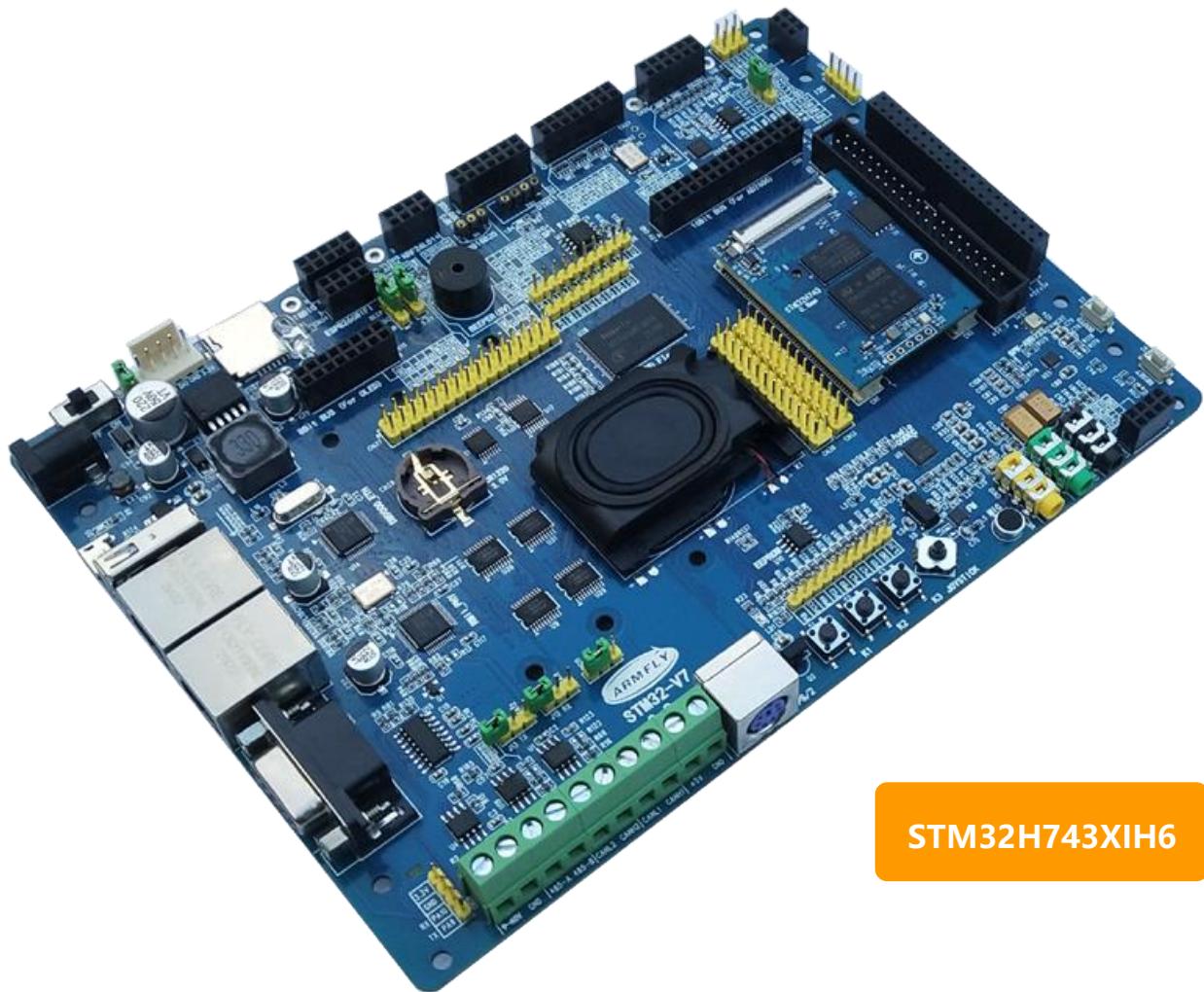


STM32-V7 开发板

数字信号处理



STM32H743XIH6

武汉安富莱电子有限公司



本文档适用的开发板：

序号	型号配置	说明
1	STM32-V7+4.3寸电阻屏	主板+4.3寸RGB屏 (480*320/电阻触摸) 有亚克力保护
2	STM32-V7+4.3寸电容屏	主板+4.3寸RGB屏 (480*272/电容触摸) 有亚克力保护
3	STM32-V7+5.0寸电阻屏	主板+5.0寸RGB屏 (800*480/电阻触摸) 有亚克力保护
4	STM32-V7+5.0寸电容屏	主板+5.0寸RGB屏 (800*480/电容触摸) 有亚克力保护
5	STM32-V7+7.0寸电阻屏	主板+7.0寸RGB屏 (800*480/电阻触摸) 有亚克力保护
6	STM32-V7+7.0寸电容屏	主板+7.0寸RGB屏 (800*480/电容触摸) 有亚克力保护
7	STM32-V7主板	仅主板 无显示屏，无亚克力保护

- (1) 开发板配套的显示模块可以单独购买。
(2) 所有配置中的主板和光盘都是相同的，差异仅在于显示模块和亚克力板。

[点击此处直接购买](#)

销售QQ: 1295744630

销售旺旺: armfly

微信公众号: 安富莱电子

销售电话: 13638617262

邮箱: armfly@qq.com

公司网址: www.armfly.com

技术支持论坛: www.armbbs.cn

淘宝直销: armfly.taobao.com



武汉安富莱电子有限公司 专业开发板、显示模块制造商 承接项目开发 (提供生产供货服务)



第1章 初学数字信号处理准备工作

本期教程开始带领大家学习 DSP 教程，学习前首先要搞明白一个概念，DSP 有两层含义，一个是 DSP 芯片也就是 Digital Signal Processor，另一个是 Digital Signal Processing，也就是我们常说的数字信号处理技术。本教程主要讲的是后者。

1.1 初学者重要提示

1.2 STM32H7 的 DSP 功能介绍

1.3 Cortex-M7 内核的 DSP 和专业 DSP 的区别

1.4 ARM 提供的 CMSIS-DSP 库

1.5 TI 提供的 32 位定点 DSP 库 IQmath

1.6 ARM DSP 软件替代模拟器件的优势

1.7 Matlab 的安装

1.8 总结

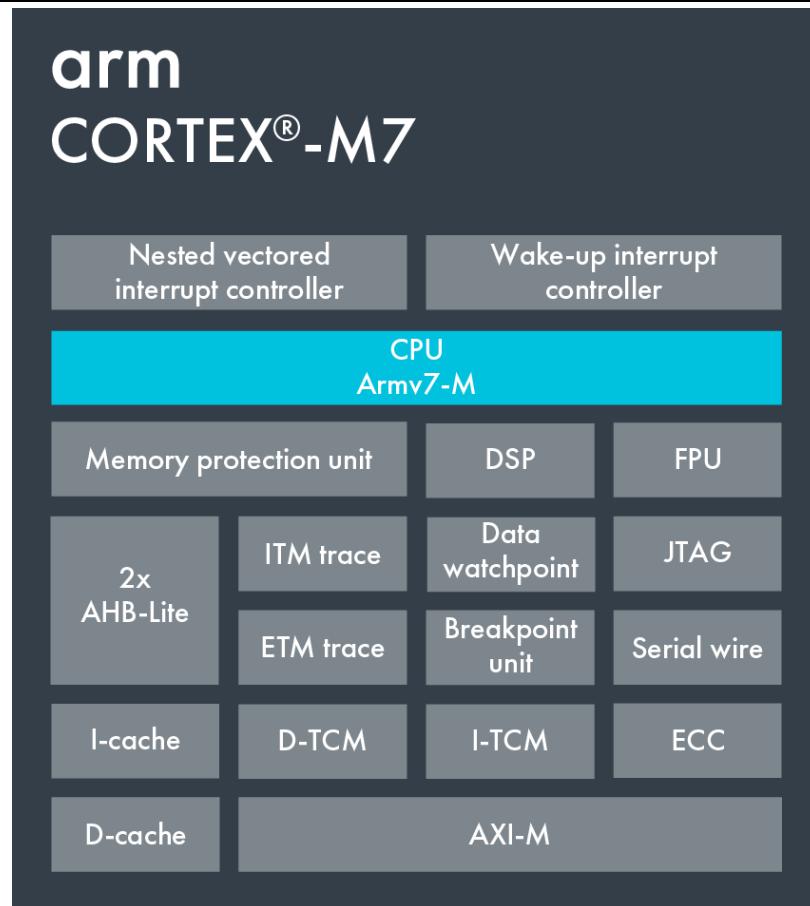
1.1 初学者重要提示

- ◆ 关于学习方法问题，可以看附件章节 A。
- ◆ 这几年单片机的性能越来越强劲，DSP 芯片的中低端应用基本都可以用单片机来做。
- ◆ 当前单片机 AI 也是有一定前景的，ARM 一直在大力推进，很多软件厂商和研究机构也在不断的努力。通过此贴可以了解下：单片机 AI 的春天真的来了，ARM 最新 DSP 库已经支持 NEON，且支持 Python

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=94406>。

1.2 STM32H7 的 DSP 功能介绍

STM32H7 是采用的 Cortex-M7 内核，而 DSP 功能是内核自带的，下面我们通过 M7 内核框图来了解下：



重点看如下两个设计单元：

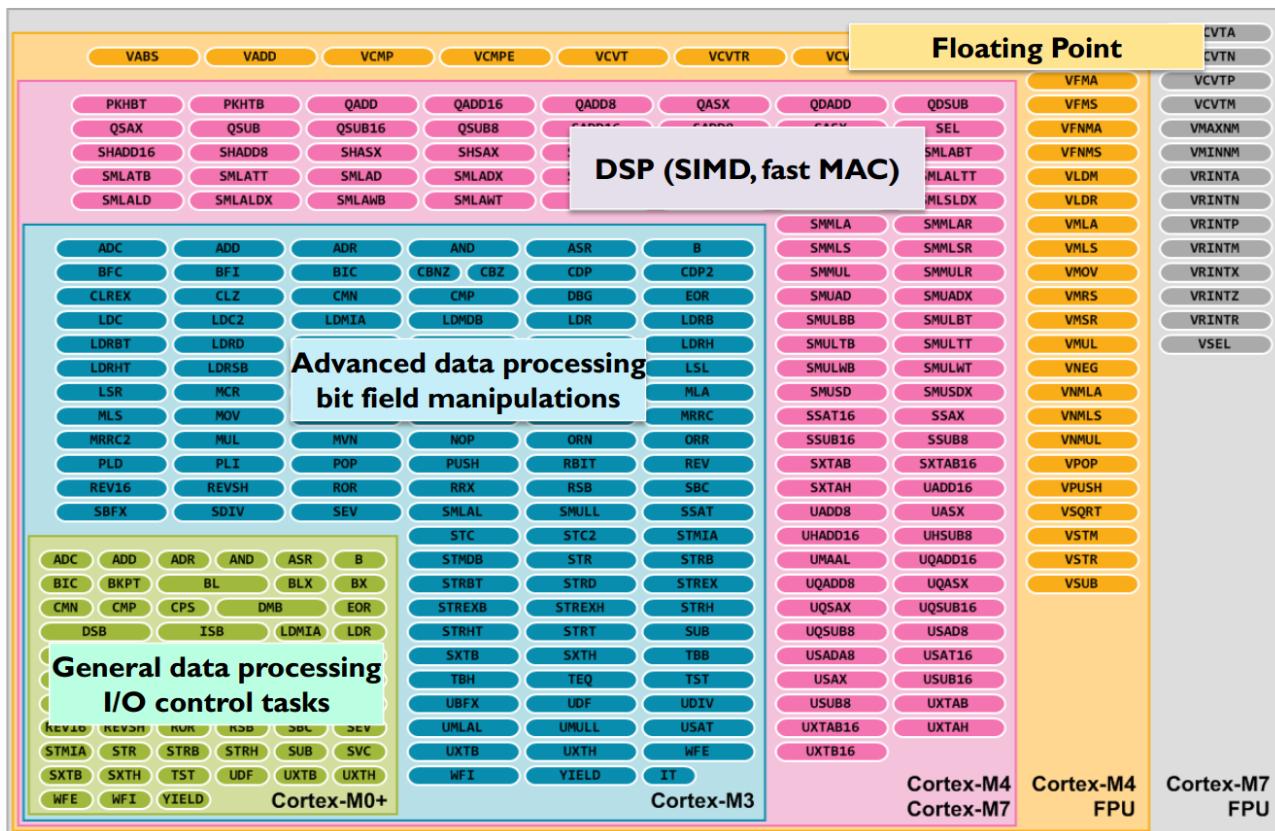
◆ DSP

DSP 单元集成了一批专用的指令集（主要是 SIMD 指令和快速 MAC 乘积累加指令），可以加速数字信号处理的执行速度。

◆ FPU

Cortex-M7 内核支持双精度浮点，可以大大加速浮点运算的处理速度。

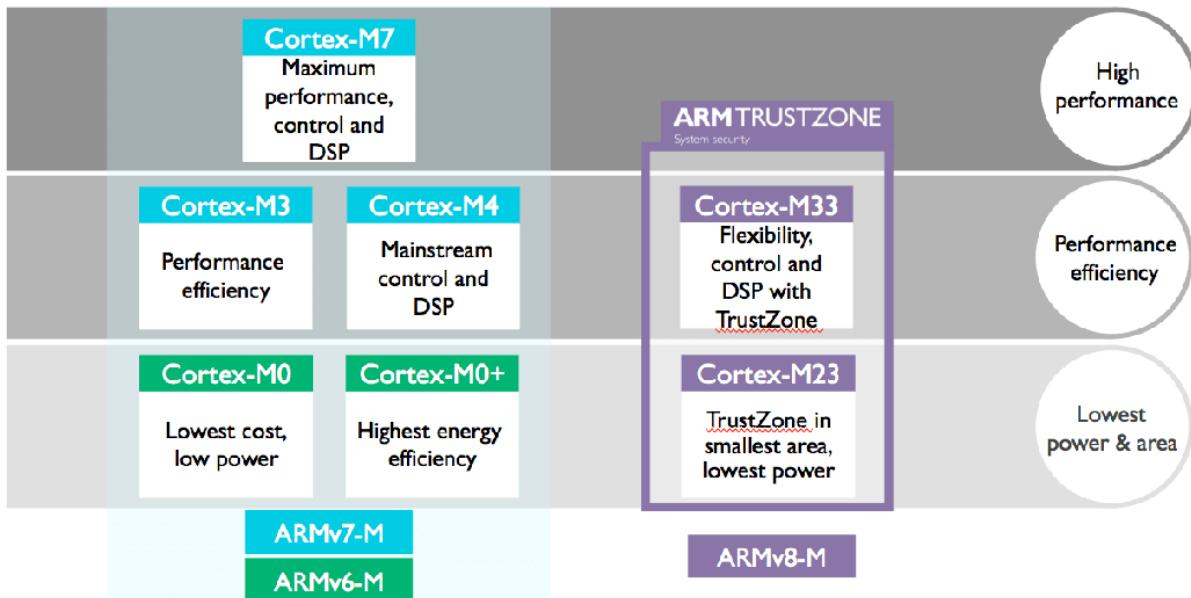
下面是 Cortex-M3, M4 和 M7 的指令集爆炸图：



通过这个图，我们可以了解到以下几点：

- ◆ M4 和 M7 系列有相同的 DSP 指令集。
- ◆ M7 相比 M4 系列要多一些浮点指令集。
- ◆ 同时这里要注意一个小细节，浮点指令都是以字符 V 开头的。通过这点，我们可以方便的验证是否正确开启了 FPU (MDK 或者 IAR 调试状态查看浮点运算对应的反汇编是否有这种指令)。

不同 M 内核的 DSP 性能比较：



- ◆ Cortex-M7 内核的 DSP 性能最强。
- ◆ Cortex-M3, M4 和 M33 是中等性能, 其中 M3 最弱。
- ◆ Cortex-M0, M0+和 M23 性能最弱。

1.3 Cortex-M7 内核的 DSP 和专业 DSP 的区别

M 核的 DSP 处理单元与专业 DSP 的区别:

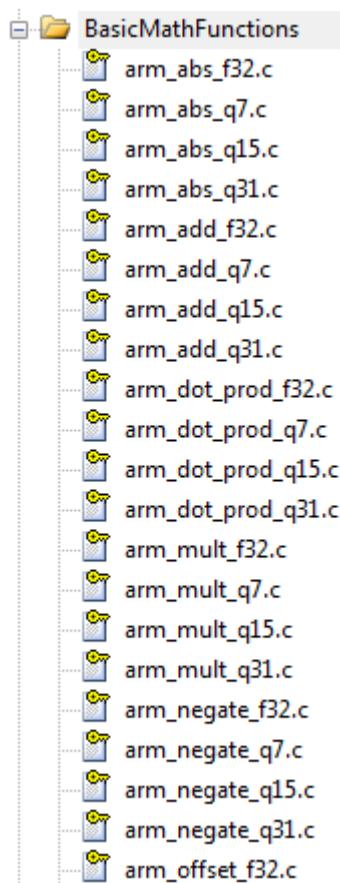
	Cortex-M3	Cortex-M4	Cortex-M7	True DSP
Single cycle MAC		Fixed-point only	Fixed and floating-point	Y
Floating-point		Y	Y	Y
Fractional and saturating math		Y	Y	Y
SIMD operations		Y	Y	Y
Load and store in parallel with math			Y (1)	Y (2)
Zero overhead loops			Y	Y
Accumulator with guard bits				Y
Circular and bit-reversed addressing				Y

1.4 ARM 提供的 CMSIS-DSP 库

为了方便用户实现 DSP 功能, ARM 专门做了一个 DSP 库 CMSIS-DSP, 主要包含以下数字信号处理算法:

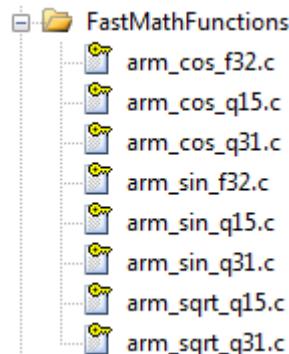
- ◆ BasicMathFunctions

提供了基本的数据运算，如加减乘除等基本运算，以_f32结尾的函数是浮点运算，以_q8, _q15, _q31结尾的函数是定点运算，下面是部分API截图：



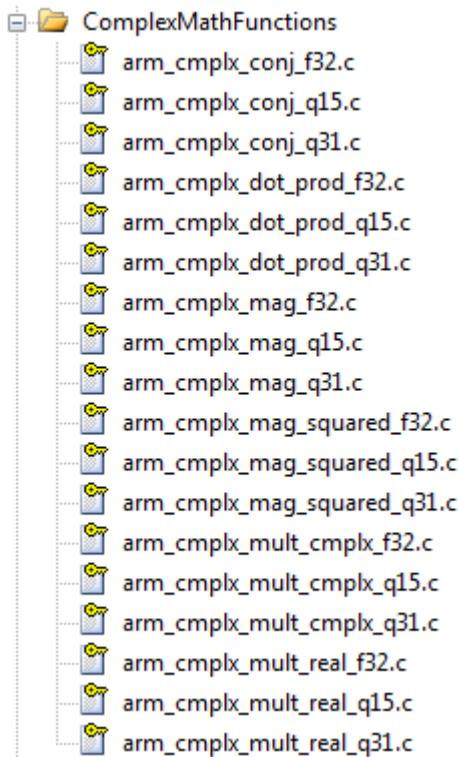
◆ FastMathFunctions

主要提供SIN, COS以及平方根SQRT的运算。



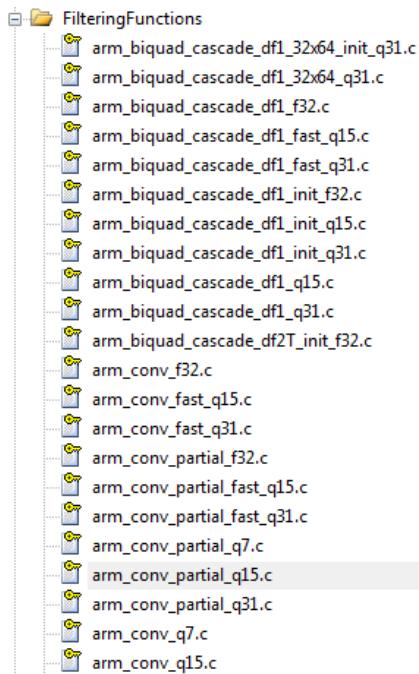
◆ ComplexMathFunctions

复杂数学运算，主要是向量，求模等运算。下面是部分API截图：



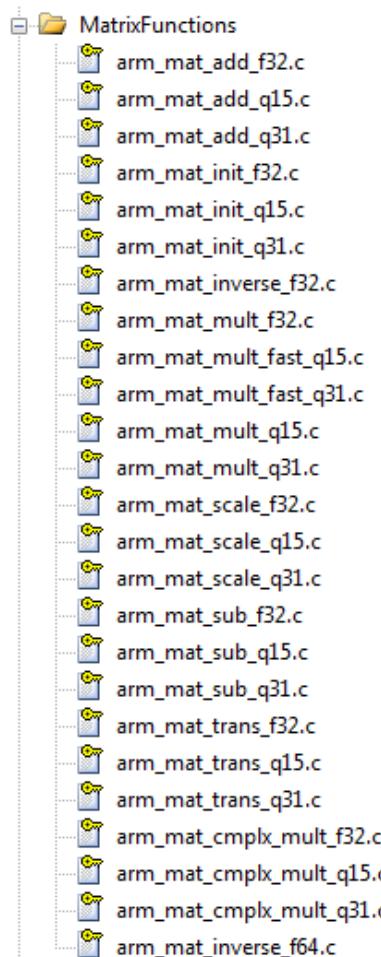
◆ FilteringFunctions

主要是滤波函数，如IIR, FIR, LMS等，下面是部分API截图：



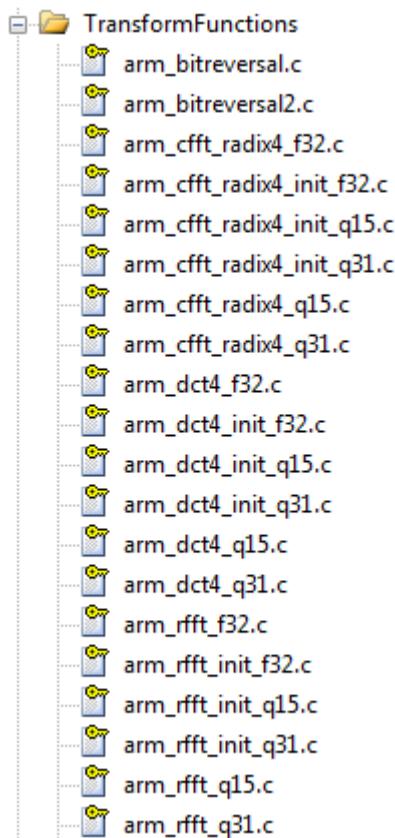
◆ MatrixFunctions

主要是矩阵运算。



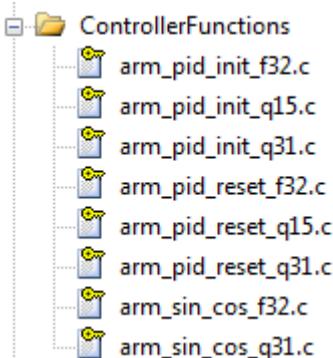
◆ TransformFunctions

变换功能。包括复数FFT (CFFT) , 复数FFT逆运算 (CIFFT) , 实数FFT (RFFT) , 实数 FFT 逆运算, 下面是部分API截图:



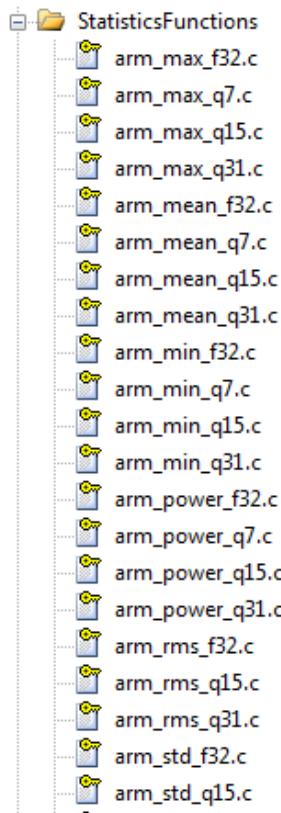
◆ ControllerFunctions

控制功能，主要是PID控制函数和正余弦函数。



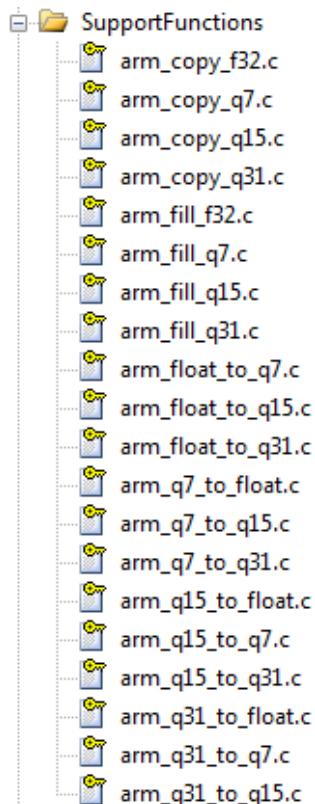
◆ StatisticsFunctions

统计功能函数，如求平均值，最大值，最小值，功率，RMS等，下面是部分API截图。



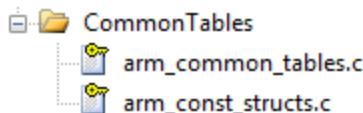
◆ SupportFunctions

支持功能函数，如数据拷贝，Q格式和浮点格式相互转换。



◆ CommonTables

arm_common_tables.c 文件提供位翻转或相关参数表。



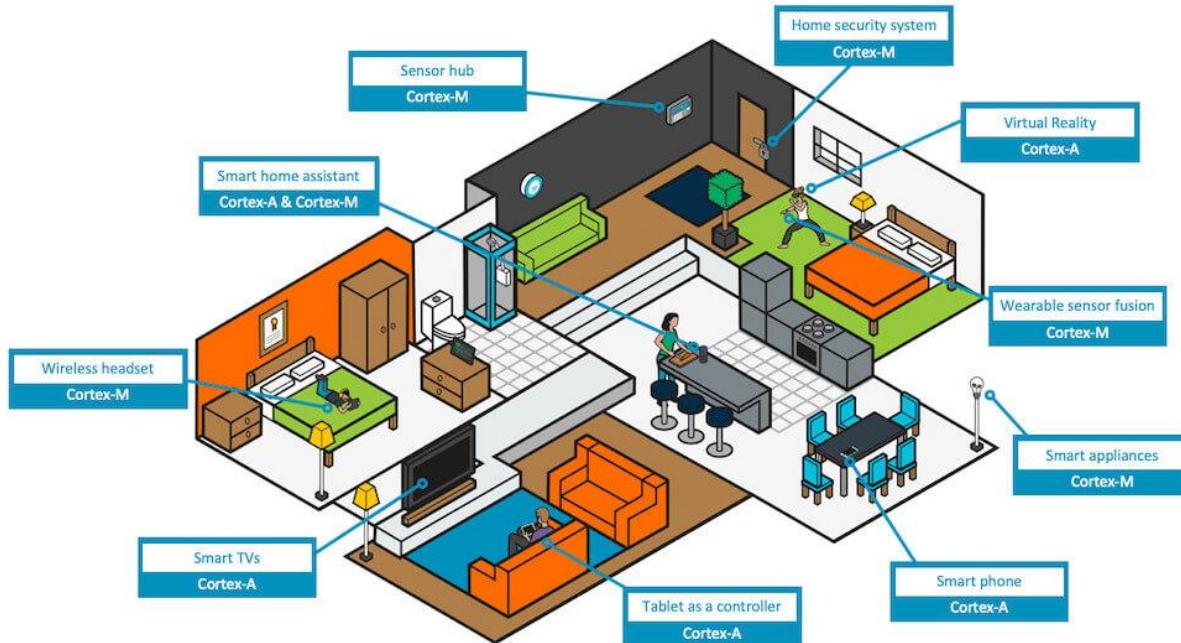
1.5 TI 提供的 32 位定点 DSP 库 IQmath

初次使用这个定点库，感觉在各种Q格式的互转、Q格式数值和浮点数的互转处理上更专业些，让人一目了然。

所以本次教程也会对IQmath做个介绍并配套一个例子。

1.6 ARM DSP 软件替代模拟器件的优势

我们日常生活中用到DSP的地方很多，以生活中的设备为例：



通过ARM DSP软件替换模拟组件可以降低成本，PCB的面积和设计时间，同时提高灵活性和适应性。

- ◆ 降低BOM成本

将模拟电路转换为软件的最明显的好处是材料清单成本 (BOM) 减少。

- ◆ 提高设计灵活性

使用模拟滤波器来不断调节电路以获得最佳性能时，这种情况并不少见。较小的电路板修改会导致新的电气特性突然改变寄生电容或电感，从而导致模拟电路达不到预期。将模拟电路转换为DSP算法不仅可以消除这种风险，还可以根据软件的需要进行调整，且更具灵活性。



◆ 减少产品尺寸

降低BOM成本具有额外的好处，也允许开发人员减少其产品的尺寸。

◆ 缩短设计周期时间

将模拟电路转换为软件有助于缩短设计周期。这有几个原因：

- 首先，有很多工具可供软件设计人员模拟和生成替换模拟电路所需的DSP算法。这通常比通过电路仿真和测试调整电路所需的时间快得多。
- 其次，如果需要进行更改，可以在软件中进行更改，这可以在几分钟内完成，而不必重新调整电路板或进行硬件修改。

◆ 现场适应性

在某些产品中，设计者很难预料用户在现场所遇到的各种情况。使用DSP算法，设计者甚至用户都可以进行实时调整，以适应现场条件，而无需进行大量硬件修改。

用数字信号处理算法替换模拟电路有很多好处。需要设计者在实际应用中权衡利益，选择最合适的方案。

1.7 Matlab 安装

Matlab 是学习 DSP 过程中非常重要的辅助工具，也是需要熟练掌握的，本教程的第 2 章到第 5 章进行了入门介绍。

1.8 总结

本期教程主要是做一些入门性的介绍，下期教程将开始实战。



第2章 Matlab R2018a 的安装

本期教程主要是讲解 Matlab R2018a 的安装过程，作为学习 DSP 的必备软件，掌握简单的 Matlab 操作是必须的。

2.1 初学者重要提示

2.2 Matlab R2018a 安装

2.3 Matlab 简介

2.4 总结

2.1 初学者重要提示

- ◆ Matlab2018a 的软件比较大，压缩包有 13GB，安装后有 20 多 GB。如果电脑速度不是很快的话，安装要花点时间，需要大家耐心等待。
- ◆ 安装前，请大家务必将安装过程通读一遍，有些地方是需要大家注意的。
- ◆ 如果想使用老版 matlab2012a，可以看我们早期的数字信号处理教程安装章节：
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=3886>。

2.2 Matlab R2018a 安装

下面将 Matlab2018a 的安装流程做个说明。

2.2.1 第 1 步，下载并解压

软件包下载地址：<http://www.armbbs.cn/forum.php?mod=viewthread&tid=94442>。软件包比较大，有 13GB，要下载一段时间。注意如下的三个文件都要下载：



R2018a_win64_dvd2.iso

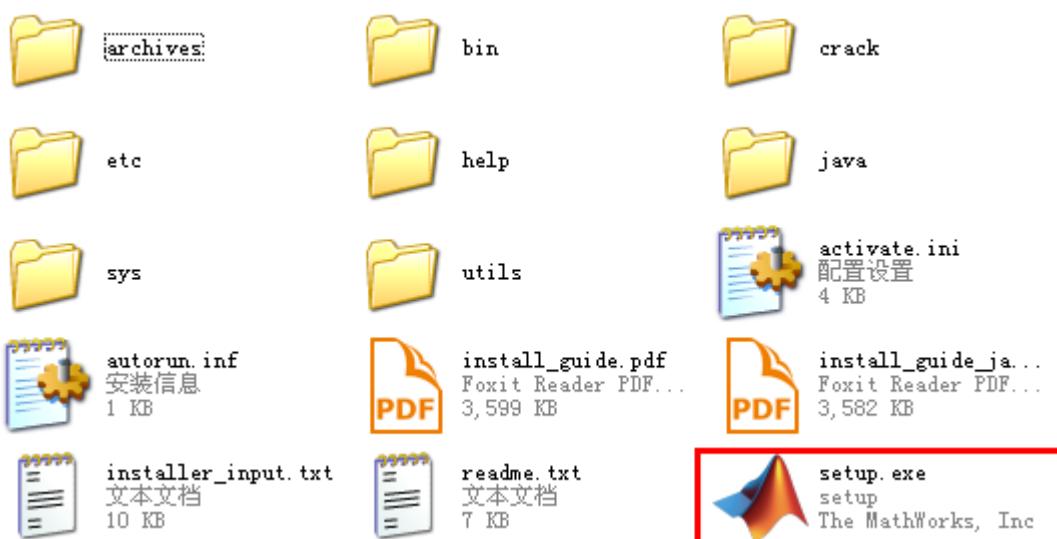


R2018a_win64_dvd1.iso



MATLAB R2018a Win64 Crack.zip

前两个是安装文件的压缩包，后面那个 Crack 是注册文件。解压时请优先解压 R2018a_win64_dvd1.iso，然后解压 R2018a_win64_dvd1.iso。



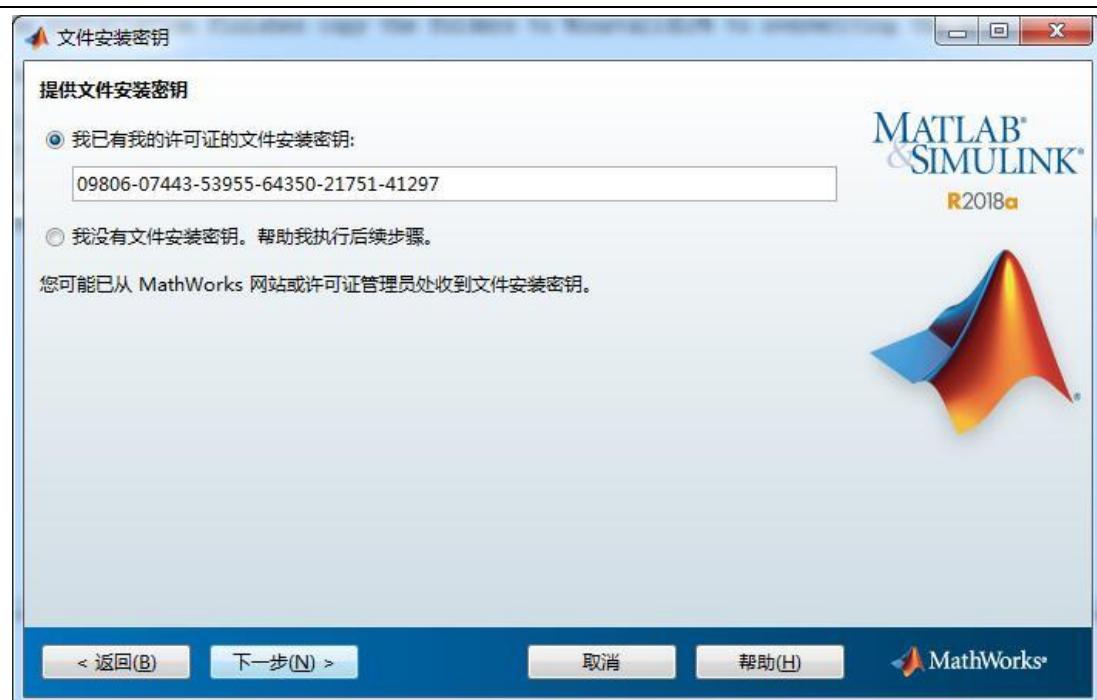
两个文件都解压出来后，点击 setup.exe 文件安装即可。

2.2.2 第 2 步，输入安装密钥

选择使用文件安装密钥。

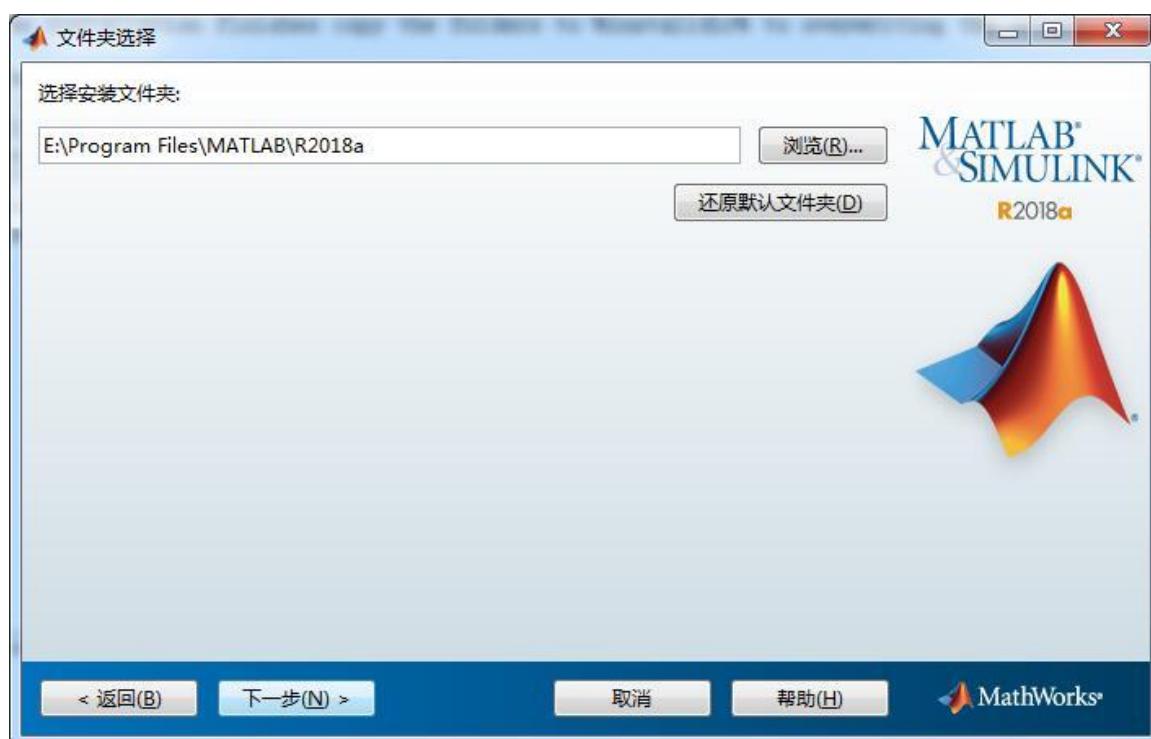


输入安全密码，输入 09806-07443-53955-64350-21751-41297 即可。

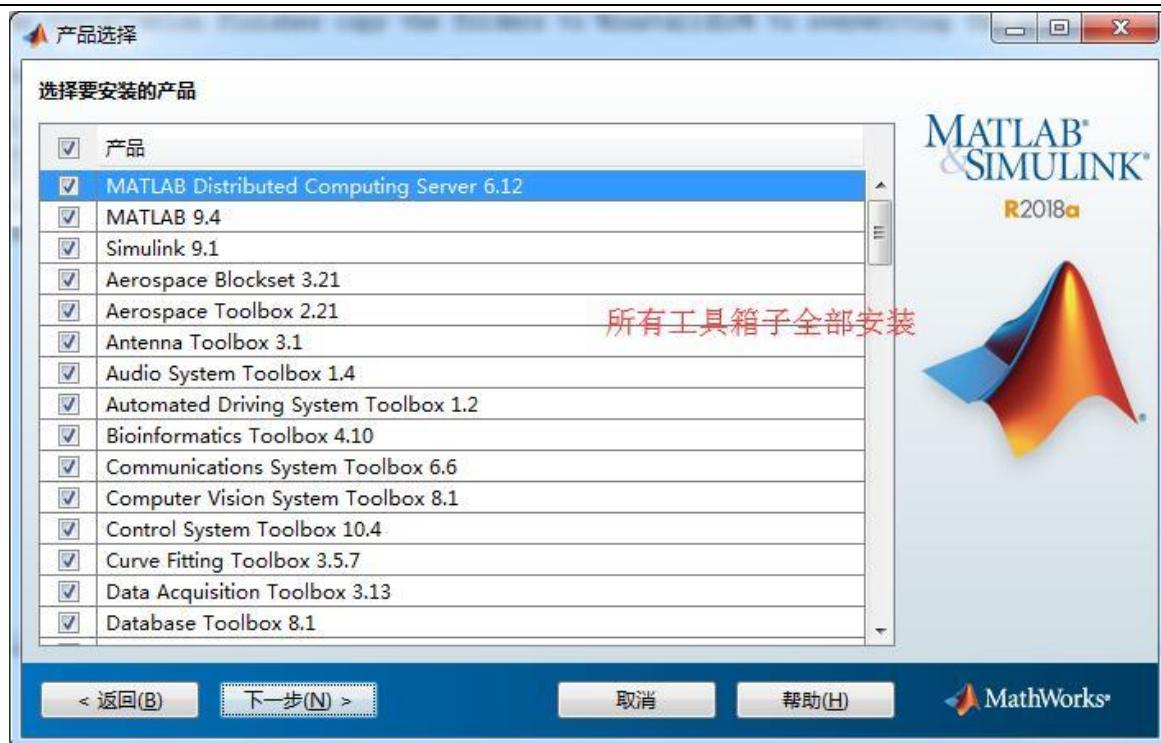


2.2.3 第3步，选择安装路径并安装所有工具箱

注意安装路径不要太长，路径不要有中文。

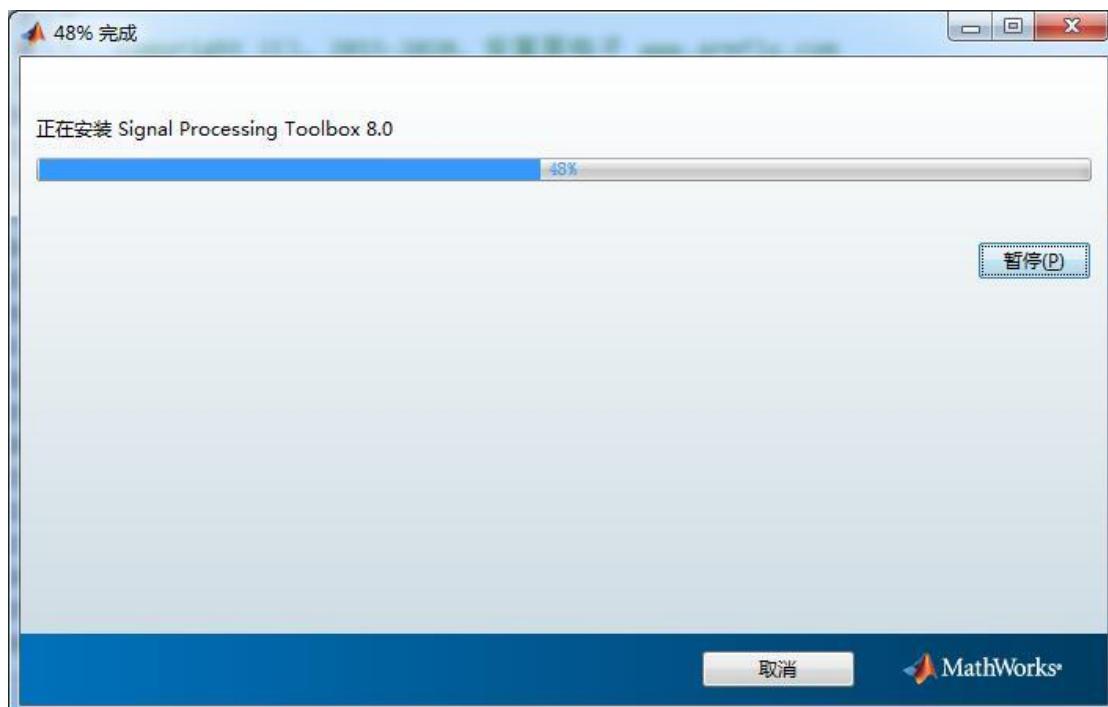


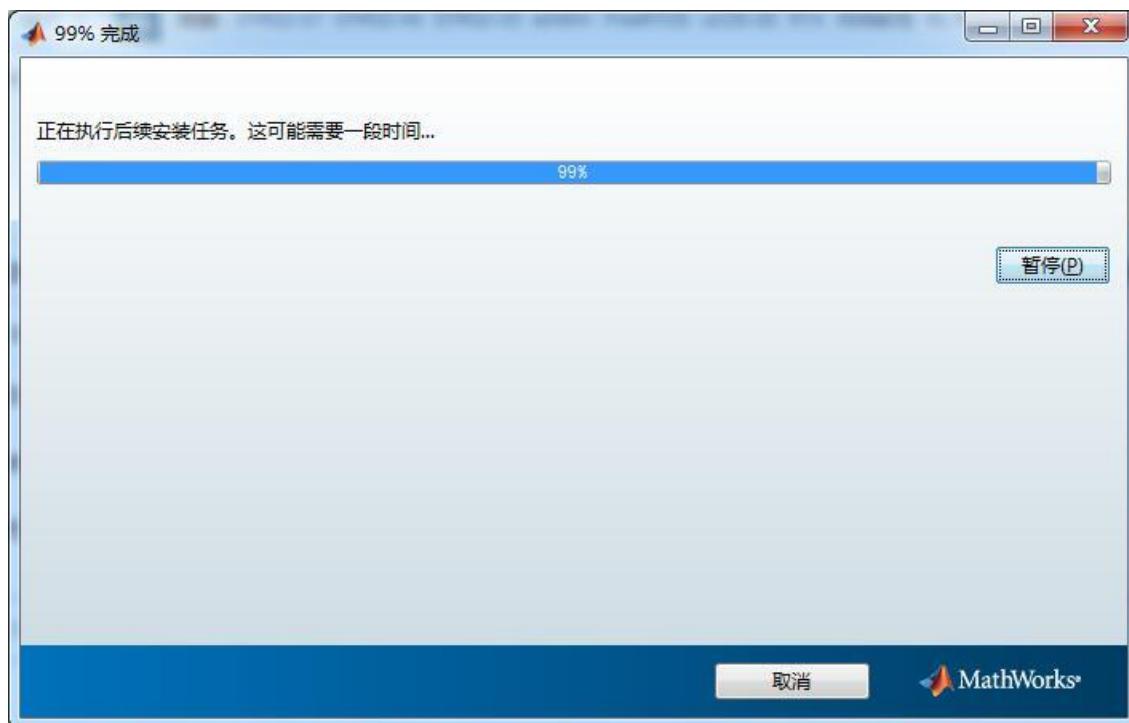
我们这里将所有的工具箱全部安装上：



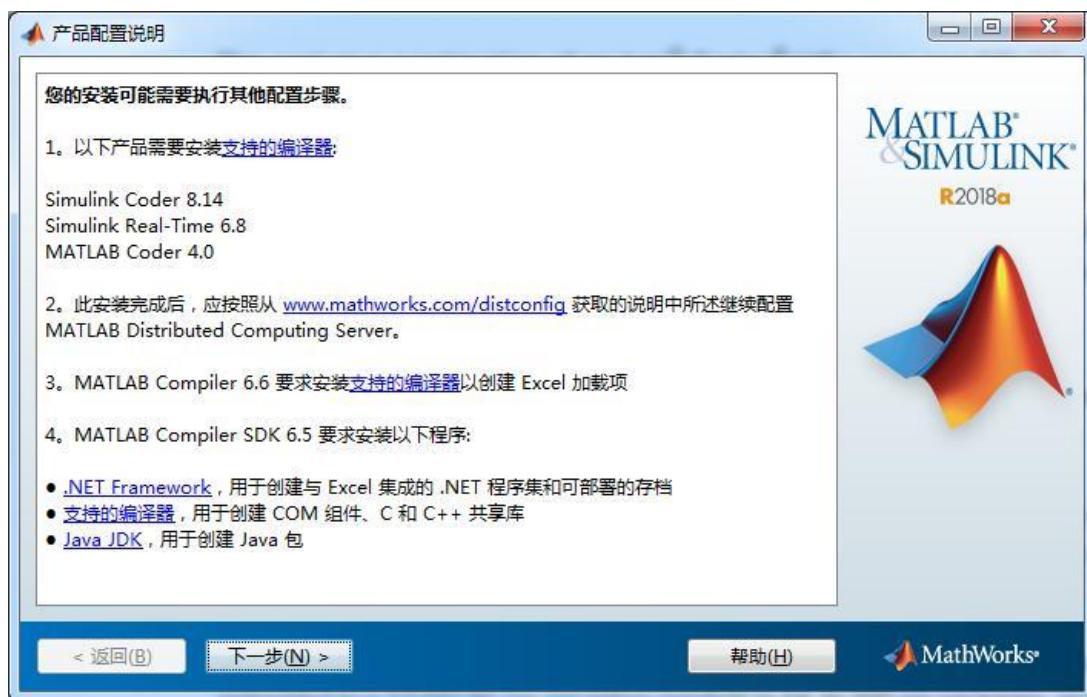
2.2.4 第4步，漫长的安装过程

接下来就是漫长的安装过程，电脑速度快的，差不多也要1个小时。





进行到下面这个界面时，就是最后一步了：



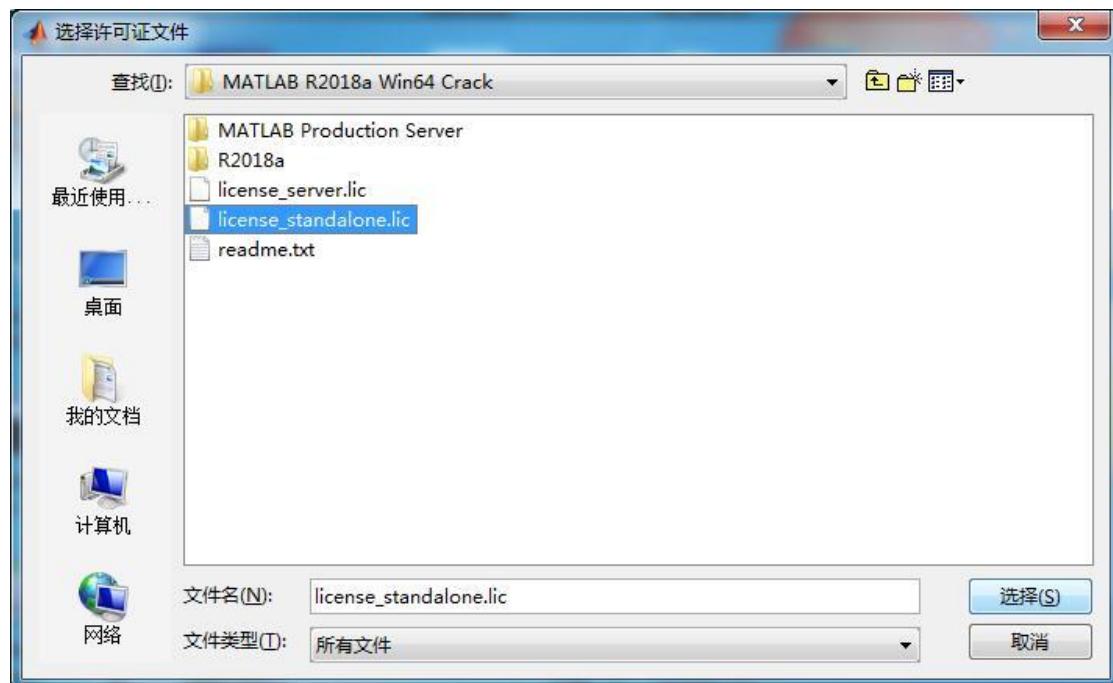
点击下一步，就安装完了。

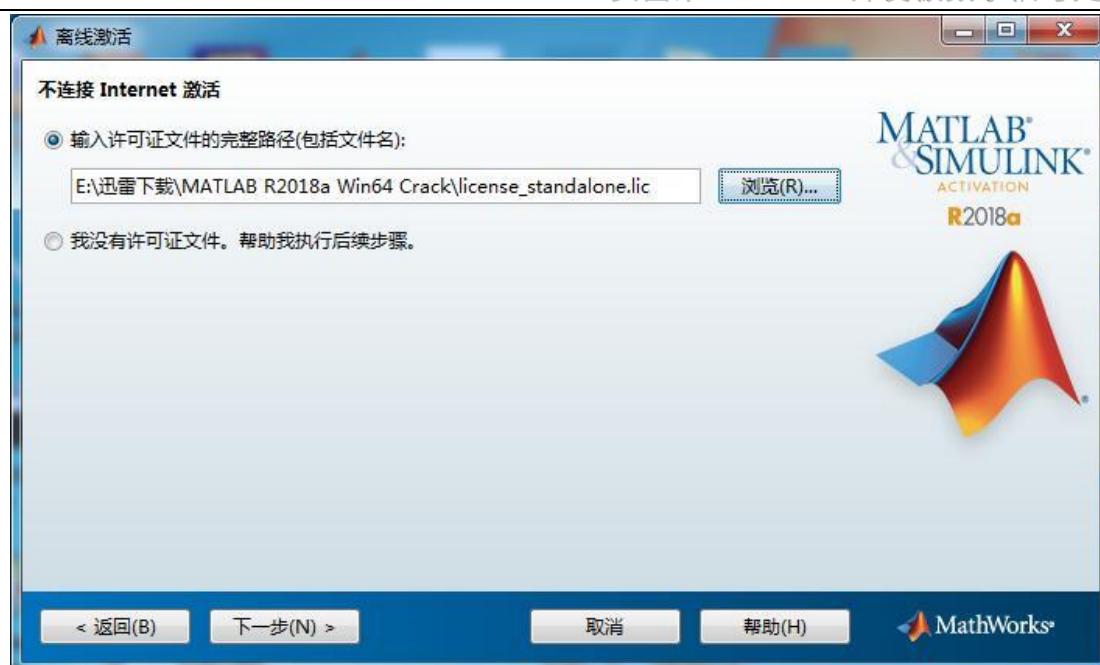
2.2.5 第 5 步，注册 matlab

下面的主要工作就是打开安装好的 matlab，如果 matlab 的图标没有出现在桌面上，需要大家到 matlab 安装目录的 bin 文件中找出启动文件 matlab.exe，首次打开会弹出如下界面：

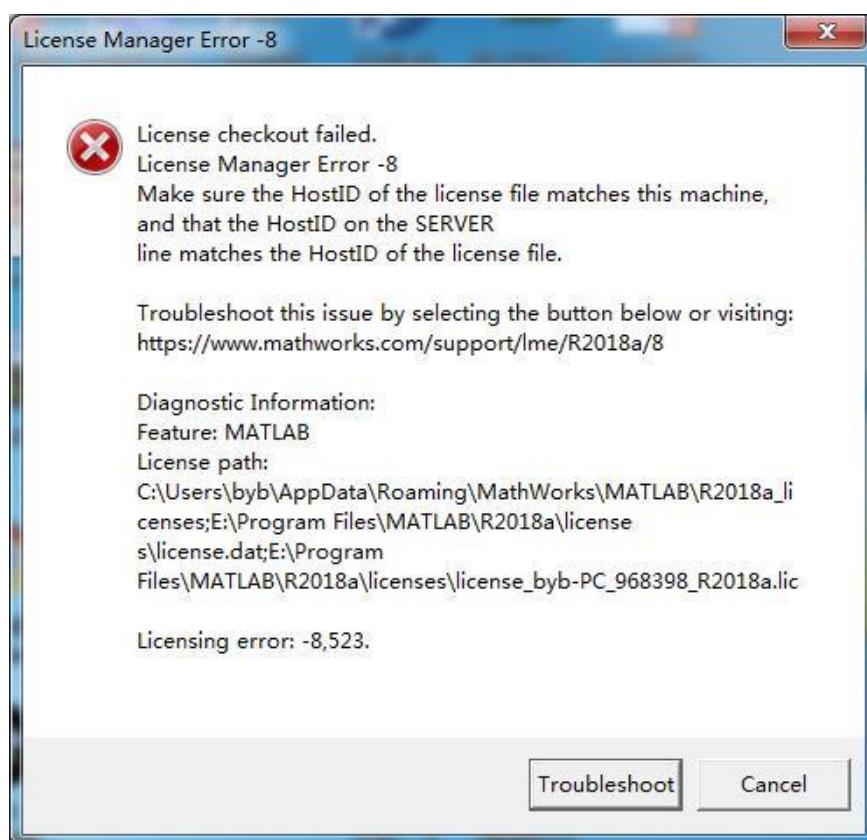


需要大家将下载的 Crack 压缩包解压下，然后点击上面截图的下一步来加载下面的 license_standalone.lic 文件。





至此就注册完成了，但是当我们再次打开 matlab.exe 文件的时候，弹出如下错误：

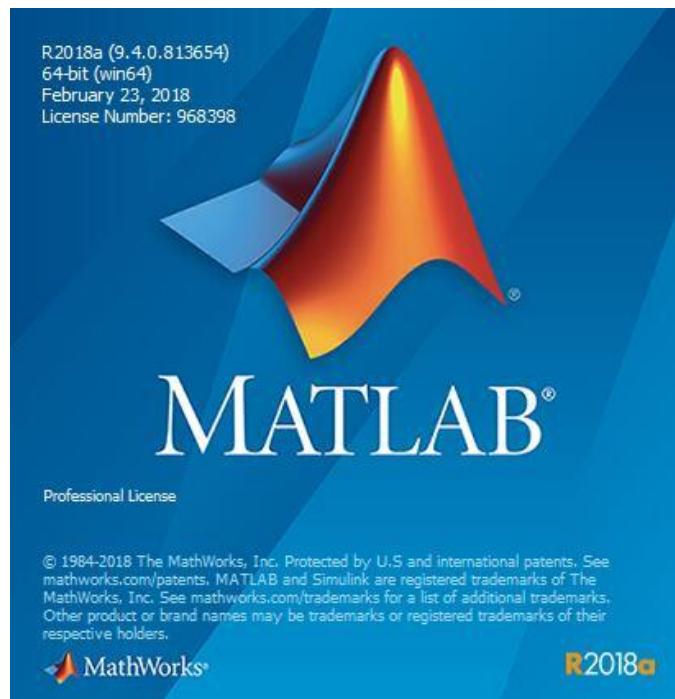


出现这个错误不要慌，解决办法也比较简单，用户只需将 Crack 文件里面 MATLAB R2018a Win64 Crack\R2018a\bin\win64\netapi32.dll 复制到 matlab 安装目录 MATLAB\R2018a\bin\win64 里面即可。

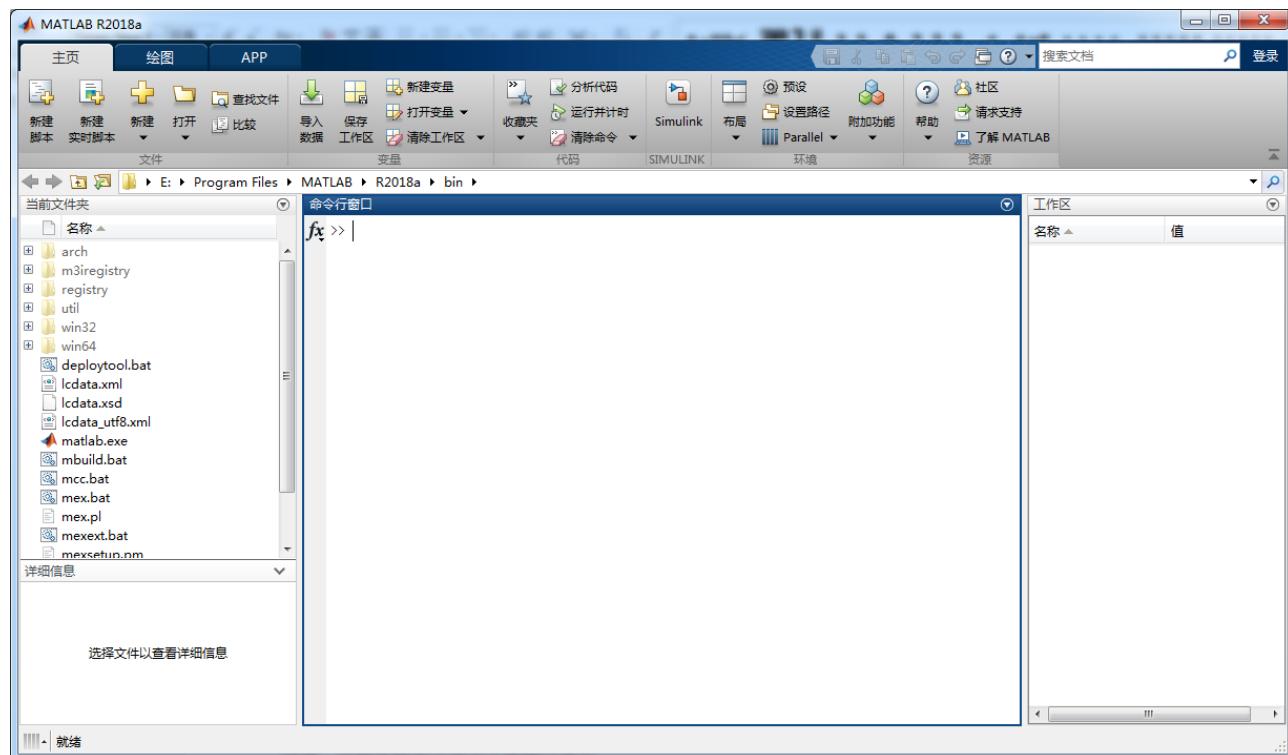


2.2.6 第 6 步，完成安装

再次打开 matlab.exe，出现如下界面，说明已经安装成功了：



打开 Matlab 的界面如下：





2.3 Matlab 简介

MATLAB (矩阵实验室) 是 MATrix LABoratory 的缩写，是一款由美国 The MathWorks 公司出品的商业数学软件。MATLAB 是一种用于算法开发、数据可视化、数据分析以及数值计算的高级技术计算语言和交互式环境。除了矩阵运算、绘制函数/数据图像等常用功能外，MATLAB 还可以用来创建用户界面以及调用其它语言（包括 C, C++, JAVA, Python, FORTRAN 等）编写的程序。

尽管 MATLAB 主要用于数值运算，但利用为数众多的附加工具箱（Toolbox）它也适合不同领域的应用，例如控制系统设计与分析、图像处理、信号处理与通讯、金融建模和分析等。另外还有一个配套软件包 Simulink，提供了一个可视化开发环境，常用于系统模拟、嵌入式系统开发等方面。

2.3.1 Matlab 历史

20世纪70年代末到80年代初，时任美国新墨西哥大学教授的克里夫·莫勒尔为了让学生更方便地使用 LINPACK 及 EISPACK（需要通过 FORTRAN 编程来实现，但当时学生们并无相关知识），独立编写了第一个版本的 MATLAB。这个版本的 MATLAB 只能进行简单的矩阵运算，例如矩阵转置、计算行列式和本征值，此版本软件分发出两三百份。

1984年，杰克·李特、克里夫·莫勒尔和斯蒂夫·班格尔特合作成立了 MathWorks 公司，正式把 MATLAB 推向市场。MATLAB 最初是由莫勒尔用 FORTRAN 编写的，李特和班格尔特花了约一年半的时间用 C 重新编写了 MATLAB 并增加了一些新功能，同时，李特还开发了第一个系统控制工具箱，其中一些代码到现在仍然在使用。C 语言版的面向 MS-DOS 系统的 MATLAB 1.0 在拉斯维加斯举行的 IEEE 决策与控制会议（IEEE Conference on Decision and Control）正式推出，它的第一份订单只售出了 10 份，而到了现在，根据 MathWorks 自己的数据，目前世界上 180 多个国家的超过三百万工程师和科学家在使用 MATLAB 和 Simulink。

1992年，学生版 MATLAB 推出；

1993年，Microsoft Windows 版 MATLAB 面世；

1995年，推出 Linux 版。

2.3.2 Matlab 主要功能

MATLAB 主要提供以下功能（部分）：

- 可用于技术计算的高级语言。
- 可对代码、文件和数据进行管理的开发环境。
- 可以按迭代的方式探查、设计及求解问题的交互式工具。
- 可用于线性代数、统计、傅立叶分析、筛选、优化以及数值积分等的数学函数。
- 可用于可视化数据的二维和三维图形函数。
- 可用于构建自定义的图形用户界面的各种工具。



- 可将基于 MATLAB 的算法与外部应用程序和语言（如 C、C++、Fortran、Java、COM 以及 Microsoft Excel）集成的各种函数。

工具箱

MATLAB 的一个重要特点是可扩展性。作为 Simulink 和其它所有 MathWorks 产品的基础，MATLAB 可以通过附加的工具箱（Toolbox）进行功能扩展，每一个工具箱就是实现特定功能的函数的集合。MathWorks 提供的工具箱分以下几大类（部分）：

- 数学和优化。
- 统计和数据分析。
- 控制系统设计和分析。
- 信号处理和通讯。
- 图像处理。
- 测试和测量。
- 金融建模和分析。
- 应用程序部署。
- 数据库连接和报表。
- 分布式计算。

这些工具箱大多是用开放式的 MATLAB 语言写成，用户不但可以查看源代码，还可根据自己的需要进行修改以及创建自定义函数。此外，常有用户在 MATLAB Central: File Exchange 发布自己编写的 MATLAB 程序或工具箱，供他人自由下载使用。

2.3.3 Matlab 语言

MATLAB 语言是一种交互性的数学脚本语言，其语法与 C/C++ 类似。它支持包括逻辑（boolean）、数值（numeric）、文本（text）、函数柄（function handle）和异质数据容器（heterogeneous container）在内的 15 种数据类型，每一种类型都定义为矩阵或阵列的形式（0 维至任意高维）。

执行 MATLAB 代码的最简单方式是在 MATLAB 程序的命令窗口（Command Window）的提示符处（>>）输入代码，MATLAB 会即时返回操作结果（如果有的话）。此时，MATLAB 可以看作是一个交互式的数学终端，简单来说，一个功能强大的“计算器”。MATLAB 代码同样可以保存在一个以.m 为后缀名的文本文件中，然后在命令窗口或其它函数中直接调用。

2.4 总结

本期教程主要是讲述了 Matlab 的安装过程，相对比较容易，下期教程开始讲解 Matlab 的使用。

第3章 Matlab 简易使用之基础操作

本期教程开始讲解 Matlab 的简易使用之基础操作，作为学习 DSP 的必备软件，掌握简单的 Matlab 操作是必须的。

3.1 初学者重要提示

3.2 Matlab 界面说明

3.3 Matlab 矩阵和阵列

3.4 Matlab 检索矩阵中的数据

3.5 Matlab 工作区中的数据保存和加载

3.6 Matlab 字符串

3.7 Matlab 函数

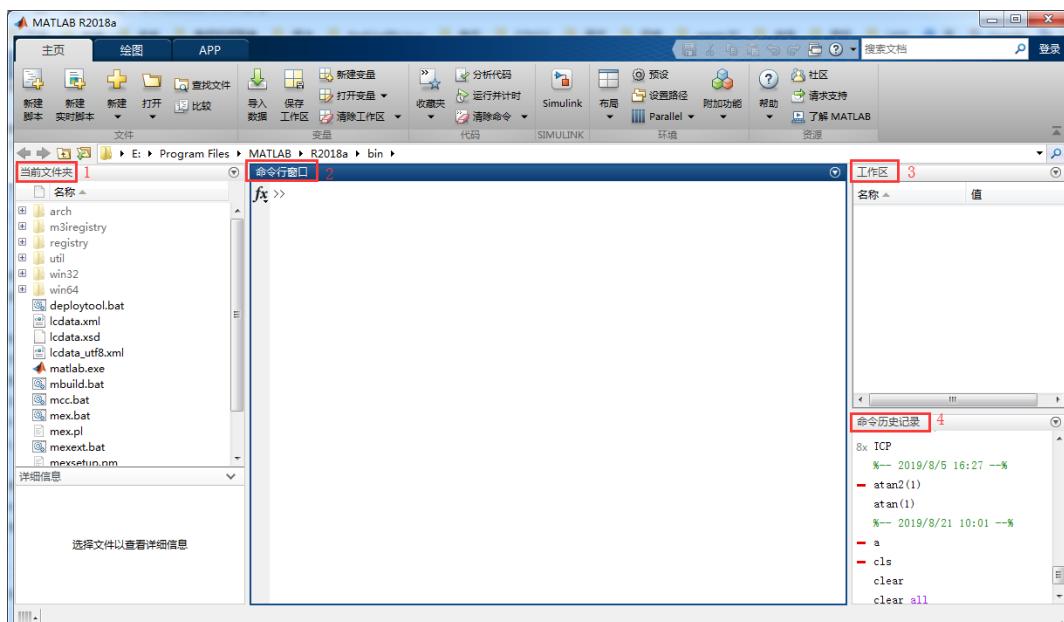
3.8 Matlab 绘图功能

3.9 总结

3.1 初学者重要提示

◆ 本章主要介绍了 matlab 的基础操作，如果之前没有接触过这方面的知识，务必要实际动手操作。

3.2 Matlab 界面说明



◆ 当前文件夹 (Current Folder)

用于访问电脑中的文件。

◆ 命令窗口 (Command Window)

用于输入命令，公式计算等也可以在这里进行。

◆ 工作区 (Workspace)

浏览用户创建的数据或者从文件中导入的数据。

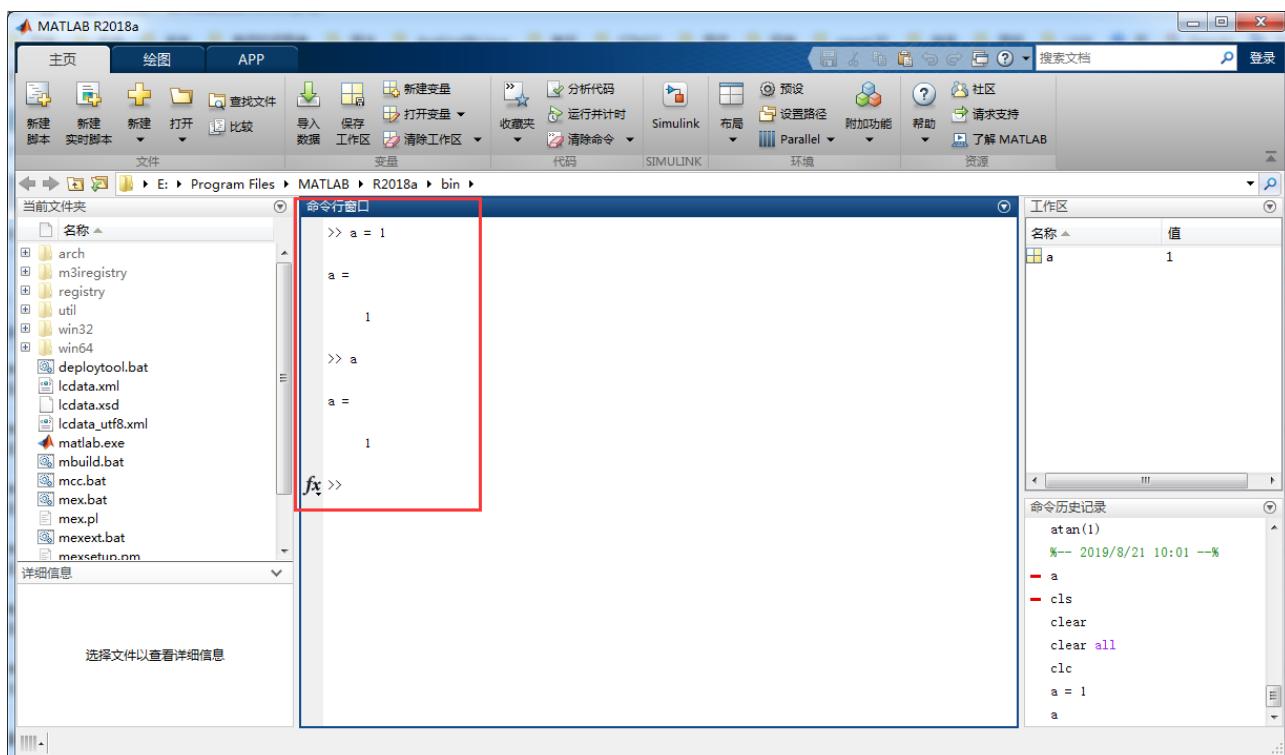
◆ 命令历史记录 (Command History)

记录用户在 command 窗口输入的命令，双击这些历史命令可以返回到 command 窗口继续执行。

下面通过简单的例子说明一下 command 窗口的使用。

3.2.1 简单计算

在 command 窗口输入变量 $a = 1$ ，然后回车，再次输入 a ，然后回车。

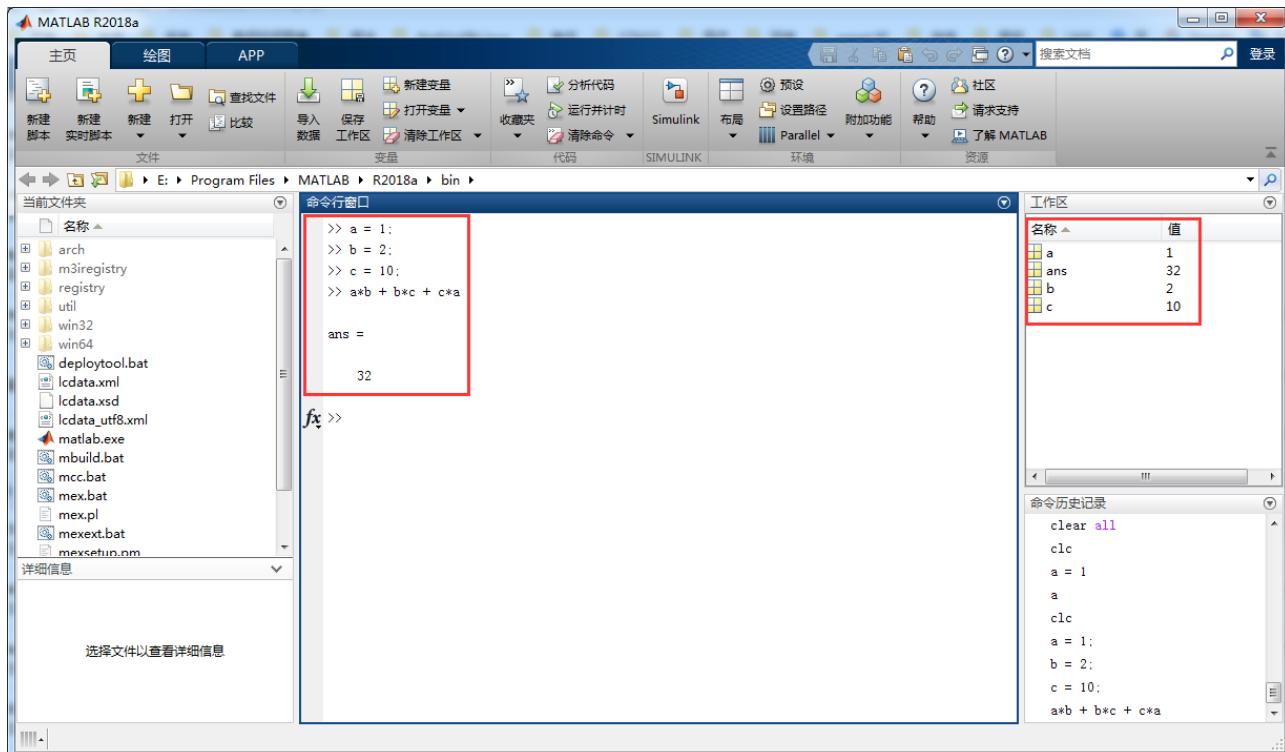


第一次输入 $a = 1$ 并回车后会将变量 a 及其数值加入到工作区 (Workspace) 中。

第二次输入 a 并回车后会将变量 a 以前的赋值显示出来。

3.2.2 稍复杂计算

在 command 窗口输入以下计算：



注意：上面输入一行后加入了分号，这个分号很重要，加上分号后再回车就可以输入下个计算，否则会输出计算结果。在需要获得结算结果的时候，就不再需要这个分号了，直接回车即可。

如果没有结果变量的话，输出会是 `ans = xxx` (使用 `ans` 作为输出变量)。

3.2.3 历史命令行调用

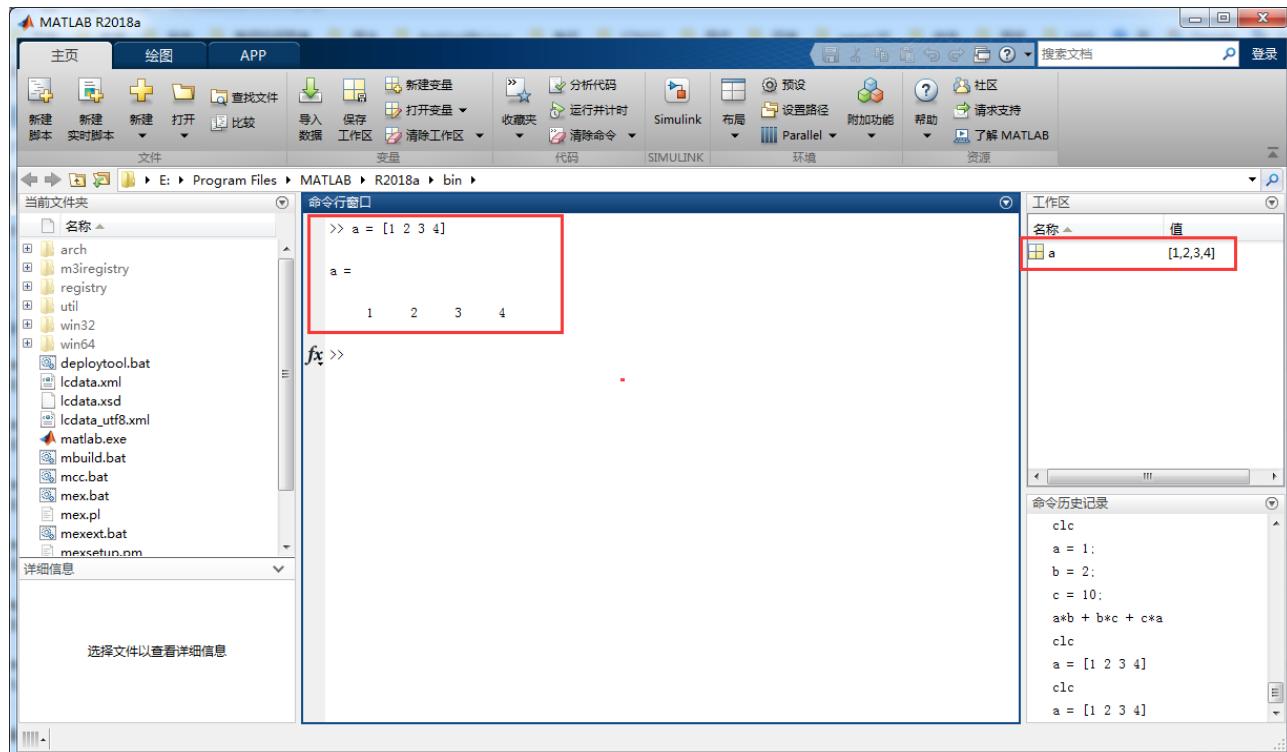
历史命令行的调用除了可以在右下角的 Command 里面调用以外，还可以通过键盘上面的按键↑和↓实现历史命令的查询。

3.3 Matlab 矩阵和阵列

Matlab 的主要设计是对整个矩阵和数组操作。不管什么类型的数据，所有 MATLAB 的变量是多维数组。矩阵是一个二维阵列通常用于线性代数。

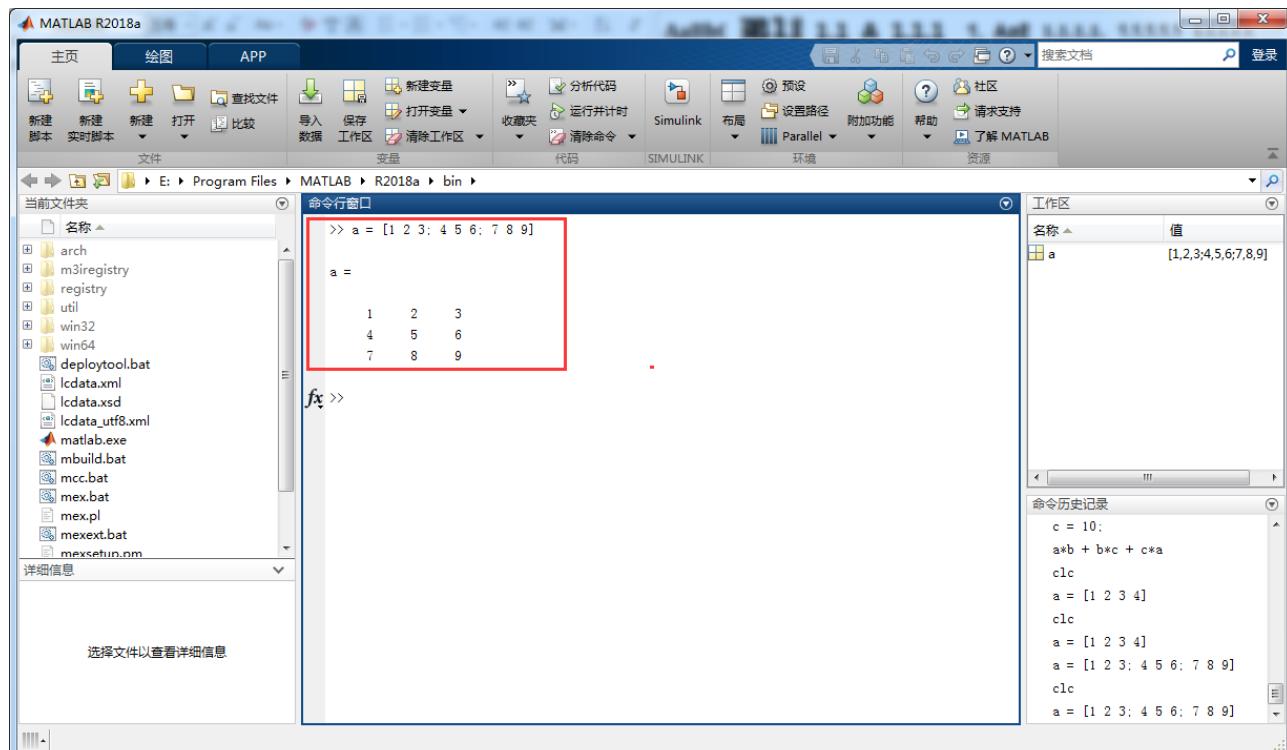
3.3.1 创建数组

下面在 matlab 中创建一个一行四列的数组，数组中的每个元素用逗号或者空格分开。比如创建数组

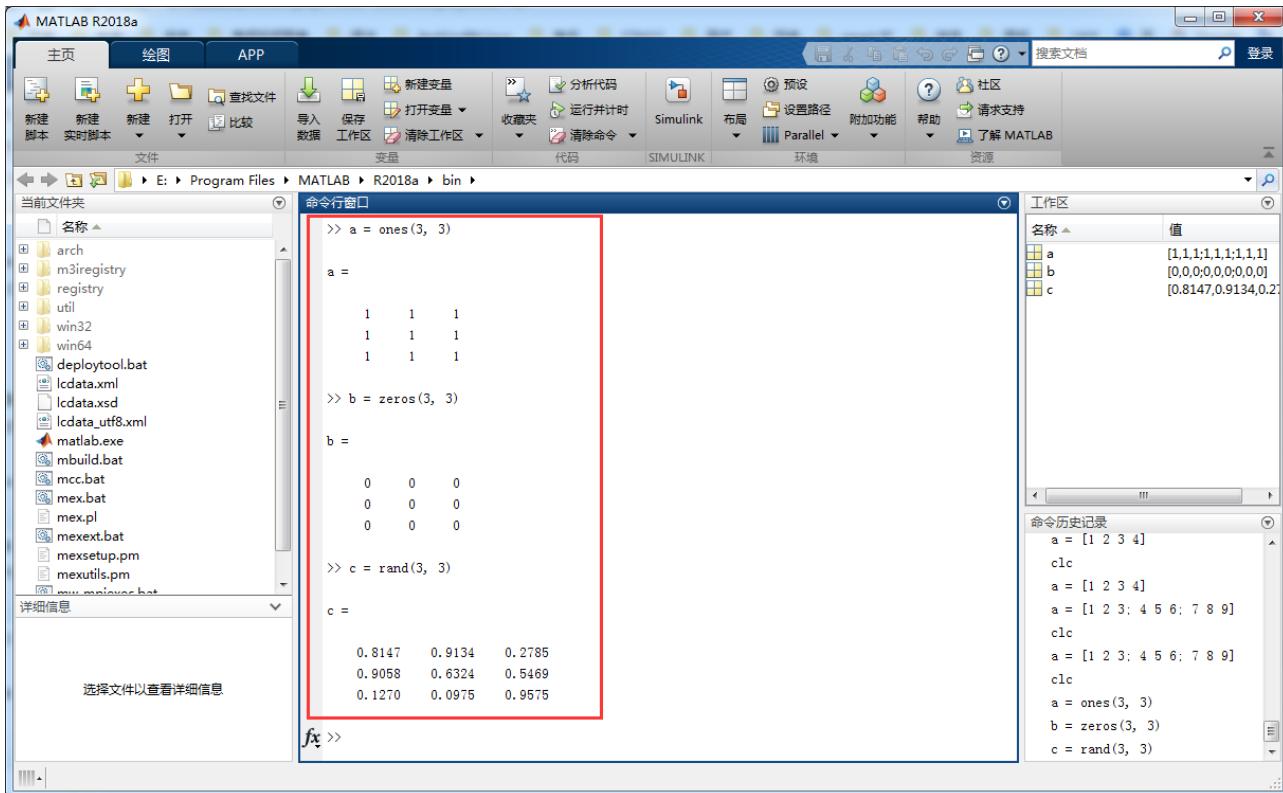


这种类型的数组也称为行向量。

下面创建一个多行的矩阵，不同的行用分号隔开：

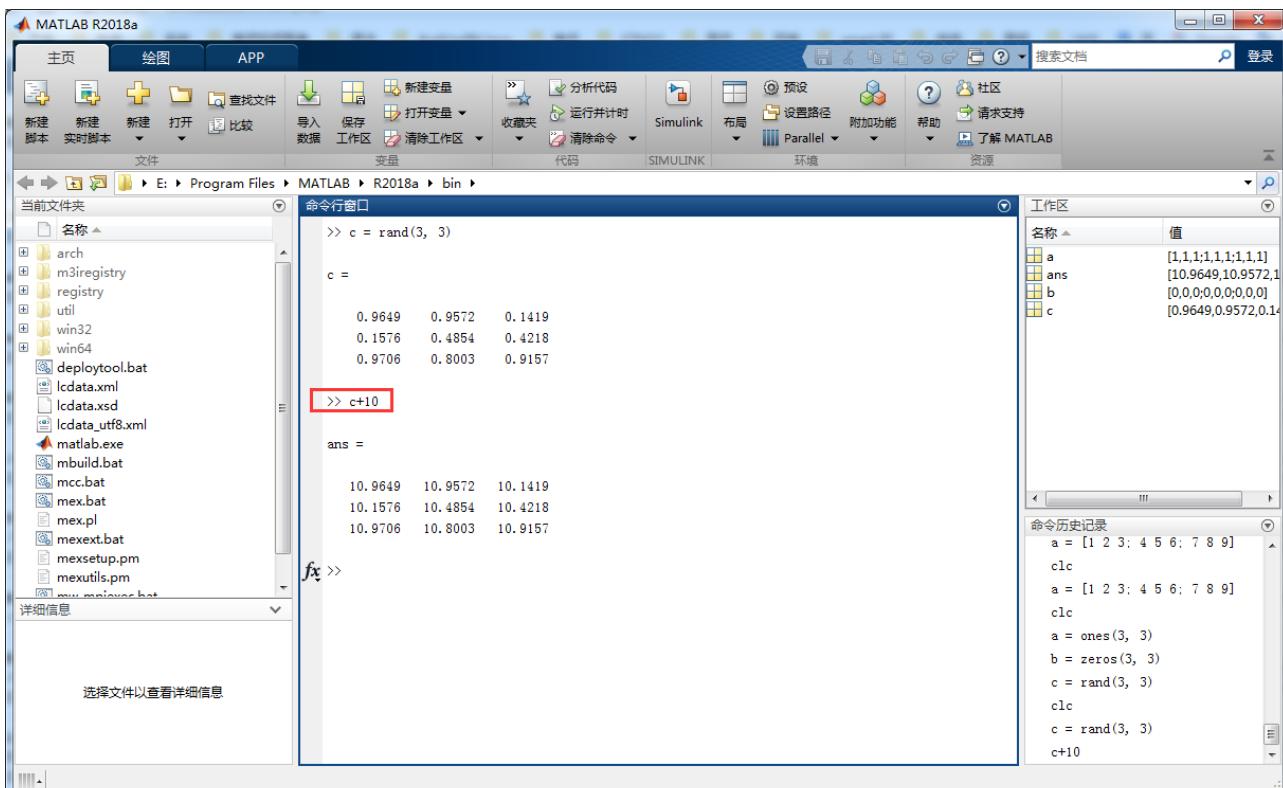


当然，也可以使用 Matlab 自带的函数进行创建，比如 `ones`, `zeros`, `rand` 等。

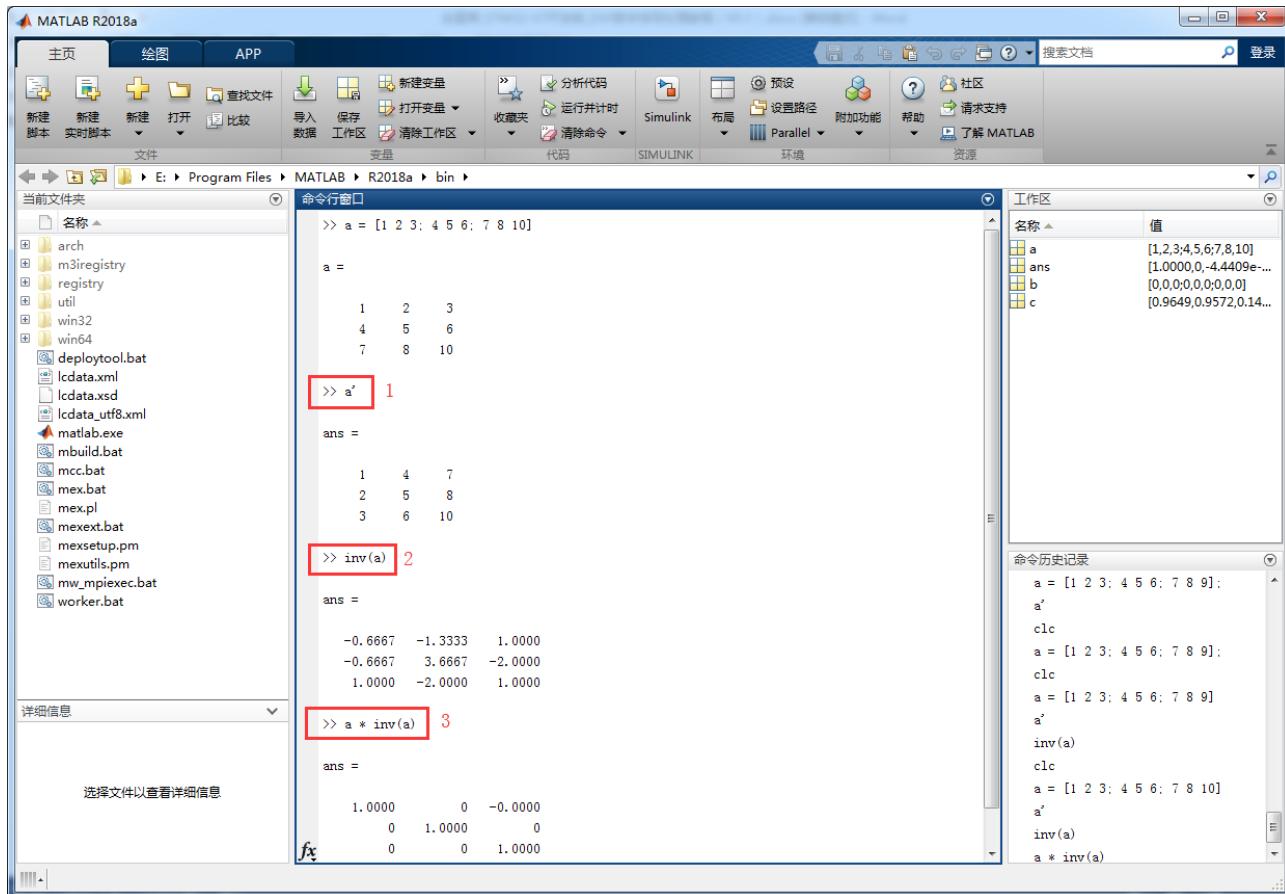


3.3.2 矩阵和阵列运算

MATLAB 允许客户使用一个单一的算术运算符或函数来处理所有在矩阵中的值。比如：



下面继续说一下 matlab 矩阵的转置，求逆矩阵。



1. 给矩阵 a 加上符号 a' 用于求转置矩阵。
2. inv() 用于求逆矩阵。
3. 矩阵 a 乘以 a 的逆矩阵就是求单位矩阵。

注意上面的 $a \cdot inv(a)$ 得到的结果已经不再是整数矩阵，Matlab 存储结果的时候会以浮点的形式进行存储，Matlab 实际存储的数值和当前命令窗口显示的数值是有区别的。为了获得更高的显示精度可以使用下面的数据格式



执行逐个元素乘法，而不是矩阵的乘法可以使用符号 `.*` 来实现：

```
>> a.*a  
ans =  
    1      4      9  
   16     25     36  
   49     64    100
```

下面是实现矩阵各个元素的 3 次方

```
>> a.*a  
ans =  
     1      4      9  
    16     25     36  
    49     64    100
```

3.3.3 矩阵的合并

矩阵的合并主要有以下两种形式：

```
>>A = [a,a]
A =
    1   2   3   1   2   3
    4   5   6   4   5   6
    7   8   10  7   8   10
```

$$\begin{aligned} >> A &= [a; a] \\ A &= \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 10 \end{matrix} \end{aligned}$$



1	2	3
4	5	6
7	8	10

3.3.4 复数

复数的表示是由实部和虚部组成的，比如我们在 matlab 命令窗口输入：

```
>>sqrt(-1)
ans =
    0 + 1.0000i
```

为了表示复数的虚部，可以使用 i 或者 j 来表示：

```
>>c = [3+4i, 4+3j, -i, 10j]
c =
    3.0000 + 4.0000i   4.0000 + 3.0000i   0 - 1.0000i   0 + 10.0000i
```

3.4 Matlab 检索矩阵中的数据

有时矩阵中的元素比较多，为了方便用户可以检索矩阵中所需要查找的元素，调用相关命令即可。比如：先用 magic 函数生成 4 阶幻方矩阵：

```
>>A = magic(4)
A =
    16     2     3     13
      5    11     10     8
      9     7     6    12
      4    14    15     1
```

◆ 如果我们要获得第 4 行第 2 列的数据（注意，行列从 1 开始算的），可以采用如下的方法：

```
>>A(4, 2)
ans =
    14
```

◆ 简单点，也可以使用如下方法进行定位：

```
>>A(8)
ans =
    14
```

◆ 如果检索超出了矩阵的范围，会报错，如下：

```
>> test = A(4, 5)
Attempted to access A(4, 5); index out of bounds because size(A)=[4, 4].
```

◆ 用户可以通过如下方法增加行和列

```
>> A(5, 5) = 14
A =
    16     2     3     13     0
      5    11     10     8     0
      9     7     6    12     0
      4    14    15     1     0
      0     0     0     0    14
```

◆ 用户可以通过如下方法访问某行某列的某些数据

```
>> A(1:3, 2)
ans =
    2
    11
    7
>> A(3, :)
```

```
ans =  
    9     7     6    12     0
```

◆ 使用冒号运算符，用户可以获得一个等间距序列，冒号隔开的数值分别表示 start:step:end

```
>> B = 0:10:100  
B =  
Columns 1 through 10  
    0    10    20    30    40    50    60    70    80    90  
Column 11  
    100
```

如果不设置 step，那么输出结果默认步是 1。

3.5 Matlab 工作区中的数据保存和加载

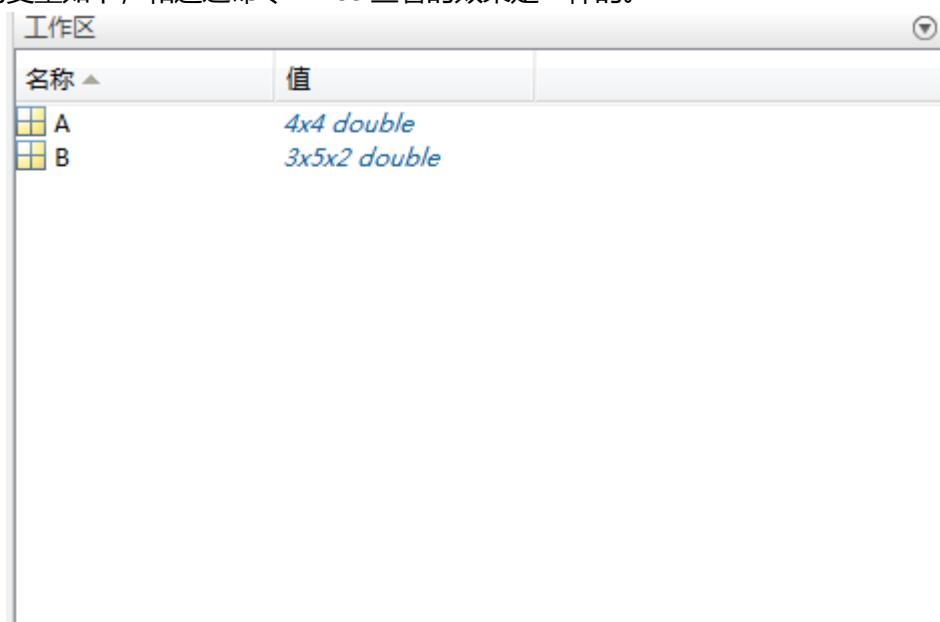
工作区中的变量来自用户创建，外部数据加载或者其它的编程加入。比如我们在命令窗户输入以下两个函数。

```
>> A = magic(4);  
B = rand(3, 5, 2);
```

通过命令 whos 可以查看工作区中的变量内容。

```
>> whos  
Name      Size            Bytes  Class       Attributes  
A            4x4             128  double          
B            3x5x2           240  double        
```

工作区中的变量如下，和通过命令 whos 查看的效果是一样的。



如果用户退出 Matlab 后，再次进入工作区中，那么里面的数据变量将被清空。如果想保持这些变量可以通过如下命令进行保存：

```
>> save myfile.mat
```

下次打开 MATLAB 可以通过如下命令加载这些变量：

```
>> load myfile.mat
```

另外用户可以通过命令 clear 清除当前工作区中的变量。



3.6 Matlab 字符串

- ◆ 在 matlab 中显示字符串跟使用 C 不一样， matlab 中使用单引号即可。比如：

```
>> myText = 'Hello, world'  
myText =  
Hello, world  
  
>> otherText = 'You''re right' %特别的注意这里，显示单引号需要写两个才可以。  
otherText =  
You're right  
  
>> whos
```

Name	Size	Bytes	Class	Attributes
myText	1x12	24	char	
otherText	1x12	24	char	

- ◆ 如果想合并两个字符串可以用如下的方法：

```
>> longText = [myText, ' - ', otherText]  
longText =  
Hello, world - You're right
```

- ◆ 如果想把数字转换成字符串显示，可以用函数 num2str 或者 int2str.

```
>> f = 71;  
c = (f-32)/1.8;  
tempText = ['Temperature is ', num2str(c), 'C']  
tempText =  
Temperature is 21.6667C
```

3.7 Matlab 函数

MATLAB 支持的函数非常多，下面举一个简单的例子说明下，后面具体用到那个函数查阅手册即可。

```
>> A = [1 3 5];  
B = [10 6 4];  
>> max(A) %求最大值  
ans =  
5  
  
>> max(A, B) %求 A, B 中的最大值  
ans =  
10 6 5  
  
>> maxA = max(A) %将最大值付给 maxA  
maxA =  
5  
  
>> [maxA, location] = max(A) %将最大值和次最大值赋给两个变量  
maxA =  
5  
location =  
3
```

- ◆ 显示任何字符串可以调用函数：

```
>> disp('hello armfly')  
hello armfly
```

- ◆ 命令窗口数据的清除可以使用命令

```
>>clc
```

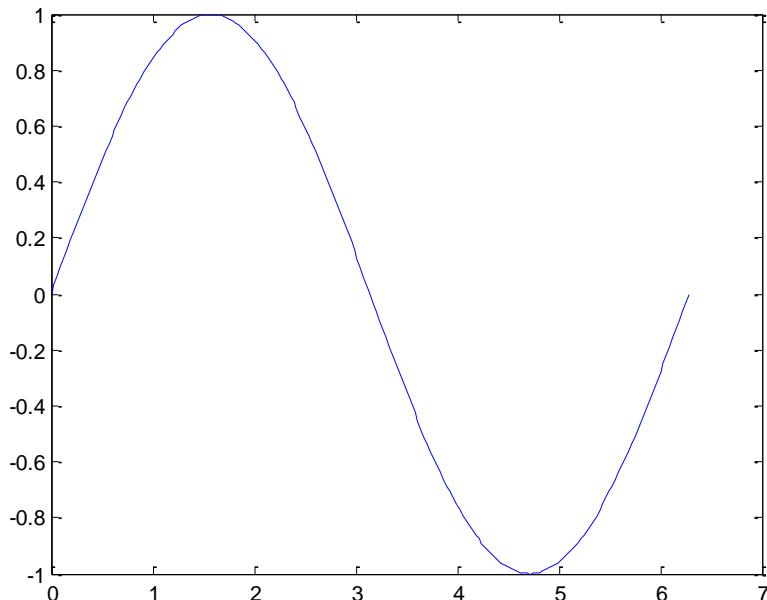
3.8 Matlab 绘图功能

Matlab 的绘图功能非常强劲，下面分别简单介绍下。

3.8.1 画线

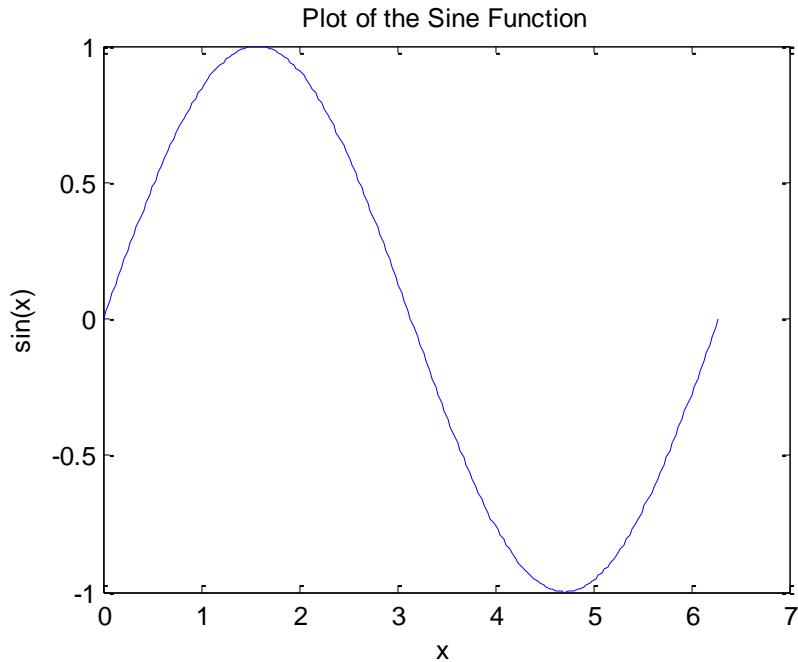
下面使用绘图功能创建一个二维图：

```
>> x = 0:pi/100:2*pi;  
y = sin(x);  
plot(x, y)
```

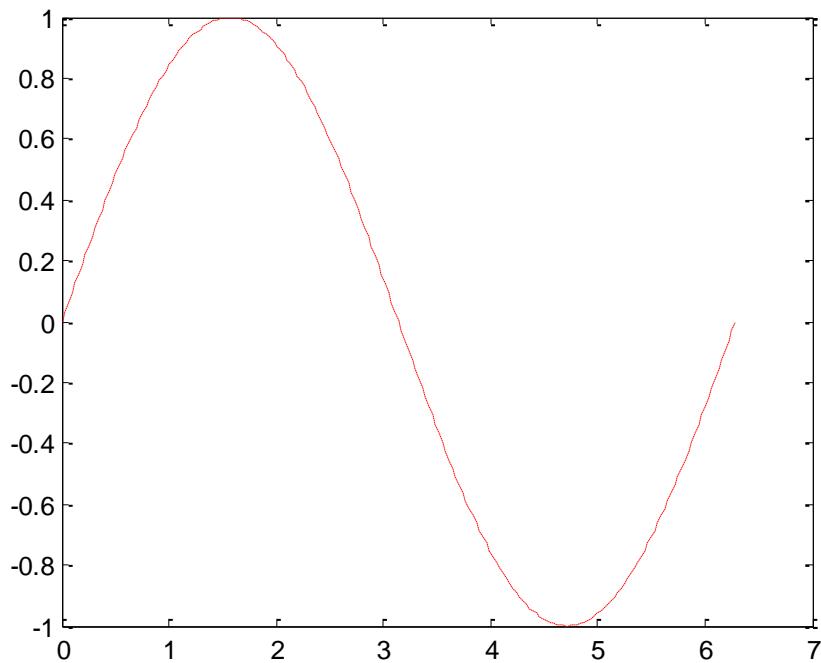


通过如下函数可以给绘图加上标题：

```
>> x = 0:pi/100:2*pi;  
y = sin(x);  
plot(x, y);  
xlabel('x');  
ylabel('sin(x)');  
title('Plot of the Sine Function')
```



通过函数 `plot(x,y,'r--')` 可以改变曲线的颜色和显示方式。



如果想把两个波形显示在一个图中，可以采用函数 `hold on`，如下所示：

```
>> x = 0:pi/100:2*pi;
```

```
y = sin(x);
```

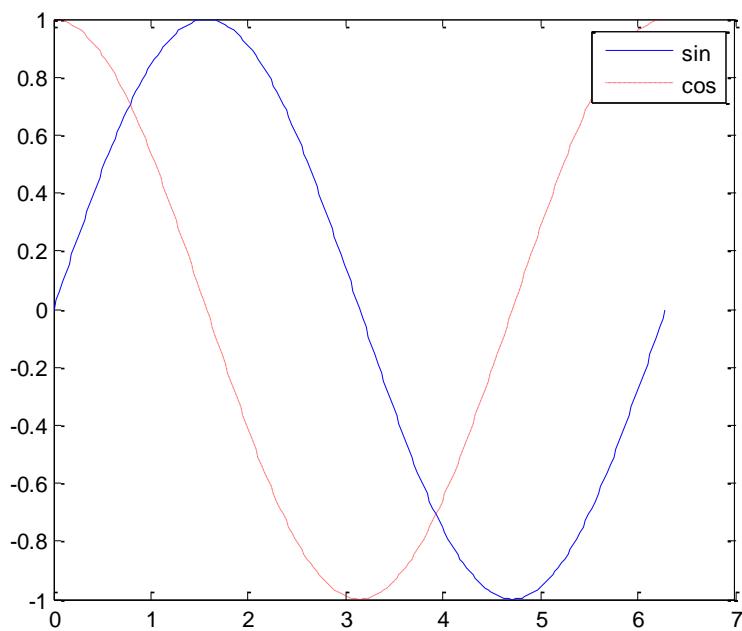
```
plot(x, y)
```

```
hold on
```

```
y2 = cos(x);
```

```
plot(x, y2, 'r:')
```

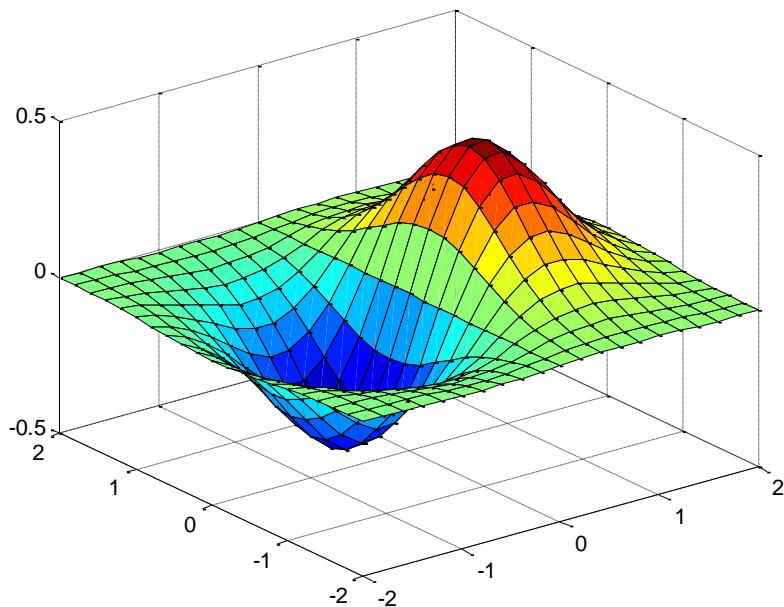
```
legend('sin', 'cos')
```



3.8.2 3-D 绘图

Matlab 也支持 3-D 绘图，下面举一个简单的例子，主要是为了说明显示效果：

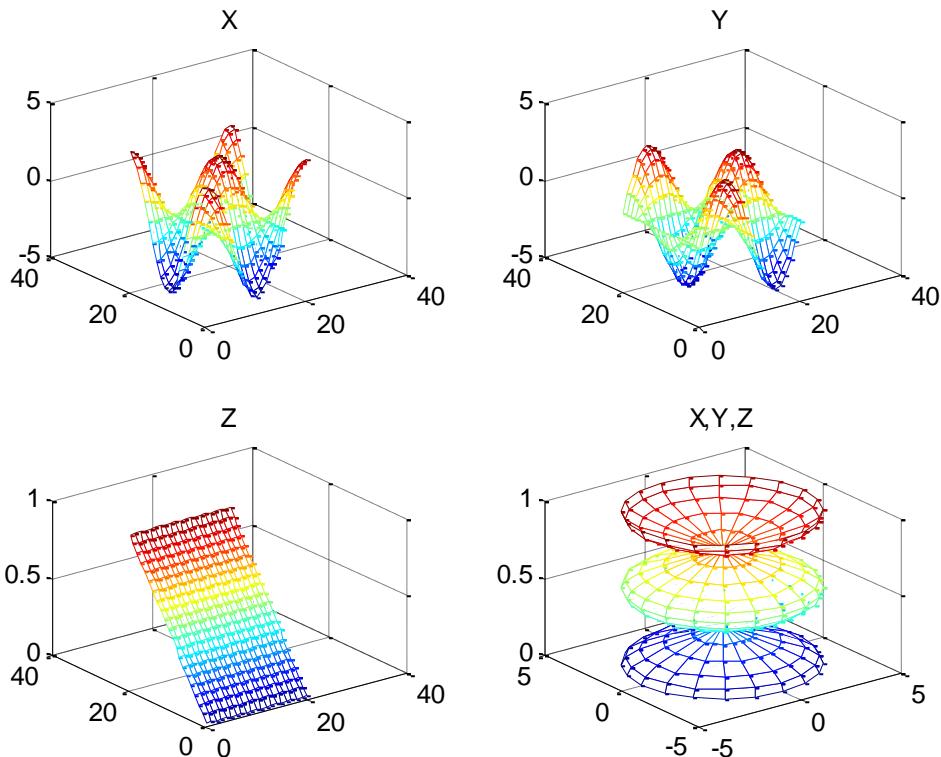
```
>> [X, Y] = meshgrid(-2:.2:2);  
Z = X .* exp(-X.^2 - Y.^2);  
surf(X, Y, Z)
```



3.8.3 多个子图的绘制

Matlab 也支持在一幅图中绘制多个子图片，主要是通过函数 subplot 实现：

```
>> t = 0:pi/10:2*pi;  
[X, Y, Z] = cylinder(4*cos(t));  
subplot(2, 2, 1); mesh(X); title('X');  
subplot(2, 2, 2); mesh(Y); title('Y');  
subplot(2, 2, 3); mesh(Z); title('Z');  
subplot(2, 2, 4); mesh(X, Y, Z); title('X, Y, Z');
```



3.9 总结

本期主要跟大家讲解了 Matlab 的简单使用方法，后面复杂的使用需要大家多查手册，多练习。



第4章 Matlab 简易使用之脚本文件

本期教程主要是讲解 Matlab 的 m 文件简易使用方法，有些内容跟上一节相同，但是比上一些更详细。

4.1 初学者重要提示

4.2 Matlab 的脚本文件.m 的使用

4.3 Matlab 中的条件和循环函数

4.4 绘图功能

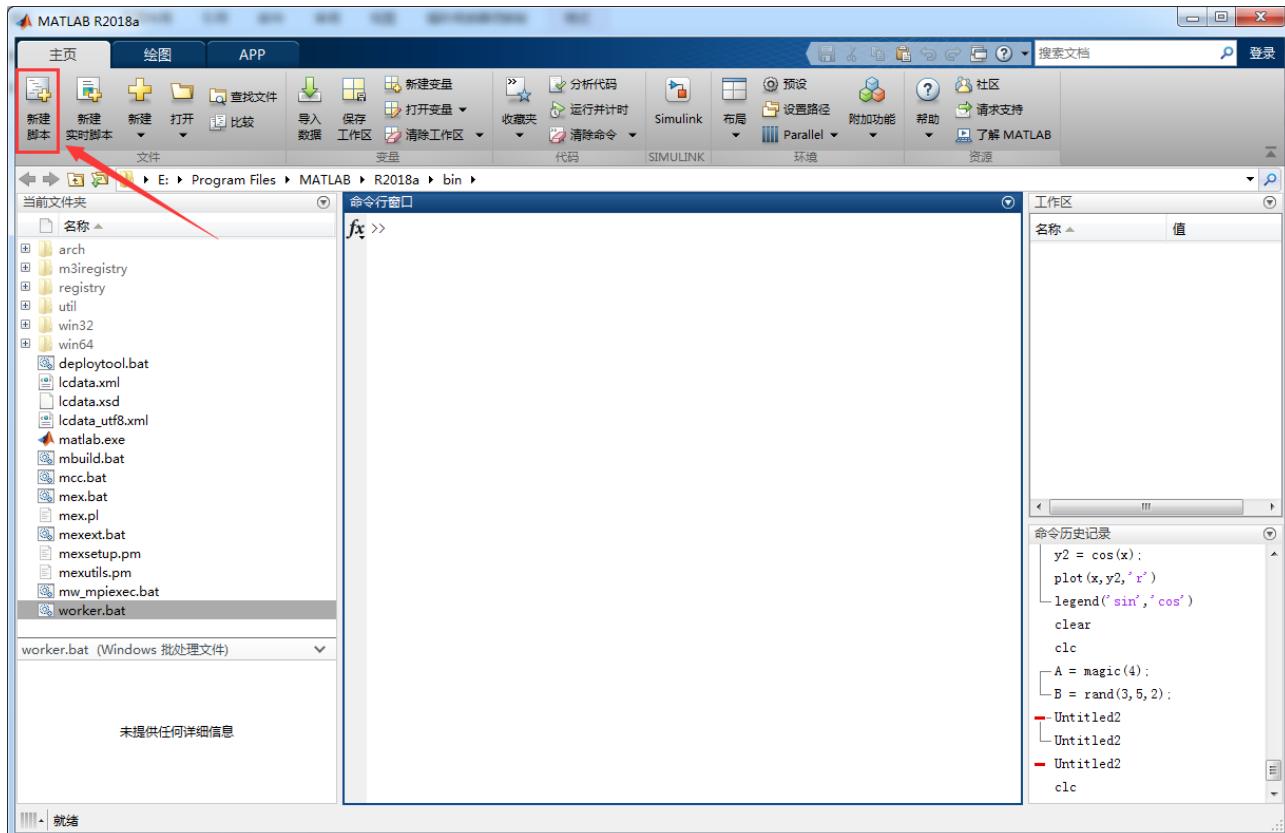
4.5 总结

4.1 初学者重要提

- ◆ 学习本章节前，务必优先学习第 3 章。
- ◆ 对于 Matlab 的 m 文件使用方法，务必要熟练掌握，后续章节都是基于 m 文件做测试。

4.2 Matlab 的脚本文件.m 的使用

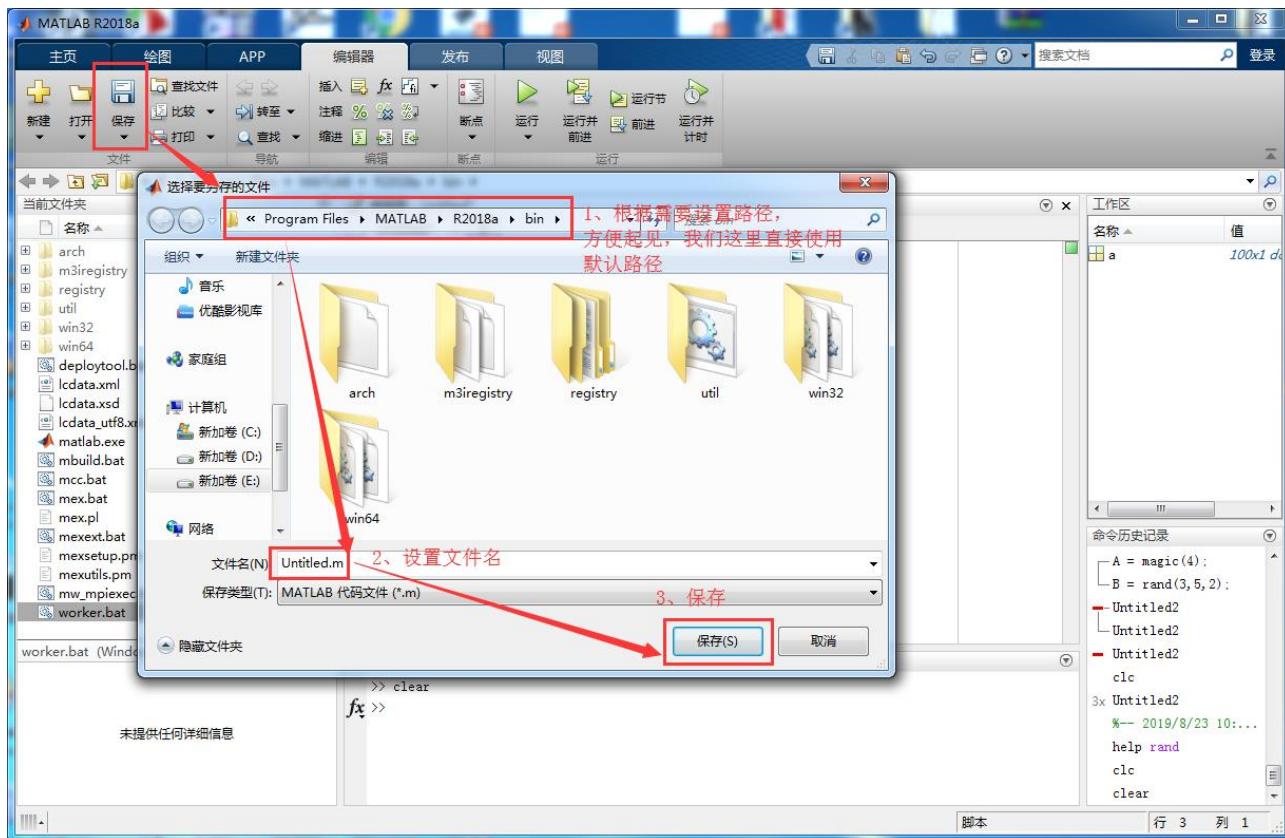
在 matlab 上创建和使用.m 文件跟在 MDK 或者 IAR 上面创建和使用.C 或者.ASM 文件是一样的。创建方法如下：



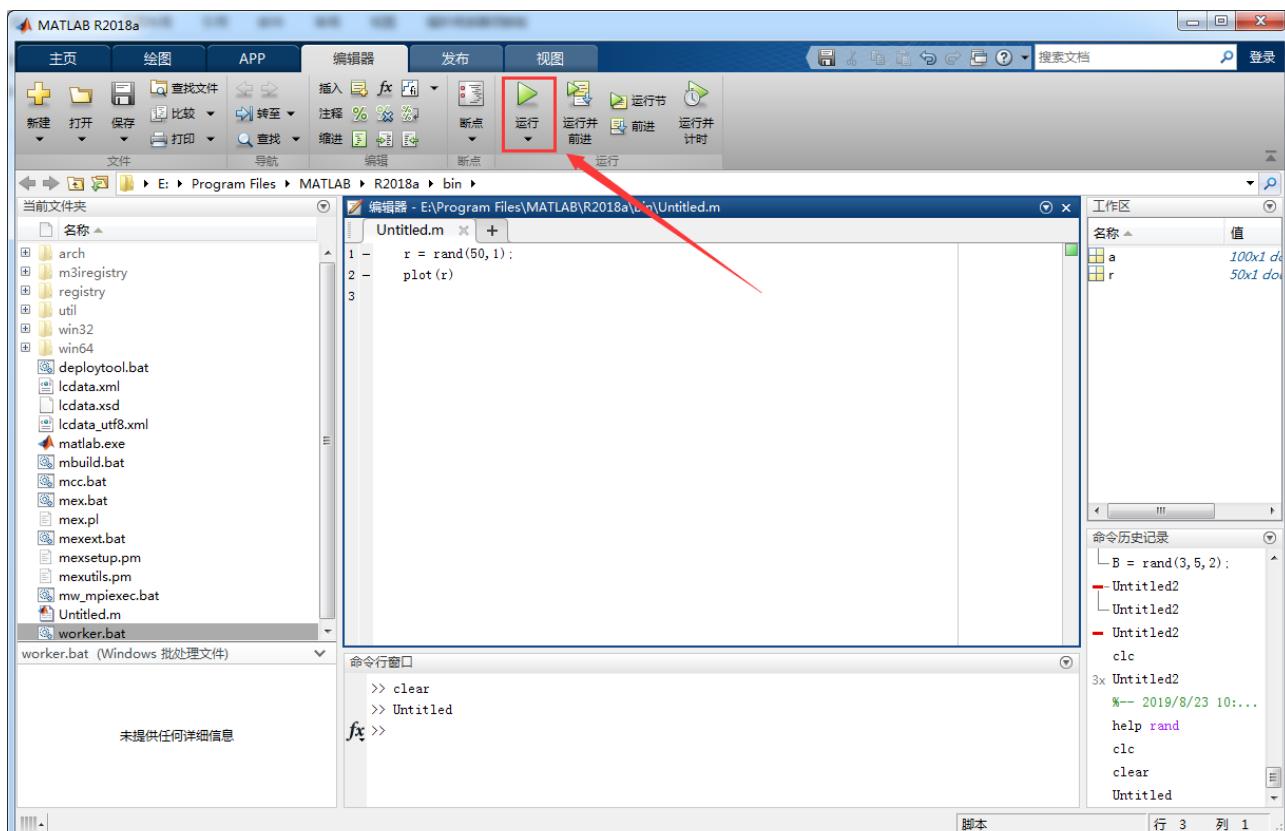
点击上图中的小图标，打开编辑窗口后，输入以下函数：

```
r = rand(50, 1);
plot(r)
```

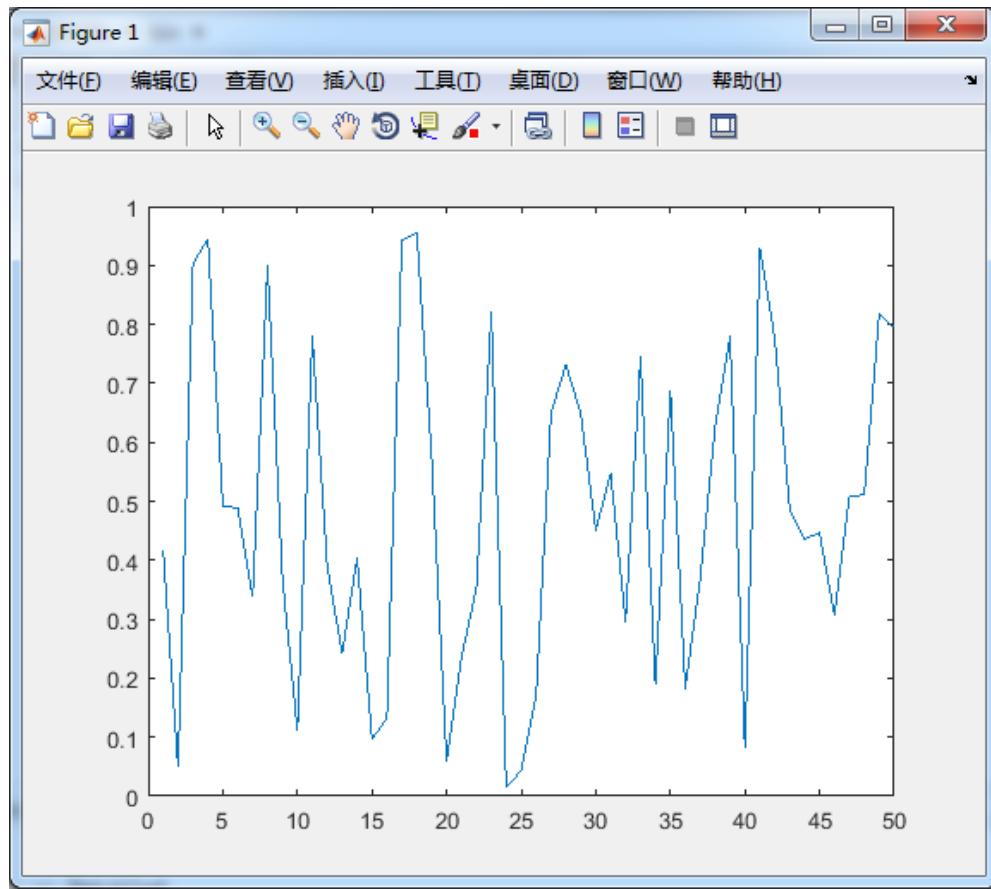
编辑好函数后需要将当前文件进行保存：



然后点击如下图标即可运行（或者按 F5）：



显示效果如下：



4.3 Matlab 中的条件和循环函数

matlab 也支持类似 C 语言中的条件和循环语句：for, while, if, switch。但在 matlab 中使用比在 C 中使用更加随意。

◆ 比如在.M 文件中输入以下函数：

```
nsamples = 5;
npoints = 50;

for k = 1 : nsamples
    currentData = rand(npoints, 1);
    sampleMean(k) = mean(currentData);
end
overallMean = mean(sampleMean)
```

在命令窗口得到输出结果：



```
命令行窗口
>> Untitled

overallMean =
0.4910

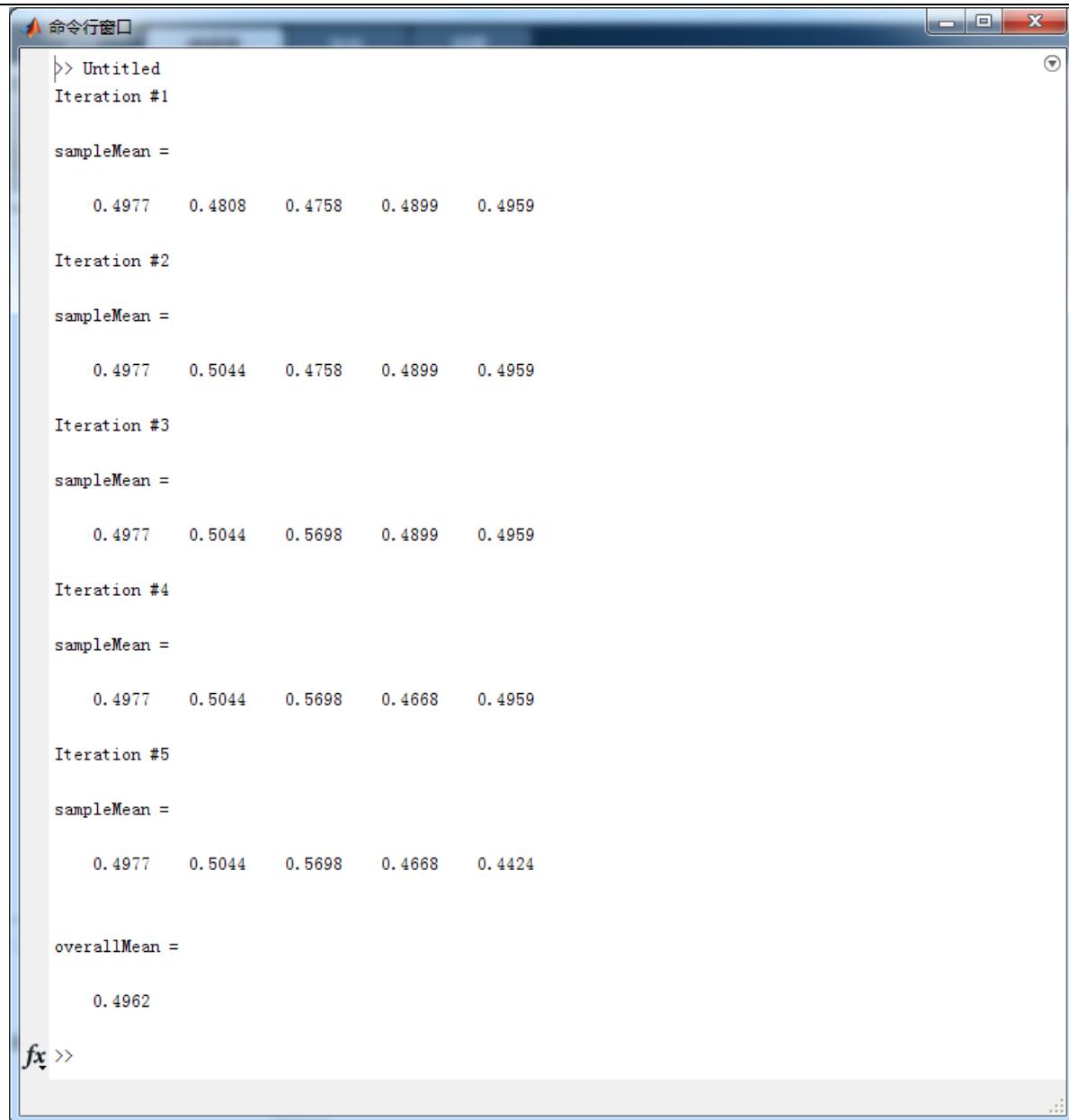
fx >>
```

- ◆ 为了将上面函数每次迭代的结果都进行输出可以采用如下方法：

```
nsamples = 5;
npoints = 50;

for k = 1:nsamples
    iterationString = ['Iteration #', int2str(k)];
    disp(iterationString) %注意这里没有分号，这样才能保证会在命令窗口输出结果
    currentData = rand(npoints, 1);
    sampleMean(k) = mean(currentData) %注意这里没有分号
end
overallMean = mean(sampleMean) %注意这里没有分号
```

在命令窗口得到输出结果：



```
>> Untitled
Iteration #1

sampleMean =

    0.4977    0.4808    0.4758    0.4899    0.4959

Iteration #2

sampleMean =

    0.4977    0.5044    0.4758    0.4899    0.4959

Iteration #3

sampleMean =

    0.4977    0.5044    0.5698    0.4899    0.4959

Iteration #4

sampleMean =

    0.4977    0.5044    0.5698    0.4668    0.4959

Iteration #5

sampleMean =

    0.4977    0.5044    0.5698    0.4668    0.4424

overallMean =

    0.4962

fx >>
```

◆ 如果在上面的函数下面加上如下语句：

```
if overallMean < .49
    disp('Mean is less than expected')
elseif overallMean > .51
    disp('Mean is greater than expected')
else
    disp('Mean is within the expected range')
end
```

命令窗口输出结果如下（这里仅列出了最后三行）：

```
命令行窗口
0.5745    0.5960    0.4858    0.4978    0.5128
overallMean =
0.5334
Mean is greater than expected
fx >>
```

4.4 绘图功能

4.4.1 基本的 plot 函数

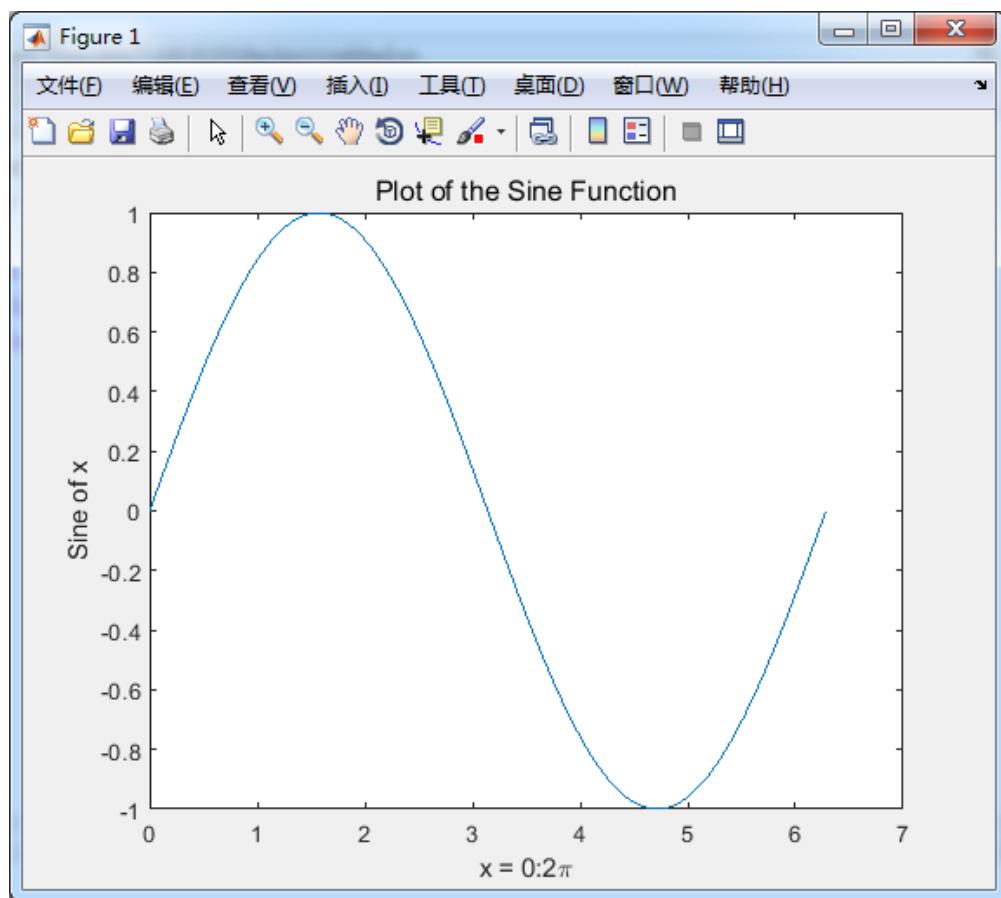
- ◆ 根据 plot 不同的输入参数，主要有两种方式：

- plot(y)，这种方式的话，主要是根据 y 的数据个数产生一个线性曲线。
- plot(x, y)以 x 轴为坐标进行绘制。

比如在命令窗口或者.m 文件中写如下函数：

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y)

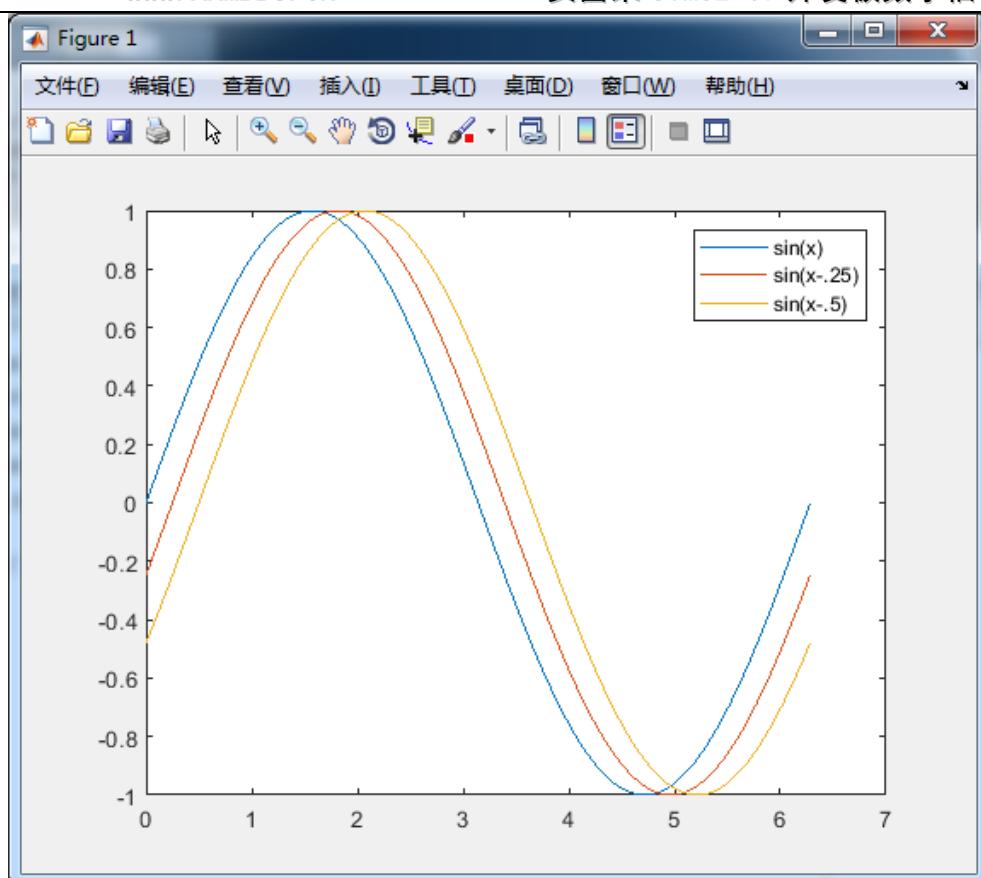
xlabel('x = 0:2\pi')
ylabel('Sine of x')
title('Plot of the Sine Function', 'FontSize', 12)
```



◆ 下面这个函数可以实现在一个图片上显示多个曲线。

```
x = 0:pi/100:2*pi;
y = sin(x);
y2 = sin(x-.25);
y3 = sin(x-.5);
plot(x,y, x,y2, x,y3)

legend('sin(x)', 'sin(x-.25)', 'sin(x-.5)')
```



- ◆ 另外曲线的样式和颜色都可以进行配置的，命令格式如下：

```
plot(x, y, 'color_style_marker')
```

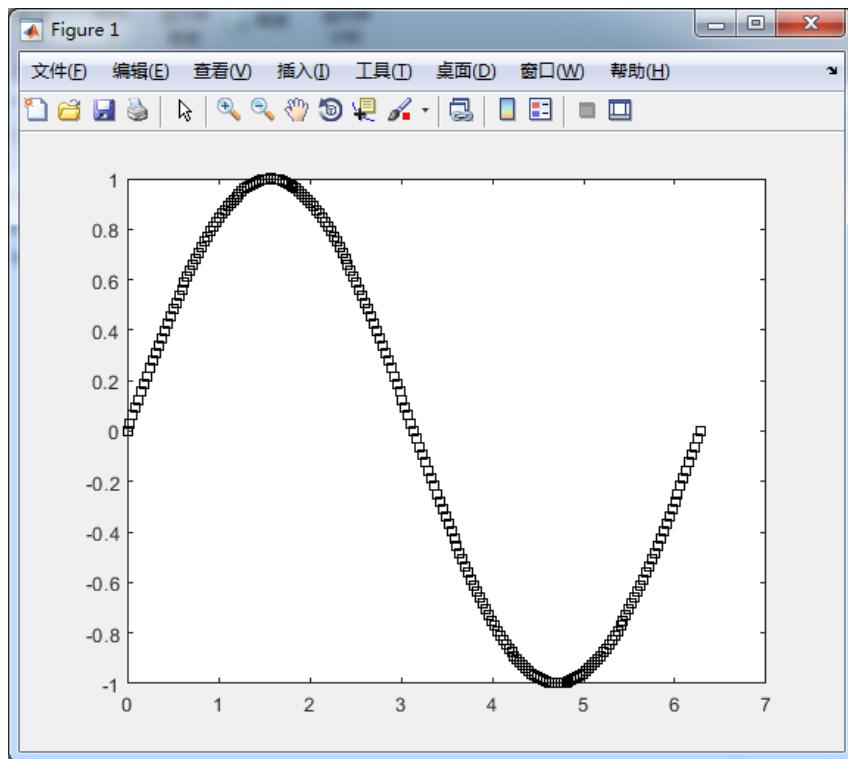


Type	Values	Meanings
Color	'c' 'm' 'y' 'r' 'g' 'b' 'w' 'k'	cyan magenta yellow red green blue white black
Line style	'-' '--' ' :' ' .-' no character	solid dashed dotted dash-dot no line
Marker type	'+' 'o' '*' 'x' 's' 'd' '^' 'v' '>' '<' 'p' 'h' no character	plus mark unfilled circle asterisk letter x filled square filled diamond filled upward triangle filled downward triangle filled right-pointing triangle filled left-pointing triangle filled pentagram filled hexagram no marker

下面通过几个实例看一下实际显示效果。

```
x = 0:pi/100:2*pi;  
y = sin(x);  
plot(x, y, 'ks')
```

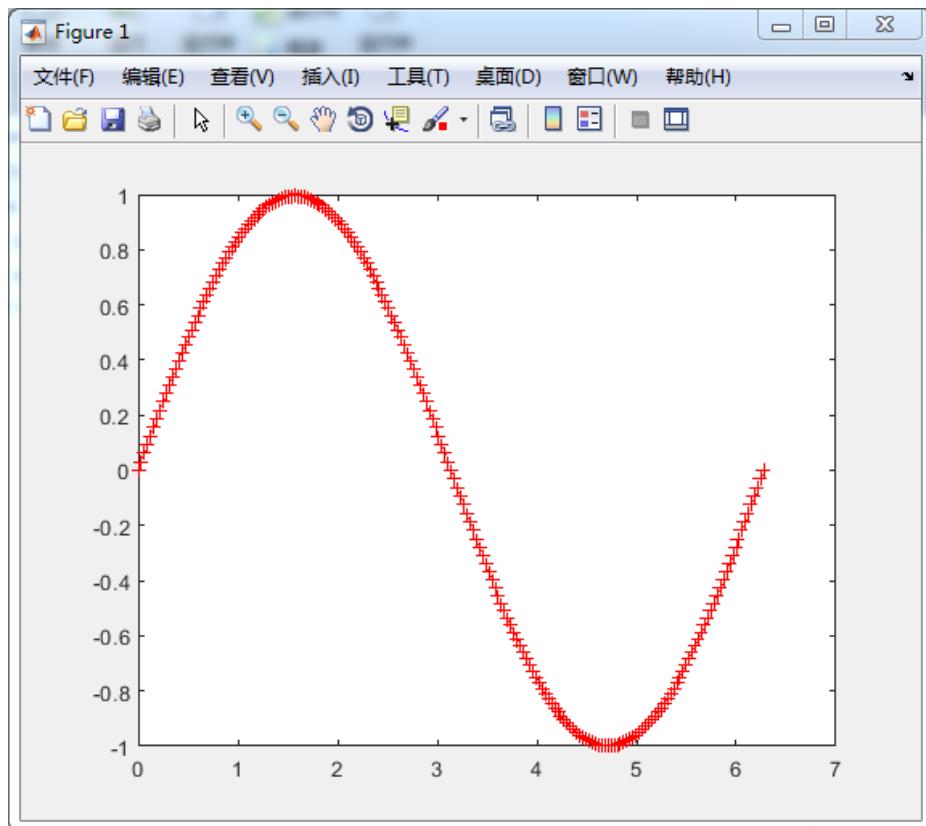
显示效果如下：



下面函数的显示效果：

```
x = 0:pi/100:2*pi;
y = sin(x);
plot(x,y, 'r:+')
```

下面函数的显示效果如下：

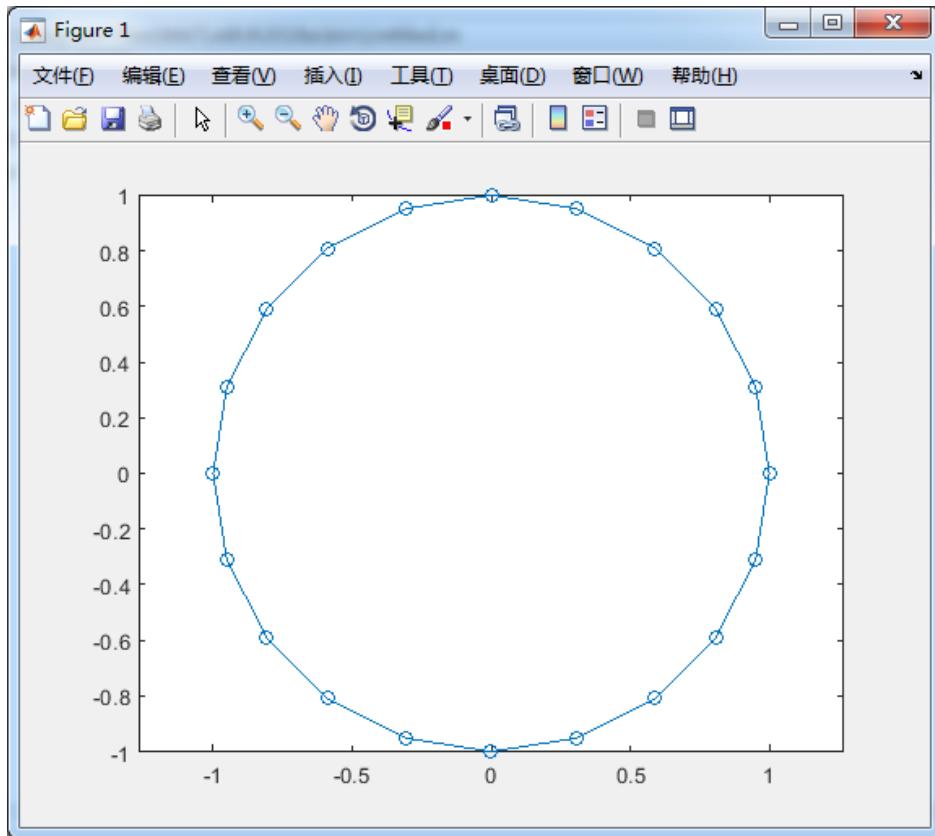


◆ 复数绘图

默认情况下函数 `plot` 只绘制数据的实部，如果是下面这种形式，实部和虚部都会进行绘制。`plot(Z)` 也就是 `plot(real(Z),imag(Z))`。下面我们在命令窗口中实现如下函数功能：

```
t = 0:pi/10:2*pi;
plot(exp(i*t), '-o')
axis equal
```

显示效果如下：

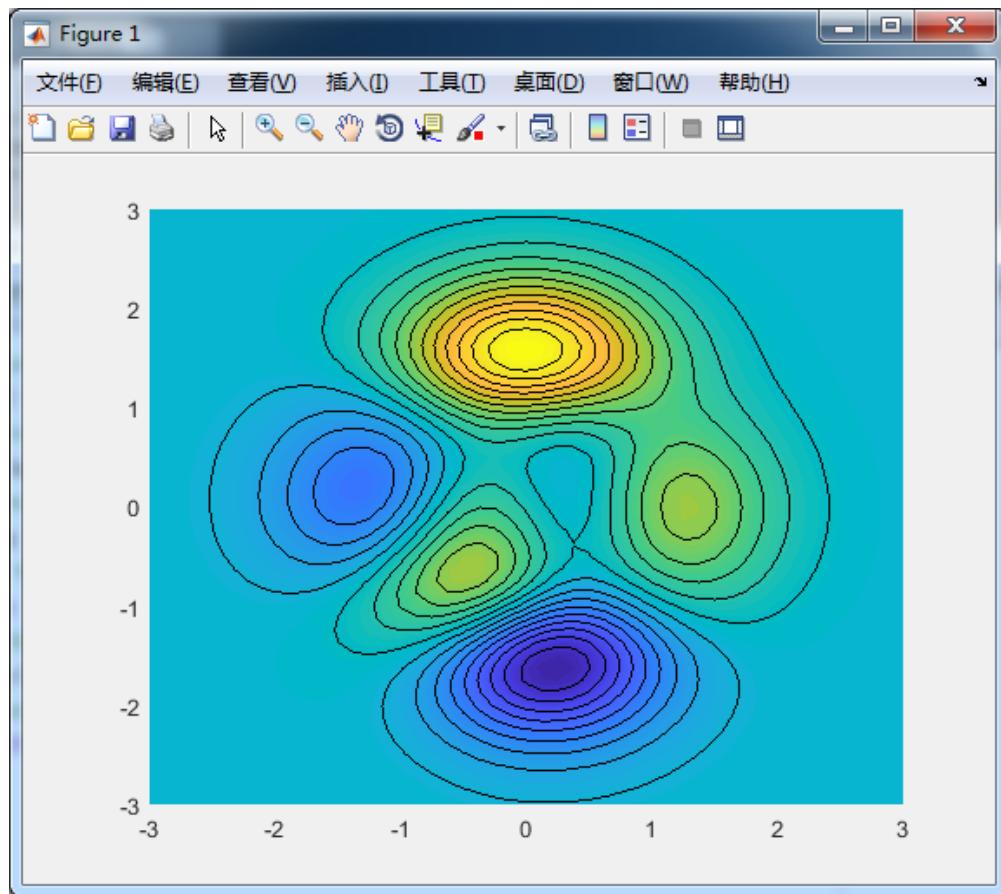


◆ 在当前的绘图中添加新的 `plot` 函数

使用函数 `hold on` 即可实现，这个函数我们在上一章节中已经使用过，作用就是在当前绘图的基础上加上一个新的绘图。

```
% Obtain data from evaluating peaks function
[x, y, z] = peaks;
% Create pseudocolor plot
pcolor(x, y, z)
% Remove edge lines a smooth colors
shading interp
% Hold the current graph
hold on
% Add the contour graph to the pcolor graph
contour(x, y, z, 20, 'k')
% Return to default
hold off
```

显示效果如下：



◆ Axis 设置

➤ 可见性设置

```
axis on    %设置可见  
axis off   %设置不可见
```

➤ 网格线设置

```
grid on    %设置可见  
grid off   %设置不可见
```

➤ 长宽比设置

```
axis square      %设置 X, Y 轴等长  
axis equal       %设置 X, Y 相同的递增。  
axis auto normal %设置自动模式。
```

➤ 设置轴界限

```
axis([xmin xmax ymin ymax])          %二维  
axis([xmin xmax ymin ymax zmin zmax]) %三维  
axis auto    %设置自动
```

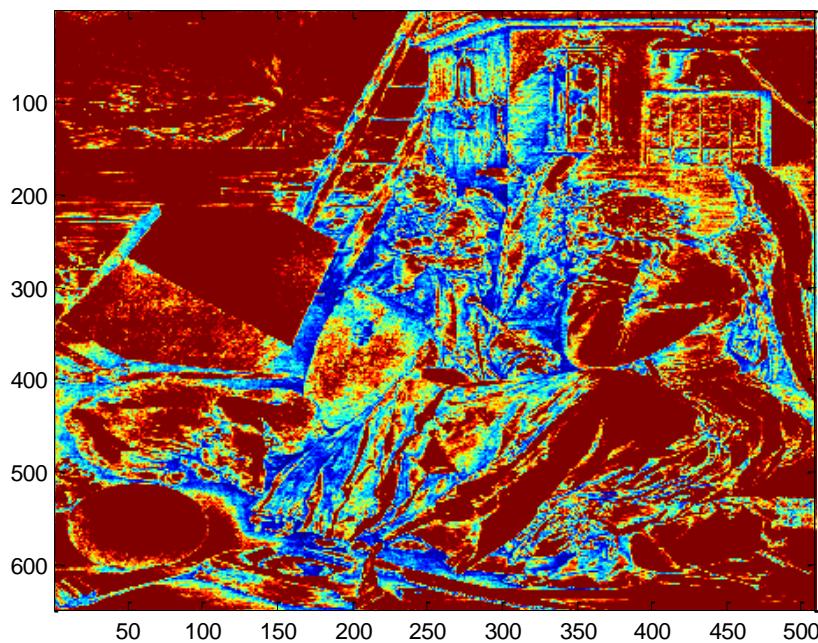
4.4.2 绘制图像数据

下面通过一个简单的实例说明一下图像数据的绘制，在命令窗口下操作即可。

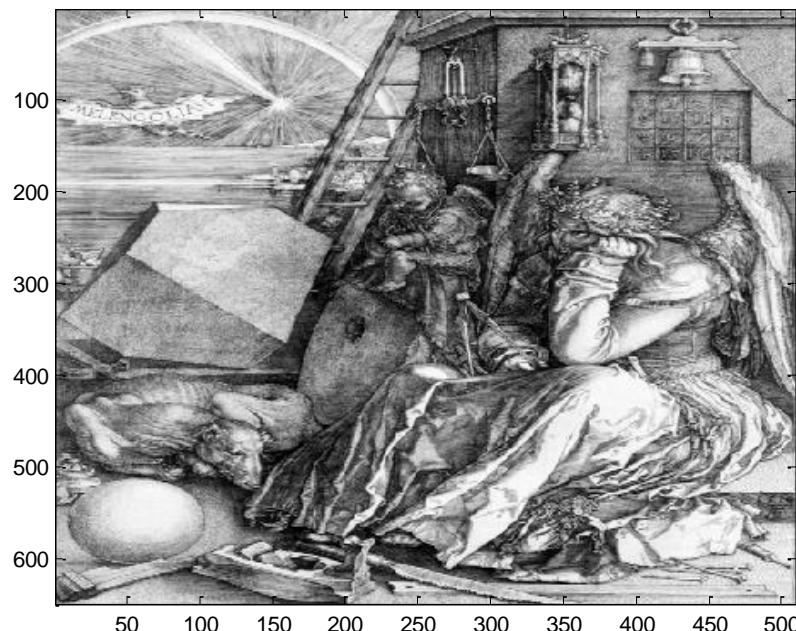
```
>> load durer  
>> whos  
Name      Size            Bytes  Class     Attributes  
X         648x509        2638656  double  
ans       648x509        2638656  double
```

```
caption      2x28          112  char
map         128x3        3072  double
```

```
>> image(X)    %显示图片
```



```
>> colormap(map)    %上色
```



```
>> axis image    %设置坐标
```



使用相同的方法，大家可以加载图片 detail 进行操作。另外用户可以使用函数 imwrite 和 imread 操作标准的 JPEG, BMP, TIFF 等类型的图片。

4.5 总结

本期主要跟大家讲解了 Matlab 的简单使用方法，需要大家多查手册，多练习。



第5章 Matlab 简易使用之常用编程语句

本期教程主要是讲解 Matlab 的一些编程语句。

5.1 初学者重要提示

5.2 Matlab 控制流

5.3 Matlab 中 help 功能的使用

5.4 总结

5.1 初学者重要提示

- ◆ 学习本章节前，务必优先学习第 4 章。
- ◆ Matlab 的编程语句类似 C，只是比 C 更加宽松。

5.2 Matlab 控制流

5.2.1 Matlab 条件控制 if, else, switch

下面我们通过三个简单的例子来说明这三个函数的使用。

◆ If 和 else 语句的使用

```
a = randi(100, 1);  
  
if a < 30  
    disp('small')  
elseif a < 80  
    disp('medium')  
else  
    disp('large')  
end
```

命令窗口输出结果如下：

```
命令行窗口  
>> Untitled  
small  
fx >>
```

◆ switch 语句的使用

```
[dayNum, dayString] = weekday(date, 'long', 'en_US');  
  
switch dayString  
    case 'Monday'  
        disp('Start of the work week')  
    case 'Tuesday'
```



```
disp('Day 2')
case 'Wednesday'
    disp('Day 3')
case 'Thursday'
    disp('Day 4')
case 'Friday'
    disp('Last day of the work week')
otherwise
    disp('Weekend!')
end
```

命令窗口输出结果如下：

```
命令行窗口
>> Untitled
Start of the work week
fx >>
```

在这里顺便介绍一个类似于 C 语言中 `scanf` 的函数 `input` 并配合上面的 `if else` 实现一个小功能：

```
yourNumber = input('Enter a number:');
```

```
if yourNumber < 0
    disp('Negative')
elseif yourNumber > 0
    disp('Positive')
else
    disp('Zero')
end
```

运行上面代码后，我们在命令窗口输入数字 22，输出结果如下：

```
命令行窗口
>> Untitled
Enter a number: 6
Positive
fx >>
```

5.2.2 Matlab 循环控制 `for`, `while`, `continue`, `break`

这里我们也通过几个简单的例子来说明这几个函数的使用。

◆ `for` 语句的使用

```
for n = 3:32
    r(n) = rank(magic(n));
end
r
```

命令窗口输出结果如下：



```
命令行窗口
>> Untitled

r =

1 至 18 列

0 0 3 3 5 5 7 3 9 7 11 3 13 9 15 3 17 11

19 至 32 列

19 3 21 13 23 3 25 15 27 3 29 17 31 3

fx >>
```

◆ while 语句的使用

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
    x = (a+b)/2;
    fx = x^3-2*x-5;
    if sign(fx) == sign(fa)
        a = x; fa = fx;
    else
        b = x; fb = fx;
    end
end
x
```

命令窗口输出结果如下：

```
命令行窗口
>> Untitled

x =

2.0946

fx >> |
```

◆ continue 语句的使用

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'% ',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
fprintf('%d lines\n',count);
fclose(fid);
```

命令窗口的输出结果如下：



```
命令行窗口
>> Untitled
31 lines
fx >>
```

◆ break 语句的使用

```
a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*b
    x = (a+b)/2;
    fx = x^3-2*x-5;
    if fx == 0
        break
    elseif sign(fx) == sign(fa)
        a = x; fa = fx;
    else
        b = x; fb = fx;
    end
end
x
```

命令窗口输出结果如下：

```
命令行窗口
>> Untitled
x =
2.0946
fx >> |
```

5.2.3 Matlab 矢量化

对于 matlab 而言，要想加快算法的执行速度可以通过算法的矢量化来实现，比如要实现如下的功能。

```
x = .01;
for k = 1:1001
    y(k) = log10(x);
    x = x + .01;
end
```

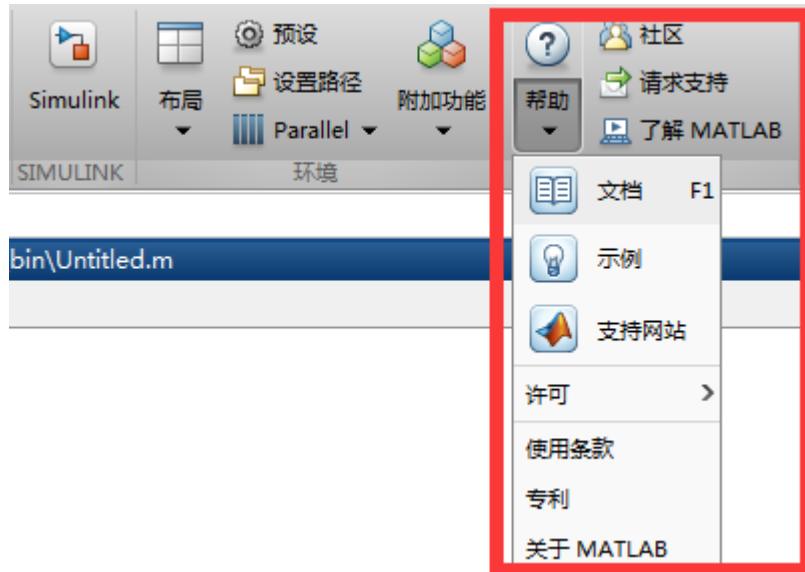
但是我们矢量化后，将更加方便和容易实现。

```
x = .01:.01:10;
y = log10(x);
```

但是有一点大家要特别注意，不是什么程序矢量化都能加快执行速度，要视具体情况而定。

5.3 Matlab 中 help 功能的使用

关于 matlab 入门方面的东西就跟大家讲这么多，基本上有这些基础就够了，后面遇到什么问题在网上查找资料即可。也可以查看 matlab 本身的 help 帮助文档。点击[这里](https://ww2.mathworks.cn/help/)就可以查看，或者直接访问地址：<https://ww2.mathworks.cn/help/>。



如果有不懂的函数，可以直接在命令窗口输入 help 再加上函数即可，比如输入：

```
命令行窗口
>> help max
max - 数组中的最大元素

此 MATLAB 函数 返回 A 的最大元素。如果 A 为向量，则 max(A) 返回 A 的最大元素。如果 A 为矩阵，则 max(A)
是包含每一列的最大值的行向量。如果 A 是多维数组，则 max(A) 沿大小不等于 1 的第一个数组维度计算，并将这些元素视为向量。此维度的大小将变为
1，而所有其他维度的大小保持不变。如果 A 是第一个维度长度为零的空数组，则 max(A) 返回与 A 大小相同的空数组。

M = max(A)
M = max(A, [], dim)
M = max(A, [], nanflag)
M = max(A, [], dim, nanflag)
[M, I] = max(___)
C = max(A, B)
C = max(A, B, nanflag)

另请参阅 bounds, mean, median, min, sort

max 的参考页
名为 max 的其他函数

fx >>
```

5.4 总结

Matlab 方面的教程就跟大家讲这么多，后面需要那方面知识的时候，我们再具体的补充。学会这些基本的操作就可以入门了。永远要记住，Matlab 只是个工具，我们只需把它当个工具来用，没有必要花大量的时间去研究，入门后用什么学什么即可。



第6章 ARM DSP 源码和库移植方法 (MDK5 的 AC5 和 AC6)

本期教程主要讲解 ARM 官方 DSP 源码和库的移植以及一些相关知识的介绍。

6.1 初学者重要提示

6.2 DSP 库的下载和说明

6.3 DSP 库版本的区别

6.4 DSP 库的几个重要的预定义宏含义

6.5 使用 MDK 的 AC6 编译器优势

6.6 DSP 库在 MDK 上的移植 (AC5 源码移植方式)

6.7 DSP 库在 MDK 上的移植 (AC5 库移植方式)

6.8 DSP 库在 MDK 上的移植 (AC6 源码移植方式)

6.9 DSP 库在 MDK 上的移植 (AC6 库移植方式)

6.10 升级到最新版 DSP 库的方法

6.11 简易 DSP 库函数验证

6.12 总结

6.1 初学者重要提示

- ◆ MDK 请使用 5.26 及其以上版本，CMSIS 软件包请使用 5.6.0 及其以上版本。
- ◆ MDK 的工程创建，下载和调试方法，在 V7 用户手册有详细说明：
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=93255>。
- ◆ 鉴于 MDK 的 AC6 (ARM Compiler 6.X) 编译器在浮点处理上的强劲性能，每个例子将必做一个 AC6 版，而且 ARM 编译好的 DSP 库也开始直接采用 AC6。
- ◆ **MDK AC6 有两个地方在使用的时候要注意：**
 - 工程目录切记不要有中文路径，而且不要太长，否则会导致无法使用 go to def 以及调试的时候不正常。
 - MDK AC6 工程模板的汉字编码问题，在本章 6.8 小节有详细说明。



6.2 DSP 库的下载和说明

下面详细的给大家讲解一下官方 DSP 库的移植。

6.2.1 DSP 库的下载

DSP 库是包含在 CMSIS 软件包 (Cortex Microcontroller Software Interface Standard) 里面，所以下载 DSP 库也就是下载 CMSIS 软件包。这里提供三个可以下载的地方：

- ◆ 方式一：STM32CubeH7 软件包里面。

每个版本的 Cube 软件包都会携带 CMSIS 文件夹，只是版本比较老，不推荐。即使是最新的 CubeH7

软件包，包含的 CMSIS 软件包版本也有点低。

- ◆ 方式二：MDK 安装目录（下面是 5.6.0 版本的路径）。

大家安装了新版 MDK 后，CMSIS 软件包会存在于路径：ARM\PACK\ARM\CMSIS\5.6.0\CMSIS。

如果有更新的版本，推荐大家使用最新版本，MDK 的软件包下载地址：

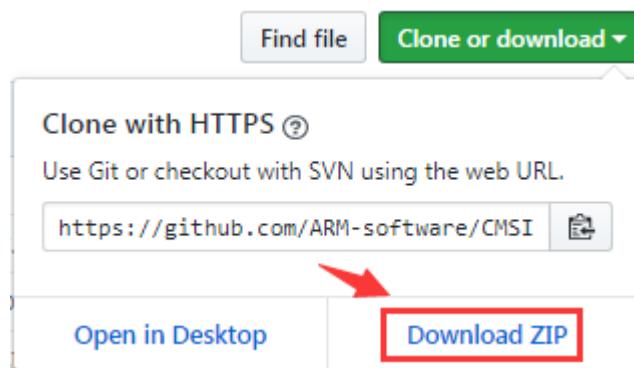
<http://www.keil.com/dd2/Pack/>。

The screenshot shows a list of software packages available for download:

- ARM mbed Client for Cortex-M devices (1.1.0)
- ARM mbed Cryptographic and SSL/TLS library for Cortex-M devices (1.6.0)
- Bundle of FreeRTOS for Cortex-M and Cortex-A (10.2.0)
- CMSIS (Cortex Microcontroller Software Interface Standard) (5.6.0) - This item is highlighted with a red border.
- CMSIS Drivers for external devices (2.4.1)
- CMSIS-Driver Validation (1.2.0)

- ◆ 方式三：GitHub。

通过 GitHub 获取也比较方便，地址：https://github.com/ARM-software/CMSIS_5。点击右上角就可以下载 CMSIS 软件包了。

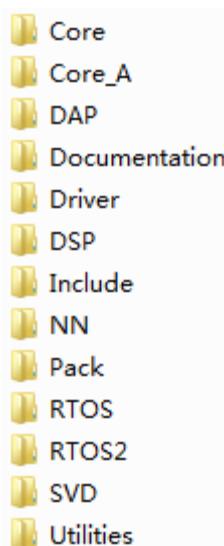




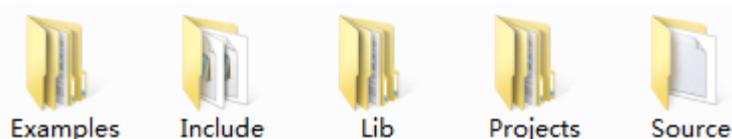
当然，也可以在 ARM 官网下载，只是这两年 ARM 官网升级得非常频繁，通过检索功能找资料非常麻烦。所以不推荐大家到 ARM 官网下载资料了。

6.2.2 DSP 库的说明

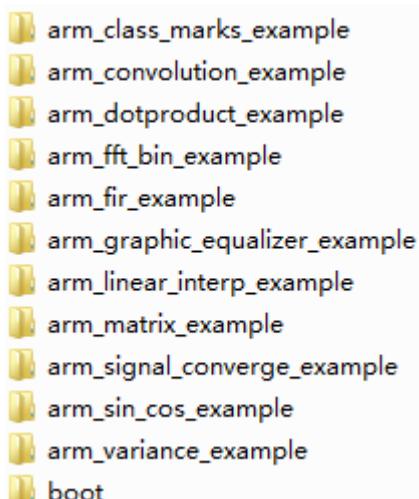
这里我们以 CMSIS V5.6.0 为标准进行移植。打开固件库里面的 CMSIS 文件，可以看到如下几个文件：



其中 DSP 文件夹是我们需要的：



Examples 文件夹中的文件如下，主要是提供了一些例子：

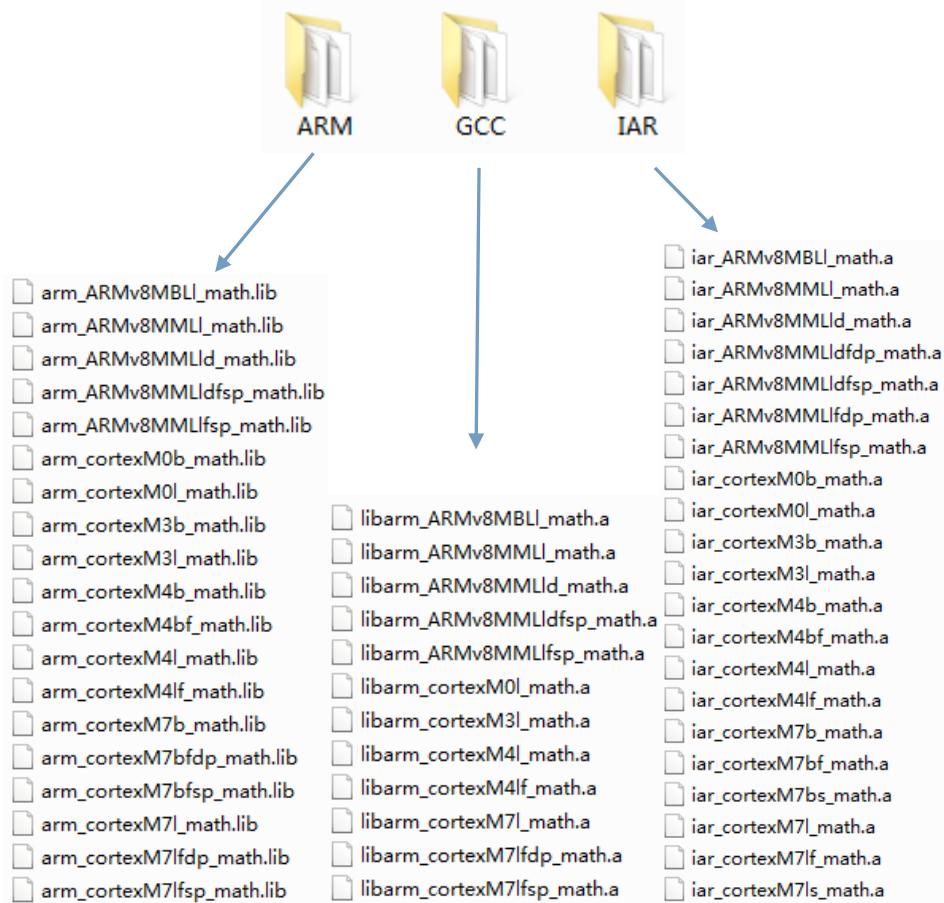


Include 文件夹里面是 DSP 库的头文件：



arm_common_tables.h
 arm_const_structs.h
 arm_math.h

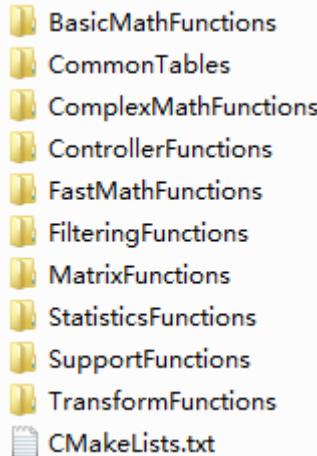
Lib 文件夹里面是 MDK (ARM) , IAR 和 CGG 版库文件:



Projects 文件夹里面的文件如下，提供了三个版本的工程模板，每个模板里面都是把所有源码文件添加了进来：

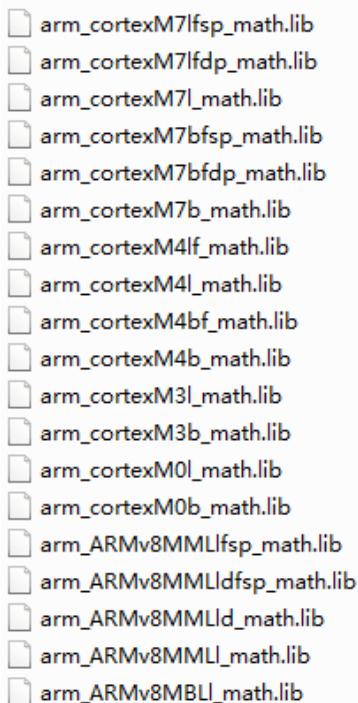


Source 文件夹中的文件如下，这个是 DSP 的源码文件：



6.3 DSP 库版本的区别

MDK 版本的 DSP 库如下：



- ◆ arm_cortexM7lfdp_math.lib

Cortex-M7 内核，l 表示小端格式，f 表示带 FPU 单元，dp 表示 Double Precision 双精度浮点。

- ◆ arm_cortexM7bfdp_math.lib

Cortex-M7 内核，b 表示大端格式，f 表示带 FPU 单元，dp 表示 Double Precision 双精度浮点。

- ◆ arm_cortexM7lfsp_math.lib

Cortex-M7 内核，l 表示小端格式，f 表示带 FPU 单元，sp 表示 Single Precision 单精度浮点。

- ◆ arm_cortexM7bfsp_math.lib



Cortex-M7 内核，b 表示大端格式，f 表示带 FPU 单元，sp 表示 Single Precision 单精度浮点。

- ◆ arm_cortexM7l_math.lib

Cortex-M7 内核，l 表示小端格式。

- ◆ arm_cortexM7b_math.lib

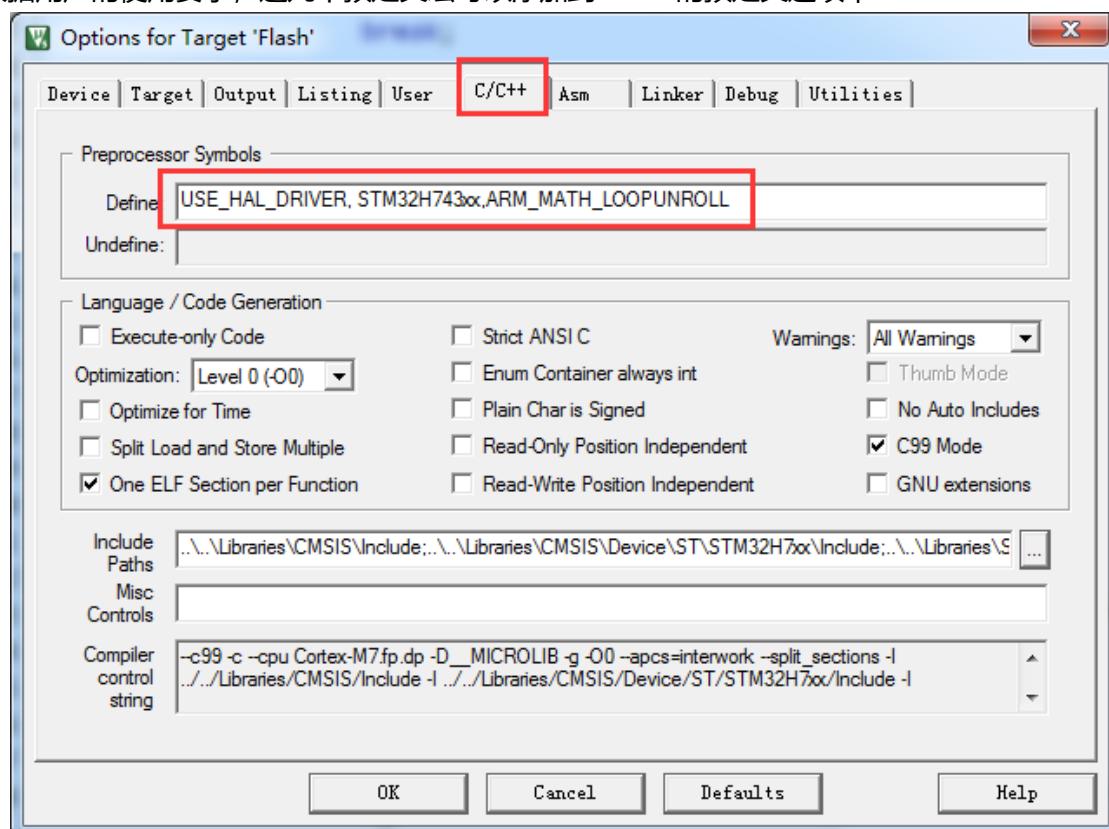
Cortex-M7 内核，b 表示大端格式。

STM32H7 是 M7 内核，双精度浮点，一般使用小端格式，所以我们选择库

arm_cortexM7lfdp_math.lib

6.4 DSP 库的几个重要的预定义宏含义

根据用户的使用要求，这几个预定义宏可以添加到 MDK 的预定义选项中：



这里将这几个预定义宏做个介绍：

- ◆ ARM_MATH_BIG_ENDIAN:

大端格式。

- ◆ ARM_MATH_MATRIX_CHECK:

检测矩阵的输入输出大小

- ◆ ARM_MATH_NEON:

ARM_MATH_NEON EXPERIMENTAL:



这两个暂时用不到，因为 M0, M3, M4 和 M7 内核不支持 NEON 指令，需要等待升级到 ARMv8.1-M 架构。

◆ ARM_MATH_ROUNDING:

主要用在浮点数转 Q32, Q15 和 Q7 时，类似四舍五入的处理上，其它函数没用到。

◆ ARM_MATH_LOOPUNROLL:

用于 4 个为一组的小批量处理上，加快执行速度。

通过下面的求绝对值函数，可以方便的看出区别：

```
void arm_abs_f32(
    const float32_t * pSrc,
    float32_t * pDst,
    uint32_t blockSize)
{
    uint32_t blkCnt; /* Loop counter */

#if defined(ARM_MATH_NEON)
    float32x4_t vec1;
    float32x4_t res;

    /* Compute 4 outputs at a time */
    blkCnt = blockSize >> 2U;

    while (blkCnt > 0U)
    {
        /* C = |A| */

        /* Calculate absolute values and then store the results in the destination buffer. */
        vec1 = vld1q_f32(pSrc);
        res = vabsq_f32(vec1);
        vst1q_f32(pDst, res);

        /* Increment pointers */
        pSrc += 4;
        pDst += 4;

        /* Decrement the loop counter */
        blkCnt--;
    }

    /* Tail */
    blkCnt = blockSize & 0x3;
}

#else
#if defined (ARM_MATH_LOOPUNROLL)

    /* Loop unrolling: Compute 4 outputs at a time */
    blkCnt = blockSize >> 2U;

    while (blkCnt > 0U)
    {
        /* C = |A| */

```



```
/* Calculate absolute and store result in destination buffer. */
*pDst++ = fabsf(*pSrc++) ;

*pDst++ = fabsf(*pSrc++) ;
*pDst++ = fabsf(*pSrc++) ;
*pDst++ = fabsf(*pSrc++) ;

/* Decrement loop counter */
blkCnt--;
}

/* Loop unrolling: Compute remaining outputs */
blkCnt = blockSize % 0x4U;

#else

/* Initialize blkCnt with number of samples */
blkCnt = blockSize;

#endif /* #if defined (ARM_MATH_LOOPUNROLL) */
#endif /* #if defined(ARM_MATH_NEON) */

while (blkCnt > 0U)
{
    /* C = |A| */

    /* Calculate absolute and store result in destination buffer. */
    *pDst++ = fabsf(*pSrc++) ;

    /* Decrement loop counter */
    blkCnt--;
}
}
```

6.5 使用 MDK 的 AC6 编译器优势

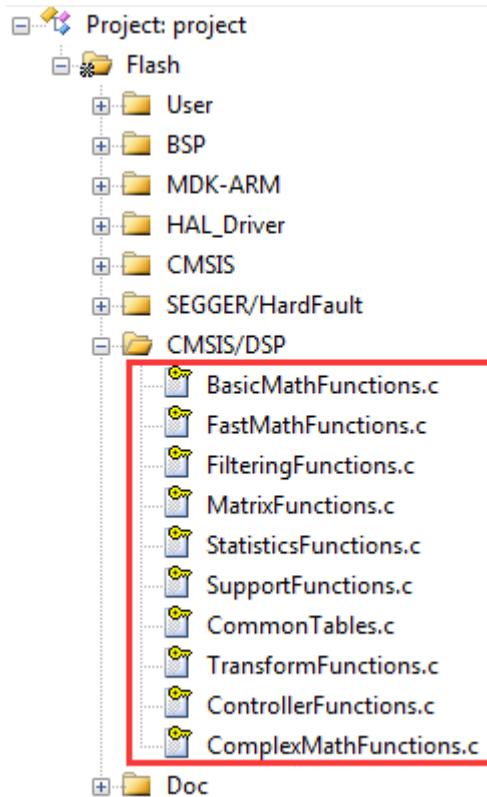
涉及到的知识点比较多，专门将其整理到[附件章节 E](#)。

6.6 DSP 库在 MDK 上的移植（AC5 源码移植方式）

下面我们讲解下如何在 MDK 上面移植 DSP 库源码，DSP 库的移植相对比较容易。

6.6.1 第一步：建立 MDK 工程并添加 DSP 库

为了方便起见，我们这里不再专门建立一个 MDK 工程了，直接以 V7 开发板中的例子：V7-001_跑马灯例程为模板进行添加即可。打开这个实例并在左侧添加分组 CMSIS/DSP：

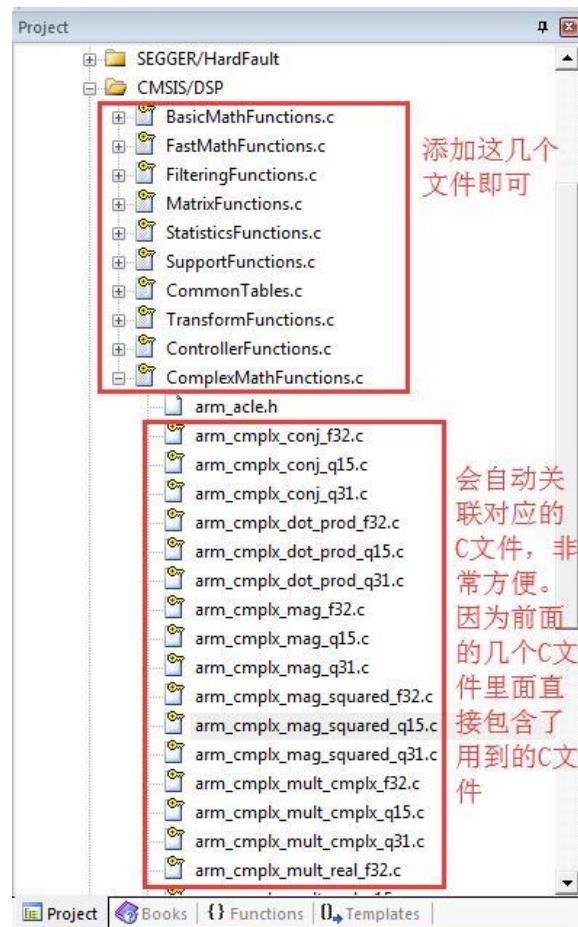


我们这里不需要添加每个 C 文件源码，仅需添加包含这些 C 文件的汇总文件，比如 BasicMathFunctions.c 文件里面包含的 C 文件就是：

```
#include "arm_abs_f32.c"
#include "arm_abs_q15.c"
#include "arm_abs_q31.c"
#include "arm_abs_q7.c"
#include "arm_add_f32.c"
#include "arm_add_q15.c"
#include "arm_add_q31.c"
#include "arm_add_q7.c"
#include "arm_dot_prod_f32.c"
#include "arm_dot_prod_q15.c"
#include "arm_dot_prod_q31.c"
#include "arm_dot_prod_q7.c"
#include "arm_mult_f32.c"
#include "arm_mult_q15.c"
#include "arm_mult_q31.c"
#include "arm_mult_q7.c"
#include "arm_negate_f32.c"
#include "arm_negate_q15.c"
#include "arm_negate_q31.c"
#include "arm_negate_q7.c"
#include "arm_offset_f32.c"
#include "arm_offset_q15.c"
#include "arm_offset_q31.c"
#include "arm_offset_q7.c"
#include "arm_scale_f32.c"
#include "arm_scale_q15.c"
#include "arm_scale_q31.c"
#include "arm_scale_q7.c"
```

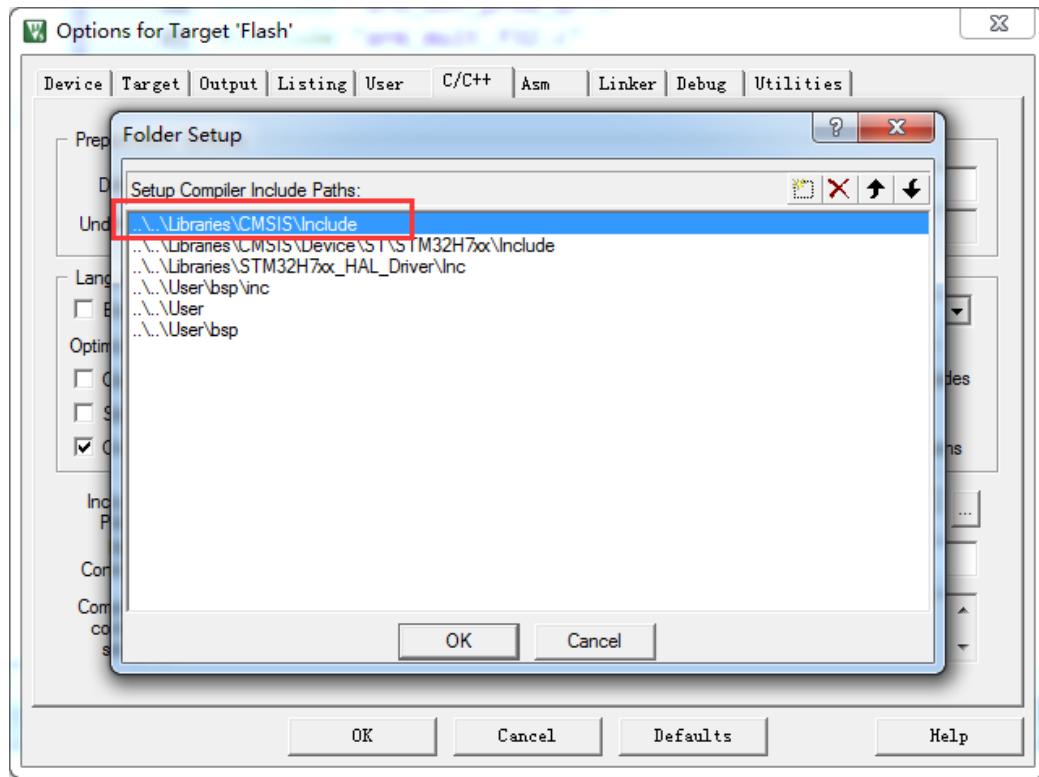
```
#include "arm_shift_q15.c"
#include "arm_shift_q31.c"
#include "arm_shift_q7.c"
#include "arm_sub_f32.c"
#include "arm_sub_q15.c"
#include "arm_sub_q31.c"
#include "arm_sub_q7.c"
```

这样一来，MDK 编译后会自动关联，查看源码非常方便：



6.6.2 第二步：添加头文件路径

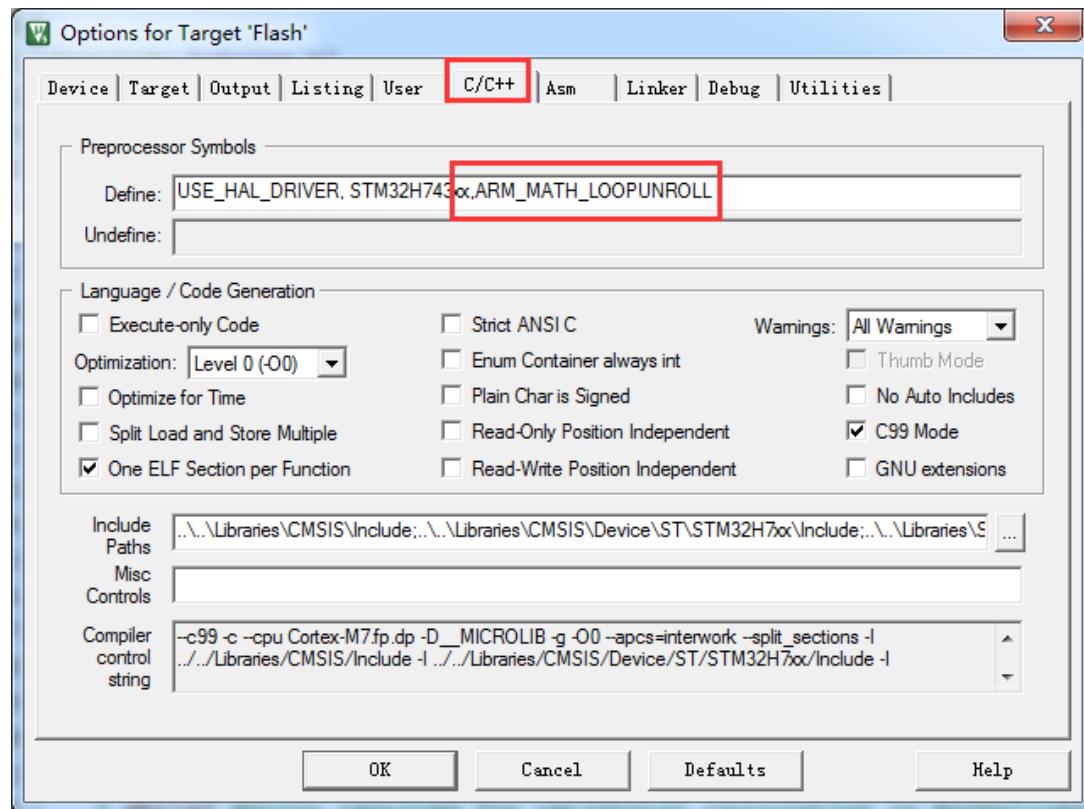
添加 DSP 所需的头文件路径，这个头文件路径是已经在模板工程中添加好的，这里只是跟大家强调一下：



这里要注意一点，为什么直接添加路径 Libraries\CMSIS\Include 里面的头文件即可，而没有添加 Libraries\CMSIS\DSP\Include，这是因为路径 Libraries\CMSIS\Include 里面已经包含了 DSP 库的头文件。

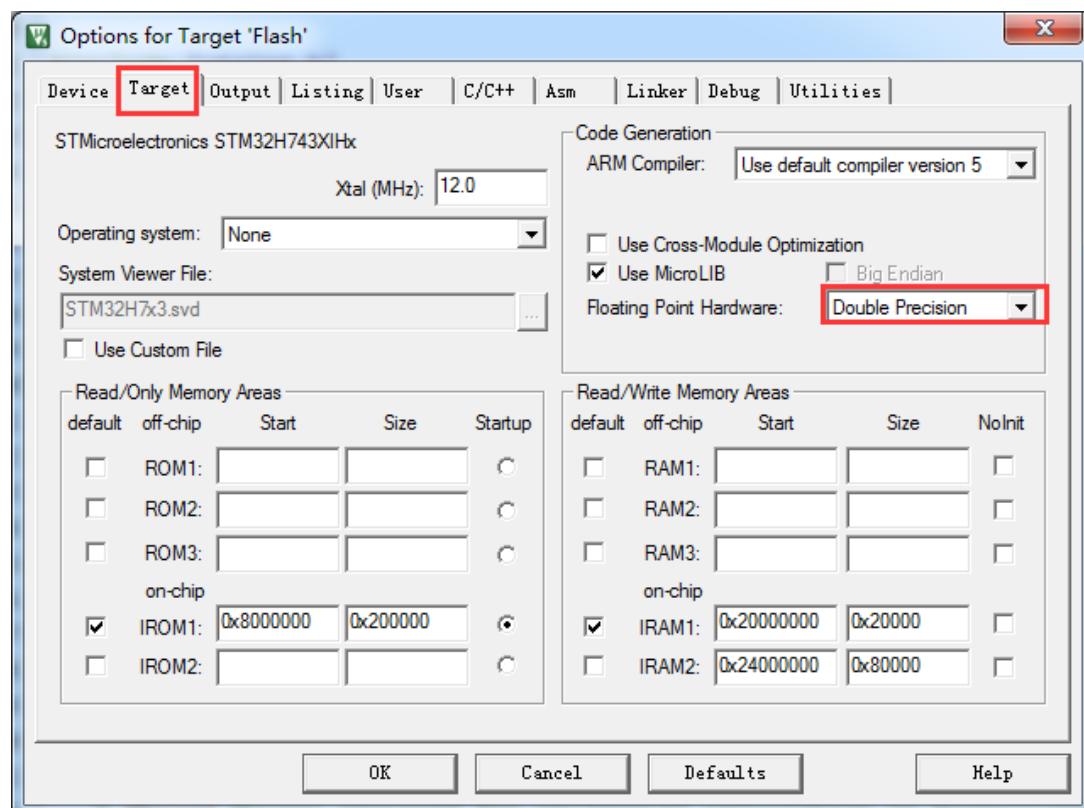
6.6.3 第三步：添加宏定义

我们这里仅使能一个宏定义 ARM_MATH_LOOPUNROLL：



6.6.4 第四步：开启 FPU

需要客户通过 MDK 开启 FPU，由于 STM32H7 支持双精度浮点，这里要开启 Double Precision。

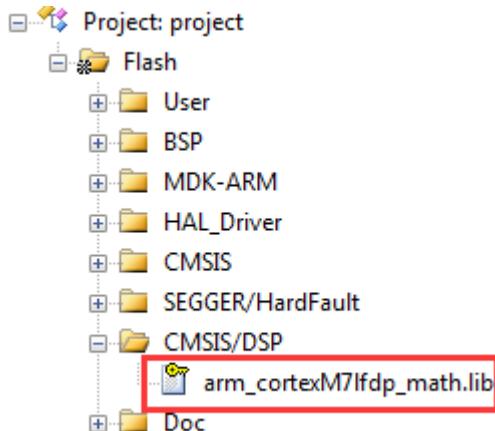


6.6.5 第五步：添加头文件 arm_math.h

用到 DSP 库函数的文件得添加`#include "arm_math.h"`就可以调用 DSP 库的 API 了。至此就完成了 DSP 库的移植。

6.7 DSP 库在 MDK 上的移植（AC5 库移植方式）

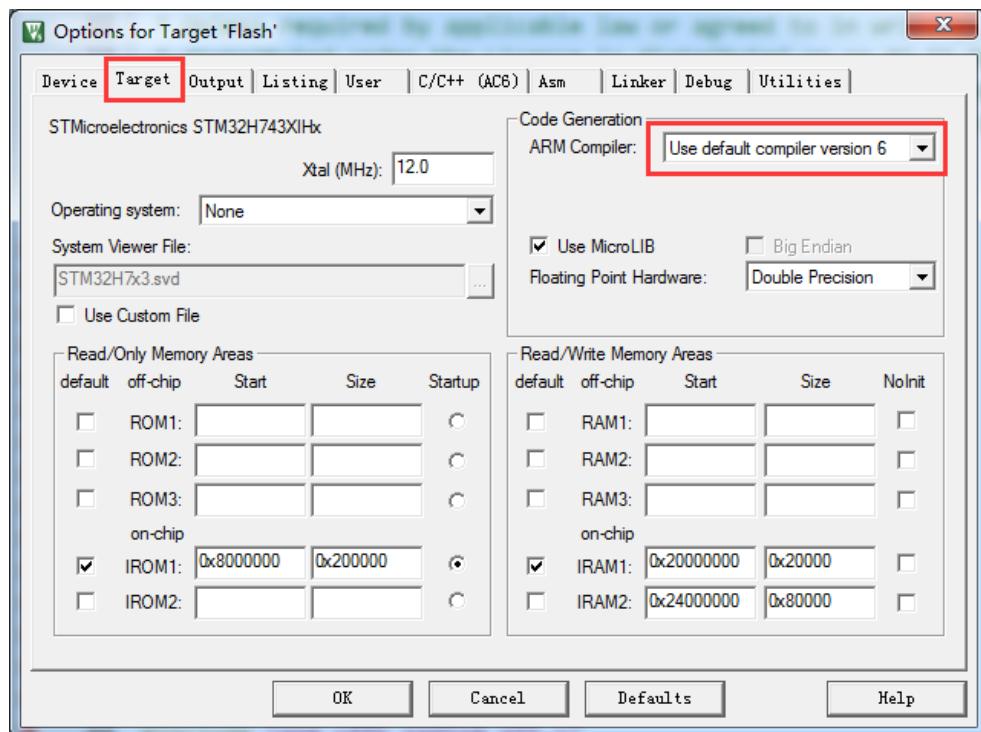
移植方法与本章 6.5 小节的相同，仅第 1 步不同，将源码的添加修改为库添加：



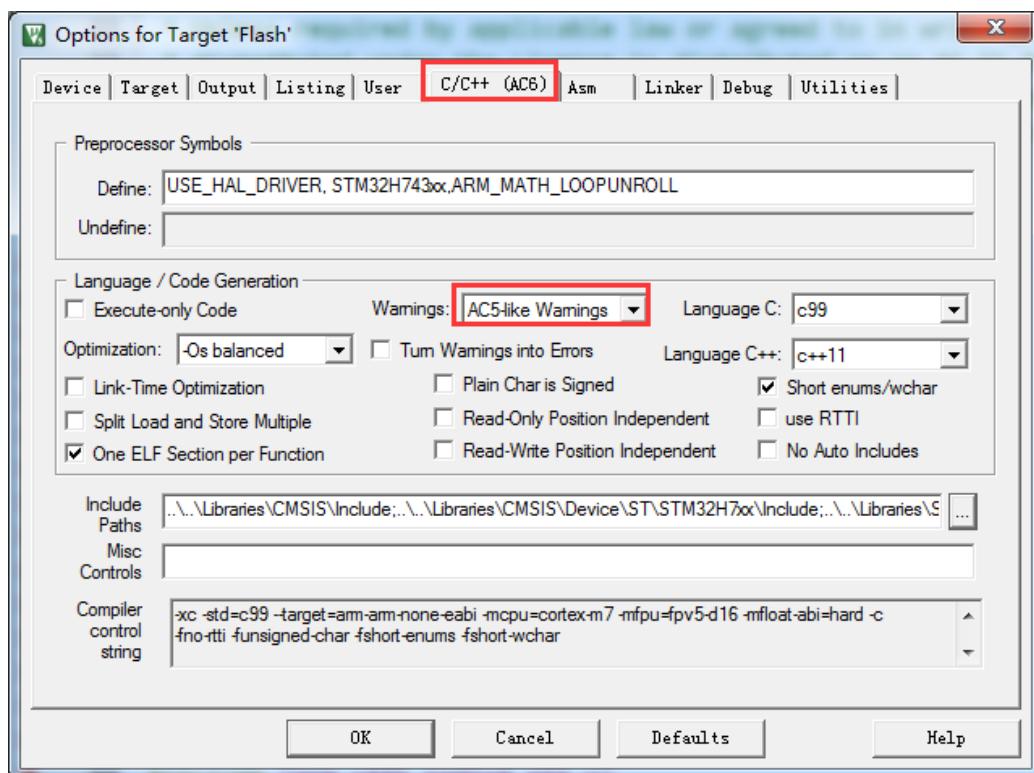
6.8 DSP 库在 MDK 上的移植（AC6 源码移植方式）

AC6 的 DSP 源码移植与本章 6.5 小节里面的 AC5 移植完全相同，没有区别。但 AC5 和 AC6 工程上有三处区别，这里着重指出下：

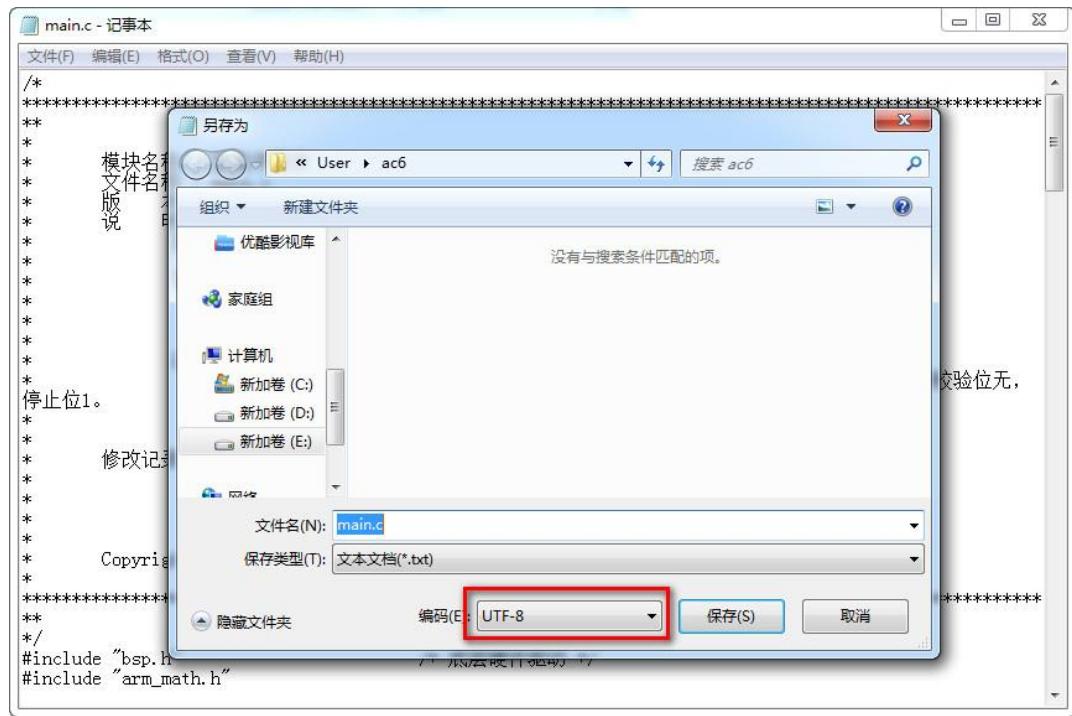
- ◆ 第 1 处，采用 AC6 编译器：



◆ 第 2 处，警告类型选择 AC5-like:

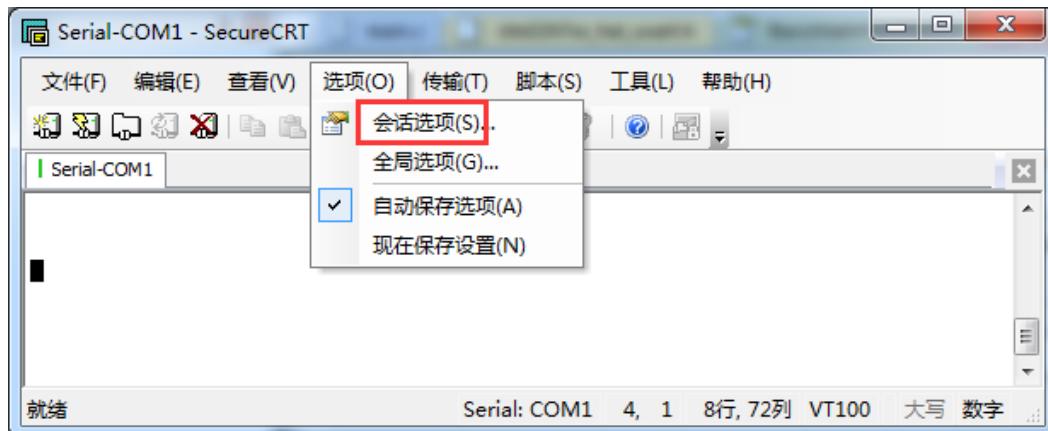


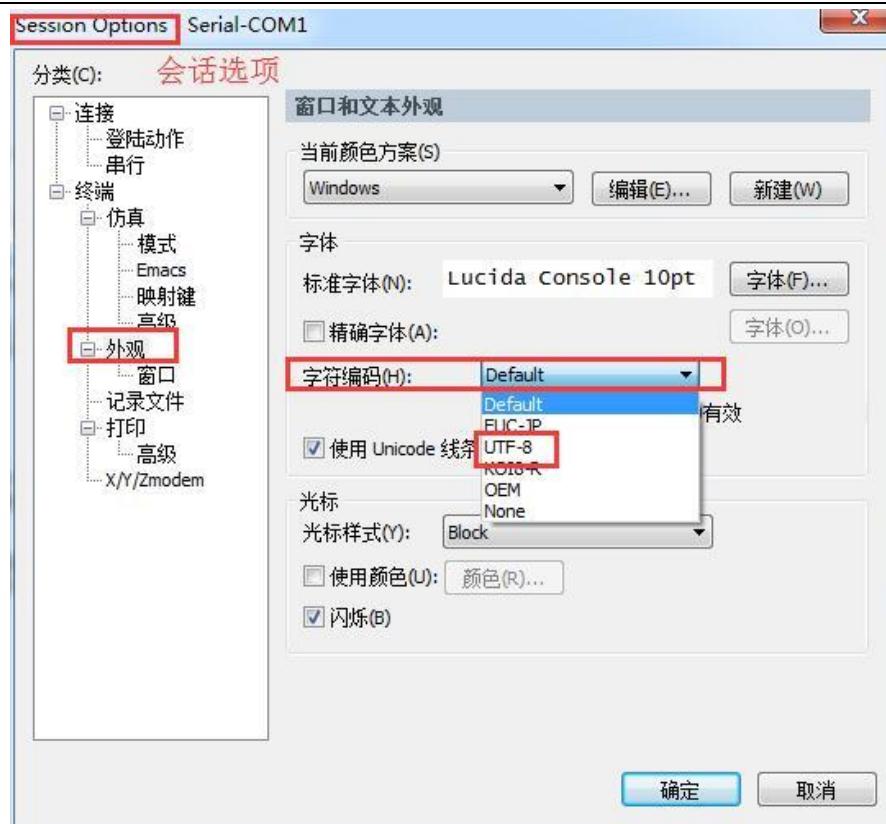
◆ 第 3 处，MDK 的 AC6 工程代码如果有源文件是 GBK 编码，而且使用汉字，MDK 编译时会报错，需要用记事本打开使用汉字的源码文件，另存为 UTF-8。比如 main.c 文件的串口打印函数 printf 用到了汉字，那么就需要做如下修改：



然后再重新编译就不会报错了。同时，串口打印时，使用的串口助手要支持 UTF-8，推荐用 SecureCRT，
下载地址：<http://www.armbbs.cn/forum.php?mod=viewthread&tid=91718>。

设置如下：





6.9 DSP 库在 MDK 上的移植 (AC6 库移植方式)

AC6 的 DSP 库移植与本章 6.6 小节里面的 AC5 移植完全相同，没有区别。不过要注意 6.8 小节中所讲解的问题即可。

6.10 升级到最新版 DSP 库的方法

由于 CMSIS 软件包是实时更新的，这里提供一种升级的简单办法，按照本章 6.1 小节的说明下载到最新版 CMSIS 软件包，然后直接覆盖 DSP 工程里面的 CMSIS 文件夹即可。

6.11 简易 DSP 库函数验证

这里我们主要运行 `arm_abs_f32`, `arm_abs_q31`, `arm_abs_q15` 这三个函数，以此来验证我们移植的 DSP 库是否正确。

配套例子：

本章配套了如下两个例子：

- ◆ V7-200_DSP 程序模板 (源码方式)
- ◆ V7-201_DSP 程序模板 (库方式)

每个例子都配套了 MDK 的 AC5 和 AC6 两个版本的工程。

实验目的：

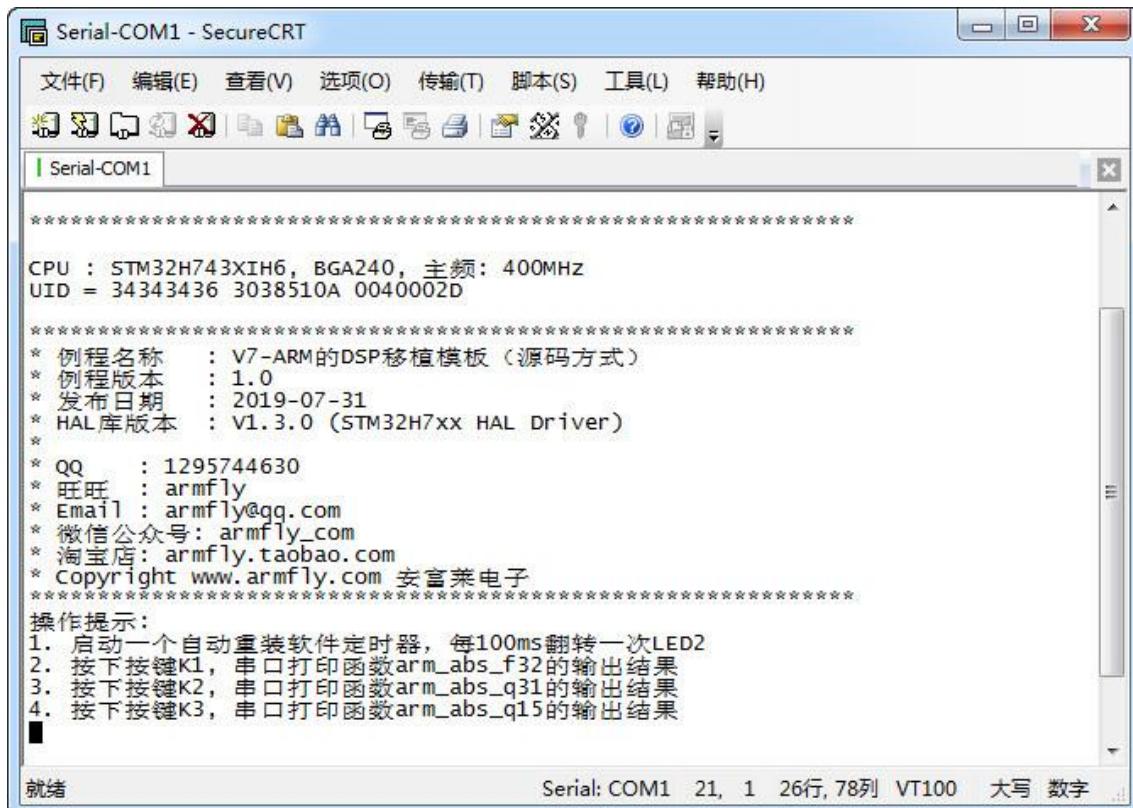
- ## 1. 学习官方 DSP 库的移植

实验内容：

1. 按下按键 K1, 串口打印函数 arm_abs_f32 的输出结果
 2. 按下按键 K2, 串口打印函数 arm_abs_q31 的输出结果
 3. 按下按键 K3, 串口打印函数 arm_abs_q15 的输出结果

实验现象：

通过串口上位机软件 SecureCRT 看打印信息现象如下（分别按几次 K1, K2, K3）。如果编译的是 MDK 的 AC6 工程，特别要注意本章 6.7 小节所说的问题。



程序设计：

程序的设计也比较简单，通过按下不同的按键从而打印不同的 DSP 库函数执行结果，主程序如下：

