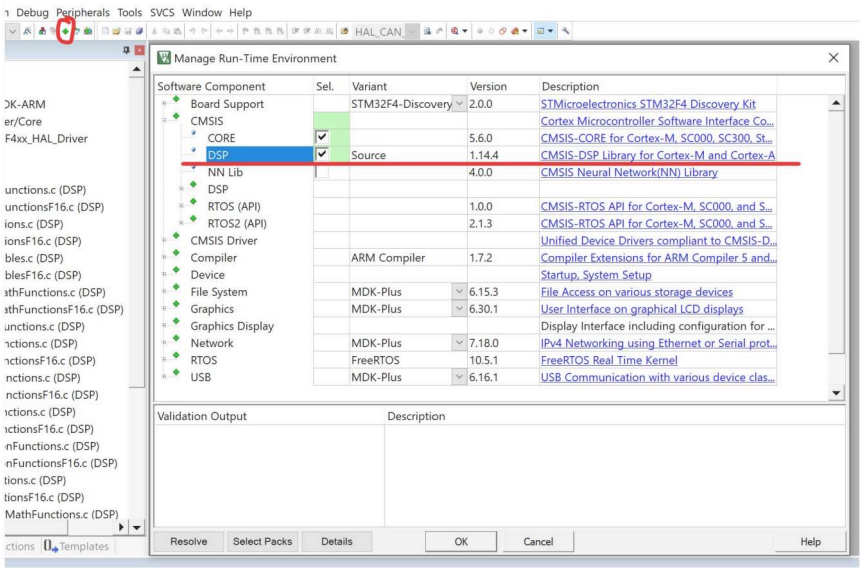
使用CMSIS-DSP库进行PID控制

CMSIS-DSP是针对嵌入式系统的优化计算库，支持Cortex-M和Cortex-A内核，可以利用内核的FPU、DSP指令，提高算法的性能。这个库为我们提供了针对内核优化的向量计算、矩阵运算、数字信号处理、电机控制、统计和机器学习算法。

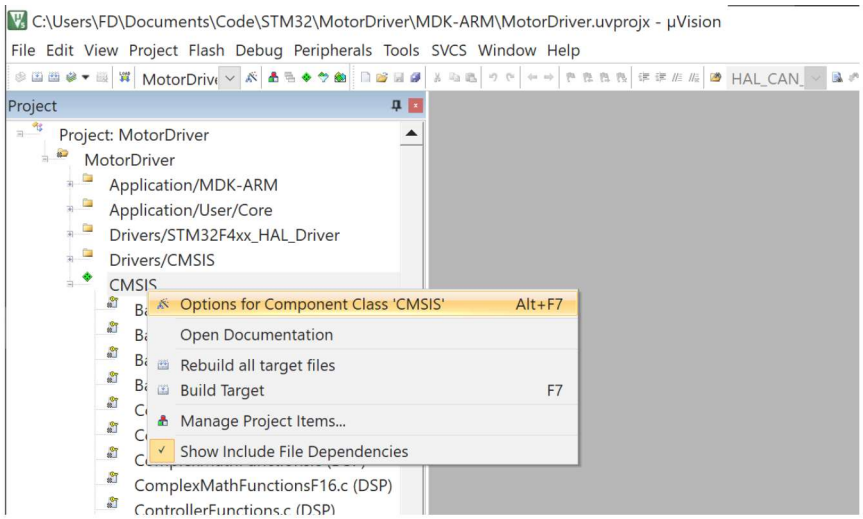
本文将介绍如何使用CMSIS-DSP库，在STM32单片机上，构建增量式PID控制程序。

引入CMSIS-DSP库

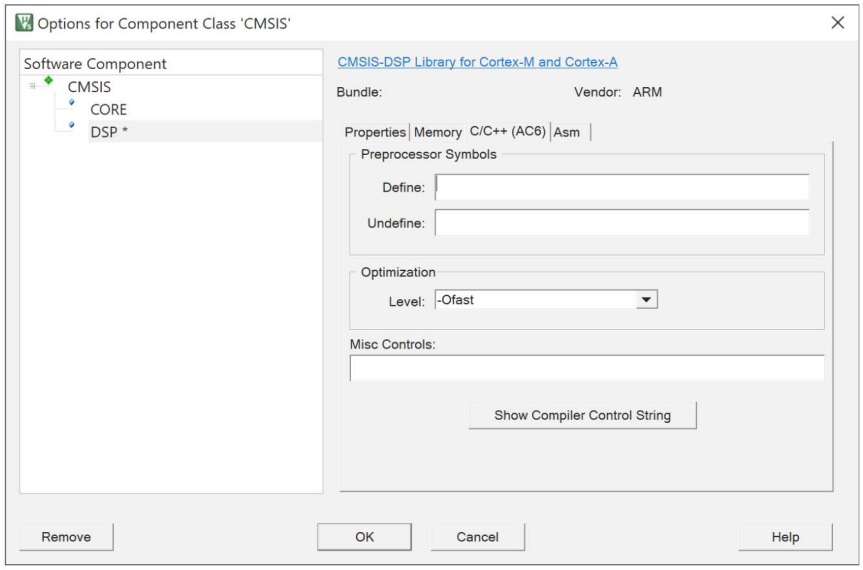
在Keil MDK-Arm中引入CMSIS-DSP库是非常方便的。建立好适用于MDK-Arm的STM32工程后，单击`Manage Run-Time Environment`，进入MDK-Arm提供的包管理器界面。随后，展开CMSIS，单击DSP右侧的单选框，即可完成库的导入。



