Handle_t	PWMC_R3_2_F3_Handle_t
	PWMC_R3_2_F7_Handle_t	PWMC_R3_2_L4_Handle_t	PWMC_R3_32_G4_Handle_t
	PWMC_R3_3_G4_Handle_t	PWMC_R3_4_F3_Handle_t	PWMC_R3_F0_Handle_t
	PWMC_R3_F4_Handle_t	PWMC_R3_HD2_Handle_t	PWMC_R3_LM1_Handle_t

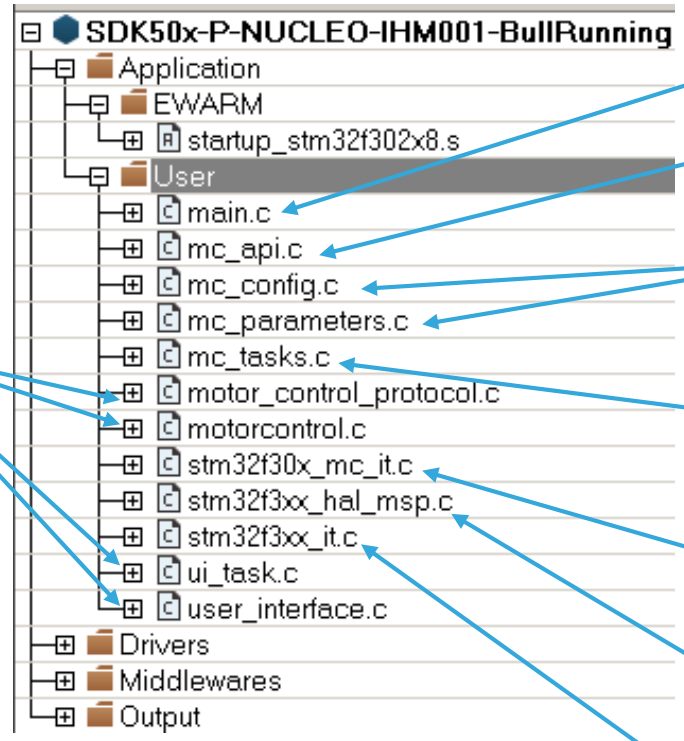
life.augmented



打开 SDK 5.x 自带的例程代码 P-NUCLEO-IHM001-BullRunning 的IAR 工程。在 IAR 的 workspace 窗口，我们可以看到在工程下有四个文件夹。

- **Application**
  和客户的应用紧密相关的程序。通常客户只需要更改这一部分的代码。我们在后面会有更详细的说明。
- **Drivers**
  MCU 的外设驱动代码，由 CubeMX 生成。另外还包含 ARM 的 CMSIS 库.
- **Middleware**
  电机控制库中的各个组件的代码和库文件。
- **Output**
  链接器输出的文件如 .out 文件,.map 文件等。

life.augmented



UI 相关

main 函数。

用户与 SDK 交互的接口函数。

实例化各组件的结构体，并初始化。

电机控制的核心环路。如电流环，转矩和速度环，状态机等。

电机控制相关的外设中断服务程序。

CubeMX 生成的各外设的 IO 配置。

用户自定义的外设中断服务程序。

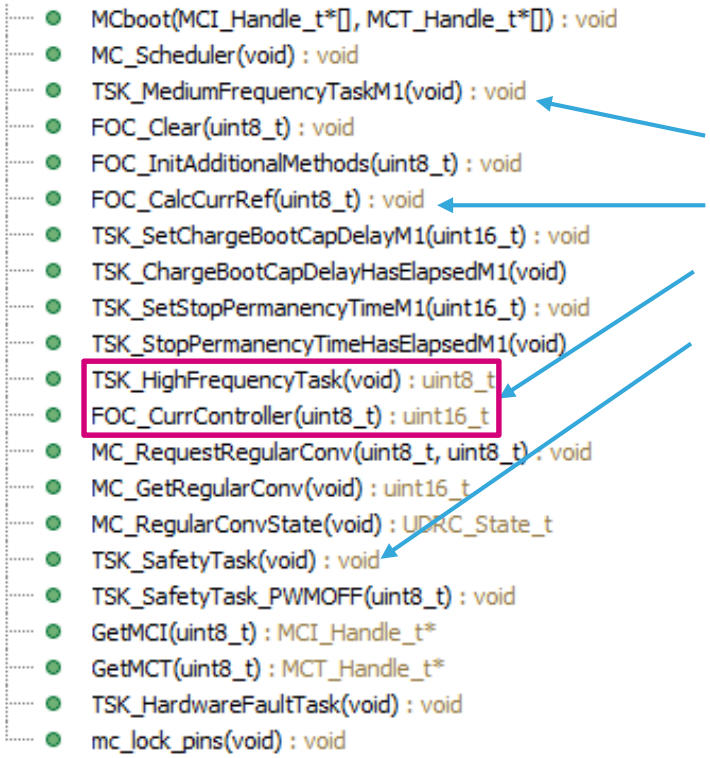
life.augmented

```
97  int main(void)
98  {
99    /* USER CODE BEGIN 1 */
100
101   /* USER CODE END 1 */
102
103   /* MCU Configuration--------------------------
104
105   /* Reset of all peripherals, Initializes th
106   HAL_Init();
107
108   /* USER CODE BEGIN Init */
109
110   /* USER CODE END Init */
111
112   /* Configure the system clock */
113   SystemClock_Config();
114
115   /* USER CODE BEGIN SysInit */
116
117   /* USER CODE END SysInit */
118
119   /* Initialize all configured peripherals */
120   MX_GPIO_Init();
121   MX_ADC1_Init();
122   MX_DAC_Init();
123   MX_TIM1_Init();
124   MX_USART2_UART_Init();
125   MX_MotorControl_Init();
126
127   /* Initialize interrupts */
128   MX_NVIC_Init();
129   /* USER CODE BEGIN 2 */
130
131   /* USER CODE END 2 */
132
133   /* Infinite loop */
134   /* USER CODE BEGIN WHILE */
135   while (1)
136   {
137
138   /* USER CODE END WHILE */
139
140   /* USER CODE BEGIN 3 */
141
142   }
143   /* USER CODE END 3 */
144
145 }
```

在 **main.c** 文件中，函数 **int main(void)** 包含以下内容：

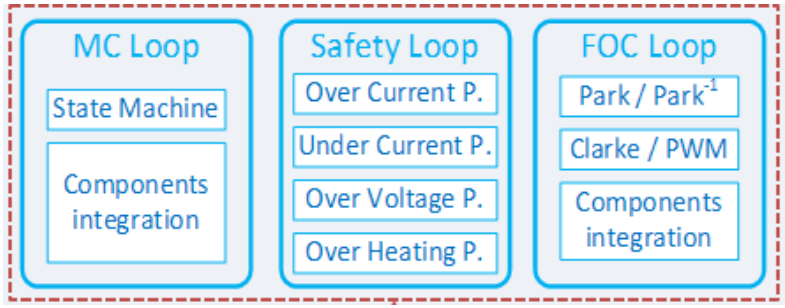
- System configure
- Peripherals configure
- Motor Control configure
- Interrupt configure
- Main loop

life.augmented



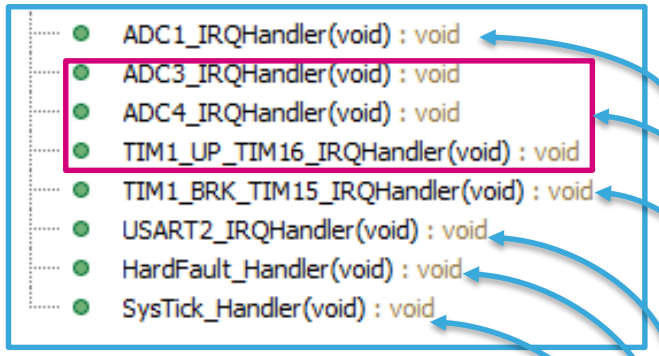
在文件 **mc_tasks.c** 中核心的是下面的几个环路:

- MC loop
- Torque or speed loop
- Current loop
- Safety loop



life.augmented

例程代码讲解




```
309  void SysTick_Handler(void)
310  {
311    /* USER CODE BEGIN SysTick_IRQn 0 */
312
313    /* USER CODE END SysTick_IRQn 0 */
314    HAL_IncTick();
315    HAL_SYSTICK_IRQHandler();
316    /* USER CODE BEGIN SysTick_IRQn 1 */
317    /* USER CODE END SysTick_IRQn 1 */
318      TB_Scheduler();
319    /* USER CODE BEGIN SysTick_IRQn 2 */
320    /* USER CODE END SysTick_IRQn 2 */
321  }
322
323  /* USER CODE BEGIN 1 */
324
325  /* USER CODE END 1 */
```

在文件 **stm32f30x_mc_it.c** 中有以下的中断服务程序：

ADC 中断: **HighFrequencyTask()** 在这个函数中被调用。调用的频率可以在 **workbench** 中设定。最大的频率可以和 **PWM** 频率相同。

- 此例程中未启用。
- **TIM1 Break** 中断。
- **UART** 中断。
- **System hard fault** 中断。
- **System Tick** 中断：周期为500us。

life.augmented

在 **mc_api.c** 中有一系列的函数来实现与对电机的控制，我们称之为 **MC API**。这些函数是用户和 **SDK** 之间的桥梁。下面列出了一些最常用的函数：

- MC_StartMotor1
- MC_StopMotor1
- MC_ProgramSpeedRampMotor1
- MC_ProgramTorqueRampMotor1
- MC_GetMecSpeedReferenceMotor1
- MC_GetMecSpeedAverageMotor1
- MC_GetSTMStateMotor1
- MC_GetOccurredFaultsMotor1
- MC_AcknowledgeFaultMotor1
- MC_GetImposedDirectionMotor1

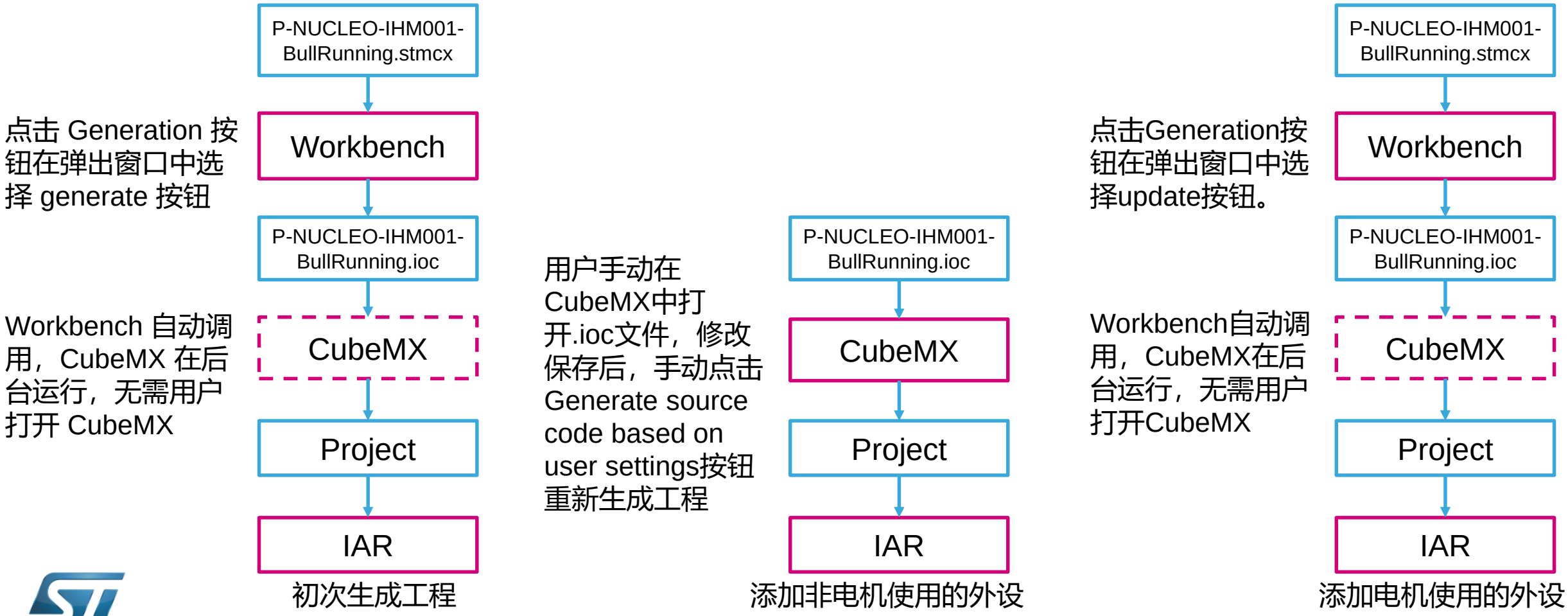
开发实战

- ST MC SDK5.x WB应用指南
  - 软件工具的下载和安装
  - ST MC Workbench及关键配置参数
  - MC Project 的生成，编译和下载
  - 电机控制及监控
- ST MC SDK5.x固件详解
  - 程序架构
  - 组件
  - 例程代码讲解
  - 开发实战
    - 如何向例程中添加外设和自己的代码
    - Step-by-Step添加一段闪灯代码

life.augmented

如何向例程中添加外设

下图说明开发者在初次生成代码，修改代码配置一个非电机使用的外设，修改一个电机使用外设三种应用场景下的流程。



life.augmented

如何向例程中添加外设

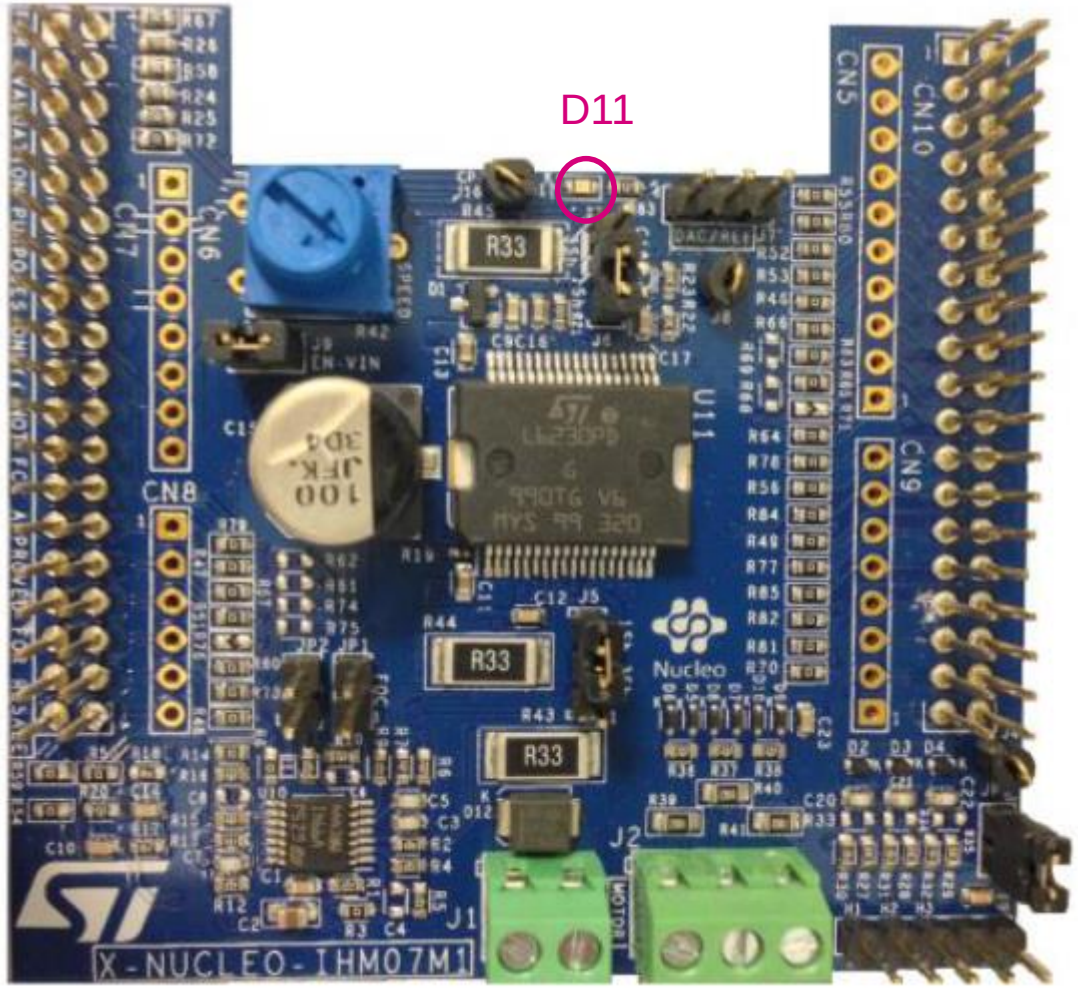

```
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------

  /* Reset of all peripherals, Initializes the
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_ADC1_Init();
  MX_DAC_Init();
  MX_TIM1_Init();
  MX_USART2_UART_Init();
  MX_MotorControl_Init();
```