```
#include "bsp.h"          /* 底层硬件驱动 */  
#include "arm_math.h"
```

```
/* 定义例程名和例程发布日期 */
#define EXAMPLE_NAME      "V7-ARM的DSP移植模板（源码方式）"
#define EXAMPLE_DATE       "2019-07-31"
#define DEMO_VER          "1.0"

static void PrintfLogo(void);
static void PrintfHelp(void);

/*
*****
```



```
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    float32_t pSrc;
    float32_t pDst;
    q31_t pSrc1;
    q31_t pDst1;
    q15_t pSrc2;
    q15_t pDst2;

    bsp_Init();           /* 硬件初始化 */

    PrintfLogo(); /* 打印例程名称和版本等信息 */
    PrintfHelp(); /* 打印操作提示 */

    bsp_StartAutoTimer(0, 100); /* 启动1个100ms的自动重装的定时器 */

    /* 主程序大循环 */
    while (1)
    {
        /* CPU 空闲时执行的函数, 在 bsp.c */
        bsp_Idle();

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            /* 翻转 LED2 的状态 */
            bsp_LedToggle(2);
        }

        /* 处理按键事件 */
        ucKeyCode = bsp_GetKey();
        if (ucKeyCode > 0)
        {
            /* 有键按下 */
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:          /* K1键按下 */
                    pSrc -= 1.23f;
                    arm_abs_f32(&pSrc, &pDst, 1);
                    printf("pDst = %f\r\n", pDst);
                    break;

                case KEY_DOWN_K2:          /* K2键按下 */
                    pSrc1 -= 1;
                    arm_abs_q31(&pSrc1, &pDst1, 1);
                    printf("pDst1 = %d\r\n", pDst1);
                    break;

                case KEY_DOWN_K3:          /* K3键按下 */
                    pSrc2 -= 1;
                    arm_abs_q15(&pSrc2, &pDst2, 1);
                    printf("pDst2 = %d\r\n", pDst2);
                    break;

                default:
                    break;
            }
        }
    }
}
```

{
}
}

6.12 总结

本期教程主要跟大家介绍了官方 DSP 库的移植，相对来说移植也比较简单，建议初学的同学按照这个步骤移植一遍。



第7章 ARM DSP 源码和库移植方法 (IAR8)

本期教程主要讲解 ARM 官方 DSP 源码和库的移植以及一些相关知识的介绍。

7.1 初学者重要提示

- 7.2 DSP 库的下载和说明
- 7.3 DSP 库版本的区别
- 7.4 DSP 库的几个重要的预定义宏含义
- 7.5 DSP 库在 IAR 上的移植 (源码移植方式)
- 7.6 DSP 库在 IAR 上的移植 (库移植方式)
- 7.7 升级到最新版 DSP 库方法
- 7.8 简易 DSP 库函数验证
- 7.9 总结

7.1 初学者重要提示

- ◆ IAR 请使用 8.30 及其以上版本，CMSIS 请使用 5.6.0 及其以上版本。
- ◆ IAR 的工程创建，下载和调试方法，在 V7 用户手册有详细说明：
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=93255>。

7.2 DSP 库的下载和说明

下面详细的给大家讲解一下官方 DSP 库的移植。

7.2.1 DSP 库的下载

DSP 库是包含在 CMSIS 软件包 (Cortex Microcontroller Software Interface Standard) 里面，所以下载 DSP 库也就是下载 CMSIS 软件包。这里提供三个可以下载的地方：

- ◆ 方式一：STM32CubeH7 软件包里面。

每个版本的 Cube 软件包都会携带 CMSIS 文件夹，只是版本比较老，不推荐。即使是最新的 CubeH7

软件包，包含的 CMSIS 软件包版本也有点低。

- ◆ 方式二：MDK 安装目录 (下面是 5.6.0 版本的路径)。

大家安装了新版 MDK 后，CMSIS 软件包会存在于路径：ARM\PACK\ARM\CMSIS\5.6.0\CMSIS。



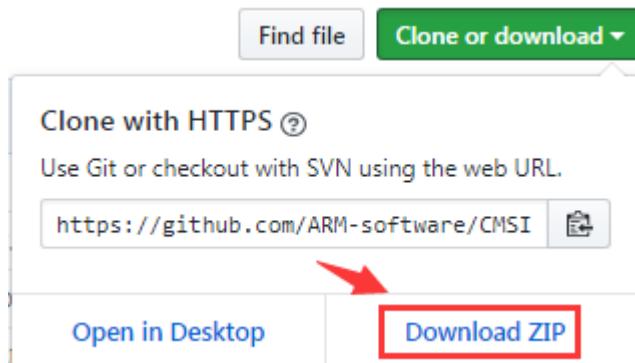
如果有更新的版本，推荐大家使用最新版本，MDK 的软包下载地址：

<http://www.keil.com/dd2/Pack/>。

➤ ARM mbed Client for Cortex-M devices	1.1.0	
➤ ARM mbed Cryptographic and SSL/TLS library for Cortex-M devices	1.6.0	
➤ Bundle of FreeRTOS for Cortex-M and Cortex-A	10.2.0	
➤ CMSIS (Cortex Microcontroller Software Interface Standard)	BSP DFP 5.6.0	
➤ CMSIS Drivers for external devices	2.4.1	
➤ CMSIS-Driver Validation	1.2.0	

◆ 方式三：GitHub。

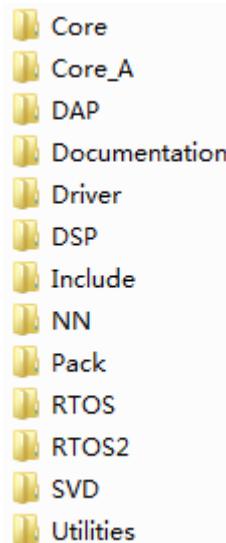
通过 GitHub 获取也比较方便，地址：https://github.com/ARM-software/CMSIS_5。点击右上角就可以下载 CMSIS 软件包了。



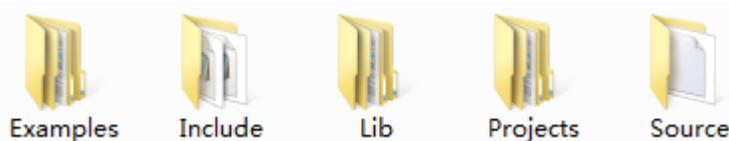
当然，也可以在 ARM 官网下载，只是这两年 ARM 官网升级得非常频繁，通过检索功能找资料非常麻烦。所以不推荐大家到 ARM 官网下载资料了。

7.2.2 DSP 库的说明

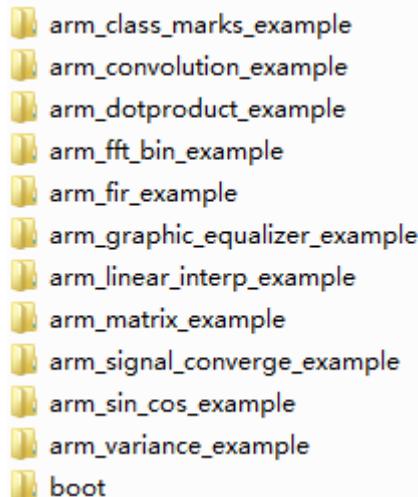
这里我们以 CMSIS V5.6.0 为标准进行移植。打开固件库里面的 CMSIS 文件，可以看到如下几个文件：



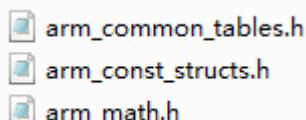
其中 DSP 文件夹是我们需要的：



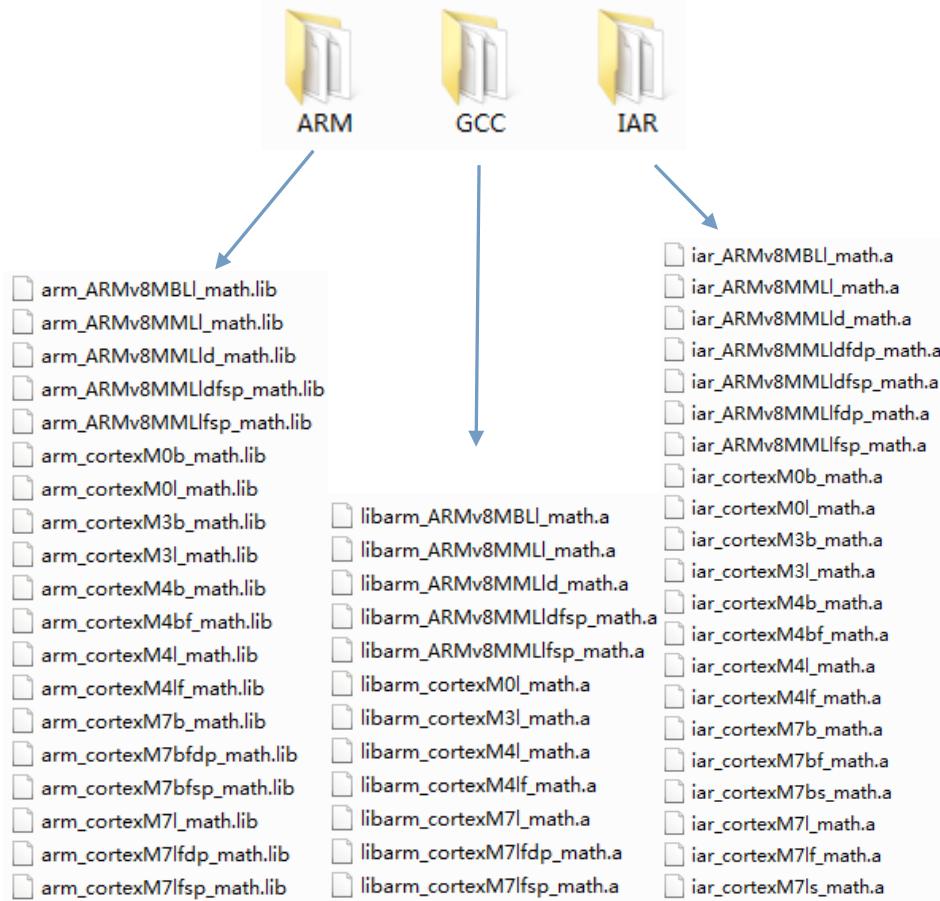
Examples 文件夹中的文件如下，主要是提供了一些例子：



Include 文件夹里面是 DSP 库的头文件：



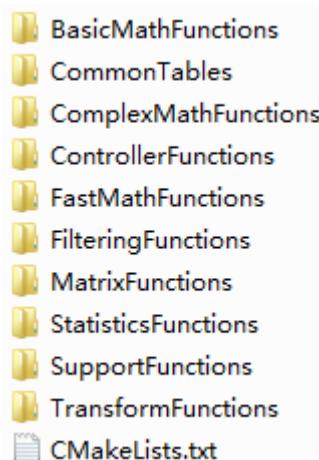
Lib 文件夹里面是 MDK (ARM) , IAR 和 CGG 版库文件：



Projects 文件夹里面的文件如下，提供了三个版本的工程模板，每个模板里面都是把所有源码文件添加了进来：



Source 文件夹中的文件如下，这个是 DSP 的源码文件：





7.3 DSP 库版本的区别

IAR 版本的 DSP 库如下：

- ❑ iar_cortexM7ls_math.a
- ❑ iar_cortexM7lf_math.a
- ❑ iar_cortexM7l_math.a
- ❑ iar_cortexM7bs_math.a
- ❑ iar_cortexM7bf_math.a
- ❑ iar_cortexM7b_math.a
- ❑ iar_cortexM4lf_math.a
- ❑ iar_cortexM4l_math.a
- ❑ iar_cortexM4bf_math.a
- ❑ iar_cortexM4b_math.a
- ❑ iar_cortexM3l_math.a
- ❑ iar_cortexM3b_math.a
- ❑ iar_cortexM0l_math.a
- ❑ iar_cortexM0b_math.a
- ❑ iar_ARMv8MMILfsp_math.a
- ❑ iar_ARMv8MMILfdp_math.a
- ❑ iar_ARMv8MMILdfsp_math.a
- ❑ iar_ARMv8MMILdfdp_math.a
- ❑ iar_ARMv8MMILd_math.a
- ❑ iar_ARMv8MMIL_math.a
- ❑ iar_ARMv8MBLl_math.a

◆ iar_cortexM7ls_math.a

Cortex-M7 内核， l 表示小端格式， s 表示带 FPU 单元， Single Precision 单精度浮点。

iar_cortexM7lf_math.a

Cortex-M7 内核， l 表示小端格式， f 表示带 FPU 单元， Double Precision 双精度浮点。

◆ iar_cortexM7l_math.a

Cortex-M7 内核， l 表示小端格式。

◆ iar_cortexM7bs_math.a

Cortex-M7 内核， b 表示大端格式， s 表示带 FPU 单元， Single Precision 单精度浮点。

◆ iar_cortexM7bf_math.a

Cortex-M7 内核， b 表示大端格式， f 表示带 FPU 单元， Double Precision 双精度浮点。

◆ iar_cortexM7b_math.a

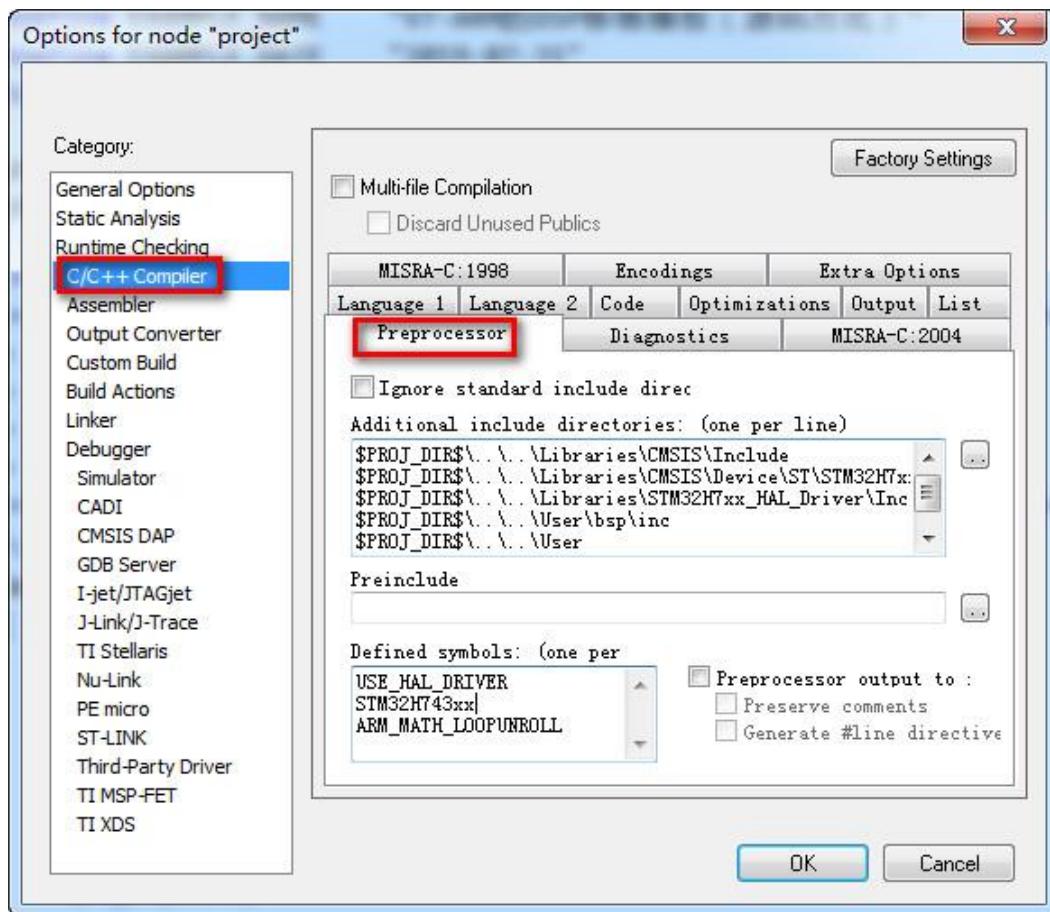
Cortex-M7 内核， b 表示大端格式。

STM32H7 是 M7 内核，双精度浮点，一般使用小端格式，所以我们选择库

iar_cortexM7lf_math.a

7.4 DSP 库的几个重要的预定义宏含义

根据用户的使用要求，这几个预定义宏可以添加到 IAR 的预定义选项中：



这里将这几个预定义宏做个介绍：

◆ ARM_MATH_BIG_ENDIAN:

大端格式。

◆ ARM_MATH_MATRIX_CHECK:

检测矩阵的输入输出大小

◆ ARM_MATH_NEON:

ARM_MATH_NEON_EXPERIMENTAL:

这两个暂时用不到，因为 M0, M3, M4 和 M7 内核不支持 NEON 指令，需要等待升级到 ARMv8.1-M 架构。

◆ ARM_MATH_ROUNDING:

主要用在浮点数转 Q32, Q15 和 Q7 时，类似四舍五入的处理上，其它函数没用到。

◆ ARM_MATH_LOOPUNROLL:

用于 4 个为一组的小批量处理上，加快执行速度。



通过下面的求绝对值函数，可以方便的看出区别：

```
void arm_abs_f32(
    const float32_t * pSrc,
    float32_t * pDst,
    uint32_t blockSize)
{
    uint32_t blkCnt; /* Loop counter */

#if defined(ARM_MATH_NEON)
    float32x4_t vec1;
    float32x4_t res;

    /* Compute 4 outputs at a time */
    blkCnt = blockSize >> 2U;

    while (blkCnt > 0U)
    {
        /* C = |A| */

        /* Calculate absolute values and then store the results in the destination buffer. */
        vec1 = vld1q_f32(pSrc);
        res = vabsq_f32(vec1);
        vst1q_f32(pDst, res);

        /* Increment pointers */
        pSrc += 4;
        pDst += 4;

        /* Decrement the loop counter */
        blkCnt--;
    }

    /* Tail */
    blkCnt = blockSize & 0x3;
}

#else
#if defined (ARM_MATH_LOOPUNROLL)

    /* Loop unrolling: Compute 4 outputs at a time */
    blkCnt = blockSize >> 2U;

    while (blkCnt > 0U)
    {
        /* C = |A| */

        /* Calculate absolute and store result in destination buffer. */
        *pDst++ = fabsf(*pSrc++);
        *pDst++ = fabsf(*pSrc++);
        *pDst++ = fabsf(*pSrc++);
        *pDst++ = fabsf(*pSrc++);

        /* Decrement loop counter */
        blkCnt--;
    }
#endif
#endif
```

```
/* Loop unrolling: Compute remaining outputs */
blkCnt = blockSize % 0x4U;

#else

/* Initialize blkCnt with number of samples */
blkCnt = blockSize;

#endif /* #if defined (ARM_MATH_LOOPUNROLL) */
#endif /* #if defined(ARM_MATH_NEON) */

while (blkCnt > 0U)
{
    /* C = |A| */

    /* Calculate absolute and store result in destination buffer. */
    *pDst++ = fabsf(*pSrc++);

    /* Decrement loop counter */
    blkCnt--;
}

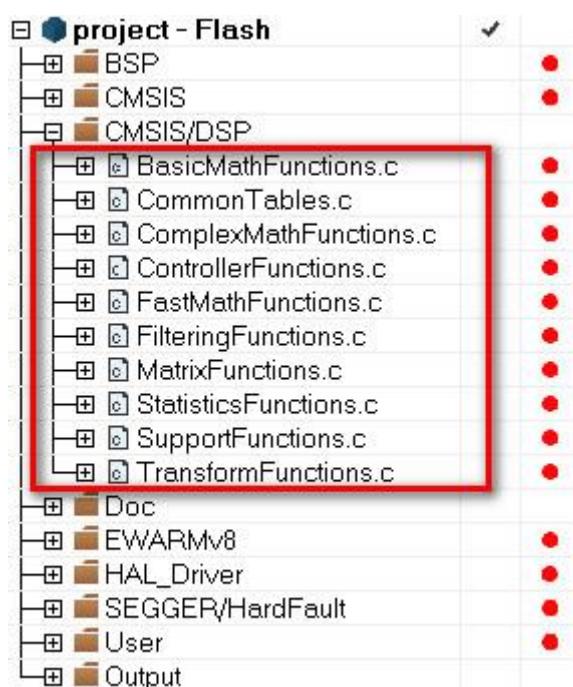
}
```

7.5 DSP 库在 IAR 上的移植（源码移植方式）

下面我们讲解下如何在 IAR 上面移植 DSP 库源码，DSP 库的移植相对比较容易。

7.5.1 第一步：建立 IAR 工程并添加 DSP 库

为了方便起见，我们这里不再专门建立一个 MDK 工程了，直接以 V7 开发板中的例子：V7-001_跑马灯例程为模板进行添加即可。打开这个实例并在左侧添加分组 CMSIS/DSP：

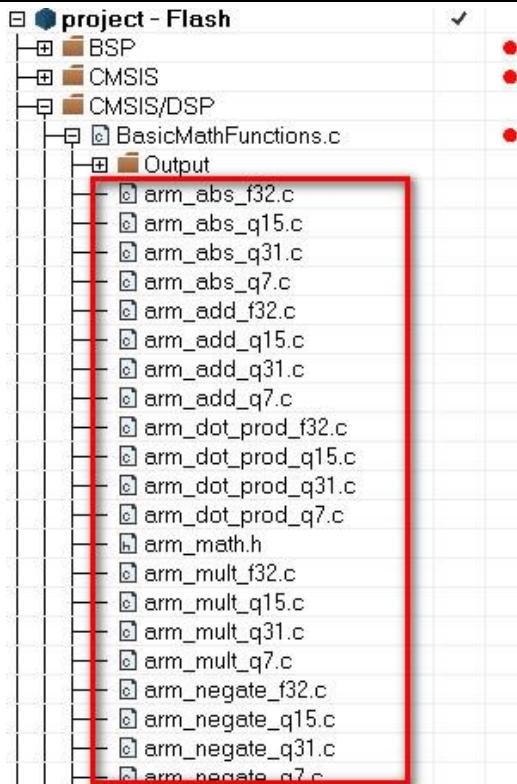




我们这里不需要添加每个 C 文件源码，仅需添加包含这些 C 文件的汇总文件，比如 BasicMathFunctions.c 文件里面包含的 C 文件就是：

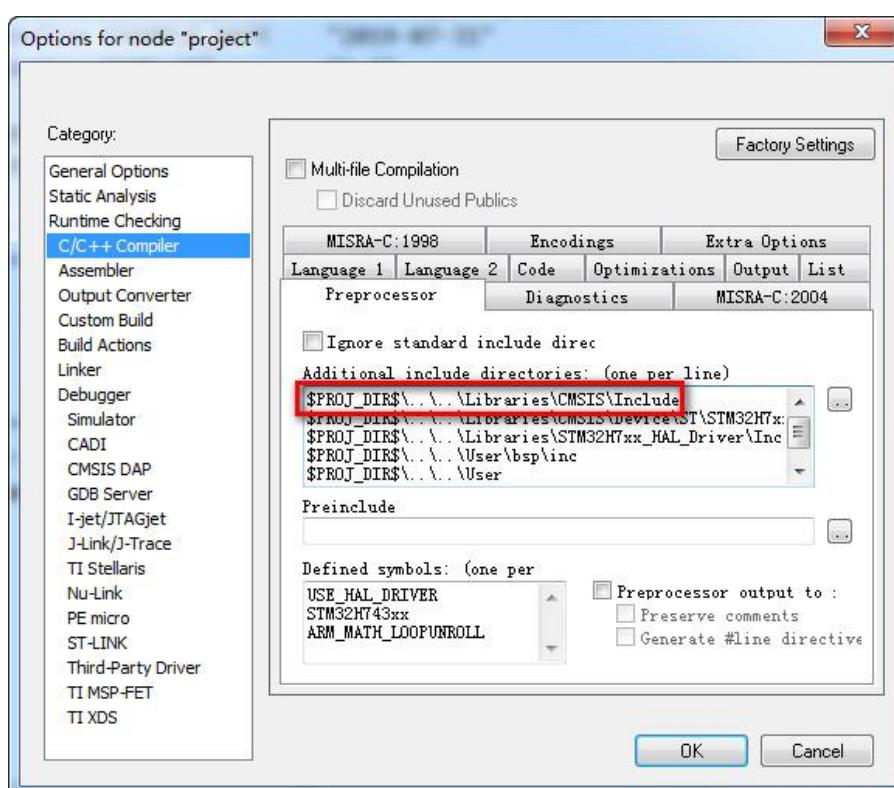
```
#include "arm_abs_f32.c"  
#include "arm_abs_q15.c"  
#include "arm_abs_q31.c"  
#include "arm_abs_q7.c"  
#include "arm_add_f32.c"  
#include "arm_add_q15.c"  
#include "arm_add_q31.c"  
#include "arm_add_q7.c"  
#include "arm_dot_prod_f32.c"  
#include "arm_dot_prod_q15.c"  
#include "arm_dot_prod_q31.c"  
#include "arm_dot_prod_q7.c"  
#include "arm_mult_f32.c"  
#include "arm_mult_q15.c"  
#include "arm_mult_q31.c"  
#include "arm_mult_q7.c"  
#include "arm_negate_f32.c"  
#include "arm_negate_q15.c"  
#include "arm_negate_q31.c"  
#include "arm_negate_q7.c"  
#include "arm_offset_f32.c"  
#include "arm_offset_q15.c"  
#include "arm_offset_q31.c"  
#include "arm_offset_q7.c"  
#include "arm_scale_f32.c"  
#include "arm_scale_q15.c"  
#include "arm_scale_q31.c"  
#include "arm_scale_q7.c"  
#include "arm_shift_q15.c"  
#include "arm_shift_q31.c"  
#include "arm_shift_q7.c"  
#include "arm_sub_f32.c"  
#include "arm_sub_q15.c"  
#include "arm_sub_q31.c"  
#include "arm_sub_q7.c"
```

这样一来，IAR 编译后会自动关联，查看源码非方便：



7.5.2 第二步：添加头文件路径

添加 DSP 所需的头文件路径，这个头文件路径是已经在模板工程中添加好的，这里只是跟大家强调一下：

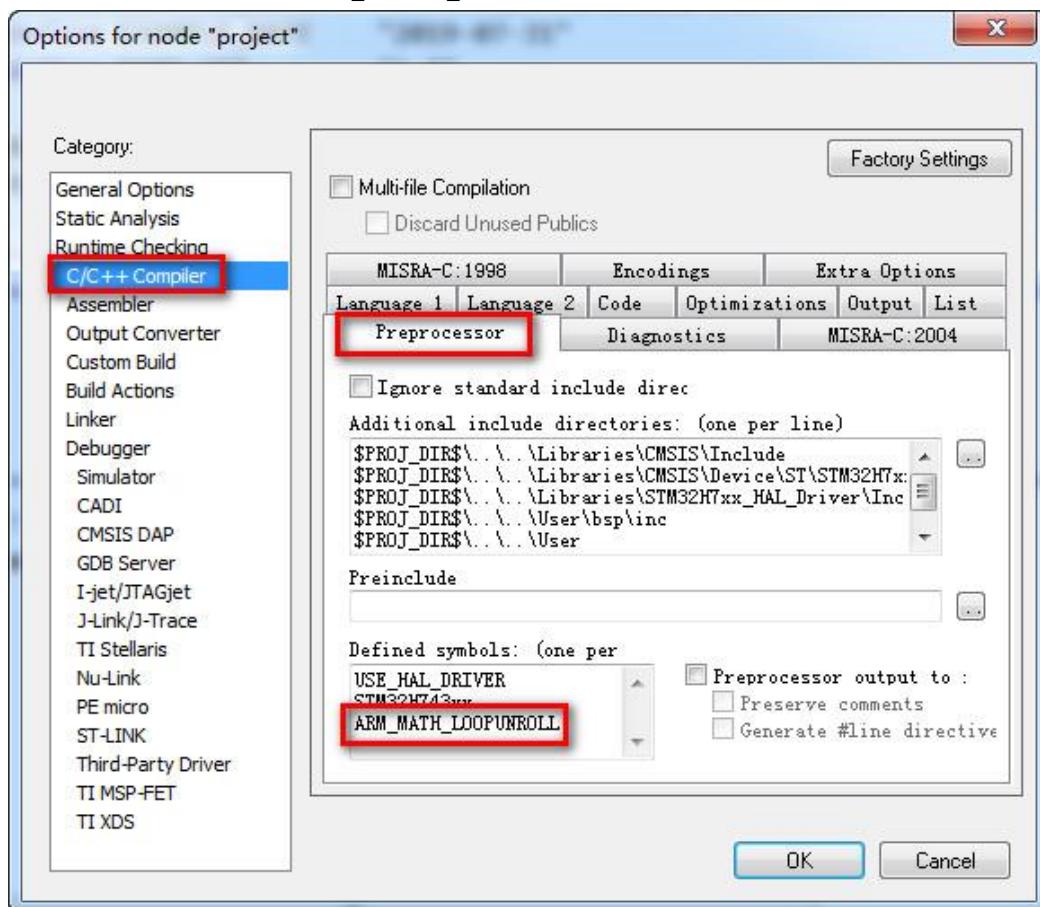


这里要注意一点，为什么直接添加路径 Libraries\CMSIS\Include 里面的头文件即可，而没有添加

Libraries\CMSIS\CMSIS\Include, 这是因为路径 Libraries\CMSIS\Include 里面已经包含了 DSP 库的头文件。

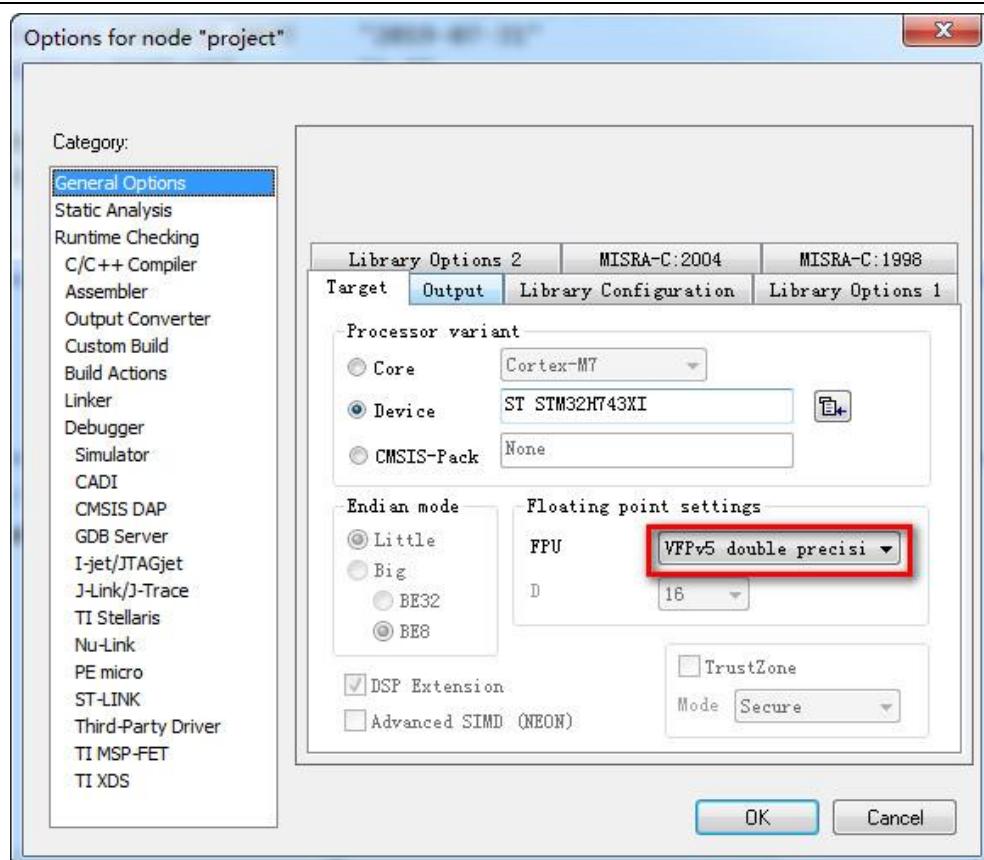
7.5.3 第三步：添加宏定义

我们这里仅使能一个宏定义 ARM_MATH_LOOPUNROLL:



7.5.4 第四步：开启 FPU

需要客户通过 MDK 开启 FPU, 由于 STM32H7 支持双精度浮点, 这里要开启 Double Precision。

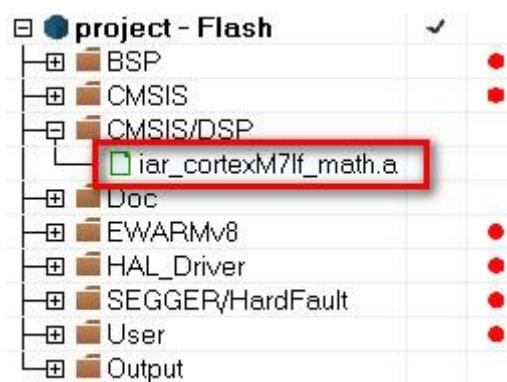


7.5.5 第五步：添加头文件 arm_math.h

用到 DSP 库函数的文件得添加`#include "arm_math.h"`就可以调用 DSP 库的 API 了。至此就完成了 DSP 库的移植。

7.6 DSP 库在 IAR 上的移植（库移植方式）

移植方法与本章 7.5 小节的相同，仅第 1 步不同，将源码的添加修改为库添加：





7.7 升级到最新版 DSP 库方法

由于 CMSIS 软件包是实时更新的，这里提供一种升级的简单办法，按照本章 7.1 小节的说明下载到最新版 CMSIS 软件包，然后直接覆盖 DSP 工程里面的 CMSIS 文件夹即可。

7.8 简易 DSP 库函数验证

这里我们主要运行 arm_abs_f32, arm_abs_q31, arm_abs_q15 这三个函数，以此来验证我们移植的 DSP 库是否正确。

配套例子：

本章配套了如下两个例子：

- ◆ V7-200_DSP 程序模板（源码方式）
- ◆ V7-201_DSP 程序模板（库方式）

实验目的：

1. 学习官方 DSP 库的移植

实验内容：

1. 按下按键 K1, 串口打印函数 arm_abs_f32 的输出结果
2. 按下按键 K2, 串口打印函数 arm_abs_q31 的输出结果
3. 按下按键 K3, 串口打印函数 arm_abs_q15 的输出结果

实验现象：

通过串口上位机软件 SecureCRT 看打印信息现象如下（分别按几次 K1, K2, K3）。如果编译的是 MDK 的 AC6 工程，特别要注意本章 7.7 小节所说的问题。



```
CPU : STM32H743XIH6, BGA240, 主频: 400MHz
UID = 34343436 3038510A 0040002D
*****
* 例程名称    : V7-ARM的DSP移植模板 (源码方式)
* 例程版本    : 1.0
* 发布日期    : 2019-07-31
* HAL库版本  : V1.3.0 (STM32H7xx HAL Driver)
*
* QQ       : 1295744630
* 旺旺     : armfly
* Email    : armfly@qq.com
* 微信公众号: armfly_com
* 淘宝店   : armfly.taobao.com
* Copyright www.armfly.com 安富莱电子
*****
操作提示:
1. 启动一个自动重装软件定时器, 每100ms翻转一次LED2
2. 按下按键K1, 串口打印函数arm_abs_f32的输出结果
3. 按下按键K2, 串口打印函数arm_abs_q31的输出结果
4. 按下按键K3, 串口打印函数arm_abs_q15的输出结果
*****
就绪
```

程序设计:

程序的设计也比较简单，通过按下不同的按键从而打印不同的 DSP 库函数执行结果，主程序如下：

```
#include "bsp.h"          /* 底层硬件驱动 */
#include "arm_math.h"

/*
 * 定义例程名和例程发布日期 */
#define EXAMPLE_NAME    "V7-ARM的DSP移植模板 (源码方式)"
#define EXAMPLE_DATE    "2019-07-31"
#define DEMO_VER        "1.0"

static void PrintfLogo(void);
static void PrintfHelp(void);

/*
 * 函数名: main
 * 功能说明: c 程序入口
 * 形参: 无
 * 返回值: 错误代码(无需处理)
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    float32_t pSrc;
    float32_t pDst;
    q31_t pSrc1;
    q31_t pDst1;
    q15_t pSrc2;
    q15_t pDst2;
```



```
bsp_Init(); /* 硬件初始化 */

PrintfLogo(); /* 打印例程名称和版本等信息 */
PrintfHelp(); /* 打印操作提示 */

bsp_StartAutoTimer(0, 100); /* 启动1个100ms的自动重装的定时器 */

/* 主程序大循环 */
while (1)
{
    /* CPU 空闲时执行的函数, 在 bsp.c */
    bsp_Idle();

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        /* 翻转 LED2 的状态 */
        bsp_LedToggle(2);
    }

    /* 处理按键事件 */
    ucKeyCode = bsp_GetKey();
    if (ucKeyCode > 0)
    {
        /* 有键按下 */
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1: /* K1键按下 */
                pSrc -= 1.23f;
                arm_abs_f32(&pSrc, &pDst, 1);
                printf("pDst = %f\r\n", pDst);
                break;

            case KEY_DOWN_K2: /* K2键按下 */
                pSrc1 -= 1;
                arm_abs_q31(&pSrc1, &pDst1, 1);
                printf("pDst1 = %d\r\n", pDst1);
                break;

            case KEY_DOWN_K3: /* K3键按下 */
                pSrc2 -= 1;
                arm_abs_q15(&pSrc2, &pDst2, 1);
                printf("pDst2 = %d\r\n", pDst2);
                break;

            default:
                break;
        }
    }
}
```

7.9 总结

本期教程主要跟大家介绍了官方 DSP 库的移植，相对来说移植也比较简单，建议初学的同学按照这个步骤移植一遍。



第8章 DSP 定点数和浮点数（重要）

本期教程主要跟大家讲解一下定点数和浮点数的基础知识，了解这些基础知识对于后面学习 ARM 官方的 DSP 库大有裨益。**特别是初学的一一定要理解这些基础知识。**

8.1 定点数和浮点数概念

8.2 IEEE 浮点数

8.3 定点数运算

8.4 总结

8.1 初学者重要提示

- ◆ 如果之前没有接触过这方面的知识点，首次学习会有点不太理解，随着后面章节的深入就慢慢理解了。

8.2 定点数和浮点数概念

如果小数点的位置事先已有约定，不再改变，此类数称为“定点数”。相比之下，如果小数点的位置可变，则称为“浮点数”（**定点数的本质是小数，整数只是其表现形式**）。

8.2.1 定点数

常用的定点数有两种表示形式：如果小数点位置约定在最低数值位的后面，则该数只能是定点整数；如果小数点位置约定在最高数值位的前面，则该数只能是定点小数。

8.2.2 浮点数

在计算机系统的发展过程中，曾经提出过多种方法表达实数。典型的比如相对于浮点数的定点数（Fixed Point Number）。在这种表达方式中，小数点固定的位于实数所有数字中间的某个位置。货币的表达就可以使用这种方式，比如 99.00 或者 00.99 可以用于表达具有四位精度，小数点后有两位的货币值。由于小数点位置固定，所以可以直接用四位数值来表达相应的数值。SQL 中的 NUMBER 数据类型就是利用定点数来定义的。还有一种提议的表达方式为有理数表达方式，即用两个整数的比值来表达实数。

定点数表达法的缺点在于其形式过于僵硬，固定的小数点位置决定了固定位数的整数部分和小数部分，不利于同时表达特别大的数或者特别小的数。最终，绝大多数现代的计算机系统采纳了所谓的浮点



数表达方式。这种表达方式利用科学计数法来表达实数，即用一个尾数（Mantissa），一个基数（Base），一个指数（Exponent）以及一个表示正负的符号来表达实数。比如 123.45 用十进制科学计数法可以表达为 1.2345×10^2 ，其中 1.2345 为尾数，10 为基数，2 为指数。浮点数利用指数达到了浮动小数点的效果，从而可以灵活地表达更大范围的实数。

提示：尾数有时也称为有效数字（Significand）。尾数实际上是有效数字的非正式说法。

同样的数值可以有多种浮点数表达方式，比如上面例子中的 123.45 可以表达为 12.345×10^1 , 0.12345×10^3 或者 1.2345×10^2 。因为这种多样性，有必要对其加以规范化以达到统一表达的目标。规范的（Normalized）浮点数表达方式具有如下形式：

$$\pm d.dd\dots d \times \beta^e, (0 \leq d_i < \beta)$$

其中 $d.dd\dots d$ 即尾数， β 为基数， e 为指数。尾数中数字的个数称为精度，在本文中用 p 来表示。每个数字 d 介于 0 和基数之间，包括 0。小数点左侧的数字不为 0。

基于规范表达的浮点数对应的具体值可由下面的表达式计算而得：

$$\pm(d^0 + d^1\beta^{-1} + \dots + d^{p-1}\beta^{-(p-1)})\beta^e, (0 \leq d_i < \beta)$$

对于十进制的浮点数，即基数 β 等于 10 的浮点数而言，上面的表达式非常容易理解，也很直白。计算机内部的数值表达是基于二进制的。从上面的表达式，我们可以知道，二进制数同样可以有小数点，也同样具有类似于十进制的表达方式。只是此时 β 等于 2，而每个数字 d 只能在 0 和 1 之间取值。比如二进制数 1001.101 相当于 $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$ ，对应于十进制的 9.625。其规范浮点数表达为 1.001101×2^3 。

8.3 IEEE 浮点数

说明：Cortex-M7 中的 FPU（浮点单元）就是用的这个 IEEE 754 标准，初学的要认真学习。

IEEE 二进制浮点数算术标准（IEEE 754）是 20 世纪 80 年代以来最广泛使用的浮点数运算标准，为许多 CPU 与浮点运算器所采用。这个标准定义了表示浮点数的格式（包括负零 -0）与反常值（denormal number），一些特殊数值（无穷 Inf）与非数值（NaN），以及这些数值的“浮点数运算符”；它也指明了四种数值舍入规则和五种例外状况（包括例外发生的时机与处理方式）。

IEEE 754 规定了四种表示浮点数值的方式：单精确度（32 位）、双精确度（64 位）、延伸单精确度（43 比特以上，很少使用）与延伸双精确度（79 比特以上，通常以 80 比特实现）。只有 32 位模式有强制要求，其他都是选择性的。大部分编程语言都有提供 IEEE 浮点数格式与算术，但有些将其列为非必需的。例如，IEEE 754 问世之前就有的 C 语言，现在有包括 IEEE 算术，但不算作强制要求（C 语言的 float 通常是指 IEEE 单精确度，而 double 是指双精确度）。

该标准的全称为 IEEE 二进制浮点数算术标准（ANSI/IEEE Std 754-1985），又称 IEC 60559:1989，微处理器系统的二进制浮点数算术（本来的编号是 IEC 559:1989）。后来还有“与基数无关的浮点数”的“IEEE 854-1987 标准”，有规定基数为 2 跟 10 的状况。现在最新标准是“IEEE 854-2008 标准”。



在六、七十年代，各家计算机公司的各个型号的计算机，有着千差万别的浮点数表示，却没有一个业界通用的标准。这给数据交换、计算机协同工作造成了极大不便。IEEE 的浮点数专业小组于七十年代末期开始酝酿浮点数的标准。在 1980 年，英特尔公司就推出了单片的 8087 浮点数协处理器，其浮点数表示法及定义的运算具有足够的合理性、先进性，被 IEEE 采用作为浮点数的标准，于 1985 年发布。而在此前，这一标准的内容已在八十年代初期被各计算机公司广泛采用，成了事实上的业界工业标准。

在 IEEE 标准中，浮点数是将特定长度的连续字节的所有二进制位分割为特定宽度的符号域，指数域和尾数域三个域，其中保存的值分别用于表示给定二进制浮点数中的符号，指数和尾数。这样，通过尾数和可以调节的指数（所以称为“浮点”）就可以表达给定的数值了。具体的格式参见下面的图例：

IEEE 单精度浮点数

符号 Sign	指数 Exponent	尾数 Mantissa
1bit	8bit	23bit

IEEE 双精度浮点数

符号 Sign	指数 Exponent	尾数 Mantissa
1bit	11bit	52bit

- 在上面的图例中，第一个域为符号域。其中 0 表示数值为正数，而 1 则表示负数。
- 第二个域为指数域。其中单精度数为 8 位，双精度数为 11 位。以单精度数为例，8 位的指数为可以表达 0 到 255 之间的 255 个指数值。但是，指数可以为正数，也可以为负数。

为了处理负指数的情况，实际的指数值按要求需要加上一个偏差（Bias）值作为保存在指数域中的值，单精度数的偏差值为 127，而双精度数的偏差值为 1023。比如，单精度的实际指数值 0 在指数域中将保存为 127；而保存在指数域中的 64 则表示实际的指数值 -63。偏差的引入使得对于单精度数，实际可以表达的指数值的范围就变成 -127 到 128 之间（包含两端）。我们后面还将看到，实际的指数值 -127（保存为全 0）以及 +128（保存为全 1）保留用作特殊值的处理。这样，实际可以表达的有效指数范围就在 -127 和 127 之间。在本文中，最小指数和最大指数分别用 e_{\min} 和 e_{\max} 来表达。

- 图例中的第三个域为尾数域，其中单精度数为 23 位长，双精度数为 52 位长。除了我们将要讲到的某些特殊值外，IEEE 标准要求浮点数必须是规范的。这意味着尾数的小数点左侧必须为 1，因此我们在保存尾数的时候，可以省略小数点前面这个 1，从而腾出一个二进制位来保存更多的尾数。这样我们实际上用 23 位长的尾数域表达了 24 位的尾数。比如：

对于单精度数而言，二进制的 1001.101（对应于十进制的 9.625）可以表达为 1.001101×2^3 ，所以实际保存在尾数域中的值为 0011 0100 0000 000 0000 0000，即去掉小数点左侧的 1，并用 0 在右侧补齐。



值得注意的是，对于单精度数，由于我们只有 24 位的指数（其中一位隐藏），所以可以表达的最大指数为 $2^{24} - 1 = 16,777,215$ 。特别的，16,777,216 是偶数，所以我们可以将它除以 2 并相应地调整指数来保存这个数，这样 16,777,216 同样可以被精确的保存。相反，数值 16,777,217 则无法被精确的保存。由此，我们可以看到单精度的浮点数可以表达的十进制数值中，真正有效的数字不高于 8 位。事实上，对相对误差的数值分析结果显示有效的精度大约为 7.22 位。参考下面的示例：

例：

true value	stored value
16,777,215	1.6777215E7
16,777,216	1.6777216E7
16,777,217	1.6777216E7
16,777,218	1.6777218E7
16,777,219	1.677722E7
16,777,220	1.677722E7
16,777,221	1.677722E7
16,777,222	1.6777222E7
16,777,223	1.6777224E7
16,777,224	1.6777224E7
16,777,225	1.6777224E7

根据标准要求，无法精确保存的值必须向最接近的可保存的值进行舍入。这有点像我们熟悉的十进制的四舍五入，即不足一半则舍，一半以上（包括一半）则进。不过对于二进制浮点数而言，还多一条规矩，就是当需要舍入的值刚好是一半时，不是简单地进，而是在前后两个等距接近的可保存的值中，取其中最后一位有效数字为零者。从上面的示例中可以看出，奇数都被舍入为偶数，且有舍有进。我们可以将这种舍入误差理解为“半位”的误差。所以，为了避免 7.22 对很多人造成的困惑，有些文章经常以 7.5 位来说明单精度浮点数的精度问题。

提示：这里采用的浮点数舍入规则有时被称为舍入到偶数（Round to Even）。相比简单地逢一半则进的舍入规则，舍入到偶数有助于从某些角度减小计算中产生的舍入误差累积问题。因此为 IEEE 标准所采用。

8.3.1 规范化浮点数

通过前面的介绍，大家应该已经了解的浮点数的基本知识，这些知识对于一个不接触浮点数应用的人应该足够了。简单总结如下：

标准的浮点数都符合如下的公式：

$$\text{Normalized Number} = (-1)^s \cdot (1 + \sum f_i \cdot 2^{-i}) \cdot 2^{e-\text{bias}} \quad (\text{with } i > 0)$$

其中 `bias` 是固定的数值，这个在前面的已经讲解过。参数的具体范围如下

Mode	Exponent	Exp. Bias	Exp. Range	Mantissa	Min. value	Max. Value
Half	5-bit	15	-14, +15	10-bit	$6.10.10^{-5}$	65504
Single	8-bit	127	-126,+127	23-bit	$1,18. 10^{-38}$	$3,40.10^{38}$
Double	11-bit	1023	-1022,+1023	52-bit	$2,23.10^{-308}$	$1,8.10^{308}$

8.3.2 非规范化浮点数

现在我们看看这两个浮点数的差值。不难得出，该差值为 0.0001×2^{-125} ，表达为规范浮点数则为 1.0×2^{-129} 。问题在于其指数大于允许的最小指数值，所以无法保存为规范浮点数。最终，只能近似为零 (Flush to Zero)。这中特殊情况意味着下面本来十分可靠的代码也可能出现问题：

```
if (x != y)
{
    z = 1 / (x -y);
}
```

正如我们精心选择的两个浮点数展现的问题一样，即使 x 不等于 y ， x 和 y 的差值仍然可能绝对值过小，而近似为零，导致除以 0 的情况发生。

为了解决此类问题，IEEE 标准中引入了非规范（Denormalized）浮点数。规定当浮点数的指数为允许的最小指数值，即 $emin$ 时，尾数不必是规范化的。比如上面例子中的差值可以表达为非规范的浮点数 0.001×2^{-126} ，其中指数 -126 等于 $emin$ 。注意，这里规定的是“不必”，这也就意味着“可以”。当浮点数实际的指数为 $emin$ ，且指数域也为 $emin$ 时，该浮点数仍是规范的，也就是说，保存时隐含着一个隐藏的尾数位。为了保存非规范浮点数，IEEE 标准采用了类似处理特殊值零时所采用的办法，即用特殊的指数域值 $emin - 1$ 加以标记，当然，此时的尾数域不能为零。这样，例子中的差值可以保存为 00000000000100000000000000000000 (0x100000)，没有隐含的尾数位。

有了非规范浮点数，去掉了隐含的尾数位的制约，可以保存绝对值更小的浮点数。而且，也由于不再受到隐含尾数域的制约，上述关于极小差值的问题也不存在了，因为所有可以保存的浮点数之间的差值同样可以保存。



8.3.3 有符号的零

因为 IEEE 标准的浮点数格式中，小数点左侧的 1 是隐藏的，而零显然需要尾数必须是零。所以，零也就无法直接用这种格式表达而只能特殊处理。

实际上，零保存为尾数域为全为 0，指数域为 $\text{emin} - 1 = -127$ ，也就是说指数域也全为 0。考虑到符号域的作用，所以存在着两个零，即 +0 和 -0。不同于正负无穷之间是有序的，IEEE 标准规定正负零是相等的。

零有正负之分，的确非常容易让人困惑。这一点是基于数值分析的多种考虑，经利弊权衡后形成的结果。有符号的零可以避免运算中，特别是涉及无穷的运算中，符号信息的丢失。举例而言，如果零无符号，则等式 $1/(1/x) = x$ 当 $x = \pm\infty$ 时不再成立。原因是如果零无符号，1 和正负无穷的比值为同一个零，然后 1 与 0 的比值为正无穷，符号没有了。解决这个问题，除非无穷也没有符号。但是无穷的符号表达了上溢发生在数轴的哪一侧，这个信息显然是不能不要的。零有符号也造成了其它问题，比如当 $x=y$ 时，等式 $1/x = 1/y$ 在 x 和 y 分别为 +0 和 -0 时，两端分别为正无穷和负无穷而不再成立。当然，解决这个问题的另一个思路是和无穷一样，规定零也是有序的。但是，如果零是有序的，则即使 `if (x==0)` 这样简单的判断也由于 x 可能是 ± 0 而变得不确定了。两害取其轻者，零还是无序的好。

8.3.4 无穷

和 NaN 一样，特殊值无穷 (Infinity) 的指数部分同样为 $\text{emax} + 1 = 128$ ，不过无穷的尾数域必须为零。无穷用于表达计算中产生的上溢 (Overflow) 问题。比如两个极大的数相乘时，尽管两个操作数本身可以用保存为浮点数，但其结果可能大到无法保存为浮点数，而必须进行舍入。根据 IEEE 标准，此时不是将结果舍入为可以保存的最大的浮点数（因为这个数可能离实际的结果相差太远而毫无意义），而是将其舍入为无穷。对于负数结果也是如此，只不过此时舍入为负无穷，也就是说符号域为 1 的无穷。有了 NaN 的经验我们不难理解，特殊值无穷使得计算中发生的上溢错误不必以终止运算为结果。

无穷和除 NaN 以外的其它浮点数一样是有序的，从小到大依次为负无穷，负的无穷非零值，正负零（随后介绍），正的无穷非零值以及正无穷。除 NaN 以外的任何非零值除以零，结果都将是无穷，而符号则由作为除数的零的符号决定。

当零除以零时得到的结果不是无穷而是 NaN。原因不难理解，当除数和被除数都逼近于零时，其商可能为任何值，所以 IEEE 标准决定此时用 NaN 作为商比较合适。

8.3.5 NaN

NaN 用于处理计算中出现的错误情况，比如 0.0 除以 0.0 或者求负数的平方根。由上面的表中可以看出，对于单精度浮点数，NaN 表示为指数为 $\text{emax} + 1 = 128$ （指数域全为 1），且尾数域不等于零的浮点数。IEEE 标准没有要求具体的尾数域，所以 NaN 实际上不是一个，而是一族。不同的实现可以自由选择尾数域的值来表达 NaN。



8.4 定点数运算

8.4.1 数的定标 (Q 格式)

在许多情况下,数学运算过程中的数不一定都是整数,而且定点 DSP 和不带 FPU 的处理器是无能为力的。那么是不是说定点 DSP 和不带 FPU 的处理器就不能处理各种小数呢?当然不是。这其中的关键就是由程序员来确定一个数的小数点处于数据中的哪一位。这就是数的定标 (由于很多时候,我们都是直接用 C 来实现浮点运算,具体的底层转化我们并没有去关心,所以也就很少有人知道数的定标)。

通过设定小数点在数据中的不同位置,就可以表示不同大小和不同精度的小数了。数的定标有 Q 表示法和 S 表示法两种。下表列出了一个 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。

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
Q6	S9.6	-512≤x≤511.9804375
Q5	S10.5	-1024≤x≤1023.96875
Q4	S11.4	-2048≤x≤2047.9375
Q3	S12.3	-4096≤x≤4095.875
Q2	S13.2	-8192≤x≤8191.75
Q1	S14.1	-16384≤x≤16383.5
Q0	S15.0	-32768≤x≤32767

从上表可以看出,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

还可以看出,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-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。因此,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。



浮点数与定点数的转换关系可表示为：

浮点数(x)转换为定点数(xq): $xq = (\text{int})x * 2^Q$

定点数(xq)转换为浮点数(x): $x = (\text{float})xq * 2^{-Q}$

例如,浮点数 $x=0.5$,定标 $Q=15$,则定点数 $xq=L0.5*32768J=16384$,式中 LJ 表示下取整。反之,一个用 $Q=15$ 表示的定点数 16384 ,其浮点数为 $16384 * 2^{-15}=16384/32768=0.5$ 。浮点数转换为定点数时,为了降低截尾误差,在取整前可以先加上 0.5 。

8.4.2 定点数的算术运算

关于定点数的算术运算会在讲解 ARM 官方的 DSP 教程时专门给大家讲解。

8.5 总结

本期教程就跟大家讲这么多,这部分知识对于初学 DSP 的非常重要,建议认真学习下,有兴趣的可以在网上多查些资料进行了解。



第9章 Matlab 的串口通信实现

本章节主要为大家讲解 Matlab 的串口方式波形数据传输和后期数据分析功能，非常实用。

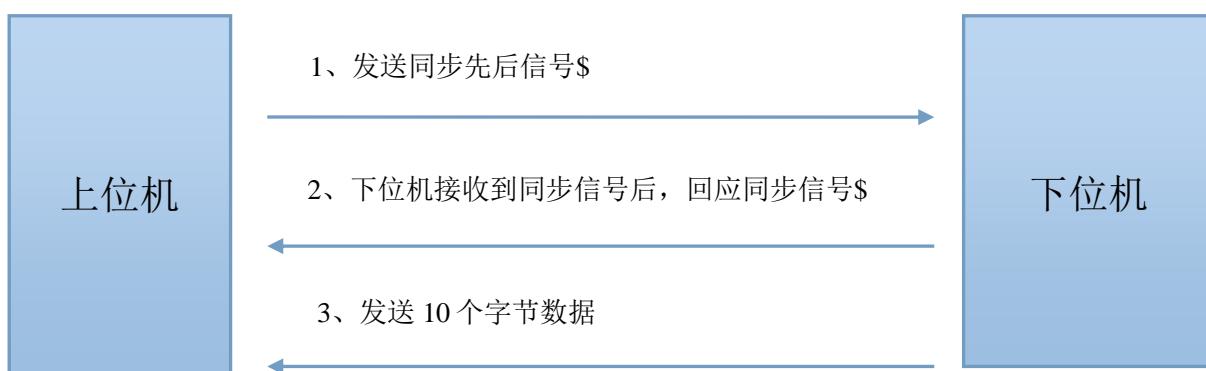
- 9.1 初学者重要提示
- 9.2 程序设计框架
- 9.3 下位机 STM32H7 程序设计
- 9.4 上位机 Matlab 程序设计
- 9.5 Matlab 上位机程序运行
- 9.6 实验例程说明 (MDK)
- 9.7 实验例程说明 (IAR)
- 9.8 总结

9.1 初学者重要提示

- ◆ 测试本章节例程注意事项。
 - 请优先运行开发板，然后运行 matlab。
 - 调试 matlab 串口数据发送前，请务必关闭串口助手。
- ◆ 函数 `delete(instrfindall);`
如果不用 matlab 了，请在 matlab 的命令输入窗口调用此函数，防止 matlab 一直占用串口。

9.2 程序设计框架

上位机和下位机的程序设计框架如下：





上位机和下位机做了一个简单的同步，保证数据通信不出错。

9.3 下位机 STM32H7 程序设计

STM32H7 端的程序设计思路。

9.3.1 第 1 步，发送的数据格式

为了方便数据发送，专门设计了一个数据格式：

```
_packed typedef struct
{
    uint16_t data1;
    uint16_t data2;
    uint16_t data3;
    uint8_t  data4;
    uint8_t  data5;
    uint8_t  data6;
    uint8_t  data7;
}
SENDPARAM_T;

SENDPARAM_T g_SendData;
```

为了保证数据的连续存储，特地在前面加了一个关键词 `_packed`。关于结构体变量占用多少字节问题，此贴进行了详细说明：<http://www.armbbs.cn/forum.php?mod=viewthread&tid=89103>。

9.3.2 第 2 步，接收同步信号\$

Matlab 发送同步信号\$ (ASCII 编码值是 13) 给开发板。

```
int main(void)
{
    /* 省略未写，仅留下关键代码 */

    /* 进入主程序循环体 */
    while (1)
    {

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        if (comGetChar(COM1, &read))
        {
            /* 接收到同步帧'$' */
            if(read == 13)
            {
                bsp_LedToggle(4);
                bsp_DelayMS(10);
                Serial_sendDataMATLAB();
            }
        }
    }
}
```



```
        }
    }
}
```

通过函数 comGetChar 获取串口接收到的数据，如果数值是 13，说明接收到 Matlab 发送过来的同步信号了。

这里要注意一点，程序这里接收到同步信号后，延迟了 10ms 再发数据给 matlab，主要是因为 matlab 的波形刷新有点快，程序这里每发送给 matlab 一次数据，matlab 就会刷新一次，10ms 就相当于 100Hz 的刷新率，也会有一定的闪烁感。

9.3.3 第 3 步，发送数据给 Matlab

下面是给 Matlab 回复同步信号和相应数据的实现：

```
/*
*****
* 函数名: Serial_sendDataMATLAB
* 功能说明: 发送串口数据给 matlab
* 形参: 无
* 返回值: 无
*****
*/
static void Serial_sendDataMATLAB(void)
{
    /* 先发同步信号'$' */
    comSendChar(COM1, 13);

    /* 发送数据,一共 10 个字节 */
    g_SendData.data1 = rand() %65536;
    g_SendData.data2 = rand() %65536;
    g_SendData.data3 = rand() %65536;
    g_SendData.data4 = rand() %256;
    g_SendData.data5 = rand() %256;
    g_SendData.data6 = rand() %256;
    g_SendData.data7 = rand() %256;
    comSendBuf(COM1, (uint8_t *)&g_SendData, 10);
}
```

开发板接收到同步信号后，会回应一个同步信号，然后将 10 个字节的数据发送给 matlab。通过这三步就完成了 STM32H7 端的程序设计。

9.4 上位机 Matlab 程序设计

Matlab 端的程序设计要略复杂些，需要大家理解 matlab 端的 API。具体说明可以看如下地址：

<https://ww2.mathworks.cn/help/matlab/serial-port-devices.html>。

9.4.1 第 1 步，配置并打开串口

下面操作是配置并打开串口：

```
close all
clear all
```



```
%删除所有已经打开的串口，这条很重要，防止之前运行没有关闭串口
```

```
delete(instrfindall);
```

```
%打开串口 COM1，波特率 115200，8 位数据位，1 位停止位，无奇偶校验，无流控制
```

```
s = serial('COM1', 'BaudRate', 115200, 'DataBits', 8, 'StopBits', 1, 'Parity', 'none', 'FlowControl', 'none');
```

```
s.ReadAsyncMode = 'continuous';
```

```
fopen(s);
```

```
fig = figure(1);
```

这里有以下几点需要大家了解：

◆ 函数`delete(instrfindall)`

这个函数比较重要，防止上次matlab操作串口，结束时没有关闭串口。通过这个函数会将其关闭。

◆ 函数`serial`

大家要特别注意打开的COM序号，务必要根据实际使用的COM号进行设置。

◆ 函数`fopen`

通过函数`fopen`打开串口。

9.4.2 第2步，相关变量设置

程序里面对这些变量的注释已经比较详细：

```
AxisMax = 65536; %坐标轴最大值
AxisMin = -65536; %坐标轴最小值
window_width = 800; %窗口宽度

g_Count = 0; %接收到的数据计数
SOF = 0; %同步帧标志
AxisValue = 1; %坐标值
RecDataDisp = zeros(1, 100000); %开辟 100000 个数据单元，用于存储接收到的数据。
RecData = zeros(1, 100); %开辟 100 个数据单元，用于数据处理。
Axis = zeros(1, 100000); %开辟 100000 个数据单元，用于 X 轴。

window = window_width * (-0.9); %窗口 X 轴起始坐标
axis([window, window + window_width, AxisMin, AxisMax]); %设置窗口坐标范围
```

%子图 1 显示串口上传的数据

```
subplot(2, 1, 1);
grid on;
title('串口数据接收');
xlabel('时间');
ylabel('数据');
```

%子图 2 显示波形的幅频响应

```
subplot(2, 1, 2);
grid on;
title('FFT');
xlabel('频率');
ylabel('幅度');

Fs = 100; % 采样率
N = 50; % 采样点数
n = 0:N-1; % 采样序列
f = n * Fs / N; %真实的频率
```



这里有以下几点需要大家了解：

◆ 变量RecDataDisp, RecData和Axis

这几个变量专门开辟好了数据空间，防止matlab警告和刷新波形慢的问题，大家根据需要可以进行加大。

◆ 采样率Fs = 100和采样点数N = 50

这个地方要根据实际的情况进行设置。

9.4.3 第3步，数据同步部分

这部分代码比较关键，matlab先发送同步信号\$出去，然后等待开发板回复同步信号\$，并读取本次通信的数据。

```
%设置同步信号标志，= 1 表示接收到下位机发送的同步帧
SOF = 0;

%发送同步帧
fwrite(s, 13);

%获取是否有数据
bytes = get(s, 'BytesAvailable');
if bytes == 0
    bytes = 1;
end

%读取下位机返回的所有数据
RecData = fread(s, bytes, 'uint8');

%检索下位机返回的数据中是否有字符$
StartData = find(RecData == 13);

%如果检索到$，读取 10 个字节的数据，也就是 5 个 uint16 的数据
if(StartData >= 1)
    RecData = fread(s, 5, 'uint16');
    SOF =1;
    StartData = 0;
end
```

这里有以下几点需要大家了解：

◆ 函数fwrite(s, 13)

用于发送同步信号\$（ASCII值是13）。

◆ 函数get(s, 'BytesAvailable')

用于获取串口缓冲中的字节数。

◆ 函数fread(s, bytes, 'uint8')

将串口缓冲的数据读取输出。

◆ 函数find(RecData == 13)

检索接收到串口数据中是否有同步信号\$。

◆ 函数fread(s, 5, 'uint16')



如果检索到\$，继续读取10个字节的数据，也就是5个uint16的数据。

9.4.4 第4步，显示串口上传的数据

下面 matlab 的数据显示波形

```
%更新接收到的数据波形
if (SOF == 1)
    %更新数据
    RecDataDisp(AxisValue) = RecData(1);
    RecDataDisp(AxisValue + 1) = RecData(2);
    RecDataDisp(AxisValue + 2) = RecData(3);
    RecDataDisp(AxisValue + 3) = RecData(4);
    RecDataDisp(AxisValue + 4) = RecData(5);

    %更新 X 轴
    Axis(AxisValue) = AxisValue;
    Axis(AxisValue + 1) = AxisValue + 1;
    Axis(AxisValue + 2) = AxisValue + 2;
    Axis(AxisValue + 3) = AxisValue + 3;
    Axis(AxisValue + 4) = AxisValue + 4;

    %更新变量
    AxisValue = AxisValue + 5;
    g_Count = g_Count + 5;

    %绘制波形
    subplot(2, 1, 1);
    plot(Axis(1:AxisValue-1), RecDataDisp(1:AxisValue-1), 'r');
    window = window + 5;
    axis([window, window + window_width, AxisMin, AxisMax]);
    grid on;
    title('串口数据接收');
    xlabel('时间');
    ylabel('数据');
    drawnow
end
```

这里有以下几点需要大家了解：

◆ 数组RecDataDisp, RecData和Axis

这里要尤其注意，matlab的数组索引是从1开始的，也是开头直接定义AxisValue = 1的原因。

◆ 函数plot

这里plot的实现尤其重要，务必要注意坐标点和数值个数要匹配。

9.4.5 第5步，FFT 数据展示

FFT部分会在在后面章节为大家详细讲解，这里也做个说明，这里是每接收够50个数据，做一次FFT：

```
if(g_Count== 50)
    subplot(2, 1, 2);
    %对原始信号做 FFT 变换
    y = fft(RecDataDisp(AxisValue-50:AxisValue-1), 50);

    %求 FFT 转换结果的模值
    Mag = abs(y)*2/N;
```



```
%绘制幅频相应曲线
plot(f, Mag, 'r');
grid on;
title('FFT');
xlabel('频率');
ylabel('幅度');
g_Count = 0;
drawnow
end
```

9.5 Matlab 上位机程序运行

M 文件的程序代码在例子 V7-203_Matlab 串口波形刷新和数据分析 m 文件里面。M 文件的运行方法在第 4 章的 4.2 小节有详细说明。

9.6 实验例程说明 (MDK)

配套例子：

V7-202_Matlab 的串口通信实现

实验目的：

1. 学习 matlab 的串口数据通信。

实验内容：

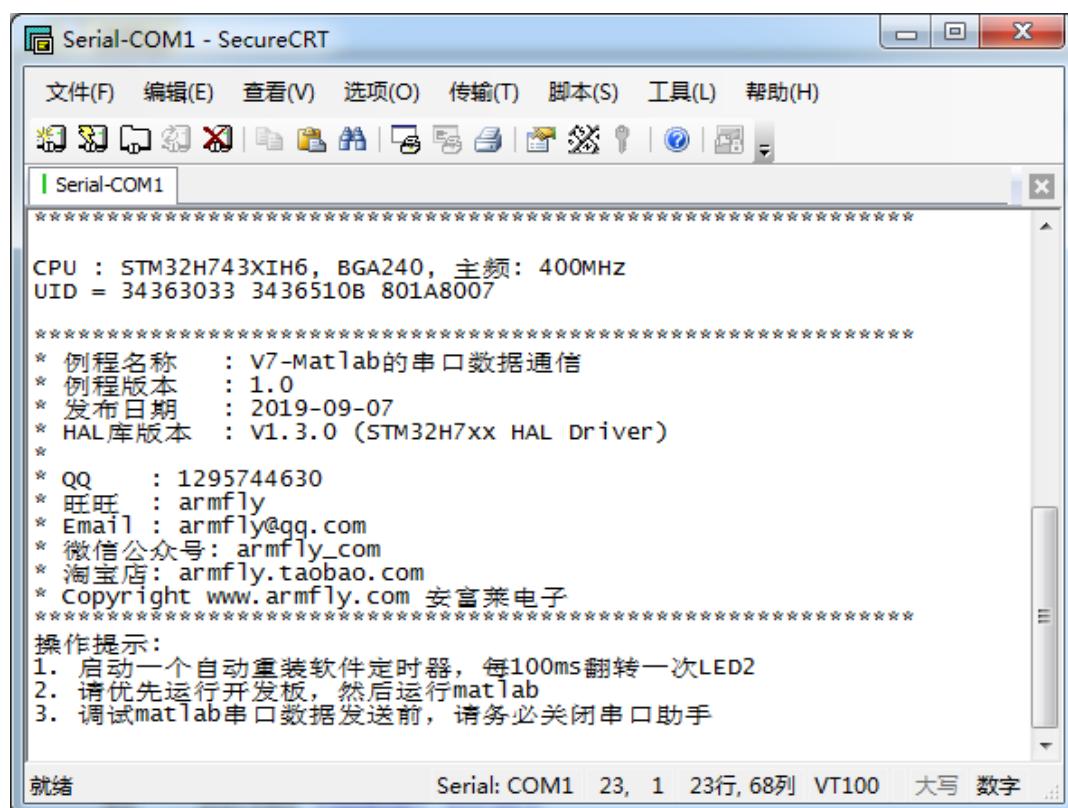
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 请优先运行开发板，然后运行 matlab。
3. 调试 matlab 串口数据发送前，请务必关闭串口助手。

使用 AC6 注意事项

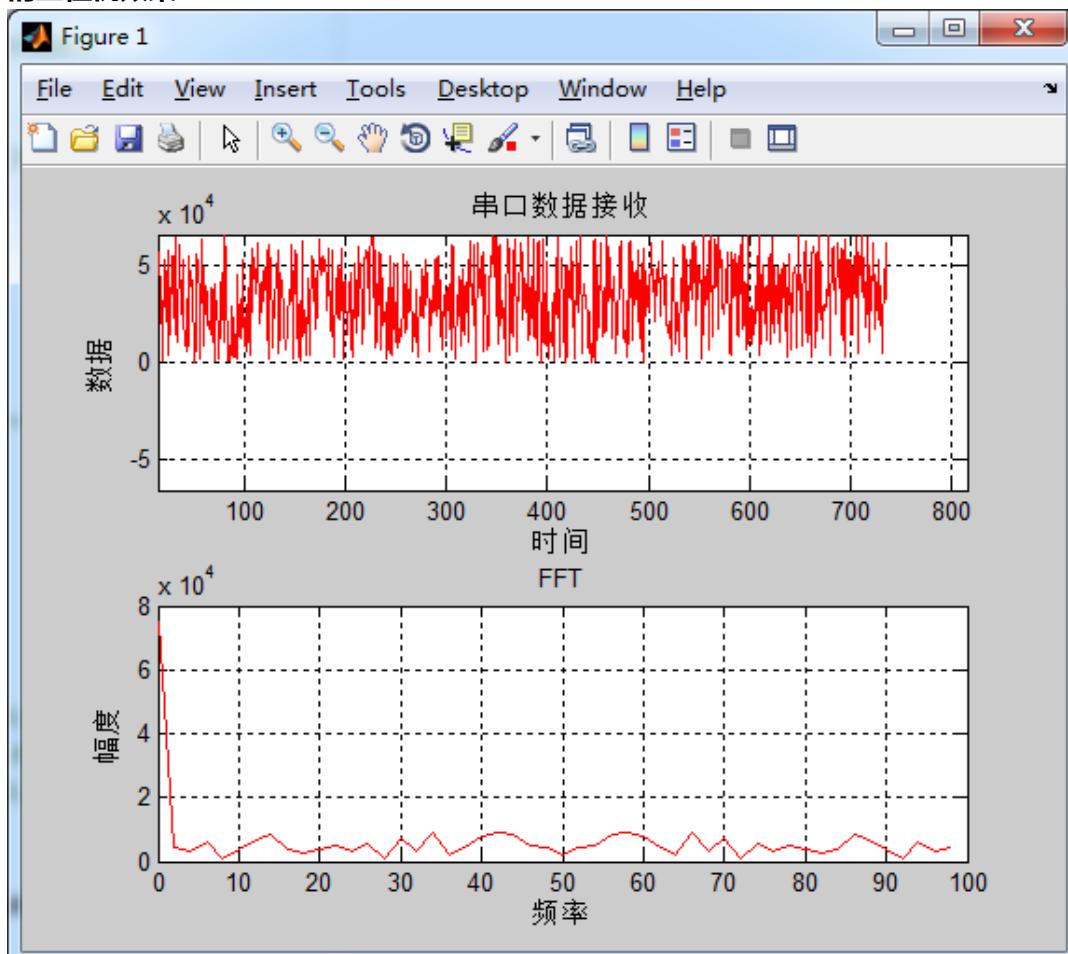
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1

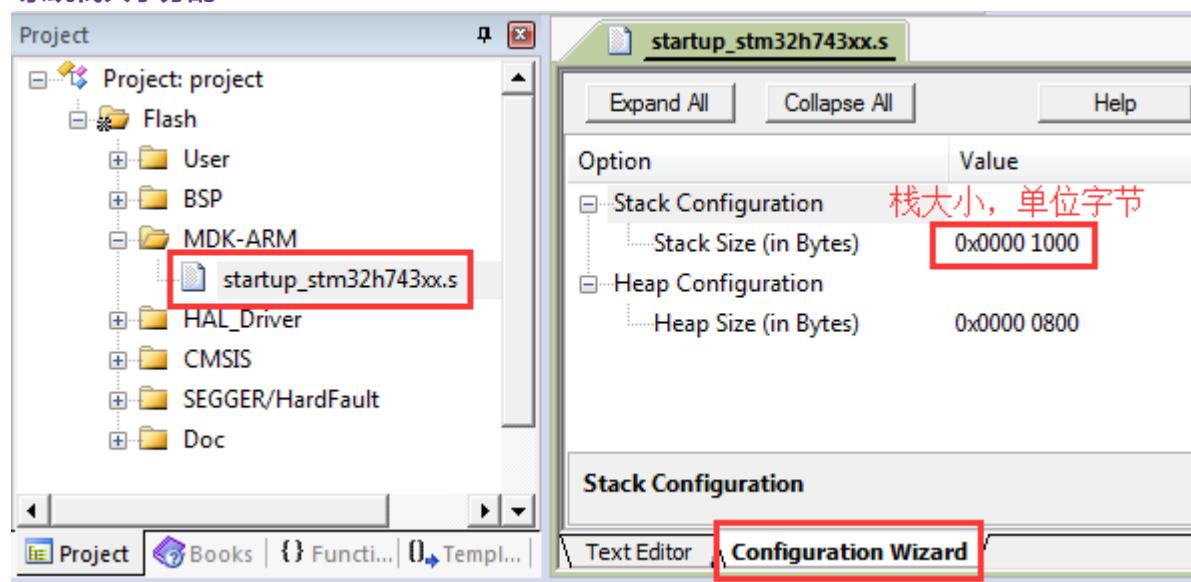


Matlab 的上位机效果:

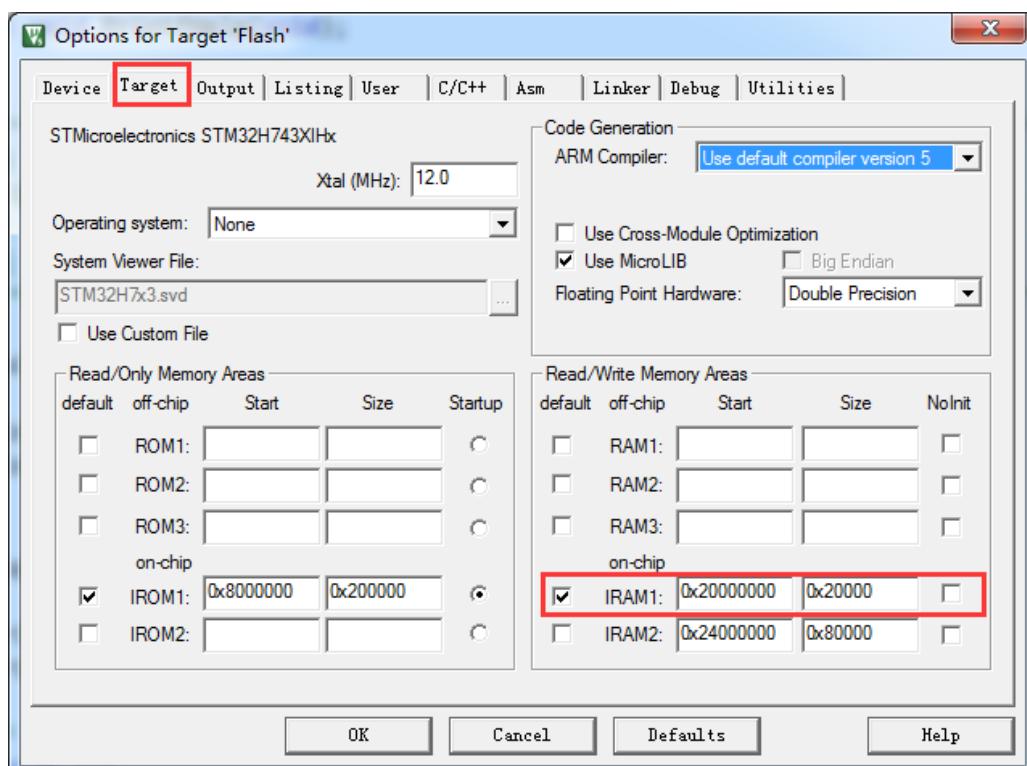


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
*****
```



```
/* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
        - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
        - 设置 NVIV 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
        - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;
```



```
/* 禁止 MPU */
HAL_MPU_Disable();

/* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:



主程序实现如下操作：

- 接收 matlab 发送过来的同步信号，并回一个同步信号后，传输相应的数据过去

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形    参: 无
***** 返  回 值: 错误代码(无需处理)
***** */

int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint8_t read;

    bsp_Init();           /* 硬件初始化 */

    PrintfLogo(); /* 打印例程名称和版本等信息 */
    PrintfHelp(); /* 打印操作提示 */

    bsp_StartAutoTimer(0, 50); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        if (comGetChar(COM1, &read))
        {
            /* 接收到同步帧' $' */
            if (read == 13)
            {
                bsp_LedToggle(4);
                bsp_DelayMS(100);
                Serial_sendDataMATLAB();
            }
        }
    }

    /* 按键滤波和检测由后台 systick 中断服务程序实现，我们只需要调用 bsp_GetKey 读取键值即可。 */
    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:           /* K1 键按下 */
                printf("K1 键按下\r\n");
                break;

            case KEY_UP_K1:            /* K1 键弹起 */
                break;
        }
    }
}
```



```
break;

case KEY_DOWN_K2:           /* K2 键按下 */
    printf("K2 键按下\r\n");
    break;

case KEY_UP_K2:             /* K2 键弹起 */
    printf("K2 键弹起\r\n");
    break;

case KEY_DOWN_K3:           /* K3 键按下 */
    printf("K3 键按下\r\n");
    break;

case KEY_UP_K3:             /* K3 键弹起 */
    printf("K3 键弹起\r\n");
    break;

default:
    /* 其它的键值不处理 */
    break;
}

}

}

}

/* ****
* 函数名: Serial_sendDataMATLAB
* 功能说明: 发送串口数据给 matlab
* 形参: 无
* 返回值: 无
*****
*/
static void Serial_sendDataMATLAB(void)
{
    /* 先发同步信号'$' */
    comSendChar(COM1, 13);

    /* 发送数据, 一共 10 个字节 */
    g_SendData.data1 = rand()%65536;
    g_SendData.data2 = rand()%65536;
    g_SendData.data3 = rand()%65536;
    g_SendData.data4 = rand()%256;
    g_SendData.data5 = rand()%256;
    g_SendData.data6 = rand()%256;
    g_SendData.data7 = rand()%256;
    comSendBuf(COM1, (uint8_t *)&g_SendData, 10);
}
```

9.7 实验例程说明 (IAR)

配套例子：

V7-202_Matlab 的串口通信实现

**实验目的:**

1. 学习 matlab 的串口数据通信。

实验内容:

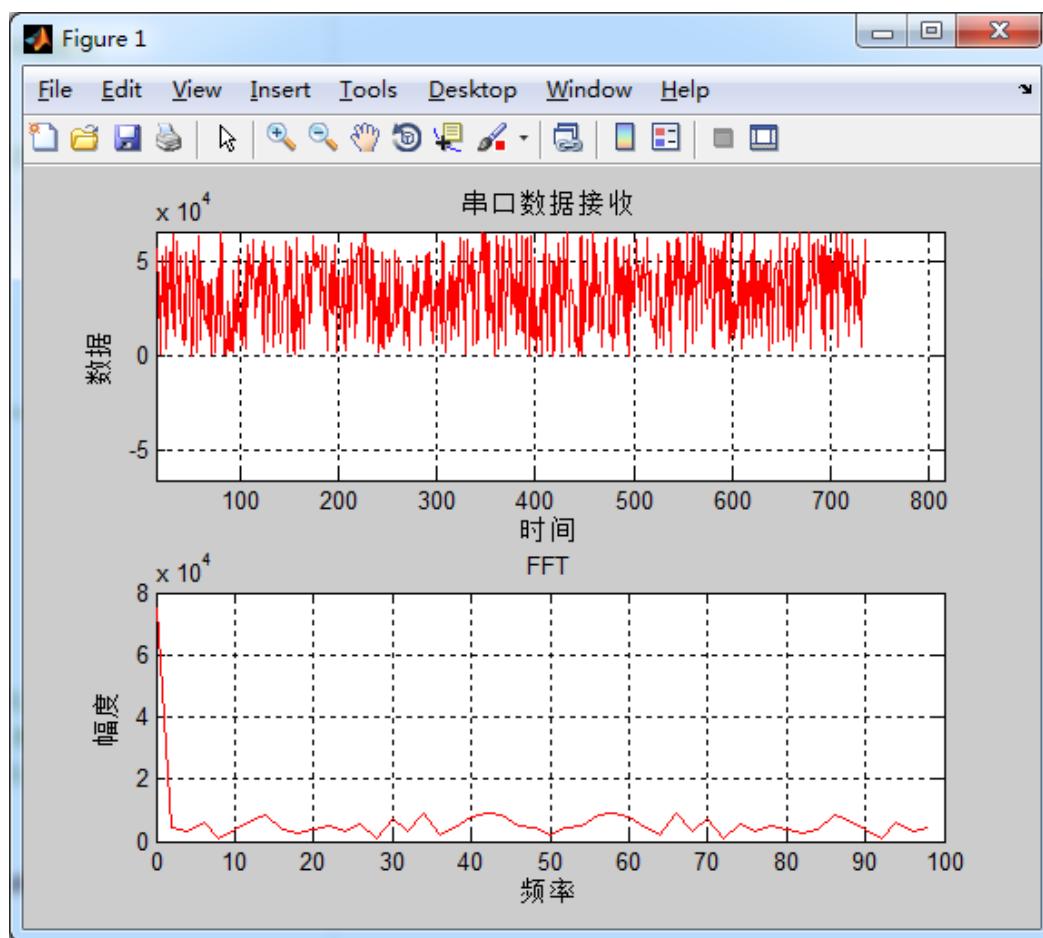
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 请优先运行开发板，然后运行 matlab。
3. 调试 matlab 串口数据发送前，请务必关闭串口助手。

上电后串口打印的信息:

波特率 115200，数据位 8，奇偶校验位无，停止位 1

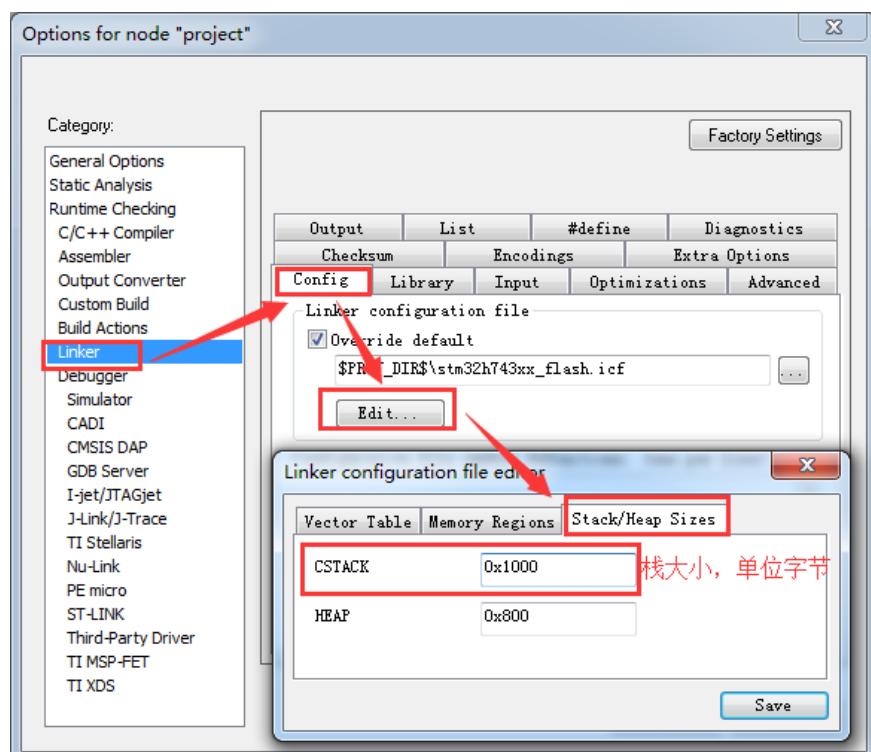
```
Serial-COM1 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM1
*****
CPU : STM32H743XIH6, BGA240, 主频: 400MHz
UID = 34363033 3436510B 801A8007
*****
* 例程名称   : V7-Matlab的串口数据通信
* 例程版本   : 1.0
* 发布日期   : 2019-09-07
* HAL库版本  : V1.3.0 (STM32H7xx HAL Driver)
*
* QQ        : 1295744630
* 旺旺      : armfly
* Email     : armfly@qq.com
* 微信公众号: armfly_com
* 淘宝店    : armfly.taobao.com
* Copyright www.armfly.com 安富莱电子
*****
操作提示:
1. 启动一个自动重装软件定时器，每100ms翻转一次LED2
2. 请优先运行开发板，然后运行matlab
3. 调试matlab串口数据发送前，请务必关闭串口助手
就绪          Serial: COM1 23, 1 23行, 68列 VT100 大写 数字
```

Matlab 的上位机效果:

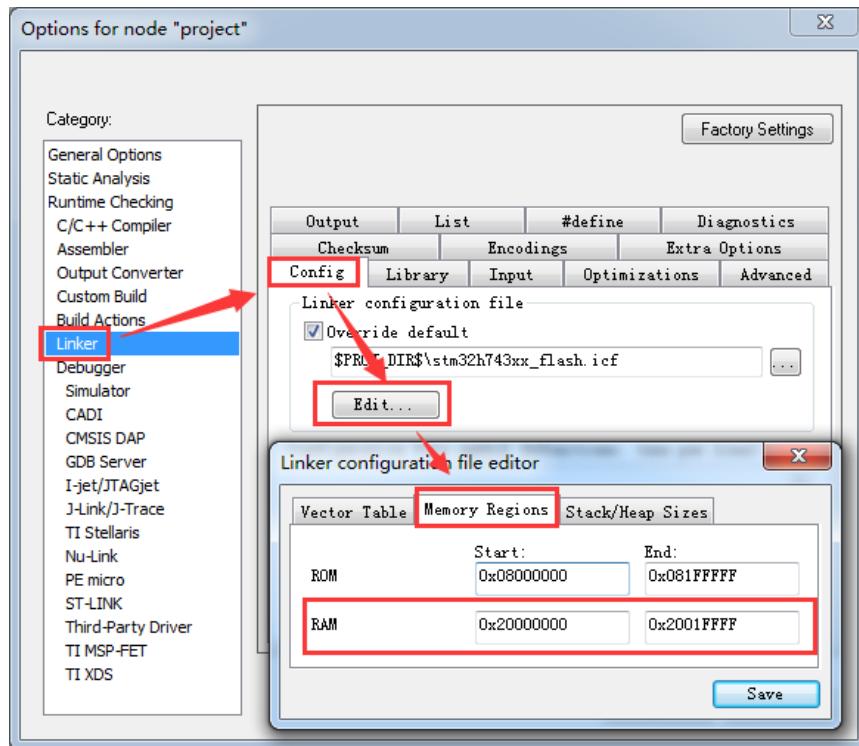


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
     * STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
     * - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
     * - 设置 NVIC 优先级分组为 4。
     */
    HAL_Init();

    /*
     * 配置系统时钟到 400MHz
     * - 切换使用 HSE。
     * - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
     */
}
```



```
SystemClock_Config();

/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
```



```
MPU_InitStruct.Size          = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable    = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable     = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number         = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 接收 matlab 发送过来的同步信号，并回一个同步信号后，传输相应的数据过去

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint8_t read;

    bsp_Init();           /* 硬件初始化 */

    PrintfLogo(); /* 打印例程名称和版本等信息 */
    PrintfHelp(); /* 打印操作提示 */

    bsp_StartAutoTimer(0, 50); /* 启动 1 个 100ms 的自动重装的定时器 */
}
```



```
/* 进入主程序循环体 */
while (1)
{
    bsp_Idle(); /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    if (comGetChar(COM1, &read))
    {
        /* 接收到同步帧'$' */
        if (read == 13)
        {
            bsp_LedToggle(4);
            bsp_DelayMS(100);
            Serial_sendDataMATLAB();
        }
    }

    /* 按键滤波和检测由后台 systick 中断服务程序实现，我们只需要调用 bsp_GetKey 读取键值即可。 */
    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1: /* K1 键按下 */
                printf("K1 键按下\r\n");
                break;

            case KEY_UP_K1: /* K1 键弹起 */
                break;

            case KEY_DOWN_K2: /* K2 键按下 */
                printf("K2 键按下\r\n");
                break;

            case KEY_UP_K2: /* K2 键弹起 */
                printf("K2 键弹起\r\n");
                break;

            case KEY_DOWN_K3: /* K3 键按下 */
                printf("K3 键按下\r\n");
                break;

            case KEY_UP_K3: /* K3 键弹起 */
                printf("K3 键弹起\r\n");
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
```



```
}

/*
***** 函数名: Serial_sendDataMATLAB
***** 功能说明: 发送串口数据给 matlab
***** 形    参: 无
***** 返  回 值: 无
*****
*/
static void Serial_sendDataMATLAB(void)
{
    /* 先发同步信号'$' */
    comSendChar(COM1, 13);

    /* 发送数据, 一共 10 个字节 */
    g_SendData.data1 = rand()%65536;
    g_SendData.data2 = rand()%65536;
    g_SendData.data3 = rand()%65536;
    g_SendData.data4 = rand()%256;
    g_SendData.data5 = rand()%256;
    g_SendData.data6 = rand()%256;
    g_SendData.data7 = rand()%256;
    comSendBuf(COM1, (uint8_t *)&g_SendData, 10);
}
```

9.8 总结

本章讲解的例程非常实用，需要大家熟练掌握。



第10章 Matlab 的 WIFI 通信实现

本章节主要为大家讲解 Matlab 的 WIFI 方式波形数据传输和后期数据分析功能，非常实用。

10.1 初学者重要提示

10.2 程序设计框架

10.3 实验操作步骤

10.4 下位机 STM32H7 程序设计

10.5 上位机 Matlab 程序设计

10.6 实验例程说明 (MDK)

10.7 实验例程说明 (IAR)

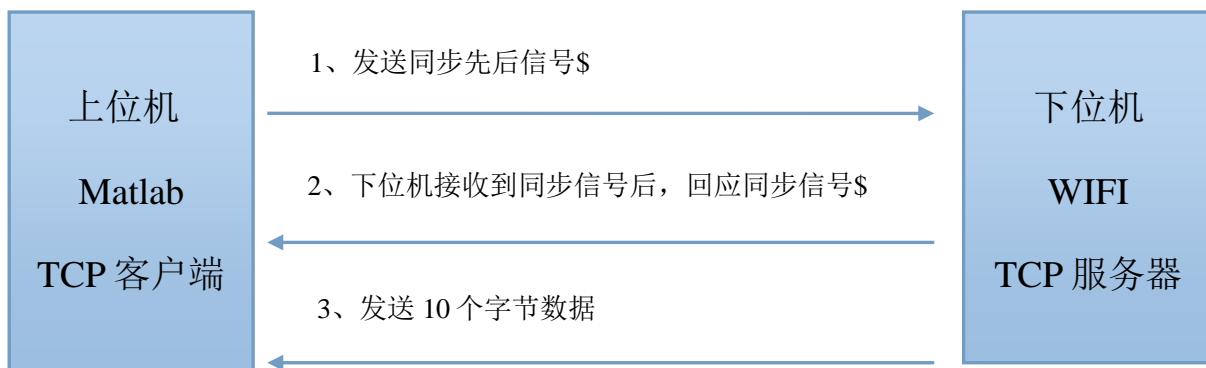
10.8 总结

10.1 初学者重要提示

- ◆ 测试本章节例程注意事项。
 - 请优先运行开发板，然后运行 matlab。
- ◆ 测试使用前，务必优先看本章第 3 小节。

10.2 程序设计框架

WIFI 模块用的 ESP8266，串口通信方式。Matlab 端是作为 TCP 客户端，而 WIFI 模块是作为 TCP 服务器。上位机和下位机的程序设计框架如下：

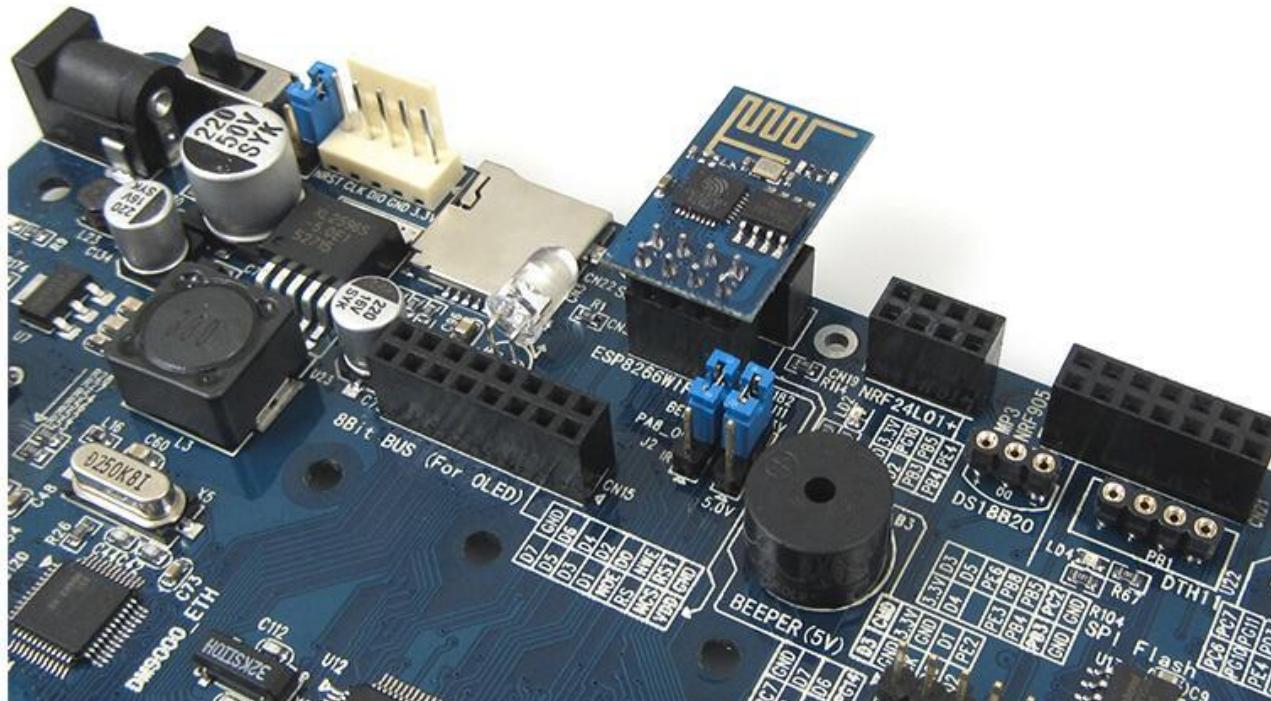


10.3 实验操作步骤

由于要用到 WiFi 模块，非常有必要把实验操作步骤说一下，主要是考虑到一些用户没有用过 WiFi。

注意：务必要保证 WiFi 模块和电脑在同一个局域网内。

10.3.1 第 1 步，WiFi 模块的插入位置



10.3.2 第 2 步，串口打印的操作说明

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

注：特别注意截图里面的注释说明。



Serial-COM1 - SecureCRT

文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)

Serial-COM1

操作提示：
ESP8266模块 STM32-V7开发板
UTXD --- PC7/USART6_RX CN16插座
GND --- GND CN6插座
CH_PD --- PIO0(控制模块掉电) CN16插座
GPIO2
GPIO16
GPIO00
VCC --- 3.3 (供电) CN6插座
URXD --- PG14/USART6_TX CN16插座

【按键操作】
K1键 : 列举AP
K2键 : 加入AP
K3键 : 9600波特率切换到115200, 并设置为Station模式
摇杆上键 : AT+CIFSR获取本地IP地址
摇杆下键 : AT+CIPSTATUS获得IP连接状态
摇杆左键 : AT+CIPSTART建立TCP服务器
摇杆右键 : AT+CIPSEND向TCP客户端发送数据

【SecureCRT串口设置】
波特率: 115200
会话选项 - 终端 - 仿真 - 模式, 勾选新行模式(w)

【1】正在给ESP8266模块上电...(波特率: 74880bsp)
ets Jan 8 2013,rst cause:2, boot mode:(3,1)
load 0x40100000, len 25052, room 16
tail 12
checksum 0x0b
ho 0 tail 12 room 4
load 0x3ffe8000, len 3312, room 12
tail 4
checksum 0x53
load 0x3ffe8cf0, len 6576, room 4
tail 12
checksum 0xd
csum 0xd

[Vendor:www.ai-thinker.com Version:0.9.2.4]

ready
【2】上电完成。波特率: 115200bsp

AT
OK
【3】模块应答AT成功

有些模块通过74880波特率打印logo信息完毕后, 可能会使用9600波特率, 这里是以115200测试下AT指令看看是否会成功, 如果不是115200, 用户按下K3可以修改波特率

就绪 Serial: COM1 53, 1 53行, 91列 VT100 大写 数字

10.3.3 第3步, K1 按键按下后, 会打印附近的 WiFi 热点

特别注意自己用的 WiFi 热点是否在识别出来的 WiFi 列表里面。



The screenshot shows a list of WiFi networks found by the AT+CWLAP command. The list includes various network names and their MAC addresses. The window title is "Serial-COM1 - SecureCRT". The status bar at the bottom right shows "Serial: COM1 30, 1 30行, 90列 VT100 大写 数字".

```
AT+CWLAP
+CW LAP:(4,"901",-90,"bc:d1:77:da:86:12",1)
+CW LAP:(4,"MERCURY_8D20D0",-76,"c0:61:18:8d:20:d0",1)
+CW LAP:(4,"ChinaNet-qnkW",-74,"40:f4:20:f0:18:22",1)
+CW LAP:(4,"801",-64,"50:0f:f5:0f:18:50",1)
+CW LAP:(4,"TP-LINK_6E64",-85,"30:fc:68:1b:6e:64",1)
+CW LAP:(4,"FANGWU901",-83,"fc:d7:33:04:78:c4",1)
+CW LAP:(4,"CMCC-Kmc3",-95,"3c:57:4f:fb:31:f8",1)
+CW LAP:(4,"@PHICOMM_20",-90,"2c:b2:1a:f4:f1:22",1)
+CW LAP:(3,"ManHuaZhu",-69,"02:21:46:a8:53:9f",6)
+CW LAP:(4,"SANY",-63,"34:96:72:26:f5:de",6)
+CW LAP:(4,"701",-74,"e4:d3:32:92:20:58",6)
+CW LAP:(4,"ziroom3C",-91,"4c:e1:73:01:4c:2e",2)
+CW LAP:(4,"Netcore_7378CB",-21,"08:10:78:73:78:cb",6)
+CW LAP:(4,"liuxiaobaoaimama",-87,"c8:3a:35:1a:55:98",5)
+CW LAP:(4,"ChinaNet-qzgf",-76,"54:66:6c:c0:a1:d0",7)
+CW LAP:(4,"ChinaNet-vvMx",-81,"54:66:6c:c0:5c:00",8)
+CW LAP:(4,"TP-LINK_B2A6",-89,"24:69:68:95:b2:a6",6)
+CW LAP:(4,"CMCC-uwDN",-71,"10:3d:3e:10:4c:78",10)
+CW LAP:(4,"CMCC-eAu6",-77,"00:cf:c0:54:18:f8",11)
+CW LAP:(4,"TP-LINK_1102",-89,"14:75:90:0d:63:1e",11)
+CW LAP:(4,"TP-803",-84,"d8:15:0d:50:a1:1a",11)
+CW LAP:(4,"TP-LINK_57EE",-92,"14:75:90:77:57:ee",11)
+CW LAP:(4,"MERCURY_7687",-88,"48:8a:d2:9a:76:87",12)
+CW LAP:(4,"MERCURY_191C",-93,"1c:60:de:f7:19:1c",6)
+CW LAP:(4,"601",-93,"bc:54:fc:dc:3c:2c",13)

OK
```

10.3.4 第4步，K2 按键按下后，加入其中一个 WIFI 热点

本章配套程序的 main.c 文件有如下一段代码：

```
case KEY_DOWN_K2: /* K2 键按下，加入某个 WIFI 网络*/
    g_TCPServerOk = 0;
    ret = ESP8266_JoinAP("Netcore_7378CB", "512464265", 15000);
    if(ret == 1)
    {
        printf("\r\nJoinAP Success\r\n");
    }
    else
    {
        printf("\r\nJoinAP fail\r\n");
    }
    break;
```

Netcore_7378CB 是热点名，而 512464265 是密码。需要大家根据自己的情况设置。

加入一次即可，以后上电会自动加入。



The screenshot shows the SecureCRT application window titled "Serial-COM1 - SecureCRT". The menu bar includes "文件(F)", "编辑(E)", "查看(V)", "选项(O)", "传输(T)", "脚本(S)", "工具(L)", and "帮助(H)". The toolbar contains icons for file operations like Open, Save, Print, and Copy/Paste. The main terminal window displays serial communication logs:

```
+CWLAP:(4,"TP-LINK_1102",-89,"14:75:90:0d:63:1e",11)
+CWLAP:(4,"TP-803",-84,"d8:15:0d:50:a1:1a",11)
+CWLAP:(4,"TP-LINK_57EE",-92,"14:75:90:77:57:ee",11)
+CWLAP:(4,"MERCURY_7687",-88,"48:8a:d2:9a:76:87",12)
+CWLAP:(4,"MERCURY_191C",-93,"1c:60:de:f7:19:1c",6)
+CWLAP:(4,"601",-93,"bc:54:fc:dc:3c:2c",13)

OK
AT+CWJAP="Netcore_7378CB","512464265"
OK
JoinAP Success
```

At the bottom, status information is shown: "就绪" (Ready), "Serial: COM1 14, 1 14行, 75列 VT100", and "大写 数字" (Caps Lock, Num Lock).

10.3.5 第5步，摇杆上键打印 WIFI 获取的 IP 地址

这个 IP 地址要记住，因为 Matlab 上位机要使用。

The screenshot shows the SecureCRT application window titled "Serial-COM1 - SecureCRT". The menu bar and toolbar are identical to the previous screenshot. The main terminal window displays serial communication logs, similar to the previous one, but with the IP address "192.168.1.5" highlighted in red:

```
+CWLAP:(4,"MERCURY_191C",-93,"1c:60:de:f7:19:1c",6)
+CWLAP:(4,"601",-93,"bc:54:fc:dc:3c:2c",13)

OK
AT+CWJAP="Netcore_7378CB","512464265"
OK
JoinAP Success
AT+CTFSR
192.168.1.5

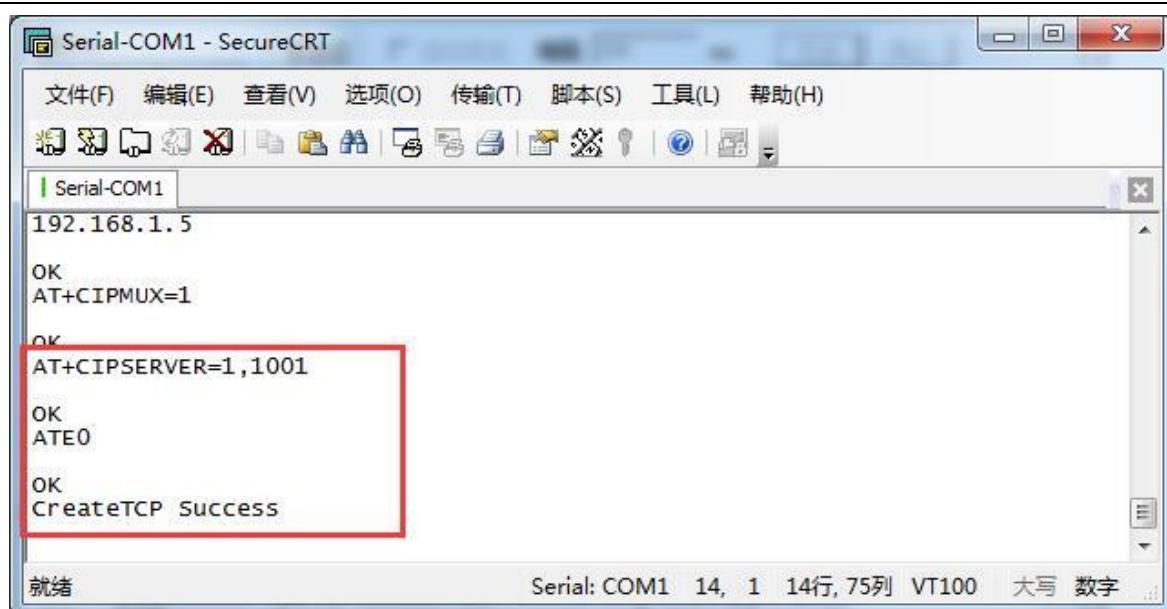
OK
■
```

At the bottom, status information is shown: "就绪" (Ready), "Serial: COM1 14, 1 14行, 75列 VT100", and "大写 数字" (Caps Lock, Num Lock).

当前从 WiFi 热点获取的 IP 是 192.168.1.5。

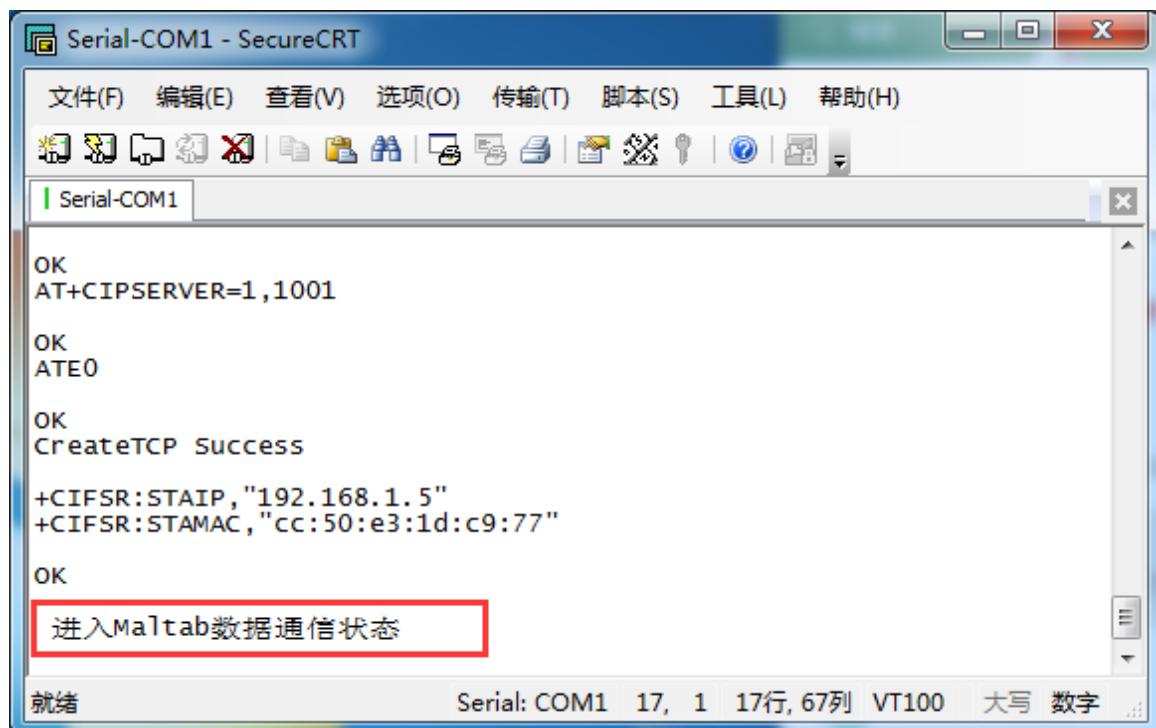
10.3.6 第6步，摇杆左键创建一个 TCP 服务器，端口号 1001

打印 CreateTCP Success 的话，表示创建成功：



10.3.7 第7步，摇杆右键进入 Matlab 通信状态

这里只是设置一下状态标志，方便进入 Matlab 通信程序：



10.3.8 第8步，修改 Matlab 上位机程序的服务器地址

将第5步获取的IP地址填写到上位机程序：

```
%*****  
%连接远程服务器，IP地址192.168.1.5，端口号1001。  
t = tcpclient('192.168.1.5', 1001);
```



10.3.9 第9步，最有一步，运行 matlab 上位机程序

M 文件的程序代码在例子 V7-205_Matlab 的 WIFI 波形刷新和数据分析 m 文件里面。M 文件的运行方法在第 4 章的 4.2 小节有详细说明。

注意，测试程序时，先将板子上电，也就是先把服务器创建好，然后运行 matlab 程序。

10.4 下位机 STM32H7 程序设计

STM32H7 端的程序设计思路。

10.4.1 第1步，发送的数据格式

数据格式比较简单，创建了 5 个 uint16_t 类型的数据：

```
uint16_t SendDATA[5];
```

10.4.2 第2步，接收同步信号\$并发送数据

Matlab 发送同步信号\$ (ASCII 编码值是 36) 给开发板。

```
int main(void)
{
    /* 省略未写，仅留下关键代码 */

    /* 进入主程序循环体 */
    while (1)
    {

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        if (g_TCPServerOk == 1)
        {
            cmd_len = ESP8266_RxNew(cmd_buf, &tcpid);
            if (cmd_len >0)
            {
                printf("\r\n接收到数据长度 = %d\r\n远程 ID =%d\r\n数据内容=%s\r\n",
                       cmd_len, tcpid, cmd_buf);

                /* 检索 matlab 发送过来的同步帧字符$，对应的 ASCII 数值是 36 */
                if (strchr((char *)cmd_buf, 36))
                {
                    /* 回复同步帧$ */
                    ESP8266_SendTcpUdp(tcpid, (uint8_t *)&SyncData, 1);
                    bsp_DelayMS(10);
                    SendDATA[0] = rand() %65536;
                    SendDATA[1] = rand() %65536;
                    SendDATA[2] = rand() %65536;
                    SendDATA[3] = rand() %65536;
                    SendDATA[4] = rand() %65536;
                }
            }
        }
    }
}
```



```
/* 发送数据, 10 个字节 */
ESP8266_SendTcpUdp(tcpid, (uint8_t *)SendDATA, 10);
printf("找到了相应的字符串\r\n");
}
else
{
    printf("没有找到了相应的字符串\r\n");
}
}
}
}
```

通过函数 `ESP8266_RxNew` 获取串口接收到的数据，如果数值是 36 (对应的 ASCII 字符是\$)，说明接收到 Matlab 发送过来的同步信号了。然后再通过函数 `ESP8266_SendTcpUdp` 回应一个同步字符\$。

回复完毕后，迟了 10ms 再发数据给 matlab，主要是因为 matlab 的波形刷新有点快，程序这里每发送给 matlab 一次数据，matlab 就会刷新一次，10ms 就相当于 100Hz 的刷新率，也会有一定的闪烁感。

通过这两步就完成了 STM32H7 端的程序设计。

10.5 上位机 Matlab 程序设计

Matlab 端的程序设计要略复杂些，需要大家理解 matlab 端的 API。具体说明可以看如下地址：

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=94598>。

10.5.1 第 1 步，配置 TCP 客户端

下面操作是配置TCP客户端连接服务器：

```
%*****
%连接远程服务器, IP 地址 192.168.1.5, 端口号 1001。
t = tcpclient('192.168.1.5', 1001);
```

务必要根据本章3.5小节获取的IP地址进行配置。

10.5.2 第 2 步，相关变量设置

程序里面对这些变量的注释已经比较详细：

```
AxisMax = 65536; %坐标轴最大值
AxisMin = -65536; %坐标轴最小值
window_width = 800; %窗口宽度

g_Count = 0; %接收到的数据计数
SOF = 0; %同步帧标志
AxisValue = 1; %坐标值

RecDataDisp = zeros(1, 100000); %开辟 100000 个数据单元, 用于存储接收到的数据。
RecData = zeros(1, 100); %开辟 100 个数据单元, 用于数据处理。
Axis = zeros(1, 100000); %开辟 100000 个数据单元, 用于 X 轴。

window = window_width * (-0.9); %窗口 X 轴起始坐标
axis([window, window + window_width, AxisMin, AxisMax]); %设置窗口坐标范围
```



```
%子图 1 显示串口上传的数据
```

```
subplot(2, 1, 1);  
grid on;  
title('串口数据接收');  
xlabel('时间');  
ylabel('数据');
```

```
%子图 2 显示波形的幅频响应
```

```
subplot(2, 1, 2);  
grid on;  
title('FFT');  
xlabel('频率');  
ylabel('幅度');  
  
Fs = 100; % 采样率  
N = 50; % 采样点数  
n = 0:N-1; % 采样序列  
f = n * Fs / N; % 真实的频率
```

这里有以下几点需要大家了解：

◆ 变量RecDataDisp, RecData和Axis

这几个变量专门开辟好了数据空间，防止matlab警告和刷新波形慢的问题，大家根据需要可以进行加大。

◆ 采样率Fs = 100和采样点数N = 50

这个地方要根据实际的情况进行设置。

10.5.3 第3步，数据同步部分

这部分代码比较关键，matlab先发送同步信号\$出去，然后等待开发板回复同步信号\$，并读取本次通信的数据。

```
%设置同步信号标志，=1表示接收到下位机发送的同步帧  
SOF = 0;  
  
%发送同步帧，36 对应字符'$'  
data(1) = 36;  
write(t, data(1));  
  
%读取返回值  
RecData = read(t, 1, 'uint8');  
  
%如果检索到$，读取 10 个字节的数据，也就是 5 个 uint16 的数据  
if (RecData == 36)  
    RecData = read(t, 5, 'uint16');  
    SOF = 1;  
    StartData = 0;  
end
```

这里有以下几点需要大家了解：

◆ 函数write(t, data(1))

用于发送同步信号\$（ASCII值是36）。

◆ 函数read(t,1,'uint8')



读取1个uint8类型的数据，也就是1个字节。

◆ 函数if(RecData == 36)

检查接收到的数据是否是同步信号\$。如果是\$，继续读取10个字节的数据，也就是5个uint16的数据。

10.5.4 第4步，显示串口上传的数据

下面 matlab 的数据显示波形

```
%更新接收到的数据波形
if(SOF == 1)
    %更新数据
    RecDataDisp(AxisValue) = RecData(1);
    RecDataDisp(AxisValue + 1) = RecData(2);
    RecDataDisp(AxisValue + 2) = RecData(3);
    RecDataDisp(AxisValue + 3) = RecData(4);
    RecDataDisp(AxisValue + 4) = RecData(5);

    %更新 X 轴
    Axis(AxisValue) = AxisValue;
    Axis(AxisValue + 1) = AxisValue + 1;
    Axis(AxisValue + 2) = AxisValue + 2;
    Axis(AxisValue + 3) = AxisValue + 3;
    Axis(AxisValue + 4) = AxisValue + 4;

    %更新变量
    AxisValue = AxisValue + 5;
    g_Count = g_Count + 5;

    %绘制波形
    subplot(2,1,1);
    plot(Axis(1:AxisValue-1), RecDataDisp(1:AxisValue-1), 'r');
    window = window + 5;
    axis([window, window + window_width, AxisMin, AxisMax]);
    grid on;
    title('串口数据接收');
    xlabel('时间');
    ylabel('数据');
    drawnow
end
```

这里有以下几点需要大家了解：

◆ 数组RecDataDisp, RecData和Axis

这里要尤其注意，matlab的数组索引是从1开始的，也是开头直接定义AxisValue = 1的原因。

◆ 函数plot

这里plot的实现尤其重要，务必要注意坐标点和数值个数要匹配。

10.5.5 第5步，FFT 数据展示

FFT部分会在在后面章节为大家详细讲解，这里也做个说明，这里是每接收够50个数据，做一次FFT：

```
if(g_Count== 50)
    subplot(2,1,2);
    %对原始信号做 FFT 变换
```



```
y = fft(RecDataDisp(AxisValue-50:AxisValue-1), 50);
```

```
%求 FFT 转换结果的模值
```

```
Mag = abs(y)*2/N;
```

```
%绘制幅频相应曲线
```

```
plot(f, Mag, 'r');  
grid on;  
title('FFT');  
xlabel('频率');  
ylabel('幅度');  
g_Count = 0;  
drawnow  
end
```

10.6 实验例程说明 (MDK)

配套例子：

V7-204_Matlab 的 WIFI 通信实现

实验目的：

1. 学习 matlab 的串口数据通信。

实验内容：

1. K1 键 :列举 AP, 就是 WIFI 热点;
2. K2 键 :加入 AP, 就是加入 WIFI 热点;
3. K3 键 :9600 波特率切换到 115200, 并设置为 Station 模式;
4. 摆杆上键 :AT+CIFSR 获取本地 IP 地址;
5. 摆杆下键 :AT+CIPSTATUS 获得 IP 连接状态;
6. 摆杆左键 :AT+CIPSTART 建立 TCP 服务器;
7. 摆杆右键 :进入 Matlab 数据通信状态;

使用 AC6 注意事项

特别注意附件章节 C 的问题。

上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1



Serial-COM1 - SecureCRT

文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)

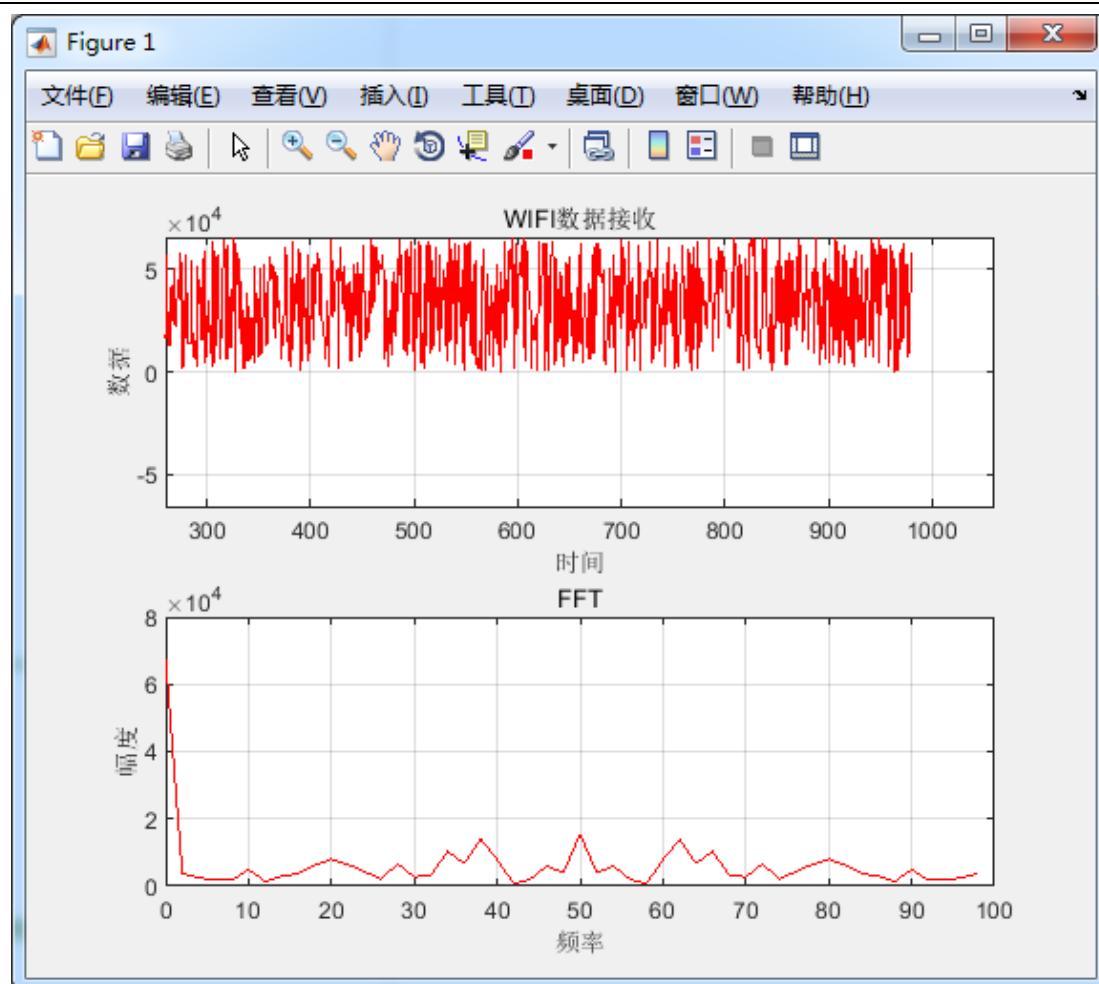
Serial-COM1

```
CPU : STM32H743XIH6, BGA240, 主频: 400MHz
UID = 34343436 3038510A 0040002D
*****
* 例程名称 : V7-Matlab的WIFI通信实现
* 例程版本 : 1.0
* 发布日期 : 2019-09-13
* HAL库版本 : V1.3.0 (STM32H7xx HAL Driver)
*
* QQ : 1295744630
* 旺旺 : armfly
* Email : armfly@qq.com
* 微信公众号: armfly_com
* 淘宝店: armfly.taobao.com
* Copyright www.armfly.com 安富莱电子
*****
操作提示:
ESP8266模块 STM32-V7开发板
UTXD --- PC7/USART6_RX CN16插座
GND --- GND CN6插座
CH_PD --- PIO(控制模块掉电) CN16插座
GPIO2
GPIO16
GPIO0
VCC --- 3.3 (供电) CN6插座
URXD --- PG14/USART6_TX CN16插座

【按键操作】
K1键 : 列举AP
K2键 : 加入AP
K3键 : 9600波特率切换到115200, 并设置为Station模式
摇杆上键 : AT+CIFSR获取本地IP地址
摇杆下键 : AT+CIPSTATUS获得IP连接状态
摇杆左键 : AT+CIPSTART建立TCP服务器
摇杆右键 : AT+CIPSEND向TCP客户端发送数据
```

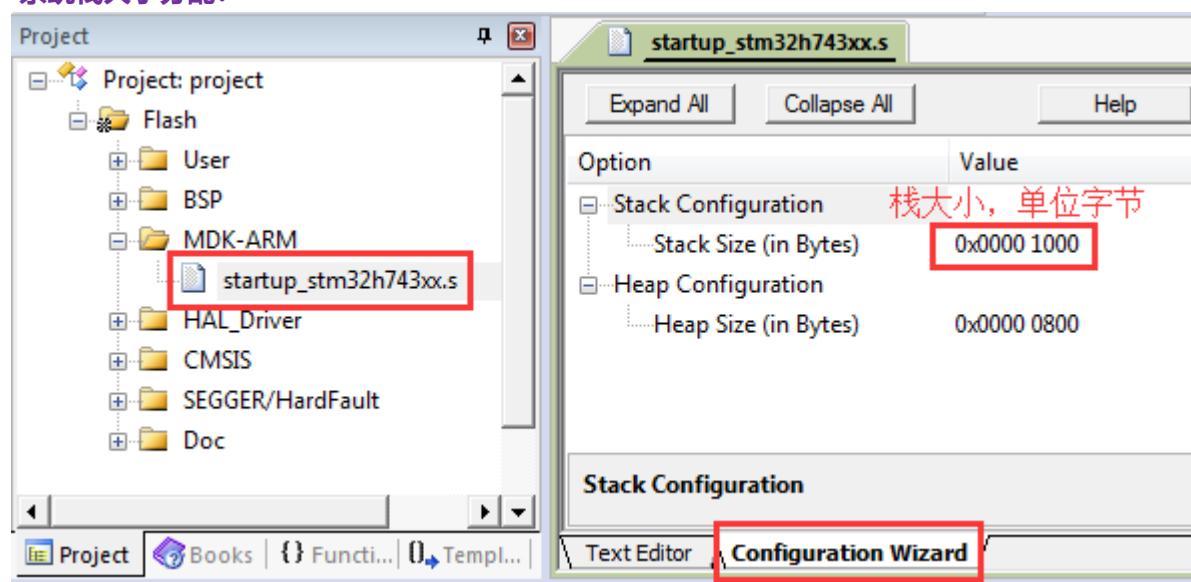
就绪 Serial: COM1 37, 1 37行, 73列 VT100 大写 数字

Matlab 的上位机效果:

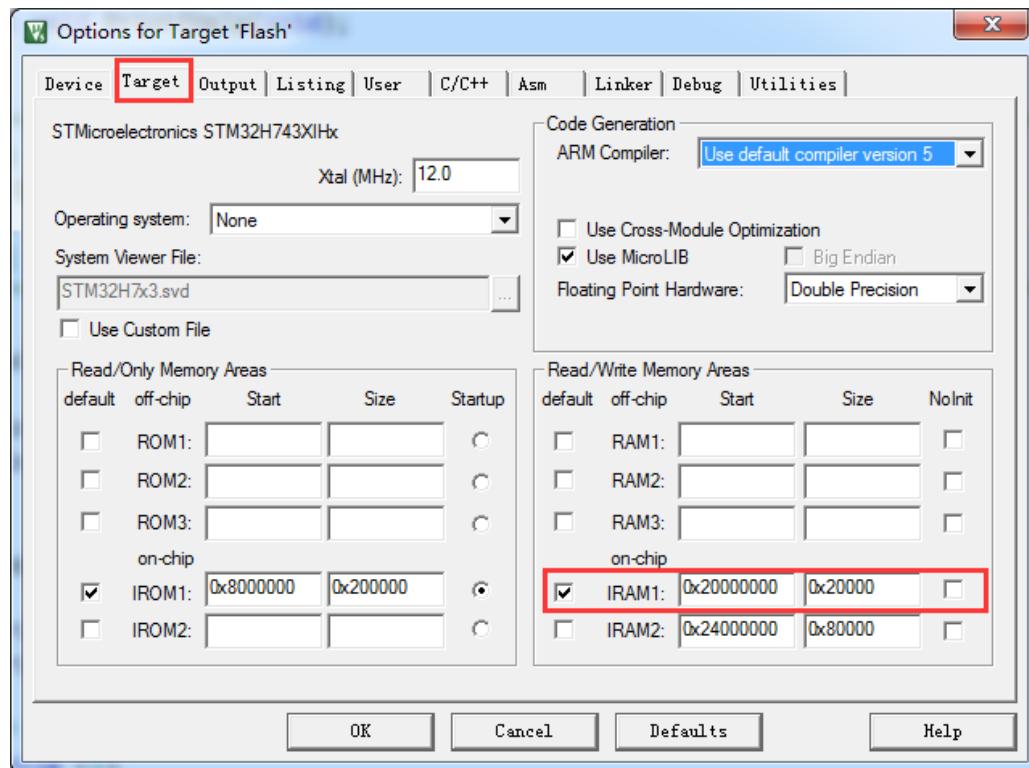


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */

bsp_InitESP8266(); /* 配置 ESP8266 模块相关的资源 */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
```



```
MPU_InitStruct.BaseAddress      = 0x60000000;
MPU_InitStruct.Size            = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable    = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable     = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable     = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number          = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec     = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 接收 matlab 发送过来的同步信号，并回一个同步信号后，传输相应的数据过去

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint8_t ucValue;
    uint8_t ret;
    uint8_t SyncData = 36;
    uint16_t SendDATA[5];

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */
```



```
PrintfHelp(); /* 打印操作提示信息 */

/* 模块上电 */
printf("\r\n【1】正在给 ESP8266 模块上电... (波特率: 74880bsp)\r\n");
ESP8266_PowerOn();

printf("\r\n【2】上电完成。波特率: 115200bsp\r\n");

/* 检测模块波特率是否为 115200 */
ESP8266_SendAT("AT");
if (ESP8266_WaitResponse("OK", 50) == 1)
{
    printf("\r\n【3】模块应答 AT 成功\r\n");
    bsp_DelayMS(1000);
}
else
{
    printf("\r\n【3】模块无应答, 请按 K3 键修改模块的波特率为 115200\r\n");
    bsp_DelayMS(1000);
}

g_TCPServerOk = 0;

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle(); /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    /* 进入 Matlab 通信状态执行下面程序 */
    if (g_TCPServerOk == 1)
    {
        cmd_len = ESP8266_RxNew(cmd_buf, &tcpid);
        if (cmd_len > 0)
        {
            printf("\r\n接收到数据长度 = %d\r\n远程 ID =%d\r\n数据内容=%s\r\n",
                   cmd_len, tcpid, cmd_buf);

            /* 检索 matlab 发送过来的同步帧字符$, 对应的 ASCII 数值是 36 */
            if (strchr((char *)cmd_buf, 36))
            {
                /* 回复同步帧$ */
                ESP8266_SendTcpUdp(tcpid, (uint8_t *)&SyncData, 1);
                bsp_DelayMS(10);
                SendDATA[0] = rand() % 65536;
                SendDATA[1] = rand() % 65536;
                SendDATA[2] = rand() % 65536;
                SendDATA[3] = rand() % 65536;
                SendDATA[4] = rand() % 65536;
            }
        }
    }
}
```



```
/* 发送数据, 10 个字节 */
ESP8266_SendTcpUdp(tcpid, (uint8_t *)SendDATA, 10);
printf("找到了相应的字符串\r\n");
}
else
{
    printf("没有找到了相应的字符串\r\n");
}
}

/* 未进入 Matlab 通信状态执行下面程序 */
else
{
    /* 从 WIFI 收到的数据发送到串口 1 */
    if (comGetChar(COM_ESP8266, &ucValue))
    {
        comSendChar(COM1, ucValue);
    }
    /* 将串口 1 的数据发送到 8266 模块 */
    if (comGetChar(COM1, &ucValue))
    {
        comSendChar(COM_ESP8266, ucValue);
    }
}

ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
if (ucKeyCode != KEY_NONE)
{
    switch (ucKeyCode)
    {
        case KEY_DOWN_K1:           /* K1 键按下, 列举当前的 WIFI 热点 */
            g_TCPServerOk = 0;
            ESP8266_SendAT("AT+CWLAP");
            break;

        case KEY_DOWN_K2:           /* K2 键按下, 加入某个 WIFI 网络*/
            g_TCPServerOk = 0;
            //ESP8266_SendAT("AT+CWJAP=\\"Netcore_7378CB\\",\\"512464265\\"");
            ret = ESP8266_JoinAP("Netcore_7378CB", "512464265", 15000);
            if(ret == 1)
            {
                printf("\r\nJoinAP Success\r\n");
            }
            else
            {
                printf("\r\nJoinAP fail\r\n");
            }

            break;

        case KEY_DOWN_K3:           /* K3 键-9600 波特率切换到 115200 */
            g_TCPServerOk = 0;
            ESP8266_9600to115200();
            break;

        case JOY_DOWN_U:           /* 摆杆上键, AT+CIFSR 获取本地 IP 地址 */
            g_TCPServerOk = 0;
```



```
ESP8266_SendAT("AT+CIFSR");
break;

case JOY_DOWN_D:                                /* 摆杆下键 AT+CIPSTATUS 获得 IP 连接状态 */
g_TCPServerOk = 0;
ESP8266_SendAT("AT+CIPSTATUS");
break;

case JOY_DOWN_L:                                /* 摆杆左键按下，创建 TCP 服务器 */
g_TCPServerOk = 0;
ret = ESP8266_CreateTCPServer(1001);
if(ret == 1)
{
    printf("\r\nCreateTCP Success\r\n");
}
else
{
    printf("\r\nCreateTCP fail\r\n");
}
break;

case JOY_DOWN_R:                                /* 摆杆右键按下，进入 Matlab 数据通信状态 */
g_TCPServerOk = 1;
printf("\r\n 进入 Matlab 数据通信状态 \r\n");
break;

case JOY_DOWN_OK:                               /* 摆杆 OK 键按下，创建 WIFI 热点 */
g_TCPServerOk = 0;
#if 0
ESP8266_SendAT("AT+CIPSTART=\"TCP\", \"WWW.ARMFLY.COM\", 80");
#endif

#if 0
{
    char ip[20], mac[32];
    ESP8266_GetLocalIP(ip, mac);
    printf("ip=%s, mac=%s\r\n", ip, mac);
}
#endif

#if 1
ESP8266_SetWiFiMode(3);
ESP8266_SendAT("AT+CWSAP=\"ESP8266\", \"1234567890\", 1, 3");
#endif
break;

default:
/* 其他的键值不处理 */
break;
}

}

}
```



10.7 实验例程说明 (IAR)

配套例子：

V7-204_Matlab 的 WIFI 通信实现

实验目的：

1. 学习 matlab 的串口数据通信。

实验内容：

1. K1 键 :列举 AP, 就是 WIFI 热点;
2. K2 键 :加入 AP, 就是加入 WIFI 热点;
3. K3 键 :9600 波特率切换到 115200,并设置为 Station 模式;
4. 摆杆上键 :AT+CIFSR 获得本地 IP 地址;
5. 摆杆下键 :AT+CIPSTATUS 获得 IP 连接状态;
6. 摆杆左键 :AT+CIPSTART 建立 TCP 服务器;
7. 摆杆右键 :进入 Maltab 数据通信状态;

使用 AC6 注意事项

特别注意附件章节 C 的问题。

上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1



Serial-COM1 - SecureCRT

文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)

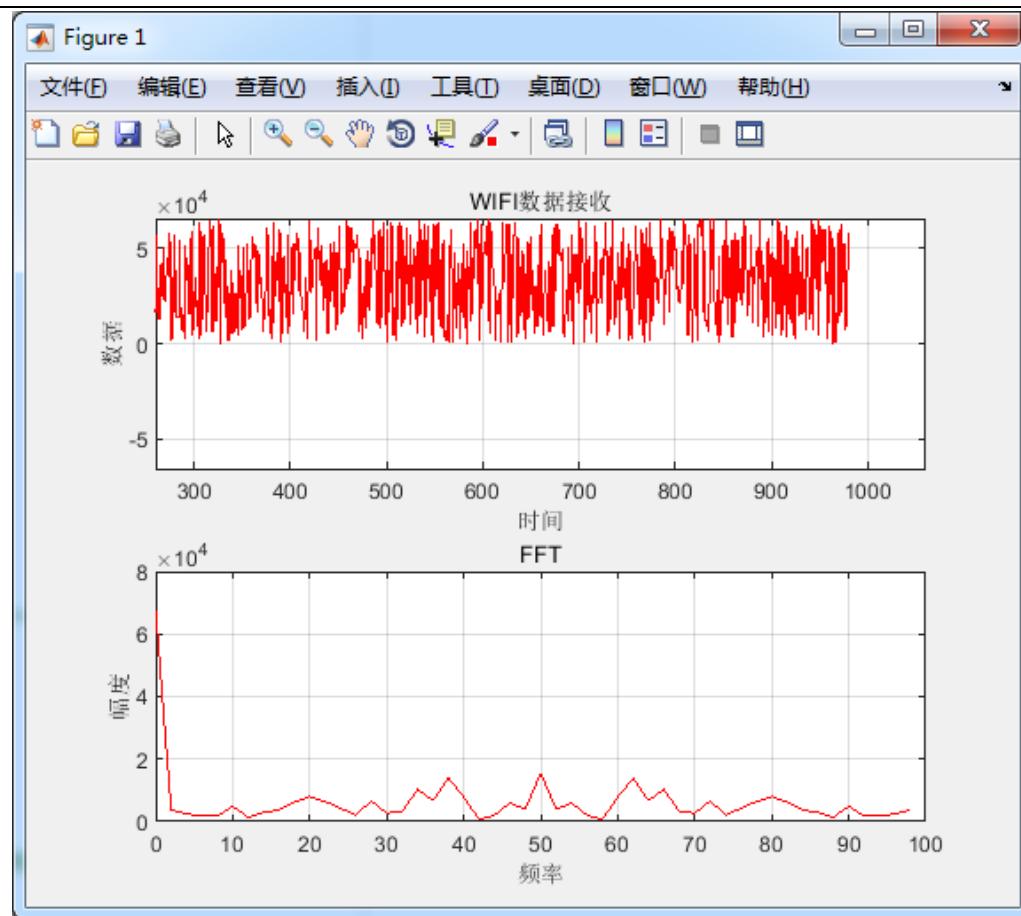
Serial-COM1

```
CPU : STM32H743XIH6, BGA240, 主频: 400MHz
UID = 34343436 3038510A 0040002D
*****
* 例程名称 : V7-Matlab的WIFI通信实现
* 例程版本 : 1.0
* 发布日期 : 2019-09-13
* HAL库版本 : V1.3.0 (STM32H7xx HAL Driver)
*
* QQ : 1295744630
* 旺旺 : armfly
* Email : armfly@qq.com
* 微信公众号: armfly_com
* 淘宝店: armfly.taobao.com
* Copyright www.armfly.com 安富莱电子
*****
操作提示:
ESP8266模块 STM32-V7开发板
UTXD --- PC7/USART6_RX CN16插座
GND --- GND CN6插座
CH_PD --- PIO(控制模块掉电) CN16插座
GPIO2
GPIO16
GPIO0
VCC --- 3.3 (供电) CN6插座
URXD --- PG14/USART6_TX CN16插座

【按键操作】
K1键 : 列举AP
K2键 : 加入AP
K3键 : 9600波特率切换到115200, 并设置为Station模式
摇杆上键 : AT+CIFSR获取本地IP地址
摇杆下键 : AT+CIPSTATUS获得IP连接状态
摇杆左键 : AT+CIPSTART建立TCP服务器
摇杆右键 : AT+CIPSEND向TCP客户端发送数据
```

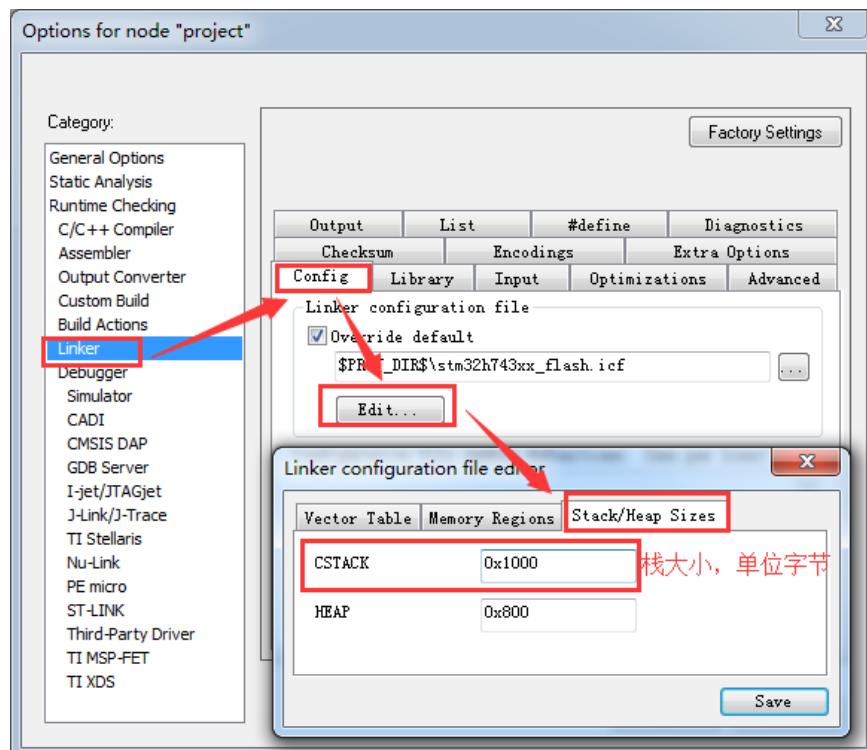
就绪 Serial: COM1 37, 1 37行, 73列 VT100 大写 数字

Matlab 的上位机效果:

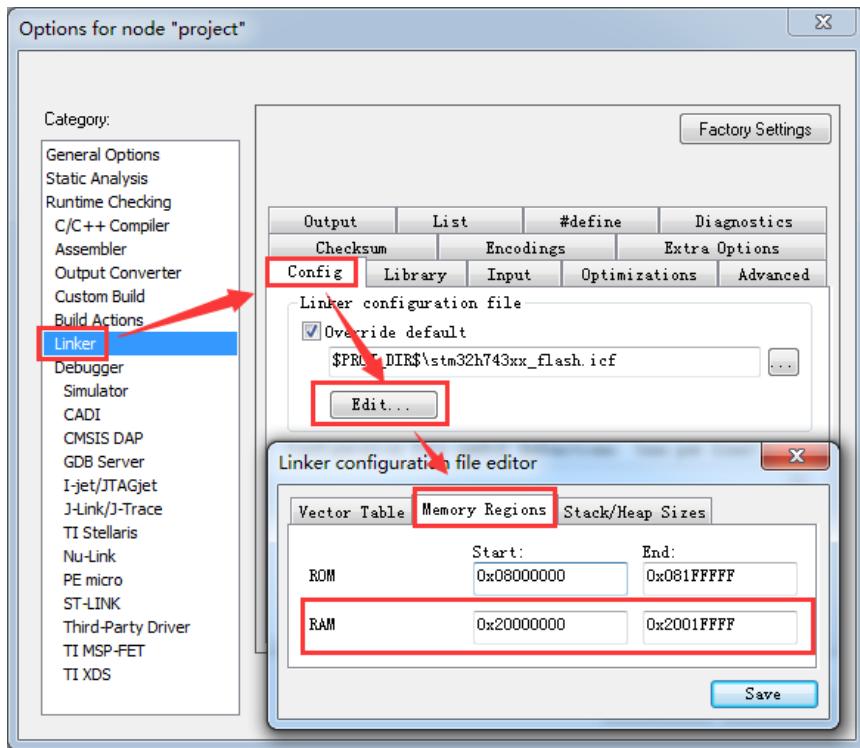


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
}
```



```
SystemClock_Config();

/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */

bsp_InitESP8266(); /* 配置 ESP8266 模块相关的资源 */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
}
```



```
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec   = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 接收 matlab 发送过来的同步信号，并回一个同步信号后，传输相应的数据过去

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint8_t ucValue;
    uint8_t ret;
    uint8_t SyncData = 36;
    uint16_t SendDATA[5];

    bsp_Init();           /* 硬件初始化 */
```



```
PrintfLogo(); /* 打印例程信息到串口 1 */

PrintfHelp(); /* 打印操作提示信息 */

/* 模块上电 */
printf("\r\n【1】正在给 ESP8266 模块上电... (波特率: 74880bsp)\r\n");
ESP8266_PowerOn();

printf("\r\n【2】上电完成。波特率: 115200bsp\r\n");

/* 检测模块波特率是否为 115200 */
ESP8266_SendAT("AT");
if (ESP8266_WaitResponse("OK", 50) == 1)
{
    printf("\r\n【3】模块应答 AT 成功\r\n");
    bsp_DelayMS(1000);
}
else
{
    printf("\r\n【3】模块无应答, 请按 K3 键修改模块的波特率为 115200\r\n");
    bsp_DelayMS(1000);
}

g_TCPServerOk = 0;

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle(); /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    /* 进入 Matlab 通信状态执行下面程序 */
    if (g_TCPServerOk == 1)
    {
        cmd_len = ESP8266_RxNew(cmd_buf, &tcpid);
        if (cmd_len > 0)
        {
            printf("\r\n接收到数据长度 = %d\r\n远程 ID =%d\r\n数据内容=%s\r\n",
                   cmd_len, tcpid, cmd_buf);

            /* 检索 matlab 发送过来的同步帧字符$, 对应的 ASCII 数值是 36 */
            if (strchr((char *)cmd_buf, 36))
            {
                /* 回复同步帧$ */
                ESP8266_SendTcpUdp(tcpid, (uint8_t *)&SyncData, 1);
                bsp_DelayMS(10);
                SendDATA[0] = rand() % 65536;
                SendDATA[1] = rand() % 65536;
                SendDATA[2] = rand() % 65536;
                SendDATA[3] = rand() % 65536;
            }
        }
    }
}
```



```
SendDATA[4] = rand()%65536;

/* 发送数据, 10 个字节 */
ESP8266_SendTcpUdp(tcpid, (uint8_t *)SendDATA, 10);
printf("找到了相应的字符串\r\n");
}

else
{
    printf("没有找到了相应的字符串\r\n");
}
}

/*
未进入 Matlab 通信状态执行下面程序 */
else
{
    /* 从 WIFI 收到的数据发送到串口 1 */
    if (comGetChar(COM_ESP8266, &ucValue))
    {
        comSendChar(COM1, ucValue);
    }
    /* 将串口 1 的数据发送到 8266 模块 */
    if (comGetChar(COM1, &ucValue))
    {
        comSendChar(COM_ESP8266, ucValue);
    }
}

ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
if (ucKeyCode != KEY_NONE)
{
    switch (ucKeyCode)
    {
        case KEY_DOWN_K1:           /* K1 键按下, 列举当前的 WIFI 热点 */
            g_TCPServerOk = 0;
            ESP8266_SendAT("AT+CWLAP");
            break;

        case KEY_DOWN_K2:           /* K2 键按下, 加入某个 WIFI 网络*/
            g_TCPServerOk = 0;
            //ESP8266_SendAT("AT+CWJAP=\\"Netcore_7378CB\\",\\"512464265\\\"");
            ret = ESP8266_JoinAP("Netcore_7378CB", "512464265", 15000);
            if(ret == 1)
            {
                printf("\r\nJoinAP Success\r\n");
            }
            else
            {
                printf("\r\nJoinAP fail\r\n");
            }

            break;

        case KEY_DOWN_K3:           /* K3 键-9600 波特率切换到 115200 */
            g_TCPServerOk = 0;
            ESP8266_9600to115200();
            break;

        case JOY_DOWN_U:             /* 摆杆上键, AT+CIFSR 获取本地 IP 地址 */
    }
}
```



```
g_TCPServerOk = 0;
ESP8266_SendAT("AT+CIFSR");
break;

case JOY_DOWN_D: /* 摆杆下键 AT+CIPSTATUS 获得 IP 连接状态 */
    g_TCPServerOk = 0;
    ESP8266_SendAT("AT+CIPSTATUS");
    break;

case JOY_DOWN_L: /* 摆杆左键按下，创建 TCP 服务器 */
    g_TCPServerOk = 0;
    ret = ESP8266_CreateTCPServer(1001);
    if(ret == 1)
    {
        printf("\r\nCreateTCP Success\r\n");
    }
    else
    {
        printf("\r\nCreateTCP fail\r\n");
    }
    break;

case JOY_DOWN_R: /* 摆杆右键按下，进入 Matlab 数据通信状态 */
    g_TCPServerOk = 1;
    printf("\r\n进入 Matlab 数据通信状态 \r\n");
    break;

case JOY_DOWN_OK: /* 摆杆 OK 键按下，创建 WIFI 热点 */
    g_TCPServerOk = 0;
#ifndef 0
    ESP8266_SendAT("AT+CIPSTART=\"TCP\", \"WWW.ARMFLY.COM\", 80");
#endif

#ifndef 0
{
    char ip[20], mac[32];
    ESP8266_GetLocalIP(ip, mac);
    printf("ip=%s, mac=%s\r\n", ip, mac);
}
#endif

#ifndef 1
    ESP8266_SetWiFiMode(3);
    ESP8266_SendAT("AT+CWSAP=\"ESP8266\", \"1234567890\", 1, 3");
#endif
    break;

default: /* 其他的键值不处理 */
    break;
}

}
```



武汉安富莱电子有限公司

WWW.ARMBBS.CN

安富莱 STM32-V7 开发板数字信号处理教程

10.8 总结

本章讲解的例程非常实用，建议大家熟练掌握。



第11章 DSP 基础函数-绝对值，求和，乘法和点

乘

本期教程开始学习 ARM 官方的 DSP 库，这里我们先从基本数学函数开始。本期教程主要讲绝对值，加法，点乘和乘法四种运算。

11.1 初学者重要提示

11.2 DSP 基础运算指令

11.3 绝对值 (Vector Absolute Value)

11.4 求和 (Vector Addition)

11.5 点乘 (Vector Dot Product)

11.6 乘法 (Vector Multiplication)

11.7 实验例程说明 (MDK)

11.8 实验例程说明 (IAR)

11.1 初学者重要提示

在这里简单的跟大家介绍一下 DSP 库中函数的通用格式，后面就不再赘述了。

- ◆ 基本所有的函数都是可重入的。
- ◆ 大部分函数都支持批量计算，比如求绝对值函数 arm_abs_f32。所以如果只是就几个数的绝对值，用这个库函数就没有什么优势了。
- ◆ 库函数基本是 CM0, CM0+, CM3, CM4 和 CM7 内核都是支持的，不限制厂家。
- ◆ 每组数据基本上都是以 4 个数为一个单位进行计算，不够四个再单独计算。大部分函数都是配有 f32, Q31, Q15 和 Q7 四种格式。
- ◆ 为什么定点 DSP 运算输出的时候容易出现结果为 0 的情况：
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=95194>。

11.2 DSP 基础运算指令

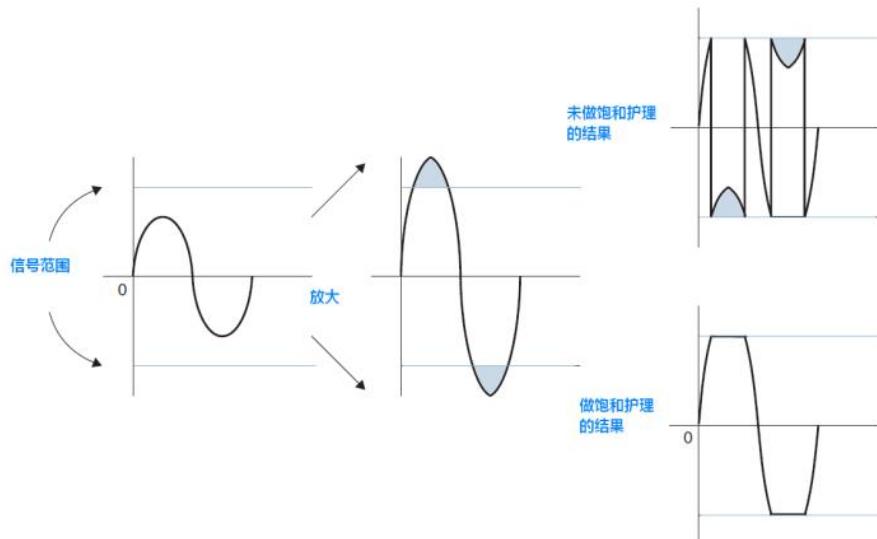
本章用到基础运算指令：

- ◆ 绝对值函数用到 QSUB, QSUB16 和 QSUB8。
- ◆ 求和函数用到 QADD, QADD16 和 QADD8。

- ◆ 点乘函数用到 SMLALD 和 SMLAD。
- ◆ 乘法用到 _PKHBT 和 _SSAT。

用到的这几个指令，在本章讲解具体函数时都有专门的讲解说明。这里重点说一下饱和运算的问题，字母 Q 打头的指令是饱和运算指令，饱和的意思超过所能表示的数值范围时，将直接取最大值，比如 QSUB16 减法指令，如果是正数，那么最大值是 0x7FFF (32767)，大于这个值将直接取 0x7FFF，如果是负数，那么最小值是 0x8000 (-32768)，比这个值还小将直接取值 0x8000。

反应到实际应用中就是下面这种效果：



11.3 绝对值 (Vector Absolute Value)

这部分函数主要用于求绝对值，公式描述如下：

$pDst[n] = \text{abs}(pSrc[n]), 0 \leq n < \text{blockSize}.$

特别注意，这部分函数支持目标指针和源指针指向相同的缓冲区。

11.3.1 函数 arm_abs_f32

函数原型：

```
1. void arm_abs_f32(
2.     const float32_t * pSrc,
3.     float32_t * pDst,
4.     uint32_t blockSize)
5. {
6.     uint32_t blkCnt; /* Loop counter */
7.
8. #if defined(ARM_MATH_NEON)
9.     float32x4_t vec1;
10.    float32x4_t res;
11.
12. /* Compute 4 outputs at a time */
13.    blkCnt = blockSize >> 2U;
14.
```



```
15.     while (blkCnt > 0U)
16.     {
17.         /* C = |A| */
18.
19.         /* Calculate absolute values and then store the results in the destination buffer. */
20.         vec1 = vld1q_f32(pSrc);
21.         res = vabsq_f32(vec1);
22.         vst1q_f32(pDst, res);
23.
24.         /* Increment pointers */
25.         pSrc += 4;
26.         pDst += 4;
27.
28.         /* Decrement the loop counter */
29.         blkCnt--;
30.     }
31.
32.     /* Tail */
33.     blkCnt = blockSize & 0x3;
34.
35. #else
36. #if defined (ARM_MATH_LOOPUNROLL)
37.
38.     /* Loop unrolling: Compute 4 outputs at a time */
39.     blkCnt = blockSize >> 2U;
40.
41.     while (blkCnt > 0U)
42.     {
43.         /* C = |A| */
44.
45.         /* Calculate absolute and store result in destination buffer. */
46.         *pDst++ = fabsf(*pSrc++);
47.
48.         *pDst++ = fabsf(*pSrc++);
49.
50.         *pDst++ = fabsf(*pSrc++);
51.
52.         *pDst++ = fabsf(*pSrc++);
53.
54.         /* Decrement loop counter */
55.         blkCnt--;
56.     }
57.
58.     /* Loop unrolling: Compute remaining outputs */
59.     blkCnt = blockSize % 0x4U;
60.
61. #else
62.
63.     /* Initialize blkCnt with number of samples */
64.     blkCnt = blockSize;
65.
66. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
67. #endif /* #if defined(ARM_MATH_NEON) */
68.
69.     while (blkCnt > 0U)
70.     {
71.         /* C = |A| */
72.     }
```



```
73.     /* Calculate absolute and store result in destination buffer. */
74.     *pDst++ = fabsf(*pSrc++);
75.
76.     /* Decrement loop counter */
77.     blkCnt--;
78. }
79.
80. }
```

函数描述：

这个函数用于求 32 位浮点数的绝对值。

函数解析：

- ◆ 第 8 到 35 行，用于 NEON 指令集，当前的 CM 内核不支持。
- ◆ 第 36 到 66 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 函数 fabsf 不是用 Cortex-M 内核支持的 DSP 指令实现的，而是用 C 库函数实现的，这个函数是被 MDK 封装了起来。
- ◆ 第 69 到 78 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是原数据地址。
- ◆ 第 2 个参数是求绝对值后目的数据地址。
- ◆ 第 3 个参数转换的数据个数，这里是指的浮点数个数。

函数描述：

函数形参的源地址和目的地址可以使用同一个缓冲。

11.3.2 函数 arm_abs_q31

函数原型：

```
1. void arm_abs_q31(
2.     const q31_t * pSrc,
3.     q31_t * pDst,
4.     uint32_t blockSize)
5. {
6.     uint32_t blkCnt;                      /* Loop counter */
7.     q31_t in;                           /* Temporary variable */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. /* Loop unrolling: Compute 4 outputs at a time */
12. blkCnt = blockSize >> 2U;
13.
14. while (blkCnt > 0U)
15. {
16.     /* C = |A| */
17.
18.     /* Calculate absolute of input (if -1 then saturated to 0x7fffffff) and store result in
19.      destination
20.      buffer. */
21.     in = *pSrc++;
22. #if defined (ARM_MATH_DSP)
```



```
22.     *pDst++ = (in > 0) ? in : (q31_t)_QSUB(0, in);
23. #else
24.     *pDst++ = (in > 0) ? in : ((in == INT32_MIN) ? INT32_MAX : -in);
25. #endif
26.
27.     in = *pSrc++;
28. #if defined (ARM_MATH_DSP)
29.     *pDst++ = (in > 0) ? in : (q31_t)_QSUB(0, in);
30. #else
31.     *pDst++ = (in > 0) ? in : ((in == INT32_MIN) ? INT32_MAX : -in);
32. #endif
33.
34.     in = *pSrc++;
35. #if defined (ARM_MATH_DSP)
36.     *pDst++ = (in > 0) ? in : (q31_t)_QSUB(0, in);
37. #else
38.     *pDst++ = (in > 0) ? in : ((in == INT32_MIN) ? INT32_MAX : -in);
39. #endif
40.
41.     in = *pSrc++;
42. #if defined (ARM_MATH_DSP)
43.     *pDst++ = (in > 0) ? in : (q31_t)_QSUB(0, in);
44. #else
45.     *pDst++ = (in > 0) ? in : ((in == INT32_MIN) ? INT32_MAX : -in);
46. #endif
47.
48.     /* Decrement loop counter */
49.     blkCnt--;
50. }
51.
52. /* Loop unrolling: Compute remaining outputs */
53. blkCnt = blockSize % 0x4U;
54.
55. #else
56.
57. /* Initialize blkCnt with number of samples */
58. blkCnt = blockSize;
59.
60. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
61.
62. while (blkCnt > 0U)
63. {
64.     /* C = |A| */
65.
66.     /* Calculate absolute of input (if -1 then saturated to 0xffffffff) and store result in
destination
67.      buffer. */
68.     in = *pSrc++;
69. #if defined (ARM_MATH_DSP)
70.     *pDst++ = (in > 0) ? in : (q31_t)_QSUB(0, in);
71. #else
72.     *pDst++ = (in > 0) ? in : ((in == INT32_MIN) ? INT32_MAX : -in);
73. #endif
74.
75.     /* Decrement loop counter */
76.     blkCnt--;
77. }
78.
```



79. }

函数描述:

用于求 32 位定点数的绝对值。

函数解析:

- ◆ 第 9 到 60 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 69 到 78 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 这个函数使用了饱和运算，其实不光这个函数，后面很多函数都是使用了饱和运算的，关于什么是饱和运算，大家看 Cortex-M3 权威指南中文版的 4.3.6 小节：汇编语言：饱和运算即可。
- ◆ 对于 Q31 格式的数据，饱和运算会使得数据 0x80000000 变成 0x7fffffff（这个数比较特殊，算是特殊处理，记住即可）。
- ◆ 这里重点说一下函数_QSUB，其实这个函数算是 Cortex-M7, M4/M3 的一个指令，用于实现饱和减法。比如函数：_QSUB(0, in1) 的作用就是实现 $0 - \text{in1}$ 并返回结果。这里_QSUB 实现的是 32 位数的饱和减法。还有_QSUB16 和_QSUB8 实现的是 16 位和 8 位数的减法。

函数参数:

- ◆ 第 1 个参数是原数据地址。
- ◆ 第 2 个参数是求绝对值后目的数据地址。
- ◆ 第 3 个参数转换的数据个数，这里是指的定点数个数。

函数描述:

函数形参的源地址和目的地址可以使用同一个缓冲。

11.3.3 函数 arm_abs_q15

函数原型:

```
1. void arm_abs_q15(
2.     const q15_t * pSrc,
3.     q15_t * pDst,
4.     uint32_t blockSize)
5. {
6.     uint32_t blkCnt; /* Loop counter */
7.     q15_t in; /* Temporary input variable */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. /* Loop unrolling: Compute 4 outputs at a time */
12. blkCnt = blockSize >> 2U;
13.
14. while (blkCnt > 0U)
15. {
16.     /* C = |A| */
17.
18.     /* Calculate absolute of input (if -1 then saturated to 0x7fff) and store result in destination
       buffer.
19. */
20.     in = *pSrc++;
21. #if defined (ARM_MATH_DSP)
```



```
22.     *pDst++ = (in > 0) ? in : (q15_t)_QSUB16(0, in);
23. #else
24.     *pDst++ = (in > 0) ? in : ((in == (q15_t) 0x8000) ? 0x7fff : -in);
25. #endif
26.
27.     in = *pSrc++;
28. #if defined (ARM_MATH_DSP)
29.     *pDst++ = (in > 0) ? in : (q15_t)_QSUB16(0, in);
30. #else
31.     *pDst++ = (in > 0) ? in : ((in == (q15_t) 0x8000) ? 0x7fff : -in);
32. #endif
33.
34.     in = *pSrc++;
35. #if defined (ARM_MATH_DSP)
36.     *pDst++ = (in > 0) ? in : (q15_t)_QSUB16(0, in);
37. #else
38.     *pDst++ = (in > 0) ? in : ((in == (q15_t) 0x8000) ? 0x7fff : -in);
39. #endif
40.
41.     in = *pSrc++;
42. #if defined (ARM_MATH_DSP)
43.     *pDst++ = (in > 0) ? in : (q15_t)_QSUB16(0, in);
44. #else
45.     *pDst++ = (in > 0) ? in : ((in == (q15_t) 0x8000) ? 0x7fff : -in);
46. #endif
47.
48.     /* Decrement loop counter */
49.     blkCnt--;
50. }
51.
52. /* Loop unrolling: Compute remaining outputs */
53. blkCnt = blockSize % 0x4U;
54.
55. #else
56.
57.     /* Initialize blkCnt with number of samples */
58.     blkCnt = blockSize;
59.
60. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
61.
62.     while (blkCnt > 0U)
63.     {
64.         /* C = |A| */
65.
66.         /* Calculate absolute of input (if -1 then saturated to 0x7fff) and store result in destination
       buffer.
67.         */
68.         in = *pSrc++;
69. #if defined (ARM_MATH_DSP)
70.         *pDst++ = (in > 0) ? in : (q15_t)_QSUB16(0, in);
71. #else
72.         *pDst++ = (in > 0) ? in : ((in == (q15_t) 0x8000) ? 0x7fff : -in);
73. #endif
74.
75.         /* Decrement loop counter */
76.         blkCnt--;
77.     }
78.
```



79. }

函数描述：

用于求 16 位定点数的绝对值。

函数解析：

- ◆ 第 9 到 55 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 62 到 77 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 对于 Q15 格式的数据，饱和运算会使得数据 0x8000 变成 0xffff。
- ◆ __QSUB16 用于实现 16 位数据的饱和减法。

函数参数：

- ◆ 第 1 个参数是原数据地址。
- ◆ 第 2 个参数是求绝对值后目的数据地址。
- ◆ 第 3 个参数转换的数据个数，这里是指的定点数个数。

函数描述：

函数形参的源地址和目的地址可以使用同一个缓冲。

11.3.4 函数 arm_abs_q7

函数原型：

```
1. void arm_abs_q7(
2.     const q7_t * pSrc,
3.     q7_t * pDst,
4.     uint32_t blockSize)
5. {
6.     uint32_t blkCnt; /* Loop counter */
7.     q7_t in; /* Temporary input variable */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. /* Loop unrolling: Compute 4 outputs at a time */
12. blkCnt = blockSize >> 2U;
13.
14. while (blkCnt > 0U)
15. {
16.     /* C = |A| */
17.
18.     /* Calculate absolute of input (if -1 then saturated to 0x7f) and store result in destination
19.      buffer. */
20.     /*
21.     #if defined (ARM_MATH_DSP)
22.         *pDst++ = (in > 0) ? in : (q7_t)__QSUB(0, in);
23.     #else
24.         *pDst++ = (in > 0) ? in : ((in == (q7_t) 0x80) ? (q7_t) 0x7f : -in);
25.     #endif
26.
27.     in = *pSrc++;
28.     #if defined (ARM_MATH_DSP)
29.         *pDst++ = (in > 0) ? in : (q7_t)__QSUB(0, in);
30.     #else
```



```
31.     *pDst++ = (in > 0) ? in : ((in == (q7_t) 0x80) ? (q7_t) 0x7f : -in);
32. #endif
33.
34.     in = *pSrc++;
35. #if defined (ARM_MATH_DSP)
36.     *pDst++ = (in > 0) ? in : (q7_t)_QSUB(0, in);
37. #else
38.     *pDst++ = (in > 0) ? in : ((in == (q7_t) 0x80) ? (q7_t) 0x7f : -in);
39. #endif
40.
41.     in = *pSrc++;ezi le mexia
42. #if defined (ARM_MATH_DSP)
43.     *pDst++ = (in > 0) ? in : (q7_t)_QSUB(0, in);
44. #else
45.     *pDst++ = (in > 0) ? in : ((in == (q7_t) 0x80) ? (q7_t) 0x7f : -in);
46. #endif
47.
48.     /* Decrement loop counter */
49.     blkCnt--;
50. }
51.
52. /* Loop unrolling: Compute remaining outputs */
53. blkCnt = blockSize % 0x4U;
54.
55. #else
56.
57.     /* Initialize blkCnt with number of samples */
58.     blkCnt = blockSize;
59.
60. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
61.
62.     while (blkCnt > 0U)
63.     {
64.         /* C = |A| */
65.
66.         /* Calculate absolute of input (if -1 then saturated to 0x7f) and store result in destination
       buffer.
67.         */
68.         in = *pSrc++;
69. #if defined (ARM_MATH_DSP)
70.         *pDst++ = (in > 0) ? in : (q7_t)_QSUB(0, in);
71. #else
72.         *pDst++ = (in > 0) ? in : ((in == (q7_t) 0x80) ? (q7_t) 0x7f : -in);
73. #endif
74.
75.         /* Decrement loop counter */
76.         blkCnt--;
77.     }
78.
79. }
```

函数描述：

用于求 8 位定点数的绝对值。

函数解析：

- ◆ 第 9 到 55 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。



- ◆ 第 62 到 77 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 对于 Q7 格式的数据，饱和运算会使得数据 0x80 变成 0x7f。
- ◆ __QSUB 用于实现 32 位数据的饱和减法。而当前的 DSP 库版本却将其用到了 Q7 函数中，导致 0x80 的饱和和出错。详情看此贴：
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=95152>。

函数参数：

- ◆ 第 1 个参数是原数据地址。
- ◆ 第 2 个参数是求绝对值后目的数据地址。
- ◆ 第 3 个参数转换的数据个数，这里是指的定点数个数。

函数描述：

函数形参的源地址和目的地址可以使用同一个缓冲。

11.3.5 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_ABS
* 功能说明: 求绝对值
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_ABS(void)
{
    float32_t pSrc;
    float32_t pDst;

    q31_t pSrc1;
    q31_t pDst1;

    q15_t pSrc2;
    q15_t pDst2;

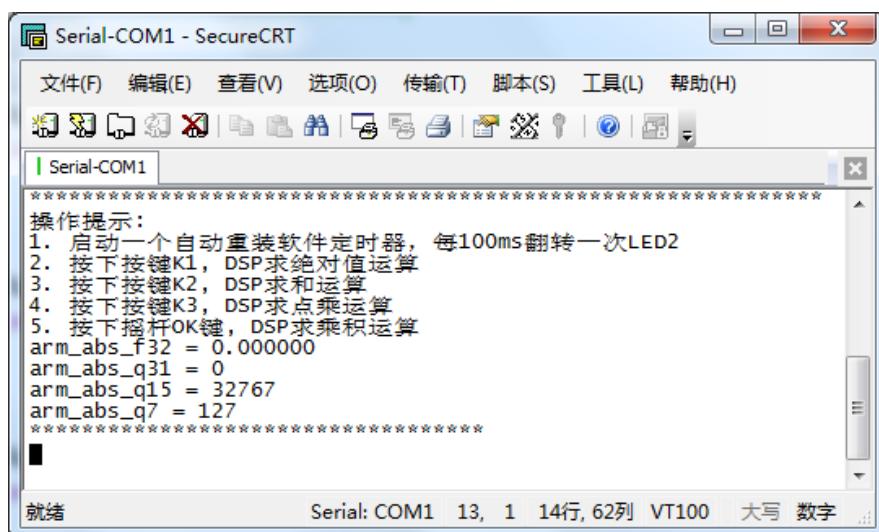
    q7_t pSrc3;
    q7_t pDst3;

    /*求绝对值******/
    pSrc = 1.23f;
    arm_abs_f32(&pSrc, &pDst, 1);
    printf("arm_abs_f32 = %f\r\n", pDst);

    pSrc1 = 1;
    arm_abs_q31(&pSrc1, &pDst1, 1);
    printf("arm_abs_q31 = %d\r\n", pDst1);

    pSrc2 = -32768;
    arm_abs_q15(&pSrc2, &pDst2, 1);
    printf("arm_abs_q15 = %d\r\n", pDst2);

    pSrc3 = 127;
    arm_abs_q7(&pSrc3, &pDst3, 1);
    printf("arm_abs_q7 = %d\r\n", pDst3);
    printf("*****\r\n");
}
```

实验现象：

这里特别注意 Q15 的计算，数值-32768 被饱和处理到 32767，即 $0 - (-32768) = 32768$ ，超出了正数所能表示的最大值，经过饱和后，输出为 32767。

11.4 求和 (Vector Addition)

这部分函数主要用于求和，公式描述如下：

$$pDst[n] = pSrcA[n] + pSrcB[n], \quad 0 \leq n < blockSize.$$

11.4.1 函数 arm_add_f32

函数原型：

```
1. void arm_add_f32(
2.     const float32_t * pSrcA,
3.     const float32_t * pSrcB,
4.     float32_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined(ARM_MATH_NEON)
10.    float32x4_t vec1;
11.    float32x4_t vec2;
12.    float32x4_t res;
13.
14.    /* Compute 4 outputs at a time */
15.    blkCnt = blockSize >> 2U;
16.
17.    while (blkCnt > 0U)
18.    {
19.        /* C = A + B */
20.
21.        /* Add and then store the results in the destination buffer. */
22.        vec1 = vld1q_f32(pSrcA);
23.        vec2 = vld1q_f32(pSrcB);
24.        res = vaddq_f32(vec1, vec2);
```



```
25.         vst1q_f32(pDst, res);
26.
27.         /* Increment pointers */
28.         pSrcA += 4;
29.         pSrcB += 4;
30.         pDst += 4;
31.
32.         /* Decrement the loop counter */
33.         blkCnt--;
34.     }
35.
36.     /* Tail */
37.     blkCnt = blockSize & 0x3;
38.
39. #else
40. #if defined(ARM_MATH_LOOPUNROLL)
41.
42.     /* Loop unrolling: Compute 4 outputs at a time */
43.     blkCnt = blockSize >> 2U;
44.
45.     while (blkCnt > 0U)
46.     {
47.         /* C = A + B */
48.
49.         /* Add and store result in destination buffer. */
50.         *pDst++ = (*pSrcA++) + (*pSrcB++);
51.         *pDst++ = (*pSrcA++) + (*pSrcB++);
52.         *pDst++ = (*pSrcA++) + (*pSrcB++);
53.         *pDst++ = (*pSrcA++) + (*pSrcB++);
54.
55.         /* Decrement loop counter */
56.         blkCnt--;
57.     }
58.
59.     /* Loop unrolling: Compute remaining outputs */
60.     blkCnt = blockSize % 0x4U;
61.
62. #else
63.
64.     /* Initialize blkCnt with number of samples */
65.     blkCnt = blockSize;
66.
67. #endif /* #if defined(ARM_MATH_LOOPUNROLL) */
68. #endif /* #if defined(ARM_MATH_NEON) */
69.
70.     while (blkCnt > 0U)
71.     {
72.         /* C = A + B */
73.
74.         /* Add and store result in destination buffer. */
75.         *pDst++ = (*pSrcA++) + (*pSrcB++);
76.
77.         /* Decrement loop counter */
78.         blkCnt--;
79.     }
80.
81. }
```



函数描述：

这个函数用于求两个 32 位浮点数的和。

函数解析：

- ◆ 第 8 到 35 行，用于 NEON 指令集，当前的 CM 内核不支持。
- ◆ 第 40 到 62 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 70 到 79 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是加数地址。
- ◆ 第 2 个参数是被加数地址。
- ◆ 第 3 个参数是和地址。
- ◆ 第 4 个参数是浮点数个数，其实就是执行加法的次数。

11.4.2 函数 arm_add_q31

函数原型：

```
1. void arm_add_q31(
2.     const q31_t * pSrcA,
3.     const q31_t * pSrcB,
4.     q31_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11.    /* Loop unrolling: Compute 4 outputs at a time */
12.    blkCnt = blockSize >> 2U;
13.
14.    while (blkCnt > 0U)
15.    {
16.        /* C = A + B */
17.
18.        /* Add and store result in destination buffer. */
19.        *pDst++ = __QADD(*pSrcA++, *pSrcB++);
20.
21.        *pDst++ = __QADD(*pSrcA++, *pSrcB++);
22.
23.        *pDst++ = __QADD(*pSrcA++, *pSrcB++);
24.
25.        *pDst++ = __QADD(*pSrcA++, *pSrcB++);
26.
27.        /* Decrement loop counter */
28.        blkCnt--;
29.    }
30.
31.    /* Loop unrolling: Compute remaining outputs */
32.    blkCnt = blockSize % 0x4U;
33.
34. #else
35.
```



```
36. /* Initialize blkCnt with number of samples */
37. blkCnt = blockSize;
38.
39. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
40.
41. while (blkCnt > 0U)
42. {
43.     /* C = A + B */
44.
45.     /* Add and store result in destination buffer. */
46.     *pDst++ = __QADD(*pSrcA++, *pSrcB++);
47.
48.     /* Decrement loop counter */
49.     blkCnt--;
50. }
51.
52. }
```

函数描述：

这个函数用于求两个 32 位定点数的和。

函数解析：

- ◆ 第 9 到 34 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 41 到 50 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ __QADD 实现 32 位数的加法饱和运算。输出结果的范围[0x80000000 0x7FFFFFFF]，超出这个结果将产生饱和结果，负数饱和到 0x80000000，正数饱和到 0x7FFFFFFF。

函数参数：

- ◆ 第 1 个参数是加数地址。
- ◆ 第 2 个参数是被加数地址。
- ◆ 第 3 个参数是和地址。
- ◆ 第 4 个参数是定点数个数，其实就是执行加法的次数。

11.4.3 函数 arm_add_q15

函数原型：

```
1. void arm_add_q15(
2.     const q15_t * pSrcA,
3.     const q15_t * pSrcB,
4.     q15_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. #if defined (ARM_MATH_DSP)
12.     q31_t inA1, inA2;
13.     q31_t inB1, inB2;
14. #endif
15.
16. /* Loop unrolling: Compute 4 outputs at a time */
```



```
17.     blkCnt = blockSize >> 2U;
18.
19.     while (blkCnt > 0U)
20.     {
21.         /* C = A + B */
22.
23. #if defined (ARM_MATH_DSP)
24.     /* read 2 times 2 samples at a time from sourceA */
25.     inA1 = read_q15x2_ia ((q15_t **) &pSrcA);
26.     inA2 = read_q15x2_ia ((q15_t **) &pSrcA);
27.     /* read 2 times 2 samples at a time from sourceB */
28.     inB1 = read_q15x2_ia ((q15_t **) &pSrcB);
29.     inB2 = read_q15x2_ia ((q15_t **) &pSrcB);
30.
31.     /* Add and store 2 times 2 samples at a time */
32.     write_q15x2_ia (&pDst, __QADD16(inA1, inB1));
33.     write_q15x2_ia (&pDst, __QADD16(inA2, inB2));
34. #else
35.     *pDst++ = (q15_t) __SSAT(((q31_t) *pSrcA++ + *pSrcB++), 16);
36.     *pDst++ = (q15_t) __SSAT(((q31_t) *pSrcA++ + *pSrcB++), 16);
37.     *pDst++ = (q15_t) __SSAT(((q31_t) *pSrcA++ + *pSrcB++), 16);
38.     *pDst++ = (q15_t) __SSAT(((q31_t) *pSrcA++ + *pSrcB++), 16);
39. #endif
40.
41.     /* Decrement loop counter */
42.     blkCnt--;
43. }
44.
45. /* Loop unrolling: Compute remaining outputs */
46. blkCnt = blockSize % 0x4U;
47.
48. #else
49.
50.     /* Initialize blkCnt with number of samples */
51.     blkCnt = blockSize;
52.
53. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
54.
55.     while (blkCnt > 0U)
56.     {
57.         /* C = A + B */
58.
59.         /* Add and store result in destination buffer. */
60. #if defined (ARM_MATH_DSP)
61.         *pDst++ = (q15_t) __QADD16(*pSrcA++, *pSrcB++);
62. #else
63.         *pDst++ = (q15_t) __SSAT(((q31_t) *pSrcA++ + *pSrcB++), 16);
64. #endif
65.
66.         /* Decrement loop counter */
67.         blkCnt--;
68.     }
69.
70. }
```

函数描述：

这个函数用于求两个 16 位定点数的和。



函数解析：

- ◆ 第 23 到 34 行，对于 M4 和 M7 带 DSP 单元的芯片使用。
- ◆ 第 35 到 38 行，对于不带 DSP 单元的 M0, M0+ 和 M3 使用。
- ◆ 第 55 到 68 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 函数 `read_q15x2_ia` 的原型如下：

```
__STATIC_FORCEINLINE q31_t read_q15x2_ia (
    q15_t ** pQ15)
{
    q31_t val;

    memcpy (&val, *pQ15, 4);
    *pQ15 += 2;

    return (val);
}
```

作用是读取两次 16 位数据，返回一个 32 位数据，并将数据地址递增，方便下次读取。

- ◆ `_QADD16` 实现两次 16 位数的加法饱和运算。输出结果的范围[0x8000 0x7FFF]，超出这个结果将产生饱和结果，负数饱和到 0x8000，正数饱和到 0x7FFF。
- ◆ `_SSAT` 也是 SIMD 指令，这里是将结果饱和到 16 位精度。

函数参数：

- ◆ 第 1 个参数是加数地址。
- ◆ 第 2 个参数是被加数地址。
- ◆ 第 3 个参数是和地址。
- ◆ 第 4 个参数是定点数个数，其实就是执行加法的次数。

11.4.4 函数 `arm_add_q7`

函数原型：

```
1. void arm_add_q7(
2.     const q7_t * pSrcA,
3.     const q7_t * pSrcB,
4.     q7_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. /* Loop unrolling: Compute 4 outputs at a time */
12. blkCnt = blockSize >> 2U;
13.
14. while (blkCnt > 0U)
15. {
16.     /* C = A + B */
17.
18. #if defined (ARM_MATH_DSP)
19.     /* Add and store result in destination buffer (4 samples at a time). */
20.     write_q7x4_ia (&pDst, __QADD8 (read_q7x4_ia ((q7_t **) &pSrcA), read_q7x4_ia ((q7_t **) &pSrcB)));
}
```



```
21. #else
22.     *pDst++ = (q7_t) __SSAT ((q15_t) *pSrcA++ + *pSrcB++, 8);
23.     *pDst++ = (q7_t) __SSAT ((q15_t) *pSrcA++ + *pSrcB++, 8);
24.     *pDst++ = (q7_t) __SSAT ((q15_t) *pSrcA++ + *pSrcB++, 8);
25.     *pDst++ = (q7_t) __SSAT ((q15_t) *pSrcA++ + *pSrcB++, 8);
26. #endif
27.
28.     /* Decrement loop counter */
29.     blkCnt--;
30. }
31.
32. /* Loop unrolling: Compute remaining outputs */
33. blkCnt = blockSize % 0x4U;
34.
35. #else
36.
37.     /* Initialize blkCnt with number of samples */
38.     blkCnt = blockSize;
39.
40. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
41.
42. while (blkCnt > 0U)
43. {
44.     /* C = A + B */
45.
46.     /* Add and store result in destination buffer. */
47.     *pDst++ = (q7_t) __SSAT((q15_t) *pSrcA++ + *pSrcB++, 8);
48.
49.     /* Decrement loop counter */
50.     blkCnt--;
51. }
52.
53. }
```

函数描述：

这个函数用于求两个 8 位定点数的和。

函数解析：

- ◆ 第 18 到 21 行，对于 M4 和 M7 带 DSP 单元的芯片使用。
- ◆ 第 22 到 25 行，对于不带 DSP 单元的 M0, M0+ 和 M3 使用。
- ◆ 第 42 到 51 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 函数 `read_q15x2_ia` 的原型如下：

```
__STATIC_FORCEINLINE void write_q7x4_ia (
    q7_t ** pQ7,
    q31_t    value)
{
    q31_t val = value;

    memcpy (*pQ7, &val, 4);
    *pQ7 += 4;
}
```

作用是读取 4 次 8 位数据，返回一个 32 位数据，并将数据地址递增，方便下次读取。



- ◆ `_QADD8` 实现四次 8 位数的加法饱和运算。输出结果的范围[0x80 0x7F]，超出这个结果将产生饱和结果，负数饱和到 0x80，正数饱和到 0x7F。

函数参数：

- ◆ 第 1 个参数是加数地址。
- ◆ 第 2 个参数是被加数地址。
- ◆ 第 3 个参数是和地址。
- ◆ 第 4 个参数是定点数个数，其实就是执行加法的次数。

11.4.5 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_Add
* 功能说明: 加法
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Add(void)
{
    float32_t pSrcA;
    float32_t pSrcB;
    float32_t pDst;

    q31_t pSrcA1;
    q31_t pSrcB1;
    q31_t pDst1;

    q15_t pSrcA2;
    q15_t pSrcB2;
    q15_t pDst2;

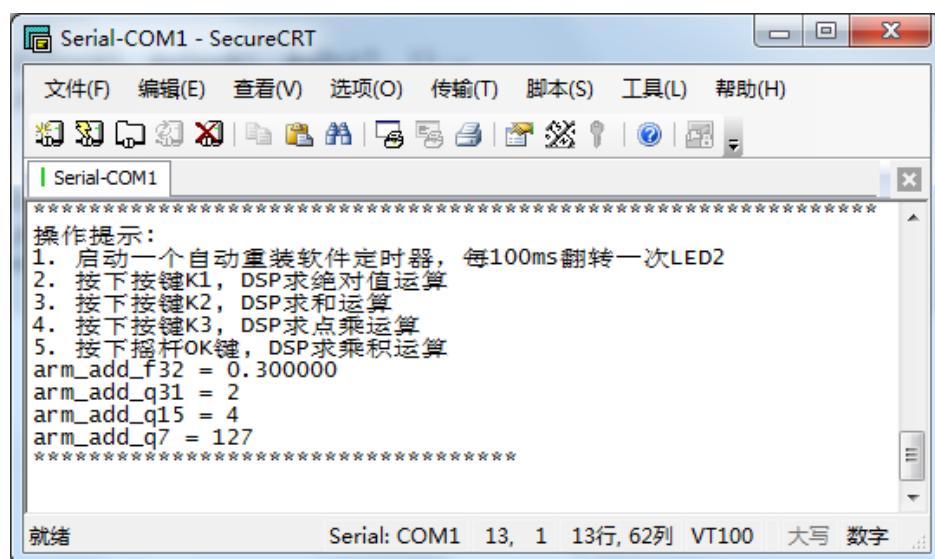
    q7_t pSrcA3;
    q7_t pSrcB3;
    q7_t pDst3;

    /*求和***** */
    pSrcA = 0.1f;
    pSrcB = 0.2f;
    arm_add_f32(&pSrcA, &pSrcB, &pDst, 1);
    printf("arm_add_f32 = %f\r\n", pDst);

    pSrcA1 = 1;
    pSrcB1 = 1;
    arm_add_q31(&pSrcA1, &pSrcB1, &pDst1, 1);
    printf("arm_add_q31 = %d\r\n", pDst1);

    pSrcA2 = 2;
    pSrcB2 = 2;
    arm_add_q15(&pSrcA2, &pSrcB2, &pDst2, 1);
    printf("arm_add_q15 = %d\r\n", pDst2);

    pSrcA3 = 30;
    pSrcB3 = 120;
    arm_add_q7(&pSrcA3, &pSrcB3, &pDst3, 1);
    printf("arm_add_q7 = %d\r\n", pDst3);
    printf("*****\r\n");
}
```

实验现象：

这里特别注意 Q7 的计算处理， $30+120$ 已经超出了 Q7 所能表示的最大值 127，经过饱和处理后，经过饱和后，输出为 127。

11.5 点乘 (Vector Dot Product)

这部分函数主要用于点乘，公式描述如下：

$\text{sum} = \text{pSrcA}[0]*\text{pSrcB}[0] + \text{pSrcA}[1]*\text{pSrcB}[1] + \dots + \text{pSrcA}[\text{blockSize}-1]*\text{pSrcB}[\text{blockSize}-1]$

11.5.1 函数 arm_dot_prod_f32

函数原型：