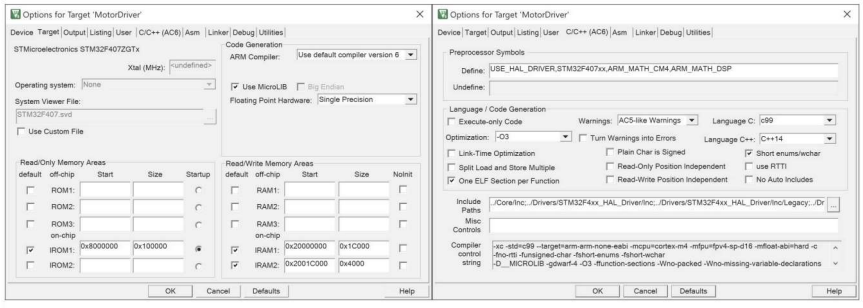
在项目管理器中，右键单击`CMSIS`库，选择`Options for Component Class 'CMSIS'`，配置DSP库的编译选项。



在弹出的界面中，选择DSP库的`C/C++`编译选项，开启`-Ofast`优化。



如果您的设备支持FPU，可以在工程的编译选项中，使能`Floating Point Hardware`并添加`ARM_MATH_CM4`和`ARM_MATH_DSP`宏，让CMSIS-DSP库能够利用硬件实现算法的加速。



重新编译工程，若未发现错误，则说明我们已经成功引入了CMSIS-DSP库。

CMSIS-DSP提供的PID算法原理与公式推导

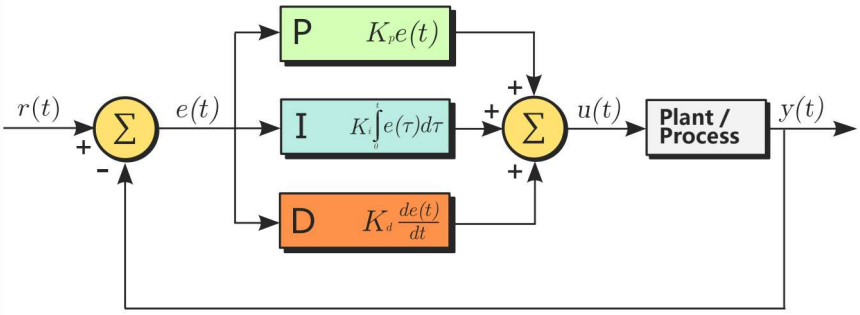
PID控制器（比例-积分-微分控制器），由比例单元（Proportional）、积分单元（Integral）和微分单元

（**D**erivative）组成，是一种在工业控制应用中常见的反馈回路部件。PID控制器可以根据历史和当前的数据，来调整对系统的控制，使系统更加准确而稳定。

PID控制器的输出是关于输入量与被控量偏差的函数，是比例单元、积分单元和微分单元的线性组合，即：

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_i \frac{de(t)}{dt} \tag{1}$$

一个完整的PID控制系统的系统框图如下所示：



在本文中，我们称这个系统中的：

- $y(t)$为**被控量**，表示我们需要控制的被控对象的输出量；
- $r(t)$为**输入量**，是我们希望被控量在t时刻达到的状态或目标；
- $e(t)$为输入量与被控量之间的**偏差**，即：$e(t) = r(t) - y(t)$；
- $u(t)$为**操纵量**（执行元件的输入）；
- K_p、K_i、K_d分别是比例项、积分项和微分项的比例系数，也是我们后续进行PID**参数整定**所需要调整的参数。

