韶关学院
SHAOGUAN UNIVERSITY

数字信号控制（DSC）技术基础

物理与机电学院自动化系
宁 宇

第六节 定标、IQmath函数库和标么化计算

1定点与浮点计算

2定点运算中数字的定标

3 IQmath函数库

4 标么化系统

1定点与浮点计算

定点与浮点DSP的基本差异在于它们对数据的数字表示法不同定点DSP严格执行整数运算，而浮点DSP既支持整数运算又支持实数运算，后者以科学计数法进行了标准化。浮点DSP将数据路径分为两部分：一是可用作整数值或实数基数的尾数，二是指数。业界标准单一精确运算的32位浮点DSP中，尾数是24位，指数是8位。动态范围大大高于定点格式提供的精确度。

浮点DSP需要的内部电路多，32位数据路径比用定点器件宽1倍。晶片面积越大，引脚数量也越多，导致封装越大，成本也更高。浮点格式中，实数运算可直接通过代码加入硬件运算中，而定点器件则须通过软件才能间接运行实数运算。增加了算法指令与延长了开发时间。浮点最初用于开发工作强度较大的情况。

定点DSP 体积小、功耗低、价格便宜, 而且现在的定点产品的速度已经可以做得很高,然而, 随之而来的问题是如何在精度要求严格的应用中, 用定点DSP 保持较高的运算精度。

F28xx的IQmath函数库

F28XX是定点DSP，但它提供了IQmath库模块，主要完成处理器优化和定点数学运算。它是在汇编库基础上创建的，可在定点DSP实现精确的浮点运算，使用库函数可方便用户进行编写浮点处理程序。对于要求高实时和高精度的系统这些函数库尤其有用。与直接用ANSI C相比，速度上明显提高且精度也很好。

2定点小数运算原理与DSP的定标

定点数运算时，操作数采用整型数表示。最大表示范围取决于DSP的字长。DSP的芯片的数以2的补码形式表示。用一位数表示数的正负，0为正，1为负。其余15位为数值大小。

二进制数0010000000000011b=8195
二进制数1111111111111100b= -4

处理小数时需确定小数处于16位数据中的具体位置。即定标。定标的表示法有Q法和S法。Q法仅列出小数的位数，S法要列出整数位置、小数点和小数位数。Q12或S4.12表示4位整数12位小数。小数的分辨率为

$$\frac{1}{2^{12}} = 0.000244140625$$

表1.1　Q表示、S表示及数值范围

Q表示	S表示	十进制数表示范围
Q15	S0.15	-1≤x≤0.9999695
Q14	S1.14	-2≤x≤1.9999390
Q13	S2.13	-4≤x≤3.9998779
Q12	S3.12	-8≤x≤7.9997559
Q11	S4.11	-16≤x≤15.9995117
Q10	S5.10	-32≤x≤31.9990234
Q9	S6.9	-64≤x≤63.9980469
Q8	S7.8	-128≤x≤127.9960938
Q7	S8.7	-256≤x≤255.9921875

表1.1列出了一个16位数的16种Q表示、S表示及它们所能表示的十进制数值范围。

A 同样一个16位数，若小数点设定的位置不同，它所表示的数也就不同。例如，

16进制数2000H=8192，用Q0表示

16进制数2000H=0.25，用Q15表示

但对于DSP芯片来说，处理方法是完全相同的。

B 不同的Q所表示的数不仅范围不同，而且精度也不相同。Q越大，数值范围越小，但精度越高；相反，Q越小，数值范围越大，但精度就越低。例如，Q0 的数值范围是一32768到+32767，其精度为1，而Q15的数值范围为-1到0.9999695，精度为1/32768=0.00003051。

对定点数而言，数值范围与精度是一对矛盾，一个变量要想能够表示比较大的数值范围，必须以牺牲精度为代价；而想精度提高，则数的表示范围就相应地减小。在实际的定点算法中，为了达到最佳的性能，必须充分考虑到这一点。

16位的DSP中加法/减法运算

Qx Qy

1 Qx>Qy, Qz=Qx,y<<(Qx-Qy),

z=x+y;