```

1. void arm_dot_prod_f32(
2.     const float32_t * pSrcA,
3.     const float32_t * pSrcB,
4.     uint32_t blockSize,
5.     float32_t * result)
6. {
7.     uint32_t blkCnt;           /* Loop counter */
8.     float32_t sum = 0.0f;      /* Temporary return variable */
9.
10. #if defined(ARM_MATH_NEON)
11.     float32x4_t vec1;
12.     float32x4_t vec2;
13.     float32x4_t res;
14.     float32x4_t accum = vdupq_n_f32(0);
15.
16.     /* Compute 4 outputs at a time */
17.     blkCnt = blockSize >> 2U;
18.
19.     vec1 = vld1q_f32(pSrcA);
20.     vec2 = vld1q_f32(pSrcB);
21.
22.     while (blkCnt > 0U)
23.     {

```



```
24.     /* C = A[0]*B[0] + A[1]*B[1] + A[2]*B[2] + ... + A[blockSize-1]*B[blockSize-1] */
25.     /* Calculate dot product and then store the result in a temporary buffer. */
26.
27.     accum = vmlaq_f32(accum, vec1, vec2);
28.
29.     /* Increment pointers */
30.     pSrcA += 4;
31.     pSrcB += 4;
32.
33.     vec1 = vld1q_f32(pSrcA);
34.     vec2 = vld1q_f32(pSrcB);
35.
36.     /* Decrement the loop counter */
37.     blkCnt--;
38. }
39.
40. #if __aarch64__
41.     sum = vpadds_f32(vpadd_f32(vget_low_f32(accum), vget_high_f32(accum)));
42. #else
43.     sum = (vpadd_f32(vget_low_f32(accum), vget_high_f32(accum))[0] + (vpadd_f32(vget_low_f32(accum),
44.         vget_high_f32(accum))[1];
45. #endif
46.
47.     /* Tail */
48.     blkCnt = blockSize & 0x3;
49.
50. #else
51. #if defined (ARM_MATH_LOOPUNROLL)
52.
53.     /* Loop unrolling: Compute 4 outputs at a time */
54.     blkCnt = blockSize >> 2U;
55.
56.     /* First part of the processing with loop unrolling. Compute 4 outputs at a time.
57.      ** a second loop below computes the remaining 1 to 3 samples. */
58.     while (blkCnt > 0U)
59.     {
60.         /* C = A[0]* B[0] + A[1]* B[1] + A[2]* B[2] + .....+ A[blockSize-1]* B[blockSize-1] */
61.
62.         /* Calculate dot product and store result in a temporary buffer. */
63.         sum += (*pSrcA++) * (*pSrcB++);
64.
65.         sum += (*pSrcA++) * (*pSrcB++);
66.
67.         sum += (*pSrcA++) * (*pSrcB++);
68.
69.         sum += (*pSrcA++) * (*pSrcB++);
70.
71.         /* Decrement loop counter */
72.         blkCnt--;
73.     }
74.
75.     /* Loop unrolling: Compute remaining outputs */
76.     blkCnt = blockSize % 0x4U;
77.
78. #else
79.
80.     /* Initialize blkCnt with number of samples */
81.     blkCnt = blockSize;
```



```
82.  
83. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */  
84. #endif /* #if defined(ARM_MATH_NEON) */  
85.  
86.     while (blkCnt > 0U)  
87.     {  
88.         /* C = A[0]* B[0] + A[1]* B[1] + A[2]* B[2] + .....+ A[blockSize-1]* B[blockSize-1] */  
89.  
90.         /* Calculate dot product and store result in a temporary buffer. */  
91.         sum += (*pSrcA++) * (*pSrcB++);  
92.  
93.         /* Decrement loop counter */  
94.         blkCnt--;  
95.     }  
96.  
97.     /* Store result in destination buffer */  
98.     *result = sum;  
99. }
```

函数描述：

这个函数用于求 32 位浮点数的点乘。

函数解析：

- ◆ 第 10 到 50 行，用于 NEON 指令集，当前的 CM 内核不支持。
- ◆ 第 51 到 78 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 86 到 95 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是乘数地址。
- ◆ 第 2 个参数是被乘数地址。
- ◆ 第 3 个参数是浮点数个数，其实就是执行点乘的次数。
- ◆ 第 4 个参数是结果地址。

11.5.2 函数 arm_dot_prod_q31

函数原型：

```
1. void arm_dot_prod_q31(  
2.     const q31_t * pSrcA,  
3.     const q31_t * pSrcB,  
4.             uint32_t blockSize,  
5.             q63_t * result)  
6. {  
7.     uint32_t blkCnt;                      /* Loop counter */  
8.     q63_t sum = 0;                         /* Temporary return variable */  
9.  
10. #if defined (ARM_MATH_LOOPUNROLL)  
11.  
12.     /* Loop unrolling: Compute 4 outputs at a time */  
13.     blkCnt = blockSize >> 2U;  
14.  
15.     while (blkCnt > 0U)  
16.     {  
17.         /* C = A[0]* B[0] + A[1]* B[1] + A[2]* B[2] + .....+ A[blockSize-1]* B[blockSize-1] */
```



```
18.     /* Calculate dot product and store result in a temporary buffer. */
19.     sum += ((q63_t) *pSrcA++ * *pSrcB++) >> 14U;
20.
21.     sum += ((q63_t) *pSrcA++ * *pSrcB++) >> 14U;
22.     sum += ((q63_t) *pSrcA++ * *pSrcB++) >> 14U;
23.     sum += ((q63_t) *pSrcA++ * *pSrcB++) >> 14U;
24.     sum += ((q63_t) *pSrcA++ * *pSrcB++) >> 14U;
25.
26.     /* Decrement loop counter */
27.     blkCnt--;
28. }
29.
30. /*
31.  * Loop unrolling: Compute remaining outputs */
32. blkCnt = blockSize % 0x4U;
33.
34.
35. #else
36.
37. /* Initialize blkCnt with number of samples */
38. blkCnt = blockSize;
39.
40. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
41.
42. while (blkCnt > 0U)
43. {
44.     /* C = A[0]* B[0] + A[1]* B[1] + A[2]* B[2] + .....+ A[blockSize-1]* B[blockSize-1] */
45.
46.     /* Calculate dot product and store result in a temporary buffer. */
47.     sum += ((q63_t) *pSrcA++ * *pSrcB++) >> 14U;
48.
49.     /* Decrement loop counter */
50.     blkCnt--;
51. }
52.
53. /* Store result in destination buffer in 16.48 format */
54. *result = sum;
55. }
```

函数描述：

这个函数用于求 32 位定点数的点乘。

函数解析：

- ◆ 第 10 到 35 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 42 到 51 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 两个 Q31 格式的 32 位数相乘，那么输出结果的格式是 $1.31 \times 1.31 = 2.62$ 。实际应用中基本不需要这么高的精度，这个函数将低 14 位的数据截取掉，反应在函数中就是两个数的乘积左移 14 位，也就是定点数的小数点也左移 14 位，那么最终的结果的格式是 16.48。所以只要乘累加的个数小于 2^{16} 就没有输出结果溢出的危险。

函数参数：

- ◆ 第 1 个参数是乘数地址。
- ◆ 第 2 个参数是被乘数地址。



- ◆ 第3个参数是定点数个数，其实就是执行点乘的次数。
- ◆ 第4个参数是结果地址。

11.5.3 函数 arm_dot_prod_q15

函数原型：

```
1. void arm_dot_prod_q15(
2.     const q15_t * pSrcA,
3.     const q15_t * pSrcB,
4.     uint32_t blockSize,
5.     q63_t * result)
6. {
7.     uint32_t blkCnt;           /* Loop counter */
8.     q63_t sum = 0;            /* Temporary return variable */
9.
10. #if defined (ARM_MATH_LOOPUNROLL)
11.
12.     /* Loop unrolling: Compute 4 outputs at a time */
13.     blkCnt = blockSize >> 2U;
14.
15.     while (blkCnt > 0U)
16.     {
17.         /* C = A[0]* B[0] + A[1]* B[1] + A[2]* B[2] + .....+ A[blockSize-1]* B[blockSize-1] */
18.
19. #if defined (ARM_MATH_DSP)
20.         /* Calculate dot product and store result in a temporary buffer. */
21.         sum = __SMLALD(read_q15x2_ia ((q15_t **) &pSrcA), read_q15x2_ia ((q15_t **) &pSrcB), sum);
22.         sum = __SMLALD(read_q15x2_ia ((q15_t **) &pSrcA), read_q15x2_ia ((q15_t **) &pSrcB), sum);
23. #else
24.         sum += (q63_t) ((q31_t) *pSrcA++ * *pSrcB++);
25.         sum += (q63_t) ((q31_t) *pSrcA++ * *pSrcB++);
26.         sum += (q63_t) ((q31_t) *pSrcA++ * *pSrcB++);
27.         sum += (q63_t) ((q31_t) *pSrcA++ * *pSrcB++);
28. #endif
29.
30.         /* Decrement loop counter */
31.         blkCnt--;
32.     }
33.
34.     /* Loop unrolling: Compute remaining outputs */
35.     blkCnt = blockSize % 0x4U;
36.
37. #else
38.
39.     /* Initialize blkCnt with number of samples */
40.     blkCnt = blockSize;
41.
42. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
43.
44.     while (blkCnt > 0U)
45.     {
46.         /* C = A[0]* B[0] + A[1]* B[1] + A[2]* B[2] + .....+ A[blockSize-1]* B[blockSize-1] */
47.
48.         /* Calculate dot product and store result in a temporary buffer. */
49. // #if defined (ARM_MATH_DSP)
50. //     sum = __SMLALD(*pSrcA++, *pSrcB++, sum);
```



```
51. //#else
52.     sum += (q63_t)((q31_t)*pSrcA++ * *pSrcB++);
53. //#endif
54.
55.     /* Decrement loop counter */
56.     blkCnt--;
57. }
58.
59. /* Store result in destination buffer in 34.30 format */
60. *result = sum;
61. }
```

函数描述：

这个函数用于求 32 位定点数的点乘。

函数解析：

- ◆ 第 10 到 37 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 44 到 57 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 两个 Q15 格式的数据相乘，那么输出结果的格式是 $1.15 \times 1.15 = 2.30$ ，这个函数将输出结果赋值给了 64 位变量，那么输出结果就是 34.30 格式。所以基本没有溢出的危险。
- ◆ __SMLALD 也是 SIMD 指令，实现两个 16 位数相乘，并把结果累加给 64 位变量。

函数参数：

- ◆ 第 1 个参数是乘数地址。
- ◆ 第 2 个参数是被乘数地址。
- ◆ 第 3 个参数是定点数个数，其实就是执行点乘的次数。
- ◆ 第 4 个参数是结果地址。

11.5.4 函数 arm_dot_prod_q7

函数原型：

```
1. void arm_dot_prod_q7(
2.     const q7_t * pSrcA,
3.     const q7_t * pSrcB,
4.     uint32_t blockSize,
5.     q31_t * result)
6. {
7.     uint32_t blkCnt;                      /* Loop counter */
8.     q31_t sum = 0;                        /* Temporary return variable */
9.
10. #if defined (ARM_MATH_LOOPUNROLL)
11.
12. #if defined (ARM_MATH_DSP)
13.     q31_t input1, input2;                /* Temporary variables */
14.     q31_t inA1, inA2, inB1, inB2;        /* Temporary variables */
15. #endif
16.
17.     /* Loop unrolling: Compute 4 outputs at a time */
18.     blkCnt = blockSize >> 2U;
19.
20.     while (blkCnt > 0U)
21.     {
```



```
22.     /* C = A[0]* B[0] + A[1]* B[1] + A[2]* B[2] + .....+ A[blockSize-1]* B[blockSize-1] */  
23.  
24. #if defined (ARM_MATH_DSP)  
25.     /* read 4 samples at a time from sourceA */  
26.     input1 = read_q7x4_ia ((q7_t **) &pSrcA);  
27.     /* read 4 samples at a time from sourceB */  
28.     input2 = read_q7x4_ia ((q7_t **) &pSrcB);  
29.  
30.     /* extract two q7_t samples to q15_t samples */  
31.     inA1 = __SXTB16(__ROR(input1, 8));  
32.     /* extract reminaing two samples */  
33.     inA2 = __SXTB16(input1);  
34.     /* extract two q7_t samples to q15_t samples */  
35.     inB1 = __SXTB16(__ROR(input2, 8));  
36.     /* extract reminaing two samples */  
37.     inB2 = __SXTB16(input2);  
38.  
39.     /* multiply and accumulate two samples at a time */  
40.     sum = __SMLAD(inA1, inB1, sum);  
41.     sum = __SMLAD(inA2, inB2, sum);  
42. #else  
43.     sum += (q31_t) ((q15_t) *pSrcA++ * *pSrcB++);  
44.     sum += (q31_t) ((q15_t) *pSrcA++ * *pSrcB++);  
45.     sum += (q31_t) ((q15_t) *pSrcA++ * *pSrcB++);  
46.     sum += (q31_t) ((q15_t) *pSrcA++ * *pSrcB++);  
47. #endif  
48.  
49.     /* Decrement loop counter */  
50.     blkCnt--;  
51. }52.  
53. /* Loop unrolling: Compute remaining outputs */  
54. blkCnt = blockSize % 0x4U;  
55.  
56. #else  
57.  
58.     /* Initialize blkCnt with number of samples */  
59.     blkCnt = blockSize;  
60.  
61. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */  
62.  
63.     while (blkCnt > 0U)  
64.     {  
65.         /* C = A[0]* B[0] + A[1]* B[1] + A[2]* B[2] + .....+ A[blockSize-1]* B[blockSize-1] */  
66.  
67.         /* Calculate dot product and store result in a temporary buffer. */  
68. // #if defined (ARM_MATH_DSP)  
69. //     sum = __SMLAD(*pSrcA++, *pSrcB++, sum);  
70. // #else  
71.     sum += (q31_t) ((q15_t) *pSrcA++ * *pSrcB++);  
72. // #endif  
73.  
74.     /* Decrement loop counter */  
75.     blkCnt--;  
76. }77.  
78. /* Store result in destination buffer in 18.14 format */  
79. *result = sum;
```



80. }

函数描述：

这个函数用于求 8 位定点数的点乘。

函数解析：

- ◆ 第 10 到 56 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 63 到 76 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 两个 Q8 格式的数据相乘，那么输出结果就是 $1.7 * 1.7 = 2.14$ 格式。这里将最终结果赋值给了 32 位的变量，那么最终的格式就是 18.14。如果乘累加的个数小于 2^{18} 那么就不会有溢出的危险。
- ◆ `_SXTB16` 也是 SIMD 指令，用于将两个 8 位的有符号数扩展成 16 位。`_ROR` 用于实现数据的循环右移。
- ◆ `_SMLAD` 也是 SIMD 指令，用于实现如下功能：

`sum = _SMLAD(x, y, z)`

`sum = z + ((short)(x>>16) * (short)(y>>16)) + ((short)x * (short)y)`

函数参数：

- ◆ 第 1 个参数是乘数地址。
- ◆ 第 2 个参数是被乘数地址。
- ◆ 第 3 个参数是定点数个数，其实就是执行点乘的次数。
- ◆ 第 4 个参数是结果地址。

11.5.5 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_DotProduct
* 功能说明: 点乘
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_DotProduct(void)
{
    float32_t pSrcA[5] = {1.0f, 1.0f, 1.0f, 1.0f, 1.0f};
    float32_t pSrcB[5] = {1.0f, 1.0f, 1.0f, 1.0f, 1.0f};
    float32_t result;

    q31_t pSrcA1[5] = {0x7fffffff, 1, 1, 1, 1};
    q31_t pSrcB1[5] = {1, 1, 1, 1, 1};
    q63_t result1;

    q15_t pSrcA2[5] = {1, 1, 1, 1, 1};
    q15_t pSrcB2[5] = {1, 1, 1, 1, 1};
    q63_t result2;

    q7_t pSrcA3[5] = {1, 1, 1, 1, 1};
    q7_t pSrcB3[5] = {1, 1, 1, 1, 1};
    q31_t result3;

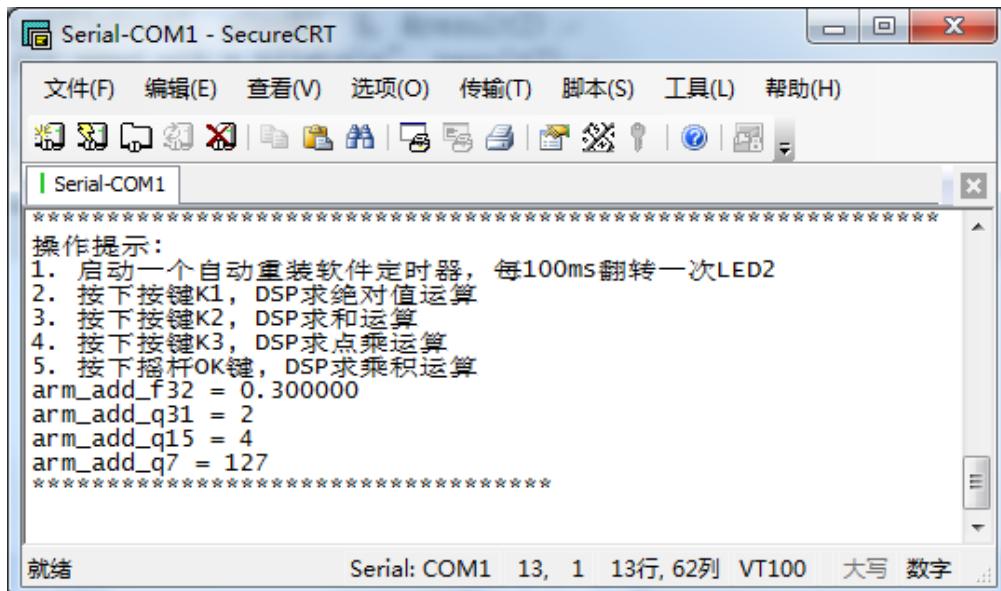
    /*求点乘*****
    arm_dot_prod_f32(pSrcA, pSrcB, 5, &result);
}
```

```
printf("arm_dot_prod_f32 = %f\r\n", result);
arm_dot_prod_q31(pSrcA1, pSrcB1, 5, &result1);
printf("arm_dot_prod_q31 = %lld\r\n", result1);

arm_dot_prod_q15(pSrcA2, pSrcB2, 5, &result2);
printf("arm_dot_prod_q15 = %lld\r\n", result2);

arm_dot_prod_q7(pSrcA3, pSrcB3, 5, &result3);
printf("arm_dot_prod_q7 = %d\r\n", result3);
printf("*****\r\n");
}
```

实验现象：



11.6 乘法 (Vector Multiplication)

这部分函数主要用于乘法，公式描述如下：

$$pDst[n] = pSrcA[n] * pSrcB[n], \quad 0 \leq n < blockSize.$$

11.6.1 函数 arm_mult_f32

函数原型：

```
1. void arm_mult_f32(
2.     const float32_t * pSrcA,
3.     const float32_t * pSrcB,
4.     float32_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined(ARM_MATH_NEON)
10.    float32x4_t vec1;
11.    float32x4_t vec2;
12.    float32x4_t res;
13.
14.    /* Compute 4 outputs at a time */
15.    blkCnt = blockSize >> 2U;
```



```
16.
17.     while (blkCnt > 0U)
18.     {
19.         /* C = A * B */
20.
21.         /* Multiply the inputs and then store the results in the destination buffer. */
22.         vec1 = vld1q_f32(pSrcA);
23.         vec2 = vld1q_f32(pSrcB);
24.         res = vmulq_f32(vec1, vec2);
25.         vst1q_f32(pDst, res);
26.
27.         /* Increment pointers */
28.         pSrcA += 4;
29.         pSrcB += 4;
30.         pDst += 4;
31.
32.         /* Decrement the loop counter */
33.         blkCnt--;
34.     }
35.
36.     /* Tail */
37.     blkCnt = blockSize & 0x3;
38.
39. #else
40. #if defined (ARM_MATH_LOOPUNROLL)
41.
42.     /* Loop unrolling: Compute 4 outputs at a time */
43.     blkCnt = blockSize >> 2U;
44.
45.     while (blkCnt > 0U)
46.     {
47.         /* C = A * B */
48.
49.         /* Multiply inputs and store result in destination buffer. */
50.         *pDst++ = (*pSrcA++) * (*pSrcB++);
51.
52.         *pDst++ = (*pSrcA++) * (*pSrcB++);
53.
54.         *pDst++ = (*pSrcA++) * (*pSrcB++);
55.
56.         *pDst++ = (*pSrcA++) * (*pSrcB++);
57.
58.         /* Decrement loop counter */
59.         blkCnt--;
60.     }
61.
62.     /* Loop unrolling: Compute remaining outputs */
63.     blkCnt = blockSize % 0x4U;
64.
65. #else
66.
67.     /* Initialize blkCnt with number of samples */
68.     blkCnt = blockSize;
69.
70. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
71. #endif /* #if defined(ARM_MATH_NEON) */
72.
73.     while (blkCnt > 0U)
```



```
74.    {
75.        /* C = A * B */
76.
77.        /* Multiply input and store result in destination buffer. */
78.        *pDst++ = (*pSrcA++) * (*pSrcB++);
79.
80.        /* Decrement loop counter */
81.        blkCnt--;
82.    }
83.
84. }
```

函数描述：

这个函数用于求 32 位浮点数的乘法。

函数解析：

- ◆ 第 9 到 39 行，用于 NEON 指令集，当前的 CM 内核不支持。
- ◆ 第 40 到 65 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 73 到 82 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是乘数地址。
- ◆ 第 2 个参数是被乘数地址。
- ◆ 第 3 个参数是结果地址。
- ◆ 第 4 个参数是数据块大小，其实就是执行乘法的次数。

11.6.2 函数 arm_mult_q31

函数原型：

```
1. void arm_mult_q31(
2.     const q31_t * pSrcA,
3.     const q31_t * pSrcB,
4.     q31_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt;                      /* Loop counter */
8.     q31_t out;                           /* Temporary output variable */
9.
10. #if defined (ARM_MATH_LOOPUNROLL)
11.
12.     /* Loop unrolling: Compute 4 outputs at a time */
13.     blkCnt = blockSize >> 2U;
14.
15.     while (blkCnt > 0U)
16.     {
17.         /* C = A * B */
18.
19.         /* Multiply inputs and store result in destination buffer. */
20.         out = ((q63_t) *pSrcA++ * *pSrcB++) >> 32;
21.         out = __SSAT(out, 31);
22.         *pDst++ = out << 1U;
23.
24.         out = ((q63_t) *pSrcA++ * *pSrcB++) >> 32;
```



```
25.     out = __SSAT(out, 31);
26.     *pDst++ = out << 1U;
27.
28.     out = ((q63_t) *pSrcA++ * *pSrcB++) >> 32;
29.     out = __SSAT(out, 31);
30.     *pDst++ = out << 1U;
31.
32.     out = ((q63_t) *pSrcA++ * *pSrcB++) >> 32;
33.     out = __SSAT(out, 31);
34.     *pDst++ = out << 1U;
35.
36.     /* Decrement loop counter */
37.     blkCnt--;
38. }
39.
40. /* Loop unrolling: Compute remaining outputs */
41. blkCnt = blockSize % 0x4U;
42.
43. #else
44.
45. /* Initialize blkCnt with number of samples */
46. blkCnt = blockSize;
47.
48. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
49.
50. while (blkCnt > 0U)
51. {
52.     /* C = A * B */
53.
54.     /* Multiply inputs and store result in destination buffer. */
55.     out = ((q63_t) *pSrcA++ * *pSrcB++) >> 32;
56.     out = __SSAT(out, 31);
57.     *pDst++ = out << 1U;
58.
59.     /* Decrement loop counter */
60.     blkCnt--;
61. }
62.
63. }
```

函数描述：

这个函数用于求 32 位定点数的乘法。

函数解析：

- ◆ 这个函数使用了饱和运算 __SSAT, 所得结果是 Q31 格式，范围 Q31 range[0x80000000 0xFFFFFFFF]
- ◆ 第 10 到 43 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 50 到 61 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 第 20 行，所得乘积左移 32 位。
- ◆ 第 21 行，实现 31 位精度的饱和运算。
- ◆ 第 22 行，右移一位，保证所得结果是 Q31 格式。

函数参数：



- ◆ 第1个参数是乘数地址。
- ◆ 第2个参数是被乘数地址。
- ◆ 第3个参数是结果地址。
- ◆ 第4个参数是数据块大小，其实就是执行乘法的次数。

11.6.3 函数 arm_mult_q15

函数原型：

```
1. void arm_mult_q15(
2.     const q15_t * pSrcA,
3.     const q15_t * pSrcB,
4.     q15_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt;           /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. #if defined (ARM_MATH_DSP)
12.     q31_t inA1, inA2, inB1, inB2;          /* Temporary input variables */
13.     q15_t out1, out2, out3, out4;          /* Temporary output variables */
14.     q31_t mul1, mul2, mul3, mul4;          /* Temporary variables */
15. #endif
16.
17. /* Loop unrolling: Compute 4 outputs at a time */
18. blkCnt = blockSize >> 2U;
19.
20. while (blkCnt > 0U)
21. {
22.     /* C = A * B */
23.
24. #if defined (ARM_MATH_DSP)
25.     /* read 2 samples at a time from sourceA */
26.     inA1 = read_q15x2_ia ((q15_t **) &pSrcA);
27.     /* read 2 samples at a time from sourceB */
28.     inB1 = read_q15x2_ia ((q15_t **) &pSrcB);
29.     /* read 2 samples at a time from sourceA */
30.     inA2 = read_q15x2_ia ((q15_t **) &pSrcA);
31.     /* read 2 samples at a time from sourceB */
32.     inB2 = read_q15x2_ia ((q15_t **) &pSrcB);
33.
34.     /* multiply mul = sourceA * sourceB */
35.     mul1 = (q31_t) ((q15_t) (inA1 >> 16) * (q15_t) (inB1 >> 16));
36.     mul2 = (q31_t) ((q15_t) (inA1        ) * (q15_t) (inB1        ));
37.     mul3 = (q31_t) ((q15_t) (inA2 >> 16) * (q15_t) (inB2 >> 16));
38.     mul4 = (q31_t) ((q15_t) (inA2        ) * (q15_t) (inB2        ));
39.
40.     /* saturate result to 16 bit */
41.     out1 = (q15_t) __SSAT(mul1 >> 15, 16);
42.     out2 = (q15_t) __SSAT(mul2 >> 15, 16);
43.     out3 = (q15_t) __SSAT(mul3 >> 15, 16);
44.     out4 = (q15_t) __SSAT(mul4 >> 15, 16);
45.
46.     /* store result to destination */
47. #ifndef ARM_MATH_BIG_ENDIAN
```



```
48.     write_q15x2_ia (&pDst, __PKHBT(out2, out1, 16));
49.     write_q15x2_ia (&pDst, __PKHBT(out4, out3, 16));
50. #else
51.     write_q15x2_ia (&pDst, __PKHBT(out1, out2, 16));
52.     write_q15x2_ia (&pDst, __PKHBT(out3, out4, 16));
53. #endif /* #ifndef ARM_MATH_BIG_ENDIAN */
54.
55. #else
56.     *pDst++ = (q15_t) __SSAT(((q31_t) (*pSrcA++) * (*pSrcB++)) >> 15), 16);
57.     *pDst++ = (q15_t) __SSAT(((q31_t) (*pSrcA++) * (*pSrcB++)) >> 15), 16);
58.     *pDst++ = (q15_t) __SSAT(((q31_t) (*pSrcA++) * (*pSrcB++)) >> 15), 16);
59.     *pDst++ = (q15_t) __SSAT(((q31_t) (*pSrcA++) * (*pSrcB++)) >> 15), 16);
60. #endif
61.
62.     /* Decrement loop counter */
63.     blkCnt--;
64. }
65.
66. /* Loop unrolling: Compute remaining outputs */
67. blkCnt = blockSize % 0x4U;
68.
69. #else
70.
71.     /* Initialize blkCnt with number of samples */
72.     blkCnt = blockSize;
73.
74. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
75.
76. while (blkCnt > 0U)
77. {
78.     /* C = A * B */
79.
80.     /* Multiply inputs and store result in destination buffer. */
81.     *pDst++ = (q15_t) __SSAT(((q31_t) (*pSrcA++) * (*pSrcB++)) >> 15), 16);
82.
83.     /* Decrement loop counter */
84.     blkCnt--;
85. }
86.
87. }
```

函数描述：

这个函数用于求 16 位定点数的乘法。

函数解析：

- ◆ 这个函数使用了饱和运算 __SSAT, 所得结果是 Q31 格式，范围 Q31 range[0x80000000 0x7FFFFFFF]。
- ◆ 第 9 到 69 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 79 到 85 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 第 26 行，一次读取两个 Q15 格式的数据。
- ◆ 第 35 到 38 行，将四组数的乘积保存到 Q31 格式的变量 mul1, mul2, mul3, mul4。
- ◆ 第 41 到 44 行，丢弃 32 位数据的低 15 位，并把最终结果饱和到 16 位精度。



- ◆ 第 51 到 52 行的 SIMD 指令 _PKHBT，将两个 Q15 格式的数据保存的结果数组中，从而一个指令周期就能完成两个数据的存储。

函数参数：

- ◆ 第 1 个参数是乘数地址。
- ◆ 第 2 个参数是被乘数地址。
- ◆ 第 3 个参数是结果地址。
- ◆ 第 4 个参数是数据块大小，其实就是执行乘法的次数。

11.6.4 函数 arm_mult_q7

函数原型：

```
1. void arm_mult_q7(
2.     const q7_t * pSrcA,
3.     const q7_t * pSrcB,
4.     q7_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. #if defined (ARM_MATH_DSP)
12.     q7_t out1, out2, out3, out4; /* Temporary output variables */
13. #endif
14.
15. /* Loop unrolling: Compute 4 outputs at a time */
16. blkCnt = blockSize >> 2U;
17.
18. while (blkCnt > 0U)
19. {
20.     /* C = A * B */
21.
22. #if defined (ARM_MATH_DSP)
23.     /* Multiply inputs and store results in temporary variables */
24.     out1 = (q7_t) __SSAT(((q15_t) (*pSrcA++) * (*pSrcB++)) >> 7), 8);
25.     out2 = (q7_t) __SSAT(((q15_t) (*pSrcA++) * (*pSrcB++)) >> 7), 8);
26.     out3 = (q7_t) __SSAT(((q15_t) (*pSrcA++) * (*pSrcB++)) >> 7), 8);
27.     out4 = (q7_t) __SSAT(((q15_t) (*pSrcA++) * (*pSrcB++)) >> 7), 8);
28.
29.     /* Pack and store result in destination buffer (in single write) */
30.     write_q7x4_ia (&pDst, __PACKq7(out1, out2, out3, out4));
31. #else
32.     *pDst++ = (q7_t) __SSAT(((q15_t) (*pSrcA++) * (*pSrcB++)) >> 7), 8);
33.     *pDst++ = (q7_t) __SSAT(((q15_t) (*pSrcA++) * (*pSrcB++)) >> 7), 8);
34.     *pDst++ = (q7_t) __SSAT(((q15_t) (*pSrcA++) * (*pSrcB++)) >> 7), 8);
35.     *pDst++ = (q7_t) __SSAT(((q15_t) (*pSrcA++) * (*pSrcB++)) >> 7), 8);
36. #endif
37.
38.     /* Decrement loop counter */
39.     blkCnt--;
40. }
41.
42. /* Loop unrolling: Compute remaining outputs */
```



```
43.     blkCnt = blockSize % 0x4U;
44.
45. #else
46.
47.     /* Initialize blkCnt with number of samples */
48.     blkCnt = blockSize;
49.
50. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
51.
52.     while (blkCnt > 0U)
53.     {
54.         /* C = A * B */
55.
56.         /* Multiply input and store result in destination buffer. */
57.         *pDst++ = (q7_t) __SSAT(((q15_t) (*pSrcA++) * (*pSrcB++)) >> 7), 8);
58.
59.         /* Decrement loop counter */
60.         blkCnt--;
61.     }
62.
63. }
```

函数描述：

这个函数用于求 8 位定点数的乘法。

函数解析：

- ◆ 这个函数使用了饱和算法 __SSAT，所得结果是 Q7 格式，范围 [0x80 0x7F]
- ◆ 第 9 到 45 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 52 到 61 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 第 24 到 27 行，将两个 Q7 格式的数据乘积左移 7 位，也就是丢掉低 7 位的数据，并将所得结果饱和到 8 位精度。
- ◆ 第 30 行，__PACKq7 函数可以在一个时钟周期就能完成相应操作。

函数参数：

- ◆ 第 1 个参数是乘数地址。
- ◆ 第 2 个参数是被乘数地址。
- ◆ 第 3 个参数是结果地址。
- ◆ 第 4 个参数是数据块大小，其实就是执行乘法的次数。

11.6.5 使用举例

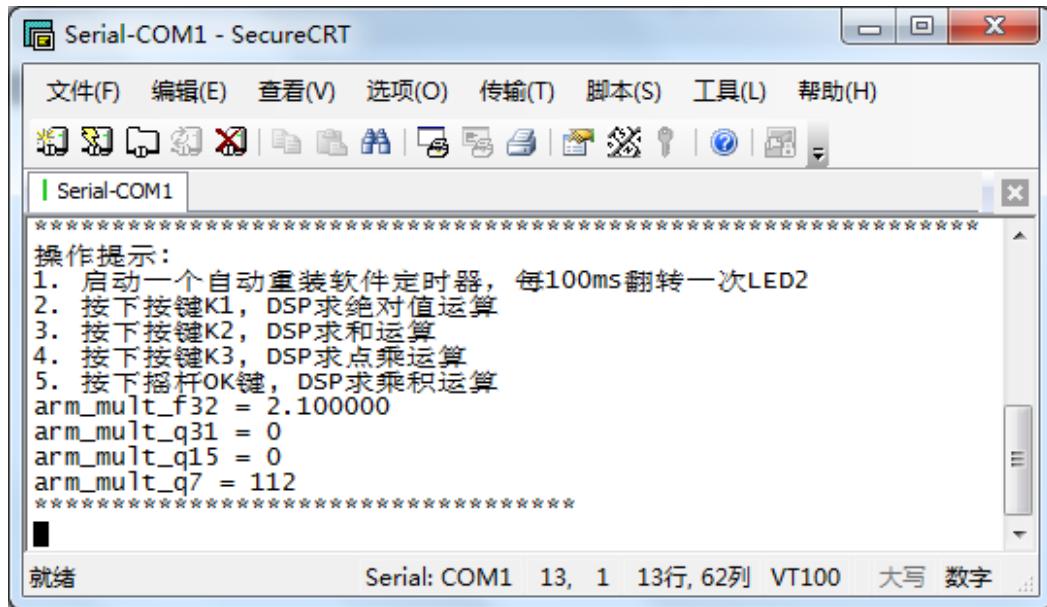
程序设计：

```
/*
*****
* 函数名: DSP_Multiplication
* 功能说明: 乘法
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Multiplication(void)
{
    float32_t pSrcA[5] = {1.0f, 1.0f, 1.0f, 1.0f, 1.0f};
```



```
float32_t pSrcB[5] = {1.0f, 1.0f, 1.0f, 1.0f, 1.0f};  
float32_t pDst[5];  
  
q31_t pSrcA1[5] = {1, 1, 1, 1, 1};  
q31_t pSrcB1[5] = {1, 1, 1, 1, 1};  
q31_t pDst1[5];  
  
q15_t pSrcA2[5] = {1, 1, 1, 1, 1};  
q15_t pSrcB2[5] = {1, 1, 1, 1, 1};  
q15_t pDst2[5];  
  
q7_t pSrcA3[5] = {0x70, 1, 1, 1, 1};  
q7_t pSrcB3[5] = {0x7f, 1, 1, 1, 1};  
q7_t pDst3[5];  
  
/*求乘积*****  
pSrcA[0] += 1.1f;  
arm_mult_f32(pSrcA, pSrcB, pDst, 5);  
printf("arm_mult_f32 = %f\r\n", pDst[0]);  
  
pSrcA1[0] += 1;  
arm_mult_q31(pSrcA1, pSrcB1, pDst1, 5);  
printf("arm_mult_q31 = %d\r\n", pDst1[0]);  
  
pSrcA2[0] += 1;  
arm_mult_q15(pSrcA2, pSrcB2, pDst2, 5);  
printf("arm_mult_q15 = %d\r\n", pDst2[0]);  
  
pSrcA3[0] += 1;  
arm_mult_q7(pSrcA3, pSrcB3, pDst3, 5);  
printf("arm_mult_q7 = %d\r\n", pDst3[0]);  
printf("*****\r\n");  
}  
}
```

实验现象：



这里特别注意为什么 Q31 和 Q15 结算的输出结果会有 0，关于这个问题，在此贴进行了详细说明：

[http://www.armbbs.cn/forum.php?mod=viewthread&tid=95194。](http://www.armbbs.cn/forum.php?mod=viewthread&tid=95194)

11.7 实验例程说明 (MDK)

配套例子：

V7-206_DSP 基础运算 (绝对值, 求和, 乘法和点乘)

实验目的：

1. 学习基础运算 (绝对值, 求和, 乘法和点乘)。

实验内容：

1. 启动一个自动重装软件定时器, 每 100ms 翻转一次 LED2。
2. 按下按键 K1, DSP 求绝对值运算。
3. 按下按键 K2, DSP 求和运算。
4. 按下按键 K3, DSP 求点乘运算。
5. 按下摇杆 OK 键, DSP 求乘积运算。

使用 AC6 注意事项

特别注意附件章节 C 的问题

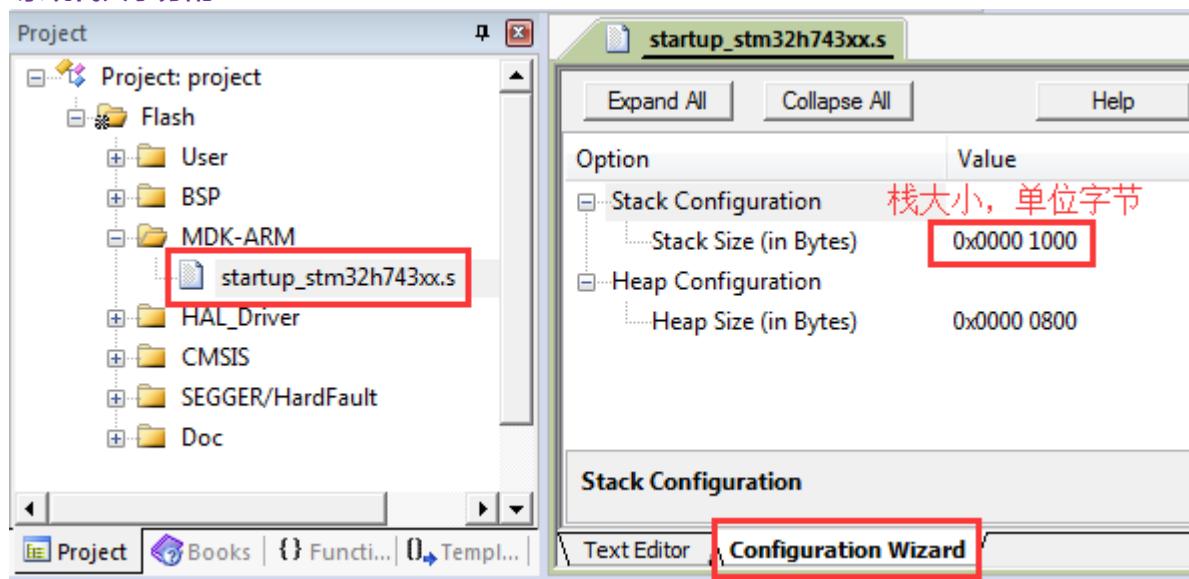
上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

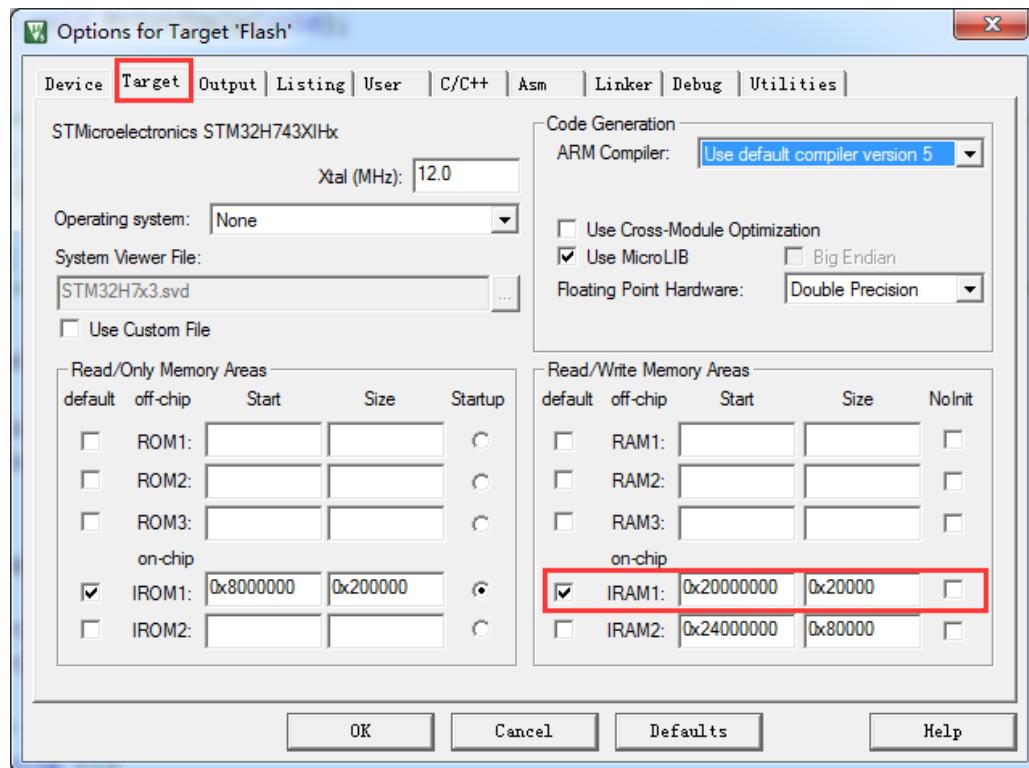
详见本章的 4.5, 5.5 和 6.5 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 按下按键 K1, DSP 求绝对值运算。
- 按下按键 K2, DSP 求和运算。
- 按下按键 K3, DSP 求点乘运算。
- 按下摇杆 OK 键, DSP 求乘积运算。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint8_t ucValue;

    bsp_Init(); /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */
}
```



```
PrintfHelp(); /* 打印操作提示信息 */

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle(); /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1: /* K1 键按下，求绝对值 */
                DSP_ABS();
                break;

            case KEY_DOWN_K2: /* K2 键按下，求和 */
                DSP_Add();
                break;

            case KEY_DOWN_K3: /* K3 键按下，求点乘 */
                DSP_DotProduct();
                break;

            case JOY_DOWN_OK: /* 摆杆 OK 键按下，求乘积 */
                DSP_Multiplication();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}

}
```

11.8 实验例程说明 (IAR)

配套例子：

V7-206_DSP 基础运算 (绝对值, 求和, 乘法和点乘)

实验目的：

1. 学习基础运算 (绝对值, 求和, 乘法和点乘)。

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1, DSP 求绝对值运算。
3. 按下按键 K2, DSP 求和运算。
4. 按下按键 K3, DSP 求点乘运算。
5. 按下摇杆 OK 键, DSP 求乘积运算。

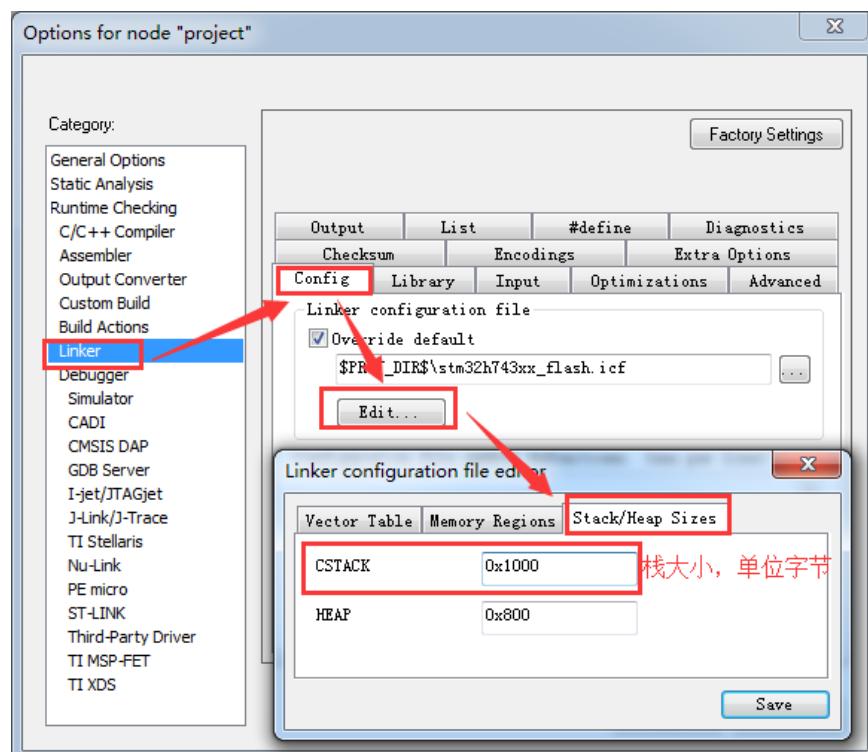
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

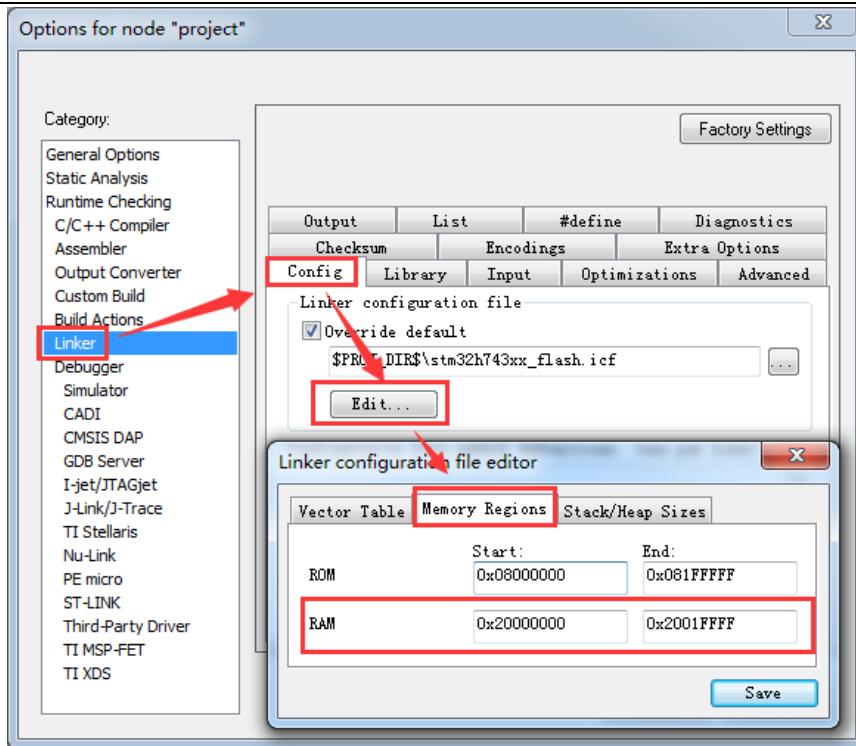
详见本章的 4.5, 5.5 和 6.5 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作:

- 按下按键 K1, DSP 求绝对值运算。
- 按下按键 K2, DSP 求和运算。
- 按下按键 K3, DSP 求点乘运算。
- 按下摇杆 OK 键, DSP 求乘积运算。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint8_t ucValue;

    bsp_Init(); /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下，求绝对值 */
                DSP_ABS();
                break;

            case KEY_DOWN_K2:          /* K2 键按下，求和 */
                DSP_Add();
                break;

            case KEY_DOWN_K3:          /* K3 键按下，求点乘 */
                DSP_DotProduct();
                break;

            case JOY_DOWN_OK:          /* 摆杆 OK 键按下，求乘积 */
                DSP_Multiplication();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}
```

11.9 总结

本期教程就跟大家讲这么多，还是那句话，可以自己写些代码调用本期教程中讲的这几个函数，如果可以的话，可以自己尝试直接调用这些 DSP 指令。



第12章 DSP 基础函数-相反数，偏移，移位，减法和比例因子

本期教程主要讲基本函数中的相反数，偏移，移位，减法和比例因子。

12.1 初学者重要提示

12.2 DSP 基础运算指令

12.3 相反数 (Vector Negate)

12.4 偏移 (Vector Offset)

12.5 移位 (Vector Shift)

12.6 减法 (Vector Sub)

12.7 比例因子 (Vector Scale)

12.8 实验例程说明 (MDK)

12.9 实验例程说明 (IAR)

12.10 总结

12.1 初学者重要提示

在这里简单的跟大家介绍一下 DSP 库中函数的通用格式，后面就不再赘述了。

- ◆ 这些函数基本都是支持重入的。
- ◆ 基本每个函数都有四种数据类型，F32, Q31, Q15, Q7。
- ◆ 函数中数值的处理基本都是 4 个为一组，这么做的原因是 F32, Q31, Q15, Q7 就可以统一采用一个程序设计架构，便于管理。更重要的是可以在 Q15 和 Q7 数据处理中很好的发挥 SIMD 指令的作用（因为 4 个为一组的话，可以用 SIMD 指令正好处理 2 个 Q15 数据或者 4 个 Q7 数据）。
- ◆ 部分函数是支持目标指针和源指针指向相同的缓冲区。
- ◆ 为什么定点 DSP 运算输出的时候容易出现结果为 0 的情况：

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=95194>

12.2 DSP 基础运算指令

本章用到基础运算指令：

- ◆ 相反数函数用到 QSUB, QSUB16 和 QSUB8。



- ◆ 偏移函数用到 QADD, QADD16 和 QADD8。
- ◆ 移位函数用到 PKHBT 和 SSAT。
- ◆ 减法函数用到 QSUB, QSUB16 和 QSUB8。
- ◆ 比例因子函数用到 PKHBT 和 SSAT。

这里特别注意饱和运算问题，在第 11 章的第 2 小节有详细说明。

12.3 相反数 (Vector Negate)

这部分函数主要用于求相反数，公式描述如下：

$pDst[n] = -pSrc[n]$, $0 \leq n < blockSize$.

特别注意，这部分函数支持目标指针和源指针指向相同的缓冲区。

12.3.1 函数 arm_negate_f32

函数原型：

```
1. void arm_negate_f32(
2.     const float32_t * pSrc,
3.     float32_t * pDst,
4.     uint32_t blockSize)
5. {
6.     uint32_t blkCnt; /* Loop counter */
7.
8. #if defined(ARM_MATH_NEON_EXPERIMENTAL)
9.     float32x4_t vec1;
10.    float32x4_t res;
11.
12.    /* Compute 4 outputs at a time */
13.    blkCnt = blockSize >> 2U;
14.
15.    while (blkCnt > 0U)
16.    {
17.        /* C = -A */
18.
19.        /* Negate and then store the results in the destination buffer. */
20.        vec1 = vld1q_f32(pSrc);
21.        res = vnegq_f32(vec1);
22.        vst1q_f32(pDst, res);
23.
24.        /* Increment pointers */
25.        pSrc += 4;
26.        pDst += 4;
27.
28.        /* Decrement the loop counter */
29.        blkCnt--;
30.    }
31.
32.    /* Tail */
33.    blkCnt = blockSize & 0x3;
34.
```



```
35. #else
36. #if defined (ARM_MATH_LOOPUNROLL)
37.
38. /* Loop unrolling: Compute 4 outputs at a time */
39. blkCnt = blockSize >> 2U;
40.
41. while (blkCnt > 0U)
42. {
43.     /* C = -A */
44.
45.     /* Negate and store result in destination buffer. */
46.     *pDst++ = -*pSrc++;
47.
48.     *pDst++ = -*pSrc++;
49.
50.     *pDst++ = -*pSrc++;
51.
52.     *pDst++ = -*pSrc++;
53.
54.     /* Decrement loop counter */
55.     blkCnt--;
56. }
57.
58. /* Loop unrolling: Compute remaining outputs */
59. blkCnt = blockSize % 0x4U;
60.
61. #else
62.
63. /* Initialize blkCnt with number of samples */
64. blkCnt = blockSize;
65.
66. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
67. #endif /* #if defined(ARM_MATH_NEON_EXPERIMENTAL) */
68.
69. while (blkCnt > 0U)
70. {
71.     /* C = -A */
72.
73.     /* Negate and store result in destination buffer. */
74.     *pDst++ = -*pSrc++;
75.
76.     /* Decrement loop counter */
77.     blkCnt--;
78. }
79.
80. }
```

函数描述：

这个函数用于求 32 位浮点数的相反数。

函数解析：

- ◆ 第 8 到 35 行，用于 NEON 指令集，当前的 CM 内核不支持。
- ◆ 第 36 到 61 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 浮点数的相反数求解比较简单，直接在相应的变量前加上负号即可。
- ◆ 第 69 到 78 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。



函数参数:

- ◆ 第 1 个参数是原数据地址。
- ◆ 第 2 个参数是求相反数后目的数据地址。
- ◆ 第 3 个参数转换的数据个数，这里是指的浮点数个数。

12.3.2 函数 arm_negate_q31

函数原型:

```
1. void arm_negate_q31(
2.     const q31_t * pSrc,
3.     q31_t * pDst,
4.     uint32_t blockSize)
5. {
6.     uint32_t blkCnt; /* Loop counter */
7.     q31_t in; /* Temporary input variable */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. /* Loop unrolling: Compute 4 outputs at a time */
12. blkCnt = blockSize >> 2U;
13.
14. while (blkCnt > 0U)
15. {
16.     /* C = -A */
17.
18.     /* Negate and store result in destination buffer. */
19.     in = *pSrc++;
20. #if defined (ARM_MATH_DSP)
21.     *pDst++ = __QSUB(0, in);
22. #else
23.     *pDst++ = (in == INT32_MIN) ? INT32_MAX : -in;
24. #endif
25.
26.     in = *pSrc++;
27. #if defined (ARM_MATH_DSP)
28.     *pDst++ = __QSUB(0, in);
29. #else
30.     *pDst++ = (in == INT32_MIN) ? INT32_MAX : -in;
31. #endif
32.
33.     in = *pSrc++;
34. #if defined (ARM_MATH_DSP)
35.     *pDst++ = __QSUB(0, in);
36. #else
37.     *pDst++ = (in == INT32_MIN) ? INT32_MAX : -in;
38. #endif
39.
40.     in = *pSrc++;
41. #if defined (ARM_MATH_DSP)
42.     *pDst++ = __QSUB(0, in);
43. #else
44.     *pDst++ = (in == INT32_MIN) ? INT32_MAX : -in;
45. #endif
46.
47.     /* Decrement loop counter */
```



```
48.     blkCnt--;
49. }
50.
51. /* Loop unrolling: Compute remaining outputs */
52. blkCnt = blockSize % 0x4U;
53.
54. #else
55.
56. /* Initialize blkCnt with number of samples */
57. blkCnt = blockSize;
58.
59. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
60.
61. while (blkCnt > 0U)
62. {
63.     /* C = -A */
64.
65.     /* Negate and store result in destination buffer. */
66.     in = *pSrc++;
67. #if defined (ARM_MATH_DSP)
68.     *pDst++ = __QSUB(0, in);
69. #else
70.     *pDst++ = (in == INT32_MIN) ? INT32_MAX : -in;
71. #endif
72.
73.     /* Decrement loop counter */
74.     blkCnt--;
75. }
76.
77. }
```

函数描述：

用于求 32 位定点数的相反数。

函数解析：

- ◆ 第 9 到 54 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 61 到 75 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 对于 Q31 格式的数据，饱和运算会使得数据 0x80000000 变成 0xffffffff，因为最小负数 0x80000000（对应浮点数-1），求相反数后，是个正的 0x80000000（对应浮点数正 1），已经超过 Q31 所能表示的最大值 0x7fffffff，因此会被饱和处理为正数最大值 0x7fffffff。
- ◆ 这里重点说一下函数__QSUB，其实这个函数算是 Cortex-M7, M4/M3 的一个指令，用于实现饱和减法。比如函数：__QSUB(0, in1) 的作用就是实现 $0 - in1$ 并返回结果。这里__QSUB 实现的是 32 位数的饱和减法。还有__QSUB16 和__QSUB8 实现的是 16 位和 8 位数的减法。

函数参数：

- ◆ 第 1 个参数是原数据地址。
- ◆ 第 2 个参数是求相反数后目的数据地址。
- ◆ 第 3 个参数转换的数据个数，这里是指的定点数个数。



12.3.3 函数 arm_negate_q15

函数原型：

```
1. void arm_negate_q15(
2.     const q15_t * pSrc,
3.     q15_t * pDst,
4.     uint32_t blockSize)
5. {
6.     uint32_t blkCnt; /* Loop counter */
7.     q15_t in; /* Temporary input variable */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. #if defined (ARM_MATH_DSP)
12.     q31_t in1; /* Temporary input variables */
13. #endif
14.
15. /* Loop unrolling: Compute 4 outputs at a time */
16. blkCnt = blockSize >> 2U;
17.
18. while (blkCnt > 0U)
19. {
20.     /* C = -A */
21.
22. #if defined (ARM_MATH_DSP)
23.     /* Negate and store result in destination buffer (2 samples at a time). */
24.     in1 = read_q15x2_ia ((q15_t **) &pSrc);
25.     write_q15x2_ia (&pDst, __QSUB16(0, in1));
26.
27.     in1 = read_q15x2_ia ((q15_t **) &pSrc);
28.     write_q15x2_ia (&pDst, __QSUB16(0, in1));
29. #else
30.     in = *pSrc++;
31.     *pDst++ = (in == (q15_t) 0x8000) ? (q15_t) 0x7fff : -in;
32.
33.     in = *pSrc++;
34.     *pDst++ = (in == (q15_t) 0x8000) ? (q15_t) 0x7fff : -in;
35.
36.     in = *pSrc++;
37.     *pDst++ = (in == (q15_t) 0x8000) ? (q15_t) 0x7fff : -in;
38.
39.     in = *pSrc++;
40.     *pDst++ = (in == (q15_t) 0x8000) ? (q15_t) 0x7fff : -in;
41. #endif
42.
43.     /* Decrement loop counter */
44.     blkCnt--;
45. }
46.
47. /* Loop unrolling: Compute remaining outputs */
48. blkCnt = blockSize % 0x4U;
49.
50. #else
51.
52. /* Initialize blkCnt with number of samples */
53. blkCnt = blockSize;
```



```
54.  
55. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */  
56.  
57.     while (blkCnt > 0U)  
58.     {  
59.         /* C = -A */  
60.  
61.         /* Negate and store result in destination buffer. */  
62.         in = *pSrc++;  
63.         *pDst++ = (in == (q15_t) 0x8000) ? (q15_t) 0x7fff : -in;  
64.  
65.         /* Decrement loop counter */  
66.         blkCnt--;  
67.     }  
68.  
69. }
```

函数描述：

用于求 16 位定点数的绝对值。

函数解析：

- ◆ 第 9 到 50 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 57 到 67 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 对于 Q15 格式的数据，饱和运算会使得数据 0x8000 求相反数后饱和为 0x7fff。因为最小负数 0x8000 (对应浮点数-1)，求相反数后，是个正的 0x8000 (对应浮点数正 1)，已经超过 Q15 所能表示的最大值 0x7fff，因此会被饱和处理为正数最大值 0x7fff。
- ◆ _QSUB16 用于实现 16 位数据的饱和减法。

函数参数：

- ◆ 第 1 个参数是原数据地址。
- ◆ 第 2 个参数是求相反数后目的数据地址。
- ◆ 第 3 个参数转换的数据个数，这里是指的定点数个数。

12.3.4 函数 arm_negate_q7

函数原型：

```
1. void arm_negate_q7(  
2.     const q7_t * pSrc,  
3.     q7_t * pDst,  
4.     uint32_t blockSize)  
5. {  
6.     uint32_t blkCnt;           /* Loop counter */  
7.     q7_t in;                /* Temporary input variable */  
8.  
9. #if defined (ARM_MATH_LOOPUNROLL)  
10.  
11. #if defined (ARM_MATH_DSP)  
12.     q31_t in1;              /* Temporary input variable */  
13. #endif  
14.  
15.     /* Loop unrolling: Compute 4 outputs at a time */  
16.     blkCnt = blockSize >> 2U;
```



```
17.
18.     while (blkCnt > 0U)
19.     {
20.         /* C = -A */
21.
22. #if defined (ARM_MATH_DSP)
23.     /* Negate and store result in destination buffer (4 samples at a time). */
24.     in1 = read_q7x4_ia ((q7_t **) &pSrc);
25.     write_q7x4_ia (&pDst, __QSUB8(0, in1));
26. #else
27.     in = *pSrc++;
28.     *pDst++ = (in == (q7_t) 0x80) ? (q7_t) 0x7f : -in;
29.
30.     in = *pSrc++;
31.     *pDst++ = (in == (q7_t) 0x80) ? (q7_t) 0x7f : -in;
32.
33.     in = *pSrc++;
34.     *pDst++ = (in == (q7_t) 0x80) ? (q7_t) 0x7f : -in;
35.
36.     in = *pSrc++;
37.     *pDst++ = (in == (q7_t) 0x80) ? (q7_t) 0x7f : -in;
38. #endif
39.
40.     /* Decrement loop counter */
41.     blkCnt--;
42. }
43.
44. /* Loop unrolling: Compute remaining outputs */
45. blkCnt = blockSize % 0x4U;
46.
47. #else
48.
49.     /* Initialize blkCnt with number of samples */
50.     blkCnt = blockSize;
51.
52. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
53.
54.     while (blkCnt > 0U)
55.     {
56.         /* C = -A */
57.
58.         /* Negate and store result in destination buffer. */
59.         in = *pSrc++;
60.
61. #if defined (ARM_MATH_DSP)
62.         *pDst++ = (q7_t) __QSUB(0, in);
63. #else
64.         *pDst++ = (in == (q7_t) 0x80) ? (q7_t) 0x7f : -in;
65. #endif
66.
67.         /* Decrement loop counter */
68.         blkCnt--;
69.     }
70.
71. }
```

函数描述:



用于求 8 位定点数的相反数。

函数解析：

- ◆ 第 9 到 47 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 54 到 69 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 对于 Q7 格式的数据，饱和运算会使得数据 0x80 变成 0x7f。因为最小负数 0x80（对应浮点数-1），求相反数后，是个正的 0x80（对应浮点数正 1），已经超过 Q7 所能表示的最大值 0x7f，因此会被饱和处理为正数最大值 0x7f。
- ◆ __QSUB8 用于实现 8 位数据的饱和减法。

函数参数：

- ◆ 第 1 个参数是原数据地址。
- ◆ 第 2 个参数是求相反数后目的数据地址。
- ◆ 第 3 个参数转换的数据个数，这里是指的定点数个数。

12.3.5 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_Negate
* 功能说明: 求相反数
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Negate(void)
{
    float32_t pSrc = 0.0f;
    float32_t pDst;

    q31_t pSrc1 = 0;
    q31_t pDst1;

    q15_t pSrc2 = 0;
    q15_t pDst2;

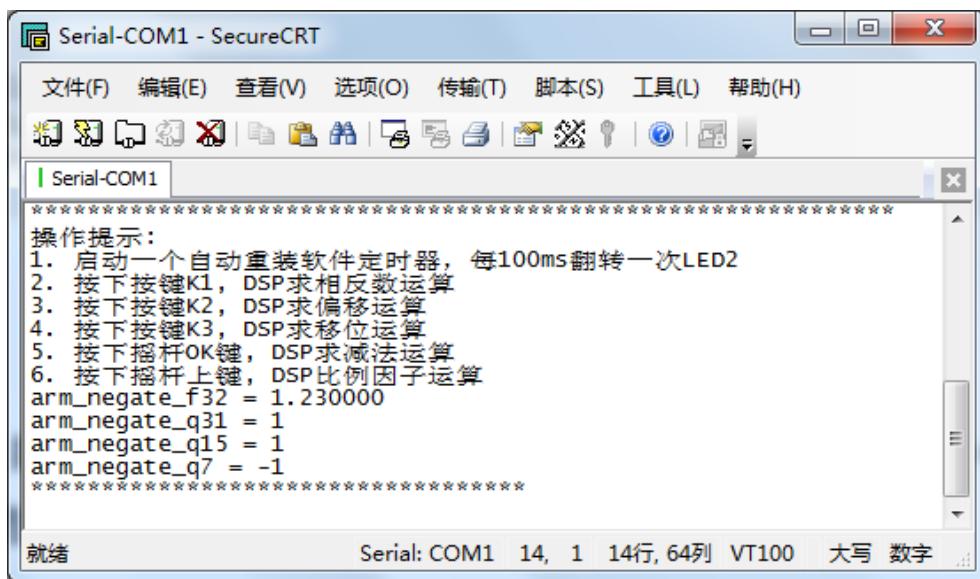
    q7_t pSrc3 = 0;
    q7_t pDst3;

    /*求相反数******/
    pSrc -= 1.23f;
    arm_negate_f32(&pSrc, &pDst, 1);
    printf("arm_negate_f32 = %f\r\n", pDst);

    pSrc1 -= 1;
    arm_negate_q31(&pSrc1, &pDst1, 1);
    printf("arm_negate_q31 = %d\r\n", pDst1);

    pSrc2 -= 1;
    arm_negate_q15(&pSrc2, &pDst2, 1);
    printf("arm_negate_q15 = %d\r\n", pDst2);

    pSrc3 += 1;
    arm_negate_q7(&pSrc3, &pDst3, 1);
    printf("arm_negate_q7 = %d\r\n", pDst3);
    printf("*****\r\n");
}
```

实验现象：

12.4 偏移 (Vector Offset)

这部分函数主要用于求偏移，公式描述如下：

$$pDst[n] = pSrc[n] + offset, \quad 0 \leq n < blockSize.$$

注意，这部分函数支持目标指针和源指针指向相同的缓冲区。

12.4.1 函数 arm_offset_f32

函数原型：

```
1. void arm_offset_f32(
2.     const float32_t * pSrc,
3.     float32_t offset,
4.     float32_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined(ARM_MATH_NEON_EXPERIMENTAL)
10.    float32x4_t vec1;
11.    float32x4_t res;
12.
13. /* Compute 4 outputs at a time */
14.    blkCnt = blockSize >> 2U;
15.
16.    while (blkCnt > 0U)
17.    {
18.        /* C = A + offset */
19.
20.        /* Add offset and then store the results in the destination buffer. */
21.        vec1 = vld1q_f32(pSrc);
22.        res = vaddq_f32(vec1, vdupq_n_f32(offset));
23.        vst1q_f32(pDst, res);
24.    }
```



```
25.     /* Increment pointers */
26.     pSrc += 4;
27.     pDst += 4;
28.
29.     /* Decrement the loop counter */
30.     blkCnt--;
31. }
32.
33. /* Tail */
34. blkCnt = blockSize & 0x3;
35.
36. #else
37. #if defined (ARM_MATH_LOOPUNROLL)
38.
39. /* Loop unrolling: Compute 4 outputs at a time */
40. blkCnt = blockSize >> 2U;
41.
42. while (blkCnt > 0U)
43. {
44.     /* C = A + offset */
45.
46.     /* Add offset and store result in destination buffer. */
47.     *pDst++ = (*pSrc++) + offset;
48.
49.     *pDst++ = (*pSrc++) + offset;
50.
51.     *pDst++ = (*pSrc++) + offset;
52.
53.     *pDst++ = (*pSrc++) + offset;
54.
55.     /* Decrement loop counter */
56.     blkCnt--;
57. }
58.
59. /* Loop unrolling: Compute remaining outputs */
60. blkCnt = blockSize % 0x4U;
61.
62. #else
63.
64.     /* Initialize blkCnt with number of samples */
65.     blkCnt = blockSize;
66.
67. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
68. #endif /* #if defined(ARM_MATH_NEON_EXPERIMENTAL) */
69.
70. while (blkCnt > 0U)
71. {
72.     /* C = A + offset */
73.
74.     /* Add offset and store result in destination buffer. */
75.     *pDst++ = (*pSrc++) + offset;
76.
77.     /* Decrement loop counter */
78.     blkCnt--;
79. }
80.
81. }
```



函数描述：

这个函数用于求 32 位浮点数的偏移。

函数解析：

- ◆ 第 9 到 36 行，用于 NEON 指令集，当前的 CM 内核不支持。
- ◆ 第 37 到 62 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 70 到 79 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是偏移量。
- ◆ 第 3 个参数是转换后的目的地址。
- ◆ 第 4 个参数是浮点数个数，其实就是执行偏移的次数。

12.4.2 函数 arm_offset_q31

函数原型：

```
1. void arm_offset_q31(
2.     const q31_t * pSrc,
3.     q31_t offset,
4.     q31_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11.    /* Loop unrolling: Compute 4 outputs at a time */
12.    blkCnt = blockSize >> 2U;
13.
14.    while (blkCnt > 0U)
15.    {
16.        /* C = A + offset */
17.
18.        /* Add offset and store result in destination buffer. */
19. #if defined (ARM_MATH_DSP)
20.        *pDst++ = __QADD(*pSrc++, offset);
21. #else
22.        *pDst++ = (q31_t) clip_q63_to_q31((q63_t) * pSrc++ + offset);
23. #endif
24.
25. #if defined (ARM_MATH_DSP)
26.        *pDst++ = __QADD(*pSrc++, offset);
27. #else
28.        *pDst++ = (q31_t) clip_q63_to_q31((q63_t) * pSrc++ + offset);
29. #endif
30.
31. #if defined (ARM_MATH_DSP)
32.        *pDst++ = __QADD(*pSrc++, offset);
33. #else
34.        *pDst++ = (q31_t) clip_q63_to_q31((q63_t) * pSrc++ + offset);
35. #endif
```



```
36.
37. #if defined (ARM_MATH_DSP)
38.     *pDst++ = __QADD(*pSrc++, offset);
39. #else
40.     *pDst++ = (q31_t) clip_q63_to_q31((q63_t) * pSrc++ + offset);
41. #endif
42.
43.     /* Decrement loop counter */
44.     blkCnt--;
45. }
46.
47. /* Loop unrolling: Compute remaining outputs */
48. blkCnt = blockSize % 0x4U;
49.
50. #else
51.
52.     /* Initialize blkCnt with number of samples */
53.     blkCnt = blockSize;
54.
55. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
56.
57. while (blkCnt > 0U)
58. {
59.     /* C = A + offset */
60.
61.     /* Add offset and store result in destination buffer. */
62. #if defined (ARM_MATH_DSP)
63.     *pDst++ = __QADD(*pSrc++, offset);
64. #else
65.     *pDst++ = (q31_t) clip_q63_to_q31((q63_t) * pSrc++ + offset);
66. #endif
67.
68.     /* Decrement loop counter */
69.     blkCnt--;
70. }
71.
72. }
```

函数描述：

这个函数用于求两个 32 位定点数的偏移。

函数解析：

- ◆ 第 9 到 50 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 57 到 70 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ __QADD 实现 32 位数的加法饱和运算。输出结果的范围[0x80000000 0x7FFFFFFF]，超出这个结果将产生饱和结果，负数饱和到 0x80000000，正数饱和到 0x7FFFFFFF。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是偏移量。
- ◆ 第 3 个参数是转换后的目的地址。
- ◆ 第 4 个参数是定点数个数，其实就是执行偏移的次数。



12.4.3 函数 arm_offset_q15

函数原型：

```
1. void arm_offset_q15(
2.     const q15_t * pSrc,
3.     q15_t offset,
4.     q15_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. #if defined (ARM_MATH_DSP)
12.     q31_t offset_packed; /* Offset packed to 32 bit */
13.
14. /* Offset is packed to 32 bit in order to use SIMD32 for addition */
15.     offset_packed = __PKHBT(offset, offset, 16);
16. #endif
17.
18. /* Loop unrolling: Compute 4 outputs at a time */
19.     blkCnt = blockSize >> 2U;
20.
21.     while (blkCnt > 0U)
22.     {
23.         /* C = A + offset */
24.
25. #if defined (ARM_MATH_DSP)
26.         /* Add offset and store result in destination buffer (2 samples at a time). */
27.         write_q15x2_ia (&pDst, __QADD16(read_q15x2_ia ((q15_t **) &pSrc), offset_packed));
28.         write_q15x2_ia (&pDst, __QADD16(read_q15x2_ia ((q15_t **) &pSrc), offset_packed));
29. #else
30.         *pDst++ = (q15_t) __SSAT(((q31_t) *pSrc++ + offset), 16);
31.         *pDst++ = (q15_t) __SSAT(((q31_t) *pSrc++ + offset), 16);
32.         *pDst++ = (q15_t) __SSAT(((q31_t) *pSrc++ + offset), 16);
33.         *pDst++ = (q15_t) __SSAT(((q31_t) *pSrc++ + offset), 16);
34. #endif
35.
36.         /* Decrement loop counter */
37.         blkCnt--;
38.     }
39.
40.     /* Loop unrolling: Compute remaining outputs */
41.     blkCnt = blockSize % 0x4U;
42.
43. #else
44.
45.     /* Initialize blkCnt with number of samples */
46.     blkCnt = blockSize;
47.
48. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
49.
50.     while (blkCnt > 0U)
51.     {
52.         /* C = A + offset */
53. }
```



```
54.     /* Add offset and store result in destination buffer. */
55. #if defined (ARM_MATH_DSP)
56.     *pDst++ = (q15_t) __QADD16(*pSrc++, offset);
57. #else
58.     *pDst++ = (q15_t) __SSAT(((q31_t) *pSrc++ + offset), 16);
59. #endif
60.
61.     /* Decrement loop counter */
62.     blkCnt--;
63. }
64.
65. }
```

函数描述：

这个函数用于求 16 位定点数的偏移。

函数解析：

- ◆ 第 9 到 43 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 50 到 63 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 函数 __PKHBT 也是 SIMD 指令，作用是将两个 16 位的数据合并成 32 位数据。用 C 实现的话，如下：

```
#define __PKHBT(ARG1, ARG2, ARG3) ( (((int32_t)(ARG1) << 0) & (int32_t)0x0000FFFF) | \
    (((int32_t)(ARG2) << ARG3) & (int32_t)0xFFFF0000) )
```

- ◆ 函数 read_q15x2_ia 的原型如下：

```
_STATIC_FORCEINLINE q31_t read_q15x2_ia (
    q15_t ** pQ15)
{
    q31_t val;

    memcpy (&val, *pQ15, 4);
    *pQ15 += 2;

    return (val);
}
```

作用是读取两次 16 位数据，返回一个 32 位数据，并将数据地址递增，方便下次读取。

- ◆ __QADD16 实现两次 16 位数的加法饱和运算。输出结果的范围[0x8000 0x7FFF]，超出这个结果将产生饱和结果，负数饱和到 0x8000，正数饱和到 0x7FFF。
- ◆ __SSAT 也是 SIMD 指令，这里是将结果饱和到 16 位精度。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是偏移量。
- ◆ 第 3 个参数是转换后的目的地址。
- ◆ 第 4 个参数是定点数个数，其实就是执行偏移的次数。

12.4.4 函数 arm_offset_q7

函数原型：

```
1. void arm_offset_q7(
```



```
2.     const q7_t * pSrc,
3.             q7_t offset,
4.             q7_t * pDst,
5.             uint32_t blockSize)
6. {
7.     uint32_t blkCnt;                                /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. #if defined (ARM_MATH_DSP)
12.     q31_t offset_packed;                          /* Offset packed to 32 bit */
13.
14.     /* Offset is packed to 32 bit in order to use SIMD32 for addition */
15.     offset_packed = __PACKq7(offset, offset, offset, offset);
16. #endif
17.
18.     /* Loop unrolling: Compute 4 outputs at a time */
19.     blkCnt = blockSize >> 2U;
20.
21.     while (blkCnt > 0U)
22.     {
23.         /* C = A + offset */
24.
25. #if defined (ARM_MATH_DSP)
26.         /* Add offset and store result in destination buffer (4 samples at a time). */
27.         write_q7x4_ia (&pDst, __QADD8(read_q7x4_ia ((q7_t **) &pSrc), offset_packed));
28. #else
29.         *pDst++ = (q7_t) __SSAT(*pSrc++ + offset, 8);
30.         *pDst++ = (q7_t) __SSAT(*pSrc++ + offset, 8);
31.         *pDst++ = (q7_t) __SSAT(*pSrc++ + offset, 8);
32.         *pDst++ = (q7_t) __SSAT(*pSrc++ + offset, 8);
33. #endif
34.
35.         /* Decrement loop counter */
36.         blkCnt--;
37.     }
38.
39.     /* Loop unrolling: Compute remaining outputs */
40.     blkCnt = blockSize % 0x4U;
41.
42. #else
43.
44.     /* Initialize blkCnt with number of samples */
45.     blkCnt = blockSize;
46.
47. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
48.
49.     while (blkCnt > 0U)
50.     {
51.         /* C = A + offset */
52.
53.         /* Add offset and store result in destination buffer. */
54.         *pDst++ = (q7_t) __SSAT((q15_t) *pSrc++ + offset, 8);
55.
56.         /* Decrement loop counter */
57.         blkCnt--;
58.     }
59.
```



60. }

函数描述：

这个函数用于求两个 8 位定点数的偏移。

函数解析：

- ◆ 第 9 到 42 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 49 到 58 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。
- ◆ 函数 write_q7x4_ia 的原型如下：

```
_STATIC_FORCEINLINE void write_q7x4_ia (
    q7_t ** pQ7,
    q31_t    value)
{
    q31_t val = value;

    memcpy (*pQ7, &val, 4);
    *pQ7 += 4;
}
```

作用是写 4 次 8 位数据，并将数据地址递增，方便下次继续写。

- ◆ _QADD8 实现四次 8 位数的加法饱和运算。输出结果的范围[0x80 0x7F]，超出这个结果将产生饱和结果，负数饱和到 0x80，正数饱和到 0x7F。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是偏移量。
- ◆ 第 3 个参数是转换后的目的地址。
- ◆ 第 4 个参数是定点数个数，其实就是执行偏移的次数。

12.4.5 使用举例

程序设计：

```
/*
*****
*  函数名: DSP_Offset
*  功能说明: 偏移
*  形参: 无
*  返回值: 无
*****
*/
static void DSP_Offset(void)
{
    float32_t    pSrcA = 0.0f;
    float32_t    Offset = 0.0f;
    float32_t    pDst;

    q31_t    pSrcA1 = 0;
    q31_t    Offset1 = 0;
    q31_t    pDst1;

    q15_t    pSrcA2 = 0;
    q15_t    Offset2 = 0;
    q15_t    pDst2;

    q7_t     pSrcA3 = 0;
    q7_t     Offset3 = 0;
```

```

q7_t pDst3;

/*求偏移*****
Offset--;
arm_offset_f32(&pSrcA, Offset, &pDst, 1);
printf("arm_offset_f32 = %f\r\n", pDst);

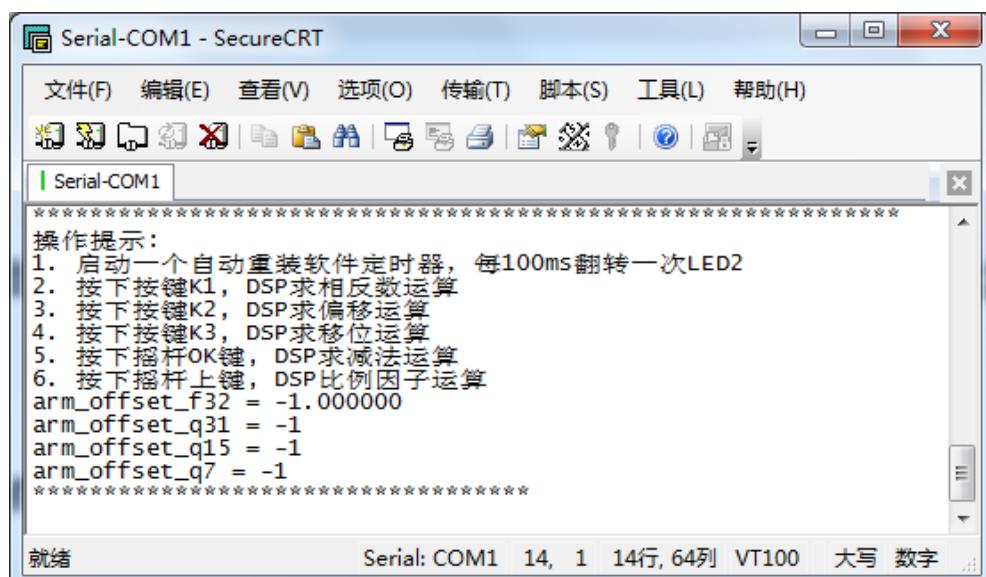
Offset1--;
arm_offset_q31(&pSrcA1, Offset1, &pDst1, 1);
printf("arm_offset_q31 = %d\r\n", pDst1);

Offset2--;
arm_offset_q15(&pSrcA2, Offset2, &pDst2, 1);
printf("arm_offset_q15 = %d\r\n", pDst2);

Offset3--;
arm_offset_q7(&pSrcA3, Offset3, &pDst3, 1);
printf("arm_offset_q7 = %d\r\n", pDst3);
printf("*****\r\n");
}

```

实验现象：



12.5 移位 (Vector Shift)

这部分函数主要用于实现移位，公式描述如下：

$pDst[n] = pSrc[n] \ll shift, 0 \leq n < blockSize.$

注意，这部分函数支持目标指针和源指针指向相同的缓冲区

12.5.1 函数 arm_shift_q31

函数原型：

```

1. void arm_shift_q31(
2.     const q31_t * pSrc,
3.     int8_t shiftBits,
4.     q31_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt;           /* Loop counter */
8.     uint8_t sign = (shiftBits & 0x80); /* Sign of shiftBits */

```



```
9.
10. #if defined (ARM_MATH_LOOPUNROLL)
11.
12.     q31_t in, out;                                /* Temporary variables */
13.
14.     /* Loop unrolling: Compute 4 outputs at a time */
15.     blkCnt = blockSize >> 2U;
16.
17.     /* If the shift value is positive then do right shift else left shift */
18.     if (sign == 0U)
19.     {
20.         while (blkCnt > 0U)
21.         {
22.             /* C = A << shiftBits */
23.
24.             /* Shift input and store result in destination buffer. */
25.             in = *pSrc++;
26.             out = in << shiftBits;
27.             if (in != (out >> shiftBits))
28.                 out = 0x7FFFFFFF ^ (in >> 31);
29.             *pDst++ = out;
30.
31.             in = *pSrc++;
32.             out = in << shiftBits;
33.             if (in != (out >> shiftBits))
34.                 out = 0x7FFFFFFF ^ (in >> 31);
35.             *pDst++ = out;
36.
37.             in = *pSrc++;
38.             out = in << shiftBits;
39.             if (in != (out >> shiftBits))
40.                 out = 0x7FFFFFFF ^ (in >> 31);
41.             *pDst++ = out;
42.
43.             in = *pSrc++;
44.             out = in << shiftBits;
45.             if (in != (out >> shiftBits))
46.                 out = 0x7FFFFFFF ^ (in >> 31);
47.             *pDst++ = out;
48.
49.             /* Decrement loop counter */
50.             blkCnt--;
51.         }
52.     }
53.     else
54.     {
55.         while (blkCnt > 0U)
56.         {
57.             /* C = A >> shiftBits */
58.
59.             /* Shift input and store results in destination buffer. */
60.             *pDst++ = (*pSrc++ >> -shiftBits);
61.             *pDst++ = (*pSrc++ >> -shiftBits);
62.             *pDst++ = (*pSrc++ >> -shiftBits);
63.             *pDst++ = (*pSrc++ >> -shiftBits);
64.
65.             /* Decrement loop counter */
66.             blkCnt--;

```



```
67.     }
68.   }
69.
70.   /* Loop unrolling: Compute remaining outputs */
71.   blkCnt = blockSize % 0x4U;
72.
73. #else
74.
75.   /* Initialize blkCnt with number of samples */
76.   blkCnt = blockSize;
77.
78. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
79.
80.   /* If the shift value is positive then do right shift else left shift */
81.   if (sign == 0U)
82.   {
83.     while (blkCnt > 0U)
84.     {
85.       /* C = A << shiftBits */
86.
87.       /* Shift input and store result in destination buffer. */
88.       *pDst++ = clip_q63_to_q31((q63_t) *pSrc++ << shiftBits);
89.
90.       /* Decrement loop counter */
91.       blkCnt--;
92.     }
93.   }
94. else
95. {
96.   while (blkCnt > 0U)
97.   {
98.     /* C = A >> shiftBits */
99.
100.    /* Shift input and store result in destination buffer. */
101.    *pDst++ = (*pSrc++ >> -shiftBits);
102.
103.    /* Decrement loop counter */
104.    blkCnt--;
105.  }
106. }
```

函数描述：

这个函数用于求 32 位定点数的左移或者右移。

函数解析：

- ◆ 第 10 到 73 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
 - 第 18 到 52 行，如果参数 shiftBits 是正数，执行左移。
 - 第 53 到 68 行，如果蚕食 shiftBits 是负数，执行右移。
 - 第 28 行，数值的左移仅支持将其左移后再右移相应的位数后数值不变的情况，如果不满足这个条件，那么要对输出结果做饱和运算，这里分两种情况：
$$\text{out} = 0x7FFFFFFF \wedge (\text{in} >> 31) \quad (\text{in 是正数})$$



```
= 0xFFFFFFFF ^ 0x00000000
= 0xFFFFFFFF
out = 0xFFFFFFFF ^ (in >> 31)    (in 是负数)
= 0xFFFFFFFF ^ 0xFFFFFFFF
= 0x80000000
```

- ◆ 第 81 到 106 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

- 第 88 行，函数 `clip_q63_to_q31` 的原型如下：

```
_STATIC_FORCEINLINE q31_t clip_q63_to_q31(
    q63_t x)
{
    return ((q31_t) (x >> 32) != ((q31_t) x >> 31)) ?
        ((0x7FFFFFFF ^ ((q31_t) (x >> 63)))) : (q31_t) x;
```

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是左移或者右移位数，正数是左移，负数是右移。
- ◆ 第 3 个参数是移位后数据地址。
- ◆ 第 4 个参数是定点数个数，其实就是执行左移或者右移的次数。

12.5.2 函数 `arm_shift_q15`

函数原型：

```
1. void arm_shift_q15(
2.     const q15_t * pSrc,
3.     int8_t shiftBits,
4.     q15_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt;                      /* Loop counter */
8.     uint8_t sign = (shiftBits & 0x80);      /* Sign of shiftBits */
9.
10. #if defined (ARM_MATH_LOOPUNROLL)
11.
12. #if defined (ARM_MATH_DSP)
13.     q15_t in1, in2;                      /* Temporary input variables */
14. #endif
15.
16.     /* Loop unrolling: Compute 4 outputs at a time */
17.     blkCnt = blockSize >> 2U;
18.
19.     /* If the shift value is positive then do right shift else left shift */
20.     if (sign == 0U)
21.     {
22.         while (blkCnt > 0U)
23.         {
24.             /* C = A << shiftBits */
25.
26. #if defined (ARM_MATH_DSP)
27.             /* read 2 samples from source */
28.             in1 = *pSrc++;
```



```
29.     in2 = *pSrc++;
30.
31.     /* Shift the inputs and then store the results in the destination buffer. */
32. #ifndef ARM_MATH_BIG_ENDIAN
33.     write_q15x2_ia (&pDst, __PKHBT(__SSAT((in1 << shiftBits), 16),
34.                                     __SSAT((in2 << shiftBits), 16), 16));
35. #else
36.     write_q15x2_ia (&pDst, __PKHBT(__SSAT((in2 << shiftBits), 16),
37.                                     __SSAT((in1 << shiftBits), 16), 16));
38. #endif /* #ifndef ARM_MATH_BIG_ENDIAN */
39.
40.     /* read 2 samples from source */
41.     in1 = *pSrc++;
42.     in2 = *pSrc++;
43.
44. #ifndef ARM_MATH_BIG_ENDIAN
45.     write_q15x2_ia (&pDst, __PKHBT(__SSAT((in1 << shiftBits), 16),
46.                                     __SSAT((in2 << shiftBits), 16), 16));
47. #else
48.     write_q15x2_ia (&pDst, __PKHBT(__SSAT((in2 << shiftBits), 16),
49.                                     __SSAT((in1 << shiftBits), 16), 16));
50. #endif /* #ifndef ARM_MATH_BIG_ENDIAN */
51.
52. #else
53.     *pDst++ = __SSAT(((q31_t) *pSrc++ << shiftBits), 16);
54.     *pDst++ = __SSAT(((q31_t) *pSrc++ << shiftBits), 16);
55.     *pDst++ = __SSAT(((q31_t) *pSrc++ << shiftBits), 16);
56.     *pDst++ = __SSAT(((q31_t) *pSrc++ << shiftBits), 16);
57. #endif
58.
59.     /* Decrement loop counter */
60.     blkCnt--;
61. }
62. }
63. else
64. {
65.     while (blkCnt > 0U)
66.     {
67.         /* C = A >> shiftBits */
68.
69. #if defined (ARM_MATH_DSP)
70.         /* read 2 samples from source */
71.         in1 = *pSrc++;
72.         in2 = *pSrc++;
73.
74.         /* Shift the inputs and then store the results in the destination buffer. */
75. #ifndef ARM_MATH_BIG_ENDIAN
76.         write_q15x2_ia (&pDst, __PKHBT((in1 >> -shiftBits),
77.                                         (in2 >> -shiftBits), 16));
78. #else
79.         write_q15x2_ia (&pDst, __PKHBT((in2 >> -shiftBits),
80.                                         (in1 >> -shiftBits), 16));
81. #endif /* #ifndef ARM_MATH_BIG_ENDIAN */
82.
83.         /* read 2 samples from source */
84.         in1 = *pSrc++;
85.         in2 = *pSrc++;
86.
```



```
87. #ifndef ARM_MATH_BIG_ENDIAN
88.     write_q15x2_ia (&pDst, __PKHBT((in1 >> -shiftBits),
89.                         (in2 >> -shiftBits), 16));
90. #else
91.     write_q15x2_ia (&pDst, __PKHBT((in2 >> -shiftBits),
92.                         (in1 >> -shiftBits), 16));
93. #endif /* #ifndef ARM_MATH_BIG_ENDIAN */
94.
95. #else
96.     *pDst++ = (*pSrc++ >> -shiftBits);
97.     *pDst++ = (*pSrc++ >> -shiftBits);
98.     *pDst++ = (*pSrc++ >> -shiftBits);
99.     *pDst++ = (*pSrc++ >> -shiftBits);
100. #endif
101.
102.    /* Decrement loop counter */
103.    blkCnt--;
104. }
105. }
106.
107. /* Loop unrolling: Compute remaining outputs */
108. blkCnt = blockSize % 0x4U;
109.
110. #else
111.
112.    /* Initialize blkCnt with number of samples */
113.    blkCnt = blockSize;
114.
115. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
116.
117.    /* If the shift value is positive then do right shift else left shift */
118.    if (sign == 0U)
119.    {
120.        while (blkCnt > 0U)
121.        {
122.            /* C = A << shiftBits */
123.
124.            /* Shift input and store result in destination buffer. */
125.            *pDst++ = __SSAT(((q31_t) *pSrc++ << shiftBits), 16);
126.
127.            /* Decrement loop counter */
128.            blkCnt--;
129.        }
130.    }
131.    else
132.    {
133.        while (blkCnt > 0U)
134.        {
135.            /* C = A >> shiftBits */
136.
137.            /* Shift input and store result in destination buffer. */
138.            *pDst++ = (*pSrc++ >> -shiftBits);
139.
140.            /* Decrement loop counter */
141.            blkCnt--;
142.        }
143.    }
144.
```



145. }

函数描述：

这个函数用于求 16 位定点数的左移或者右移。

函数解析：

- ◆ 第 10 到 115 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
 - 第 20 到 62 行，如果参数 shiftBits 是正数，执行左移。
 - 第 63 到 105 行，如果蚕食 shiftBits 是负数，执行右移。
 - 第 79 行，函数 write_q15x2_ia 的原型如下，用于实现将两个 Q15 组成合并成一个 Q31。

```
_STATIC_FORCEINLINE void write_q15x2_ia (
    q15_t ** pQ15,
    q31_t     value)
{
    q31_t val = value;

    memcpy (*pQ15, &val, 4);
    *pQ15 += 2;
}
```

函数 _PKHBT 也是 SIMD 指令，作用是将两个 16 位的数据合并成 32 位数据。用 C 实现的话，如下：

```
#define __PKHBT(ARG1, ARG2, ARG3) (((int32_t)(ARG1) << 0) & (int32_t)0x0000FFFF) | \
    (((int32_t)(ARG2) << ARG3) & (int32_t)0xFFFF0000)
```

- ◆ 第 118 到 143 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是左移或者右移位数，正数是左移，负数是右移。
- ◆ 第 3 个参数是移位后数据地址。
- ◆ 第 4 个参数是定点数个数，其实就是执行左移或者右移的次数。

12.5.3 函数 arm_shift_q7

函数原型：

```
1. void arm_shift_q7(
2.     const q7_t * pSrc,
3.     int8_t shiftBits,
4.     q7_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt;                      /* Loop counter */
8.     uint8_t sign = (shiftBits & 0x80);      /* Sign of shiftBits */
9.
10. #if defined (ARM_MATH_LOOPUNROLL)
11.
12. #if defined (ARM_MATH_DSP)
13.     q7_t in1,  in2,  in3,  in4;          /* Temporary input variables */
14. #endif
15.
16.     /* Loop unrolling: Compute 4 outputs at a time */
17.     blkCnt = blockSize >> 2U;
18.
19.     /* If the shift value is positive then do right shift else left shift */
```



```
20.    if (sign == 0U)
21.    {
22.        while (blkCnt > 0U)
23.        {
24.            /* C = A << shiftBits */
25.
26. #if defined (ARM_MATH_DSP)
27.     /* Read 4 inputs */
28.     in1 = *pSrc++;
29.     in2 = *pSrc++;
30.     in3 = *pSrc++;
31.     in4 = *pSrc++;
32.
33.     /* Pack and store result in destination buffer (in single write) */
34.     write_q7x4_ia (&pDst, __PACKq7(__SSAT((in1 << shiftBits), 8),
35.                                     __SSAT((in2 << shiftBits), 8),
36.                                     __SSAT((in3 << shiftBits), 8),
37.                                     __SSAT((in4 << shiftBits), 8) ));
38. #else
39.     *pDst++ = (q7_t) __SSAT(((q15_t) *pSrc++ << shiftBits), 8);
40.     *pDst++ = (q7_t) __SSAT(((q15_t) *pSrc++ << shiftBits), 8);
41.     *pDst++ = (q7_t) __SSAT(((q15_t) *pSrc++ << shiftBits), 8);
42.     *pDst++ = (q7_t) __SSAT(((q15_t) *pSrc++ << shiftBits), 8);
43. #endif
44.
45.     /* Decrement loop counter */
46.     blkCnt--;
47. }
48. }
49. else
50. {
51.     while (blkCnt > 0U)
52.     {
53.         /* C = A >> shiftBits */
54.
55. #if defined (ARM_MATH_DSP)
56.     /* Read 4 inputs */
57.     in1 = *pSrc++;
58.     in2 = *pSrc++;
59.     in3 = *pSrc++;
60.     in4 = *pSrc++;
61.
62.     /* Pack and store result in destination buffer (in single write) */
63.     write_q7x4_ia (&pDst, __PACKq7((in1 >> -shiftBits),
64.                                     (in2 >> -shiftBits),
65.                                     (in3 >> -shiftBits),
66.                                     (in4 >> -shiftBits) ));
67. #else
68.     *pDst++ = (*pSrc++ >> -shiftBits);
69.     *pDst++ = (*pSrc++ >> -shiftBits);
70.     *pDst++ = (*pSrc++ >> -shiftBits);
71.     *pDst++ = (*pSrc++ >> -shiftBits);
72. #endif
73.
74.     /* Decrement loop counter */
75.     blkCnt--;
76. }
77. }
```



```
78.  
79.     /* Loop unrolling: Compute remaining outputs */  
80.     blkCnt = blockSize % 0x4U;  
81.  
82. #else  
83.  
84.     /* Initialize blkCnt with number of samples */  
85.     blkCnt = blockSize;  
86.  
87. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */  
88.  
89.     /* If the shift value is positive then do right shift else left shift */  
90.     if (sign == 0U)  
91.     {  
92.         while (blkCnt > 0U)  
93.         {  
94.             /* C = A << shiftBits */  
95.  
96.             /* Shift input and store result in destination buffer. */  
97.             *pDst++ = (q7_t) __SSAT((q15_t) *pSrc++ << shiftBits), 8);  
98.  
99.             /* Decrement loop counter */  
100.            blkCnt--;  
101.        }  
102.    }  
103. else  
104. {  
105.     while (blkCnt > 0U)  
106.     {  
107.         /* C = A >> shiftBits */  
108.  
109.         /* Shift input and store result in destination buffer. */  
110.         *pDst++ = (*pSrc++ >> -shiftBits);  
111.  
112.         /* Decrement loop counter */  
113.         blkCnt--;  
114.     }  
115. }
```

函数描述：

这个函数用于求 8 位定点数的左移或者右移。

函数解析：

- ◆ 第 10 到 87 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
 - 第 20 到 48 行，如果参数 shiftBits 是正数，执行左移。
 - 第 49 到 77 行，如果蚕食 shiftBits 是负数，执行右移。
 - 第 79 行，函数 write_q7x4_ia 的原型如下，作用是写入 4 次 8 位数据，并将数据地址递增，方便下次写入。

```
_STATIC_FORCEINLINE void write_q7x4_ia (  
    q7_t ** pQ7,  
    q31_t    value)  
{
```



```
q31_t val = value;  
  
memcpy (*pQ7, &val, 4);  
*pQ7 += 4;  
}
```

函数 __PACKq7 作用是将 4 个 8 位的数据合并成 32 位数据，实现代码如下：

```
#define __PACKq7(v0,v1,v2,v3) (((int32_t)(v0) << 0) & (int32_t)0x000000FF) | \  
((int32_t)(v1) << 8) & (int32_t)0x0000FF00) | \  
((int32_t)(v2) << 16) & (int32_t)0x00FF0000) | \  
((int32_t)(v3) << 24) & (int32_t)0xFF000000)
```

- ◆ 第 90 到 115 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

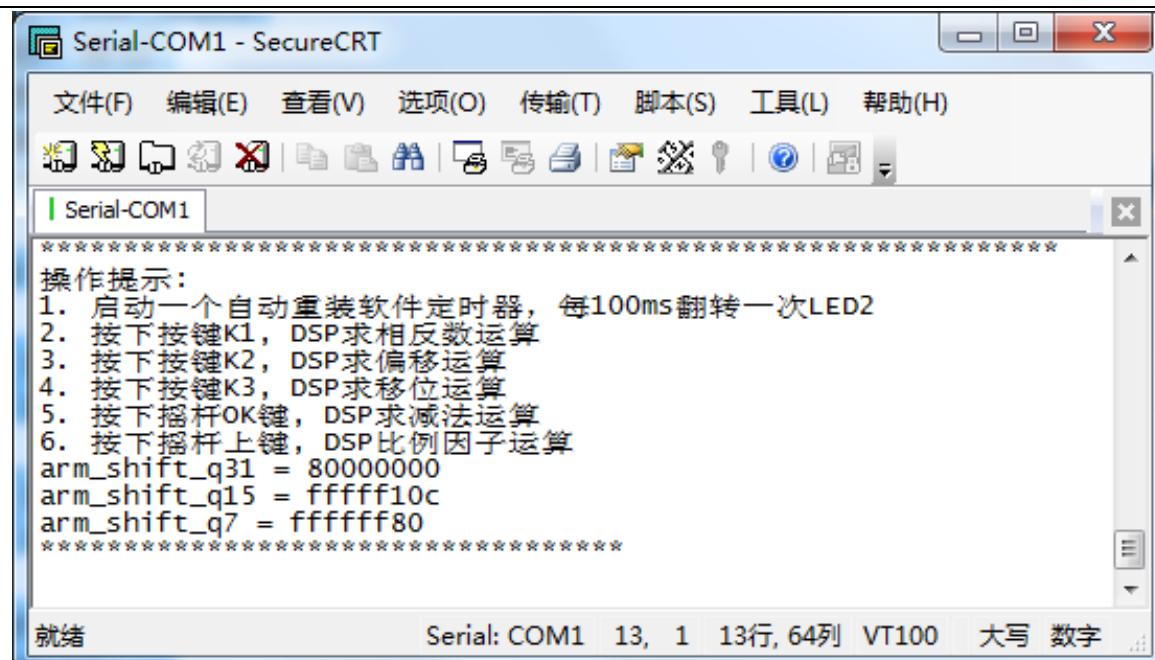
- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是左移或者右移位数，正数是左移，负数是右移。
- ◆ 第 3 个参数是移位后数据地址。
- ◆ 第 4 个参数是定点数个数，其实就是执行左移或者右移的次数

12.5.4 使用举例

程序设计：

```
/*  
*****  
* 函数名: DSP_Shift  
* 功能说明: 移位  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_Shift(void)  
{  
    q31_t pSrcA1 = 0x88886666;  
    q31_t pDst1;  
  
    q15_t pSrcA2 = 0x8866;  
    q15_t pDst2;  
  
    q7_t pSrcA3 = 0x86;  
    q7_t pDst3;  
  
    /*求移位*****  
    arm_shift_q31(&pSrcA1, 3, &pDst1, 1);  
    printf("arm_shift_q31 = %8x\r\n", pDst1);  
  
    arm_shift_q15(&pSrcA2, -3, &pDst2, 1);  
    printf("arm_shift_q15 = %4x\r\n", pDst2);  
  
    arm_shift_q7(&pSrcA3, 3, &pDst3, 1);  
    printf("arm_shift_q7 = %2x\r\n", pDst3);  
    printf("*****\r\n");  
}
```

实验现象：



这里特别注意 Q31 和 Q7 的计算结果，表示负数已经饱和到了最小值。另外注意，对于负数来说，右移时，右侧补 1，左移时，左侧补 0。

12.6 减法 (Vector Sub)

这部分函数主要用于实现减法，公式描述如下：

$$pDst[n] = pSrcA[n] - pSrcB[n], \quad 0 \leq n < blockSize.$$

12.6.1 函数 arm_sub_f32

函数原型：

```

1. void arm_sub_f32(
2.     const float32_t * pSrcA,
3.     const float32_t * pSrcB,
4.     float32_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined(ARM_MATH_NEON)
10.    float32x4_t vec1;
11.    float32x4_t vec2;
12.    float32x4_t res;
13.
14.    /* Compute 4 outputs at a time */
15.    blkCnt = blockSize >> 2U;
16.
17.    while (blkCnt > 0U)
18.    {
19.        /* C = A - B */
20.
21.        /* Subtract and then store the results in the destination buffer. */
22.        vec1 = vld1q_f32(pSrcA);

```



```
23.     vec2 = vld1q_f32(pSrcB);
24.     res = vsubq_f32(vec1, vec2);
25.     vst1q_f32(pDst, res);
26.
27.     /* Increment pointers */
28.     pSrcA += 4;
29.     pSrcB += 4;
30.     pDst += 4;
31.
32.     /* Decrement the loop counter */
33.     blkCnt--;
34. }
35.
36. /* Tail */
37. blkCnt = blockSize & 0x3;
38.
39. #else
40. #if defined(ARM_MATH_LOOPUNROLL)
41.
42.     /* Loop unrolling: Compute 4 outputs at a time */
43.     blkCnt = blockSize >> 2U;
44.
45.     while (blkCnt > 0U)
46.     {
47.         /* C = A - B */
48.
49.         /* Subtract and store result in destination buffer. */
50.         *pDst++ = (*pSrcA++) - (*pSrcB++);
51.
52.         *pDst++ = (*pSrcA++) - (*pSrcB++);
53.
54.         *pDst++ = (*pSrcA++) - (*pSrcB++);
55.
56.         *pDst++ = (*pSrcA++) - (*pSrcB++);
57.
58.         /* Decrement loop counter */
59.         blkCnt--;
60.     }
61.
62.     /* Loop unrolling: Compute remaining outputs */
63.     blkCnt = blockSize % 0x4U;
64.
65. #else
66.
67.     /* Initialize blkCnt with number of samples */
68.     blkCnt = blockSize;
69.
70. #endif /* #if defined(ARM_MATH_LOOPUNROLL) */
71. #endif /* #if defined(ARM_MATH_NEON) */
72.
73.     while (blkCnt > 0U)
74.     {
75.         /* C = A - B */
76.
77.         /* Subtract and store result in destination buffer. */
78.         *pDst++ = (*pSrcA++) - (*pSrcB++);
79.
80.         /* Decrement loop counter */
```



```
81.     blkCnt--;
82. }
83.
84. }
```

函数描述：

这个函数用于求 32 位浮点数的减法。

函数解析：

- ◆ 第 9 到 39 行，用于 NEON 指令集，当前的 CM 内核不支持。
- ◆ 第 40 到 65 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 73 到 82 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是减数地址。
- ◆ 第 2 个参数是被减数地址。
- ◆ 第 3 个参数是结果地址。
- ◆ 第 4 个参数是数据块大小，其实就是执行减法的次数。

12.6.2 函数 arm_sub_q31

函数原型：

```
1. void arm_sub_q31(
2.     const q31_t * pSrcA,
3.     const q31_t * pSrcB,
4.     q31_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. /* Loop unrolling: Compute 4 outputs at a time */
12. blkCnt = blockSize >> 2U;
13.
14. while (blkCnt > 0U)
15. {
16.     /* C = A - B */
17.
18.     /* Subtract and store result in destination buffer. */
19.     *pDst++ = __QSUB(*pSrcA++, *pSrcB++);
20.
21.     *pDst++ = __QSUB(*pSrcA++, *pSrcB++);
22.
23.     *pDst++ = __QSUB(*pSrcA++, *pSrcB++);
24.
25.     *pDst++ = __QSUB(*pSrcA++, *pSrcB++);
26.
27.     /* Decrement loop counter */
28.     blkCnt--;
29. }
30.
31. /* Loop unrolling: Compute remaining outputs */
```



```
32.     blkCnt = blockSize % 0x4U;
33.
34. #else
35.
36.     /* Initialize blkCnt with number of samples */
37.     blkCnt = blockSize;
38.
39. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
40.
41.     while (blkCnt > 0U)
42.     {
43.         /* C = A - B */
44.
45.         /* Subtract and store result in destination buffer. */
46.         *pDst++ = __QSUB(*pSrcA++, *pSrcB++);
47.
48.         /* Decrement loop counter */
49.         blkCnt--;
50.     }
51.
52. }
```

函数描述：

这个函数用于求 32 位定点数的减法。

函数解析：

- ◆ 这个函数使用了饱和减法__QSUB，所得结果是 Q31 格式，范围[0x80000000 0x7FFFFFFF]。
- ◆ 第 9 到 34 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 41 到 50 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是减数地址。
- ◆ 第 2 个参数是被减数地址。
- ◆ 第 3 个参数是结果地址。
- ◆ 第 4 个参数是数据块大小，其实就是执行减法的次数。

12.6.3 函数 arm_sub_q15

函数原型：

```
1. void arm_sub_q15(
2.     const q15_t * pSrcA,
3.     const q15_t * pSrcB,
4.     q15_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt;           /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. #if defined (ARM_MATH_DSP)
12.     q31_t inA1, inA2;
13.     q31_t inB1, inB2;
14. #endif
```



```
15.
16. /* Loop unrolling: Compute 4 outputs at a time */
17. blkCnt = blockSize >> 2U;
18.
19. while (blkCnt > 0U)
20. {
21.     /* C = A - B */
22.
23. #if defined (ARM_MATH_DSP)
24.     /* read 2 times 2 samples at a time from sourceA */
25.     inA1 = read_q15x2_ia ((q15_t **) &pSrcA);
26.     inA2 = read_q15x2_ia ((q15_t **) &pSrcA);
27.     /* read 2 times 2 samples at a time from sourceB */
28.     inB1 = read_q15x2_ia ((q15_t **) &pSrcB);
29.     inB2 = read_q15x2_ia ((q15_t **) &pSrcB);
30.
31.     /* Subtract and store 2 times 2 samples at a time */
32.     write_q15x2_ia (&pDst, __QSUB16(inA1, inB1));
33.     write_q15x2_ia (&pDst, __QSUB16(inA2, inB2));
34. #else
35.     *pDst++ = (q15_t) __SSAT(((q31_t) *pSrcA++ - *pSrcB++), 16);
36.     *pDst++ = (q15_t) __SSAT(((q31_t) *pSrcA++ - *pSrcB++), 16);
37.     *pDst++ = (q15_t) __SSAT(((q31_t) *pSrcA++ - *pSrcB++), 16);
38.     *pDst++ = (q15_t) __SSAT(((q31_t) *pSrcA++ - *pSrcB++), 16);
39. #endif
40.
41.     /* Decrement loop counter */
42.     blkCnt--;
43. }
44.
45. /* Loop unrolling: Compute remaining outputs */
46. blkCnt = blockSize % 0x4U;
47.
48. #else
49.
50.     /* Initialize blkCnt with number of samples */
51.     blkCnt = blockSize;
52.
53. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
54.
55. while (blkCnt > 0U)
56. {
57.     /* C = A - B */
58.
59.     /* Subtract and store result in destination buffer. */
60. #if defined (ARM_MATH_DSP)
61.     *pDst++ = (q15_t) __QSUB16(*pSrcA++, *pSrcB++);
62. #else
63.     *pDst++ = (q15_t) __SSAT(((q31_t) *pSrcA++ - *pSrcB++), 16);
64. #endif
65.
66.     /* Decrement loop counter */
67.     blkCnt--;
68. }
69.
70. }
```

函数描述:



这个函数用于求 16 位定点数的减法。

函数解析：

- ◆ 第 9 到 48 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
 - 第 25 行，函数 read_q15x2_ia 一次读取两个 Q15 格式的数据，组成一个 Q31 格式。
 - 第 32 行，函数 write_q15x2_ia 一次写入两个 Q15 格式的数据，获得一个 Q31 格式数据。
 - 第 32 行，函数 _QSUB16 实现两次 16bit 的饱和减法。
- ◆ 第 55 到 68 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是减数地址。
- ◆ 第 2 个参数是被减数地址。
- ◆ 第 3 个参数是结果地址。
- ◆ 第 4 个参数是数据块大小，其实就是执行减法的次数。

12.6.4 函数 arm_sub_q7

函数原型：

```
1. void arm_sub_q7(
2.     const q7_t * pSrcA,
3.     const q7_t * pSrcB,
4.     q7_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8.
9. #if defined (ARM_MATH_LOOPUNROLL)
10.
11. /* Loop unrolling: Compute 4 outputs at a time */
12. blkCnt = blockSize >> 2U;
13.
14. while (blkCnt > 0U)
15. {
16.     /* C = A - B */
17.
18. #if defined (ARM_MATH_DSP)
19.     /* Subtract and store result in destination buffer (4 samples at a time). */
20.     write_q7x4_ia (&pDst, __QSUB8(read_q7x4_ia ((q7_t **) &pSrcA), read_q7x4_ia ((q7_t **) &pSrcB)));
21. #else
22.     *pDst++ = (q7_t) __SSAT((q15_t) *pSrcA++ - *pSrcB++, 8);
23.     *pDst++ = (q7_t) __SSAT((q15_t) *pSrcA++ - *pSrcB++, 8);
24.     *pDst++ = (q7_t) __SSAT((q15_t) *pSrcA++ - *pSrcB++, 8);
25.     *pDst++ = (q7_t) __SSAT((q15_t) *pSrcA++ - *pSrcB++, 8);
26. #endif
27.
28.     /* Decrement loop counter */
29.     blkCnt--;
30. }
31.
32. /* Loop unrolling: Compute remaining outputs */
33. blkCnt = blockSize % 0x4U;
```



```
34.  
35. #else  
36.  
37. /* Initialize blkCnt with number of samples */  
38. blkCnt = blockSize;  
39.  
40. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */  
41.  
42. while (blkCnt > 0U)  
43. {  
44.     /* C = A - B */  
45.  
46.     /* Subtract and store result in destination buffer. */  
47.     *pDst++ = (q7_t) __SSAT((q15_t) *pSrcA++ - *pSrcB++, 8);  
48.  
49.     /* Decrement loop counter */  
50.     blkCnt--;  
51. }  
52.  
53. }
```

函数描述：

这个函数用于求 8 位定点数的乘法。

函数解析：

- ◆ 第 9 到 35 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
 - 第 20 行，函数 write_q7x4_ia 实现一次写入 4 个 Q7 格式数据到 Q31 各种中。函数_QSUB8 实现一次计算 4 个 Q7 格式减法。
- ◆ 第 42 到 51 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是减数地址。
- ◆ 第 2 个参数是被减数地址。
- ◆ 第 3 个参数是结果地址。
- ◆ 第 4 个参数是数据块大小，其实就是执行减法的次数。

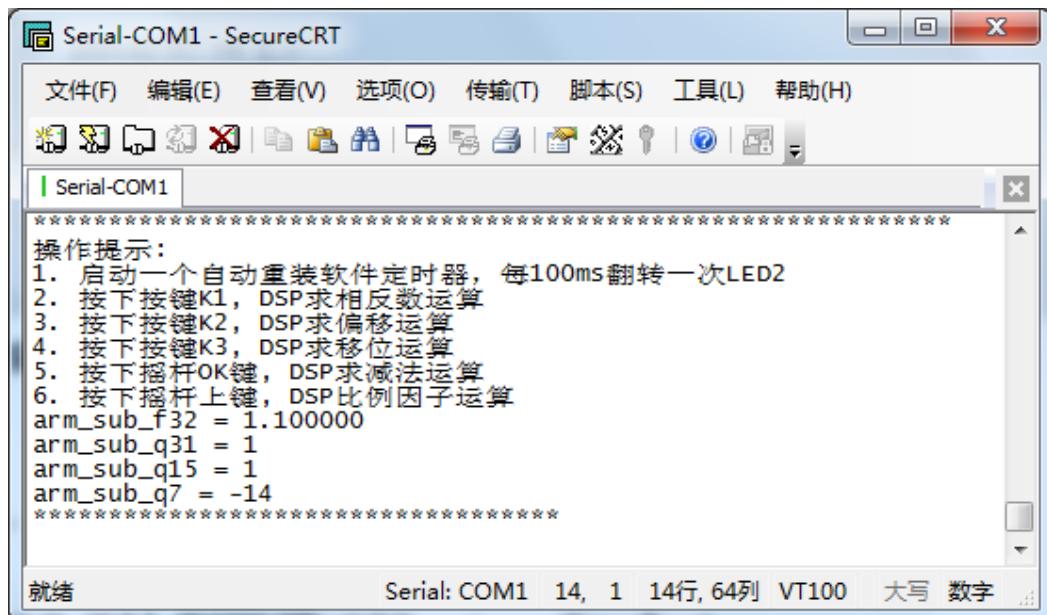
12.6.5 使用举例

程序设计：

```
/*  
*****  
* 函数名: DSP_Sub  
* 功能说明: 减法  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_Sub(void)  
{  
    float32_t pSrcA[5] = {1.0f, 1.0f, 1.0f, 1.0f, 1.0f};  
    float32_t pSrcB[5] = {1.0f, 1.0f, 1.0f, 1.0f, 1.0f};  
    float32_t pDst[5];  
  
    q31_t pSrcA1[5] = {1, 1, 1, 1, 1};  
    q31_t pSrcB1[5] = {1, 1, 1, 1, 1};
```

```
q31_t pDst1[5];  
  
q15_t pSrcA2[5] = {1, 1, 1, 1, 1};  
q15_t pSrcB2[5] = {1, 1, 1, 1, 1};  
q15_t pDst2[5];  
  
q7_t pSrcA3[5] = {0x70, 1, 1, 1, 1};  
q7_t pSrcB3[5] = {0x7f, 1, 1, 1, 1};  
q7_t pDst3[5];  
  
/*求减法*****  
pSrcA[0] += 1.1f;  
arm_sub_f32(pSrcA, pSrcB, pDst, 5);  
printf("arm_sub_f32 = %f\r\n", pDst[0]);  
  
pSrcA1[0] += 1;  
arm_sub_q31(pSrcA1, pSrcB1, pDst1, 5);  
printf("arm_sub_q31 = %d\r\n", pDst1[0]);  
  
pSrcA2[0] += 1;  
arm_sub_q15(pSrcA2, pSrcB2, pDst2, 5);  
printf("arm_sub_q15 = %d\r\n", pDst2[0]);  
  
pSrcA3[0] += 1;  
arm_sub_q7(pSrcA3, pSrcB3, pDst3, 5);  
printf("arm_sub_q7 = %d\r\n", pDst3[0]);  
printf("*****\r\n");  
}  
}
```

实验现象：



12.7 比例因子 (Vector Scale)

这部分函数主要用于实现数据的比例放大和缩小，浮点数据公式描述如下：

$$pDst[n] = pSrc[n] * scale, \quad 0 \leq n < blockSize.$$

如果是 Q31, Q15, Q7 格式的数据，公式描述如下：

$$pDst[n] = (pSrc[n] * scaleFract) << shift, \quad 0 \leq n < blockSize.$$

这种情况下，比例因子就是：



scale = scaleFract * 2^shift.

注意，这部分函数支持目标指针和源指针指向相同的缓冲区

12.7.1 函数 arm_scale_f32

函数原型：

```
1. void arm_scale_f32(
2.     const float32_t *pSrc,
3.     float32_t scale,
4.     float32_t *pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt; /* Loop counter */
8. #if defined(ARM_MATH_NEON_EXPERIMENTAL)
9.     float32x4_t vec1;
10.    float32x4_t res;
11.
12.    /* Compute 4 outputs at a time */
13.    blkCnt = blockSize >> 2U;
14.
15.    while (blkCnt > 0U)
16.    {
17.        /* C = A * scale */
18.
19.        /* Scale the input and then store the results in the destination buffer. */
20.        vec1 = vld1q_f32(pSrc);
21.        res = vmulq_f32(vec1, vdupq_n_f32(scale));
22.        vst1q_f32(pDst, res);
23.
24.        /* Increment pointers */
25.        pSrc += 4;
26.        pDst += 4;
27.
28.        /* Decrement the loop counter */
29.        blkCnt--;
30.    }
31.
32.    /* Tail */
33.    blkCnt = blockSize & 0x3;
34.
35. #else
36. #if defined (ARM_MATH_LOOPUNROLL)
37.
38.    /* Loop unrolling: Compute 4 outputs at a time */
39.    blkCnt = blockSize >> 2U;
40.
41.    while (blkCnt > 0U)
42.    {
43.        /* C = A * scale */
44.
45.        /* Scale input and store result in destination buffer. */
46.        *pDst++ = (*pSrc++) * scale;
47.
48.        *pDst++ = (*pSrc++) * scale;
49.
50.        *pDst++ = (*pSrc++) * scale;
```



```
51.     *pDst++ = (*pSrc++) * scale;
52.
53.
54.     /* Decrement loop counter */
55.     blkCnt--;
56. }
57.
58. /* Loop unrolling: Compute remaining outputs */
59. blkCnt = blockSize % 0x4U;
60.
61. #else
62.
63.     /* Initialize blkCnt with number of samples */
64.     blkCnt = blockSize;
65.
66. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
67. #endif /* #if defined(ARM_MATH_NEON_EXPERIMENTAL) */
68.
69.     while (blkCnt > 0U)
70.     {
71.         /* C = A * scale */
72.
73.         /* Scale input and store result in destination buffer. */
74.         *pDst++ = (*pSrc++) * scale;
75.
76.         /* Decrement loop counter */
77.         blkCnt--;
78.     }
79. }
```

函数描述：

这个函数用于求 32 位浮点数的比例因子计算。

函数解析：

- ◆ 第 8 到 35 行，用于 NEON 指令集，当前的 CM 内核不支持。
- ◆ 第 36 到 61 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
- ◆ 第 69 到 78 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是数据源地址。
- ◆ 第 2 个参数是比例因子
- ◆ 第 3 个参数是结果地址。
- ◆ 第 4 个参数是数据块大小，其实就是执行比例因子计算的次数。

12.7.2 函数 arm_scale_q31

函数原型：

```
1. void arm_scale_q31(
2.     const q31_t *pSrc,
3.             q31_t scaleFract,
4.             int8_t shift,
5.             q31_t *pDst,
```



```
6.         uint32_t blockSize)
7.     {
8.         uint32_t blkCnt;                      /* Loop counter */
9.         q31_t in, out;                      /* Temporary variables */
10.        int8_t kShift = shift + 1;           /* Shift to apply after scaling */
11.        int8_t sign = (kShift & 0x80);
12.
13. #if defined (ARM_MATH_LOOPUNROLL)
14.
15. /* Loop unrolling: Compute 4 outputs at a time */
16. blkCnt = blockSize >> 2U;
17.
18. if (sign == 0U)
19. {
20.     while (blkCnt > 0U)
21.     {
22.         /* C = A * scale */
23.
24.         /* Scale input and store result in destination buffer. */
25.         in = *pSrc++;                      /* read input from source */
26.         in = ((q63_t) in * scaleFract) >> 32;    /* multiply input with scalar value */
27.         out = in << kShift;                 /* apply shifting */
28.         if (in != (out >> kShift))          /* saturate the result */
29.             out = 0x7FFFFFFF ^ (in >> 31);
30.         *pDst++ = out;                     /* Store result destination */
31.
32.         in = *pSrc++;
33.         in = ((q63_t) in * scaleFract) >> 32;
34.         out = in << kShift;
35.         if (in != (out >> kShift))
36.             out = 0x7FFFFFFF ^ (in >> 31);
37.         *pDst++ = out;
38.
39.         in = *pSrc++;
40.         in = ((q63_t) in * scaleFract) >> 32;
41.         out = in << kShift;
42.         if (in != (out >> kShift))
43.             out = 0x7FFFFFFF ^ (in >> 31);
44.         *pDst++ = out;
45.
46.         in = *pSrc++;
47.         in = ((q63_t) in * scaleFract) >> 32;
48.         out = in << kShift;
49.         if (in != (out >> kShift))
50.             out = 0x7FFFFFFF ^ (in >> 31);
51.         *pDst++ = out;
52.
53.         /* Decrement loop counter */
54.         blkCnt--;
55.     }
56. }
57. else
58. {
59.     while (blkCnt > 0U)
60.     {
61.         /* C = A * scale */
62.
63.         /* Scale input and store result in destination buffer. */
```



```
64.     in = *pSrc++;
65.     in = ((q63_t) in * scaleFract) >> 32;
66.     out = in >> -kShift;
67.     *pDst++ = out;
68.
69.     in = *pSrc++;
70.     in = ((q63_t) in * scaleFract) >> 32;
71.     out = in >> -kShift;
72.     *pDst++ = out;
73.
74.     in = *pSrc++;
75.     in = ((q63_t) in * scaleFract) >> 32;
76.     out = in >> -kShift;
77.     *pDst++ = out;
78.
79.     in = *pSrc++;
80.     in = ((q63_t) in * scaleFract) >> 32;
81.     out = in >> -kShift;
82.     *pDst++ = out;
83.
84.     /* Decrement loop counter */
85.     blkCnt--;
86. }
87. }
88.
89. /* Loop unrolling: Compute remaining outputs */
90. blkCnt = blockSize % 0x4U;
91.
92. #else
93.
94.     /* Initialize blkCnt with number of samples */
95.     blkCnt = blockSize;
96.
97. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
98.
99.     if (sign == 0U)
100.    {
101.        while (blkCnt > 0U)
102.        {
103.            /* C = A * scale */
104.
105.            /* Scale input and store result in destination buffer. */
106.            in = *pSrc++;
107.            in = ((q63_t) in * scaleFract) >> 32;
108.            out = in << kShift;
109.            if (in != (out >> kShift))
110.                out = 0x7FFFFFFF ^ (in >> 31);
111.            *pDst++ = out;
112.
113.            /* Decrement loop counter */
114.            blkCnt--;
115.        }
116.    }
117.    else
118.    {
119.        while (blkCnt > 0U)
120.        {
121.            /* C = A * scale */
```



```
122.  
123.     /* Scale input and store result in destination buffer. */  
124.     in = *pSrc++;  
125.     in = ((q63_t) in * scaleFract) >> 32;  
126.     out = in >> -kShift;  
127.     *pDst++ = out;  
128.  
129.     /* Decrement loop counter */  
130.     blkCnt--;  
131. }
```

函数描述：

这个函数用于求 32 位定点数的比例因子计算。

函数解析：

- ◆ 第 13 到 92 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。
 - 第 18 行到 56 行，如果函数的移位形参 shift 是正数，那么执行左移。
 - 第 57 行到 87 行，如果函数的移位形参 shift 是负数，那么执行右移。
 - 这里特别注意一点，两个 Q31 函数相乘是 2.62 格式，而函数的结果要是 Q31 格式的，所以程序里面做了专门处理。
第 26 行，左移 32 位，那么结果就是 2.30 格式。
第 27 行，kShift = shift + 1，也就是 out = in << (shift + 1) 多执行了一次左移操作。
相当于 2.30 格式，转换为 2.31 格式。
 - 第 28 到 29 行，做了一个 Q31 的饱和处理，也就是将 2.31 格式转换为 1.31。
数值的左移仅支持将其左移后再右移相应的位数后数值不变的情况，如果不满足这个条件，那么要对输出结果做饱和运算，这里分两种情况：
$$\begin{aligned} \text{out} &= 0xFFFFFFFF \wedge (\text{in} >> 31) \quad (\text{in 是正数}) \\ &= 0xFFFFFFFF \wedge 0x00000000 \\ &= 0xFFFFFFFF \\ \text{out} &= 0xFFFFFFFF \wedge (\text{in} >> 31) \quad (\text{in 是负数}) \\ &= 0xFFFFFFFF \wedge 0xFFFFFFFF \\ &= 0x80000000 \end{aligned}$$
- ◆ 第 99 到 132 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是数据源地址。
- ◆ 第 2 个参数是比例因子。
- ◆ 第 3 个参数是移位参数，正数表示右移，负数表示左移。
- ◆ 第 4 参数是结果地址。
- ◆ 第 5 参数是数据块大小，其实就是执行比例因子计算的次数。



12.7.3 函数 arm_scale_q15

函数原型：

```
1. void arm_shift_q15(
2.     const q15_t * pSrc,
3.     int8_t shiftBits,
4.     q15_t * pDst,
5.     uint32_t blockSize)
6. {
7.     uint32_t blkCnt;                      /* Loop counter */
8.     uint8_t sign = (shiftBits & 0x80);    /* Sign of shiftBits */
9.
10. #if defined (ARM_MATH_LOOPUNROLL)
11.
12. #if defined (ARM_MATH_DSP)               /* Temporary input variables */
13.     q15_t in1, in2;
14. #endif
15.
16. /* Loop unrolling: Compute 4 outputs at a time */
17. blkCnt = blockSize >> 2U;
18.
19. /* If the shift value is positive then do right shift else left shift */
20. if (sign == 0U)
21. {
22.     while (blkCnt > 0U)
23.     {
24.         /* C = A << shiftBits */
25.
26. #if defined (ARM_MATH_DSP)
27.         /* read 2 samples from source */
28.         in1 = *pSrc++;
29.         in2 = *pSrc++;
30.
31.         /* Shift the inputs and then store the results in the destination buffer. */
32. #ifndef ARM_MATH_BIG_ENDIAN
33.         write_q15x2_ia (&pDst, __PKHBT(__SSAT((in1 << shiftBits), 16),
34.                                         __SSAT((in2 << shiftBits), 16), 16));
35. #else
36.         write_q15x2_ia (&pDst, __PKHBT(__SSAT((in2 << shiftBits), 16),
37.                                         __SSAT((in1 << shiftBits), 16), 16));
38. #endif /* #ifndef ARM_MATH_BIG_ENDIAN */
39.
40.         /* read 2 samples from source */
41.         in1 = *pSrc++;
42.         in2 = *pSrc++;
43.
44. #ifndef ARM_MATH_BIG_ENDIAN
45.         write_q15x2_ia (&pDst, __PKHBT(__SSAT((in1 << shiftBits), 16),
46.                                         __SSAT((in2 << shiftBits), 16), 16));
47. #else
48.         write_q15x2_ia (&pDst, __PKHBT(__SSAT((in2 << shiftBits), 16),
49.                                         __SSAT((in1 << shiftBits), 16), 16));
50. #endif /* #ifndef ARM_MATH_BIG_ENDIAN */
51.
52. #else
53.     *pDst++ = __SSAT((q31_t) *pSrc++ << shiftBits), 16);
54.
```



```
54.     *pDst++ = __SSAT(((q31_t) *pSrc++ << shiftBits), 16);
55.     *pDst++ = __SSAT(((q31_t) *pSrc++ << shiftBits), 16);
56.     *pDst++ = __SSAT(((q31_t) *pSrc++ << shiftBits), 16);
57. #endif
58.
59.     /* Decrement loop counter */
60.     blkCnt--;
61. }
62. }
63. else
64. {
65.     while (blkCnt > 0U)
66.     {
67.         /* C = A >> shiftBits */
68.
69. #if defined (ARM_MATH_DSP)
70.         /* read 2 samples from source */
71.         in1 = *pSrc++;
72.         in2 = *pSrc++;
73.
74.         /* Shift the inputs and then store the results in the destination buffer. */
75. #ifndef ARM_MATH_BIG_ENDIAN
76.         write_q15x2_ia (&pDst, __PKHBT((in1 >> -shiftBits),
77.                             (in2 >> -shiftBits), 16));
78. #else
79.         write_q15x2_ia (&pDst, __PKHBT((in2 >> -shiftBits),
80.                             (in1 >> -shiftBits), 16));
81. #endif /* #ifndef ARM_MATH_BIG_ENDIAN */
82.
83.         /* read 2 samples from source */
84.         in1 = *pSrc++;
85.         in2 = *pSrc++;
86.
87. #ifndef ARM_MATH_BIG_ENDIAN
88.         write_q15x2_ia (&pDst, __PKHBT((in1 >> -shiftBits),
89.                             (in2 >> -shiftBits), 16));
90. #else
91.         write_q15x2_ia (&pDst, __PKHBT((in2 >> -shiftBits),
92.                             (in1 >> -shiftBits), 16));
93. #endif /* #ifndef ARM_MATH_BIG_ENDIAN */
94.
95. #else
96.     *pDst++ = (*pSrc++ >> -shiftBits);
97.     *pDst++ = (*pSrc++ >> -shiftBits);
98.     *pDst++ = (*pSrc++ >> -shiftBits);
99.     *pDst++ = (*pSrc++ >> -shiftBits);
100. #endif
101.
102.    /* Decrement loop counter */
103.    blkCnt--;
104. }
105. }
106.
107. /* Loop unrolling: Compute remaining outputs */
108. blkCnt = blockSize % 0x4U;
109.
110. #else
111.
```



```
112. /* Initialize blkCnt with number of samples */
113. blkCnt = blockSize;
114.
115. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
116.
117. /* If the shift value is positive then do right shift else left shift */
118. if (sign == OU)
119. {
120.     while (blkCnt > OU)
121.     {
122.         /* C = A << shiftBits */
123.
124.         /* Shift input and store result in destination buffer. */
125.         *pDst++ = __SSAT((q31_t) *pSrc++ << shiftBits), 16);
126.
127.         /* Decrement loop counter */
128.         blkCnt--;
129.     }
130. }
131. else
132. {
133.     while (blkCnt > OU)
134.     {
135.         /* C = A >> shiftBits */
136.
137.         /* Shift input and store result in destination buffer. */
138.         *pDst++ = (*pSrc++ >> -shiftBits);
139.
140.         /* Decrement loop counter */
141.         blkCnt--;
142.     }
143. }
144.
145. }
```

函数描述：

这个函数用于求 16 位定点数的比例因子计算。

函数解析：

◆ 第 10 到 110 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。

- 第 20 到 62 行，如果函数的移位形参 shiftBits 是正数，执行左移。
- 第 63 到 105 行，如果函数的移位形参 shiftBits 是负数，执行右移。
- 第 33 行，函数 __PKHBT 也是 SIMD 指令，作用是将两个 16 位的数据合并成 32 位数据。用 C 实现的话，如下：

```
#define __PKHBT(ARG1, ARG2, ARG3) ( (((int32_t)(ARG1) << 0) & (int32_t)0x0000FFFF) | \
                                (((int32_t)(ARG2) << ARG3) & (int32_t)0xFFFF0000) )
```

函数 write_q15x2_ia 的原型如下：

```
_STATIC_FORCEINLINE void write_q15x2_ia (
    q15_t ** pQ15,
    q31_t    value)
{
    q31_t val = value;

    memcpy (*pQ15, &val, 4);
```



```
*pQ15 += 2;  
}
```

作用是写入两次 Q15 格式数据，组成一个 Q31 格式数据，并将数据地址递增，方便下次写入。

- ◆ 第 118 到 143 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理

函数参数：

- ◆ 第 1 个参数是数据源地址。
- ◆ 第 2 个参数是比例因子。
- ◆ 第 3 个参数是移位参数，正数表示右移，负数表示左移。
- ◆ 第 4 参数是结果地址。
- ◆ 第 5 参数是数据块大小，其实就是执行比例因子计算的次数。

12.7.4 函数 arm_scale_q7

函数原型：

```
1. void arm_scale_q7(  
2.     const q7_t * pSrc,  
3.             q7_t scaleFract,  
4.             int8_t shift,  
5.             q7_t * pDst,  
6.             uint32_t blockSize)  
7. {  
8.     uint32_t blkCnt;                      /* Loop counter */  
9.     int8_t kShift = 7 - shift;              /* Shift to apply after scaling */  
10.  
11. #if defined (ARM_MATH_LOOPUNROLL)  
12.  
13. #if defined (ARM_MATH_DSP)  
14.     q7_t in1,   in2,   in3,   in4;          /* Temporary input variables */  
15.     q7_t out1,  out2,  out3,  out4;         /* Temporary output variables */  
16. #endif  
17.  
18. /* Loop unrolling: Compute 4 outputs at a time */  
19. blkCnt = blockSize >> 2U;  
20.  
21. while (blkCnt > 0U)  
22. {  
23.     /* C = A * scale */  
24.  
25. #if defined (ARM_MATH_DSP)  
26.     /* Reading 4 inputs from memory */  
27.     in1 = *pSrc++;  
28.     in2 = *pSrc++;  
29.     in3 = *pSrc++;  
30.     in4 = *pSrc++;  
31.  
32.     /* Scale inputs and store result in the temporary variable. */  
33.     out1 = (q7_t) (_SSAT(((in1) * scaleFract) >> kShift, 8));  
34.     out2 = (q7_t) (_SSAT(((in2) * scaleFract) >> kShift, 8));  
35.     out3 = (q7_t) (_SSAT(((in3) * scaleFract) >> kShift, 8));  
36.     out4 = (q7_t) (_SSAT(((in4) * scaleFract) >> kShift, 8));  
37.  
38.     /* Pack and store result in destination buffer (in single write) */
```



```
39.     write_q7x4_ia (&pDst, __PACKq7(out1, out2, out3, out4));
40. #else
41.     *pDst++ = (q7_t) (__SSAT(((q15_t) *pSrc++ * scaleFract) >> kShift), 8));
42.     *pDst++ = (q7_t) (__SSAT(((q15_t) *pSrc++ * scaleFract) >> kShift), 8));
43.     *pDst++ = (q7_t) (__SSAT(((q15_t) *pSrc++ * scaleFract) >> kShift), 8));
44.     *pDst++ = (q7_t) (__SSAT(((q15_t) *pSrc++ * scaleFract) >> kShift), 8));
45. #endif
46.
47.     /* Decrement loop counter */
48.     blkCnt--;
49. }
50.
51. /* Loop unrolling: Compute remaining outputs */
52. blkCnt = blockSize % 0x4U;
53.
54. #else
55.
56.     /* Initialize blkCnt with number of samples */
57.     blkCnt = blockSize;
58.
59. #endif /* #if defined (ARM_MATH_LOOPUNROLL) */
60.
61. while (blkCnt > 0U)
62. {
63.     /* C = A * scale */
64.
65.     /* Scale input and store result in destination buffer. */
66.     *pDst++ = (q7_t) (__SSAT(((q15_t) *pSrc++ * scaleFract) >> kShift), 8));
67.
68.     /* Decrement loop counter */
69.     blkCnt--;
70. }
71.
72. }
```

函数描述：

这个函数用于求 8 位定点数的比例因子计算。

函数解析：

- ◆ 第 9 行，这个变量设计很巧妙，这样下面处理正数左移和负数右移就很方便了，可以直接使用一个右移就可以实现。
- ◆ 第 11 到 54 行，实现四个为一组进行计数，好处是加快执行速度，降低 while 循环占用时间。

- 33 到 36 行，对输入的数据做 8 位的饱和处理。比如：

$$\begin{aligned} & (\text{in1} * \text{scaleFract}) >> \text{kShift} \\ &= (\text{in1} * \text{scaleFract}) * 2^{\text{shift} - 7} \\ &= ((\text{in1} * \text{scaleFract}) >> 7) * (2^{\text{shift}}) \end{aligned}$$

源数据 in1 格式 Q7 乘以比例因子 scaleFract 格式 Q7，也就是 2.14 格式，再右移 7bit 就是 2.7 格式，

此时如果 shift 正数，那么就是当前结果左移 shift 位，如果 shift 是负数，那么就是当前结果右移 shift 位。最终结果通过 __SSAT 做个饱和运算。



- ◆ 第 61 到 70 行，四个为一组剩余数据的处理或者不采用四个为一组时数据处理。

函数参数：

- ◆ 第 1 个参数是数据源地址。
- ◆ 第 2 个参数是比例因子。
- ◆ 第 3 个参数是移位参数，正数表示右移，负数表示左移。
- ◆ 第 4 参数是结果地址。
- ◆ 第 5 参数是数据块大小，其实就是执行比例因子计算的次数。

12.7.5 使用举例

程序设计：

```
/*
***** 函数名: DSP_Scale
***** 功能说明: 比例因子
***** 形参: 无
***** 返回值: 无
*/
static void DSP_Scale(void)
{
    float32_t pSrcA[5] = {1.0f, 1.0f, 1.0f, 1.0f, 1.0f};
    float32_t scale = 0.0f;
    float32_t pDst[5];

    q31_t pSrcA1[5] = {0x6fffffff, 1, 1, 1, 1};
    q31_t scale1 = 0x6fffffff;
    q31_t pDst1[5];

    q15_t pSrcA2[5] = {0x6fff, 1, 1, 1, 1};
    q15_t scale2 = 0x6fff;
    q15_t pDst2[5];

    q7_t pSrcA3[5] = {0x70, 1, 1, 1, 1};
    q7_t scale3 = 0x6f;
    q7_t pDst3[5];

    /*求比例因子计算*****
    scale += 0.1f;
    arm_scale_f32(pSrcA, scale, pDst, 5);
    printf("arm_scale_f32 = %f\r\n", pDst[0]);

    scale1 += 1;
    arm_scale_q31(pSrcA1, scale1, 0, pDst1, 5);
    printf("arm_scale_q31 = %x\r\n", pDst1[0]);

    scale2 += 1;
    arm_scale_q15(pSrcA2, scale2, 0, pDst2, 5);
    printf("arm_scale_q15 = %x\r\n", pDst2[0]);

    scale3 += 1;
    arm_scale_q7(pSrcA3, scale3, 0, pDst3, 5);
    printf("arm_scale_q7 = %x\r\n", pDst3[0]);
    printf("*****\r\n");
}
```

实验现象：



12.8 实验例程说明 (MDK)

配套例子：

V7-207_DSP 基础运算 (相反数, 偏移, 移位, 减法和比例因子)

实验目的：

1. 学习基础运算 (相反数, 偏移, 移位, 减法和比例因子)

实验内容：

1. 启动一个自动重装软件定时器, 每 100ms 翻转一次 LED2。
2. 按下按键 K1, DSP 求相反数运算。
3. 按下按键 K2, DSP 求偏移运算。
4. 按下按键 K3, DSP 求移位运算。
5. 按下摇杆 OK 键, DSP 求减法运算。
6. 按下摇杆上键, DSP 比例因子运算。

使用 AC6 注意事项

特别注意附件章节 C 的问题

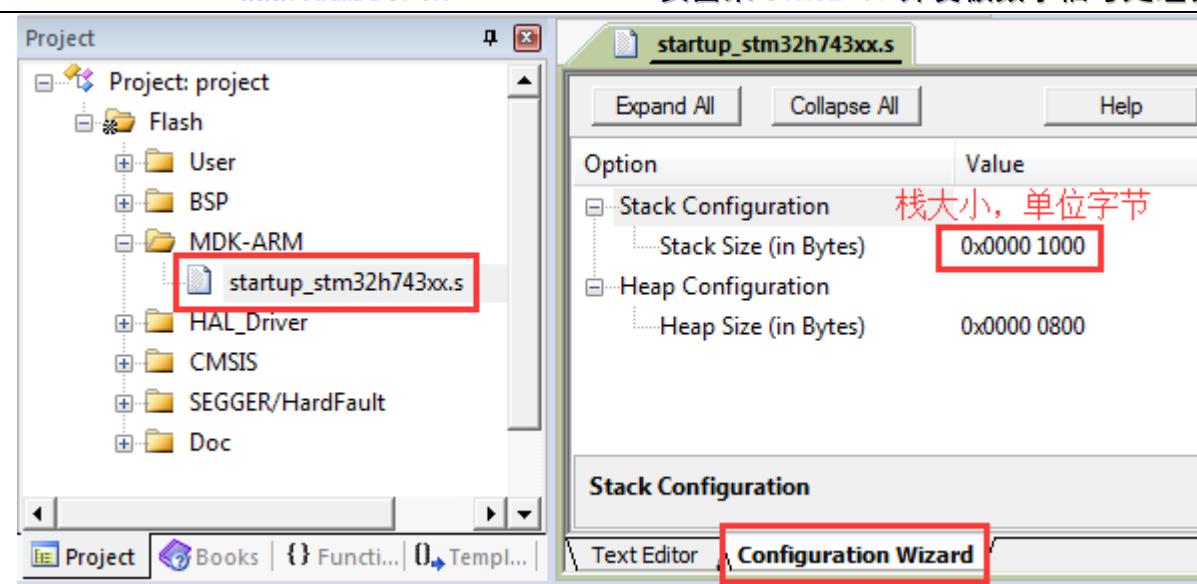
上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

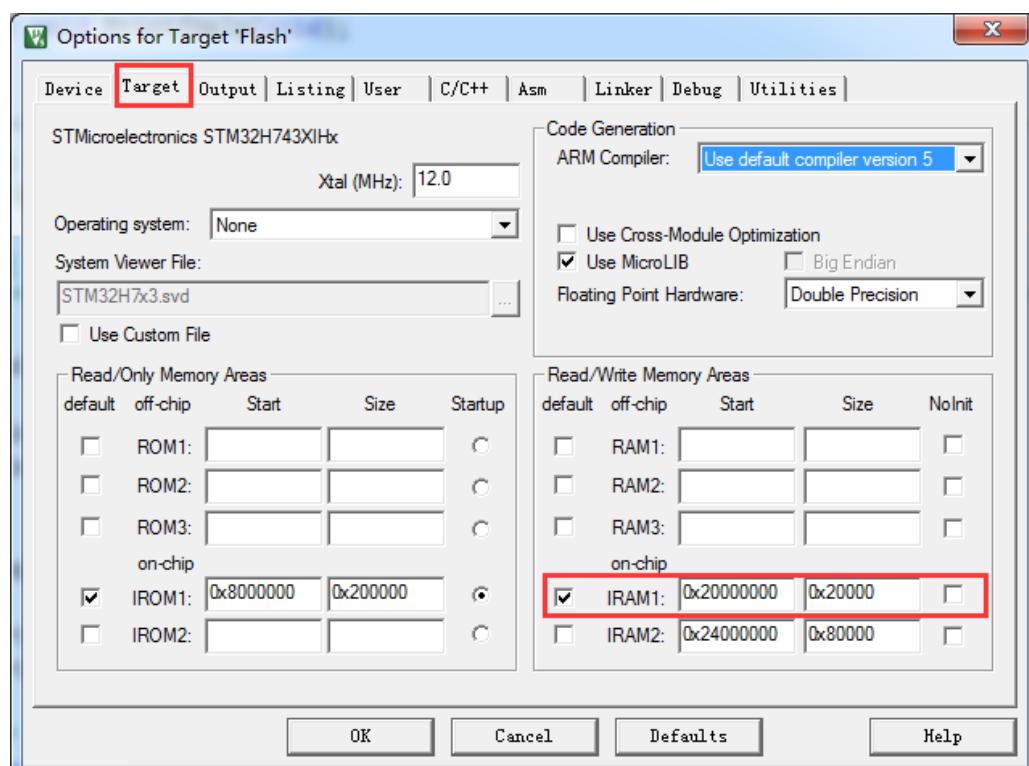
详见本章的 3.5, 4.5, 5.4 和 6.5 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现:

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
```



```
{  
    /* 配置 MPU */  
    MPU_Config();  
  
    /* 使能 L1 Cache */  
    CPU_CACHE_Enable();  
  
    /*  
     * STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：  
     * - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。  
     * - 设置 NVIC 优先级分组为 4。  
     */  
    HAL_Init();  
  
    /*  
     * 配置系统时钟到 400MHz  
     * - 切换使用 HSE。  
     * - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。  
     */  
    SystemClock_Config();  
  
    /*  
     * Event Recorder:  
     * - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。  
     * - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章  
     */  
#if Enable_EventRecorder == 1  
    /* 初始化 EventRecorder 并开启 */  
    EventRecorderInitialize(EventRecordAll, 1U);  
    EventRecorderStart();  
#endif  
  
    bsp_InitKey(); /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */  
    bsp_InitTimer(); /* 初始化滴答定时器 */  
    bsp_InitUart(); /* 初始化串口 */  
    bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */  
    bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();
```



```
/* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。



- 按下按键 K1, DSP 求相反数运算。
- 按下按键 K2, DSP 求偏移运算。
- 按下按键 K3, DSP 求移位运算。
- 按下摇杆 OK 键, DSP 求减法运算。
- 按下摇杆上键, DSP 比例因子运算。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下, 求相反数 */
                    DSP_Negate();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下, 求偏移 */
                    DSP_Offset();
                    break;

                case KEY_DOWN_K3:           /* K3 键按下, 求移位 */
                    DSP_Shift();
                    break;
            }
        }
    }
}
```



```
case JOY_DOWN_OK:           /* 摆杆 OK 键按下, 求減法 */
    DSP_Sub();
    break;

case JOY_DOWN_U:            /* 摆杆上键按下, 求比例因子计算 */
    DSP_Scale();
    break;

default:
    /* 其他的键值不处理 */
    break;
}
}

}
```

12.9 实验例程说明 (IAR)

配套例子:

V7-207_DSP 基础运算 (相反数, 偏移, 移位, 减法和比例因子)

实验目的:

1. 学习基础运算 (相反数, 偏移, 移位, 减法和比例因子)

实验内容:

1. 启动一个自动重装软件定时器, 每 100ms 翻转一次 LED2。
2. 按下按键 K1, DSP 求相反数运算。
3. 按下按键 K2, DSP 求偏移运算。
4. 按下按键 K3, DSP 求移位运算。
5. 按下摇杆 OK 键, DSP 求减法运算。
6. 按下摇杆上键, DSP 比例因子运算。

使用 AC6 注意事项

特别注意附件章节 C 的问题

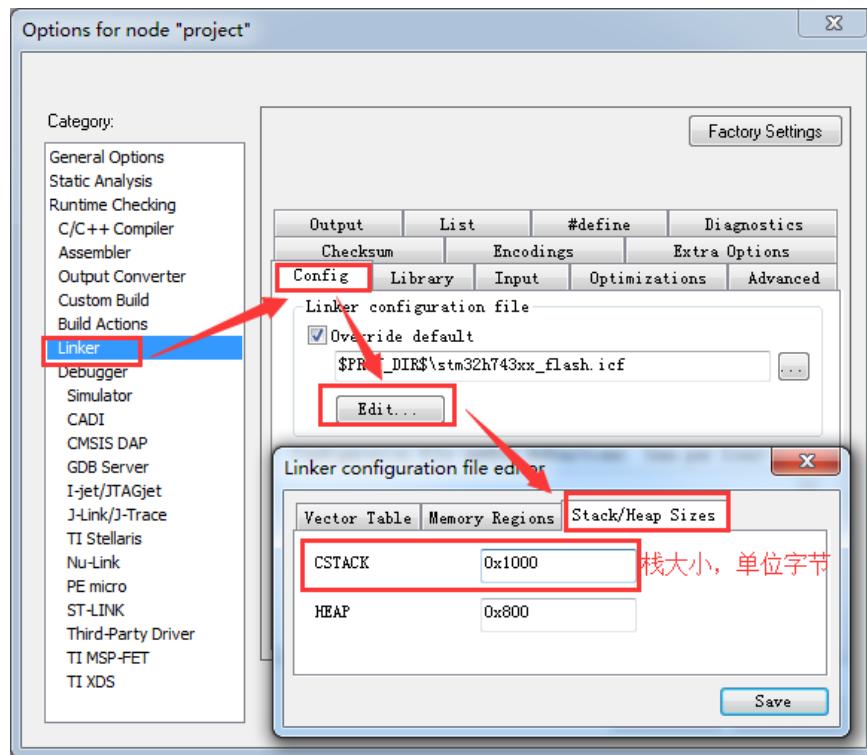
上电后串口打印的信息:

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

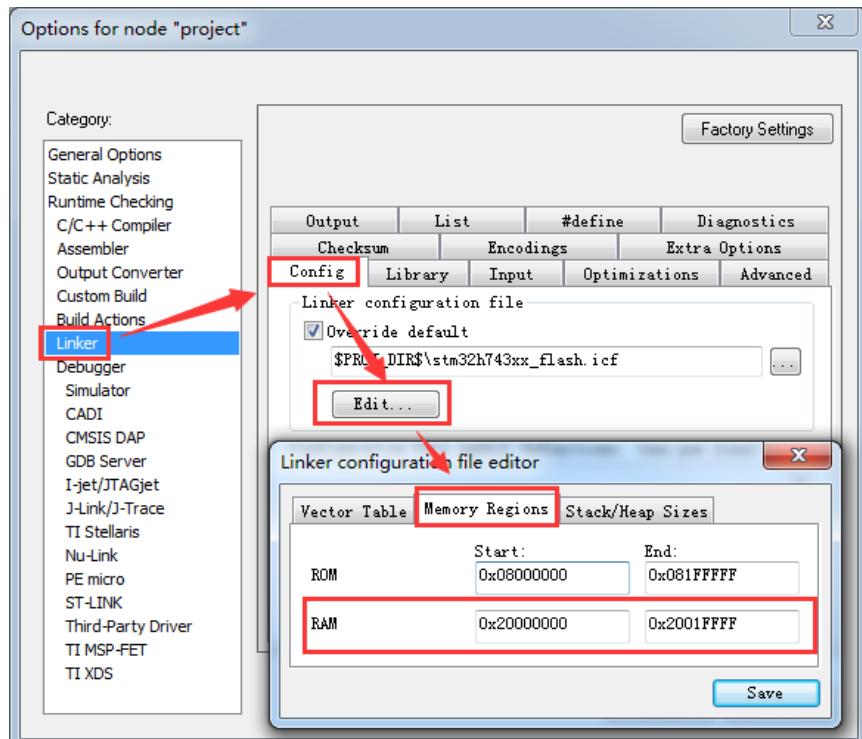
详见本章的 3.5, 4.5, 5.4 和 6.5 小节。

程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
*****
```



```
/* 功能说明：初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形    参：无
* 返回值：无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
        - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
        - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名：MPU_Config
* 功能说明：配置 MPU
* 形    参：无
* 返回值：无
*****
*/
static void MPU_Config( void )
```



```
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL MPU_ConfigRegion(&MPU_InitStruct);  
  
    /*使能 MPU */  
    HAL MPU_Enable(MPU_PRIVILEGED_DEFAULT);  
}  
  
/*  
*****  
* 函数名: CPU_CACHE_Enable  
* 功能说明: 使能 L1 Cache  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void CPU_CACHE_Enable(void)  
{  
    /* 使能 I-Cache */  
    SCB_EnableICache();  
  
    /* 使能 D-Cache */  
    SCB_EnableDCache();  
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1, DSP 求相反数运算。
- 按下按键 K2, DSP 求偏移运算。
- 按下按键 K3, DSP 求移位运算。
- 按下摇杆 OK 键, DSP 求减法运算。
- 按下摇杆上键, DSP 比例因子运算。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下, 求相反数 */
                    DSP_Negate();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下, 求偏移 */
                    DSP_Offset();
                    break;
            }
        }
    }
}
```



```
break;

case KEY_DOWN_K3:           /* K3 键按下, 求移位 */
    DSP_Shift();
    break;

case JOY_DOWN_OK:          /* 摆杆 OK 键按下, 求減法 */
    DSP_Sub();
    break;

case JOY_DOWN_U:            /* 摆杆上键按下, 求比例因子计算 */
    DSP_Scale();
    break;

default:
    /* 其他的键值不处理 */
    break;
}
}

}
```

12.10 总结

DSP 基础函数就跟大家讲这么多，希望初学的同学多多的练习，并在自己以后的项目中多多使用，效果必将事半功倍。



第13章 DSP 快速计算函数-三角函数和平方根

本期教程开始，我们将不再专门的分析 DSP 函数的源码，主要是有些 DSP 函数的公式分析较麻烦，有兴趣的同学可以自行研究，本期教程开始主要讲解函数如何使用。

13.1 初学者重要提示

13.2 DSP 基础运算指令

13.3 三角函数 (Cosine)

13.4 三角函数 (Sine)

13.5 平方根 (Sqrt)

13.6 实验例程说明 (MDK)

13.7 实验例程说明 (IAR)

13.8 总结

13.1 初学者重要提示

- ◆ 特别注意本章 13.5.2 小节的问题，定点数求解平方根。
- ◆ 本章 13.6 小节给出了 Matlab2018a 手动加载数据的方法。如果要看 Matlab2012，参考第 1 版 DSP 教程：<http://www.armbbs.cn/forum.php?mod=viewthread&tid=3886>。

13.2 DSP 基础运算指令

本章用到基础运算指令：

- ◆ 平方根函数用到 _CLZ 指令，全称是 Count Leading Zero
用于求解 32 位数据中从 bit31 开始的 0 的个数。
- ◆ 平方根函数用到 _sqrtf 指令。
用于求解浮点数的平方根，用户可以直接调用此指令，求平方根非常方便。

13.3 三角函数 (Cosine)

三角函数 cosine 的计算是通过查表并配合直线插补实现的。

13.3.1 函数 arm_cos_f32

函数原型：

```
float32_t arm_cos_f32(float32_t x)
```

函数描述：

这个函数用于求 32 位浮点数的 cos 值。

函数参数：

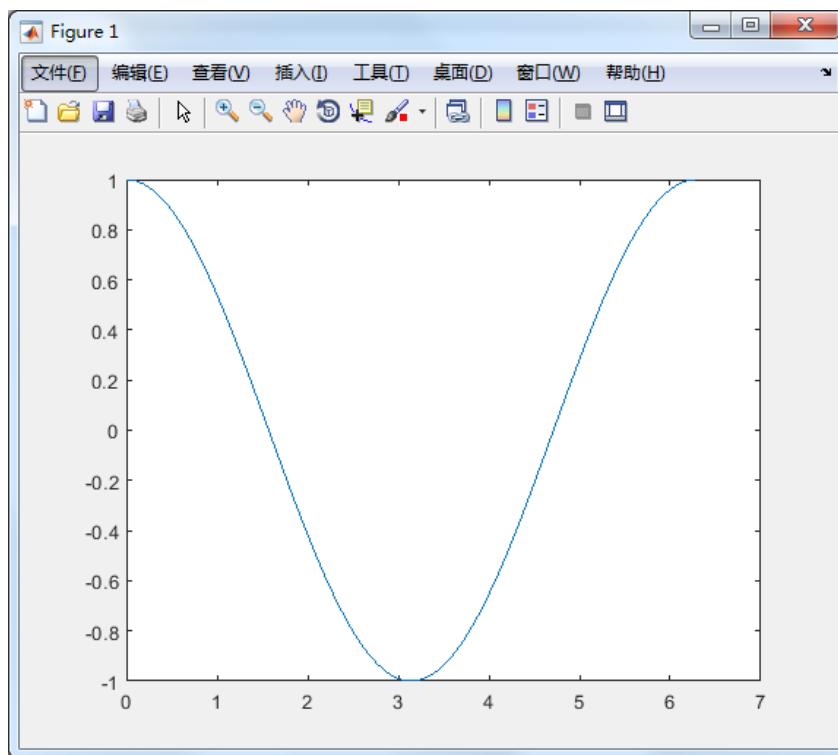
- ◆ 第 1 个参数 x 是弧度制，也就是 cos 函数的一个周期对应于弧度[0 2*PI)。
 $\text{PI} = 3.14159265358979f$
- ◆ 返回值，函数返回计算结果。

Matlab 计算：

下面我们先通过 Matlab 绘制一个周期的 cos 曲线。新建一个.m 格式的脚本文件，并写入如下函数：

```
x = 0:0.01:2*pi;  
plot(x, cos(x))
```

运行后显示效果如下：



点击上面截图中的Tools->Data statistics（工具->数据统计信息）获取数据的分析结果，我们主要看Y轴。



最大值和最小值分别对应1和-1，这个与我们所学的理论知识是相符的。

13.3.2 函数 arm_cos_q31

函数原型:

```
q31_t arm_cos_q31(q31_t x)
```

函数描述:

用于求 32 位定点数的 cos 值。

函数参数:

- ◆ 第 1 个参数 x 是弧度制，参数范围[0 0xFFFFFFFF]（对于的浮点范围是[0 +0.9999]）相当于弧度 [0 2*PI]。
- ◆ 返回值，函数返回计算结果。

13.3.3 函数 arm_cos_q15

函数原型:

```
q15_t arm_cos_q15(q15_t x)
```

函数描述:

用于求 16 位定点数的 cos 值。

函数参数:

- ◆ 第 1 个参数 x 是弧度制，参数范围[0 0xFFFF]（对于的浮点范围是[0 +0.9999]）相当于弧度[0 2*PI]。
- ◆ 返回值，函数返回计算结果。

13.3.4 使用举例

程序设计:

```
/*
***** 函数名: DSP_Cosine
* 功能说明: 求cos函数
```



```
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Cosine(void)
{
    uint16_t i;

    /*cos函数*******/
    for(i = 0; i < 256; i++)
    {
        /* 参数的输入范围是[0 2*pi) */
        printf("%f\r\n", arm_cos_f32(i * PI / 128));
    }
    printf("*****\r\n");

    for(i = 0; i < 256; i++)
    {
        /* 这里是0 到 0xFFFF对应[0 2*pi) */
        printf("%d\r\n", arm_cos_q15(i*128));
    }
    printf("*****\r\n");

    for(i = 0; i < 256; i++)
    {
        /* 这里是0 到 0xFFFFFFFF对应[0 2*pi) */
        printf("%d\r\n", arm_cos_q31(i*8388608));
    }
    printf("*****\r\n");
}
```

实验现象：

The screenshot shows the SecureCRT application window titled "Serial-COM1 - SecureCRT". The menu bar includes File(F), Edit(E), View(V), Options(O), Transfer(T), Scripts(S), Tools(L), and Help(H). The toolbar contains icons for connection, session management, and file operations. The main terminal window displays a list of numerical values representing cosine calculations. Above the terminal, a message box titled "操作提示:" lists four steps related to the experiment. The status bar at the bottom right shows "Serial: COM1 25, 1 25行, 84列 VT100 大写 数字".

操作提示:
1. 启动一个自动重装软件定时器，每100ms翻转一次LED2
2. 按下按键K1, DSP求Cosine
3. 按下按键K2, DSP求Sine
4. 按下按键K3, DSP求平方根

1.000000
0.999699
0.998795
0.997290
0.995185
0.992480
0.989177
0.985278
0.980785
0.975702
0.970031
0.963776
0.956940
0.949528
0.941544
0.932993
0.923880
0.914210
0.903989
0.893224

上面是部分计算结果截图。

13.4 三角函数 (Sine)

三角函数 sine 的计算是通过查表并配合直线插补实现的。具体的实现方法大家可以查阅相关资料进行了解。

13.4.1 函数 arm_sin_f32

函数原型:

```
float32_t arm_sin_f32(float32_t x)
```

函数描述:

这个函数用于求 32 位浮点数的 sin 值。

函数参数:

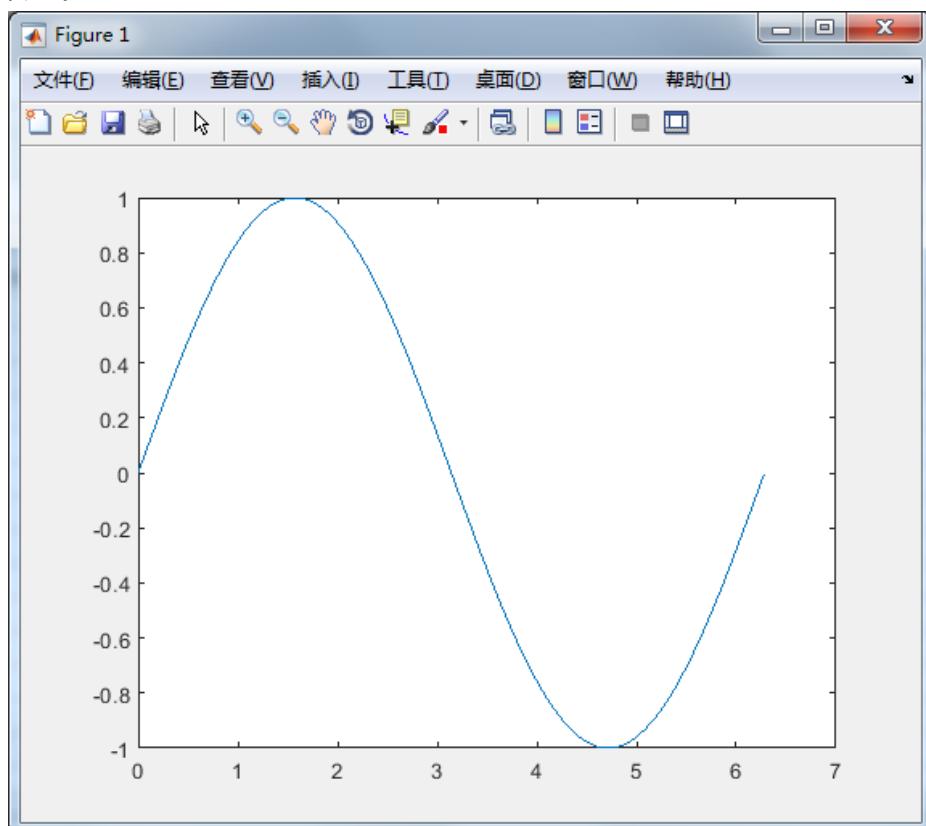
- ◆ 第 1 个参数 x 是弧度制，也就是 sin 函数的一个周期对应于弧度 [0 2*PI) 。
- PI = 3.14159265358979f
- ◆ 返回值，函数返回计算结果。

Matlab 计算：

下面我们先通过 Matlab 绘制一个周期的 sin 曲线。新建一个.m 格式的脚本文件，并写入如下函数：

```
x = 0:0.01:2*pi;
plot(x, sine(x))
```

运行后显示效果如下：



点击上面截图中的 Tools->Data statistics (工具->数据统计信息) 获取数据的分析结果，我们主要看 Y 轴。



最大值和最小值分别对应1和-1，这个与我们所学的理论知识是相符的。

13.4.2 函数 arm_sin_q31

函数原型:

```
q31_t arm_sin_q31(q31_t x)
```

函数描述:

用于求 32 位定点数的 sin 值。

函数参数:

- ◆ 第 1 个参数 x 是弧度制，参数范围[0 0xFFFFFFFF]（对于的浮点范围是[0 +0.9999]）相当于弧度 [0 2*PI]。
- ◆ 返回值，函数返回计算结果。

13.4.3 函数 arm_sin_q15

函数原型:

```
q15_t arm_cos_q15(q15_t x)
```

函数描述:

用于求 16 位定点数的 sin 值。

函数参数:

- ◆ 第 1 个参数 x 是弧度制，参数范围[0 0xFFFF]（对于的浮点范围是[0 +0.9999]）相当于弧度[0 2*PI]。
- ◆ 返回值，函数返回计算结果

13.4.4 使用举例

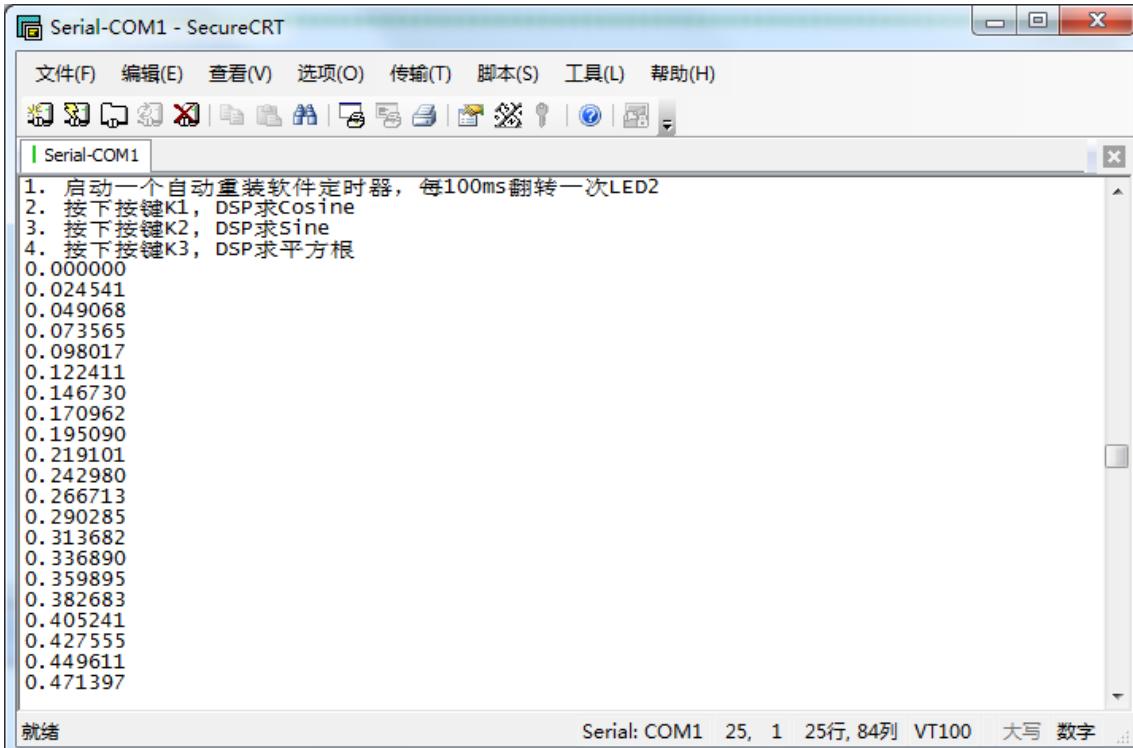
程序设计:

```
/*
```



```
*****  
* 函数名: DSP_Sine  
* 功能说明: 求sine函数  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_Sine(void)  
{  
    uint16_t i;  
  
    /*sin函数*****  
    for(i = 0; i < 256; i++)  
    {  
        /* 参数的输入范围是[0 2*pi) */  
        printf("%f\r\n", arm_sin_f32(i * PI / 128));  
    }  
    printf("*****\r\n");  
  
    for(i = 0; i < 256; i++)  
    {  
        /* 这里是0 - 0xFFFF 对应 [0 2*pi) */  
        printf("%d\r\n", arm_sin_q15(i*128));  
    }  
    printf("*****\r\n");  
  
    for(i = 0; i < 256; i++)  
    {  
        /* 这里是0 - 0xFFFFFFFF 对应 [0 2*pi) */  
        printf("%d\r\n", arm_sin_q31(i*0x800000));  
    }  
    printf("*****\r\n");  
}
```

实验现象：



上面是部分计算结果截图。



13.5 平方根 (Sqrt)

浮点数的平方根计算只需调用一条浮点指令即可，而定点数的计算要稍显麻烦。

13.5.1 函数 arm_sqrt_f32

函数原型：

```
_STATIC_FORCEINLINE arm_status arm_sqrt_f32(  
    float32_t in,  
    float32_t * p0ut)
```

函数描述：

这个函数用于求 32 位定点数的平方根，对于带 FPU 的处理器来说，浮点数的平方根求解很简单，只需调用指令 `_sqrtf`，仅需要 14 个时钟周期就可以完成。

函数形参：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数求平方根后的数据地址。

13.5.2 函数 arm_sqrt_q31

函数原型：

```
arm_status arm_sqrt_q31(  
    q31_t in,  
    q31_t * p0ut)
```

函数描述：

这个函数用于求 32 位定点数的平方根。

函数参数：

- ◆ 第 1 个参数是源数据地址，参数范围 0x00000000 到 0x7FFFFFFF。
- ◆ 第 2 个参数是求平方根后的数据地址。
- ◆ 返回值，返回 ARM_MATH_SUCCESS 表示计算成功，返回 ARM_MATH_ARGUMENT_ERROR 表示计算出错。

注意事项：

这里 `in` 的输入范围是 0x00000000 到 0x7FFFFFFF，转化成浮点数范围就是 [0 +1)。在使用这个函数的时候有一点要特别的注意，比如我们要求 1000 的平方根，而获得结果是 1465429，这是为什么呢，分析如下：

定点数 1000 = 浮点数 $1000 / (2^{31}) = 4.6566e-07$ (用 Q31 表示)。

对 $4.6566e-07$ 求平方根可得 $6.8239e-04$ 。

定点数 1465429 = 浮点数 $1465429 / (2^{31}) = 6.8239e-04$ 。

简单的总结下上面的意思就是说，求定点数 1000 的平方根，实际是求浮点数 $4.6566e-07$ (用 Q31 表示) 的平方根。



13.5.3 函数 arm_sqrt_q15

函数原型:

```
arm_status arm_sqrt_q15(  
    q15_t in,  
    q15_t * p0ut)
```

函数描述:

这个函数用于求 16 点数的平方根

函数参数:

- ◆ 第 1 个参数是源数据地址，参数范围 0x0000 到 0x7FFF (转化成浮点数范围就是[0 +1)) 。
- ◆ 第 2 个参数是求平方根后的数据地址。
- ◆ 返回值，返回 ARM_MATH_SUCCESS 表示计算成功，返回 ARM_MATH_ARGUMENT_ERROR 表示计算出错

13.5.4 使用举例

程序设计:

```
/*  
*****  
* 函数名: DSP_Shift  
* 功能说明: 移位  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_Shift(void)  
{  
    q31_t pSrcA1 = 0x88886666;  
    q31_t pDst1;  
  
    q15_t pSrcA2 = 0x8866;  
    q15_t pDst2;  
  
    q7_t pSrcA3 = 0x86;  
    q7_t pDst3;  
  
    /*求移位*****  
    arm_shift_q31(&pSrcA1, 3, &pDst1, 1);  
    printf("arm_shift_q31 = %8x\r\n", pDst1);  
  
    arm_shift_q15(&pSrcA2, -3, &pDst2, 1);  
    printf("arm_shift_q15 = %4x\r\n", pDst2);  
  
    arm_shift_q7(&pSrcA3, 3, &pDst3, 1);  
    printf("arm_shift_q7 = %2x\r\n", pDst3);  
    printf("*****\r\n");  
}
```

实验现象:

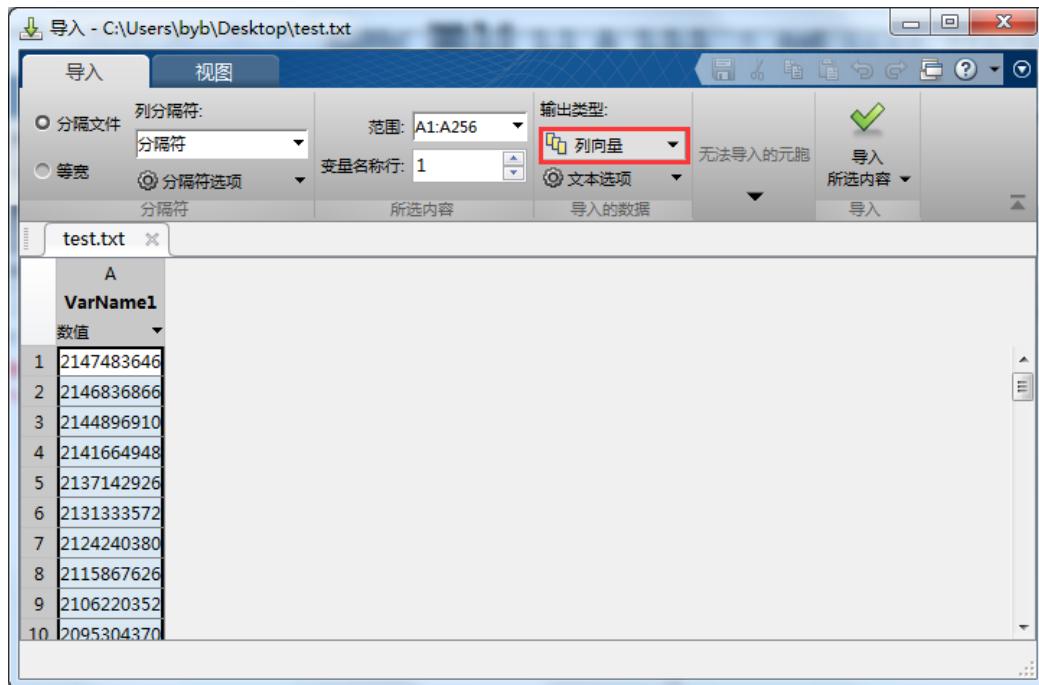


13.6 Matlab 验证（手动加载数据到 Matlab 的方法）

这里我们采样了 cos 曲线一个周期中的 256 个点。为了验证结果是否正确，我们可以将这些数据保存到 txt 文件中，复制这 256 个数据即可，然后保存并关闭文件。通过 matlab 加载这个 txt 文件，加载方法如下：



加载保存好数据的 txt 文件（特别注意输出类型选择列向量）：

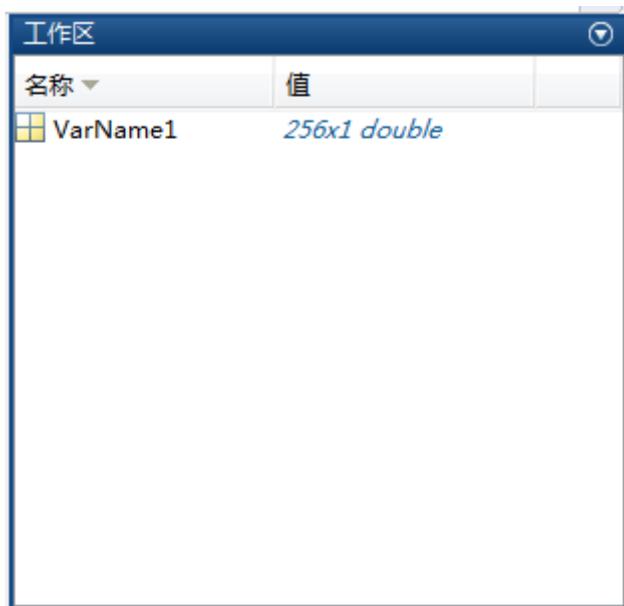


然后点击右上角那个绿色对勾，会提示变量已经导入：

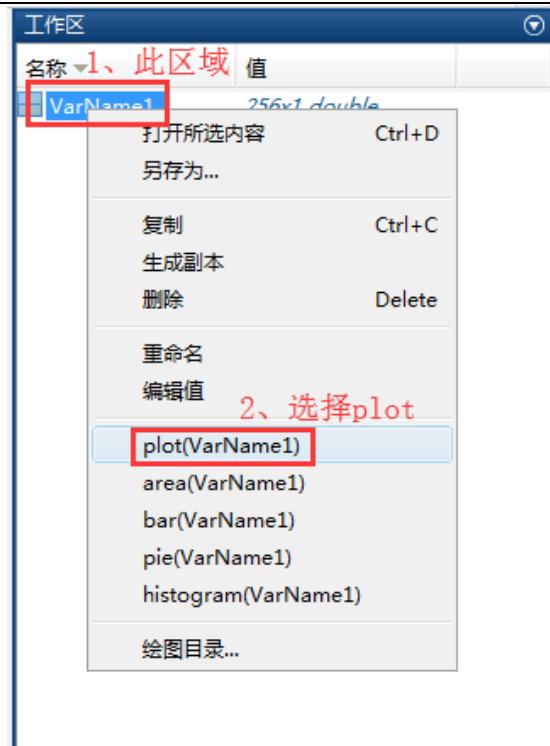


以下变量已导入:
VarName1 (256x1)

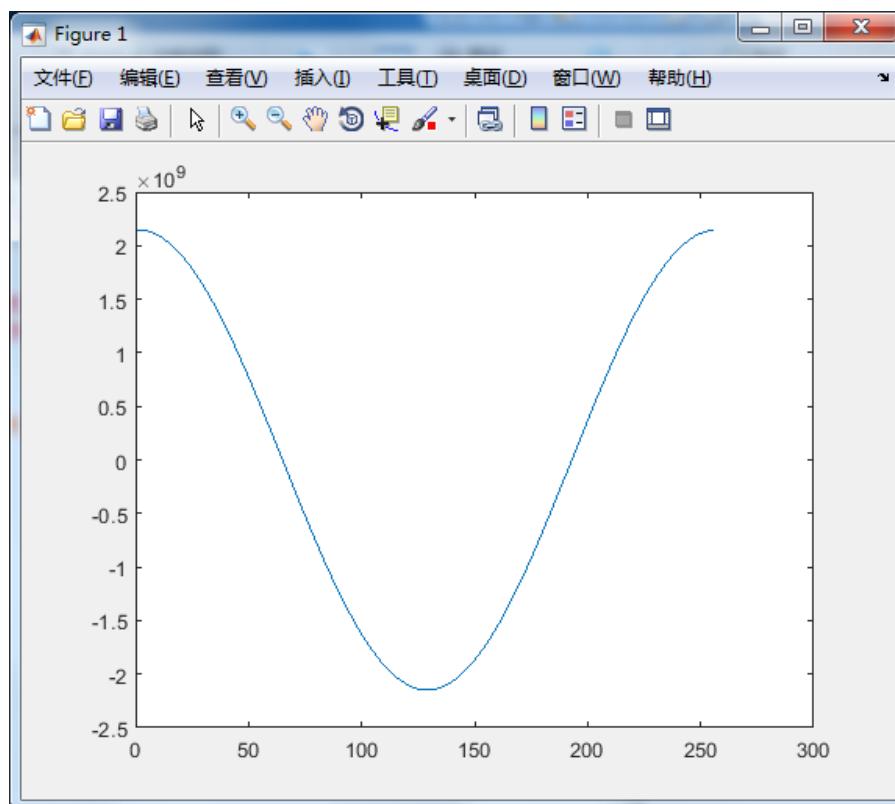
然后再看工作区 (Workspace) 就能看到添加的数组变量了：



现在我们通过 matlab 中的 plot 功能绘制下这些数据，在的 VarName1 的地方右击鼠标，选择 plot



绘制后的结果如下：



从波形上看基本是一个周期的 cos 函数曲线。

13.7 实验例程说明 (MDK)

配套例子：

V7-208_DSP 快速运算 (三角函数和平方根)

实验目的：

1. 学习 DSP 快速运算 (三角函数和平方根)

实验内容：

1. 按下按键 K1, DSP 求 Cosine。
2. 按下按键 K2, DSP 求 Sine。
3. 按下按键 K3, DSP 求平方根。

使用 AC6 注意事项

特别注意附件章节 C 的问题

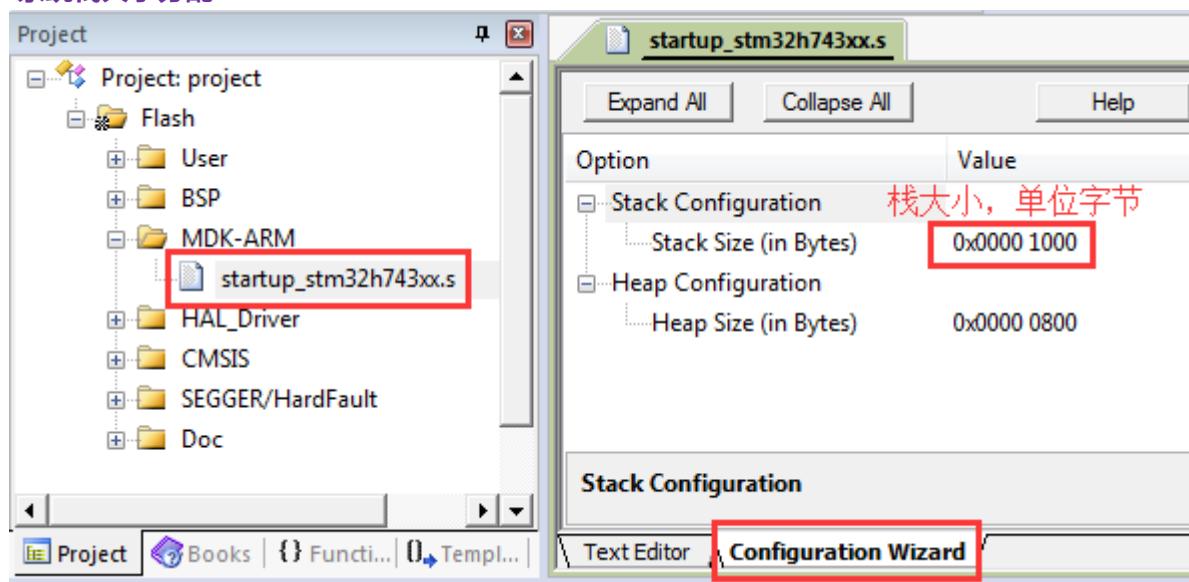
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

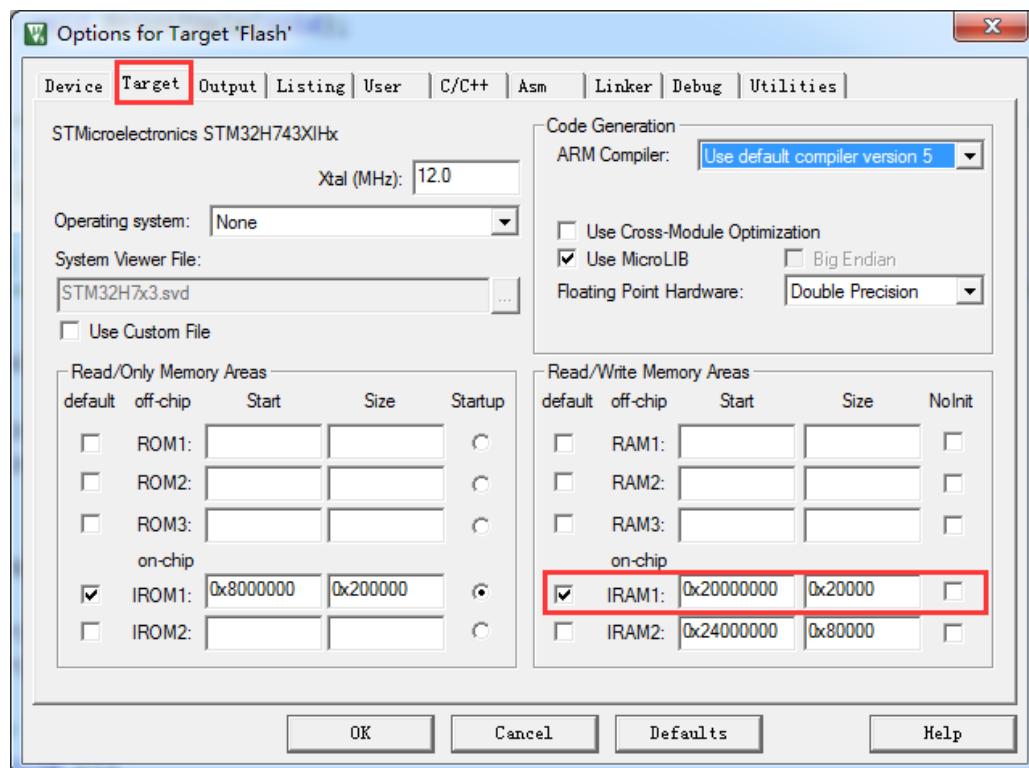
详见本章的 3.4 4.4, 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1, DSP 求 Cosine。
- 按下按键 K2, DSP 求 Sine。
- 按下按键 K3, DSP 求平方根。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */
}
```



```
PrintfHelp(); /* 打印操作提示信息 */
```

```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:           /* K1 键按下，求 Cosine */
                DSP_Cosine();
                break;

            case KEY_DOWN_K2:           /* K2 键按下，求 Sine */
                DSP_Sine();
                break;

            case KEY_DOWN_K3:           /* K3 键按下，求平方根 */
                DSP_Sqrt();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}
```

13.8 实验例程说明 (IAR)

配套例子：

V7-208_DSP 快速运算 (三角函数和平方根)

实验目的：

1. 学习 DSP 快速运算 (三角函数和平方根)

实验内容：

1. 按下按键 K1, DSP 求 Cosine。
2. 按下按键 K2, DSP 求 Sine。

3. 按下按键 K3, DSP 求平方根。

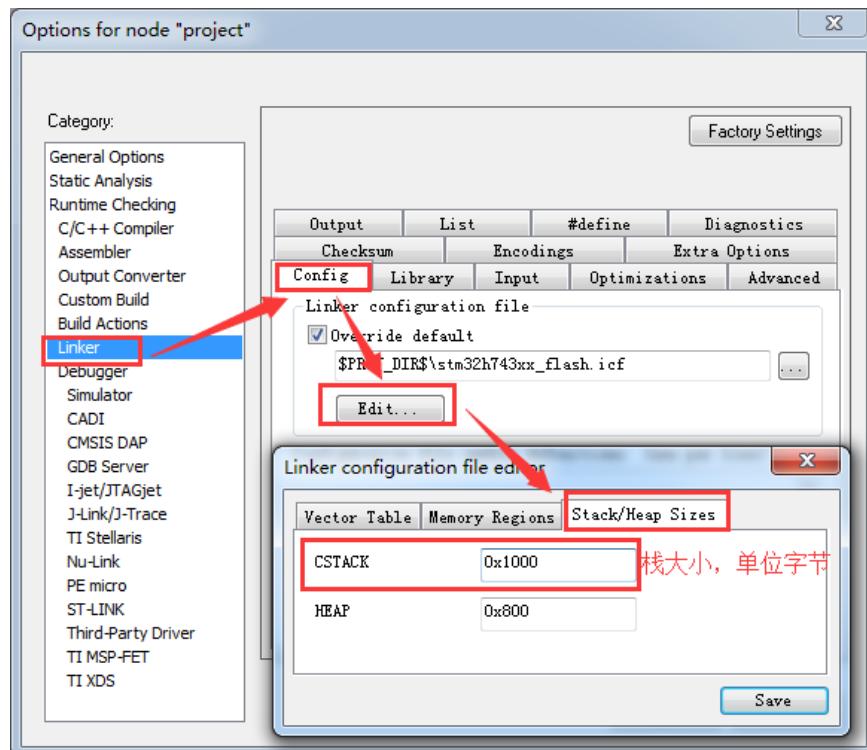
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

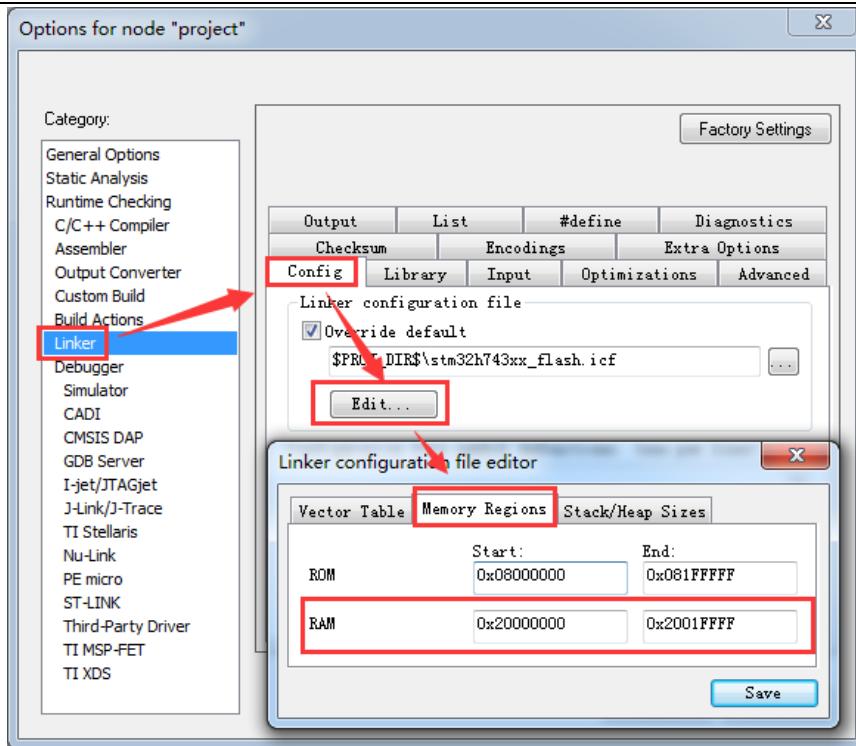
详见本章的 3.4 4.4, 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作:

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1, DSP 求 Cosine。
- 按下按键 K2, DSP 求 Sine。
- 按下按键 K3, DSP 求平方根。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ukeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下，求 Cosine */
                DSP_Cosine();
                break;

            case KEY_DOWN_K2:          /* K2 键按下，求 Sine */
                DSP_Sine();
                break;

            case KEY_DOWN_K3:          /* K3 键按下，求平方根 */
                DSP_Sqrt();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}

}
```

13.9 总结

本期教程就跟大家讲这么多，有兴趣的可以深入研究这些函数源码的实现。



第14章 DSP 统计函数-最大值，最小值，平均值

和功率

本期教程主要讲解统计函数中的最大值，最小值，平均值和功率的计算。

14.1 初学者重要提示

14.2 DSP 基础运算指令

14.3 最大值 (Maximum)

14.4 最小值 (Minimum)

14.5 平均值 (Mean)

14.6 功率 (Power)

14.7 实验例程说明 (MDK)

14.8 实验例程说明 (IAR)

14.9 总结

14.1 初学者重要提示

- ◆ 特别注意本章 13.5.2 小节的问题，定点数求解平方根。
- ◆ 本章 13.6 小节给出了 Matlab2018a 手动加载数据的方法。如果要看 Matlab2012，参考第 1 版 DSP 教程：<http://www.armbbs.cn/forum.php?mod=viewthread&tid=3886>。

14.2 DSP 基础运算指令

本章没有用到 DSP 指令。

14.3 最大值 (Maximum)

这部分函数用于计算数组中的最大值，并返回数组中的最大值和最大值在数组中的位置。

14.3.1 函数 arm_max_f32

函数原型：

```
void arm_max_f32(  
    const float32_t * pSrc,  
    uint32_t blockSize,
```



```
float32_t * pResult,  
uint32_t * pIndex)
```

函数描述：

这个函数用于求 32 位浮点数的最大值。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的最大值。
- ◆ 第 4 个参数是求解出来的最大值在源数据中的位置。

14.3.2 函数 arm_max_q31

函数原型：

```
void arm_max_q31(  
    const q31_t * pSrc,  
    uint32_t blockSize,  
    q31_t * pResult,  
    uint32_t * pIndex)
```

函数描述：

用于求 32 位定点数的最大值。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的最大值。
- ◆ 第 4 个参数是求解出来的最大值在源数据中的位置。

14.3.3 函数 arm_max_q15

函数原型：

```
void arm_max_q15(  
    const q15_t * pSrc,  
    uint32_t blockSize,  
    q15_t * pResult,  
    uint32_t * pIndex)
```

函数描述：

用于求 16 位定点数的最大值。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的最大值。
- ◆ 第 4 个参数是求解出来的最大值在源数据中的位置。



14.3.4 函数 arm_max_q7

函数原型:

```
void arm_max_q7(
    const q7_t * pSrc,
    uint32_t blockSize,
    q7_t * pResult,
    uint32_t * pIndex)
```

函数描述:

用于求 8 位定点数的最大值。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的最大值。
- ◆ 第 4 个参数是求解出来的最大值在源数据。

14.3.5 使用举例

程序设计:

```
/*
*****
* 函数名: DSP_Max
* 功能说明: 求最大值
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Max(void)
{
    float32_t pSrc[10] = {0.6948f, 0.3171f, 0.9502f, 0.0344f, 0.4387f, 0.3816f, 0.7655f, 0.7952f, 0.1869f,
                           0.4898f};
    float32_t pResult;
    uint32_t pIndex;

    q31_t pSrc1[10];
    q31_t pResult1;

    q15_t pSrc2[10];
    q15_t pResult2;

    q7_t pSrc3[10];
    q7_t pResult3;

    arm_max_f32(pSrc, 10, &pResult, &pIndex);
    printf("arm_max_f32 : pResult = %f  pIndex = %d\r\n", pResult, pIndex);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc1[pIndex] = rand();
    }
    arm_max_q31(pSrc1, 10, &pResult1, &pIndex);
    printf("arm_max_q31 : pResult = %d  pIndex = %d\r\n", pResult1, pIndex);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc2[pIndex] = rand()%32768;
    }
```

```
}

arm_max_q15(pSrc2, 10, &pResult2, &pIndex);
printf("arm_max_q15 : pResult = %d pIndex = %d\r\n", pResult2, pIndex);

/***********************/
for(pIndex = 0; pIndex < 10; pIndex++)
{
    pSrc3[pIndex] = rand()%128;
}
arm_max_q7(pSrc3, 10, &pResult3, &pIndex);
printf("arm_max_q7 : pResult = %d pIndex = %d\r\n", pResult3, pIndex);
printf("*****\r\n");

}
```

实验现象：



14.4 最小值 (Minimum)

这部分函数用于计算数组中的最小值，并返回数组中的最小值和最小值在数组中的位置。

14.4.1 函数 arm_min_f32

函数原型：

```
void arm_min_f32(
    const float32_t * pSrc,
    uint32_t blockSize,
    float32_t * pResult,
    uint32_t * pIndex)
```

函数描述：

这个函数用于求 32 位浮点数的最小值。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的最小值。
- ◆ 第 4 个参数是求解出来的最小值在源数据中的位置。



14.4.2 函数 arm_min_q31

函数原型:

```
void arm_min_q31(
    const q31_t * pSrc,
    uint32_t blockSize,
    q31_t * pResult,
    uint32_t * pIndex)
```

函数描述:

用于求 32 位定点数的最小值。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的最小值。
- ◆ 第 4 个参数是求解出来的最小值在源数据中的位置。

14.4.3 函数 arm_min_q15

函数原型:

```
void arm_min_q15(
    const q15_t * pSrc,
    uint32_t blockSize,
    q15_t * pResult,
    uint32_t * pIndex)
```

函数描述:

用于求 16 位定点数的最小值。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的最小值。
- ◆ 第 4 个参数是求解出来的最小值在源数据中的位置。

14.4.4 函数 arm_min_q7

函数原型:

```
void arm_min_q7(
    const q7_t * pSrc,
    uint32_t blockSize,
    q7_t * pResult,
    uint32_t * pIndex)
```

函数描述:

用于求 8 位定点数的最小值。

函数参数:

- ◆ 第 1 个参数源数据地址。



- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的最小值。
- ◆ 第 4 个参数是求解出来的最小值在源数据中的位置。

14.4.5 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_Min
* 功能说明: 求最小值
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Min(void)
{
    float32_t pSrc[10] = {0.6948f, 0.3171f, 0.9502f, 0.0344f, 0.4387f, 0.3816f, 0.7655f, 0.7952f, 0.1869f,
                           0.4898f};
    float32_t pResult;
    uint32_t pIndex;

    q31_t pSrc1[10];
    q31_t pResult1;

    q15_t pSrc2[10];
    q15_t pResult2;

    q7_t pSrc3[10];
    q7_t pResult3;

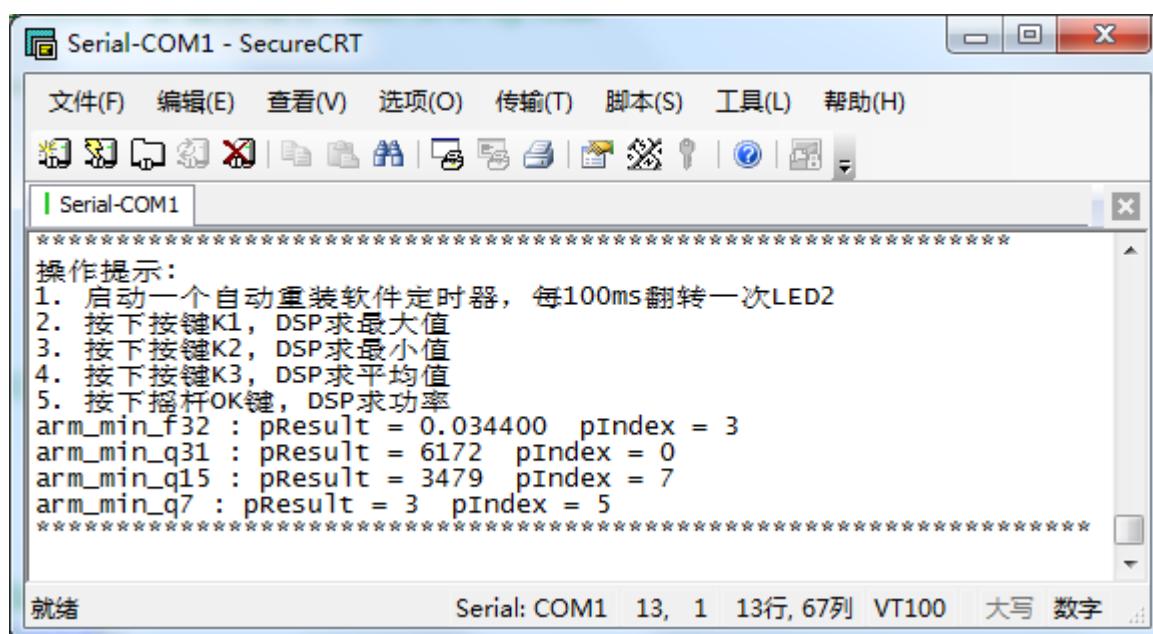
    arm_min_f32(pSrc, 10, &pResult, &pIndex);
    printf("arm_min_f32 : pResult = %f  pIndex = %d\r\n", pResult, pIndex);

    *****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc1[pIndex] = rand();
    }
    arm_min_q31(pSrc1, 10, &pResult1, &pIndex);
    printf("arm_min_q31 : pResult = %d  pIndex = %d\r\n", pResult1, pIndex);

    *****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc2[pIndex] = rand()%32768;
    }
    arm_min_q15(pSrc2, 10, &pResult2, &pIndex);
    printf("arm_min_q15 : pResult = %d  pIndex = %d\r\n", pResult2, pIndex);

    *****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc3[pIndex] = rand()%128;
    }
    arm_min_q7(pSrc3, 10, &pResult3, &pIndex);
    printf("arm_min_q7 : pResult = %d  pIndex = %d\r\n", pResult3, pIndex);
    printf("*****\r\n");
}
```