当然，我们在使用MCU进行编程时，我们往往用的是PID控制器的离散形式，即将积分项变为累加项，微分项变为差分项，如下式所示：

$$u[t] = K_p e[t] + K_i \sum_{\tau=0}^{t} e[\tau]\Delta t + K_d \frac{e[t] - e[t-1]}{\Delta t} \tag{2}$$

式中，Δt为两次相邻采样的时间间隔。我们称这样的PID控制方式为**位置式PID**。

考虑相邻时刻，操纵量$u[t]$的变化量$\Delta u[t] = u[t] - u[t-1]$，我们可以得到：

$$\Delta u[t] = K_p \left(e[t] - e[t-1]\right) + K_i e[t]\Delta t + \frac{K_d}{\Delta t}\left(e[t] - 2e[t-1] + e[t-2]\right) \tag{3}$$

将上式化简，我们发现，PID控制器操纵量的增量，是当前时刻t、$t-1$时刻和$t-2$时刻的偏差的线性组合，即：

$$\Delta u[t] = (K_p + K_i\Delta t + \frac{K_d}{\Delta t})e[t] - (K_p + \frac{2K_d}{\Delta t})e[t-1] + \frac{K_d}{\Delta t}e[t-2] \tag{4}$$

我们称通过计算系统输入的变化量$\Delta u[t]$实现PID控制的控制方式，称为**增量式PID。CMSIS-DSP库中提供的PID算法，使用增量式PID的实现。**在增量式PID控制器中，有：

$$u[t] = u[t-1] + \Delta u[t] \tag{5}$$

在CMSIS-DSP中的PID实现中，认为$\Delta t = 1$（或者说假定用户已经将Δt加权至K_i和K_d构成新的K_i和K_d）。那么，$u[t]$的变化量$\Delta u[t]$可以用下面的式子如式(6)：

$$\Delta u[t] = A_0 e[t] + A_1 e[t-1] + A_2 e[t-2] \tag{6}$$

其中：

$$\begin{aligned} A_0 &= K_p + K_i + K_d \\ A_1 &= -K_p - 2K_d \\ A_2 &= K_d \end{aligned} \tag{7}$$

因此，在增量式PID系统中，合并式(5)和式(6)，我们可以通过式(8)计算操纵量$u[t]$。这个式子，也是CMSIS-DSP库中，PID的实现原理：

$$u[t] = u[t-1] + A_0 e[t] + A_1 e[t-1] + A_2 e[t-2] \tag{8}$$

对于拥有DSP的指令的芯片，在编译器强大的优化功能下，可以在一个指令周期内完成式(8)所示的乘加运算。这大大提高了PID的计算速度。

CMSIS-DSP库的PID控制接口

数据类型

CMSIS-DSP库的PID控制器，有基于三种基于不同数据类型的实现。这三种数据类型分别是`q15`，`q31`和`f32`，分别表示：16位有符号整数`int16_t`、32位有符号整数`int32_t`、32位浮点数`float32_t`。

在CMSIS-DSP库实现的PID控制器，函数和结构体的命名都是“名称+数据类型后缀”的命名方式。例如：`arm_pid_instance_q15`、`arm_pid_instance_q31`和`arm_pid_instance_f32`是基于不同数据类型实现的PID控制的结构体；`arm_pid_q15`、`arm_pid_q31`和`arm_pid_f32`是基于不同数据类型实现的计算PID增量的函数。

在接下来的文段中，我们将以`f32`类型为例，介绍如何使用CMSIS-DSP库的PID控制器进行PID控制，若要使用其它数据类型的PID实现，只需按照这种命名方式，换用相应数据类型的函数、结构体即可。

PID控制结构体的定义

在CMSIS-DSP中，PID的参数信息、历史误差信息，都被记录在`arm_pid_instance_xxx`结构体中，例如：

```
/**
 * @ingroup PID
 * @brief Instance structure for the floating-point PID Contro
 */
typedef struct {
    float32_t A0;         /**< The derived gain, A0 = Kp + Ki +
    float32_t A1;         /**< The derived gain, A1 = -Kp - 2Kd
    float32_t A2;         /**< The derived gain, A2 = Kd . */
    float32_t state[3];   /**< The state array of length 3. */
    float32_t Kp;         /**< The proportional gain. */
    float32_t Ki;         /**< The integral gain. */
    float32_t Kd;         /**< The derivative gain. */
} arm_pid_instance_f32;
```

若我们认为当前时刻为t时刻，则结构体中各个参数的定义如下所示：

1. `Kp`、`Ki`和`Kd`需要我们自行设置，分别是比例项、积分项和微分项的比例系数K_p、K_i和K_d；
2. `A0`、`A1`和`A2`分别对应式(7)中的A_0，A_1和A_2，其具体意义与前文所述一致；
3. `state`是PID控制的状态数组。
    1. `state[0]`表示$t-1$时刻的偏差，即$e[t-1]$；
    2. `state[1]`表示$t-2$时刻的偏差，即$e[t-2]$；
    3. **`state[2]`表示的是前1时刻的操纵量，即**$u[t-1]$