2Qz=Qy x>>(Qx-Qy),

z=x+y

如果X，Y，Z的定标互不相同，则需将X，Y重新定标再运算。定标就是小数点对齐的过程。

$$x = 1.5$$

$$y = 0.8$$

$$Qx = 14$$

$$Qy = 13$$

$$x = 24576, y = 6553$$

$$Qz = 14$$

$$x >> (14 - 13) = 12288$$

$$z = x + y = 12288 + 6553 = 18841$$

$$z = 18841 \bullet \frac{1}{2^{13}} = \frac{18842}{8192} = 2.3$$

乘法运算

Qx,Qy,Qz,Qz'=Qx+Qy是32位

1 Qz=Qx,Z>>Qy,则取低16位为乘积值；

2 Qz=Qy，z>>Qx,则取低16位作为乘积值。

$$x = 1.5$$

$$y = 0.8$$

$$Qx = 14$$

$$Qy = 13$$

$$x = 24576, y = 6553$$

$$z = xy = 24576 \times 6553 = 161046528 = \$999600$$

$$Qz = 13$$

$$z = (\$999600) >> 14 = \$2665 = 9829$$

$$z = 9829 \bullet \frac{1}{2^{13}} = \frac{9829}{8192} \approx 1.2$$

除法运算

Qx,Qy,Qz,若Z=X/Y，Qz'=Qx-Qy。

$Qz \neq Qz'$，$Qz > Qz'$，商需要左移 $Qz - Qz'$ 位；

$Qz < Qz'$，商需要右移 $Qz' - Qz$ 位。

$x = 1.25$

$y = 0.8$

$Qx = 12$

$Qy = 10$

$x = 5120, y = 819$

$z = x / y = 5120 / 819 = 6$

$z = \$0006 << Qz - Qz' = 10 - (12 - 10) = 8 = \$600 = 1536$

$$z = 1536 \bullet \frac{1}{2^{10}} = \frac{1536}{1024} \approx 1.5$$

1.25除以0.8实际值为1.5625，而经过定标后的结果为1.5，有较大误差。主要原因是，在计算过程中，由于作除法运算时商的定标值等于被除数与除数定标值之差，因此商的精度大大降低，从而产生较大的误差。为防止这种现象发生，可在作除法运算前，首先将被除数的定标值提高，使之等于除数定标值与商的定标值之和，再作除法运算，就可保证商的精度了。

首先，将被除数X重定标为Qx+Qy=20

$$x = 1310720, y = 819$$

$$z = x / y = 1310720 / 819 = \$640 = 1600$$

$$Qz = Qz'$$

$$z = 1536 \bullet \frac{1}{2^{10}} = \frac{1600}{1024} \approx 1.5625$$

经过数字定标的变量在运算中要注意以下几个问题：

1 溢出

由于定点数的表示范围是一定的，运算时其结果有可能出现超出数值表示范围的情况。称为溢出。大于最大值，称为上溢；小于最小值，称为下溢。无论何种溢出，都会产生意想不到的结果。必须采取保护性措施。

A自动增加字长 B将定标值减为1，C饱和处理，即溢出后为最大值/最小值。

2 舍入及截尾

对某个数的取整处理有舍入法和截尾法两种。即上取整和下取整法。DSP由于实际操作数是整数，所以采取截尾法取整。为提高精度，运算后定标的数，通过取整运算可得到不同的有效值的小数。一般舍入误差的绝对值小于截尾误差的绝对值值。

采用固定Q15定标的运算规则

采用固定Q15的定标运算可避免计算过程中反复进行移位的麻烦，也简化了算法，提高了算法的可移植性。同时，Q15也可保证较高的运算精度。

加法 只需要考虑饱和处理。

Ia=20446(0.624), Ib=3276(0.1)

Ic=Ia+Ib=20446+3276=23722(0.724)

乘法

两个Q15相乘得到32位结果；由于乘积也是Q15，因此，只需将32位乘积左移1位即可，取高16位为最终乘积即可。

a=0.6 b=0.5 A=19660,B=16383

A*B=322089780=$1332B334

左移一位（乘以2）再取高16位则

（$1332B334）<<1=$26656668

C=$2665=9829

实际值为

c=C/32767=0.29996

除法

先将被除数左移15位，用32位数表示被除法（与乘法对偶），再进行除法运算。注意的是，被除数必须小于除数；否则结果绝对值大于1，超出了Q15的表示范围。这时可适当处理。最简单的办法是先除被除数缩小N倍，再将结果扩大N倍。保证结果不变。

a=0.6,b=0.5,A=19660=$4CCC,B=16383=$3FFF 则

C=（B<<15）/A=$1FFF800/$4CCC=27306

实际值为

c=C/32767=0.83333

3 IQmath函数库

针对不同系统或同一系统不同模块的需求，用户可利用IQmath库函数更灵活地选择合适的数据格式。但由于数据精度和动态范围本身是相互矛盾的两个指标，软件设计中需要根据具体要求来确定，进行折中以便达到系统最优。默认格式为Q24。各模块对精度和动态范围要求不同，需要定义局部Q格式，以便在具体的模块中取代全局GLOBAL_Q格式。