实验现象：



14.5 平均值 (Mean)

这部分函数用于计算数组的平均值，公式描述如下：

$$\text{Result} = (\text{pSrc}[0] + \text{pSrc}[1] + \text{pSrc}[2] + \dots + \text{pSrc}[\text{blockSize}-1]) / \text{blockSize}.$$

14.5.1 函数 arm_mean_f32

函数原型：

```
void arm_mean_f32(
    const float32_t * pSrc,
    uint32_t blockSize,
    float32_t * pResult)
```

函数描述：

用于求解 32 位浮点数的平均值。

函数形参：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是数据结果。

14.5.2 函数 arm_mean_q31

函数原型：

```
void arm_mean_q31(
    const q31_t * pSrc,
    uint32_t blockSize,
    q31_t * pResult)
```

函数描述：



用于求 32 位定点数的平均值。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是数据结果。

注意事项:

求平均前的数据之和是赋值给了 64 位累加器，然后再求平均。

14.5.3 函数 arm_mean_q15

函数原型:

```
void arm_mean_q15(  
    const q15_t * pSrc,  
    uint32_t blockSize,  
    q15_t * pResult)
```

函数描述:

用于求 16 位定点数的平均值。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是数据结果。

注意事项:

求平均前的数据之和是赋值给了 32 位累加器，然后再求平均。

14.5.4 函数 arm_mean_q7

函数原型:

```
void arm_mean_q7(  
    const q7_t * pSrc,  
    uint32_t blockSize,  
    q7_t * pResult)
```

函数描述:

用于求 8 位定点数的平均值。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是数据结果。

注意事项:

- ◆ 求平均前的数据之和是赋值给了 16 位累加器，然后再求平均。



14.5.5 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_Mean
* 功能说明: 求平均
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Mean(void)
{
    float32_t pSrc[10] = {0.6948f, 0.3171f, 0.9502f, 0.0344f, 0.4387f, 0.3816f, 0.7655f, 0.7952f, 0.1869f,
                          0.4898f};
    float32_t pResult;
    uint32_t pIndex;

    q31_t pSrc1[10];
    q31_t pResult1;

    q15_t pSrc2[10];
    q15_t pResult2;

    q7_t pSrc3[10];
    q7_t pResult3;

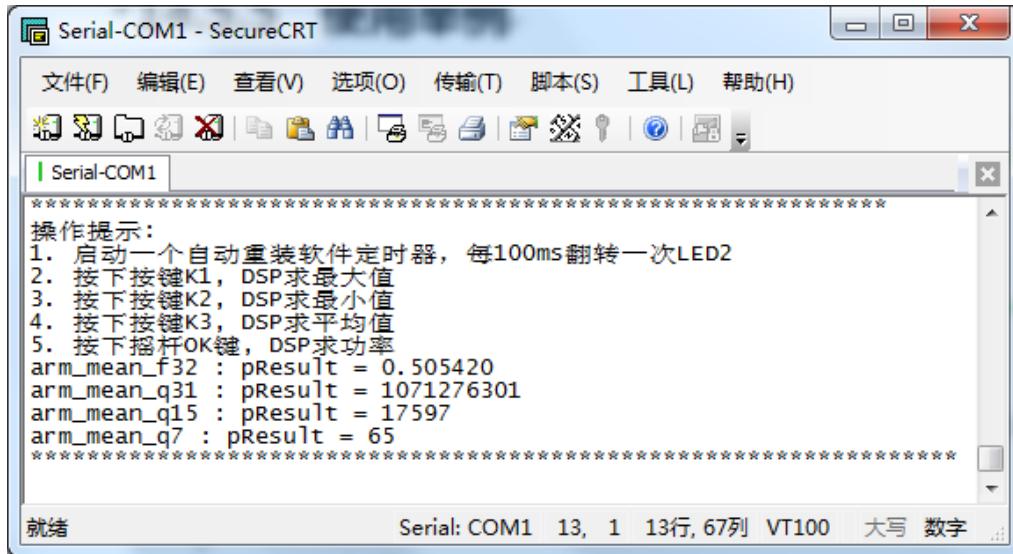
    arm_mean_f32(pSrc, 10, &pResult);
    printf("arm_mean_f32 : pResult = %f\r\n", pResult);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc1[pIndex] = rand();
    }
    arm_mean_q31(pSrc1, 10, &pResult1);
    printf("arm_mean_q31 : pResult = %d\r\n", pResult1);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc2[pIndex] = rand()%32768;
    }
    arm_mean_q15(pSrc2, 10, &pResult2);
    printf("arm_mean_q15 : pResult = %d\r\n", pResult2);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc3[pIndex] = rand()%128;
    }
    arm_mean_q7(pSrc3, 10, &pResult3);
    printf("arm_mean_q7 : pResult = %d\r\n", pResult3);
    printf("*****\r\n");
    */
}
```

实验现象：



14.6 功率 (Power)

这部分函数用于计算数组的功率。公式描述如下：

```
Result = pSrc[0] * pSrc[0] + pSrc[1] * pSrc[1] + pSrc[2] * pSrc[2] + ... + pSrc[blockSize-1] *  
pSrc[blockSize-1];
```

14.6.1 函数 arm_power_f32

函数原型:

```
void arm_power_f32(  
    const float32_t * pSrc,  
    uint32_t blockSize,  
    float32_t * pResult)
```

函数描述:

用于求 32 位浮点数的功率值。

函数形参:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是数据结果。

14.6.2 函数 arm_power_q31

函数原型:

```
void arm_power_q31(  
    const q31_t * pSrc,  
    uint32_t blockSize,  
    q63_t * pResult)
```

函数描述:

用于求 32 位定点数的功率值。

**函数参数:**

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是数据结果。

注意事项:

输入参数是 1.31 格式，两个数据的乘积就是 $1.31 \times 1.31 = 2.62$ 格式，这里将此结果右移 14 位，也就是将低 14 位数据截取掉，最终的输出做 64 位饱和运算，结果是 16.48 格式。

14.6.3 函数 arm_power_q15

函数原型:

```
void arm_power_q15(  
    const q15_t * pSrc,  
    uint32_t blockSize,  
    q63_t * pResult)
```

函数描述:

用于求 16 位定点数的功率值。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是数据结果。

注意事项:

输入参数是 1.15 格式，两个数据的乘积就是 $1.15 \times 1.15 = 2.30$ 格式，最终的输出做 64 位饱和运算，结果是 34.30 格式。

14.6.4 函数 arm_power_q7

函数原型:

```
void arm_min_q7(  
    const q7_t * pSrc,  
    uint32_t blockSize,  
    q7_t * pResult,  
    uint32_t * pIndex)
```

函数描述:

用于求 8 位定点数的功率值。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是数据结果。

注意事项:



输入参数是 1.7 格式，两个数据的乘积就是 $1.7 \times 1.7 = 2.14$ 格式，最终的输出做 32 位饱和运算，结果是 18.14 格式。

14.6.5 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_Power
* 功能说明: 求功率
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Power(void)
{
    float32_t pSrc[10] = {0.6948f, 0.3171f, 0.9502f, 0.0344f, 0.4387f, 0.3816f, 0.7655f, 0.7952f, 0.1869f,
                          0.4898f};
    float32_t pResult;
    uint32_t pIndex;

    q31_t pSrc1[10];
    q63_t pResult1;

    q15_t pSrc2[10];
    q63_t pResult2;

    q7_t pSrc3[10];
    q31_t pResult3;

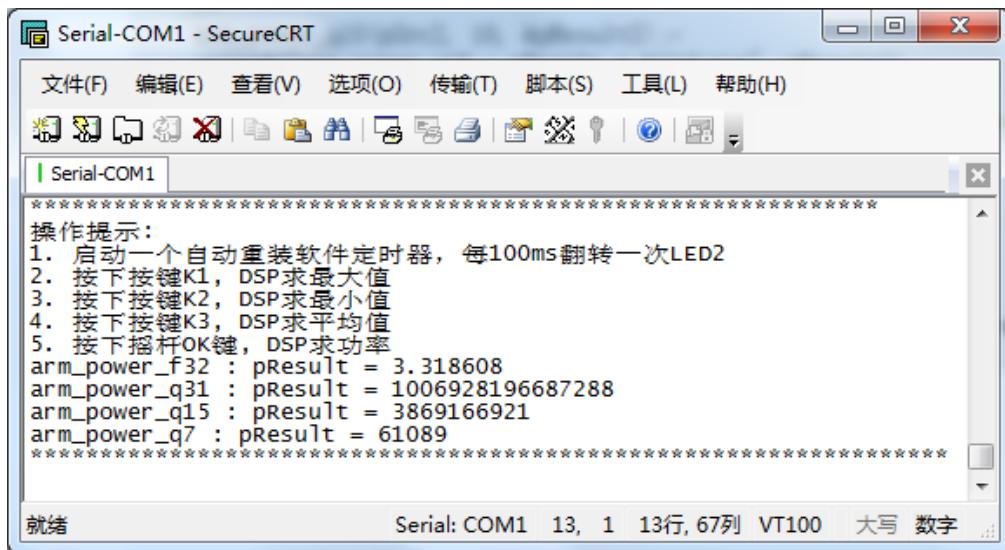
    arm_power_f32(pSrc, 10, &pResult);
    printf("arm_power_f32 : pResult = %f\r\n", pResult);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc1[pIndex] = rand();
    }
    arm_power_q31(pSrc1, 10, &pResult1);
    printf("arm_power_q31 : pResult = %lld\r\n", pResult1);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc2[pIndex] = rand()%32768;
    }
    arm_power_q15(pSrc2, 10, &pResult2);
    printf("arm_power_q15 : pResult = %lld\r\n", pResult2);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc3[pIndex] = rand()%128;
    }
    arm_power_q7(pSrc3, 10, &pResult3);
    printf("arm_power_q7 : pResult = %d\r\n", pResult3);
    printf("*****\r\n");
    */
}
```

实验现象：



14.7 实验例程说明 (MDK)

配套例子：

V7-209_DSP 统计运算 (最大值, 最小值, 平均值和功率)

实验目的：

1. 学习 DSP 快速运算 (三角函数和平方根)

实验内容：

1. 启动一个自动重装软件定时器, 每 100ms 翻转一次 LED2。
2. 按下按键 K1, DSP 求最大值。
3. 按下按键 K2, DSP 求最小值。
4. 按下按键 K3, DSP 求平均值。
5. 按下摇杆 OK 键, DSP 求功率。

使用 AC6 注意事项

特别注意附件章节 C 的问题

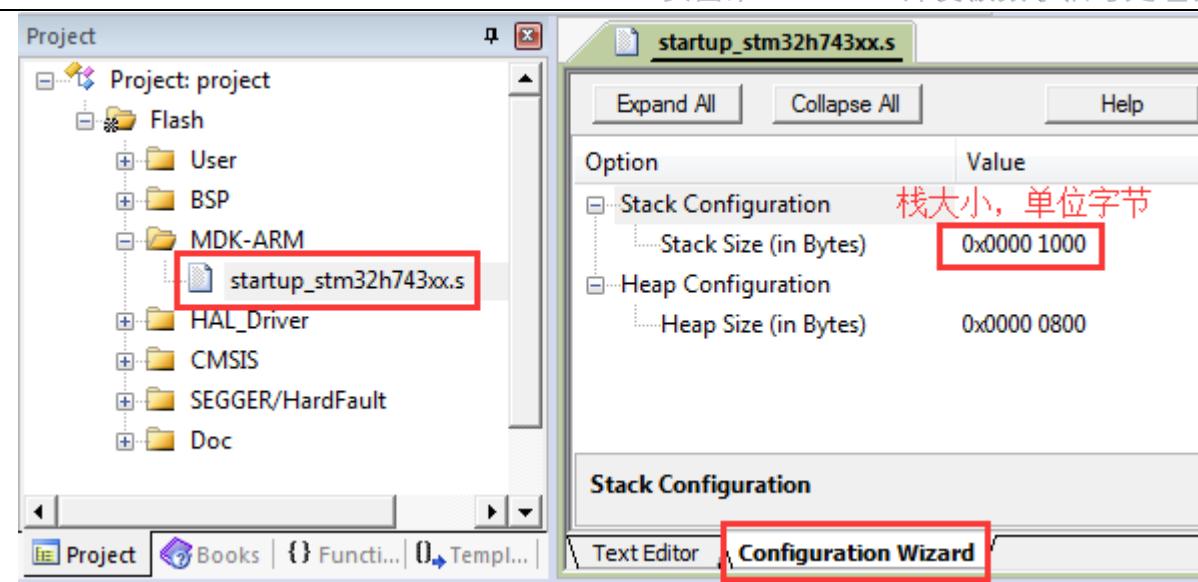
上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

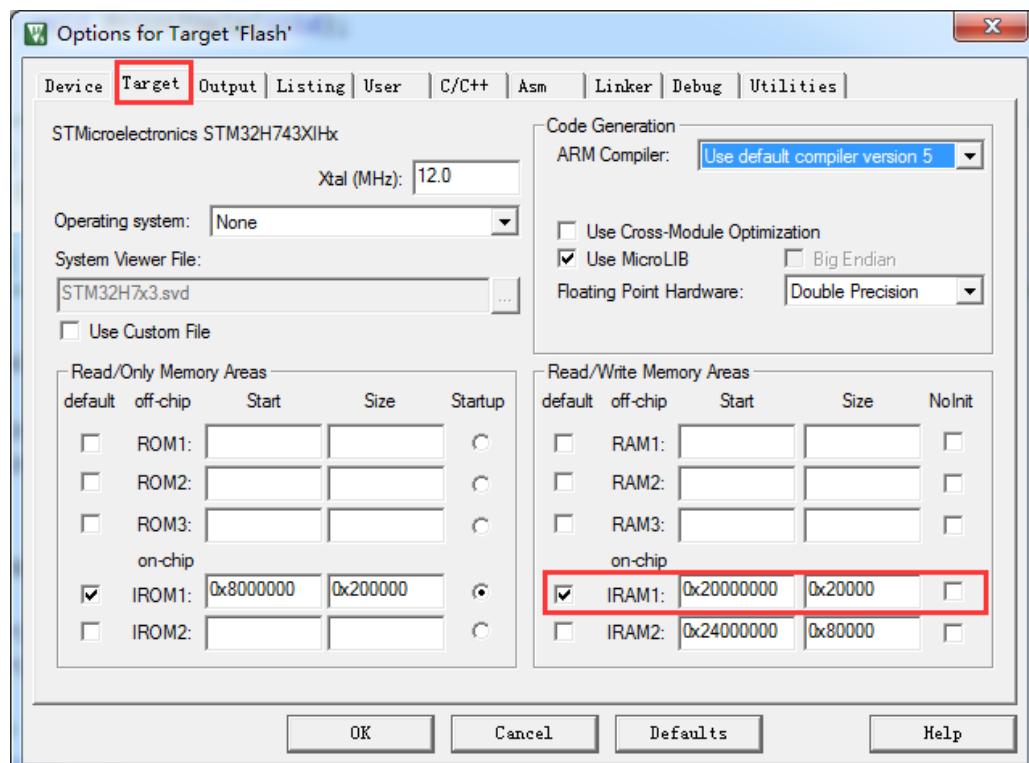
详见本章的 3.5 4.5, 5.5 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现:

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
```



```
{  
    /* 配置 MPU */  
    MPU_Config();  
  
    /* 使能 L1 Cache */  
    CPU_CACHE_Enable();  
  
    /*  
     * STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：  
     * - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。  
     * - 设置 NVIC 优先级分组为 4。  
     */  
    HAL_Init();  
  
    /*  
     * 配置系统时钟到 400MHz  
     * - 切换使用 HSE。  
     * - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。  
     */  
    SystemClock_Config();  
  
    /*  
     * Event Recorder:  
     * - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。  
     * - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章  
     */  
#if Enable_EventRecorder == 1  
    /* 初始化 EventRecorder 并开启 */  
    EventRecorderInitialize(EventRecordAll, 1U);  
    EventRecorderStart();  
#endif  
  
    bsp_InitKey(); /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */  
    bsp_InitTimer(); /* 初始化滴答定时器 */  
    bsp_InitUart(); /* 初始化串口 */  
    bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */  
    bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();
```



```
/* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。



- 按下按键 K1, DSP 求最大值。
- 按下按键 K2, DSP 求最小值。
- 按下按键 K3, DSP 求平均值。
- 按下摇杆 OK 键, DSP 求功率。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下, 求最大值 */
                    DSP_Max();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下, 求小值 */
                    DSP_Min();
                    break;

                case KEY_DOWN_K3:           /* K3 键按下, 求平方根 */
                    DSP_Mean();
                    break;

                case JOY_DOWN_OK:          /* 摆杆上键, 求功率 */
                    DSP_Power();
                    break;
            }
        }
    }
}
```



```
DSP_Power();  
break;  
  
default:  
/* 其他的键值不处理 */  
break;  
}  
}  
}
```

14.8 实验例程说明 (IAR)

配套例子：

V7-209_DSP 统计运算 (最大值, 最小值, 平均值和功率)

实验目的：

1. 学习 DSP 快速运算 (三角函数和平方根)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1, DSP 求最大值。
3. 按下按键 K2, DSP 求最小值。
4. 按下按键 K3, DSP 求平均值。
5. 按下摇杆 OK 键, DSP 求功率。

使用 AC6 注意事项

特别注意附件章节 C 的问题

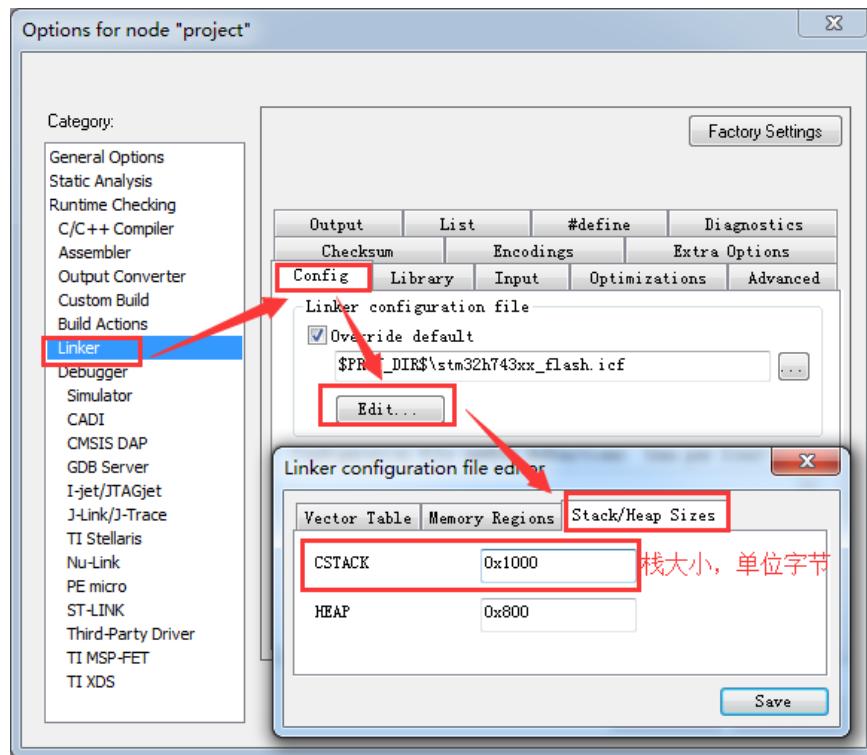
上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

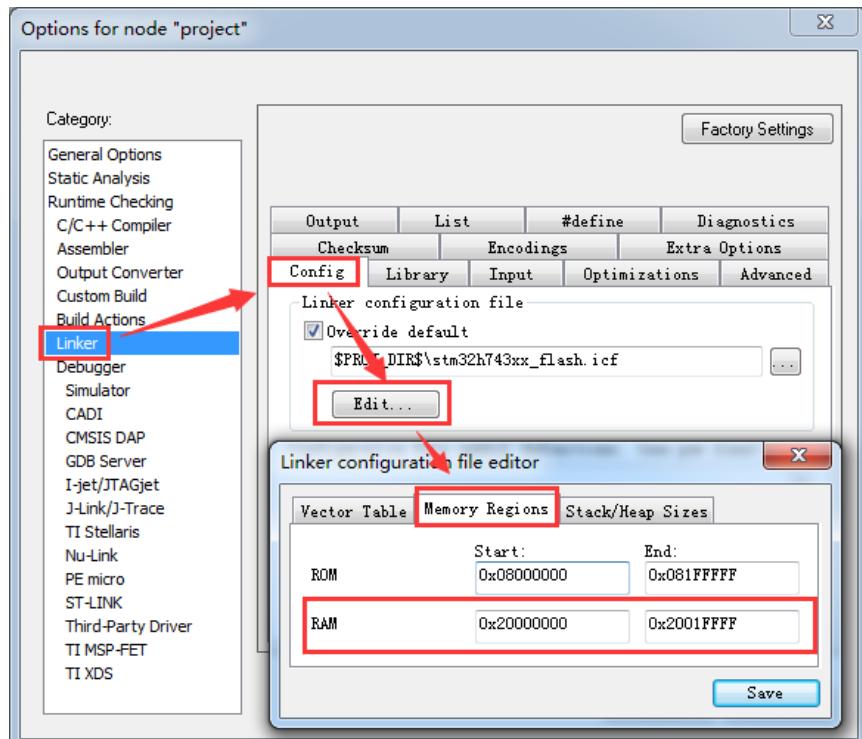
详见本章的 3.5 4.5, 5.5 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
*****
```



```
/* 功能说明：初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形    参：无
* 返回值：无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
        - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
        - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名：MPU_Config
* 功能说明：配置 MPU
* 形    参：无
* 返回值：无
*****
*/
static void MPU_Config( void )
```



```
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /*使能 MPU */  
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);  
}  
  
/*  
*****  
* 函数名: CPU_CACHE_Enable  
* 功能说明: 使能 L1 Cache  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void CPU_CACHE_Enable(void)  
{  
    /* 使能 I-Cache */  
    SCB_EnableICache();  
  
    /* 使能 D-Cache */  
    SCB_EnableDCache();  
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1, DSP 求最大值。
- 按下按键 K2, DSP 求最小值。
- 按下按键 K3, DSP 求平均值。
- 按下摇杆 OK 键, DSP 求功率。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下, 求最大值 */
                    DSP_Max();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下, 求小值 */
                    DSP_Min();
                    break;
            }
        }
    }
}
```



```
case KEY_DOWN_K3:           /* K3 键按下, 求平方根 */
    DSP_Mean();
    break;

case JOY_DOWN_OK:          /* 摆杆上键, 求功率 */
    DSP_Power();
    break;

default:
    /* 其他的键值不处理 */
    break;
}
}

}
```

14.9 总结

本期教程就跟大家讲这么多，有兴趣的可以深入研究这些函数源码的实现。



第15章 DSP 统计函数-标准偏差、均方根和方差

本期教程主要讲解统计函数中的标准偏差，均方根和方差的计算。

15.1 初学者重要提示

15.2 DSP 基础运算指令

15.3 标准偏差 (Standard Deviation)

15.4 均方根 (RMS)

15.5 方差 (Variance)

15.7 实验例程说明 (MDK)

15.8 实验例程说明 (IAR)

15.9 总结

15.1 初学者重要提示

- ◆ 特别注意本章 13.5.2 小节的问题，定点数求解平方根，本章节几个函数的源码都有调用到求平方根。
- ◆ 正确理解 RMS 均方根（重要，推荐必读）：
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=95470>。

15.2 DSP 基础运算指令

本章用到的 DSP 指令在前面章节都已经讲解过。

15.3 标准偏差 (Standard deviation)

这部分函数用于计算标准偏差，公式描述如下：

Result = sqrt((sumOfSquares - sum^2 / blockSize) / (blockSize - 1))

其中：

sumOfSquares = pSrc[0] * pSrc[0] + pSrc[1] * pSrc[1] + ... + pSrc[blockSize-1] *
pSrc[blockSize-1]

sum = pSrc[0] + pSrc[1] + pSrc[2] + ... + pSrc[blockSize-1]



15.3.1 函数 arm_std_f32

函数原型:

```
void arm_std_f32(
    const float32_t * pSrc,
    uint32_t blockSize,
    float32_t * pResult)
```

函数描述:

这个函数用于求 32 位浮点数的标准偏差。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出的标准偏差。

15.3.2 函数 arm_std_q31

函数原型:

```
void arm_std_q31(
    const q31_t * pSrc,
    uint32_t blockSize,
    q31_t * pResult)
```

函数描述:

这个函数用于求 32 位定点数的标准偏差。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出的标准偏差。

注意事项:

输入参数是 1.31 格式的，相乘后输出就是 $1.31 \times 1.31 = 2.62$ 格式，这种情况下，函数内部使用的 64 位累加器很容易溢出，并且这个函数不支持饱和运算。

15.3.3 函数 arm_std_q15

函数原型:

```
void arm_std_q15(
    const q31_t * pSrc,
    uint32_t blockSize,
    q31_t * pResult)
```

函数描述:

这个函数用于求 15 位定点数的标准偏差。

函数参数:

- ◆ 第 1 个参数源数据地址。



- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出的标准偏差。

注意事项：

输入参数是 1.15 格式，相乘后的结果就是 $1.15 \times 1.15 = 2.30$ 格式，这种情况下，内部 64 位累加器的格式就是 34.30。最终的输出结果要截取到低 15 位数据，然后通过饱和运算最终输出数据格式 1.15。

15.3.4 使用举例

程序设计：

```
/*
***** DSP_Std *****
* 函数名: DSP_Std
* 功能说明: 求标准偏差
* 形参: 无
* 返回值: 无
***** DSP_Std *****
*/
static void DSP_Std(void)
{
    float32_t pSrc[10] = {0.6557f, 0.0357f, 0.8491f, 0.9340f, 0.6787f, 0.7577f, 0.7431f, 0.3922f,
                          0.6555f, 0.1712f};
    float32_t pResult;
    uint32_t pIndex;

    q31_t pSrc1[10];
    q31_t pResult1;

    q15_t pSrc2[10];
    q15_t pResult2;

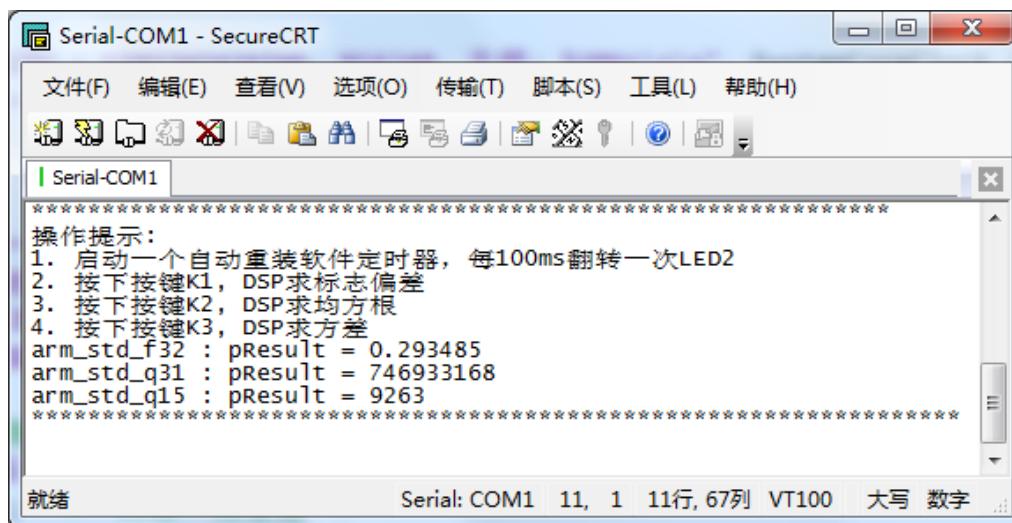
    arm_std_f32(pSrc, 10, &pResult);
    printf("arm_std_f32 : pResult = %f\r\n", pResult);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc1[pIndex] = rand();
    }
    arm_std_q31(pSrc1, 10, &pResult1);
    printf("arm_std_q31 : pResult = %d\r\n", pResult1);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc2[pIndex] = rand()%32768;
    }
    arm_std_q15(pSrc2, 10, &pResult2);
    printf("arm_std_q15 : pResult = %d\r\n", pResult2);

    printf("*****\r\n");
    */
}
```

实验现象：



15.4 均方根 (RMS)

这部分函数用于计算标准偏差，公式描述如下：

```
Result = sqrt(((pSrc[0] * pSrc[0] + pSrc[1] * pSrc[1] + ... + pSrc[blockSize-1] *  
                pSrc[blockSize-1]) / blockSize));
```

15.4.1 函数 arm_rms_f32

函数原型：

```
void arm_rms_f32(  
    const float32_t * pSrc,  
    uint32_t blockSize,  
    float32_t * pResult)
```

函数描述：

这个函数用于求 32 位浮点数的均方根。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的均方根。

15.4.2 函数 arm_rms_q31

函数原型：

```
void arm_rms_q31(  
    const q31_t * pSrc,  
    uint32_t blockSize,  
    q31_t * pResult)
```

函数描述：

这个函数用于求 32 位定点数的均方根。

函数参数：



- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的均方根。

注意事项：

输入参数是 1.31 格式的，相乘后输出就是 $1.31 * 1.31 = 2.62$ 格式，这种情况下，函数内部使用的 64 位累加器很容易溢出，并且这个函数不支持饱和运算。

15.4.3 函数 arm_rms_q15

函数原型：

```
void arm_rms_q15(
    const q15_t * pSrc,
    uint32_t blockSize,
    q15_t * pResult)
```

函数描述：

这个函数用于求 16 位定点数的均方根。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的均方根。

注意事项：

输入参数是 1.15 格式，相乘后的结果就是 $1.15 * 1.15 = 2.30$ 格式，这种情况下，内部 64 位累加器的格式就是 34.30。最终的输出结果要截取到低 15 位数据，然后通过饱和运算最终输出数据格式 1.15。

15.4.4 使用举例

程序设计：

```
/*
*****
*  函数名: DSP_RMS
*  功能说明: 求均方根
*  形  参: 无
*  返  值: 无
*****
*/
static void DSP_RMS(void)
{
    float32_t pSrc[10] = {0.7060f, 0.0318f, 0.2769f, 0.0462f, 0.0971f, 0.8235f, 0.6948f, 0.3171f,
                          0.9502f, 0.0344f};
    float32_t pResult;
    uint32_t pIndex;

    q31_t pSrc1[10];
    q31_t pResult1;

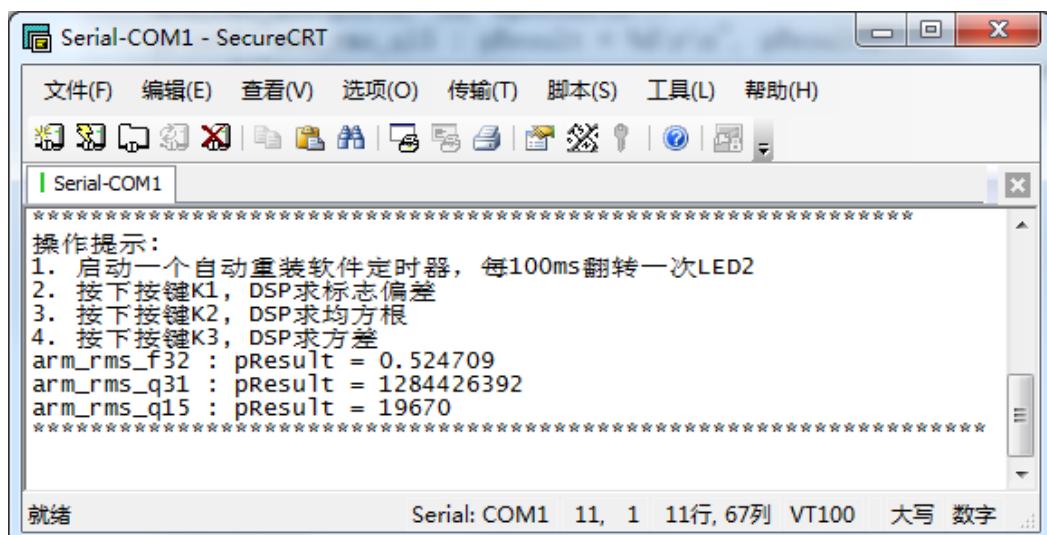
    q15_t pSrc2[10];
    q15_t pResult2;

    arm_rms_f32(pSrc, 10, &pResult);
    printf("arm_rms_f32 : pResult = %f\r\n", pResult);
```

```
/*
for(pIndex = 0; pIndex < 10; pIndex++)
{
    pSrc1[pIndex] = rand();
}
arm_rms_q31(pSrc1, 10, &pResult1);
printf("arm_rms_q31 : pResult = %d\r\n", pResult1);

/*
for(pIndex = 0; pIndex < 10; pIndex++)
{
    pSrc2[pIndex] = rand()%32768;
}
arm_rms_q15(pSrc2, 10, &pResult2);
printf("arm_rms_q15 : pResult = %d\r\n", pResult2);
printf("*****\r\n");
}
```

实验现象：



15.5 方差 (Variance)

这部分函数用于计算标准偏差，公式描述如下：

```
Result = sqrt(((pSrc[0] * pSrc[0] + pSrc[1] * pSrc[1] + ... + pSrc[blockSize-1] *
                pSrc[blockSize-1]) / blockSize));
```

15.5.1 函数 arm_var_f32

函数原型：

```
void arm_var_f32(
    const float32_t * pSrc,
    uint32_t blockSize,
    float32_t * pResult)
```

函数描述：

这个函数用于求 32 位浮点数的方差。

函数参数：

- ◆ 第 1 个参数源数据地址。



- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是求解出来的方差。

15.5.2 函数 arm_var_q31

函数原型:

```
void arm_var_q31(
    const q31_t * pSrc,
    uint32_t blockSize,
    q31_t * pResult)
```

函数描述:

用于求 32 位定点数的方差。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是计算出来的方差。

注意事项:

输入参数是 1.31 格式的，相乘后输出就是 $1.31 \times 1.31 = 2.62$ 格式，这种情况下，函数内部使用的 64 位累加器很容易溢出，并且这个函数不支持饱和运算

15.5.3 函数 arm_var_q15

函数原型:

```
void arm_var_q15(
    const q15_t * pSrc,
    uint32_t blockSize,
    q15_t * pResult)
```

函数描述:

用于求 16 位定点数的方差。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是源数据个数。
- ◆ 第 3 个参数是计算出来的方差结果。

注意事项:

输入参数是 1.15 格式，相乘后的结果就是 $1.15 \times 1.15 = 2.30$ 格式，这种情况下，内部 64 位累加器的格式就是 34.30。最终的输出结果要截取到低 15 位数据，然后通过饱和运算最终输出数据格式 1.15。

15.5.4 使用举例

程序设计:

```
/*
***** DSP_Var.c *****
* 函数名: DSP_Var
* 功能说明: 求方差
```

```
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Var(void)
{
    float32_t pSrc[10] = {0.4387f, 0.3816f, 0.7655f, 0.7952f, 0.1869f, 0.4898f, 0.4456f, 0.6463f,
                          0.7094f, 0.7547f};
    float32_t pResult;
    uint32_t pIndex;

    q31_t pSrc1[10];
    q31_t pResult1;

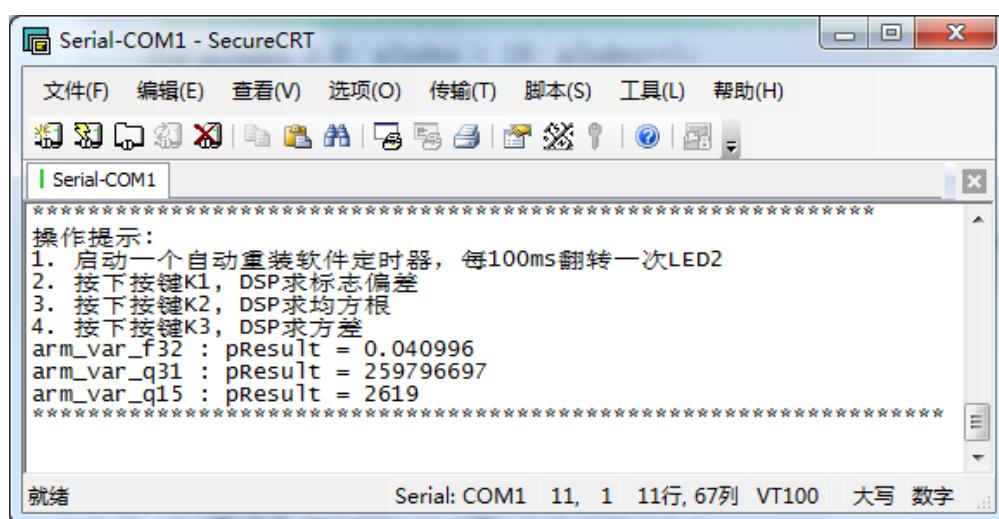
    q15_t pSrc2[10];
    q15_t pResult2;

    arm_var_f32(pSrc, 10, &pResult);
    printf("arm_var_f32 : pResult = %f\r\n", pResult);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc1[pIndex] = rand();
    }
    arm_var_q31(pSrc1, 10, &pResult1);
    printf("arm_var_q31 : pResult = %d\r\n", pResult1);

    /*****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc2[pIndex] = rand()%32768;
    }
    arm_var_q15(pSrc2, 10, &pResult2);
    printf("arm_var_q15 : pResult = %d\r\n", pResult2);
    printf("*****\r\n");
    */
}
```

实验现象：



15.6 Matlab 求标准偏差，均方差和方差

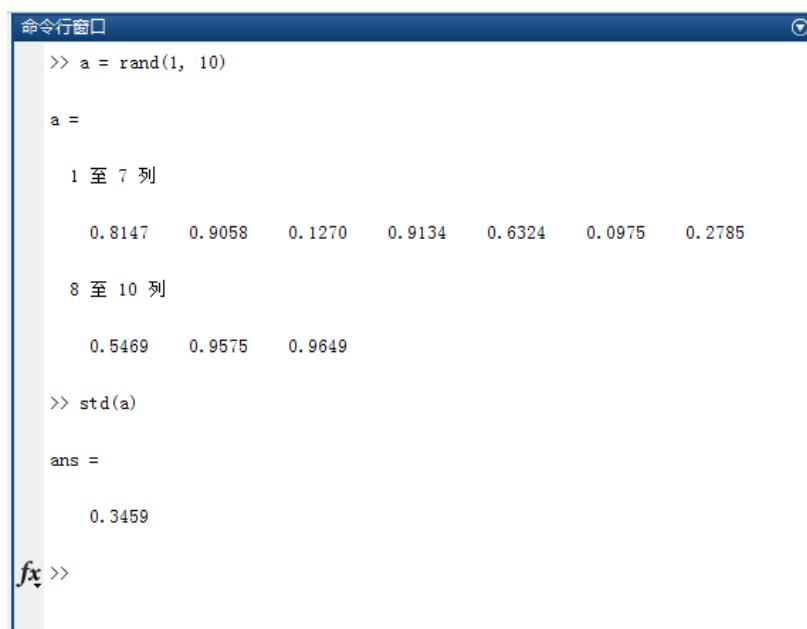
15.6.1 Matlab 求标准偏差

在 matlab 的命令窗口输入如下命令:

```
a = rand(1, 10) %1行10列
```

然后再通过命令 std 获得标准偏差:

```
std(a)
```



The screenshot shows the MATLAB Command Window with the following text:
>> a = rand(1, 10)
a =
1 至 7 列
0.8147 0.9058 0.1270 0.9134 0.6324 0.0975 0.2785
8 至 10 列
0.5469 0.9575 0.9649
>> std(a)
ans =
0.3459
fx >>

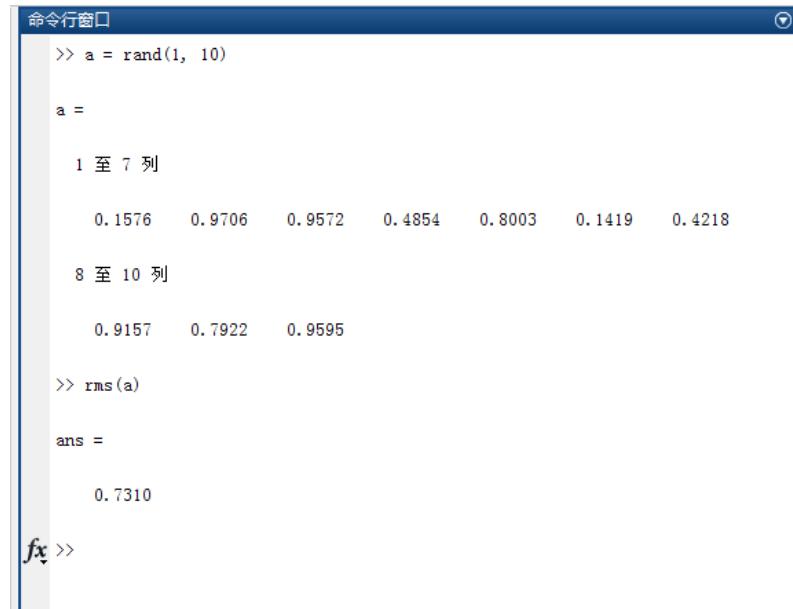
15.6.2 Matlab 求均方根

在 matlab 的命令窗口输入如下命令:

```
a = rand(1, 10) %1行10列
```

然后再通过命令 rms 获得均方根。

```
rms(a)
```



```
命令行窗口
>> a = rand(1, 10)

a =

    1 至 7 列

    0.1576    0.9706    0.9572    0.4854    0.8003    0.1419    0.4218

    8 至 10 列

    0.9157    0.7922    0.9595

>> rms(a)

ans =

    0.7310

fx >>
```

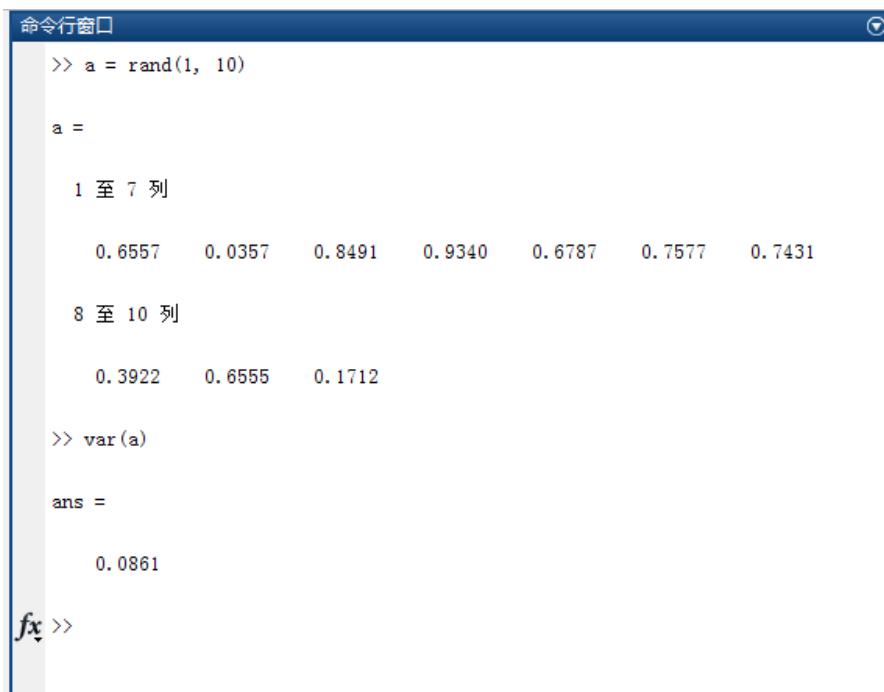
15.6.3 Matlab 求方差

在 matlab 的命令窗口输入如下命令:

```
a = rand(1, 10) %1行10列
```

然后再通过命令 var 获得方差。

```
var(a)
```



```
命令行窗口
>> a = rand(1, 10)

a =

    1 至 7 列

    0.6557    0.0357    0.8491    0.9340    0.6787    0.7577    0.7431

    8 至 10 列

    0.3922    0.6555    0.1712

>> var(a)

ans =

    0.0861

fx >>
```

15.7 实验例程说明 (MDK)

配套例子：

V7-210_DSP 统计运算 (标准偏差, 均方根和方差)

实验目的：

1. 学习统计运算 (标准偏差, 均方根和方差)

实验内容：

1. 启动一个自动重装软件定时器, 每 100ms 翻转一次 LED2。
2. 按下按键 K1, DSP 求标准偏差。
3. 按下按键 K2, DSP 求均方根。
4. 按下按键 K3, DSP 求方差。

使用 AC6 注意事项

特别注意附件章节 C 的问题

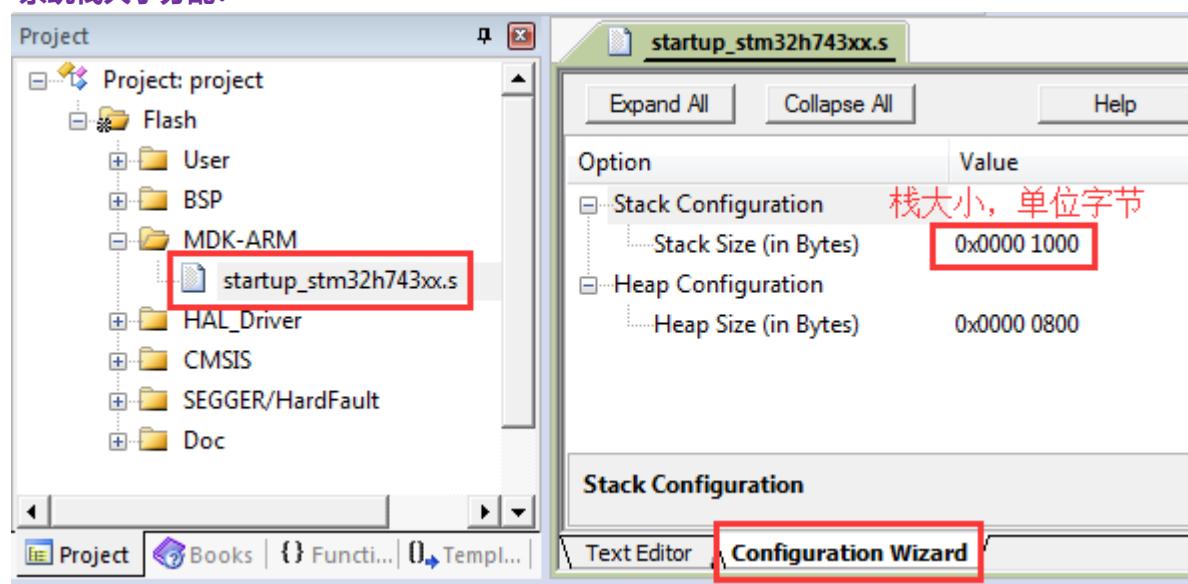
上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

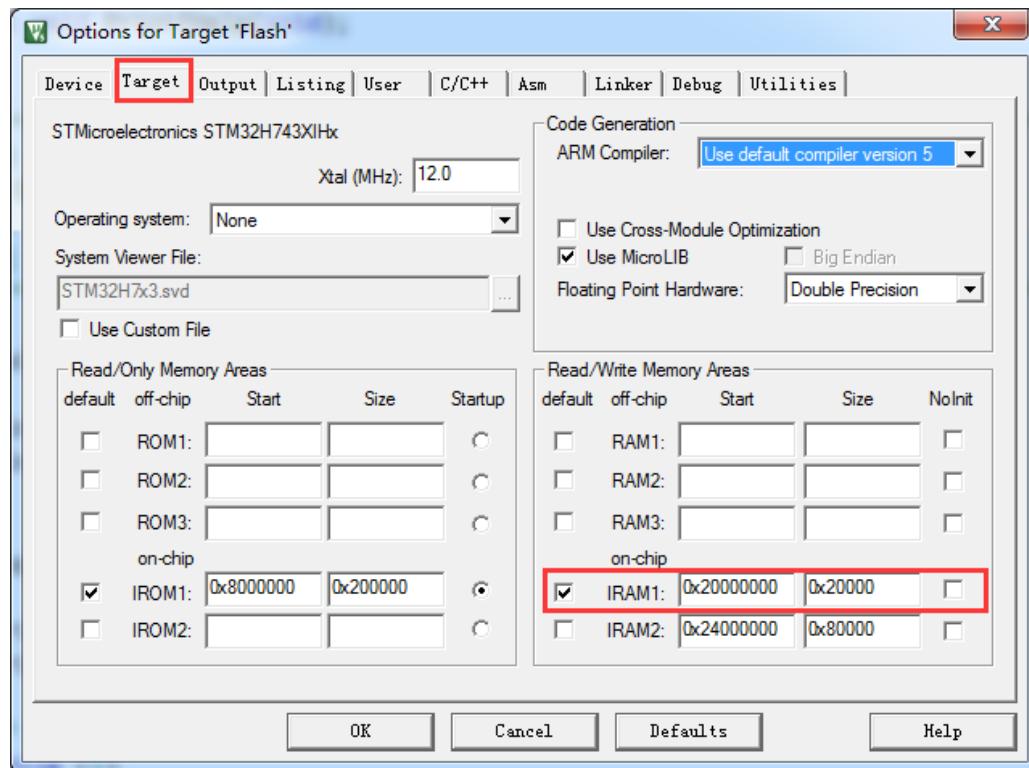
详见本章的 3.5 4.5, 5.5 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1, DSP 求标准偏差。
- 按下按键 K2, DSP 求均方根。
- 按下按键 K3, DSP 求方差。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */
}
```



```
PrintfHelp(); /* 打印操作提示信息 */
```

```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下，求标准偏差 */
                DSP_Std();
                break;

            case KEY_DOWN_K2:          /* K2 键按下，求均方根 */
                DSP_RMS();
                break;

            case KEY_DOWN_K3:          /* K3 键按下，求方差 */
                DSP_Var();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}
```

15.8 实验例程说明 (IAR)

配套例子：

V7-210_DSP 统计运算 (标准偏差, 均方根和方差)

实验目的：

1. 学习统计运算 (标准偏差, 均方根和方差)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1, DSP 求标准偏差。

3. 按下按键 K2, DSP 求均方根。
4. 按下按键 K3, DSP 求方差。

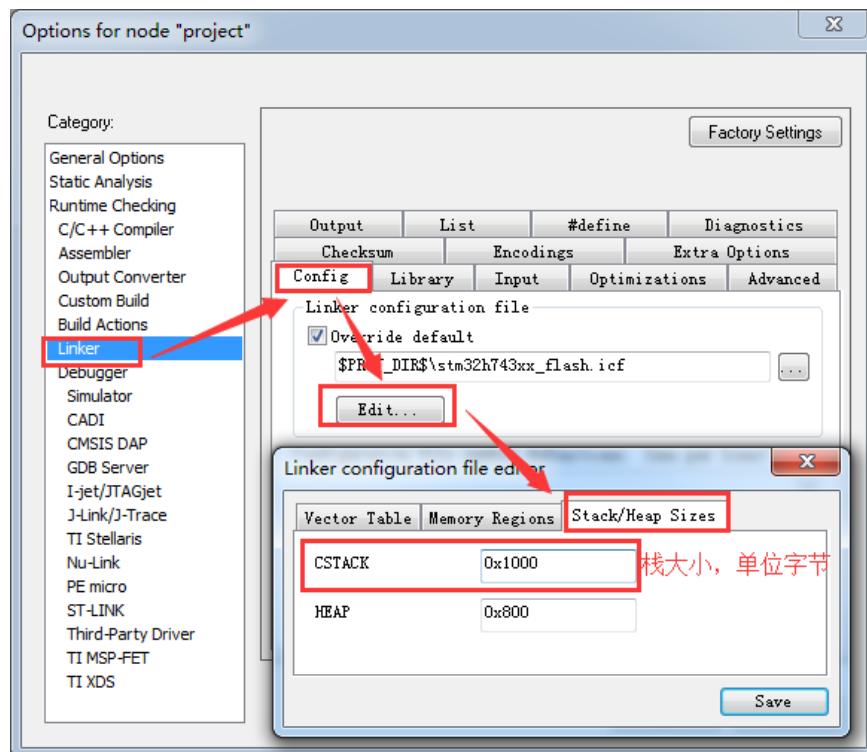
上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

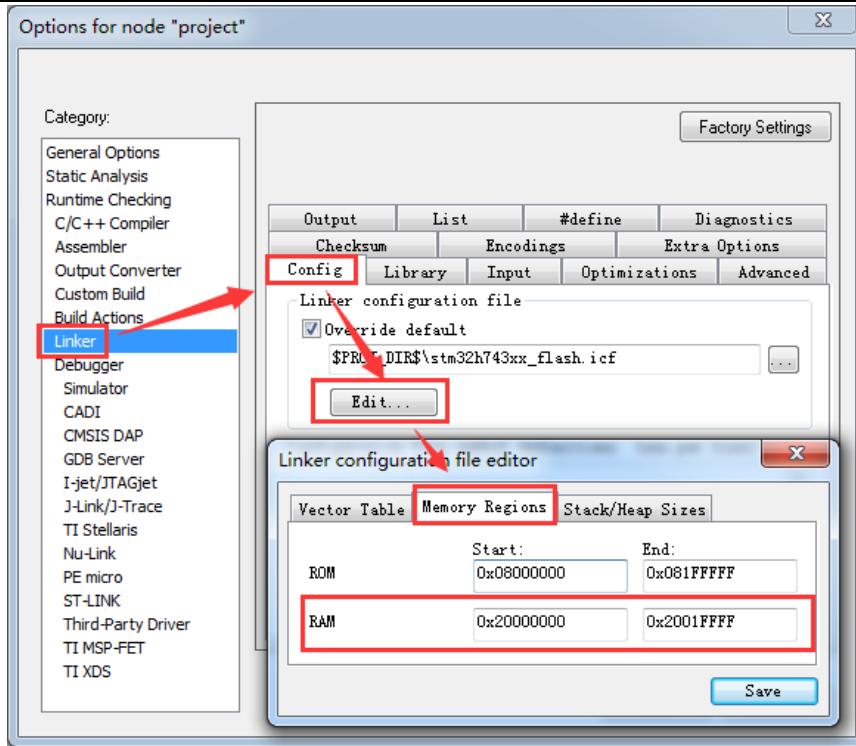
详见本章的 3.5 4.5, 5.5 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作:

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1, DSP 求标准偏差。
- 按下按键 K2, DSP 求均方根。
- 按下按键 K3, DSP 求方差。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ukeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下，求标准偏差 */
                DSP_Std();
                break;

            case KEY_DOWN_K2:          /* K2 键按下，求均方根 */
                DSP_RMS();
                break;

            case KEY_DOWN_K3:          /* K3 键按下，求方差 */
                DSP_Var();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}

}
```

15.9 总结

本期教程就跟大家讲这么多，有兴趣的可以深入研究这些函数源码的实现。



第16章 DSP 功能函数-数据拷贝，数据填充和浮点转定点

点转定点

本期教程主要讲解功能函数中的数据拷贝，数据填充和浮点数转换为定点数。

16.1 初学者重要提示

- 16.2 DSP 基础运算指令
- 16.3 数据拷贝 (Copy)
- 16.4 数据填充 (Fill)
- 16.5 浮点数转定点数 (Float to Fix)
- 16.6 总结

16.1 初学者重要提示

- ◆ 浮点数的四舍五入处理：
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=95149>。
- ◆ C 库的浮点数四舍五入函数 round, roundf, round 使用说明：
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=95156>。

16.2 DSP 基础运算指令

本章用到的 DSP 指令在前面章节都已经讲解过。

16.3 数据拷贝 (copy)

这部分函数用于数据拷贝，公式描述如下：

$pDst[n] = pSrc[n]; \quad 0 \leq n < blockSize$

16.3.1 函数 arm_copy_f32

函数原型：

```
void arm_copy_f32(  
    const float32_t * pSrc,  
    float32_t * pDst,  
    uint32_t blockSize)
```

**函数描述：**

这个函数用于 32 位浮点数的复制。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是目的数据地址。
- ◆ 第 3 个参数是复制的个数。

16.3.2 函数 arm_copy_q31

函数原型：

```
void arm_copy_q31(
    const q31_t * pSrc,
    q31_t * pDst,
    uint32_t blockSize)
```

函数描述：

这个函数用于 32 位定点数的复制。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是目的数据地址。
- ◆ 第 3 个参数是复制的个数。

16.3.3 函数 arm_copy_q15

函数原型：

```
void arm_copy_q15(
    const q15_t * pSrc,
    q15_t * pDst,
    uint32_t blockSize)
```

函数描述：

这个函数用于 16 位定点数的复制。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是目的数据地址。
- ◆ 第 3 个参数是复制的个数。

16.3.4 函数 arm_copy_q7

函数原型：

```
void arm_copy_q7(
    const q7_t * pSrc,
    q7_t * pDst,
    uint32_t blockSize)
```

函数描述：



这个函数用于 8 位定点数的复制。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是目的数据地址。
- ◆ 第 3 个参数是复制的个数。

16.3.5 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_Copy
* 功能说明: 数据拷贝
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Copy(void)
{
    float32_t pSrc[10] = {0.6557, 0.0357, 0.8491, 0.9340, 0.6787, 0.7577, 0.7431, 0.3922, 0.6555,
0.1712};
    float32_t pDst[10];
    uint32_t pIndex;

    q31_t pSrc1[10];
    q31_t pDst1[10];

    q15_t pSrc2[10];
    q15_t pDst2[10];

    q7_t pSrc3[10];
    q7_t pDst3[10];

    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("pSrc[%d] = %f\r\n", pIndex, pSrc[pIndex]);
    }
    arm_copy_f32(pSrc, pDst, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_copy_f32: pDst[%d] = %f\r\n", pIndex, pDst[pIndex]);
    }

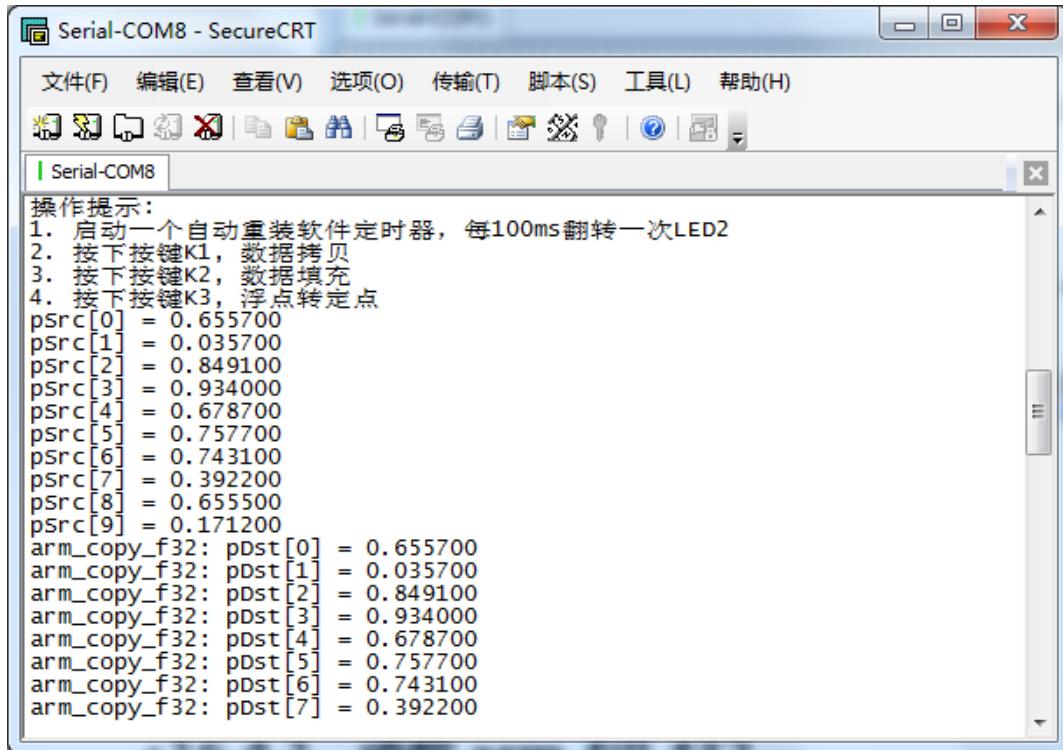
    ****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc1[pIndex] = rand();
        printf("pSrc1[%d] = %d\r\n", pIndex, pSrc1[pIndex]);
    }
    arm_copy_q31(pSrc1, pDst1, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_copy_q31: pDst1[%d] = %d\r\n", pIndex, pDst1[pIndex]);
    }

    ****
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc2[pIndex] = rand()%32768;
        printf("pSrc2[%d] = %d\r\n", pIndex, pSrc2[pIndex]);
    }
    arm_copy_q15(pSrc2, pDst2, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
```



```
printf("arm_copy_q15: pDst2[%d] = %d\r\n", pIndex, pDst2[pIndex]);  
}/******  
for(pIndex = 0; pIndex < 10; pIndex++)  
{  
    pSrc3[pIndex] = rand()%128;  
    printf("pSrc3[%d] = %d\r\n", pIndex, pSrc3[pIndex]);  
}  
arm_copy_q7(pSrc3, pDst3, 10);  
for(pIndex = 0; pIndex < 10; pIndex++)  
{  
    printf("arm_copy_q7: pDst3[%d] = %d\r\n", pIndex, pDst3[pIndex]);  
}  
*****\nprintf("*****\r\n");  
}
```

实验现象 (部分截图):



16.4 数据填充 (Fill)

这部分函数用于数据填充，公式描述如下：

$pDst[n] = value; \quad 0 \leq n < blockSize$

16.4.1 函数 arm_fill_f32

函数原型:

```
void arm_fill_f32(  
    float32_t value,  
    float32_t * pDst,  
    uint32_t blockSize)
```

函数描述:

这个函数用于填充 32 位浮点数。

**函数参数:**

- ◆ 第 1 个参数是要填充的数值。
- ◆ 第 2 个参数是要填充的数据地址。
- ◆ 第 3 个参数是要填充的数据个数。

16.4.2 函数 arm_fill_q31

函数原型:

```
void arm_fill_q31(  
    q31_t value,  
    q31_t * pDst,  
    uint32_t blockSize)
```

函数描述:

这个函数用于填充 32 位定点数。

函数参数:

- ◆ 第 1 个参数是要填充的数值。
- ◆ 第 2 个参数是要填充的数据地址。
- ◆ 第 3 个参数是要填充的数据个数。

16.4.3 函数 arm_fill_q15

函数原型:

```
void arm_fill_q15(  
    q15_t value,  
    q15_t * pDst,  
    uint32_t blockSize)
```

函数描述:

这个函数用于填充 16 位定点数。

函数参数:

- ◆ 第 1 个参数是要填充的数值。
- ◆ 第 2 个参数是要填充的数据地址。
- ◆ 第 3 个参数是要填充的数据个数。

16.4.4 函数 arm_fill_q7

函数原型:

```
void arm_fill_q7(  
    q7_t value,  
    q7_t * pDst,  
    uint32_t blockSize)
```

函数描述:

这个函数用于填充 8 位定点数。

函数参数:



- ◆ 第1个参数是要填充的数值。
- ◆ 第2个参数是要填充的数据地址。
- ◆ 第3个参数是要填充的数据个数。

16.4.5 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_Fill
* 功能说明: 数据填充
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Fill(void)
{
    float32_t pDst[10];
    uint32_t pIndex;
    q31_t pDst1[10];
    q15_t pDst2[10];
    q7_t pDst3[10];

    arm_fill_f32(3.33f, pDst, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_fill_f32: pDst[%d] = %f\r\n", pIndex, pDst[pIndex]);
    }

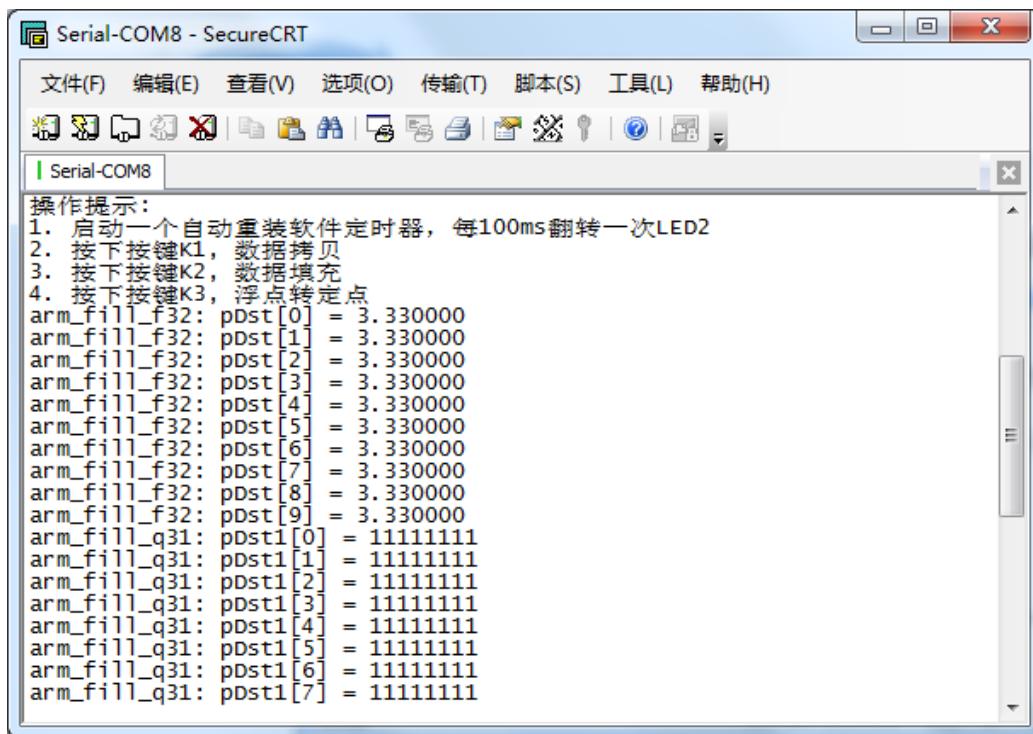
    *****
    arm_fill_q31(0x11111111, pDst1, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_fill_q31: pDst1[%d] = %x\r\n", pIndex, pDst1[pIndex]);
    }

    *****
    arm_fill_q15(0x1111, pDst2, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_fill_q15: pDst2[%d] = %d\r\n", pIndex, pDst2[pIndex]);
    }

    *****
    arm_fill_q7(0x11, pDst3, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_fill_q7: pDst3[%d] = %d\r\n", pIndex, pDst3[pIndex]);
    }

    *****
    printf("*****\r\n");
}
```

实验现象：



16.5 浮点数转定点数 (Float to Fix)

浮点数转 Q31 公式描述:

$$pDst[n] = (q31_t)(pSrc[n] * 2147483648); \quad 0 \leq n < blockSize.$$

浮点数转 Q15 公式描述:

$$pDst[n] = (q15_t)(pSrc[n] * 32768); \quad 0 \leq n < blockSize$$

浮点数转 Q7 公式描述:

$$pDst[n] = (q7_t)(pSrc[n] * 128); \quad 0 \leq n < blockSize$$

16.5.1 函数 arm_float_to_q31

函数原型:

```
void arm_float_to_q31(
    const float32_t * pSrc,
    q31_t * pDst,
    uint32_t blockSize)
```

函数描述:

这个函数用于将浮点数转换为 32 位定点数。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数是转换的次数。

注意事项:



- ◆ 这个函数使用了饱和运算。
- ◆ 输出结果的范围是[0x80000000 0x7FFFFFFF]。

16.5.2 函数 arm_float_to_q15

函数原型:

```
void arm_var_q31(
    const q31_t * pSrc,
    uint32_t blockSize,
    q31_t * pResult)
```

函数描述:

这个函数用于将浮点数转换为 16 位定点数。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数是转换的次数。

注意事项:

- ◆ 这个函数使用了饱和运算。
- ◆ 输出结果的范围是[0x8000 0x7FFF]。

16.5.3 函数 arm_float_to_q7

函数原型:

```
void arm_float_to_q7(
    const float32_t * pSrc,
    q7_t * pDst,
    uint32_t blockSize)
```

函数描述:

这个函数用于将浮点数转换为 8 位定点数。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数是转换的次数。

注意事项:

- ◆ 这个函数使用了饱和运算。
- ◆ 输出结果的范围是[0x80 0x7F]。

16.5.4 使用举例

程序设计:

```
/*
***** DSP_FloatToFix *****
*  功能说明：浮点数转定点数
```

```
* 形 参: 无
* 返回值: 无
*****
*/
static void DSP_FloatToFix(void)
{
    float32_t pSrc[10] = {0.6557, 0.0357, 0.8491, 0.9340, 0.6787, 0.7577, 0.7431, 0.3922, 0.6555,
                          0.1712};
    uint32_t pIndex;
    q31_t pDst1[10];
    q15_t pDst2[10];
    q7_t pDst3[10];

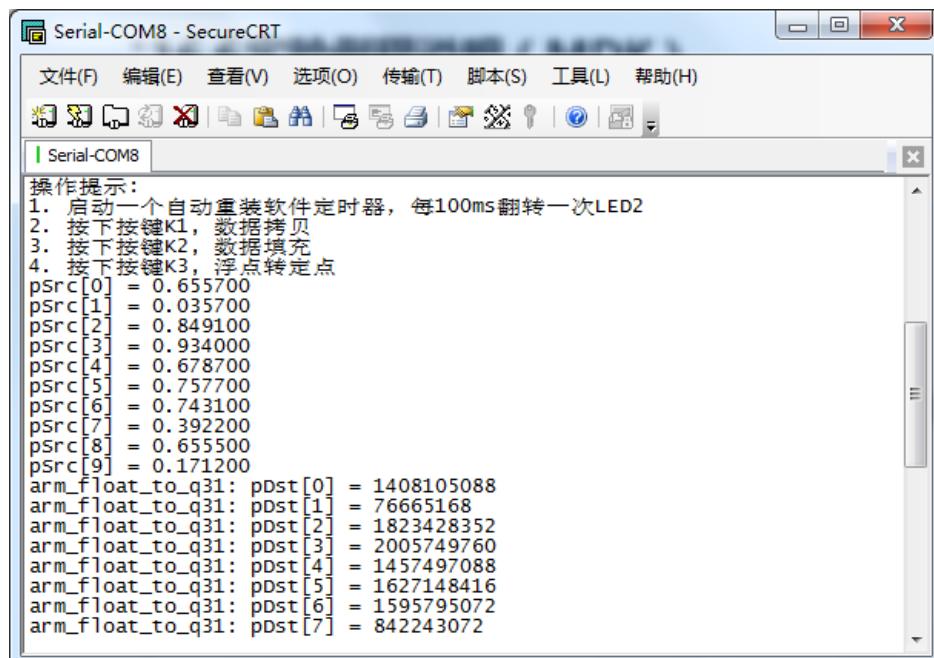
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("pSrc[%d] = %f\r\n", pIndex, pSrc[pIndex]);
    }

    /*****
    arm_float_to_q31(pSrc, pDst1, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_float_to_q31: pDst[%d] = %d\r\n", pIndex, pDst1[pIndex]);
    }

    ****/
    arm_float_to_q15(pSrc, pDst2, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_float_to_q15: pDst1[%d] = %d\r\n", pIndex, pDst2[pIndex]);
    }

    /*****
    arm_float_to_q7(pSrc, pDst3, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_float_to_q7: pDst2[%d] = %d\r\n", pIndex, pDst3[pIndex]);
    }
    ****/
    printf("*****\r\n");
}
```

实验现象：



16.6 实验例程说明 (MDK)

配套例子：

V7-211_DSP 功能函数 (数据拷贝, 数据填充和浮点转定点)

实验目的：

1. 学习功能函数 (数据拷贝, 数据填充和浮点转定点)

实验内容：

1. 启动一个自动重装软件定时器, 每 100ms 翻转一次 LED2。
2. 按下按键 K1, 串口打印函数 DSP_Copy 的输出结果。
3. 按下按键 K2, 串口打印函数 DSP_Fill 的输出结果。
4. 按下按键 K3, 串口打印函数 DSP_FloatToFix 的输出结果。

使用 AC6 注意事项

特别注意附件章节 C 的问题

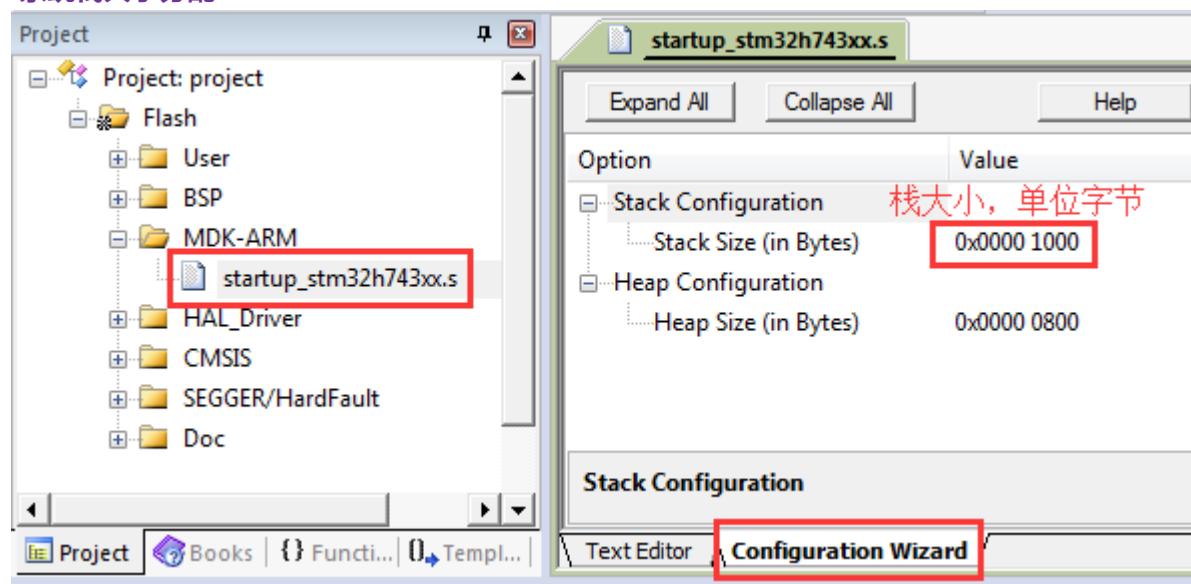
上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

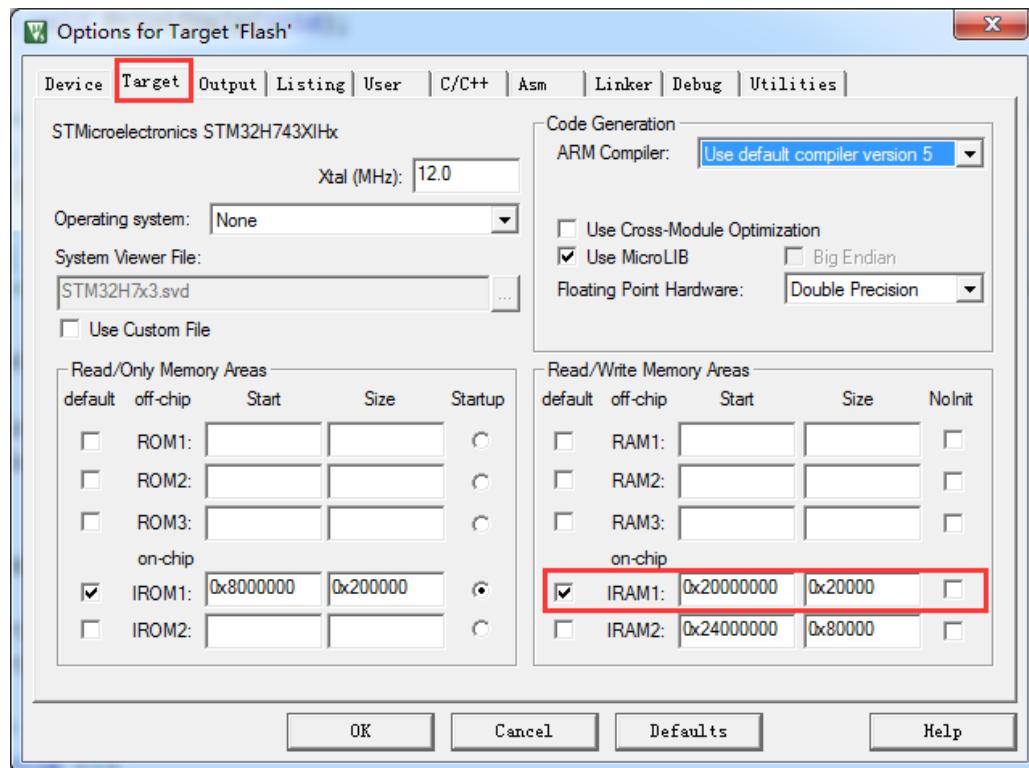
详见本章的 3.5 4.5, 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印函数 DSP_Copy 的输出结果
- 按下按键 K2，串口打印函数 DSP_Fill 的输出结果
- 按下按键 K3，串口打印函数 DSP_FloatToFix 的输出结果

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */
}
```



```
PrintfHelp(); /* 打印操作提示信息 */
```

```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:           /* K1 键按下，数据复制 */
                DSP_Copy();
                break;

            case KEY_DOWN_K2:           /* K2 键按下，数据填充 */
                DSP_Fill();
                break;

            case KEY_DOWN_K3:           /* K3 键按下，浮点转定点 */
                DSP_FloatToFix();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}
```

16.7 实验例程说明 (IAR)

配套例子：

V7-211_DSP 功能函数 (数据拷贝, 数据填充和浮点转定点)

实验目的：

1. 学习功能函数 (数据拷贝, 数据填充和浮点转定点)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1, 串口打印函数 DSP_Copy 的输出结果。

3. 按下按键 K2, 串口打印函数 DSP_Fill 的输出结果。
4. 按下按键 K3, 串口打印函数 DSP_FloatToFix 的输出结果。

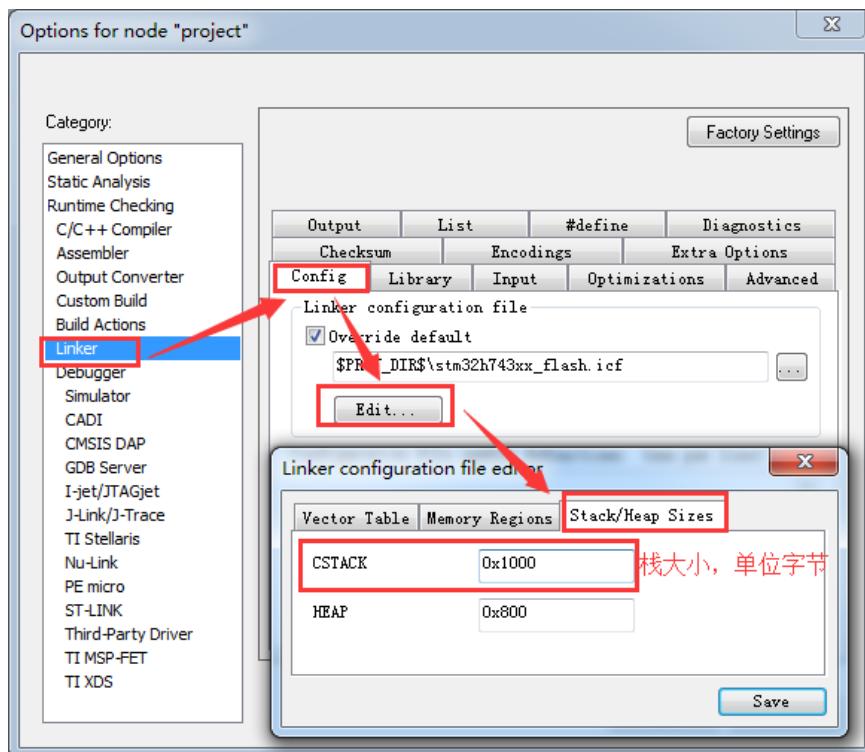
上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

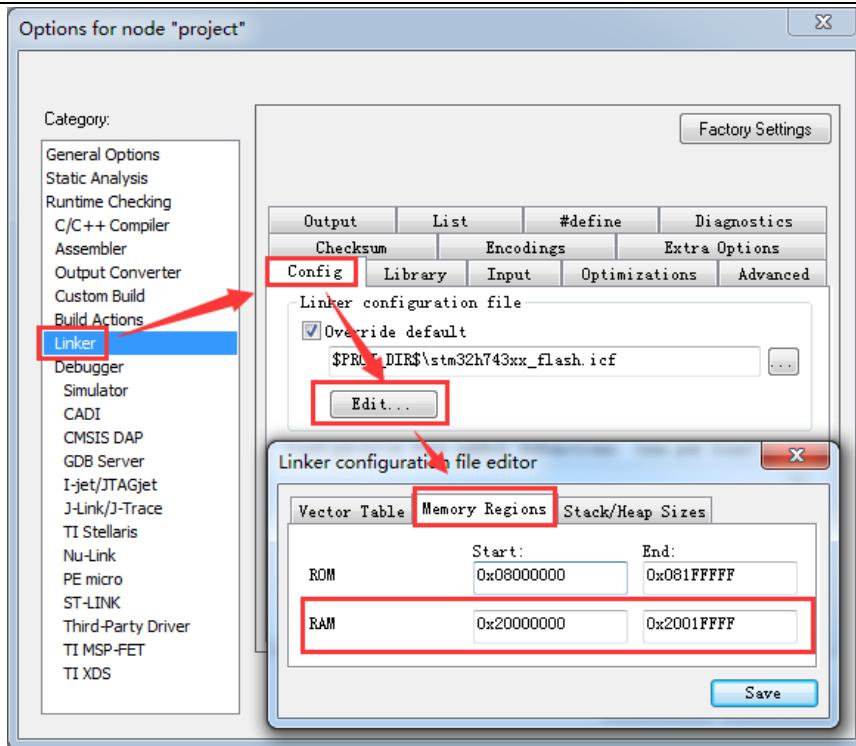
详见本章的 3.5 4.5, 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印函数 DSP_Copy 的输出结果
- 按下按键 K2，串口打印函数 DSP_Fill 的输出结果
- 按下按键 K3，串口打印函数 DSP_FloatToFix 的输出结果

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */\n\n/* 进入主程序循环体 */\nwhile (1)\n{\n    bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */\n\n    /* 判断定时器超时时间 */\n    if (bsp_CheckTimer(0))\n    {\n        /* 每隔 100ms 进来一次 */\n        bsp_LedToggle(2);\n    }\n\n    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */\n    if (ucKeyCode != KEY_NONE)\n    {\n        switch (ucKeyCode)\n        {\n            case KEY_DOWN_K1:           /* K1 键按下，数据复制 */\n                DSP_Copy();\n                break;\n\n            case KEY_DOWN_K2:           /* K2 键按下，数据填充 */\n                DSP_Fill();\n                break;\n\n            case KEY_DOWN_K3:           /* K3 键按下，浮点转定点 */\n                DSP_FloatToFix();\n                break;\n\n            default:\n                /* 其他的键值不处理 */\n                break;\n        }\n    }\n}\n}
```

16.8 总结

本期教程就跟大家讲这么多，有兴趣的可以深入研究这些函数源码的实现。



第17章 DSP 功能函数-定点数互转

本期教程主要讲解功能函数中的 Q7, Q15 和 Q31 分别向其它类型数据转换。

17.1 初学者重要提示

17.2 DSP 基础运算指令

17.3 定点数 Q7 转换

17.4 定点数 Q15 转换

17.5 定点数 Q31 转换

17.6 总结

17.1 初学者重要提示

◆ 浮点数的四舍五入处理:

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=95149>。

◆ C 库的浮点数四舍五入函数 round, roundf, round 使用说明:

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=95156>。

17.2 DSP 基础运算指令

本章用到的 DSP 指令在前面章节都已经讲解过。

17.3 定点数 Q7 转换

Q7 转浮点数:

$pDst[n] = (float32_t) pSrc[n] / 128;$ $0 \leq n < blockSize.$

Q7 转 Q31:

$pDst[n] = (q31_t) pSrc[n] << 24;$ $0 \leq n < blockSize.$

Q7 转 Q15:

$pDst[n] = (q15_t) pSrc[n] << 8;$ $0 \leq n < blockSize.$

17.3.1 函数 arm_q7_to_float

函数原型:

```
void arm_q7_to_float(  
  const q7_t * pSrc,
```



```
float32_t * pDst,  
       uint32_t blockSize)
```

函数描述：

这个函数用于定点数 Q7 转浮点数。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数转换个数。

17.3.2 函数 arm_q7_to_q31

函数原型：

```
void arm_q7_to_q31(  
    const q7_t * pSrc,  
    q31_t * pDst,  
    uint32_t blockSize)
```

函数描述：

这个函数用于定点 Q7 转定点数 Q31。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数是转换的次数。

17.3.3 函数 arm_q7_to_q15

函数原型：

```
void arm_q7_to_q15(  
    const q7_t * pSrc,  
    q15_t * pDst,  
    uint32_t blockSize)
```

函数描述：

这个函数用于定点数 Q7 转定点数 Q15。

函数参数：

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数是转换的次数。

17.3.4 使用举例

程序设计：

```
/*  
*****  
* 函数名: DSP_Q7  
* 功能说明: Q7格式数据向其它格式转换  
* 形参: 无  
* 返回值: 无  
*/
```



```
*****
*/
static void DSP_Q7(void)
{
    float32_t pDst[10];
    uint32_t pIndex;

    q31_t pDst1[10];
    q15_t pDst2[10];
    q7_t pSrc[10];

    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc[pIndex] = rand()%128;
        printf("pSrc[%d] = %d\r\n", pIndex, pSrc[pIndex]);
    }

    /******arm_q7_to_float(pSrc, pDst, 10);*****/
    arm_q7_to_float(pSrc, pDst, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_q7_to_float: pDst[%d] = %f\r\n", pIndex, pDst[pIndex]);
    }

    /******arm_q7_to_q31(pSrc, pDst1, 10);*****/
    arm_q7_to_q31(pSrc, pDst1, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_q7_to_q31: pDst1[%d] = %d\r\n", pIndex, pDst1[pIndex]);
    }

    /******arm_q7_to_q15(pSrc, pDst2, 10);*****/
    arm_q7_to_q15(pSrc, pDst2, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_q7_to_q15: pDst2[%d] = %d\r\n", pIndex, pDst2[pIndex]);
    }

    /******printf("*****\r\n");*****/
    printf("*****\r\n");
}
```

实验现象：

```
操作提示:  
1. 启动一个自动重装软件定时器，每100ms翻转一次LED2  
2. 按下按键K1, Q7数据格式转换  
3. 按下按键K2, Q15数据格式转换  
4. 按下按键K3, Q31数据格式转换  
pSrc[0] = 28  
pSrc[1] = 63  
pSrc[2] = 111  
pSrc[3] = 22  
pSrc[4] = 122  
pSrc[5] = 69  
pSrc[6] = 125  
pSrc[7] = 12  
pSrc[8] = 56  
pSrc[9] = 43  
arm_q7_to_float: pDst[0] = 0.218750  
arm_q7_to_float: pDst[1] = 0.492188  
arm_q7_to_float: pDst[2] = 0.867188  
arm_q7_to_float: pDst[3] = 0.171875  
arm_q7_to_float: pDst[4] = 0.953125  
arm_q7_to_float: pDst[5] = 0.539063  
arm_q7_to_float: pDst[6] = 0.976563  
arm_q7_to_float: pDst[7] = 0.093750
```

17.4 定点数 Q15 转换

Q15 转浮点数:

```
pDst[n] = (float32_t) pSrc[n] / 32768; 0 <= n < blockSize.
```

Q15 转 Q31:

```
pDst[n] = (q31_t) pSrc[n] << 16; 0 <= n < blockSize.
```

Q15 转 Q7:

```
pDst[n] = (q7_t) pSrc[n] >> 8; 0 <= n < blockSize.
```

17.4.1 函数 arm_q15_to_float

函数原型:

```
void arm_q15_to_float(  
    const q15_t * pSrc,  
    float32_t * pDst,  
    uint32_t blockSize)
```

函数描述:

这个函数用于定点数 Q15 转浮点数。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数是转换的次数。



17.4.2 函数 arm_q15_to_q31

函数原型:

```
void arm_q15_to_q31(
    const q15_t * pSrc,
    q31_t * pDst,
    uint32_t blockSize)
```

函数描述:

这个函数用于定点数 Q15 转定点数 Q31。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数是转换的次数。

17.4.3 函数 arm_q15_to_q7

函数原型:

```
void arm_q15_to_q7(
    const q15_t * pSrc,
    q7_t * pDst,
    uint32_t blockSize)
```

函数描述:

这个函数用于定点数 Q15 转定点数 Q7。

函数参数:

- ◆ 第 1 个参数源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数是转换的次数。

17.4.4 使用举例

程序设计:

```
/*
*****
* 函数名: DSP_Q15
* 功能说明: Q15格式数据向其它格式转换
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Q15(void)
{
    float32_t pDst[10];
    uint32_t pIndex;
    q31_t pDst1[10];
    q15_t pSrc[10];
    q7_t pDst2[10];

    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc[pIndex] = rand()%32678;
```



```
    printf("pSrc[%d] = %d\r\n", pIndex, pSrc[pIndex]);  
}  
  
/*********************************************/  
arm_q15_to_float(pSrc, pDst, 10);  
for(pIndex = 0; pIndex < 10; pIndex++)  
{  
    printf("arm_q15_to_float: pDst[%d] = %f\r\n", pIndex, pDst[pIndex]);  
}  
  
/*********************************************/  
arm_q15_to_q31(pSrc, pDst1, 10);  
for(pIndex = 0; pIndex < 10; pIndex++)  
{  
    printf("arm_q15_to_q31: pDst1[%d] = %d\r\n", pIndex, pDst1[pIndex]);  
}  
  
/*********************************************/  
arm_q15_to_q7(pSrc, pDst2, 10);  
for(pIndex = 0; pIndex < 10; pIndex++)  
{  
    printf("arm_q15_to_q7: pDst2[%d] = %d\r\n", pIndex, pDst2[pIndex]);  
}  
  
/*********************************************/  
printf("*****\r\n");  
}
```

实验现象：

```
操作提示：  
1. 启动一个自动重装软件定时器，每100ms翻转一次LED2  
2. 按下按键K1, Q7数据格式转换  
3. 按下按键K2, Q15数据格式转换  
4. 按下按键K3, Q31数据格式转换  
pSrc[0] = 6172  
pSrc[1] = 15097  
pSrc[2] = 29817  
pSrc[3] = 8878  
pSrc[4] = 7362  
pSrc[5] = 31995  
pSrc[6] = 18887  
pSrc[7] = 14008  
pSrc[8] = 10228  
pSrc[9] = 20581  
arm_q15_to_float: pDst[0] = 0.188354  
arm_q15_to_float: pDst[1] = 0.460724  
arm_q15_to_float: pDst[2] = 0.909943  
arm_q15_to_float: pDst[3] = 0.270935  
arm_q15_to_float: pDst[4] = 0.224670  
arm_q15_to_float: pDst[5] = 0.976410  
arm_q15_to_float: pDst[6] = 0.576385  
arm_q15_to_float: pDst[7] = 0.427490
```

17.5 定点数 Q31 转换

Q31 转浮点数：

$$pDst[n] = (\text{float32_t}) pSrc[n] / 2147483648; \quad 0 \leq n < \text{blockSize}.$$

Q31 转 Q15：

$$pDst[n] = (\text{q15_t}) pSrc[n] \gg 16; \quad 0 \leq n < \text{blockSize}.$$



Q31 转 Q7:

```
pDst[n] = (q7_t) pSrc[n] >> 24; 0 <= n < blockSize.
```

17.5.1 函数 arm_q31_to_float

函数原型:

```
void arm_q31_to_float(
    const q31_t * pSrc,
    float32_t * pDst,
    uint32_t blockSize)
```

函数描述:

这个函数用于 32 位定点数转浮点数。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数是转换个数。

17.5.2 函数 arm_q31_to_q15

函数原型:

```
void arm_q31_to_q15(
    const q31_t * pSrc,
    q15_t * pDst,
    uint32_t blockSize)
```

函数描述:

用于 32 位定点数转 16 位定点数。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。
- ◆ 第 3 个参数是转换的数据个数。

17.5.3 函数 arm_q31_to_q7

函数原型:

```
void arm_q31_to_q7(
    const q31_t * pSrc,
    q7_t * pDst,
    uint32_t blockSize)
```

函数描述:

用于 32 位定点数转 8 位定点数。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是转换后的数据地址。



- ◆ 第3个参数是转换的数据个数。

17.5.4 使用举例

程序设计：

```
/*
***** DSP_Q31 *****
* 函数名: DSP_Q31
* 功能说明: Q31格式数据向其它格式转换
* 形参: 无
* 返回值: 无
***** DSP_Q31 *****
*/
static void DSP_Q31(void)
{
    float32_t pDst[10];
    uint32_t pIndex;

    q31_t pSrc[10];
    q15_t pDst1[10];
    q7_t pDst2[10];

    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        pSrc[pIndex] = rand();
        printf("pSrc[%d] = %d\r\n", pIndex, pSrc[pIndex]);
    }

    /****** arm_q31_to_float *****/
    arm_q31_to_float(pSrc, pDst, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_q31_to_float: pDst[%d] = %f\r\n", pIndex, pDst[pIndex]);
    }

    /****** arm_q31_to_q15 *****/
    arm_q31_to_q15(pSrc, pDst1, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_q31_to_q15: pDst1[%d] = %d\r\n", pIndex, pDst1[pIndex]);
    }

    /****** arm_q31_to_q7 *****/
    arm_q31_to_q7(pSrc, pDst2, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_q31_to_q7: pDst2[%d] = %d\r\n", pIndex, pDst2[pIndex]);
    }

    /****** */
    printf("*****\r\n");
}
```

实验现象：

操作提示：

1. 启动一个自动重装软件定时器，每100ms翻转一次LED2
2. 按下按键K1, Q7数据格式转换
3. 按下按键K2, Q15数据格式转换
4. 按下按键K3, Q31数据格式转换

```
psrc[0] = 6172
psrc[1] = 1777208127
psrc[2] = 1401033711
psrc[3] = 1798475286
psrc[4] = 114641786
psrc[5] = 1628409413
psrc[6] = 525775229
psrc[7] = 1720641420
psrc[8] = 1470977720
psrc[9] = 275594155
arm_q31_to_float: pdst[0] = 0.000003
arm_q31_to_float: pdst[1] = 0.827577
arm_q31_to_float: pdst[2] = 0.652407
arm_q31_to_float: pdst[3] = 0.837480
arm_q31_to_float: pdst[4] = 0.053384
arm_q31_to_float: pdst[5] = 0.758287
arm_q31_to_float: pdst[6] = 0.244833
arm_q31_to_float: pdst[7] = 0.801236
```

17.6 实验例程说明 (MDK)

配套例子：

V7-212_DSP 功能函数 (定点数互转)

实验目的：

1. 学习 DSP 功能函数 (定点数互转)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1, 串口打印 Q7 转换其它数据格式。
3. 按下按键 K2, 串口打印 Q15 转换其它数据格式。
4. 按下按键 K3, 串口打印 Q31 转换其它数据格式。

使用 AC6 注意事项

特别注意附件章节 C 的问题

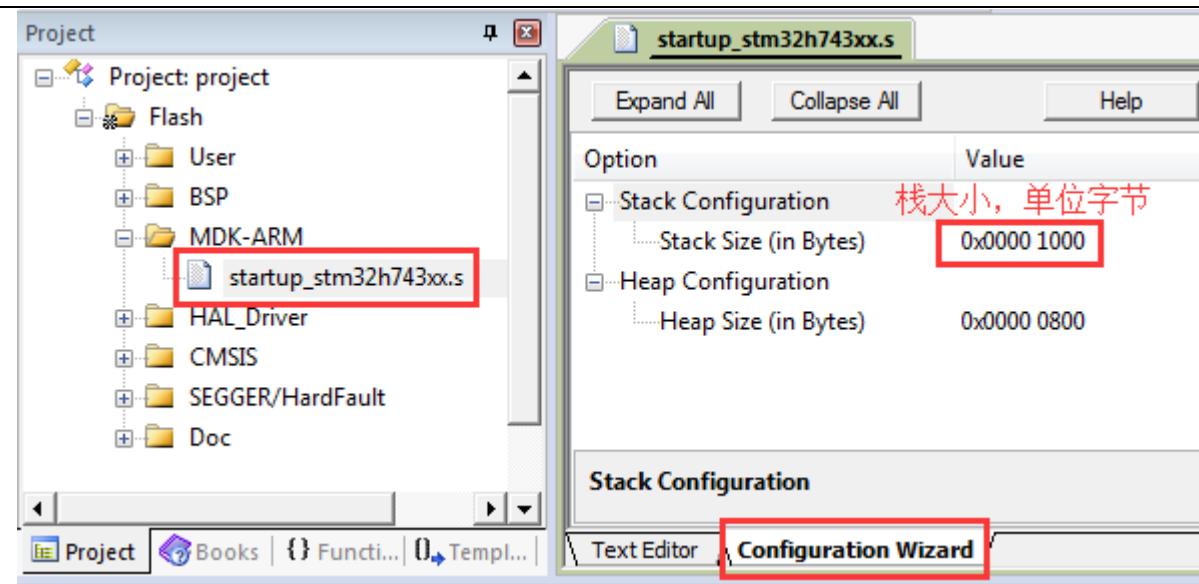
上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

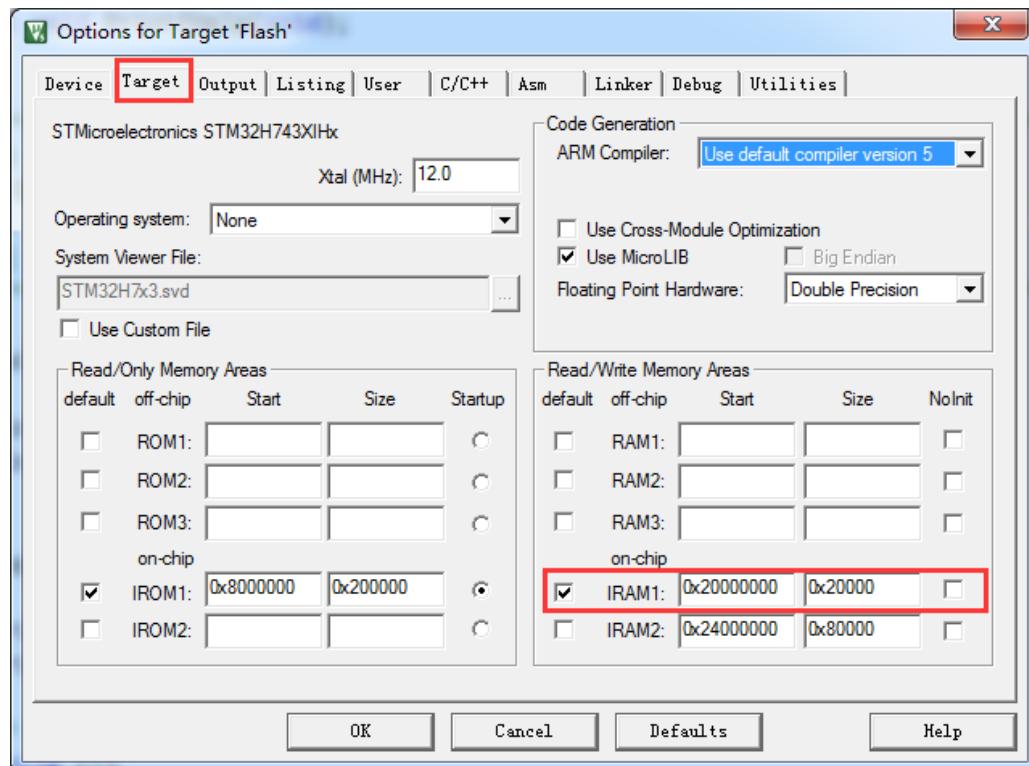
详见本章的 3.4 4.4, 5.4 小节。

程序设计：

- ◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
```



```
{  
    /* 配置 MPU */  
    MPU_Config();  
  
    /* 使能 L1 Cache */  
    CPU_CACHE_Enable();  
  
    /*  
     * STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：  
     * - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。  
     * - 设置 NVIC 优先级分组为 4。  
     */  
    HAL_Init();  
  
    /*  
     * 配置系统时钟到 400MHz  
     * - 切换使用 HSE。  
     * - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。  
     */  
    SystemClock_Config();  
  
    /*  
     * Event Recorder:  
     * - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。  
     * - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章  
     */  
#if Enable_EventRecorder == 1  
    /* 初始化 EventRecorder 并开启 */  
    EventRecorderInitialize(EventRecordAll, 1U);  
    EventRecorderStart();  
#endif  
  
    bsp_InitKey(); /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */  
    bsp_InitTimer(); /* 初始化滴答定时器 */  
    bsp_InitUart(); /* 初始化串口 */  
    bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */  
    bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();
```



```
/* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。



- 按下按键 K1, 串口打印 Q7 转换其它数据格式。
- 按下按键 K2, 串口打印 Q15 转换其它数据格式。
- 按下按键 K3, 串口打印 Q31 转换其它数据格式。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形    参: 无
***** 返  回 值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下, Q7 转换其它数据格式 */
                    DSP_Q7();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下, Q15 转换其它数据格式 */
                    DSP_Q15();
                    break;

                case KEY_DOWN_K3:           /* K3 键按下, Q31 转换其它数据格式 */
                    DSP_Q31();
                    break;

                default:
                    /* 其他的键值不处理 */
                    break;
            }
        }
    }
}
```

```
}
```

17.7 实例程说明 (IAR)

配套例子：

V7-212_DSP 功能函数 (定点数互转)

实验目的：

1. 学习 DSP 功能函数 (定点数互转)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1, 串口打印 Q7 转换其它数据格式。
3. 按下按键 K2, 串口打印 Q15 转换其它数据格式。
4. 按下按键 K3, 串口打印 Q31 转换其它数据格式。

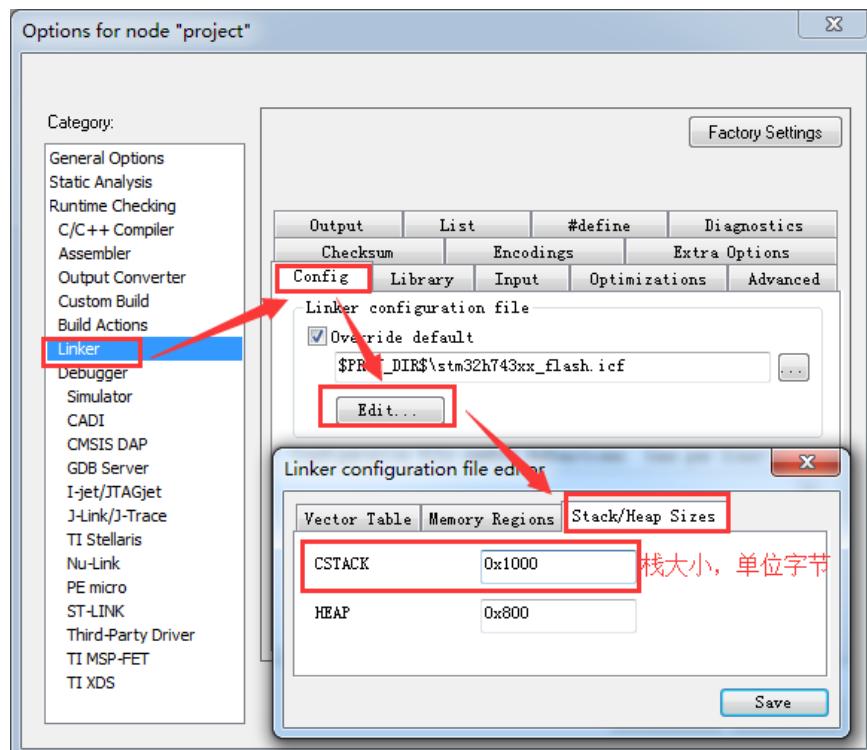
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

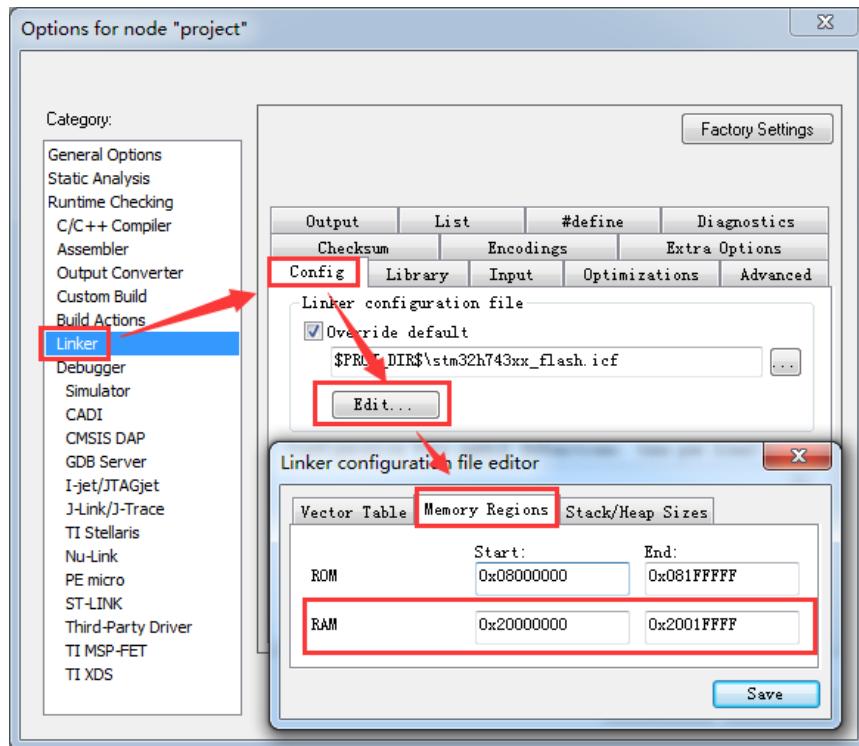
详见本章的 3.4 4.4, 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
}
```



```
SystemClock_Config();

/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
```



```
MPU_InitStruct.Size          = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable    = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable     = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number         = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1, 串口打印 Q7 转换其它数据格式。
- 按下按键 K2, 串口打印 Q15 转换其它数据格式。
- 按下按键 K3, 串口打印 Q31 转换其它数据格式。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */
}
```



```
PrintfHelp(); /* 打印操作提示信息 */
```

```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */
```

```
/* 进入主程序循环体 */
```

```
while (1)
```

```
{
```

```
    bsp_Idle(); /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */
```

```
    /* 判断定时器超时时间 */
```

```
    if (bsp_CheckTimer(0))
```

```
{
```

```
    /* 每隔 100ms 进来一次 */
```

```
    bsp_LedToggle(2);
```

```
}
```

```
    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
```

```
    if (ucKeyCode != KEY_NONE)
```

```
{
```

```
    switch (ucKeyCode)
```

```
{
```

```
        case KEY_DOWN_K1: /* K1 键按下，Q7 转换其它数据格式 */
```

```
            DSP_Q7();
```

```
            break;
```

```
        case KEY_DOWN_K2: /* K2 键按下，Q15 转换其它数据格式 */
```

```
            DSP_Q15();
```

```
            break;
```

```
        case KEY_DOWN_K3: /* K3 键按下，Q31 转换其它数据格式 */
```

```
            DSP_Q31();
```

```
            break;
```

```
        default:
```

```
            /* 其他的键值不处理 */
```

```
            break;
```

```
}
```

```
}
```

```
}
```

17.8 总结

本期教程就跟大家讲这么多，有兴趣的可以深入研究这些函数源码的实现。



第18章 DSP 控制函数-更好用的 SIN, COS 计 算

本期教程主要讲解控制函数中的 cos 和 sin 函数数值的获取，这里使用的函数比起第 13 章中使用的 sin 和 cos 函数数值的获取要方便很多。

18.1 初学者重要提示

18.2 DSP 基础运算指令

18.3 浮点数 SIN 和 COS

18.4 定点数 SIN 和 COS

18.5 Clark 正变换和逆变换

18.6 Park 正变换和逆变换

18.7 实验例程说明 (MDK)

18.8 实验例程说明 (IAR)

18.9 总结

18.1 初学者重要提示

◆ Matlab2018a 手动加载数据的方法在第 13 章的 13.6 章节进行了说明。

如果要看 Matlab2012，参考第 1 版 DSP 教程：

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=3886>。

18.2 DSP 基础运算指令

本章用到的 DSP 指令在前面章节都已经讲解过。

18.3 浮点数 SIN 和 COS

使用表查找法和线性插值方式来计算正弦和余弦值。

18.3.1 函数 arm_sin_cos_f32

函数原型：

```
void arm_sin_cos_f32(
    float32_t theta,
    float32_t * pSinVal,
```

float32_t * pCosVal)

函数描述：

这个函数用于浮点方式计算正弦和余弦值。

函数参数：

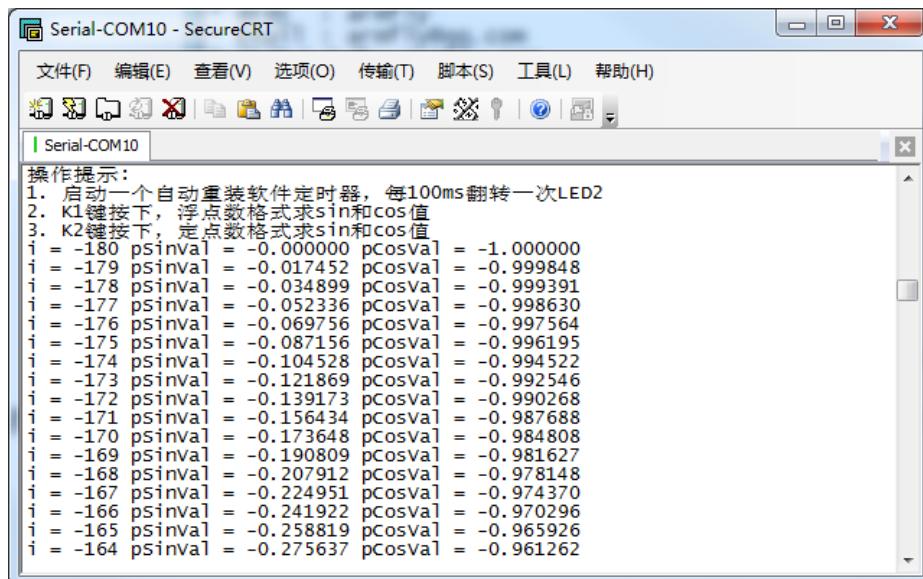
- ◆ 第 1 个参数参数是角度。这里输入角度-180 到 179 就能得到一个周期的正弦或者余弦数值。
- ◆ 第 2 个参数是转换后求出的 sin 值。
- ◆ 第 3 个参数是转换后求出的 cos 值。

18.3.2 使用举例

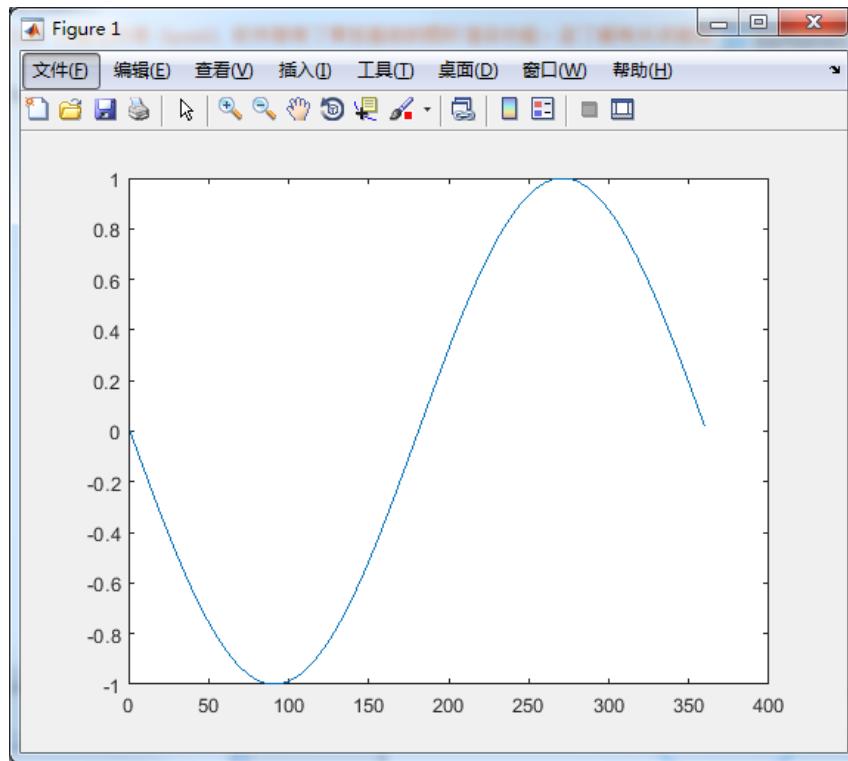
程序设计：

```
/*
*****
* 函数名: DSP_SIN_COS
* 功能说明: 浮点数cos和sin计算
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_SIN_COS(void)
{
    int16_t i;
    float32_t pSinVal;
    float32_t pCosVal;

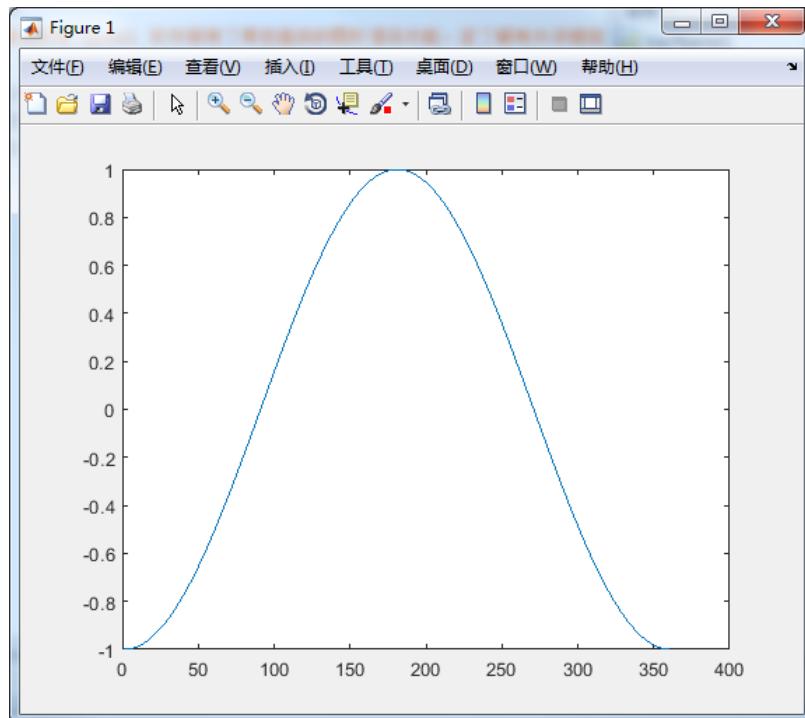
    for(i = -180; i < 180; i++)
    {
        arm_sin_cos_f32(i, &pSinVal, &pCosVal);
        printf("i = %d pSinVal = %f pCosVal = %f\r\n", i, pSinVal, pCosVal);
    }
}
```

实验现象：

通过 matlab 绘制 sin 函数的输出数据的曲线 (绘制方法见第 10 章的 10.4 小节)



1. 通过 matlab 绘制 cos 函数的输出数据的曲线 (绘制方法见第 10 章的 10.4 小节)



参数 theta 的单位是角度。这里输入角度 $-2^{31} \sim 2^{31}-1$ 就能得到一个周期的正弦或者余弦数值



18.4 定点数 SIN 和 COS

使用表查找法和线性插值方式来计算正弦和余弦值。

18.4.1 函数 arm_sin_cos_q31

函数原型:

```
void arm_sin_cos_f32(  
    float32_t theta,  
    float32_t * pSinVal,  
    float32_t * pCosVal)
```

函数描述:

这个函数用于定点方式计算正弦和余弦值。

函数参数:

- ◆ 第 1 个参数参数是角度。这里输入角度 $-2^{31} \sim 2^{31}-1$ 就能得到一个周期的正弦或者余弦数值。
- ◆ 第 2 个参数是转换后求出的 sin 值。
- ◆ 第 3 个参数是转换后求出的 cos 值。

18.4.2 使用举例

程序设计:

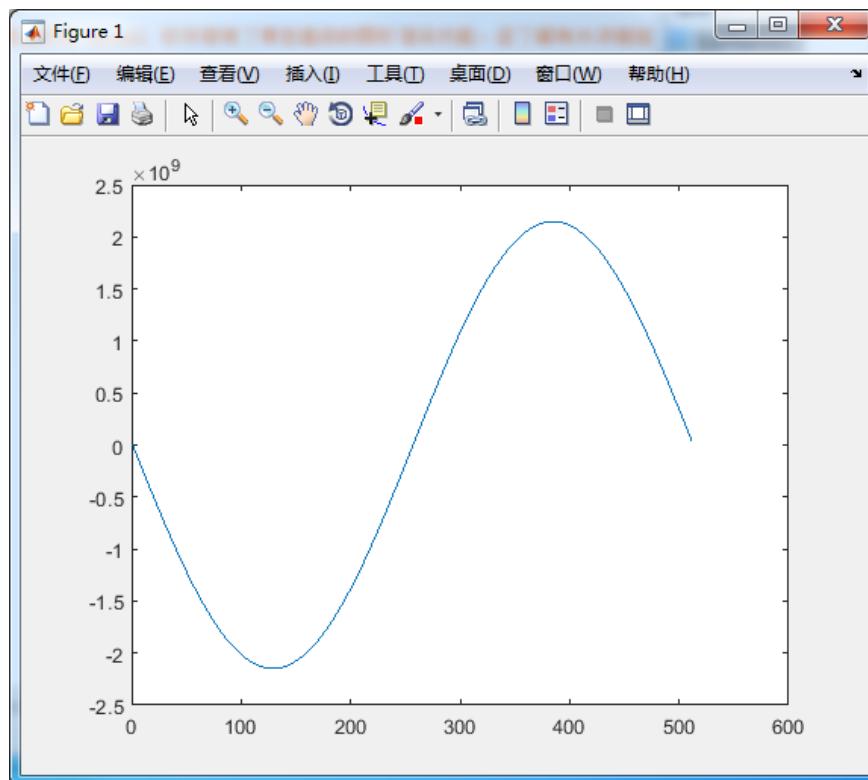
```
/*  
*****  
* 函数名: DSP_SIN_COS  
* 功能说明: 浮点数cos和sin计算  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_SIN_COS(void)  
{  
    int16_t i;  
    float32_t pSinVal;  
    float32_t pCosVal;  
  
    for(i = -180; i < 180; i++)  
    {  
        arm_sin_cos_f32(i, &pSinVal, &pCosVal);  
        printf("i = %d pSinVal = %f pCosVal = %f\r\n", i, pSinVal, pCosVal);  
    }  
}
```

实验现象:

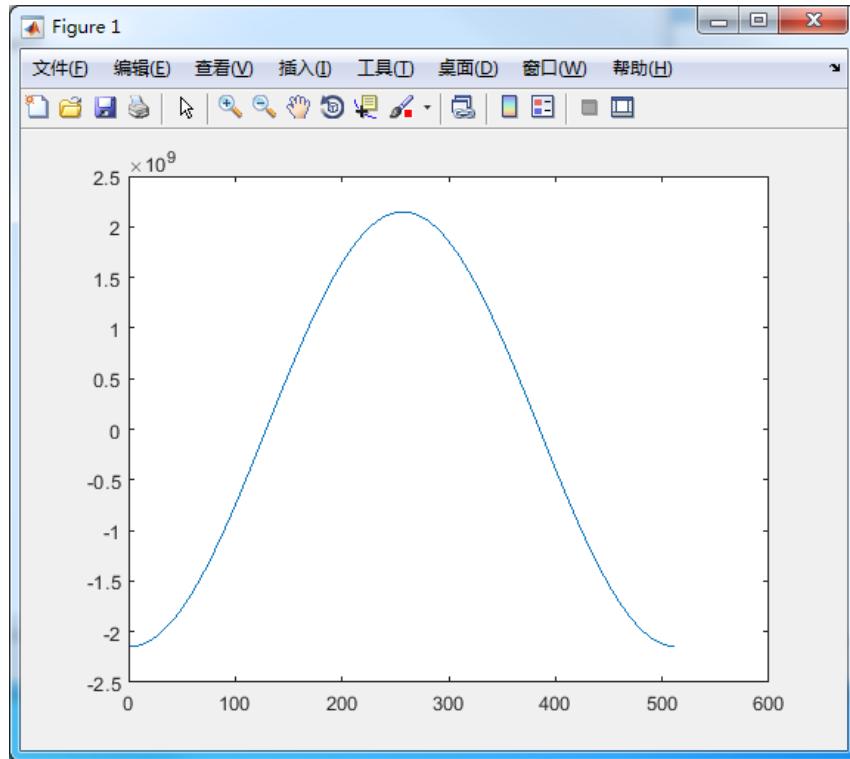
The screenshot shows a window titled "Serial-COM10 - SecureCRT". The menu bar includes: 文件(F), 编辑(E), 查看(V), 选项(O), 传输(T), 脚本(S), 工具(L), 帮助(H). The toolbar contains icons for copy, paste, find, etc. A status bar at the bottom says "Serial-COM10". The main pane displays the following text:

```
操作提示:  
1. 启动一个自动重装软件定时器, 每100ms翻转一次LED2  
2. K1键按下, 浮点数格式求sin和cos值  
3. K2键按下, 定点数格式求sin和cos值  
i = -256 psinVal = 0 pCosVal = -2147483648  
i = -255 psinVal = -26352928 pCosVal = -2147321946  
i = -254 psinVal = -52701887 pCosVal = -2146836866  
i = -253 psinVal = -79042909 pCosVal = -2146028480  
i = -252 psinVal = -105372028 pCosVal = -2144896910  
i = -251 psinVal = -131685278 pCosVal = -2143442326  
i = -250 psinVal = -157978697 pCosVal = -2141664948  
i = -249 psinVal = -184248325 pCosVal = -2139565043  
i = -248 psinVal = -210490206 pCosVal = -2137142927  
i = -247 psinVal = -236700388 pCosVal = -2134398966  
i = -246 psinVal = -262874923 pCosVal = -2131333572  
i = -245 psinVal = -289009871 pCosVal = -2127947206  
i = -244 psinVal = -315101295 pCosVal = -2124240380  
i = -243 psinVal = -341145265 pCosVal = -2120213651  
i = -242 psinVal = -367137861 pCosVal = -2115867626  
i = -241 psinVal = -393075166 pCosVal = -2111202959  
i = -240 psinVal = -418953276 pCosVal = -2106220352
```

通过 matlab 绘制 sin 函数的输出数据的曲线 (绘制方法见第 13 章的 13.6 小节)



2. 通过 matlab 绘制 cos 函数的输出数据的曲线 (绘制方法见第 13 章的 13.6 小节)



18.5 Clarke 正变换和逆变换

暂时没有研究。

18.6 Park 正变换和逆变换

暂时没有研究。

18.7 实验例程说明 (MDK)

配套例子：

V7-213_DSP 控制函数 (三角函数)

实验目的：

1. 学习 DSP 控制函数 (三角函数)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，浮点数格式求 sin 和 cos 值。
3. 按下按键 K2，定点数格式求 sin 和 cos 值。

使用 AC6 注意事项

特别注意附件章节 C 的问题

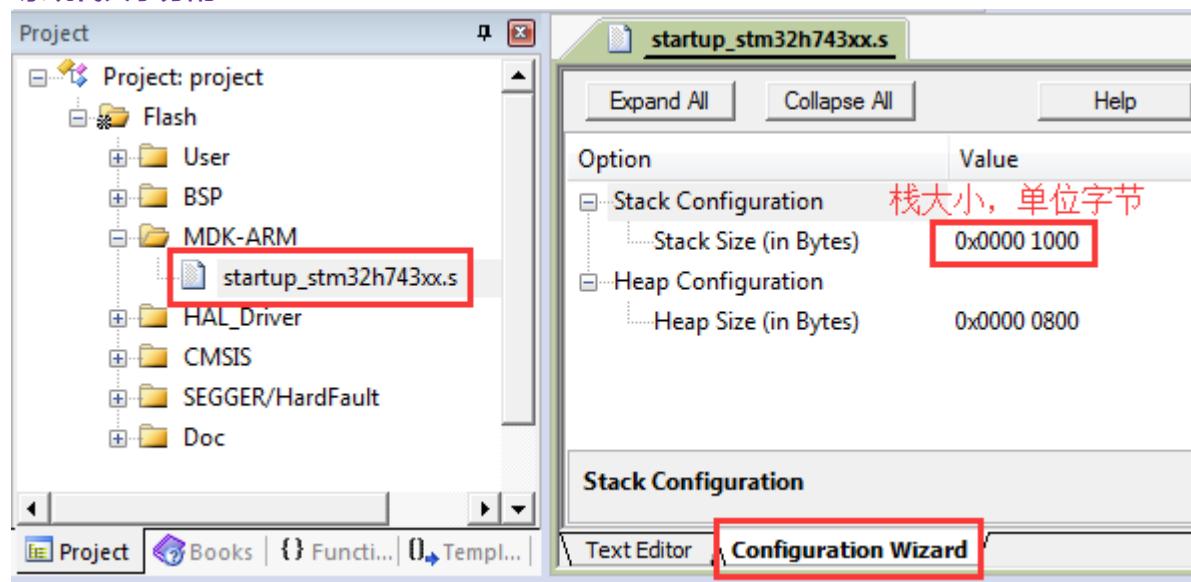
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

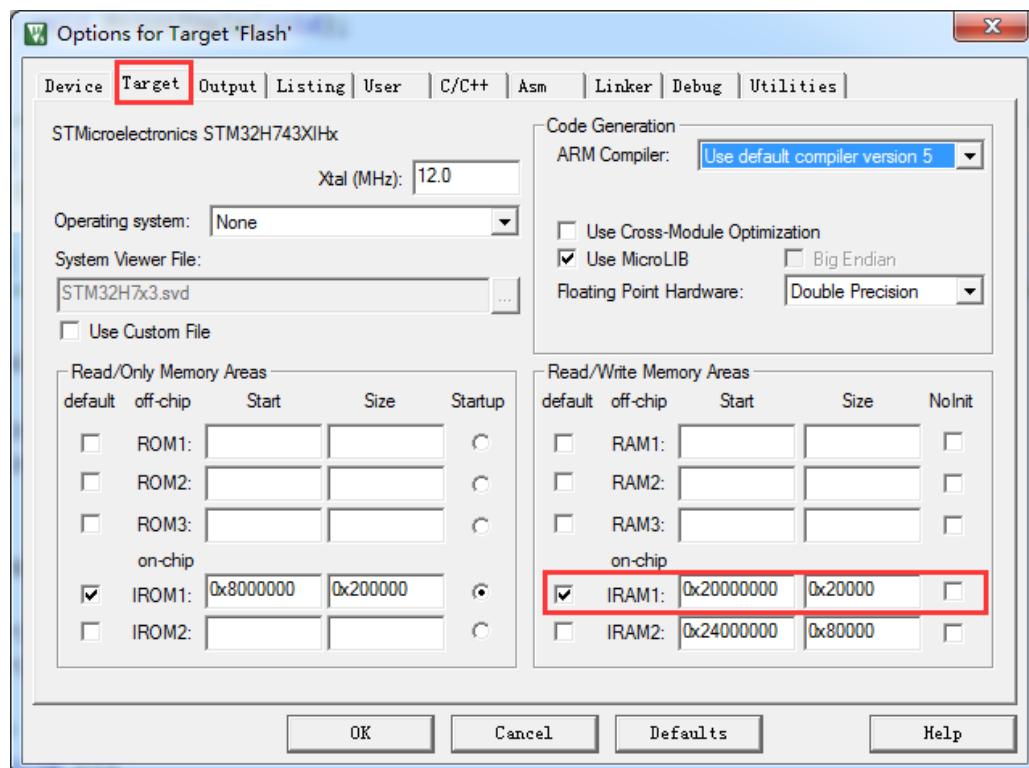
详见本章的 3.4 4.4，5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：



```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIIV 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
        - 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
```



```
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /*使能 MPU */
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
}
```



```
    SCB_EnableDCache();  
}
```

◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，浮点数格式求 sin 和 cos 值。
- 按下按键 K2，定点数格式求 sin 和 cos 值。

```
/*  
*****  
* 函数名: main  
* 功能说明: c 程序入口  
* 形参: 无  
* 返回值: 错误代码(无需处理)  
*****  
*/  
int main(void)  
{  
    uint8_t ucKeyCode; /* 按键代码 */  
  
    bsp_Init(); /* 硬件初始化 */  
    PrintfLogo(); /* 打印例程信息到串口 1 */  
  
    PrintfHelp(); /* 打印操作提示信息 */  
  
    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */  
  
    /* 进入主程序循环体 */  
    while (1)  
    {  
        bsp_Idle(); /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */  
  
        /* 判断定时器超时时间 */  
        if (bsp_CheckTimer(0))  
        {  
            /* 每隔 100ms 进来一次 */  
            bsp_LedToggle(2);  
        }  
  
        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */  
        if (ucKeyCode != KEY_NONE)  
        {  
            switch (ucKeyCode)  
            {  
                case KEY_DOWN_K1: /* K1 键按下, 浮点数格式求 sin 和 cos 值 */  
                    DSP_SIN_COS();  
                    break;  
  
                case KEY_DOWN_K2: /* K2 键按下, 定点数格式求 sin 和 cos 值 */  
                    DSP_SIN_COS_Q31();  
                    break;  
            }  
        }  
    }  
}
```



```
default:  
    /* 其他的键值不处理 */  
    break;  
}  
}  
}
```

18.8 实例例程说明 (IAR)

配套例子：

V7-213_DSP 控制函数 (三角函数)

实验目的：

1. 学习 DSP 控制函数 (三角函数)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，浮点数格式求 sin 和 cos 值。
3. 按下按键 K2，定点数格式求 sin 和 cos 值。

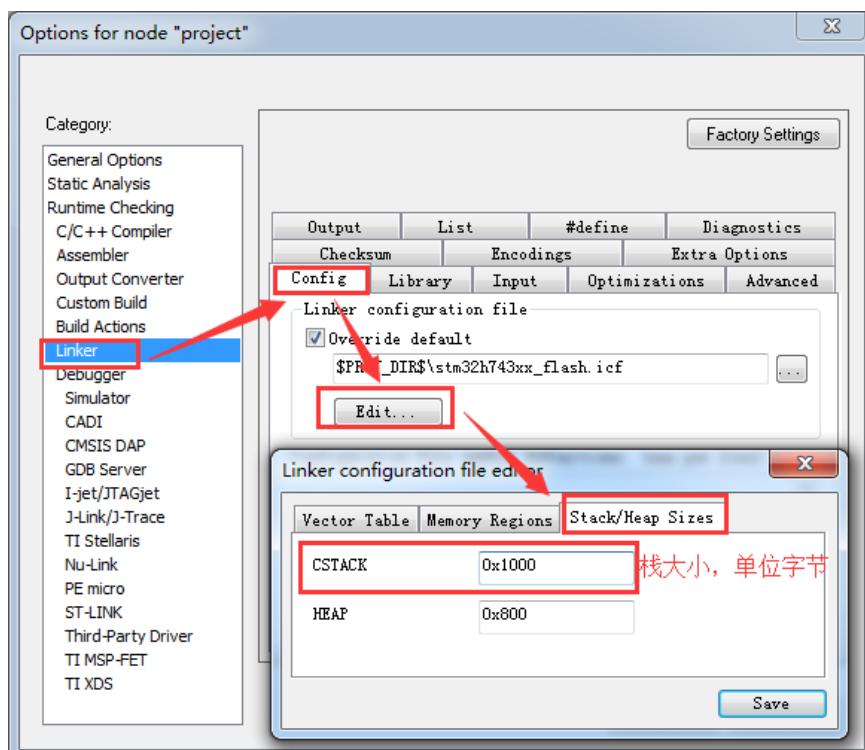
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

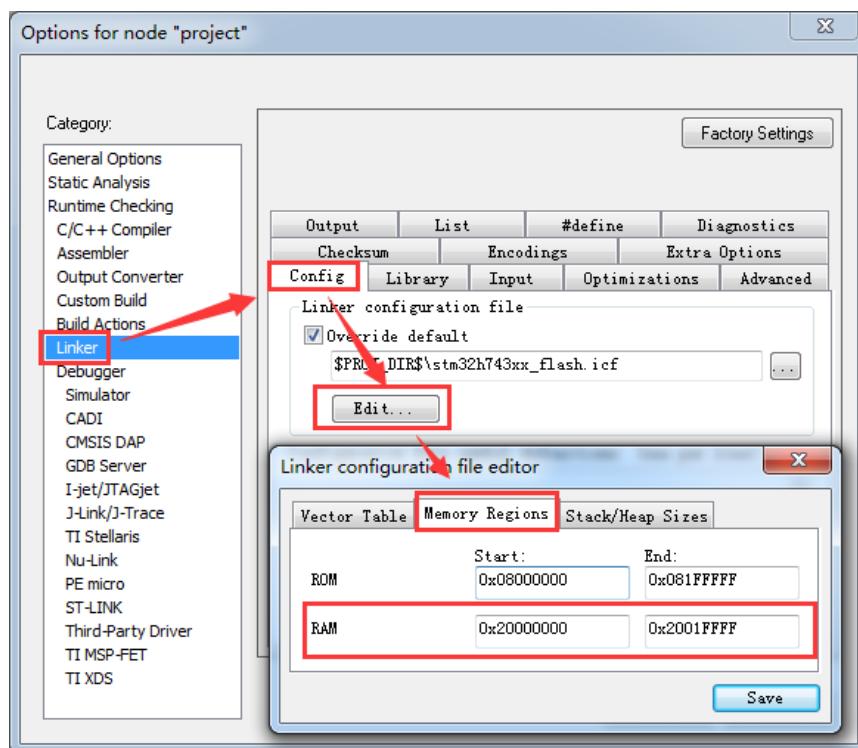
详见本章的 3.4 4.4, 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
     * STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
     * - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
     * - 设置 NVIC 优先级分组为 4。
     */
    HAL_Init();

    /*
     * 配置系统时钟到 400MHz
     * - 切换使用 HSE。
     * - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
     */
}
```



```
SystemClock_Config();

/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
```



```
MPU_InitStruct.Size          = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable    = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable     = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number         = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，浮点数格式求 sin 和 cos 值。
- 按下按键 K2，定点数格式求 sin 和 cos 值。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下，浮点数格式求 sin 和 cos 值 */
                DSP_SIN_COS();
                break;

            case KEY_DOWN_K2:          /* K2 键按下，定点数格式求 sin 和 cos 值 */
                DSP_SIN_COS_Q31();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}
```

18.9 总结

本期教程就跟大家讲这么多，有兴趣的可以深入研究下算法的实现。



第19章 DSP 复数运算-共轭，点乘和求模

本期教程主要讲解复数运算中的共轭，点乘和模的求解。

19.1 初学者重要提示

19.2 DSP 基础运算指令

19.3 复数共轭运算 (ComplexConj)

19.4 复数点乘 (ComplexDotProduct)

19.5 复数求模 (ComplexMag)

19.6 实验例程说明 (MDK)

19.7 实验例程说明 (IAR)

19.8 总结

19.1 初学者重要提示

◆ 复数运算比较重要，后面 FFT 章节要用到，如果印象不深的话，需要温习下高数知识了。

19.2 DSP 基础运算指令

本章用到的 DSP 指令在前面章节都已经讲解过。

19.3 复数共轭运算 (ComplexConj)

这部分函数用于复数共轭运算，公式描述如下：

```
for(n=0; n<numSamples; n++)  
{  
    pDst[(2*n)+0] = pSrc[(2*n)+0]; // 实部  
    pDst[(2*n)+1] = -pSrc[(2*n)+1]; // 虚部  
}
```

用代数式来表示 $a+bi$ 的共轭就是 $a-bi$ 。

19.3.1 函数 arm_cmplx_conj_f32

函数原型：

```
void arm_cmplx_conj_f32(  
    const float32_t * pSrc,
```



```
float32_t * pDst,  
uint32_t numSamples)
```

函数描述：

这个函数用于浮点数的复位共轭求解。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是求共轭后的数据地址。
- ◆ 第 3 个参数是转换的数据个数。

注意事项：

参数pSrc中存储的数据格式是（实部，虚部，实部，虚部.....），一定要按照这个顺序存储数据，比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3}。（注意第三个数据是0）。函数的输出结果pDst也是按照这个顺序存储的。

19.3.2 函数 arm_cmplx_conj_q31

函数原型：

```
void arm_cmplx_conj_q31(  
    const q31_t * pSrc,  
    q31_t * pDst,  
    uint32_t numSamples)
```

函数描述：

这个函数用于定点数 Q31 的复数共轭求解。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是求共轭后的数据地址。
- ◆ 第 3 个参数是转换的数据个数。

注意事项：

1. 数组 pSrc 中存储的数据格式是（实部，虚部，实部，虚部.....），一定要按照这个顺序存储数据，比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3}。（注意第三个数据是 0）。函数的输出结果 pDst 也是按照这个顺序存储的。
2. 这个函数使用了饱和运算。数值 0x80000000 由于饱和运算（源码中的_QSUB(0, in)）将变成 0xFFFFFFFF。

19.3.3 函数 arm_cmplx_conj_q15

函数原型：

```
void arm_cmplx_conj_q15(  
    const q15_t * pSrc,  
    q15_t * pDst,  
    uint32_t numSamples)
```

函数描述：



这个函数用于定点数 Q15 的复数共轭求解。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是求共轭后的数据地址。
- ◆ 第 3 个参数是转换的数据个数。

注意事项：

1. 数组 pSrc 中存储的数据格式是（实部，虚部，实部，虚部.....），一定要按照这个顺序存储数据，比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3}。
(注意第三个数据是 0)。函数的输出结果 pDst 也是按照这个顺序存储的。
2. 这个函数使用了饱和运算。数值 0x8000 由于饱和运算（源码中的_QSAX(0, in1)）将变成 0xFFFF。

19.3.4 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_Fill
* 功能说明: 数据填充
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_Fill(void)
{
    float32_t pDst[10];
    uint32_t pIndex;
    q31_t pDst1[10];
    q15_t pDst2[10];
    q7_t pDst3[10];

    arm_fill_f32(3.33f, pDst, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_fill_f32: pDst[%d] = %f\r\n", pIndex, pDst[pIndex]);
    }

    /*****
    arm_fill_q31(0x11111111, pDst1, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_fill_q31: pDst1[%d] = %x\r\n", pIndex, pDst1[pIndex]);
    }

    /*****
    arm_fill_q15(0x1111, pDst2, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_fill_q15: pDst2[%d] = %d\r\n", pIndex, pDst2[pIndex]);
    }

    /*****
    arm_fill_q7(0x11, pDst3, 10);
    for(pIndex = 0; pIndex < 10; pIndex++)
    {
        printf("arm_fill_q7: pDst3[%d] = %d\r\n", pIndex, pDst3[pIndex]);
    }

    ****/
    printf("*****\r\n");
}
```



```
/*
***** 函数名: DSP_CONJ
***** 功能说明: 复数求共轭
***** 形参: 无
***** 返回值: 无
*****
*/
static void DSP_CONJ(void)
{
    uint8_t i;
    float32_t pSrc[10] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f, 5.1f};
    float32_t pDst[10];

    q31_t pSrc1[10] = {1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
    q31_t pDst1[10];

    q15_t pSrc2[10] = {1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
    q15_t pDst2[10];

    /**浮点数共轭*****
    arm_cmplx_conj_f32(pSrc, pDst, 5);
    printf("****浮点数共轭*****\r\n");
    for(i = 0; i < 5; i++)
    {
        printf("pSrc[%d] = %f %f j      pDst[%d] = %f %f j\r\n", i, pSrc[2*i], pSrc[2*i+1], i, pDst[2*i], pDst[2*i+1]);
    }

    /**定点数共轭Q31*****
    printf("****定点数共轭Q31*****\r\n");
    arm_cmplx_conj_q31(pSrc1, pDst1, 5);
    for(i = 0; i < 5; i++)
    {
        printf("pSrc1[%d] = %d %d j      pDst1[%d] = %d %d j\r\n", i, pSrc1[2*i], pSrc1[2*i+1], i, pDst1[2*i], pDst1[2*i+1]);
    }

    /**定点数共轭Q15*****
    printf("****定点数共轭Q15*****\r\n");
    arm_cmplx_conj_q15(pSrc2, pDst2, 5);
    for(i = 0; i < 5; i++)
    {
        printf("pSrc2[%d] = %d %d j      pDst2[%d] = %d %d j\r\n", i, pSrc2[2*i], pSrc2[2*i+1], i, pDst2[2*i], pDst2[2*i+1]);
    }
}
```

实验现象：

```
Serial-COM10 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 帮助(H)
Serial-COM10
操作提示：
1. 启动一个自动重装软件定时器，每100ms翻转一次LED2
2. 按下按键K1，串口打函数DSP_CONJ的输出数据
3. 按下按键K2，串口打函数DSP_CmplxDotProduct的输出数据
4. 按下按键K3，串口打函数DSP_CmplxMag的输出数据
***浮点数共轭*****
psrc[0] = 1.100000 1.100000j      pdst[0] = 1.100000 -1.100000j
psrc[1] = 2.100000 2.100000j      pdst[1] = 2.100000 -2.100000j
psrc[2] = 3.100000 3.100000j      pdst[2] = 3.100000 -3.100000j
psrc[3] = 4.100000 4.100000j      pdst[3] = 4.100000 -4.100000j
psrc[4] = 5.100000 5.100000j      pdst[4] = 5.100000 -5.100000j
***定点数共轭Q31*****
psrc1[0] = 1 1j      pdst1[0] = 1 -1j
psrc1[1] = 2 2j      pdst1[1] = 2 -2j
psrc1[2] = 3 3j      pdst1[2] = 3 -3j
psrc1[3] = 4 4j      pdst1[3] = 4 -4j
psrc1[4] = 5 5j      pdst1[4] = 5 -5j
***定点数共轭Q15*****
psrc2[0] = 1 1j      pdst2[0] = 1 -1j
psrc2[1] = 2 2j      pdst2[1] = 2 -2j
psrc2[2] = 3 3j      pdst2[2] = 3 -3j
psrc2[3] = 4 4j      pdst2[3] = 4 -4j
psrc2[4] = 5 5j      pdst2[4] = 5 -5j
```

19.4 复数点乘 (ComplexDotProduct)

这部分函数用于复数共轭运算，公式描述如下：

```
realResult = 0;
imagResult = 0;
for (n = 0; n < numSamples; n++) {
    realResult += pSrcA[(2*n)+0] * pSrcB[(2*n)+0] - pSrcA[(2*n)+1] * pSrcB[(2*n)+1]; //实部
    imagResult += pSrcA[(2*n)+0] * pSrcB[(2*n)+1] + pSrcA[(2*n)+1] * pSrcB[(2*n)+0]; //虚部
}
```

用代数式来表示复数乘法就是：

$$(a+bi)(c+di)=(ac-bd)+(ad+bc)i.$$

19.4.1 函数 arm_cmplx_dot_prod_f32

函数原型：

```
void arm_cmplx_dot_prod_f32(
    const float32_t * pSrcA,
    const float32_t * pSrcB,
    uint32_t numSamples,
    float32_t * realResult,
    float32_t * imagResult)
```

函数描述：

这个函数用于浮点数的复数点乘。

函数参数：



- ◆ 第 1 个参数是源数据 A 地址。
- ◆ 第 2 个参数是源数据 B 地址。
- ◆ 第 3 个参数是点乘的数据个数。
- ◆ 第 4 个参数是点乘后的实数地址。
- ◆ 第 5 个参数是点乘后的虚数地址。

注意事项：

数组 pSrcA 和 pSrcB 中存储的数据格式是（实部，虚部，实部，虚部.....），一定要按照这个顺序存储数据，比如数据 1-j, j, 2+3j 这个三个数在数组中的存储格式就是：pSrcA[6] = {1, -1, 0, 1, 2, 3}。（注意第三个数据是 0）。而输出结果的实部和虚部是分开存储的。

19.4.2 函数 arm_cmplx_dot_prod_q31

函数原型：

```
void arm_cmplx_dot_prod_q31(  
    const q31_t * pSrcA,  
    const q31_t * pSrcB,  
    uint32_t numSamples,  
    q63_t * realResult,  
    q63_t * imagResult)
```

函数描述：

这个函数用于定点数 Q31 的复数点乘。

函数参数：

- ◆ 第 1 个参数是源数据 A 地址。
- ◆ 第 2 个参数是源数据 B 地址。
- ◆ 第 3 个参数是点乘的数据个数。
- ◆ 第 4 个参数是点乘后的实数地址。
- ◆ 第 5 个参数是点乘后的虚数地址。

注意事项：

1. 数组 pSrcA 和 pSrcB 中存储的数据格式是（实部，虚部，实部，虚部.....），一定要按照这个顺序存储数据，比如数据 1-j, j, 2+3j 这个三个数在数组中的存储格式就是：pSrcA[6] = {1, -1, 0, 1, 2, 3}。（注意第三个数据是 0）。而输出结果的实部和虚部是分开存储的。
2. 这个函数的内部使用了 64 累加器，1.31 格式数据乘以 1.31 格式数据结果就是 2.62 格式，这里我们将所得结果右移 14 位，那么数据就是 16.48 格式。由于加数是不支持饱和运算，所以只要 numSamples 的个数小于 32768 就不会有溢出的危险。

19.4.3 函数 arm_cmplx_dot_prod_q15

函数原型：

```
void arm_cmplx_dot_prod_q15(  
    const q15_t * pSrcA,  
    const q15_t * pSrcB,
```



```
uint32_t numSamples,  
q31_t * realResult,  
q31_t * imagResult)
```

函数描述：

这个函数用于定点数 Q15 的复数点乘。

函数参数：

- ◆ 第 1 个参数是源数据 A 地址。
- ◆ 第 2 个参数是源数据 B 地址。
- ◆ 第 3 个参数是点乘的数据个数。
- ◆ 第 4 个参数是点乘后的实数地址。
- ◆ 第 5 个参数是点乘后的虚数地址。

注意事项：

1. 数组 pSrcA 和 pSrcB 中存储的数据格式是（实部，虚部，实部，虚部.....），一定要按照这个顺序存储数据，比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是：pSrcA[6] = {1, -1, 0, 1, 2, 3}。（注意第三个数据是 0）。而输出结果的实部和虚部是分开存储的。
2. 这个函数的内部使用了 64 累加器，1.15 格式数据乘以 1.15 格式数据结果就是 2.30 格式，对应到 64bit 就是 34.30，然后将最终的计算结果转换为 8.24。

19.4.4 使用举例

程序设计：

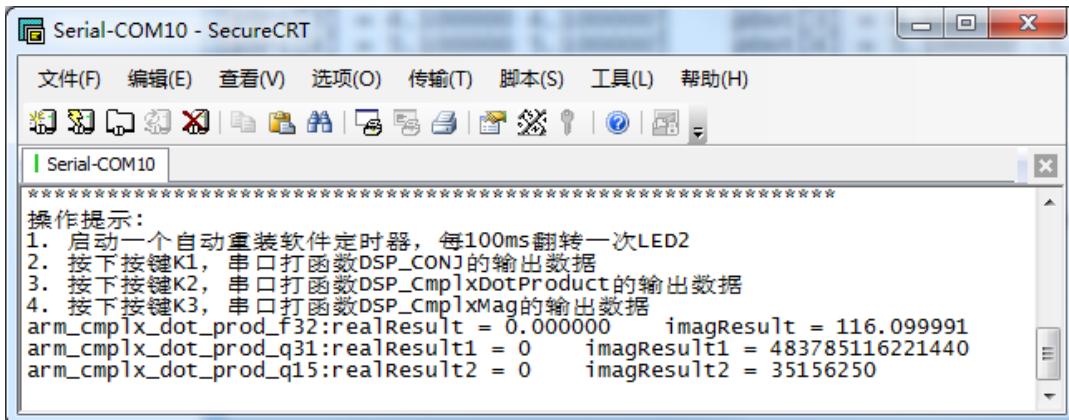
```
/*  
*****  
* 函数名: DSP_CmplxDotProduct  
* 功能说明: 复数点乘  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_CmplxDotProduct(void)  
{  
    float32_t pSrcA[10] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f, 5.1f};  
    float32_t pSrcB[10] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f, 5.1f};  
    float32_t realResult;  
    float32_t imagResult;  
  
    q31_t pSrcA1[10] = {1*268435456, 1*268435456, 2*268435456, 2*268435456, 3*268435456, 3*268435456,  
                        4*268435456, 4*268435456, 5*268435456, 5*268435456};  
    q31_t pSrcB1[10] = {1*268435456, 1*268435456, 2*268435456, 2*268435456, 3*268435456, 3*268435456,  
                        4*268435456, 4*268435456, 5*268435456, 5*268435456};  
    q63_t realResult1;  
    q63_t imagResult1;  
  
    q15_t pSrcA2[10] = {5000, 10000, 15000, 20000, 25000, 5000, 10000, 15000, 20000, 25000};  
    q15_t pSrcB2[10] = {5000, 10000, 15000, 20000, 25000, 5000, 10000, 15000, 20000, 25000};  
    q31_t realResult2;  
    q31_t imagResult2;  
  
    /**浮点数点乘*****  
    arm_cmplx_dot_prod_f32(pSrcA, pSrcB, 5, &realResult, &imagResult);  
    printf("arm_cmplx_dot_prod_f32:realResult = %f      imagResult = %f\r\n", realResult, imagResult);  
  
    /**定点数点乘Q31*****
```



```
arm_cmplx_dot_prod_q31(pSrcA1, pSrcB1, 5, &realResult1, &imagResult1);
printf("arm_cmplx_dot_prod_q31:realResult1 = %lld      imagResult1 = %lld\r\n", realResult1,
imagResult1);

/**定点数点乘Q15*****
arm_cmplx_dot_prod_q15(pSrcA2, pSrcB2, 5, &realResult2, &imagResult2);
printf("arm_cmplx_dot_prod_q15:realResult2 = %d      imagResult2 = %d\r\n", realResult2, imagResult2);
}
```

实验现象：



19.5 复数求模 ComplexMag

这部分函数用于复数求模，公式描述如下：

```
for (n = 0; n < numSamples; n++) {
    pDst[n] = sqrt(pSrc[(2*n)+0]^2 + pSrc[(2*n)+1]^2);
}
```

用代数式来表示复数乘法就是：

$$|a+bi| = \sqrt{a^2 + b^2}$$

19.5.1 函数 arm_cmplx_mag_f32

函数原型：

```
void arm_cmplx_mag_f32(
    const float32_t * pSrc,
    float32_t * pDst,
    uint32_t numSamples)
```

函数描述：

这个函数用于浮点数类型的复数求模。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是求模后的数据地址。
- ◆ 第 3 个参数是要求解的复数个数。

注意事项：



数组 pSrc 中存储的数据格式是（实部，虚部，实部，虚部.....），一定要按照这个顺序存储数据，比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3}。（注意第三个数据是 0）。而模值的结果存到到 pDst 里面。

19.5.2 函数 arm_cmplx_mag_q31

函数原型：

```
void arm_cmplx_mag_q31(
    const q31_t * pSrc,
    q31_t * pDst,
    uint32_t numSamples)
```

函数描述：

这个函数用于定点数 Q31 类型的复数求模。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是求模后的数据地址。
- ◆ 第 3 个参数是要求解的复数个数。

注意事项：

- ◆ 数组 pSrc 中存储的数据格式是（实部，虚部，实部，虚部.....），一定要按照这个顺序存储数据，比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3}。（注意第三个数据是 0）。而模值的结果存到到 pDst 里面。
- ◆ 1.31 格式的数据乘 1.31 格式的数据，并经过移位处理后结果是 2.30 格式。

19.5.3 函数 arm_cmplx_mag_q15

函数原型：

```
void arm_cmplx_mag_q15(
    const q15_t * pSrc,
    q15_t * pDst,
    uint32_t numSamples)
```

函数描述：

这个函数用于定点数 Q15 类型的复数求模。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是求模后的数据地址。
- ◆ 第 3 个参数是要求解的复数个数

注意事项：

- ◆ 数组 pSrc 中存储的数据格式是（实部，虚部，实部，虚部.....），一定要按照这个顺序存储数据，比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3}。（注意第三个数据是 0）。而模值的结果存到到 pDst 里面。



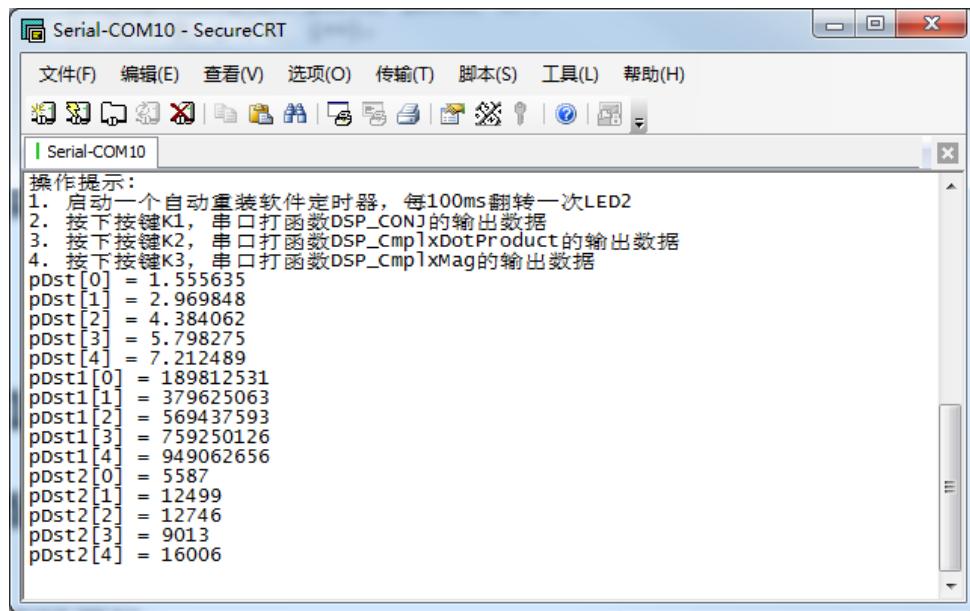
- ◆ 1.15 格式的数据乘 1.15 格式的数据，并经过移位处理后结果是 2.14 格式。

19.5.4 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_CmplxMag
* 功能说明: 复数求模
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_CmplxMag(void)
{
    uint8_t i;
    float32_t pSrc[10] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f, 5.1f};
    float32_t pDst[10];
    q31_t pSrc1[10] = {1*268435456, 1*268435456, 2*268435456, 2*268435456, 3*268435456, 3*268435456,
                        4*268435456, 4*268435456, 5*268435456, 5*268435456};
    q31_t pDst1[10];
    q15_t pSrc2[10] = {5000, 10000, 15000, 20000, 25000, 5000, 10000, 15000, 20000, 25000};
    q15_t pDst2[10];
    /*浮点数求模*******/
    arm_cmplx_mag_f32(pSrc, pDst, 5);
    for(i = 0; i < 5; i++)
    {
        printf("pDst[%d] = %f\r\n", i, pDst[i]);
    }
    /*定点数求模Q31*******/
    arm_cmplx_mag_q31(pSrc1, pDst1, 5);
    for(i = 0; i < 5; i++)
    {
        printf("pDst1[%d] = %d\r\n", i, pDst1[i]);
    }
    /*定点数求模Q15*******/
    arm_cmplx_mag_q15(pSrc2, pDst2, 5);
    for(i = 0; i < 5; i++)
    {
        printf("pDst2[%d] = %d\r\n", i, pDst2[i]);
    }
}
```

实验现象：



```
操作提示:
1. 启动一个自动重装软件定时器，每100ms翻转一次LED2
2. 按下按键K1，串口打函数DSP_CONJ的输出数据
3. 按下按键K2，串口打函数DSP_CmplxDotProduct的输出数据
4. 按下按键K3，串口打函数DSP_CmplxMag的输出数据
pdst[0] = 1.555635
pdst[1] = 2.969848
pdst[2] = 4.384062
pdst[3] = 5.798275
pdst[4] = 7.212489
pdst1[0] = 189812531
pdst1[1] = 379625063
pdst1[2] = 569437593
pdst1[3] = 759250126
pdst1[4] = 949062656
pdst2[0] = 5587
pdst2[1] = 12499
pdst2[2] = 12746
pdst2[3] = 9013
pdst2[4] = 16006
```

19.6 实验例程说明 (MDK)

配套例子：

V7-214_DSP 复数运算 (共轭, 点乘和求模)

实验目的：

1. 学习 DSP 复数运算 (共轭, 点乘和求模)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打函数 DSP_CONJ 的输出数据。
3. 按下按键 K2，串口打函数 DSP_CmplxDotProduct 的输出数据。
4. 按下按键 K3，串口打函数 DSP_CmplxMag 的输出数据。

使用 AC6 注意事项

特别注意附件章节 C 的问题

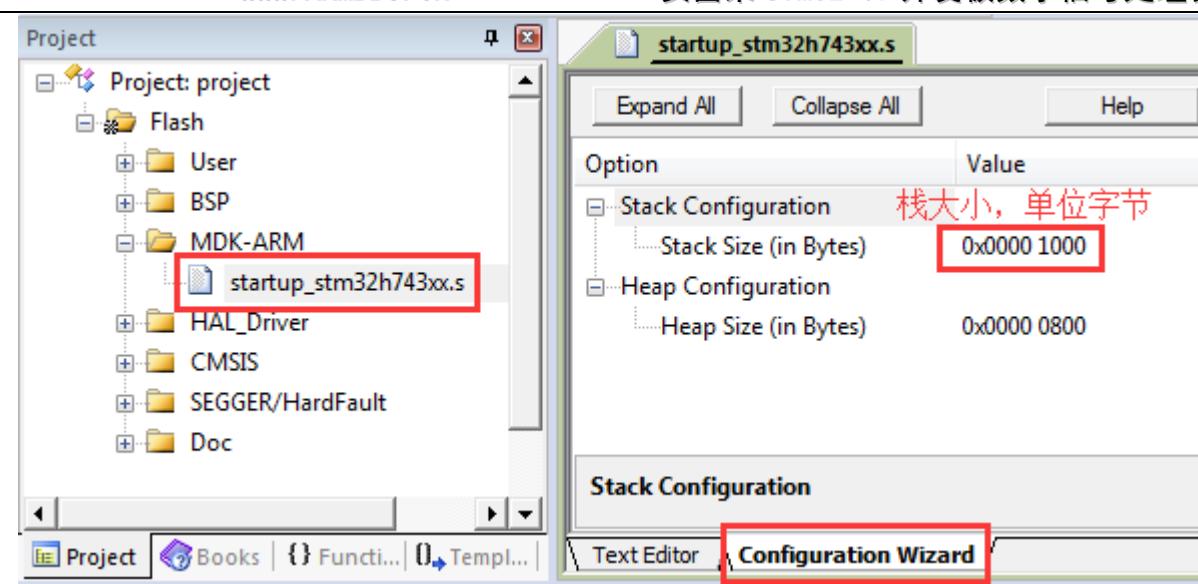
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

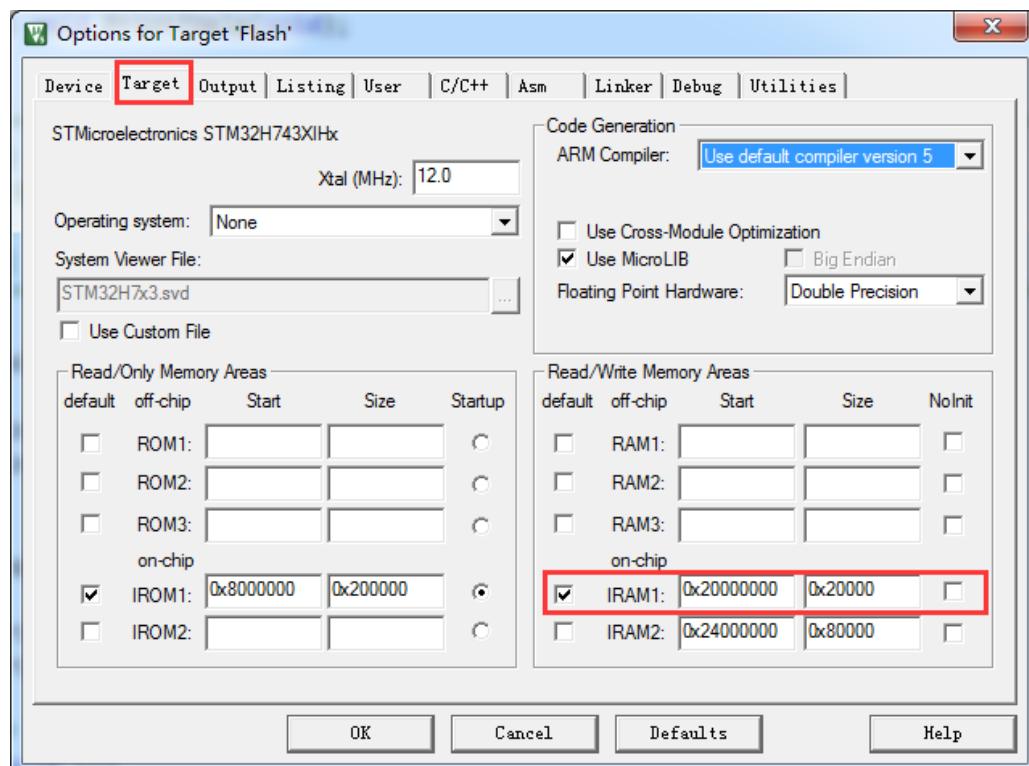
详见本章的 3.4 4.4, 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现:

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
```



```
{  
    /* 配置 MPU */  
    MPU_Config();  
  
    /* 使能 L1 Cache */  
    CPU_CACHE_Enable();  
  
    /*  
     * STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：  
     * - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。  
     * - 设置 NVIC 优先级分组为 4。  
     */  
    HAL_Init();  
  
    /*  
     * 配置系统时钟到 400MHz  
     * - 切换使用 HSE。  
     * - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。  
     */  
    SystemClock_Config();  
  
    /*  
     * Event Recorder:  
     * - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。  
     * - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章  
     */  
#if Enable_EventRecorder == 1  
    /* 初始化 EventRecorder 并开启 */  
    EventRecorderInitialize(EventRecordAll, 1U);  
    EventRecorderStart();  
#endif  
  
    bsp_InitKey(); /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */  
    bsp_InitTimer(); /* 初始化滴答定时器 */  
    bsp_InitUart(); /* 初始化串口 */  
    bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */  
    bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();
```



```
/* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。



- 按下按键 K1，串口打函数 DSP_CONJ 的输出数据。
- 按下按键 K2，串口打函数 DSP_CmplxDotProduct 的输出数据。
- 按下按键 K3，串口打函数 DSP_CmplxMag 的输出数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形    参: 无
***** 返  回 值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下, 求共轭 */
                    DSP_CONJ();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下, 求点乘 */
                    DSP_CmplxDotProduct();
                    break;

                case KEY_DOWN_K3:           /* K3 键按下, 求模 */
                    DSP_CmplxMag();
                    break;

                default:
                    /* 其他的键值不处理 */
                    break;
            }
        }
    }
}
```



```
        }  
    }  
}
```

19.7 实验例程说明 (IAR)

配套例子：

V7-214_DSP 复数运算 (共轭, 点乘和求模)

实验目的：

1. 学习 DSP 复数运算 (共轭, 点乘和求模)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打函数 DSP_CONJ 的输出数据。
3. 按下按键 K2，串口打函数 DSP_CmplxDotProduct 的输出数据。
4. 按下按键 K3，串口打函数 DSP_CmplxMag 的输出数据。

使用 AC6 注意事项

特别注意附件章节 C 的问题

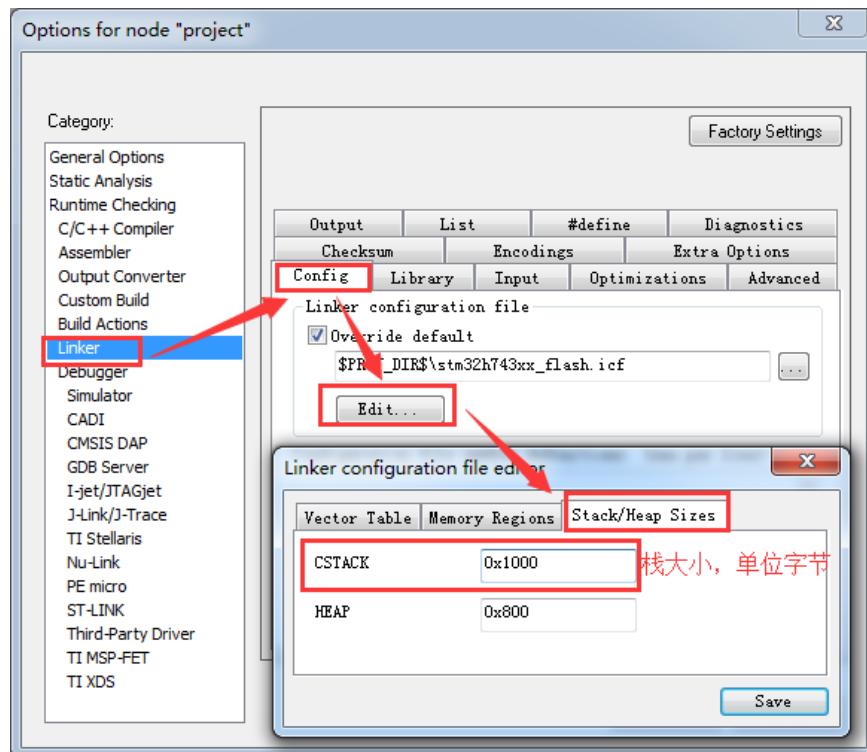
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

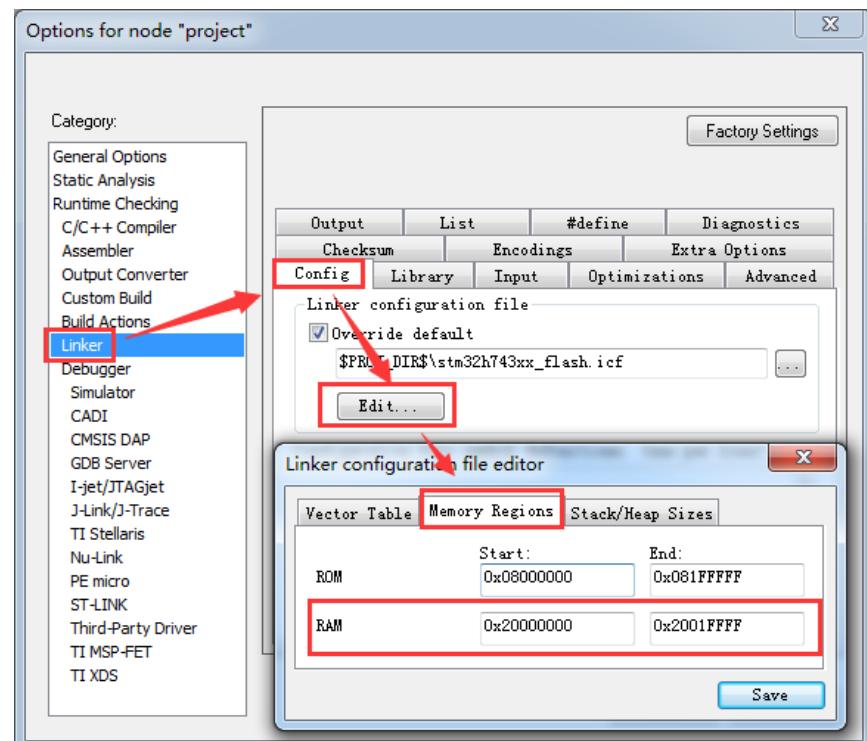
详见本章的 3.4 4.4, 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
*****
```



```
/* 功能说明：初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形    参：无
* 返回值：无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
     STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
     - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
     - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
     配置系统时钟到 400MHz
     - 切换使用 HSE。
     - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
     Event Recorder:
     - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
     - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名：MPU_Config
* 功能说明：配置 MPU
* 形    参：无
* 返回值：无
*****
*/
static void MPU_Config( void )
```



```
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec   = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec   = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /*使能 MPU */  
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);  
}  
  
/*  
*****  
* 函数名: CPU_CACHE_Enable  
* 功能说明: 使能 L1 Cache  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void CPU_CACHE_Enable(void)  
{  
    /* 使能 I-Cache */  
    SCB_EnableICache();  
  
    /* 使能 D-Cache */  
    SCB_EnableDCache();  
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打函数 DSP_CONJ 的输出数据。
- 按下按键 K2，串口打函数 DSP_CmplxDotProduct 的输出数据。
- 按下按键 K3，串口打函数 DSP_CmplxMag 的输出数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下, 求共轭 */
                    DSP_CONJ();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下, 求点乘 */
                    DSP_CmplxDotProduct();
                    break;

                case KEY_DOWN_K3:           /* K3 键按下, 求模 */
                    DSP_CmplxMag();
                    break;
            }
        }
    }
}
```



```
        break;  
  
    default:  
        /* 其他的键值不处理 */  
        break;  
    }  
}  
}
```

19.8 总结

本期教程就跟大家讲这么多，有兴趣的可以深入研究下算法的具体实现。



第20章 DSP 复数运算-模平方，乘法和复数乘实

数

本期教程主要讲解复数运算中的模平方，乘法和复数乘实数。

20.1 初学者重要提示

20.2 DSP 基础运算指令

20.3 复数模平方 (ComplexMagSquared)

20.4 复数乘法 (ComplexMultComplex)

20.5 复数乘实数 (ComplexMultComplex)

20.6 实验例程说明 (MDK)

20.7 实验例程说明 (IAR)

20.8 总结

20.1 初学者重要提示

- ◆ 复数运算比较重要，后面 FFT 章节要用到，如果印象不深的话，需要温习下高数知识了。

20.2 DSP 基础运算指令

本章用到的 DSP 指令在前面章节都已经讲解过。

20.3 复数模平方 (ComplexMagSquared)

这部分函数用于复数求模平方，公式描述如下：

```
for(n=0; n<numSamples; n++) {  
    pDst[n] = pSrc[(2*n)+0]^2 + pSrc[(2*n)+1]^2;  
}
```

用代数式来表示模平方：

$a+bi$ 模平方 = $a^2 + b^2$ 。



20.3.1 函数 arm_cmplx_mag_squared_f32

函数原型:

```
void arm_cmplx_mag_f32(
    const float32_t * pSrc,
    float32_t * pDst,
    uint32_t numSamples)
```

函数描述:

这个函数用于浮点数类型的复数求模平方。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是求模平方后的数据地址。
- ◆ 第 3 个参数是要求解的复数个数。

注意事项:

数组 pSrc 中存储的数据格式是 (实部, 虚部, 实部, 虚部.....) , 一定要按照这个顺序存储数据, 比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是: pSrc[6] = {1, -1, 0, 1, 2, 3} (**注意第三个数据是 0**) 。而模值的结果存到 pDst 里面。

20.3.2 函数 arm_cmplx_mag_squared_q31

函数原型:

```
void arm_cmplx_mag_squared_q31(
    const q31_t * pSrc,
    q31_t * pDst,
    uint32_t numSamples)
```

函数描述:

这个函数用于定点数 Q31 类型的复数求模平方。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是求模平方后的数据地址。
- ◆ 第 3 个参数是要求解的复数个数。

注意事项:

1. 两个1.31格式的定点数相乘为2.62, 程序中将此结果做了放缩, 此函数的最终结果转换后为3.29。
2. 数组pSrc中存储的数据格式是 (实部, 虚部, 实部, 虚部.....) , 一定要按照这个顺序存储数据, 比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是: pSrc[6] = {1, -1, 0, 1, 2, 3} (**注意第三个数据是0**) 。而模值的结果存到pDst里面。

20.3.3 函数 arm_cmplx_mag_squared_q15

函数原型:

```
void arm_cmplx_mag_squared_q15(
    const q15_t * pSrc,
```



```
q15_t * pDst,  
      uint32_t numSamples)
```

函数描述：

这个函数用于定点数 Q15 类型的复数求模平方。

函数参数：

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是求模平方后的数据地址。
- ◆ 第 3 个参数是要求解的复数个数。

注意事项：

1. 两个1.15格式的定点数相乘为2.30，程序中将此结果做了放缩，此函数的最终结果转换后为3.13。
2. 数组pSrc中存储的数据格式是（实部，虚部，实部，虚部.....），一定要按照这个顺序存储数据，比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3}（注意第三个数据是0）。而模值的结果存到pDst里面。

20.3.4 使用举例

程序设计：

```
/*  
*****  
* 函数名: DSP_MagSquared  
* 功能说明: 复数模的平方  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_MagSquared(void)  
{  
    uint8_t i;  
    float32_t pSrc[10] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f, 5.1f};  
    float32_t pDst[10];  
  
    q31_t pSrc1[10] = {1*268435456, 1*268435456, 2*268435456, 2*268435456, 3*268435456, 3*268435456,  
                        4*268435456, 4*268435456, 5*268435456, 5*268435456};  
    q31_t pDst1[10];  
  
    q15_t pSrc2[10] = {5000, 10000, 15000, 20000, 25000, 5000, 10000, 15000, 20000, 25000};  
    q15_t pDst2[10];  
  
    /**浮点数模平方*****  
    arm_cmplx_mag_squared_f32(pSrc, pDst, 5);  
    for(i = 0; i < 5; i++)  
    {  
        printf("pDst[%d] = %f\r\n", i, pDst[i]);  
    }  
  
    /**定点数模平方Q31*****  
    arm_cmplx_mag_squared_q31(pSrc1, pDst1, 5);  
    for(i = 0; i < 5; i++)  
    {  
        printf("pDst1[%d] = %d\r\n", i, pDst1[i]);  
    }  
  
    /**定点数模平方Q15*****  
    arm_cmplx_mag_squared_q15(pSrc2, pDst2, 5);  
    for(i = 0; i < 5; i++)  
    {  
        printf("pDst2[%d] = %d\r\n", i, pDst2[i]);  
    }
```



```
}

/*
* 函数名: DSP_MatInit
* 功能说明: 矩阵数据初始化
* 形参: 无
* 返回值: 无
*/
static void DSP_MatInit(void)
{
    uint8_t i;

    /***浮点数数组*****
    float32_t pDataA[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};

    arm_matrix_instance_f32 pSrcA; //3行3列数据

    /***定点数Q31数组*****
    q31_t pDataA1[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};

    arm_matrix_instance_q31 pSrcA1; //3行3列数据

    /***定点数Q15数组*****
    q15_t pDataA2[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};

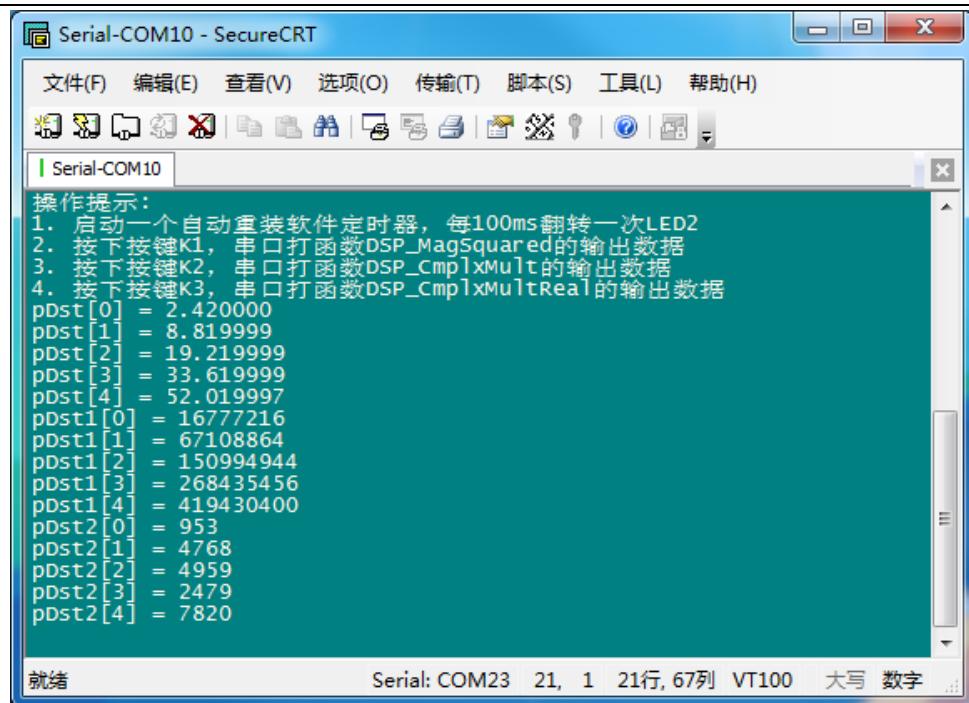
    arm_matrix_instance_q15 pSrcA2; //3行3列数据

    /***浮点数*****
    printf("****浮点数*****\r\n");
    arm_mat_init_f32(&pSrcA, 3, 3, pDataA);
    for(i = 0; i < 9; i++)
    {
        printf("pDataA[%d] = %f\r\n", i, pDataA[i]);
    }

    /***定点数Q31*****
    printf("****浮点数*****\r\n");
    arm_mat_init_q31(&pSrcA1, 3, 3, pDataA1);
    for(i = 0; i < 9; i++)
    {
        printf("pDataA1[%d] = %d\r\n", i, pDataA1[i]);
    }

    /***定点数Q15*****
    printf("****浮点数*****\r\n");
    arm_mat_init_q15(&pSrcA2, 3, 3, pDataA2);
    for(i = 0; i < 9; i++)
    {
        printf("pDataA2[%d] = %d\r\n", i, pDataA2[i]);
    }
}
```

实验现象 (按下 K1 按键后串口打印模平方):



20.4 复数乘法 (ComplexMultComplex)

这部分函数用于复数乘复数，公式描述如下：

```
for (n = 0; n < numSamples; n++) {
    pDst[(2*n)+0] = pSrcA[(2*n)+0] * pSrcB[(2*n)+0] - pSrcA[(2*n)+1] * pSrcB[(2*n)+1];
    pDst[(2*n)+1] = pSrcA[(2*n)+0] * pSrcB[(2*n)+1] + pSrcA[(2*n)+1] * pSrcB[(2*n)+0];
}
```

用代数式来表示复数乘法：

$$(a+bi) * (c+di) = (ac - bd) + (ad + bc)i.$$

20.4.1 函数 arm_cmplx_mult_cmplx_f32

函数原型：

```
void arm_cmplx_mult_cmplx_f32(
    const float32_t * pSrcA,
    const float32_t * pSrcB,
    float32_t * pDst,
    uint32_t numSamples)
```

函数描述：

这个函数用于浮点数的复数乘复位。

函数参数：

- ◆ 第 1 个参数是源数据 A 的地址。
- ◆ 第 2 个参数是源数据 B 的地址。



- ◆ 第 3 个参数是存储数组 A 和数组 B 乘积地址。
- ◆ 第 4 个参数是要求解的复数个数。

注意事项：

数组 pSrcA, pSrcB 和 pDst 中存储的数据格式是（实部，虚部，实部，虚部.....），源数据 A 和 B 一定要按照这个顺序存储数据，比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3} (注意第三个数据是 0)。而乘积的结果存到 pDst 里面。

20.4.2 函数 arm_cmplx_mult_cmplx_q31

函数原型：

```
void arm_cmplx_mult_cmplx_q31(
    const q31_t * pSrcA,
    const q31_t * pSrcB,
    q31_t * pDst,
    uint32_t numSamples)
```

函数描述：

这个函数用于 Q31 格式定点数的复数乘复数。

函数参数：

- ◆ 第 1 个参数是源数据 A 的地址。
- ◆ 第 2 个参数是源数据 B 的地址。
- ◆ 第 3 个参数是存储数组 A 和数组 B 乘积地址。
- ◆ 第 4 个参数是要求解的复数个数。

注意事项：

1. 两个 1.31 格式的定点数相乘为 2.62，程序中将此结果做了放缩，此函数的最终结果转换后为 3.29。
2. 数组 pSrcA, pSrcB 和 pDst 中存储的数据格式是（实部，虚部，实部，虚部.....），源数据 A 和 B 一定要按照这个顺序存储数据，比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3} (注意第三个数据是 0)。而乘积的结果存到 pDst 里面。

20.4.3 函数 arm_cmplx_mult_cmplx_f15

函数原型：

```
void arm_cmplx_mult_cmplx_f32(
    const float32_t * pSrcA,
    const float32_t * pSrcB,
    float32_t * pDst,
    uint32_t numSamples)
```

函数描述：

这个函数用于 Q15 格式定点数的复数乘复数。

函数参数：

- ◆ 第 1 个参数是源数据 A 的地址。



- ◆ 第 2 个参数是源数据 B 的地址。
- ◆ 第 3 个参数是存储数组 A 和数组 B 乘积地址。
- ◆ 第 4 个参数是要求解的复数个数。

注意事项：

1. 两个 1.15 格式的定点数相乘为 2.30，程序中将此结果做了放缩，此函数的最终结果转换后为 3.13。
2. 数组 pSrcA, pSrcB 和 pDst 中存储的数据格式是（实部，虚部，实部，虚部.....），源数据 A 和 B 一定要按照这个顺序存储数据，比如数据 1-j, j, 2+3j 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3} (注意第三个数据是 0)。而乘积的结果存到 pDst 里面。

20.4.4 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_CmplxMult
* 功能说明: 复数乘法
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_CmplxMult(void)
{
    uint8_t i;
    float32_t pSrcA[10] = {1.1f, 1.2f, 2.1f, 2.2f, 3.1f, 3.2f, 4.1f, 4.2f, 5.1f, 5.2f};
    float32_t pSrcB[10] = {1.2f, 1.2f, 2.2f, 2.2f, 3.2f, 3.2f, 4.2f, 4.2f, 5.2f, 5.2f};
    float32_t pDst[10];

    q31_t pSrcA1[10] = {1*268435456, 1*268435456, 2*268435456, 2*268435456, 3*268435456, 3*268435456,
                        4*268435456, 4*268435456, 5*268435456, 5*268435456};
    q31_t pSrcB1[10] = {1*268435456, 1*268435456, 2*268435456, 2*268435456, 3*268435456, 3*268435456,
                        4*268435456, 4*268435456, 5*268435456, 5*268435456};
    q31_t pDst1[10];

    q15_t pSrcA2[10] = {5000, 10000, 15000, 20000, 25000, 5000, 10000, 15000, 20000, 25000};
    q15_t pSrcB2[10] = {6000, 11000, 15000, 20000, 25000, 5000, 10000, 15000, 20000, 25000};
    q15_t pDst2[10];

    /***浮点数乘法*****
    arm_cmplx_mult_cmplx_f32(pSrcA, pSrcB, pDst, 5);
    for(i = 0; i < 5; i++)
    {
        printf("pDst[%d] = %f %fj\r\n", i, pDst[2*i], pDst[2*i+1]);
    }

    /***定点数乘法Q31*****
    arm_cmplx_mult_cmplx_q31(pSrcA1, pSrcB1, pDst1, 5);
    for(i = 0; i < 5; i++)
    {
        printf("pDst1[%d] = %d %dj\r\n", i, pDst1[2*i], pDst1[2*i+1]);
    }

    /***定点数乘法Q15*****
    arm_cmplx_mult_cmplx_q15(pSrcA2, pSrcB2, pDst2, 5);
    for(i = 0; i < 5; i++)
    {
        printf("pDst1[%d] = %d %dj\r\n", i, pDst2[2*i], pDst2[2*i+1]);
    }
}
```

实验现象 (按下 K2 按键后串口打印复数乘法):



操作提示：
1. 启动一个自动重装软件定时器，每100ms翻转一次LED2
2. 按下按键K1，串口打函数DSP_MagSquared的输出数据
3. 按下按键K2，串口打函数DSP_CmplxMult的输出数据
4. 按下按键K3，串口打函数DSP_CmplxMultReal的输出数据

就绪 Serial: COM23 21, 1 21行, 67列 VT100 大写 数字

20.5 复数乘实数 (ComplexMultReal)

这部分函数用于复数乘实数，公式描述如下：

```
for(n=0; n<numSamples; n++) {  
    pCmplxDst[(2*n)+0] = pSrcCmplx[(2*n)+0] * pSrcReal[n];  
    pCmplxDst[(2*n)+1] = pSrcCmplx[(2*n)+1] * pSrcReal[n];  
}
```

用代数式来表示复数乘法：

$$a^* (c+di) = ac + adi.$$

20.5.1 函数 arm_cmplx_mult_real_f32

函数原型：

```
void arm_cmplx_mult_real_f32(  
    const float32_t * pSrcCmplx,  
    const float32_t * pSrcReal,  
    float32_t * pCmplxDst,  
    uint32_t numSamples)
```

函数描述：

这个函数用于浮点数的复数乘实数。

函数参数：

- ◆ 第 1 个参数是复数的源地址。
- ◆ 第 2 个参数是实数的源地址。



- ◆ 第 3 个参数是复数和实数乘积地址。
- ◆ 第 4 个参数是要进行复数乘实数的个数。

注意事项：

1. 数组 pSrcCmplx 和 pCmplxDst 中存储的数据格式是 (实部, 虚部, 实部, 虚部.....) , 源数据 pSrcCmplx 一定要按照这个顺序存储数据, 比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是: pSrc[6] = {1, -1, 0, 1, 2, 3} (注意第三个数据是 0) 。而乘积的结果存到 pCmplxDst 里面。

20.5.2 函数 arm_cmplx_mult_real_q31

函数原型：

```
void arm_cmplx_mult_real_q31(
    const q31_t * pSrcCmplx,
    const q31_t * pSrcReal,
    q31_t * pCmplxDst,
    uint32_t numSamples)
```

函数描述：

这个函数用于 Q31 格式定点数的复数乘实数。

函数参数：

- ◆ 第 1 个参数是复数的源地址。
- ◆ 第 2 个参数是实数的源地址。
- ◆ 第 3 个参数是复数和实数乘积地址。
- ◆ 第 4 个参数是要进行复数乘实数的个数。

注意事项：

1. 输出结果做了饱和运算, 输出范围[0x80000000 0x7FFFFFFF]。
2. 数组 pSrcCmplx 和 pCmplxDst 中存储的数据格式是 (实部, 虚部, 实部, 虚部.....) , 源数据 pSrcCmplx 一定要按照这个顺序存储数据, 比如数据 $1-j, j, 2+3j$ 这个三个数在数组中的存储格式就是: pSrc[6] = {1, -1, 0, 1, 2, 3} (注意第三个数据是 0) 。而乘积的结果存到 pCmplxDst 里面。

20.5.3 函数 arm_cmplx_mult_real_q15

函数原型：

```
void arm_cmplx_mult_real_q15(
    const q15_t * pSrcCmplx,
    const q15_t * pSrcReal,
    q15_t * pCmplxDst,
    uint32_t numSamples)
```

函数描述：

这个函数用于 Q15 格式定点数的复数乘实数。

函数参数：



- ◆ 第1个参数是复数的源地址。
- ◆ 第2个参数是实数的源地址。
- ◆ 第3个参数是复数和实数乘积地址。
- ◆ 第4个参数是要进行复数乘实数的个数。

注意事项：

1. 输出结果做了饱和运算，输出范围[0x8000 0x7FFF]。
2. 数组 pSrcCmplx 和 pCmplxDst 中存储的数据格式是（实部，虚部，实部，虚部.....），源数据 pSrcCmplx 一定要按照这个顺序存储数据，比如数据 1-j, j, 2+3j 这个三个数在数组中的存储格式就是：pSrc[6] = {1, -1, 0, 1, 2, 3} (**注意第三个数据是0**)。而乘积的结果存到 pCmplxDst 里面。

20.5.4 使用举例

程序设计：

```
/*
*****
* 函数名: DSP_CmplxMultReal
* 功能说明: 复数乘实数
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_CmplxMultReal(void)
{
    uint8_t i;
    float32_t pSrcCmplx[10] = {1.1f, 1.2f, 2.1f, 2.2f, 3.1f, 3.2f, 4.1f, 4.2f, 5.1f, 5.2f};
    float32_t pSrcReal[5] = {1.2f, 1.2f, 2.2f, 2.2f, 3.2f};
    float32_t pCmplxDst[10];

    q31_t pSrcCmplx1[10] = {1*268435456, 1*268435456, 2*268435456, 2*268435456, 3*268435456, 3*268435456,
                           4*268435456, 4*268435456, 5*268435456, 5*268435456};

    q31_t pSrcReal1[10] = {1*268435456, 1*268435456, 2*268435456, 2*268435456, 3*268435456, 3*268435456};
    q31_t pCmplxDst1[10];

    q15_t pSrcCmplx2[10] = {14000, 16000, 20000, 20000, 30000, 31000, 12000, 13000, 14000, 25000};
    q15_t pSrcReal2[10] = {15000, 17000, 20000, 20000, 30000};
    q15_t pCmplxDst2[10];

    /**浮点数*****
arm_cmplx_mult_cmplx_f32(pSrcCmplx, pSrcReal, pCmplxDst, 5);
for(i = 0; i < 5; i++)
{
    printf("pCmplxDst[%d] = %f %f j\r\n", i, pCmplxDst[2*i], pCmplxDst[2*i+1]);
}

    /**定点数Q31*****
arm_cmplx_mult_cmplx_q31(pSrcCmplx1, pSrcReal1, pCmplxDst1, 5);
for(i = 0; i < 5; i++)
{
    printf("pCmplxDst1[%d] = %d %d j\r\n", i, pCmplxDst1[2*i], pCmplxDst1[2*i+1]);
}

    /**定点数Q15*****
arm_cmplx_mult_cmplx_q15(pSrcCmplx2, pSrcReal2, pCmplxDst2, 5);
for(i = 0; i < 5; i++)
{
    printf("pCmplxDst2[%d] = %d %d j\r\n", i, pCmplxDst2[2*i], pCmplxDst2[2*i+1]);
}
}
```



实验现象 (按下 K3 按键后串口打印实数乘复数):

```
操作提示:
1. 启动一个自动重装软件定时器, 每100ms翻转一次LED2
2. 按下按键K1, 串口打函数DSP_MagSquared的输出数据
3. 按下按键K2, 串口打函数DSP_CmplxMult的输出数据
4. 按下按键K3, 串口打函数DSP_CmplxMultReal的输出数据
pcmplxDst[0] = -0.120000 2.760000j
pcmplxDst[1] = -0.220000 9.460000j
pcmplxDst[2] = 9.920000 10.240000j
pcmplxDst[3] = 0.000000 0.000000j
pcmplxDst[4] = 0.000000 0.000000j
pcmplxDst1[0] = 0 16777216j
pcmplxDst1[1] = 0 67108864j
pcmplxDst1[2] = 75497472 75497472j
pcmplxDst1[3] = 0 0j
pcmplxDst1[4] = 0 0j
pcmplxDst2[0] = -473 3646j
pcmplxDst2[1] = 0 6102j
pcmplxDst2[2] = 6866 7095j
pcmplxDst2[3] = 0 0j
pcmplxDst2[4] = 0 0j
```

20.6 实验例程说明 (MDK)

配套例子:

V7-215_DSP 复数运算 (模平方, 复数乘复数和复数乘实数)

实验目的:

1. 学习 DSP 复数运算 (模平方, 复数乘复数和复数乘实数)

实验内容:

1. 启动一个自动重装软件定时器, 每 100ms 翻转一次 LED2。
2. 按下按键 K1, 串口打函数 DSP_MagSquared 的输出数据。
3. 按下按键 K2, 串口打函数 DSP_CmplxMult 的输出数据。
4. 按下按键 K3, 串口打函数 DSP_CmplxMultReal 的输出数据。

使用 AC6 注意事项

特别注意附件章节 C 的问题

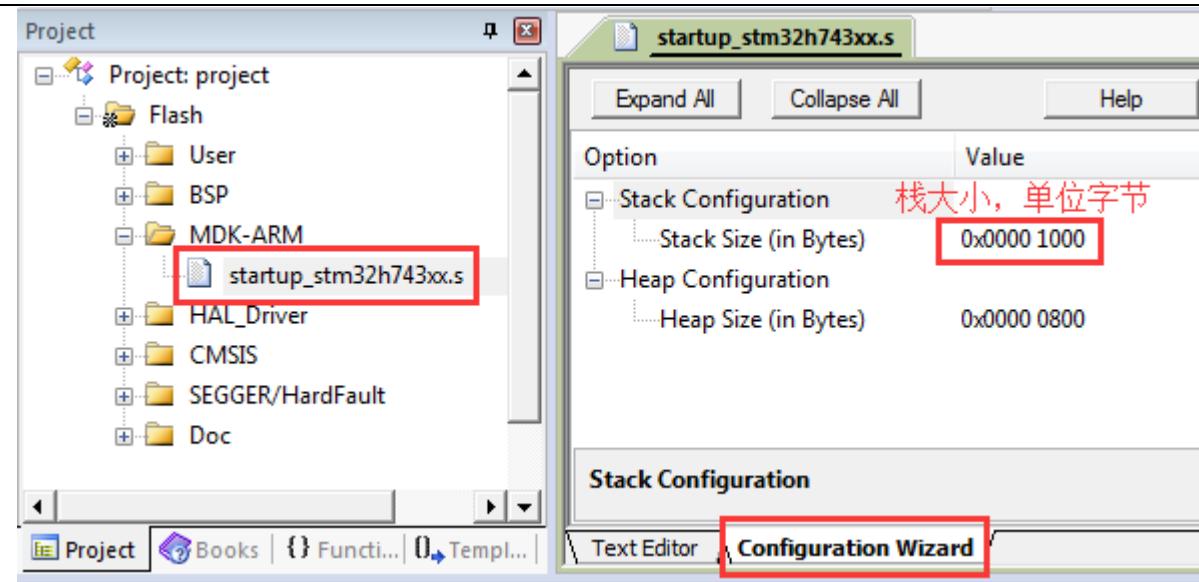
上电后串口打印的信息:

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

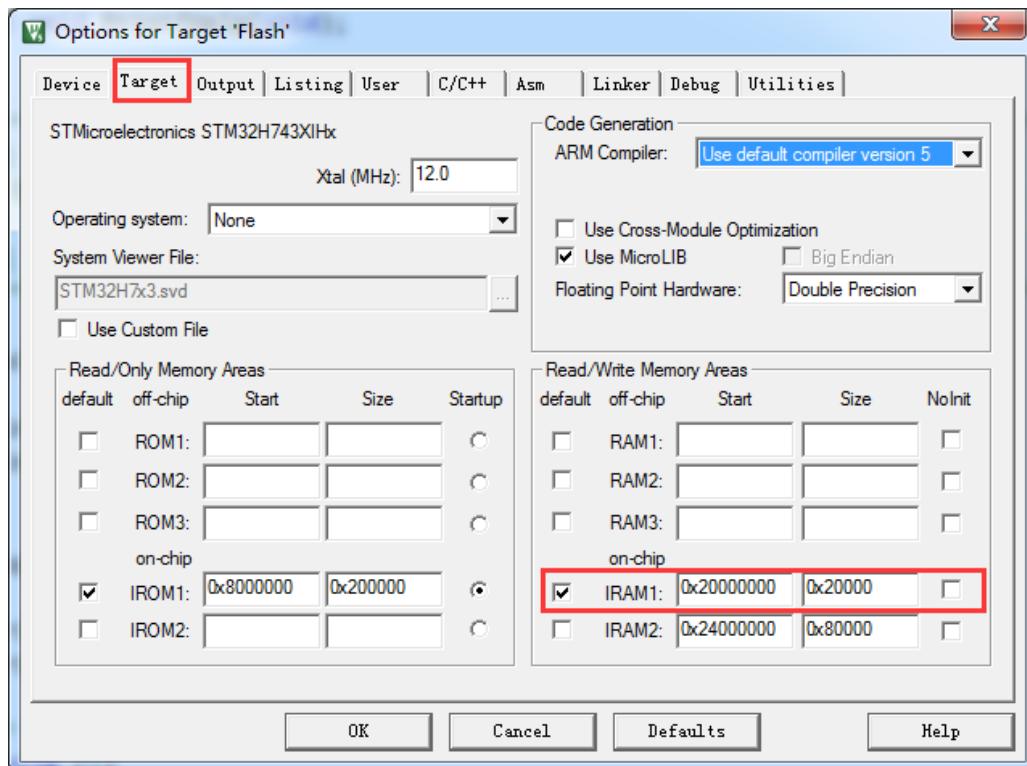
详见本章的 3.4 4.4, 5.4 小节。

程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现:

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
```



```
{  
    /* 配置 MPU */  
    MPU_Config();  
  
    /* 使能 L1 Cache */  
    CPU_CACHE_Enable();  
  