因为 **Workbench** 生成的工程中，绝大部分的代码是以源码的形式提供给大家的。所以大家可以很方便的修改和添加自己的应用代码。

左图是例程代码 **P-NUCLEO-IHM001-BullRunning** 中的 **main** 函数。我们可以看到很多注释如 **/* USER CODE BEGIN 1 */** 和 **/* USER CODE END 1 */** 。开发者应把自己的应用代码添加在 USER CODE BEGIN 和 USER CODE END 之间。这样当我们需要修改外设配置重新生成代码时，我们之前添加的代码不会被修改掉。否则代码将被覆盖为 SDK 的默认代码。

life.augmented

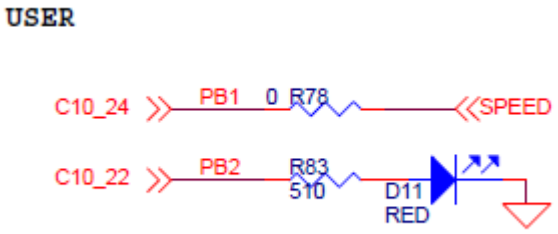
Step-by-Step添加一段闪灯代码



SDK5.x 中自带代码例程 P-NUCLEO-IHM001-BullRunning 。与之配套的硬件是控制板 NUCLEO F302R8 和功率板 X-NUCLEO-IHM07M1。

下面我们将一步一步修改代码,实现 LED 灯 D11 在电机进入正常运行状态后以1.0s的频率闪烁。

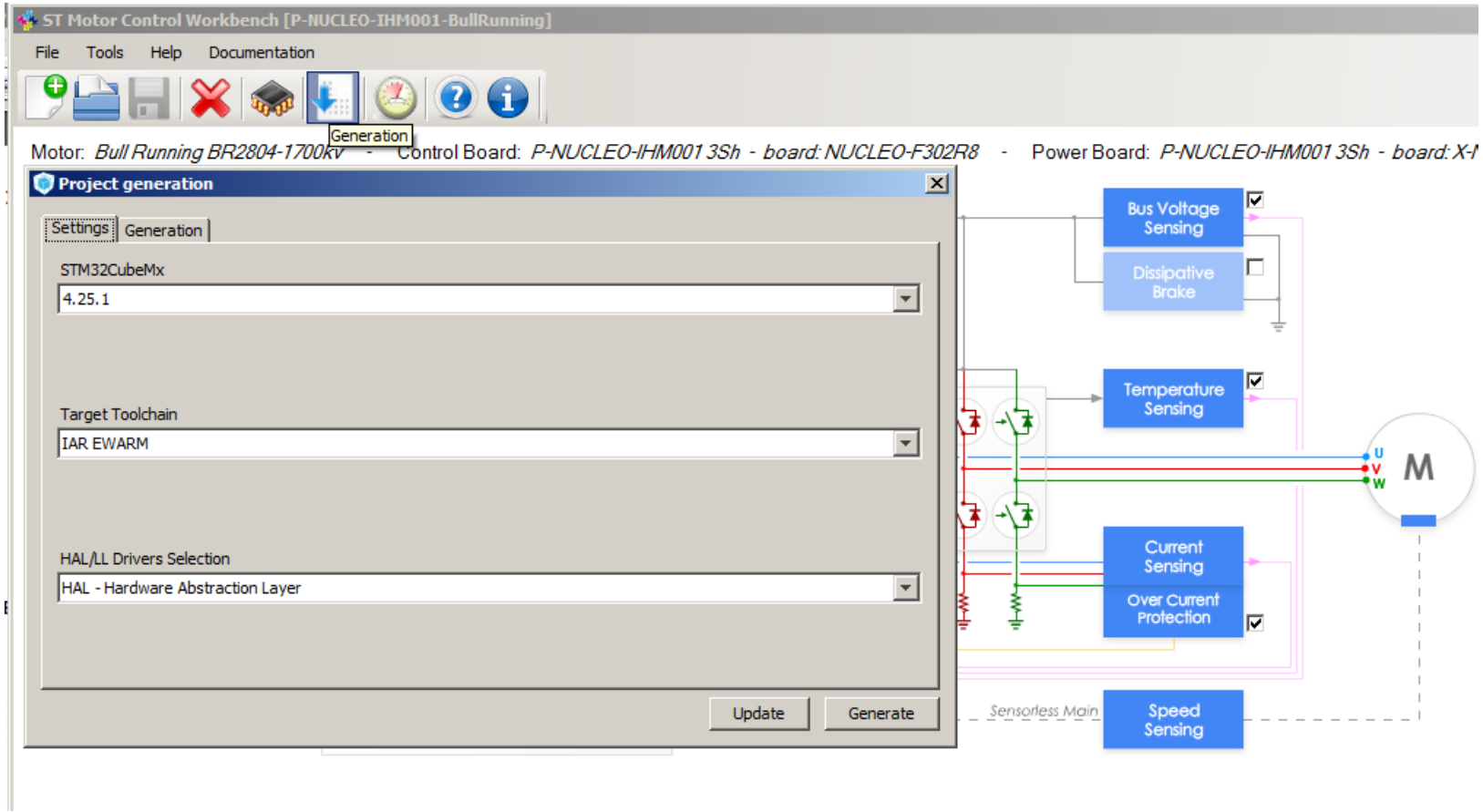
D11 连到了 STM32F302R8 的 PB2 引脚。



life.augmented

Step 1：Motor Control Workbench 初次生成代码

- 用 **Workbench** 打开 **SDK** 自带的 **P-NUCLEO-IHM001-BullRunning.stmcx**。
- 点击下图的 **Generation** 按钮，会跳出 **Project generation** 窗口。
- 在窗口中点击 **Generate** 按钮。