PID控制结构体的初始化

当我们在初始化这个结构体的时候，我们需要且只需手动对`Kp`、`Ki`和`Kd`这三个成员进行赋值。之后，我们需要调用`arm_pid_init_xxx`函数，初始化这个结构体。

`arm_pid_init_xxx`函数接受两个参数，第一个参数是PID控制结构体的指针，第二个参数是一个整数标志位，当它为0时，只计算`A0`、`A1`和`A2`的值，不初始化`state`数组；当它为1时，则初始将`state`数组，将其置为0。在编程中，如果我们首次初始化该结构体，则应当将`resetStateFlag`置为1；当我们非首次初始化（如PID参数整定过程中），我们需要更新`Kp`、`Ki`或`Kd`的值，我们可以将其置为0，以提高性能。

这个函数初始化的过程分两步：

1. 利用`Kp`、`Ki`和`Kd`，通过我们先前推导的公式，计算`A0`、`A1`和`A2`的值。
2. 若`resetStateFlag`参数为1，则将`state`数组置为0。

```
/**
 * @brief         Initialization function for the floating-poi
 * @param[in,out] S               points to an instance of the
```

```
 * @param[in]     resetStateFlag
 *                  - value = 0: no change in state
 *                  - value = 1: reset state
 * @return        none
 */
void arm_pid_init_f32(arm_pid_instance_f32 *S, int32_t resetSt
    /* Derived coefficient A0 */
    S->A0 = S->Kp + S->Ki + S->Kd;
    /* Derived coefficient A1 */
    S->A1 = (-S->Kp) - ((float32_t) 2.0f * S->Kd);
    /* Derived coefficient A2 */
    S->A2 = S->Kd;
    /* Check whether state needs reset or not */
    if (resetStateFlag) {
        /* Reset state to zero, The size will be always 3 samp
        memset(S->state, 0, 3U * sizeof(float32_t));
    }
}
```

当我们进行PID参数整定时，需要不断地调整K_p、K_i和K_d的值。**我们每次调整K_p、K_i和K_d的值，都应该调用这个函数，重新计算A_0、A_1和A_2的值。**但是此时，我们可以不重置`state`数组。

操纵量的更新

作为一种反馈控制算法，PID控制算法是一种按偏差进行控制的过程。由于扰动或输入量变化等因素的影响，偏差往往不恒为o；而我们的控制系统，为了达到尽量让偏差为o的目标，需要对输出量进行采样，与输入量不断地更新操纵量$u[t]$，以使得被控量贴合输入量。

对于增量式PID算法，其核心步骤就在于计算操纵量的变化量$\Delta u[t]$，并更新操纵量$u[t]$。$u[t]$的计算，可以通过`arm_pid_xxx`进行计算。这个函数的**输入是当前时刻的偏差$e[t]$，输出的是操纵量$u[t]$。**

值得注意的是，`arm_pid_xxx`函数不会对输出进行限幅，关于这一点，我们将在后续进行更深入的讨论。

```
/**
 * @ingroup PID
 * @brief         Process function for the floating-point PID
 * @param[in,out] S   is an instance of the floating-point PID
 * @param[in]     in  input sample to process
 * @return        processed output sample.
 */
__STATIC_FORCEINLINE float32_t arm_pid_f32(arm_pid_instance_f3
    float32_t out;

    /* u[t] = u[t - 1] + A0 * e[t] + A1 * e[t - 1] + A2 * e[t
    out = (S->A0 * in) +
      (S->A1 * S->state[0]) + (S->A2 * S->state[1]) + (S->stat

    /* Update state */
    S->state[1] = S->state[0];
    S->state[0] = in;
    S->state[2] = out;
```

```
19
20      /* return to application */
21      return (out);
22  }
```

状态的清除

在`arm_pid_xxx`函数中，是否清空状态数组`state`取决于我们输入的参数。当我们未修改`Kp`、`Ki`或`Kd`的值，不需要重新更新`A0`、`A1`和`A2`的值时，若我们要清空状态数组`state`时，可以使用`arm_pid_reset_xxx`函数。

```
1  /**
2   * @brief         Reset function for the floating-point PID Co
3   * @param[in,out] S  points to an instance of the floating-poi
4   * @return        none
5   * @par           Details
6   *                The function resets the state buffer to zero
7   */
8  void arm_pid_reset_f32(arm_pid_instance_f32 *S) {
9      /* Reset state to zero, The size will be always 3 samples
10     memset(S->state, 0, 3U * sizeof(float32_t));
11 }
12
```