    /*  
     * STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：  
     * - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。  
     * - 设置 NVIC 优先级分组为 4。  
     */  
    HAL_Init();  
  
    /*  
     * 配置系统时钟到 400MHz  
     * - 切换使用 HSE。  
     * - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。  
     */  
    SystemClock_Config();  
  
    /*  
     * Event Recorder:  
     * - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。  
     * - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章  
     */  
#if Enable_EventRecorder == 1  
    /* 初始化 EventRecorder 并开启 */  
    EventRecorderInitialize(EventRecordAll, 1U);  
    EventRecorderStart();  
#endif  
  
    bsp_InitKey(); /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */  
    bsp_InitTimer(); /* 初始化滴答定时器 */  
    bsp_InitUart(); /* 初始化串口 */  
    bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */  
    bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();
```



```
/* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
```

```
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;
```

```
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

```
/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
```

```
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;
```

```
HAL_MPU_ConfigRegion(&MPU_InitStruct);
```

```
/*使能 MPU */
```

```
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
```

```
}
```

```
/*
```

```
*****
```

```
* 函数名: CPU_CACHE_Enable
```

```
* 功能说明: 使能 L1 Cache
```

```
* 形参: 无
```

```
* 返回值: 无
```

```
*****
```

```
*/
```

```
static void CPU_CACHE_Enable(void)
```

```
{
```

```
/* 使能 I-Cache */
```

```
SCB_EnableICache();
```

```
/* 使能 D-Cache */
```

```
SCB_EnableDCache();
```

```
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。



- 按下按键 K1，串口打函数 DSP_MagSquared 的输出数据。
- 按下按键 K2，串口打函数 DSP_CmplxMult 的输出数据。
- 按下按键 K3，串口打函数 DSP_CmplxMultReal 的输出数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形    参: 无
***** 返  回 值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下, 求模平方 */
                    DSP_MagSquared();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下, 求复数乘复数 */
                    DSP_CmplxMult();
                    break;

                case KEY_DOWN_K3:           /* K3 键按下, 求复数乘实数 */
                    DSP_CmplxMultReal();
                    break;

                default:
                    /* 其他的键值不处理 */
                    break;
            }
        }
    }
}
```

```
}
```

20.7 实例程说明 (IAR)

配套例子：

V7-215_DSP 复数运算 (模平方, 复数乘复数和复数乘实数)

实验目的：

1. 学习 DSP 复数运算 (模平方, 复数乘复数和复数乘实数)

实验内容：

1. 启动一个自动重装软件定时器, 每 100ms 翻转一次 LED2。
2. 按下按键 K1, 串口打函数 DSP_MagSquared 的输出数据。
3. 按下按键 K2, 串口打函数 DSP_CmplxMult 的输出数据。
4. 按下按键 K3, 串口打函数 DSP_CmplxMultReal 的输出数据。

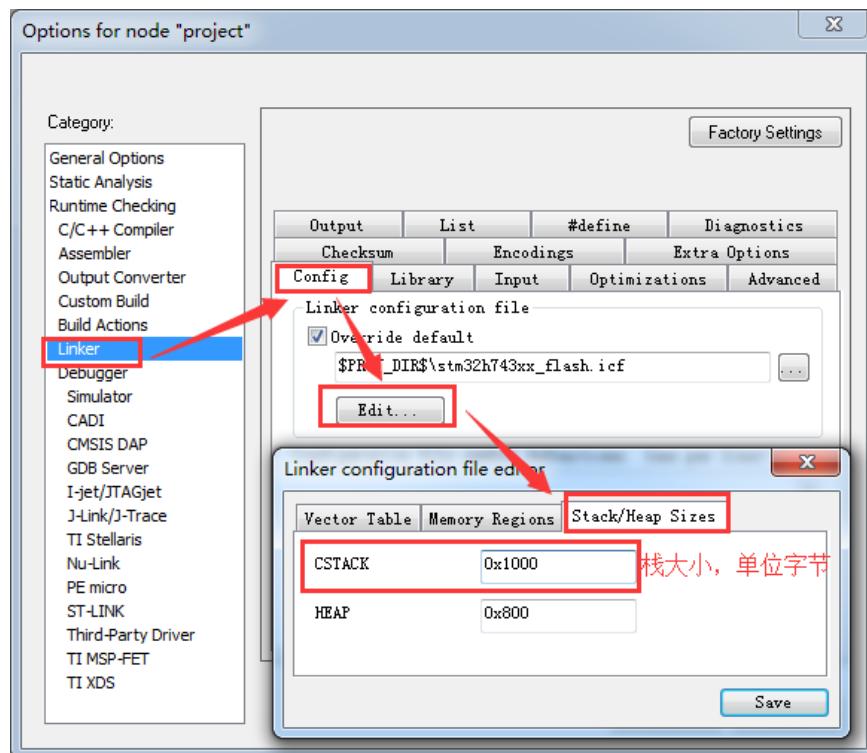
上电后串口打印的信息：

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

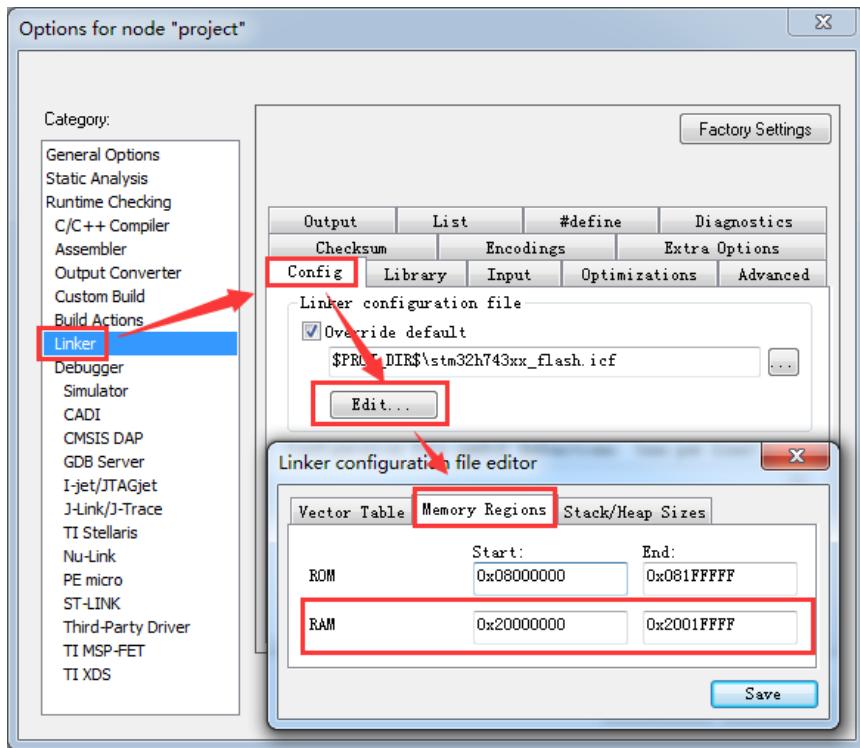
详见本章的 3.4 4.4, 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
     * STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
     * - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
     * - 设置 NVIC 优先级分组为 4。
     */
    HAL_Init();

    /*
     * 配置系统时钟到 400MHz
     * - 切换使用 HSE。
     * - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
     */
}
```



```
SystemClock_Config();

/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
```



```
MPU_InitStruct.Size          = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable    = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable     = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number         = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打函数 DSP_MagSquared 的输出数据。
- 按下按键 K2，串口打函数 DSP_CmplxMult 的输出数据。
- 按下按键 K3，串口打函数 DSP_CmplxMultReal 的输出数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */
}
```



```
PrintfHelp(); /* 打印操作提示信息 */
```

```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下，求模平方 */
                DSP_MagSquared();
                break;

            case KEY_DOWN_K2:          /* K2 键按下，求复数乘复数 */
                DSP_CmplxMult();
                break;

            case KEY_DOWN_K3:          /* K3 键按下，求复数乘实数 */
                DSP_CmplxMultReal();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}
```

20.8 总结

本期教程就跟大家讲这么多，有兴趣的可以深入研究下算法的具体实现。



第21章 DSP 矩阵运算-加法，减法和逆矩阵

本期教程主要讲解矩阵运算中的初始化，加法，逆矩阵和减法。

21.1 初学者重要提示

21.2 DSP 基础运算指令

21.3 矩阵初始化 (MatInit)

21.4 矩阵加法 (MatAdd)

21.5 矩阵减法 (MatSub)

21.6 逆矩阵 (MatInverse)

21.7 实验例程说明 (MDK)

21.8 实验例程说明 (IAR)

21.9 总结

21.1 初学者重要提示

- ◆ 复数运算比较重要，后面 FFT 章节要用到，如果印象不深的话，需要温习下高数知识了。
- ◆ ARM 提供的 DSP 库逆矩阵求法有局限性，通过 Matlab 验证是可以求逆矩阵的，而 DSP 库却不能正确求解。

21.2 DSP 基础运算指令

本章用到的 DSP 指令在前面章节都已经讲解过。

21.3 矩阵初始化 (MatInit)

主要用于矩阵结构体成员的初始化，浮点格式矩阵结构体定义如下：

```
typedef struct
{
    uint16_t numRows;      // 矩阵行数.
    uint16_t numCols;      // 矩阵列数
    float32_t *pData;      // 矩阵地址
} arm_matrix_instance_f32
```

定点数 Q31 格式矩阵结构体定义如下：

```
typedef struct
{
    uint16_t numRows;      //矩阵行数
    uint16_t numCols;      //矩阵列数
```



```
q31_t *pData;           //矩阵地址  
} arm_matrix_instance_q31;
```

定点数 Q15 格式矩阵结构体定义如下：

```
typedef struct  
{  
    uint16_t numRows;      //矩阵行数  
    uint16_t numCols;      //矩阵列数  
    q15_t *pData;          //矩阵地址  
} arm_matrix_instance_q15;
```

21.3.1 函数 arm_mat_init_f32

函数原型：

```
void arm_mat_init_f32(  
    arm_matrix_instance_f32 * S,  
    uint16_t nRows,  
    uint16_t nColumns,  
    float32_t * pData)
```

函数描述：

这个函数用于浮点格式的矩阵数据初始化。

函数参数：

- ◆ 第 1 个参数是 arm_matrix_instance_f32 类型矩阵结构体指针变量。
- ◆ 第 2 个参数是矩阵行数。
- ◆ 第 3 个参数是矩阵列数。
- ◆ 第 4 个参数是矩阵数据地址。

21.3.2 函数 arm_mat_init_q31

函数原型：

```
void arm_mat_init_f32(  
    arm_matrix_instance_f32 * S,  
    uint16_t nRows,  
    uint16_t nColumns,  
    float32_t * pData)
```

函数描述：

这个函数用于定点数 Q31 格式的矩阵数据初始化。

函数参数：

- ◆ 第 1 个参数是 arm_matrix_instance_q31 类型矩阵结构体指针变量。
- ◆ 第 2 个参数是矩阵行数。
- ◆ 第 3 个参数是矩阵列数。
- ◆ 第 4 个参数是矩阵数据地址。



21.3.3 函数 arm_mat_init_q15

函数原型:

```
void arm_mat_init_f32(  
    arm_matrix_instance_f32 * S,  
    uint16_t nRows,  
    uint16_t nColumns,  
    float32_t * pData)
```

函数描述:

这个函数用于定点数 Q15 格式的矩阵数据初始化。

函数参数:

- ◆ 第 1 个参数是 arm_matrix_instance_q15 类型矩阵结构体指针变量。
- ◆ 第 2 个参数是矩阵行数。
- ◆ 第 3 个参数是矩阵列数。
- ◆ 第 4 个参数是矩阵数据地址。

21.3.4 使用举例

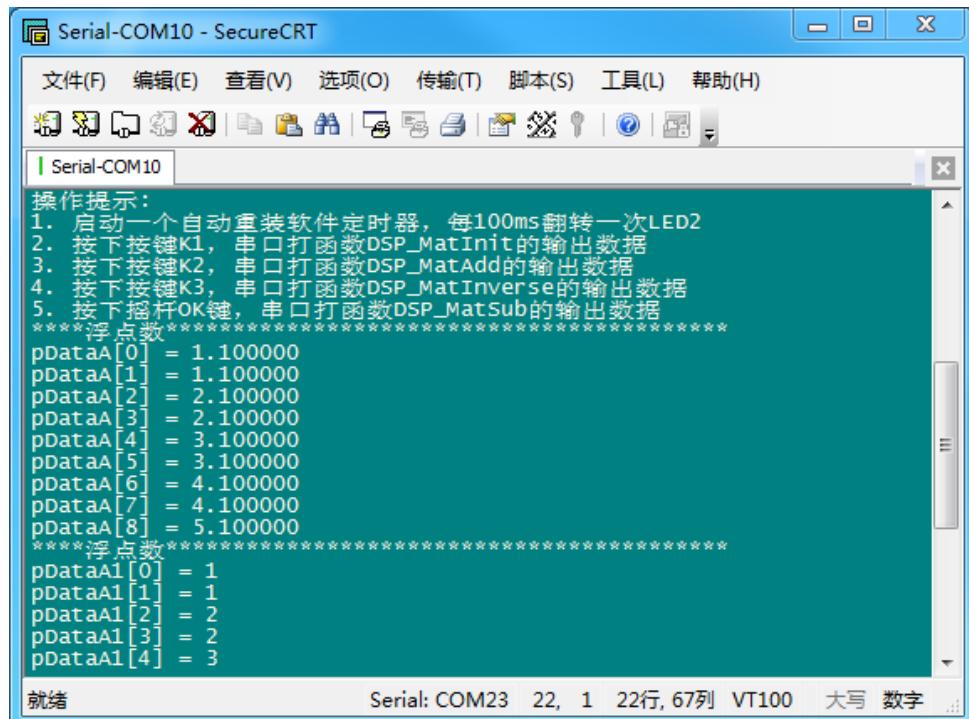
程序设计:

```
/*  
*****  
* 函数名: DSP_MatInit  
* 功能说明: 矩阵数据初始化  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_MatInit(void)  
{  
    uint8_t i;  
  
    /****浮点数数组*****  
    float32_t pDataA[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};  
  
    arm_matrix_instance_f32 pSrcA; //3行3列数据  
  
    /****定点数Q31数组*****  
    q31_t pDataA1[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
  
    arm_matrix_instance_q31 pSrcA1; //3行3列数据  
  
    /****定点数Q15数组*****  
    q15_t pDataA2[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
  
    arm_matrix_instance_q15 pSrcA2; //3行3列数据  
  
    /****浮点数*****  
    printf("****浮点数*****\r\n");  
    arm_mat_init_f32(&pSrcA, 3, 3, pDataA);  
    for(i = 0; i < 9; i++)  
    {  
        printf("pDataA[%d] = %f\r\n", i, pDataA[i]);  
    }  
  
    /****定点数Q31*****  
    printf("****定点数Q31*****\r\n");  
    arm_mat_init_q31(&pSrcA1, 3, 3, pDataA1);  
    for(i = 0; i < 9; i++)
```

```
{
    printf("pDataA1[%d] = %d\r\n", i, pDataA1[i]);
}

/**定点数Q15*****浮点数*****\r\n*/
printf("*****浮点数*****\r\n");
arm_mat_init_q15(&pSrcA2, 3, 3, pDataA2);
for(i = 0; i < 9; i++)
{
    printf("pDataA2[%d] = %d\r\n", i, pDataA2[i]);
}
}
```

实验现象 (按下 K1 按键后串口打印模平方):)



21.4 矩阵加法 (MatAdd)

以 3*3 矩阵为例，矩阵加法的实现公式如下：

$$\begin{bmatrix} \underline{\underline{a_{11}}} & \underline{\underline{a_{12}}} & \underline{\underline{a_{13}}} \\ \underline{\underline{a_{21}}} & \underline{\underline{a_{22}}} & \underline{\underline{a_{23}}} \\ \underline{\underline{a_{31}}} & \underline{\underline{a_{32}}} & \underline{\underline{a_{33}}} \end{bmatrix} + \begin{bmatrix} \underline{\underline{b_{11}}} & \underline{\underline{b_{12}}} & \underline{\underline{b_{13}}} \\ \underline{\underline{b_{21}}} & \underline{\underline{b_{22}}} & \underline{\underline{b_{23}}} \\ \underline{\underline{b_{31}}} & \underline{\underline{b_{32}}} & \underline{\underline{b_{33}}} \end{bmatrix} = \begin{bmatrix} \underline{\underline{a_{11}+b_{11}}} & \underline{\underline{a_{12}+b_{12}}} & \underline{\underline{a_{13}+b_{13}}} \\ \underline{\underline{a_{21}+b_{21}}} & \underline{\underline{a_{22}+b_{22}}} & \underline{\underline{a_{23}+b_{23}}} \\ \underline{\underline{a_{31}+b_{31}}} & \underline{\underline{a_{32}+b_{32}}} & \underline{\underline{a_{33}+b_{33}}} \end{bmatrix}$$

21.4.1 函数 arm_mat_add_f32

函数原型：

```
arm_status arm_mat_add_f32(
    const arm_matrix_instance_f32 * pSrcA,
```



```
const arm_matrix_instance_f32 * pSrcB,  
arm_matrix_instance_f32 * pDst)
```

函数描述：

这个函数用于浮点数的矩阵加法。

函数参数：

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A + 矩阵 B 计算结果存储的地址。
- ◆ 返回值，ARM_MATH_SUCCESS 表示成功，ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项：

1. pSrcA, pSrcB, pDst 的行数和列数必须是相同的，否则没有办法使用加法运算。
2. 矩阵在数组中的存储是从左到右，再从上到下。

21.4.2 函数 arm_mat_add_q31

函数原型：

```
arm_status arm_mat_add_f32(  
    const arm_matrix_instance_f32 * pSrcA,  
    const arm_matrix_instance_f32 * pSrcB,  
    arm_matrix_instance_f32 * pDst)
```

函数描述：

这个函数用于定点数 Q31 的矩阵加法。

函数参数：

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A + 矩阵 B 计算结果存储的地址。
- ◆ 返回值，ARM_MATH_SUCCESS 表示成功，ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项：

1. 使用了饱和运算，输出结果范围[0x80000000 0x7FFFFFFF]。
2. pSrcA, pSrcB, pDst 的行数和列数必须是相同的，否则没有办法使用加法运算。
3. 矩阵在数组中的存储是从左到右，再从上到下。

21.4.3 函数 arm_mat_add_q15

函数原型：

```
arm_status arm_mat_add_f32(  
    const arm_matrix_instance_f32 * pSrcA,
```



```
const arm_matrix_instance_f32 * pSrcB,  
arm_matrix_instance_f32 * pDst)
```

函数描述：

这个函数用于定点数 Q15 的矩阵加法。

函数参数：

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A + 矩阵 B 计算结果存储的地址。
- ◆ 返回值，ARM_MATH_SUCCESS 表示成功，ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项：

1. 使用了饱和运算，输出结果范围[0x8000 0x7FFF]。
2. pSrcA, pSrcB, pDst 的行数和列数必须是相同的，否则没有办法使用加法运算。
3. 矩阵在数组中的存储是从左到右，再从上到下。

21.4.4 用举例（含 Matlab 实现）

程序设计：

```
/*  
*****  
* 函数名: DSP_MatAdd  
* 功能说明: 矩阵求和  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_MatAdd(void)  
{  
    uint8_t i;  
  
    /*浮点数数组*****  
    float32_t pDataA[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};  
    float32_t pDataB[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};  
    float32_t pDataDst[9];  
  
    arm_matrix_instance_f32 pSrcA; //3行3列数据  
    arm_matrix_instance_f32 pSrcB; //3行3列数据  
    arm_matrix_instance_f32 pDst;  
  
    *****定点数Q31数组*****  
    q31_t pDataA1[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
    q31_t pDataB1[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
    q31_t pDataDst1[9];  
  
    arm_matrix_instance_q31 pSrcA1; //3行3列数据  
    arm_matrix_instance_q31 pSrcB1; //3行3列数据  
    arm_matrix_instance_q31 pDst1;  
  
    *****定点数Q15数组*****  
    q15_t pDataA2[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
    q15_t pDataB2[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
    q15_t pDataDst2[9];  
  
    arm_matrix_instance_q15 pSrcA2; //3行3列数据  
    arm_matrix_instance_q15 pSrcB2; //3行3列数据  
    arm_matrix_instance_q15 pDst2;  
  
    ****浮点数*****
```



```
pSrcA.numCols = 3;
pSrcA.numRows = 3;
pSrcA.pData = pDataA;

pSrcB.numCols = 3;
pSrcB.numRows = 3;
pSrcB.pData = pDataB;

pDst.numCols = 3;
pDst.numRows = 3;
pDst.pData = pDataDst;

printf("****浮点数*****\r\n");
arm_mat_add_f32(&pSrcA, &pSrcB, &pDst);
for(i = 0; i < 9; i++)
{
    printf("pDataDst[%d] = %f\r\n", i, pDataDst[i]);
}

/*定点数Q31*****
pSrcA1.numCols = 3;
pSrcA1.numRows = 3;
pSrcA1.pData = pDataA1;

pSrcB1.numCols = 3;
pSrcB1.numRows = 3;
pSrcB1.pData = pDataB1;

pDst1.numCols = 3;
pDst1.numRows = 3;
pDst1.pData = pDataDst1;

printf("****定点数Q31*****\r\n");
arm_mat_add_q31(&pSrcA1, &pSrcB1, &pDst1);
for(i = 0; i < 9; i++)
{
    printf("pDataDst1[%d] = %d\r\n", i, pDataDst1[i]);
}

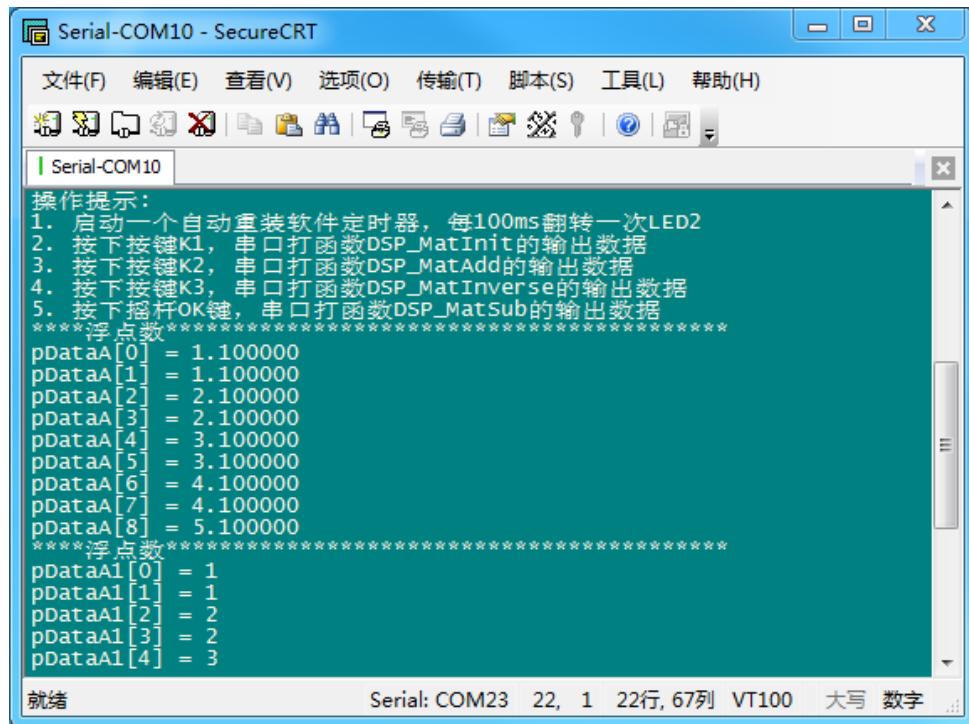
/*定点数Q15*****
pSrcA2.numCols = 3;
pSrcA2.numRows = 3;
pSrcA2.pData = pDataA2;

pSrcB2.numCols = 3;
pSrcB2.numRows = 3;
pSrcB2.pData = pDataB2;

pDst2.numCols = 3;
pDst2.numRows = 3;
pDst2.pData = pDataDst2;

printf("****定点数Q15*****\r\n");
arm_mat_add_q15(&pSrcA2, &pSrcB2, &pDst2);
for(i = 0; i < 9; i++)
{
    printf("pDataDst2[%d] = %d\r\n", i, pDataDst2[i]);
}
```

实验现象 (按下 K2 按键后串口打印矩阵加法):



下面通过 Matlab 来求解矩阵和 (在命令窗口输入):

```

命令行窗口
>> a = [1 2 3; 2 3 4];
>> b = [1 2 3; 2 3 4];
>> c = a+b

c =
    2     4     6
    4     6     8

fx >> |

```

21.5 矩阵减法 (MatSub)

以 3*3 矩阵为例, 矩阵减法的实现公式如下:

$$\begin{array}{ccc|c}
 \boxed{a_{11}} & a_{12} & \boxed{a_{13}} & \\
 a_{21} & a_{22} & a_{23} & \\
 a_{31} & a_{32} & a_{33} & \\ \hline
 & - & & \\
 & \boxed{b_{11}} & b_{12} & \boxed{b_{13}} \\
 & b_{21} & b_{22} & b_{23} \\
 & b_{31} & b_{32} & b_{33} \\ \hline
 & & & \\
 \boxed{a_{11}-b_{11}} & a_{12}-b_{12} & \boxed{a_{13}-b_{13}} & \\
 a_{21}-b_{21} & a_{22}-b_{22} & a_{23}-b_{23} & \\
 a_{31}-b_{31} & a_{32}-b_{32} & a_{33}-b_{33} &
 \end{array}$$



21.5.1 函数 arm_mat_sub_f32

函数原型:

```
arm_status arm_mat_sub_f32(  
    const arm_matrix_instance_f32 * pSrcA,  
    const arm_matrix_instance_f32 * pSrcB,  
    arm_matrix_instance_f32 * pDst)
```

函数描述:

这个函数用于浮点数的矩阵减法。

函数参数:

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A 减去矩阵 B 计算结果存储的地址。
- ◆ 返回值, ARM_MATH_SUCCESS 表示成功, ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项:

1. pSrcA, pSrcB, pDst 的行数和列数必须是相同的, 否则没有办法使用加法运算。
2. 矩阵在数组中的存储是从左到右, 再从上到下。

21.5.2 函数 arm_mat_sub_q31

函数原型:

```
arm_status arm_mat_sub_q31(  
    const arm_matrix_instance_q31 * pSrcA,  
    const arm_matrix_instance_q31 * pSrcB,  
    arm_matrix_instance_q31 * pDst)
```

函数描述:

这个函数用于定点数 Q31 的矩阵减法。

函数参数:

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A 减去矩阵 B 计算结果存储的地址。
- ◆ 返回值, ARM_MATH_SUCCESS 表示成功, ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项:

1. 使用了饱和运算, 输出结果范围[0x80000000 0x7FFFFFFF]。
2. pSrcA, pSrcB, pDst 的行数和列数必须是相同的, 否则没有办法使用加法运算。
3. 矩阵在数组中的存储是从左到右, 再从上到下。



21.5.3 函数 arm_mat_sub_q15

函数原型:

```
arm_status arm_mat_sub_q15(  
    const arm_matrix_instance_q15 * pSrcA,  
    const arm_matrix_instance_q15 * pSrcB,  
    arm_matrix_instance_q15 * pDst)
```

函数描述:

这个函数用于定点数 Q15 的矩阵减法。

函数参数:

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A 减去矩阵 B 计算结果存储的地址。
- ◆ 返回值, ARM_MATH_SUCCESS 表示成功, ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项:

1. 使用了饱和运算, 输出结果范围[0x8000 0x7FFF]。
2. pSrcA, pSrcB, pDst 的行数和列数必须是相同的, 否则没有办法使用加法运算。
3. 矩阵在数组中的存储是从左到右, 再从上。

21.5.4 使用举例 (含 Matlab 实现)

程序设计:

```
/*  
*****  
* 函数名: DSP_MatSub  
* 功能说明: 矩阵减法  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_MatSub(void)  
{  
    uint8_t i;  
  
    /****浮点数数组*****  
    float32_t pDataA[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};  
    float32_t pDataB[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};  
    float32_t pDataDst[9];  
  
    arm_matrix_instance_f32 pSrcA; //3行3列数据  
    arm_matrix_instance_f32 pSrcB; //3行3列数据  
    arm_matrix_instance_f32 pDst;  
  
    /****定点数Q31数组*****  
    q31_t pDataA1[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
    q31_t pDataB1[9] = {2, 2, 2, 2, 2, 2, 2, 2, 2};  
    q31_t pDataDst1[9];  
  
    arm_matrix_instance_q31 pSrcA1; //3行3列数据
```



```
arm_matrix_instance_q31 pSrcB1; //3行3列数据
arm_matrix_instance_q31 pDst1;

/*定点数Q15数组***** */
q15_t pDataA2[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};
q15_t pDataB2[9] = {2, 2, 2, 2, 23, 2, 2, 2, 2};
q15_t pDataDst2[9];

arm_matrix_instance_q15 pSrcA2; //3行3列数据
arm_matrix_instance_q15 pSrcB2; //3行3列数据
arm_matrix_instance_q15 pDst2;

/*浮点数***** */
pSrcA.numCols = 3;
pSrcA.numRows = 3;
pSrcA.pData = pDataA;

pSrcB.numCols = 3;
pSrcB.numRows = 3;
pSrcB.pData = pDataB;

pDst.numCols = 3;
pDst.numRows = 3;
pDst.pData = pDataDst;

printf("*****浮点数*****\r\n");
arm_mat_sub_f32(&pSrcA, &pSrcB, &pDst);
for(i = 0; i < 9; i++)
{
    printf("pDataDst[%d] = %f\r\n", i, pDataDst[i]);
}

/*定点数Q31***** */
pSrcA1.numCols = 3;
pSrcA1.numRows = 3;
pSrcA1.pData = pDataA1;

pSrcB1.numCols = 3;
pSrcB1.numRows = 3;
pSrcB1.pData = pDataB1;

pDst1.numCols = 3;
pDst1.numRows = 3;
pDst1.pData = pDataDst1;

printf("*****定点数Q31*****\r\n");
arm_mat_sub_q31(&pSrcA1, &pSrcB1, &pDst1);
for(i = 0; i < 9; i++)
{
    printf("pDataDst1[%d] = %d\r\n", i, pDataDst1[i]);
}

/*定点数Q15***** */
pSrcA2.numCols = 3;
pSrcA2.numRows = 3;
pSrcA2.pData = pDataA2;

pSrcB2.numCols = 3;
pSrcB2.numRows = 3;
pSrcB2.pData = pDataB2;

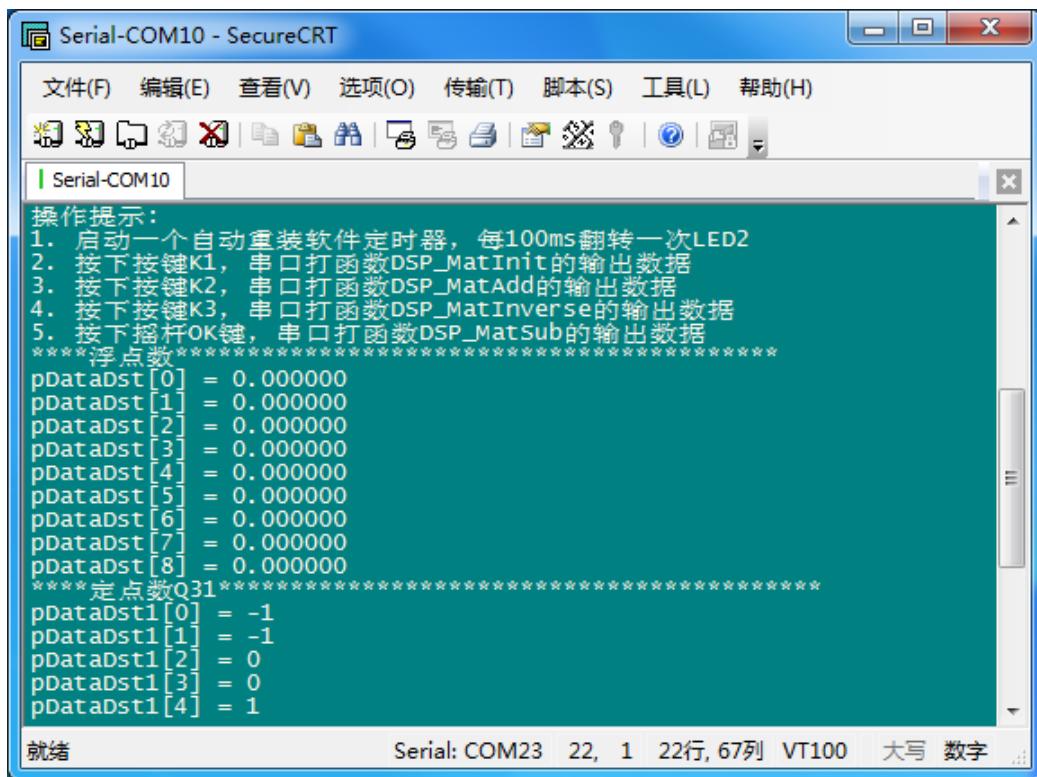
pDst2.numCols = 3;
pDst2.numRows = 3;
pDst2.pData = pDataDst2;

printf("*****定点数Q15*****\r\n");
arm_mat_sub_q15(&pSrcA2, &pSrcB2, &pDst2);
for(i = 0; i < 9; i++)
```

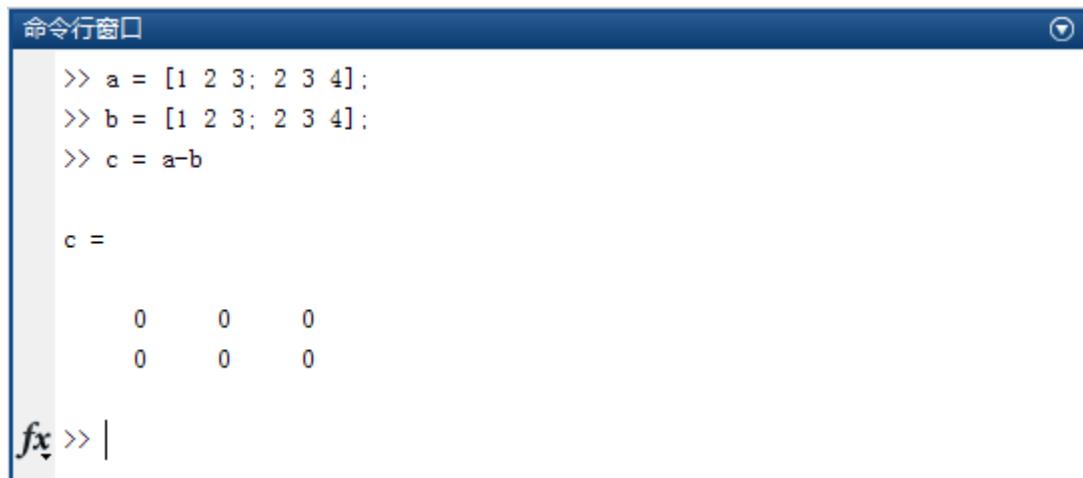


```
{  
    printf("pDataDst2[%d] = %d\r\n", i, pDataDst2[i]);  
}
```

实验现象 (按下 OK 按键后串口打印矩阵减法):



下面通过 Matlab 来求解矩阵减法 (在命令窗口输入)。



21.6 逆矩阵 (MatInverse)

以 3*3 矩阵为例，逆矩阵的实现公式如下 (Gauss-Jordan 法求逆矩阵):



$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array} \right] = \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & X_{11} & X_{21} & X_{31} \\ 0 & 1 & 0 & X_{12} & X_{22} & X_{32} \\ 0 & 0 & 1 & X_{13} & X_{23} & X_{33} \end{array} \right]$$

A is a 3×3 matrix and its inverse is X

21.6.1 函数 arm_mat_inverse_f64

函数原型:

```
arm_status arm_mat_inverse_f64(  
    const arm_matrix_instance_f64 * pSrc,  
    arm_matrix_instance_f64 * pDst)
```

函数描述:

这个函数用于 64bit 浮点数的逆矩阵求解。

函数参数:

- ◆ 第 1 个参数是矩阵源地址。
- ◆ 第 2 个参数是求逆后的矩阵地址。
- ◆ 返回值, ARM_MATH_SUCCESS 表示成功, ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。
ARM_MATH_SINGULAR 表示矩阵不可逆。

注意事项:

1. pSrc 必须得是方阵 (行数和列数相同)。
2. pSrc 和 pDst 必须是相同的方阵。
3. 输入的矩阵可逆, 函数会返回 ARM_MATH_SUCCESS, 如果不可逆, 返回
ARM_MATH_SINGULAR。
4. ARM 官方库只提供了浮点数矩阵求逆矩阵。

21.6.2 函数 arm_mat_inverse_f32

函数原型:

```
arm_status arm_mat_inverse_f32(  
    const arm_matrix_instance_f32 * pSrc,  
    arm_matrix_instance_f32 * pDst)
```

函数描述:

这个函数用于 32bit 浮点数的逆矩阵求解。

函数参数:



- ◆ 第1个参数是矩阵源地址。
- ◆ 第2个参数是求逆后的矩阵地址。
- ◆ 返回值，ARM_MATH_SUCCESS 表示成功，ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。
ARM_MATH_SINGULAR 表示矩阵不可逆。

注意事项：

1. pSrc 必须得是方阵（行数和列数相同）。
2. pSrc 和 pDst 必须是相同的方阵。
3. 输入的矩阵可逆，函数会返回 ARM_MATH_SUCCESS，如果不可逆，返回 ARM_MATH_SINGULAR。
4. ARM 官方库只提供了浮点数矩阵求逆矩阵。

21.6.3 使用举例（含 Matlab 实现）

程序设计：

```
/*
*****
* 函数名: DSP_MatInverse
* 功能说明: 求逆矩阵
* 形参: 无
* 返回值: 无
*****
*/
static void DSP_MatInverse(void)
{
    uint8_t i;

    arm_status sta;

    /****浮点数数组*****/
    float32_t pDataB[36];
    float32_t pDataA[36] = {
        1.0f,    0.0f,    0.0f,    0.0f,    0.0f,    0.0f,
        0.0f,    1.0f,    0.0f,    0.0f,    1.0f,    0.0f,
        0.0f,    0.0f,    2.0f,    0.0f,    0.0f,    0.0f,
        0.0f,    0.0f,    0.0f,    2.0f,    0.0f,    1.0f,
        0.0f,    0.0f,    0.0f,    0.0f,    3.0f,    0.0f,
        0.0f,    0.0f,    0.0f,    0.0f,    0.0f,    4.0f};

    arm_matrix_instance_f32 pSrcA; //6行6列数据
    arm_matrix_instance_f32 pSrcB; //6行6列数据;

    /****浮点数*****/
    pSrcA.numCols = 6;
    pSrcA.numRows = 6;
    pSrcA.pData = pDataA;

    pSrcB.numCols = 6;
    pSrcB.numRows = 6;
    pSrcB.pData = pDataB;

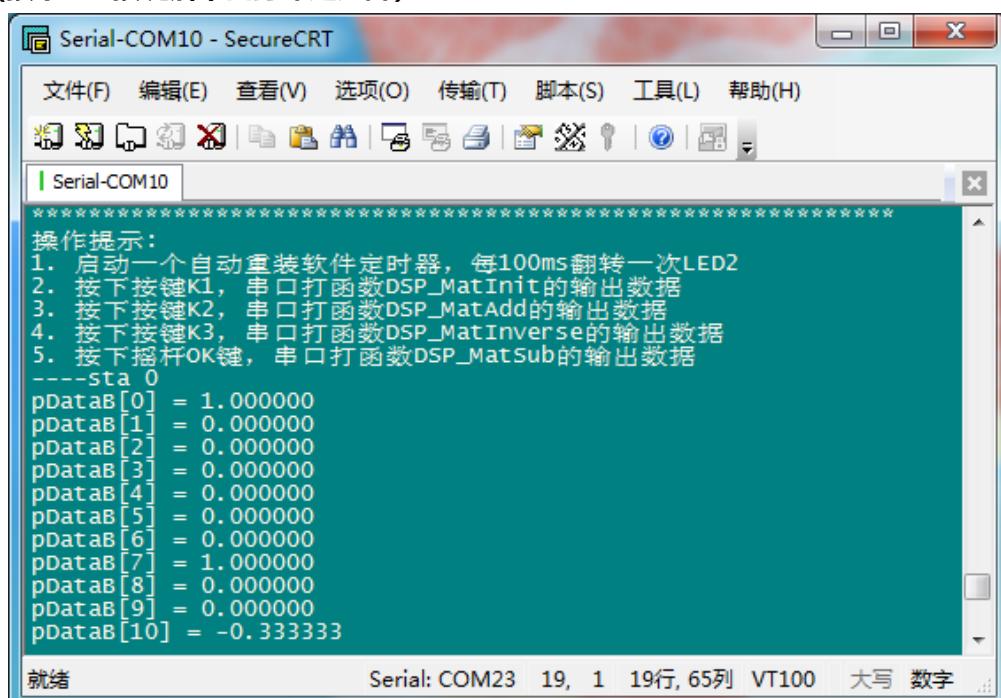
    sta = arm_mat_inverse_f32(&pSrcA, &pSrcB);

    /*
        sta = ARM_MATH_SUCCESS, 即返回0, 表示求逆矩阵成功。
        sta = ARM_MATH_SINGULAR, 即返回-5, 表示求逆矩阵失败, 也表示不可逆。
        注意, ARM提供的DSP库逆矩阵求发有局限性, 通过Matlab验证是可以求逆矩阵的, 而DSP库却不能正确求解。
    */
}
```

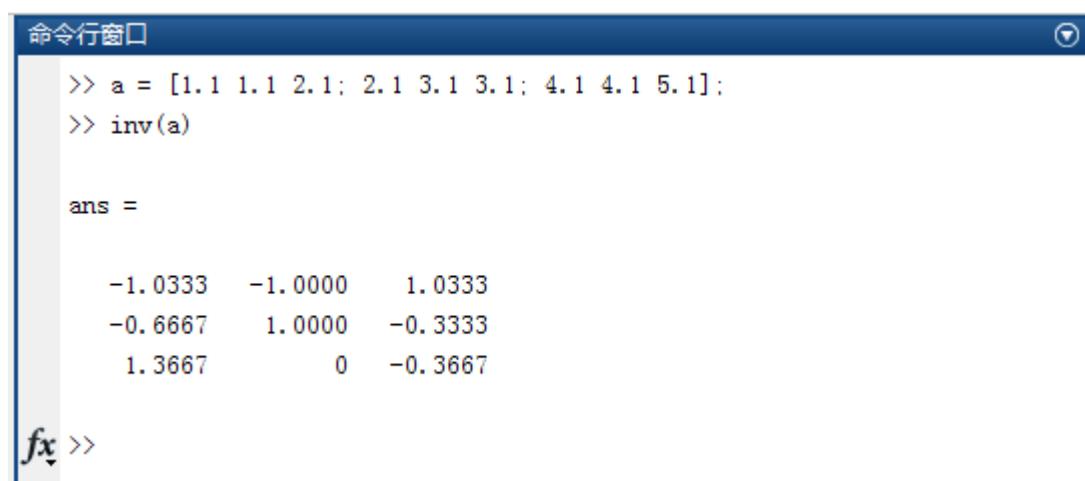
```
/*
printf("----sta %d\r\n", sta);

for(i = 0; i < 36; i++)
{
    printf("pDataB[%d] = %f\r\n", i, pDataB[i]);
}
}
```

实验现象 (按下 K3 按键后串口打印逆矩阵):



下面我们通过 Matlab 来实现求逆矩阵 (在命令窗口输入):



21.7 实验例程说明 (MDK)

配套例子:

V7-216_DSP 矩阵运算 (加法, 减法和逆矩阵)

实验目的:

1. 学习 DSP 复数运算 (加法, 减法和逆矩阵)

实验内容:

1. 启动一个自动重装软件定时器, 每 100ms 翻转一次 LED2。
2. 按下按键 K1, 串口打函数 DSP_MatInit 的输出数据。
3. 按下按键 K2, 串口打函数 DSP_MatAdd 的输出数据。
4. 按下按键 K3, 串口打函数 DSP_MatInverse 的输出数据。
5. 按下摇杆 OK 键, 串口打函数 DSP_MatSub 的输出数据。

使用 AC6 注意事项

特别注意附件章节 C 的问题

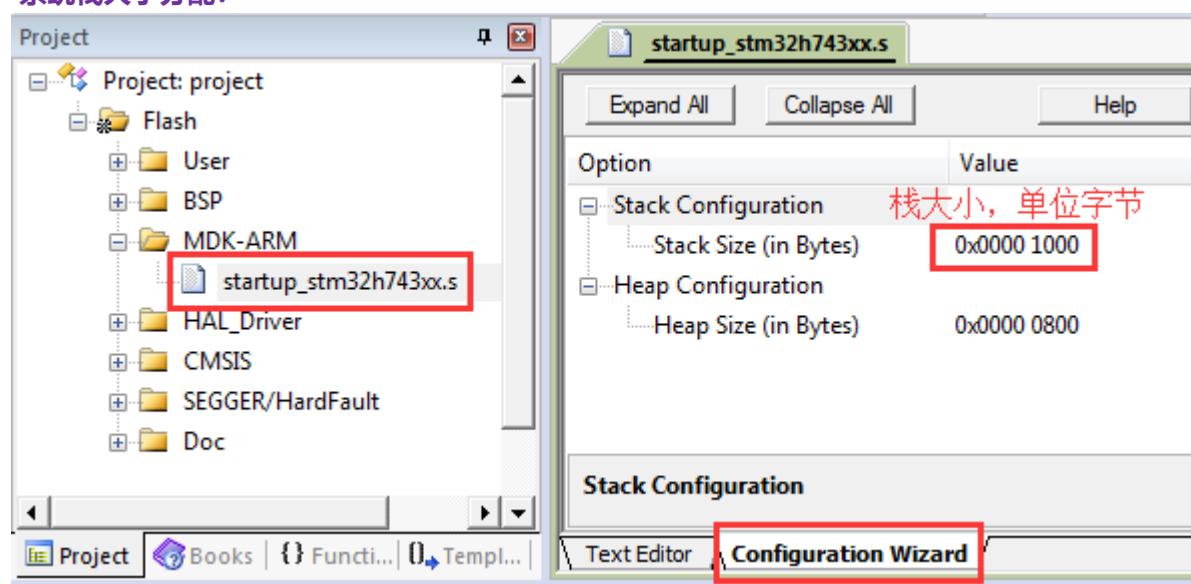
上电后串口打印的信息:

波特率 115200, 数据位 8, 奇偶校验位无, 停止位 1。

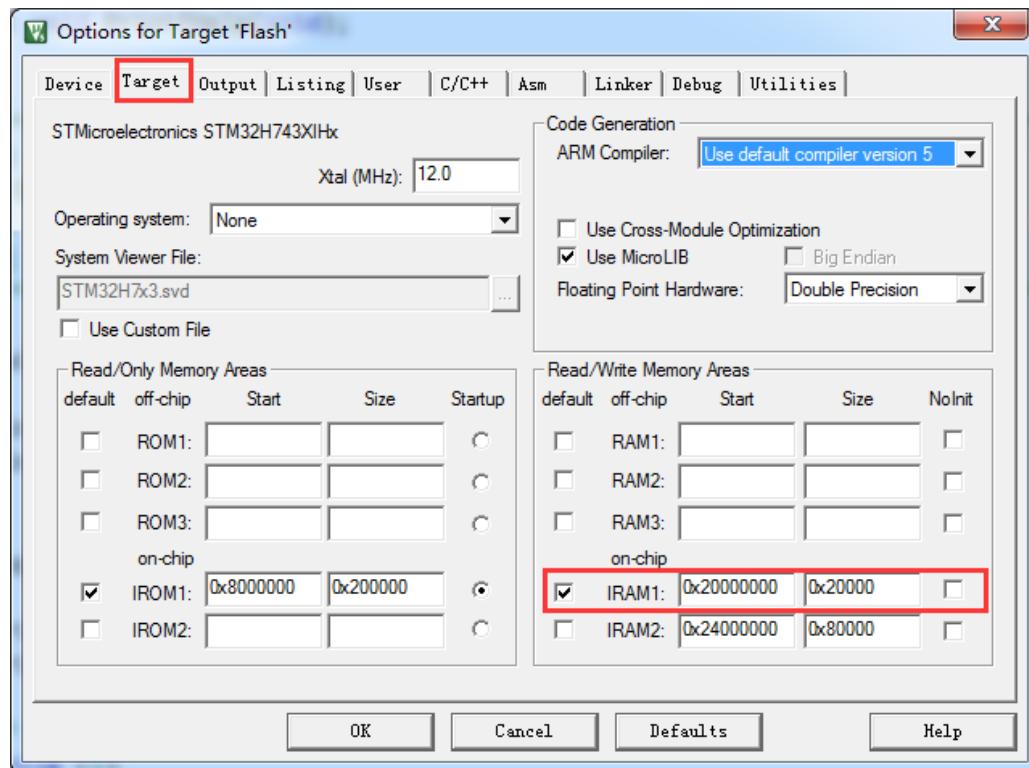
详见本章的 3.4, 4.4, 5.4 和 6.3 小节。

程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打函数 DSP_MatInit 的输出数据。
- 按下按键 K2，串口打函数 DSP_MatAdd 的输出数据。
- 按下按键 K3，串口打函数 DSP_MatInverse 的输出数据。
- 按下摇杆 OK 键，串口打函数 DSP_MatSub 的输出数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
```



```
PrintfLogo(); /* 打印例程信息到串口 1 */

PrintfHelp(); /* 打印操作提示信息 */

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle(); /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1: /* 按下按键 K1, 串口打函数 DSP_MatInit 的输出数据 */
                DSP_MatInit();
                break;

            case KEY_DOWN_K2: /* 按下按键 K2, 串口打函数 DSP_MatAdd 的输出数据 */
                DSP_MatAdd();
                break;

            case KEY_DOWN_K3: /* 按下按键 K3, 串口打函数 DSP_MatInverse 的输出数据 */
                DSP_MatInverse();
                break;

            case JOY_DOWN_OK: /* 按下摇杆 OK 键, 串口打函数 DSP_MatSub 的输出数据 */
                DSP_MatSub();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}

}
```

21.8 实验例程说明 (IAR)

配套例子：

V7-216_DSP 矩阵运算 (加法, 减法和逆矩阵)

实验目的：

1. 学习 DSP 复数运算 (加法, 减法和逆矩阵)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打函数 DSP_MatInit 的输出数据。
3. 按下按键 K2，串口打函数 DSP_MatAdd 的输出数据。
4. 按下按键 K3，串口打函数 DSP_MatInverse 的输出数据。
5. 按下摇杆 OK 键，串口打函数 DSP_MatSub 的输出数据。

使用 AC6 注意事项

特别注意附件章节 C 的问题

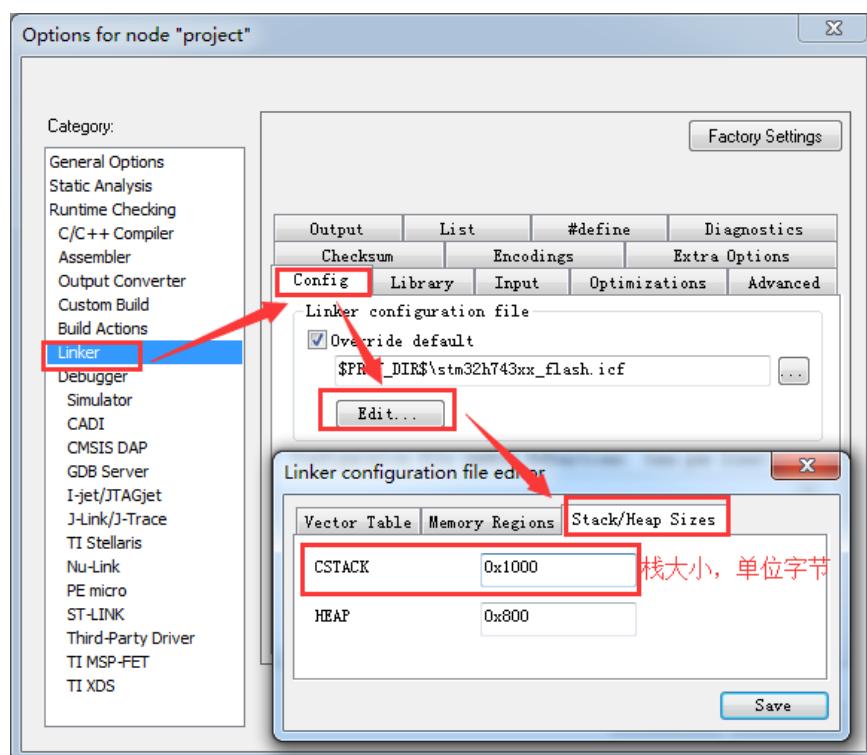
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

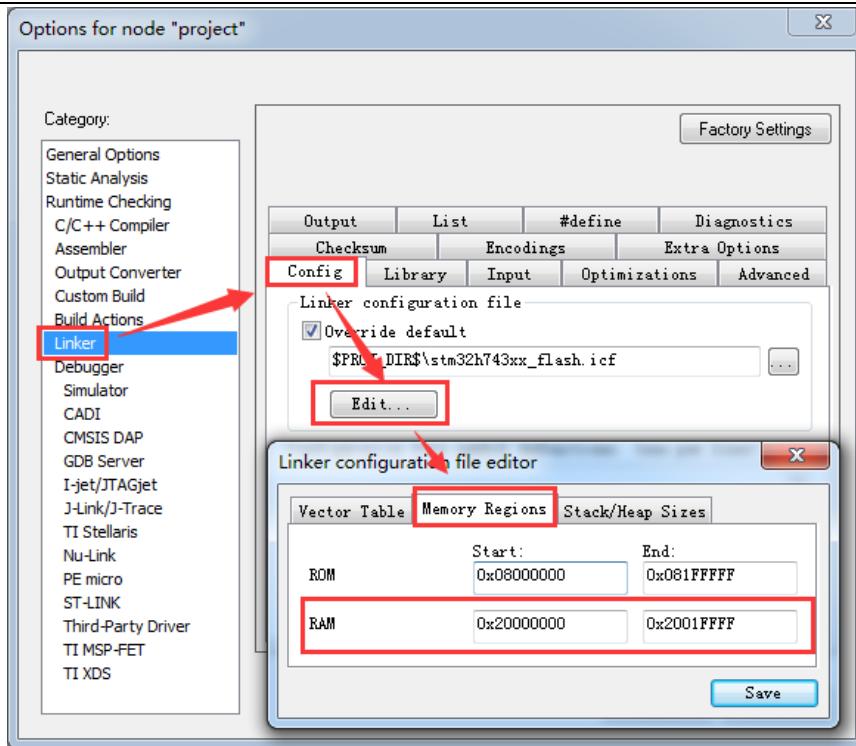
详见本章的 3.4，4.4，5.4 和 6.3 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打函数 DSP_MatInit 的输出数据。
- 按下按键 K2，串口打函数 DSP_MatAdd 的输出数据。
- 按下按键 K3，串口打函数 DSP_MatInverse 的输出数据。
- 按下摇杆 OK 键，串口打函数 DSP_MatSub 的输出数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */\n\n/* 进入主程序循环体 */\nwhile (1)\n{\n    bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */\n\n    /* 判断定时器超时时间 */\n    if (bsp_CheckTimer(0))\n    {\n        /* 每隔 100ms 进来一次 */\n        bsp_LedToggle(2);\n    }\n\n    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */\n    if (ucKeyCode != KEY_NONE)\n    {\n        switch (ucKeyCode)\n        {\n            case KEY_DOWN_K1:           /* 按下按键 K1，串口打函数 DSP_MatInit 的输出数据 */\n                DSP_MatInit();\n                break;\n\n            case KEY_DOWN_K2:           /* 按下按键 K2，串口打函数 DSP_MatAdd 的输出数据 */\n                DSP_MatAdd();\n                break;\n\n            case KEY_DOWN_K3:           /* 按下按键 K3，串口打函数 DSP_MatInverse 的输出数据 */\n                DSP_MatInverse();\n                break;\n\n            case JOY_DOWN_OK:          /* 按下摇杆 OK 键，串口打函数 DSP_MatSub 的输出数据 */\n                DSP_MatSub();\n                break;\n\n            default:\n                /* 其他的键值不处理 */\n                break;\n        }\n    }\n}
```

21.9 总结

本期教程就跟大家讲这么多，有兴趣的可以深入研究下算法的具体实现。



第22章 DSP 矩阵运算-放缩，乘法和转置矩阵

本期教程主要讲解矩阵运算中的放缩，乘法和转置。

22.1 初学者重要提示

22.2 DSP 基础运算指令

22.3 矩阵放缩 (MatScale)

22.4 矩阵乘法 (MatMult)

22.5 转置矩阵 (MatTrans)

22.6 实验例程说明 (MDK)

22.7 实验例程说明 (IAR)

22.8 总结

22.1 初学者重要提示

- ◆ ARM 提供的 DSP 库逆矩阵求法有局限性，通过 Matlab 验证是可以求逆矩阵的，而 DSP 库却不能正确求解。
- ◆ 注意定点数的矩阵乘法运算中溢出问题。

22.2 DSP 基础运算指令

本章用到的 DSP 指令在前面章节都已经讲解过。

22.3 矩阵放缩 (MatScale)

以 3*3 矩阵为例，矩阵放缩的实现公式如下：

$$\begin{bmatrix} \underline{\mathbf{a}_{11}} & \underline{\mathbf{a}_{12}} & \underline{\mathbf{a}_{13}} \\ \underline{\mathbf{a}_{21}} & \underline{\mathbf{a}_{22}} & \underline{\mathbf{a}_{23}} \\ \underline{\mathbf{a}_{31}} & \underline{\mathbf{a}_{32}} & \underline{\mathbf{a}_{33}} \end{bmatrix} \times K = \begin{bmatrix} \underline{\mathbf{a}_{11} \times K} & \underline{\mathbf{a}_{12} \times K} & \underline{\mathbf{a}_{13} \times K} \\ \underline{\mathbf{a}_{21} \times K} & \underline{\mathbf{a}_{22} \times K} & \underline{\mathbf{a}_{23} \times K} \\ \underline{\mathbf{a}_{31} \times K} & \underline{\mathbf{a}_{32} \times K} & \underline{\mathbf{a}_{33} \times K} \end{bmatrix}$$



22.3.1 函数 arm_mat_scale_f32

函数原型:

```
arm_status arm_mat_scale_f32(  
    const arm_matrix_instance_f32 * pSrc,  
    float32_t scale,  
    arm_matrix_instance_f32 * pDst)
```

函数描述:

这个函数用于浮点格式的矩阵数据的放缩。

函数参数:

- ◆ 第 1 个参数是源矩阵地址。
- ◆ 第 2 个参数是放缩系数。
- ◆ 第 3 个参数是放缩后的目的数据地址。
- ◆ 返回值, ARM_MATH_SIZE_MISMATCH 表示源矩阵和目标矩阵行列不一致,
ARM_MATH_SUCCESS 表示成功。

注意事项:

矩阵在数组中的存储是从左到右, 再从上到下。

22.3.2 函数 arm_mat_scale_q31

函数原型:

```
arm_status arm_mat_scale_q31(  
    const arm_matrix_instance_q31 * pSrc,  
    q31_t scaleFract,  
    int32_t shift,  
    arm_matrix_instance_q31 * pDst)
```

函数描述:

这个函数用于定点 Q31 格式的矩阵数据的放缩。

函数参数:

- ◆ 第 1 个参数是源矩阵地址。
- ◆ 第 2 个参数是放缩系数。
- ◆ 第 3 个参数是移位的 bit 数。
- ◆ 第 4 个参数是放缩后的目的数据地址。
- ◆ 返回值, ARM_MATH_SIZE_MISMATCH 表示源矩阵和目标矩阵行列不一致,
ARM_MATH_SUCCESS 表示成功。

注意事项:

1. 两个 1.31 格式的数据相乘产生 2.62 格式的数据, 最终结果要做偏移和饱和运算产生 1.31 格式数。
2. 定点数的最终放缩比例计算是: $scale = scaleFract \times 2^shift$ 。
3. 矩阵在数组中的存储是从左到右, 再从上到下。



22.3.3 函数 arm_mat_scale_q15

函数原型:

```
arm_status arm_mat_scale_q15(  
    const arm_matrix_instance_q15 * pSrc,  
    q15_t scaleFract,  
    int32_t shift,  
    arm_matrix_instance_q15 * pDst)
```

函数描述:

这个函数用于定点 Q15 格式的矩阵数据的放缩。

函数参数:

- ◆ 第 1 个参数是源矩阵地址。
- ◆ 第 2 个参数是放缩系数。
- ◆ 第 3 个参数是移位的 bit 数。
- ◆ 第 4 个参数是放缩后的目的数据地址。
- ◆ 返回值, ARM_MATH_SIZE_MISMATCH 表示源矩阵和目标矩阵行列不一致,
ARM_MATH_SUCCESS 表示成功。

注意事项:

1. 两个1.15格式的数据相乘产生2.30格式的数据, 最终结果要做偏移和饱和运算产生1.15格式数据。
2. 定点数的最终放缩比例计算是: scale = scaleFract * 2^shift。
3. 矩阵在数组中的存储是从左到右, 再从上到下。

22.3.4 使用举例 (含 matlab 实现)

程序设计:

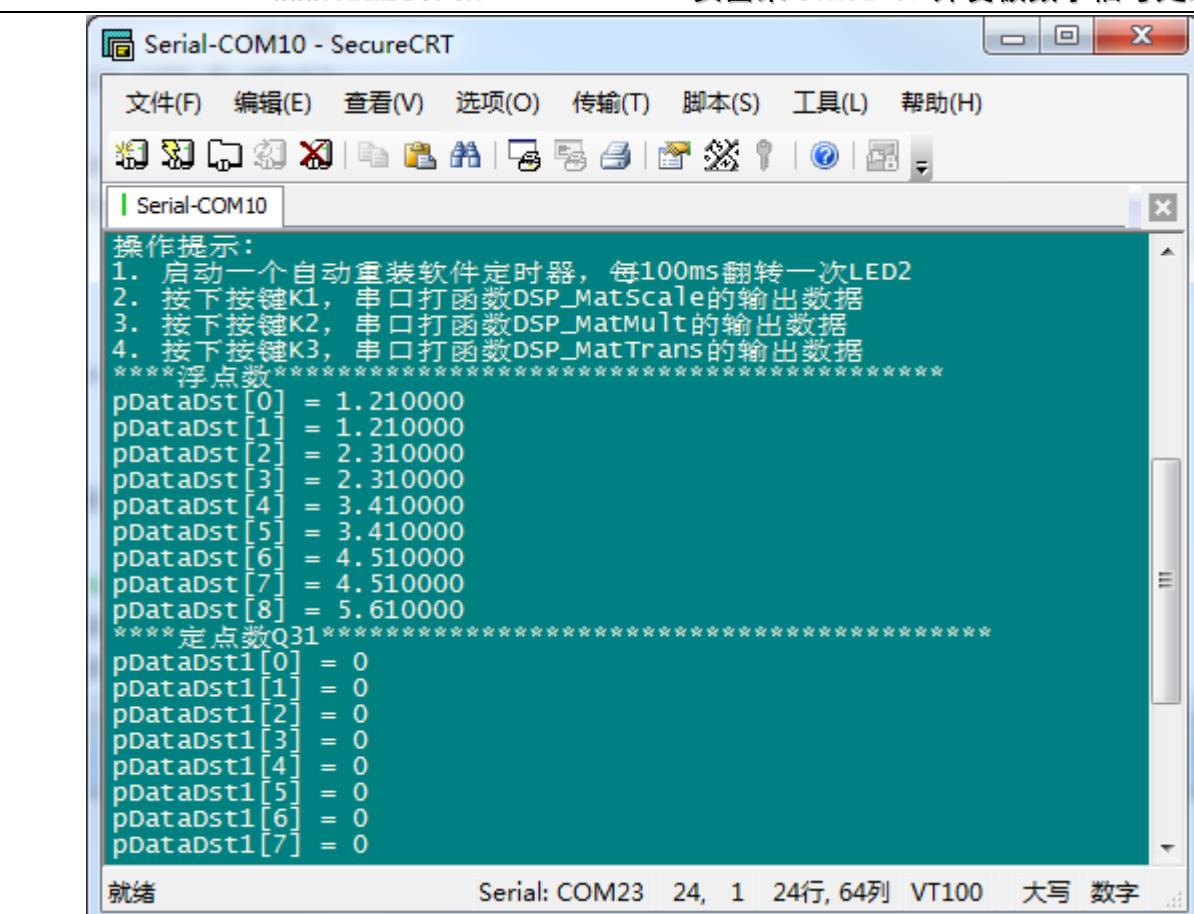
```
/*  
*****  
* 函数名: DSP_MatScale  
* 功能说明: 矩阵放缩  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_MatScale(void)  
{  
    uint8_t i;  
  
    /****浮点数数组*****  
    float32_t pDataA[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};  
    float32_t scale = 1.1f;  
    float32_t pDataDst[9];  
  
    arm_matrix_instance_f32 pSrcA; //3行3列数据  
    arm_matrix_instance_f32 pDst;  
  
    /****定点数Q31数组*****  
    q31_t pDataA1[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
    q31_t scaleFract = 10;  
    int32_t shift = 0;  
    q31_t pDataDst1[9];  
  
    arm_matrix_instance_q31 pSrcA1; //3行3列数据
```



```
arm_matrix_instance_q31 pDst1;

/*定点数Q15数组*****  
q15_t pDataA2[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
q15_t scaleFract1 = 10;  
int32_t shift1 = 0;  
q15_t pDataDst2[9];  
  
arm_matrix_instance_q15 pSrcA2; //3行3列数据  
arm_matrix_instance_q15 pDst2;  
  
/*浮点数*****  
pSrcA.numCols = 3;  
pSrcA.numRows = 3;  
pSrcA.pData = pDataA;  
  
pDst.numCols = 3;  
pDst.numRows = 3;  
pDst.pData = pDataDst;  
  
printf("*****浮点数*****\r\n");  
arm_mat_scale_f32(&pSrcA, scale, &pDst);  
for(i = 0; i < 9; i++)  
{  
    printf("pDataDst[%d] = %f\r\n", i, pDataDst[i]);  
}  
  
/*定点数Q31*****  
pSrcA1.numCols = 3;  
pSrcA1.numRows = 3;  
pSrcA1.pData = pDataA1;  
  
pDst1.numCols = 3;  
pDst1.numRows = 3;  
pDst1.pData = pDataDst1;  
  
printf("*****定点数Q31*****\r\n");  
arm_mat_scale_q31(&pSrcA1, scaleFract, shift, &pDst1);  
for(i = 0; i < 9; i++)  
{  
    printf("pDataDst1[%d] = %d\r\n", i, pDataDst1[i]);  
}  
  
/*定点数Q15*****  
pSrcA2.numCols = 3;  
pSrcA2.numRows = 3;  
pSrcA2.pData = pDataA2;  
  
pDst2.numCols = 3;  
pDst2.numRows = 3;  
pDst2.pData = pDataDst2;  
  
printf("*****定点数Q15*****\r\n");  
arm_mat_scale_q15(&pSrcA2, scaleFract1, shift1, &pDst2);  
for(i = 0; i < 9; i++)  
{  
    printf("pDataDst2[%d] = %d\r\n", i, pDataDst2[i]);  
}
```

实验现象 (按下 K1 按键后串口打印):



下面通过 matlab 来实现矩阵的放缩：

```

命令行窗口
>> a = [1 2 3; 4 5 6];
>> b = 3.4;
>> c = a * b

c =

    3.4000    6.8000   10.2000
   13.6000   17.0000   20.4000

fx >>

```

22.4 矩阵乘法 (MatMult)

以 3*3 矩阵为例，矩阵乘法的实现公式如下：

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} a_{11}x b_{11} + a_{12}x b_{21} + a_{13}x b_{31} & a_{11}x b_{12} + a_{12}x b_{22} + a_{13}x b_{32} & a_{11}x b_{13} + a_{12}x b_{23} + a_{13}x b_{33} \\ a_{21}x b_{11} + a_{22}x b_{21} + a_{23}x b_{31} & a_{21}x b_{12} + a_{22}x b_{22} + a_{23}x b_{32} & a_{21}x b_{13} + a_{22}x b_{23} + a_{23}x b_{33} \\ a_{31}x b_{11} + a_{32}x b_{21} + a_{33}x b_{31} & a_{31}x b_{12} + a_{32}x b_{22} + a_{33}x b_{32} & a_{31}x b_{13} + a_{32}x b_{23} + a_{33}x b_{33} \end{bmatrix}$$



22.4.1 函数 arm_mat_mult_f32

函数原型:

```
arm_status arm_mat_mult_f32(  
    const arm_matrix_instance_f32 * pSrcA,  
    const arm_matrix_instance_f32 * pSrcB,  
    arm_matrix_instance_f32 * pDst)
```

函数描述:

这个函数用于浮点数的矩阵乘法。

函数参数:

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A 乘以矩阵 B 计算结果存储的地址。
- ◆ 返回值, ARM_MATH_SUCCESS 表示成功, ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项:

1. 两个矩阵 $M \times N$ 和 $N \times P$ 相乘的结果是 $M \times P$ (必须保证一个矩阵的列数等于另一个矩阵的行数)。
2. 矩阵在数组中的存储是从左到右, 再从上到下。

22.4.2 函数 arm_mat_mult_q31

函数原型:

```
arm_status arm_mat_mult_q31(  
    const arm_matrix_instance_q31 * pSrcA,  
    const arm_matrix_instance_q31 * pSrcB,  
    arm_matrix_instance_q31 * pDst)
```

函数描述:

这个函数用于定点数 Q31 的矩阵乘法。

函数参数:

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A 乘以矩阵 B 计算结果存储的地址。
- ◆ 返回值, ARM_MATH_SUCCESS 表示成功, ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项:

1. 两个1.31格式的数据相乘产生2.62格式的数据, 函数的内部使用了64位的累加器, 最终结果要做偏移和饱和运算产生1.31格式数据。
2. 两个矩阵 $M \times N$ 和 $N \times P$ 相乘的结果是 $M \times P$ 。 (必须保证一个矩阵的列数等于另一个矩阵的行数)。
3. 矩阵在数组中的存储是从左到右, 再从上到下。

22.4.3 函数 arm_mat_mult_q15

函数原型:



```
arm_status arm_mat_mult_q15(
    const arm_matrix_instance_q15 * pSrcA,
    const arm_matrix_instance_q15 * pSrcB,
    arm_matrix_instance_q15 * pDst,
    q15_t * pState)
```

函数描述：

这个函数用于定点数 Q15 的矩阵乘法。

函数参数：

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A 乘以矩阵 B 计算结果存储的地址。
- ◆ 第 4 个参数用于存储内部计算结果。
- ◆ 返回值，ARM_MATH_SUCCESS 表示成功，ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项：

1. 两个1.15格式数据相乘是2.30格式，函数的内部使用了64位的累加器，34.30格式，最终结果将低15位截取掉并做饱和处理为1.15格式。
2. 两个矩阵M x N和N x P相乘的结果是M x P. (必须保证一个矩阵的列数等于另一个矩阵的行数)。
3. 矩阵在数组中的存储是从左到右，再从上到下。

22.4.4 函数 arm_mat_mult_fast_q31

函数原型：

```
arm_status arm_mat_mult_fast_q31(
    const arm_matrix_instance_q31 * pSrcA,
    const arm_matrix_instance_q31 * pSrcB,
    arm_matrix_instance_q31 * pDst)
```

函数描述：

这个函数用于定点数 Q31 的矩阵乘法。

函数参数：

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A 乘以矩阵 B 计算结果存储的地址。
- ◆ 返回值，ARM_MATH_SUCCESS 表示成功，ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项：

1. 两个1.31格式的数据相乘产生2.62格式的数据，函数的内部使用了64位的累加器，最终结果要做偏移和饱和运算产生1.31格式数据。
2. 两个矩阵M x N和N x P相乘的结果是M x P. (必须保证一个矩阵的列数等于另一个矩阵的行数)。
3. 矩阵在数组中的存储是从左到右，再从上到下。
4. 函数arm_mat_mult_fast_q31是arm_mat_mult_q31的快速算法。



22.4.5 函数 arm_mat_mult_fast_q15

函数原型:

```
arm_status arm_mat_mult_q15(  
    const arm_matrix_instance_q15 * pSrcA,  
    const arm_matrix_instance_q15 * pSrcB,  
    arm_matrix_instance_q15 * pDst,  
    q15_t * pState)
```

函数描述:

这个函数用于定点数 Q15 的矩阵乘法。

函数参数:

- ◆ 第 1 个参数是矩阵 A 的源地址。
- ◆ 第 2 个参数是矩阵 B 的源地址。
- ◆ 第 3 个参数是矩阵 A 乘以矩阵 B 计算结果存储的地址。
- ◆ 第 4 个参数用于存储内部计算结果。
- ◆ 返回值, ARM_MATH_SUCCESS 表示成功, ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项:

1. 两个1.15格式数据相乘是2.30格式, 函数的内部使用了64位的累加器, 34.30格式, 最终结果将低15位截取掉并做饱和处理为1.15格式。
2. 两个矩阵M x N和N x P相乘的结果是M x P. (必须保证一个矩阵的列数等于另一个矩阵的行数)。
3. 矩阵在数组中的存储是从左到右, 再从上到下。

22.4.6 使用举例 (含 matlab 实现)

程序设计:

```
/*  
*****  
* 函数名: DSP_MatMult  
* 功能说明: 矩阵乘法  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_MatMult(void)  
{  
    uint8_t i;  
  
    /****浮点数数组*****  
    float32_t pDataA[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};  
    float32_t pDataB[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};  
    float32_t pDataDst[9];  
  
    arm_matrix_instance_f32 pSrcA; //3行3列数据  
    arm_matrix_instance_f32 pSrcB; //3行3列数据  
    arm_matrix_instance_f32 pDst;  
  
    /****定点数Q31数组*****  
    q31_t pDataA1[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
    q31_t pDataB1[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
    q31_t pDataDst1[9];  
  
    arm_matrix_instance_q31 pSrcA1; //3行3列数据
```



```
arm_matrix_instance_q31 pSrcB1; //3行3列数据
arm_matrix_instance_q31 pDst1;

/*定点数Q15数组*****
q15_t pDataA2[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};
q15_t pDataB2[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};
q15_t pDataDst2[9];

arm_matrix_instance_q15 pSrcA2; //3行3列数据
arm_matrix_instance_q15 pSrcB2; //3行3列数据
arm_matrix_instance_q15 pDst2;
q15_t pState[9];

/*浮点数*****
pSrcA.numCols = 3;
pSrcA.numRows = 3;
pSrcA.pData = pDataA;

pSrcB.numCols = 3;
pSrcB.numRows = 3;
pSrcB.pData = pDataB;

pDst.numCols = 3;
pDst.numRows = 3;
pDst.pData = pDataDst;

printf("*****浮点数*****\r\n");
arm_mat_mult_f32(&pSrcA, &pSrcB, &pDst);
for(i = 0; i < 9; i++)
{
    printf("pDataDst[%d] = %f\r\n", i, pDataDst[i]);
}

/*定点数Q31*****
pSrcA1.numCols = 3;
pSrcA1.numRows = 3;
pSrcA1.pData = pDataA1;

pSrcB1.numCols = 3;
pSrcB1.numRows = 3;
pSrcB1.pData = pDataB1;

pDst1.numCols = 3;
pDst1.numRows = 3;
pDst1.pData = pDataDst1;

printf("*****定点数Q31*****\r\n");
arm_mat_mult_q31(&pSrcA1, &pSrcB1, &pDst1);
arm_mat_mult_fast_q31(&pSrcA1, &pSrcB1, &pDst1);
for(i = 0; i < 9; i++)
{
    printf("pDataDst1[%d] = %d\r\n", i, pDataDst1[i]);
}

/*定点数Q15*****
pSrcA2.numCols = 3;
pSrcA2.numRows = 3;
pSrcA2.pData = pDataA2;

pSrcB2.numCols = 3;
pSrcB2.numRows = 3;
pSrcB2.pData = pDataB2;

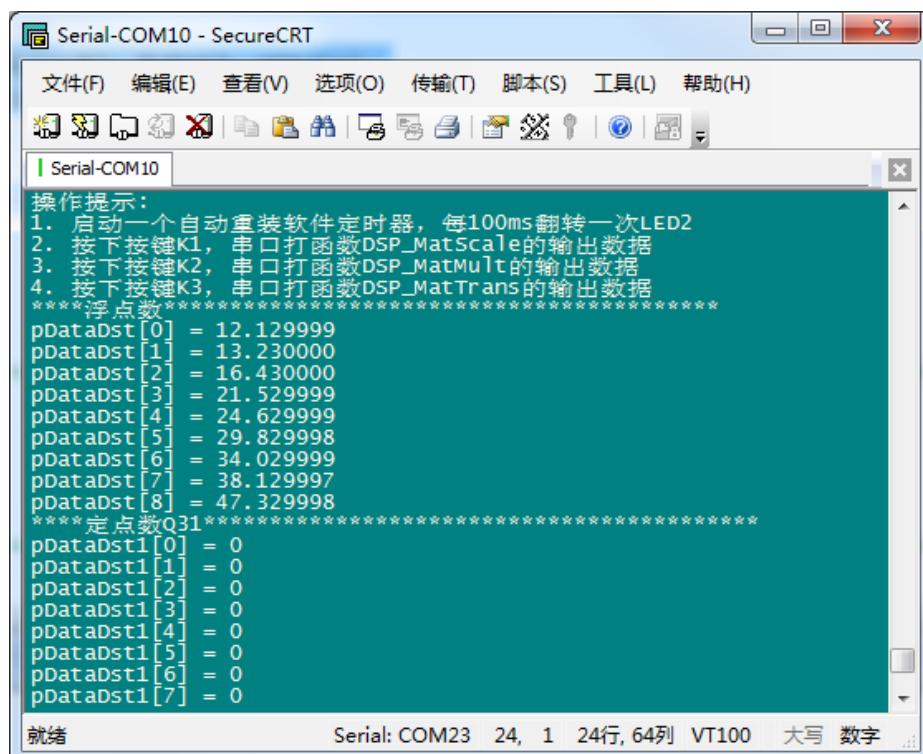
pDst2.numCols = 3;
pDst2.numRows = 3;
pDst2.pData = pDataDst2;

printf("*****定点数Q15*****\r\n");
```

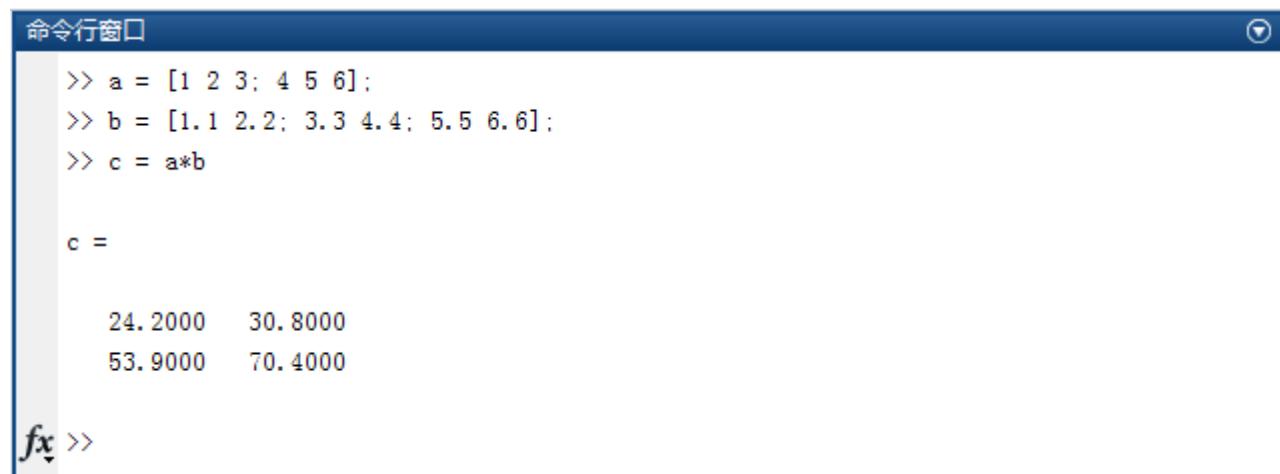


```
arm_mat_mult_q15(&pSrcA2, &pSrcB2, &pDst2, pState);
arm_mat_mult_fast_q15(&pSrcA2, &pSrcB2, &pDst2, pState);
for(i = 0; i < 9; i++)
{
    printf("pDataDst2[%d] = %d\r\n", i, pDataDst2[i]);
}
}
```

实验现象 (按下 K2 按键后串口打印):



下面通过 matlab 来实现矩阵的放缩:



22.5 转置矩阵 MatTrans

以 3*3 矩阵为例, 转置矩阵的实现公式如下:

$$\begin{bmatrix} \underline{\mathbf{a}_{11}} & \mathbf{a}_{12} & \underline{\mathbf{a}_{13}} \\ \mathbf{a}_{21} & \mathbf{a}_{22} & \mathbf{a}_{23} \\ \underline{\mathbf{a}_{31}} & \mathbf{a}_{32} & \underline{\mathbf{a}_{33}} \end{bmatrix}^T = \begin{bmatrix} \underline{\mathbf{a}_{11}} & \mathbf{a}_{21} & \underline{\mathbf{a}_{31}} \\ \mathbf{a}_{12} & \mathbf{a}_{22} & \mathbf{a}_{32} \\ \underline{\mathbf{a}_{13}} & \mathbf{a}_{23} & \underline{\mathbf{a}_{33}} \end{bmatrix}$$

22.5.1 函数 arm_mat_trans_f32

函数原型:

```
arm_status arm_mat_trans_f32(  
    const arm_matrix_instance_f32 * pSrc,  
    arm_matrix_instance_f32 * pDst)
```

函数描述:

这个函数用于浮点数的转置矩阵求解。

函数参数:

- ◆ 第 1 个参数是矩阵源地址。
- ◆ 第 2 个参数是转置后的矩阵地址。
- ◆ 返回值, ARM_MATH_SUCCESS 表示成功, ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项:

1. 矩阵 M × N 转置后是 N × M。也就是说 pSrc 源地址存储的矩阵是 M × N 格式的话, 那么 pDst 地址必须是 N × M 格式。

22.5.2 函数 arm_mat_trans_q31

函数原型:

```
arm_status arm_mat_trans_q31(  
    const arm_matrix_instance_q31 * pSrc,  
    arm_matrix_instance_q31 * pDst)
```

函数描述:

这个函数用于定点数 Q31 的转置矩阵求解。

函数参数:

- ◆ 第 1 个参数是矩阵源地址。
- ◆ 第 2 个参数是转置后的矩阵地址。
- ◆ 返回值, ARM_MATH_SUCCESS 表示成功, ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项:

矩阵 M × N 转置后是 N × M。也就是说 pSrc 源地址存储的矩阵是 M × N 格式的话, 那么 pDst 地址必须是 N × M 格式。



22.5.3 函数 arm_mat_trans_q15

函数原型:

```
arm_status arm_mat_trans_q15(  
    const arm_matrix_instance_q15 * pSrc,  
    arm_matrix_instance_q15 * pDst)
```

函数描述:

这个函数用于定点数 Q15 的转置矩阵求解。

函数参数:

- ◆ 第 1 个参数是矩阵源地址。
- ◆ 第 2 个参数是转置后的矩阵地址。
- ◆ 返回值, ARM_MATH_SUCCESS 表示成功, ARM_MATH_SIZE_MISMATCH 表示矩阵大小不一致。

注意事项:

矩阵 $M \times N$ 转置后是 $N \times M$ 。也就是说 pSrc 源地址存储的矩阵是 $M \times N$ 格式的话, 那么 pDst 地址必须是 $N \times M$ 格式。

22.5.4 使用举例 (含 matlab 实现)

程序设计:

```
/*  
*****  
* 函数名: DSP_MatScale  
* 功能说明: 矩阵放缩  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void DSP_MatScale(void)  
{  
    uint8_t i;  
  
    /****浮点数数组*****  
    float32_t pDataA[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};  
    float32_t scale = 1.1f;  
    float32_t pDataDst[9];  
  
    arm_matrix_instance_f32 pSrcA; //3行3列数据  
    arm_matrix_instance_f32 pDst;  
  
    /****定点数Q31数组*****  
    q31_t pDataA1[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
    q31_t scaleFract = 10;  
    int32_t shift = 0;  
    q31_t pDataDst1[9];  
  
    arm_matrix_instance_q31 pSrcA1; //3行3列数据  
    arm_matrix_instance_q31 pDst1;  
  
    /****定点数Q15数组*****  
    q15_t pDataA2[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};  
    q15_t scaleFract1 = 10;  
    int32_t shift1 = 0;  
    q15_t pDataDst2[9];  
  
    arm_matrix_instance_q15 pSrcA2; //3行3列数据  
    arm_matrix_instance_q15 pDst2;
```



```
/* ****浮点数*****  
pSrcA.numCols = 3;  
pSrcA.numRows = 3;  
pSrcA.pData = pDataA;  
  
pDst.numCols = 3;  
pDst.numRows = 3;  
pDst.pData = pDataDst;  
  
printf("****浮点数*****\r\n");  
arm_mat_scale_f32(&pSrcA, scale, &pDst);  
for(i = 0; i < 9; i++)  
{  
    printf("pDataDst[%d] = %f\r\n", i, pDataDst[i]);  
}  
  
/* ****定点数Q31*****  
pSrcA1.numCols = 3;  
pSrcA1.numRows = 3;  
pSrcA1.pData = pDataA1;  
  
pDst1.numCols = 3;  
pDst1.numRows = 3;  
pDst1.pData = pDataDst1;  
  
printf("****定点数Q31*****\r\n");  
arm_mat_scale_q31(&pSrcA1, scaleFract, shift, &pDst1);  
for(i = 0; i < 9; i++)  
{  
    printf("pDataDst1[%d] = %d\r\n", i, pDataDst1[i]);  
}  
  
/* ****定点数Q15*****  
pSrcA2.numCols = 3;  
pSrcA2.numRows = 3;  
pSrcA2.pData = pDataA2;  
  
pDst2.numCols = 3;  
pDst2.numRows = 3;  
pDst2.pData = pDataDst2;  
  
printf("****定点数Q15*****\r\n");  
arm_mat_scale_q15(&pSrcA2, scaleFract1, shift1, &pDst2);  
for(i = 0; i < 9; i++)  
{  
    printf("pDataDst2[%d] = %d\r\n", i, pDataDst2[i]);  
}  
}  
  
/*  
* 函数名: DSP_MatTrans  
* 功能说明: 矩阵数据初始化  
* 形参: 无  
* 返回值: 无  
*/  
static void DSP_MatTrans(void)  
{  
    uint8_t i;  
  
    /* ****浮点数数组*****  
    float32_t pDataA[9] = {1.1f, 1.1f, 2.1f, 2.1f, 3.1f, 3.1f, 4.1f, 4.1f, 5.1f};  
    float32_t pDataDst[9];  
  
    arm_matrix_instance_f32 pSrcA; //3行3列数据  
    arm_matrix_instance_f32 pDst;
```



```
*****定点数Q31数组*****
q31_t pDataA1[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};
q31_t pDataDst1[9];

arm_matrix_instance_q31 pSrcA1; //3行3列数据
arm_matrix_instance_q31 pDst1;

*****定点数Q15数组*****
q15_t pDataA2[9] = {1, 1, 2, 2, 3, 3, 4, 4, 5};
q15_t pDataDst2[9];

arm_matrix_instance_q15 pSrcA2; //3行3列数据
arm_matrix_instance_q15 pDst2;

*****浮点数*****
pSrcA.numCols = 3;
pSrcA.numRows = 3;
pSrcA.pData = pDataA;

pDst.numCols = 3;
pDst.numRows = 3;
pDst.pData = pDataDst;

printf("*****浮点数*****\r\n");
arm_mat_trans_f32(&pSrcA, &pDst);
for(i = 0; i < 9; i++)
{
    printf("pDataDst[%d] = %f\r\n", i, pDataDst[i]);
}

*****定点数Q31*****
pSrcA1.numCols = 3;
pSrcA1.numRows = 3;
pSrcA1.pData = pDataA1;

pDst1.numCols = 3;
pDst1.numRows = 3;
pDst1.pData = pDataDst1;

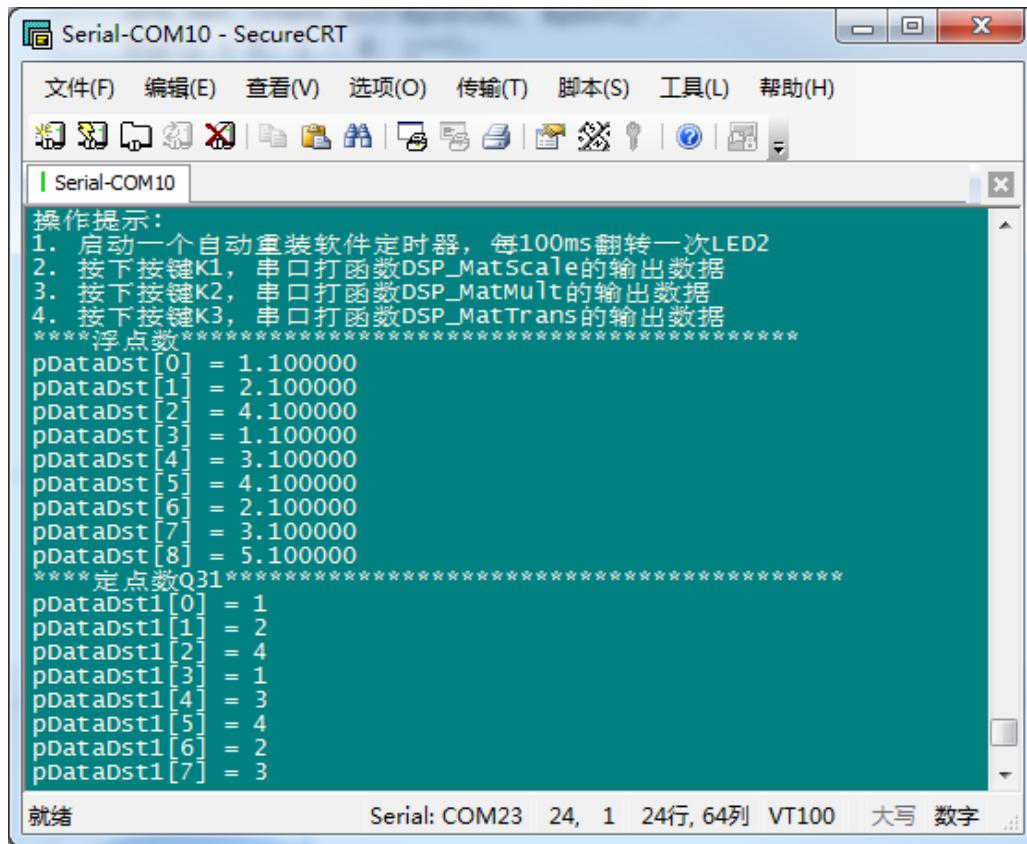
printf("*****定点数Q31*****\r\n");
arm_mat_trans_q31(&pSrcA1, &pDst1);
for(i = 0; i < 9; i++)
{
    printf("pDataDst1[%d] = %d\r\n", i, pDataDst1[i]);
}

*****定点数Q15*****
pSrcA2.numCols = 3;
pSrcA2.numRows = 3;
pSrcA2.pData = pDataA2;

pDst2.numCols = 3;
pDst2.numRows = 3;
pDst2.pData = pDataDst2;

printf("*****定点数Q15*****\r\n");
arm_mat_trans_q15(&pSrcA2, &pDst2);
for(i = 0; i < 9; i++)
{
    printf("pDataDst2[%d] = %d\r\n", i, pDataDst2[i]);
}
```

实验现象 (按下 K3 按键后串口打印):



下面通过 matlab 来实现矩阵的放缩：

```
命令行窗口 ▾

>> a = [1 2 3; 4 5 6];
>> a'
ans =
    1     4
    2     5
    3     6

fx >> |
```

22.6 实验例程说明 (MDK)

配套例子：

V7-217 DSP 矩阵运算 (放缩, 乘法和转置)

实验目的：

- ## 1. 学习 DSP 复数运算 (放缩, 乘法和转置)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打函数 DSP_MatScale 的输出数据。
3. 按下按键 K2，串口打函数 DSP_MatMult 的输出数据。
4. 按下按键 K3，串口打函数 DSP_MatTrans 的输出数据。

使用 AC6 注意事项

特别注意附件章节 C 的问题

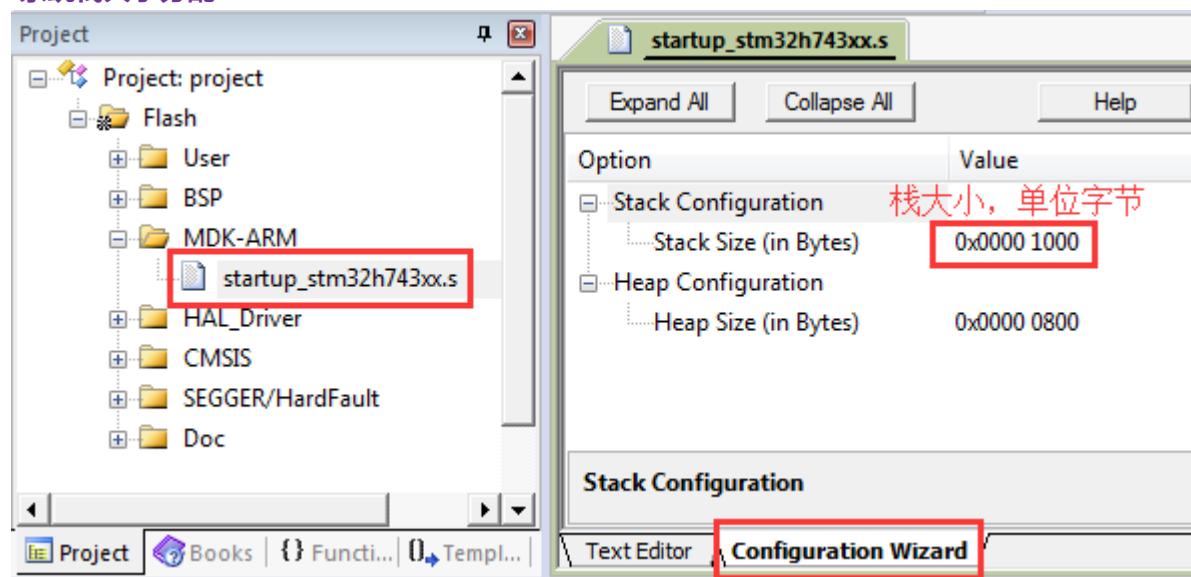
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

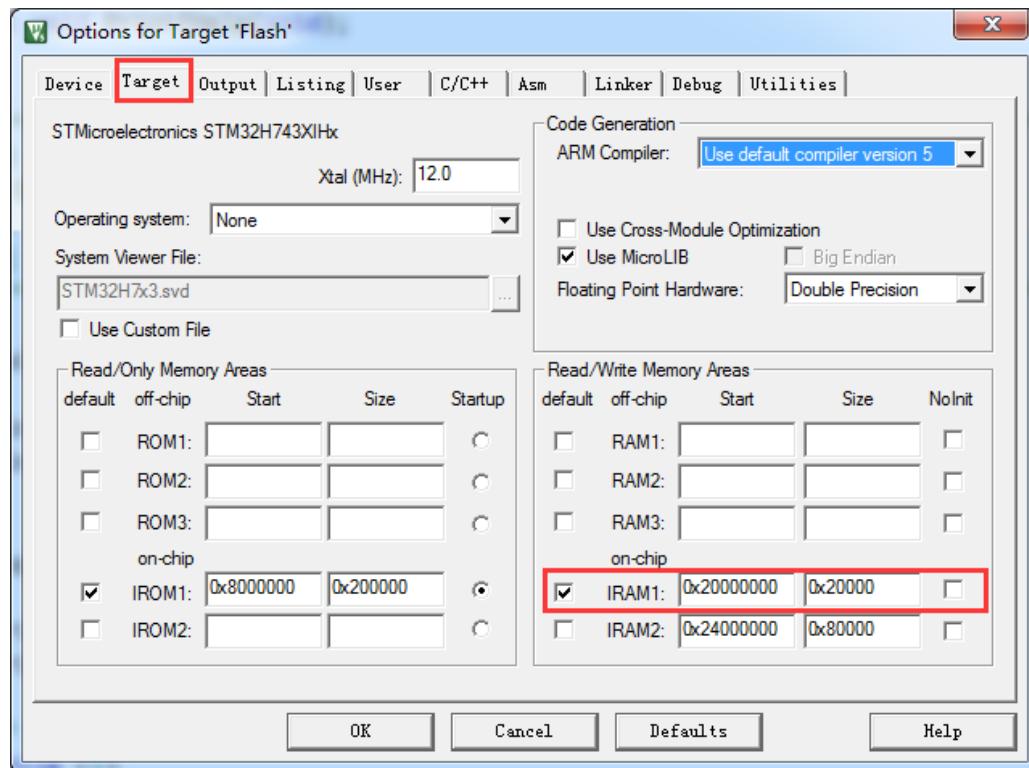
详见本章的 3.4，4.6 和 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打函数 DSP_MatScale 的输出数据。
- 按下按键 K2，串口打函数 DSP_MatMult 的输出数据。
- 按下按键 K3，串口打函数 DSP_MatTrans 的输出数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */
}
```



```
PrintfHelp(); /* 打印操作提示信息 */
```

```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:           /* 按下按键 K1，串口打函数 DSP_MatScale 的输出数据 */
                DSP_MatScale();
                break;

            case KEY_DOWN_K2:           /* 按下按键 K2，串口打函数 DSP_MatMult 的输出数据 */
                DSP_MatMult();
                break;

            case KEY_DOWN_K3:           /* 按下按键 K3，串口打函数 DSP_MatTrans 的输出数据 */
                DSP_MatTrans();
                break;

            default:
                /* 其他的键值不处理 */
                break;
        }
    }
}
```

22.7 实验例程说明 (IAR)

配套例子：

V7-217_DSP 矩阵运算 (放缩, 乘法和转置)

实验目的：

1. 学习 DSP 复数运算 (放缩, 乘法和转置)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打函数 DSP_MatScale 的输出数据。

3. 按下按键 K2，串口打函数 DSP_MatMult 的输出数据。
4. 按下按键 K3，串口打函数 DSP_MatTrans 的输出数据。

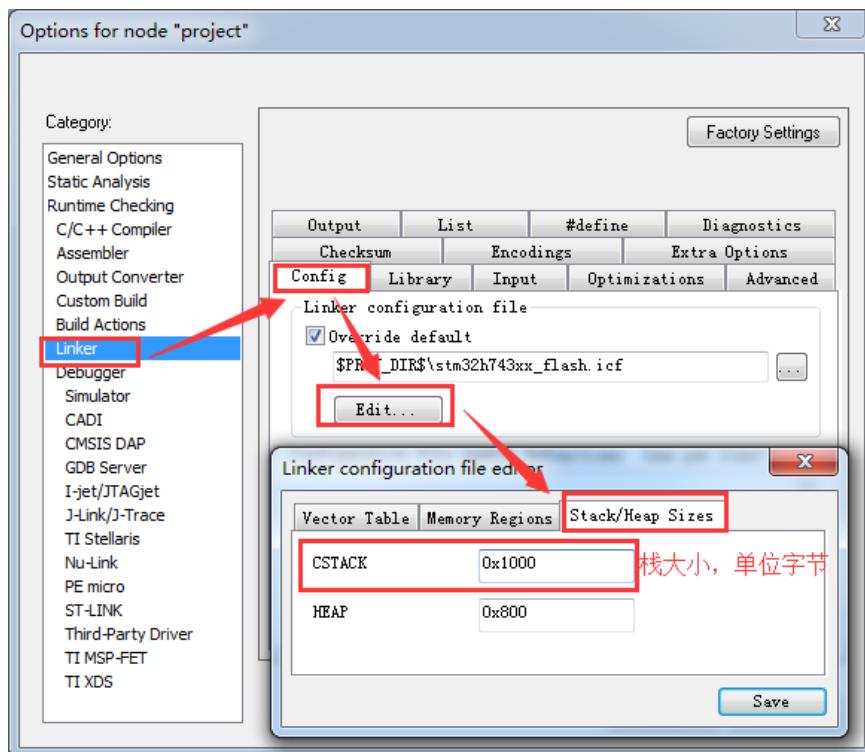
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

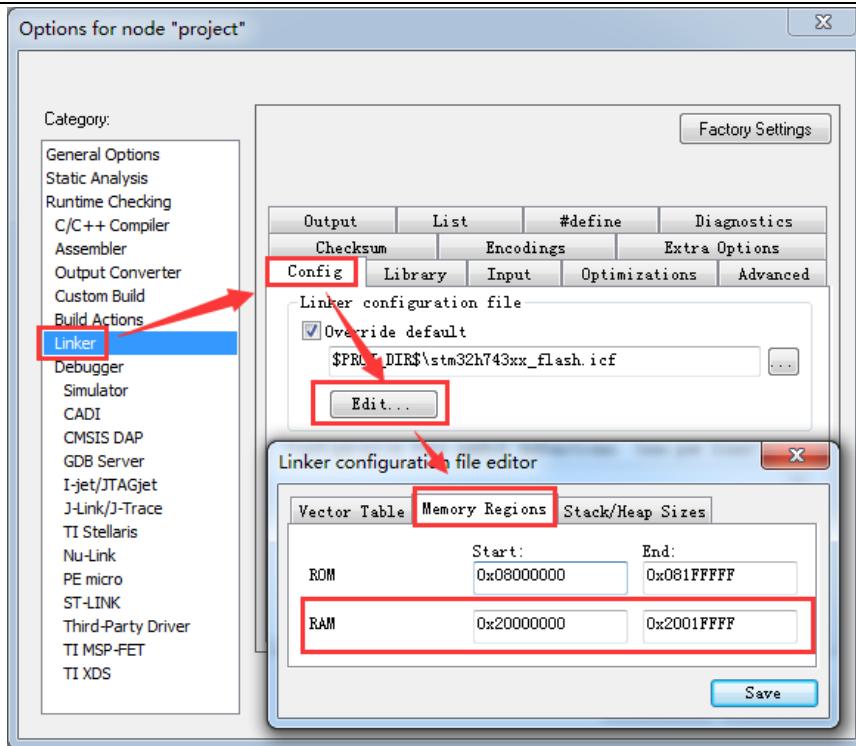
详见本章的 3.4，4.6 和 5.4 小节。

程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打函数 DSP_MatScale 的输出数据。
- 按下按键 K2，串口打函数 DSP_MatMult 的输出数据。
- 按下按键 K3，串口打函数 DSP_MatTrans 的输出数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */\n\n/* 进入主程序循环体 */\nwhile (1)\n{\n    bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */\n\n    /* 判断定时器超时时间 */\n    if (bsp_CheckTimer(0))\n    {\n        /* 每隔 100ms 进来一次 */\n        bsp_LedToggle(2);\n    }\n\n    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */\n    if (ucKeyCode != KEY_NONE)\n    {\n        switch (ucKeyCode)\n        {\n            case KEY_DOWN_K1:           /* 按下按键 K1，串口打函数 DSP_MatScale 的输出数据 */\n                DSP_MatScale();\n                break;\n\n            case KEY_DOWN_K2:           /* 按下按键 K2，串口打函数 DSP_MatMult 的输出数据 */\n                DSP_MatMult();\n                break;\n\n            case KEY_DOWN_K3:           /* 按下按键 K3，串口打函数 DSP_MatTrans 的输出数据 */\n                DSP_MatTrans();\n                break;\n\n            default:\n                /* 其他的键值不处理 */\n                break;\n        }\n    }\n}\n}
```

22.8 总结

本期教程就跟大家讲这么多，有兴趣的可以深入研究下算法的具体实现。



第23章 DSP 辅助运算-math_help 中函数的使

用

本期教程主要讲解 math_help 文件中函数的使用，这个文件也是 ARM 官方提供的，这些函数相对都比较容易，同时使用频率也很高。希望初学的同学学习并掌握。

23.1 初学者重要提示

23.2 函数目录

23.3 函数讲解

23.4 总结

23.1 初学者重要提示

◆ math_help 文件路径，本教程配套的任意一个例子如下路径都有此文件：

\Libraries\CMSIS\DSP\Examples\ARM\arm_graphic_equalizer_example.

23.2 函数目录

在文件 math_help 文件中主要有以下函数：

```
float arm_snr_f32(float *pRef, float *pTest, uint32_t bufferSize)

uint32_t arm_compare_fixed_q31(q31_t *pIn, q31_t *pOut, uint32_t numSamples)
uint32_t arm_compare_fixed_q15(q15_t *pIn, q15_t *pOut, uint32_t numSamples)

void arm_provide_guard_bits_q31 (q31_t * input_buf, uint32_t blockSize, uint32_t guard_bits)
void arm_provide_guard_bits_q15 (q15_t * input_buf, uint32_t blockSize, uint32_t guard_bits)
void arm_provide_guard_bits_q7 (q7_t * input_buf, uint32_t blockSize, uint32_t guard_bits)

uint32_t arm_calc_guard_bits (uint32_t num_adds)
void arm_apply_guard_bits (float32_t *pIn, uint32_t numSamples, uint32_t guard_bits)

uint32_t arm_calc_2pow(uint32_t numShifts)
void arm_clip_f32 (float *pIn, uint32_t numSamples)

void arm_float_to_q12_20(float *pIn, q31_t *pOut, uint32_t numSamples)
void arm_float_to_q14 (float *pIn, q15_t *pOut, uint32_t numSamples)
void arm_float_to_q28 (float *pIn, q31_t *pOut, uint32_t numSamples)
void arm_float_to_q29 (float *pIn, q31_t *pOut, uint32_t numSamples)
void arm_float_to_q30 (float *pIn, q31_t *pOut, uint32_t numSamples)
```



23.3 函数讲解

23.3.1 函数 arm_snr_f32

这个函数用于求信噪比：

```
/**  
 * @brief Caluclation of SNR  
 * @param[in] pRef Pointer to the reference buffer  
 * @param[in] pTest Pointer to the test buffer  
 * @param[in] bufferSize total number of samples  
 * @return SNR  
 * The function Caluclates signal to noise ratio for the reference output  
 * and test output  
 */  
float arm_snr_f32(float *pRef, float *pTest, uint32_t bufferSize)  
{  
    float EnergySignal = 0.0, EnergyError = 0.0;  
    uint32_t i;  
    float SNR;  
    int temp;  
    int *test;  
  
    for (i = 0; i < bufferSize; i++)  
    {  
        /* 检测是否为 NAN 非值 */  
        test = (int *)(&pRef[i]);  
        temp = *test;  
  
        if (temp == 0x7FC00000)  
        {  
            return(0);  
        }  
  
        test = (int *)(&pTest[i]);  
        temp = *test;  
  
        if (temp == 0x7FC00000)  
        {  
            return(0);  
        }  
        EnergySignal += pRef[i] * pRef[i];  
        EnergyError += (pRef[i] - pTest[i]) * (pRef[i] - pTest[i]);  
    }  
  
    test = (int *)(&EnergyError);  
    temp = *test;  
  
    if (temp == 0x7FC00000)  
    {  
        return(0);  
    }  
  
    SNR = 10 * log10 (EnergySignal / EnergyError);  
  
    return (SNR);  
}
```

信噪比，即 SNR (Signal to Noise Ratio) 是指一个电子设备或者电子系统中信号与噪声的比例。这里面的信号指的是来自设备外部需要通过这台设备进行处理的电子信号，噪声是指经过该设备后产生的原



信号中并不存在的无规则的额外信号（或信息），并且该种信号并不随原信号的变化而变化。

狭义来讲是指放大器的输出信号功率与输出噪声功率的比，常常用分贝数表示，设备的信噪比越高表明它产生的杂音越少。一般来说，信噪比越大，说明混在信号里的噪声越小。

公式如下：有用信号功率（Power of Signal）与噪声功率（Power of Noise）的比，那么幅度（Amplitude）平方的比如下：

$$\text{SNR} = \frac{P_{\text{signal}}}{P_{\text{noise}}} = \frac{A_{\text{signal}}^2}{A_{\text{noise}}^2}$$

单位一般使用分贝，其值为十倍对数的信号功率与噪声功率比：

$$\text{SNR(dB)} = 10 \log_{10} \left(\frac{P_{\text{signal}}}{P_{\text{noise}}} \right) = 20 \log_{10} \left(\frac{A_{\text{signal}}}{A_{\text{noise}}} \right)$$

其中

P_{signal} 为信号功率（Power of Signal）。

P_{noise} 为噪声功率（Power of Noise）。

A_{signal} 为信号幅度（Amplitude of Signal）。

A_{noise} 为噪声幅度（Amplitude of Noise）。

程序里面的 $10 * \log10(\text{EnergySignal} / \text{EnergyError})$ 对应的就是上面的公式实现。

23.3.2 函数 arm_compare_fixed_q31

```
/**  
 * @brief Compare MATLAB Reference Output and ARM Test output  
 * @param q31_t* Pointer to Ref buffer  
 * @param q31_t* Pointer to Test buffer  
 * @param uint32_t    number of samples in the buffer  
 * @return none  
 */  
uint32_t arm_compare_fixed_q31(q31_t *pIn, q31_t *pOut, uint32_t numSamples)  
{  
    uint32_t i;  
    int32_t diff, diffCrnt = 0;  
    uint32_t maxDiff = 0;  
  
    for (i = 0; i < numSamples; i++)  
    {  
        diff = pIn[i] - pOut[i];  
        diffCrnt = (diff > 0) ? diff : -diff;  
  
        if (diffCrnt > maxDiff)  
        {  
            maxDiff = diffCrnt;  
        }  
    }  
  
    return (maxDiff);  
}
```

这个函数用于对比 matlab 和 ARM 实际计算的输出，并返回最大的差值（Q31）。



23.3.3 函数 arm_compare_fixed_q15

```
/**  
 * @brief Compare MATLAB Reference Output and ARM Test output  
 * @param q15_t* Pointer to Ref buffer  
 * @param q15_t* Pointer to Test buffer  
 * @param uint32_t    number of samples in the buffer  
 * @return none  
 */  
uint32_t arm_compare_fixed_q15(q15_t *pIn, q15_t *pOut, uint32_t numSamples)  
{  
    uint32_t i;  
    int32_t diff, diffCrnt = 0;  
    uint32_t maxDiff = 0;  
  
    for (i = 0; i < numSamples; i++)  
    {  
        diff = pIn[i] - pOut[i];  
        diffCrnt = (diff > 0) ? diff : -diff;  
  
        if (diffCrnt > maxDiff)  
        {  
            maxDiff = diffCrnt;  
        }  
    }  
  
    return (maxDiff);  
}
```

这个函数用于对比 matlab 和 ARM 实际计算的输出，并返回最大的差值 (Q15)。

23.3.4 函数 arm_provide_guard_bits_q31

```
/**  
 * @brief Provide guard bits for Input buffer  
 * @param q31_t* Pointer to input buffer  
 * @param uint32_t    blockSize  
 * @param uint32_t    guard_bits  
 * @return none  
 * The function Provides the guard bits for the buffer  
 * to avoid overflow  
 */  
void arm_provide_guard_bits_q31 (q31_t * input_buf,  
                                 uint32_t blockSize,  
                                 uint32_t guard_bits)  
{  
    uint32_t i;  
  
    for (i = 0; i < blockSize; i++)  
    {  
        input_buf[i] = input_buf[i] >> guard_bits;  
    }  
}
```

这个函数用于给 q31_t 类型的数据提供保护位，防止溢出。从函数的实现上看，保护位的实现是通过右移数据实现的。

23.3.5 函数 arm_provide_guard_bits_q15

```
/**  
 * @brief Provide guard bits for Input buffer  
 * @param q15_t*    Pointer to input buffer
```



```
* @param uint32_t blockSize
* @param uint32_t guard_bits
* @return none
* The function Provides the guard bits for the buffer
* to avoid overflow
*/
void arm_provide_guard_bits_q15 (q15_t * input_buf, uint32_t blockSize,
                                 uint32_t guard_bits)
{
    uint32_t i;

    for (i = 0; i < blockSize; i++)
    {
        input_buf[i] = input_buf[i] >> guard_bits;
    }
}
```

这个函数用于给 q15_t 类型的数据提供保护位，防止溢出。

23.3.6 函数 arm_provide_guard_bits_q7

```
/** 
* @brief Provide guard bits for Input buffer
* @param[in,out] input_buf Pointer to input buffer
* @param[in]     blockSize block Size
* @param[in]     guard_bits guard bits
* @return none
* The function Provides the guard bits for the buffer
* to avoid overflow
*/
void arm_provide_guard_bits_q7 (q7_t * input_buf,
                                uint32_t blockSize,
                                uint32_t guard_bits)
{
    uint32_t i;

    for (i = 0; i < blockSize; i++)
    {
        input_buf[i] = input_buf[i] >> guard_bits;
    }
}
```

这个函数用于给 q7_t 类型的数据提供保护位，防止溢出。

23.3.7 函数 arm_calc_guard_bits

```
/** 
* @brief Caluclates number of guard bits
* @param[in] num_adds number of additions
* @return guard bits
* The function Caluclates the number of guard bits
* depending on the numtaps
*/
uint32_t arm_calc_guard_bits (uint32_t num_adds)
{
    uint32_t i = 1, j = 0;

    if (num_adds == 1)
    {
        return (0);
    }

    while (i < num_adds)
```



```
{  
    i = i * 2;  
    j++;  
}  
  
return (j);
```

这个函数是根据加数的个数来计算最终结果需要的保护位格式，从而防止溢出。

23.3.8 函数 arm_apply_guard_bits

```
/**  
 * @brief Apply guard bits to buffer  
 * @param[in,out] pIn pointer to input buffer  
 * @param[in] numSamples number of samples in the input buffer  
 * @param[in] guard_bits guard bits  
 * @return none  
 */  
void arm_apply_guard_bits (float32_t *pIn,  
                           uint32_t numSamples,  
                           uint32_t guard_bits)  
{  
    uint32_t i;  
  
    for (i = 0; i < numSamples; i++)  
    {  
        pIn[i] = pIn[i] * arm_calc_2pow(guard_bits);  
    }  
}
```

这个函数不是很理解在实际应用中的作用。

23.3.9 函数 arm_calc_2pow

```
/**  
 * @brief Calculates pow(2, numShifts)  
 * @param uint32_t number of shifts  
 * @return pow(2, numShifts)  
 */  
uint32_t arm_calc_2pow(uint32_t numShifts)  
{  
  
    uint32_t i, val = 1;  
  
    for (i = 0; i < numShifts; i++)  
    {  
        val = val * 2;  
    }  
  
    return (val);  
}
```

这个函数用于求解 2 的 n 次方。

23.3.10 函数 arm_clip_f32

```
/**  
 * @brief Clip the float values to +/- 1  
 * @param[in,out] pIn input buffer  
 * @param[in] numSamples number of samples in the buffer  
 * @return none  
 * The function converts floating point values to fixed point values  
 */  
void arm_clip_f32 (float *pIn, uint32_t numSamples)
```



```
{  
    uint32_t i;  
  
    for (i = 0; i < numSamples; i++)  
    {  
        if (pIn[i] > 1.0f)  
        {  
            pIn[i] = 1.0;  
        }  
        else if (pIn[i] < -1.0f)  
        {  
            pIn[i] = -1.0;  
        }  
    }  
}
```

浮点数裁剪函数，大于 1 的赋值为 1，小于-1 的赋值为-1。

23.3.11 函数 arm_float_to_q12_20

```
/**  
 * @brief Converts float to fixed in q12.20 format  
 * @param uint32_t number of samples in the buffer  
 * @return none  
 * The function converts floating point values to fixed point(q12.20) values  
 */  
void arm_float_to_q12_20(float *pIn, q31_t *pOut, uint32_t numSamples)  
{  
    uint32_t i;  
  
    for (i = 0; i < numSamples; i++)  
    {  
        /* 1048576.0f corresponds to pow(2, 20) */  
        pOut[i] = (q31_t) (pIn[i] * 1048576.0f);  
  
        pOut[i] += pIn[i] > 0 ? 0.5 : -0.5;  
  
        if (pIn[i] == (float) 1.0)  
        {  
            pOut[i] = 0x000FFFFF;  
        }  
    }  
}
```

- ◆ 这个函数用于将浮点数转换成 Q12.20 格式的数据。
- ◆ 浮点数转换成 Q12.20 格式需要乘以 2^{20} ，然后再做舍入处理，特别注意这里的浮点数末尾要加符号 f，这样才能保证是单精度。

23.3.12 函数 arm_float_to_q14

```
/**  
 * @brief Converts float to fixed q14  
 * @param uint32_t number of samples in the buffer  
 * @return none  
 * The function converts floating point values to fixed point values  
 */  
void arm_float_to_q14 (float *pIn, q15_t *pOut,  
                      uint32_t numSamples)  
{  
    uint32_t i;  
  
    for (i = 0; i < numSamples; i++)  
    {
```



```
/* 16384.0f corresponds to pow(2, 14) */
pOut[i] = (q15_t) (pIn[i] * 16384.0f);

pOut[i] += pIn[i] > 0 ? 0.5 : -0.5;

if (pIn[i] == (float) 2.0)
{
    pOut[i] = 0x7FFF;
}

}
```

- ◆ 这个函数用于将浮点数转换成 Q14 格式的数据。
- ◆ 浮点数转换成 Q14 格式需要乘以 2^{14} , 然后再做舍入处理。

23.3.13 函数 arm_float_to_q28

```
/** 
 * @brief Converts float to fixed q28 format
 * @param[in] pIn pointer to input buffer
 * @param[out] pOut pointer to output buffer
 * @param[in] numSamples number of samples in the buffer
 * @return none
 * The function converts floating point values to fixed point values
 */

void arm_float_to_q28 (float *pIn, q31_t *pOut, uint32_t numSamples)
{
    uint32_t i;

    for (i = 0; i < numSamples; i++)
    {
        /* 268435456.0f corresponds to pow(2, 28) */
        pOut[i] = (q31_t) (pIn[i] * 268435456.0f);

        pOut[i] += pIn[i] > 0 ? 0.5 : -0.5;

        if (pIn[i] == (float) 8.0)
        {
            pOut[i] = 0x7FFFFFFF;
        }
    }
}
```

- ◆ 这个函数用于将浮点数转换成 Q28 格式的数据。
- ◆ 浮点数转换成 Q28 格式需要乘以 2^{28} , 然后再做舍入处理。

23.3.14 函数 arm_float_to_q29

```
/** 
 * @brief Converts float to fixed q30 format
 * @param uint32_t number of samples in the buffer
 * @return none
 * The function converts floating point values to fixed point values
 */
void arm_float_to_q29 (float *pIn, q31_t *pOut,
                      uint32_t numSamples)
{
    uint32_t i;

    for (i = 0; i < numSamples; i++)
    {
        /* 536870912.0f corresponds to pow(2, 29) */
        pOut[i] = (q31_t) (pIn[i] * 536870912.0f);
    }
}
```



```
pOut[i] += pIn[i] > 0 ? 0.5 : -0.5;

if (pIn[i] == (float) 4.0)
{
    pOut[i] = 0x7FFFFFFF;
}
}
```

- ◆ 这个函数用于将浮点数转换成 Q29 格式的数据。
- ◆ 浮点数转换成 Q29 格式需要乘以 2^{29} , 然后再做舍入处理。

23.3.15 函数 arm_float_to_q30

```
/** @brief Converts float to fixed q30 format
 * @param uint32_t number of samples in the buffer
 * @return none
 * The function converts floating point values to fixed point values
 */
void arm_float_to_q29 (float *pIn, q31_t *pOut,
                      uint32_t numSamples)
{
    uint32_t i;

    for (i = 0; i < numSamples; i++)
    {
        /* 1073741824.0f corresponds to pow(2, 30) */
        pOut[i] = (q31_t) (pIn[i] * 536870912.0f);

        pOut[i] += pIn[i] > 0 ? 0.5 : -0.5;

        if (pIn[i] == (float) 4.0)
        {
            pOut[i] = 0x7FFFFFFF;
        }
    }
}
```

- ◆ 这个函数用于将浮点数转换成 Q30 格式的数据。
- ◆ 浮点数转换成 Q30 格式需要乘以 2^{30} , 然后再做舍入处理。

23.4 总结

本期教程就跟大家讲这么多，本期教程就不配套例子了，这些函数会在后面的应用中都用到。



第24章 DSP 变换运算-傅里叶变换

本章节开始进入此教程最重要的知识点之一傅里叶变换。关于傅里叶变换，本章主要是把傅里叶相关的基础知识进行必要的介绍，没有这些基础知识的话，后面学习 FFT（快速傅里叶变换）时会比较困难。本章节的内容主要来自百度百科，wiki 百科以及网络和书籍中整理的一些资料。

24.1 初学者重要提示

24.2 傅里叶人物简介

24.3 傅里叶变换概念

24.4 傅里叶的特殊形式

24.5 傅里叶变换相关知识

24.6 总结

24.1 初学者重要提示

- ◆ 为大家推荐西电的国家级精品课程信号与系统，含课堂视频，书籍和课堂 PPT
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=94886>。
- ◆ 非非常好的 DSP 基础知识普及书籍：
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=97312>。
- ◆ 通过本章学习要搞清楚为什么计算机要选择 DFT 作为运算基础，而不是 DTFT 或其它。
- ◆ 通过傅里叶的学习，我们知道任何波形都可以使用正弦波无限逼近，但是为什么选择的是正弦波，而不是三角波或者方波，本章也进行了解释。

24.2 傅里叶人物简介

学习傅里叶变换前，一定要对傅里叶这个人有所了解，这样更加有利于学习他提出的理论。

让·巴普蒂斯·约瑟夫·傅立叶 (Jean Baptiste Joseph Fourier, 1768 –1830)，法国著名数学家、物理学家，1817 年当选为科学院院士，1822 年任该院终身秘书，后又任法兰西学院终身秘书和理工科大学校务委员会主席，主要贡献是在研究热的传播时创立了一套数学理论。

傅立叶生于法国中部欧塞尔 (Auxerre) 一个裁缝家庭，8 岁时沦为孤儿，就读于地方军校，1795 年任巴黎综合工科大学助教，1798 年随拿破仑军队远征埃及，受到拿破仑器重，回国后被任命为格伦诺布尔省省长。

傅立叶早在 1807 年就写成关于热传导的基本论文《热的传播》，向巴黎科学院呈交，但经拉格朗日、



拉普拉斯和勒让德审阅后被科学院拒绝，1811 年又提交了经修改的论文，该文获科学院大奖，却未正式发表。傅立叶在论文中推导出著名的热传导方程，并在求解该方程时发现解函数可以由三角函数构成的级数形式表示，从而提出任一函数都可以展成三角函数的无穷级数。傅立叶级数（即三角级数）、傅立叶分析等理论均由此创始。

傅立叶由于对传热理论的贡献于 1817 年当选为巴黎科学院院士。

1822 年，傅立叶终于出版了专著《热的解析理论》(Theorieanalytique de la Chaleur , Didot , Paris, 1822)。这部经典著作将欧拉、伯努利等人在一些特殊情形下应用的三角级数方法发展成内容丰富的一般理论，三角级数后来就以傅立叶的名字命名。傅立叶应用三角级数求解热传导方程，为了处理无穷区域的热传导问题又导出了当前所称的“傅立叶积分”，这一切都极大地推动了偏微分方程边值问题的研究。然而傅立叶的工作意义远不止此，它迫使人们对函数概念作修正、推广，特别是引起了对不连续函数的探讨；三角级数收敛性问题更刺激了集合论的诞生。因此，《热的解析理论》影响了整个 19 世纪分析严格化的进程。傅立叶 1822 年成为科学院终身秘书。

24.3 傅里叶变换概念

关于傅里叶变换的概念和公式方面的内容需要大家重新拿起当年的《信号与系统》课本温习下，不过在温习课本之前强烈建议看下这个帖子：<http://zhuanlan.zhihu.com/wille/19763358>（傅里叶分析之掐死教程（完整版））。这个帖子非常适合没有傅里叶变换方面知识的同学看，如果有傅里叶方面的基础认识的话，看完这个帖子，你的认识会更加的深刻。

这里我只把傅里叶简单的定义贴上进行说明。傅立叶变换是一种分析信号的方法，它可分析信号的成分，也可用这些成分合成信号。许多波形可作为信号的成分，比如正弦波、方波、锯齿波等，傅立叶变换用正弦波作为信号的成分。

假设 $f(t)$ 是 t 的周期函数，如果 t 满足狄里赫莱条件：在一个周期内具有有限个间断点，且在这些间断点上，函数是有限值；在一个周期内具有有限个极值点；绝对可积。则有下图①式成立。称为积分运算 $f(t)$ 的傅立叶变换，②式的积分运算叫做 $F(\omega)$ 的傅立叶逆变换。 $F(\omega)$ 叫做 $f(t)$ 的像函数， $f(t)$ 叫做 $F(\omega)$ 的像原函数。 $F(\omega)$ 是 $f(t)$ 的像。 $f(t)$ 是 $F(\omega)$ 原像。

①傅立叶变换

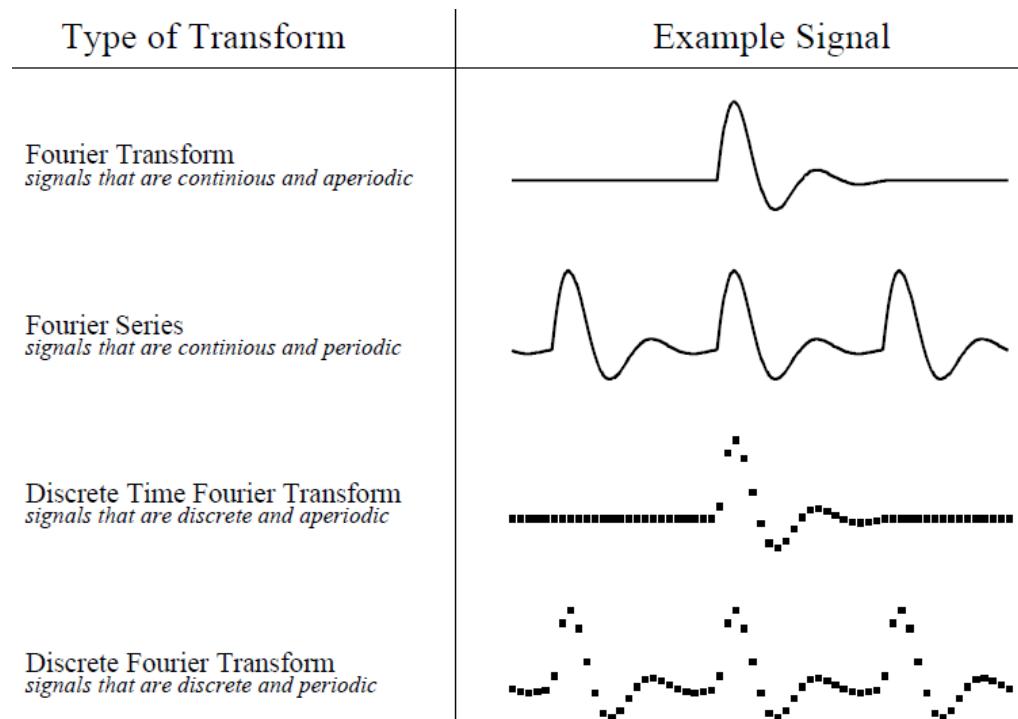
$$F(\omega) = \mathcal{F}[f(t)] = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt$$

②傅立叶逆变换

$$f(t) = \mathcal{F}^{-1}[F(\omega)] = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{i\omega t} d\omega$$

24.4 傅里叶的特殊变换形式

下面要讲到的这几种概念很重要，初学时不要搞混淆了。



上图做的非常好，从上到下依次是：

1	非周期性连续信号	傅立叶变换 (Fourier Transform)
2	周期性连续信号	傅立叶级数(Fourier Series)
3	非周期性离散信号	离散时间傅立叶变换 (Discrete Time Fourier Transform)
4	周期性离散信号	离散傅立叶变换(Discrete Fourier Transform)

24.4.1 连续傅里叶变换 (Fourier Transform)

一般情况下，若“傅里叶变换”一词的前面未加任何限定语，则指的是连续傅里叶变换。连续傅里叶变换将平方可积的函数表示成复指数函数的积分形式：

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{i\omega t} d\omega$$

上式其实表示的是连续傅里叶变换的逆变换，即将时间域的函数表示为频率域的函数 $F(\omega)$ 的积分。反过来，其正变换恰好是将频率域的函数 $F(\omega)$ 表示为时间域的函数 $f(t)$ 的积分形式。一般称函数 $f(t)$ 为



原函数，而称函数 $F(\omega)$ 为傅里叶变换的像函数，原函数和像函数构成一个傅里叶变换对 (transform pair)。

当 $f(t)$ 为奇函数 (或偶函数) 时，其余弦 (或正弦) 分量为零，而可以称这时的变换为余弦变换 (或正弦变换)。

24.4.2 傅里叶级数 (Fourier series)

连续形式的傅里叶变换其实是傅里叶级数的推广，因为积分其实是一种极限形式的求和算子而已。

对于周期函数，它的傅里叶级数 (Fourier series) 表示被定义为：

$$f(t) = \sum_{n=-\infty}^{\infty} F_n e^{i2\pi nt/T}$$

其中 T 为函数的周期， F_n 为傅里叶展开系数，它们等于

$$F_n = \frac{1}{T} \int_{-T/2}^{T/2} f(t) e^{-i2\pi nt/T} dt$$

对于实值函数，函数的傅里叶级数可以写成：

$$f(t) = \frac{a_0}{2} + \sum_{n=1}^{\infty} [a_n \cos\left(\frac{2\pi nt}{T}\right) + b_n \sin\left(\frac{2\pi nt}{T}\right)]$$

其中 a_n 和 b_n 是实频率分量的振幅。

24.4.3 离散时间傅里叶变换 (Discrete-time Fourier transform)

离散时间傅里叶变换 (discrete-time Fourier transform, DTFT) 针对的是定义域为 Z 的数列。设 $\{x_n\}_{n=-\infty}^{\infty}$ 为某一数列，则其 DTFT 被定义为

$$X(\omega) = \sum_{-\infty}^{\infty} x_n e^{-i\omega n}$$

相应的逆变换为

$$x_n = \frac{1}{2\pi} \int_{-\infty}^{+\infty} X(\omega) e^{i\omega n} d\omega$$

DTFT 在时域上离散，在频域上则是周期的，它一般用来对离散时间信号进行频谱分析。DTFT 可以被看作是傅里叶级数的逆。

24.4.4 离散傅里叶变换 (Discrete Fourier transform)

为了在科学计算和数字信号处理等领域使用计算机进行傅里叶变换，必须将函数定义在离散点上而非连续域内，且须满足有限性或周期性条件。这种情况下，序列 $\{x_n\}_{n=0}^{N-1}$ 的离散傅里叶变换 (discrete Fourier transform, DFT) 为

$$X(k) = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$$



其逆变换为

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{i2\pi kn/N}$$

直接使用 DFT 的定义计算的复杂度为 $O(N^2)$ ，而快速傅里叶变换 (fast Fourier transform, FFT) 可以将复杂度改进为 $O(n \log n)$ 。计算复杂度的降低以及数字电路计算能力的发展使得 DFT 成为在信号处理领域十分实用且重要的方法。

24.4.5 傅里叶变换家族

下表列出了傅里叶变换家族的成员。容易发现，函数在时（频）域的离散对应于其像函数在频（时）域的周期性，反之连续则意味着在对应域的信号的非周期性。

变换	时间域	频率域
连续傅里叶变换	连续， 非周期性	连续， 非周期性
傅里叶级数	连续， 周期性	离散， 非周期性
离散时间傅里叶变换	离散， 非周期性	连续， 周期性
离散傅里叶变换	离散， 周期性	离散， 周期性

24.5 傅里叶变换相关知识 (重要)

24.5.1 傅里叶变换的提出

傅里叶是一位法国数学家和物理学家的名字，英语原名是 Jean Baptiste Joseph Fourier(1768-1830), Fourier 对热传递很感兴趣，于 1807 年在法国科学学会上发表了一篇论文，运用正弦曲线来描述温度分布，论文里有个在当时具有争议性的决断：任何连续周期信号可以由一组适当的正弦曲线组合而成。当时审查这个论文的人，其中有两位是历史上著名的数学家拉格朗日(Joseph Louis Lagrange, 1736-1813)和拉普拉斯(Pierre Simon de Laplace, 1749-1827)，当拉普拉斯和其它审查者投票通过并要发表这个论文时，拉格朗日坚决反对，在他此后生命的六年中，拉格朗日坚持认为傅里叶的方法无法表示带有棱角的信号，如在方波中出现非连续变化斜率。法国科学学会屈服于拉格朗日的威望，拒绝了傅里叶的工作，幸运的是，傅里叶还有其它事情可忙，他参加了政治运动，随拿破仑远征埃及，法国大革命后因会被推上断头台而一直在逃避。直到拉格朗日死后 15 年这个论文才被发表出来。

拉格朗日是对的：正弦曲线无法组合成一个带有棱角的信号。但是，我们可以用正弦曲线来非常逼近地表示它，逼近到两种表示方法不存在能量差别，基于此，傅里叶是对的。

用正弦曲线来代替原来的曲线而不用方波或三角波来表示的原因在于，分解信号的方法是无穷的，但分解信号的目的是为了更加简单地处理原来的信号。用正余弦来表示原信号会更加简单，因为正余弦拥有原信号所不具有的性质：正弦曲线保真度。一个正弦曲线信号输入后，输出的仍是正弦曲线，只有



幅度和相位可能发生变化，但是频率和波的形状仍是一样的。且只有正弦曲线才拥有这样的性质，正因如此我们才不用方波或三角波来表示。

24.5.2 傅里叶变换分类

根据原信号的不同类型，我们可以把傅里叶变换分为四种类别：

这四种傅里叶变换都是针对正无穷大和负无穷大的信号，即信号的长度是无穷大的，我们知道这对于计算机处理来说是不可能的，那么有没有针对长度有限的傅里叶变换呢？没有。因为正余弦波被定义成从负无穷大到正无穷大，我们无法把一个长度无限的信号组合成长度有限的信号。面对这种困难，方法是把长度有限的信号表示成长度无限的信号，可以把信号无限地从左右进行延伸，延伸的部分用零来表示，这样，这个信号就可以被看成是非周期性离散信号，我们就可以用到离散时间傅里叶变换的方法。还有，也可以把信号用复制的方法进行延伸，这样信号就变成了周期性离散信号，这时我们就可以用离散傅里叶变换方法进行变换。这里我们要学的是离散信号，对于连续信号我们不作讨论，因为计算机只能处理离散的数值信号，我们的最终目的是运用计算机来处理信号的。

但是对于非周期性的信号，我们需要用无穷多不同频率的正弦曲线来表示，这对于计算机来说是不可能实现的。所以对于离散信号的变换只有离散傅里叶变换（DFT）才能被适用，对于计算机来说只有离散的和有限长度的数据才能被处理，对于其它的变换类型只有在数学演算中才能用到，在计算机面前我们只能用 DFT 方法，后面我们要理解的也正是 DFT 方法。这里要理解的是我们使用周期性的信号目的是为了能够用数学方法来解决问题，至于考虑周期性信号是从哪里得到或怎样得到是无意义的。

每种傅里叶变换都分成实数和复数两种方法，对于实数方法是最好理解的，但是复数方法就相对复杂许多了，需要懂得有关复数的理论知识，不过，如果理解了实数离散傅里叶变换(real DFT)，再去理解复数傅里叶就更容易了。

还有，这里我们所要说的变换(transform)虽然是数学意义上的变换，但跟函数变换是不同的，函数变换是符合一一映射准则的，对于离散数字信号处理（DSP），有许多的变换：傅里叶变换、拉普拉斯变换、Z 变换、希尔伯特变换、离散余弦变换等，这些都扩展了函数变换的定义，允许输入和输出有多种的值，简单地说变换就是把一堆的数据变成另一堆数据的方法。

24.5.3 傅里叶变换的意义

傅里叶变换是数字信号处理领域一种很重要的算法。要知道傅里叶变换算法的意义，首先要了解傅里叶原理的意义。傅里叶原理表明：任何连续测量的时序或信号，都可以表示为不同频率的正弦波信号的无限叠加。而根据该原理创立的傅里叶变换算法利用直接测量到的原始信号，以累加的方式来计算该信号中不同正弦波信号的频率、振幅和相位。

和傅里叶变换算法对应的是傅里叶逆变换算法。该逆变换从本质上说也是一种累加处理，这样就可以将单独改变的正弦波信号转换成一个信号。因此，可以说，傅里叶变换将原来难以处理的时域信号转换成了易于分析的频域信号（信号的频谱），可以利用一些工具对这些频域信号进行处理、加工。最后还可以利用傅里叶逆变换将这些频域信号转换成时域信号。



从现代数学的眼光来看，傅里叶变换是一种特殊的积分变换。它能将满足一定条件的某个函数表示成正弦基函数的线性组合或者积分。在不同的研究领域，傅里叶变换具有多种不同的变体形式，如连续傅里叶变换和离散傅里叶变换。

在数学领域，尽管最初傅里叶分析是作为热过程的解析分析的工具，但是其思想方法仍然具有典型的还原论和分析主义的特征。“任意”的函数通过一定的分解，都能够表示为正弦函数的线性组合的形式。

傅里叶变换在物理学、数论、组合数学、信号处理、概率、统计、密码学、声学、光学等领域都有着广泛的应用。

24.6 总结

通过本章节，一定要搞明白傅里叶变换，傅里叶级数，离散时间傅里叶变换和离散傅里叶变换直接的关系，特别是 24.4.2 小节的知识点。



第25章 DSP 变换运算-快速傅里叶变换原理

(FFT)

在数字信号处理中常常需要用到离散傅立叶变换(DFT)，以获取信号的频域特征。尽管传统的 DFT 算法能够获取信号频域特征，但是算法计算量大，耗时长，不利于计算机实时对信号进行处理。因此导致 DFT 被发现以来，在很长的一段时间内都不能被应用到实际工程项目中，直到一种快速的离散傅立叶计算方法——FFT 被发现，离散傅立叶变换才在实际的工程中得到广泛应用。需要强调的是，FFT 并不是一种新的频域特征获取方式，而是 DFT 的一种快速实现算法。

特别声明：FFT 原理的讲解来自网络和书籍。

25.1 FFT 由来

25.3 直接计算 DFT 的问题及改进路径

25.3 改善 DFT 运算效率的基本途径

25.4 按时间抽选的基 2-FFT 算法

25.5 按频率抽选的基 2-FFT 算法

25.6 总结

25.1 初学者重要提示

- ◆ 为大家推荐西电的国家级精品课程信号与系统，含课堂视频，书籍和课堂 PPT
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=94886>。
- ◆ 非常好的 DSP 基础知识普及书籍：
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=97312>。

25.2 FFT 的由来

离散傅里叶变换 (Discrete Fourier Transform, DFT) 是数字信号处理最重要的基石之一，也是对信号进行分析处理时，最常用的工具之一。在 200 多年前，法国数学家、物理学家傅里叶提出以他的名字命名的傅里叶级数之后，用 DFT 这个工具来分析信号就已经被人们所知。

历史上最伟大的数学家之一，欧拉是第一个使用“函数”一词来描述包含各种参数的表达式，例如： $y = f(x)$ 。他是把微积分应用于物理学的先驱者之一，给出了一个用实变量函数表示傅立叶系数的方程，用三角级数来描述离散声音在媒介中传播，发现某些函数可以通过余弦函数之和来表达。但在很长时间



内，这种分析方法并没有引起更多的重视，最主要的原因在于这种方法运算量比较大。直到 1965 年，Cooley 和 Tukey 在《计算机科学》发表著名的《机器计算傅立叶级数的一种算法》论文，FFT 才开始大规模应用。

那个年代，有个肯尼迪总统科学咨询委员会，其中有项研究主题是对苏联核测试进行检测，Tukey 就是其中一员。美国/苏联核测试提案的批准，主要取决于不实地访问核测试设施而做出检测方法。其中一个想法是，分析离海岸的地震情况，这种计算需要快速算法来计算 DFT。其它应用是国家安全，如用声学探测远距离的核潜艇。所以在军事上，迫切需要一种快速的傅立叶变换算法，这也促进了 FFT 的正式提出。

FFT 充分利用了 DFT 运算中的对称性和周期性，从而将 DFT 运算量从 N^2 减少到 $\log_2 N$ 。当 N 比较小，FFT 优势并不明显。但当 N 大于 32 开始，点数越大，FFT 对运算量的改善越明显。比如当 N 为 1024 时，FFT 的运算效率比 DFT 提高了 100 倍。在库利和图基提出的 FFT 算法中，其基本原理是先将一个 N 点时域序列的 DFT 分解为 N 个 1 点序列的 DFT，然后将这样计算出来的 N 个 1 点序列 DFT 的结果进行组合，得到最初的 N 点时域序列的 DFT 值。实际上，这种基本的思想很早就由德国伟大的数学家高斯提出过，在某种情况下，天文学计算（也是现在 FFT 应用的领域之一）与等距观察的有限集中的行星轨道的内插值有关。由于当时计算都是靠手工，所以产生一种快速算法的迫切需要。而且，更少的计算量同时也代表着错误的机会更少，正确性更高。高斯发现，一个富氏级数有宽度 $N=N1*N2$ ，可以分成几个部分。计算 $N2$ 子样本 DFT 的 $N1$ 长度和 $N1$ 子样本 DFT 的 $N2$ 长度。只是由于当时尚欠东风——计算机还没发明。在 20 世纪 60 年代，伴随着计算机的发展和成熟，库利和图基的成果掀起了数字信号处理的革命，因而 FFT 发明者的桂冠才落在他们头上。

之后，桑德 (G.Sand) -图基等快速算法相继出现，几经改进，很快形成了一套高效运算方法，这就是现在的快速傅立叶变换 (FFT)。这种算法使 DFT 的运算效率提高 1 到 2 个数量级，为数字信号处理技术应用于各种信号的实时处理创造了良好的条件，大大推进了数学信号处理技术。1984 年，法国的杜哈梅 (P.Dohamel) 和霍尔曼 (H.Hollamann) 提出的分裂基块快速算法，使运算效率进一步提高。

库利和图基的 FFT 算法的最基本运算为蝶形运算，每个蝶形运算包括两个输入点，因而也称为基-2 算法。在这之后，又有一些新的算法，进一步提高了 FFT 的运算效率，比如基-4 算法，分裂基算法等。这些新算法对 FFT 运算效率的提高一般在 50% 以内，远远不如 FFT 对 DFT 运算的提高幅度。从这个意义上说，FFT 算法是里程碑式的。可以说，正是计算机技术的发展和 FFT 的出现，才使得数字信号处理迎来了一个崭新的时代。除了运算效率的大幅度提高外，FFT 还大大降低了 DFT 运算带来的累计量化误差，这点常为人们所忽略。

25.3 直接计算 DFT 的问题及改进路径

25.3.1 问题的提出

设有限长序列 $x(n)$ ，非零值长度为 N，若对 $x(n)$ 进行一次 DFT 运行，共需要多大的运算工作量。



25.3.2 DFT 的运算量

DFT 和 IDFT 的变换式：

$$X(k) = \text{DFT}[x(n)] = \sum_{n=0}^{N-1} x(n) W_N^{nk} \quad 0 \leq k \leq N - 1$$

$$x(n) = \text{IDFT}[X(k)] = \frac{1}{N} \sum_{n=0}^{N-1} X(k) W^{-nk} \quad 0 \leq k \leq N - 1$$

注意：

1. $x(n)$ 为复数， $W_N^{nk} = e^{-j\frac{2\pi}{N}nk}$ 也为复数。

2. DFT 与 IDFT 的计算量相当。

下面以 DFT 为例说明计算量：

计算机运算时（编程实现）：

$$k = 0 \quad X(0) = x(0)W_N^{00} + x(1)W_N^{10} + \dots + x(N-1)W_N^{(N-1)0}$$

$$k = 1 \quad X(1) = x(0)W_N^{01} + x(1)W_N^{11} + \dots + x(N-1)W_N^{(N-1)1}$$

$$k = 2 \quad X(2) = x(0)W_N^{02} + x(1)W_N^{12} + \dots + x(N-1)W_N^{(N-1)2}$$

.

.

$$k = N-1 \quad X(N-1) = x(0)W_N^{0(N-1)} + x(1)W_N^{1(N-1)} + \dots + x(N-1)W_N^{(N-1)(N-1)}$$

N 个点

N 次复乘，N-1 次复加

由上面的结算可得 DFT 的计算量如下：

	复数乘法	复数加法
一个 $X(k)$	N	$N-1$
N 个 $X(k)$ (N 点 DFT)	N^2	$N(N-1)$

复数乘法的计算量： $(a+jb)(c+jd)=(ab-bd)+j(bc+ad)$

	实数乘法	复数加法
一次复乘	4	2
一次复加		2
一个 $X(k)$	$4N$	$2N+2(N-1)=2(2N-1)$
N 个 $X(k)$ (N 点 DFT)	$4N^2$	

下面通过两个实例来说明计算量：

例一：计算一个 N 点 DFT，共需 N^2 次复乘。以做一次复乘 $1\mu\text{s}$ 计算，若 $N=4096$ ，所需时间为



$$(4096)^2 = 16777216 \mu s \approx 17s$$

例二：石油勘探，有 24 个通道的记录，每通道波形记录长度为 5 秒，若每秒抽样 500 点/秒。

1. 每道总抽样点数： $500 \times 5 = 2500$ 点。
2. 24 道总抽样点数： $24 \times 2500 = 6$ 万点。
3. DFT 复乘运算时间： $N^2 = (60000)^2 = 36 \times 10^8$ 次。

$$(60000)^2 = 36 \times 10^8 \mu s = 3600s$$

由于计算量大，且要求相当大的内存，难以实现实时处理，限制了 DFT 的应用，人们一直在寻求一种能提高 DFT 运算速度的方法。

FFT 便是 Cooley 和 Tukey 在 1955 年提出来的快速算法，它可以使运算速度提高几百倍，从而使数字信号处理成为一个新兴的应用学科。

25.4 改善 DFT 运算效率的基本途径

1、利用 DFT 运算的系数 W_N^{kn} 的固有对称性和周期性，改善 DFT 的运算效率。

- 1) 对称性
- 2) 周期性
- 3) 可约性

$$W_N^{kn} \text{ 的特性 } W_N^{kn} = e^{-j\frac{2\pi}{N}nk}$$

$$\text{对称性 } (W_N^{kn})^* = W_N^{-nk} = W_N^{(N-n)k} = W_N^{n(N-k)}$$
$$\downarrow \quad \downarrow$$

$$W_N^{Nk} \cdot W_N^{-nk} \quad W_N^{nN} \cdot W_N^{-nk}$$

$$\text{周期性 } W_N^{kn} = W_N^{(N+n)k} = W_N^{n(N+k)}$$

$$\text{可约性 } W_N^{kn} = W_{mN}^{mnk} \quad W_N^{kn} = W_{N/m}^{nk/m}$$

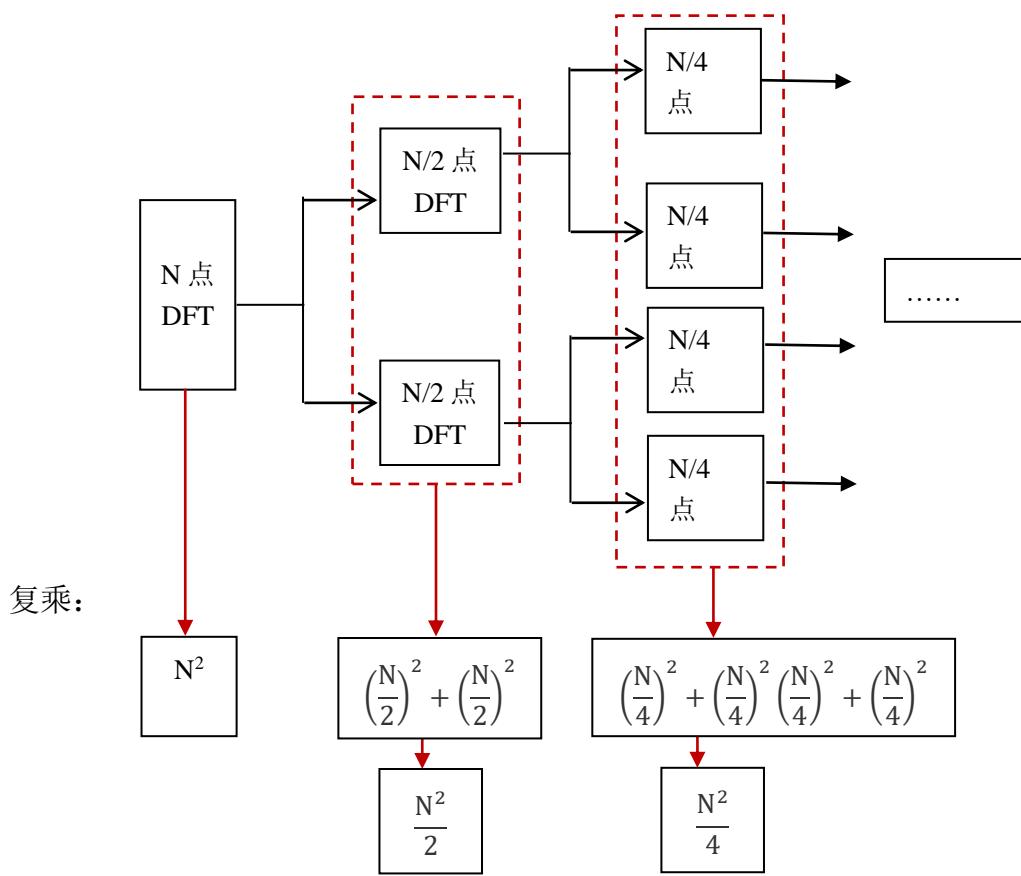
$$\downarrow$$
$$e^{-j\frac{2\pi}{mN}mnk} \quad e^{-j\frac{2\pi N}{m^2}} = e^{-j\pi} = -1$$

↑

$$\text{特殊点 } W_N^0 = 1 \quad W_N^{N/2} = -1 \quad W_N^{(k+N/2)} = -W_N^k$$

2、将长序列 DFT 利用对称性和周期性分解为短序列 DFT 的思路

因为 DFT 的运算量与 N^2 成正比，如果一个大点数 N 的 DFT 能分解为若干小点数 DFT 的组合，则显然可以达到减少运算工作量的效果。



FFT 算法的基本思想：

- 利用 DFT 系数的特性，合并 DFT 运算中的某些项。
- 把长序列 DFT → 短序列 DFT，从而减少运算量。

FFT 算法分类：

时间抽选法

DIT: Decimation-In-Time

频率抽选法

DIF: Decimation-In-Frequency

25.5 按时间抽选的基 2-FFT 算法

25.5.1 算法原理

设输入序列长度为 $N = 2^M$ (M 为正整数)，将该序列按时间顺序的奇偶分解为越来越短的子序列，称为基 2 按时间抽取的 FFT 算法。也称为 Coolkey-Tukey 算法。

其中基 2 表示： $N = 2^M$ ， M 为整数。若不满足这个条件，可以人为地加上若干零值（加零补长）使



其达到 $N = 2^M$ 。

25.5.2 算法步骤

➤ 分组，变量置换

$$X(k) = DFT[x(n)] = \sum_{n=0}^{N-1} x(n)W_N^{nk} \quad 0 \leq k \leq N-1$$

先将 $x(n)$ 按 n 的奇偶分为两组，作变量置换：

当 $n =$ 偶数时，令 $n = 2r$ ；

当 $n =$ 奇数时，令 $n = 2r+1$ ；

得到：

$$x(2r) = x_1(r)$$

$$x(2r+1) = x_2(r) \quad r = 0, \dots, \frac{N}{2}-1$$

➤ 分组，变量置换

$$X(k) = DFT[x(n)]$$

$$\begin{aligned} &= \sum_{n=0}^{N-1} x(n)W_N^{nk} \\ &= \sum_{\substack{n=0 \\ n \text{ 为偶数}}}^{N-1} x(n)W_N^{nk} + \sum_{\substack{n=0 \\ n \text{ 为奇数}}}^{N-1} x(n)W_N^{nk} \\ &= \sum_{r=0}^{\frac{N}{2}-1} x(2r)W_N^{2rk} + \sum_{r=0}^{\frac{N}{2}-1} x(2r+1)W_N^{(2r+1)k} \end{aligned}$$

$$= \sum_{r=0}^{\frac{N}{2}-1} x_1(2r)W_N^{2rk} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} x_2(2r+1)W_N^{(2r+1)k}$$

$$\text{由于 } W_N^{nk} = e^{-j\frac{2\pi}{N}nk} = e^{-j\frac{2\pi}{N/2}nk} = W_{N/2}^n$$

$$\begin{aligned} X(k) &= \sum_{r=0}^{\frac{N}{2}-1} x_1(2r)W_N^{2rk} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} x_2(2r+1)W_N^{(2r+1)k} \\ &= \sum_{r=0}^{\frac{N}{2}-1} x_1(2r)W_N^{2rk} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} x_2(2r+1)W_{N/2}^{rk} \\ &= X_1(k) + W_N^k X_2(k) \end{aligned}$$

其中 $k = 0, 1, \dots, \frac{N}{2}-1$ 。 $X_1(k)$ 和 $X_2(k)$ 只有 $\frac{N}{2}$ 个点，以 $\frac{N}{2}$ 为周期；而 $X(k)$ 却有 N 个点，以 N

为周期。要用 $X_1(k)$ 和 $X_2(k)$ 表达全部的 $X(k)$ 值。还必须利用 W_N 系数的周期特性。

$$\text{因为: } W_{N/2}^{r(\frac{N}{2}+k)} = W_{N/2}^{rk}$$

$$\text{所以: } X_1\left(\frac{N}{2} + k\right) = \sum_{r=0}^{N/2-1} x_1(2r) W_{\frac{N}{2}}^{r(\frac{N}{2}+K)} = \sum_{r=0}^{N/2-1} x_1(2r+1) W_{N/2}^{rk}$$

又考虑到 W_N^k 的对称性:

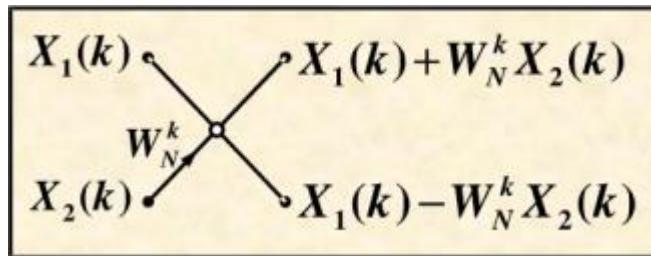
$$W_N^{(\frac{N}{2}+k)} = W_N^{(\frac{N}{2})} W_N^k = -W_N^k, \text{ 有:}$$

$$X(k) = X_1(k) + W_N^k X_2(k) \quad k = 0, 1, \dots, N/2 - 1$$

$$X\left(\frac{N}{2} + k\right) = X_1\left(\frac{N}{2} + k\right) + W_N^{(\frac{N}{2}+K)} X_2\left(\frac{N}{2} + k\right)$$

$$= X_1(k) - W_N^k X_2(k) \quad k = 0, 1, \dots, N/2 - 1$$

有了上面的计算结果后，我们可以得到如下的蝶形运算流图符号:



关于这个蝶形运算流图符号说明如下:

1. 1 个蝶形运算需要 1 次复乘, 2 次复加。
2. 左边两路为输入。
3. 右边两路为输出。
4. 中间以一个小圆表示加减运算 (右上路为相加输出, 右下路为相减输出)。

➤ 分解后的运算量

	复数乘法	复数加法
一个 N 点 DFT	N^2	$N(N-1)$
一个 $N/2$ 点 DFT	N^2	$N/2(N/2-1)$
两个 $N/2$ 点 DFT	$N^2/2$	$N(N/2-1)$
一个蝶形	1	2
$N/2$ 个蝶形	$N/2$	N
总计	$N^2 + N/2 \approx N^2/2$	$N(N/2-1)+N \approx N^2/2$

运算量减少了近一半。

例子: 求 $N=2^3=8$ 点 FFT 变化。按 $N=8 \rightarrow N/2=4$, 做 4 点的 DFT, 先将 $N=8$ 点的 DFT 分解成 2 个 4

点的 DFT:

可知: 时域上 $x(0), x(2), x(4), x(6)$ 为偶子序列。

$x(1), x(3), x(5), x(7)$ 为奇子序列。

频域上 $X(0)$ 到 $X(3)$ 由 $X(k)$ 给出

$X(4)$ 到 $X(7)$ 由 $X(k+N/2)$ 给出

$N=8$ 点的直接 DFT 的计算量为:

复乘: N^2 次 = 64 次

复加: $N(N-1)$ 次 = $8 \times 7 = 56$ 次

$$X(k) = X_1(k) + W_N^k X_2(k)$$

$$X(k+N/2) = X_1(k) - W_N^k X_2(k) \quad k=0, \dots, N/2-1$$

得到 $X_1(k)$ 和 $X_2(k)$ 需要:

复乘: $(N/2)^2 + (N/2)^2$ 次 = 32 次

复加: $N/2(N/2-1) + N/2(N/2-1) = 12 + 12 = 24$ 次

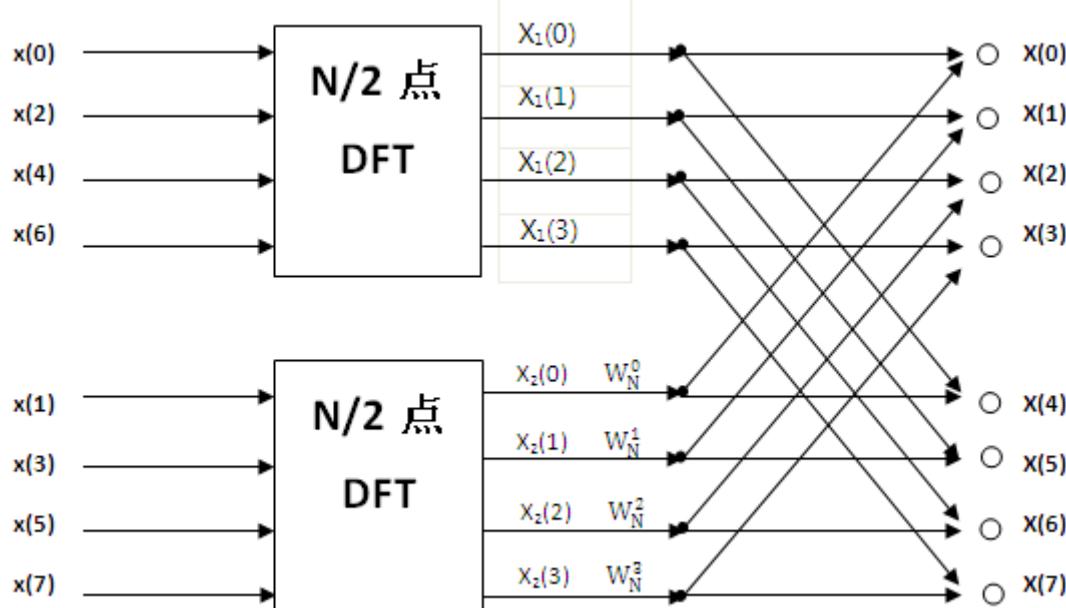
此外, 还有 4 个蝶形结, 每个蝶形结需要 1 次复乘, 2 次复加。一共是: 复乘 4 次, 复加 8 次。

用分解的方法得到 $X(k)$ 需要:

复乘: $32 + 4 = 36$ 次

复加: $24 + 8 = 32$ 次

$N = 2^3 = 8$ 按时间抽取的 DFT 分解过程:



因为 4 点 DFT 还是比较麻烦, 所以再继续分解。

若将 $N/2$ (4 点)子序列按奇/偶分解成两个 $N/4$ 点(2 点)子序列。即对将 $x_1(r)$ 和 $x_2(r)$ 分解成奇、偶两



个 $N/4$ 点(2 点)的子序列。

$$x_1(r): \begin{cases} x(0), x(4) \text{ 偶序列} \\ x(2), x(6) \text{ 奇序列} \end{cases} \quad \text{同理: } x_2(r): \begin{cases} x(1), x(5) \text{ 偶序列} \\ x(3), x(7) \text{ 奇序列} \end{cases}$$

$$\text{设: } \begin{cases} x_1(2l) = x_3(l) & \text{偶序列} \\ x_1(2l+1) = x_4(l) & \text{奇序列} \end{cases} \quad (l=0 \dots \frac{N}{4}-1) \quad \text{此处, } l=0,1$$

$$\text{设: } \begin{cases} x_2(2l) = x_5(l) & \text{偶序列} \\ x_2(2l+1) = x_6(l) & \text{奇序列} \end{cases} \quad (l=0 \dots \frac{N}{4}-1) \quad \text{此处, } l=0,1$$

那么, $X_1(k)$ 又可表示为

$$X_1(k) = \sum_{l=0}^{\frac{N}{4}-1} x_1(2l) W_{N/2}^{2lk} + \sum_{l=0}^{\frac{N}{4}-1} x_1(2l+1) W_{N/2}^{(2l+1)k}$$

$$= \sum_{l=0}^{\frac{N}{4}-1} x_3(l) W_{N/4}^{lk} + W_{N/2}^k \sum_{l=0}^{\frac{N}{4}-1} x_4(l) W_{N/4}^{lk}$$

$$= X_3(k) + W_{N/2}^k X_4(k)$$

$$\left. \begin{array}{l} X_1(k) = X_3(k) + W_{N/2}^k X_4(k) \\ X_1(k + \frac{N}{4}) = X_3(k) - W_{N/2}^k X_4(k) \end{array} \right\} k=0,1,\dots,\frac{N}{4}-1$$

$X_2(k)$ 因为可以进行相同的分解:

$$X_2(k) = \sum_{l=0}^{\frac{N}{4}-1} x_2(2l) W_{N/2}^{2lk} + \sum_{l=0}^{\frac{N}{4}-1} x_2(2l+1) W_{N/2}^{(2l+1)k}$$

$$= \sum_{l=0}^{\frac{N}{4}-1} x_5(l) W_{N/4}^{lk} + W_{N/2}^k \sum_{l=0}^{\frac{N}{4}-1} x_6(l) W_{N/4}^{lk}$$

$$= X_5(k) + W_{N/2}^k X_6(k)$$

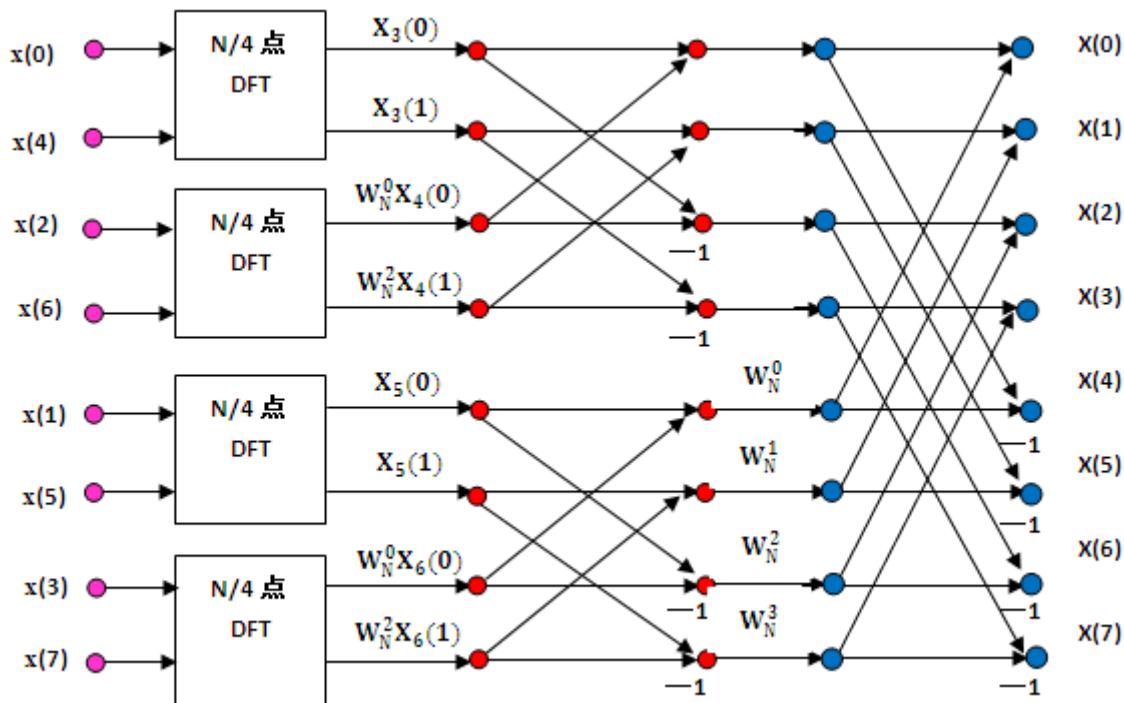
$$\left. \begin{array}{l} X_2(k) = X_5(k) + W_{N/2}^k X_6(k) \\ X_2(k + \frac{N}{4}) = X_5(k) - W_{N/2}^k X_6(k) \end{array} \right\} k=0,1,\dots,\frac{N}{4}-1$$

注意: 通常我们会把 $W_{N/2}^k$ 写成 W_N^{2k} 。

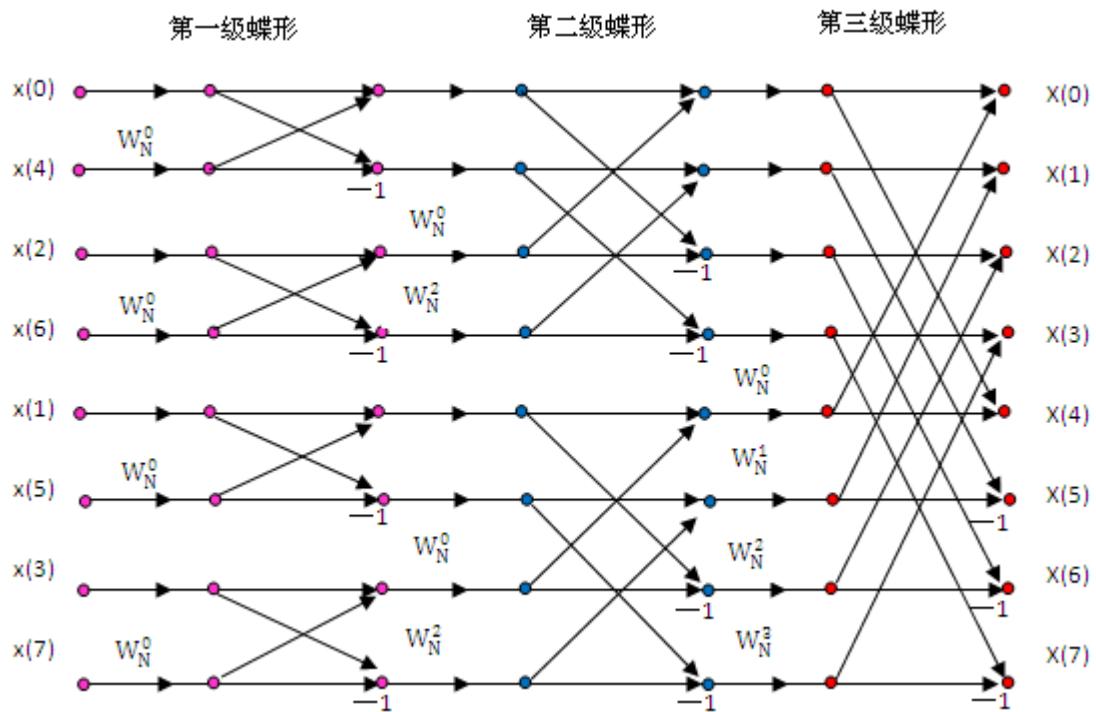
因此可以对两个 $N/2$ 点的 DFT 再分别作进一步的分解。将一个 8 点的 DFT 可以分解成四个 2 点的 DFT, 直到最后得到两两点的 DFT 为止。

由于这种方法每一步分解都是按输入序列是属于偶数还是奇数来抽取的, 所以称为“按时间抽取的 FFT 算法”。

下图是由 4 个两点 DFT 组成的 8 点 DFT:



下图是按 8 点抽取的 FFT 运算流图：



这里注意观察蝶形图的系数 W_N^{nk}

观察 $N=2^3=8$ 点 FFT 的蝶形系数 W_N^{nk} :

第一级：有一种类型的蝶形运算系数 W_8^0

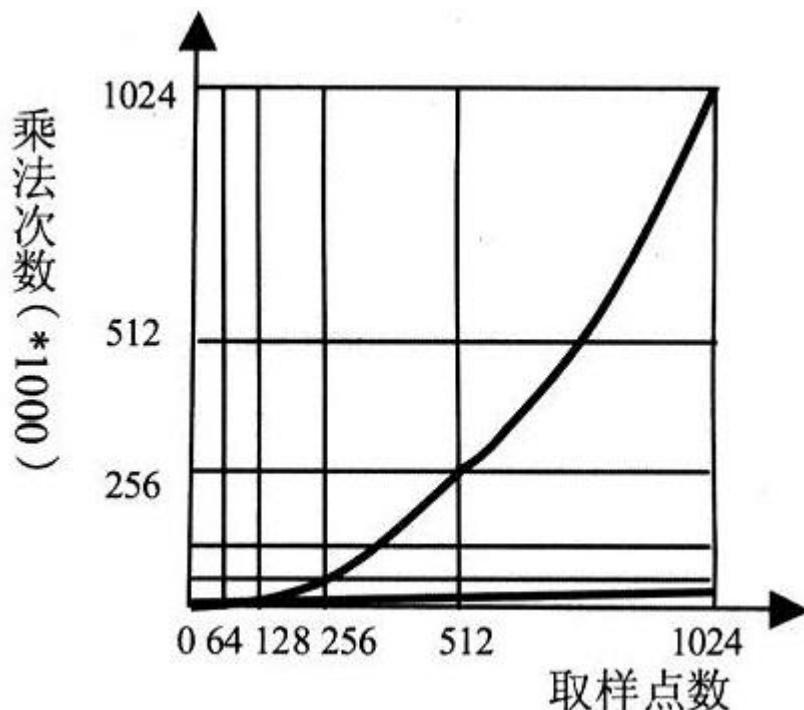
第二级：有两种类型的蝶形运算系数 W_8^0, W_8^2

第三级：有四种类型的蝶形运算系数 $W_8^0, W_8^1, W_8^2, W_8^3$

第 L 级：有 2^{L-1} 种蝶形运算系数 $W_N^0, W_N^1, W_N^2, \dots, W_N^{N/2-1}$

25.5.3 FFT 算法和直接计算 DFT 运算量的比较

FFT 算法与直接计算 DFT 所需乘法次数的比较曲线



25.6 按频率抽选的基 2-FFT 算法

在基 2 快速算法中，频域抽取法 FFT 也是一种常用的快速算法，简称 DIF-FFT。

鉴于网上和课本中关于 FFT 原理已经讲解非常详细了，在这里就不再赘述了。有兴趣的查阅相关书籍进行学习即可。

25.7 总结

本章节主要讲解了 FFT 的基 2 算法实现原理，讲解稍显枯燥，不过还是希望初学的同学认真学习，搞懂一种快速傅里叶算法的实现即可。



第26章 FFT 变换结果的物理意义

FFT 是离散傅立叶变换的快速算法，可以将一个信号变换到频域。有些信号在时域上是很难看出什么特征的，但是如果变换到频域之后，就很容易看出特征了。这就是很多信号分析采用 FFT 变换的原因。另外，FFT 可以将一个信号的频谱提取出来，这在频谱分析方面也是经常用的。本章节的主要内容是讲解 FFT 变换结果的物理意义。

特别声明：部分知识整理自网络。

- 26.1 FFT 变换结果的物理意义
- 26.2 FFT 变换的频谱泄露问题
- 26.4 总结

26.1 初学者重要提示

- ◆ 本章为大家介绍 FFT 结果的物理意义，如果之前没有了解过，有必要了解下。
- ◆ 下个章节为大家介绍两个重要知识点：频谱泄露和栅栏效应，推荐学习完毕本章后看一下。

26.2 FFT 变换结果的物理意义

26.2.1 理论阐释

虽然很多人都知道 FFT 是什么，可以用来做什么，怎么去做，但是却不知道 FFT 之后的结果是什么意思、如何决定要使用多少点来做 FFT。

一个模拟信号，经过 ADC 采样之后，就变成了数字信号。采样定理告诉我们，采样频率要大于信号频率的两倍（要满足奈奎斯特采样定律）。

采样得到的数字信号，就可以做 FFT 变换了。N 个采样点，经过 FFT 之后，就可以得到 N 个点的 FFT 结果。为了方便进行 FFT 运算，通常 N 取 2 的整数次方。

假设采样频率为 F_s ，信号频率 F ，采样点数为 N。那么 FFT 之后结果就是一个为 N 点的复数。每一个点就对应着一个频率点。这个点的模值，就是该频率值下的幅度特性。具体跟原始信号的幅度有什么关系呢？假设原始信号的峰值为 A，那么 FFT 的结果的每个点（除了第一个点直流分量之外）的模值就是 A 的 $N/2$ 倍。而第一个点就是直流分量，它的模值就是直流分量的 N 倍。而每个点的相位呢，就是在该频率下的信号的相位。第一个点表示直流分量（即 0Hz），而最后一个点 N 的再下一个点（实际上这个点是不存在的，这里是假设的第 $N+1$ 个点，可以看做是将第一个点分做两半分，另一半移到最后）



则表示采样频率 F_s , 这中间被 $N-1$ 个点平均分成 N 等份, 每个点的频率依次增加。例如某点 n 所表示的频率为:

$$F_n = \frac{(n-1)xF_s}{N}$$

由上面的公式可以看出, F_n 所能分辨到频率为 F_s/N , 如果采样频率 F_s 为 1024Hz, 采样点数为 1024 点, 则可以分辨到 1Hz。1024Hz 的采样率采样 1024 点, 刚好是 1 秒, 也就是说, 采样 1 秒时间的信号并做 FFT, 则结果可以分析到 1Hz, 如果采样 2 秒时间的信号并做 FFT, 则结果可以分析到 0.5Hz。如果要提高频率分辨率, 则必须增加采样点数, 也即采样时间。频率分辨率和采样时间是倒数关系。假设 FFT 之后某点 n 用复数 $a+bi$ 表示, 那么这个复数的模就是:

$$A_n = \sqrt{a^2 + b^2}$$

相位就是:

$$\phi_n = \text{atan2}(b, a)$$

根据以上的结果, 就可以计算出 n 点 ($n \neq 1$, 且 $n < N/2$) 对应的信号的表达式为:

$$\frac{A_n}{N/2} \times \cos(2\pi \times F_n \times t + \phi_n) \text{ 即 } \frac{A_n \times 2}{N} \times \cos(2\pi \times F_n \times t + \phi_n)$$

对于 $n=1$ 点的信号, 是直流分量, 幅度即为 A_1/N 。由于 FFT 结果的对称性, 通常我们只使用前半部分的结果, 即小于采样频率一半的结果。

26.2.2 理论计算和 Matlab 实际计算结果对比

下面以一个实际的信号来做说明:

假设我们有一个信号, 它含有 2V 的直流分量, 频率为 50Hz、相位为 -30 度、幅度为 3V 的交流信号, 以及一个频率为 75Hz、相位为 90 度、幅度为 1.5V 的交流信号。用数学表达式就是如下:

$$x = 2 + 3 \cdot \cos(2\pi \cdot 50 \cdot t - \pi \cdot 30/180) + 1.5 \cdot \cos(2\pi \cdot 75 \cdot t + \pi \cdot 90/180)$$

式中 cos 参数为弧度, 所以 -30 度和 90 度要分别换算成弧度。我们以 256Hz 的采样率对这个信号进行采样, 总共采样 256 点。按照我们上面的分析, $F_n = (n-1) \cdot F_s / N$, 我们可以知道, 每两个点之间的间距就是 1Hz, 第 n 个点的频率就是 $n-1$ 。我们的信号有 3 个频率: 0Hz、50Hz、75Hz, 应该分别在第 1 个点、第 51 个点、第 76 个点上出现峰值, 其它各点应该接近 0。实际情况如何如下:

◆ 第一步: 在 matlab 上新建.m 文件, 文件内容如下:

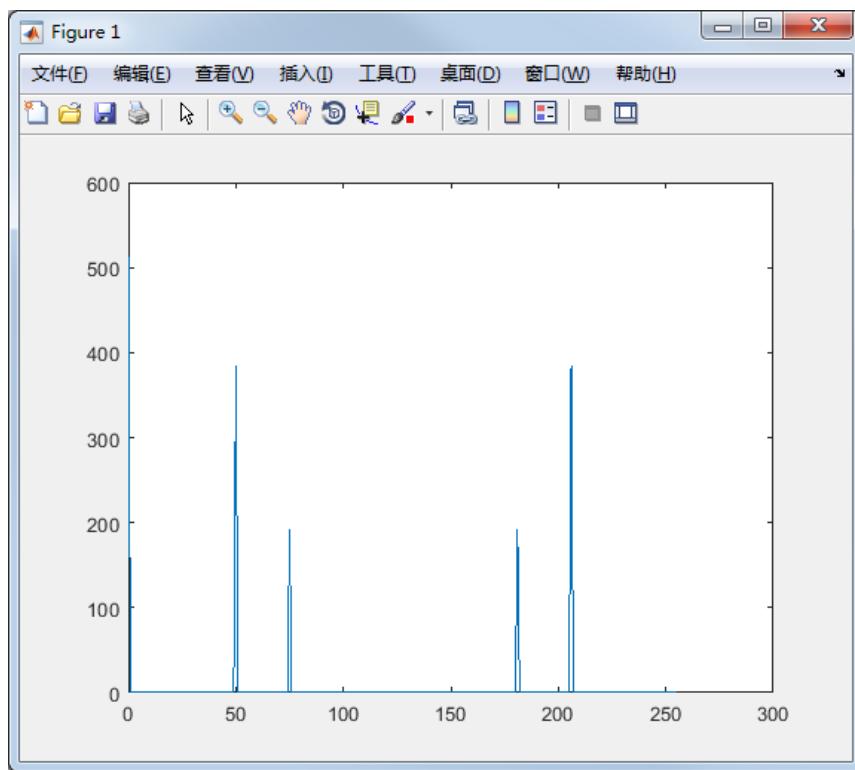
```
Fs = 256; % 采样率  
N = 256; % 采样点数  
n = 0:N-1; % 采样序列  
t = 0:1/Fs:1-1/Fs; % 时间序列
```

```
x = 2 + 3 * cos(2 * pi * 50 * t - pi * 30 / 180) + 1.5 * cos(2 * pi * 75 * t + pi * 90 / 180); % 原始信号
```

```
y = fft(x); % 对原始信号做 FFT 变换
```

```
M = abs(y); %求 FFT 转换结果的模值  
plot(n, M); %绘制 FFT 转换模值的曲线
```

◆ 第二步：运行后显示效果如下：



◆ 第三步：从 matlab 的工作区获得几个关键点及其附近两个点的幅值：

工作区	
名称	值
Fs	256
M	1x256 double
n	1x256 double
N	256
t	1x256 double
x	1x256 double
y	1x256 complex do...

1 点，2 点，3 点的数值如下：

1	2	3
512.0000	1.4516e-13	1.4353e-13

50 点，51 点，52 点的数值如下：

50	51	52
2.2573e-12	384	2.2587e-12



75 点, 76 点, 77 点的数值如下:

75	76	77
1.0385e-12	192.0000	7.5670e-13

按照上面说的公式, 可以计算出:

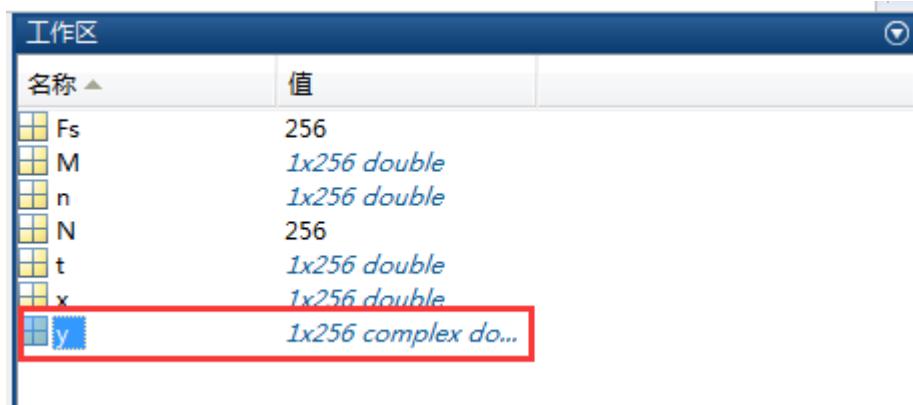
直流分量为: $512/N=512/256=2$;

50Hz 信号的幅度为: $384/(N/2)=384/(256/2)=3$;

75Hz 信号的幅度为: $192/(N/2)=192/(256/2)=1.5$ 。可见, 从频谱分析出来的幅度是正确的。

◆ 第四步：计算相位

计算相位要获取 FFT 变换后相应频率点幅值的实部和虚部, 这里看第一步代码中的 y 变量数值即可。



由于直流信号没有相位可言。这里主要看 50Hz 的相位和 75Hz 的相位。

1. 计算 50Hz 信号的相位。

y 变量的第 51 点对应数值: $332.553755053225 - 192.0000000000000i$

那么 $\text{atan2}(-192, 332.553)=-0.5236$, 这个结果是弧度, 换算成角度 $180*(-0.5236)/\pi=-30.0001$ 。

这个结果与 $\cos(2*\pi*50*t-\pi*30/180)$ 中相位是相符的。

2. 计算 75Hz 信号的相位。

y 变量的第 76 点对应数值: $3.43858275186904e-12 + 192.0000000000000i$

那么 $\text{atan2}(192, 3.43858275186904e-12)=1.5708$ 弧度, 换算成角度 $180*1.5708/\pi=90.0002$ 。

这个结果与 $\cos(2*\pi*75*t+\pi*90/180)$ 中的相位是相符的。

➤ 总结

根据 FFT 结果以及上面的分析计算, 我们就可以写出信号的表达式了, 它就是我们开始提供的信号。

总的来说, 这个过程就是这样: 假设采样频率为 F_s , 采样点数为 N , 做 FFT 之后, 某一点 n (n 从 1 开始) 表示的频率为: $F_n=(n-1)*F_s/N$; 该点的模值除以 $N/2$ 就是对应该频率下的信号的幅度 (对于直流信号是除以 N) ; 该点的相位即是对应该频率下的信号的相位。相位的计算可用函数 $\text{atan2}(b,a)$ 计算。 $\text{atan2}(b,a)$ 是求坐标为 (a,b) 点的角度值, 范围从 $-\pi$ 到 π 。要精确到 x Hz, 则需要采样长度为 $1/x$ 秒的信号, 并做 FFT。要提高频率分辨率, 就需要增加采样点数, 这在一些实际的应用中是不现实的, 需要在较短的时间内完成分析。解决这个问题的方法有频率细分法, 比较简单的方法是采样比较短时间的信

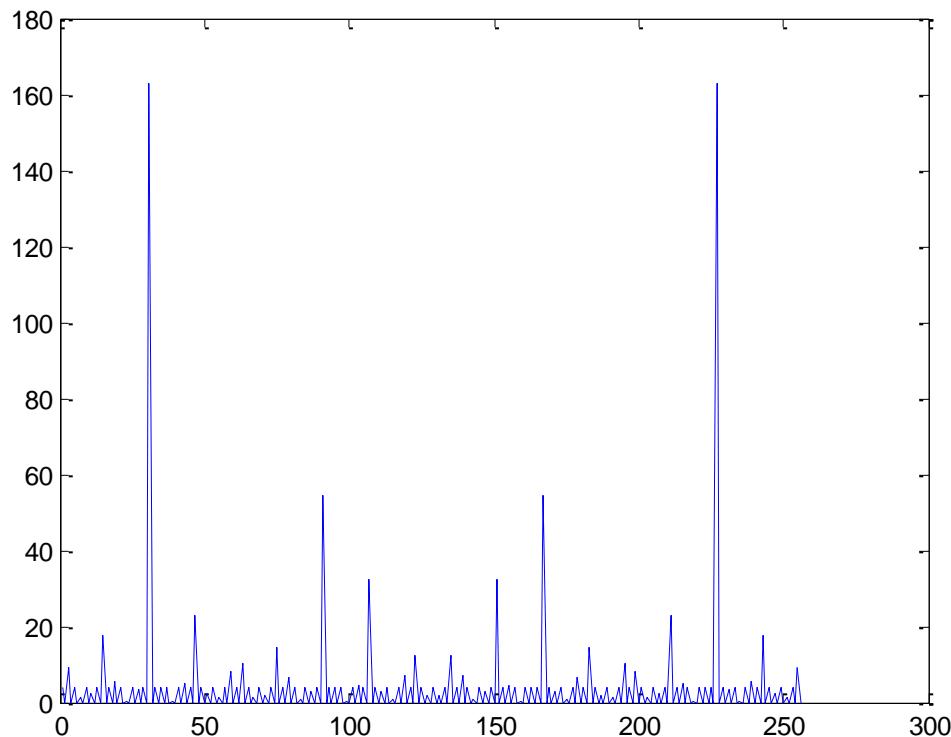
号，然后在后面补充一定数量的 0，使其长度达到需要的点数，再做 FFT，这在一定程度上能够提高频率分辨率。具体的频率细分法大家可参考相关文献。

26.3 FFT 变换的频谱泄露问题

为了说明频谱泄露的问题，这里我们做一个求解方波 FFT 变换的例子。在 matlab 中运行如下代码：