Step 1：Motor Control Workbench 初次生成代码

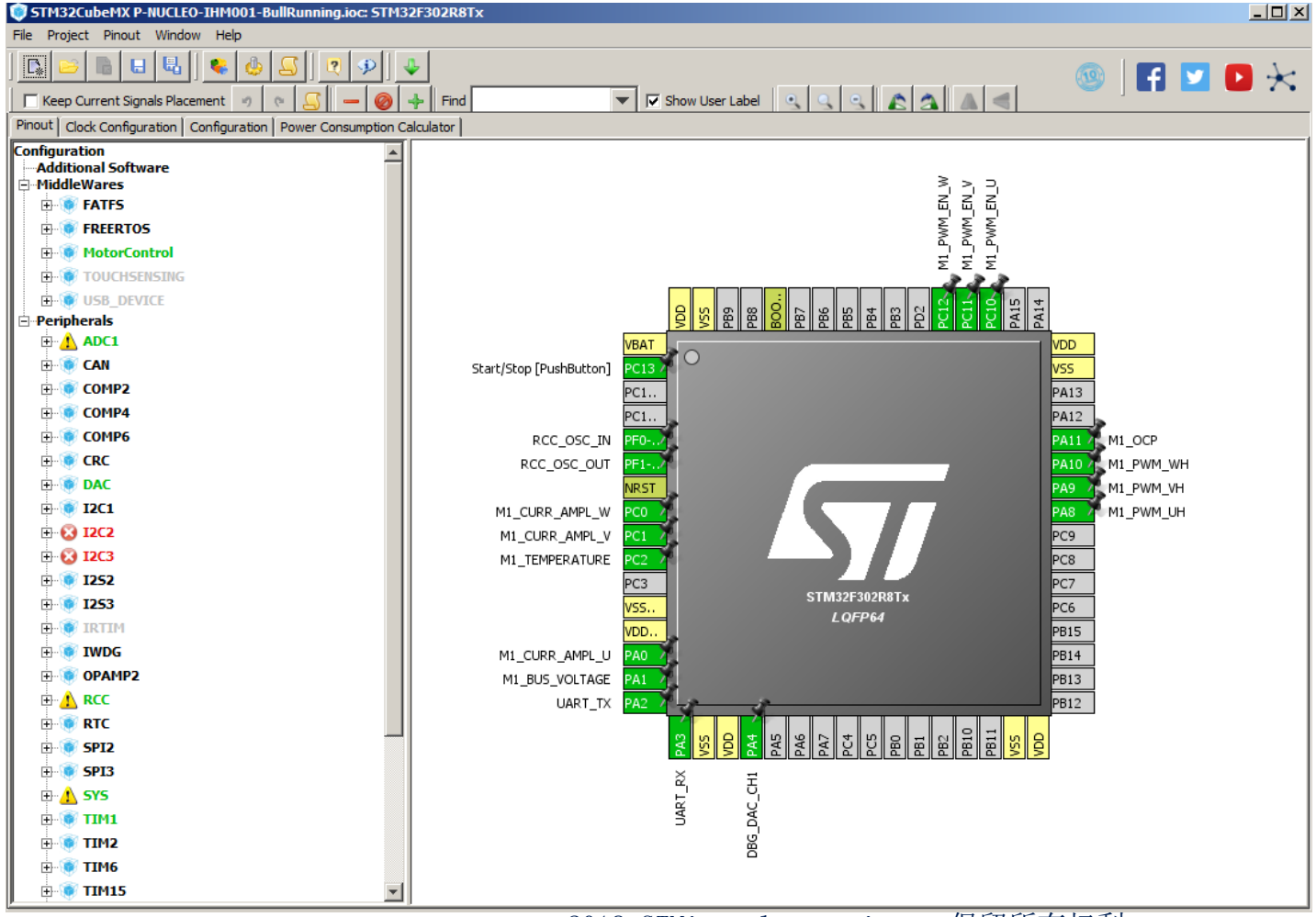
- 打开 IAR 工程，我们可以看到如下的 GPIO 初始化代码。

```
static void MX_GPIO_Init(void)
{

  GPIO_InitTypeDef GPIO_InitStruct;

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOC, M1_PWM_EN_U_Pin|M1_PWM_EN_V_Pin|M1_PWM_EN_W_Pin, GPIO_PIN_RESET);

  /*Configure GPIO pin : Start_Stop_Pin */
  GPIO_InitStruct.Pin = Start_Stop_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(Start_Stop_GPIO_Port, &GPIO_InitStruct);

  /*Configure GPIO pins : M1_PWM_EN_U_Pin M1_PWM_EN_V_Pin M1_PWM_EN_W_Pin */
  GPIO_InitStruct.Pin = M1_PWM_EN_U_Pin|M1_PWM_EN_V_Pin|M1_PWM_EN_W_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLDOWN;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

}
```

life.augmented

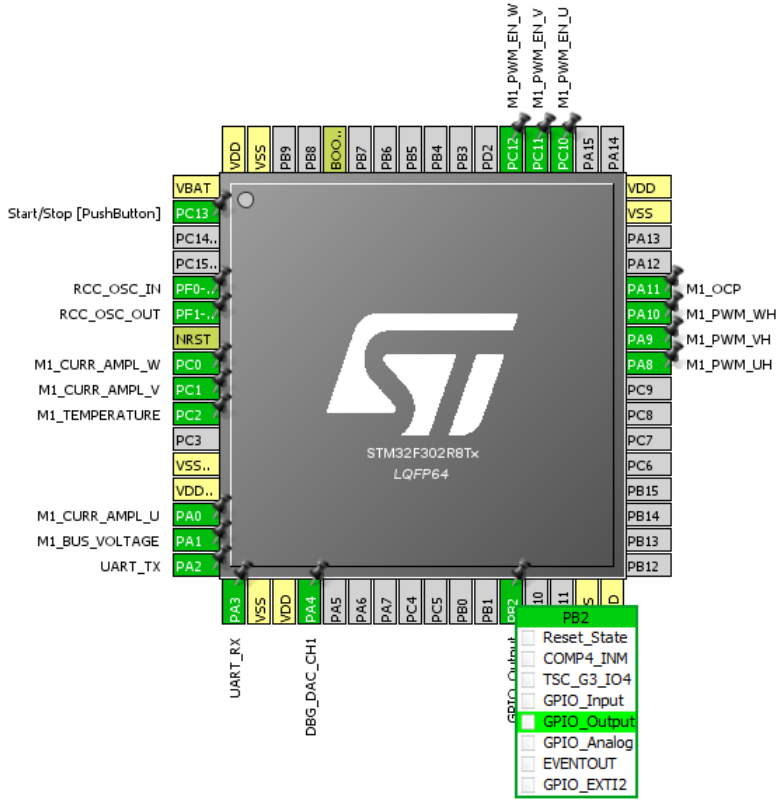
Step 2：在 CubeMX 中添加一个 GPIO

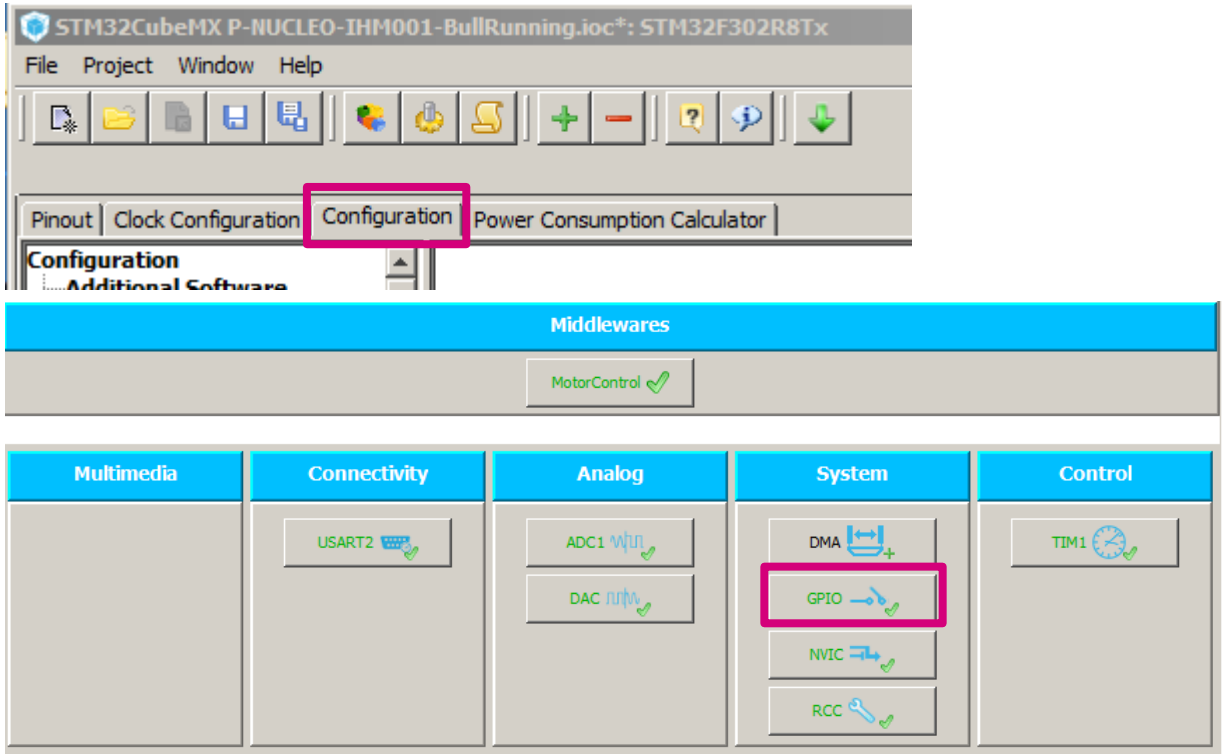
- 打开 P-NUCLEO-IHM001-BullRunning.ioc ， 在 CubeMX 里配置 GPIO 的 PB2 引脚。
- 从下图我们可以看到 PB2 尚未被占用。