PID控制的实现

基础控制

根据上面的介绍，我们使用PID进行积分控制的方法，也就呼之欲出了。其伪代码如下所示：

```
1  void pid_control(float Kp, float Ki, float Kp, float delta_t)
2      // 在CMSIS-DSP中的PID实现中，认为Δt = 1，
3      // 故需要调整Kp和Ki的值，保证系统在采样率变化时的鲁棒性
4      arm_pid_instance_f32 controller = {
5          .Kp = Kp,
6          .Ki = Ki * delta_t,
7          .Kd = Kd / delta_t
8      };
9      arm_pid_init_f32(&controller, 1);       // 初始化结构体，要
10     while (1) {
11         const float r = get_reference();    // 读取当前系统的输
12         const float y = get_status();       // 读取当前的被控量
13         const float error = r - y;          // 计算偏差
14         const float u = arm_pid_f32(&controller, error);    //
15         execute(u);        // 使用最新的操纵量，调整执行元件，实现
16         delay(delta_t);    // 实际可以使用定时器等机制，实现循环
17     }
18 }
```

输出限幅的控制

在实际工程中，受硬件条件的限制，操纵量往往是有一定范围的。如矩形波占空比的大小，只能是0~100%；输出电压的大小，不会超过系统中的最高电压；电流的大小，也不会越过欧姆定律的限制……

但是，CMSIS-DSP库的PID实现，为了追求极致的效率，未将这一点纳入考量。当我们在这样的控制系统中应用上面的控制程序，就有可能出现$u[t]$趋向于无穷的情况。此时，当PID控制器的输入量发生变化，PID控制器可能不会及时的响应。这时，我们要人为地限定操纵量的大小，使其符合工程实际。我们称这样的操作为“输出限幅”。

使用CMSIS-DSP库实现PID，当我们需要进行输出限幅地时候，需要修改`stete`数组的第三个成员——记录着上一时刻操纵量的$u[t-1]$的`state[2]`成员，保证其在合理的范围内，避免超出工程实际的幅值。具体实现如下所示：

```
void pid_control(float Kp, float Ki, float Kp, float delta_t)
    // 在CMSIS-DSP中的PID实现中，认为Δt = 1，
    // 故需要调整Kp和Ki的值，保证系统在采样率变化时的鲁棒性
    arm_pid_instance_f32 controller = {
        .Kp = Kp,
        .Ki = Ki * delta_t,
        .Kd = Kd / delta_t
    };
    arm_pid_init_f32(&controller, 1);       // 初始化结构体，要
    while (1) {
        const float r = get_reference();    // 读取当前系统的输
        const float y = get_status();       // 读取当前的被控量
        const float error = r - y;          // 计算偏差
        arm_pid_f32(&controller, error);    // 计算PID的操纵量
        // 进行输出限幅
        if (controller.state[2] > OUTPUT_MAX) {
            controller.state[2] = OUTPUT_MAX;
        } else if (controller.state[2] < OUTPUT_MIN) {
            controller.state[2] = OUTPUT_MIN;
        }
        // 完成输出限幅，用限幅后的操纵量控制系统
        execute(controller.state[2]);   // 使用最新的操纵量，调
        delay(delta_t);      // 实际可以使用定时器等机制，实现循
    }
}

```

总结

关于使用CMSIS-DSP库实现PID控制，无论是中外的互联网上，资料都很少。官方文档目前仍没有提供例程。但是当我们理解了增量式PID的计算公式后，再来看CMSIS-DSP库的PID控制接口，实际上它是非常简单，甚至是简陋的。这也是为什么我以解读源码的方式，写下这篇文章。

CMSIS-DSP库的PID控制接口，其优化思路，主要还是在于写出利于编译器优化的代码（如将复杂的PID计算过程，转换为式(8)所示的乘加运算），借助现代编译器强大的优化功能，提高代码的性能。此外，对于被反复调用的`arm_pid_xxx`函数，这个库也使用了`__STATIC_FORCEINLINE`，强制编译器进行内联。这些优化技巧，是值得我们学习的。

参考文献

1. https://github.com/ARM-software/CMSIS-DSP
2. https://en.wikipedia.org/wiki/PID_controller
3. https://arm-software.github.io/CMSIS-DSP/latest/
4. https://arm-software.github.io/CMSIS-DSP/latest/group__PID.html

标签: 嵌入式 , STM32 , Arm , CMSIS , PID , 自动控制 , 数字信号处理