```
Fs = 256; % 采样率  
N = 256; % 采样点数  
n = 0:N-1; % 采样序列  
t = 0:1/Fs:1-1/Fs; % 时间序列  
x = square(2*pi*30*t, 50); % 原始信号  
y = fft(x); % 对原始信号做 FFT 变换  
M = abs(y); % 求 FFT 转换结果的模值  
plot(n, M); % 绘制 FFT 转换模值的曲线
```

运行代码，输出结果如下：



与方波的理论计算值相比，上面的幅频响应图中出现了很多小毛刺，其实这个就是频谱泄露的结果导致的。

下面就说说什么是频谱泄露：



对于频率为 f_s 的正弦序列，它的频谱应该只是在 f_s 处有离散谱。但是，在利用 DFT 求它的频谱做了截短，结果使信号的频谱不只是在 f_s 处有离散谱，而是在以 f_s 为中心的频带范围内都有谱线出现，它们可以理解为是从 f_s 频率上“泄露”出去的，这种现象称为频谱“泄露”（结合上面的例子就更形象了）。

在实际问题中遇到的离散时间序列 $x(n)$ 通常是无限长序列，因而处理这个序列的时候需要将它截断。截断相当于将序列乘以窗函数 $w(n)$ 。根据频域卷积定理，时域中 $x(n)$ 和 $w(n)$ 相乘对应于频域中它们的离散傅立叶变换 $X(jw)$ 和 $W(jw)$ 的卷积。因此， $x(n)$ 截短后的频谱不同于它以前的频谱。

为了减小频谱“泄露”的影响，往往在 FFT 处理中采用加窗技术，典型的加窗序列有 Hamming、Blackman、Gaussian 等窗序列。此外，增加窗序列的长度也可以减少频谱“泄露”。

时域上乘上窗函数，相当于频域进行卷积。长度为无穷长的常数窗函数，频域为 delta 函数，卷积后的结果和原来一样。如果是有限矩形窗，频域是 Sa 函数，旁瓣电平起伏大，和原频谱卷积完，会产生较大的失真。

窗的频谱，越像 delta 函数（主瓣越窄，旁瓣越小），频谱的还原度越高。加窗就不可避免频谱泄漏，典型的加权序列有 Hamming、Blackman、Gaussian 等窗序列主要是为了降低降低旁瓣，对于降低频谱泄漏效果远不如增加窗序列的长度明显。

周期信号加窗后做 DFT 仍然有可能引起频谱泄露，设 f_s 为采样频率， N 为采样序列长度，分析频率为： $m \cdot f_s / N$ ($m=0, 1, \dots$)，以 cos 函数为例，设其频率为 f_0 ，如果 f_0 不等于 $m \cdot f_s / N$ ，就会引起除 f_0 以外的其他 $m \cdot f_s / N$ 点为非零值，即出现了泄露。

DFT 作为有限长的运算，对于无限长的信号必须要进行一定程度的截断，既然信号已经不完整了，那么截断后的信号频谱肯定就会发生畸变，截断由窗函数来完成，实际的窗函数都存在着不同幅度的旁瓣，所以在卷积时，除了离散点的频率上有幅度分量外，在相邻的两个频率点之间也有不同程度的幅度，这些应该就是截断函数旁瓣所造成的。

26.4 总结

通过本章节的讲解，大家应该对 FFT 变换结果的物理意义应该有更深入的理解了，通过后面章节的继续会让大家有更加深入的认识。



第27章 FFT 的示波器应用

特别声明：本章节内容整理自力科示波器基础应用系列文档，原名《FFT 的前世今生》。

FFT (Fast Fourier Transform, 快速傅立叶变换) 是离散傅立叶变换的快速算法，也是我们在数字信号处理技术中经常会提到的一个概念。在大学的理工科课程中，完成高等数学的课程后，数字信号处理一般会作为通信电子类专业的专业基础课程进行学习，原因是其中涉及了大量的高等数学的理论推导，同时又是各类应用技术的理论基础。关于傅立叶变换的经典著作和文章非常多，但是看到满篇的复杂公式推导和罗列，我们还是很难从直观上去理解这一复杂的概念，对于普通的测试工程师来说，掌握 FFT 的概念首先应该搞清楚这样几个问题（在这篇文章中尝试用更加浅显的讲解，尽量不使用公式推导来说一说 FFT 的那些事儿）：

- 27.1 为什么需要 FFT
- 27.2 变换究竟是如何进行的
- 27.3 变换前后信号有何种对应关系
- 27.4 在使用测试工具（示波器或者其它软件平台）进行 FFT 的方法和需要注意的问题
- 27.5 力科示波器与泰克示波器的 FFT 计算方法的比较
- 27.6 珊栏现象
- 27.7 窗函数对于 FFT 结果的影响
- 27.8 窗函数选择指南

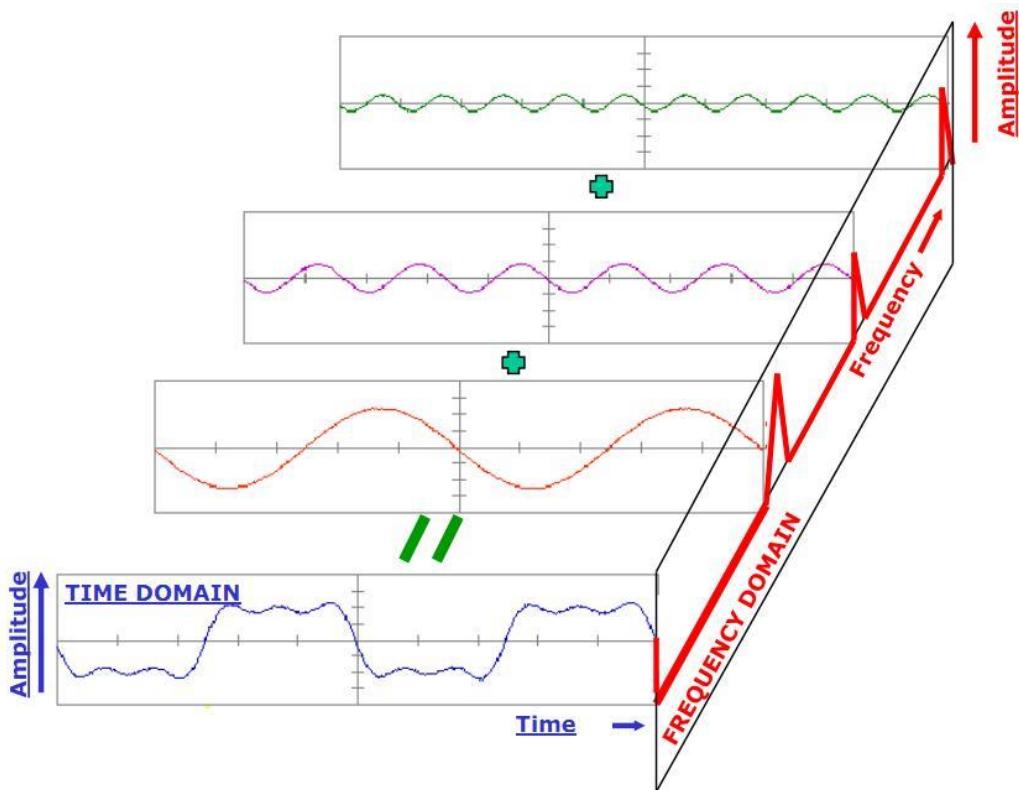
27.1 初学者重要提示

- ◆ 本章的知识点非常重要，强烈推荐初学者学习下，尤其注意两个关键知识点频谱泄露和栅栏效应。

27.2 傅里叶变换的物理意义

傅立叶原理表明：任何连续测量的时序或信号，都可以表示为不同频率的正弦波信号的无限叠加。而根据该原理创立的傅立叶变换算法利用直接测量到的原始信号，以累加的方式来计算该信号中不同正弦波信号的频率、振幅和相位。当然这是从数学的角度去看傅立叶变换。

那么从物理的角度去看待傅立叶变换，它其实是帮助我们改变传统的时间域分析信号的方法转到从频率域分析问题的思维，下面的一幅立体图形可以帮助我们更好得理解这种角度的转换：



所以，最前面的时域信号在经过傅立叶变换的分解之后，变为了不同正弦波信号的叠加，我们再去分析这些正弦波的频率，可以将一个信号变换到频域。有些信号在时域上是很难看出什么特征的，但是如果变换到频域之后，就很容易看出特征了。这就是很多信号分析采用 FFT 变换的原因。另外，FFT 可以将一个信号的频谱提取出来，这在频谱分析方面也是经常用的。

傅立叶变换提供给我们这种换一个角度看问题的工具，看问题的角度不同了，问题也许就迎刃而解！

27.3 FFT 变换是如何进行的

首先，按照被变换的输入信号类型不同，傅立叶变换可以分为 4 种类型：

- 1、非周期性连续信号傅立叶变换 (Fourier Transform)
- 2、周期性连续信号傅立叶级数(Fourier Series)
- 3、非周期性离散信号离散时域傅立叶变换 (Discrete Time Fourier Transform)
- 4、周期性离散信号离散傅立叶变换(Discrete Fourier Transform)

下面是四种原信号图例：

Type of Transform	Example Signal
Fourier Transform <i>signals that are continuous and aperiodic</i>	
Fourier Series <i>signals that are continuous and periodic</i>	
Discrete Time Fourier Transform <i>signals that are discrete and aperiodic</i>	
Discrete Fourier Transform <i>signals that are discrete and periodic</i>	

这里我们要讨论是离散信号，对于连续信号我们不作讨论，因为计算机只能处理离散的数值信号，我们的最终目的是运用计算机来处理信号的。所以对于离散信号的变换只有离散傅立叶变换（DFT）才能被适用，对于计算机来说只有离散的和有限长度的数据才能被处理，对于其它的变换类型只有在数学演算中才能用到，在计算机面前我们只能用 DFT 方法，我们要讨论的 FFT 也只不过是 DFT 的一种快速的算法。DFT 的运算过程是这样的：

$$X(k) = \text{DFT}[x(n)] = \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j\pi n t/N}$$

X(k)—频域值

X(n)—时域采样点

n—时域采样点的序列索引

k—频域值的索引

N—进行转换的采样点数量

可见，在计算机或者示波器上进行的 DFT，使用的输入值是数字示波器经过 ADC 后采集到的采样值，也就是时域的信号值，输入采样点的数量决定了转换的计算规模。变换后的频谱输出包含同样数量的采样点，但是其中有一半的值是冗余的，通常不会显示在频谱中，所以真正有用的信息是 N/2+1 个点。FFT 的过程大大简化了在计算机中进行 DFT 的过程，简单来说，如果原来计算 DFT 的复杂度是 N² 次运算（N 代表输入采样点的数量），进行 FFT 的运算复杂度是：

$$N \log_{10} N$$



因此，计算一个 1,000 采样点的 DFT，使用 FFT 算法只需要计算 3,000 次，而常规的 DFT 算法需要计算 1,000,000 次！

我们以一个 4 个点的 DFT 变换为例来简单说明 FFT 是怎样实现快速算法的：

$$X(k) = \frac{1}{4} \sum_{n=0}^3 x(n)e^{-j\pi nt/4}$$

计算得出：

$$\begin{aligned}x(0) &= x(0)e^{-j0} + x(1)e^{-j0} + x(2)e^{-j0} + x(3)e^{-j0} \\x(1) &= x(0)e^{-j0} + x(1)e^{-jx/2} + x(2)e^{-jx} + x(3)e^{-j3x/2} \\x(2) &= x(0)e^{-j0} + x(1)e^{-jx} + x(2)e^{-j2x} + x(3)e^{-j3x} \\x(3) &= x(0)e^{-j0} + x(1)e^{-j3x/2} + x(2)e^{-j3x} + x(3)e^{-j9x/2}\end{aligned}$$

其中的红色部分在 FFT 中是必须计算的分量，其它蓝色部分不需要直接计算，可以由红色的分量直接推导得到，比如：

$$x(1)e^{-j0} = -1 * x(1)e^{-j\pi}$$

$$x(2)e^{-j0} = x(2)e^{-j2\pi}$$

.....

这样，已经计算出的红色分量只需要计算机将结果保存下来用于之后计算时调用即可，因此大大减少了 DFT 的计算量。

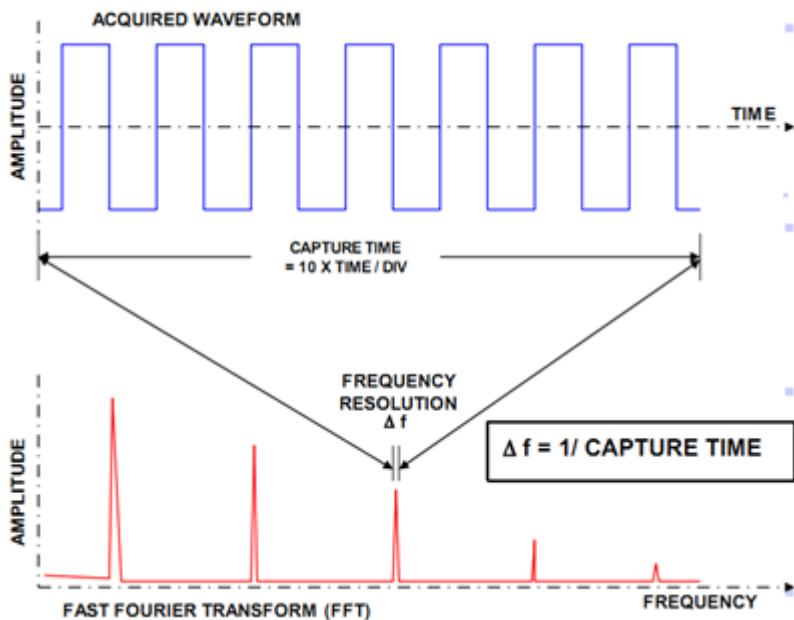
27.4 FFT 变换前后有何种对应关系

我们以一个实际的信号为例来说明：

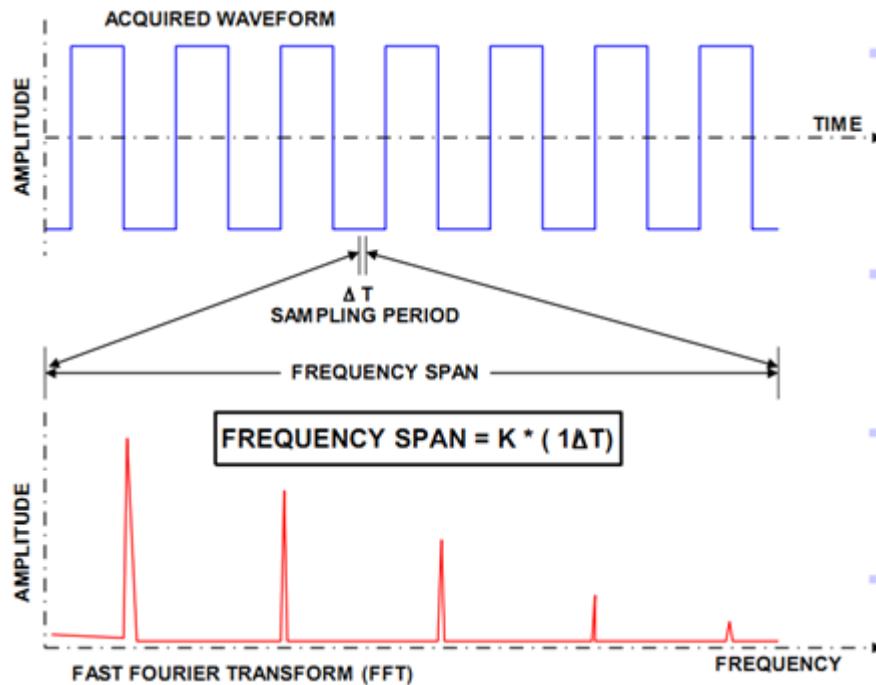
示波器采样得到的数字信号，就可以做 FFT 变换了。N 个采样点，经过 FFT 之后，就可以得到 N 个点的 FFT 结果。为了方便进行 FFT 运算，通常 N 取 2 的整数次方。假设采样频率为 F_s ，信号频率 F，采样点数为 N。那么 FFT 之后结果就是一个为 N 点的复数。每一个点就对应着一个频率点。这个点的模值，就是该频率值下的幅度特性。具体跟原始信号的幅度有什么关系呢？假设原始信号的峰值为 A，那么 FFT 的结果的每个点（除了第一个点直流分量之外）的模值就是 A 的 $N/2$ 倍。而第一个点就是直流分量，它的模值就是直流分量的 N 倍。而每个点的相位呢，就是在该频率下的信号的相位。第一个点表示直流分量（即 0Hz），而最后一个点 N 的再下一个点（实际上这个点是不存在的，这里是假设的第 $N+1$ 个点，也可以看做是将第一个点分做两半分，另一半移到最后）则表示采样频率 F_s ，这中间被 $N-1$ 个点平均分成 N 等份，每个点的频率依次增加。例如某点 n 所表示的频率为： $F_n = (n-1)*F_s/N$ 。由上面的公式可以看出， F_n 所能分辨到频率为 F_s/N ，如果采样频率 F_s 为 1024Hz，采样点数为 1024 点，则可以分辨到 1Hz。1024Hz 的采样率采样 1024 点，刚好是 1 秒，也就是说，采样 1 秒时间的信号并做 FFT，则结果可以分析精确到 1Hz，如果采样 2 秒时间的信号并做 FFT，则结果可以分析精确到

0.5Hz。如果要提高频率分辨率，则必须增加采样点数，也即采样时间。频率分辨率和采样时间是倒数关系。

下面这幅图更能够清晰地表示这种对应关系：



变换之后的频谱的宽度 (Frequency Span) 与原始信号也存在一定的对应关系。根据 Nyquist 采样定理，FFT 之后的频谱宽度 (Frequency Span) 最大只能是原始信号采样率的 1/2，如果原始信号采样率是 4GS/s，那么 FFT 之后的频宽最多只能是 2GHz。时域信号采样周期 (Sample Period) 的倒数，即采样率 (Sample Rate) 乘上一个固定的系数即是变换之后频谱的宽度，即 Frequency Span = K * $(1/\Delta T)$ ，其中 ΔT 为采样周期，K 值取决于我们在进行 FFT 之前是否对原始信号进行降采样 (抽点)，因为这样可以降低 FFT 的运算量。如下图所示：



可见，更高的频谱分辨率要求有更长的采样时间，更宽的频谱分布需要提高对于原始信号的采样率，当然我们希望频谱更宽，分辨率更精确，那么示波器的长存储就是必要的！它能提供您在高采样率下采集更长时间信号的能力！值得强调的是，力科示波器可以支持计算 128Mpts 的 FFT，而其它某品牌则只有 3.2Mpts。

27.5 使用示波器进行 FFT 的方法和需要注意的问题

我们先来看一个简单的例子：

Problem：在示波器上采集一个连续的，周期性的信号，我们希望在示波器上进行 FFT 计算之后，观察到信号中心频率（Center Frequency）在 2.48GHz，频宽（Frequency Span）为 5MHz，频谱分辨率（Bandwidth Resolution）为 10KHz 的频谱图，应该如何设置示波器的采集？

首先，根据频谱分辨率（Bandwidth Resolution）10KHz 可以推算出，至少需要采集信号的时间长度为 $1/10\text{KHz} = 100\mu\text{s}$ ，因此至少要设置示波器时基为 10us/Div；为了尽量保证 FFT 之后频谱图在各个频点的信号能量精度，测量时需要时域信号幅值占满整个栅格的 90%以上；采样率设置应至少满足 Nyquist 采样率，即至少设置 >5GS/s 采样率才能够看到中心频率在 2.48GHz 的频率谱线；选择合适的窗函数（Von Hann 汉宁窗）和频谱显示方式（power spectrum）；使用 Zoom 工具，将频谱移动到 Center 2.48GHz，Scale 500KHz/Div 位置，Zoom 设置方法如下图所示：



Zoom Tab Used
To Setup Span
And Center

在力科示波器中进行 FFT 的运算有几种不同的输出类型：

Linear Magnitude(Volts),

Phase(Degrees),

Power Spectrum(dBm),

Power Spectral Density(dBm)

这几种输出类型都是由 FFT 计算之后的结果换算而来，我们知道 FFT 计算之后的结果包含实部 (Real) 和虚部 (Imaginary) 成分，它们的单位都是 Volts。具体的换算方式如下：

$$\text{Linear Magnitude(Volts)} = \sqrt{R^2 + I^2}$$

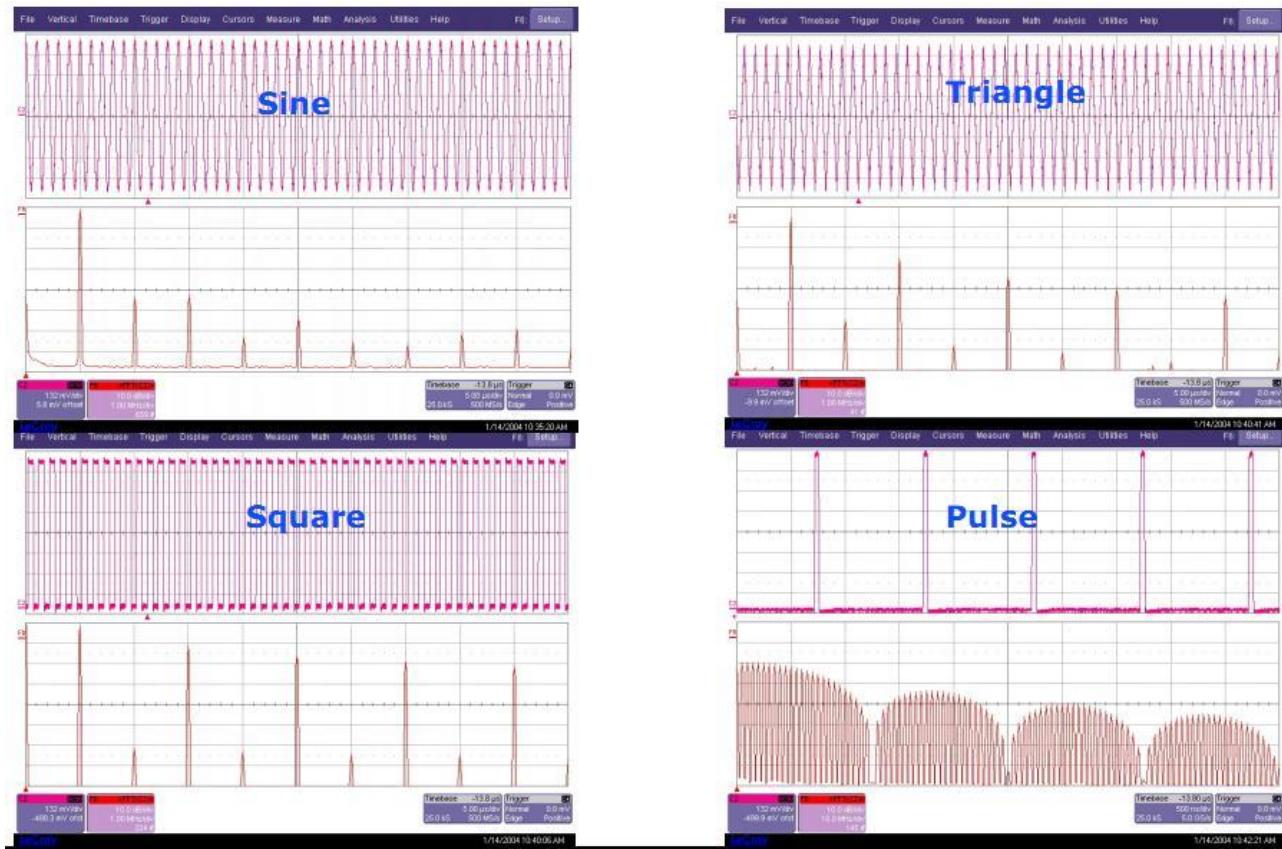
$$\text{Phase(Degrees)} = \frac{1}{\tan(I/R)}$$

$$\text{Power Spectrum(dBm)} = \log \left(\frac{I^2 + R^2}{2mW} \right)$$

Power Spectral Density(dBm) = $\log \left(\frac{I^2 + R^2}{2mW} \times \Delta f \times ENBW \right)$ ，其中 Δf 为频谱分辨率，ENBW 为与所选加权

函数（窗）相关的有效噪声带宽。

几种典型周期函数的频谱图：



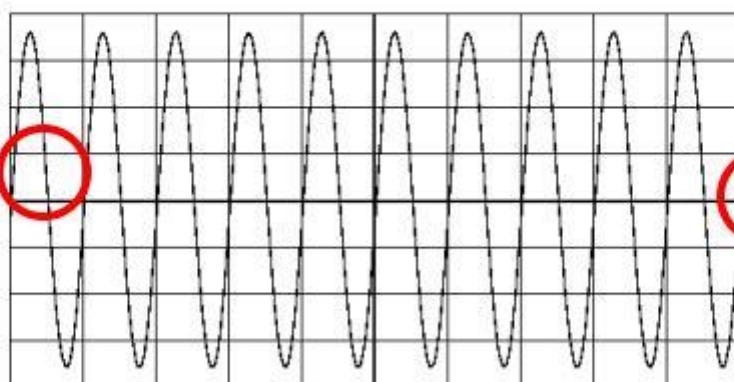
频谱泄露：

所谓频谱泄露，就是信号频谱中各谱线之间相互干扰，使测量的结果偏离实际值，同时在真实谱线的两侧的其它频率点上出现一些幅值较小的假谱。产生频谱泄露的主要原因是采样频率和原始信号频率不同步，造成周期的采样信号的相位在始端和终端不连续。简单来说就是因为计算机的 FFT 运算能力有限，只能处理有限点数的 FFT，所以在截取时域的周期信号时，没有能够截取整数倍的周期。信号分析时不可能取无限大的样本。只要有截断不同步就会有泄露。如下图所示：

Signal Frequency 2.00 MHz

22-Jun-95
16:39:58

.5 μ s
1.00 V



SETUP OF A

use Math?
No Yes

Math Type

1 Enh.Res
Extrema
FFT
FFTAVG
Functions

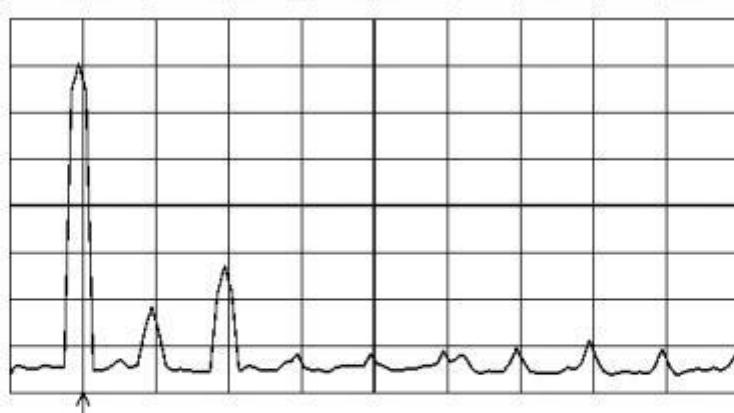
FFT result

Phase
Power Dens
Power Spect
Real
Real+Imag

with
Von Hann
(window)

of
1 2 3 4 B C D
M1 M2 M3 M4

B:PS(AVG(A))
2 MHz
10.0 dB
-42.101 dB
100 swps



B

.5 μ s

1 V 50Ω
2 .2 V 50Ω
3 .2 V 50Ω
4 50 mV AC

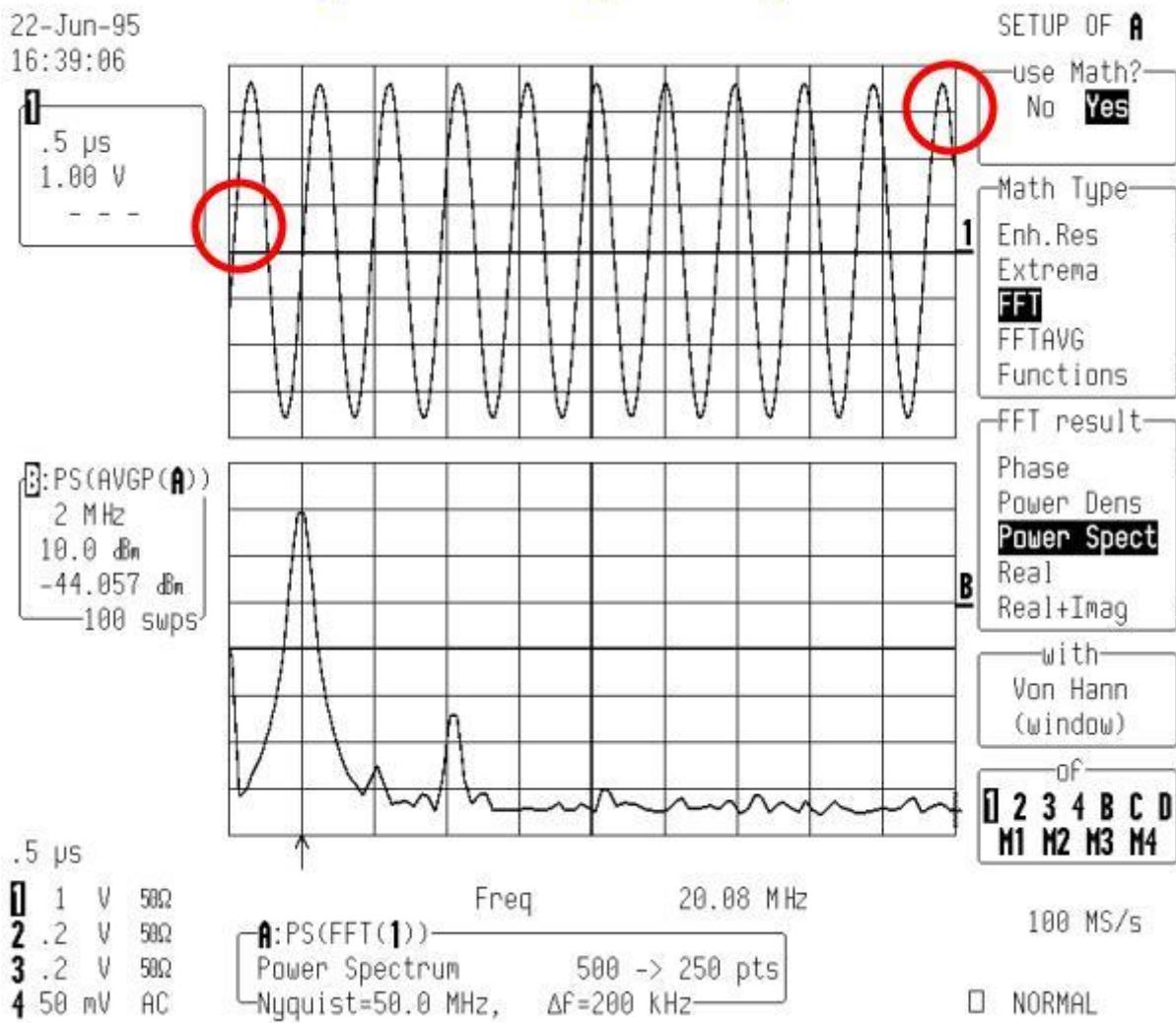
Freq 20.08 MHz
A:PS(FFT(1))
Power Spectrum 500 -> 250 pts
Nyquist=50.0 MHz, $\Delta f=200$ kHz

100 MS/s

 NORMAL

图中被测信号的开始端相位和截止端相位相同，表示在采集时间内有整数倍周期的信号被采集到，所以此时经行 FFT 运算后得出的频谱不会出现泄露。

Signal Frequency 2.1 MHz



上图的信号频率为 2.1MHz，采集时间内没有截取整数倍周期的信号，FFT 运算之后谱线的泄露现象严重，可以看到能量较低的谱线很容易被临近的能量较高的谱线的泄露给淹没住。

因此，避免频谱泄露的方法除了尽量使采集速率与信号频率同步之外，还可以采用适当的窗函数。

另外一个方法是采集信号时间足够长，基本上可以覆盖到整个有效信号的时间跨度。这种方法经常在瞬态捕捉中被使用到，比如说冲击试验，如果捕捉的时间够长，捕捉到的信号可以一直包括了振动衰减为零的时刻。在这种情况下，可以不加窗函数。窗函数其实就是一个加权函数，它在截取的信号时间段内有值，时间段之外值为 0；记为：

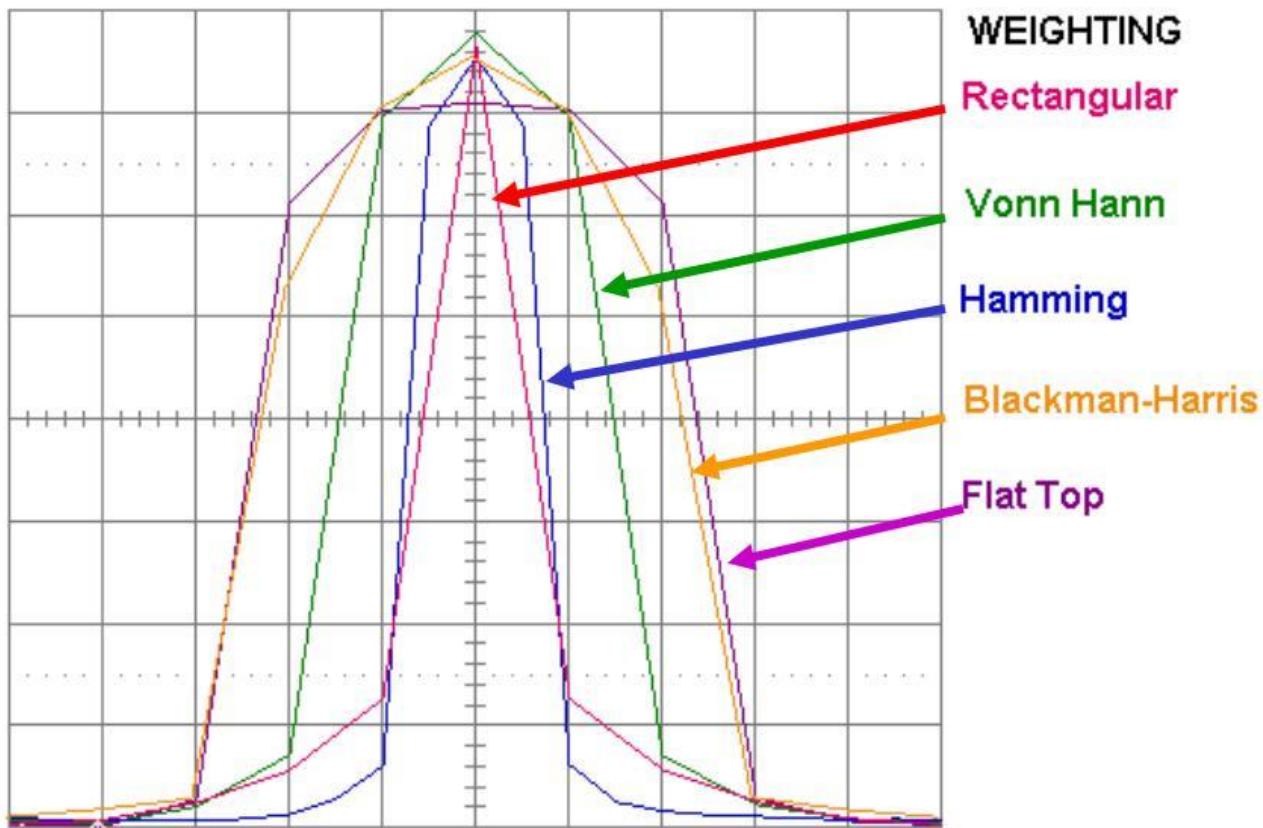
$$w(t) = g(t) \quad -T/2 < t < T/2$$

$$w(t) = 0 \text{ 其它}$$

加窗在时域上表现的是点乘，因此在频域上则表现为卷积。卷积可以被看成是一个平滑的过程。这个平滑过程可以被看出是由一组具有特定函数形状的滤波器，因此，原始信号中在某一频率点上的能量会结合滤波器的形状表现出来，从而减小泄漏。基于这个原理，人们通常在时域上直接加窗。

大多数的信号分析仪一般使用矩形窗 (rectangular) , 汉宁 (hann) , flattop 和其它的一些窗函数。

不同的窗函数对频谱谱线的影响不同，基本形状可以参看下图：



可以看到，不同的窗函数的主瓣宽度和旁瓣的衰减速度都不一样，所以对于不同信号的频谱应该使用适当的窗函数进行处理。

矩形窗(Rectangular)：加矩形窗等于不加窗，因为在截取时域信号时本身就是采用矩形截取，所以矩形窗适用于瞬态变化的信号，只要采集的时间足够长，信号宽度基本可以覆盖整个有效的瞬态部分。

汉宁窗(Von Hann)：如果测试信号有多个频率分量，频谱表现的十分复杂，且测试的目的更多关注频率点而非能量的大小。在这种情况下，需要选择一个主瓣够窄的窗函数，汉宁窗是一个很好的选择。

flattop 窗：如果测试的目的更多的关注某周期信号频率点的能量值，比如，更关心其 EUpeak, EUpeak-peak, EURms，那么其幅度的准确性则更加的重要，可以选择一个主瓣稍宽的窗，flattop 窗在这样的情况下经常被使用。

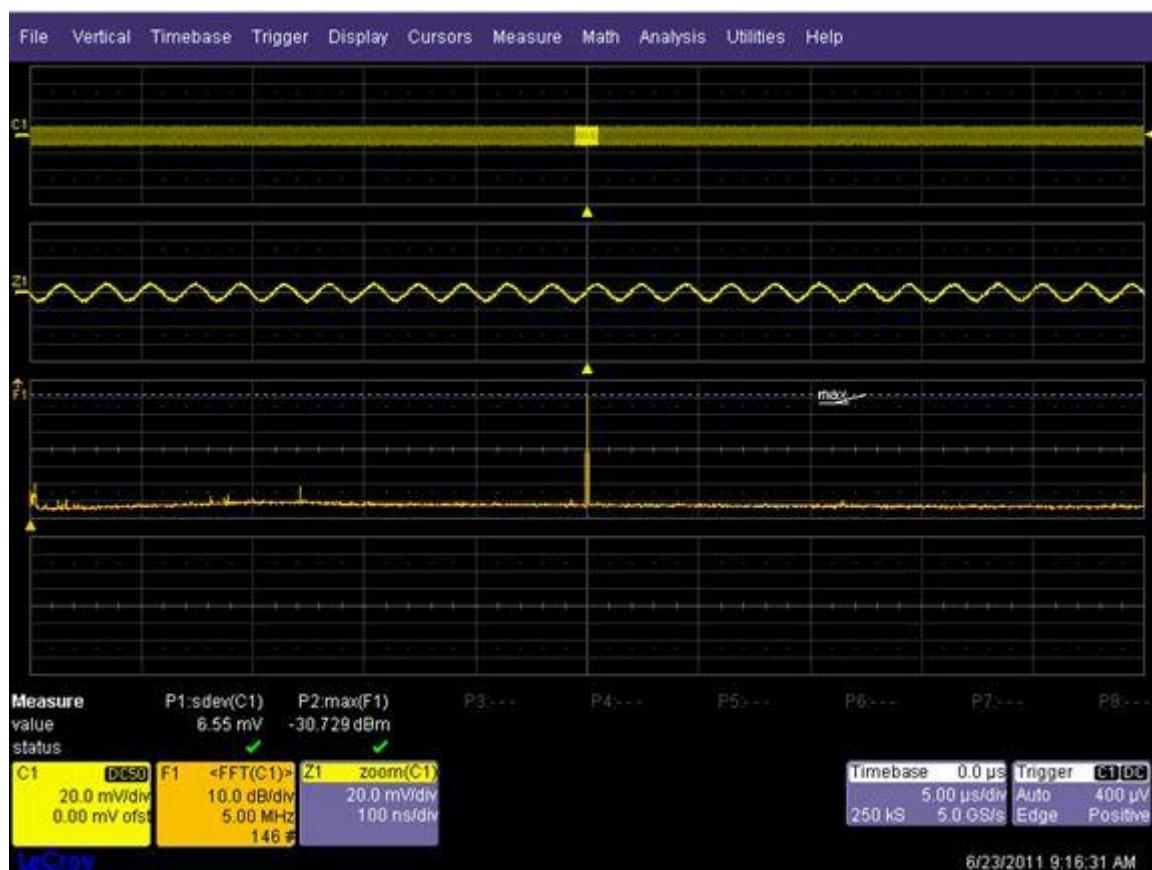
27.6 力科示波器和 Tek 示波器的 FFT 计算方法的比较

您可能也已经发现了这个问题：在示波器上进行 FFT 运算时，使用力科示波器和使用 Tek 示波器的计算结果似乎相差很大。产生这种差别的原因一方面可能是两者有效运算的采样点不一样。另外一个重要原因是 LeCroy 和 Tek 所使用的 FFT 运算的参考值不同，LeCroy 使用 dBm 为单位（参考值是 1mW

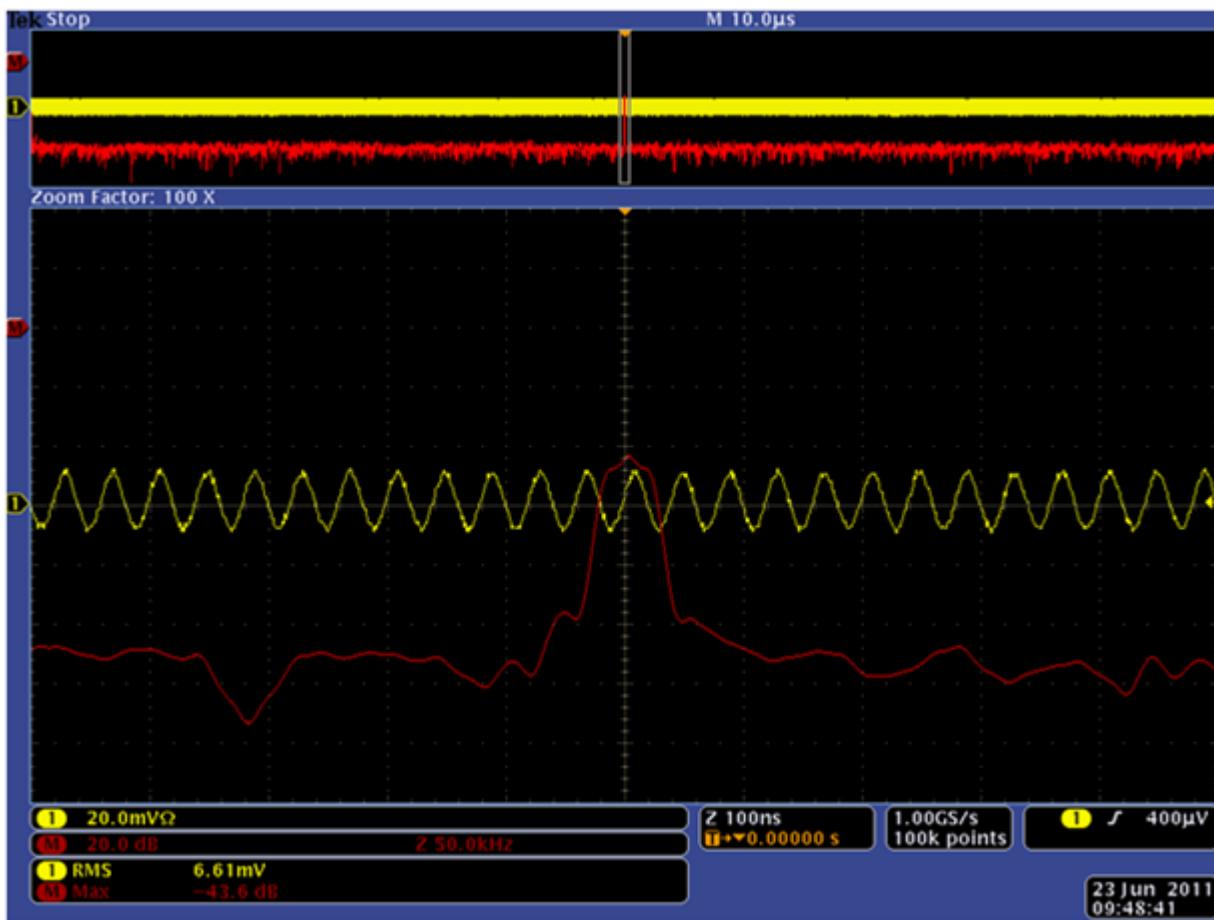
的功率值），而 Tek 使用 dB 为单位（参考值是 1V rms 的电压值），参考值不同产生的计算结果当然不一样！

dB(Deci-bel, 分贝) 是一个纯计数单位，本意是表示两个量的比值大小，没有单位。在工程应用中经常看到貌似不同的定义方式（仅仅是看上去不同）。对于功率， $dB = 10 \cdot \lg(A/B)$ 。对于电压或电流， $dB = 20 \cdot \lg(A/B)$ 。此处 A, B 代表参与比较的功率值或者电流、电压值。dB 的意义其实再简单不过了，就是把一个很大（后面跟一长串 0 的）或者很小（前面有一长串 0 的）的数比较简短地表示出来。dBm 是一个考征功率绝对值的值，计算公式为： $10\lg(\text{功率值}/1\text{mw})$ 。

此外，还有 dBV、dBuV、dBW 等等，仅仅是参考值选择的不同而已。如下是一个实测的例子，使用同一信号分别用 LeCroy 和 Tek 示波器进行 FFT 运算，下图是使用 LeCroy WaveRunner 64Xi 的测试结果：



下图是使用 Tek DPO4104 的测试结果：



测试所使用的信号幅值是 6.55 mV rms , 信号频率是 25 MHz

力科使用的计算方式如下:

$$dBm = 10 \log_{10} (((vrms^2)/50)/0.001) = 10 \log_{10} ((4.29E-5/50)/0.001) =$$

$$10 \log_{10}(8.5E-7/0.001) = 10 \log_{10} (8.5e-4) = 10 (-3.066) = -30.66 \text{ dBm}$$

Tek 使用的计算方式如下:

$$dB = 20 \lg (6.61E-3) = 10(-4.3596) = -43.59$$

换算关系如下:

$U = 20 \cdot \log_{10} \left(\frac{u}{u_0} \right)$ U = -43.67029995663982	$u_0 = \begin{cases} 1V & [\text{dBV}] \\ 1\mu V & [\text{dB}\mu V] \end{cases}$ dBV Convert
Amplitude : U =	U = 76.32970004336017
	dBuV Convert
u =	u = 0.006553676531174671
	V Convert

$P = 10 \cdot \log_{10} \left(\frac{p}{p_0} \right)$	$p_0 = 1\text{mW}$
$U = 20 \cdot \log_{10} \left(\frac{u}{u_0} \right)$	$u_0 = 1\mu\text{V}$
$p = \frac{u^2}{Z_c}$	
Impedance : <input type="text" value="50"/> Ω	
Power : <input type="text" value="-30.66"/> dBm	<input type="button" value="Convert"/>
Amplitude : <input type="text" value="76.32970004336017"/> dBuV	<input type="button" value="Convert"/>

不仅仅只是 FFT 计算方式的差别，我们以力科的 WaveMaster 8Zi-A 和 Tek 的 DPO70000 系列为例，在 WaveMaster 上您可以做最多 128M 个采样点的 FFT 运算，而在 DPO70000 上只能做 3.2M 个点的 FFT 运算，所以，这种差别才是本质上的！

27.7 栅栏现象

从直观上讲，时域分析清晰易见，示波器即是进行时域观察的主要工具，可观察波形形状、测量脉宽、相差等信息。但对于信号的进一步分析，比如测量各次谐波在所占的比重和能量分布，时域上的分析就力不从心了，但是利用从连续时间傅里叶变换发展而来的快速傅里叶变换 FFT 进行分析就很有意义了。通信系统中必不可少的要使用频谱分析技术，例如频分复用技术。频谱分析一般利用快速傅里叶变换 FFT 计算频率谱和功率谱，可直接用来提取特征频率和谱特征。因为计算机只能处理离散的数据点，但 FFT 是傅里叶变换的一种近似，与傅里叶变换存在差别，且具有固有的局限：栅栏现象。本小节就是在上面小节的基础上，从测试测量的角度，谈一谈在示波器的 FFT 运算中容易被大家忽略的一些问题。

27.7.1 频率分辨率与时基设置 (TimeBase)

频率分辨率的定义是：在使用 FFT 运算时，在频率谱上所能得到的最小的两个频率点间的间隔。

$$\Delta F = F_s / N = 1 / NT = 1 / T_p$$

称 ΔF 为频率分辨率，即：采样率/采样点数， ΔF 越小说明频率分辨率越高。 ΔF 仅与信号的实际长度成反比，即待分析的信号持续时间越长， ΔF 越小，频率分辨率越高。

27.7.2 栅栏效应与频率分辨率

示波器输入的信号一般都为非周期的连续信号 $x_a(t)$ ，它的频谱也是连续的，但是示波器所做的工作是将 $x_a(t)$ 进行等间隔采样并且截断，然后进行 FFT 的运算得到一个离散的频谱图，相当于对连续的频谱图也进行了采样。这样有一部分频谱分量将被“挡在”采样点之外，就好像我们在通过一个栅栏观察频



谱图，这种现象称为“栅栏效应”。这样就有可能发生一些频谱的峰点或谷点被栅栏所拦住，不可能被我们观察到。

不管是时域采样还是频域采样，都有相应的栅栏效应。只是当时域采样满足采样定理时，栅栏效应不会有什么影响。而频域采样的栅栏效应则影响很大，“挡住”或丢失的频率成分有可能是重要的或具有特征的成分，使信号处理失去意义。

栅栏效应是制约频谱分析谐波分析精度的一个瓶颈。栅栏效应在非同步采样的时候，影响尤为严重。在非同步采样时，由于各次谐波分量并未能正好落在频率分辨率上，而是落在两个频率分辨率之间。这样通过 FFT 不能直接得到各次谐波分量的准确值，而只能以临近的频率分辨率的值来近似代替，这就是栅栏效应降低频谱分析精度的原因。

由此我们可以得出这样的结论：**减小栅栏效应可用通过提高频谱采样间隔也就是频率分辨率的方法来解决。间隔小，频率分辨率高，被“挡住”或丢失的频率成分就会越少。但是频率分辨率的提高会增加采样点数，使计算工作量增加。**

我们可以通过两种方式增加频率分辨率：

a：物理分辨率=采样频率/采样点数。

物理分辨率的实际意义在于它可以衡量 FFT 实际上可以区分的频率分量的间隔。提高物理分辨率的方法一般是通过增加数据的有效长度，这相当于在模拟域增加了矩形窗的宽度。从而在模拟域减小了 sinc

主旁瓣宽度，减小了相邻频率分量的混叠。

这种增加采样点的方法主要针对无限长序列的 FFT 计算。对于无限长序列，不像有限长序列那样必须补零来提高视在分辨率，无限长序列可以通过增加数据长度来提高物理分辨率。

b：视在分辨率=采样频率/分析点数

在序列尾部补零的方法可以使得分析点数增大，故补零的方法可以提高频谱的视在分辨率。对序列的尾部补零的方法主要针对有限长序列。对于有限长序列，有时只能用补零或者插值来改善频率分辨率。通过补零处理，使得频域采样密度增大，得到高密度谱。补零的方法所得到的频谱图所改善的只是图形的视在分辨率，并不能得到频谱的更多细节。增加采样点数，增加了输入序列的阶次，从而提供频谱的更多细节，这是真正的分辨率（物理分辨率）。对序列只补零而不增加数据，输入序列和它的频谱阶次依旧没有提高，只是把频谱画的密一些，所以改善的只是图形的视在分辨率，并不能得到频谱的更多细节。增加序列的长度能够改善频谱的真正分辨率，这是基本的规律。

上面的讨论可知，改善分辨率的具体方法有如下两种

- (1) 对有限长序列采取尾部补零的方法提高视在分辨率
- (2) 对无限长序列通过真正增加采样点来提高物理分辨率

有限长序列和无限长序列是针对实际信号来说的，例如非周期的但是包含无限长信息的信号可以称为无限长序列，严格的周期信号和脉冲信号（脉冲之前和之后无限长时间内都是无效信息）都可以称为有限长序列，当然实际上严格的周期信号是不存在的。对于示波器来说，时间窗口内采集到的可以是有

限长序列的全部信息或者是无限长序列的一部分信息。所以，如果采集到的是有限长序列的全部信息，那么只能通过补零的方式增加视在分辨率，如果采集到的是无限长序列的一部分信息，那么可以通过增加时间窗口的长度（不是采样点）来增加物理频率分辨率。

请看下面的实例：

图 1 中正弦波测试使用的时基是 5ns/div，波形时间长度是 50ns，计算 FFT 之后的频谱分辨率是 20MHz (1/50ns)

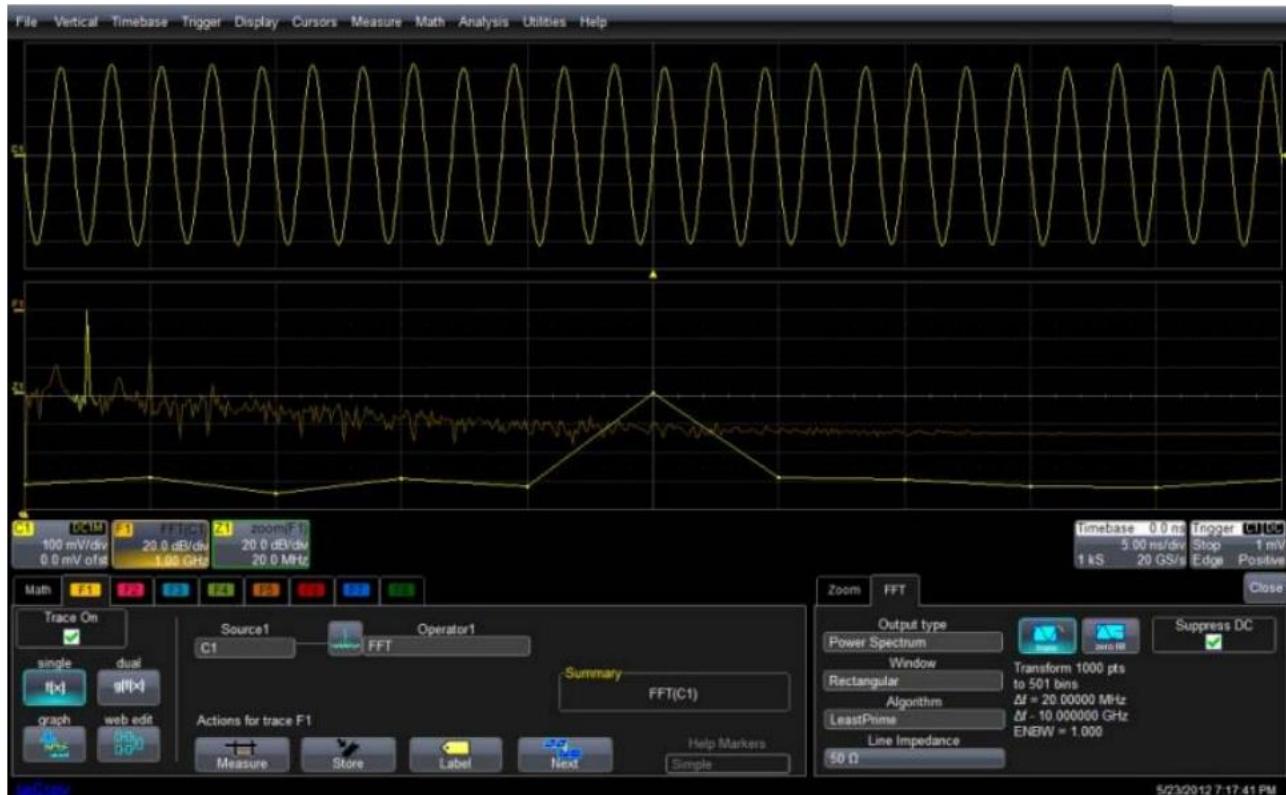


图 1 捕获 50ns 的信号，频率分辨率是 20MHz

如果改变时基设置，频谱分辨率会有变化。如图 2 所示：将时基设置为 10ns/div，波形长度是 100ns，频谱分辨率可以提高到 10MHz。

对于通过补零的方法增加 FFT 频谱的视在分辨率，力科的示波器也有相应的解决方案。力科示波器使用了两种非常常用的 FFT 算法供用户选择：Cooley-Tukey 算法和 LeastPrime 算法。Cooley-Tukey 算法也称为 Power2 算法，它提供了计算机一种非常快速的 FFT 计算方式，计算的 FFT 点数规模是 2 的整数方次，因此它会在示波器时域采集的信号中截取 2 的 N 次方的整数来作为 FFT 计算的时域样本，该截取的整数是最接近于采样点的整数。如下图 2 所示：

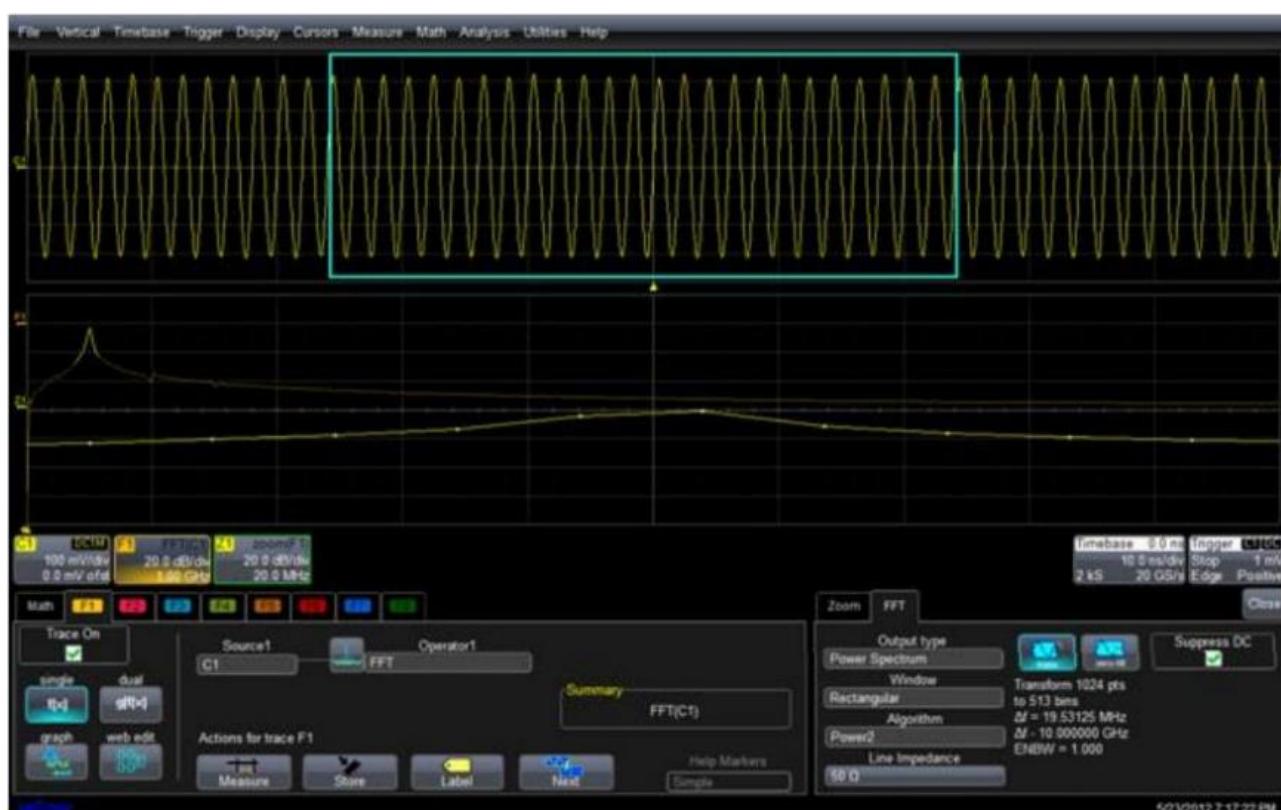


图 2 捕获 100ns 的信号，频率分辨率是 10MHz

图中的正弦波频率为 500MHz，时基设置为 10ns/div，采样率为 20GS/s，时域采样点数为 2000points，

使用 Power2 算法截取 2000 点中的 1024 点 (2^{10})，如图中的蓝框所示（注意是从信号的中间部分截取），因此截取的时间窗口为 $1024 \times 20\text{ps} = 51.2\text{ns}$ ，是 500MHz 信号的 25.6 个周期，由于截取的周期非整数倍，不可避免会产生频谱泄露，如图中 FFT 的旁瓣所示，此时的频率分辨率可以达到 19.35125MHz。

如果采用另外一种 FFT 算法 LeastPrime，可以将整个示波器时域采集的采样点进行 FFT 运算，LeastPrime 算法计算的 FFT 点数规模是 $2^N + 5^k$ ，因此 2000 点 = $2^4 + 5^3$ ，不需要截取原始数据就可以运算，但是代价是计算的速度可能会慢一些（尽管我们可能觉察不到），频率分辨率可以提高到 10MHz。使用 Power2 算法也可以不采用截取原始波形的方式，此时我们可以选择 Zero Fill（补零）的方式，增加采样点数。比如，在 2000 点中补 48 个点， $2048 = 211$ ，如图 3 所示：



图 3 补零的放出提高频率分辨率

这 48 个点补的方式是头尾各补一半，但是有可能补的不是 0，头 24 个点与第一个采样点值相同，尾 24 个点与最后一个采样点值相同（所以称之为 Zero Fill 是不完全准确的）。这里我们推荐 Zero Fill 的方法只在分析冲击信号 FFT 频谱的情况下使用。

补零法虽然能增加频谱图的视在分辨率，但是由于补的都是无效数据，所以对于频率分辨率真正的改善没有帮助，但是补零有它的好处：

1. 补零后，其实是对 FFT 结果做了插值，克服“栅栏”效应，使谱外观平滑化；我把“栅栏”效应形象理解为，就像站在栅栏旁边透过栅栏看外面风景，肯定有被栅栏挡住比较多风景，此时就可能漏掉较大频域分量，但是补零以后，相当于你站远了，改变了栅栏密度，风景就看的越来越清楚了。
2. 由于对时域数据的截短必然造成频谱泄露，因此在频谱中可能出现难以辨认的谐峰，补零在一定程度上能消除这种现象。除此之外，很多人都有这样的误区：认为通过增加待分析的计算点数而不是增加采样时间就可以使 FFT 之后的频谱更加“精细”（频率分辨率更高）。这样的误解一般来自于示波器的用户，因为当示波器采样点比较少时，FFT 的计算出来的频谱图也会很少，频谱看起来非常粗糙。这时工程师会非常有冲动把时域的采样点增多（用示波器上的插值算法很容易实现），但是如果采集信号的时间长度是不变的，工程师会发现 FFT 计算之后的频谱并没有显得更加“精细”，频率分辨率并没有任何改善。实际上使用插值或者增加采样率的方式仅仅是展宽了 FFT 之后的频谱带宽。如下图 4 所示：



图 4 插值方式并不能改变频率分辨率

左上方使用了较少的时域采样点 C1，右上方使用了较高的采样率 C2，但是采样时间是相同的。左下是对 C1 进行 FFT 之后的频谱 F1，右中是对 C2 进行 FFT 之后的频谱 F2，右下是对 F2 相同频段进行了放大。可以看到 F2 比 F1 的频宽增加了，但是对 F1 频段放大之后的频谱和 F1 一样，没有任何频率分辨率的改善。

由此我们可以得出结论，对 C1 进行插值后，额外的采样点仅仅存在于较高频段，会展宽频谱的带宽，但是插值方式对于增加我们感兴趣频段的频谱分辨率没有任何帮助。

那么如果我们只对对 FFT 之后的频谱进行插值效果如何呢？如下图 5 所示：

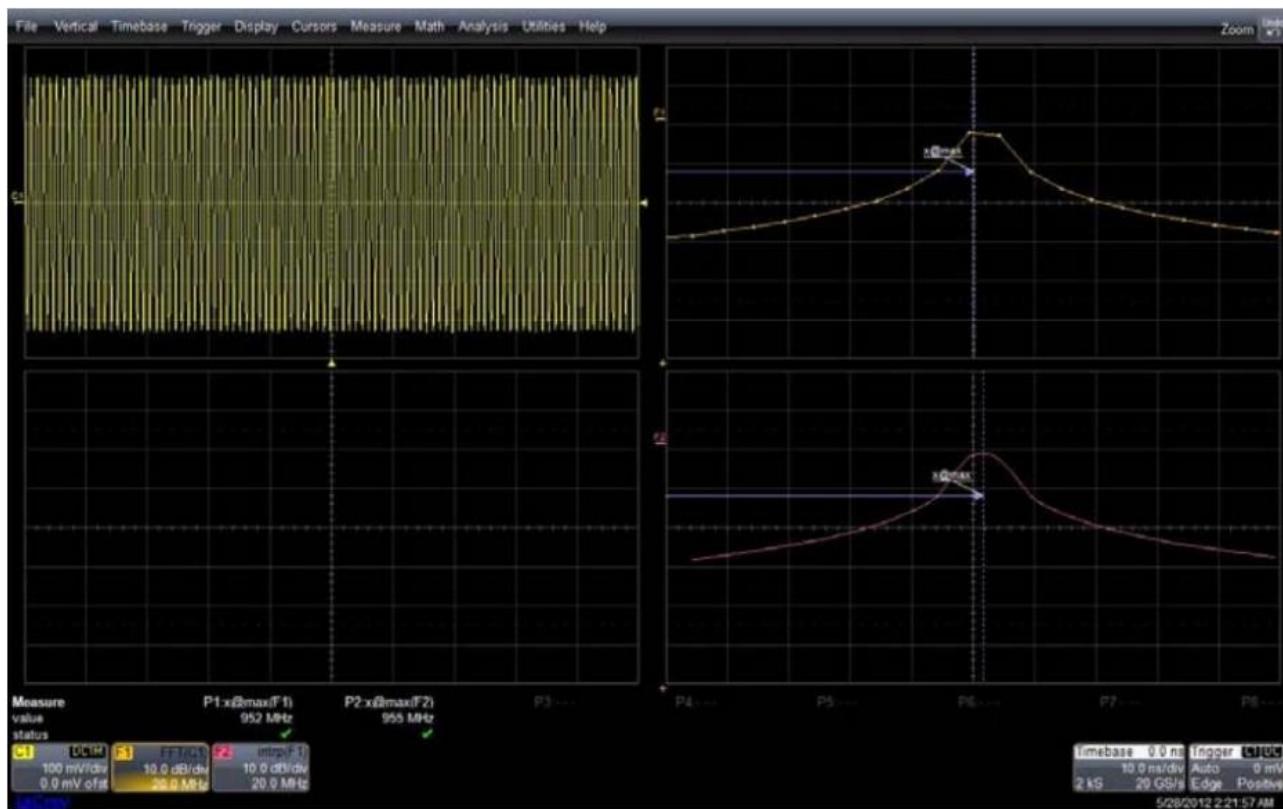


图 5 频域插值方式是频谱图看起来更密

图中展示了对频域插值之后的效果，并没有使频谱看起来更“窄”（毕竟插值出来的点都是假点），但是我们注意到，频域插值可以使频谱的测量更加精确。图中正弦波的频率是 955MHz，插值之后频谱的 Peak 频率读数 P2 是 955MHz，插值之前 P1 的读数为 952MHz。

总之，FFT 是进行信号频域分析的最广泛使用的标准化方法，也是现代数字示波器中标配的数学运算函数，我们更多了解 FFT 应用的细节，能更加有效地利用好这个工具，从 FFT 中得到更多有价值的信息。

27.8 窗函数对于 FFT 结果的影响

所谓频谱泄露，就是信号频谱中各谱线之间相互干扰，使测量的结果偏离实际值，同时在真实谱线的两侧的其它频率点上出现一些幅值较小的假谱。产生频谱泄露的主要原因是采样频率和原始信号频率不同步，造成周期的采样信号的相位在始端和终端不连续。简单来说就是因为计算机的 FFT 运算能力有限，只能处理有限点数的 FFT，所以在截取时域的周期信号时，没有能够截取整数倍的周期。信号分析时不可能取无限大的样本。只要有截断不同步就会有泄露。

在图 6 和图 7 中，为了最大化 FFT 运算之后的频率分辨率，我们使用了矩形窗。图中的时域信号是 500MHz 正弦波信号，在频谱上应该仅在 500MHz 频点上看到谱线。FFT 运算研究的是整个时间域 ($-\infty, +\infty$) 与频域的关系，所以对于矩形窗函数截取的波形应该认为是无穷延续的，因此，矩形窗 100ns

时间窗内，包含了 500MHz 正弦波整 50 个周期，所以波形的首尾能够整周期得无缝连接，FFT 之后的频谱会在 500MHz 频点看到较为纯净的能量值。如下图 6 所示

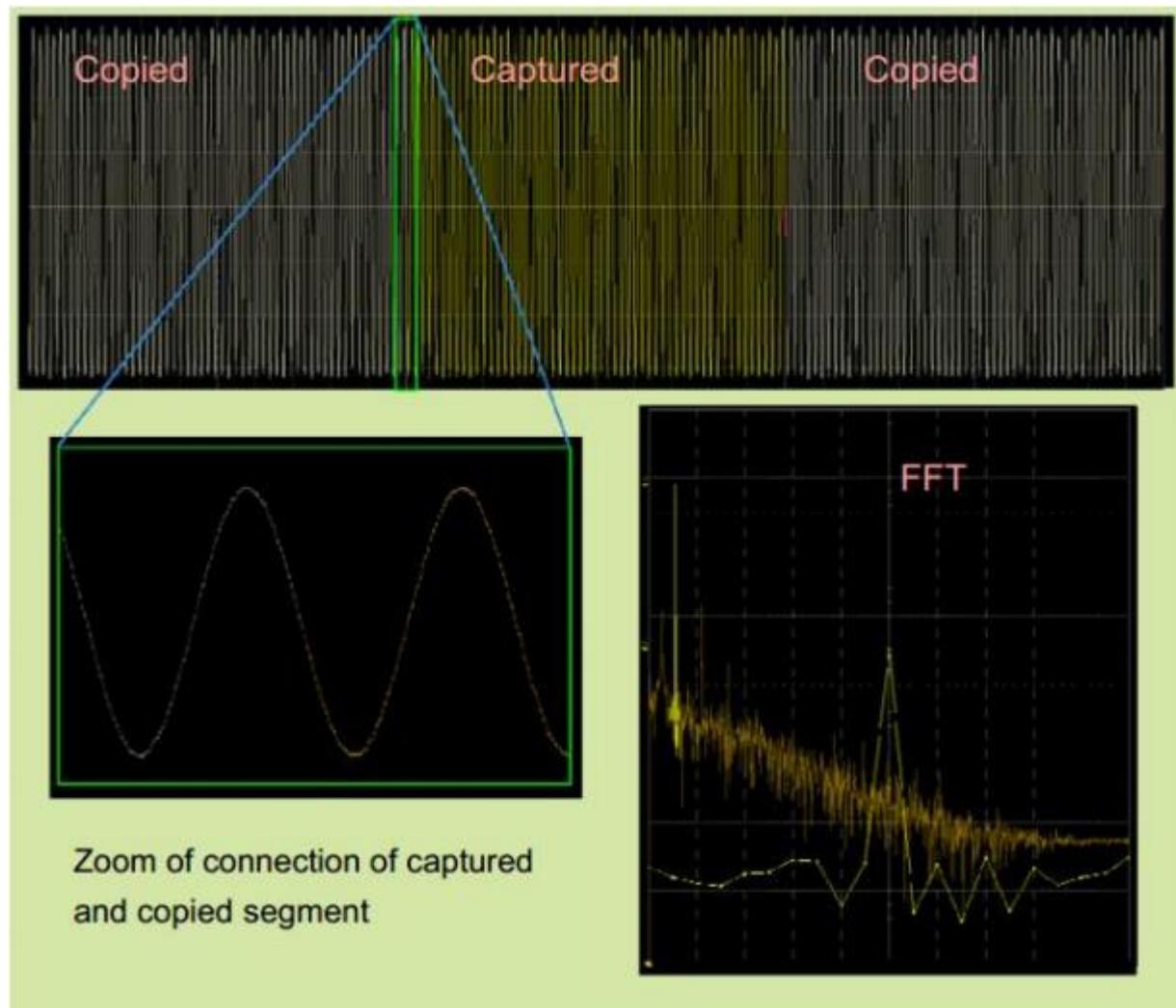


图 6：矩形时间窗口内包含整数倍周期的信号，首尾可以“无缝”连接

事实上，大多数类型的信号都不满足上面的这种特殊情况，绝大多数信号在时间窗口内都不是整周期的倍数，在这种情况下，FFT 之后的频谱就不能看做连续的正弦波了。例如，如果该正弦波的频率是 495MHz，在 100ns 时间窗口内包含 49.5 个周期，因此在截取窗口的首尾部分就存在很大程度上的“不连续”，这种“不连续”会直接影响 FFT 之后的结果。“不连续”部分的能量会散落在整个频谱范围内，使用 100ns 时间窗口，FFT 之后的频率分辨率是 10MHz，495MHz 频点即落在 490MHz 与 500MHz 之间，所以 495MHz 正弦波信号的能量分成两部分，所以从频谱上看，峰值谱线明显降低了，这被称作是频谱泄露（Leakage）。如下图 7 所示：

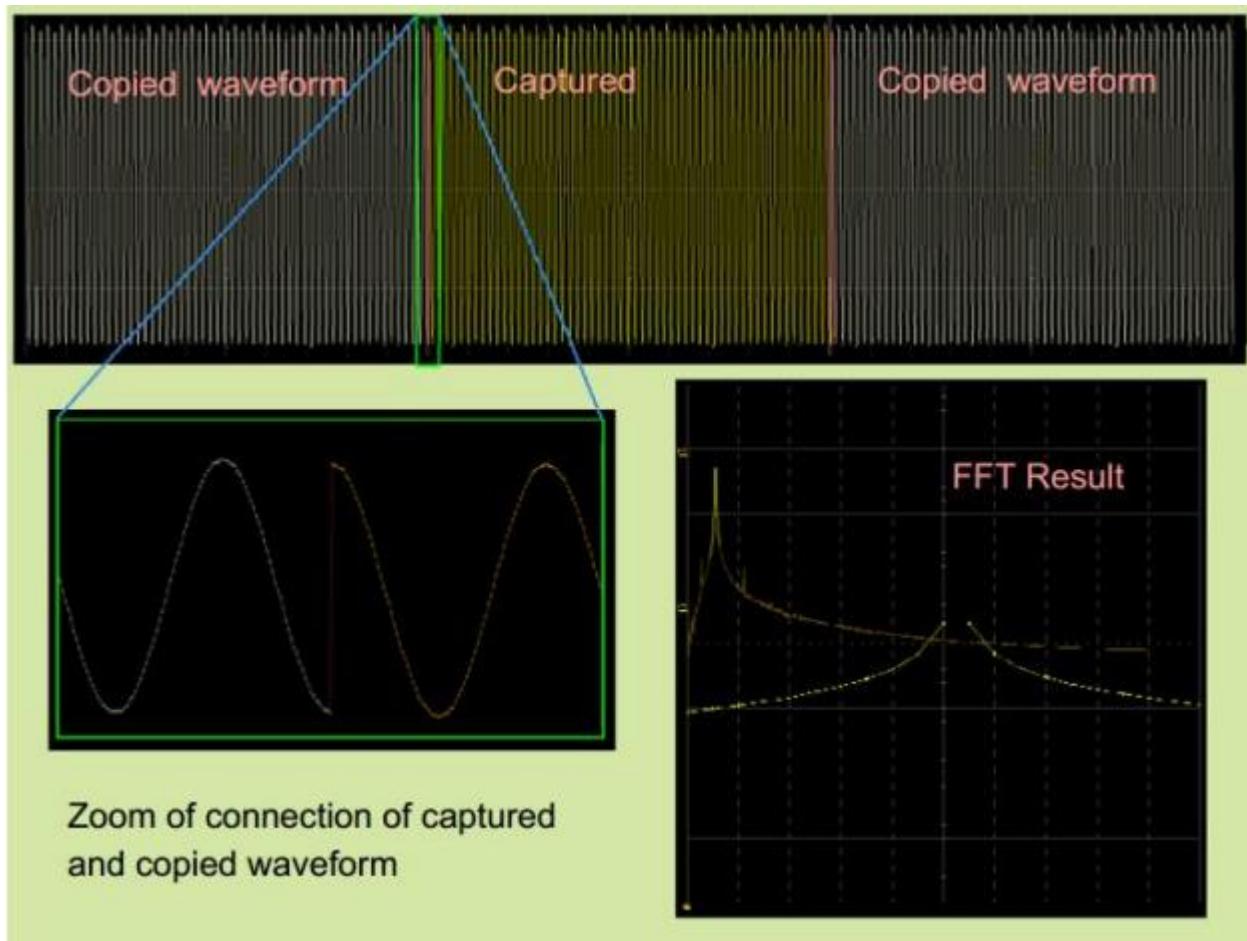


图 7：对于非整数倍周期信号进行 FFT 运算的效果

不同的窗函数对信号频谱的影响是不一样的，这主要是因为不同的窗函数，产生泄漏的大小不一样，频率分辨能力也不一样。信号的截短产生了能量泄漏，而用 FFT 算法计算频谱又产生了栅栏效应，从原理上讲这两种误差都是不能消除的，但是我们可以通过选择不同的窗函数对它们的影响进行抑制。(矩形窗

主瓣窄，旁瓣大，频率识别精度最高，幅值识别精度最低；布莱克曼窗主瓣宽，旁瓣小，频率识别精度最低，但幅值识别精度最高)

为了减少频谱旁瓣和栅栏效应的影响，我们在 FFT 运算中使用窗函数，图 8 显示了 Hanning (汉宁窗) 使用后的效果。窗函数位于下图中左上角的栅格中红色的波形，叠加在黄色的时域信号上。窗函数与时域信号时域相乘。结果显示在左下角的蓝色波形。右下角的粉色波形显示了进行 FFT 计算之后的频谱图，相对于右上角的使用窗函数之前的频谱图来说，旁瓣的幅度已经大大减低。

对于不同的应用需求还有多种不同的窗函数供工程师选择，Hanning (汉宁窗) 是使用最广泛的一种

窗函数，除此之外，Hamming (海明窗)，Flat-top 窗和 Balckman-Harris 窗的效果，在下图中做了对比，图中的信号使用 500MHz 正弦波，矩形窗产生最窄的谱线，加 Flat-top 窗谱线最宽。

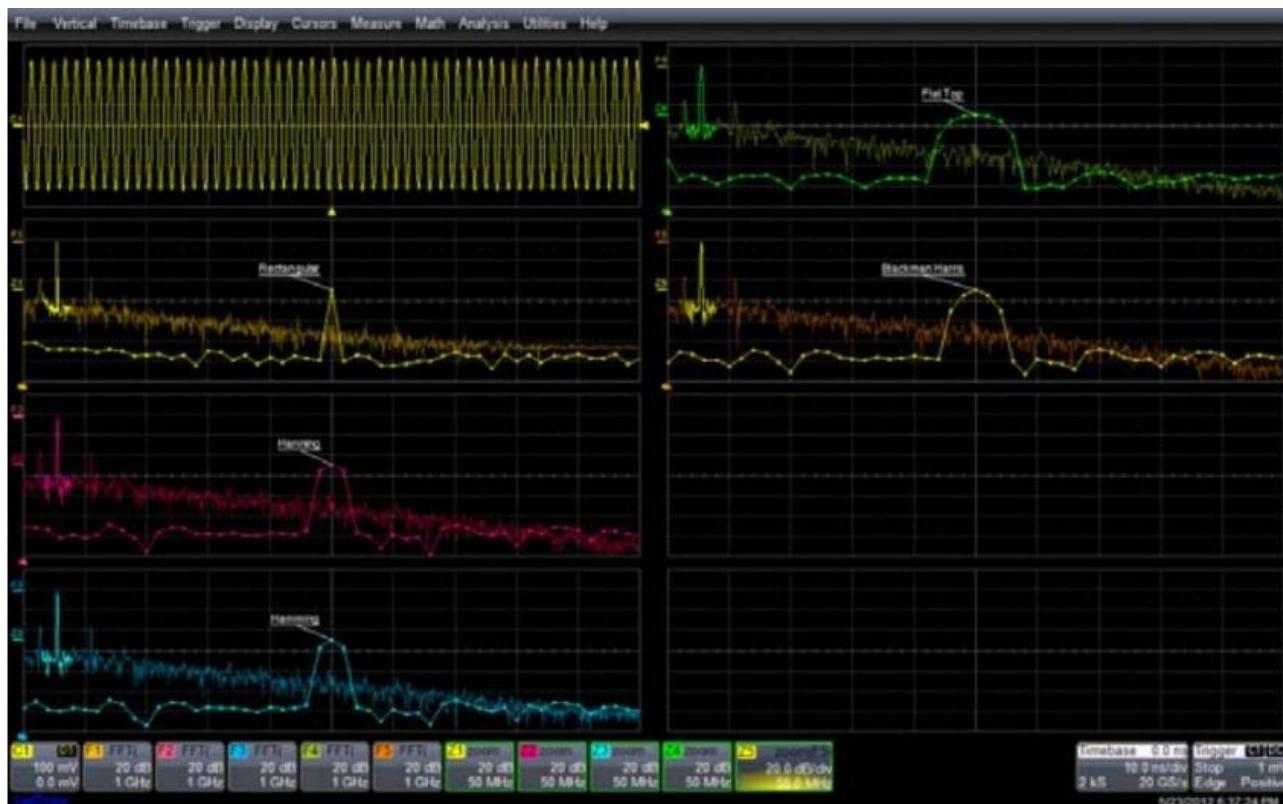


图 8: 500MHz 正弦波频谱在不同窗函数下的对比

下图 9 中显示了同样的窗函数对比，但是采用 495MHz 正弦波进行 FFT 运算，矩形窗显示了最差旁瓣效果，Flat-top 窗函数基本上保持了与图 8 一样的旁瓣效果，所以我们看到旁瓣的影响和精确频率分辨率有时候是不可兼得的。(矩形窗主瓣窄，旁瓣大，频率识别精度最高，幅值识别精度最低；Flat-top 窗主瓣宽，旁瓣小，频率识别精度最低，但幅值识别精度最高)。

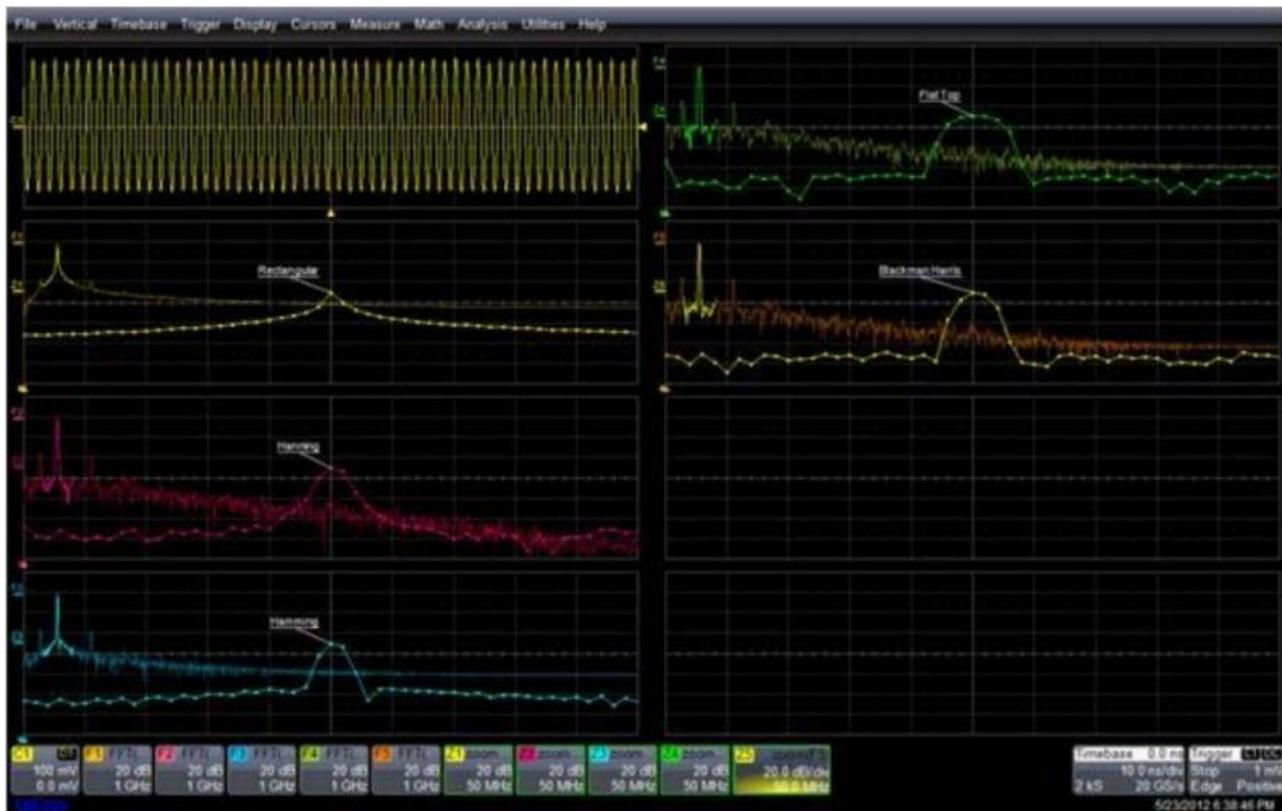


图 9：495MHz 正弦波频谱在不同窗函数下的对比

图 10 中显示了不同的窗函数对于栅栏效应的抑制效果，图中的正弦波频率从 450MHz 增加到 550MHz，步进值为 500KHz，Flat-top 窗在整个频段上基本保持相同的值，矩形窗函数有约 4dB 的差值。

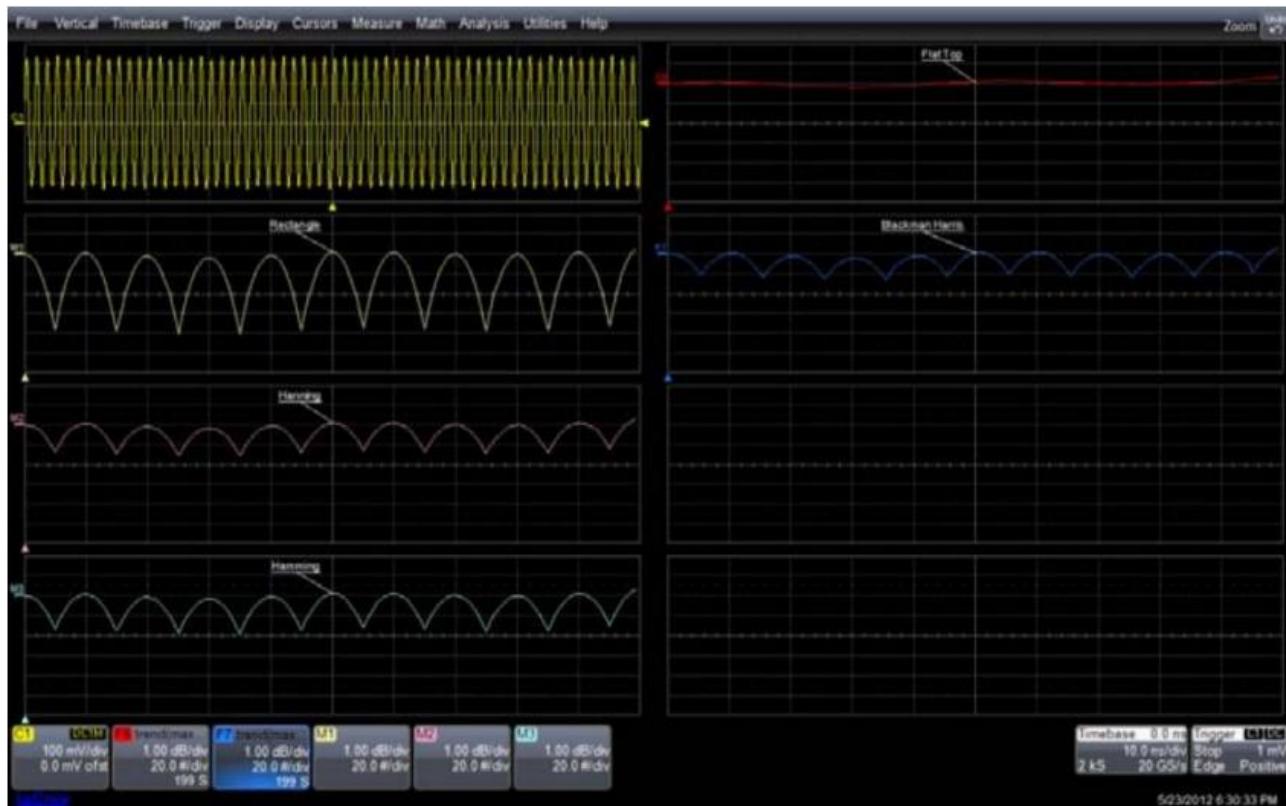


图 10：从 500MHz 到 600MHz，不同窗函数的峰值变化

我们把关于窗函数的一些重要的结论总结如下：

- 1、连续的 FFT 运算并没有窗函数的概念，因为信号是充满时间坐标轴的，FFT 之后的频率分辨率是 0，并不存在栅栏效应。但是，示波器采集和处理的信号全部是离散的采样点，是非连续的，所以 DFT 之后的频谱一定存在栅栏效应。
- 2、如果能够保证示波器时间窗口内的信号是整数倍周期的（并且在信号时间窗口之前和之后的信号都是严格周期重复的），或者采集信号时间足够长，基本上可以覆盖到整个有效信号的时间跨度。这种方法经常在瞬态捕捉中被使用到，比如说冲击试验，如果捕捉的时间够长，捕捉到的信号可以一直包括了振动衰减为零的时刻。在这种情况下，可以不加窗函数。
- 3、如果不满足 1 和 2，那么 FFT 计算之后的频谱就不可避免受到频谱泄露（Leakage）的影响，如频点分裂，幅值能量不精确等等，总之就是频谱线比较难看，这时候就需要使用适当的窗函数，以满足我们工程测量的需要。
- 4、示波器中的 FFT 运算，不加窗和加矩形窗是一回事。
- 5、窗函数会改变频域波形，让频谱形成人们“喜欢”的形状，但是不会本质上消除频谱泄露，不同的窗函数都有其独特的特性，我们只需要根据工程测试的需要，选择一款合适的就可以了。



27.9 窗函数选择指南

如果在测试中可以保证不会有泄露的发生，则不需要用任何的窗函数（在软件中可选择 uniform）。但是如同刚刚讨论的那样，这种情况只是发生在时间足够长的瞬态捕捉和一帧数据中正好包含信号整周期的情况。

如果测试信号有多个频率分量，频谱表现的十分复杂，且测试的目的更多关注频率点而非能量的大小。在这种情况下，需要选择一个主瓣够窄的窗函数，汉宁窗是一个很好的选择。

如果测试的目的更多的关注某周期信号频率点的能量值，比如，更关心其 EUpeak,EUpeak-peak,EUrms 或者 EUrms2，那么其幅度的准确性则更加的重要，可以选择一个主瓣稍宽的窗，flat-top 窗在这样的情况下经常被使用。

如果被测信号是随机或者未知的，选择汉宁窗。

27.10 总结

本章节的内容非常值得初学者看，如果你是初学者的话，建议认真的多看几遍。



第28章 FFT 和 IFFT 的 Matlab 实现 (幅频响应 和相频响应)

本章主要讲解 fft, ifft 和 fftshift 在 matlab 上的实现。

28.1 初学者重要提示

28.2 Matlab 的 FFT 函数

28.3 Matlab 的 IFFT 函数

28.4 Matlab 的 FFTSHIFT 函数

28.5 总结

28.1 初学者重要提示

- ◆ 求解 FFT 相频时的修正比较重要，本章做了一个简易修正方法。重点看 28.2.5 小节。

28.2 Matlab 的 FFT 函数

28.2.1 函数语法

$Y = \text{fft}(x)$

$Y = \text{fft}(X, n)$

$Y = \text{fft}(X, n, \text{dim})$

28.2.2 函数定义

$Y = \text{fft}(x)$ 和 $y = \text{ifft}(X)$ 分别用于实现正变换和逆变换，公式描述如下：

$$Y(k) = \sum_{j=1}^n X(j) W_n^{(j-1)(k-1)}$$
$$X(j) = \frac{1}{n} \sum_{k=1}^n Y(k) W_n^{-(j-1)(k-1)},$$

其中

$$W_n = e^{-2\pi i / n}$$



28.2.3 函数描述

Y = fft(X)

用快速傅里叶变换 (FFT) 算法计算 X 的离散傅里叶变换 (DFT)。

- ◆ 如果 X 是向量，则 fft(X) 返回该向量的傅里叶变换。
- ◆ 如果 X 是矩阵，则 fft(X) 将 X 的各列视为向量，并返回每列的傅里叶变换。
- ◆ 如果 X 是一个多维数组，则 fft(X) 将尺寸大小不等于 1 的第一个数组维度的值视为向量，并返回每个向量的傅里叶变换。

注意这里第一个尺寸不为 1 是指一个矩阵的第一个尺寸不为 1 的维。

比如一个矩阵是 2×1 ，那么第一个尺寸不为 1 的维就是行（尺寸为 2）。

X 是 $1 \times 2 \times 3$ 表示第一个尺寸不为 1 的维就是列（尺寸为 2）。

X 为维数 $5 \times 6 \times 2$ 的话，第一个尺寸不为 1 的维就是行（尺寸为 5）。

Y = fft(X, n)

返回 n 点 DFT。如果未指定任何值，则 Y 的大小与 X 相同。

- ◆ 如果 X 是向量且 X 的长度小于 n，则为 X 补上尾零以达到长度 n。
- ◆ 如果 X 是向量且 X 的长度大于 n，则对 X 进行截断以达到长度 n。
- ◆ 如果 X 是矩阵，则每列的处理与在向量情况下相同。
- ◆ 如果 X 为多维数组，则大小不等于 1 的第一个数组维度的处理与在向量情况下相同。

Y = fft(X, n, dim)

返回沿维度 dim 的傅里叶变换。例如，如果 X 是矩阵，则 fft(X,n,2) 返回每行的 n 点傅里叶变换。

28.2.4 FFT 实例一：幅频响应

傅里叶变换的一个常见用途就是查找埋藏在噪声信号中的实际信号的频率成分。下面我们考虑一个这样的例子：

采样率是 1000Hz，信号由如下三个波形组成。

- (1) 50Hz 的正弦波、振幅 0.7。
- (2) 70Hz 正弦波、振幅 1。
- (3) 均值为 0 的随机噪声。

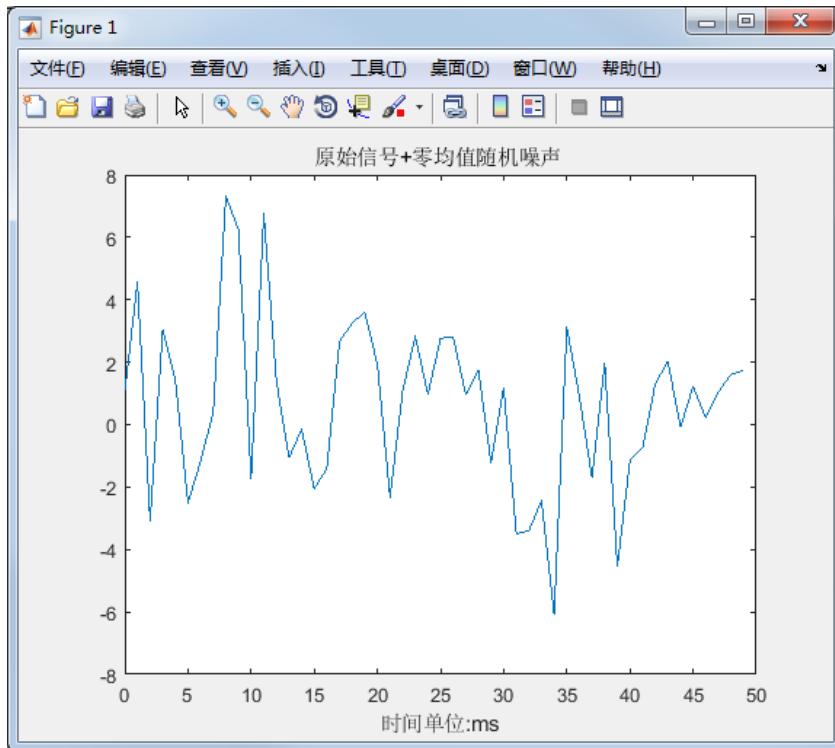
实际运行代码如下：

```
Fs = 1000;          %采样率
T = 1/Fs;           %采样时间单位
L = 1000;           %信号长度
t = (0:L-1)*T;     %时间序列

x = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t); %原始信号
y = x + 2*randn(size(t));                      %原始信号叠加了噪声后
plot(Fs*t(1:50), y(1:50));                   %绘制波形
title('原始信号+零均值随机噪声');
```

```
xlabel('时间单位:ms');
```

运行 Matlab 后，显示波形如下：



通过上面的截图，我们是很难发现波形中的频率成分，下面我们通过 FFT 变换，从频域观察就很方便了，

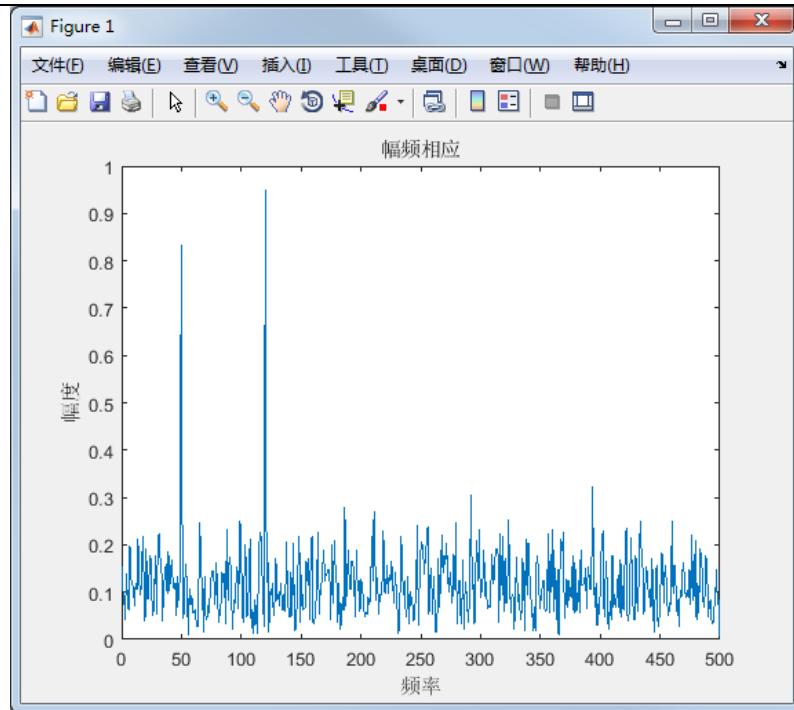
Matlab 运行代码如下：

```
Fs = 1000; %采样率
T = 1/Fs; %采样时间单位
L = 1000; %信号长度
t = (0:L-1)*T; %时间序列

x = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t); %原始信号
y = x + 2*randn(size(t)); %原始信号叠加了噪声后

NFFT = 2^nextpow2(L); %求得最接近采样点的2^n，由于上面是1000点，那么最近的就是1024点。
Y=fft(y,NFFT)/L; %进行FFT变换，除以总的采样点数，方便观察实际值。
f = Fs/2*linspace(0, 1, NFFT/2+1); %频率轴，这里只显示Fs/2部分，另一半是对称的。

plot(f,2*abs(Y(1:NFFT/2+1))) %绘制波形
title('幅频相应');
xlabel('频率');
ylabel('幅度');
```



从上面的幅频响应，我们可以看出，两个正弦波的频谱并不是准确的 0.7 和 1，而是比较接近，这个就是我们在上节教程中所示的频谱泄露以及噪声的干扰。

28.2.5 FFT 实例二：相频响应（重要）

这里我们以采样两个余弦波组成的信号为例进行说明，并求出其幅频和相频响应。

- (1) 50Hz 的余弦波，初始相位 60° ，振幅 1.5。
- (2) 90Hz 的余弦波、初始相位 60° ，振幅 1。
- (3) 采样率 256Hz，采集 256 个点。

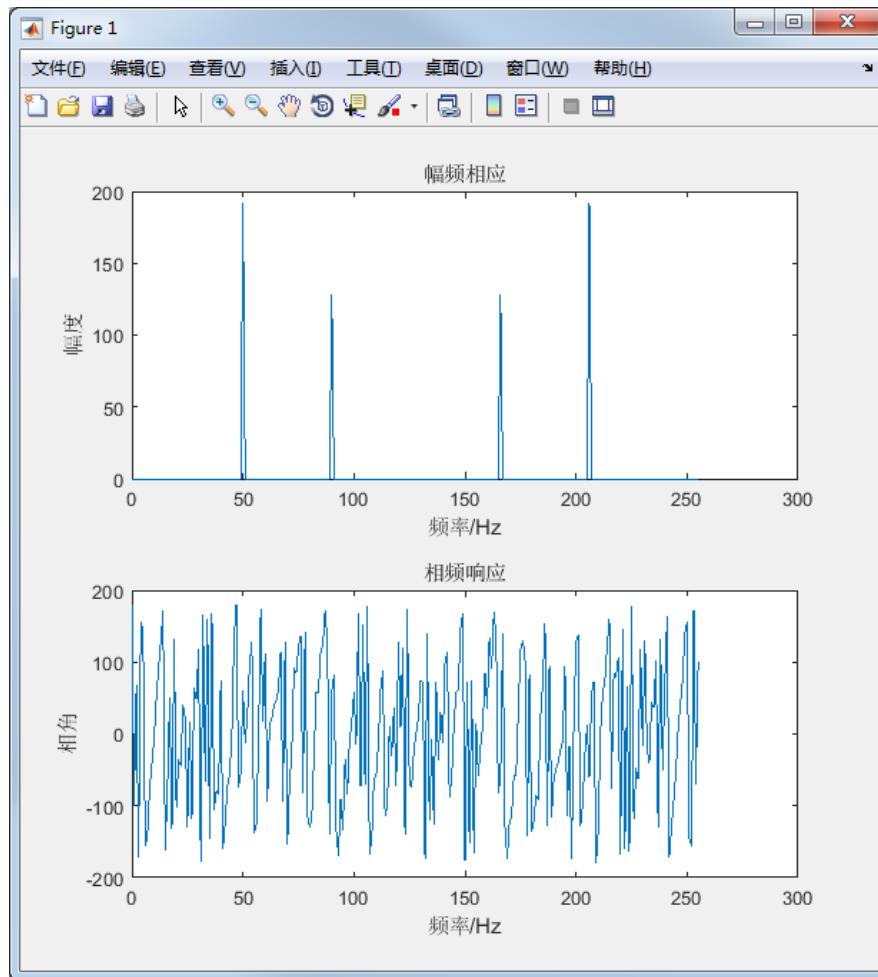
Matlab上面运行的代码如下：

```
Fs = 256; % 采样率
N = 256; % 采样点数
n = 0:N-1; % 采样序列
t = 0:1/Fs:1-1/Fs; % 时间序列
f = n * Fs / N; % 真实的频率

x = 1.5*cos(2*pi*50*t+pi/3)+cos(2*pi*90*t+pi/3); % 原始信号
y = fft(x, N); % 对原始信号做FFT变换
Mag = abs(y); % 求FFT转换结果的模值
subplot(2, 1, 1);
plot(f, Mag); % 绘制幅频响应曲线
title('幅频响应');
xlabel('频率/Hz');
ylabel('幅度');

subplot(2, 1, 2);
plot(f, angle(y)*180/pi); % 绘制相频响应曲线，注意这将弧度转换成了角度
title('相频响应方式一');
xlabel('频率/Hz');
ylabel('相角');
```

运行后求出的幅频相应和相频响应结果如下：



求出的幅频响应没问题，而相频响应杂乱无章，造成这个问题的根本原因很多频段的幅值非常小，他们的相角可以不显示出来，这样就可以方便的查看相频响应了。基于此，修正后的代码如下：

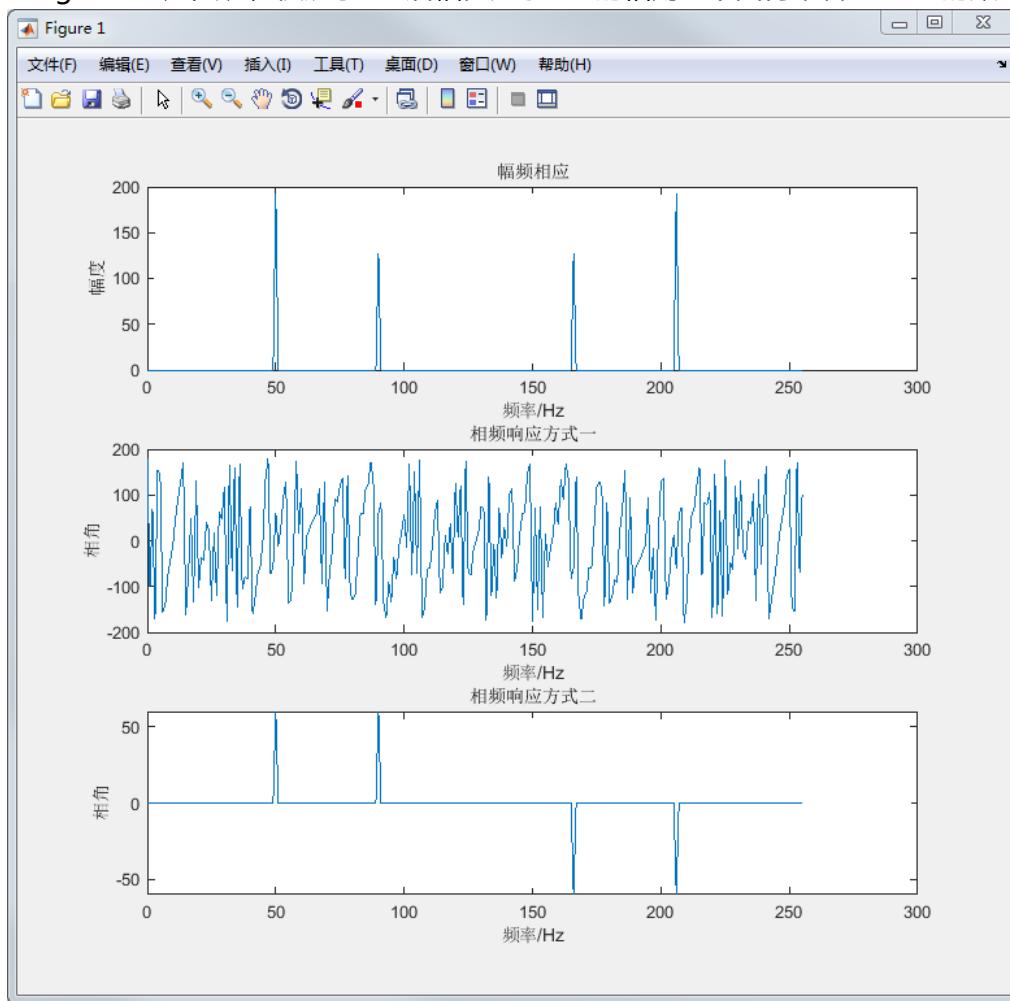
```
Fs = 256; % 采样率
N = 256; % 采样点数
n = 0:N-1; % 采样序列
t = 0:1/Fs:1-1/Fs; % 时间序列
f = n * Fs / N; % 真实的频率

x = 1.5*cos(2*pi*50*t+pi/3)+ cos(2*pi*90*t+pi/3); % 原始信号
y = fft(x, N); % 对原始信号做FFT变换
Mag = abs(y); % 求FFT转换结果的模值
subplot(3, 1, 1);
plot(f, Mag); % 绘制幅频相应曲线
title('幅频相应');
xlabel('频率/Hz');
ylabel('幅度');

subplot(3, 1, 2);
plot(f, angle(y)*180/pi); % 绘制相频响应曲线，注意这将弧度转换成了角度
title('相频响应方式一');
xlabel('频率/Hz');
ylabel('相角');

subplot(3, 1, 3);
plot(f, angle(y)*180/pi.* (Mag>=100)); % 绘制相频响应曲线，注意这将弧度转换成了角度
title('相频响应方式二');
xlabel('频率/Hz');
ylabel('相角');
```

上面代码中 $\text{Mag} >= 100$ 是关键，仅展示FFT后幅值大于100的相角。下面再来看Matlab的效果：



可以看到已经完全没问题了，求出了频率50Hz的余弦初相为60°左右，频率90Hz的余弦初相也是60°。

28.3 Matlab 的 IFFT 函数

28.3.1 函数语法

```
y = ifft(X)
y = ifft(X,n)
y = ifft(X,[],dim)
y = ifft(X,n,dim)
y = ifft(..., 'symmetric')
y = ifft(..., 'nonsymmetric')
```

28.3.2 函数描述

y = ifft(X)

此函数用于返回向量 X 的离散傅立叶变换 (DFT) 逆变换结果，计算时使用快速傅里叶算法 (Fast Fourier transform (FFT))。

y = ifft(X,n)

此函数用于返回 n 点的 IDFT。

y = ifft(X,[],dim)

y = ifft(X,n,dim)

上面两个函数用于实现指定维度的 IFFT 运算。

28.3.3 IFFT 实例

下面我们对信号： $0.7\sin(2\pi \cdot 50 \cdot t) + \sin(2\pi \cdot 120 \cdot t)$ 求 FFT 和 IFFT，并绘制原始信号和转换后的信号。Matlab 上运行的代码如下：

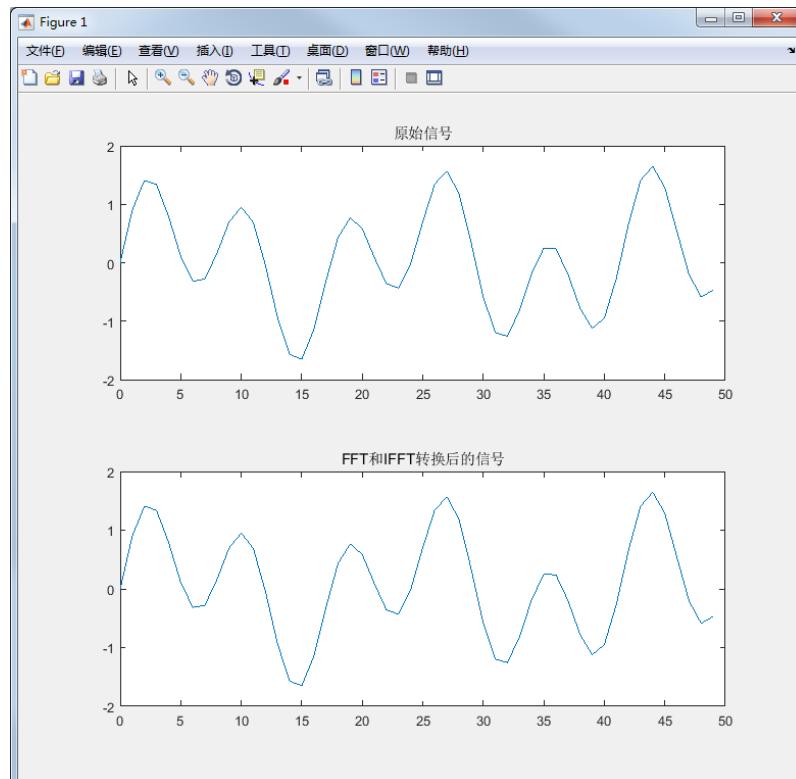
```
Fs = 1000; %采样率
T = 1/Fs; % 采样时间
L = 1024; % 信号长度
t = (0:L-1)*T; % 时间序列

y = 0.7*sin(2*pi*50*t) + sin(2*pi*120*t); %50Hz正弦波和120Hz的正弦波的叠加

subplot(2, 1, 1);
plot(Fs*t(1:50), y(1:50)); %绘制原始信号
title('原始信号');

Y = fft(y, L); %分别调用正变换和逆变换
Z = ifft(Y);
subplot(2, 1, 2);
plot(Fs*t(1:50), Z(1:50)); %绘制逆变换后的波形
title('FFT和IFFT转换后的信号');
```

运行后求出的结果如下：





通过上面的运行结果可以看出，转换后的波形与原始的波形基本是一样的。

28.4 Matlab 的 FFTSHIFT 函数

fftshift 的作用正是让正半轴部分和负半轴部分的图像分别关于各自的中心对称。因为直接用 fft 得出的数据与频率不是对应的，fftshift 可以纠正过来

以下是 Matlab 的帮助文件中对 fftshift 的说明：

$Y = \text{fftshift}(X)$ rearranges the outputs of fft, fft2, and fftn by moving the zero-frequency component to the center of the array. It is useful for visualizing a Fourier transform with the zero-frequency component in the middle of the spectrum. For vectors, $\text{fftshift}(X)$ swaps the left and right halves of X .

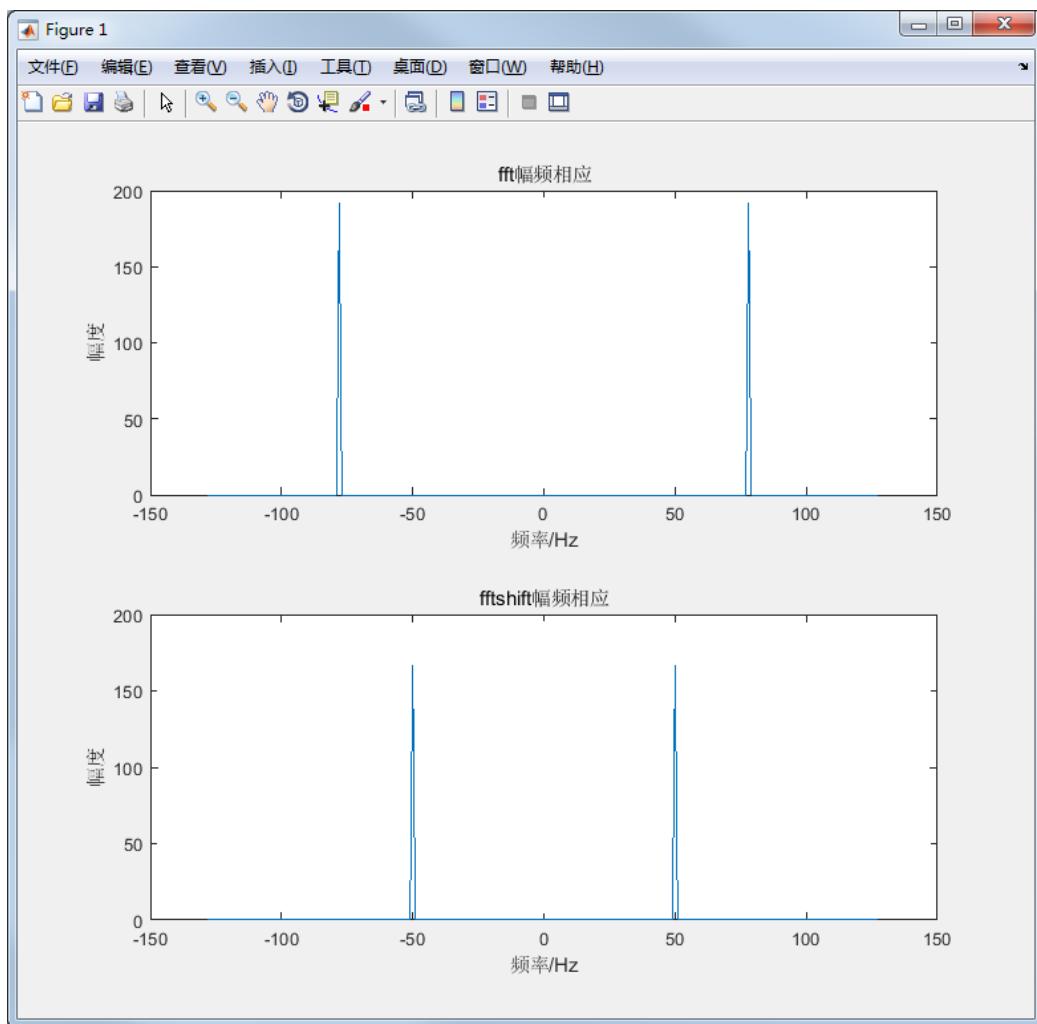
下面我们在 Matlab 上面实现一个如下的代码来说明 fftshift 的使用：

```
Fs = 256;          % 采样率
N = 256;          % 采样点数
n = 0:N-1;        % 采样序列
t = 0:1/Fs:1-1/Fs; % 时间序列
f = (-N/2:N/2-1) * Fs / N; % 真实的频率

x = 1.5*sin(2*pi*50*t+pi/3); % 原始信号
y = fft(x, N);           % 对原始信号做FFT变换
Mag = abs(y);            % 求FFT转换结果的模值
subplot(2, 1, 1);
plot(f, Mag);            % 绘制幅频相应曲线
title('fft幅频相应');
xlabel('频率/Hz');
ylabel('幅度');

z = fftshift(y);          % 对FFT转换后的结果做偏移。
subplot(2, 1, 2);
plot(f, z);              % 绘制幅频相应曲线
title('fftshift幅频相应');
xlabel('频率/Hz');
ylabel('幅度');
```

Matlab 的运行结果如下：



通过上面的运行结果我们可以看到，经过 `fftshift` 的调节后，正弦波的中心频率正好对应在了相应的 50Hz 频率点。使用 `fftshift` 还有很多其它的好处，有兴趣的可以查找相关的资料进行了解。

28.5 总结

本章节主要讲解了 `fft`, `ifft` 和 `fftshift` 的基本用法，如果要深入了解，一定要多练习，多查资料和翻阅相关书籍。



第29章 STM32H7 汇编定点 FFT 库 (64 点, 256 点和 1024 点)

本章主要讲解 ST 官方汇编 FFT 库的应用，包括 1024 点，256 点和 64 点 FFT 的实现。

29.1 汇编 FFT 库说明

29.2 函数 cr4_fft_1024_stm32 的使用(含幅频和相频响应)

29.3 函数 cr4_fft_256_stm32 的使用

29.4 函数 cr4_fft_64_stm32 的使用

29.5 实验例程说明(MDK)

29.6 实验例程说明(IAR)

29.7 总结。

29.1 汇编 FFT 库说明

29.1.1 描述

这个汇编的FFT库是来自STM32F10x DSP library，由于是汇编实现的，而且是基4算法，所以实现FFT在速度上比较快。

如果x[N]是采样信号的话，使用FFT时必须满足如下两条：

- N得满足 4^n ($n = 1, 2, 3\dots$)，也就是以4为基数。
- 采样信号必须是32位数据，高16位存实部，低16位存虚部（这个是针对大端模式），小端模式是高位存虚部，低位存虚部。一般常用的是小端模式。

汇编FFT的实现主要包括以下三个函数：

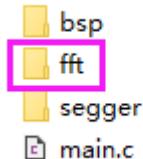
1. cr4_fft_64_stm32：实现64点FFT。
2. cr4_fft_256_stm32：实现256点FFT。
3. cr4_fft_1024_stm32：实现1024点FFT。

29.1.2 汇编库的移植

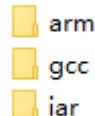
注：这里以MDK为例进行说明，IAR是一样的。



这个汇编库的移植比较简单，从本章配套例子User文件夹复制fft文件夹到自己的工程：



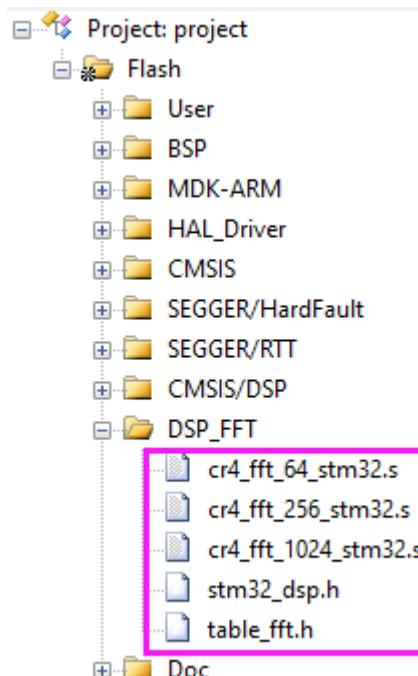
注意路径\User\fft\src\asm下有三个文件夹，分布是arm, gcc和iar，其中arm可用于MDK, gcc可用于Embedded Studio, iar可用于IAR FOR ARM。



三个文件夹里面都是如下几个文件，只是用于不用的编译器：

- cr4_fft_64_stm32.s
- cr4_fft_256_stm32.s
- cr4_fft_1024_stm32.s
- fir_stm32.s
- iirarma_stm32.s
- PID_stm32.s

然后把FFT源文件的三个FFT汇编文件和两个头文件添加上即可，添加后效果如下([注意不同编译器添相应汇编文件](#))：

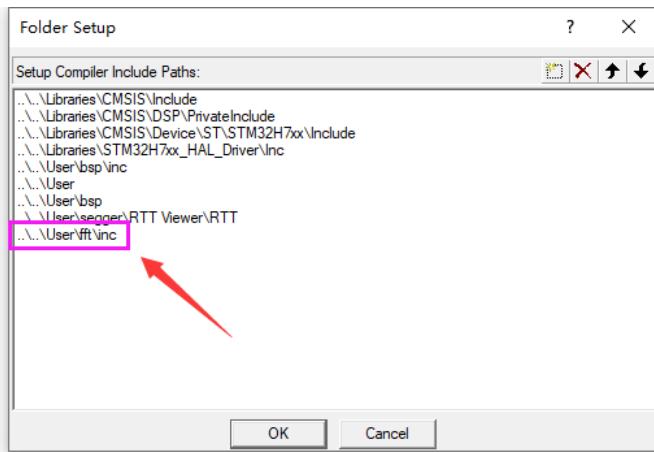


相应文件添加后还有最重要一条，要把stm32_DSP.h文件中的STM32H7的头文件：



```
22 /* Define to prevent recursive inclusion */
23 ifndef __STM32F10x_DSP_H
24 define __STM32F10x_DSP_H
25
26 /* Includes */
27 #include "stm32h7xx.h"
28
29
```

最后别忘了添加路径：



经过上面的操作，汇编FFT库的移植就完成了。

29.2 函数 cr4_fft_1024_stm32 的使用(含幅频和相频响应)

cr4_fft_1024_stm32用于实现1024点数据的FFT计算。下面通过在开发板上运行这个函数并计算幅频相应，然后再与Matlab计算的结果做对比。

```
uint32_t input[1024], output[1024], Mag[1024]; /* 输入，输出和幅值 */
float32_t Phase[1024]; /* 相位 */

/*
*****
* 函数名: PowerMag
* 功能说明: 求模值
* 形参: _usFFTPoints FFT点数
* 返回值: 无
*****
*/
void PowerMag(uint16_t _usFFTPoints)
{
    int16_t lX, lY;
    uint16_t i;
    float32_t mag;

    /* 计算幅值 */
    for (i=0; i < _usFFTPoints; i++)
    {
        lX= (output[i]<<16);           /* 实部*/
        lY= (output[i]>> 16);          /* 虚部 */
        arm_sqrt_f32((float32_t)(lX*lX+ lY*lY), &mag); /* 求模 */
        Mag[i]= mag*2;                  /* 求模后乘以2才是实际模值，直流分量不需要乘2 */
    }

    /* 由于上面多乘了2，所以这里直流分量要除以2 */
    Mag[0] = Mag[0]>>1;
}
```



```
/*
*****
* 函数名: Power_Phase_Radians
* 功能说明: 求相位
* 形参: _usFFTPoints FFT点数, uiCmpValue 阈值
* 返回值: 无
*****
*/
void Power_Phase_Radians(uint16_t _usFFTPoints, uint32_t _uiCmpValue)
{
    int16_t lX, lY;
    uint16_t i;
    float32_t phase;
    float32_t mag;

    for (i=0; i < _usFFTPoints; i++)
    {
        lX= (output[i]<<16)>>16; /* 实部 */
        lY= (output[i] >> 16); /* 虚部 */

        phase = atan2(lY, lX); /* atan2求解的结果范围是(-pi, pi], 弧度制 */
        arm_sqrt_f32((float32_t)(lX*lX+ lY*lY), &mag); /* 求模 */

        if(_uiCmpValue > mag)
        {
            Phase[i] = 0;
        }
        else
        {
            Phase[i] = phase* 180.0f/3.1415926f; /* 将求解的结果由弧度转换为角度 */
        }
    }
}

/*
*****
* 函数名: DSP_FFTPhase
* 功能说明: 1024点FFT的相位求解
* 形参: 无
* 返回值: 无
*****
*/
void DSP_FFTPhase(void)
{
    uint16_t i;

    /* 获得1024个采样点 */
    for (i = 0; i < 1024; i++)
    {
        /* 波形是由直流分量, 50Hz正弦波组成, 波形采样率1024 */
        input[i] = 1024 + 1024*cos(2*3.1415926f*50*i/1024 + 3.1415926f/3);
    }

    /* 计算1024点FFT
       output: 输出结果, 高16位是虚部, 低16位是实部。
       input : 输入数据, 高16位是虚部, 低16位是实部。
       第三个参数必须是1024。
    */
    cr4_fft_1024_stm32(output, input, 1024);

    /* 求幅值 */
    PowerMag(1024);

    /* 打印输出结果 */
    for (i = 0; i < 1024; i++)
    {
        printf("%d\r\n", Mag[i]);
    }
}
```



```
printf("=====\\r\\n");

/* 求相频 */
Power_Phase_Radians(1024, 100);

/* 打印输出结果 */
for (i = 0; i < 1024; i++)
{
    printf("%f\\r\\n", Phase[i]);
}
```

运行函数DSP_FFTPhase可以通过串口打印出计算的模值和相角，下面我们就通过Matlab计算的模值和相角跟cr4_fft_1024_stm32计算的做对比。

对比前需要先将串口打印出的数据加载到 Matlab 中，并给这个数组起名 sampledata，加载方法在前面的教程的[第 13 章 13.6 小结已经讲解](#)，这里不做赘述了。Matlab 中运行的代码如下：

```
Fs = 1024; % 采样率
N = 1024; % 采样点数
n = 0:N-1; % 采样序列
t = 0:1/Fs:1-1/Fs; % 时间序列
f = n * Fs / N; % 真实的频率

% 波形是由直流分量, 50Hz正弦波正弦波组成
x = 1024 + 1024*cos(2*pi*50*t + pi/3);
y = fft(x, N); % 对原始信号做FFT变换
Mag = abs(y);

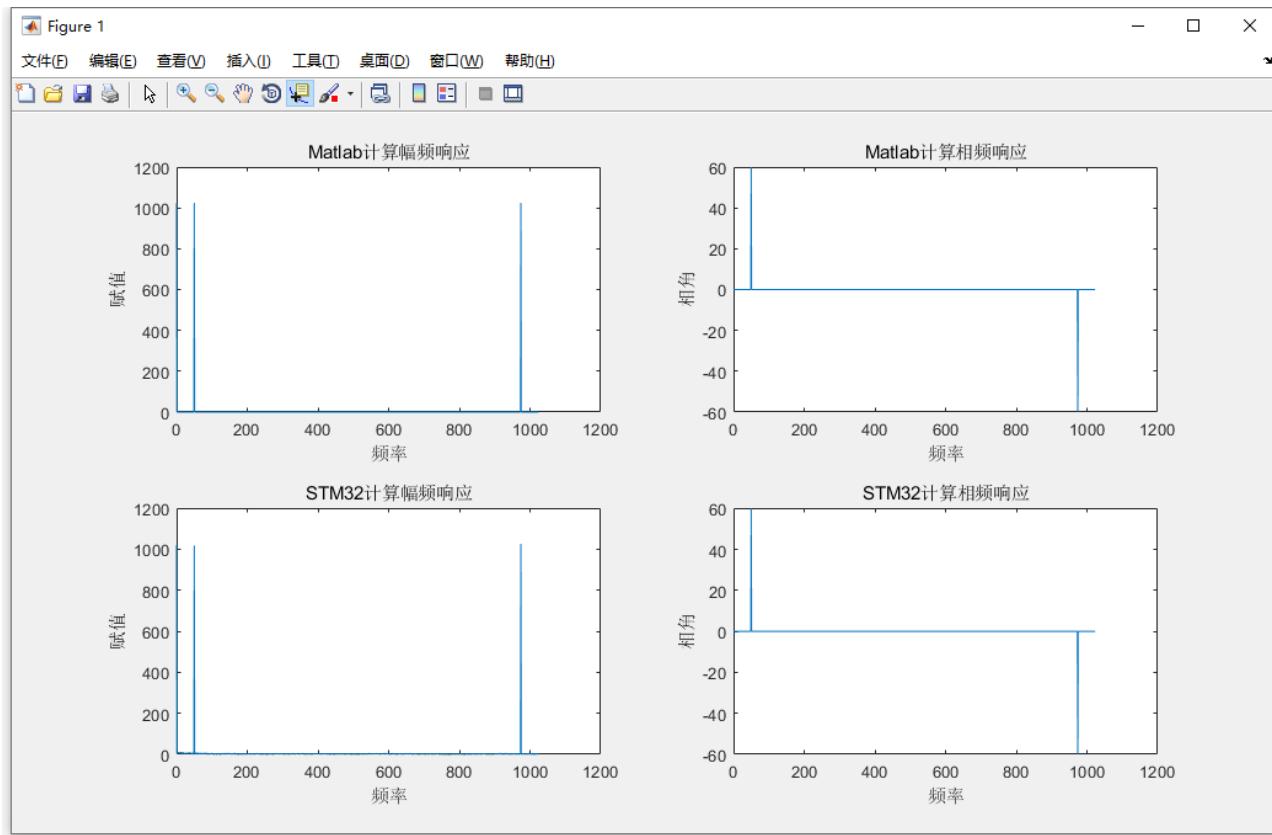
subplot(2, 2, 1);
MagAct = Mag *2 / N;
MagAct(1) = MagAct(1)/2;
plot(f, MagAct);
title('Matlab计算幅频响应');
xlabel('频率');
ylabel('赋值');

subplot(2, 2, 2);
realvalue = real(y);
imagvalue = imag(y);
plot(f, atan2(imagvalue, realvalue)*180/pi.*((Mag>=1024*20)));
title('Matlab计算相频响应');
xlabel('频率');
ylabel('相角');

subplot(2, 2, 3);
plot(f, sampledata1); % 绘制STM32计算的幅频相应
title('STM32计算幅频响应');
xlabel('频率');
ylabel('赋值');

subplot(2, 2, 4);
plot(f, sampledata2); % 绘制STM32计算的相频相应
title('STM32计算相频响应');
xlabel('频率');
ylabel('相角');
```

运行 Matlab 后的输出结果如下：



从上面的对比结果中可以看出，Matlab 和函数 cr4_fft_1024_stm32 计算的结果基本是一致的。幅频响应求出的幅值和相频响应中的求出的初始相角都是没问题的。

29.3 函数 cr4_fft_256_stm32 的使用

cr4_fft_256_stm32 和 cr4_fft_1024_stm32 的用法是一样的，下面通过一个实例进行说明：

```
/*
*****
* 函数名: DSP_FFT256
* 功能说明: 256点FFT实现
* 形参: 无
* 返回值: 无
*****
*/
void DSP_FFT256(void)
{
    uint16_t i;

    /* 获得256个采样点 */
    for (i = 0; i < 256; i++)
    {
        /* 波形是由直流分量, 50Hz正弦波和20Hz正弦波组成, 波形采样率200Hz */
        input[i] = 1024 + 1024*sin(2*3.1415926f*50*i/200) + 512*sin(2*3.1415926f*20*i/200) ;
    }

    /* 计算256点FFT
       output: 输出结果, 高16位是虚部, 低16位是实部。
       input : 输入数据, 高16位是虚部, 低16位是实部。
       第三个参数必须是1024。
    */
    cr4_fft_256_stm32(output, input, 256);
}
```



```
/* 求幅值 */
PowerMag(256);

/* 打印输出结果 */
for (i = 0; i < 256; i++)
{
    printf("%d\r\n", Mag[i]);
}
```

运行函数DSP_FFT256可以通过串口打印出计算的模值，下面我们就通过Matlab计算的模值跟cr4_fft_256_stm32计算的模值做对比。

对比前需要先将串口打印出的数据加载到Matlab中，并给这个数组起名sampledata，Matlab中

运行的代码如下：

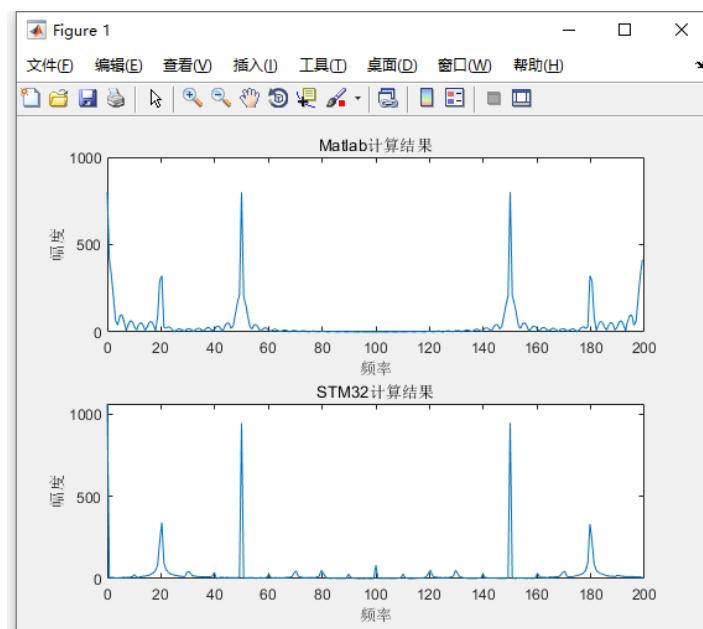
```
Fs = 200; %采样率
N = 256; %采样点数
n = 0:N-1; %采样序列
t = 0:1/Fs:1-1/Fs; %时间序列
f = n * Fs / N; %真实的频率

%波形是由直流分量, 50Hz正弦波和20Hz正弦波组成
x = 1024 + 1024*sin(2*pi*50*t) + 512*sin(2*pi*20*t); %
y = fft(x, N); %对原始信号做FFT变换

subplot(2, 1, 1);
Mag = abs(y);
MagAct = Mag *2 / N;
MagAct(1) = MagAct(1)/2;
plot(f, MagAct); %绘制幅频相应曲线
title('Matlab计算结果');
xlabel('频率');
ylabel('幅度');

subplot(2, 1, 2);
plot(f, sampledata); %绘制STM32计算的幅频相应
title('STM32计算结果');
xlabel('频率');
ylabel('幅度');
```

运行Matlab后的输出结果如下：





从上面的对比结果中可以看出，Matlab 和函数 cr4_fft_256_stm32 计算的结果基本是一致的，但频率泄露略多。

29.4 函数 cr4_fft_64_stm32 的使用

cr4_fft_64_stm32 和 cr4_fft_1024_stm32 的用法也是一样的，下面通过一个实例进行说明：

```
/*
***** DSP_FFT64 *****
* 函数名: DSP_FFT64
* 功能说明: 64点FFT实现
* 形参: 无
* 返回值: 无
*****
*/
void DSP_FFT64(void)
{
    uint16_t i;

    /* 获得64个采样点 */
    for (i = 0; i < 64; i++)
    {
        /* 波形是由直流分量, 5Hz正弦波和10Hz正弦波组成, 波形采样率60Hz */
        input[i] = 1024 + 1024*sin(2*3.1415926f*5*i/60) + 512*sin(2*3.1415926f*10*i/60) ;
    }

    /* 计算64点FFT
       output: 输出结果, 高16位是虚部, 低16位是实部。
       input : 输入数据, 高16位是虚部, 低16位是实部。
       第三个参数必须是1024。
    */
    cr4_fft_64_stm32(output, input, 64);

    /* 求幅值 */
    PowerMag(64);

    /* 打印输出结果 */
    for (i = 0; i < 64; i++)
    {
        printf("%d\r\n", Mag[i]);
    }
}
```

运行函数 DSP_FFT64 可以通过串口打印出计算的模值，下面我们就通过 Matlab 计算的模值跟 cr4_fft_64_stm32 计算的模值做对比。

对比前需要先将串口打印出的数据加载到 Matlab 中，并给这个数组起名 sampledata，Matlab 中运行的代码如下：

```
Fs = 60;          % 采样率
N = 64;          % 采样点数
n = 0:N-1;        % 采样序列
t = 0:1/Fs:1-1/Fs; % 时间序列
f = n * Fs / N;  % 真实的频率

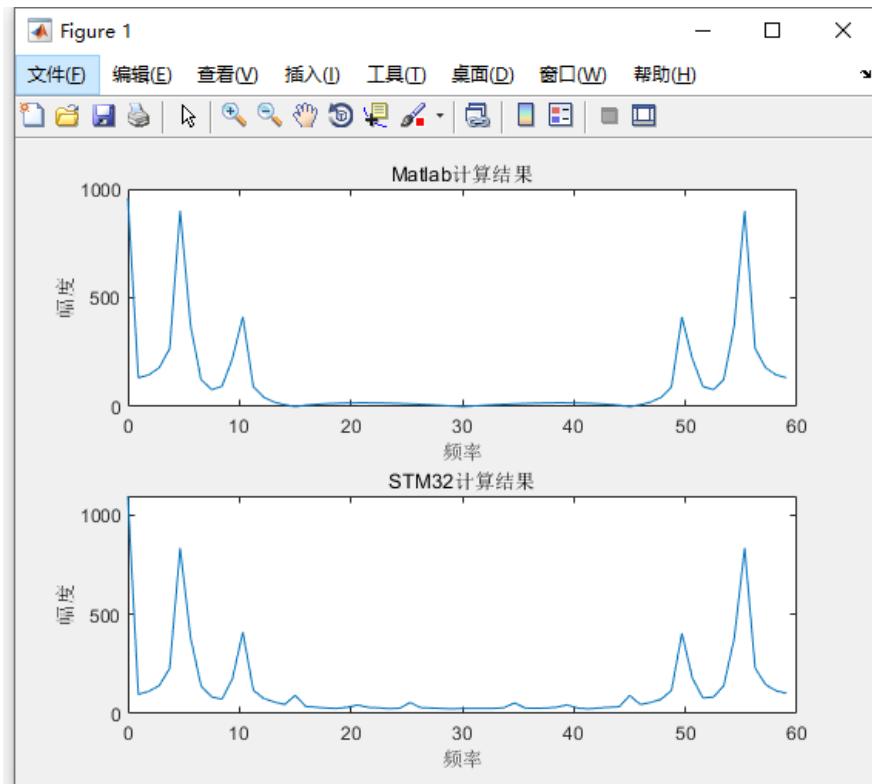
% 波形是由直流分量, 5Hz正弦波和10Hz正弦波组成
x = 1024 + 1024*sin(2*pi*5*t) + 512*sin(2*pi*10*t) ;
y = fft(x, N);      % 对原始信号做FFT变换

subplot(2, 1, 1);
Mag = abs(y);
MagAct = Mag *2 / N;
MagAct(1) = MagAct(1)/2;
plot(f, MagAct);    % 绘制幅频相应曲线
```

```
title('Matlab计算结果');
xlabel('频率');
ylabel('幅度');

subplot(2, 1, 2);
plot(f, sampledata); %绘制STM32计算的幅频相应
title('STM32计算结果');
xlabel('频率');
ylabel('幅度');
```

运行 Matlab 后的输出结果如下：



从上面的对比结果中可以看出，Matlab 和函数 cr4_fft_64_stm32 计算的结果基本是一致的，但是计算的效果都比较差，主要是因为采样点数太少。

29.5 实验例程说明 (MDK)

配套例子：

V7-219_STM32H7 移植 ST 汇编定点 FFT 库(64 点, 256 点和 1024 点)

实验目的：

1. 学习 ST 汇编定点 FFT 库 (64 点, 256 点和 1024 点)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打印 1024 点 FFT 的幅频响应和相频响应。
3. 按下按键 K2，串口打印 256 点 FFT 的幅频响应。
4. 按下按键 K3，串口打印 64 点 FFT 的幅频响应。

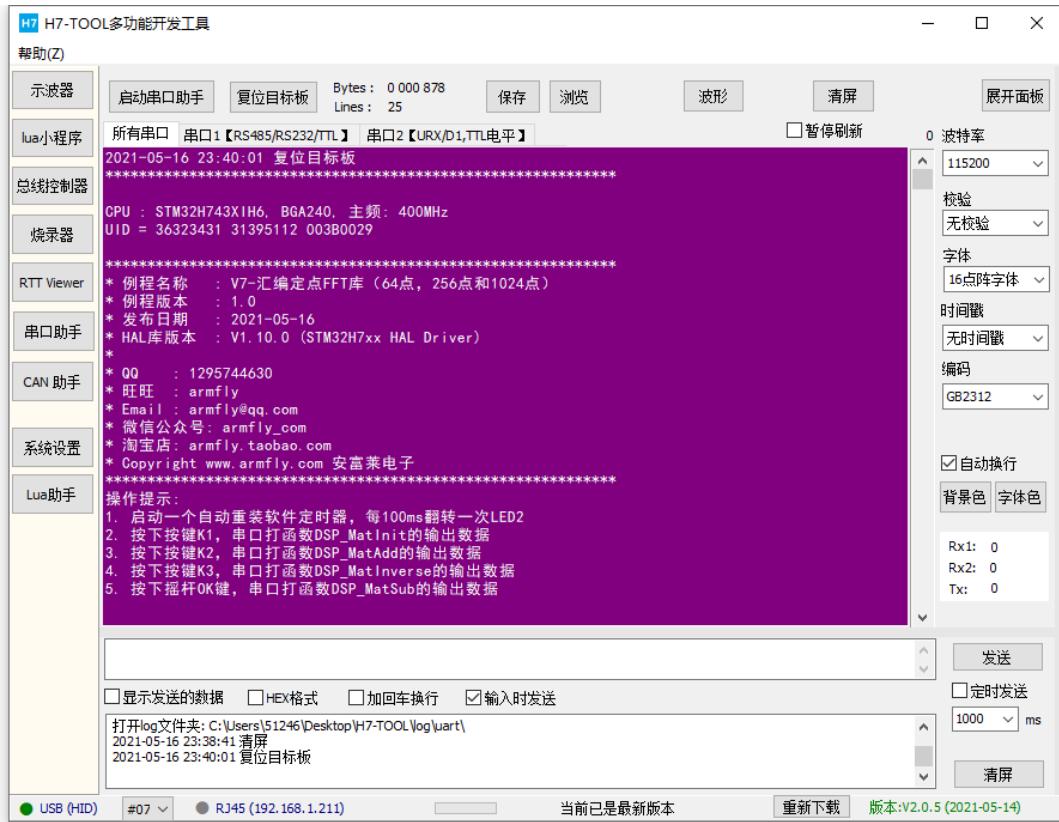


使用 AC6 注意事项

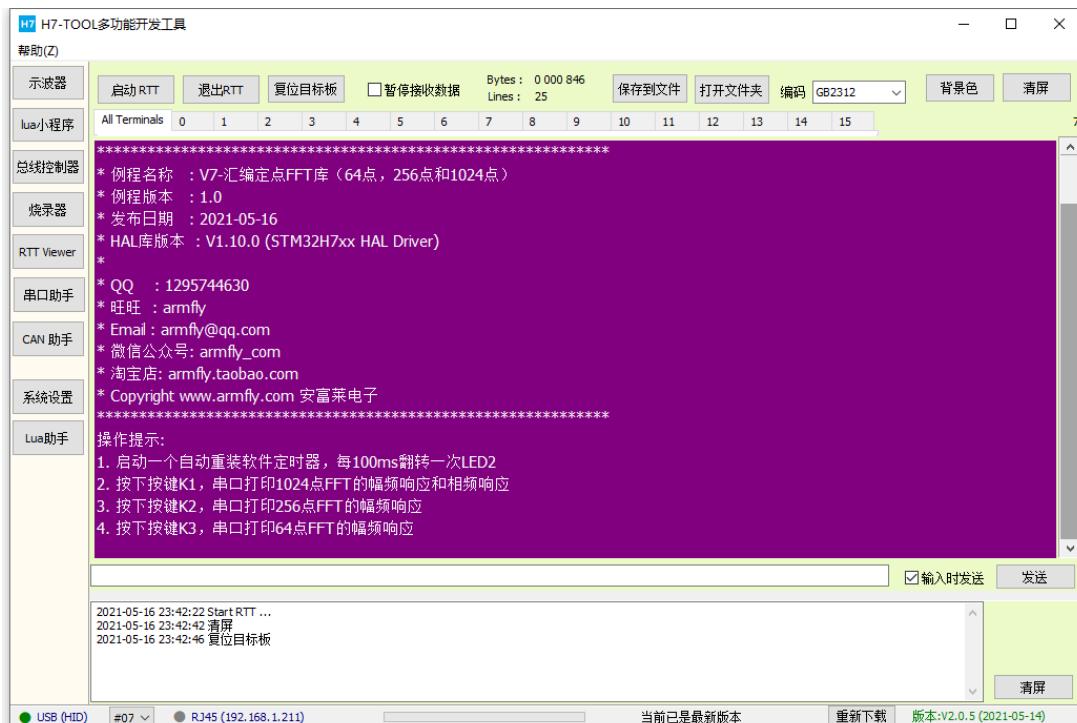
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

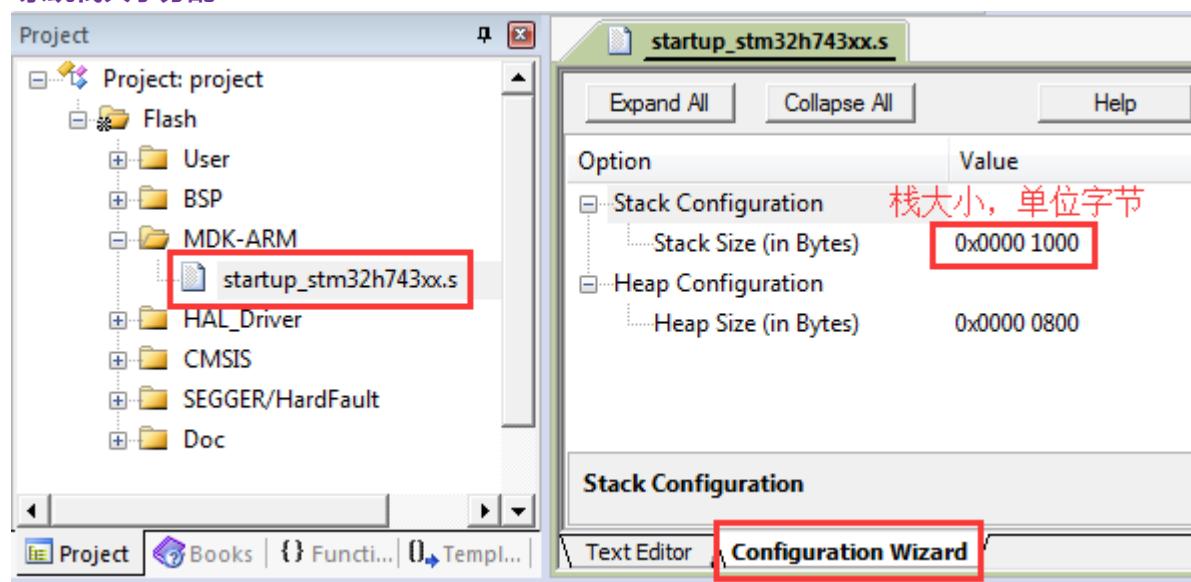


RTT 方式打印信息：

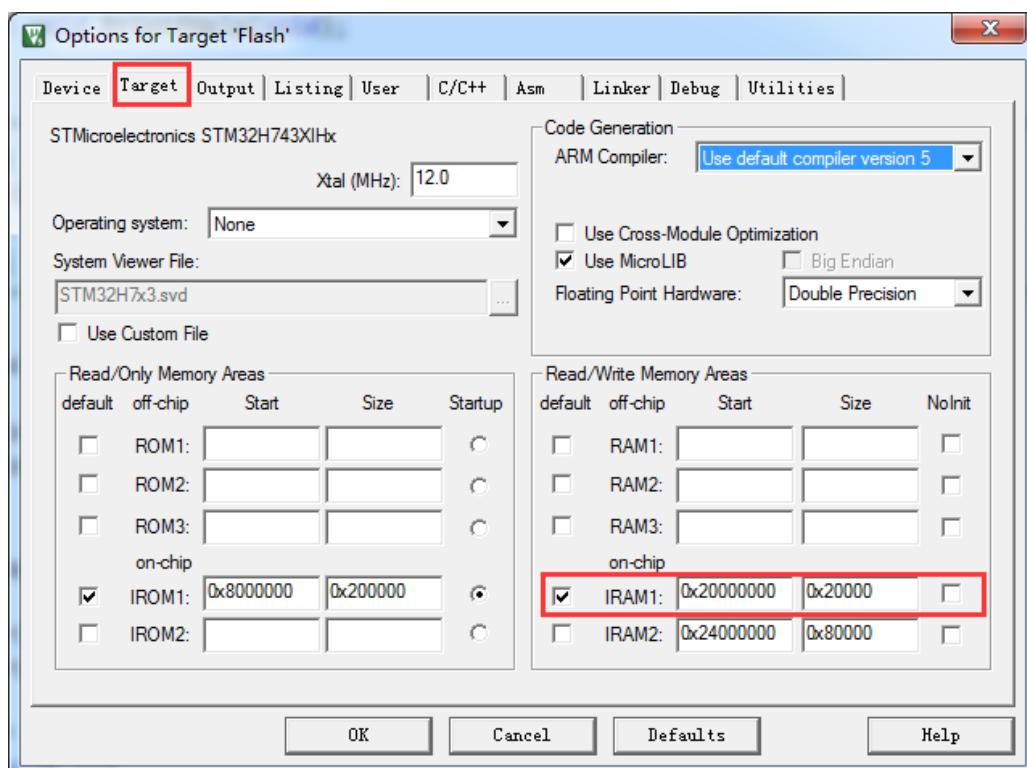


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
*****
```



```
/* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
        - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
        - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;
```



```
/* 禁止 MPU */
HAL_MPU_Disable();

/* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:



主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印 1024 点 FFT 的幅频响应和相频响应。
- 按下按键 K2，串口打印 256 点 FFT 的幅频响应。
- 按下按键 K3，串口打印 64 点 FFT 的幅频响应。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
***** */

int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 500ms 进来一次 */
            bsp_LedToggle(4); /* 翻转 LED2 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    DSP_FFTPhase();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下 */
                    DSP_FFT256();
                    break;

                case KEY_DOWN_K3:           /* K3 键按下 */
                    DSP_FFT64();
                    break;
            }
        }
    }
}
```



```
default:  
    /* 其它的键值不处理 */  
    break;  
}  
}  
}
```

29.6 实验例程说明 (IAR)

配套例子：

V7-219_STM32H7 移植 ST 汇编定点 FFT 库(64 点, 256 点和 1024 点)

实验目的：

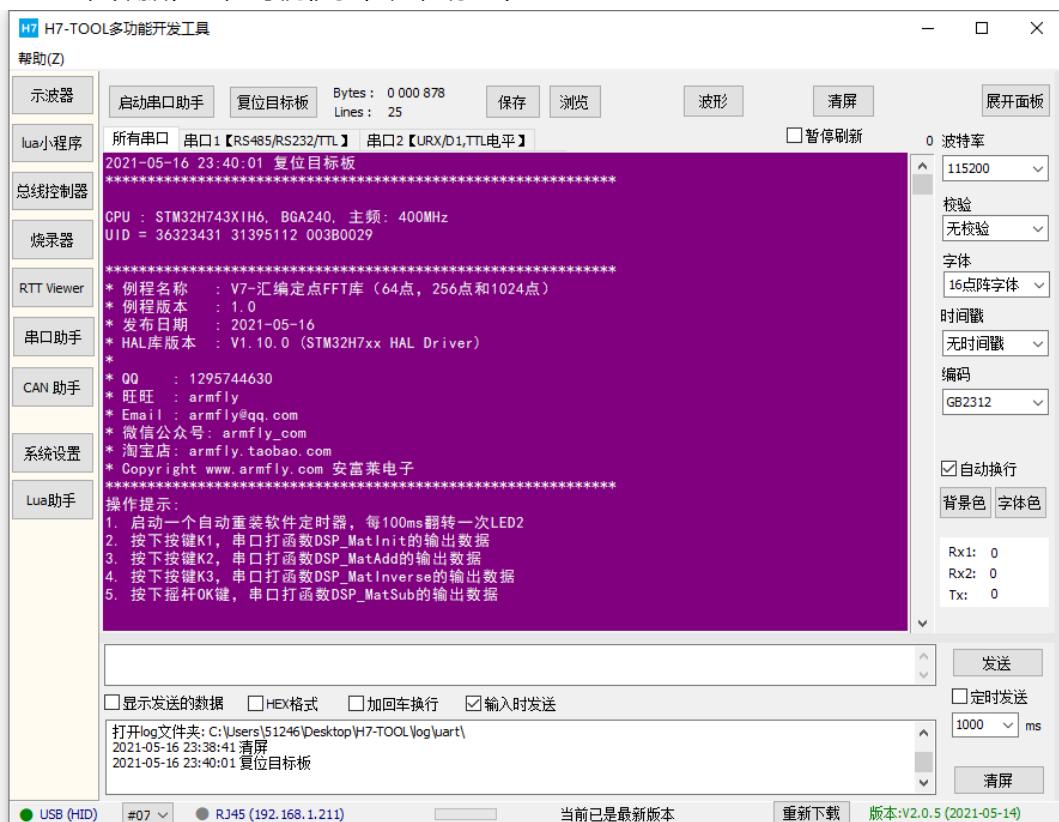
1. 学习 ST 汇编定点 FFT 库 (64 点, 256 点和 1024 点)

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打印 1024 点 FFT 的幅频响应和相频响应。
3. 按下按键 K2，串口打印 256 点 FFT 的幅频响应。
4. 按下按键 K3，串口打印 64 点 FFT 的幅频响应。

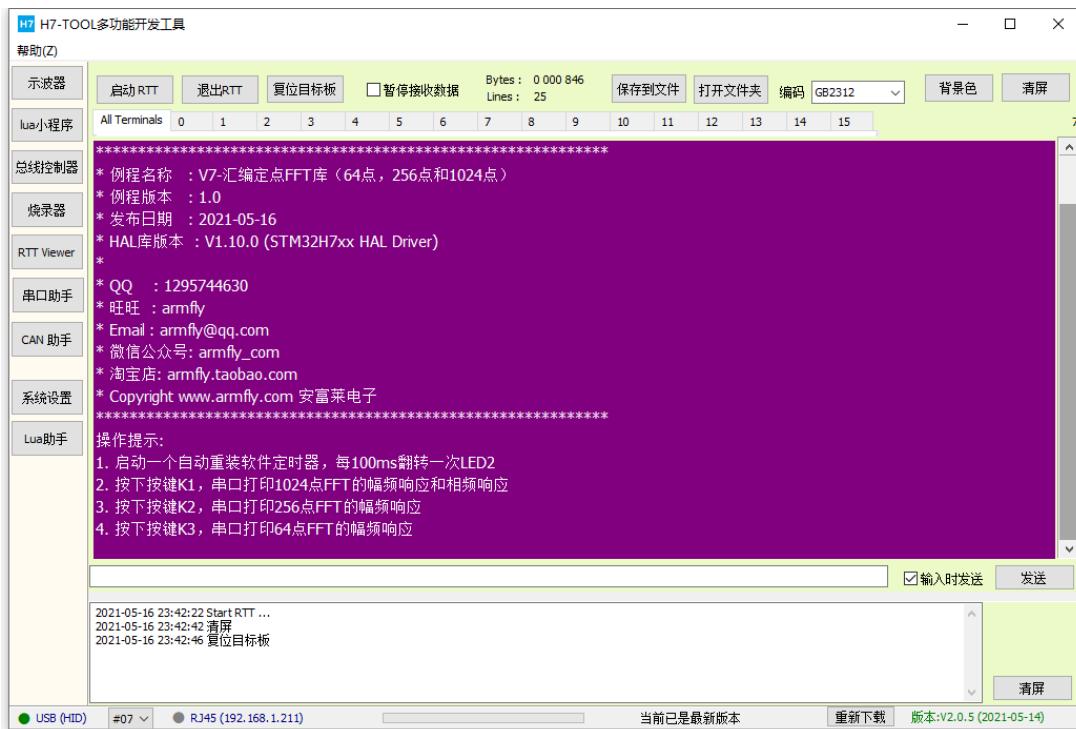
上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。



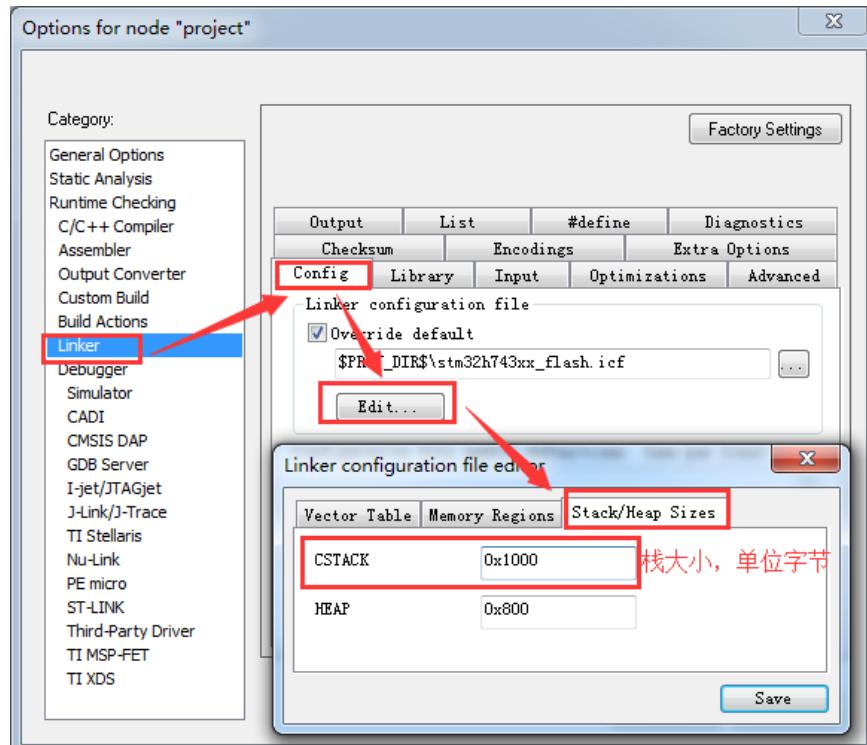


RTT 方式打印信息：

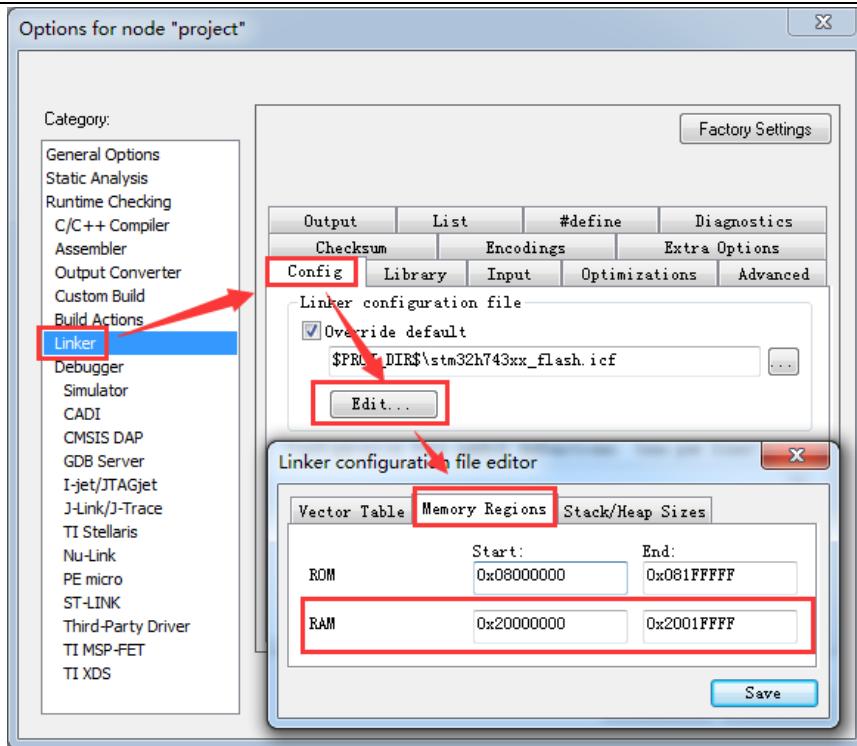


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM：



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印 1024 点 FFT 的幅频响应和相频响应。
- 按下按键 K2，串口打印 256 点 FFT 的幅频响应。
- 按下按键 K3，串口打印 64 点 FFT 的幅频响应。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */\n\n/* 进入主程序循环体 */\nwhile (1)\n{\n    bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */\n\n    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */\n    {\n        /* 每隔 500ms 进来一次 */\n        bsp_LedToggle(4); /* 翻转 LED2 的状态 */\n    }\n\n    ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */\n    if (ucKeyCode != KEY_NONE)\n    {\n        switch (ucKeyCode)\n        {\n            case KEY_DOWN_K1:           /* K1 键按下 */\n                DSP_FFTPhase();\n                break;\n\n            case KEY_DOWN_K2:           /* K2 键按下 */\n                DSP_FFT256();\n                break;\n\n            case KEY_DOWN_K3:           /* K3 键按下 */\n                DSP_FFT64();\n                break;\n\n            default:\n                /* 其它的键值不处理 */\n                break;\n        }\n    }\n}\n}
```

29.7 总结

本章节主要讲解了汇编 FFT 的 1024 点, 256 点和 64 点使用方法, 有兴趣的可以深入了解汇编代码的实现。



第30章 STM32H7 复数浮点 FFT (支持单精度)

和双精度)

本章主要讲解复数浮点 FFT，支持单精度和双精度。

30.1 初学者重要提示

30.2 复数浮点 FFT 说明

30.3 单精度函数 arm_cfft_f32 的使用 (含幅频和相频)

30.4 双精度函数 arm_cfft_f64 的使用 (含幅频和相频)

30.5 实验例程说明(MDK)

30.6 实验例程说明(IAR)

30.7 总结

30.1 初学者重要提示

- ◆ 新版 DSP 库浮点 FFT 推荐使用混合基函数 arm_cfft_f32，而基 2 函数 arm_cfft_radix2_f32 和基 4 函数 arm_cfft_radix4_f32 将废弃。ARM 说明如下：

Earlier releases of the library provided separate radix-2 and radix-4 algorithms that operated on floating-point data. These functions are still provided but are deprecated. The older functions are slower and less general than the new functions.

DSP 库的早期发行版提供了单独的 radix-2 和 radix-4 对浮点数据进行运算的算法。这些功能仍然提供，但已弃用。相比新版函数，老版的功能较慢且通用性较低

30.2 复数浮点 FFT 说明

30.2.1 功能描述

当前复数 FFT 函数支持三种数据类型，分别是浮点，定点 Q31 和 Q15。这些 FFT 函数有一个共同的特点，就是用于输入信号的缓冲，在转化结束后用来存储输出结果。这样做的好处是节省了 RAM 空间，不需要为输入和输出结果分别设置缓存。由于是复数 FFT，所以输入和输出缓存要存储实部和虚部。存储顺序如下：{real[0], imag[0], real[1], imag[1],……}，在使用中切记不要搞错。



30.2.2 浮点 FFT

浮点复数 FFT 使用了一个混合基数算法，通过多个基 8 与单个基 2 或基 4 算法实现。根据需要，该算法支持的长度[16, 32, 64, ..., 4096]和每个长度使用不同的旋转因子表。

浮点复数 FFT 使用了标准的 FFT 定义，FFT 正变换的输出结果会被放大 fftLen 倍数，计算 FFT 逆变换的时候会缩小到 1/fftLen。这样就与教科书中的定义一致了。

定义好的旋转因子和位反转表已经在头文件 arm_const_structs.h 中定义好了，调用浮点 FFT 函数 arm_cfft_f32 时，包含相应的头文件即可。比如：

```
arm_cfft_f32(arm_cfft_sR_f32_len64, pSrc, 1, 1)
```

上式就是计算一个 64 点的 FFT 逆变换包括位反转。数据结构 arm_cfft_sR_f32_len64 可以认为是常数，计算的过程中是不能修改的。同样是这种数据结构还能用于混合基的 FFT 正变换和逆变换。

早期发布的浮点复数 FFT 函数版本包含基 2 和基 4 两种方法实现的，但是不推荐大家再使用。现在全部用 arm_cfft_f32 代替了。

30.3 单精度函数 arm_cfft_f32 的使用(含幅频和相频)

30.3.1 函数说明

函数原型：

```
void arm_cfft_f32(
    const arm_cfft_instance_f32 * S,
    float32_t * p1,
    uint8_t ifftFlag,
    uint8_t bitReverseFlag)
```

函数描述：

这个函数用于单精度浮点复数 FFT。

函数参数：

◆ 第 1 个参数是封装好的浮点 FFT 例化，支持的参数如下：

- arm_cfft_sR_f32_len16, 16 点 FFT
- arm_cfft_sR_f32_len32, 32 点 FFT
- arm_cfft_sR_f32_len64, 64 点 FFT
- arm_cfft_sR_f32_len128, 128 点 FFT
- arm_cfft_sR_f32_len256, 256 点 FFT
- arm_cfft_sR_f32_len512, 512 点 FFT
- arm_cfft_sR_f32_len1024, 1024 点 FFT
- arm_cfft_sR_f32_len2048, 2048 点 FFT



- arm_cfft_sR_f32_len4096, 4096 点 FFT
- ◆ 第 2 个参数是复数地址，存储顺序是实部，虚部，实部，虚部，依次类推。
- ◆ 第 3 个参数用于设置正变换和逆变换，ifftFlag=0 表示正变换，ifftFlag=1 表示逆变换。
- ◆ 第 4 个参数用于设置输出位反转，bitReverseFlag=1 表示使能，bitReverseFlag=0 表示禁止。

30.3.2 使用举例并和 Matlab 比较

下面通过在开发板上运行这个函数并计算幅频相应，然后再与Matlab计算的结果做对比。

```
/*
*****
* 函数名: arm_cfft_f32_app
* 功能说明: 调用函数 arm_cfft_f32 计算幅频和相频
* 形参: 无
* 返回值: 无
*****
*/
static void arm_cfft_f32_app(void)
{
    uint16_t i;

    ifftFlag = 0;
    doBitReverse = 1;

    /* 按照实部, 虚部, 实部, 虚部..... 的顺序存储数据 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 波形是由直流分量, 50Hz 正弦波组成, 波形采样率 1024, 初始相位 60° */
        testInput_f32[i*2] = 1 + cos(2*3.1415926f*50*i/1024 + 3.1415926f/3);
        testInput_f32[i*2+1] = 0;
    }

    /* CFFT 变换 */
    arm_cfft_f32(&arm_cfft_sR_f32_len1024, testInput_f32, ifftFlag, doBitReverse);

    /* 求解模值 */
    arm_cmplx_mag_f32(testInput_f32, testOutput_f32, TEST_LENGTH_SAMPLES);

    printf("=====\\r\\n");

    /* 求相频 */
    PowerPhaseRadians_f32(testInput_f32, Phase_f32, TEST_LENGTH_SAMPLES, 0.5f);

    /* 串口打印求解的模值 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        printf("%f, %f\\r\\n", testOutput_f32[i], Phase_f32[i]);
    }
}
```

运行函数arm_cfft_f32_app可以通过串口打印出计算的模值和相角，下面我们就通过Matlab计算的模值和相角跟arm_cfft_f32计算的做对比。

对比前需要先将串口打印出的数据加载到 Matlab 中，并给这个数组起名 sampledata，加载方法在前面的教程的[第 13 章 13.6 小结已经讲解](#)，这里不做赘述了。Matlab 中运行的代码如下：

```
Fs = 1024;          % 采样率
N = 1024;           % 采样点数
```

```

n = 0:N-1;          % 采样序列
t = 0:1/Fs:1-1/Fs;   % 时间序列
f = n * Fs / N;      % 真实的频率

% 波形是由直流分量, 50Hz正弦波正弦波组成
x = 1 + cos(2*pi*50*t + pi/3); 
y = fft(x, N);           % 对原始信号做FFT变换
Mag = abs(y);

subplot(2, 2, 1);
plot(f, Mag);
title('Matlab计算幅频响应');
xlabel('频率');
ylabel('赋值');

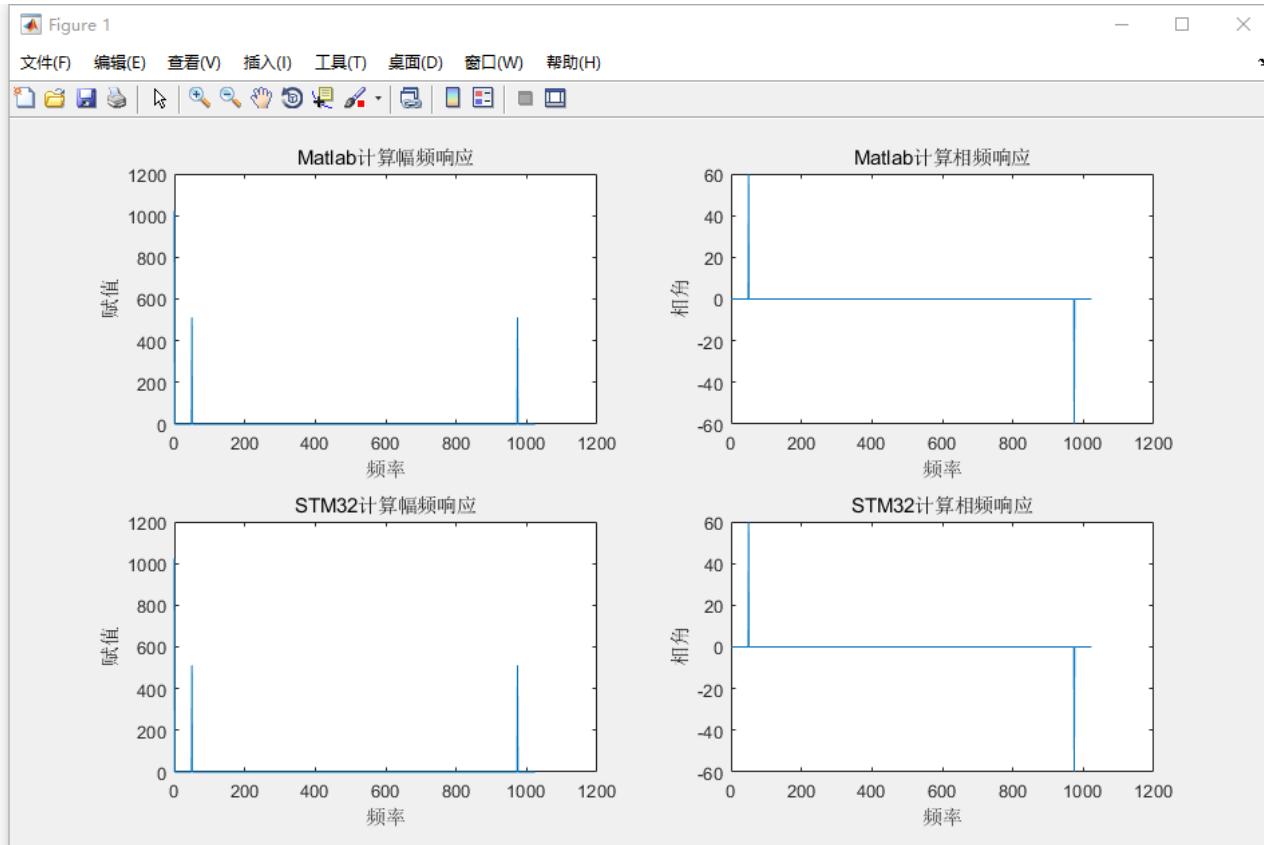
subplot(2, 2, 2);
realvalue = real(y);
imagvalue = imag(y);
plot(f, atan2(imagvalue, realvalue)*180/pi.* (Mag>=200));
title('Matlab计算相频响应');
xlabel('频率');
ylabel('相角');

subplot(2, 2, 3);
plot(f, sampledata1); % 绘制STM32计算的幅频相应
title('STM32计算幅频响应');
xlabel('频率');
ylabel('赋值');

subplot(2, 2, 4);
plot(f, sampledata2); % 绘制STM32计算的相频相应
title('STM32计算相频响应');
xlabel('频率');
ylabel('相角');

```

运行 Matlab 后的输出结果如下：





从上面的对比结果中可以看出，Matlab 和函数 arm_cfft_f32 计算的结果基本是一致的。幅频响应求出的幅值和相频响应中的求出的初始相角都是没问题的。

30.4 双精度函数 arm_cfft_f64 的使用(含幅频和相频)

30.4.1 函数说明

函数原型：

```
void arm_cfft_f64(
    const arm_cfft_instance_f64 * S,
    float64_t * p1,
    uint8_t ifftFlag,
    uint8_t bitReverseFlag)
```

函数描述：

这个函数用于双精度浮点复数 FFT。

函数参数：

- ◆ 第 1 个参数是封装好的浮点 FFT 例化，支持的参数如下：
 - arm_cfft_sR_f64_len16, 16 点 FFT
 - arm_cfft_sR_f64_len32, 32 点 FFT
 - arm_cfft_sR_f64_len64, 64 点 FFT
 - arm_cfft_sR_f64_len128, 128 点 FFT
 - arm_cfft_sR_f64_len256, 256 点 FFT
 - arm_cfft_sR_f64_len512, 512 点 FFT
 - arm_cfft_sR_f64_len1024, 1024 点 FFT
 - arm_cfft_sR_f64_len2048, 2048 点 FFT
 - arm_cfft_sR_f64_len4096, 4096 点 FFT
- ◆ 第 2 个参数是复数地址，存储顺序是实部，虚部，实部，虚部，依次类推。
- ◆ 第 3 个参数用于设置正变换和逆变换，ifftFlag=0 表示正变换，ifftFlag=1 表示逆变换。
- ◆ 第 4 个参数用于设置输出位反转，bitReverseFlag=1 表示使能，bitReverseFlag=0 表示禁止。

30.4.2 使用举例并和 Matlab 比较

下面通过在开发板上运行这个函数并计算幅频相应，然后再与Matlab计算的结果做对比。

```
/*
*****
*  函数名: arm_cfft_f64_app
*  功能说明: 调用函数 arm_cfft_f64 计算幅频和相频
*  形  参: 无
*  返  值: 无
*****
```



```
*****
*/
static void arm_cfft_f64_app(void)
{
    uint16_t i;
    float64_t lX, lY;

    ifftFlag = 0;
    doBitReverse = 1;

    /* 按照实部, 虚部, 实部, 虚部..... 的顺序存储数据 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 波形是由直流分量, 50Hz 正弦波组成, 波形采样率 1024, 初始相位 60° */
        testInput_f64[i*2] = 1 + cos(2*3.1415926*50*i/1024 + 3.1415926/3);
        testInput_f64[i*2+1] = 0;
    }

    /* CFFT 变换 */
    arm_cfft_f64(&arm_cfft_sR_f64_len1024, testInput_f64, ifftFlag, doBitReverse);

    /* 求解模值 */
    for (i = 0; i < TEST_LENGTH_SAMPLES; i++)
    {
        lX = testInput_f64[2*i];           /* 实部*/
        lY = testInput_f64[2*i+1];         /* 虚部 */
        testOutput_f64[i] = sqrt(lX*lX+ lY*lY); /* 求模 */
    }

    printf("=====\\r\\n");

    /* 求相频 */
    PowerPhaseRadians_f64(testInput_f64, Phase_f64, TEST_LENGTH_SAMPLES, 0.5);

    /* 串口打印求解的模值 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        printf("%.11f, %.11f\\r\\n", testOutput_f64[i], Phase_f64[i]);
    }
}
```

运行函数arm_cfft_f64_app可以通过串口打印出计算的模值和相角，下面我们就通过Matlab计算的模值和相角跟arm_cfft_f64计算的做对比。

对比前需要先将串口打印出的数据加载到 Matlab 中，并给这个数组起名 sampledata，加载方法在前面的教程的**第 13 章 13.6 小结已经讲解**，这里不做赘述了。Matlab 中运行的代码如下：

```
Fs = 1024;          % 采样率
N = 1024;          % 采样点数
n = 0:N-1;          % 采样序列
t = 0:1/Fs:1-1/Fs; % 时间序列
f = n * Fs / N;    % 真实的频率

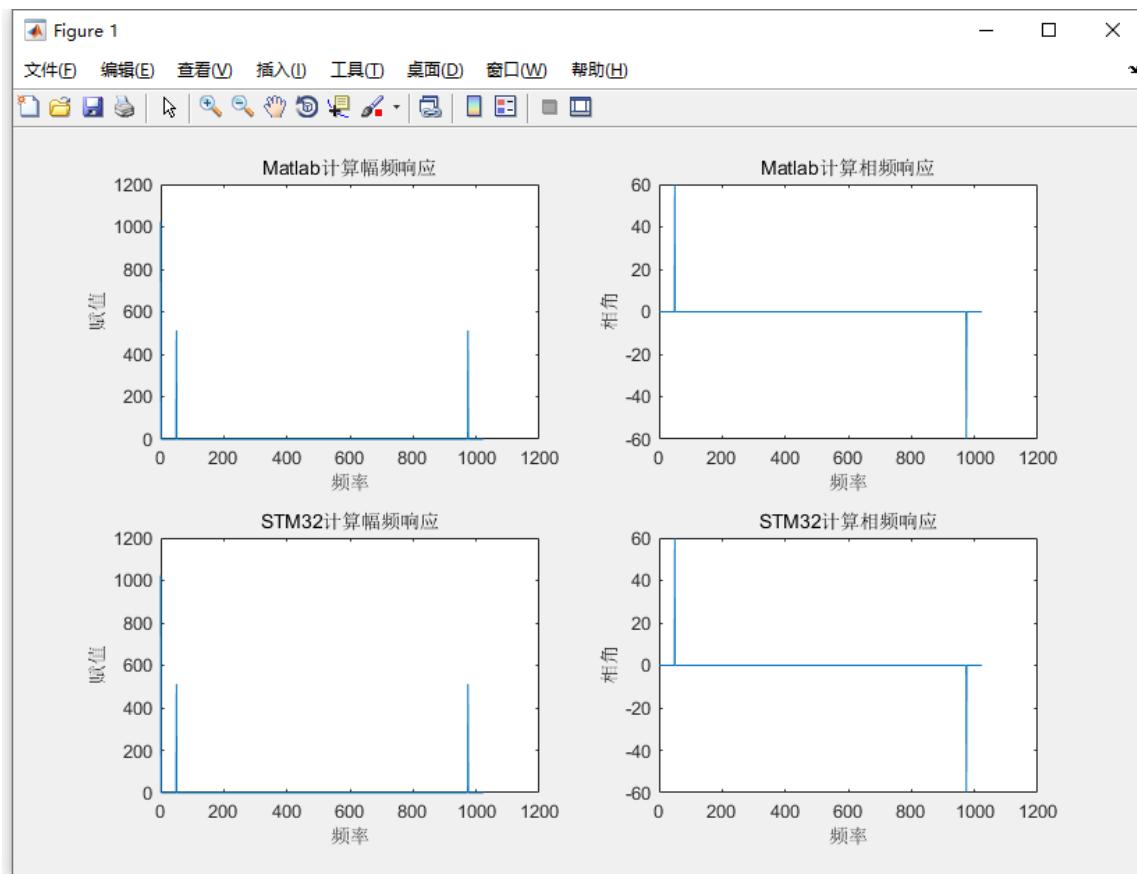
% 波形是由直流分量, 50Hz 正弦波正弦波组成
x = 1 + cos(2*pi*50*t + pi/3);
y = fft(x, N);      % 对原始信号做FFT变换
Mag = abs(y);

subplot(2, 2, 1);
plot(f, Mag);
title('Matlab计算幅频响应');
xlabel('频率');
ylabel('幅值');

subplot(2, 2, 2);
```

```
realvalue = real(y);  
imagvalue = imag(y);  
plot(f, atan2(imagvalue, realvalue)*180/pi.* (Mag>=200));  
title('Matlab计算相频响应');  
xlabel('频率');  
ylabel('相角');  
  
subplot(2, 2, 3);  
plot(f, sampledata1); %绘制STM32计算的幅频相应  
title('STM32计算幅频响应');  
xlabel('频率');  
ylabel('赋值');  
  
subplot(2, 2, 4);  
plot(f, sampledata2); %绘制STM32计算的相频相应  
title('STM32计算相频响应');  
xlabel('频率');  
ylabel('相角');
```

运行 Matlab 后的输出结果如下：



从上面的对比结果中可以看出，Matlab 和函数 arm_cfft_f64 计算的结果基本是一致的。幅频响应求出的幅值和相频响应中的求出的初始相角都是没问题的。

30.5 实验例程说明 (MDK)

配套例子：

V7-220_复数浮点 FFT(支持单精度和双精度)

实验目的：



1. 学习复数浮点 FFT，支持单精度浮点和双精度浮点

实验内容：

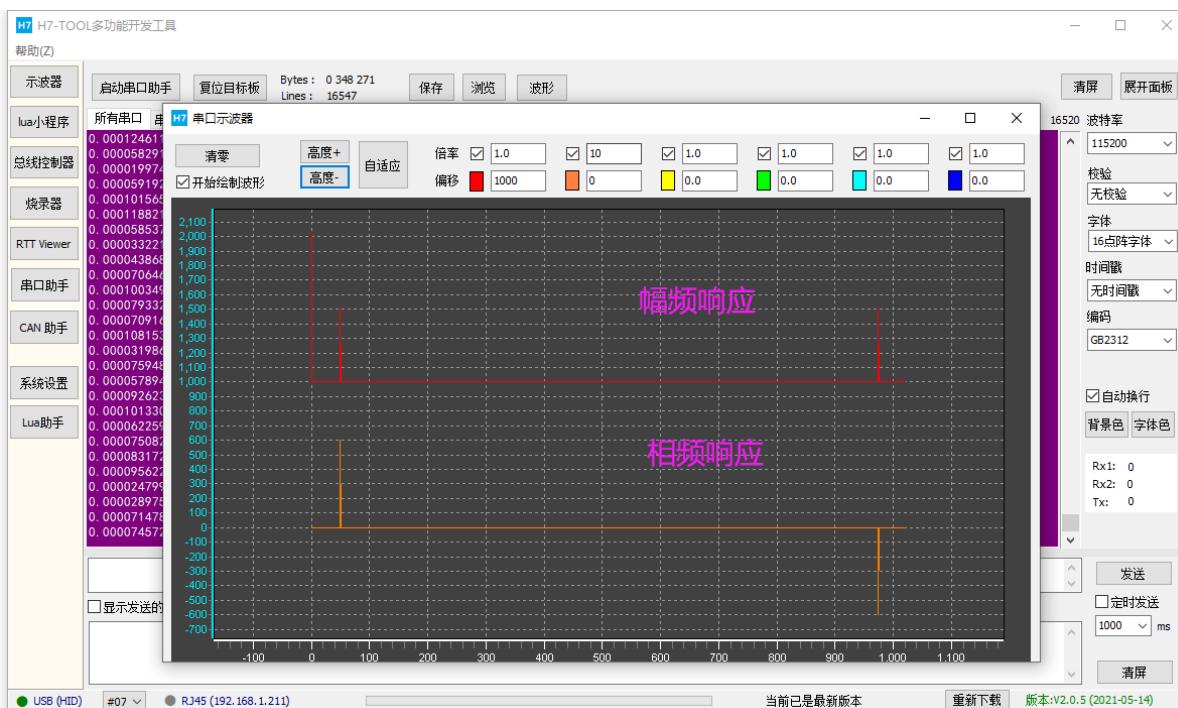
- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印 1024 点复数单精度 FFT 的幅频响应和相频响应。
- 按下按键 K2，串口打印 1024 点复数双精度 FFT 的幅频响应和相频响应。

使用 AC6 注意事项

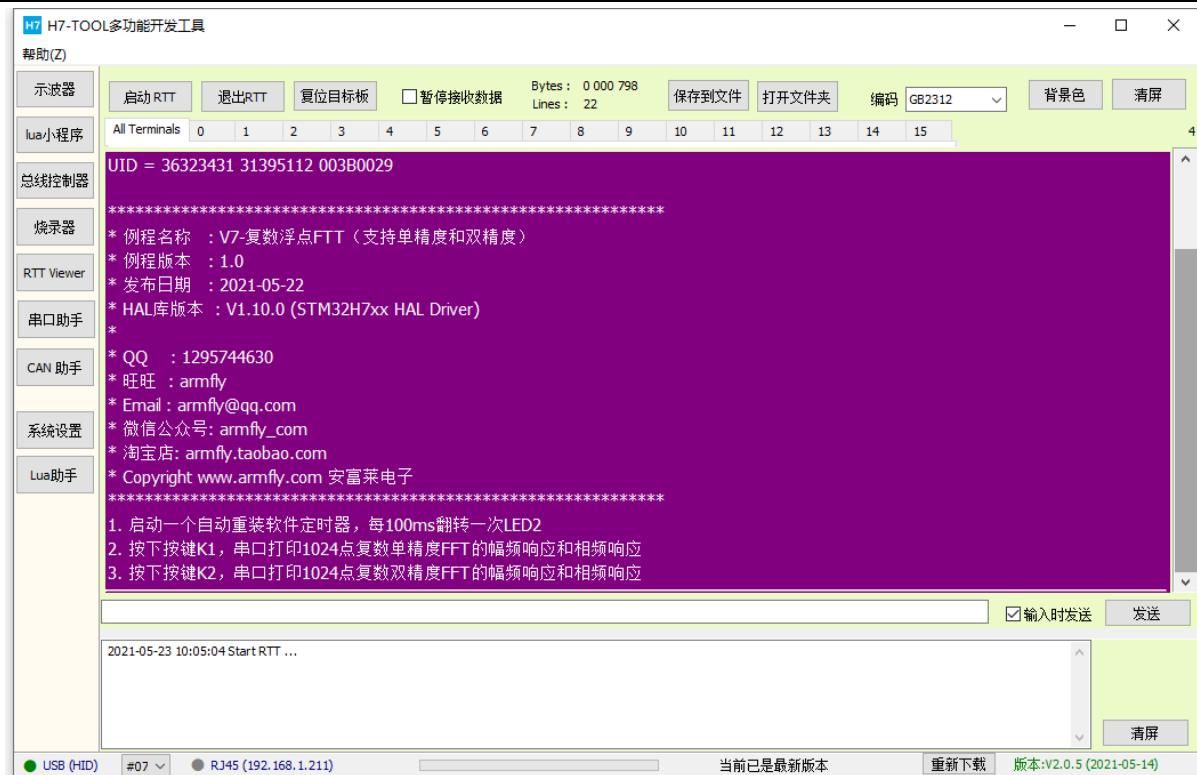
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

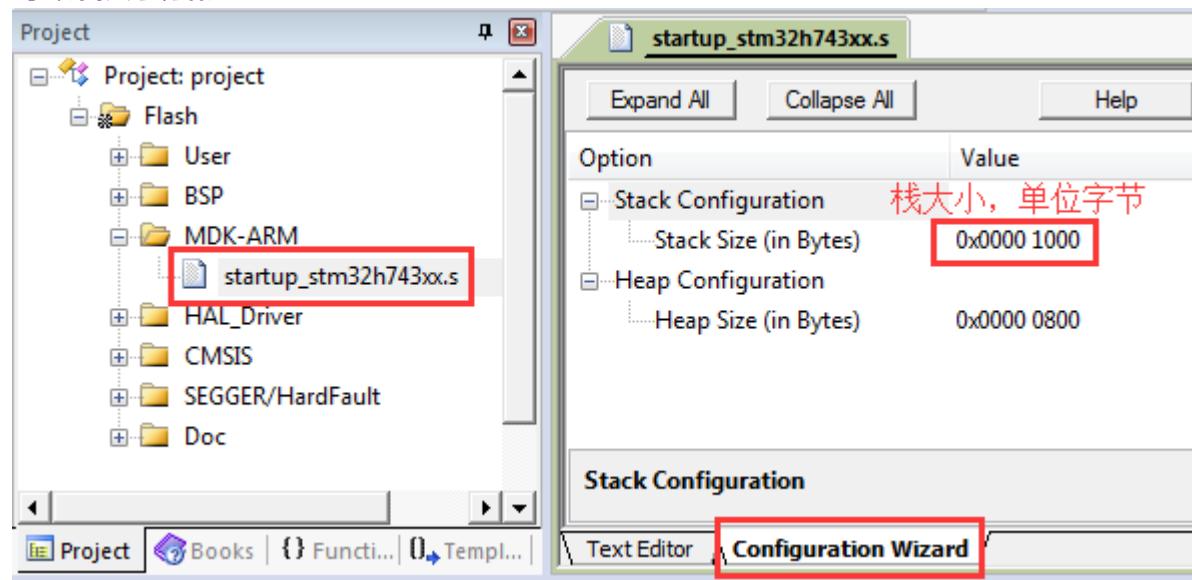


RTT 方式打印信息：

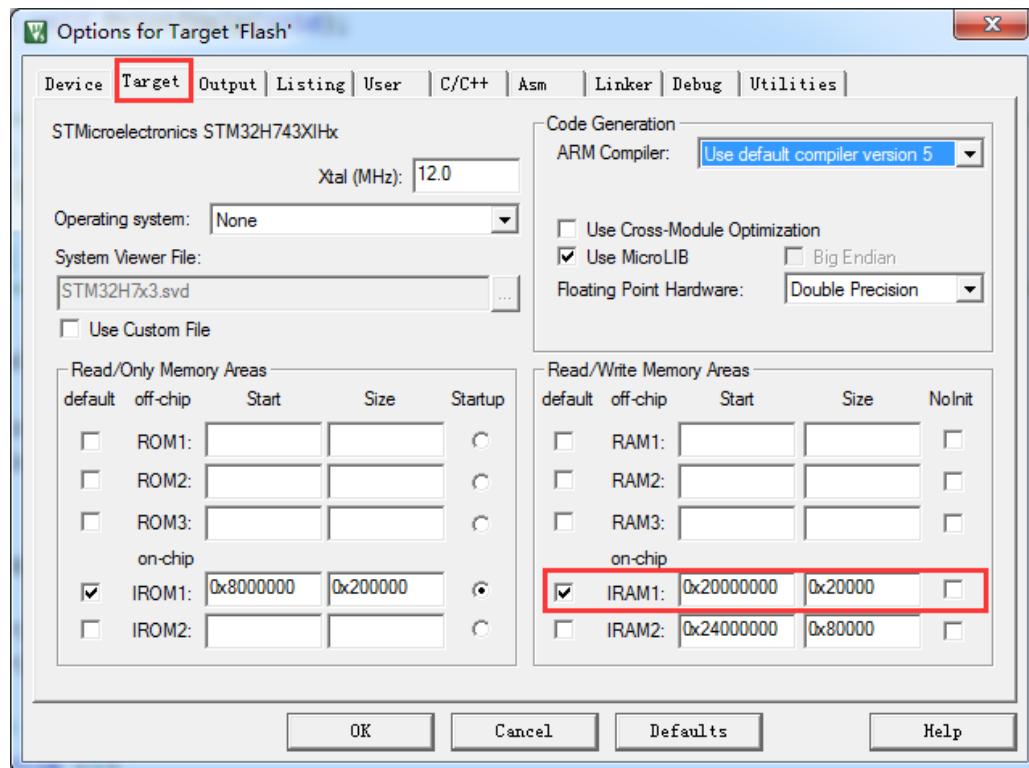


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印 1024 点复数单精度 FFT 的幅频响应和相频响应。
- 按下按键 K2，串口打印 1024 点复数双精度 FFT 的幅频响应和相频响应。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(4); /* 翻转 LED2 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_cfft_f32_app();
                break;

            case KEY_DOWN_K2:          /* K2 键按下 */
                arm_cfft_f64_app();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
```

30.6 实验例程说明 (IAR)

配套例子：

V7-220_复数浮点 FFT(支持单精度和双精度)

实验目的：

1. 学习复数浮点 FFT，支持单精度浮点和双精度浮点

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打印 1024 点复数单精度 FFT 的幅频响应和相频响应。
3. 按下按键 K2，串口打印 1024 点复数双精度 FFT 的幅频响应和相频响应。

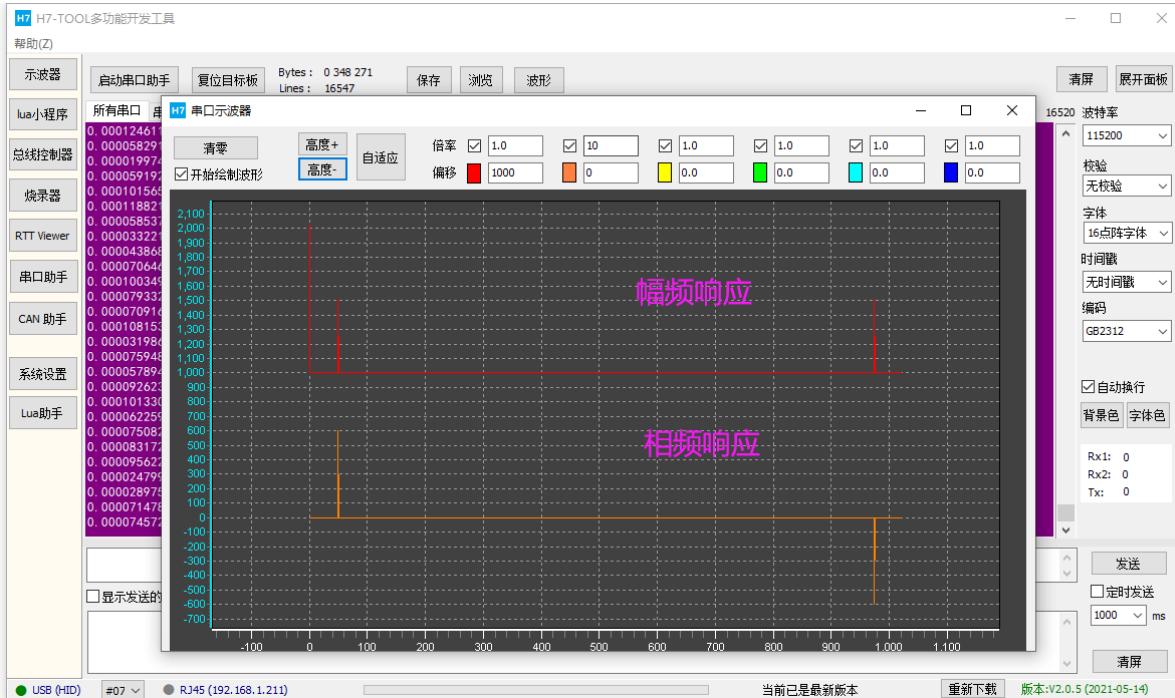
使用 AC6 注意事项



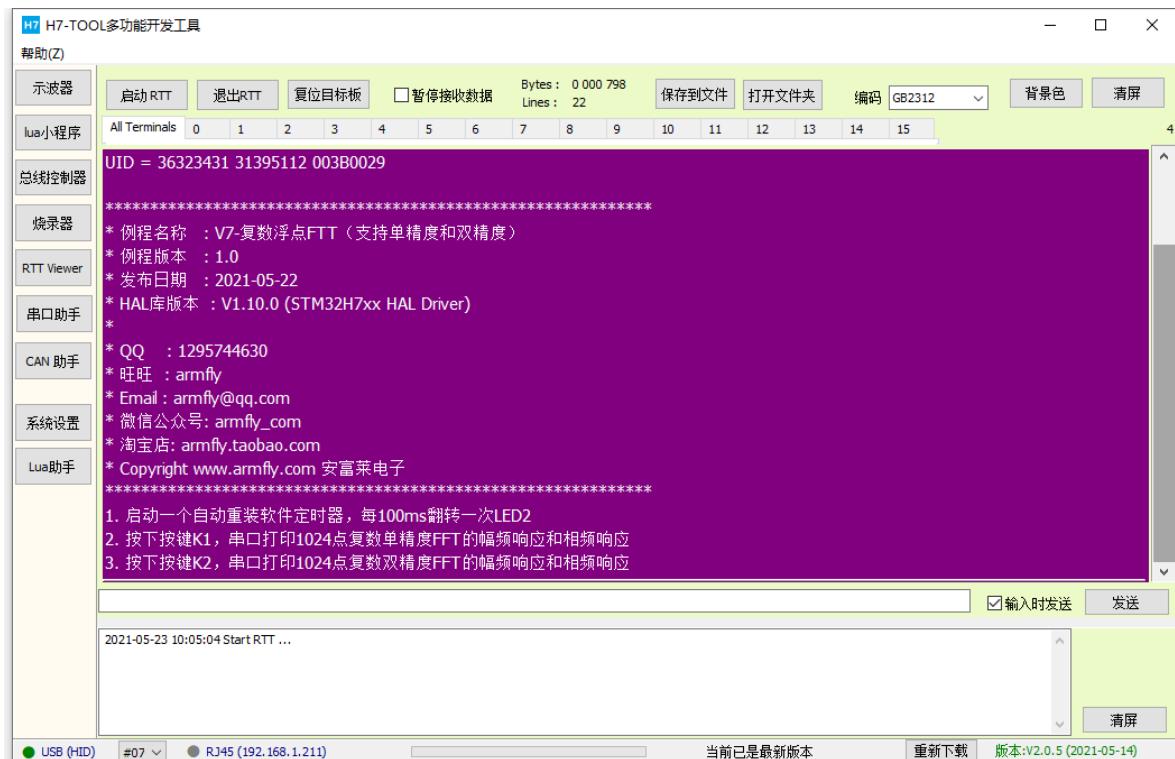
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

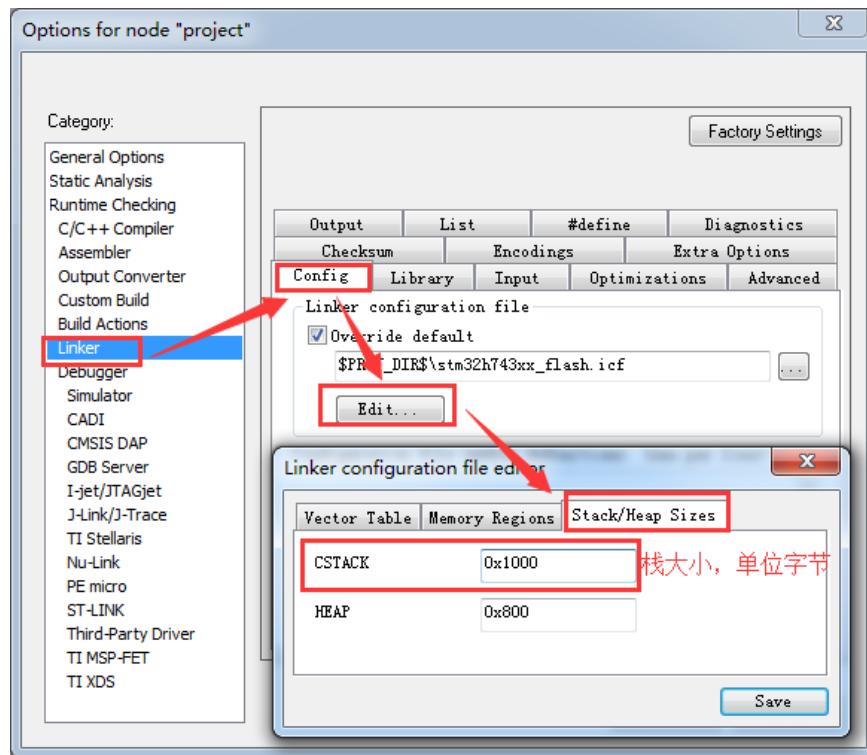


RTT 方式打印信息：

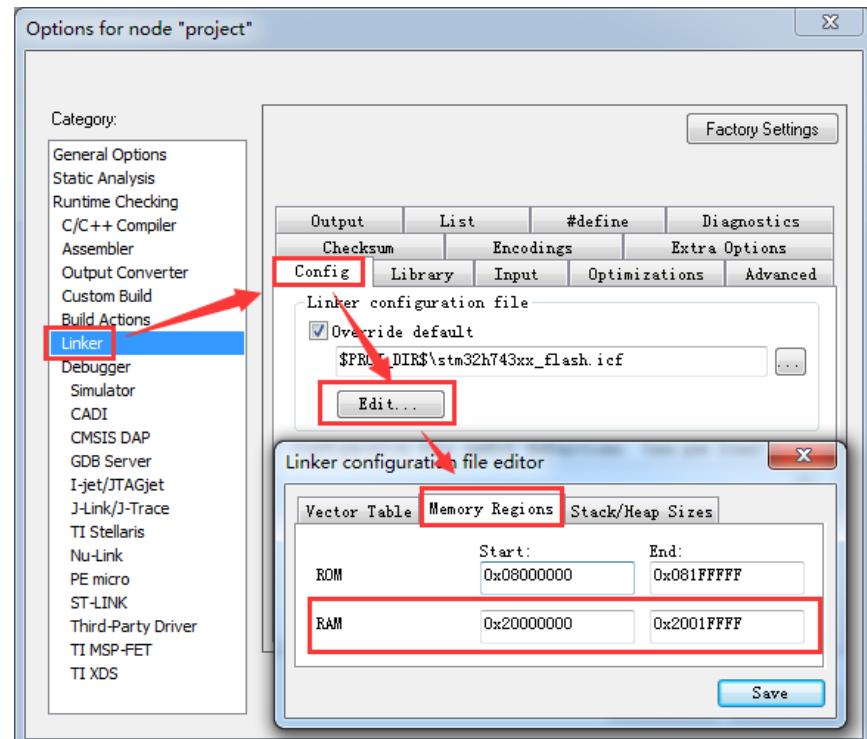


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
*****
```



```
/* 功能说明：初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形    参：无
* 返回值：无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
        - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
        - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名：MPU_Config
* 功能说明：配置 MPU
* 形    参：无
* 返回值：无
*****
*/
static void MPU_Config( void )
```



```
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /*使能 MPU */  
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);  
}  
  
/*  
*****  
* 函数名: CPU_CACHE_Enable  
* 功能说明: 使能 L1 Cache  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void CPU_CACHE_Enable(void)  
{  
    /* 使能 I-Cache */  
    SCB_EnableICache();  
  
    /* 使能 D-Cache */  
    SCB_EnableDCache();  
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印 1024 点复数单精度 FFT 的幅频响应和相频响应。
- 按下按键 K2，串口打印 1024 点复数双精度 FFT 的幅频响应和相频响应。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
***** */

int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(4); /* 翻转 LED2 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_cfft_f32_app();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下 */
                    arm_cfft_f64_app();
                    break;

                default:
                    /* 其它的键值不处理 */
            }
        }
    }
}
```



```
        break;
```

```
}
```

```
}
```

30.7 总结

本章节设计到 FFT 实现，有兴趣的可以深入了解源码的实现。

第31章 STM32H7 实数浮点 FFT (支持单精度

和双精度)

本章主要讲解实数浮点 FFT，支持单精度和双精度。

31.1 初学者重要提示

31.2 实数浮点 FFT 说明

31.3 单精度函数 arm_rfft_fast_f32 的使用 (含幅频和相频)

31.4 双精度函数 arm_rfft_fast_f64 的使用 (含幅频和相频)

31.5 实验例程说明(MDK)

31.6 实验例程说明(IAR)

31.7 总结

31.1 初学者重要提示

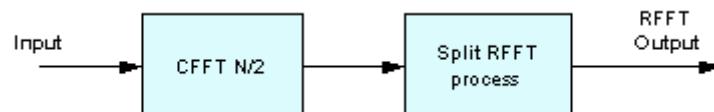
- ◆ 与上一章节的复数 FFT 相比，实数 FFT 仅需用户输入实部即可。输出结果根据 FFT 的对称性，也仅输出一半的频谱。

31.2 实数浮点 FFT 说明

CMSIS DSP 库里面包含一个专门用于计算实数序列的 FFT 库，很多情况下，用户只需要计算实数序列即可。计算同样点数 FFT 的实数序列要比计算同样点数的虚数序列有速度上的优势。

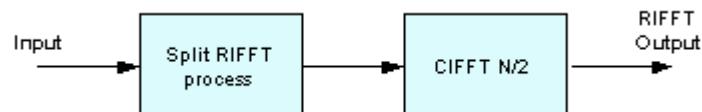
快速的 rfft 算法是基于混合基 cfft 算法实现的。

一个 N 点的实数序列 FFT 正变换采用下面的步骤实现：



由上面的框图可以看出，实数序列的 FFT 是先计算 $N/2$ 个实数的 CFFT，然后再重塑数据进行处理从而获得半个 FFT 频谱即可（利用了 FFT 变换后频谱的对称性）。

一个 N 点的实数序列 FFT 逆变换采用下面的步骤实现：





实数 FFT 支持浮点, Q31 和 Q15 三种数据类型。

31.3 单精度函数 arm_rfft_fast_f32 的使用(含幅频和相频)

31.3.1 函数说明

函数原型:

```
void arm_rfft_fast_f32(
    const arm_rfft_fast_instance_f32 * S,
    float32_t * p,
    float32_t * pOut,
    uint8_t ifftFlag)
```

函数描述:

这个函数用于单精度浮点实数 FFT。

函数参数:

- ◆ 第 1 个参数是封装好的浮点 FFT 例化, 需要用户先调用函数 arm_rfft_fast_init_f32 初始化, 然后供此函数 arm_rfft_fast_f32 调用。支持 32, 64, 128, 256, 512, 1024, 2048, 4096 点 FFT。
比如做 1024 点 FFT, 代码如下:

```
arm_rfft_fast_instance_f32 S;
arm_rfft_fast_init_f32(&S, 1024);
arm_rfft_fast_f32(&S, testInput_f32, testOutput_f32, ifftFlag);
```
- ◆ 第 2 个参数是实数地址, 比如我们要做 1024 点实数 FFT, 要保证有 1024 个缓冲。
- ◆ 第 3 个参数是 FFT 转换结果, 转换结果不是实数了, 而是复数, 按照实部, 虚拟, 实部, 虚部, 依次排列。比如做 1024 点 FFT, 这里的输出也会有 1024 个数据, 即 512 个复位。
- ◆ 第 4 个参数用于设置正变换和逆变换, ifftFlag=0 表示正变换, ifftFlag=1 表示逆变换。

31.3.2 使用举例并和 Matlab 比较

下面通过在开发板上运行这个函数并计算幅频相应, 然后再与Matlab计算的结果做对比。

```
/*
*****
* 函数名: arm_rfft_f32_app
* 功能说明: 调用函数 arm_rfft_fast_f32 计算幅频和相频
* 形参: 无
* 返回值: 无
*****
*/
static void arm_rfft_f32_app(void)
{
    uint16_t i;
    arm_rfft_fast_instance_f32 S;
```



```
/* 正变换 */
ifftFlag = 0;

/* 初始化结构体 S 中的参数 */
arm_rfft_fast_init_f32(&S, TEST_LENGTH_SAMPLES);

for(i=0; i<1024; i++)
{
    /* 波形是由直流分量, 50Hz 正弦波组成, 波形采样率 1024, 初始相位 60° */
    testInput_f32[i] = 1 + cos(2*3.1415926f*50*i/1024 + 3.1415926f/3);
}

/* 1024 点实序列快速 FFT */
arm_rfft_fast_f32(&S, testInput_f32, testOutput_f32, ifftFlag);

/* 为了方便跟函数 arm_cfft_f32 计算的结果做对比, 这里求解了 1024 组模值, 实际函数 arm_rfft_fast_f32
只求解出了 512 组
*/
arm_cmplx_mag_f32(testOutput_f32, testOutputMag_f32, TEST_LENGTH_SAMPLES);

printf("=====\\r\\n");

/* 求相频 */
PowerPhaseRadians_f32(testOutput_f32, Phase_f32, TEST_LENGTH_SAMPLES, 0.5f);

/* 串口打印求解的幅频和相频 */
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    printf("%f, %f\\r\\n", testOutputMag_f32[i], Phase_f32[i]);
}
```

运行函数arm_rfft_f32_app可以通过串口打印出计算的模值和相角，下面我们就通过Matlab计算的模值和相角跟arm_rfft_fast_f32计算的做对比。

对比前需要先将串口打印出的数据加载到 Matlab 中，并给这个数组起名 sampledata，加载方法在前面的教程的**第 13 章 13.6 小结已经讲解**，这里不做赘述了。Matlab 中运行的代码如下：

```
Fs = 1024;          % 采样率
N = 1024;          % 采样点数
n = 0:N-1;          % 采样序列
t = 0:1/Fs:1-1/Fs; % 时间序列
f = n * Fs / N;    % 真实的频率

% 波形是由直流分量, 50Hz 正弦波正弦波组成
x = 1 + cos(2*pi*50*t + pi/3); 
y = fft(x, N);           % 对原始信号做FFT变换
Mag = abs(y);