格式变换函数atoIQ,IqtoF IQtoIQN

算术运算函数 Iqmpy IQdiv

三角运算函数 Iqsin Iqcos IQatan2

数学计算函数 Iasqrt IQisqrt

其它函数 Iqabs IAsat

GLOBAL_Q格式的选择

由于数据精度和动态范围本身相互矛盾，必须根据具体要求确定其中的折中，以达到系统最优。默认为Q24。系统中各种模块Q格式不一样，因此，应定义局部Q格式，以便在具体模块中取代全局Q格式。由于IQmathTables包含Iqmath函数使用的所有查表数据已固化在BOOTROM中，.cmd文件该段必须定义为NOLOAD类型。程序会自动定位查表符号。使用Simulator时，由于没有目标板，须将库中的数据表一并加载。

但是由于定点DSP 本身不能进行浮点运算,再加上目前C 编译器的优化功能还不尽完善,用C 语言编写的浮点运算程序在定点DSP 上的执行效率远远低于人们的预期。为此许多工程技术人员在开发定点DSP 程序时经常将浮点运算转换为定点运算,但是由程序员自己手工实现三角函数、对数等数学函数的定点运算是非常烦琐的工作,费时费力,

而且程序的可靠性也难以保证。TI 公司推出的针对C28X 系列定点DSP 的IQmath 数学函数库,用定点算法优化实现了一些常用的数学函数,在一定程度上解决了这个问题。合理使用这些函数,可以大幅度的提高C 语言浮点运算程序的执行效率。

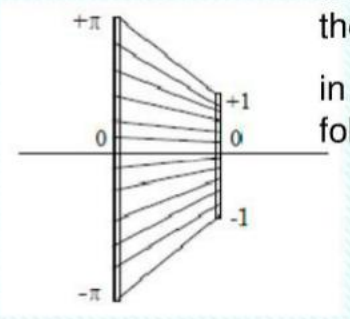
c编译器带有浮点运算库，因此可将浮点算法和定点算法的结果进行比较，对于4路各1024点数据处理，用浮点算法实现约需3．6秒，而用定点算法只需1．3秒。

F2812为定点芯片，而实际的数据为浮点数，所以要将浮点数转化为定点数之后才能运算，这就涉及到数的定标问题。TI公司推出的数学函数库(IQmath.lib)，用它来进行数的运算，可以缩短开发周期。因为这个函数库是高精度、高优化的函数库，它提供了标准的C／C++语言的无缝连接。

Function Name	Execution Cycles	Accuracy	Program Memory	Input format	Output format
QSIN	33 cycles	15 bits	31 words	Normalized Q1.15	Signed Q1.15
QSINLT	25 cycles	14 bits	17 words	Normalized Q1.15	Signed Q1.15
QCOS	33 cycles	15 bits	32 words	Normalized Q1.15	Signed Q1.15
QCOSLT	25 cycles	14 bits	19 words	Normalized Q1.15	Signed Q1.15
QATAN	65 cycles	15 bits	50 words	Signed Q16.16	Normalized Q1.15
QSQRT	45 cycles	14.5 bits	44 words	Unsigned Q16.16	Unsigned Q8.8
QLOG10	38 cycles	15.5 bits	38 words	Unsigned Q16.16	Signed Q4.12
QLOGN	38 cycles	15.5 bits	38 words	Unsigned Q16.16	Signed Q5.11
QINV1	51 cycles	32 bits	15 words	Signed Q(x)	Signed Q(31-x)
QINV2	35 cycles	16 bits	14 words	Signed Q(x)	Signed Q(15-x)
QDIV	54 cycles	32 bits	16 words	Signed Q(x), Q(y)	Signed Q(16+x-y)

C28x QMATH LIBRARY

$$\sin(x) = 3.1406625x + 0.0202636\,7x^2 - 5.325196x^3 + 0.5446778x^4 + 1.800293x^5$$



the x is the radians within the range -∏ to + ∏ , then the normalized value of 'x'

in Q15 representation can be obtained by the following simple equation

$$X = \frac{x}{\pi} \times 2^{15}$$

C/C-Callable ASM Interface

Declaration	int qsin(int x)
Input Format	The argument 'x' is a fixed-point number in Q15 format that contains the angle in radians between $[-\pi,+\pi]$ normalized between $[-1,+1]$
Output Format	This function returns the sine of the input argument as fixed-point number in Q15 format.
Example	The following sample code obtains the $\sin(0.25\times\pi)$=0.707