Step 2：在 CubeMX 中添加一个 GPIO

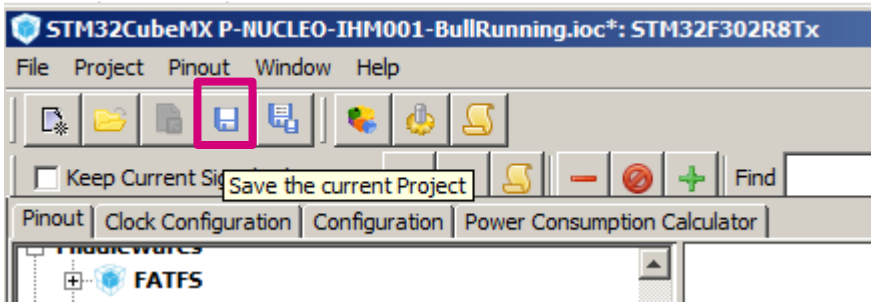
- 点击左图中的 **PB2** 引脚，选择 **GPIO_Output**。
- 如右图，在 **CubeMX** 的窗口中选择 **Configuration** 选项卡。在窗口中我们可以看到 **System** 区域内有 **GPIO** 的按钮。点击设置 **PB2** 的初始配置。我们把 PB2 初始化为输出高电平，来点亮 LED。

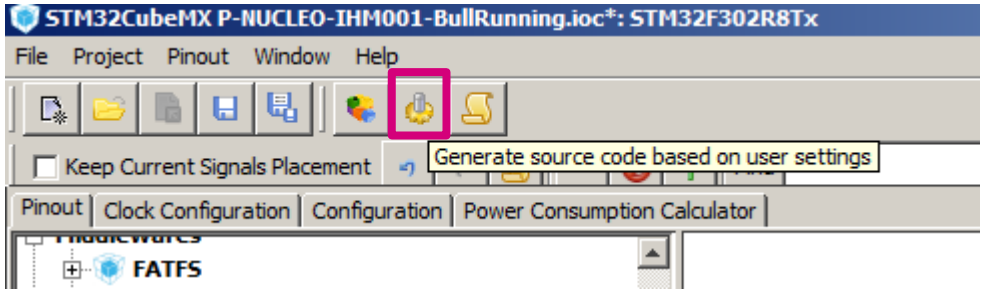




Step 3：保存 P-NUCLEO-IHM001-BullRunning.ioc，重新生成代码

- 点击 **Save the current Project** 按钮。保存对 **PB2** 的设置到 **P-NUCLEO-IHM001-BullRunning.ioc**。
- 点击 **Generate source code based on user settings** 。





Step 3：保存 P-NUCLEO-IHM001-BullRunning.ioc，重新生成代码

- 用 IAR 打开新生成的代码，我们可以看到在 main.h 中增加了 PB2 的宏定义。
- 在 main.c 的函数 static void MX_GPIO_Init(void)中也新增了初始化代码。
- 运行新生成的代码，我们可以看到功率板 X-NUCLEO-IHM07M1 上的 LED 亮起来。

main.h

```
81  #define LED_D11_Pin GPIO_PIN_2
82  #define LED_D11_GPIO_Port GPIOB
```

main.c

MX_GPIO_Init()

```
457  */
458  static void MX_GPIO_Init(void)
459  {
460
461    GPIO_InitTypeDef GPIO_InitStruct;
462
463    /* GPIO Ports Clock Enable */
464    __HAL_RCC_GPIOC_CLK_ENABLE();
465    __HAL_RCC_GPIOF_CLK_ENABLE();
466    __HAL_RCC_GPIOA_CLK_ENABLE();
467    __HAL_RCC_GPIOB_CLK_ENABLE();
468
469    /*Configure GPIO pin Output Level */
470    HAL_GPIO_WritePin(LED_D11_GPIO_Port, LED_D11_Pin, GPIO_PIN_SET);
471
472    /*Configure GPIO pin Output Level */
473    HAL_GPIO_WritePin(GPIOC, M1_PWM_EN_U_Pin|M1_PWM_EN_V_Pin|M1_PWM_EN_W_Pin, GPIO_PIN_RESET);
474
475    /*Configure GPIO pin : Start_Stop_Pin */
476    GPIO_InitStruct.Pin = Start_Stop_Pin;
477    GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
478    GPIO_InitStruct.Pull = GPIO_NOPULL;
479    HAL_GPIO_Init(Start_Stop_GPIO_Port, &GPIO_InitStruct);
480
481    /*Configure GPIO pin : LED_D11_Pin */
482    GPIO_InitStruct.Pin = LED_D11_Pin;
483    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
484    GPIO_InitStruct.Pull = GPIO_NOPULL;
485    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
486    HAL_GPIO_Init(LED_D11_GPIO_Port, &GPIO_InitStruct);
```

life.augmented

Step 4: 添加代码使 LED 根据电机的运行状态亮灭

- 在 mc_api.c 中查找用户 API 后， 我们发现可以调用函数 State_t MC_GetSTMStateMotor1(void) 得到电机状态。
- 通过查询 State_t 的定义，我们可以在 state_machine.h 找到电机的各个状态。各状态的含义见程序注释。当电机正常运行的时候，对应的状态因为 RUN = 6。
- 在程序中添加如图代码，实现在电机进入RUN后点亮LED。注意所添加的代码都在注释 USER CODE BEGIN 和 USER CODE END 之间。除了红框中的一行代码。
- 通过以上的修改我们实现了让 LED 根据电机状态点亮和熄灭。