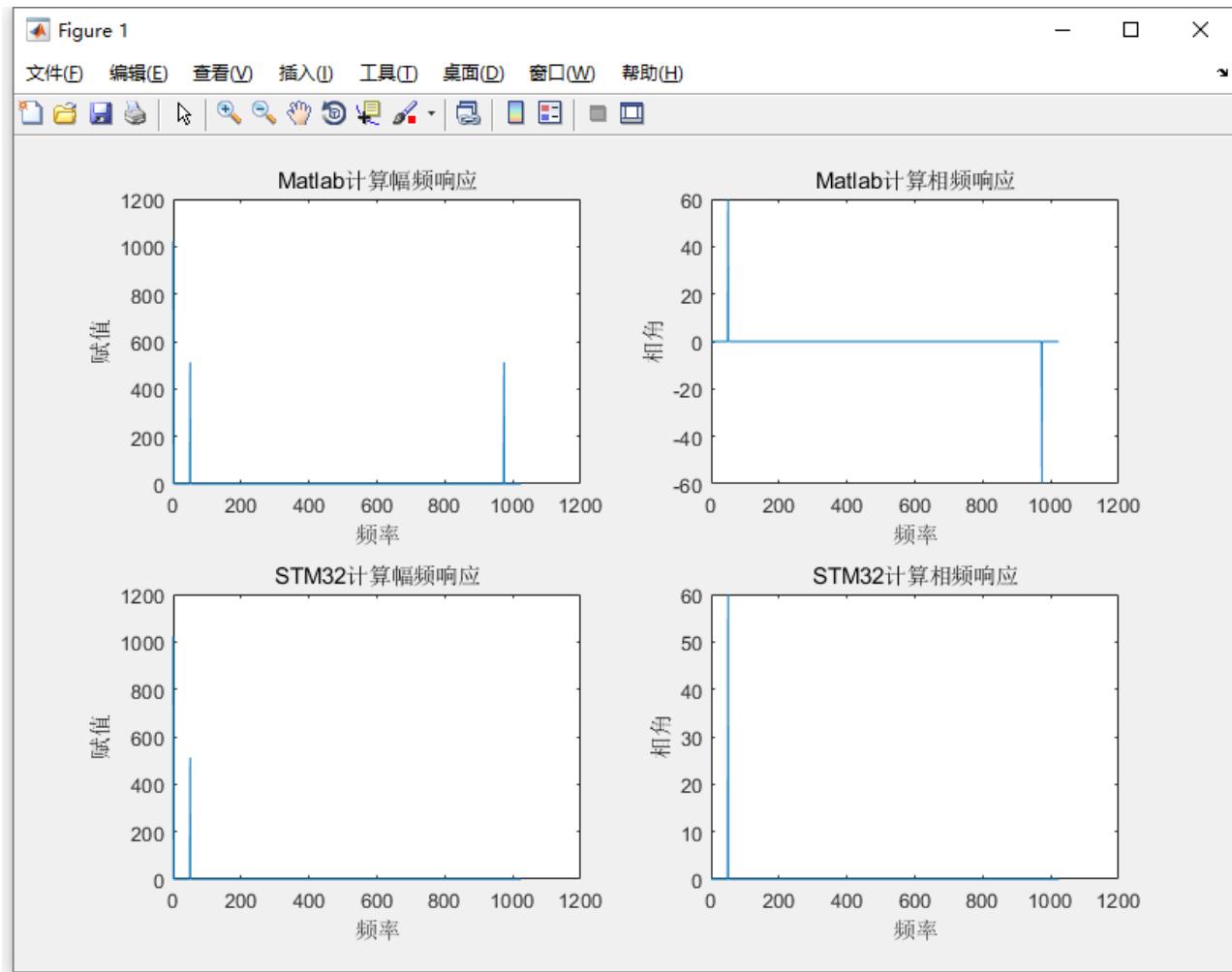
subplot(2, 2, 1);
plot(f, Mag);
title('Matlab计算幅频响应');
xlabel('频率');
ylabel('幅值');

subplot(2, 2, 2);
realvalue = real(y);
imagvalue = imag(y);
plot(f, atan2(imagvalue, realvalue)*180/pi.* (Mag>=200));
title('Matlab计算相频响应');
xlabel('频率');
ylabel('相角');

subplot(2, 2, 3);
```

```
plot(f, sampledata1); %绘制STM32计算的幅频响应  
title('STM32计算幅频响应');  
xlabel('频率');  
ylabel('赋值');  
  
subplot(2, 2, 4);  
plot(f, sampledata2); %绘制STM32计算的相频响应  
title('STM32计算相频响应');  
xlabel('频率');  
ylabel('相角');
```

运行 Matlab 后的输出结果如下：



从上面的对比结果中可以看出，从上面的前 512 点对比中，我们可以看出两者的计算结果是相符的。Matlab 和函数 arm_rfft_fast_f32 计算的结果基本是一致的。幅频响应求出的幅值和相频响应中的求出的初始相角都是没问题的。

31.4 双精度函数 arm_rfft_fast_f64 的使用(含幅频和相频)

31.4.1 函数说明

函数原型：



```
void arm_rfft_fast_f64(
    arm_rfft_fast_instance_f64 * S,
    float64_t * p,
    float64_t * pOut,
    uint8_t ifftFlag)
```

函数描述：

这个函数用于双精度浮点实数 FFT。

函数参数：

- ◆ 第 1 个参数是封装好的浮点 FFT 例化，需要用户先调用函数 arm_rfft_fast_init_f64 初始化，然后供此函数 arm_rfft_fast_f64 调用。支持 32, 64, 128, 256, 512, 1024, 2048, 4096 点 FFT。
比如做 1024 点 FFT，代码如下：

```
arm_rfft_fast_instance_f64 S;
arm_rfft_fast_init_f64(&S, 1024);
arm_rfft_fast_f64(&S, testInput_f64, testOutput_f64, ifftFlag);
```
- ◆ 第 2 个参数是实数地址，比如我们要做 1024 点实数 FFT，要保证有 1024 个缓冲。
- ◆ 第 3 个参数是 FFT 转换结果，转换结果不是实数了，而是复数，按照实部，虚拟，实部，虚拟，依次排列。比如做 1024 点 FFT，这里的输出也会有 1024 个数据，即 512 个复位。
- ◆ 第 4 个参数用于设置正变换和逆变换，ifftFlag=0 表示正变换，ifftFlag=1 表示逆变换

31.4.2 使用举例并和 Matlab 比较

下面通过在开发板上运行这个函数并计算幅频相应，然后再与Matlab计算的结果做对比。

```
/*
*****
* 函数名: arm_rfft_f64_app
* 功能说明: 调用函数arm_rfft_fast_f64计算幅频和相频
* 形参: 无
* 返回值: 无
*****
*/
static void arm_rfft_f64_app(void)
{
    uint16_t i;
    float64_t lX, lY;
    arm_rfft_fast_instance_f64 S;

    /* 正变换 */
    ifftFlag = 0;

    /* 初始化结构体S中的参数 */
    arm_rfft_fast_init_f64(&S, TEST_LENGTH_SAMPLES);

    for(i=0; i<1024; i++)
    {
        /* 波形是由直流分量, 50Hz正弦波组成, 波形采样率1024, 初始相位60° */
        testInput_f64[i] = 1 + cos(2*3.1415926*50*i/1024 + 3.1415926/3);
    }

    /* 1024点实序列快速FFT */
    arm_rfft_fast_f64(&S, testInput_f64, testOutput_f64, ifftFlag);
```



```
/* 求解模值 */
for (i =0; i < TEST_LENGTH_SAMPLES; i++)
{
    lX = testOutput_f64[2*i];           /* 实部*/
    lY = testOutput_f64[2*i+1];         /* 虚部 */
    testOutputMag_f64[i] = sqrt(lX*lX+ lY*lY); /* 求模 */
}

printf("=====\\r\\n");

/* 求相频 */
PowerPhaseRadians_f64(testOutput_f64, Phase_f64, TEST_LENGTH_SAMPLES, 0.5);

/* 串口打印幅值和相频 */
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    printf("%.11f, %.11f\\r\\n", testOutputMag_f64[i], Phase_f64[i]);
}

}
```

运行函数arm_rfft_f64_app可以通过串口打印出计算的模值和相角，下面我们就通过Matlab计算的模值和相角跟arm_rfft_fast_f32计算的做对比。

对比前需要先将串口打印出的数据加载到 Matlab 中，并给这个数组起名 sampledata，加载方法在前面的教程的[第13章 13.6 小结已经讲解](#)，这里不做赘述了。Matlab 中运行的代码如下：

```
Fs = 1024;          % 采样率
N = 1024;          % 采样点数
n = 0:N-1;          % 采样序列
t = 0:1/Fs:1-1/Fs; % 时间序列
f = n * Fs / N;    % 真实的频率

% 波形是由直流分量，50Hz正弦波正弦波组成
x = 1 + cos(2*pi*50*t + pi/3) ;
y = fft(x, N);      % 对原始信号做FFT变换
Mag = abs(y);

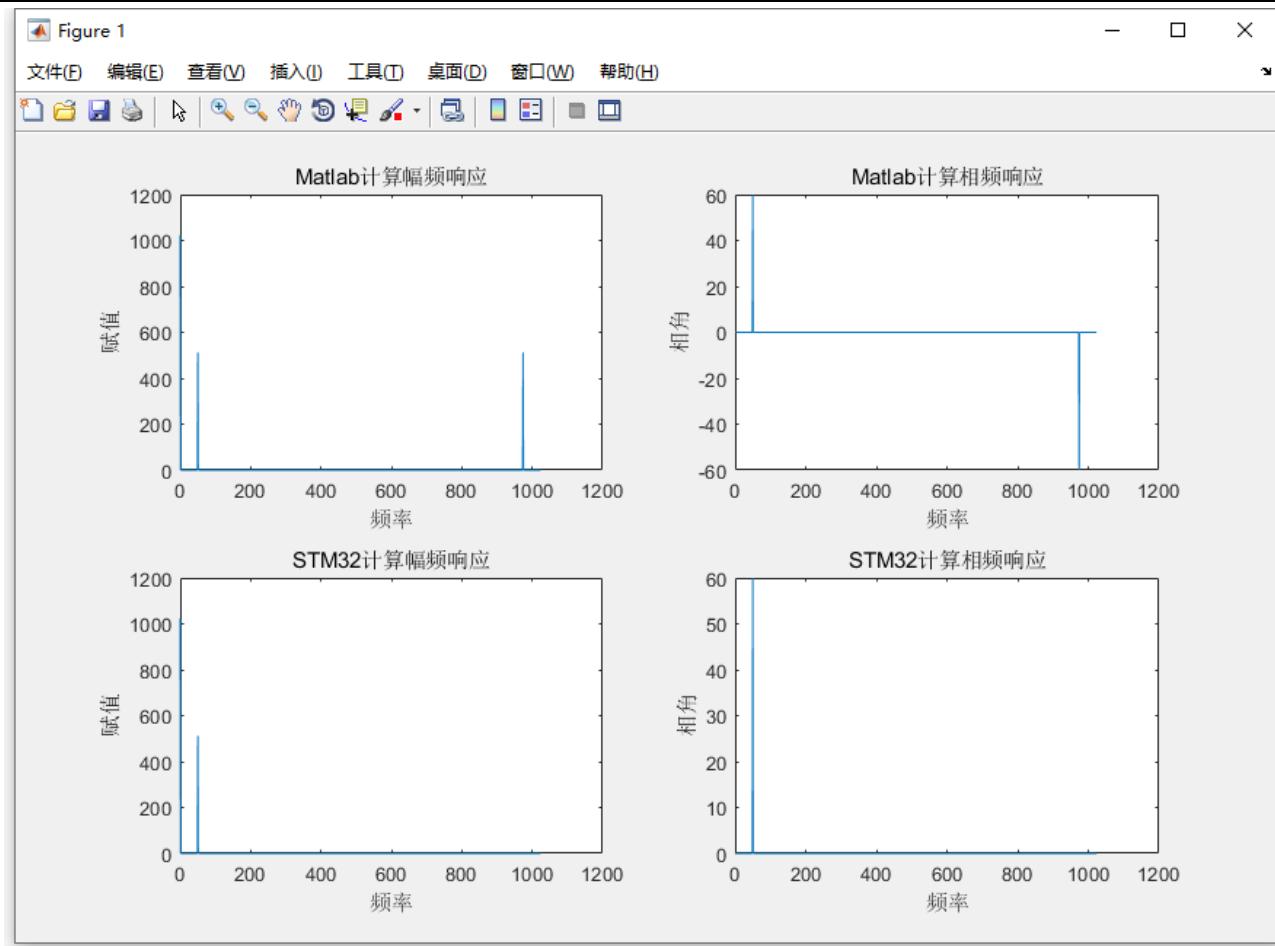
subplot(2, 2, 1);
plot(f, Mag);
title('Matlab计算幅频响应');
xlabel('频率');
ylabel('赋值');

subplot(2, 2, 2);
realvalue = real(y);
imagvalue = imag(y);
plot(f, atan2(imagvalue, realvalue)*180/pi.* (Mag>=200));
title('Matlab计算相频响应');
xlabel('频率');
ylabel('相角');

subplot(2, 2, 3);
plot(f, sampledata1); % 绘制STM32计算的幅频相应
title('STM32计算幅频响应');
xlabel('频率');
ylabel('赋值');

subplot(2, 2, 4);
plot(f, sampledata2); % 绘制STM32计算的相频相应
title('STM32计算相频响应');
xlabel('频率');
ylabel('相角');
```

运行 Matlab 后的输出结果如下：



从上面的对比结果中可以看出，从上面的前 512 点对比中，我们可以看出两者的计算结果是相符的。Matlab 和函数 arm_rfft_fast_f64 计算的结果基本是一致的。幅频响应求出的幅值和相频响应中的求出的初始相角都是没问题的。

31.5 实验例程说明 (MDK)

配套例子：

V7-221_实数浮点 FFT(支持单精度和双精度)

实验目的：

1. 学习实数浮点 FFT，支持单精度浮点和双精度浮点

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打印 1024 点实数单精度 FFT 的幅频响应和相频响应。
3. 按下按键 K2，串口打印 1024 点实数双精度 FFT 的幅频响应和相频响应。

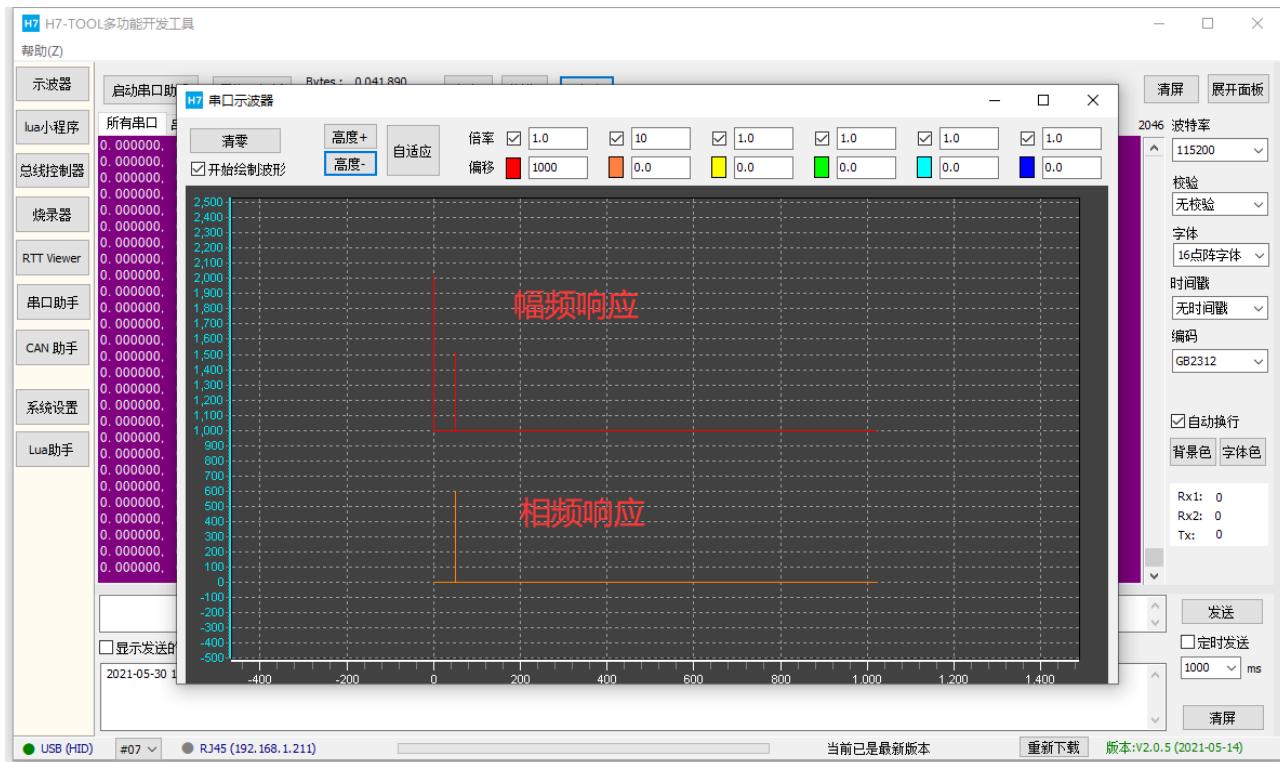
使用 AC6 注意事项

特别注意附件章节 C 的问题

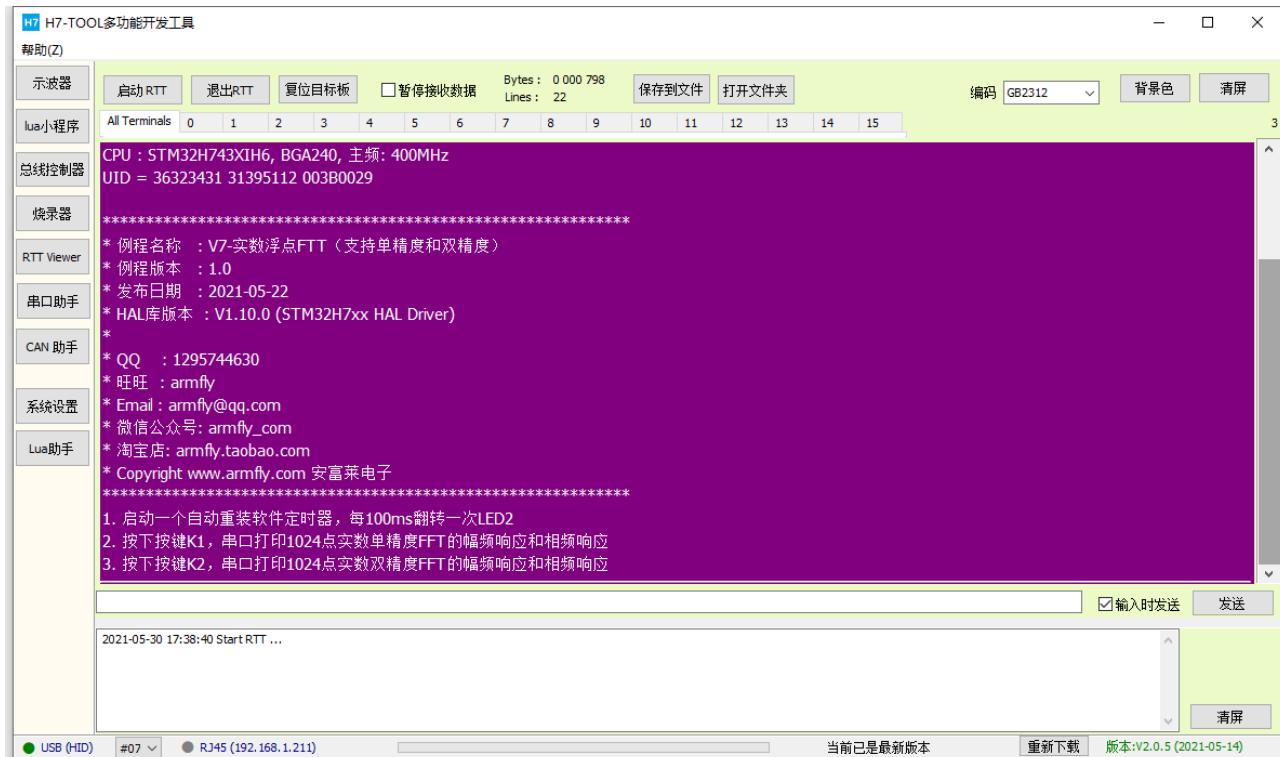
上电后串口打印的信息：



波特率 115200，数据位 8，奇偶校验位无，停止位 1。

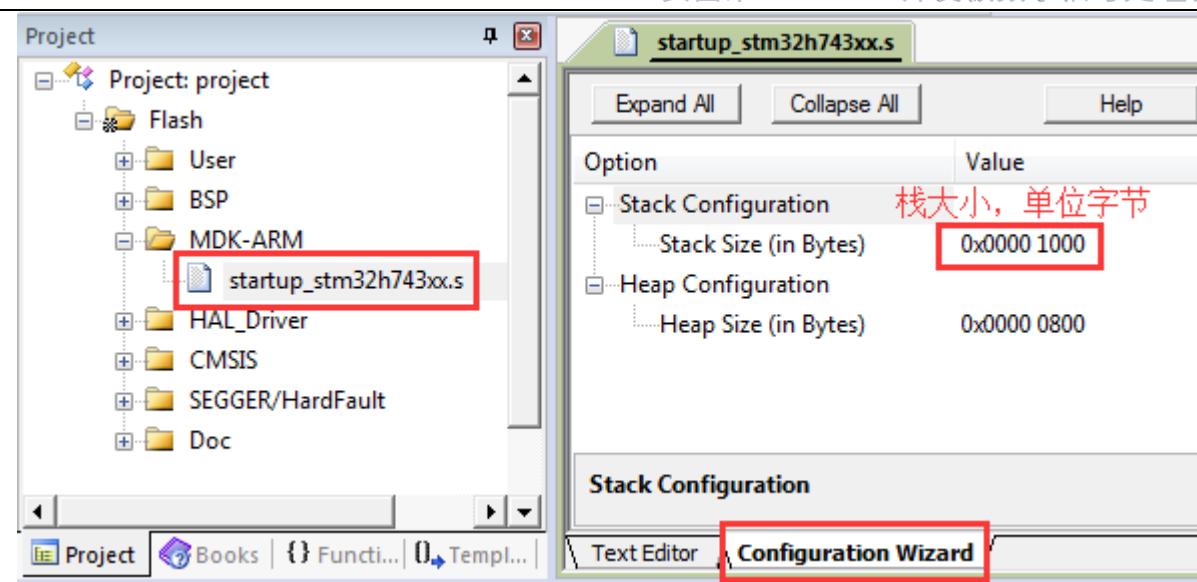


RTT 方式打印信息：

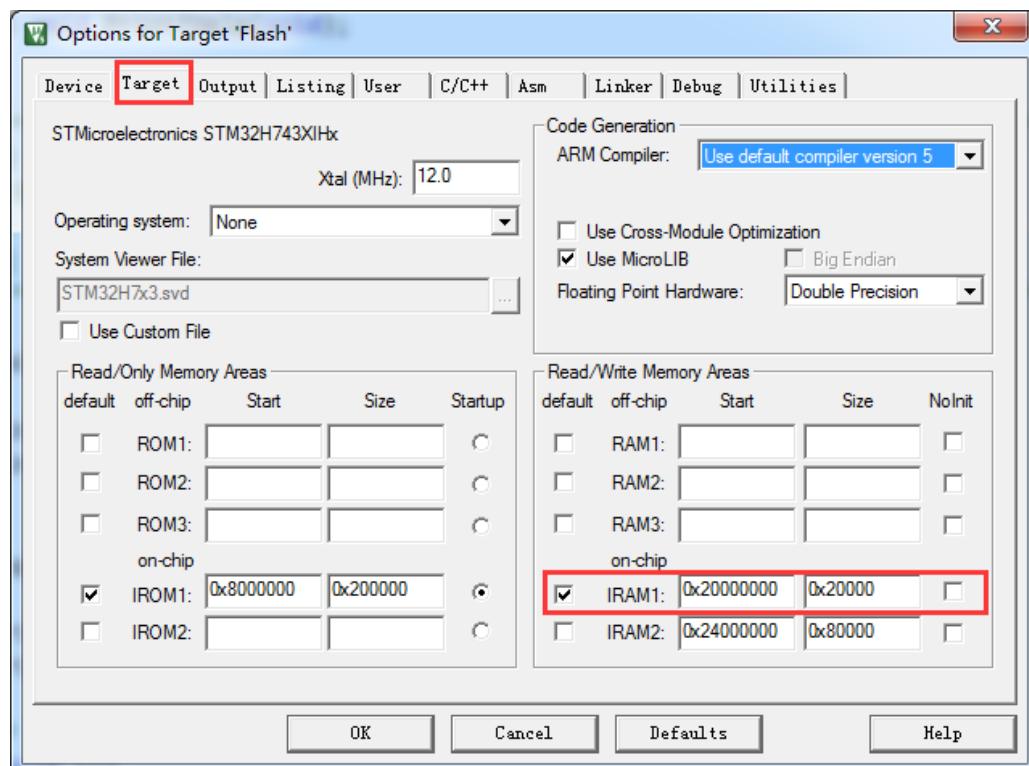


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现:

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
```



```
{  
    /* 配置 MPU */  
    MPU_Config();  
  
    /* 使能 L1 Cache */  
    CPU_CACHE_Enable();  
  
    /*  
     * STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：  
     * - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。  
     * - 设置 NVIC 优先级分组为 4。  
     */  
    HAL_Init();  
  
    /*  
     * 配置系统时钟到 400MHz  
     * - 切换使用 HSE。  
     * - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。  
     */  
    SystemClock_Config();  
  
    /*  
     * Event Recorder:  
     * - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。  
     * - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章  
     */  
#if Enable_EventRecorder == 1  
    /* 初始化 EventRecorder 并开启 */  
    EventRecorderInitialize(EventRecordAll, 1U);  
    EventRecorderStart();  
#endif  
  
    bsp_InitKey(); /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */  
    bsp_InitTimer(); /* 初始化滴答定时器 */  
    bsp_InitUart(); /* 初始化串口 */  
    bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */  
    bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();
```



```
/* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。



- 按下按键 K1，串口打印 1024 点实数单精度 FFT 的幅频响应和相频响应。
- 按下按键 K2，串口打印 1024 点实数双精度 FFT 的幅频响应和相频响应。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(4); /* 翻转 LED2 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_rfft_f32_app();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下 */
                    arm_rfft_f64_app();
                    break;

                default:
                    /* 其它的键值不处理 */
                    break;
            }
        }
    }
}
```



{}

31.6 实验例程说明 (IAR)

配套例子：

V7-221_实数浮点 FFT(支持单精度和双精度)

实验目的：

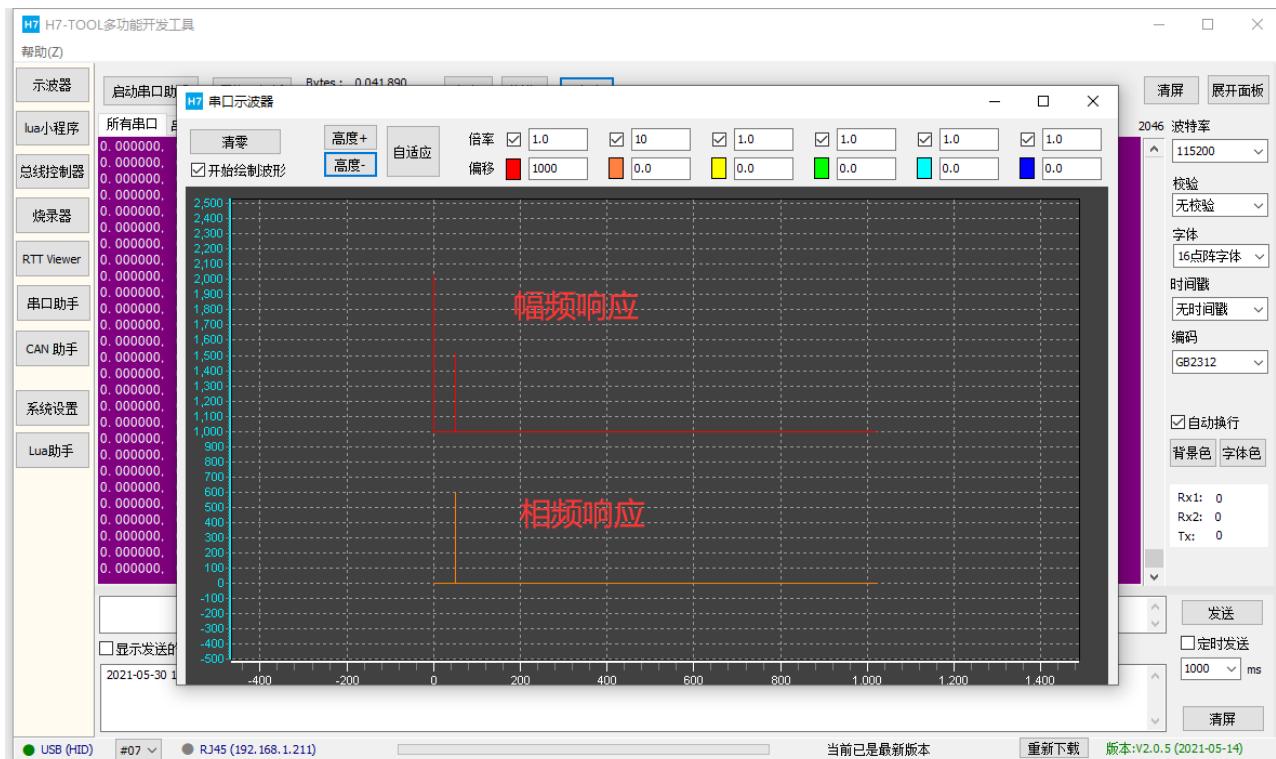
1. 学习实数浮点 FFT，支持单精度浮点和双精度浮点

实验内容：

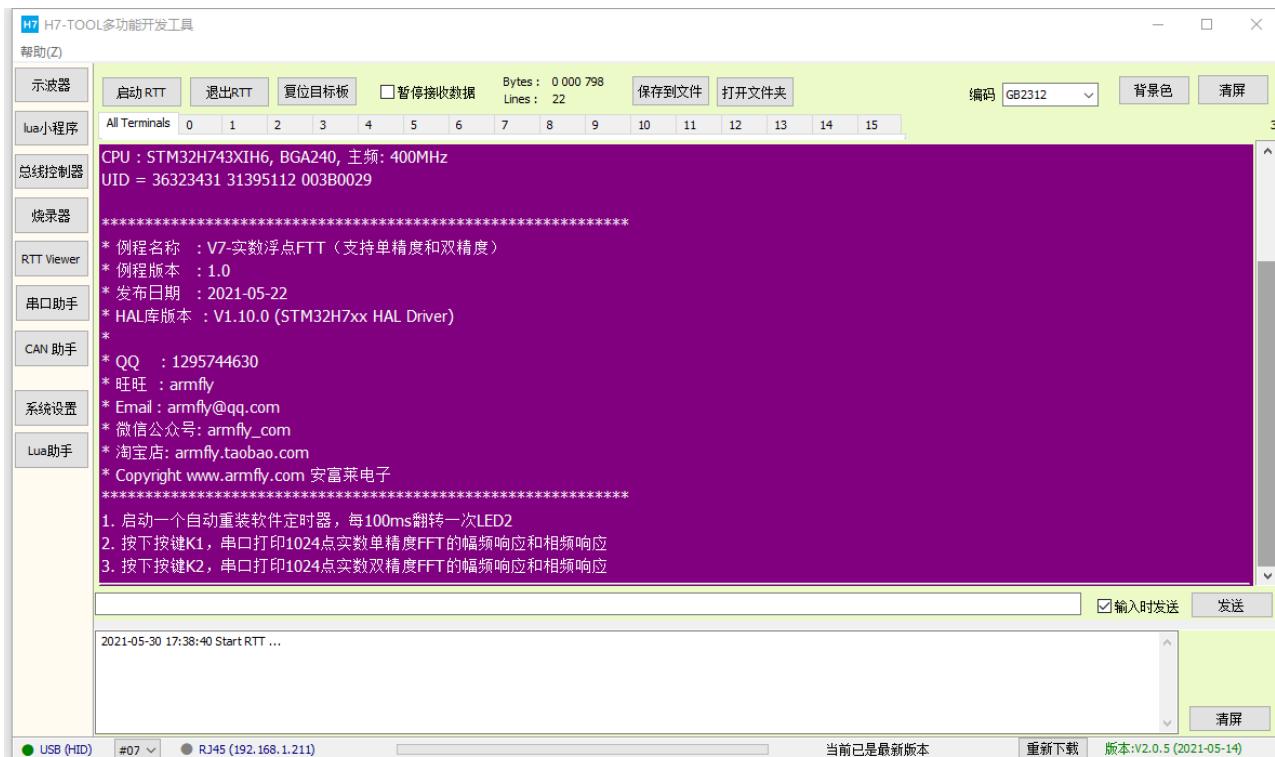
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打印 1024 点实数单精度 FFT 的幅频响应和相频响应。
3. 按下按键 K2，串口打印 1024 点实数双精度 FFT 的幅频响应和相频响应。

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

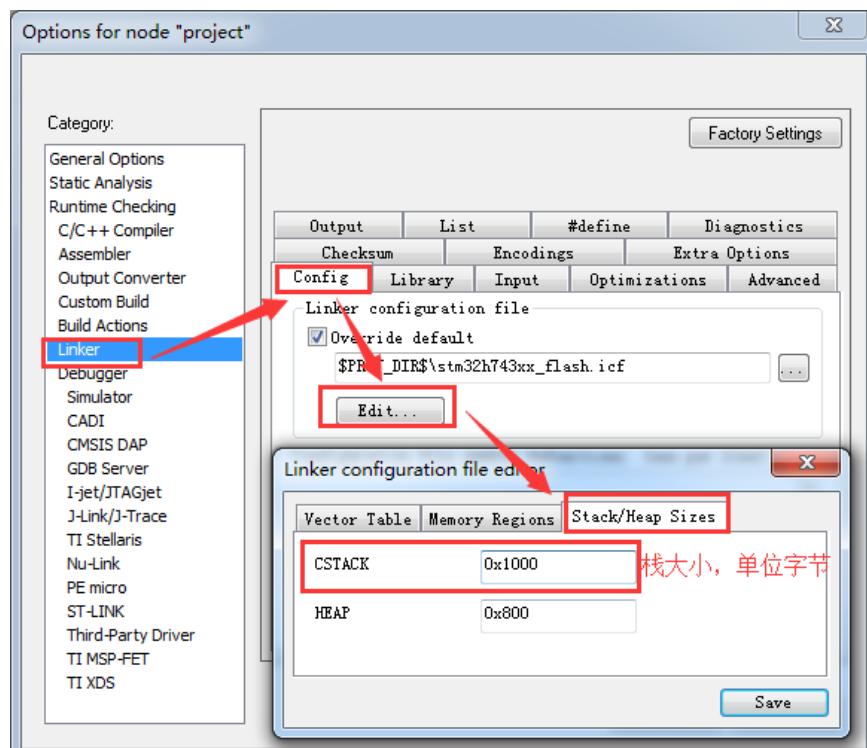


RTT 方式打印信息：

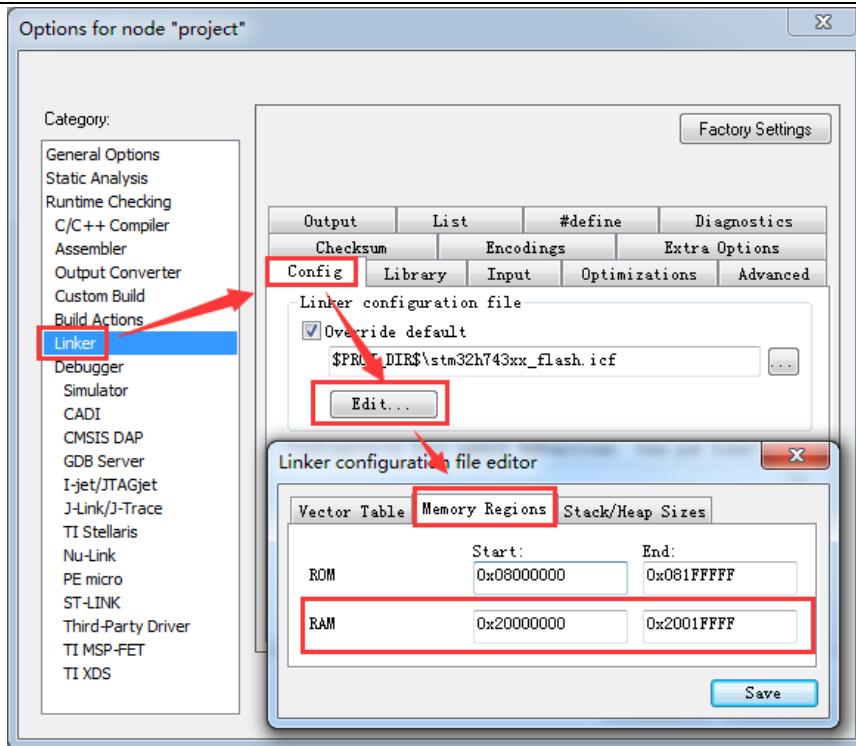


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印 1024 点实数单精度 FFT 的幅频响应和相频响应。
- 按下按键 K2，串口打印 1024 点实数双精度 FFT 的幅频响应和相频响应。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */
```



```
/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(4); /* 翻转 LED2 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:           /* K1 键按下 */
                arm_rfft_f32_app();
                break;

            case KEY_DOWN_K2:           /* K2 键按下 */
                arm_rfft_f64_app();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
```

31.7 总结

本章节设计到实数 FFT 实现，有兴趣的可以深入了解源码的实现。



第32章 STM32H7 的实数 FFT 的逆变换(支持单精度和双精度)

精度和双精度)

本章主要讲解实数 FFT 的逆变换实现。通过 FFT 变换将波形从时域转换到频域，通过 IFFT 逆变换实现从频域到时域变换。

通过本章为大家展示一个波形 FFT 变换，然后 IFFT 还原波形。

32.1 初学者重要提示

32.2 利用 FFT 库实现 IFFT 的思路

32.3 Matlab 实现 FFT 正变换和逆变换

32.4 单精度函数 arm_rfft_fast_f32 实现 FFT 正变换和逆变换

32.5 双精度函数 arm_rfft_fast_f64 实现 FFT 正变换和逆变换

32.6 实验例程说明(MDK)

32.7 实验例程说明(IAR)

32.8 总结

32.1 初学者重要提示

- ◆ STM32H7 支持硬件单精度浮点和硬件双精度浮点，计算 FFT 正变换和逆变换速度都会非常快。而 STM32F4 仅支持硬件单精度浮点。

32.2 利用 FFT 库实现 IFFT 的思路

如果希望直接调用 FFT 程序计算 IFFT，可以用下面的方法：

$$\begin{aligned} X(k) &= \text{DFT}[x(n)] = \sum_{n=0}^{N-1} x(n) W_N^{nk} \quad 0 \leq k \leq N - 1 \\ x^*(n) &= \text{IDFT}[X(k)] = \frac{1}{N} \sum_{n=0}^{N-1} X^*(k) W^{-nk} \quad 0 \leq k \leq N - 1 \end{aligned}$$

对上式两边同时去共轭，得：

$$x(n) = \text{IDFT}[X(k)] = \frac{1}{N} \left[\sum_{k=0}^{N-1} X^*(k) W_N^{kn} \right]^* = \frac{1}{N} \{ \text{DFT}[X^*(k)] \}^*$$

简单的说就是先对原始信号做 FFT 变换，然后对转换结果取共轭，再次带到 FFT 中计算，并将结果再次取共轭就可以实现 IFFT。

32.3 Matlab 实现 FFT 正变换和逆变换

根据上面小节的实现思路，我们在 Matlab 上面做一个验证，验证代码如下：

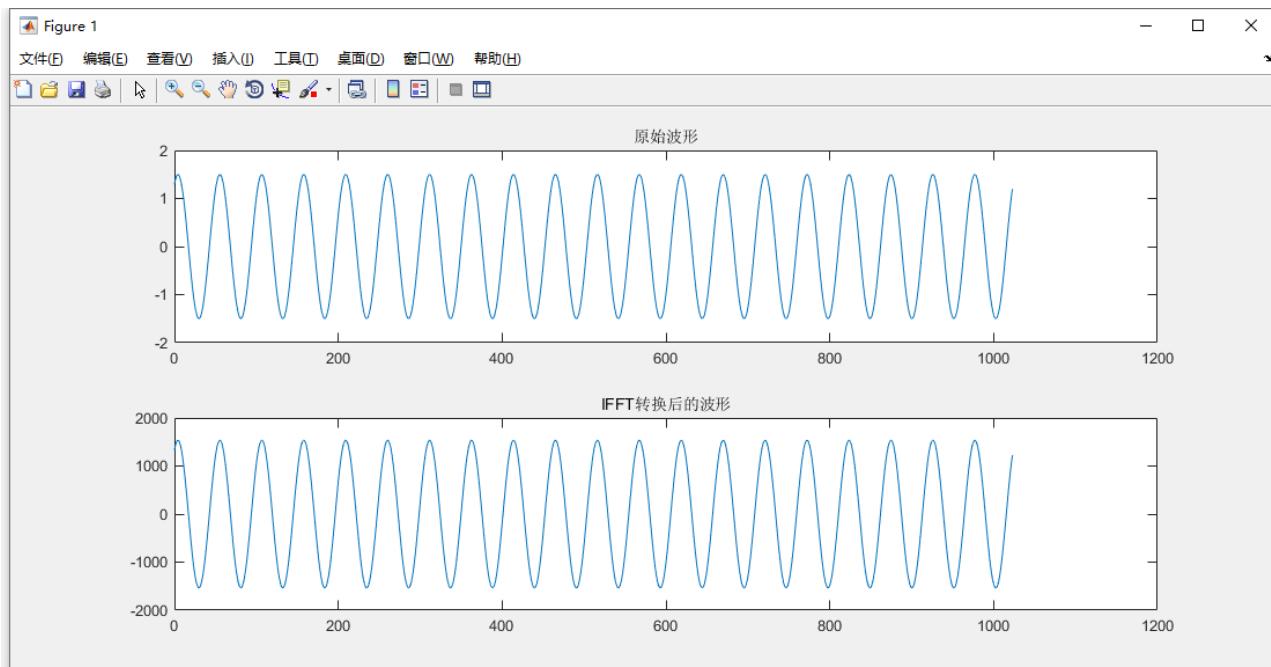
```
Fs = 1024; % 采样率
N = 1024; % 采样点数
n = 0:N-1; % 采样序列
t = 0:1/Fs:1-1/Fs; % 时间序列
f = n * Fs / N; % 真实的频率

x = 1.5*sin(2*pi*20*t+pi/3); % 原始信号
y = fft(x, N); % 对原始信号做FFT变换
z = conj(y); % 对转换结果取共轭

subplot(2, 1, 2);
z = fft(z, N); % 再次做FFT
k = conj(z); % 对转换结果去共轭
plot(f, real(k)); % 绘制转换后的波形
title('IFFT转换后的波形');

subplot(2, 1, 1);
plot(f, x); % 绘制原始波形
title('原始波形');
```

Matab 的运行结果如下：



从上面的转换结果看，两个波形信号基本是一致的。



32.4 单精度函数 arm_rfft_fast_f32 实现 FFT 正变换和逆变换

32.4.1 函数说明

函数原型：

```
void arm_rfft_fast_f32(
    const arm_rfft_fast_instance_f32 * S,
    float32_t * p,
    float32_t * pOut,
    uint8_t ifftFlag)
```

函数描述：

这个函数用于单精度浮点实数 FFT。

函数参数：

- ◆ 第 1 个参数是封装好的浮点 FFT 例化，需要用户先调用函数 arm_rfft_fast_init_f32 初始化，然后供此函数 arm_rfft_fast_f32 调用。支持 32, 64, 128, 256, 512, 1024, 2048, 4096 点 FFT。
比如做 1024 点 FFT，代码如下：

```
arm_rfft_fast_instance_f32 S;
arm_rfft_fast_init_f32(&S, 1024);
arm_rfft_fast_f32(&S, testInput_f32, testOutput_f32, ifftFlag);
```

- ◆ 第 2 个参数是实数地址，比如我们要做 1024 点实数 FFT，要保证有 1024 个缓冲。
- ◆ 第 3 个参数是 FFT 转换结果，转换结果不是实数了，而是复数，按照实部，虚拟，实部，虚拟，依次排列。比如做 1024 点 FFT，这里的输出也会有 1024 个数据，即 512 个复位。
- ◆ 第 4 个参数用于设置正变换和逆变换，ifftFlag=0 表示正变换，ifftFlag=1 表示逆变换。

32.4.2 使用举例

下面通过函数 arm_rfft_fast_f32 将正弦波做 FFT 变换，并再次通过函数 arm_rfft_fast_f32 做 FFT 逆变换来比较原始波形和转换后波形效果。

```
/*
*****
* 函数名：arm_rfft_f32_app
* 功能说明：调用函数 arm_rfft_fast_f32 计算幅频和相频
* 形参：无
* 返回值：无
*****
*/
static void arm_rfft_f32_app(void)
{
    uint16_t i;
    arm_rfft_fast_instance_f32 S;

    /* 正变换 */
}
```



```
ifftFlag = 0;

/* 初始化结构体 S 中的参数 */
arm_rfft_fast_init_f32(&S, TEST_LENGTH_SAMPLES);

for(i=0; i<1024; i++)
{
    /* 波形是由直流分量, 50Hz 正弦波组成, 波形采样率 1024, 初始相位 60° */
    testInput_f32[i] = 1 + cos(2*3.1415926f*50*i/1024 + 3.1415926f/3);
}

/* 1024 点实序列快速 FFT */
arm_rfft_fast_f32(&S, testInput_f32, testOutput_f32, ifftFlag);

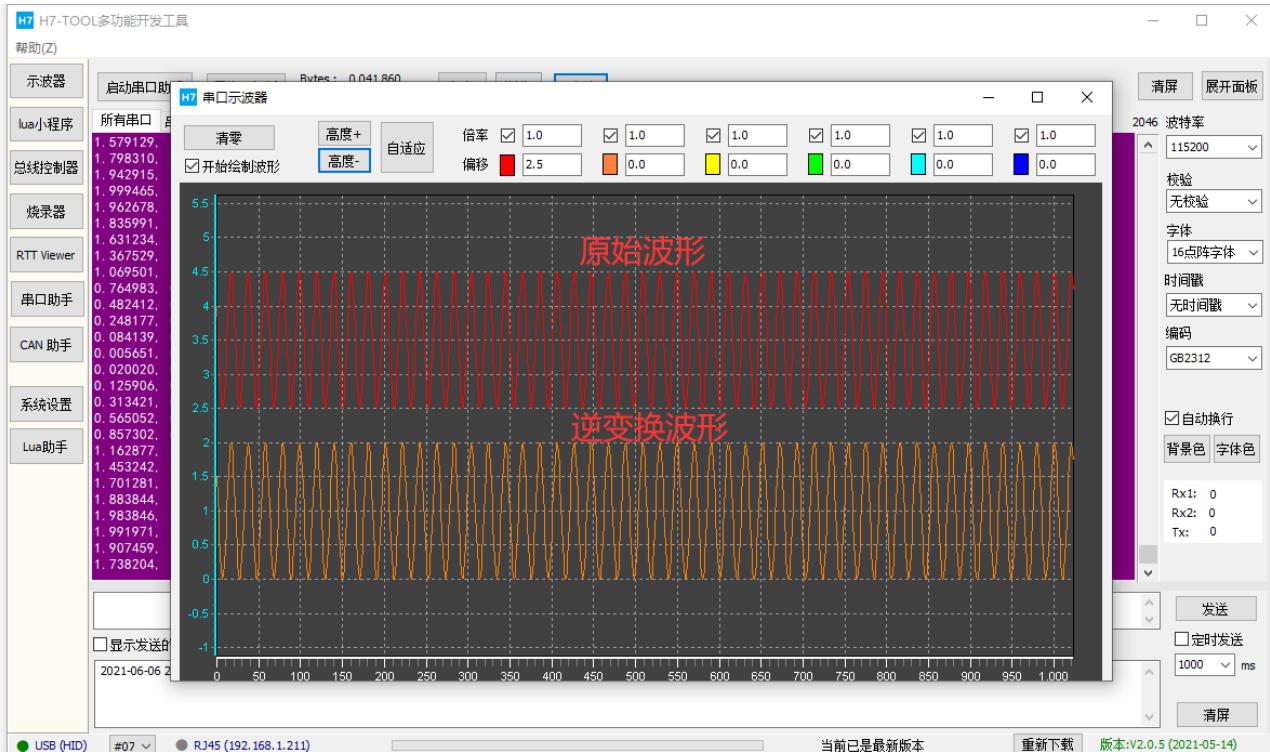
/* 为了方便跟函数 arm_cfft_f32 计算的结果做对比, 这里求解了 1024 组模值, 实际函数 arm_rfft_fast_f32
只求解出了 512 组
*/
arm_cmplx_mag_f32(testOutput_f32, testOutputMag_f32, TEST_LENGTH_SAMPLES);

printf("=====\\r\\n");

/* 求相频 */
PowerPhaseRadians_f32(testOutput_f32, Phase_f32, TEST_LENGTH_SAMPLES, 0.5f);

/* 串口打印求解的幅频和相频 */
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    printf("%f, %f\\r\\n", testOutputMag_f32[i], Phase_f32[i]);
}
```

运行函数arm_rfft_f32_app可以通过串口打印原始波形和还原后波形效果：



从上面的对比结果中可以看出原始波形和还原后的波形是一致的。



32.5 双精度函数 arm_rfft_fast_f64 实现 FFT 正变换和逆变换

32.5.1 函数说明

函数原型：

```
void arm_rfft_fast_f64(
    arm_rfft_fast_instance_f64 * S,
    float64_t * p,
    float64_t * pOut,
    uint8_t ifftFlag)
```

函数描述：

这个函数用于双精度浮点实数 FFT。

函数参数：

- ◆ 第 1 个参数是封装好的浮点 FFT 例化，需要用户先调用函数 arm_rfft_fast_init_f64 初始化，然后供此函数 arm_rfft_fast_f64 调用。支持 32, 64, 128, 256, 512, 1024, 2048, 4096 点 FFT。
比如做 1024 点 FFT，代码如下：

```
arm_rfft_fast_instance_f64 S;
arm_rfft_fast_init_f64(&S, 1024);
arm_rfft_fast_f64(&S, testInput_f64, testOutput_f64, ifftFlag);
```

- ◆ 第 2 个参数是实数地址，比如我们要做 1024 点实数 FFT，要保证有 1024 个缓冲。
- ◆ 第 3 个参数是 FFT 转换结果，转换结果不是实数了，而是复数，按照实部，虚拟，实部，虚拟，依次排列。比如做 1024 点 FFT，这里的输出也会有 1024 个数据，即 512 个复位。
- ◆ 第 4 个参数用于设置正变换和逆变换，ifftFlag=0 表示正变换，ifftFlag=1 表示逆变换。

32.5.2 使用举例

下面通过函数 arm_rfft_fast_f64 将正弦波做 FFT 变换，并再次通过函数 arm_rfft_fast_f64 做 FFT 逆变换来比较原始波形和转换后波形效果：

```
/*
*****
* 函数名: arm_rfft_f64_app
* 功能说明: 调用函数arm_rfft_fast_f64计算FFT逆变换和正变换
* 形参: 无
* 返回值: 无
*****
*/
static void arm_rfft_f64_app(void)
{
    uint16_t i;
    arm_rfft_fast_instance_f64 S;

    /* 正变换 */
    ifftFlag = 0;
```



```
/* 初始化结构体S中的参数 */
arm_rfft_fast_init_f64(&S, TEST_LENGTH_SAMPLES);

for(i=0; i<1024; i++)
{
    /* 波形是由直流分量, 50Hz正弦波组成, 波形采样率1024, 初始相位60° */
    testInput_f64[i] = 1 + cos(2*3.1415926*50*i/1024 + 3.1415926/3);
    testOutputIn_f64[i] = testInput_f64[i];
}

/* 1024点实序列快速FFT, testInput_f64是输入数据, testOutput_f64是输出 */
arm_rfft_fast_f64(&S, testInput_f64, testOutput_f64, ifftFlag);

/* 逆变换 */
ifftFlag = 1;

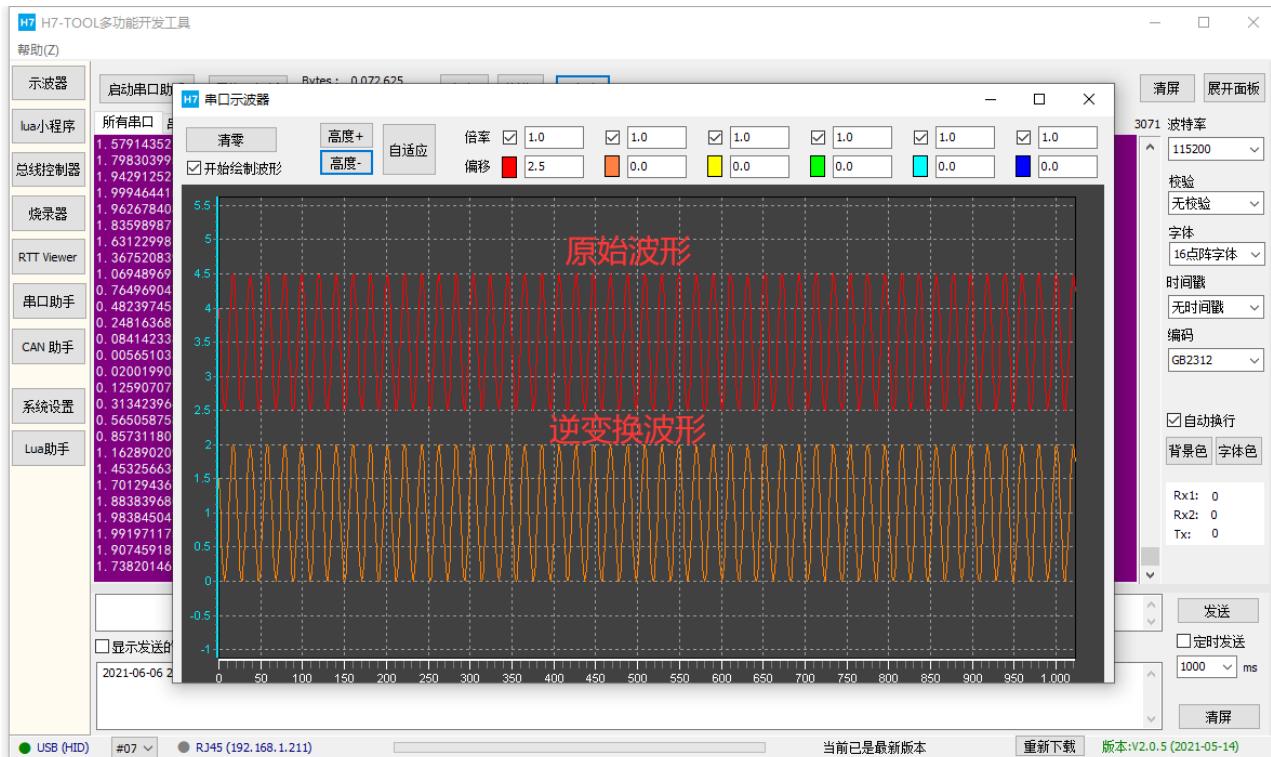
/* 1024点实序列快速FFT逆变换, testOutput_f64是输入数据, testInput_f64是输出数据 */
arm_rfft_fast_f64(&S, testOutput_f64, testInput_f64, ifftFlag);

printf("=====\\r\\n");

/* 串口打印, testOutputIn_f32原始信号, testInput_f32逆变换后的信号 */
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    printf("%.11f, %.11f\\r\\n", testOutputIn_f64[i], testInput_f64[i]);
}

}
```

运行函数arm_rfft_f64_app可以通过串口打印原始波形和还原后波形效果：



从上面的对比结果中可以看出原始波形和还原后的波形是一致的。

32.6 实验例程说明 (MDK)

配套例子：



V7-222_实数浮点 FFT 逆变换(支持单精度和双精度)

实验目的：

1. 学习实数浮点 FFT 逆变换，支持单精度浮点和双精度浮点

实验内容：

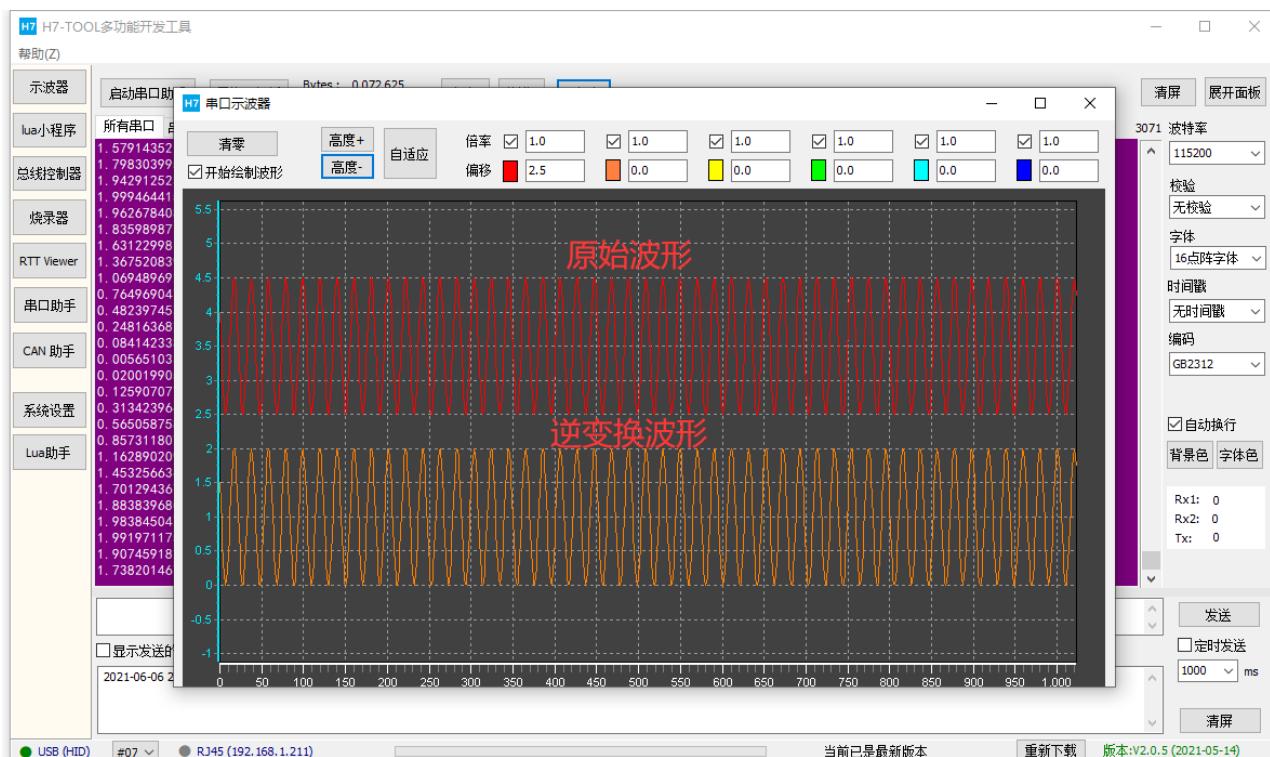
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打印 1024 点实数单精度 FFT 逆变换。
3. 按下按键 K2，串口打印 1024 点实数双精度 FFT 逆变换。

使用 AC6 注意事项

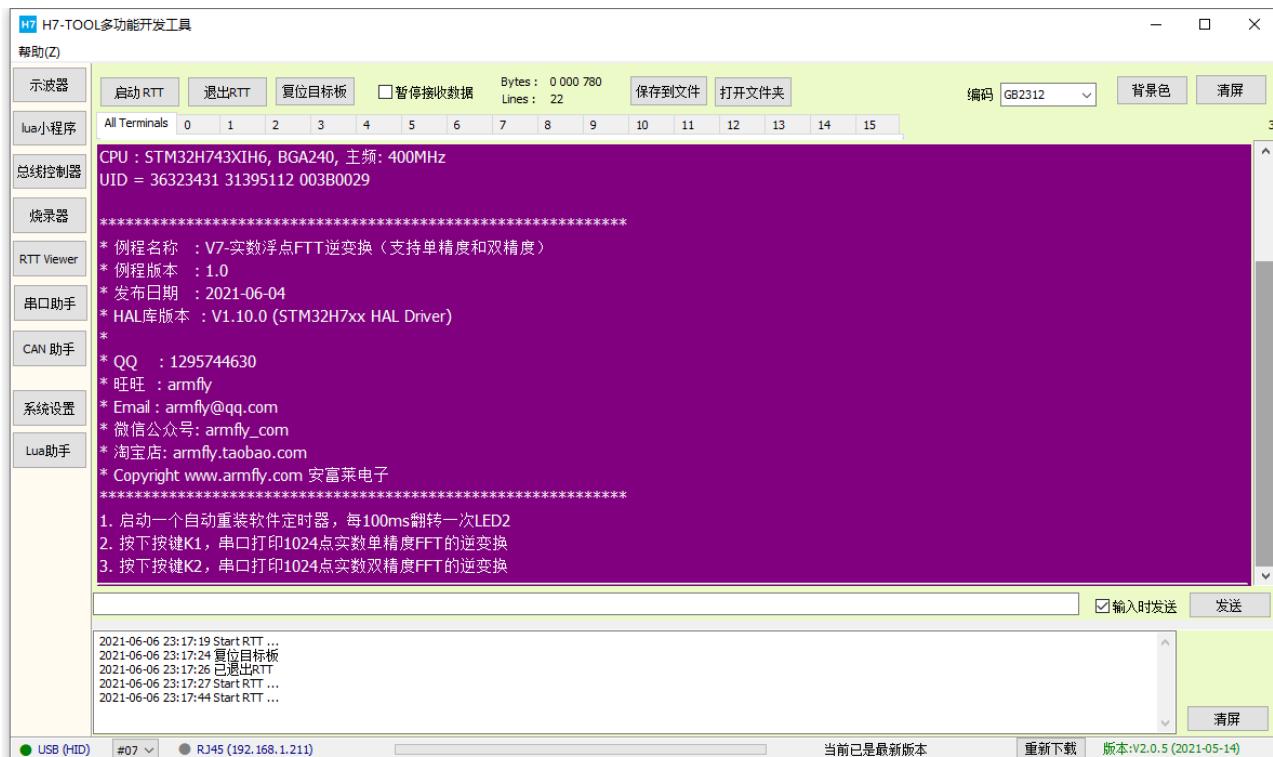
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

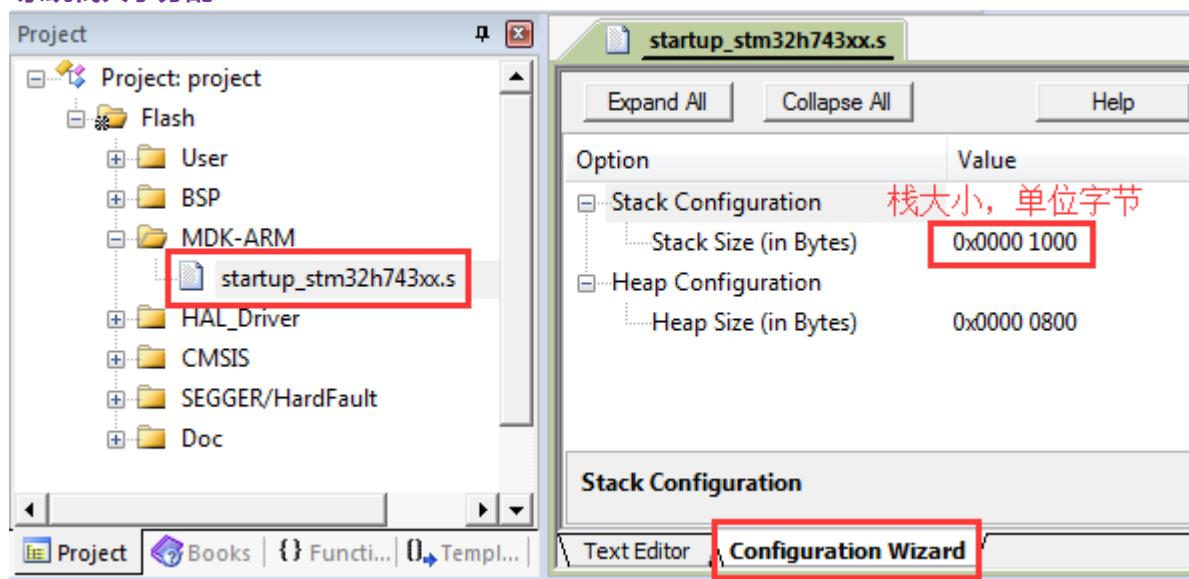


RTT 方式打印信息：

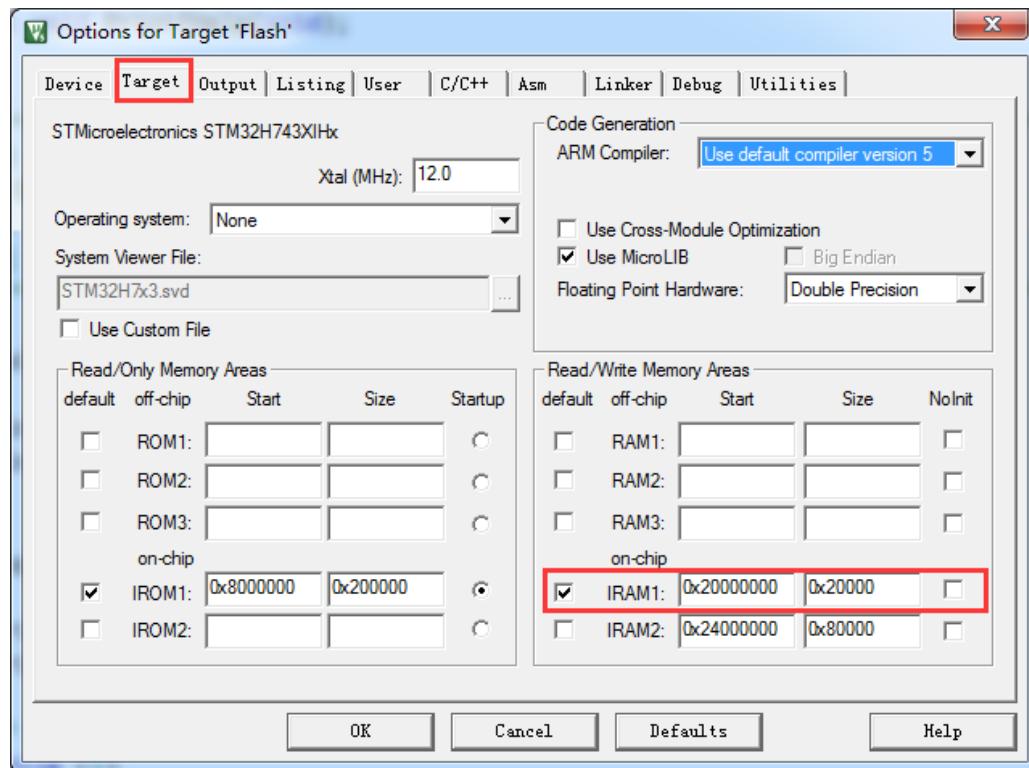


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印 1024 点实数单精度 FFT 逆变换。
- 按下按键 K2，串口打印 1024 点实数双精度 FFT 逆变换。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(4); /* 翻转 LED2 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_rfft_f32_app();
                break;

            case KEY_DOWN_K2:          /* K2 键按下 */
                arm_rfft_f64_app();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
```

32.7 实验例程说明 (IAR)

配套例子：

V7-222_实数浮点 FFT 逆变换(支持单精度和双精度)

实验目的：

1. 学习实数浮点 FFT 逆变换，支持单精度浮点和双精度浮点

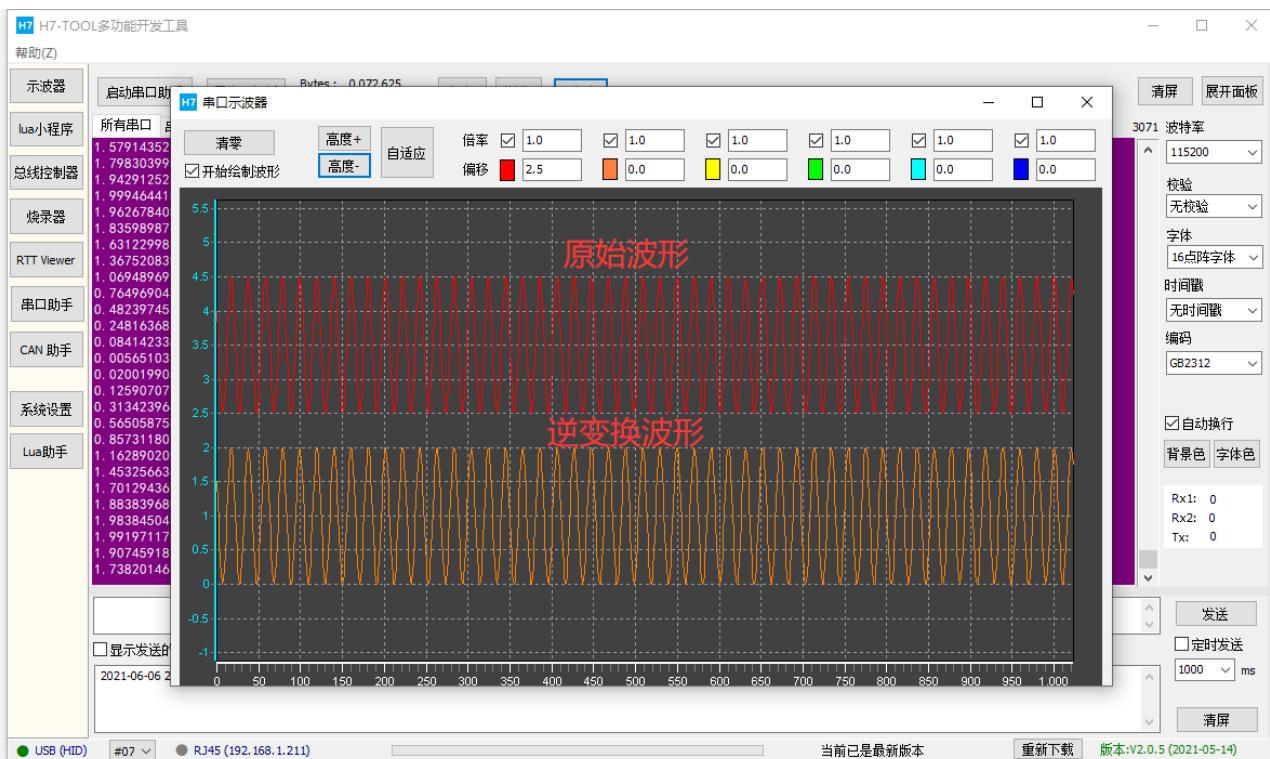
实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打印 1024 点实数单精度 FFT 逆变换。
3. 按下按键 K2，串口打印 1024 点实数双精度 FFT 逆变换。

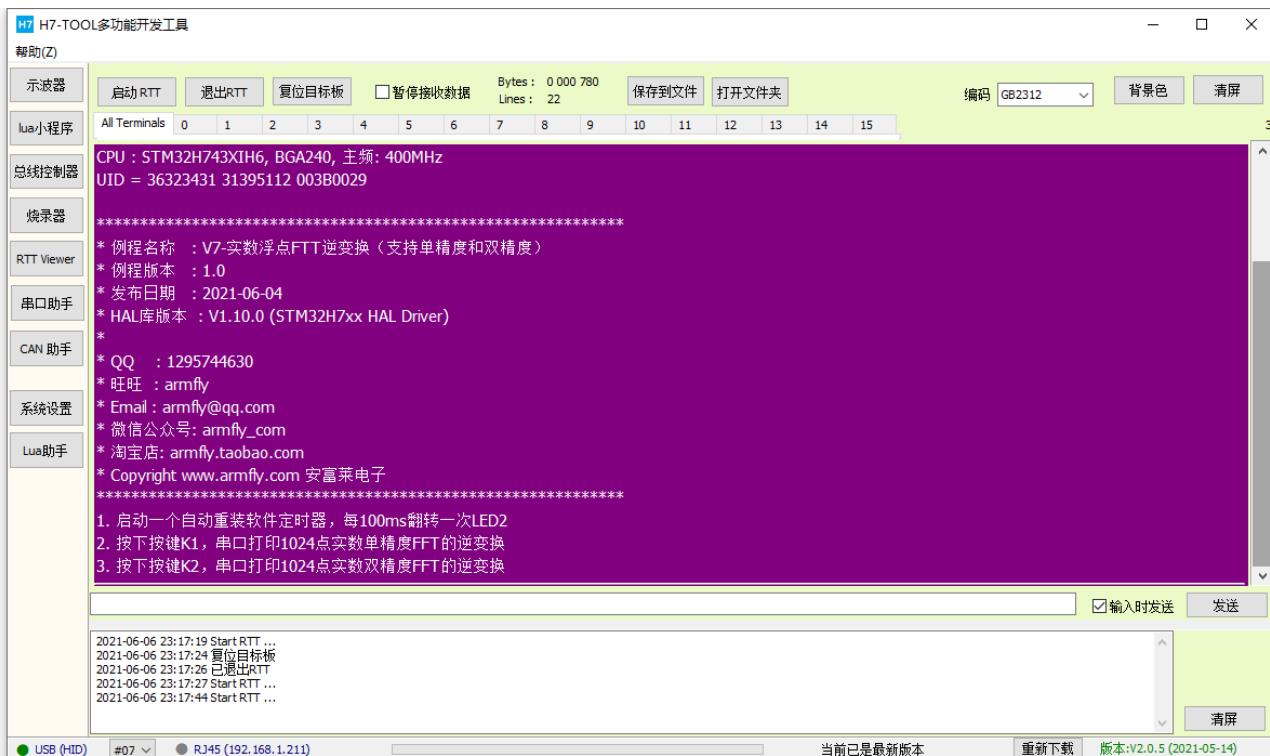
上电后串口打印的信息：



波特率 115200，数据位 8，奇偶校验位无，停止位 1。

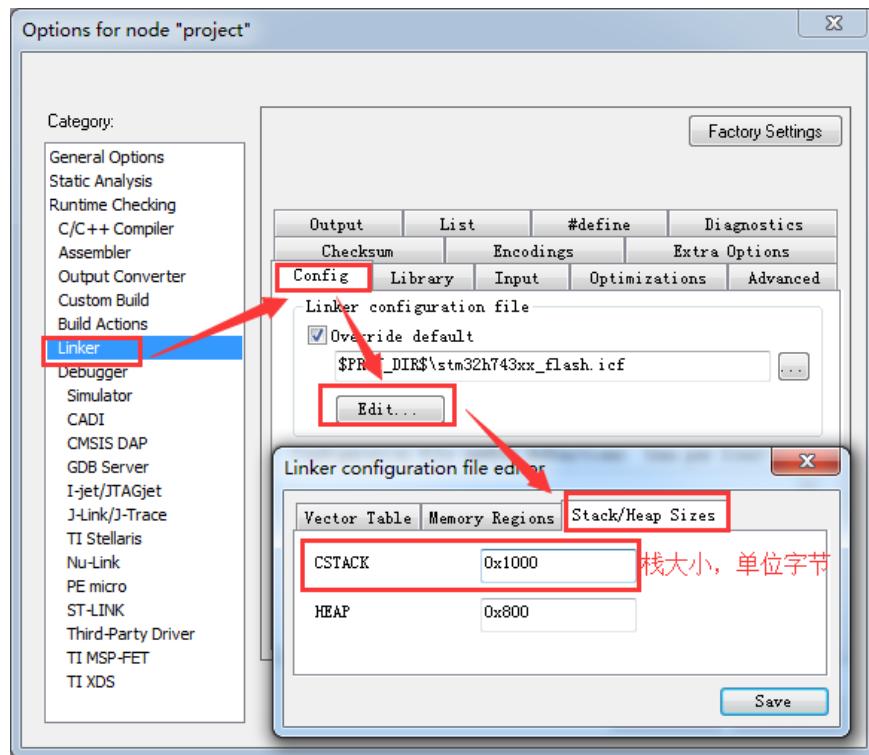


RTT 方式打印信息：

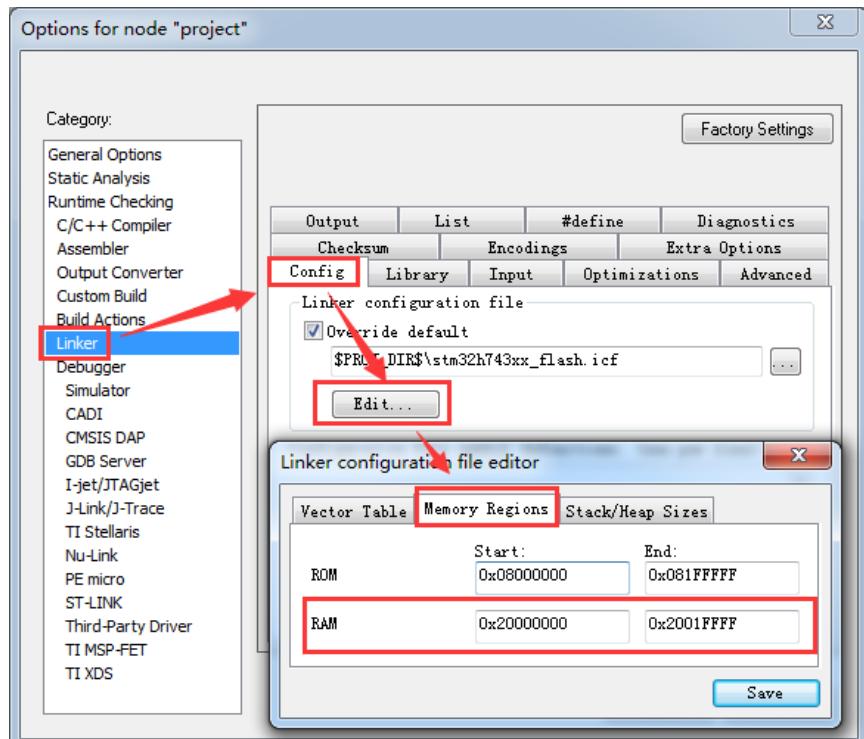


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
*****
```



```
/* 功能说明：初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形    参：无
* 返回值：无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
     STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
     - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
     - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
     配置系统时钟到 400MHz
     - 切换使用 HSE。
     - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
     Event Recorder:
     - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
     - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名：MPU_Config
* 功能说明：配置 MPU
* 形    参：无
* 返回值：无
*****
*/
static void MPU_Config( void )
```



```
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /*使能 MPU */  
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);  
}  
  
/*  
*****  
* 函数名: CPU_CACHE_Enable  
* 功能说明: 使能 L1 Cache  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void CPU_CACHE_Enable(void)  
{  
    /* 使能 I-Cache */  
    SCB_EnableICache();  
  
    /* 使能 D-Cache */  
    SCB_EnableDCache();  
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印 1024 点实数单精度 FFT 逆变换。
- 按下按键 K2，串口打印 1024 点实数双精度 FFT 逆变换。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
***** */

int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(4); /* 翻转 LED2 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_rfft_f32_app();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下 */
                    arm_rfft_f64_app();
                    break;

                default:
                    /* 其它的键值不处理 */
            }
        }
    }
}
```



```
        break;  
    }  
}  
}
```

32.8 总结

本章节主要验证了函数 arm_rfft_fast_f32 正变换和逆变换，有兴趣的可以验证 Q31 和 Q15 两种数据类型的正变换和逆变换。



第33章 STM32H7 不限制点数 FFT 实现

本章主要讲解不限制点数 FFT 的实现。

33.1 初学者重要提示

33.2 不限制点数 FFT 移植

33.3 不限制点数 FFT 应用说明

33.4 实验例程说明(MDK)

33.5 实验例程说明(IAR)

33.6 总结

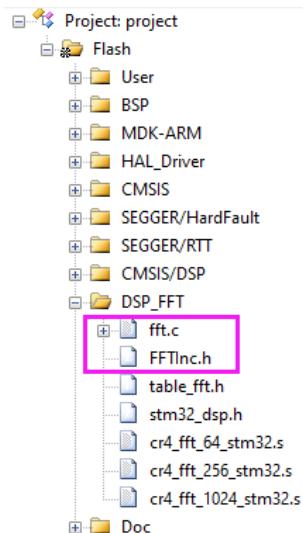
33.1 初学者重要提示

- ◆ 由于 ARM DSP 库限制最大只能 4096 点，有点不够用，所以整理了个更大点数的。不限制点数，满足 2^n 即可，n 最小值 4，即 16 个点的 FFT，而最大值不限。
- ◆ 此 FFT 算法没有再使用 ARM DSP 库，重新实现了一个。

33.2 不限制点数 FFT 移植

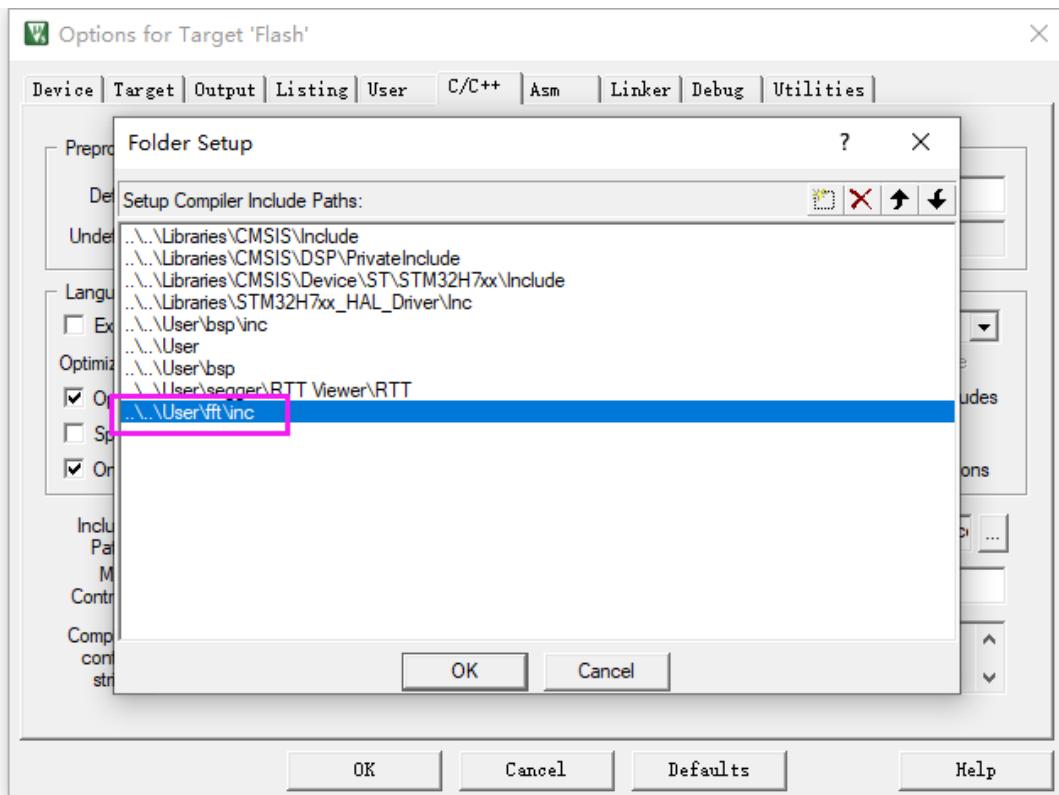
33.2.1 移植 FFT 相关文件

移植下面两个文件 fft.c 和 FFTInc.h 到工程：



33.2.2 添加路径

添加路径，大家根据自己的工程来设置即可：



33.3 不限制点数 FFT 应用说明

33.3.1 支持的点数范围

支持最小 16 点 FFT，最大值不限，但需满足 2^n 。

33.3.2 函数 InitTableFFT

函数原型：

```
/*
*****
* 函数名: Int_FFT_TAB
* 功能说明: 正弦和余弦表
* 形参: 点数
* 返回值: 无
*****
*/
#ifndef MAX_FFT_N
#define MAX_FFT_N 8192
#endif
float32_t costab[MAX_FFT_N/2];
```



```
float32_t    sintab[MAX_FFT_N/2];
void InitTableFFT(uint32_t n)
{
    uint32_t i;

/* 正常使用下面获取 cos 和 sin 值 */
#ifndef __FPU_PRESENT
    if (n <= 8192)
    {
        for (i = 0; i < n; i++)
        {
            sintab[i] = sin( 2 * PI * i / MAX_FFT_N );
            costab[i] = cos( 2 * PI * i / MAX_FFT_N );
        }
    }

/* 打印出来是方便整理 cos 值和 sin 值数组，将其放到内部 Flash，从而节省 RAM */
#else
    printf("const float32_t sintab[%d] = {\r\n", n);
    for (i = 0; i < n; i++)
    {
        sintab[i] = sin( 2 * PI * i / MAX_FFT_N );
        printf("%.11ff,\r\n", sintab[i]);
    }
    printf("};\r\n");

    printf("const float32_t costab[%d] = {\r\n", n);
    for (i = 0; i < n; i++)
    {
        costab[i] = cos( 2 * PI * i / MAX_FFT_N );
        printf("%.11ff,\r\n", costab[i]);
    }
    printf("};\r\n");
#endif
}
#endif
```

函数描述：

这个函数用于 FFT 计算过程中需要用到的正弦和余弦表。对于 8192 点和 16384 点已经专门制作了数值表，存到内部 Flash，其它点数继续使用的 RAM 空间，大家可以根据所使用芯片的 RAM 和 Flash 大小，选择正弦和余弦值存到 RAM 还是 Flash。

函数参数：

- ◆ 第 1 个参数是 FFT 点数。

33.3.3 函数 cfft

函数原型：

```
void cfft(struct compx *_ptr, uint32_t FFT_N )

/*
*****
*  函数名: cfft
*  功能说明: 对输入的复数组进行快速傅里叶变换 (FFT)
*  形    参: *_ptr 复数结构体组的首地址指针 struct 型
*          FFT_N 表示点数
*****
```



```
*  返回 值: 无
*****
*/
void cfft(struct compx *_ptr, uint32_t FFT_N )
{
    float32_t TempReal1, TempImag1, TempReal2, TempImag2;
    uint32_t k, i, j, z;
    uint32_t Butterfly_NoPerColumn;           /* 每级蝶形的蝶形组数 */
    uint32_t Butterfly_NoOfGroup;            /* 蝶形组的第几组 */
    uint32_t Butterfly_NoPerGroup;           /* 蝶形组的第几个蝶形 */
    uint32_t ButterflyIndex1,ButterflyIndex2,P,J;
    uint32_t L;
    uint32_t M;

    z=FFT_N/2;      /* 变址运算, 即把自然顺序变成倒位序, 采用雷德算法 */

    for(i=0, j=0; i<FFT_N-1; i++)
    {
        /*
         * 如果 i<j, 即进行变址 i=j 说明是它本身, i>j 说明前面已经变换过了, 不许再变化, 注意这里一般是实数
         * 有虚数部分 设置成结合体
        */
        if(i<j)
        {
            TempReal1 = _ptr[j].real;
            _ptr[j].real= _ptr[i].real;
            _ptr[i].real= TempReal1;
        }

        k=z;          /*求 j 的下一个倒位序 */

        while(k<=j)  /* 如果 k<=j, 表示 j 的最高位为 1 */
        {
            j=j-k; /* 把最高位变成 0 */
            k=k/2; /* k/2, 比较次高位, 依次类推, 逐个比较, 直到某个位为 0, 通过下面那句 j=j+k 使其变为 1 */
        }

        j=j+k;      /* 求下一个反序号, 如果是 0, 则把 0 改为 1 */
    }

/* 第 L 级蝶形(M)第 Butterfly_NoOfGroup 组(Butterfly_NoPerColumn)第 J 个蝶形(Butterfly_NoPerGroup)***** */
/* 蝶形的组数以 2 的倍数递减 Butterfly_NoPerColumn, 每组中蝶形的个数以 2 的倍数递增 Butterfly_NoPerGroup */
/* 在计算蝶形时, 每 L 列的蝶形组数, 一共有 M 列, 每组蝶形中蝶形的个数, 蝶形的阶数(0, 1, 2, ..., M-1) */

    Butterfly_NoPerColumn = FFT_N;
    Butterfly_NoPerGroup = 1;
    M = log2(FFT_N);
    for ( L = 0; L < M; L++ )
    {
        Butterfly_NoPerColumn >>= 1;          /* 蝶形组数 假如 N=8, 则(4, 2, 1) */

        //第 L 级蝶形 第 Butterfly_NoOfGroup 组 (0, 1, ..., Butterfly_NoOfGroup-1)
        for ( Butterfly_NoOfGroup = 0; Butterfly_NoOfGroup < Butterfly_NoPerColumn; Butterfly_NoOfGroup++ )
        {
            /* 第 Butterfly_NoOfGroup 组中的第 J 个 */
            for ( J = 0; J < Butterfly_NoPerGroup; J++ )
            {
                /* 第 ButterflyIndex1 和第 ButterflyIndex2 个元素作蝶形运算, WNC */
                /* (0, 2, 4, 6) (0, 1, 4, 5) (0, 1, 2, 3) */
                /* 两个要做蝶形运算的数相距 Butterfly_NoPerGroup, ge=1, 2, 4 */
            }
        }
    }
}
```



```
/* 这里相当于 P=J*2^(M-L), 做了一个换算下标都是 N (0, 0, 0, 0) (0, 2, 0, 2) (0, 1, 2, 3) */
ButterflyIndex1 = ( ( Butterfly_NoOfGroup * Butterfly_NoPerGroup ) << 1 ) + J;
ButterflyIndex2 = ButterflyIndex1 + Butterfly_NoPerGroup;
P = J * Butterfly_NoPerColumn;

/* 计算和转换因子乘积 */
TempReal2 = _ptr[ButterflyIndex2].real * costab[ P ] + _ptr[ButterflyIndex2].imag *
            sintab[ P ];
TempImag2 = _ptr[ButterflyIndex2].imag * costab[ P ] - _ptr[ButterflyIndex2].real *
            sintab[ P ];
TempReal1 = _ptr[ButterflyIndex1].real;
TempImag1 = _ptr[ButterflyIndex1].imag;

/* 蝶形运算 */
_ptr[ButterflyIndex1].real = TempReal1 + TempReal2;
_ptr[ButterflyIndex1].imag = TempImag1 + TempImag2;
_ptr[ButterflyIndex2].real = TempReal1 - TempReal2;
_ptr[ButterflyIndex2].imag = TempImag1 - TempImag2;
}

}

Butterfly_NoPerGroup<<=1; /* 一组中蝶形的个数(1, 2, 4) */
}
}
```

函数描述：

这个函数用于复数 FFT 变换。

函数参数：

- ◆ 第 1 个参数是复数格式。
- ◆ 第 2 个参数是 FFT 点数，最小值 16，最大值不限，满足满足 2^n 即可。

33.3.4 FFT 幅频响应举例

下面通过函数cff将正弦波做16384点FFT变换：

```
/*
*****
* 函数名: cfft_f32_mag
* 功能说明: 计算幅频
* 形参: 无
* 返回值: 无
*****
*/
static void cfft_f32_mag(void)
{
    uint32_t i;

    /* 计算一批sin, cos系数 */
#if (MAX_FFT_N != 8192) && (MAX_FFT_N != 16384)
    InitTableFFT(MAX_FFT_N);
#endif

    for(i=0; i<MAX_FFT_N; i++)
    {
        /* 波形是由直流分量, 500Hz正弦波组成, 波形采样率MAX_FFT_N, 初始相位60° */
        s[i].real = 1 + cos(2*3.1415926f*500*i/MAX_FFT_N + 3.1415926f/3);
        s[i].imag = 0;
    }
}
```

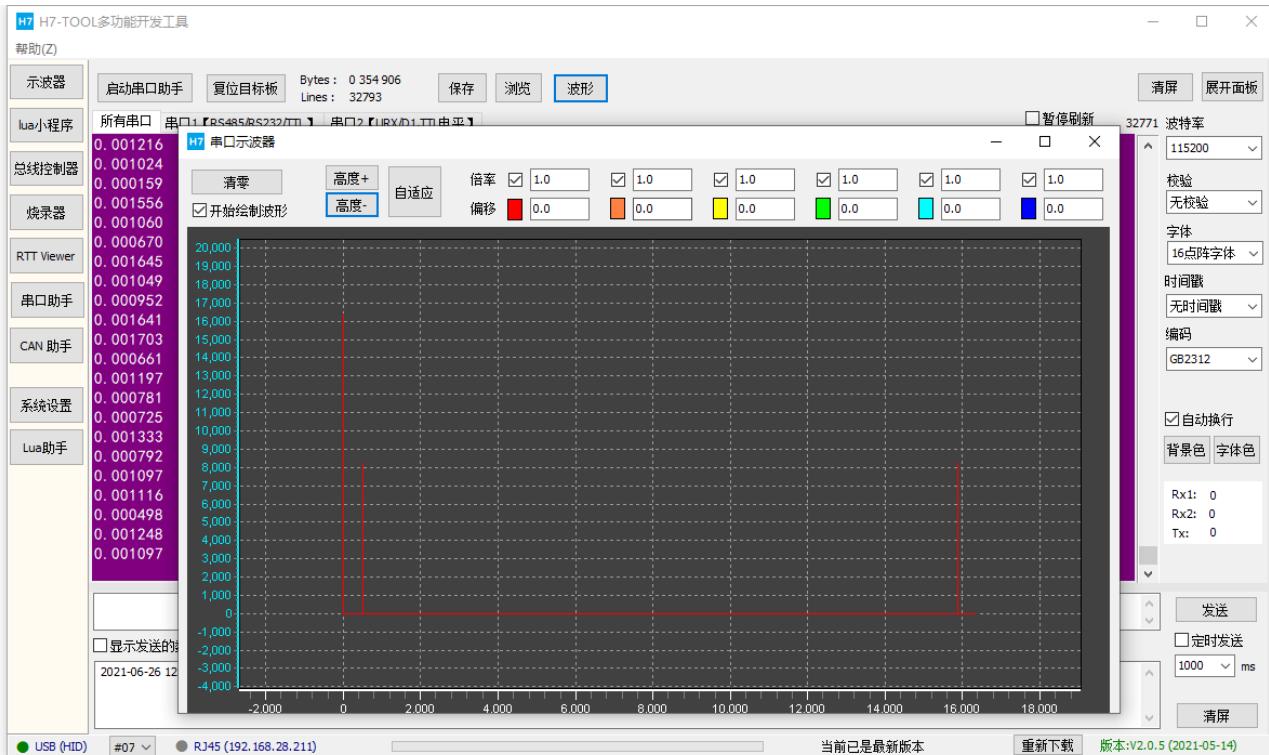


```
/* MAX_FFT_N点快速FFT */
cfft(s, MAX_FFT_N);

/* 计算幅频 */
for(i=0; i<MAX_FFT_N; i++)
{
    arm_sqrt_f32((float32_t)(s[i].real * s[i].real+ s[i].imag*s[i].imag ), &s[i].real);
}

/* 串口打印求解的幅频 */
for(i=0; i<MAX_FFT_N; i++)
{
    printf("%f\r\n", s[i].real);
}
```

运行函数cfft_f32_mag可以通过串口打印FFT结果：



从上面的结果中可以出直流分量和正弦波幅值都是没有问题的。

33.3.5 FFT 相频响应举例

下面通过函数cff将正弦波做16384点FFT变换：

```
/*
*****
* 函数名: PowerPhaseRadians_f32
* 功能说明: 求相位
* 形参: _usFFTPoints 复数个数, 每个复数是两个float32_t数值
*       _uiCmpValue 比较值, 需要求出相位的数值
* 返回值: 无
*****
*/
void PowerPhaseRadians_f32(uint16_t _usFFTPoints, float32_t _uiCmpValue)
{
    float32_t 1X, 1Y;
```



```
uint32_t i;
float32_t phase;
float32_t mag;

for (i=0; i <_usFFTPoints; i++)
{
    lX= s[i].real;      /* 实部 */
    lY= s[i].imag;     /* 虚部 */

    phase = atan2f(lY, lX);           /* atan2求解的结果范围是(-pi, pi], 弧度制 */
    arm_sqrt_f32((float32_t)(lX*lX+ lY*lY), &mag); /* 求模 */

    if(_uiCmpValue > mag)
    {
        s[i].real = 0;
    }
    else
    {
        s[i].real= phase* 180.0f/3.1415926f; /* 将求解的结果由弧度转换为角度 */
    }
}

/*
***** 函数名: cfft_f32_phase
* 功能说明: 计算相频
* 形参: 无
* 返回值: 无
*****
*/
static void cfft_f32_phase(void)
{
    uint32_t i;

    /* 计算一批sin, cos系数 */
#if (MAX_FFT_N != 8192) && (MAX_FFT_N != 16384)
    InitTableFFT(MAX_FFT_N);
#endif

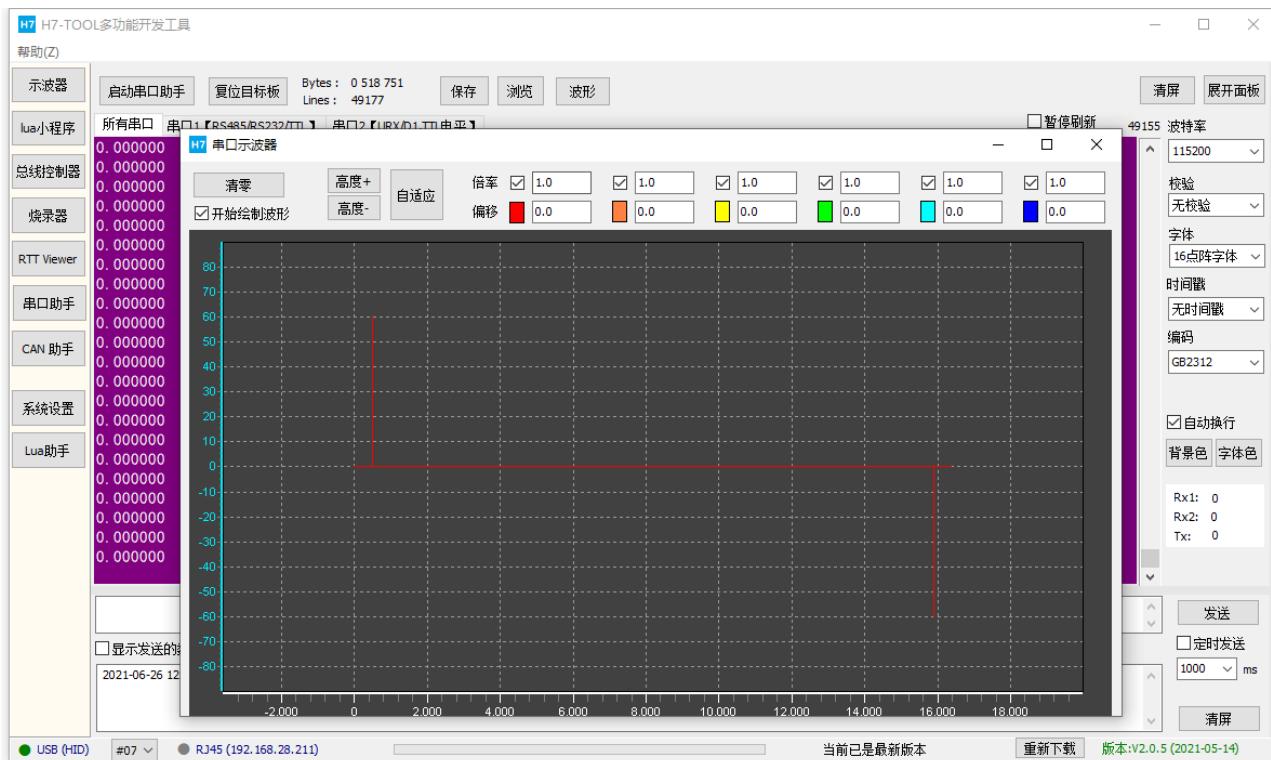
    for(i=0; i<MAX_FFT_N; i++)
    {
        /* 波形是由直流分量, 500Hz正弦波组成, 波形采样率MAX_FFT_N, 初始相位60° */
        s[i].real = 1 + cos(2*3.1415926f*500*i/MAX_FFT_N + 3.1415926f/3);
        s[i].imag = 0;
    }

    /* MAX_FFT_N点快速FFT */
    cfft(s, MAX_FFT_N);

    /* 求相频 */
    PowerPhaseRadians_f32(MAX_FFT_N, 0.5f);

    /* 串口打印求解相频 */
    for(i=0; i<MAX_FFT_N; i++)
    {
        printf("%f\r\n", s[i].real);
    }
}
```

运行函数cfft_f32_phase可以通过串口打印FFT结果:



从上面的结果中可以得出计算的初始相位是没有问题的。

33.4 实验例程说明 (MDK)

配套例子：

V7-223_不限制点数 FFT 实现

实验目的：

1. 学习不限制点数 FFT。

实验内容：

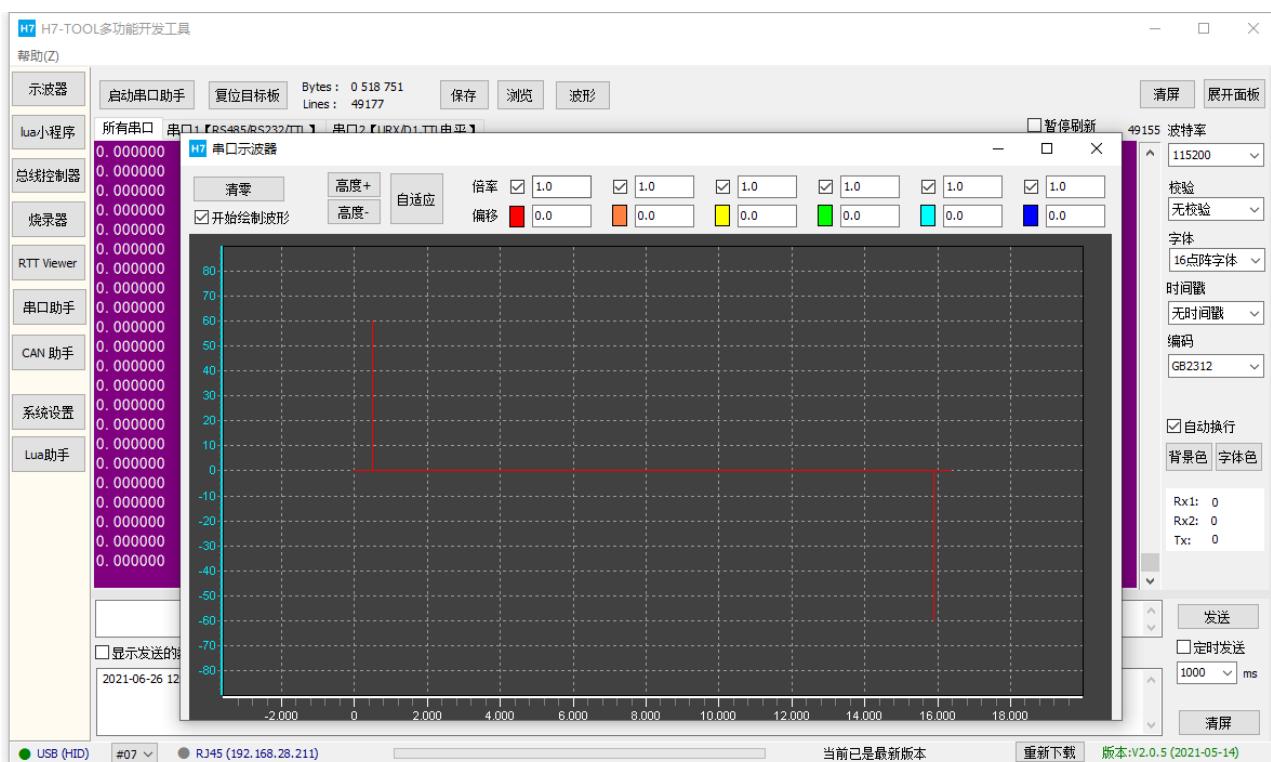
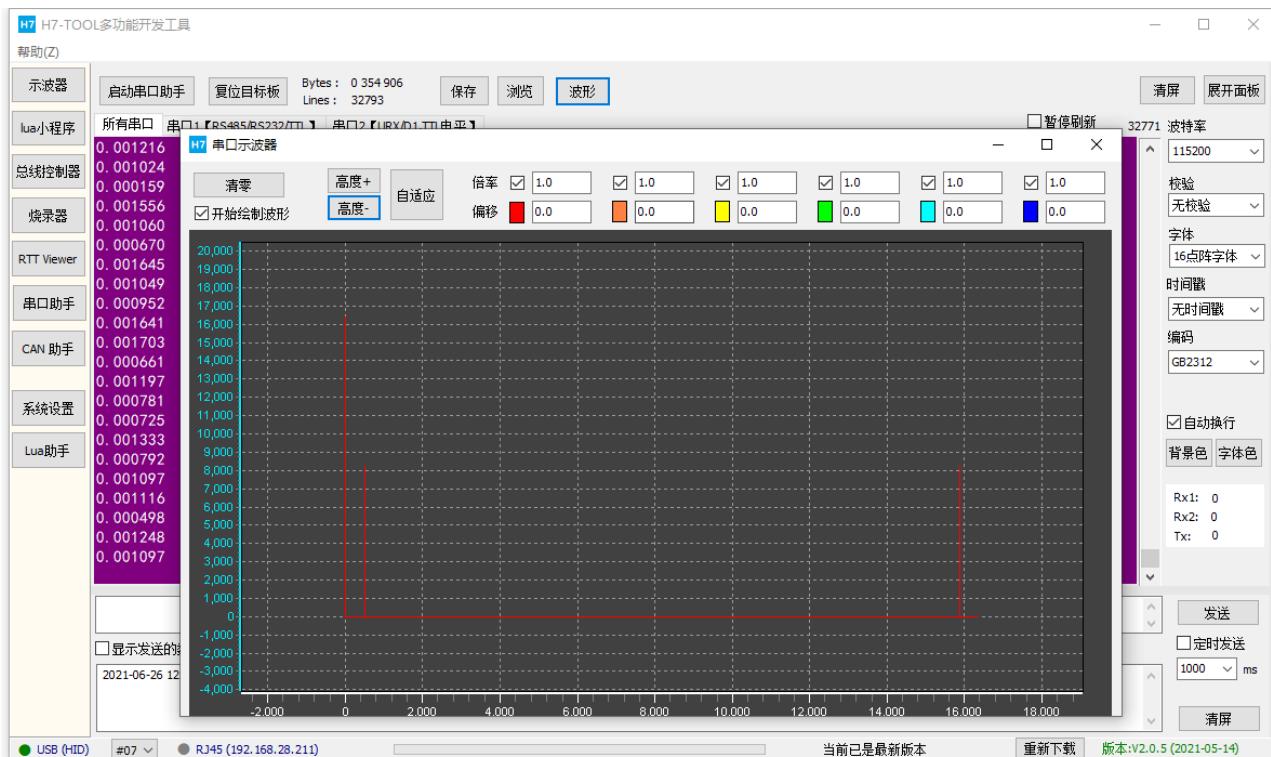
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打印 16384 点 FFT 的幅频响应。
3. 按下按键 K2，串口打印 16384 点 FFT 的相频响应。

使用 AC6 注意事项

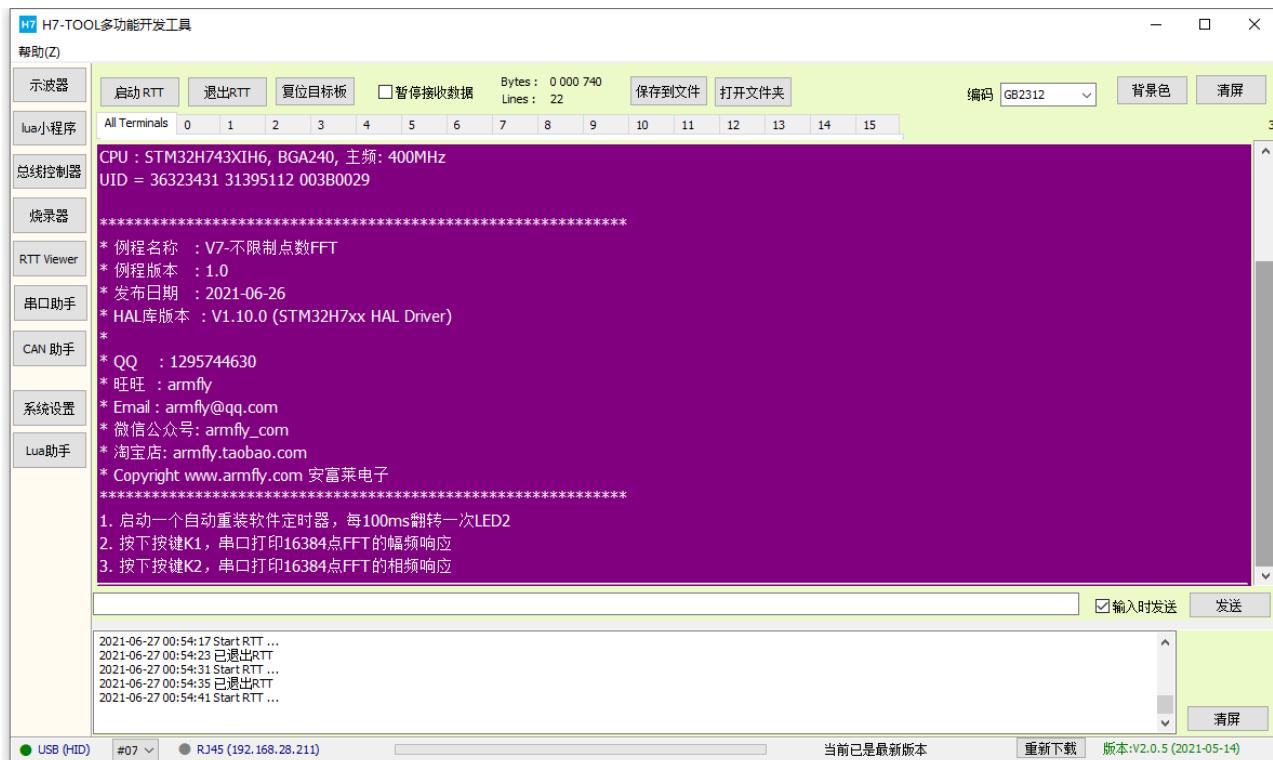
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

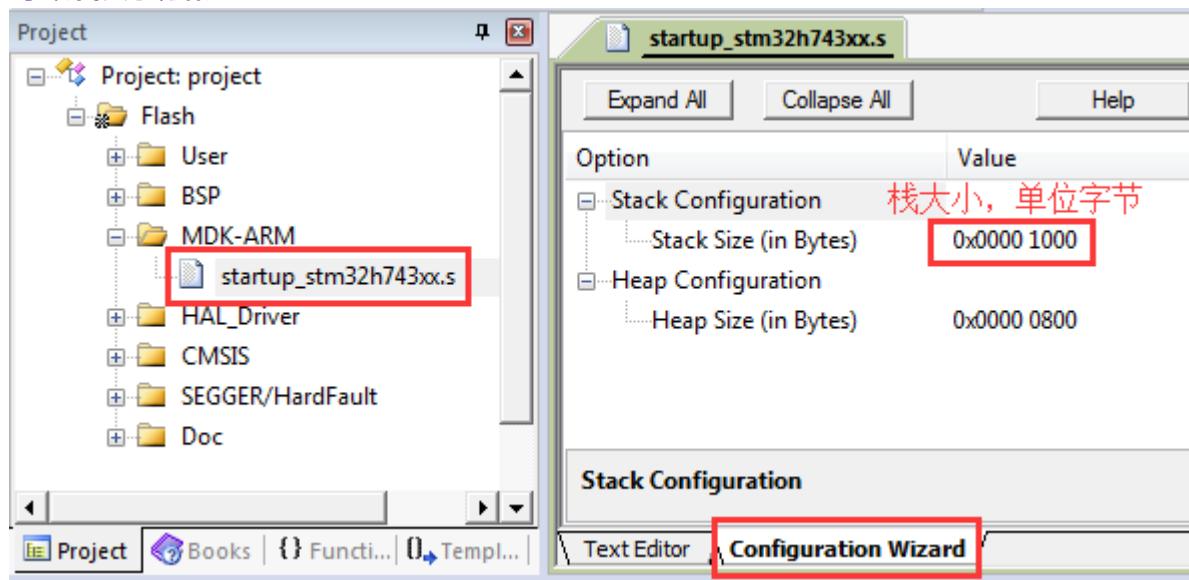


RTT 方式打印信息：

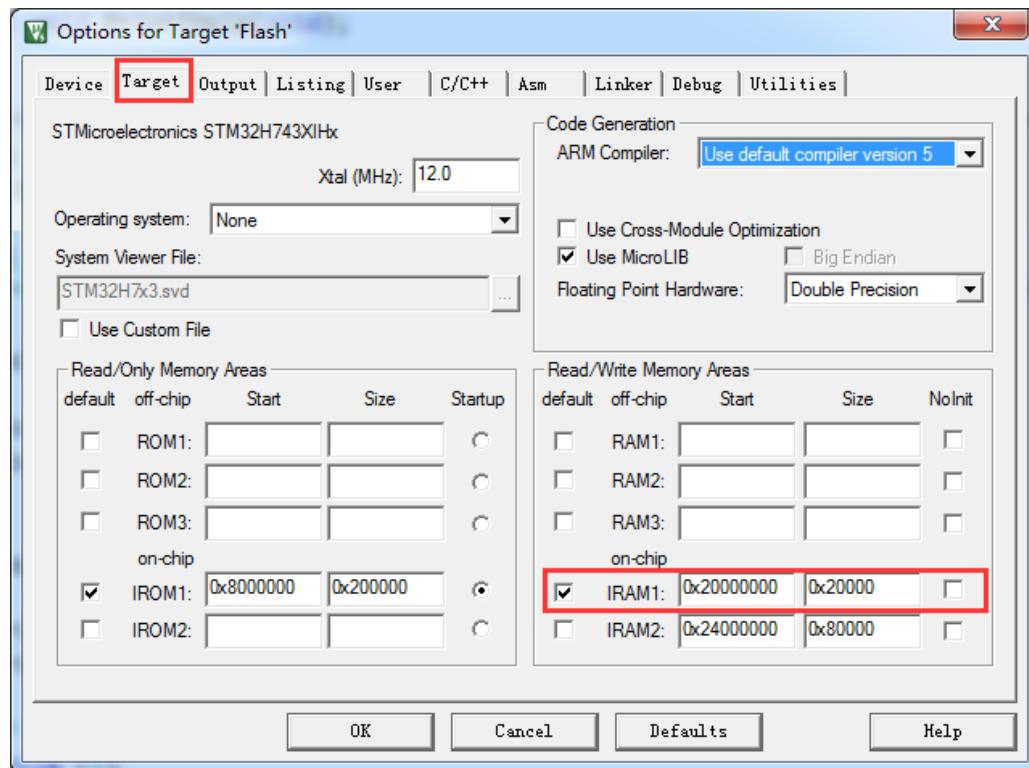


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印 16384 点 FFT 的幅频响应。
- 按下按键 K2，串口打印 16384 点 FFT 的相频响应。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(4); /* 翻转 LED2 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                cfft_f32_mag();
                break;

            case KEY_DOWN_K2:          /* K2 键按下 */
                cfft_f32_phase();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
```

33.5 实验例程说明 (IAR)

配套例子：

V7-223_不限制点数 FFT 实现

实验目的：

1. 学习不限制点数 FFT。

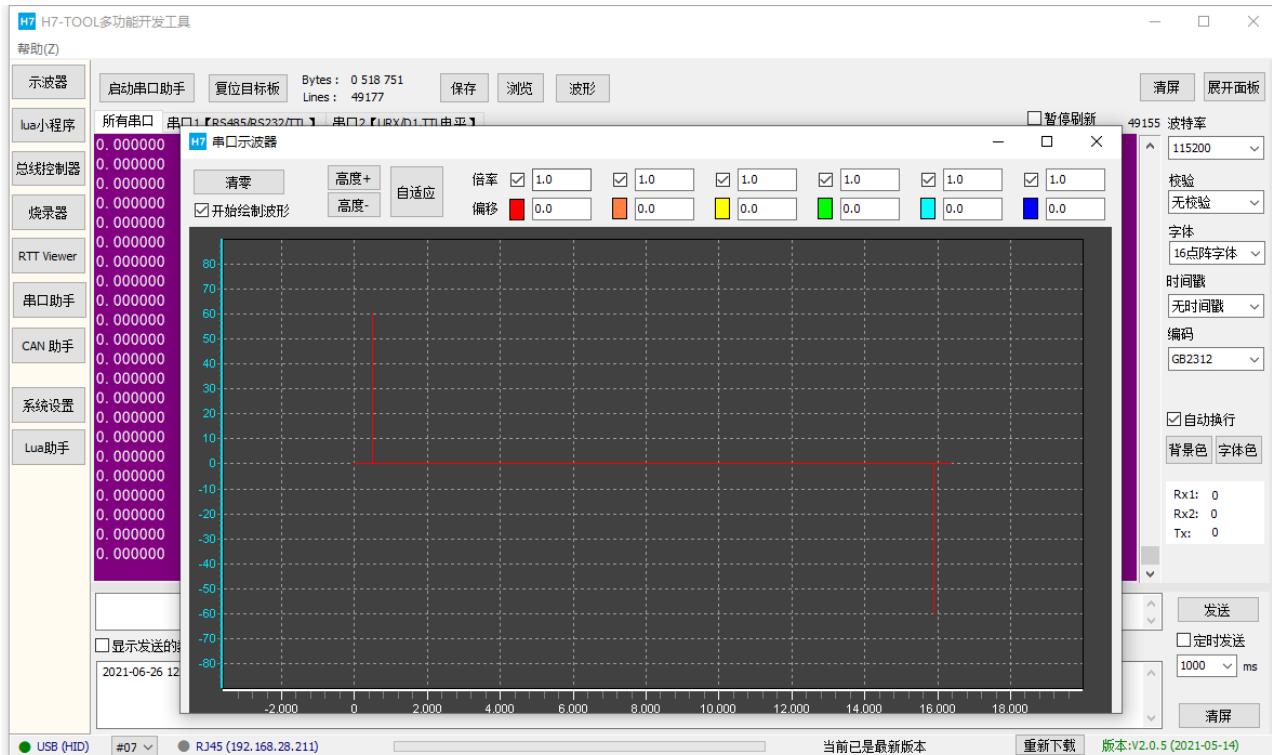
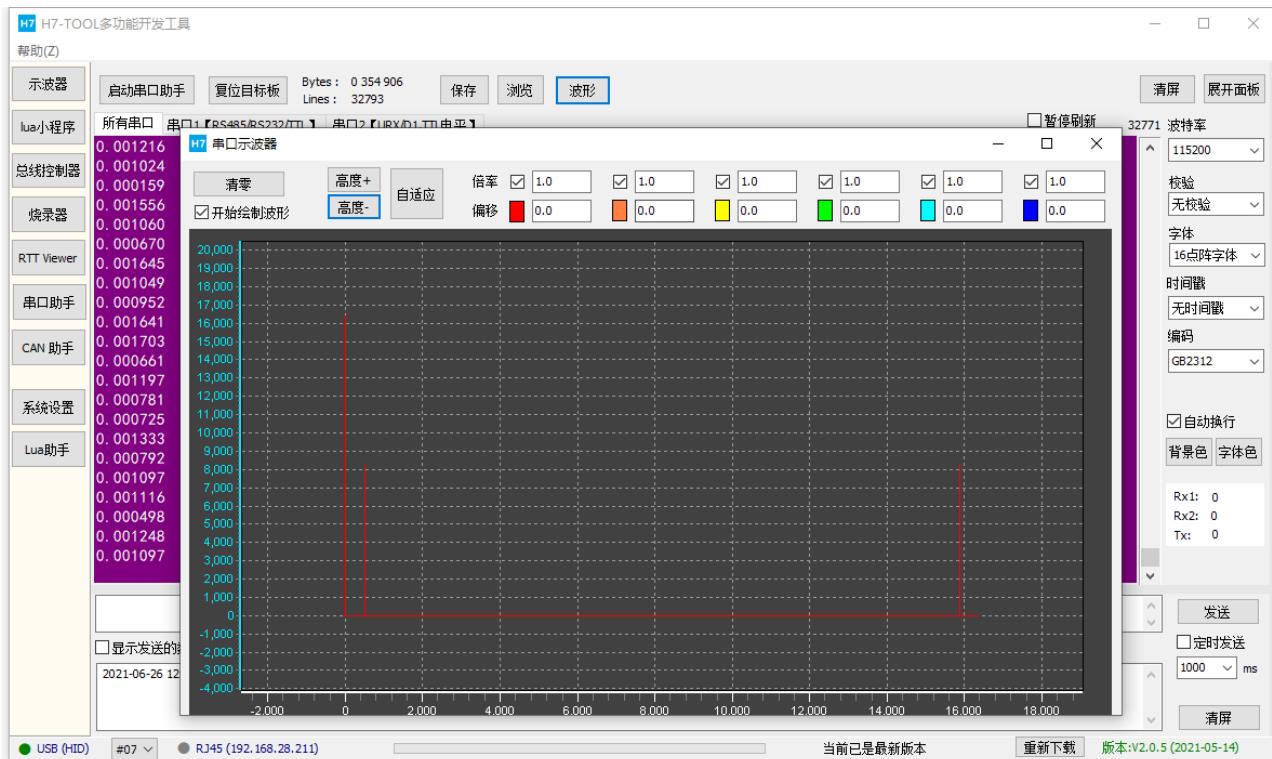
实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，串口打印 16384 点 FFT 的幅频响应。
3. 按下按键 K2，串口打印 16384 点 FFT 的相频响应。

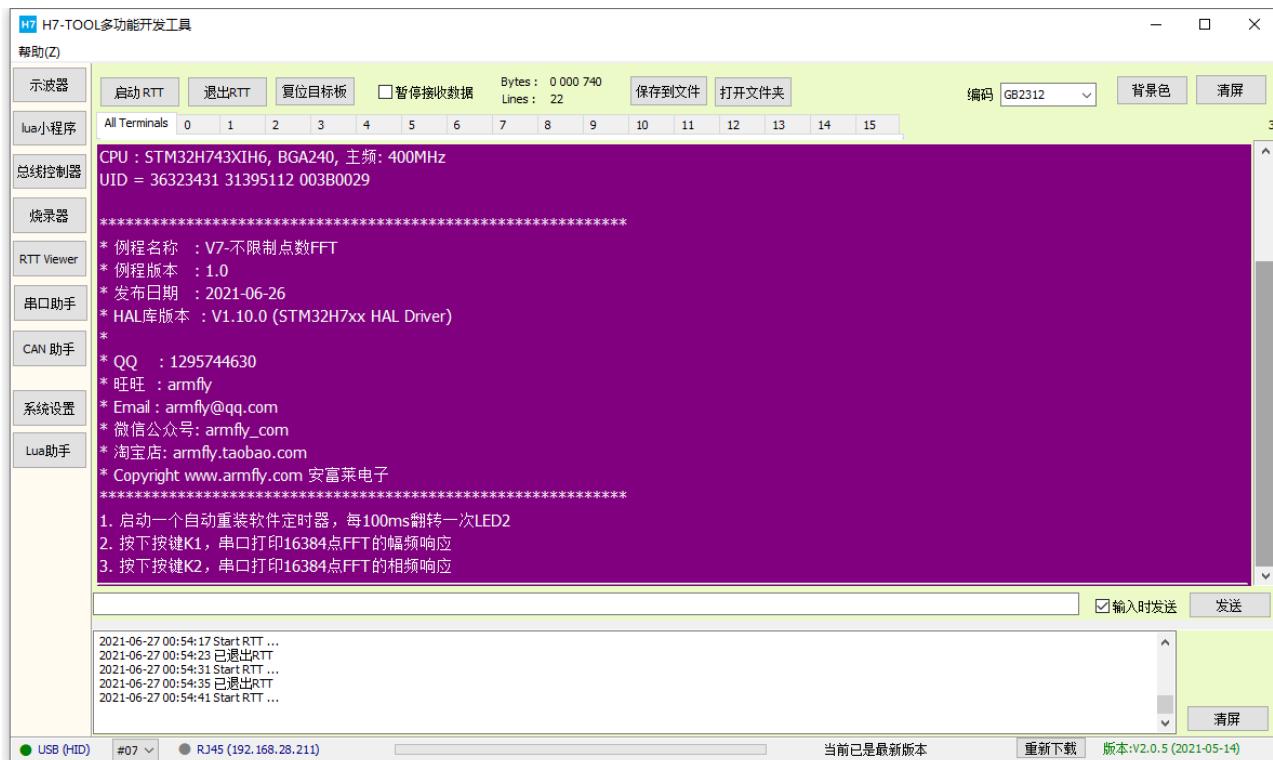
上电后串口打印的信息：



波特率 115200，数据位 8，奇偶校验位无，停止位 1。

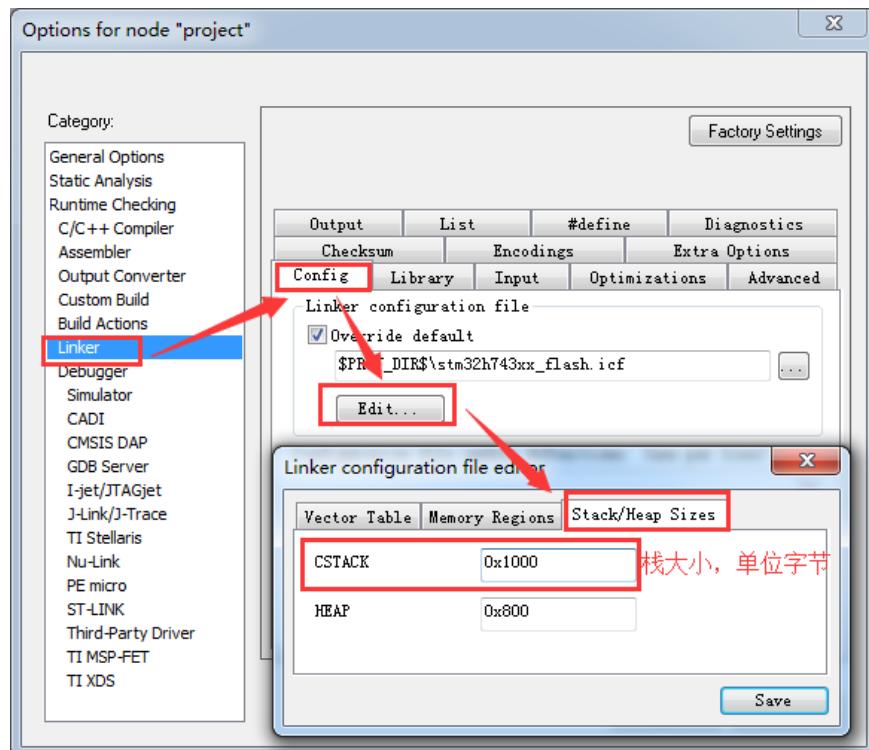


RTT 方式打印信息：

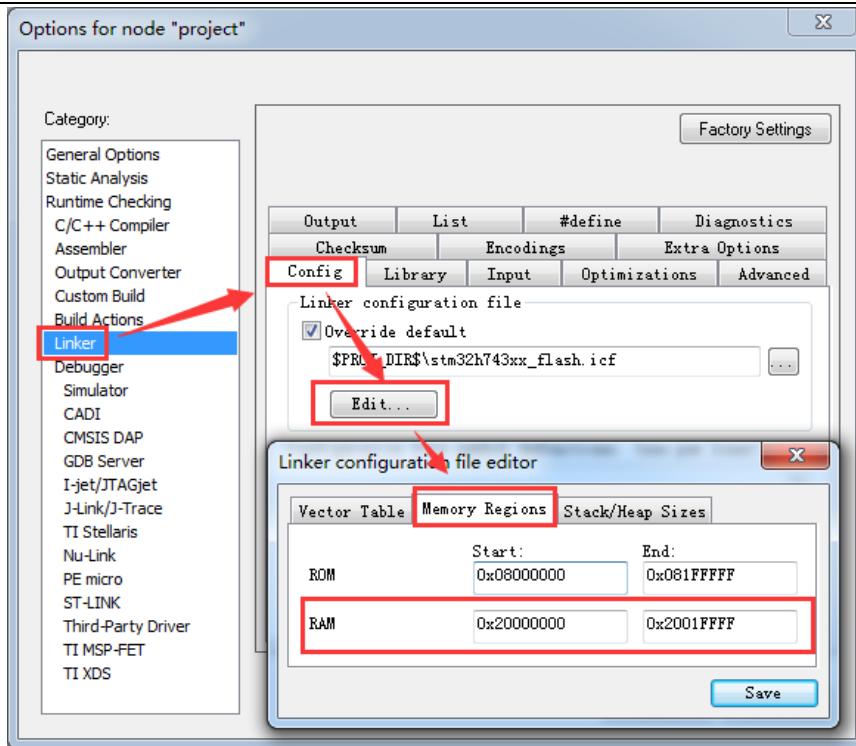


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，串口打印 16384 点 FFT 的幅频响应。
- 按下按键 K2，串口打印 16384 点 FFT 的相频响应。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */
}
```



```
/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(4); /* 翻转 LED2 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:           /* K1 键按下 */
                cfft_f32_mag();
                break;

            case KEY_DOWN_K2:           /* K2 键按下 */
                cfft_f32_phase();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
```

33.6 总结

本章节主要设计一个不限制点数的 FFT 功能，实际项目用到的地方也比较多，望初学者掌握。



第34章 DSP 滤波器基础知识

在数字信号处理中，滤波器占有及其重要的地位。数字滤波器是语音处理，图像处理，模式识别，频谱分析等应用的基本处理算法。从本章起，我们将开始讲解滤波器设计。

34.1 滤波器介绍

34.2 数字滤波器

34.2 总结

34.1 滤波器介绍

1917 年美国和德国科学家分别发明了 LC 滤波器，次年导致了美国第一个多路复用系统的出现。20 世纪 50 年代无源滤波器日趋成熟。自 60 年代起由于计算机技术、集成工艺和材料工业的发展，滤波器发展上了一个新台阶，并且朝着低功耗、高精度、小体积、多功能、稳定可靠和价廉方向努力，其中小体积、多功能、高精度、稳定可靠成为 70 年代以后的主攻方向。

34.1.1 滤波器的发展引言

凡是有能力进行信号处理的装置都可以称为滤波器。在近代电信设备和各类控制系统中，滤波器应用极为广泛；在所有的电子部件中，使用最多，技术最为复杂的要算滤波器了。滤波器的优劣直接决定产品的优劣，所以，对滤波器的研究和生产历来为各国所重视。

导致 RC 有源滤波器、数字滤波器、开关电容滤波器和电荷转移器等各种滤波器的飞速发展，到 70 年代后期，上述几种滤波器的单片集成已被研制出来并得到应用。80 年代，致力于各类新型滤波器的研究，努力提高性能并逐渐扩大应用范围。90 年代至现在主要致力于把各类滤波器应用于各类产品的开发和研制。当然，对滤波器本身的研究仍在不断进行。

我国广泛使用滤波器是 50 年代后期的事，当时主要用于话路滤波和报路滤波。经过半个世纪的发展，我国滤波器在研制、生产和应用等方面已纳入国际发展步伐，但由于缺少专门研制机构，集成工艺和材料工业跟不上来，使得我国许多新型滤波器的研制应用与国际发展有一段距离。

34.1.2 滤波器的分类

滤波器有各种不同的分类，一般有如下几种。



(1) 按处理信号类型分类

按处理信号类型分类，可分为模拟滤波器和离散滤波器两大类。其中模拟滤波器又可分为有源、无源、异类三个分类；离散滤波器又可分为数字、取样模拟、混合三个分类。当然，每个分类又可继续分下去，总之，它们的分类可以形成一个树形结构，实际上有些滤波器很难归于哪一类，例如开关电容滤波器既可属于取样模拟滤波器，又可属于混合滤波器，还可属于有源滤波器。因此，我们不必苛求这种“精确”分类，只是让人们了解滤波器的大体类型，有个总体概念就行了。

(2) 按选择物理量分类

按选择物理量分类，滤波器可分为频率选择、幅度选择、时间选择（例如 PCM 制中的话路信号）和信息选择（例如匹配滤波器）等四类滤波器。

(3) 按频率通带范围分类

按频率通带范围分类，滤波器可分为低通、高通、带通、带阻、全通五个类别，而梳形滤波器属于带通和带阻滤波器，因为它有周期性的通带和阻带。

滤波器种类繁多，有些是众所周知的，有些可能不为大家所熟悉，下面着重介绍近年来发展很快的几种滤波器。

34.1.3 有源滤波器

有源滤波器由下列一些有源元件组成：运算放大器、负电阻、负电容、负电感、频率变阻器（FDNR）、广义阻抗变换器（GIC）、负阻抗变换器（NIC）、正阻抗变换器（PIC）、负阻抗倒置器（NII）、正阻抗倒置器（PII）、四种受控源，另外，还有病态元件极子和零子。

1965 年单片集成运算放大器问世后，为有源滤波器开辟了广阔的前景。70 年代初期，有源滤波器发展引人注目，1978 年单片 RC 有源滤波器问世，为滤波器集成迈进了可喜的一步。由于运放的增益和相移均为频率的函数，这就限制了 RC 有源滤波器的频率范围，一般工作频率为 20kHz 左右，经过补偿后，工作频率也限制在 100kHz 以内。1974 年产生了更高频的 RC 有源滤波器，使工作频率可达 GB/4（GB 为运放增益与带宽之积）。由于 R 的存在，给集成工艺造成困难，于是又出现了有源 C 滤波器：就是滤波器由 C 和运放组成。这样容易集成，更重要的是提高了滤波器的精度，因为有源 C 滤波器的性能只取决于电容之比，与电容绝对值无关。但它有一个主要问题：由于各支路元件均为电容，所以运放没有直流反馈通道，使稳定性成为难题。1982 年由 Geiger、Allen 和 Ngo 提出用连续的开关电阻（SR）去替代有源 RC 滤波器中的电阻 R，就构成了 SRC 滤波器，它仍属于模拟滤波器。但由于采用前置电路和复杂的相位时钟，使这种滤波器发展前途不大。

总之，由 RC 有源滤波器为原型的各类变种有源滤波器去掉了电感器，体积小，Q 值可达 1000，克服了 RLC 无源滤波器体积大，Q 值小的缺点。但它仍有许多课题有待进一步研究：理想运放与实际特性的偏差的研究；由于有源滤波器混合集成工艺的不断改进，单片集成有待进一步研究；应用线性变换方法探索最少有源元件的滤波器需要继续探索；元件的绝对值容差的存在，影响滤波器精度和性能等问题。



仍未解决；由于 R 存在，集成占芯片面积大，电阻误差大（20%~30%），线性度差等缺点，使大规模集成仍然有困难。尽管有这么多问题，RC 有源滤波器的理论和应用仍在持续发展中。

34.1.4 开关电容滤波器 (SCF)

20 世纪 80 年代技术改造一个重大课题是实现各种电子系统全面大规模集成 (LSI)。使用最多的滤波器成为“拦路虎”，RC 有源滤波器不能实现 LSI，无源滤波器和机械滤波器更不用说了，于是，人们只能另辟新径。50 年代曾有人提出 SCF 的概念，由于当时集成工艺不过关，并没有引起人们的重视。1972 年，美国一个叫 Fried 的科学家发表了用开关和电容模拟电阻 R，说 SCF 的性能只取决于电容之比，与电容绝对值无关，这样才引起人们的重视。1979 年一些发达国家单片 SCF 已成为商品（属于高度保密技术）。现在 SC 技术已趋成熟。SCF 采用 MOS 工艺加以实现，被公认为 80 年代网络理论与集成工艺的一个重大突破。当前 MOS 电容值一般为几皮法至 100pF 之内，它具有 $(10 \sim 100) \times 10^{-6}/V$ 的电压系数与 $(10 \sim 100) \times 10^{-6}/^{\circ}C$ 的温度系数，这两个系数几乎接近理想的境界。SCF 具有下列一些优点：SCF 可以大规模集成；SCF 精度高，因为其性能取决于电容之比，而 MOS 电容之比的误差小于千分之一；功能多，几乎所有电子部件和功能均可以由 SC 技术来实现；比数字滤波器简单，因为不需要 A/D、D/A 转换；功能小，可以做到小于 10mW。

SCF 的应用以声频范围应用为主体，工作频率在 100kHz 之内。在信号处理方面的应用有：程控 SCF、模拟信号处理、振动分析、自适应性滤波器、音乐综合、共振谱、语言综合器、音调选择、语声编码、声频分析、均衡器、解调器、锁相电路、离散傅氏变换…… 总之，SCF 在仪表测量、医疗仪器、数据或信息处理等许多领域都有广泛的应用前景。

34.1.5 几种新型数字滤波器 (DF)

(1) 自适应 DF

最优控制、自适应控制和自学习控制都涉及到多参数、多变量的复杂控制系统，都属于现代控制理论研究的课题。自适应 DF 具有很强的自学习、自跟踪功能。它在雷达和声纳的波束形成、缓变噪声干扰的抑制、噪声信号的处理、通信信道的自适应均衡、远距离电话的回声抵消等领域获得了广泛的应用，促进了现代控制理论的发展。

自适应 DF 有如下一些简单算法：W—LMS 算法、M—LMS 算法、TDO 算法、差值 LMS 算法和 C—LMS 算法。

(2) 复数 DF

在输入信号为窄带信号处理系统中，常采用复数 DF 技术。为了降低采样率而又保存信号所包含的全部信息，可利用正交双路检波法，取出窄带信号的复包络，然后通过 A/D 变换，将复包络转化为复数序列进行处理，这个信号处理系统即为复数 DF。它具有许多功能：MTI 雷达中抑制具有卜勒频移的杂波干扰；数字通信网与模拟通信网之间多路 TDM/FDM 信号变换复接……



(3)多维 DF

在图像处理、地震、石油勘探的数据处理中都用到多维 DF (常用是二维 DF)，多维 DF 的设计，往往将一维 DF 优化设计直接推广到多维 DF 中去。对于模糊和随机噪声干扰的二维图像的处理，多维 DF 也能发挥很好的作用。

此外，还有波 DF，它便于实现大规模集成，便于无源和有源滤波网络的数字模拟。因此，正受到人们的重视和加以研究。

对于 DF 有待研究的课题有：系数灵敏度、舍入噪声和极限环、多维逆归滤波器的稳定性、各种硬件和软件实现 DF 的研究等等。总之，DF 在数字信号处理技术中占有极为重要的地位，对于它的研究、生产和应用等工作均是很有意义的。

34.1.6 其它新型滤波器

为适应各种需要，出现了一批新型滤波器，这里介绍几种已得到广泛应用的新型滤波器。

(1)电控编程 CCD 横向滤波器 (FPCCDTF)

电荷耦合器 (CCD) 固定加权的横向滤波器 (TF) 在信号处理中，其性能和造价均可与数字滤波器和各种信号处理部件媲美。这种滤波器主要用于自适应滤波；P-N 序列和 Chirp 波形的匹配滤波；通用化的频域滤波器及相关积运算；语音信号和相位均衡；相阵系统的波束合成和电视信号的重影消除等均有应用。当然，更多的应用有待进一步开拓。总之，FPCCDTF 是最有希望的发展方向。

(2)晶体滤波器

它是适应单边带技术而发展起来的。在 20 世纪 70 年代，集成晶体滤波器的产生，使它的发展产生一个飞跃。近十年来，晶体滤波器致力于下面一些研究：实现最佳设计，除具有优良的选择外，还具有良好的时域响应；寻求新型材料；扩展工作频率；改造工艺，使其向集成化发展。它广泛应用于多路复用系统中作为载波滤波器，在收发信中，单边带通信机中作为选频滤波器，在频谱分析仪和声纳装置中作为中频滤波器。

(3)声表面滤波器

它是理想的超高频器件。它的幅频特性和相位特性可以分别控制，以达到要求，而且它还有体积小，长时间稳定性好和工艺简单等特点。通常应用于：电视广播发射机中作为残留边带滤波器；在彩色电视接收机中调谐系统的表面梳形滤波器。此外，在国防卫星通信系统中已广泛采用。声表面滤波器是电子学和声学相结合的产物，而且可以集成，所以，它在所有无源滤波器中最有发展前途的。

各种新型滤波器太繁多，限于篇幅，不再一一叙述。

我国目前各种滤波器的应用比例

我国现有滤波器的种类和所覆盖的频率已基本上满足现有各种电信设备。从整体而言，我国有源滤波器发展比无源滤波器缓慢，尚未大量生产和应用。从下面的生产应用比例可以看出我国各类滤波器的应用情况：LC 滤波器占 50%；晶体滤波器占 20%；机械滤波器占 15%；陶瓷和声表面滤波器各占 1%；



其余各类滤波器共占 13%。从这些应用比例来看，我国电子产品要想实现大规模集成，滤波器集成化仍然是个重要课题。

随着电子工业的发展，对滤波器的性能要求越来越高，功能也越来越多，并且要求它们向集成方向发展。我国滤波器研制和生产与上述要求相差甚远，为缩短这个差距，电子工程和科技人员负有重大的历史责任。

34.2 数字滤波器

34.2.1 数字滤波器和模拟滤波器区别

数字滤波器是对数字信号进行滤波处理以得到期望的响应特性的离散时间系统。作为一种电子滤波器，数字滤波器与完全工作在模拟信号域的模拟滤波器不同。数字滤波器工作在数字信号域，它处理的对象是经由采样器件将模拟信号转换而得到的数位信号。

数字滤波器的工作方式与模拟滤波器也完全不同：后者完全依靠电阻器、电容器、晶体管等电子元件组成的物理网络实现滤波功能；而前者是通过数字运算器件对输入的数字信号进行运算和处理，从而实现设计要求的特性。

数字滤波器理论上可以实现任何可以用数学算法表示的滤波效果。数字滤波器的两个主要限制条件是它们的速度和成本。数字滤波器不可能比滤波器内部的数字电路的运算速度更快。但是随着集成电路成本的不断降低，数字滤波器变得越来越常见并且已经成为了如收音机、蜂窝电话、立体声接收机这样的日常用品的重要组成部分。

数字滤波器一般由寄存器、延时器、加法器和乘法器等基本数字电路实现。随着集成电路技术的发展，其性能不断提高而成本却不断降低，数字滤波器的应用领域也因此越来越广。按照数字滤波器的特性，它可以被分为线性与非线性、因果与非因果、无限脉冲响应 (IIR) 与有限脉冲响应 (FIR) 等等。其中，线性时不变的数字滤波器是最基本的类型；而由于数字系统可以对延时器加以利用，因此可以引入一定程度的非因果性，获得比传统的因果滤波器更灵活强大的特性；相对于 IIR 滤波器，FIR 滤波器有着易于实现和系统绝对稳定的优势，因此得到广泛的应用；对于时变系统滤波器的研究则导致了以卡尔曼滤波为代表的自适应滤波理论。

34.2.2 数字滤波器特性

数字滤波器具有比模拟滤波器更高的精度，甚至能够实现后者在理论上也无法达到的性能。例如，对于数字滤波器来说很容易就能够做到一个 1000Hz 的低通滤波器允许 999Hz 信号通过并且完全阻止 1001Hz 的信号，模拟滤波器无法区分如此接近的信号。

数字滤波器相比模拟滤波器有更高的信噪比。这主要是因为数字滤波器是以数字器件执行运算，从



而避免了模拟电路中噪声（如电阻热噪声）的影响。数字滤波器中主要的噪声源是在数字系统之前的模拟电路引入的电路噪声以及在数字系统输入端的模数转换过程中产生的量化噪声。这些噪声在数字系统的运算中可能会被放大，因此在设计数字滤波器时需要采用合适的结构，以降低输入噪声对系统性能的影响。

数字滤波器还具有模拟滤波器不能比拟的可靠性。组成模拟滤波器的电子元件的电路特性会随着时间、温度、电压的变化而漂移，而数字电路就没有这种问题。只要在数字电路的工作环境下，数字滤波器就能够稳定可靠的工作。

由于奈奎斯特采样定理 (Nyquist sampling theorem)，数字滤波器的处理能力受到系统采样频率的限制。如果输入信号的频率分量包含超过滤波器 $1/2$ 采样频率的分量时，数字滤波器因为数字系统的“混叠”而不能正常工作。如果超出 $1/2$ 采样频率的频率分量不占主要地位，通常的解决办法是在模数转换电路之前放置一个低通滤波器（即抗混叠滤波器）将超过的高频成分滤除。否则就必须用模拟滤波器实现要求的功能。

34.2.3 经典滤波器和数字滤波器

一般滤波器可以分为经典滤波器和数字滤波器。

经典滤波器：假定输入信号中的有用成分和希望去除的成分各自占有不同的频带。如果信号和噪声的频谱相互重迭，经典滤波器无能为力。比如 FIR 和 IIR 滤波器等。

现代滤波器：从含有噪声的时间序列中估计出信号的某些特征或信号本身。现代滤波器将信号和噪声都视为随机信号。包括 Wiener Filter、Kalman Filter、线性预测器、自适应滤波器等。

34.2.4 IIR 滤波器和 FIR 滤波器

线性移不变的数字滤波器包括无限长脉冲响应滤波器 (IIR 滤波器) 和有限长脉冲响应滤波器 (FIR 滤波器) 两种。这两种滤波器的系统函数可以统一以 Z 变换表示为：

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_N z^{-N}}{1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_M z^{-M}}$$

当 $M \geq 1$ 时， M 就是 IIR 滤波器的阶数，表示系统中反馈环的个数。由于反馈的存在，IIR 滤波器的脉冲响应为无限长，因此得名。若 $A(z) = 1$ ，则系统的脉冲响应的长度为 $N+1$ ，故而被称作 FIR 滤波器。

IIR 滤波器的优点在于，其设计可以直接利用模拟滤波器设计的成果，因为模拟滤波器本身就是无限长冲激响应的。通常 IIR 滤波器设计的过程如下：首先根据滤波器参数要求设计对应的模拟滤波器（如巴特沃斯滤波器、切比雪夫滤波器等等），然后通过映射（如脉冲响应不变法、双线性映射等等）将模拟滤波器变换为数字滤波器，从而决定 IIR 滤波器的参数。IIR 滤波器的重大缺点在于，由于存在反馈其稳定性不能得到保证。另外，反馈还使 IIR 滤波器的数字运算可能溢出。



FIR 滤波器最重要的优点就是由于不存在系统极点，FIR 滤波器是绝对稳定的系统。FIR 滤波器还确保了线性相位，这在信号处理中也非常重要。此外，由于不需要反馈，FIR 滤波器的实现也比 IIR 滤波器简单。FIR 滤波器的缺点在于它的性能不如同样阶数的 IIR 滤波器，不过由于数字计算硬件的飞速发展，这一点已经不成为问题。再加上引入计算机辅助设计，FIR 滤波器的设计也得到极大的简化。基于上述原因，FIR 滤波器比 IIR 滤波器的应用更广。

34.3 总结

本章节主要介绍了滤波器方面的基础知识，帮助初学者对滤波器有一个全面的认识。



第35章 DSP FIR 有限冲击响应滤波器设计

FIR 滤波器设计到的内容比较多，本章节主要进行了总结性的介绍，以帮助没有数字信号处理基础的读者能够有个整体的认识，有了这个整体的认识之后再去查阅相关资料可以到达事半功倍的效果。

35.1 基本概念

35.2 FIR 数字滤波器的基本网络结构

35.2 FIR 数字滤波器的设计方法

35.4 总结

35.1 基本概念

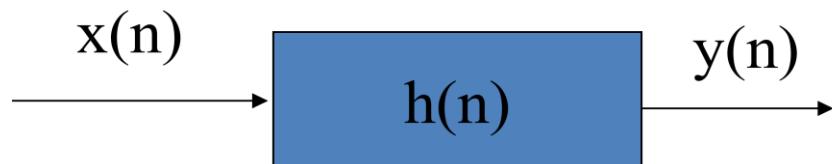
35.1.1 什么是数字滤波器

数字滤波器的作用是对输入信号起到滤波的作用；即 DF (Digital Filter) 是由差分方程描述的一类特殊的离散时间系统。它的功能是把输入序列通过一定的运算转换成输出序列。不同的运算处理方法决定了滤波器的实现结构的不同。

35.1.2 数字滤波器的工作原理

设 $x(n)$ 是系统的输入， $X(e^{j\omega})$ 是其傅氏变换。

设 $y(n)$ 是系统的输出， $Y(e^{j\omega})$ 是其傅氏变换。则 LSI (线性时不变系统) 的输出为：

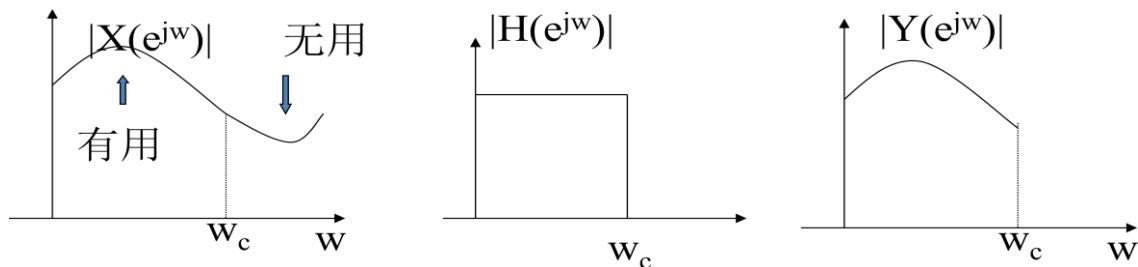


$$\text{频域: } Y(e^{j\omega}) = X(e^{j\omega})H(e^{j\omega})$$

$$\text{时域: } y(n) = h(n) * x(n)$$

$$y(n) = \sum_{m=-\infty}^{\infty} h(n-m)x(m) = F^{-1}[H(e^{j\omega})H(e^{j\omega})]$$

由上面的公式可以看出输入序列的频谱 $X(e^{j\omega})$ 经过滤波器后 (其系统性能由 $H(e^{j\omega})$ 表示) 变成 $H(e^{j\omega})X(e^{j\omega})$ 。简单的说，选取合适的 $H(e^{j\omega})$ ，使滤波器的输出 $H(e^{j\omega})X(e^{j\omega})$ 符合我们的要求，这就是数字滤波器的工作原理。



上面的截图可以形象的解释 $Y(e^{j\omega}) = X(e^{j\omega})H(e^{j\omega})$ 。

35.1.3 数字滤波器的分类

滤波器的种类很多，分类方法也不同。

- 1.从功能上分；低通、带通、高通、带阻。
- 2.从实现方法上分:FIR、IIR
- 3.从设计方法上来分：Chebyshev(切比雪夫) ,Butterworth (巴特沃斯)
- 4.从处理信号分：经典滤波器、现代滤波器

经典滤波器从功能上分又可分为：

低通滤波器 (LPAF/LPDF):Low pass analog filter

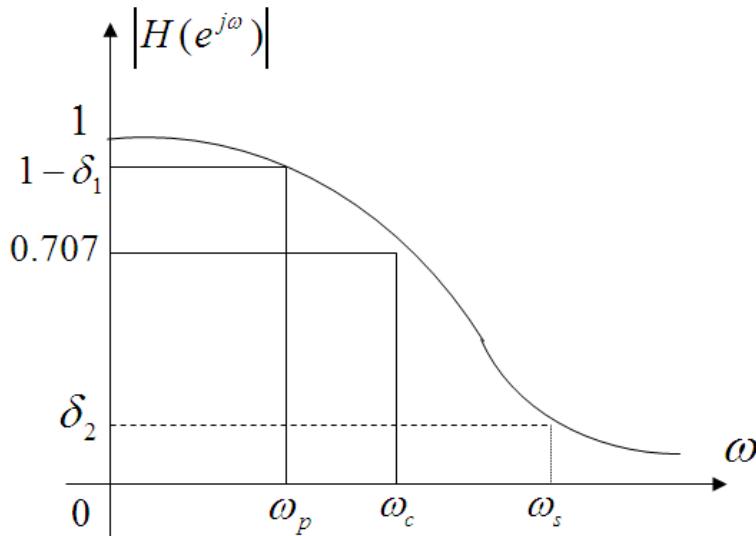
带通滤波器(BPAF/BPDF):Bandpass analog filter

高通滤波器(HPAF/HPDF):High pass analog filter

带阻滤波器(BSAF/BSDF):Bandstop analog filter

35.1.4 滤波器的技术要求

数字滤波器的传输函数： $H(e^{j\omega}) = |H(e^{j\omega})|e^{j\varphi(\omega)}$



ω_p : 通带截止频率

α_p : 通带允许的最大衰减

ω_s : 阻带截止频率

α_s : 阻带允许的最大衰减

ω_c : 3dB 通带截止频率

δ_1 δ_2 : 通带、阻带的容限 (允许误差)

α_p α_s : 分别定义为: (P-Pass, S-Stop)

$$\alpha_p = 20 \log \frac{|H(e^{j\omega_p})|}{|H(e^{j0})|} = -20 \log |H(e^{j\omega_p})| \text{ (dB)}$$

$$\alpha_s = 20 \log \frac{|H(e^{j\omega_s})|}{|H(e^{j0})|} = -20 \log |H(e^{j\omega_s})| \text{ (dB)}$$

式中均假定 $H(e^{j0}) = 1$ (归一化)

当 $\omega_p = \omega_c$ 时, $\alpha_p = 3\text{dB}$

35.1.5 滤波器的基本运算

基本运算: 相乘, 延迟, 相加;

表示方法: 线性差分方程、系统函数、框图或流图。

差分方程: $y(n) = \sum_{k=1}^N a_k \cdot y(n-k) + \sum_{k=0}^M b_k \cdot x(n-k)$

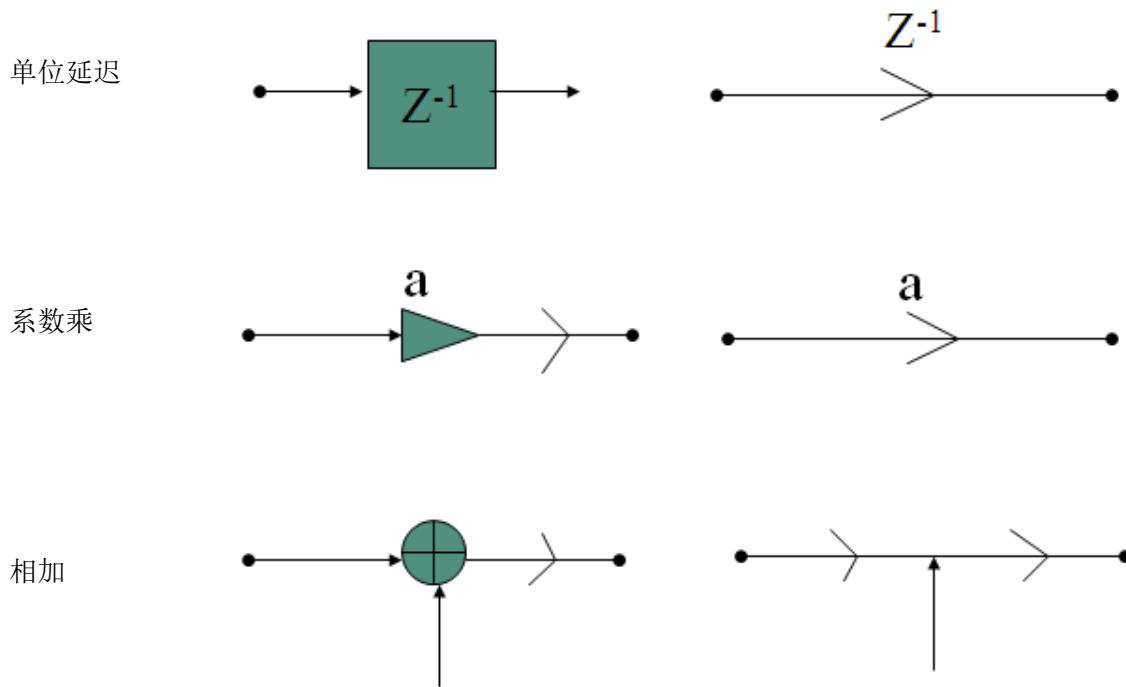
系统函数: $H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^M b_k \cdot z^{-k}}{1 - \sum_{k=1}^N a_k \cdot z^{-k}} = \frac{B(z)}{A(z)}$

由上式得: $Y(z) = \frac{B(z)}{A(z)} X(z)$

35.1.6 数字滤波器的表述方法

方框图表示法:

信号流图表示法:



把上述三个基本单元互联，可构成不同数字网络或运算结构，也有方框图表示法和流图表示法。

35.2 FIR 数字滤波器的基本网络结构

一个线性位移不变系统的输出序列 $y(n)$ 和输入序列 $x(n)$ 之间的关系，应满足常系数线性差分方程：

$$y(n) = \sum_{i=0}^{N-1} b_i x(n-i) - \sum_{i=1}^M a_i y(n-i) \quad n \geq 0$$

$x(n)$: 输入序列, $y(n)$: 输出序列, a_i 、 b_i : 滤波器系数, N : 滤波器的阶数。

若所有的 a_i 均为 0，则得 FIR 滤波器的差分方程：

$$y(n) = \sum_{i=0}^{N-1} b_i x(n-i)$$

对式其进行 z 变换，可得 FIR 滤波器的传递函数：

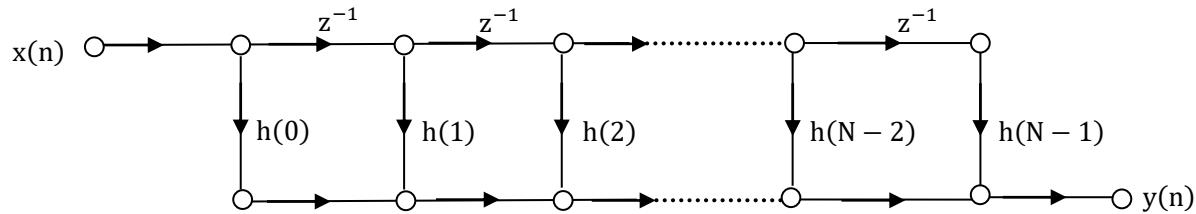
$$H(z) = \frac{Y(z)}{X(z)} = \sum_{i=0}^{N-1} b_i z^{-i}$$

35.2.1 直接型结构

差分方程： $y(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$

系统函数: $H(z) = \sum_{n=0}^{N-1} h(n)z^{-n}$

$$= h(0) + h(1)z^{-1} + h(2)z^{-2} + \dots + h(N-1)z^{-(N-1)}$$

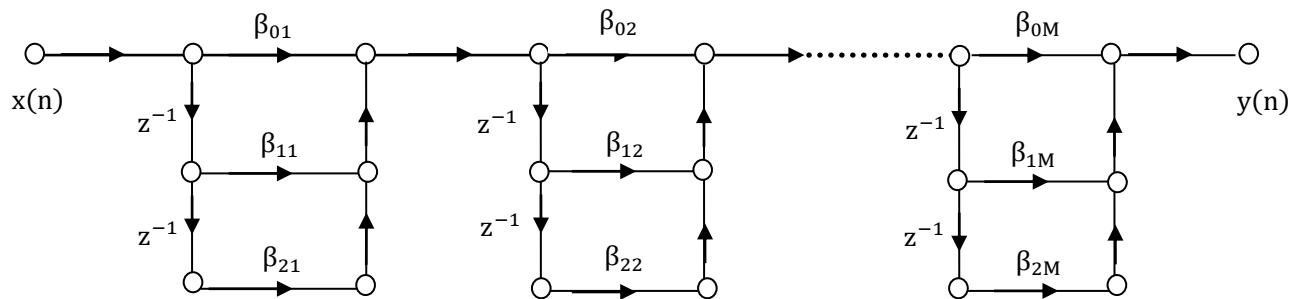


这种结构的特点是只含前向回路。

35.2.2 级联型结构

系统函数: $H(z) = \sum_{k=0}^{N-1} h(k)z^{-k}$

$$= \prod_{k=1}^M (\beta_{0k} + \beta_{1k} \cdot z^{-1} + \beta_{2k} \cdot z^{-2}), \quad h(n) \text{ 为实系数;}$$



这种结构的特点是每一种基本节控制一对零点，乘法器较多，并且遇到高阶 $H(z)$ 时难分解

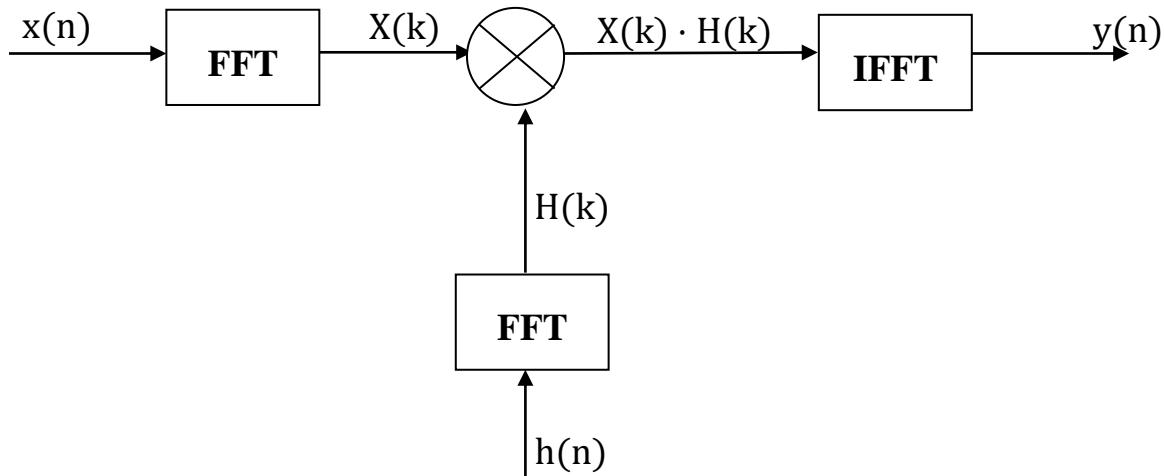
35.2.3 快速卷积型结构

已知两个长度为 N 的序列的线性卷积，可用 $2N-1$ 点的循环卷积来代替。

FIR 滤波器输出: $y(n) = x(n) * h(n)$

1. 延长 $x(n)$ 、 $h(n)$ 使: $x(n) * h(n) = x(n) \otimes h(n)$
2. 计算: $X(k) = \text{FFT}[x(n)]$, $H(k) = \text{FFT}[h(n)]$
3. 计算: $Y(k) = X(k) * H(k);$

4. 计算: $y(n) = \text{IFFT}\{X(k) * H(k)\}$



这种结构的特点是能对信号进行高速处理。需要实时处理时采用此结构。

35.2.4 线性相位型结构

这种结构涉及到的内容较多，我们这里不做讨论，大家查询相关书籍进行了解即可。

35.2.5 频率抽样型结构

这种结构涉及到的内容较多，我们这里不做讨论，大家查询相关书籍进行了解即可。

35.3 FIR 数字滤波器的设计方法

基本特性：

1. FIR 滤波器永远是稳定的（系统只有零点）；
2. FIR 滤波器的冲激响应是有限长序列；
3. FIR 滤波器的系统函数为多项式；
4. FIR 滤波器具有线性相位。

目前，FIR 数字滤波器的设计方法主要是建立在对理想滤波器频率特性做某种近似的基础上。这些近似方法有窗函数法，频率抽样法及最佳一直逼近法。

关于窗函数法，频率抽样法及最佳一直逼近法我们这里不做讨论了，这三种方法涉及的内容都比较多，大家有兴趣的可以查阅相关书籍资料进行了解。



35.4 总结

本期教程主要对 FIR 滤波进行了总结性的介绍，每个知识点并没有进行详细的介绍，如果将这些知识点也进行展开的话将占用大量的篇幅，而且大家不容易看懂。尽管这样，还是希望有兴趣的读者去查阅相关的书籍进行深入的了解，只有你对这些理论有了深入的理解，你的实际应用才能事半功倍。还是那句经常说的话：理论高度决定实践高度。



第36章 DSP FIR 滤波器的 Matlab 设计(含低通, 高通, 带通和带阻)

本章节讲解 FIR 滤波器的 Matlab 设计。主要是函数 fir1 和 fir2 的使用。

36.1 窗函数

36.2 fir1 函数

36.2 fir2 函数

36.4 总结

36.1 窗函数

在数字信号处理中不可避免地要用到数据截取的问题。例如，在应用 DFT 的时候，数据 $x(n)$ 总是有限长的，在滤波器设计中遇到了对理想滤波器抽样响应 $h(n)$ 的截取问题，在功率谱估计中也要遇到对自相关函数的截取问题。总之，我们在实际工作中所能处理的离散序列总是有限长，把一个长序列转换成有限长的序列不可避免的要用到窗函数。因此，窗函数本身的研究及其应用是信号处理中的一个基本问题。

不同的窗函数对信号频谱的影响是不一样的，这主要是因为不同的窗函数，产生泄漏的大小不一样，频率分辨能力也不一样。信号的截断产生了能量泄漏，而用 FFT 算法计算频谱又产生了栅栏效应，从原理上讲这两种误差都是不能消除的，但是我们可以通过选择不同的窗函数对它们的影响进行抑制。(矩形窗主瓣窄，旁瓣大，频率识别精度最高，幅值识别精度最低；布莱克曼窗主瓣宽，旁瓣小，频率识别精度最低，但幅值识别精度最高)。

对于窗函数的选择，应考虑被分析信号的性质与处理要求。如果仅要求精确读出主瓣频率，而不考虑幅值精度，则可选用主瓣宽度比较窄而便于分辨的矩形窗，例如测量物体的自振频率等；如果分析窄带信号，且有较强的干扰噪声，则应选用旁瓣幅度小的窗函数，如汉宁窗、三角窗等；对于随时间按指数衰减的函数，可采用指数窗来提高信噪比。

● 矩形窗：

矩形窗属于时间变量的零次幕窗。矩形窗使用最多，习惯上不加窗就是使信号通过了矩形窗。这种窗的优点是主瓣比较集中，缺点是旁瓣较高，并有负旁瓣，导致变换中带进了高频干扰和泄漏，甚至出现负谱现象。

● 三角窗：



三角窗亦称费杰 (Fejer) 窗，是幕窗的一次方形式。与矩形窗比较，主瓣宽约等于矩形窗的两倍，但旁瓣小，而且无负旁瓣。

● 汉宁窗：

汉宁窗又称升余弦窗，汉宁窗可以看作是 3 个矩形时间窗的频谱之和，或者说是 3 个 $\text{sinc}(t)$ 型函数之和，而括号中的两项相对于第一个谱窗向左、右各移动了 π/T ，从而使旁瓣互相抵消，消去高频干扰和漏能。可以看出，汉宁窗主瓣加宽并降低，旁瓣则显著减小，从减小泄漏观点出发，汉宁窗优于矩形窗。但汉宁窗主瓣加宽，相当于分析带宽加宽，频率分辨率下降。

● 海明窗：

海明窗也是余弦窗的一种，又称改进的升余弦窗。海明窗与汉宁窗都是余弦窗，只是加权系数不同。海明窗加权的系数能使旁瓣达到更小。分析表明，海明窗的第一旁瓣衰减为 -42dB 。海明窗的频谱也是由 3 个矩形时窗的频谱合成，但其旁瓣衰减速度为 $20\text{dB}/(10\text{oct})$ ，这比汉宁窗衰减速度慢。海明窗与汉宁窗都是很有用的窗函数。

● 高斯窗：

三角窗亦称费杰 (Fejer) 窗，是幕窗的一次方形式。与矩形窗比较，主瓣宽约等于矩形窗的两倍，但旁瓣小，而且无负旁瓣。

还有很多其它的窗口这里就不做介绍了，需更详细的了解的话，可以看 matlab 中 help 文档中的如下部分：

The screenshot shows the MATLAB Documentation Center search results for the term "Window". The search bar at the top contains the word "Window". Below the search bar, there is a sidebar with categories like "筛选" (Filter), "关闭" (Close), and sections for "所有产品" (All products) and "按类型分类" (Classify by type). The main search results area displays several entries:

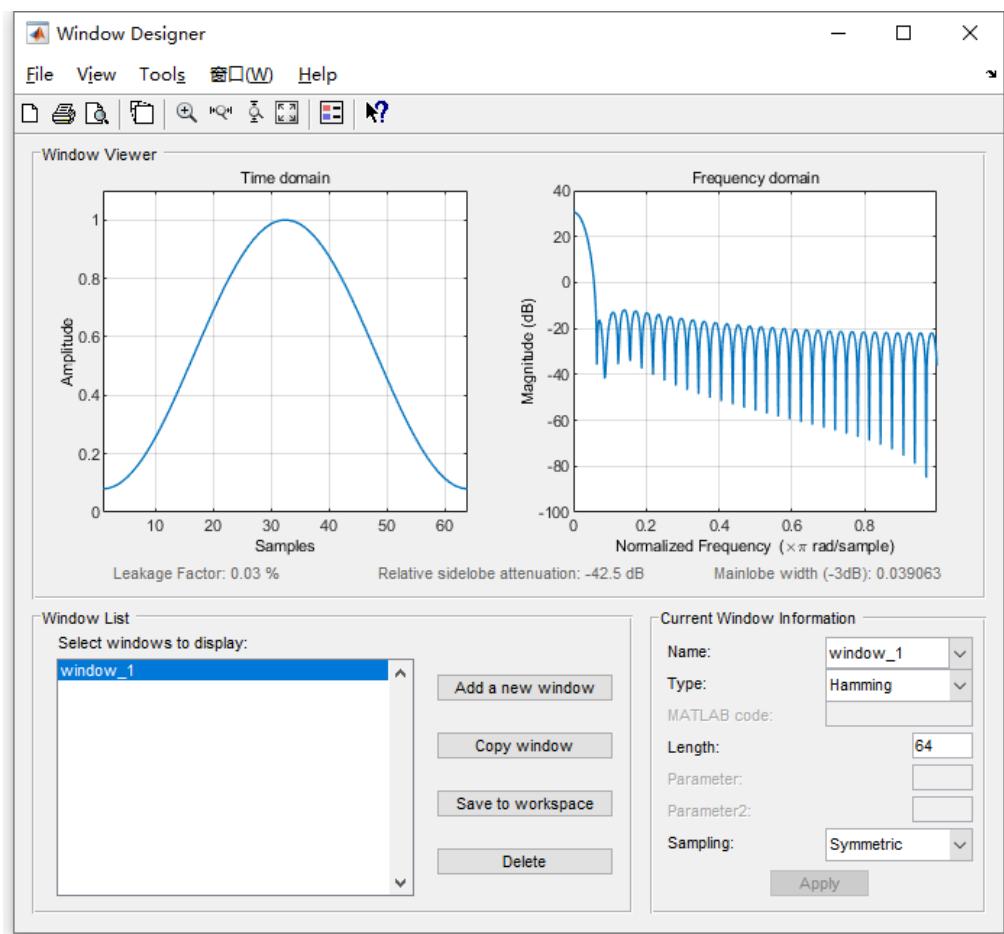
- window - Window function gateway**
This MATLAB function opens the Window Designer app.
文档 > Signal Processing Toolbox
- window - FIR filter using windowed impulse response**
This MATLAB function designs a single-rate digital filter System object using the specifications in filter specification object d.
文档 > DSP System Toolbox > Filter Design and Analysis > Filter Design
- dsp.Window System object - Window object**
The Window object applies a window to an input signal.
文档 > DSP System Toolbox > Signal Generation, Manipulation, and Analysis > Signal Operations
- Chebyshev Window**
The Chebyshev window minimizes the mainlobe width for a particular sidelobe level and exhibits equiripple sidelobe behavior.
文档 > Signal Processing Toolbox > Spectral Analysis > Windows
- Window Designer - Design and analyze spectral windows**
The Window Designer app enables you to design and analyze spectral windows.

The first result is highlighted with a red box.

或者直接在命令窗口输入 `windowDesigner` 可以打开窗口工具：



打开后界面如下：



36.2 fir1 函数

36.2.1 fir1 函数介绍

函数 fir1 用来设计标准频率响应的基于窗函数的 FIR 滤波器，可实现加窗线性相位 FIR 滤波器设计。

语法：

```
b = fir1(n,Wn)
b = fir1(n,Wn,'ftype')
b = fir1(n,Wn>window)
b = fir1(n,Wn,'ftype>window)
b = fir1(...,'normalization')
```



其中, n: 为了滤波器的阶数;

Wn: 为滤波器的截止频率;

ftype: 参数用来决定滤波器的类型, 当 ftype=high 时, 可设计高通滤波器, 当 ftype=stop 时, 可设计带阻滤波器。Window 参数用来指导滤波器采用的窗函数类型。其默认值为汉明 (Hamming) 窗。

使用 fir1 函数可设计标准的低通, 高通, 带通和带阻滤波器。滤波器的系数包含在返回值 b 中, 可表示为:

$$b(z) = b(1) + b(2)z^{-1} + \dots + b(n+1)z^{-n}$$

(1) 采用汉明窗设计低通 FIR 滤波器

使用 `b=fir1(n, Wn)` 可得到低通滤波器。其中, $0 \leq Wn \leq 1$, $Wn=1$ 相当于 $0.5f_s$ 。其语法格式为

$$b=fir1(n, Wn)$$

(2) 采用汉明窗设计高通 FIR 滤波器

在 `b=fir1(n, Wn, 'ftype')` 中, 当 `ftype=high` 时, 可设计高通滤波器。其语法格式为

$$b=fir1(n, Wn, 'high')$$

(3) 采用汉明窗设计带通 FIR 滤波器

在 `b=fir1(n, Wn)` 中, 当 $W_n=[W_1 \ W_2]$ 时, fir1 函数可得到带通滤波器, 其通带为 $W_1 < W < W_2$

W_1 和 W_2 分别为通带的下限频率和上限频率。其语法格式为

$$b=fir1(n, [W_1 \ W_2])$$

(4) 采用汉明窗设计带阻 FIR 滤波器

在 `b = fir1(n, Wn, 'ftype')` 中, 当 `ftype=stop`, $W_n=[W_1 \ W_2]$ 时, fir1 函数可得到带阻滤波器, 其语法格式为

$$b=fir1(n, [W_1 \ W_2], 'stop')$$

(5) 采用其他窗口函数设计 FIR 滤波器

使用 Window 参数, 可以用其他窗口函数设计出各种加窗滤波器, Window 参数可采用的窗口函数有 Boxcar, Hamming, Bartlett, Blackman, Kasier 和 Chebwin 等。其默认时为 Hamming 窗。例如, 采用 Bartlett 窗设计带阻滤波器, 其语法结构为

$$b=fir1(n, [W_1 \ W_2], 'stop', Bartlett[n+1])$$

注意: 用 fir1 函数设计高通和带阻滤波器时, 所使用的阶数 n 应为偶数, 当输入的阶数 n 为奇数时, fir1 函数会自动将阶数增加 1 形成偶数。

36.2.2 fir1 设计低通滤波器实例

下面我们通过一个实例来讲解 fir1 的用法。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成, 将 200Hz 的正弦波当做噪声滤掉, 下面通过函数 fir1 设计一组低通滤波器系数, 其阶数是 30, 截止频率为 0.25 (也就是 125Hz)。Matlab 运行代码如下:



```
%*****FIR 低通滤波器设计*****
```

```
%*****
```

```
fs=1000; %设置采样频率 1k
```

```
N=1024; %采样点数
```

```
n=0:N-1;
```

```
t=0:1/fs:1-1/fs; %时间序列
```

```
f=n*fs/N; %频率序列
```

```
Signal_Original=sin(2*pi*50*t); %信号 50Hz 正弦波
```

```
Signal_Noise=sin(2*pi*200*t); %噪声 200Hz 正弦波
```

```
Mix_Signal=Signal_Original+Signal_Noise; %将信号 Signal_Original 和 Signal_Original 合成一个信号进行采样
```

```
subplot(221);
```

```
plot(t, Mix_Signal); %绘制信号 Mix_Signal 的波形
```

```
xlabel('时间');
```

```
ylabel('幅值');
```

```
title('原始信号');
```

```
grid on;
```

```
subplot(222);
```

```
y=fft(Mix_Signal, N); %对信号 Mix_Signal 做 FFT
```

```
plot(f, abs(y));
```

```
xlabel('频率/Hz');
```

```
ylabel('振幅');
```

```
title('原始信号 FFT');
```

```
grid on;
```

```
b = fir1(30, 0.25); %30 阶 FIR 低通滤波器, 截止频率 125Hz
```

```
%y2= filter(b, 1, x);
```

```
y2=filtfilt(b, 1, x); %经过 FIR 滤波器后得到的信号
```

```
Ps=sum(Signal_Original.^2); %信号的总功率
```

```
Pu=sum((y2-Signal_Original).^2); %剩余噪声的功率
```

```
SNR=10*log10(Ps/Pu); %信噪比
```

```
y3=fft(y2, N); %经过 FIR 滤波器后得到的信号做 FFT
```

```
subplot(223);
```

```
plot(f, abs(y3));
```

```
xlabel('频率/Hz');
```

```
ylabel('振幅');
```

```
title('滤波后信号 FFT');
```

```
grid on;
```

```
[H, F]=freqz(b, 1, 512); %通过 fir1 设计的 FIR 系统的频率响应
```

```
subplot(224);
```

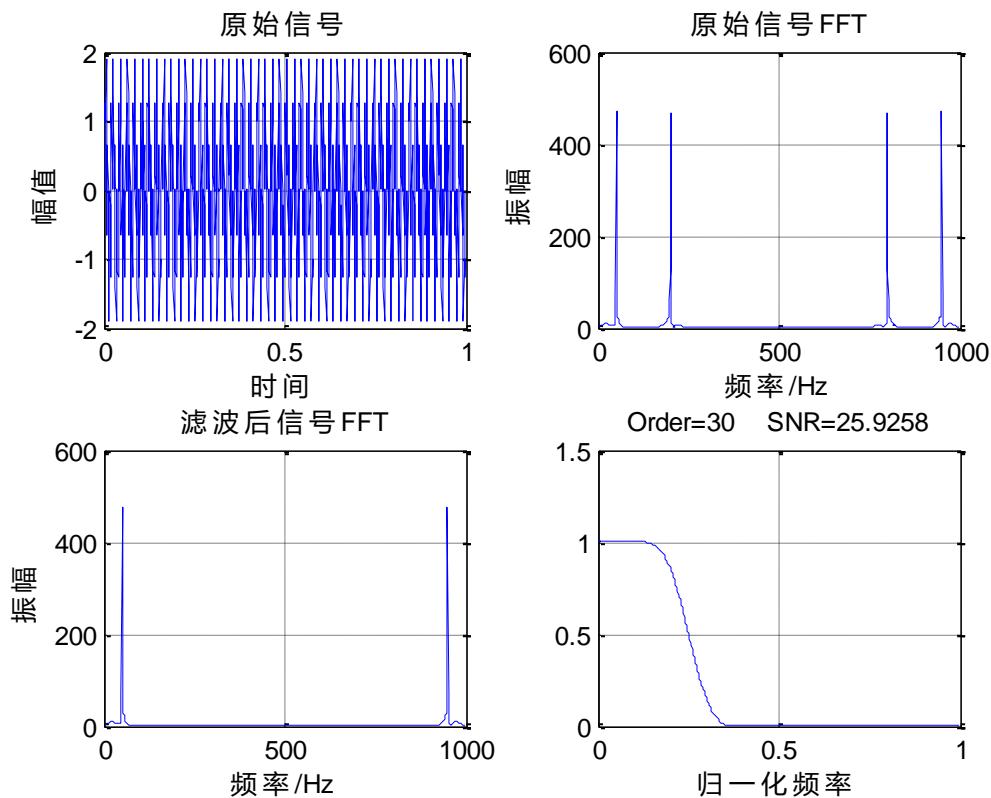
```
plot(F/pi, abs(H)); %绘制幅频响应
```

```
xlabel('归一化频率');
```

```
title(['Order=', int2str(30), ' SNR=', num2str(SNR)]);
```

```
grid on;
```

Matlab 的运行结果如下：



从运行结果的 FFT 和信噪比来看，滤波效果比较明显。

36.2.3 fir1 设计高通滤波器实例

下面我们通过一个实例来讲解 fir1 的高通滤波器的用法。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，将 50Hz 的正弦波当做噪声滤掉，下面通过函数 fir1 设计一组高通滤波器系数，其阶数是 30，截止频率为 0.25 (也就是 125Hz)。Matlab 运行代码如下：

```
%*****
% FIR 高通滤波器设计
%%%%%
fs=1000; %设置采样频率 1k
N=1024; %采样点数
n=0:N-1;
t=0:1/fs:1-1/fs; %时间序列
f=n*fs/N; %频率序列

Signal_Original=sin(2*pi*200*t); %信号 200Hz 正弦波
Signal_Noise=sin(2*pi*50*t); %噪声 50Hz 正弦波

Mix_Signal=Signal_Original+Signal_Noise; %将信号 Signal_Original 和 Signal_Original 合成一个信号进行采样
subplot(221);
plot(t, Mix_Signal); %绘制信号 Mix_Signal 的波形
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;
```



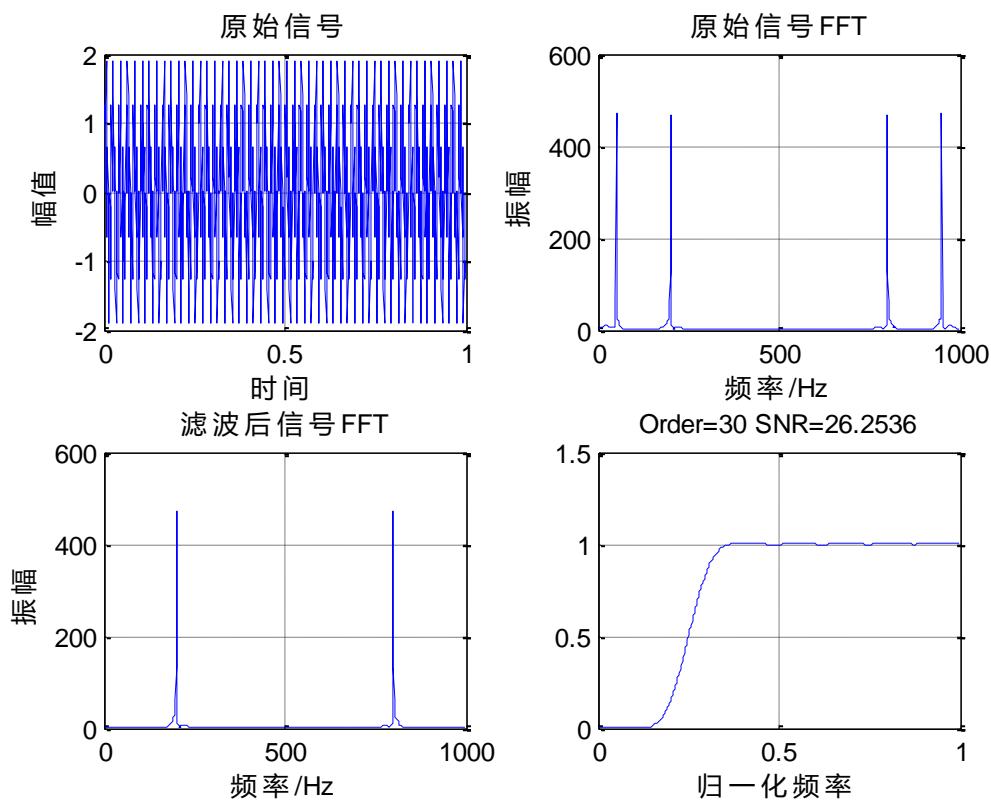
```
subplot(222);
y=fft(Mix_Signal, N);      %对信号 Mix_Signal 做 FFT
plot(f,abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

b = fir1(30, 0.25, 'high');    %30 阶 FIR 低通滤波器，截止频率 125Hz
%y2= filter(b, 1, x);
y2=filtfilt(b, 1, x);          %经过 FIR 滤波器后得到的信号
Ps=sum(Signal_Original.^2);     %信号的总功率
Pu=sum((y2-Signal_Original).^2); %剩余噪声的功率
SNR=10*log10(Ps/Pu);           %信噪比

y3=fft(y2, N);                %经过 FIR 滤波器后得到的信号做 FFT
subplot(223);
plot(f,abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

[H, F]=freqz(b, 1, 512);       %通过 fir1 设计的 FIR 系统的频率响应
subplot(224);
plot(F/pi, abs(H));            %绘制幅频响应
xlabel('归一化频率');
title(['Order=' int2str(30), ' SNR=' num2str(SNR)]);
grid on;
```

Matlab 的运行结果如下：



从运行结果的 FFT 和信噪比来看，滤波效果比较明显。

36.2.4 fir1 设计带通滤波器实例

下面我们通过一个实例来讲解 fir1 的带通滤波器的用法。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，设计通带为 125Hz 到 300Hz，下面通过函数 fir1 设计一组带通滤波器系数，其阶数是 30，通带为 $0.25 < W < 0.6$ 。Matlab 运行代码如下：

```
%*****FIR 带通滤波器设计*****
%设置采样频率 1k
N=1024; %采样点数
n=0:N-1;
t=0:1/fs:1-1/fs; %时间序列
f=n*fs/N; %频率序列

Signal_Original=sin(2*pi*200*t); %信号 200Hz 正弦波
Signal_Noise=sin(2*pi*50*t); %噪声 50Hz 正弦波

Mix_Signal=Signal_Original+Signal_Noise; %将信号 Signal_Original 和 Signal_Original 合成一个信号进行采样
subplot(2,1,1);
plot(t, Mix_Signal); %绘制信号 Mix_Signal 的波形
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;
```



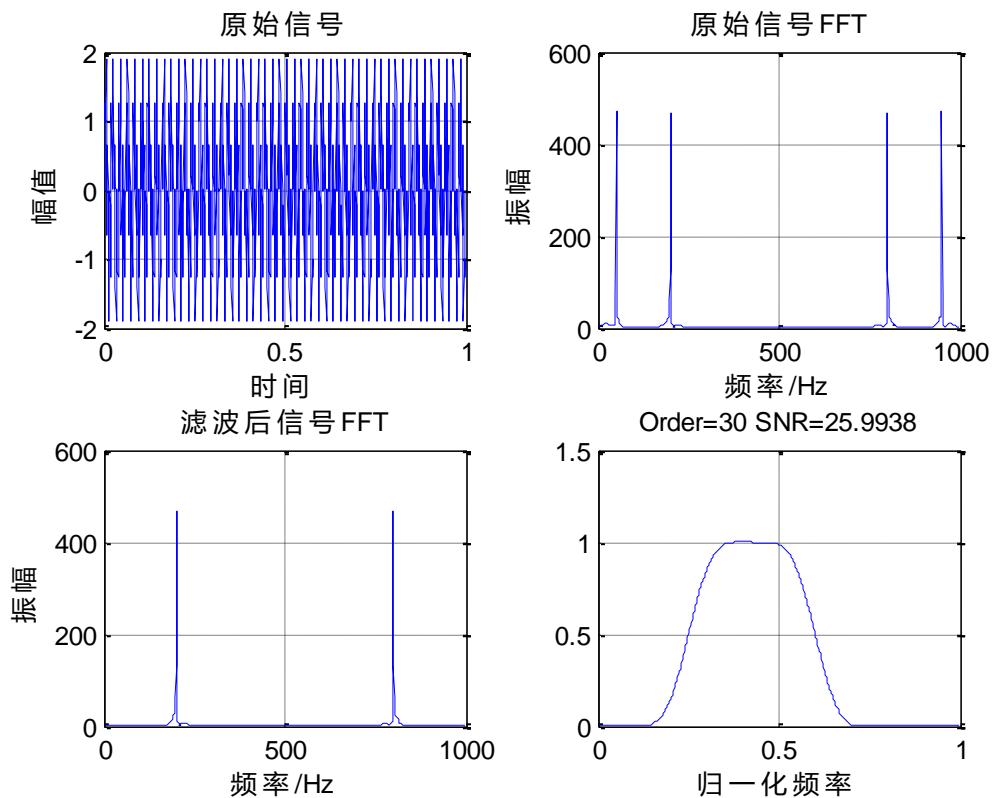
```
subplot(222);
y=fft(Mix_Signal, N);      %对信号 Mix_Signal 做 FFT
plot(f,abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

b = fir1(30, [0.25 0.6]);    %30 阶 FIR 低通滤波器, 截止频率 125Hz
%y2= filter(b, 1, x);
y2=filtfilt(b, 1, x);        %经过 FIR 滤波器后得到的信号
Ps=sum(Signal_Original.^2);    %信号的总功率
Pu=sum((y2-Signal_Original).^2);    %剩余噪声的功率
SNR=10*log10(Ps/Pu);          %信噪比

y3=fft(y2, N);              %经过 FIR 滤波器后得到的信号做 FFT
subplot(223);
plot(f,abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

[H, F]=freqz(b, 1, 512);    %通过 fir1 设计的 FIR 系统的频率响应
subplot(224);
plot(F/pi, abs(H));         %绘制幅频响应
xlabel('归一化频率');
title(['Order=' int2str(30), ' SNR=' num2str(SNR)]);
grid on;
```

Matlab 运行结果如下：



从运行结果的 FFT 和信噪比来看，滤波效果比较明显。

36.2.5 fir1 设计带阻滤波器实例

下面我们通过一个实例来讲解 fir1 的带阻滤波器的用法。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，设计阻带为 125Hz 到 300Hz，下面通过函数 fir1 设计一组带阻滤波器系数，其阶数是 30，阻

带为 $0.25 < W < 0.6$ 。Matlab 运行代码如下：

```
%*****FIR 带阻滤波器设计*****
%*****设置采样频率 1k
fs=1000;
N=1024;
n=0:N-1;
t=0:1/fs:1-1/fs;
f=n*fs/N;

Signal_Original=sin(2*pi*50*t); %信号 50Hz 正弦波
Signal_Noise=sin(2*pi*200*t); %噪声 200Hz 正弦波

Mix_Signal=Signal_Original+Signal_Noise; %将信号 Signal_Original 和 Signal_Original 合成一个信号进行采样
subplot(221);
plot(t, Mix_Signal); %绘制信号 Mix_Signal 的波形
xlabel('时间');
ylabel('幅值');
```



```
title('原始信号');
grid on;

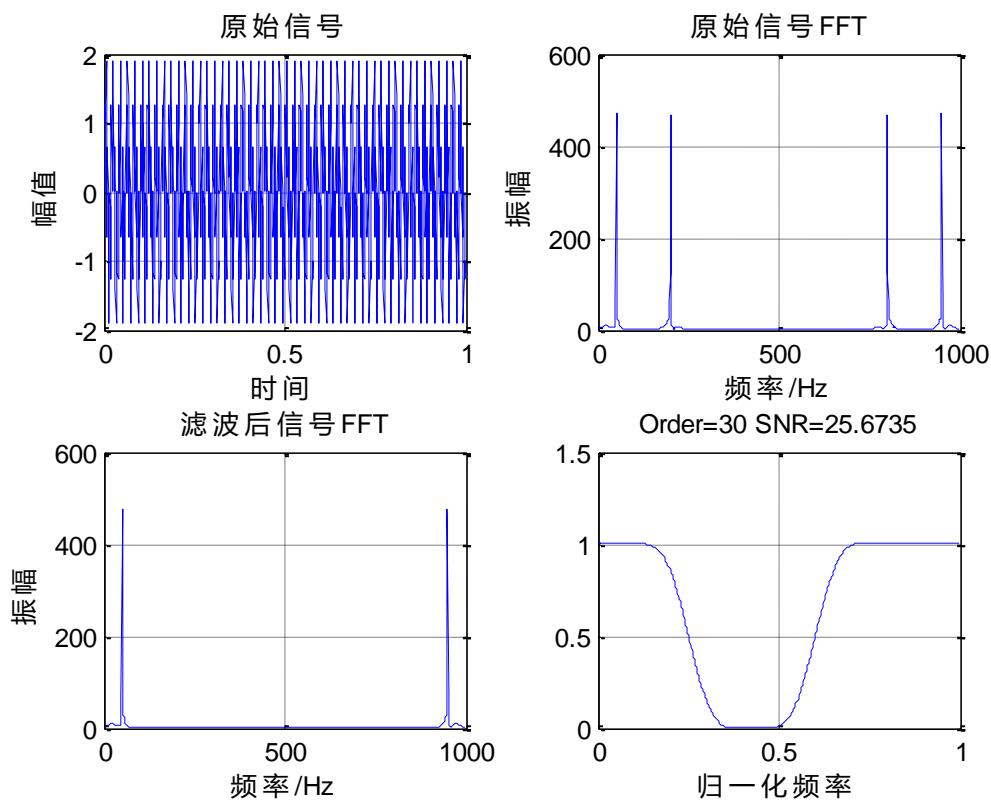
subplot(222);
y=fft(Mix_Signal, N);      %对信号 Mix_Signal 做 FFT
plot(f,abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

b = fir1(30, [0.25 0.6], 'stop');    %30 阶 FIR 低通滤波器，截止频率 125Hz
%y2= filter(b, 1, x);
y2=filtfilt(b, 1, x);                %经过 FIR 滤波器后得到的信号
Ps=sum(Signal_Original.^2);    %信号的总功率
Pu=sum((y2-Signal_Original).^2);   %剩余噪声的功率
SNR=10*log10(Ps/Pu);            %信噪比

y3=fft(y2, N);                  %经过 FIR 滤波器后得到的信号做 FFT
subplot(223);
plot(f,abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

[H, F]=freqz(b, 1, 512);        %通过 fir1 设计的 FIR 系统的频率响应
subplot(224);
plot(F/pi, abs(H));           %绘制幅频响应
xlabel('归一化频率');
title(['Order=', int2str(30), ' SNR=', num2str(SNR)]);
grid on;
```

Matlab 运行结果如下：



从运行结果的 FFT 和信噪比来看，滤波效果比较明显。

36.2.6 切比雪夫窗口函数设计带通滤波器实例

下面我们通过一个实例来讲解 fir1 设计切比雪夫窗口的的带通滤波器。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，设计通带为 125Hz 到 300Hz，下面通过函数 fir1 设计一组带通滤波器系数，其阶数是 30，通带为 $0.25 < W < 0.6$ ，并且具有 25db 波纹的切比雪夫窗。Matlab 运行代码如下：

```
%*****切比雪夫窗口函数设计带通滤波器*****
%设置采样频率 1k
fs=1000;
N=1024;
n=0:N-1;
t=0:1/fs:1-1/fs;
f=n*fs/N;

Signal_Original=sin(2*pi*200*t); %信号 200Hz 正弦波
Signal_Noise=sin(2*pi*50*t); %噪声 50Hz 正弦波

Mix_Signal=Signal_Original+Signal_Noise; %将信号 Signal_Original 和 Signal_Original 合成一个信号进行采样
subplot(221);
plot(t, Mix_Signal); %绘制信号 Mix_Signal 的波形
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;
```



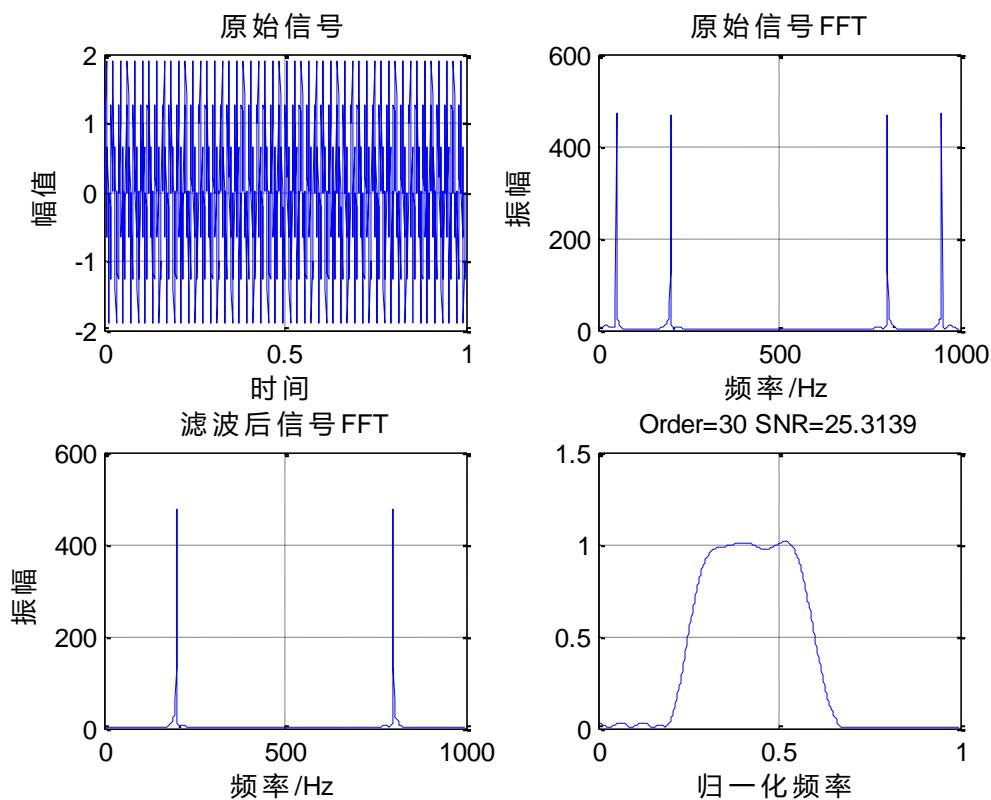
```
subplot(222);
y=fft(Mix_Signal, N);      %对信号 Mix_Signal 做 FFT
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

Window = chebwin(31, 25);      %25db 的切比雪夫窗
b = fir1(30, [0.25 0.6], Window);    %30 阶 FIR 低通滤波器, 截止频率 125Hz
%y2= filter(b, 1, x);
y2=filtfilt(b, 1, x);          %经过 FIR 滤波器后得到的信号
Ps=sum(Signal_Original.^2);    %信号的总功率
Pu=sum((y2-Signal_Original).^2); %剩余噪声的功率
SNR=10*log10(Ps/Pu);          %信噪比

y3=fft(y2, N);                %经过 FIR 滤波器后得到的信号做 FFT
subplot(223);
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

[H, F]=freqz(b, 1, 512);      %通过 fir1 设计的 FIR 系统的频率响应
subplot(224);
plot(F/pi, abs(H));           %绘制幅频响应
xlabel('归一化频率');
title(['Order=' int2str(30), ' SNR=' num2str(SNR)]);
grid on;
```

Matlab 运行结果如下：



通过归一化频率可以看出切比雪夫窗口是有一定纹波的。不过从 FFT 结果和信噪比来看，通过切比雪夫窗口做的滤波效果也是比较明显的。

36.3 fir2 函数

36.3.1 fir2 函数介绍

函数 fir2 用来设计有任意频率响应的各种加窗 FIR 滤波器。

语法：