Input

The sample input $0.25\times\pi$ in normalized Q15 format

$=\left(\frac{0.25\times\pi}{\pi}\right)\times 2^{15}$ =2000h

Output

The output is 0.707 in Q15 format = $0.707 * 2^{15}$ =23170 (5A82h)

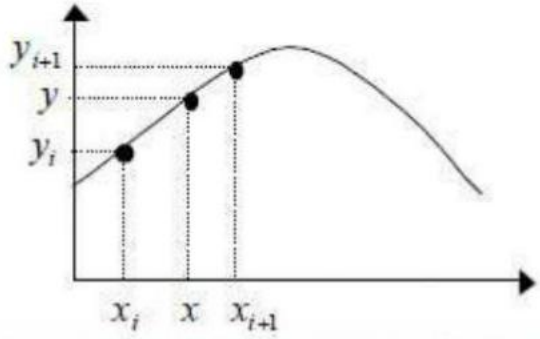
```
#include<qmath.h> /* Header file for fixed point math routine */
void main(void )
{
int x,y;
x=0x2000; /* 0.25=2000h */
y=qsin(x); /* 'y' will have sin(0.25*p)=0.707 in Q15 */
} /* 0.707 in Q15 format = 0.707 *215 =23170 (5A82h) */
```

Computation

The SIN computation is performed using table interpolation technique that is comprised of two steps viz.,

1. Direct look-up: Looking through the table to find the interval $[x_i, x_{i+1}]$ at which the considered angle 'x' is located, with $x_i < x < x_{i+1}$.
2. Interpolation: Solving the equation given below to obtain 'y'

$$y = y_i + \frac{x - x_i}{x_{i+1} - x_i} \times (y_{i+1} - y_i) \qquad (1)$$



C/C-Callable ASM Interface

Declaration	int qsinlt(int x)
Input Format	The argument 'x' is a fixed-point number in Q15 format that contains the angle in radians between $[-\pi,+\pi]$ normalized between $[-1,+1]$
Output Format	This function returns the sine of the input argument as fixed-point number in Q15 format.
Example	The following sample code obtains the $\sin(0.25\times\pi)$=0.707

Input

The sample input $0.25\times\pi$ in normalized Q15 format

$=\left(\frac{0.25\times\pi}{\pi}\right)\times 2^{15}$=2000h

Output

The output is 0.707 in Q15 format = 0.707 *2^{15} =23170 (5A82h)

QSINLT

```
#include<qmath.h> /* Header file for fixed point math routine */
void main(void )
{
int x, y;
x=0x2000; /* 0.25 ´ p in normalized Q15 format = 215=2000h */
y=qsinlt(x); /* 'y' will have sin(0.25*p)=0.707 in Q15 */
} /* 0.707 in Q15 format = 0.707 *215 =23170 (5A82h) */
```

标么化系统

$$\text{标么值} = \frac{\text{实际值}}{\text{基准值}}$$

电机理论分析和设计计算中，常用标么值来表示电机中各物理量的大小。如电压、电流、功率或容量、转矩、转速、时间或频率、磁链和反电势及电阻、电感等。从电机角度看，对不同容量的电机其参数和性能的实际值差别很大，但标么值却在一定范围变化，具有可比性；从计算的角度看，原来不同的物理量在数值上差别很大，可能达到几个数量级，但标么值可使不同的物理量在数值上等同起来，简化了计算。利用这一方法，可为不同额定值的电机的数字控制器的设计带来了便利。不仅可简化算法，提高运算精度，还为控制软件的模块化和通用性带来了方便。尤其是可将不同容量电机参数的定标加一统一。

DSC中都可能对电压、电流进行控制。首先要确定基值。就选取系统中该变量绝对值的最大值作为基值。这样，标么化后的变量，其范围在-1~+1之间。这个范围刚好可用Q15来表示。

$$I_{bmax} = 50A$$

例 电流最大值为50A，则取电流基值 $i_a = 31.1A$

按Q15定标可得：

$$I_a = \frac{i_a}{I_{bmax}} = 0.624$$

$$I_A = I_a \times (2^{15} - 1) = 0.624 \times 32767 = \$4FDE = 20446$$

是因为16位有符号数的最大值是$7FFF。而标么值的1与定标的32767相对应。

如电压信号经信号调理电路转换为0~3.3V的电压信号，供DSP的ADC进行模/数转换。首先，对信号调理电路进行设计，使传感器的输入信号有正、负值。

例如 输入信号为-10A~+10A的交流信号，通过信号调理电路转换为0~3.3V的电压信号。当 0A时，输出电压为1.65V，同样，-10A时，电压为0V；10A时为3.3V，有时2.5V.所以信号调理电路的转换系数为

$$k = 0.165$$

$$I_{base} = 10A$$

$$I_{in} = \frac{1.5}{I_{base}} \times (2^{12} - 1) = \frac{1.5}{10} \times 4095 = 614$$