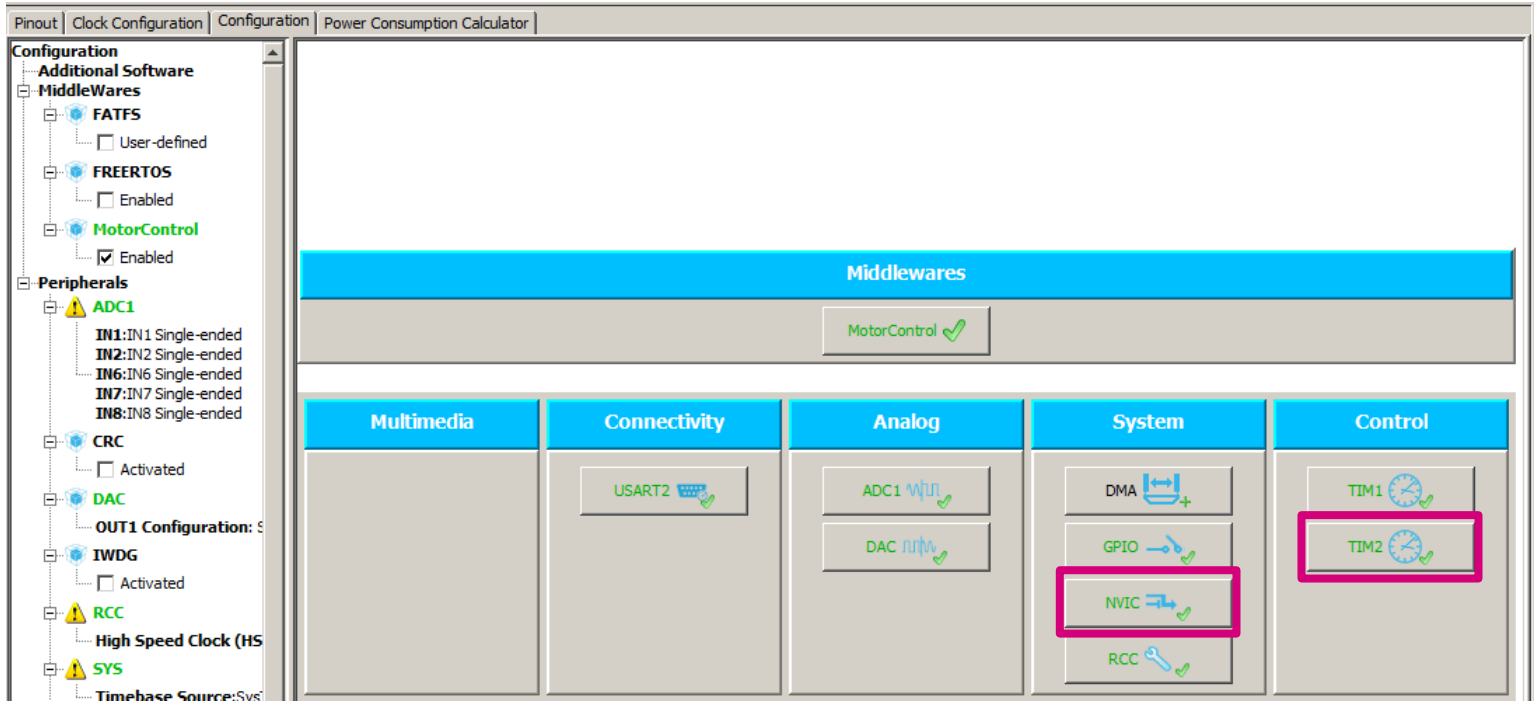
main.c

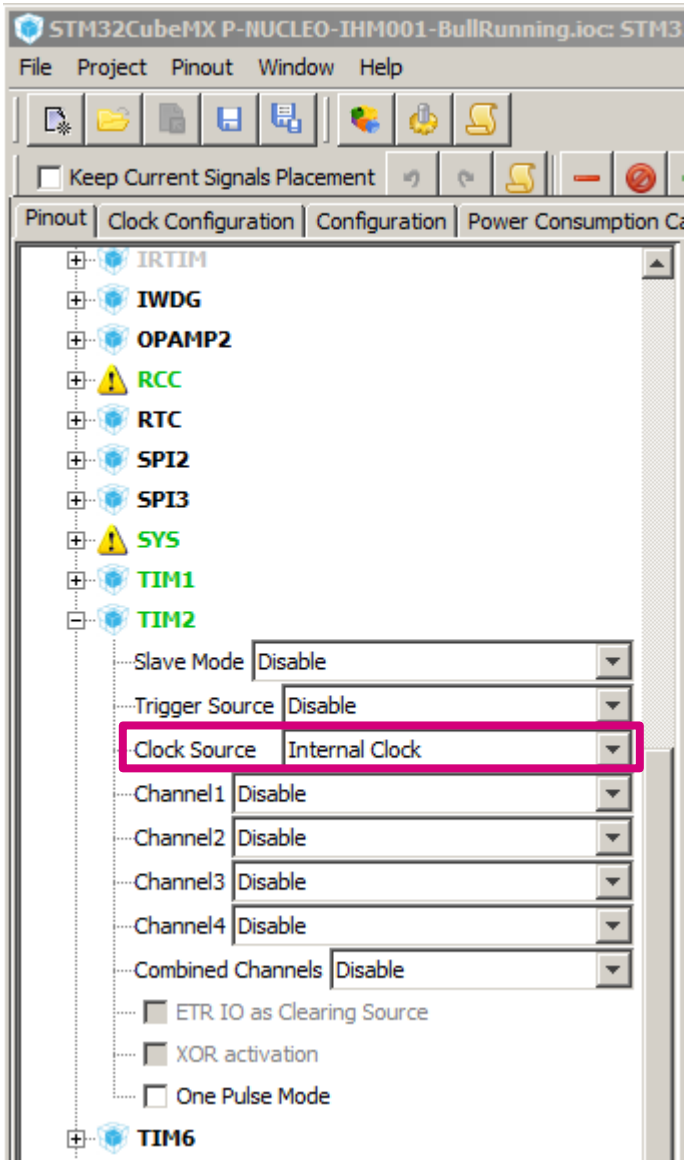
```
135   /* Infinite loop */
136   HAL_GPIO_WritePin(LED_D11_GPIO_Port, LED_D11_Pin, GPIO_PIN_RESET);
137   /* USER CODE BEGIN WHILE */
138   while (1)
139   {
140
141   /* USER CODE END WHILE */
142
143   /* USER CODE BEGIN 3 */
144     if(MC_GetSTMStateMotor1() == RUN)
145     {
146       if(LEDIndicationFlag == 0)
147       {
148         LEDIndicationFlag = 1;
149         HAL_GPIO_WritePin(LED_D11_GPIO_Port, LED_D11_Pin, GPIO_PIN_SET);
150       }
151     }
152     else
153     {
154       if(LEDIndicationFlag == 1)
155       {
156         LEDIndicationFlag = 0;
157         HAL_GPIO_WritePin(LED_D11_GPIO_Port, LED_D11_Pin, GPIO_PIN_RESET);
158       }
159     }
160   }
161   /* USER CODE END 3 */
162
163 }
```

life.augmented

Step 5：添加一个定时器 TIM2 来实现500ms的定时中断

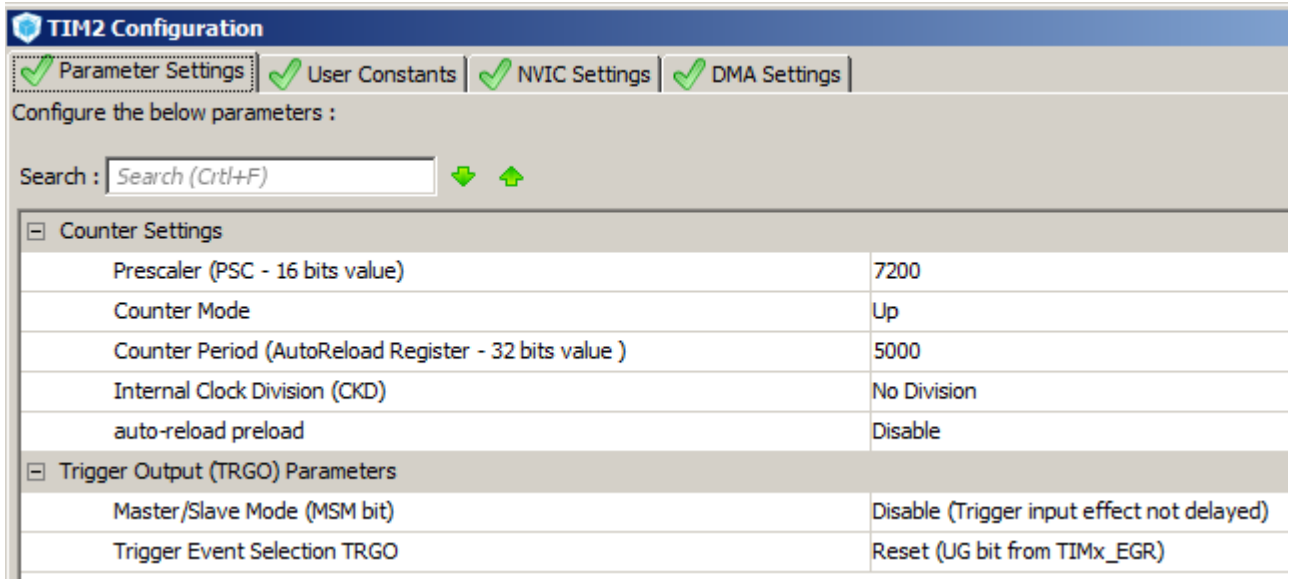
- 打开 P-NUCLEO-IHM001-BullRunning.ioc 在右图所示的 Pinout 窗口中点击 TIM2 。为选择 TIM2 选择时钟源。
- 在如下图在窗口 Configuration 中有对各外设的具体配置。我们需要点击下图中的 TIM2 和 NVIC 这两个按钮，分别配置 TIM2 及其中断。

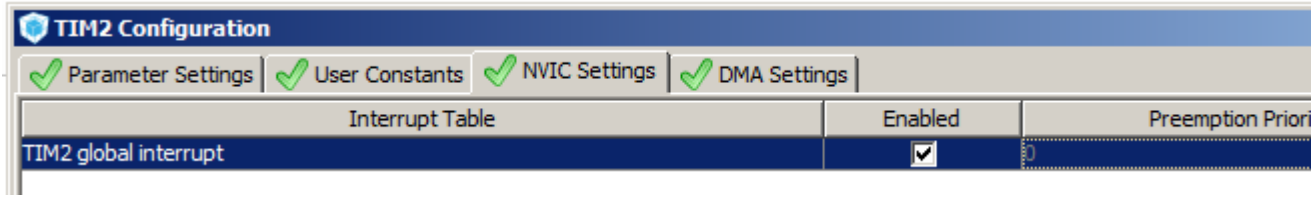




Step 5：添加一个定时器 TIM2 来实现500ms的定时中断

- 点击 **TIM2** 后弹出下图所示的 **TIM2 Configuration** 窗口。
- 在窗口内的 **Parameter** 页面设置时钟的周期为500ms。
- 在窗口内的 **NVIC Settings** 页面使能 **TIM2 global interrupt**。