将之左移3位，得到Q15定标的标么化电流为

$$I_{in}^{'} = I_{in} << 3 = 4912$$

则转换后的实际电流值为

$$i_{in} = I_{in}^{'} I_{base} /(2^{15} - 1) = \frac{4912 \times 10}{32767} = 1.4991A$$

误差为0.06%。

由此可见，标么值系统仅需对-1~0.9999695之间的数进行处理。能保持比较高的相对精度。采用Q15固定定标值的有符号数计算，使16位定点DSP能实现标么化控制系统的设计。标么化系统不用考虑定点DSP计算时的溢出，也不必关心定标后数据的表示精度，使软件设计更加简洁、高效和准确。另一个特点是可移植性。控制器的标么化设计能使控制器适应不同容量的电机。这是因为电机容量不同但标么值参数大体相同。还可适应不同的硬件便于软件移植。

第五节 用C++类实现软件模块化

C语言中常将函数看作是一个模块。本质上是将一些代码和数据封装在一起的实体，封装的数据包括局部变量和静态局部变量。C语言又可看作是面向过程的语言。

对象是在更高层次上将代码和数据封装在一起的实体。封装的代码可以包括若干个函数而且数据类型更加多样。而且通过继承等还能实现更高层次的抽象。所以面向对象语言非常适合编写大程序。主要针对对象、成员变量、成员函数（方法）等做简单的讲解不涉及继承、封装等。

类是一种类型，对象是类的实例。定义类时编译器并不会分配任何内存空间，定义对象才分配内存。类似于结构体与结构体变量的关系。

Class定义的成员变量和函数默认对外界隐藏，而struct对外界可见。对象就是类定义的变量。

使用C++类的优点

1函数名比变量名稳定。所有的接口使用公有的成员函数而不是公有的数据成员。代价是即便是简单的输入输出（本可以通过变量赋值来实现的）也须通过成员函数来完成。

函数调用是有额外开销的。是以降低效率来换取可维护性的一种策略。C++可通过在类的内部定义函数或用关键字inline实现内联，来减少函数开销。TI采用赋值语句来完成对象的输出输入功能。

2相对于C，C++可节省全局符号资源。

3相对于函数，对象是更抽象的概念。

4 对象是更大的容器。

6 提供了改进如构造函数、关键词private等

构造函数保证了对象的自动初始化，防止程序员遗忘；pvivate有利于信息的隐藏，防止误引用。对于函数，对象可实现重入性。

5 以数据为中心是更为合理的模块。

嵌入式系统中要完成模拟量采样输入、模拟量保护、开关量输入和保护。分别用四个函数来实现四个功能。如果要编写一个保护对象，则保护对象和输入对象并不是合理的对象，原因是保护对象中的两个保护函数处理的大量数据都来自于输入对象，大量数据往往要通过破坏模块化的全局变量（或全局指针变量）来传送。所以就将模拟量采样输入和保护归为一类。这样模拟量对象和开关量对象是两个独立的模块，两个对象之间没有大量变换的数据。这是以数据为中心的设计。

C语言，数据和函数的分离理所当然，而C++以数据为中心，合理形成对象，程序更加模块化。

6 提供了改进如构造函数、关键词private等

构造函数保证了对象的自动初始化，防止程序员遗忘；pvivate有利于信息的隐藏，防止误引用。对于函数，对象可实现重入性。

如何用C语言实现对象

C语言没有关键字class,但可用结构体来实现类。

1 直接用类名来定义简化的类名


```
Typedef struct STUD  STUD;
STUD stud1;
```

更为简洁的实现方法是在定义结构体时直接用

```
Typedet struct
{
.....
        Char name[20];
        Int num;
      } STUD
STUD stud1
```

2 通过**typedef** 实现指针的简洁定义

以类名加后缀”_Handle”表示指针

Typedef STUD STUD_Handle;*

*STUD *p; STUD_Handle p;等价。*

3 通过函数名前加前缀来近似实现：：运算符，减少重名。

4 用含有函数指针的结构体实现类

Void stud_display(STUD_Handle p){}

定义一个指向函数stud_display()的函数指针display

*Void(*display)(STUD_Handle); void (*display)(void*)*

C语言没有自动初始化的构造函数机制，每次定义对象时，可以用结构体的初始化来模拟构造函数，实现对象的初始化。

STUD stud1={stud_display,1};但函数指针不能直接初始化，因为函数指针和定义的函数名不是一种类型，应加强制类型转换“(void(*)(void*))

STUD stud1={{(void(*)(void*))stud_display,1};

作为简化，初始化值用宏定义来代替，

```
#define STUD_DEFAULTS
{(void(*)(void*))stud_display,1}

STUD stud1 = STUD_DEFAULTS

STUD stud2 = STUD_DEFAULTS
```

TI的例程中， 对象中所变量都作了初始化；防止漏写，所有类的定义都是变量在前，函数指针在后。必须初始化所有的变量后才能对必须有所指的函数指针做初始化。

别忘了函数声明

STUD stud1 = STUD_DEFAULTS

初始化中用到了**stud_display**，所以头文件中必须有函数声明。

Void stud_display(STUD_Handle)

稍有区别的引用对象形式

C++中定义对象并调用成员函数的方式是

STUD stud1;

Stud1.display();

C语言中则是

STUD stud1=STUD_DEFAULTS;

Stud1.display(&stud1);

对象在初始化时，函数指针变量display指向了函数stud_display,可将stud_display看作是类的成员函数。C语言必须要传递结构变量的地址&stud1，才能实现调用成员函数。

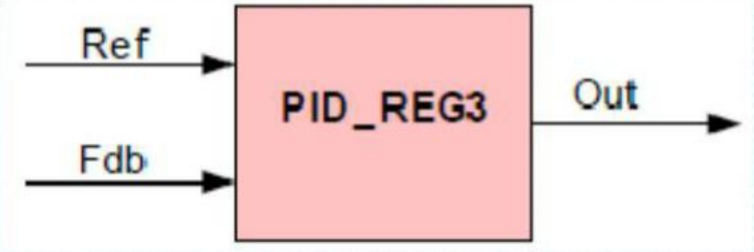
主要用于软件模块的编写、封装。以PID为例。

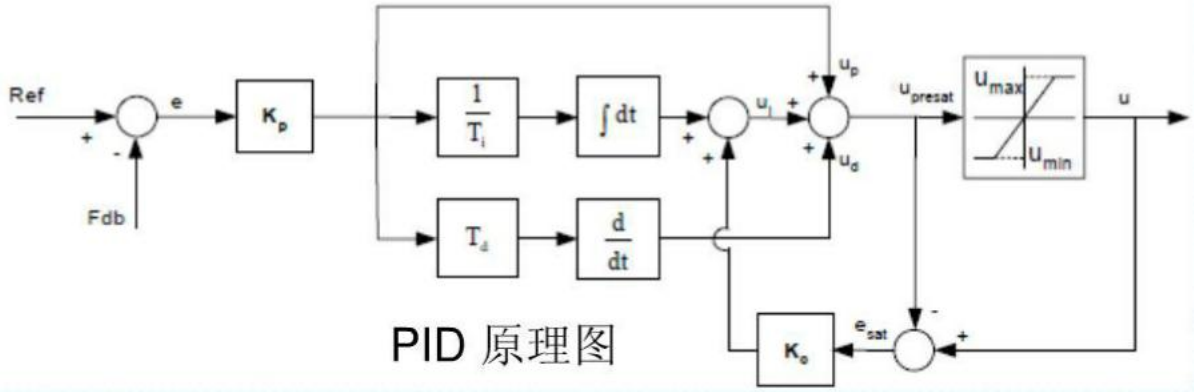
简单说来，PID控制器各校正环节的作用如下：

(1)比例环节。即时成比例地反映控制系统的偏差信号‘(0，偏差一旦产生，控制器立即产生控制作用，以减少偏差。

(2)积分环节。主要用于消除静差，提高系统的无差度。积分作用的强弱取决于积分时间常数rj，21越大，积分作用越弱，反之则越强。

(3)微分环节。能反映偏差信号的变化趋势(变化速率)，并能在偏差信号值变得太大之前，在系统中引入一个有效的早期修正传号，从而加快系统的动作速度，减小调节时间。





PID 原理图

$$u_{presat}(t)=u_p(t)+u_i(t)+u_d(t)$$

$$u_p(t)=K_p e(t)$$

$$u_i(t)=\frac{K_p}{T_i}\int_0^t e(\varsigma)d\varsigma+K_c(u(t)-u_{presat}(t))$$