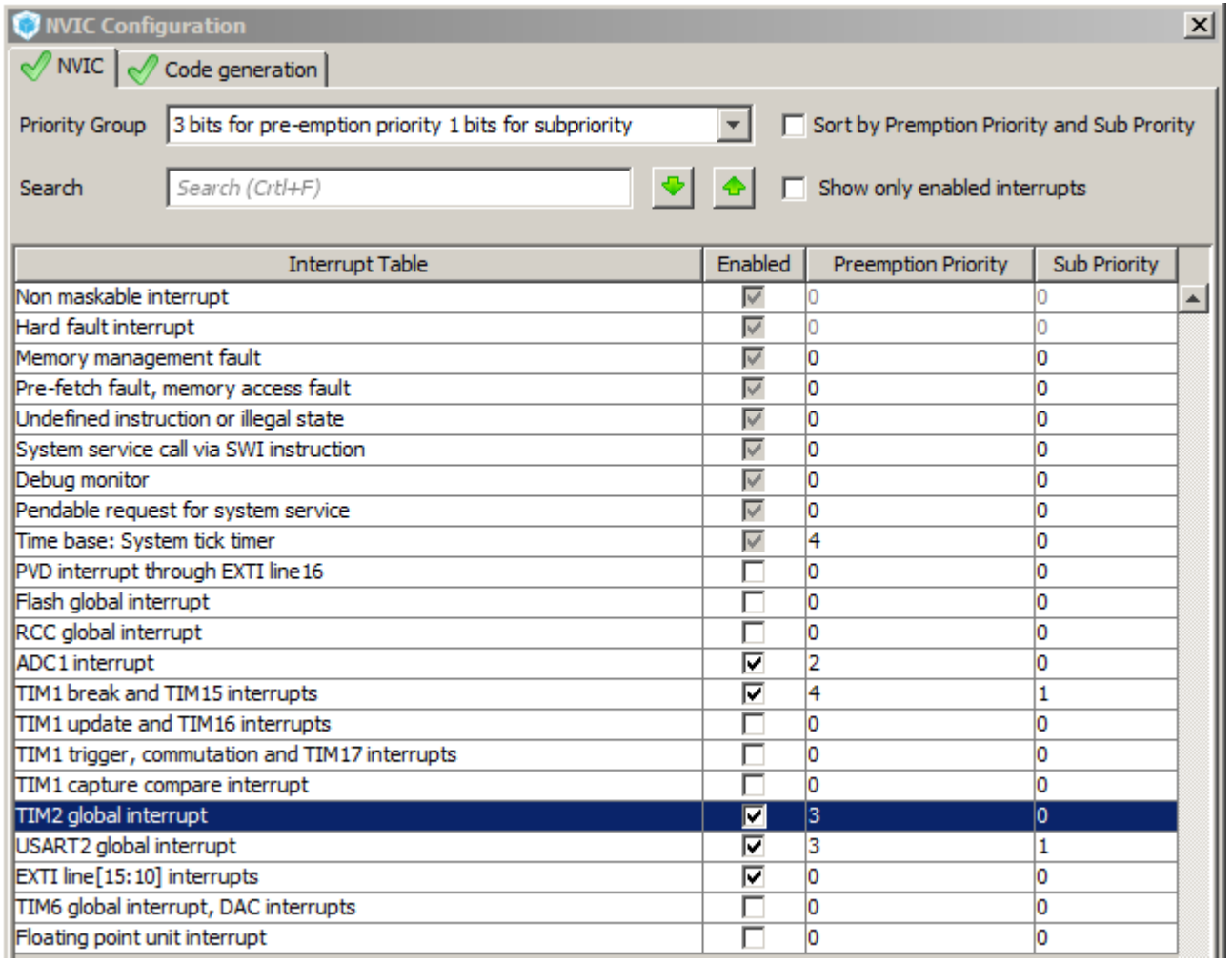




life.augmented

Step 5：添加一个定时器 TIM2 来实现500ms的定时中断

- 关闭 **TIM2 Configuration** 窗口，回到 **Configuration** 窗口，点击 **NVIC** 按钮。
- 在 **NVIC Configuration** 窗口选择 **TIM2 global interrupt** 的中断优先级。



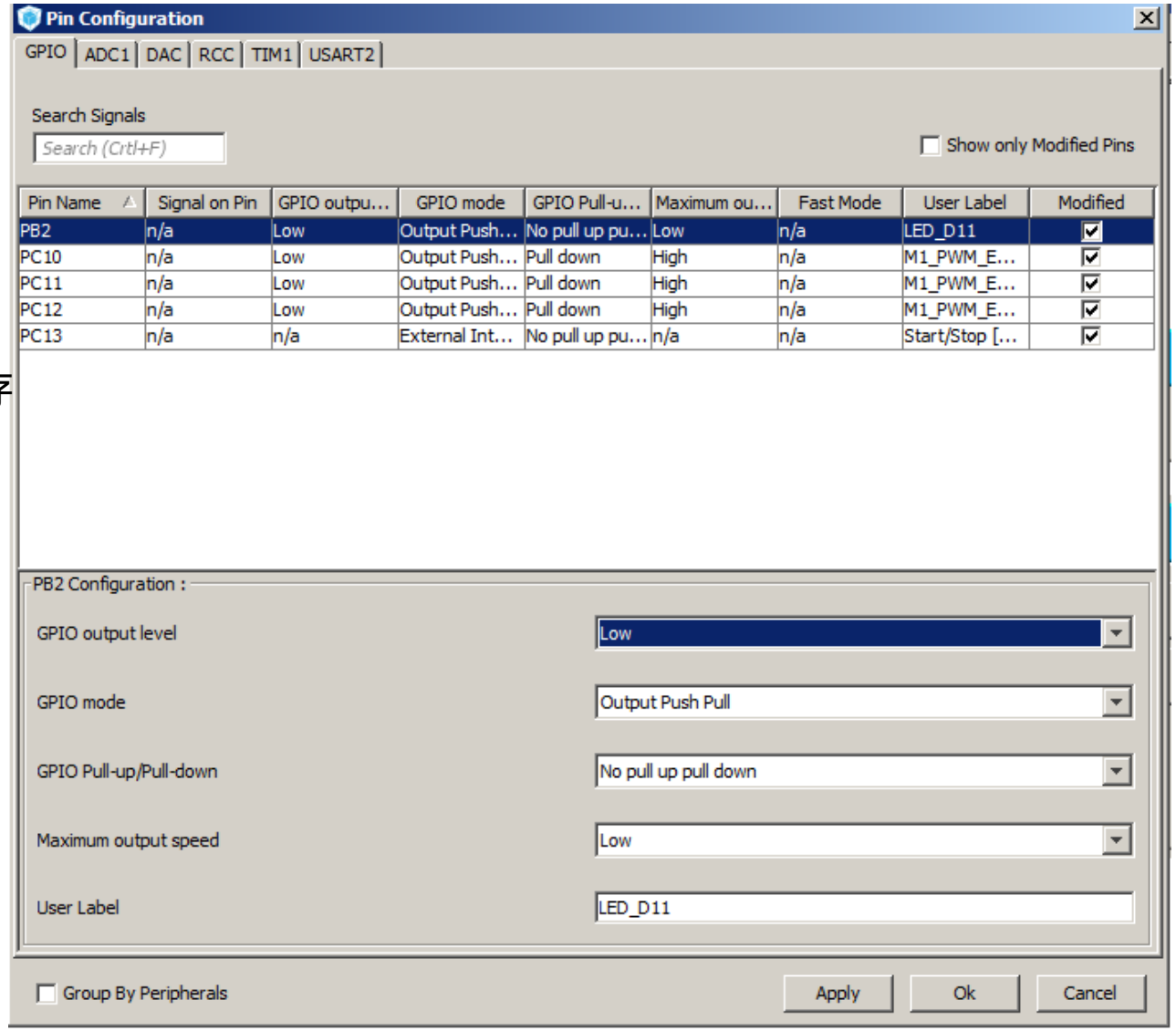
Interrupt Table	Enabled	Preemption Priority	Sub Priority
Non maskable interrupt	☑	0	0
Hard fault interrupt	☑	0	0
Memory management fault	☑	0	0
Pre-fetch fault, memory access fault	☑	0	0
Undefined instruction or illegal state	☑	0	0
System service call via SWI instruction	☑	0	0
Debug monitor	☑	0	0
Pendable request for system service	☑	0	0
Time base: System tick timer	☑	4	0
PVD interrupt through EXTI line 16	☐	0	0
Flash global interrupt	☐	0	0
RCC global interrupt	☐	0	0
ADC1 interrupt	☑	2	0
TIM1 break and TIM15 interrupts	☑	4	1
TIM1 update and TIM16 interrupts	☐	0	0
TIM1 trigger, commutation and TIM17 interrupts	☐	0	0
TIM1 capture compare interrupt	☐	0	0
TIM2 global interrupt	☑	3	0
USART2 global interrupt	☑	3	1
EXTI line[15:10] interrupts	☑	0	0
TIM6 global interrupt, DAC interrupts	☐	0	0
Floating point unit interrupt	☐	0	0