$$u_d(t)=K_p T_d\frac{de(t)}{dt}$$

预饱和输出 $u_{presat}(k)=u_p(k)+u_i(k)+u_d(k)$

$$u_i(k)=u_i(k-1)+\frac{K_pT}{T_i}e(k)+K_c(u(k)-u_{presat}(k))$$

$$u_p(k)=K_pe(k)$$

$$u_d(k)=K_p\frac{T_d}{T}(e(k)-e(k-1))$$

$$K_i=\frac{T}{T_i}\qquad \mathrm{K_d}=\frac{T_d}{T}$$

$$u_i(k)=u_i(k-1)+K_ie(k)+K_c(u(k)-u_{presat}(k))$$

$$u_d(k)=K_d(e(k)-e(k-1))$$

方程变量与程序变量对应关系

	Equation Variables	Program Variables
Inputs	Ref	Ref
	Fdb	Fdb
Output	$u(k)$	Out
Others	$e(k)$	Err
	$u_p(k)$	Up
	$u_p(k-1)$	Up1
	$u_i(k)$	Ui
	$u_d(k)$	Ud
	$u_{presat}(k)$	OutPreSat
	$e_{sat}(k)$	SatErr
	K_p	Kp
	K_i	Ki
	K_d	Kd
	K_c	Kc

PID模块变量及其数据格式

韶关学院 SHAOGUAN UNIVERSITY

本模块实现32位数字带有抗饱和积分的PID控制器。也可用作PI或PD控制器。控制器采用反步接近法将微分方程变成一个差分方程。

模块变量/功能

Item	Name	Description	Format	Range(Hex)
Input	Ref	Reference input	GLOBAL_Q	80000000-7FFFFFFF
	Fdb	Feedback input	GLOBAL_Q	80000000-7FFFFFFF
	OutMax	Maximum PID32 module output	GLOBAL_Q	80000000-7FFFFFFF
	OutMin	Minimum PID32 module output	GLOBAL_Q	80000000-7FFFFFFF
Output	Out	PID Output (Saturated)	GLOBAL_Q	80000000-7FFFFFFF
PID parameter	Kp	Proportional gain	GLOBAL_Q	80000000-7FFFFFFF
	Ki	Integral gain	GLOBAL_Q	80000000-7FFFFFFF
	Kd	Derivative gain	GLOBAL_Q	80000000-7FFFFFFF
	Kc	Integral correction gain	GLOBAL_Q	80000000-7FFFFFFF
Internal	Err	Error=Reference-feedback	GLOBAL_Q	80000000-7FFFFFFF
	SatErr	SatErr=output-preSatOut	GLOBAL_Q	80000000-7FFFFFFF
	Up	Proportional output	GLOBAL_Q	80000000-7FFFFFFF
	Up1	Previous proportional output	GLOBAL_Q	80000000-7FFFFFFF
	Ui	Integral output	GLOBAL_Q	80000000-7FFFFFFF
	Ud	Differential output	GLOBAL_Q	80000000-7FFFFFFF
	OutPreSat	PID output before saturation	GLOBAL_Q	80000000-7FFFFFFF

```
//文件名    PID_REG3.C  (IQ version)
//功能说明：带anti-windup 的 PID 控制器
#include "IQmathLib.h"        // Include header for IQmath library
// Don't forget to set a proper GLOBAL_Q in "IQmathLib.h" file
#include "dmctype.h"
#include "pid_reg3.h"
```

专用常数和数据类型

PIDREG3 模块定义创建了一个数字类型。便于建立一个PID模块类的实例。为创建模块的多个实例，只需简单地声明为PIDRGE3即可。

PIDREG3_handle 指向PID_REG3模块的指针。

方法 void pid_reg3_calc(PIDREG3_handle)

这个函数用反步接近技术实现了PID数字控制器（IQ格式）。

文件名： PID_REG3.H (IQ version)

描述：该头文件包含常数、数据类型定义和PIDREG3函数原型

韶关学院 SHAOGUAN UNIVERSITY

```
typeset struct {
_iq  Ref;   // 输入：参考输入 Input: Reference input
_iq  Fdb;   // 输入：反馈输入 Input: Feedback input
_iq  Err;   // 变量：错误 Variable: Error
_iq  Kp;// 参数：比例增益 Parameter: Proportional gain
_iq  Up;   // 变量：比例输出 Variable: Proportional output
_iq  Up;   // 变量：比例输出 Variable: Proportional output
_iq  Ui;   // 变量：积分输出 Variable: Integral output
_iq  Ud;   // 变量：微分输出 Variable: Derivative output
_iq  OutPreSat; // 变量：饱和输出 Variable: Pre-saturated output
_iq  OutMax;    // 参数：最大输出 Parameter: Maximum output
_iq  OutMin;                  // 参数：最小输出 Parameter: Minimum output
_iq  Out;                  // 输出：PID输出 Output: PID output
_iq  SatErr;               // 变量：饱和差值 Variable: Saturated difference
_iq  Ki;    // 参数：积分增益 Parameter: Integral gain
_iq  Kc;      // 参数：积分修正增益 Parameter: Integral correction gain
_iq  Kd;      // 参数：微分增益Parameter: Derivative gain
_iq  Up1;    // 历史：先前的比例输出 History: Previous proportional output
void  (*calc)(); // 计算函数指针 Pointer to calculation function
```

```
void pid_reg3_calc(PIDREG3 *v)
{
    // 计算误差
    v->Err = v->Ref - v->Fdb;
    // 计算比例输出
    v->Up = _IQmpy(v->Kp,v->Err);
    // 计算积分输出
    //v->Ui = v->Ui + _IQmpy(v->Ki,v->Up) + _IQmpy(v->Kc,v->SatErr);
    v->Ui = v->Ui + _IQmpy(v->Ki,v->Err) + _IQmpy(v->Kc,v->SatErr);
    // 计算微分输出
    v->Ud = _IQmpy(v->Kd,(v->Up - v->Up1));
    // 计算预饱和输出
    v->OutPreSat = v->Up + v->Ui + v->Ud;
    // 饱和输出
```

```
if (v->OutPreSat > v->OutMax)
    v->Out =  v->OutMax;
  else if (v->OutPreSat < v->OutMin)
    v->Out =  v->OutMin;
  else
    v->Out = v->OutPreSat;
  // 计算饱和差
  v->SatErr = v->Out - v->OutPreSat;
  // 上载先前的比例输出
  v->Up1 = v->Up;
}
```

模块应用

实例化 **PID**对象的实例

```
PIDREG3 pid1, pid2;
```

初始化 对象的初始化

```
PIDREG3 pid1 = PIDREG3_DEFAULTS;
PIDREG3 pid2 = PIDREG3_DEFAULTS;
```

计算函数

```
pid1.calc(&pid1);
pid2.calc(&pid2);
```

PID模块使用范例

韶关学院 SHAOGUAN UNIVERSITY

```
/* Instance the PID_REG3 module */
PIDREG3 pid1=PIDREG3_DEFAULTS;
PIDREG3 pid2=PIDREG3_DEFAULTS;
main()
{
pid1.Kp = _IQ(0.5); // Pass _iq parameters to pid1
pid1.Ki = _IQ(0.001); // Pass _iq parameters to pid1
pid1.Kd = _IQ(0.01); // Pass _iq parameters to pid1
pid1.Kc = _IQ(0.9); // Pass _iq parameters to pid1
pid2.Kp = _IQ(0.8); // Pass _iq parameters to pid2
pid2.Ki = _IQ(0.0001); // Pass _iq parameters to pid2
pid2.Kd = _IQ(0.02); // Pass _iq parameters to pid2
pid2.Kc = _IQ(0.8); // Pass _iq parameters to pid2
```

```
void interrupt periodic_interrupt_isr()
{
pid1.Ref = input1_1; // Pass _iq inputs to pid1
pid1.Fdb = input1_2; // Pass _iq inputs to pid1
pid2.Ref = input2_1; // Pass _iq inputs to pid2
pid2.Fdb = input2_2; // Pass _iq inputs to pid2
pid1.calc(&pid1); // Call compute function for pid1
pid2.calc(&pid2); // Call compute function for pid2
output1 = pid1.Out; // Access the output of pid1
output2 = pid2.Out; // Access the output of pid2
}
```

Module.h

```
typedef struct module_data {
int in, pastIn ,result; // Persistent data, coefficients etc.
void (*calc)() // Pointer to calculation function
} MODULE;
#define MODULE_DEFAULTS { 0,0,0, (void (*)(Uint32))module_calc
```

Module.c

```
void module_calc(MODULE *p)
{
// compute code
p->result = p->pastIn+p->In;
p->pastIn = p->In;
}
```

```
Client.c
MODULE mod = MODULE_DEFAULTS;
void somefunc(void)
{
int foo;
mod.in = 10; // coefficient configuration
mod.calc(&mod);
foo = mod.result; // Use result
}
```