life.augmented

Step 6：修改PB2的初始输出状态

- 在 **Configuration** 窗口点击 **GPIO** 按钮。出现如下 **Pin Confuguration** 窗口。
- 在 Pin Confuguration 窗口中设置 **GPIO output level** 为 **low**。

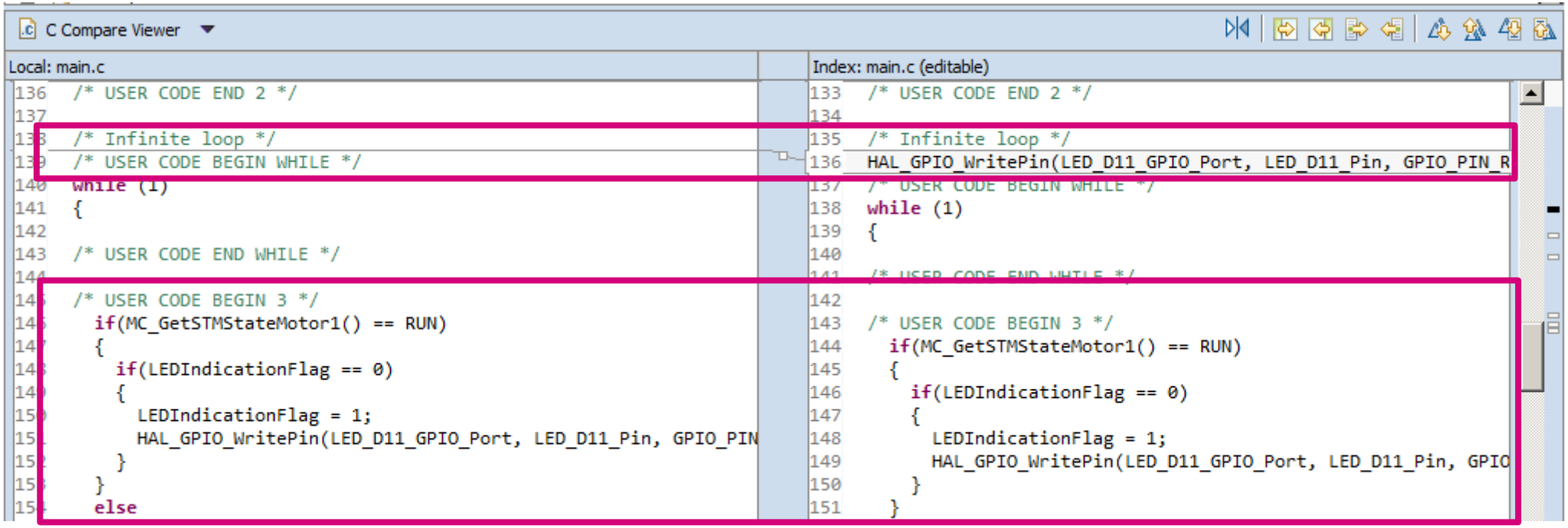
到这一步我们已经完成了所有的CubeMX的设置。保存并重新生成代码。



Step 7：修改代码

对比新生成的代码我们看到下面的文件发生了变 **main.c**， **stm32f3xx_hal_msp.c**， **stm32f3xx_it.c**， **stm32f3xx_it.h**

这里着重提示一点：从下图中可以看到之前写在 **USER CODE BEGIN** 和 **USER CODE END** 之间的代码仍然存在。**写在USER CODE BEGIN 和 USER CODE END之外的代码被覆盖。**



Step 7：修改代码

添加 TIM2 启动的代码。CubeMX 只是配置了 TIM2， 要启动 TIM2 还需要添加如下代码。

main.c

main()

```
130   MX_TIM2_Init();
131
132   /* Initialize interrupts */
133   MX_NVIC_Init();
134   /* USER CODE BEGIN 2 */
135   HAL_TIM_Base_Start_IT(&htim2);
136   /* USER CODE END 2 */
```

我们在 main.c 中的 main 函数配置完 TIM2 和中断之后调用了函数 HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim)。这个函数启动 TIM2 的计数器并使能 update 中断。函数的具体定义如下

```
HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim)
{
  /* Check the parameters */
  assert_param(IS_TIM_INSTANCE(htim->Instance));

   /* Enable the TIM Update interrupt */
   __HAL_TIM_ENABLE_IT(htim, TIM_IT_UPDATE);

   /* Enable the Peripheral */
  __HAL_TIM_ENABLE(htim);

  /* Return function status */
  return HAL_OK;
}
```

life.augmented

Step 7：修改代码

在右图 TIM2 的中断服务程序中将变量 Flag500ms 置为1。以此来标识 500ms 的时间周期。这个中断服务程序可以由 CubeMX 生成，我们只需要在函数中 USER CODE BEGIN 和 USER CODE END 之间添加自己的应用代码。

```
/**
* @brief This function handles TIM2 global interrupt.
*/
void TIM2_IRQHandler(void)
{
  /* USER CODE BEGIN TIM2_IRQn 0 */
  Flag500ms = 1;
  /* USER CODE END TIM2_IRQn 0 */
  HAL_TIM_IRQHandler(&htim2);
  /* USER CODE BEGIN TIM2_IRQn 1 */

  /* USER CODE END TIM2_IRQn 1 */
}
```

另外我们对在 main 函数中的用户代码做了如右图的修改。

```
/* USER CODE BEGIN 3 */
  if(MC_GetSTMStateMotor1() == RUN)
  {
    if(Flag500ms == 1)
    {
      Flag500ms = 0;
      if(LEDIndicationFlag == 0)
      {
        LEDIndicationFlag = 1;
        HAL_GPIO_WritePin(LED_D11_GPIO_Port, LED_D11_Pin, GPIO_PIN_SET);
      }
      else
      {
        LEDIndicationFlag = 0;
        HAL_GPIO_WritePin(LED_D11_GPIO_Port, LED_D11_Pin, GPIO_PIN_RESET);
      }
    }
  }
  else
```

life.augmented

小结

- **Motor Control Workbench** 初次生成代码和用于 **CubeMX** 的 **.ioc**文件。
- 添加非电机驱动的外设配置可以打开由 **Workbench** 生成的 **.ioc** 文件，然后在 **CubeMX** 中配置并重新生成代码。
- 用户代码要写在 **USER CODE BEGIN** 和 **USER CODE END** 之间。

Releasing your creativity

- Thank you -

life.augmented

重要通知 – 请仔细阅读

意法半导体公司及其子公司（“ST”）保留随时对ST 产品和/ 或本文档进行变更、更正、增强、修改和改进的权利，恕不另行通知。买方在订货之前应获取关于ST 产品的最新信息。ST 产品的销售依照订单确认时的相关ST 销售条款。

买方自行负责对ST 产品的选择和使用， ST 概不承担与应用协助或买方产品设计相关的任何责任。

ST 不对任何知识产权进行任何明示或默示的授权或许可。

转售的ST 产品如有不同于此处提供的信息的规定，将导致ST 针对该产品授予的任何保证失效。

ST 和ST 徽标是ST 的商标。所有其他产品或服务名称均为其各自所有者的财产。

本文档中的信息取代本文档所有早期版本中提供的信息。

版权声明