```
b = fir2(n,f,m)
b = fir2(n,f,m>window)
b = fir2(n,f,m,npt)
b = fir2(n,f,m,npt>window)
b = fir2(n,f,m,npt,lap)
b = fir2(n,f,m,npt,lap>window)
```

参数 n 为滤波器的阶数。

参数 f 为频率点矢量，且 $f \in [0, 1]$, $f=1$ 对应于 $0.5f_s$ 。矢量 f 按升序排列，且第一个元素必须为 0，最后一个必须为 1，并可以包含重复的频率点。



参数 m 为幅度点矢量，在矢量 m 中包含了与 f 相对应的期望得到的滤波器幅度。

参数 Window 用来指导所使用的窗函数类型，其默认值为汉明窗。

参数 npt 用来指定 fir2 函数对频率响应进行内插的点数。

参数 lap 用来指定 fir2 函数在重复频率点附近插入的区域大小。

36.3.2 fir2 设计低通滤波器

fir2 函数是用来设计任意频率响应的各种加窗 FIR 滤波器，此函数使用也比较简单，但是要采样的频率点和幅值不好把握，关于这个函数我们仅提供一个低通滤波器的设计。

原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，将 200Hz 的正弦波当做噪声滤掉，下面通过函数 fir2 进行设计。其中频率点矢量和幅度点矢量配置如下：

F = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 1];

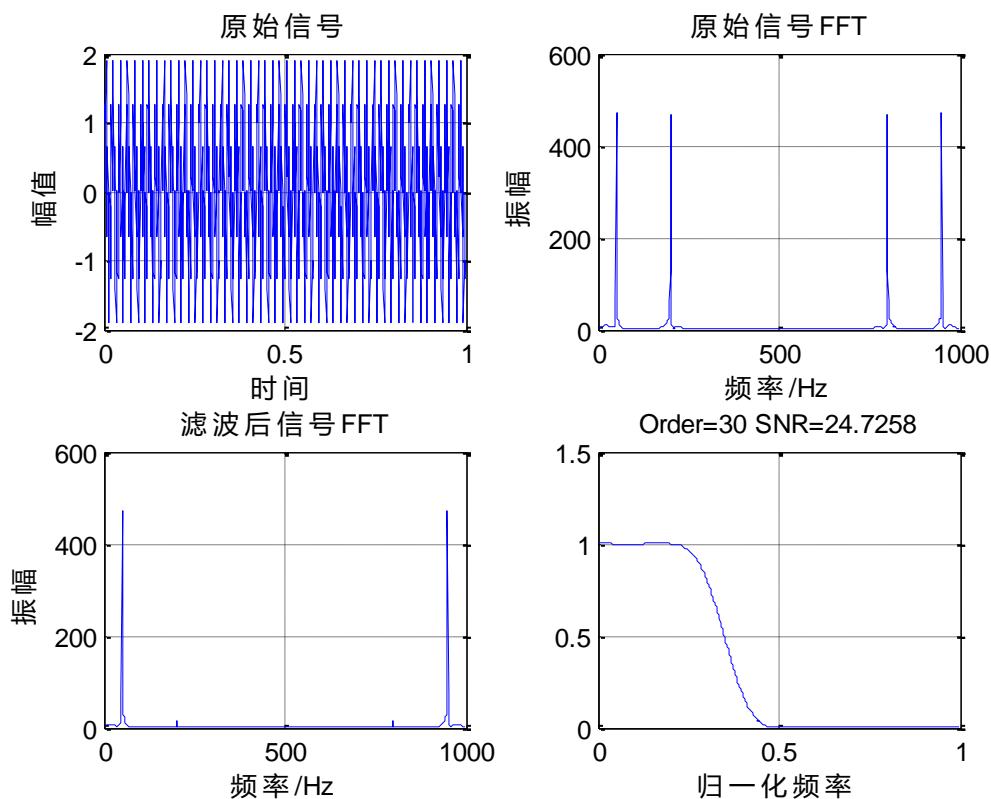
A = [1 1 1 1 0 0 0 0 0 0];

Matlab 运行的代码如下：

```
%*****  
%          fir2 设计低通滤波器  
%*****  
fs=1000;           %设置采样频率 1k  
N=1024;           %采样点数  
n=0:N-1;  
t=0:1/fs:1-1/fs;    %时间序列  
f=n*fs/N;          %频率序列  
  
Signal_Original=sin(2*pi*50*t);      %信号 50Hz 正弦波  
Signal_Noise=sin(2*pi*200*t);       %噪声 200Hz 正弦波  
  
Mix_Signal=Signal_Original+Signal_Noise;    %将信号 Signal_Original 和 Signal_Original 合成一个信号进行采样  
subplot(221);  
plot(t, Mix_Signal);    %绘制信号 Mix_Signal 的波形  
xlabel('时间');  
ylabel('幅值');  
title('原始信号');  
grid on;  
  
subplot(222);  
y=fft(Mix_Signal, N);    %对信号 Mix_Signal 做 FFT  
plot(f, abs(y));  
xlabel('频率/Hz');  
ylabel('振幅');  
title('原始信号 FFT');  
grid on;  
  
F = [0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 1];  %表示要采样的点  
A = [1 1 1 1 0 0 0 0 0 0];    %表示采样点的幅值  
b = fir2(30, F, A);    %30 阶 FIR 低通滤波器  
%y2= filter(b, 1, x);  
y2=filtfilt(b, 1, x);    %经过 FIR 滤波器后得到的信号  
Ps=sum(Signal_Original.^2);    %信号的总功率
```

```
Pu=sum((y2-Signal_Original).^2); %剩余噪声的功率  
SNR=10*log10(Ps/Pu); %信噪比  
  
y3=fft(y2, N); %经过 FIR 滤波器后得到的信号做 FFT  
subplot(223);  
plot(f, abs(y3));  
xlabel('频率/Hz');  
ylabel('振幅');  
title('滤波后信号 FFT');  
grid on;  
  
[H, F]=freqz(b, 1, 512); %通过 fir1 设计的 FIR 系统的频率响应  
subplot(224);  
plot(F/pi, abs(H)); %绘制幅频响应  
xlabel('归一化频率');  
title(['Order=' int2str(30), ' SNR=' num2str(SNR)]);  
grid on;
```

Matlab 运行结果如下：



从 FFT 结果和信噪比来看，fir2 任意滤波器设计效果也是比较明显的。

36.4 总结

本章节主要讲解了函数 fir1 和函数 fir2 的使用，想深入的掌握这两个函数，还需要大家多多练习。



第37章 STM32H7 的 FIR 低通滤波器实现（支持逐个数据的实时滤波）

本章节讲解 FIR 低通滤波器实现。

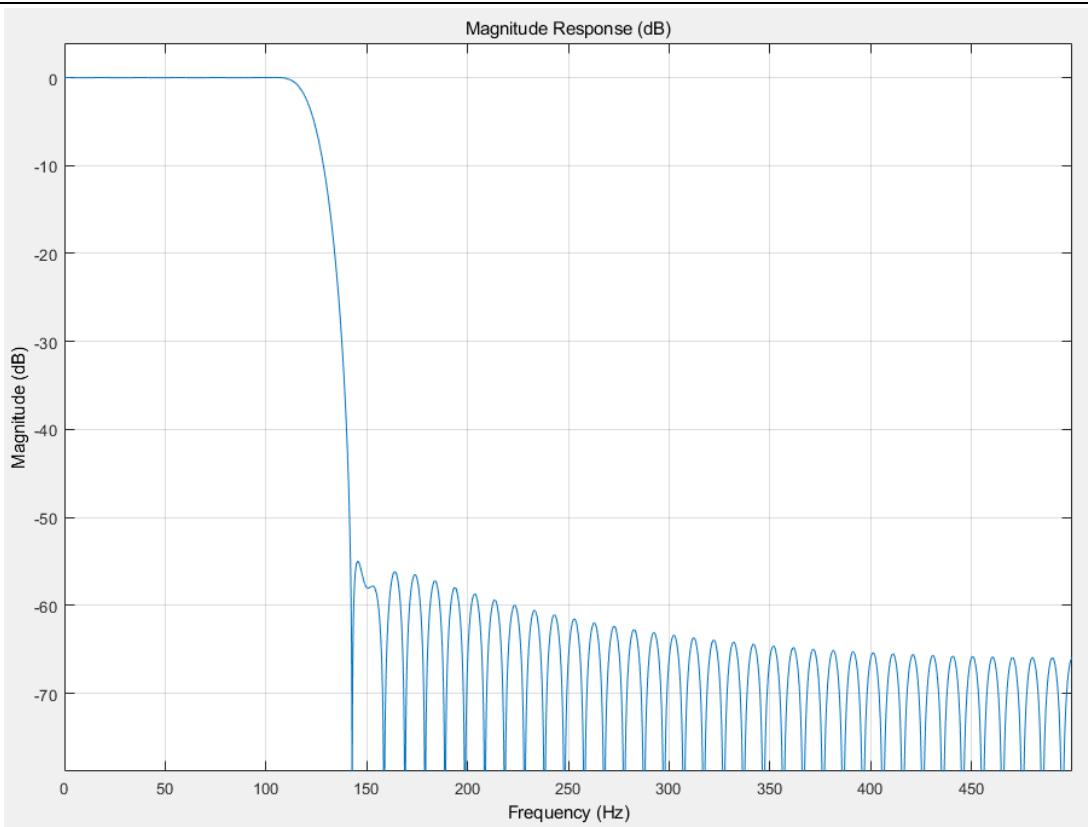
- 37.1 初学者重要提示
- 37.2 低通滤波器介绍
- 37.3 FIR 滤波器介绍
- 37.4 Matlab 工具箱 filterDesigner 生成低通滤波器 C 头文件
- 37.5 FIR 低通滤波器设计
- 37.6 实验例程说明 (MDK)
- 37.7 实验例程说明 (IAR)
- 37.8 总结

37.1 初学者重要提示

- ◆ 本章节提供的低通滤波器支持实时滤波，每次可以滤波一个数据，也可以多个数据，不限制大小。但要注意以下两点：
 - 所有数据是在同一个采样率下依次采集的数据。
 - 每次过滤数据个数一旦固定下来，运行中不可再修改。
- ◆ FIR 滤波器的群延迟是一个重要的知识点，详情在本教程第 41 章有详细说明。

37.2 低通滤波器介绍

允许低频信号通过，而减弱高于截止频率的信号通过。比如混合信号含有 50Hz + 200Hz 信号，我们可通过低通滤波器，过滤掉 200Hz 信号，让 50Hz 信号通过。



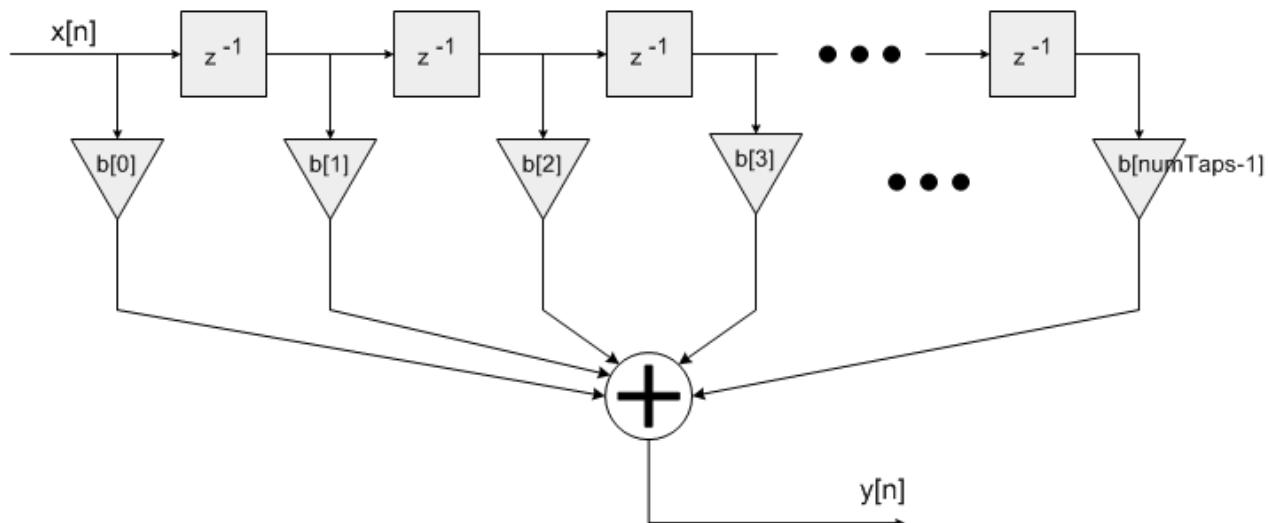
37.3 FIR 滤波器介绍

ARM 官方提供的 FIR 库支持 Q7, Q15, Q31 和浮点四种数据类型。其中 Q15 和 Q31 提供了快速算法版本。

FIR 滤波器的基本算法是一种乘法-累加 (MAC) 运行，输出表达式如下：

$$y[n] = b[0] * x[n] + b[1] * x[n-1] + b[2] * x[n-2] + \dots + b[\text{numTaps}-1] * x[n-\text{numTaps}+1]$$

结构图如下：



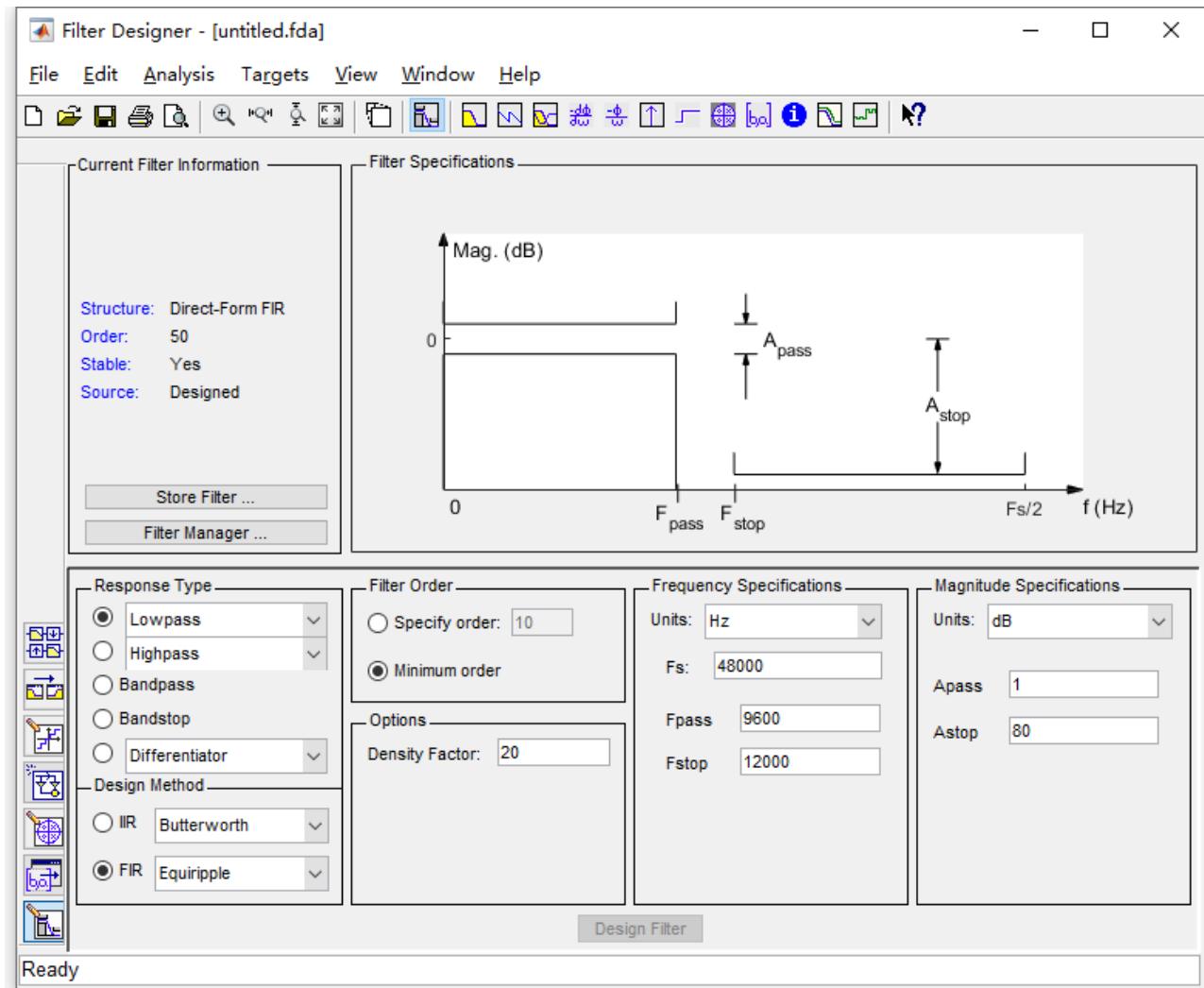
这种网络结构就是在 35.2.1 小节所讲的直接型结构。

37.4 Matlab 工具箱 filterDesigner 生成低通滤波器 C 头文件

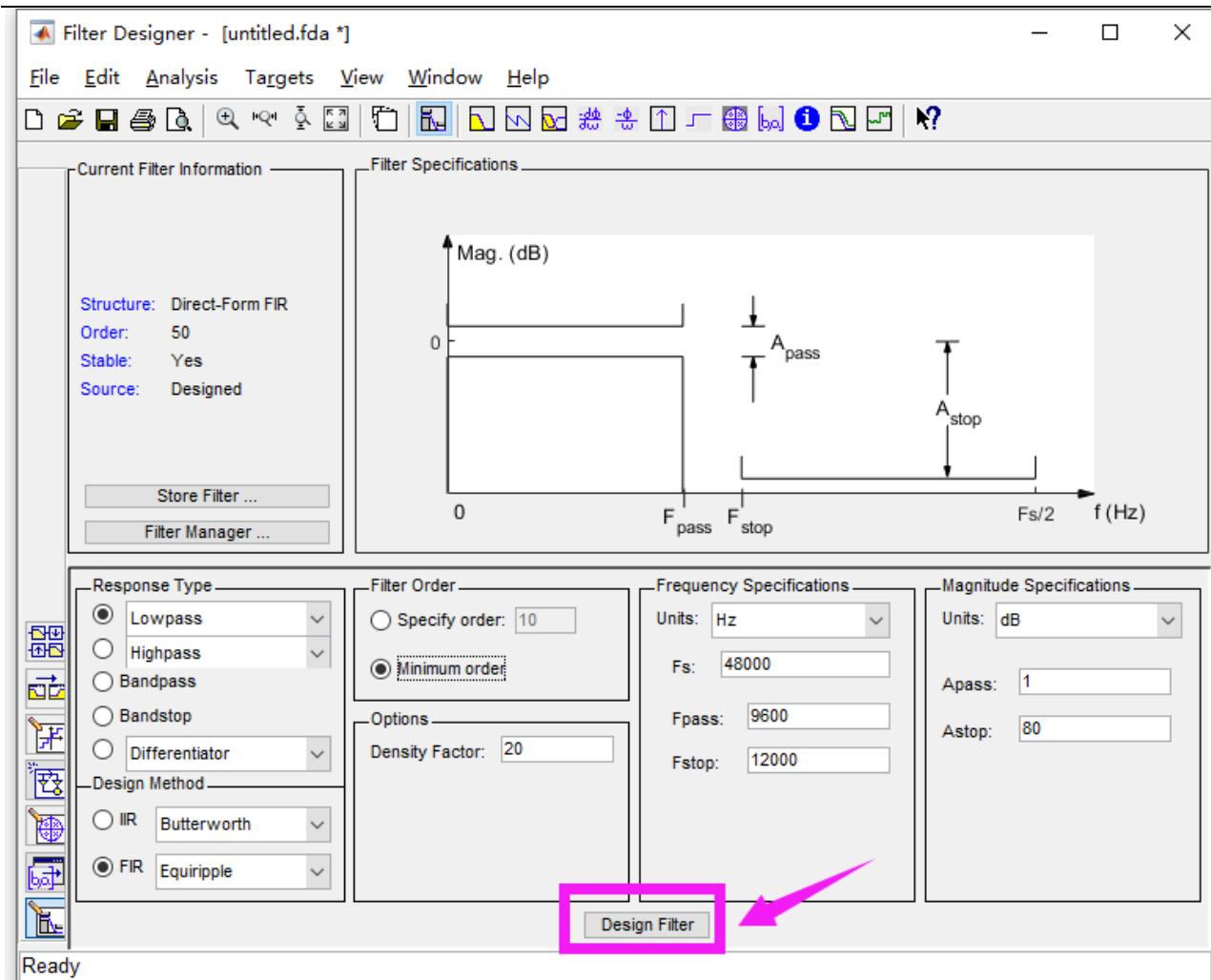
下面我们讲解下如何通过 filterDesigner 工具生成 C 头文件，也就是生成滤波器系数。首先在 matlab 的命窗口输入 filterDesigner 就能打开这个工具箱：



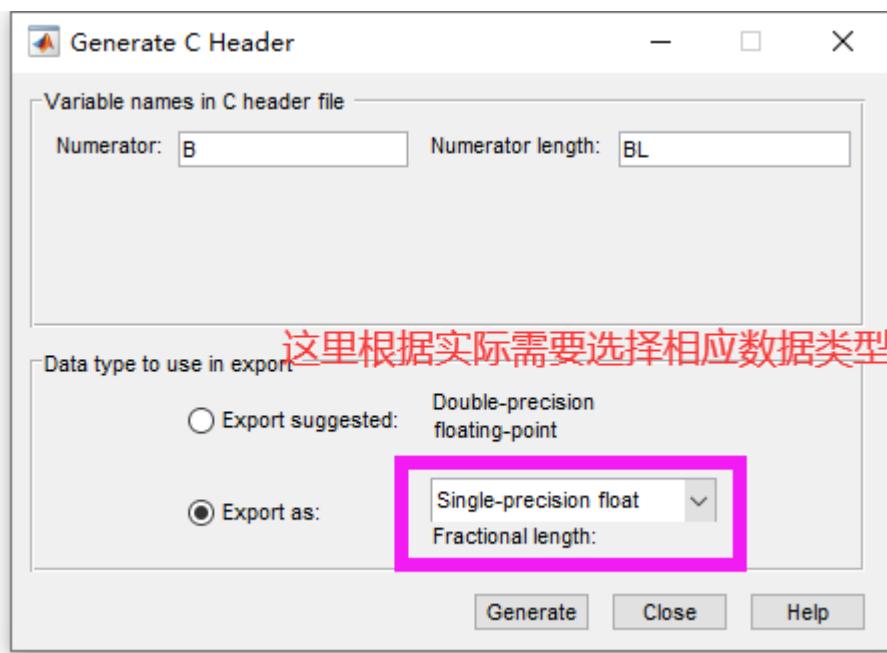
filterDesigner 界面打开效果如下：



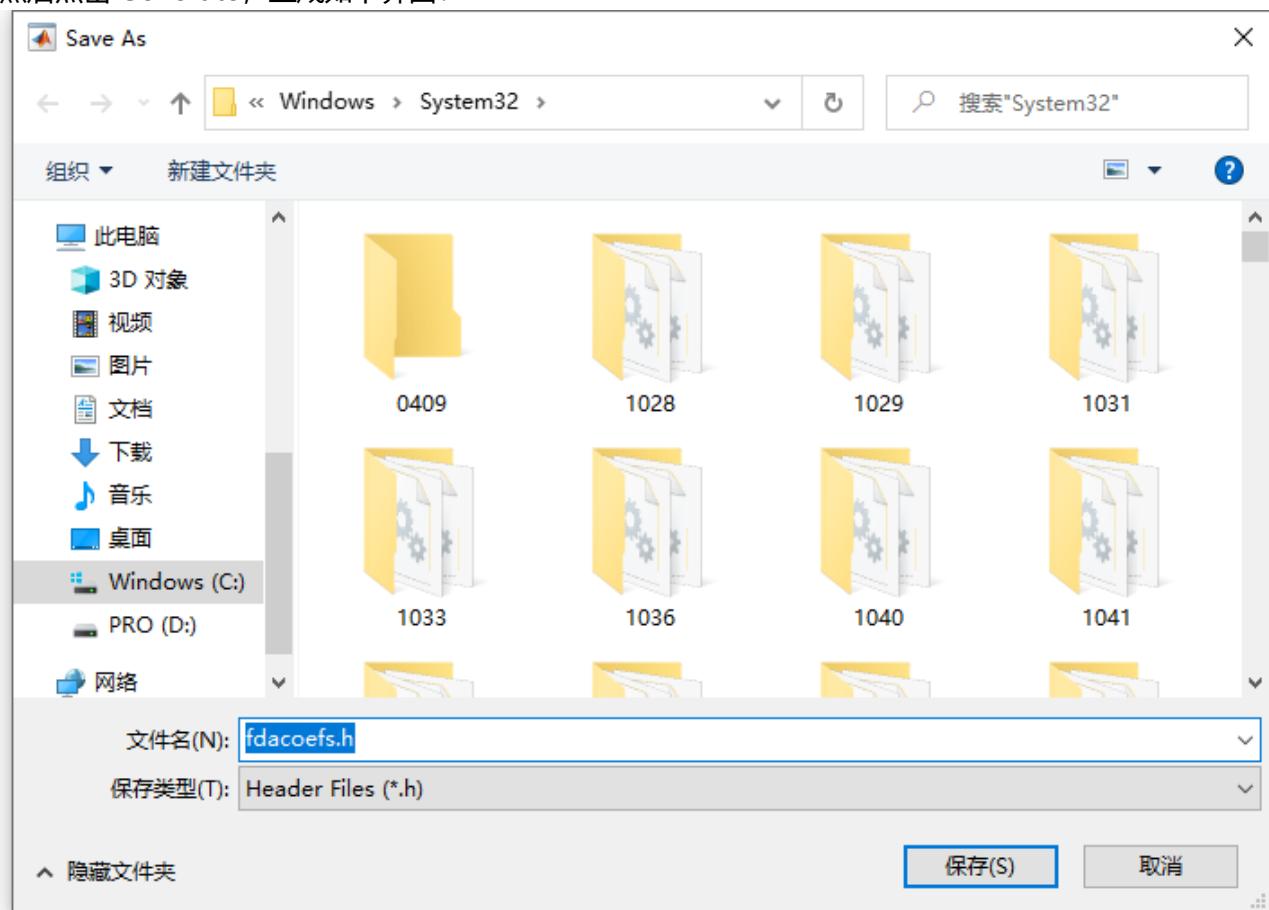
FIR 滤波器的低通，高通，带通，带阻滤波的设置会在后面逐个讲解，这里重点介绍设置后相应参数后如何生成滤波器系数。参数设置好以后点击如下按钮：



点击 Design Filter 按钮以后就生成了所需的滤波器系数，生成滤波器系数以后点击 filterDesigner 界面上的菜单 Targets->Generate C header ,打开后显示如下界面：



然后点击 Generate，生成如下界面：



再点击保存，并打开 fdatool.h 文件，可以看到生成的系数：

```
/*
 * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
 * Generated by MATLAB(R) 9.4 and Signal Processing Toolbox 8.0.
 * Generated on: 20-Jul-2021 12:19:30
 */

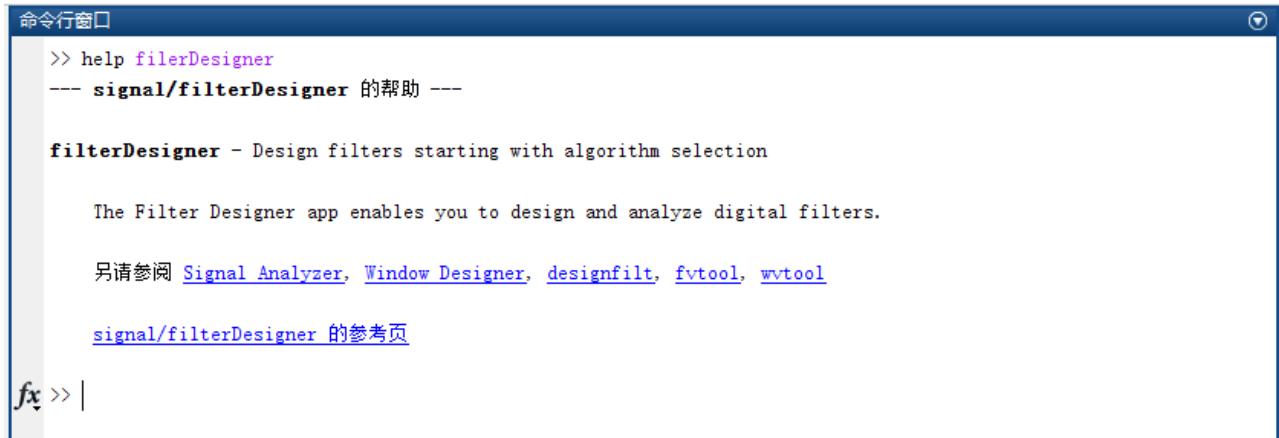
/*
 * Discrete-Time FIR Filter (real)
 * -----
 * Filter Structure : Direct-Form FIR
 * Filter Length   : 51
 * Stable          : Yes
 * Linear Phase    : Yes (Type 1)
 */

/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"
/*
 * Expected path to tmwtypes.h
 * D:\Program Files\MATLAB\R2018a\extern\include\tmwtypes.h
 */
/*
 * Warning - Filter coefficients were truncated to fit specified data type.
 * The resulting response may not match generated theoretical response.
 * Use the Filter Design & Analysis Tool to design accurate
 * single-precision filter coefficients.
 */
const int BL = 51;
const real32_T B[51] = {
    -0.0009190982091, -0.00271769613, -0.002486952813, 0.003661438357, 0.0136509249,
    0.01735116541, 0.00766530633, -0.006554719061, -0.007696784101, 0.006105459295,
```



```
0.01387391612, 0.0003508617228, -0.01690892503, -0.008905642666, 0.01744112931,  
0.02074504457, -0.0122964941, -0.03424086422, -0.001034529647, 0.04779030383,  
0.02736303769, -0.05937951803, -0.08230702579, 0.06718690693, 0.3100151718,  
0.4300478697, 0.3100151718, 0.06718690693, -0.08230702579, -0.05937951803,  
0.02736303769, 0.04779030383, -0.001034529647, -0.03424086422, -0.0122964941,  
0.02074504457, 0.01744112931, -0.008905642666, -0.01690892503, 0.0003508617228,  
0.01387391612, 0.006105459295, -0.007696784101, -0.006554719061, 0.00766530633,  
0.01735116541, 0.0136509249, 0.003661438357, -0.002486952813, -0.00271769613,  
-0.0009190982091  
};
```

上面数组 B[51]中的数据就是滤波器系数。下面小节讲解如何使用 filterDesigner 配置 FIR 低通，高通，带通和带阻滤波。关于 Filter Designer 的其它用法，大家可以在 matlab 命令窗口中输入 help filterDesigner 打开帮助文档进行学习。



37.5 FIR 低通滤波器设计

本章使用的 FIR 滤波器函数是 arm_fir_f32。使用此函数可以设计 FIR 低通，高通，带通和带阻滤波器。

37.5.1 函数 arm_fir_init_f32

函数原型：

```
void arm_fir_init_f32(  
    arm_fir_instance_f32 * S,  
    uint16_t numTaps,  
    const float32_t * pCoeffs,  
    float32_t * pState,  
    uint32_t blockSize);
```

函数描述：

这个函数用于 FIR 初始化。

函数参数：

- ◆ 第 1 个参数是 arm_fir_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是滤波器系数的个数。
- ◆ 第 3 个参数是滤波器系数地址。
- ◆ 第 4 个参数是缓冲状态地址。



- ◆ 第 5 个参数是每次处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

注意事项：

结构体 arm_fir_instance_f32 的定义如下（在文件 arm_math.h 文件）：

```
typedef struct
{
    uint16_t numTaps;      /**< number of filter coefficients in the filter. */
    float32_t *pState;     /**< points to the state variable array. The array is of length */
                           numTaps+blockSize-1;
    float32_t *pCoeffs;    /**< points to the coefficient array. The array is of length numTaps. */
} arm_fir_instance_f32;
```

1. 参数 pCoeffs 指向滤波因数，滤波因数数组长度为 numTaps。但要注意 pCoeffs 指向的滤波因数应该按照如下的逆序进行排列：
 $\{b[numTaps-1], b[numTaps-2], b[N-2], \dots, b[1], b[0]\}$
但满足线性相位特性的 FIR 滤波器具有奇对称或者偶对称的系数，偶对称时逆序排列还是他本身。
2. pState 指向状态变量数组，这个数组用于函数内部计算数据的缓存。
3. blockSize 这个参数的大小没有特殊要求，最小可以每次处理 1 个数据，最大可以每次全部处理完。

37.5.2 函数 arm_fir_f32

函数原型：

```
void arm_fir_f32(
    const arm_fir_instance_f32 * S,
    const float32_t * pSrc,
    float32_t * pDst,
    uint32_t blockSize)
```

函数描述：

这个函数用于 FIR 滤波。

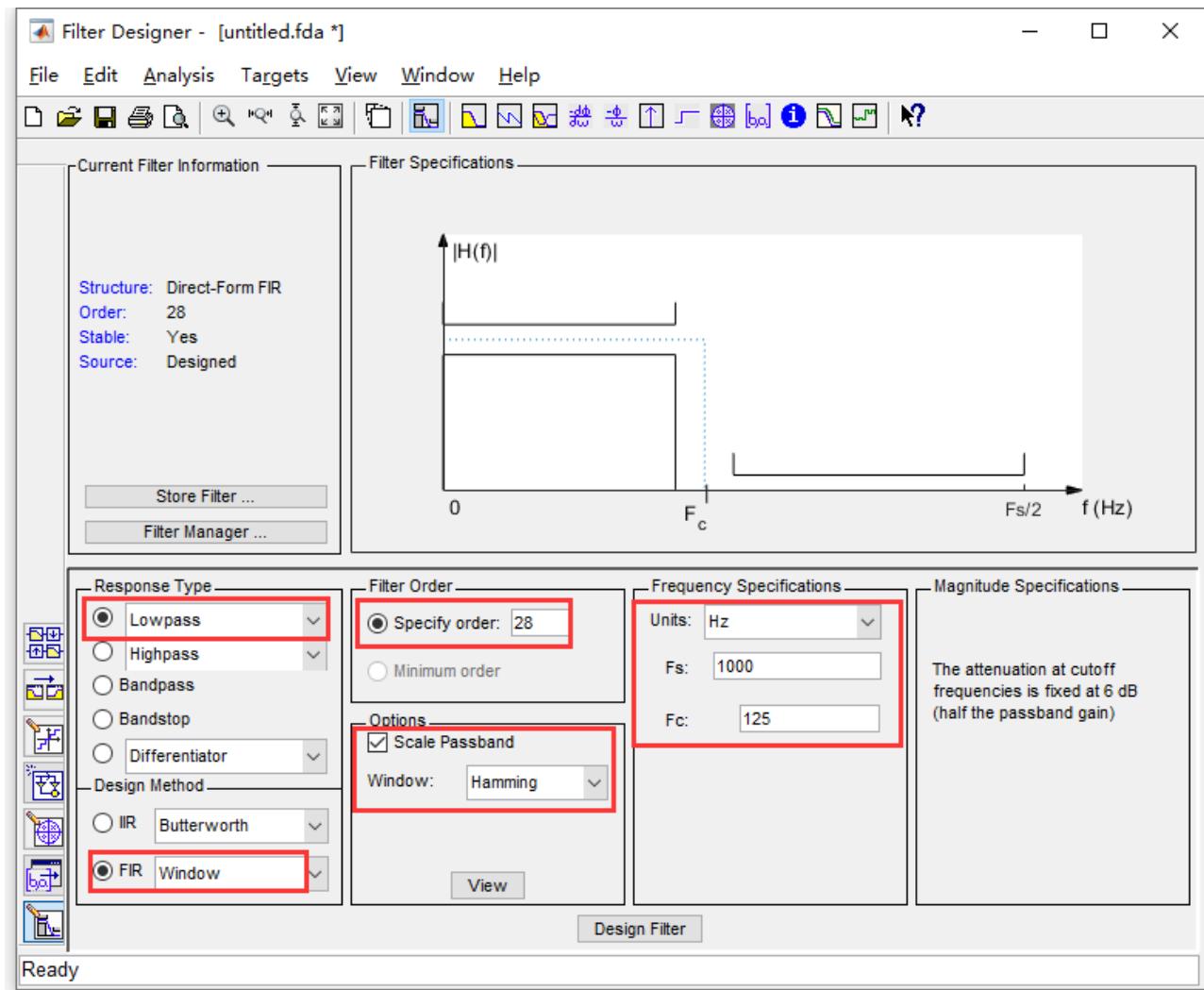
函数参数：

- ◆ 第 1 个参数是 arm_fir_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是源数据地址。
- ◆ 第 3 个参数是滤波后的数据地址。
- ◆ 第 4 个参数是每次调用处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

37.5.3 filterDesigner 获取低通滤波器系数

设计一个如下的例子：

信号由 50Hz 正弦波和 200Hz 正弦波组成，采样率 1Kbps，现设计一个低通滤波器，截止频率 125Hz，采样 1024 个数据，采用函数 fir1 进行设计（注意这个函数是基于窗口的方法设计 FIR 滤波，默认是 hamming 窗），滤波器阶数设置为 28。filterDesigner 的配置如下：



配置好低通滤波器后，具体滤波器系数的生成大家参考本章第4小节的方法即可。

37.5.4 低通滤波器实现

通过工具箱filterDesigner获得低通滤波器系数后在开发板上运行函数arm_fir_f32 来测试低通滤波器的效果。

```
#define TEST_LENGTH_SAMPLES 1024 /* 采样点数 */
#define BLOCK_SIZE 1 /* 调用一次arm_fir_f32处理的采样点个数 */
#define NUM_TAPS 29 /* 滤波器系数个数 */

uint32_t blockSize = BLOCK_SIZE;
uint32_t numBlocks = TEST_LENGTH_SAMPLES/BLOCK_SIZE; /* 需要调用arm_fir_f32的次数 */

static float32_t testInput_f32_50Hz_200Hz[TEST_LENGTH_SAMPLES]; /* 采样点 */
static float32_t testOutput[TEST_LENGTH_SAMPLES]; /* 滤波后的输出 */
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1]; /* 状态缓存，大小numTaps + blockSize - 1 */

/* 低通滤波器系数 通过fadtool获取 */
const float32_t firCoeffs32LP[NUM_TAPS] = {
    -0.001822523074f, -0.001587929321f, 1.226008847e-18f, 0.003697750857f, 0.008075430058f,
    0.008530221879f, -4.273456581e-18f, -0.01739769801f, -0.03414586186f, -0.03335915506f,
    8.073562366e-18f, 0.06763084233f, 0.1522061825f, 0.2229246944f, 0.2504960895f,
```



```
0.2229246944f,    0.1522061825f,    0.06763084233f,    8.073562366e-18f, -0.03335915506f,
-0.03414586186f, -0.01739769801f,   -4.273456581e-18f,  0.008530221879f,  0.008075430058f,
0.003697750857f, 1.226008847e-18f,   -0.001587929321f,  -0.001822523074f
};

/*
*****
* 函数名: arm_fir_f32_1p
* 功能说明: 调用函数arm_fir_f32_1p实现低通滤波器
* 形参: 无
* 返回值: 无
*****
*/
static void arm_fir_f32_1p(void)
{
    uint32_t i;
    arm_fir_instance_f32 S;
    float32_t *inputF32, *outputF32;

    /* 初始化输入输出缓存指针 */
    inputF32 = &testInput_f32_50Hz_200Hz[0];
    outputF32 = &testOutput[0];

    /* 初始化结构体S */
    arm_fir_init_f32(&S,
                      NUM_TAPS,
                      (float32_t *)&firCoeffs32LP[0],
                      &firStateF32[0],
                      blockSize);

    /* 实现FIR滤波, 这里每次处理1个点 */
    for(i=0; i < numBlocks; i++)
    {
        arm_fir_f32(&S, inputF32 + (i * blockSize), outputF32 + (i * blockSize), blockSize);
    }

    /* 打印滤波后结果 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        printf("%f, %f\r\n", testOutput[i], inputF32[i]);
    }
}
```

运行如上函数可以通过串口打印出函数arm_fir_f32滤波后的波形数据，下面通过Matlab绘制波形来对比Matlab计算的结果和ARM官方库计算的结果。

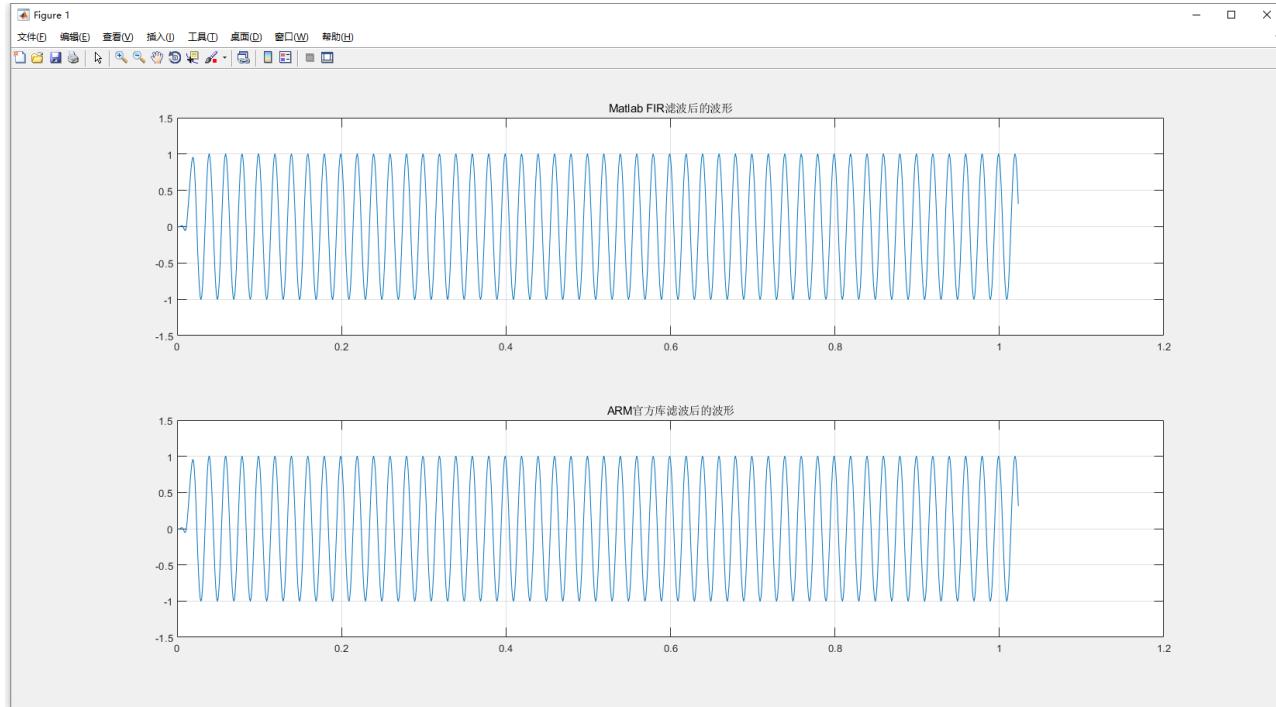
对比前需要先将串口打印出的一组数据加载到 Matlab 中，arm_fir_f32 的计算结果起名 sampledata，加载方法在第 13 章 13.6 小结已经讲解，这里不做赘述了。Matlab 中运行的代码如下：

```
%*****FIR低通滤波器设计*****
%设置采样频率 1K
fs=1000;
N=1024;
n=0:N-1;
t=n/fs;
f=n*fs/N;

x=sin(2*pi*50*t)+sin(2*pi*200*t);      %50Hz和200Hz正弦波混合
b=fir1(28, 0.25);
y=filter(b, 1, x);
subplot(211);
plot(t, y);
title('Matlab FIR滤波后的波形');
grid on;
```

```
subplot(212);
plot(t, sampledata);
title('ARM官方库滤波后的波形');
grid on;
```

Matlab 运行结果如下：



从上面的波形对比来看，matlab 和函数 arm_fir_f32 计算的结果基本是一致的。为了更好的说明滤波效果，下面从频域的角度来说明这个问题，Matlab 上面运行如下代码：

```
%*****
% FIR低通滤波器设计
%*****
fs=1000; %设置采样频率 1K
N=1024; %采样点数
n=0:N-1;
t=n/fs; %时间序列
f=n*fs/N; %频率序列

x = sin(2*pi*50*t) + sin(2*pi*200*t); %50Hz和200Hz正弦波合成

subplot(221);
plot(t, x); %绘制信号Mix_Signal的波形
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;

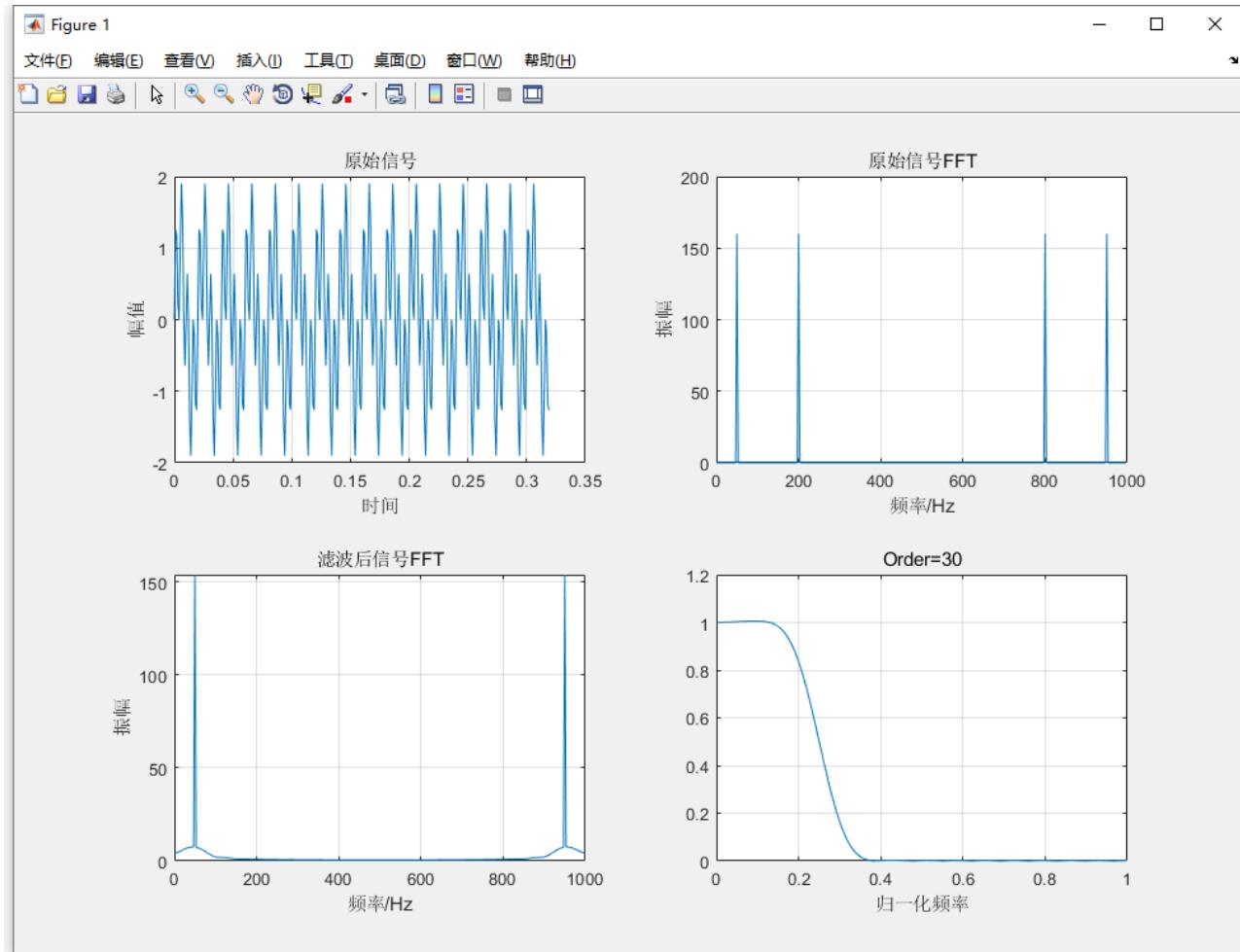
subplot(222);
y=fft(x, N); %对信号 Mix_Signal做FFT
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号FFT');
grid on;

y3=fft(sampledata, N); %经过FIR滤波器后得到的信号做FFT
subplot(223);
plot(f, abs(y3));
xlabel('频率/Hz');
```

```
ylabel('振幅');
title('滤波后信号FFT');
grid on;

b=fir1(28, 0.25); %28阶FIR低通滤波器，截止频率125Hz
[H, F]=freqz(b, 1, 512); %通过fir1设计的FIR系统的频率响应
subplot(224);
plot(F/pi, abs(H)); %绘制幅频响应
xlabel('归一化频率');
title(['Order= ', int2str(30)]);
grid on;
```

Matlab 显示效果如下：



上面波形变换前的 FFT 和变换后 FFT 可以看出，200Hz 的正弦波基本被滤除。

37.6 实验例程说明 (MDK)

配套例子：

V7-225_FIR 低通滤波器设计(支持逐个数据的实时滤波)

实验目的：

1. 学习 FIR 低通滤波器的实现，支持实时滤波

实验内容：



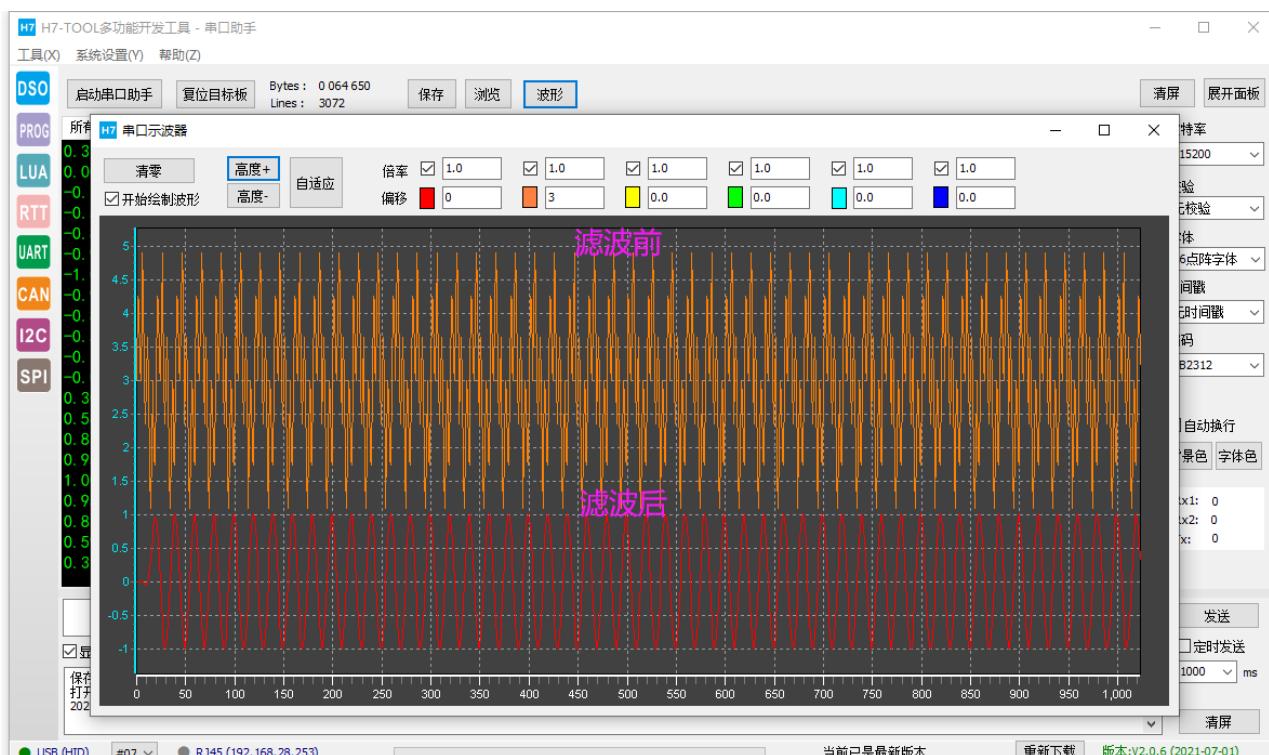
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

使用 AC6 注意事项

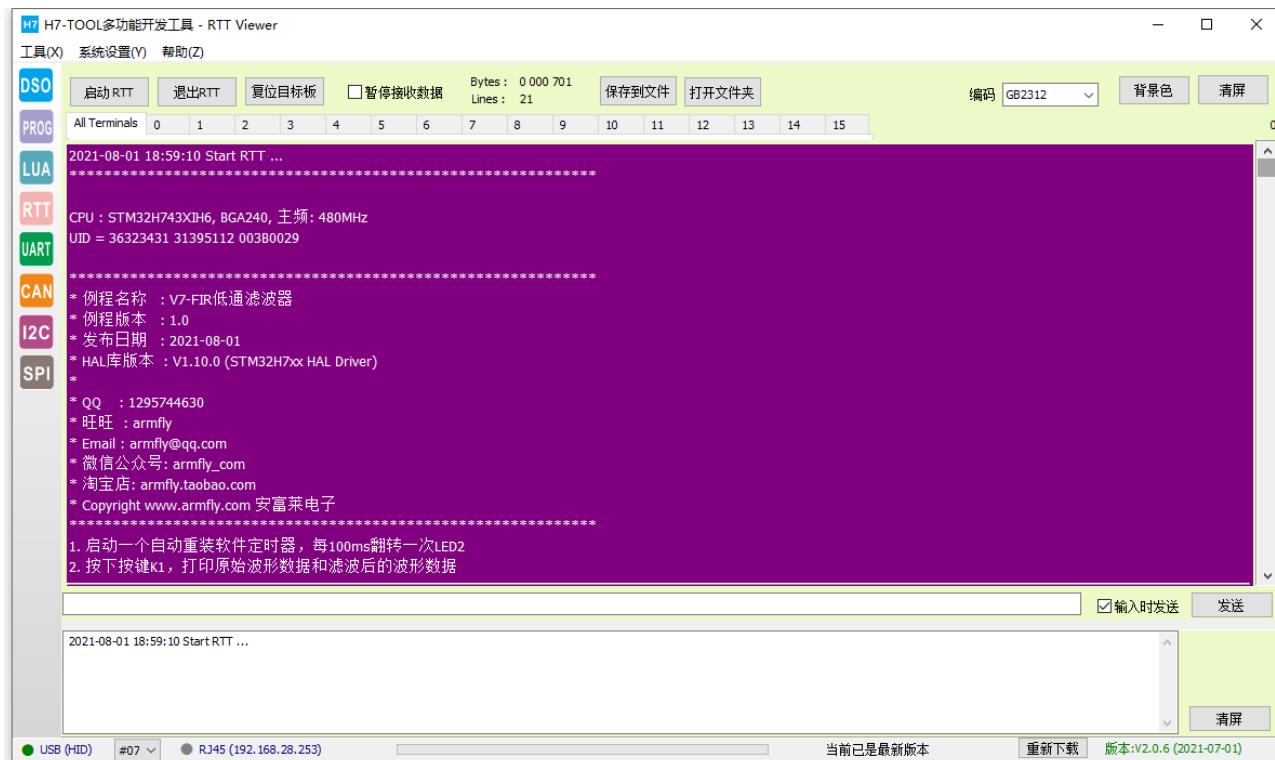
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

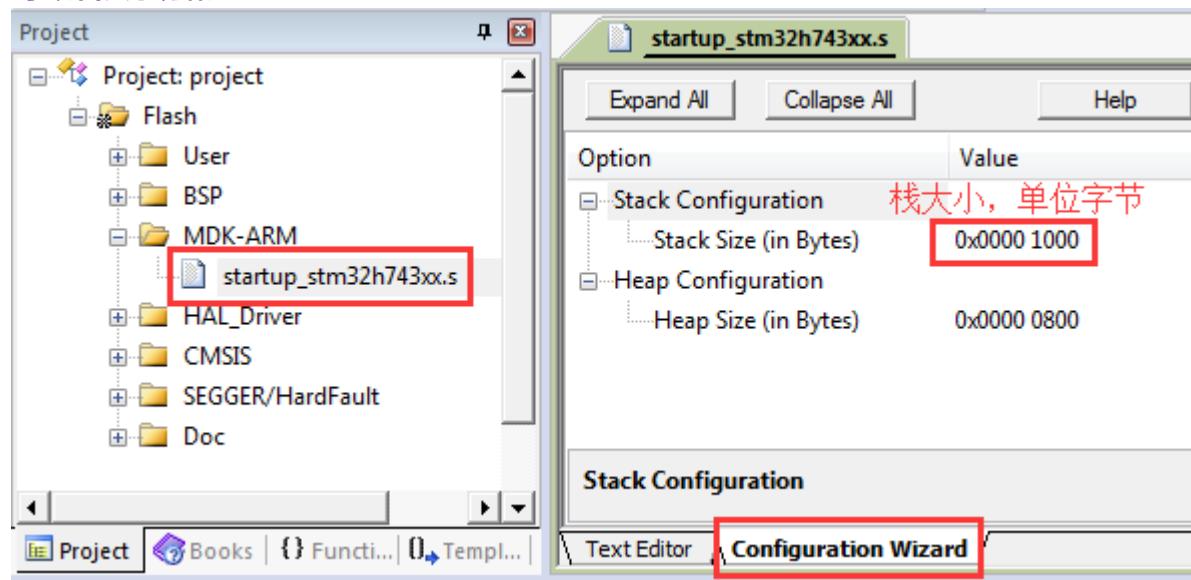


RTT 方式打印信息：

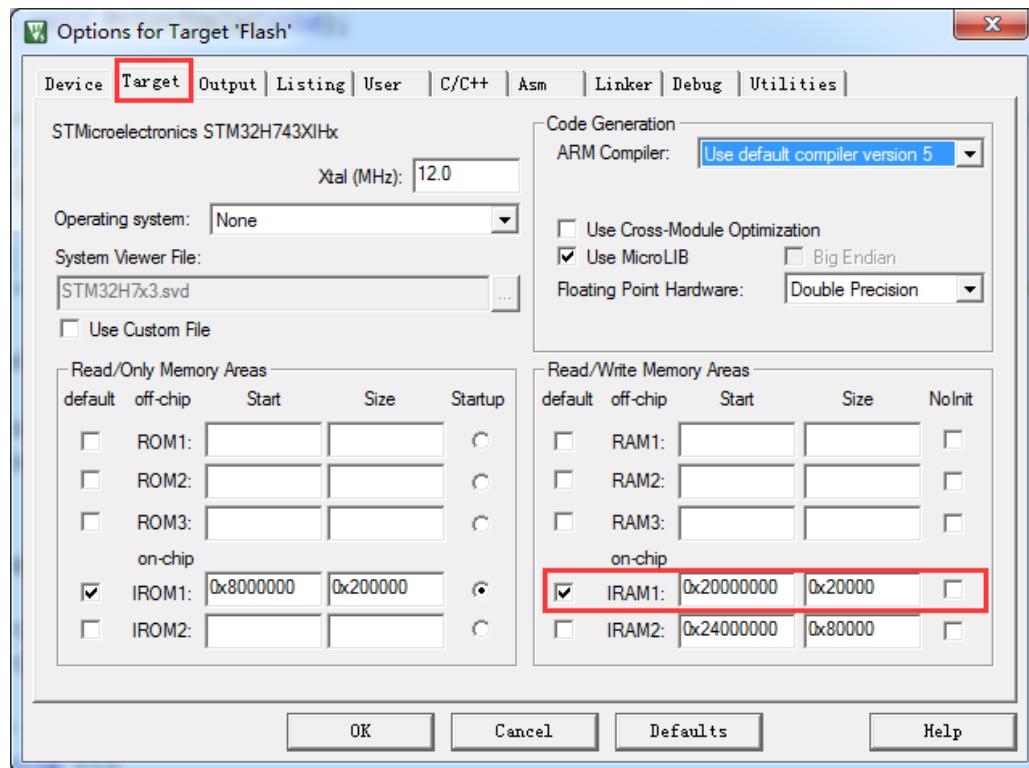


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
    testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                    arm_sin_f32(2*3.1415926f*200*i/1000);
}

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2); /* 翻转 LED 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_fir_f32_lp();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
}
```

37.7 实验例程说明 (IAR)

配套例子：

V7-225_FIR 低通滤波器设计(支持逐个数据的实时滤波)

实验目的：

1. 学习 FIR 低通滤波器的实现，支持实时滤波

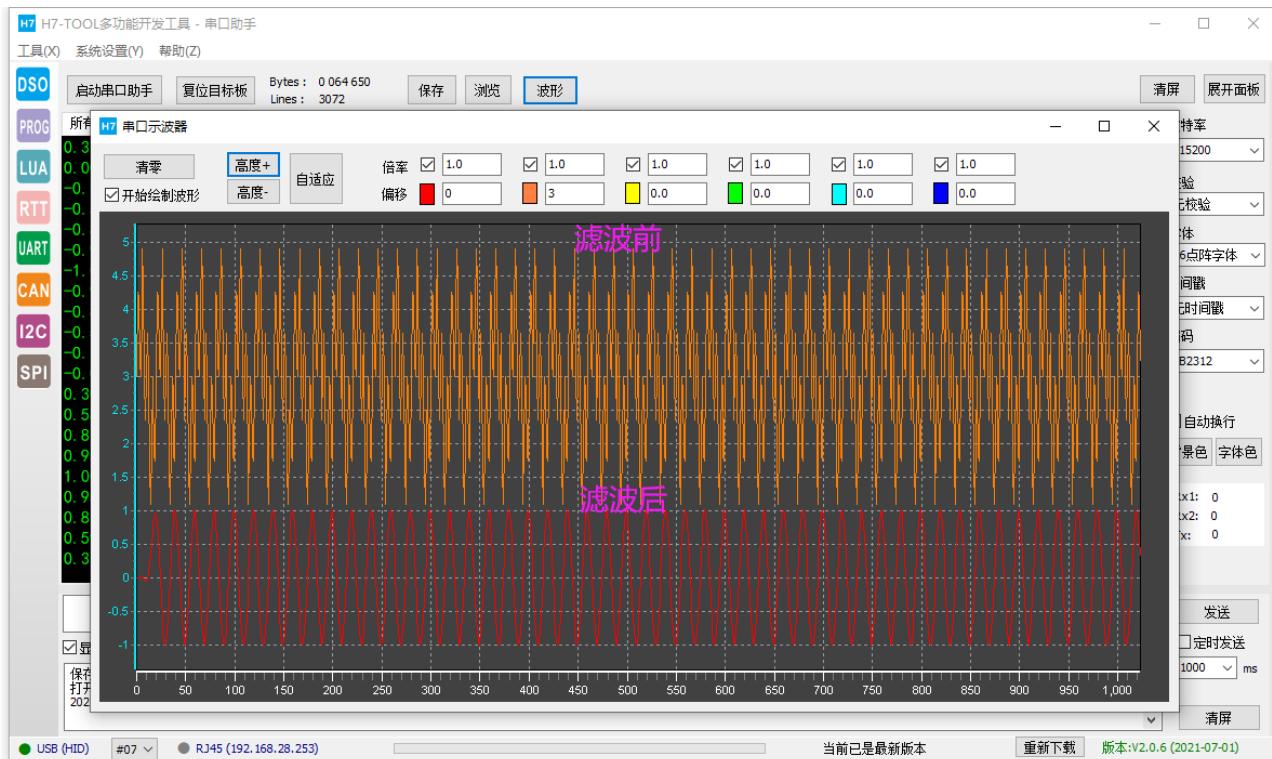
实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

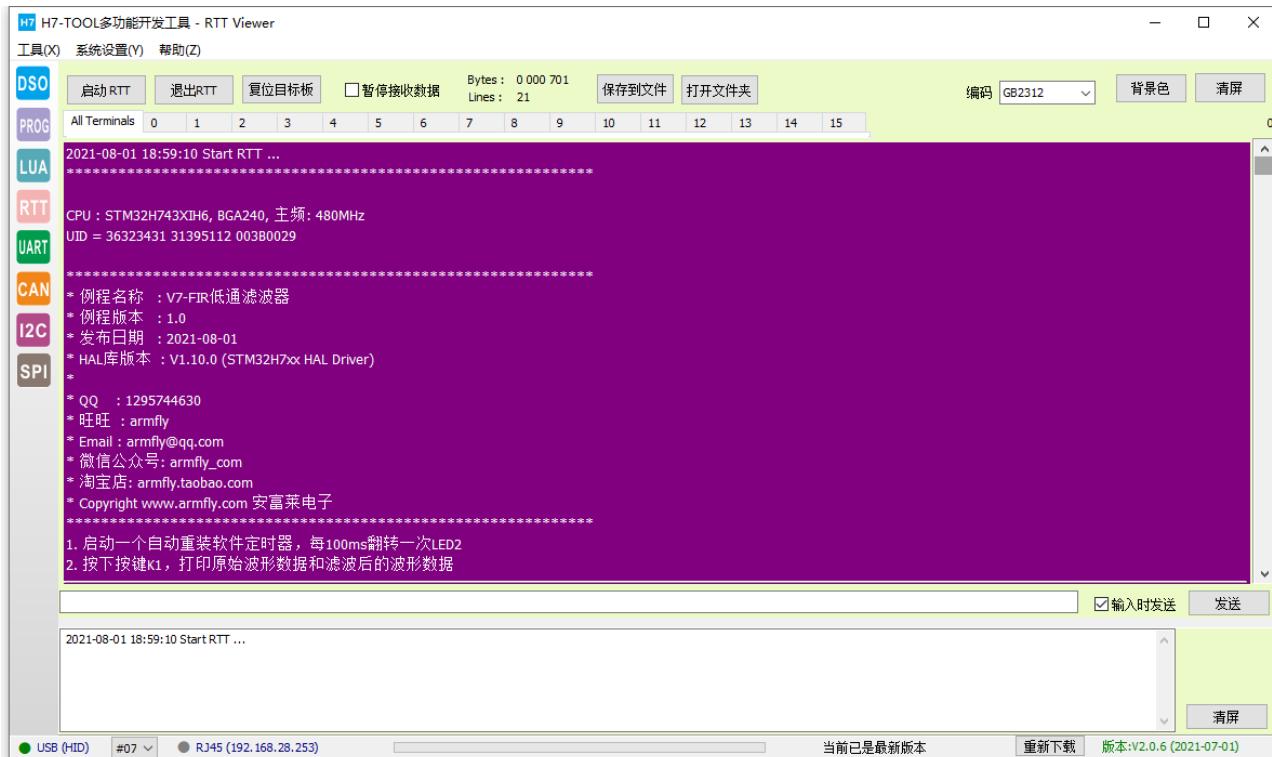
上电后串口打印的信息：



波特率 115200，数据位 8，奇偶校验位无，停止位 1。

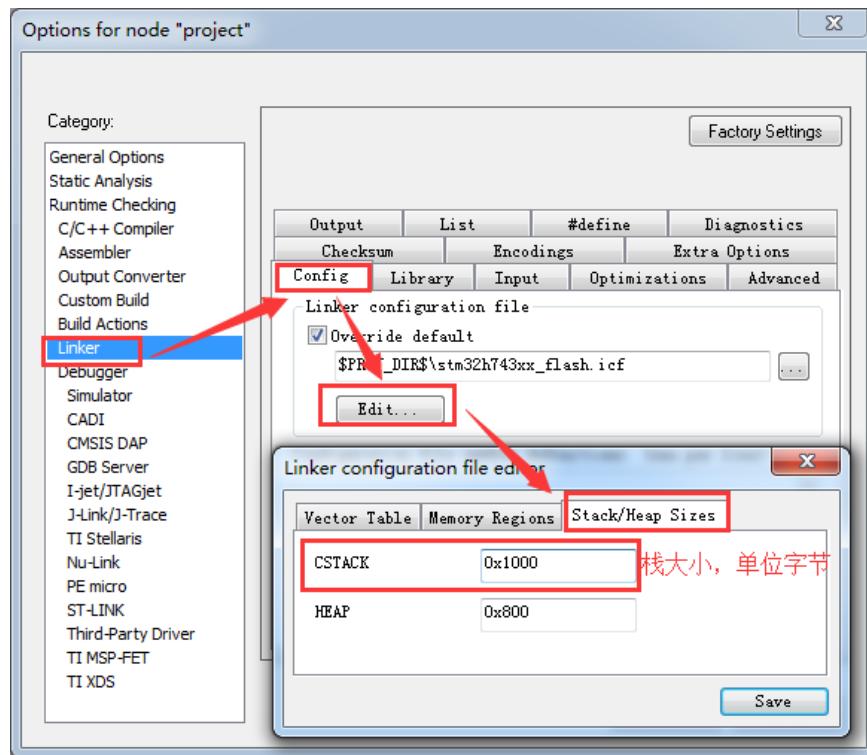


RTT 方式打印信息：

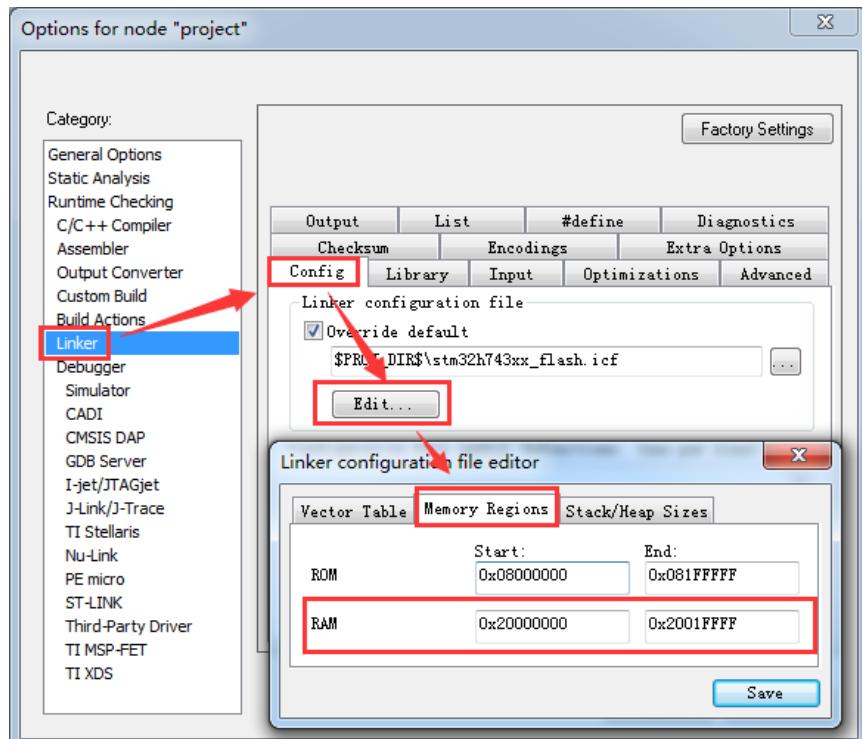


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
*****
```



```
/* 功能说明：初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形    参：无
* 返回 值：无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
        - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
        - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名：MPU_Config
* 功能说明：配置 MPU
* 形    参：无
* 返回 值：无
*****
*/
static void MPU_Config( void )
```



```
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /*使能 MPU */  
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);  
}  
  
/*  
*****  
* 函数名: CPU_CACHE_Enable  
* 功能说明: 使能 L1 Cache  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void CPU_CACHE_Enable(void)  
{  
    /* 使能 I-Cache */  
    SCB_EnableICache();  
  
    /* 使能 D-Cache */  
    SCB_EnableDCache();  
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
        testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                      arm_sin_f32(2*3.1415926f*200*i/1000);
    }

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2); /* 翻转 LED 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_fir_f32_lp();
                    break;
            }
        }
    }
}
```



```
default:  
    /* 其它的键值不处理 */  
    break;  
}  
}  
}
```

37.8 总结

本章节主要讲解了 FIR 滤波器的低通实现，同时一定要注意线性相位 FIR 滤波器的群延迟问题，详见本教程的第 41 章。



第38章 STM32H7 的 FIR 高通滤波器实现（支持逐个数据的实时滤波）

本章节讲解 FIR 高通滤波器实现。

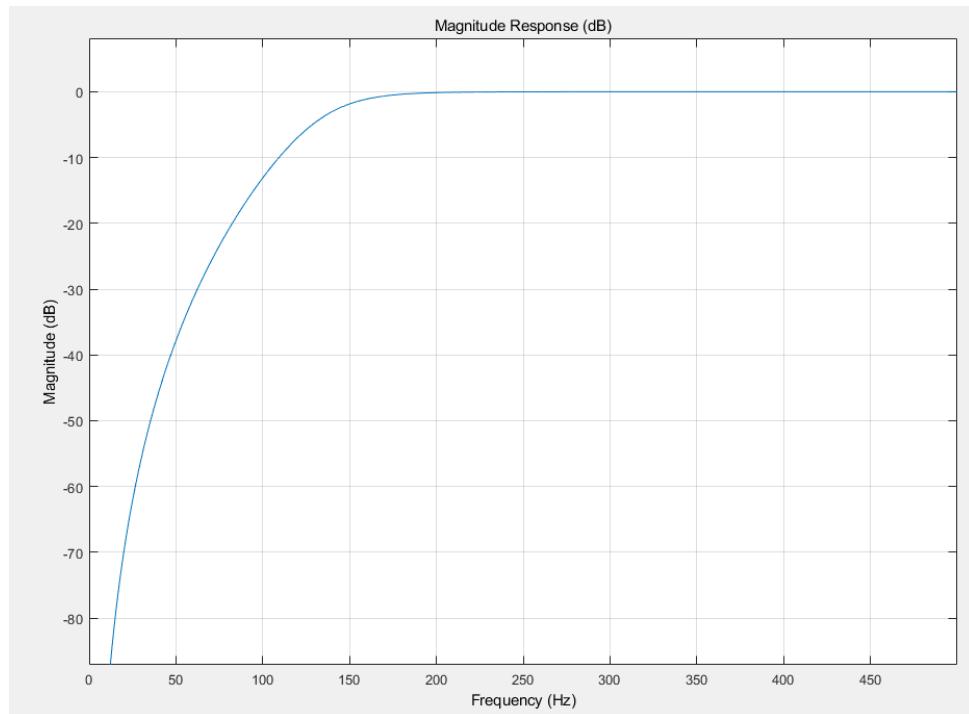
- 38.1 初学者重要提示
- 38.2 高通滤波器介绍
- 38.3 FIR 滤波器介绍
- 38.4 Matlab 工具箱 filterDesigner 生成高通滤波器 C 头文件
- 38.5 FIR 高通滤波器设计
- 38.6 实验例程说明 (MDK)
- 38.7 实验例程说明 (IAR)
- 38.8 总结

38.1 初学者重要提示

- ◆ 本章节提供的高通滤波器支持实时滤波，每次可以滤波一个数据，也可以多个数据，不限制大小。但要注意以下两点：
 - 所有数据是在同一个采样率下依次采集的数据。
 - 每次过滤数据个数一旦固定下来，运行中不可再修改。
- ◆ FIR 滤波器的群延迟是一个重要的知识点，详情在本教程第 41 章有详细说明。

38.2 高通滤波器介绍

允许高频信号通过，而减弱低于截止频率的信号通过。比如混合信号含有 50Hz + 200Hz 信号，我们可通过高通滤波器，过滤掉 50Hz 信号，让 200Hz 信号通过。



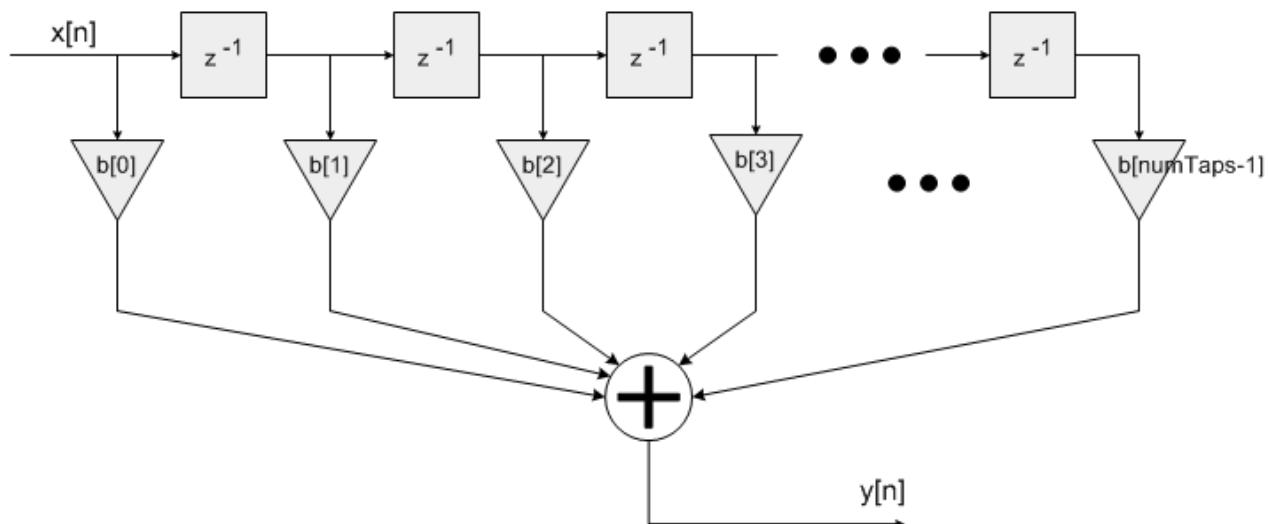
38.3 FIR 滤波器介绍

ARM 官方提供的 FIR 库支持 Q7, Q15, Q31 和浮点四种数据类型。其中 Q15 和 Q31 提供了快速算法版本。

FIR 滤波器的基本算法是一种乘法-累加 (MAC) 运行，输出表达式如下：

$$y[n] = b[0] * x[n] + b[1] * x[n-1] + b[2] * x[n-2] + \dots + b[\text{numTaps}-1] * x[n-\text{numTaps}+1]$$

结构图如下：



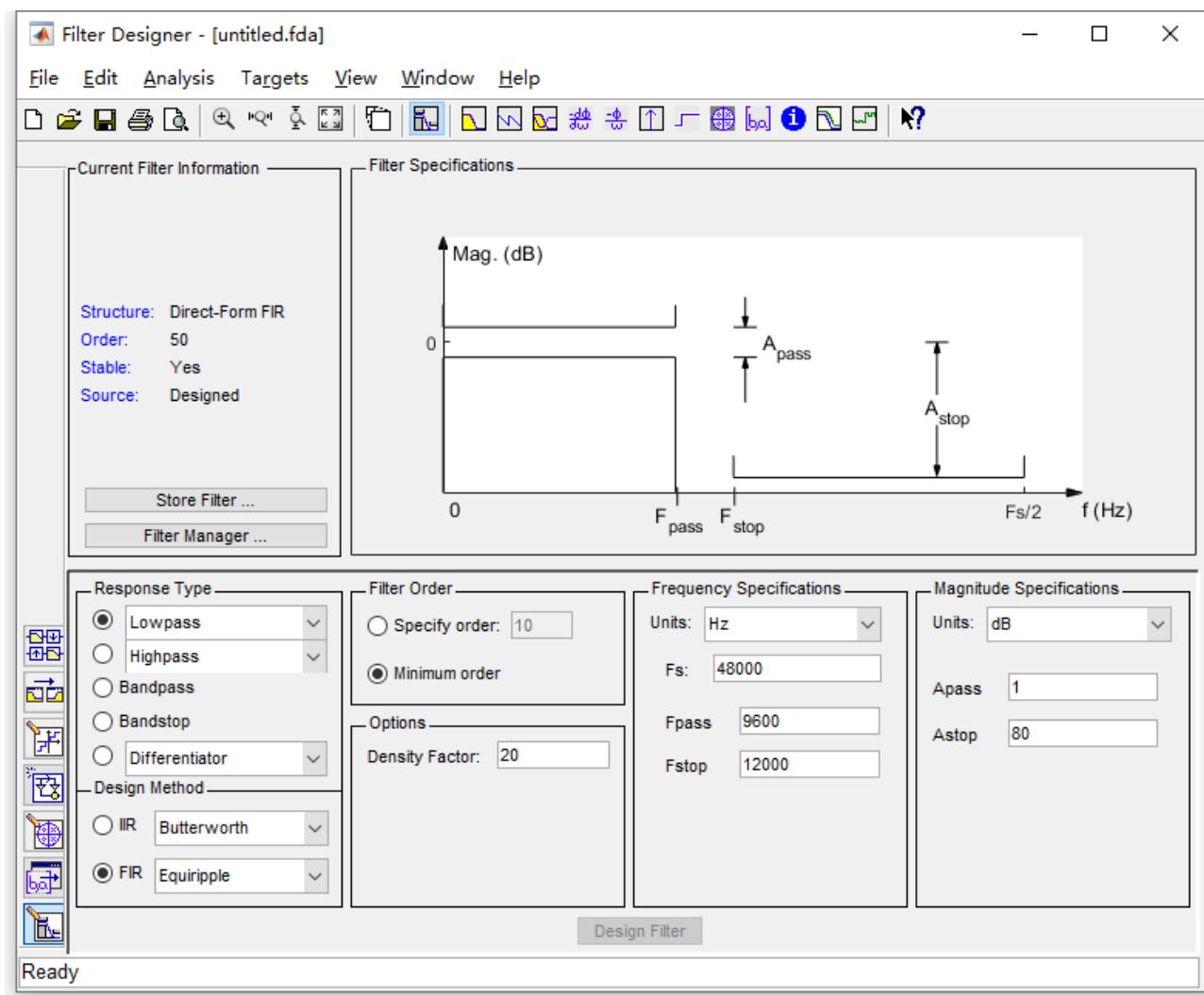
这种网络结构就是在 35.2.1 小节所讲的直接型结构。

38.4 Matlab 工具箱 filterDesigner 生成高通滤波器 C 头文件

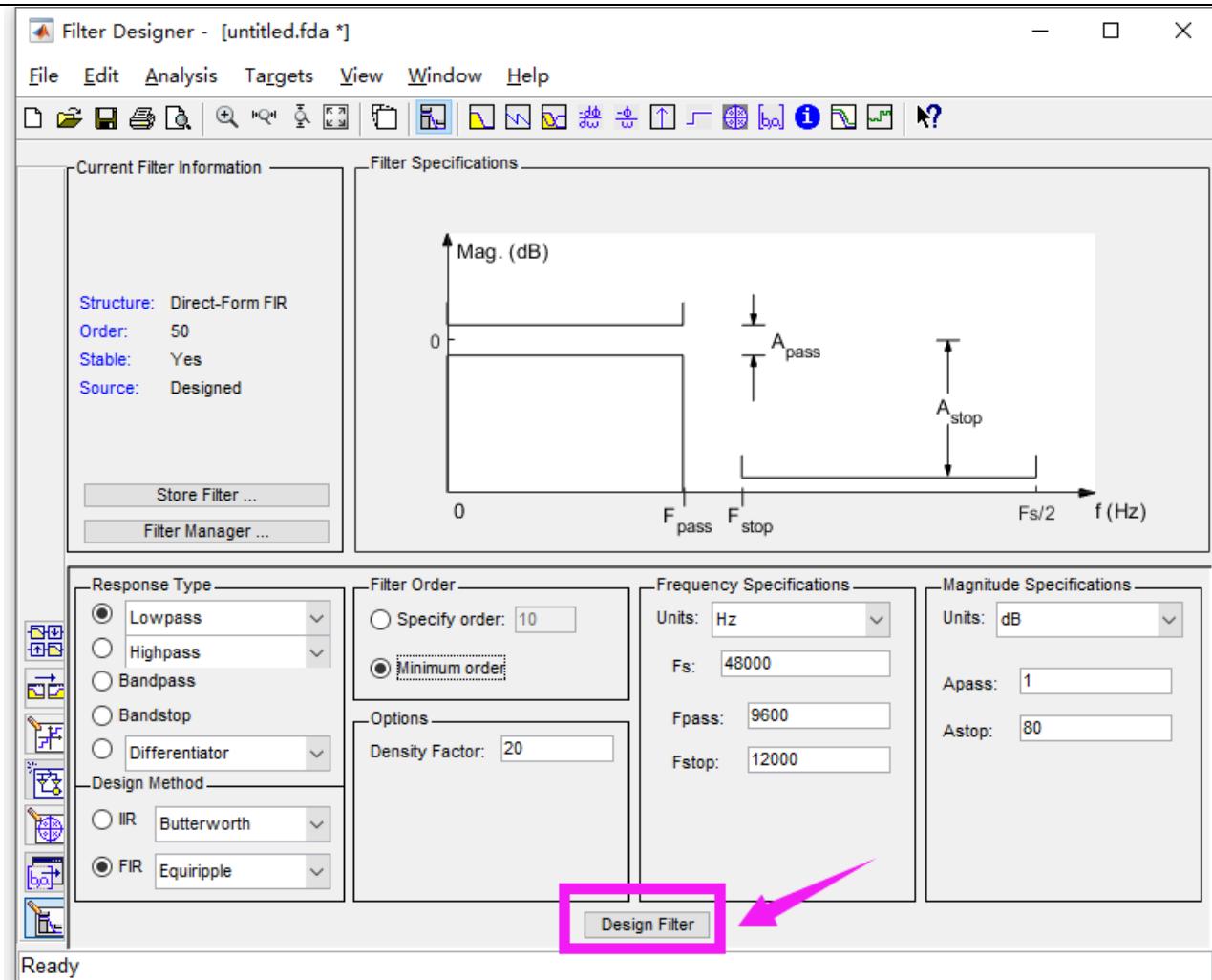
下面我们讲解下如何通过 filterDesigner 工具生成 C 头文件，也就是生成滤波器系数。首先在 matlab 的命窗口输入 filterDesigner 就能打开这个工具箱：



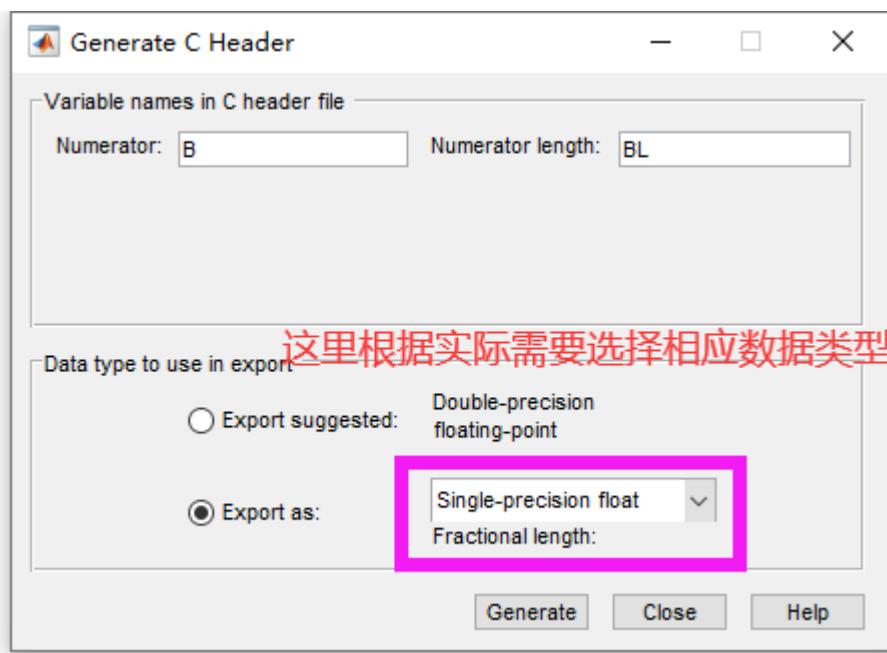
filterDesigner 界面打开效果如下：



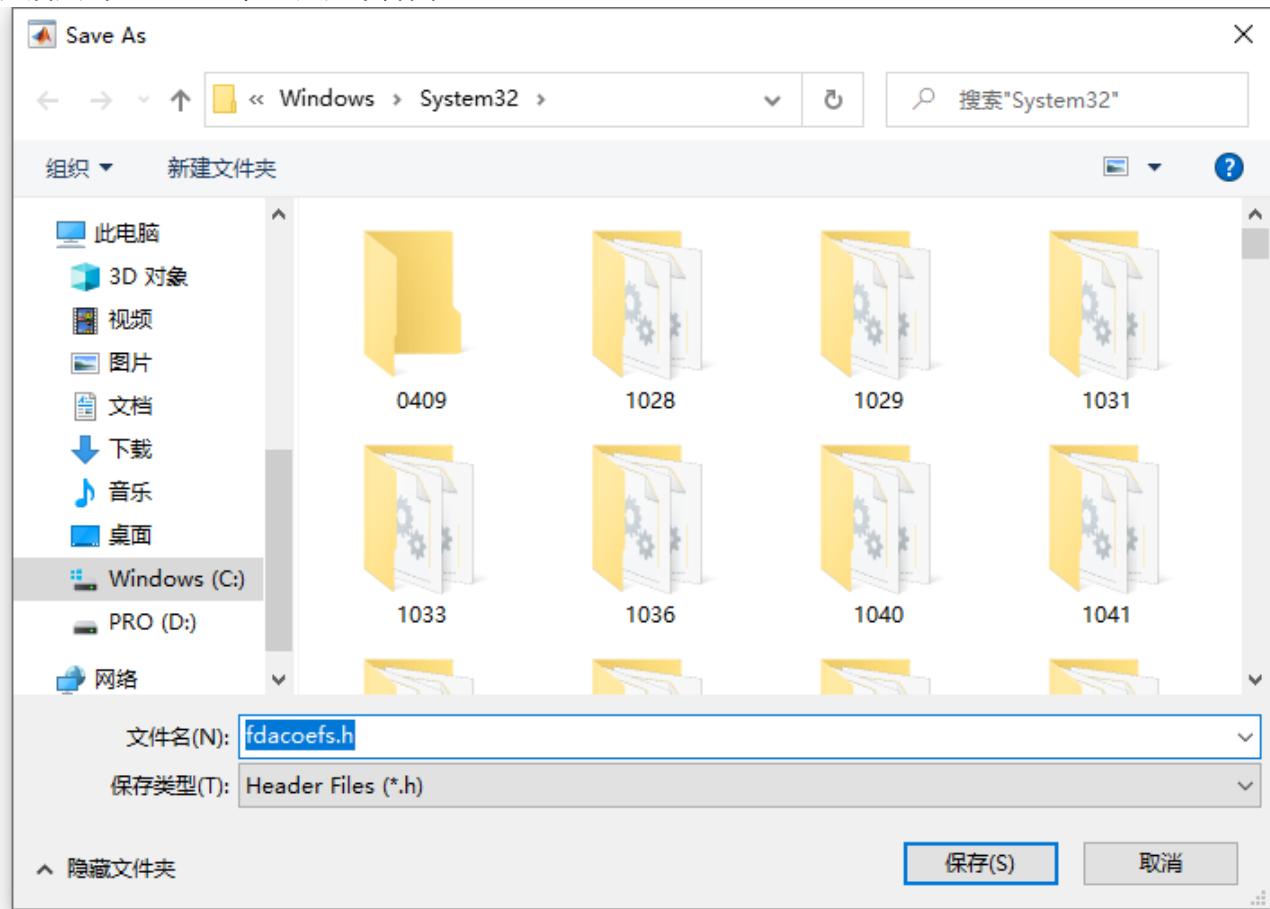
FIR 滤波器的低通，高通，带通，带阻滤波的设置会在后面逐个讲解，这里重点介绍设置后相应参数后如何生成滤波器系数。参数设置好以后点击如下按钮：



点击 Design Filter 按钮以后就生成了所需的滤波器系数，生成滤波器系数以后点击 filterDesigner 界面上的菜单 Targets->Generate C header ,打开后显示如下界面：



然后点击 Generate，生成如下界面：



再点击保存，并打开 fdatool.h 文件，可以看到生成的系数：

```
/*
 * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
 * Generated by MATLAB(R) 9.4 and Signal Processing Toolbox 8.0.
 * Generated on: 20-Jul-2021 12:19:30
 */

/*
 * Discrete-Time FIR Filter (real)
 * -----
 * Filter Structure : Direct-Form FIR
 * Filter Length   : 51
 * Stable          : Yes
 * Linear Phase    : Yes (Type 1)
 */

/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"
/*
 * Expected path to tmwtypes.h
 * D:\Program Files\MATLAB\R2018a\extern\include\tmwtypes.h
 */
/*
 * Warning - Filter coefficients were truncated to fit specified data type.
 * The resulting response may not match generated theoretical response.
 * Use the Filter Design & Analysis Tool to design accurate
 * single-precision filter coefficients.
 */
const int BL = 51;
const real32_T B[51] = {
    -0.0009190982091, -0.00271769613, -0.002486952813, 0.003661438357, 0.0136509249,
    0.01735116541, 0.00766530633, -0.006554719061, -0.007696784101, 0.006105459295,
```



```
0.01387391612, 0.0003508617228, -0.01690892503, -0.008905642666, 0.01744112931,  
0.02074504457, -0.0122964941, -0.03424086422, -0.001034529647, 0.04779030383,  
0.02736303769, -0.05937951803, -0.08230702579, 0.06718690693, 0.3100151718,  
0.4300478697, 0.3100151718, 0.06718690693, -0.08230702579, -0.05937951803,  
0.02736303769, 0.04779030383, -0.001034529647, -0.03424086422, -0.0122964941,  
0.02074504457, 0.01744112931, -0.008905642666, -0.01690892503, 0.0003508617228,  
0.01387391612, 0.006105459295, -0.007696784101, -0.006554719061, 0.00766530633,  
0.01735116541, 0.0136509249, 0.003661438357, -0.002486952813, -0.00271769613,  
-0.0009190982091  
};
```

上面数组 B[51]中的数据就是滤波器系数。下面小节讲解如何使用 filterDesigner 配置 FIR 低通，高通，带通和带阻滤波。关于 Filter Designer 的其它用法，大家可以在 matlab 命令窗口中输入 help filterDesigner 打开帮助文档进行学习。

```
命令行窗口  
>> help filterDesigner  
--- signal/filterDesigner 的帮助 ---  
  
filterDesigner - Design filters starting with algorithm selection  
  
The Filter Designer app enables you to design and analyze digital filters.  
  
另请参阅 Signal Analyzer, Window Designer, designfilt, fvtool, wvttool  
signal/filterDesigner 的参考页  
fx >> |
```

38.5 FIR 高通滤波器设计

本章使用的 FIR 滤波器函数是 arm_fir_f32。使用此函数可以设计 FIR 低通，高通，带通和带阻滤波器。

38.5.1 函数 arm_fir_init_f32

函数原型：

```
void arm_fir_init_f32(  
    arm_fir_instance_f32 * S,  
    uint16_t numTaps,  
    const float32_t * pCoeffs,  
    float32_t * pState,  
    uint32_t blockSize);
```

函数描述：

这个函数用于 FIR 初始化。

函数参数：

- ◆ 第 1 个参数是 arm_fir_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是滤波器系数的个数。
- ◆ 第 3 个参数是滤波器系数地址。
- ◆ 第 4 个参数是缓冲状态地址。



- ◆ 第 5 个参数是每次处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

注意事项：

结构体 arm_fir_instance_f32 的定义如下（在文件 arm_math.h 文件）：

```
typedef struct
{
    uint16_t numTaps;      /**< number of filter coefficients in the filter. */
    float32_t *pState;     /**< points to the state variable array. The array is of length */
                           numTaps+blockSize-1;
    float32_t *pCoeffs;    /**< points to the coefficient array. The array is of length numTaps. */
} arm_fir_instance_f32;
```

1. 参数 pCoeffs 指向滤波因数，滤波因数数组长度为 numTaps。但要注意 pCoeffs 指向的滤波因数应该按照如下的逆序进行排列：
 $\{b[numTaps-1], b[numTaps-2], b[N-2], \dots, b[1], b[0]\}$
但满足线性相位特性的 FIR 滤波器具有奇对称或者偶对称的系数，偶对称时逆序排列还是他本身。
2. pState 指向状态变量数组，这个数组用于函数内部计算数据的缓存。
3. blockSize 这个参数的大小没有特殊要求，最小可以每次处理 1 个数据，最大可以每次全部处理完。

38.5.2 函数 arm_fir_f32

函数原型：

```
void arm_fir_f32(
    const arm_fir_instance_f32 * S,
    const float32_t * pSrc,
    float32_t * pDst,
    uint32_t blockSize)
```

函数描述：

这个函数用于 FIR 滤波。

函数参数：

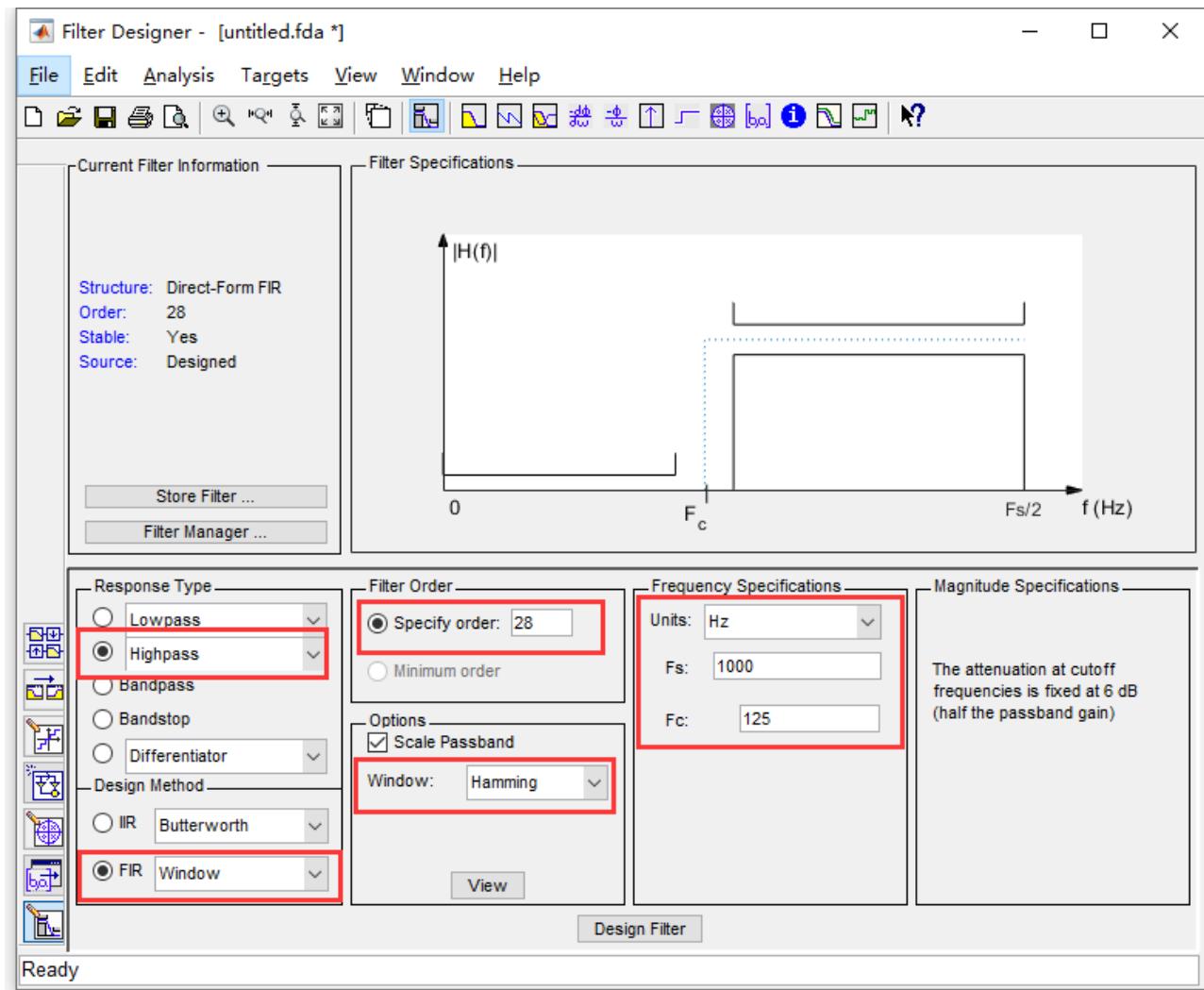
- ◆ 第 1 个参数是 arm_fir_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是源数据地址。
- ◆ 第 3 个参数是滤波后的数据地址。

第 4 个参数是每次调用处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

38.5.3 filterDesigner 获取高通滤波器系数

设计一个如下的例子：

信号由 50Hz 正弦波和 200Hz 正弦波组成，采样率 1Kbps，现设计一个高通滤波器，截止频率 125Hz，采样 1024 个数据，采用函数 fir1 进行设计（注意这个函数是基于窗口的方法设计 FIR 滤波，默认是 hamming 窗），滤波器阶数设置为 28。filterDesigner 的配置如下：



配置好高通滤波器后，具体滤波器系数的生成大家参考本章第4小节的方法即可。

38.5.4 高通滤波器实现

通过工具箱filterDesigner获得高通滤波器系数后在开发板上运行函数arm_fir_f32 来测试高通滤波器的效果。

```
#define TEST_LENGTH_SAMPLES 1024 /* 采样点数 */
#define BLOCK_SIZE 1 /* 调用一次arm_fir_f32处理的采样点个数 */
#define NUM_TAPS 29 /* 滤波器系数个数 */

uint32_t blockSize = BLOCK_SIZE;
uint32_t numBlocks = TEST_LENGTH_SAMPLES/BLOCK_SIZE; /* 需要调用arm_fir_f32的次数 */

static float32_t testInput_f32_50Hz_200Hz[TEST_LENGTH_SAMPLES]; /* 采样点 */
static float32_t testOutput[TEST_LENGTH_SAMPLES]; /* 滤波后的输出 */
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1]; /* 状态缓存，大小numTaps + blockSize - 1 */

/* 高通滤波器系数 通过fadtool获取*/
const float32_t firCoeffs32HP[NUM_TAPS] = {
    0.0018157335f, 0.001582013792f, -6.107207639e-18f, -0.003683975432f, -0.008045346476f,
    -0.008498443291f, -1.277260999e-17f, 0.01733288541f, 0.03401865438f, 0.0332348831f,
    -4.021742543e-17f, -0.06737889349f, -0.1516391635f, -0.2220942229f, 0.7486887574f,
```



```
-0.2220942229f, -0.1516391635f, -0.06737889349f, -4.021742543e-17f, 0.0332348831f,
0.03401865438f, 0.01733288541f, -1.277260999e-17f, -0.008498443291f, -0.008045346476f,
-0.003683975432f, -6.107207639e-18f, 0.001582013792f, 0.0018157335f
};
```

```
/*
*****
* 函数名: arm_fir_f32_hp
* 功能说明: 调用函数arm_fir_f32_hp实现高通滤波器
* 形参: 无
* 返回值: 无
*****
*/
static void arm_fir_f32_hp(void)
{
    uint32_t i;
    arm_fir_instance_f32 S;
    float32_t *inputF32, *outputF32;

    /* 初始化输入输出缓存指针 */
    inputF32 = &testInput_f32_50Hz_200Hz[0];
    outputF32 = &testOutput[0];

    /* 初始化结构体S */
    arm_fir_init_f32(&S,
                      NUM_TAPS,
                      (float32_t *)&firCoeffs32HP[0],
                      &firStateF32[0],
                      blockSize);

    /* 实现FIR滤波, 这里每次处理1个点 */
    for(i=0; i < numBlocks; i++)
    {
        arm_fir_f32(&S, inputF32 + (i * blockSize), outputF32 + (i * blockSize), blockSize);
    }

    /* 打印滤波后结果 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        printf("%f, %f\r\n", testOutput[i], inputF32[i]);
    }
}
```

运行如上函数可以通过串口打印出函数arm_fir_f32滤波后的波形数据，下面通过Matlab绘制波形来对比Matlab计算的结果和ARM官方库计算的结果。

对比前需要先将串口打印出的一组数据加载到 Matlab 中，arm_fir_f32 的计算结果起名为 sampledata，加载方法在前面的教程中已经讲解过，这里不做赘述了。Matlab 中运行的代码如下：

```
%%%%%
% FIR低通滤波器设计
%%%%%
fs=1000;          %设置采样频率 1K
N=320;            %采样点数
n=0:N-1;
t=n/fs;           %时间序列
f=n*fs/N;          %频率序列

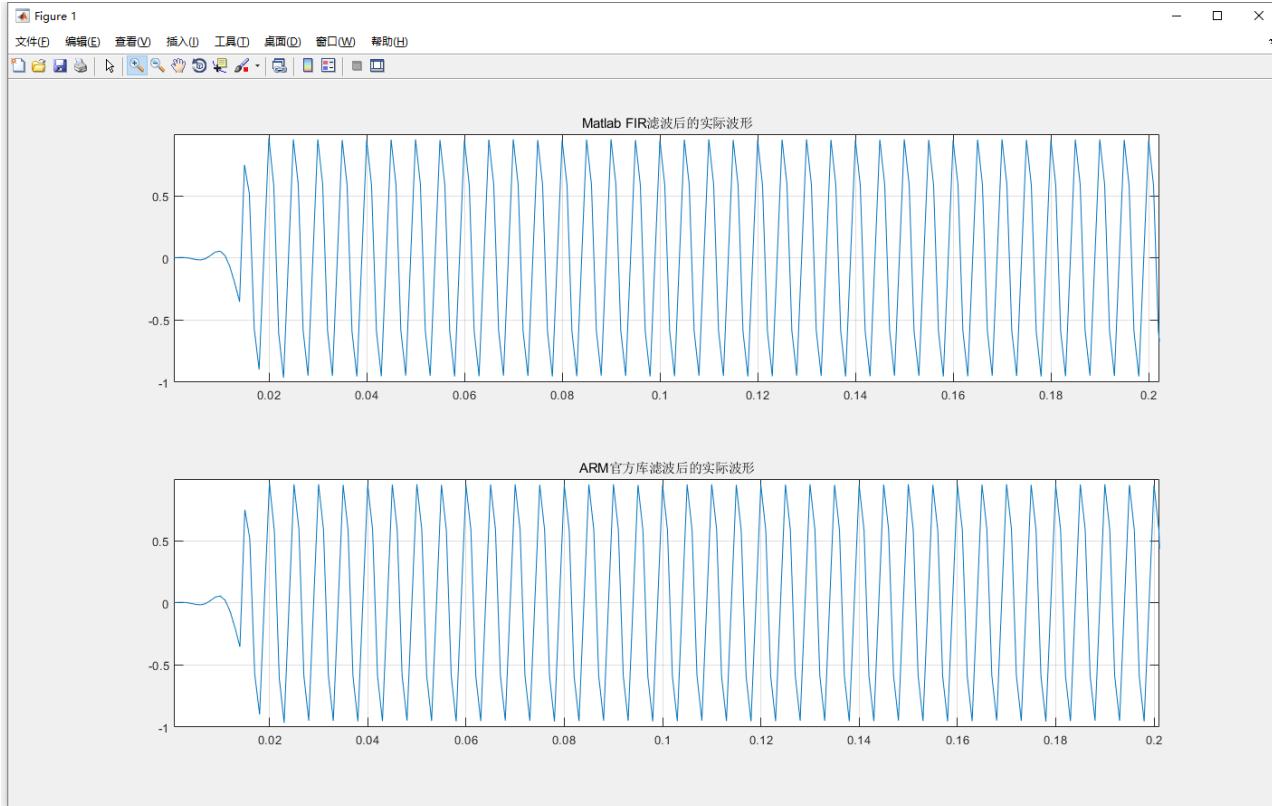
x=sin(2*pi*50*t)+sin(2*pi*200*t);      %50Hz和200Hz正弦波混合
b=fir1(28, 0.25);
y=filter(b, 1, x);
subplot(211);
plot(t, y);
title('Matlab FIR滤波后的波形');
```



```
grid on;

subplot(212);
plot(t, sampledata);
title('ARM官方库滤波后的波形');
grid on;
```

Matlab 运行结果如下：



从上面的波形对比来看，matlab 和函数 arm_fir_f32 计算的结果基本是一致的。为了更好的说明滤波效果，下面从频域的角度来说明这个问题，Matlab 上面运行如下代码：

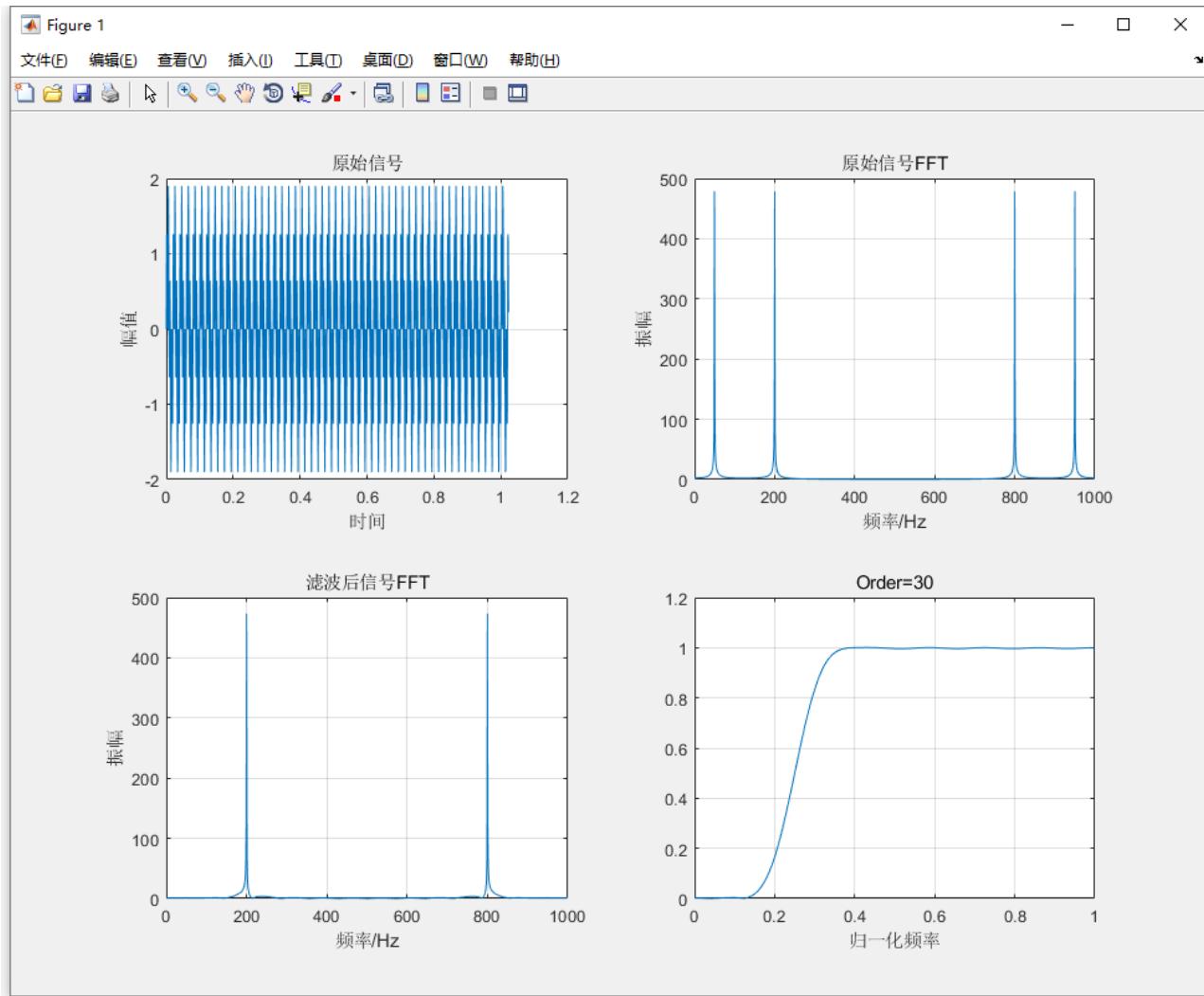
```
%*****
%          FIR高通滤波器设计
%*****
fs=1000;           %设置采样频率 1K
N=320;             %采样点数
n=0:N-1;
t=n/fs;            %时间序列
f=n*fs/N;          %频率序列

x=sin(2*pi*50*t)+sin(2*pi*200*t); %50Hz和200Hz正弦波混合
subplot(221);
plot(t, x);         %绘制信号x的波形
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;

subplot(222);
y=fft(x, N);       %对信号x做FFT
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号FFT');
grid on;
```

```
y3=fft(sampledata, N); %经过FIR滤波器后得到的信号做FFT  
subplot(223);  
plot(f, abs(y3));  
xlabel('频率/Hz');  
ylabel('振幅');  
title('滤波后信号FFT');  
grid on;  
  
b=fir1(28, 125/500, 'high'); %获得滤波器系数，截止频率125Hz，高通滤波。  
[H, F]=freqz(b, 1, 512); %通过fir1设计的FIR系统的频率响应  
subplot(224);  
plot(F/pi, abs(H)); %绘制幅频响应  
xlabel('归一化频率');  
title(['Order=', int2str(30)]);  
grid on;
```

Matlab 显示效果如下:



上面波形变换前的 FFT 和变换后 FFT 可以看出，50Hz 的正弦波基本被滤除。

38.6 实验例程说明 (MDK)

配套例子：



V7-226_FIR 高通滤波器设计(支持逐个数据的实时滤波)

实验目的：

1. FIR 高通滤波器的实现，支持实时滤波。

实验内容：

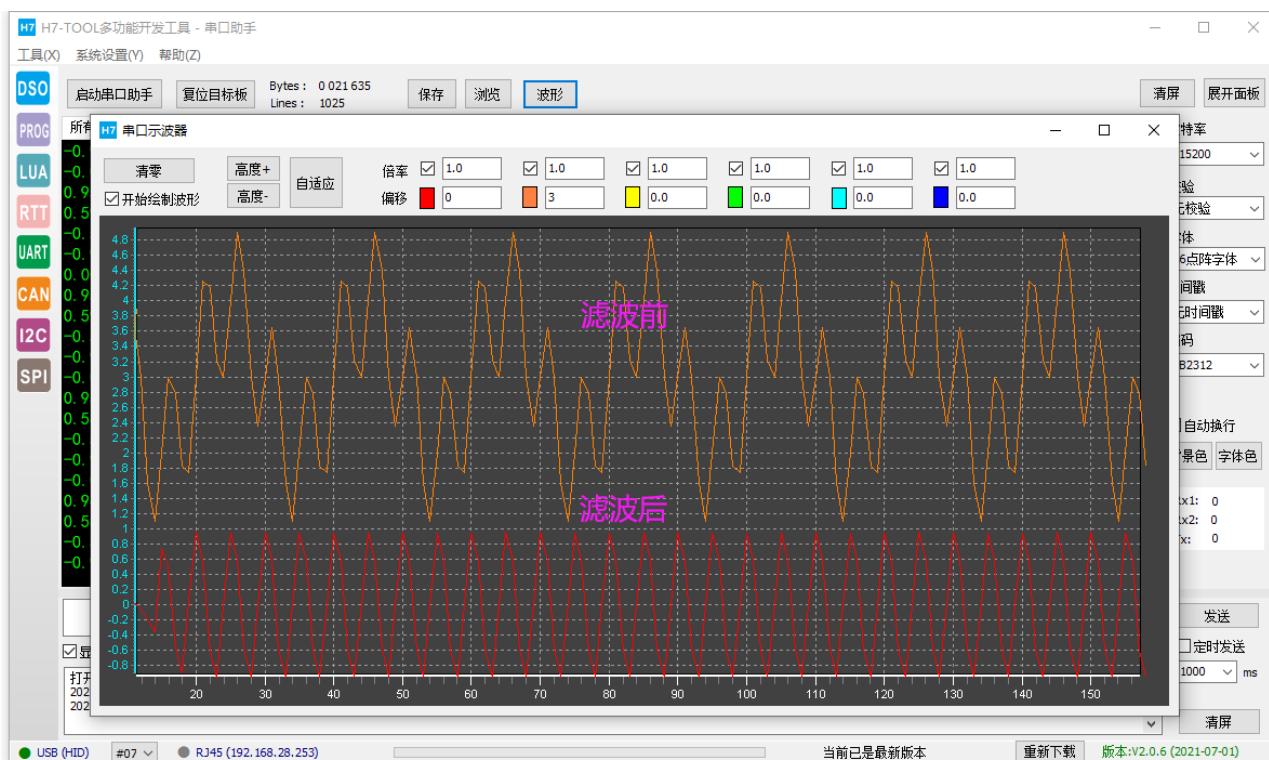
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

使用 AC6 注意事项

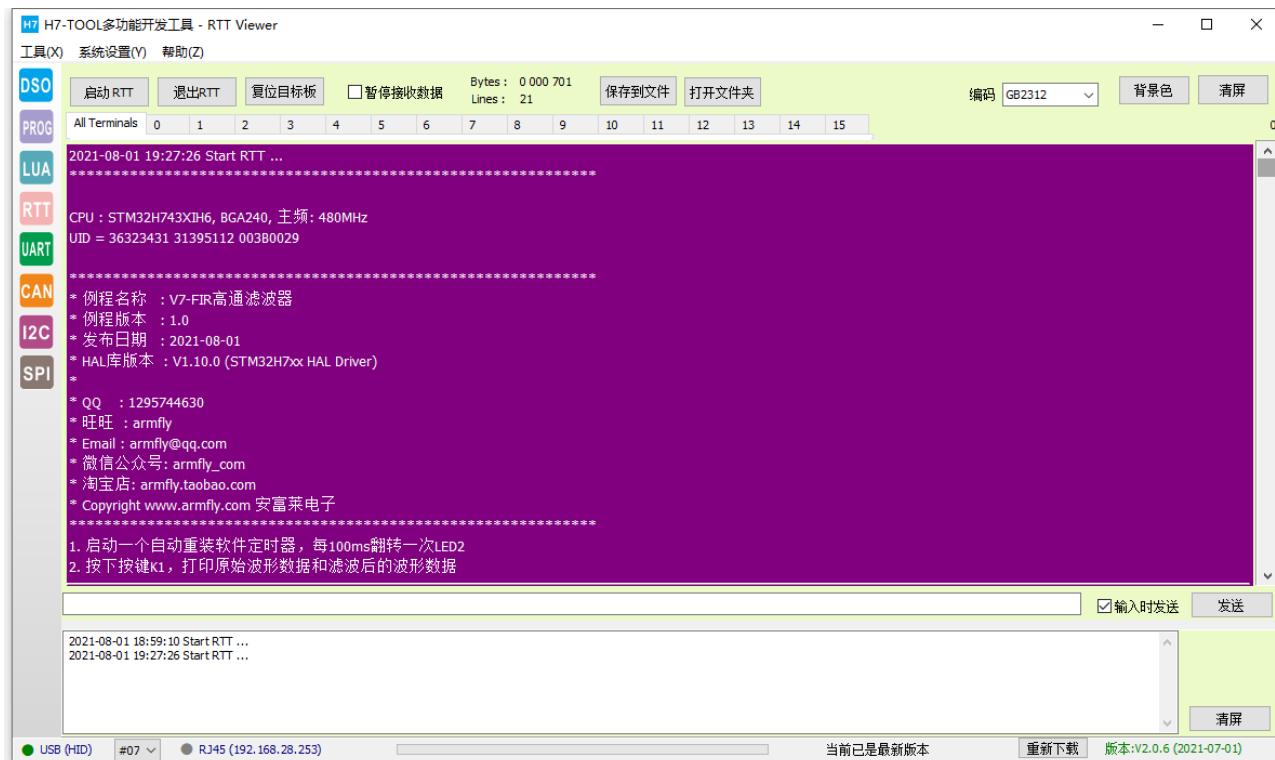
特别注意附件章节 C 的问题。

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

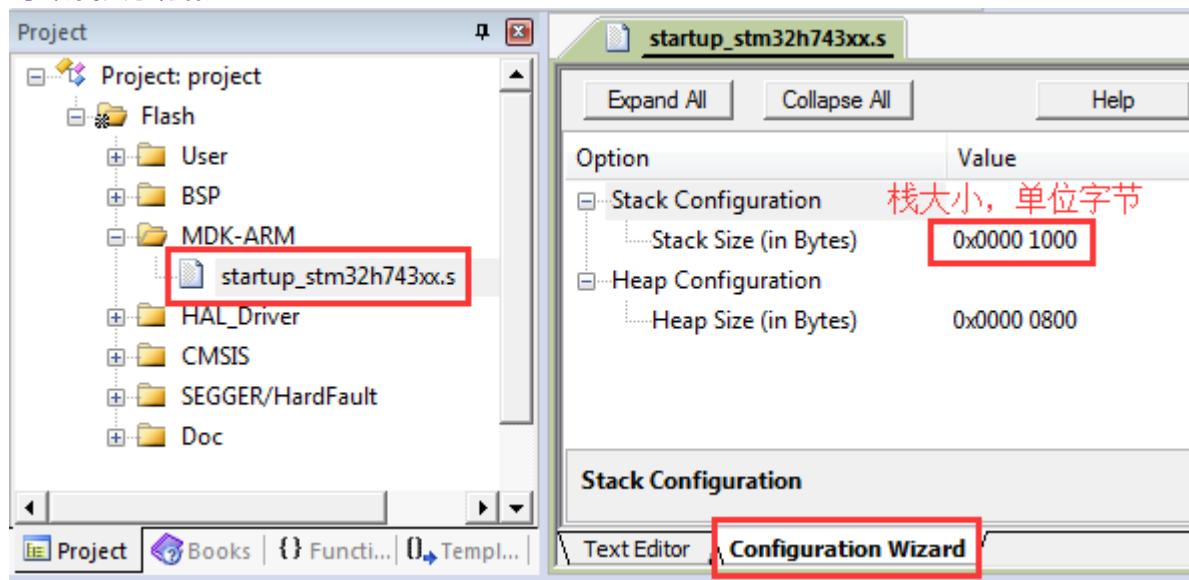


RTT 方式打印信息：

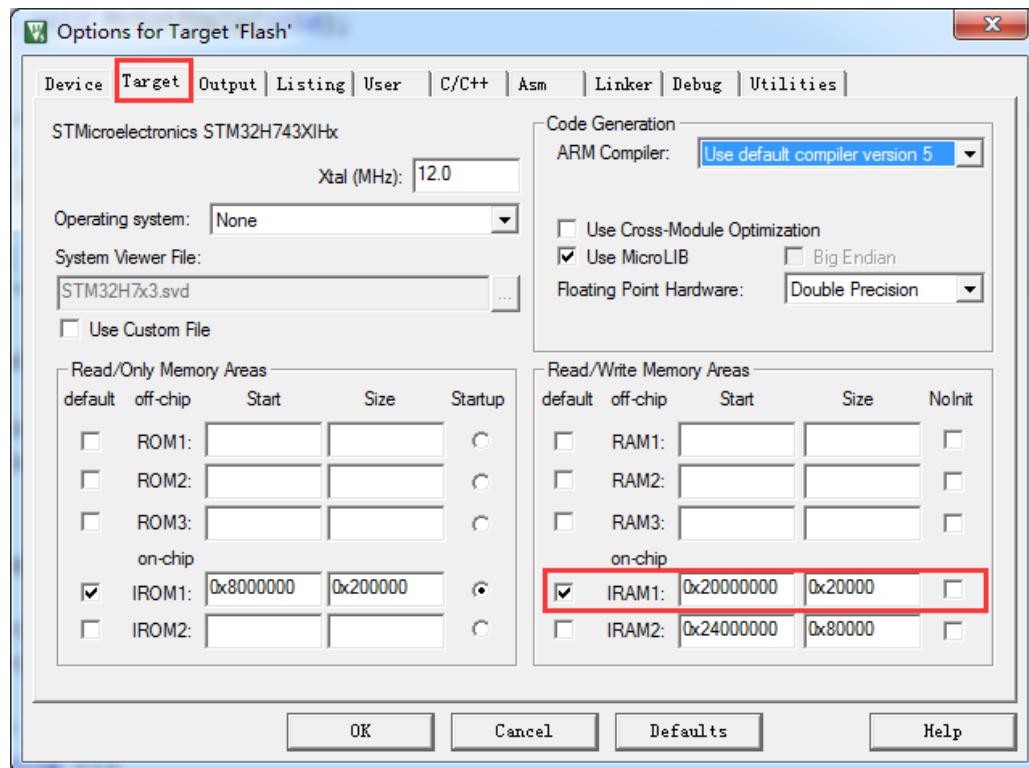


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
    testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                    arm_sin_f32(2*3.1415926f*200*i/1000);
}

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2); /* 翻转 LED2 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_fir_f32_hp();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
}
```

38.7 实验例程说明 (IAR)

配套例子：

V7-226_FIR 高通滤波器设计(支持逐个数据的实时滤波)

实验目的：

1. FIR 高通滤波器的实现，支持实时滤波。

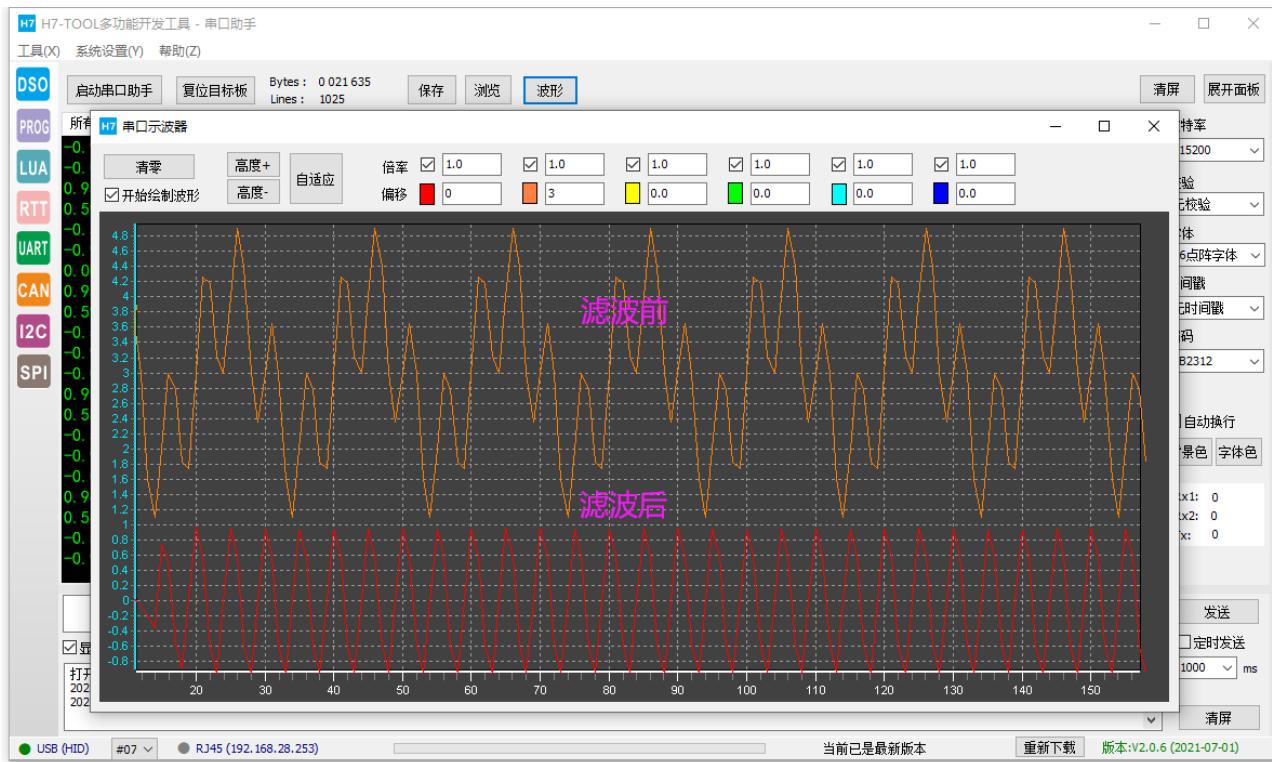
实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

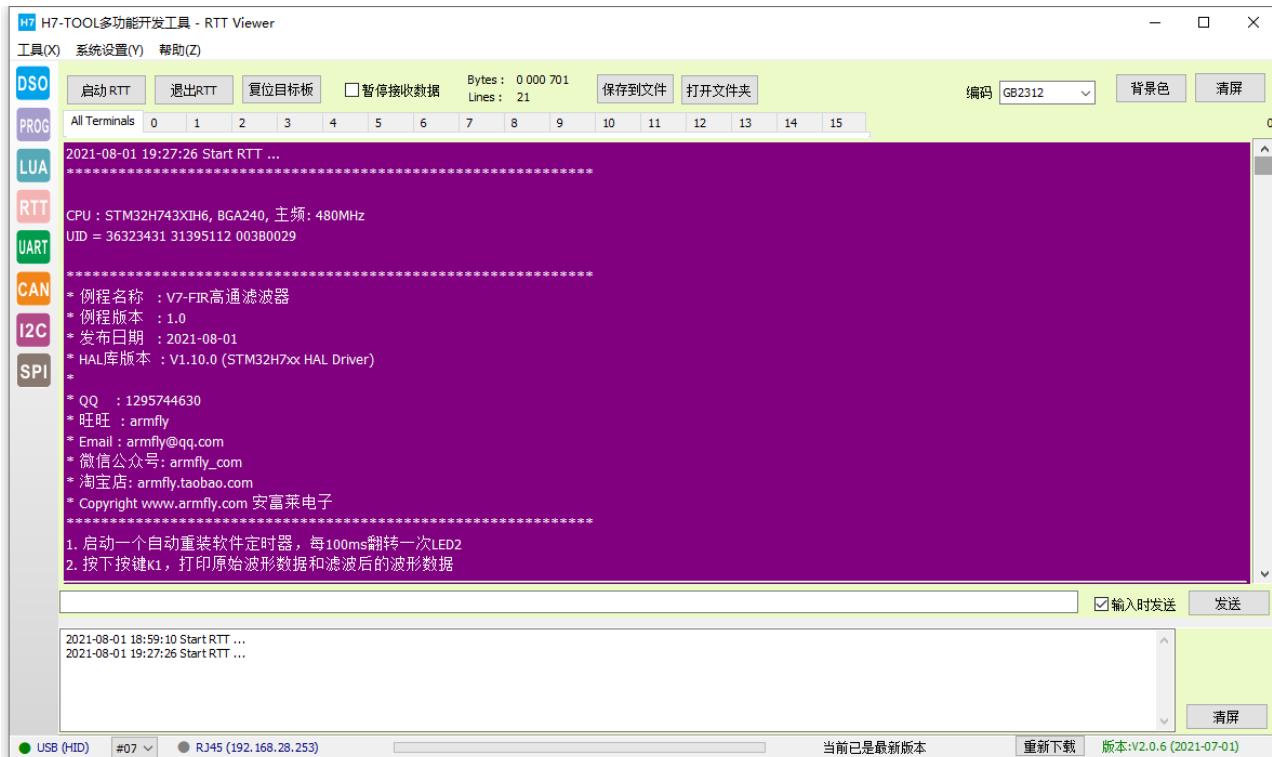
上电后串口打印的信息：



波特率 115200，数据位 8，奇偶校验位无，停止位 1。

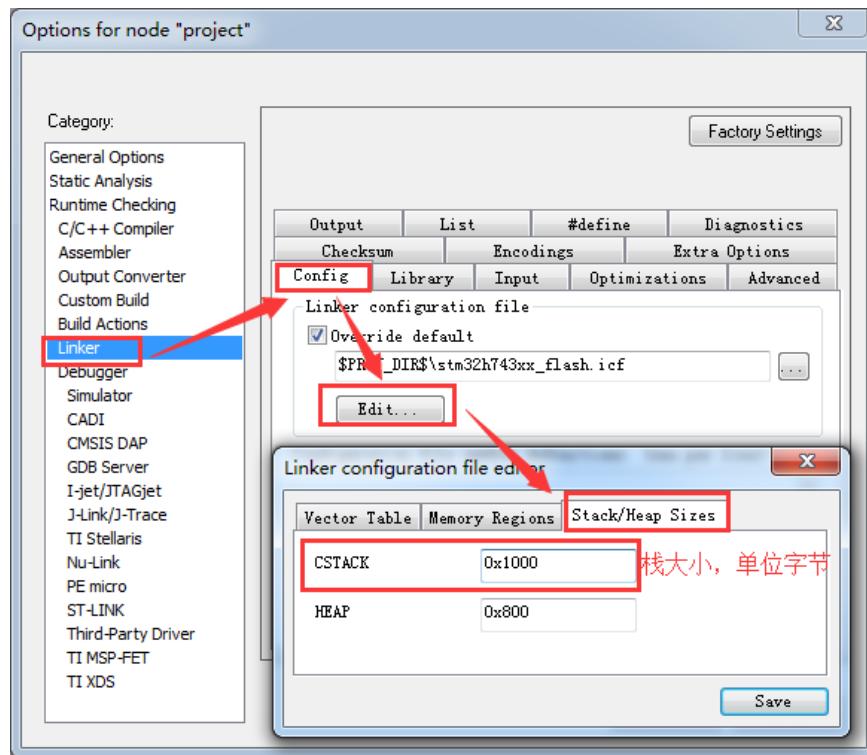


RTT 方式打印信息：

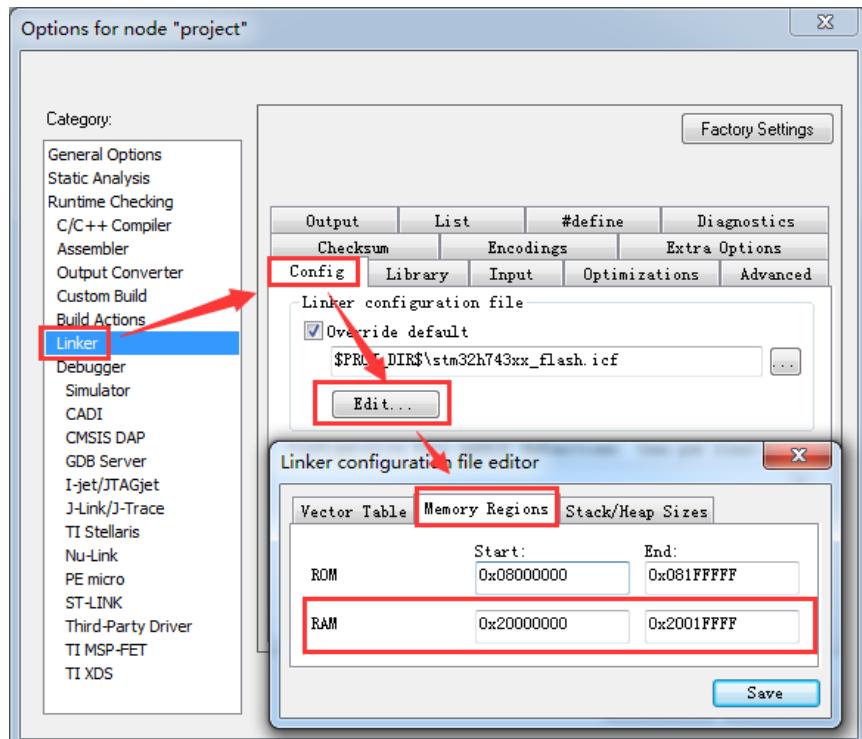


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
*****
```



```
/* 功能说明：初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形    参：无
* 返回值：无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
     STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
     - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
     - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
     配置系统时钟到 400MHz
     - 切换使用 HSE。
     - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
     Event Recorder:
     - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
     - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名：MPU_Config
* 功能说明：配置 MPU
* 形    参：无
* 返回值：无
*****
*/
static void MPU_Config( void )
```



```
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /*使能 MPU */  
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);  
}  
  
/*  
*****  
* 函数名: CPU_CACHE_Enable  
* 功能说明: 使能 L1 Cache  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void CPU_CACHE_Enable(void)  
{  
    /* 使能 I-Cache */  
    SCB_EnableICache();  
  
    /* 使能 D-Cache */  
    SCB_EnableDCache();  
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
        testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                      arm_sin_f32(2*3.1415926f*200*i/1000);
    }

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2); /* 翻转 LED2 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_fir_f32_hp();
                    break;
            }
        }
    }
}
```



```
default:  
    /* 其它的键值不处理 */  
    break;  
}  
}  
}
```

38.8 总结

本章节主要讲解了 FIR 滤波器的高通实现，同时一定要注意线性相位 FIR 滤波器的群延迟问题，详见本教程的第 41 章。



第39章 STM32H7 的 FIR 带通滤波器实现（支持逐个数据的实时滤波）

本章节讲解 FIR 带通滤波器实现。

39.1 初学者重要提示

39.2 带通滤波器介绍

39.3 FIR 滤波器介绍

39.4 Matlab 工具箱 filterDesigner 生成带通滤波器 C 头文件

39.5 FIR 带通滤波器设计

39.6 实验例程说明 (MDK)

39.7 实验例程说明 (IAR)

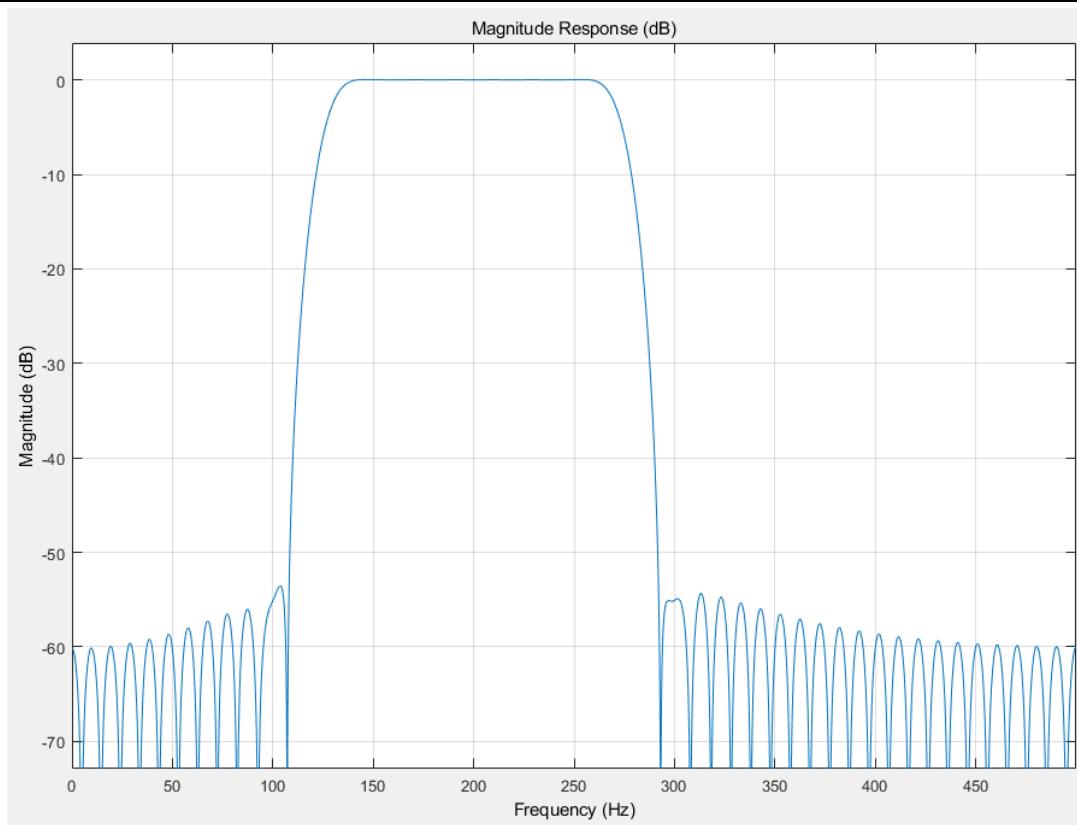
39.8 总结

39.1 初学者重要提示

- ◆ 本章节提供的带通滤波器支持实时滤波，每次可以滤波一个数据，也可以多个数据，不限制大小。但要注意以下两点：
 - 所有数据是在同一个采样率下依次采集的数据。
 - 每次过滤数据个数一旦固定下来，运行中不可再修改。
- ◆ FIR 滤波器的群延迟是一个重要的知识点，详情在本教程第 41 章有详细说明。

39.2 带通滤波器介绍

允许一个范围内的频率信号通过，而减弱范围之外频率的信号通过。比如混合信号含有 50Hz + 200Hz + 400Hz 信号，我们可通过带通滤波器，仅让 200Hz 信号通过。



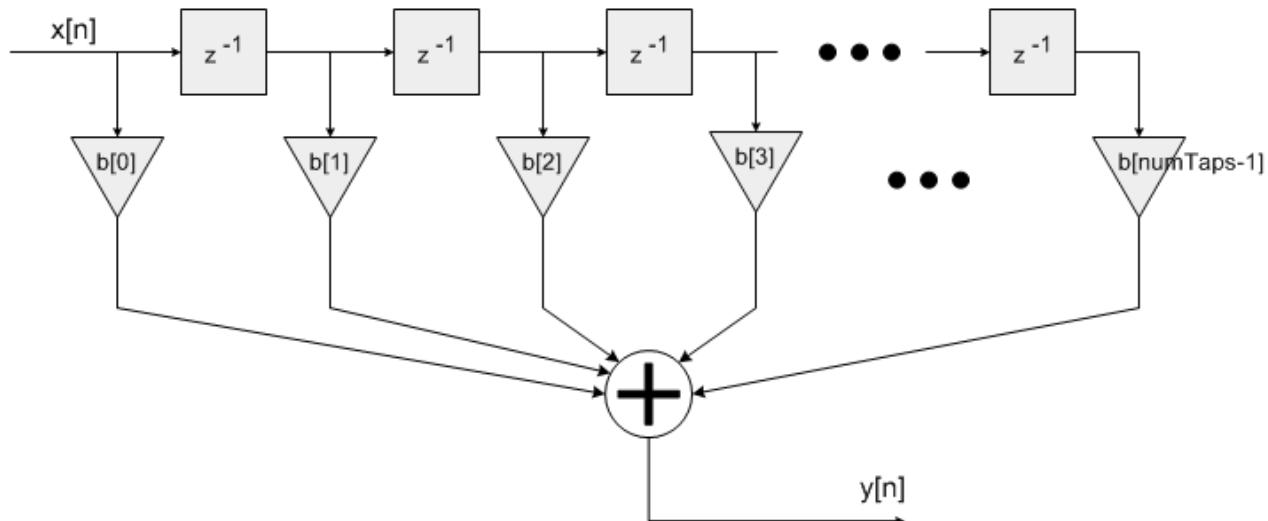
39.3 FIR 滤波器介绍

ARM 官方提供的 FIR 库支持 Q7, Q15, Q31 和浮点四种数据类型。其中 Q15 和 Q31 提供了快速算法版本。

FIR 滤波器的基本算法是一种乘法-累加 (MAC) 运行，输出表达式如下：

$$y[n] = b[0] * x[n] + b[1] * x[n-1] + b[2] * x[n-2] + \dots + b[\text{numTaps}-1] * x[n-\text{numTaps}+1]$$

结构图如下：



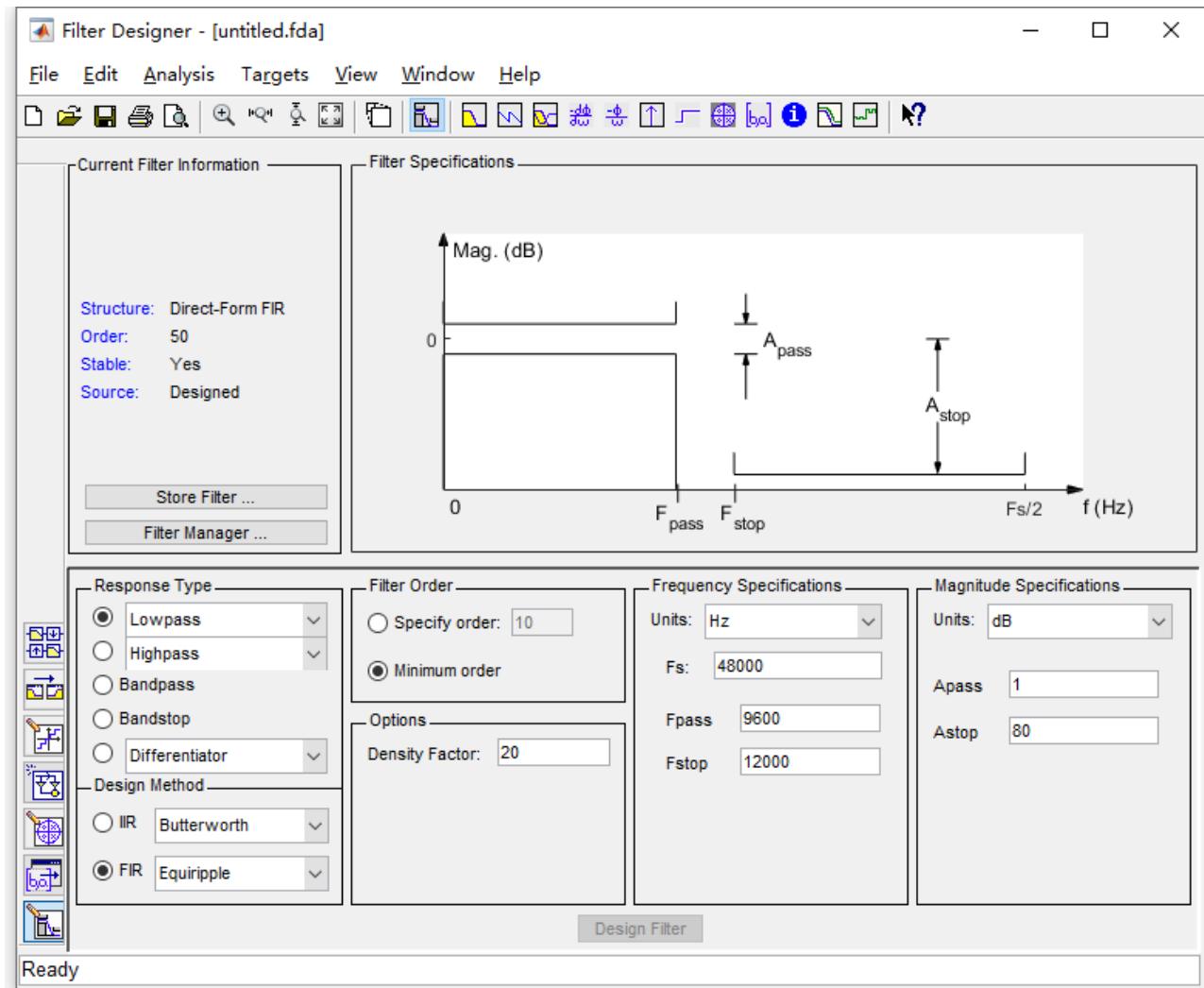
这种网络结构就是在 35.2.1 小节所讲的直接型结构。

39.4 Matlab 工具箱 filterDesigner 生成带通滤波器 C 头文件

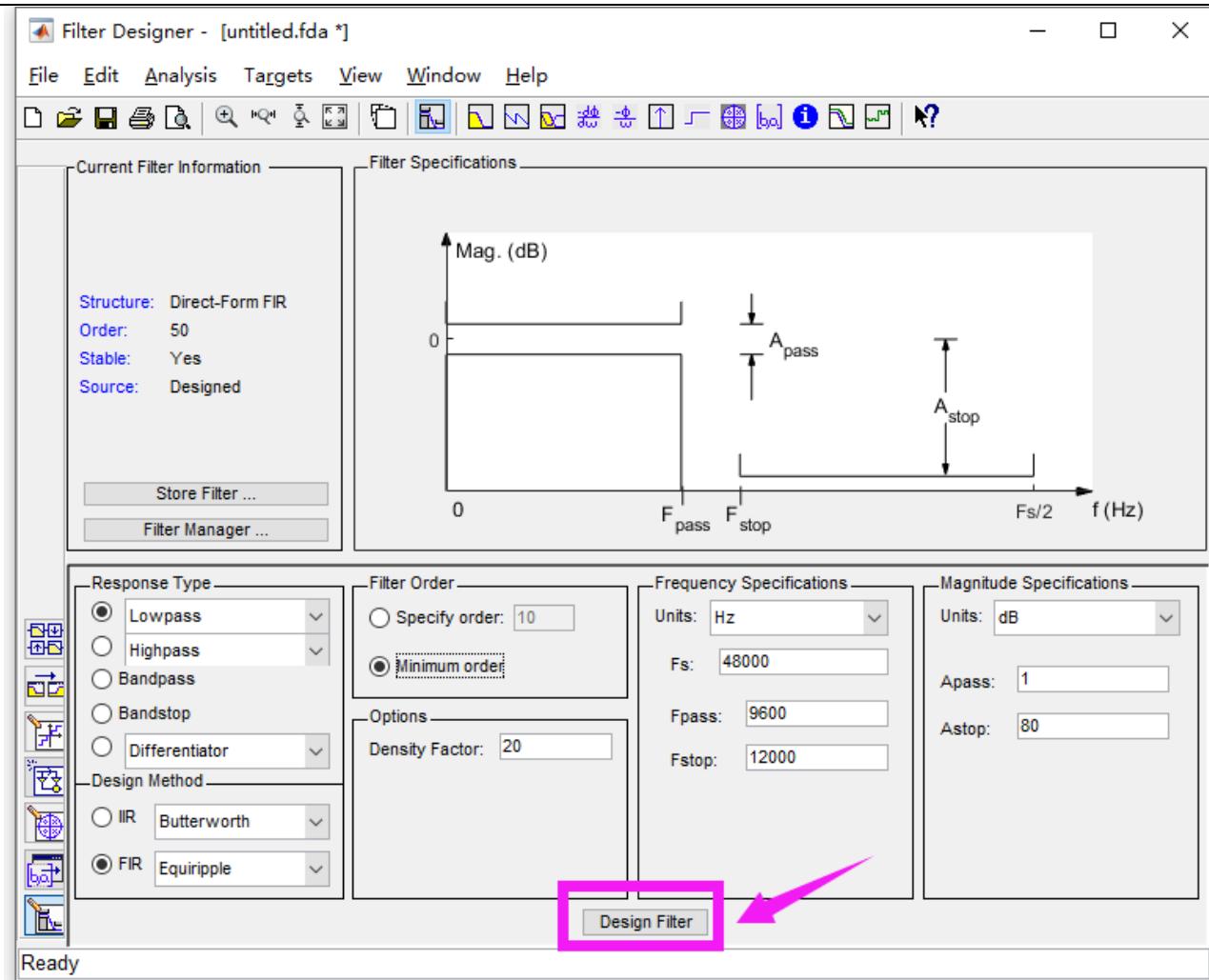
下面我们讲解下如何通过 filterDesigner 工具生成 C 头文件，也就是生成滤波器系数。首先在 matlab 的命窗口输入 filterDesigner 就能打开这个工具箱：



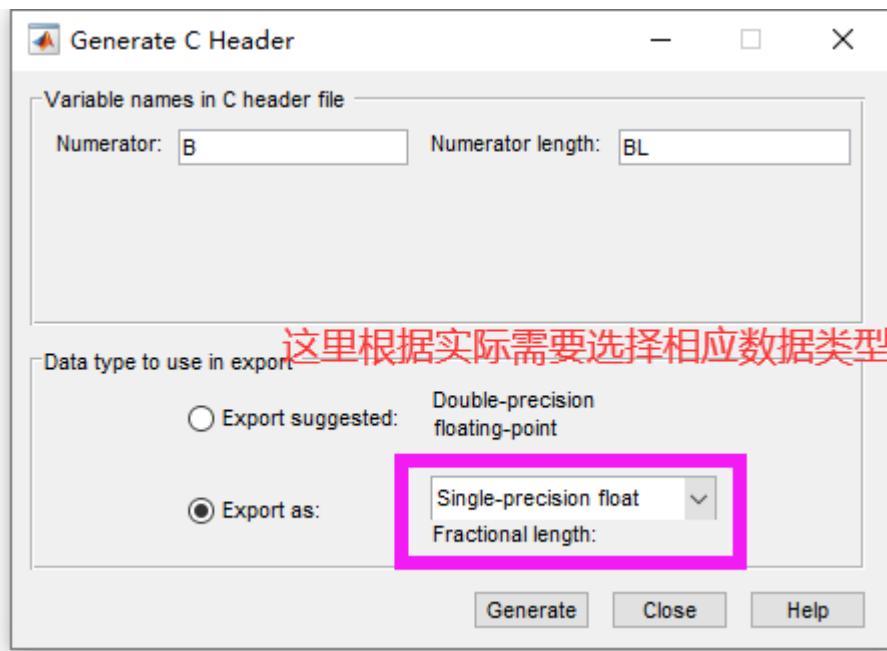
filterDesigner 界面打开效果如下：



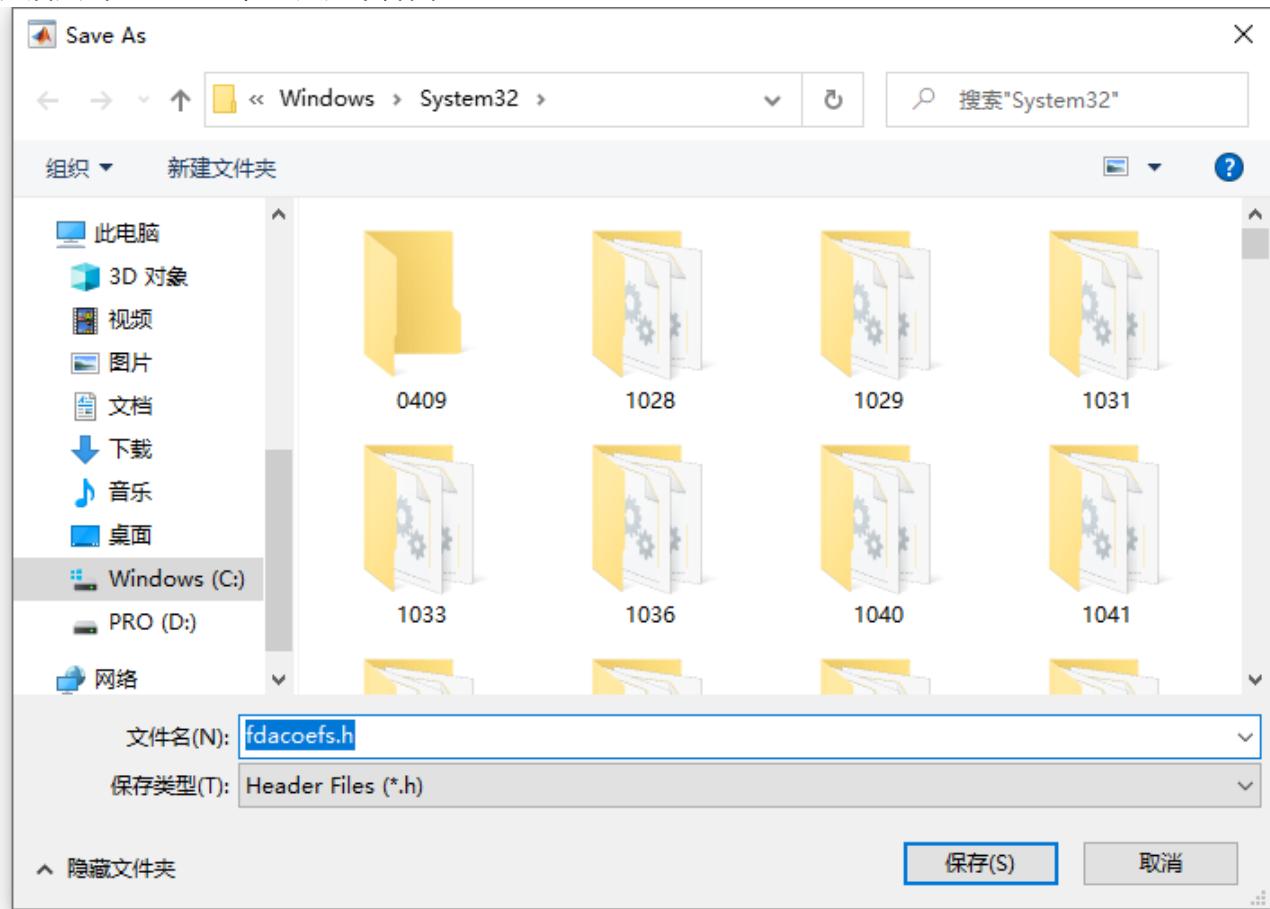
FIR 滤波器的低通，高通，带通，带阻滤波的设置会在后面逐个讲解，这里重点介绍设置后相应参数后如何生成滤波器系数。参数设置好以后点击如下按钮：



点击 Design Filter 按钮以后就生成了所需的滤波器系数，生成滤波器系数以后点击 filterDesigner 界面上的菜单 Targets->Generate C header ,打开后显示如下界面：



然后点击 Generate，生成如下界面：



再点击保存，并打开 fdatool.h 文件，可以看到生成的系数：

```
/*
 * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
 * Generated by MATLAB(R) 9.4 and Signal Processing Toolbox 8.0.
 * Generated on: 20-Jul-2021 12:19:30
 */

/*
 * Discrete-Time FIR Filter (real)
 * -----
 * Filter Structure : Direct-Form FIR
 * Filter Length   : 51
 * Stable          : Yes
 * Linear Phase    : Yes (Type 1)
 */

/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"
/*
 * Expected path to tmwtypes.h
 * D:\Program Files\MATLAB\R2018a\extern\include\tmwtypes.h
 */
/*
 * Warning - Filter coefficients were truncated to fit specified data type.
 * The resulting response may not match generated theoretical response.
 * Use the Filter Design & Analysis Tool to design accurate
 * single-precision filter coefficients.
 */
const int BL = 51;
const real32_T B[51] = {
    -0.0009190982091, -0.00271769613, -0.002486952813, 0.003661438357, 0.0136509249,
    0.01735116541, 0.00766530633, -0.006554719061, -0.007696784101, 0.006105459295,
```



```
0.01387391612, 0.0003508617228, -0.01690892503, -0.008905642666, 0.01744112931,  
0.02074504457, -0.0122964941, -0.03424086422, -0.001034529647, 0.04779030383,  
0.02736303769, -0.05937951803, -0.08230702579, 0.06718690693, 0.3100151718,  
0.4300478697, 0.3100151718, 0.06718690693, -0.08230702579, -0.05937951803,  
0.02736303769, 0.04779030383, -0.001034529647, -0.03424086422, -0.0122964941,  
0.02074504457, 0.01744112931, -0.008905642666, -0.01690892503, 0.0003508617228,  
0.01387391612, 0.006105459295, -0.007696784101, -0.006554719061, 0.00766530633,  
0.01735116541, 0.0136509249, 0.003661438357, -0.002486952813, -0.00271769613,  
-0.0009190982091  
};
```

上面数组 B[51]中的数据就是滤波器系数。下面小节讲解如何使用 filterDesigner 配置 FIR 低通，高通，带通和带阻滤波。关于 Filter Designer 的其它用法，大家可以在 matlab 命令窗口中输入 help filterDesigner 打开帮助文档进行学习。

```
命令行窗口  
>> help filterDesigner  
--- signal/filterDesigner 的帮助 ---  
  
filterDesigner - Design filters starting with algorithm selection  
  
The Filter Designer app enables you to design and analyze digital filters.  
  
另请参阅 Signal Analyzer, Window Designer, designfilt, fvtool, wvttool  
signal/filterDesigner 的参考页  
fx >> |
```

39.5 FIR 带通滤波器设计

本章使用的 FIR 滤波器函数是 arm_fir_f32。使用此函数可以设计 FIR 低通，高通，带通和带阻滤波器。

39.5.1 函数 arm_fir_init_f32

函数原型：

```
void arm_fir_init_f32(  
    arm_fir_instance_f32 * S,  
    uint16_t numTaps,  
    const float32_t * pCoeffs,  
    float32_t * pState,  
    uint32_t blockSize);
```

函数描述：

这个函数用于 FIR 初始化。

函数参数：

- ◆ 第 1 个参数是 arm_fir_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是滤波器系数的个数。
- ◆ 第 3 个参数是滤波器系数地址。
- ◆ 第 4 个参数是缓冲状态地址。



- ◆ 第 5 个参数是每次处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

注意事项：

结构体 arm_fir_instance_f32 的定义如下（在文件 arm_math.h 文件）：

```
typedef struct
{
    uint16_t numTaps;      /**< number of filter coefficients in the filter. */
    float32_t *pState;     /**< points to the state variable array. The array is of length */
                           numTaps+blockSize-1.
    float32_t *pCoeffs;    /**< points to the coefficient array. The array is of length numTaps. */
} arm_fir_instance_f32;
```

1. 参数 pCoeffs 指向滤波因数，滤波因数数组长度为 numTaps。但要注意 pCoeffs 指向的滤波因数应该按照如下的逆序进行排列：
 $\{b[numTaps-1], b[numTaps-2], b[N-2], \dots, b[1], b[0]\}$
但满足线性相位特性的 FIR 滤波器具有奇对称或者偶对称的系数，偶对称时逆序排列还是他本身。
2. pState 指向状态变量数组，这个数组用于函数内部计算数据的缓存。
3. blockSize 这个参数的大小没有特殊要求，最小可以每次处理 1 个数据，最大可以每次全部处理完。

39.5.2 函数 arm_fir_f32

函数原型：

```
void arm_fir_f32(
    const arm_fir_instance_f32 * S,
    const float32_t * pSrc,
    float32_t * pDst,
    uint32_t blockSize)
```

函数描述：

这个函数用于 FIR 滤波。

函数参数：

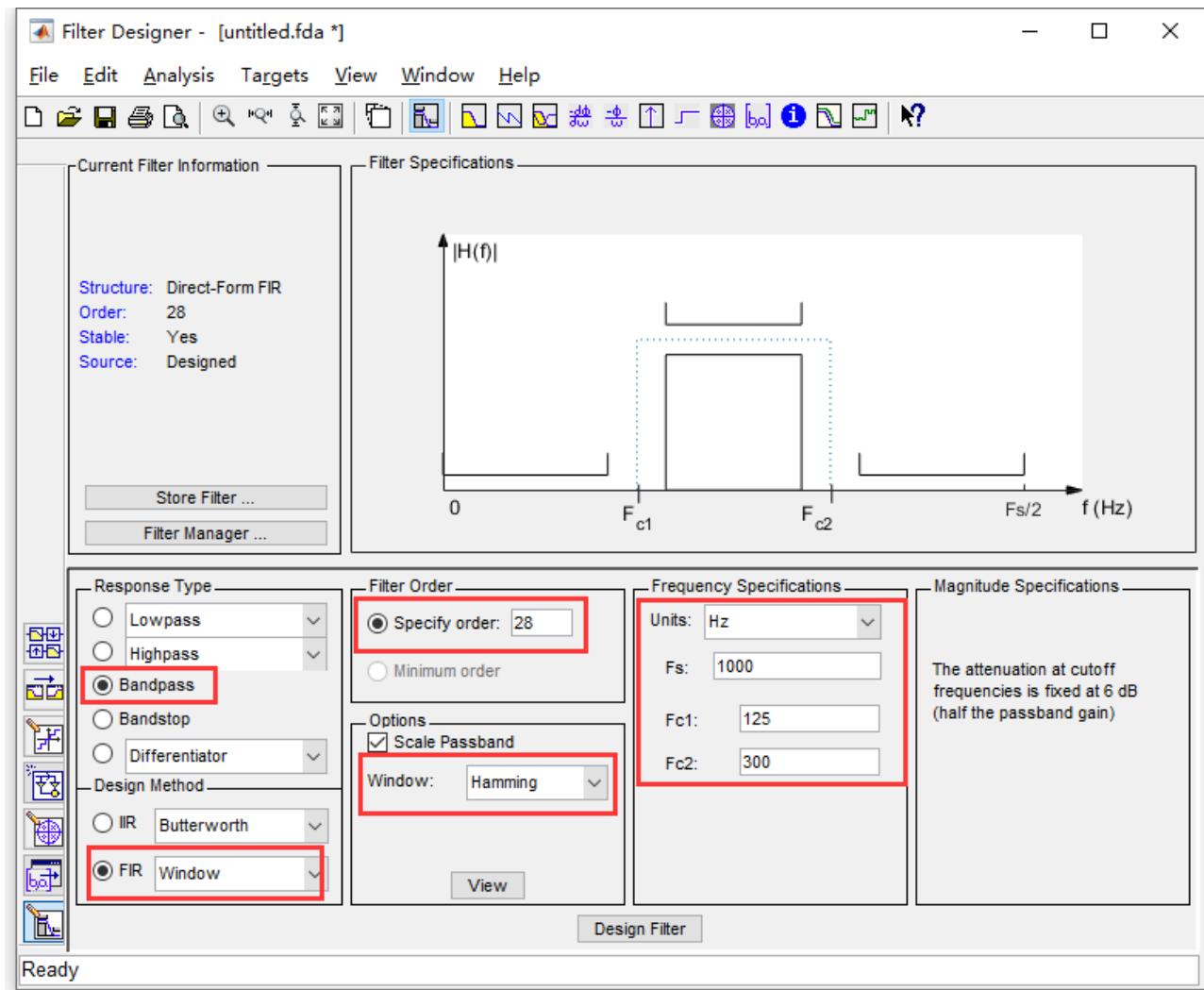
- ◆ 第 1 个参数是 arm_fir_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是源数据地址。
- ◆ 第 3 个参数是滤波后的数据地址。

第 4 个参数是每次调用处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

39.5.3 filterDesigner 获取低通滤波器系数

设计一个如下的例子：

信号由 50Hz 正弦波和 200Hz 正弦波组成，采样率 1Kbps，现设计一个带通滤波器，截止频率 125Hz 和 300Hz，采样 1024 个数据，采用函数 fir1 进行设计（注意这个函数是基于窗口的方法设计 FIR 滤波，默认是 hamming 窗），滤波器阶数设置为 28。filterDesigner 的配置如下：



配置好带通滤波器后，具体滤波器系数的生成大家参考本章第4小节的方法即可。

39.5.4 带通滤波器实现

通过工具箱filterDesigner获得带通滤波器系数后在开发板上运行函数arm_fir_f32 来测试带通滤波器的效果。

```
#define TEST_LENGTH_SAMPLES 1024 /* 采样点数 */
#define BLOCK_SIZE 1 /* 调用一次arm_fir_f32处理的采样点个数 */
#define NUM_TAPS 29 /* 滤波器系数个数 */

uint32_t blockSize = BLOCK_SIZE;
uint32_t numBlocks = TEST_LENGTH_SAMPLES/BLOCK_SIZE; /* 需要调用arm_fir_f32的次数 */

static float32_t testInput_f32_50Hz_200Hz[TEST_LENGTH_SAMPLES]; /* 采样点 */
static float32_t testOutput[TEST_LENGTH_SAMPLES]; /* 滤波后的输出 */
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1]; /* 状态缓存，大小numTaps + blockSize - 1 */

/* 低通滤波器系数 通过fadtool获取 */
const float32_t firCoeffs32LP[NUM_TAPS] = {
    -0.001822523074f, -0.001587929321f, 1.226008847e-18f, 0.003697750857f, 0.008075430058f,
    0.008530221879f, -4.273456581e-18f, -0.01739769801f, -0.03414586186f, -0.03335915506f,
    8.073562366e-18f, 0.06763084233f, 0.1522061825f, 0.2229246944f, 0.2504960895f,
```



```
0.2229246944f,    0.1522061825f,    0.06763084233f,    8.073562366e-18f, -0.03335915506f,
-0.03414586186f, -0.01739769801f,   -4.273456581e-18f,  0.008530221879f,  0.008075430058f,
0.003697750857f, 1.226008847e-18f,   -0.001587929321f,  -0.001822523074f
};
```

```
/*
*****
* 函数名: arm_fir_f32_1p
* 功能说明: 调用函数arm_fir_f32_1p实现低通滤波器
* 形参: 无
* 返回值: 无
*****
*/
static void arm_fir_f32_1p(void)
{
    uint32_t i;
    arm_fir_instance_f32 S;
    float32_t *inputF32, *outputF32;

    /* 初始化输入输出缓存指针 */
    inputF32 = &testInput_f32_50Hz_200Hz[0];
    outputF32 = &testOutput[0];

    /* 初始化结构体S */
    arm_fir_init_f32(&S,
                      NUM_TAPS,
                      (float32_t *)&firCoeffs32LP[0],
                      &firStateF32[0],
                      blockSize);

    /* 实现FIR滤波, 这里每次处理1个点 */
    for(i=0; i < numBlocks; i++)
    {
        arm_fir_f32(&S, inputF32 + (i * blockSize), outputF32 + (i * blockSize), blockSize);
    }

    /* 打印滤波后结果 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        printf("%f, %f\r\n", testOutput[i], inputF32[i]);
    }
}
```

运行如上函数可以通过串口打印出函数arm_fir_f32滤波后的波形数据，下面通过Matlab绘制波形来对比Matlab计算的结果和ARM官方库计算的结果。

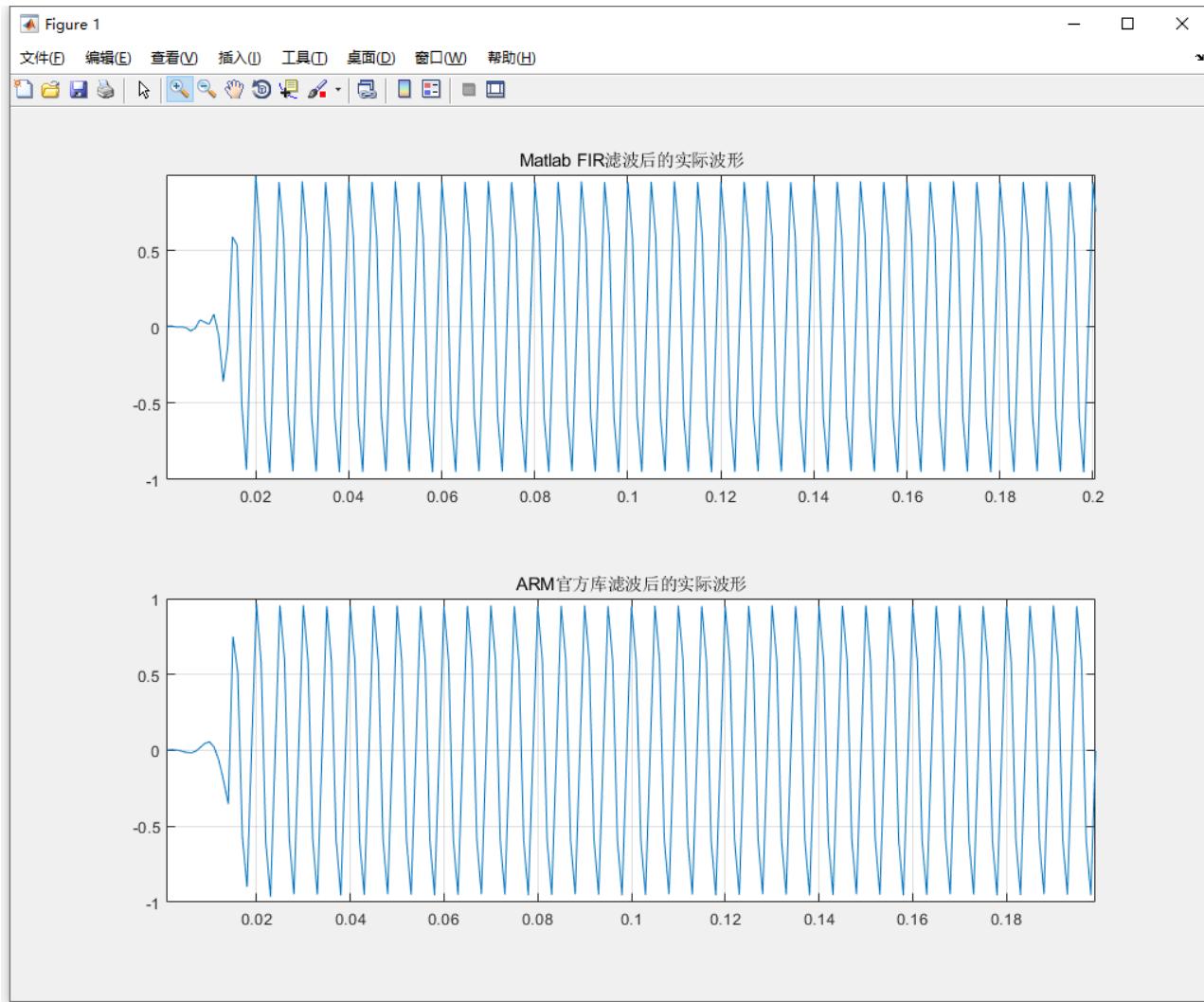
对比前需要先将串口打印出的一组数据加载到 Matlab 中，arm_fir_f32 的计算结果起名 sampledata，加载方法在前面的教程中已经讲解过，这里不做赘述了。Matlab 中运行的代码如下：

```
*****
% FIR带通滤波器设计
*****
fs=1000;          %设置采样频率 1K
N=1024;           %采样点数
n=0:N-1;
t=n/fs;           %时间序列
f=n*fs/N;          %频率序列

x=sin(2*pi*50*t)+sin(2*pi*200*t); %50Hz和200Hz正弦波混合
b=fir1(28, [125/500 300/500]);      %获得滤波器系数, 截止频率125Hz和300Hz, 带通滤波。
y=filter(b, 1, x);                  %获得滤波后的波形
subplot(211);
plot(t, y);
title('Matlab FIR滤波后的实际波形');
grid on;
```

```
subplot(212);
plot(t, sampledata); %绘制ARM官方库滤波后的波形。
title('ARM官方库滤波后的实际波形');
grid on;
```

Matlab 运行结果如下：



从上面的波形对比来看，matlab 和函数 arm_fir_f32 计算的结果基本是一致的。为了更好的说明滤波效果，下面从频域的角度来说明这个问题，Matlab 上面运行如下代码：

```
%*****
% FIR带通滤波器设计
%*****
fs=1000; %设置采样频率 1K
N=1024; %采样点数
n=0:N-1;
t=n/fs; %时间序列
f=n*fs/N; %频率序列

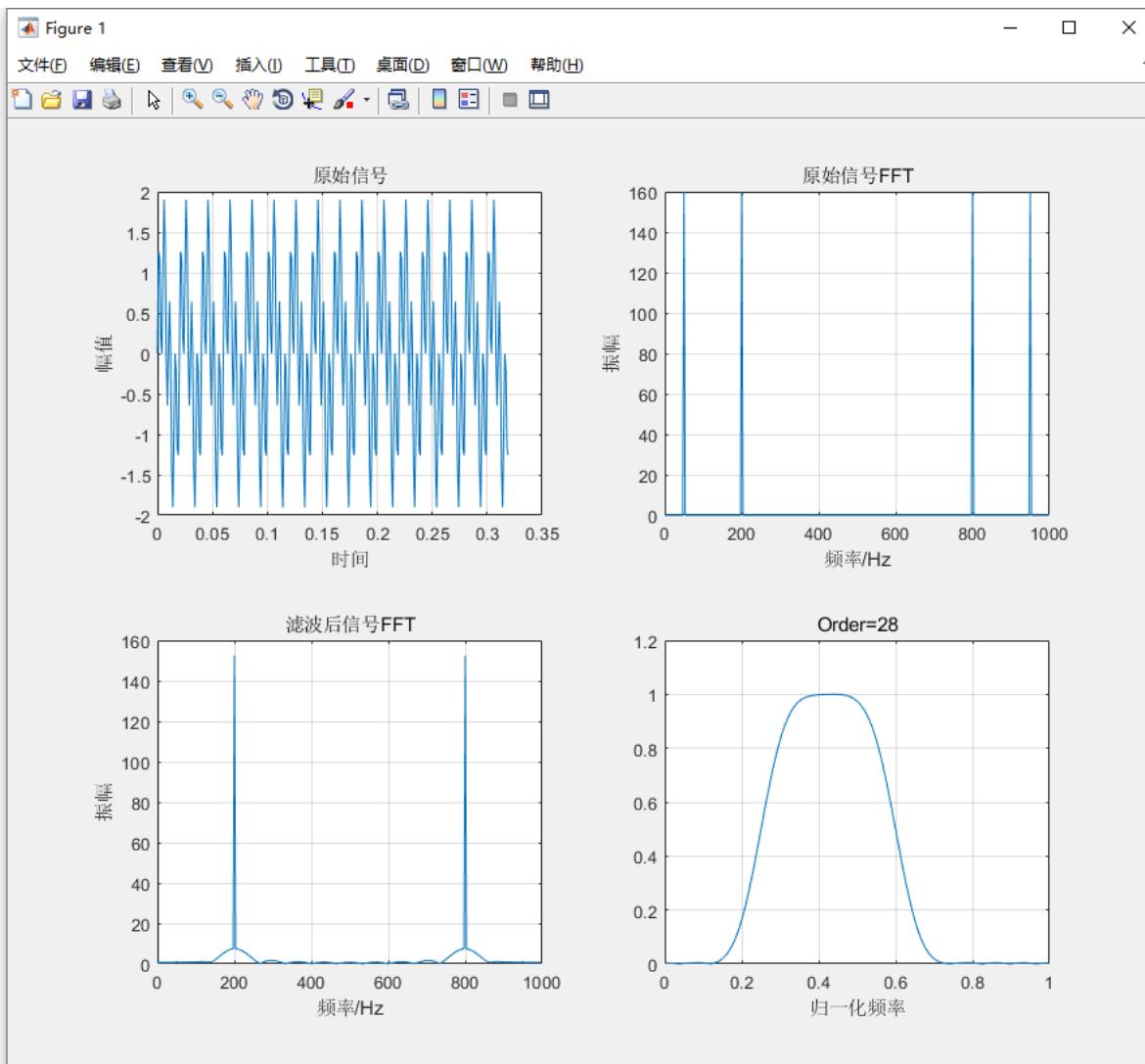
x=sin(2*pi*50*t)+sin(2*pi*200*t); %50Hz和200Hz正弦波混合
subplot(221);
plot(t, x); %绘制信号x的波形
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;
```

```
subplot(222);
y=fft(x, N);      %对信号x做FFT
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号FFT');
grid on;

y3=fft(sampledata, N);    %经过FIR滤波器后得到的信号做FFT
subplot(223);
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号FFT');
grid on;

b=fir1(28, [125/500 300/500]);    %获得滤波器系数，截止频率125Hz，高通滤波。
[H, F]=freqz(b, 1, 160);           %通过fir1设计的FIR系统的频率响应
subplot(224);
plot(F/pi, abs(H));               %绘制幅频响应
xlabel('归一化频率');
title(['Order=', int2str(28)]);
grid on;
```

Matlab 显示效果如下:





上面波形变换前的 FFT 和变换后 FFT 可以看出，50Hz 的正弦波基本被滤除。

39.6 实验例程说明 (MDK)

配套例子：

V7-227_FIR 带通滤波器设计(支持逐个数据的实时滤波)

实验目的：

1. 学习 FIR 带通滤波器的实现，支持实时滤波

实验内容：

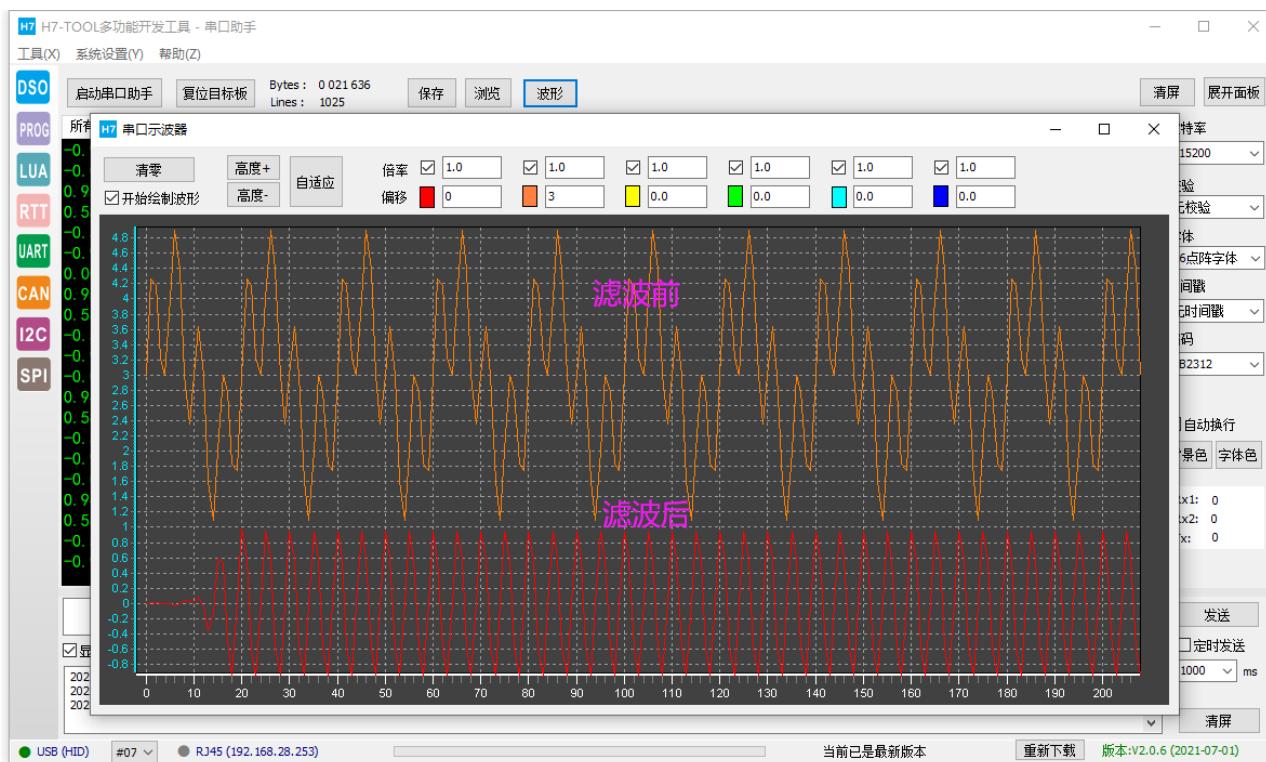
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据

使用 AC6 注意事项

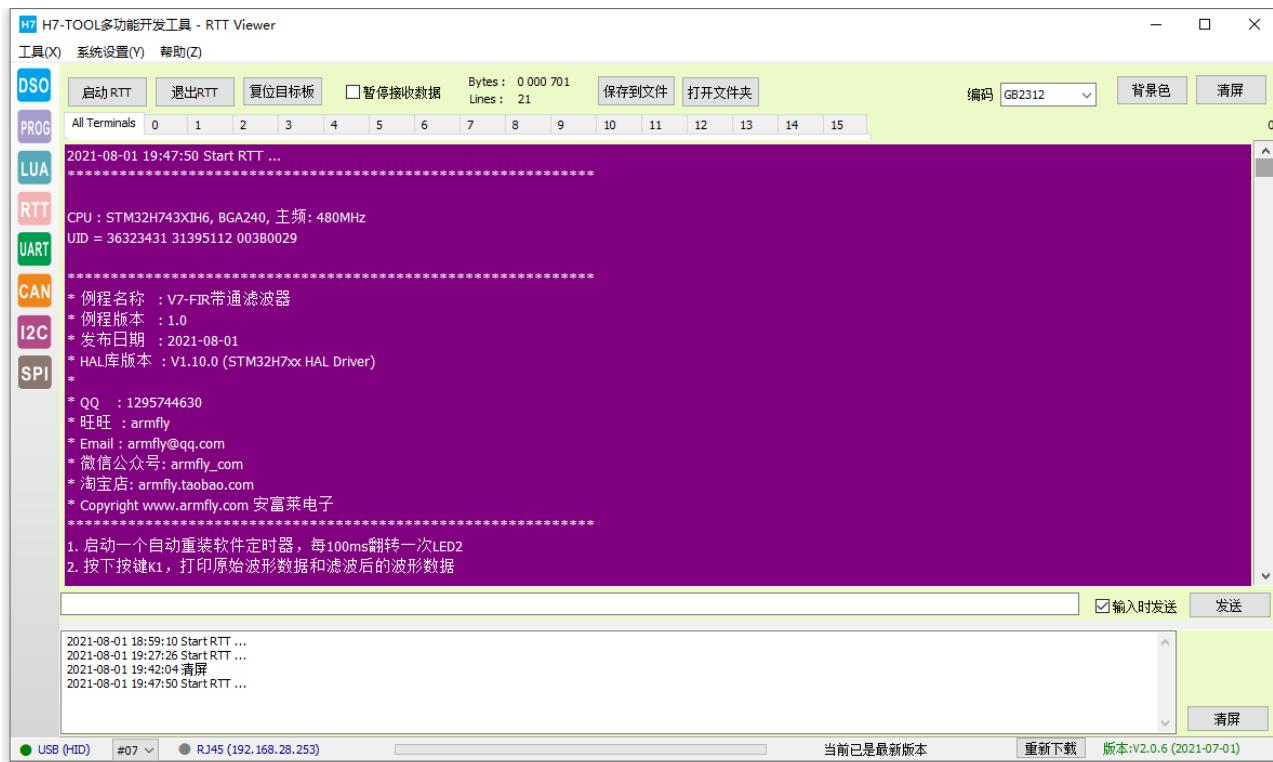
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

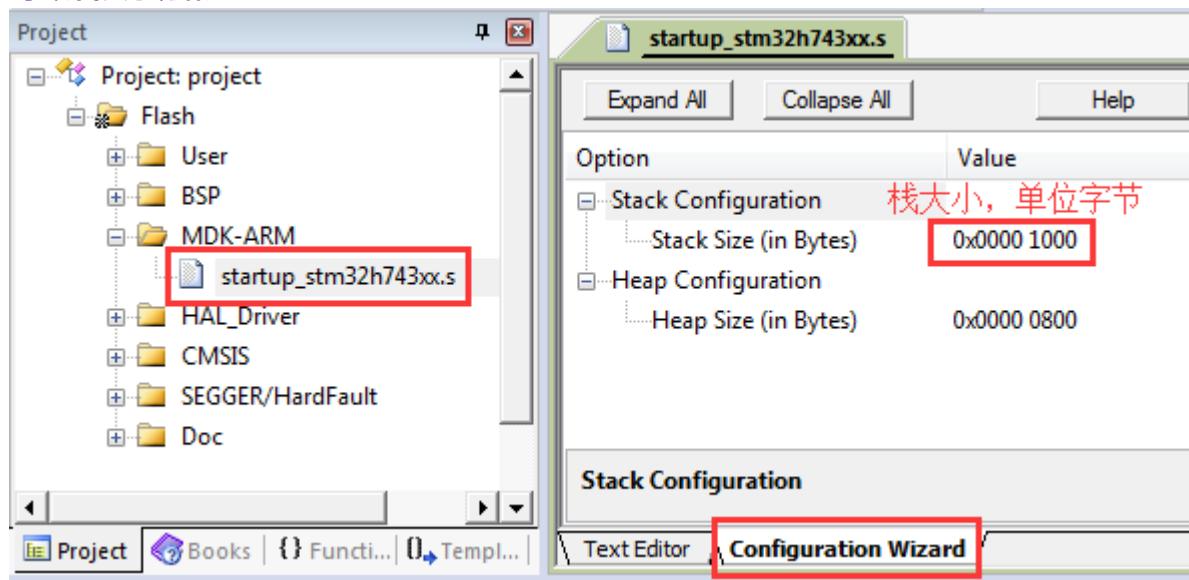


RTT 方式打印信息：

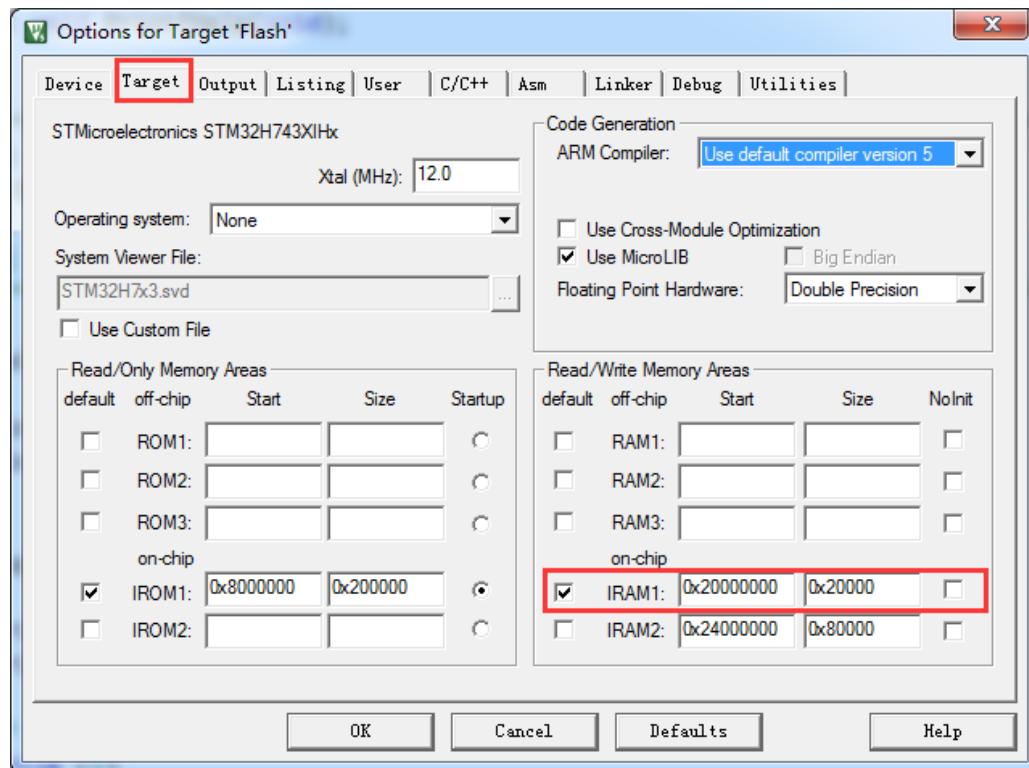


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
    testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                    arm_sin_f32(2*3.1415926f*200*i/1000);
}

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2); /* 翻转 LED2 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_fir_f32_bp();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
}
```

39.7 实验例程说明 (IAR)

配套例子：

V7-227_FIR 带通滤波器设计(支持逐个数据的实时滤波)

实验目的：

1. 学习 FIR 带通滤波器的实现，支持实时滤波

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据

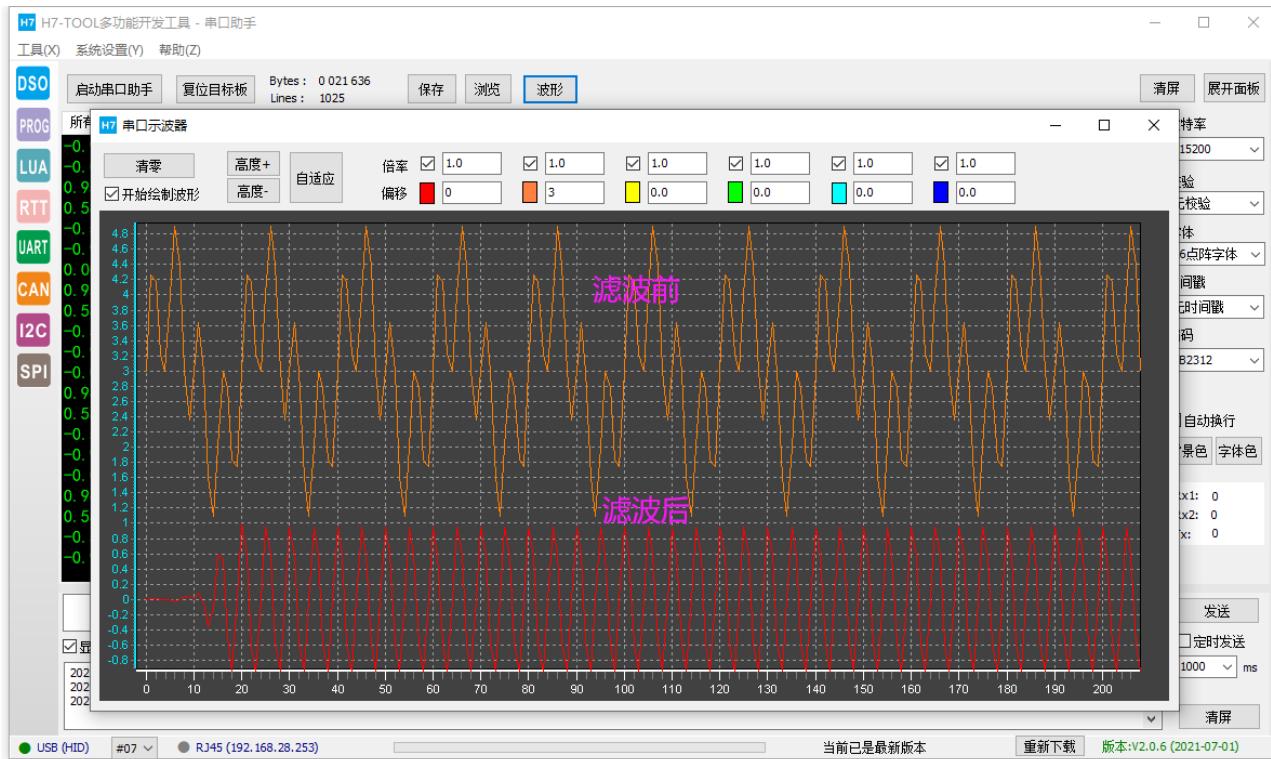
使用 AC6 注意事项



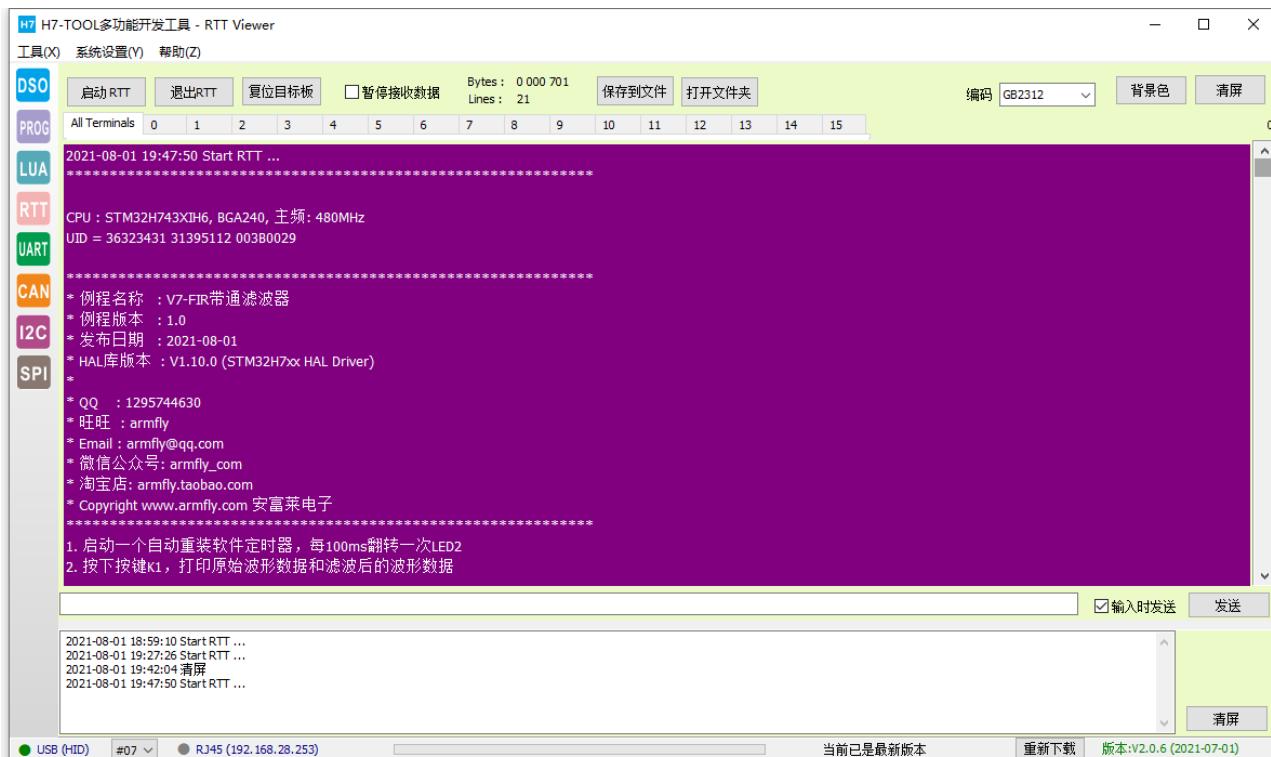
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。



RTT 方式打印信息：



程序设计：

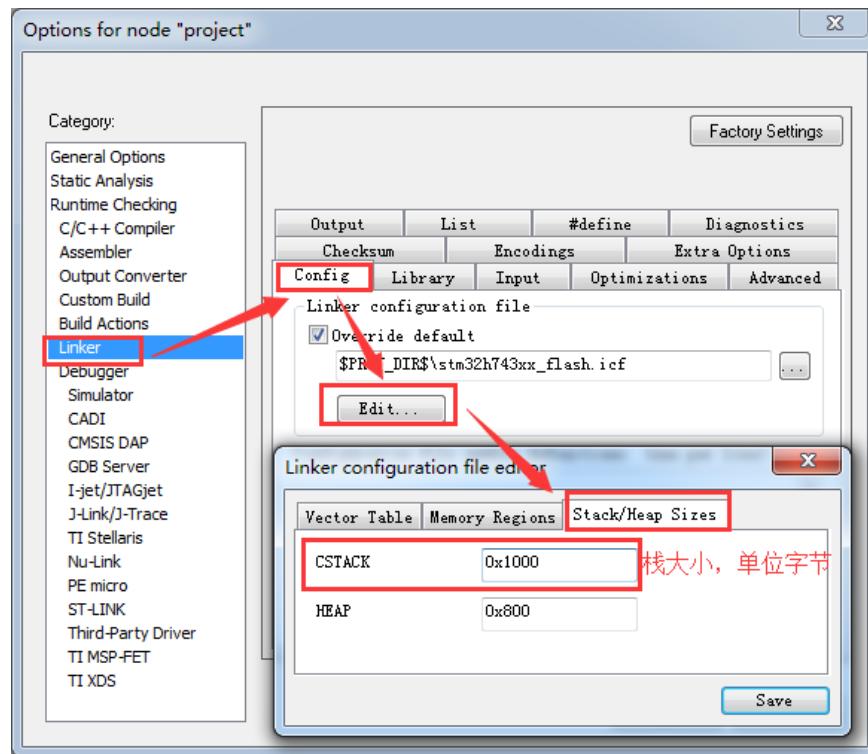
2021年11月01日

版本：2.7

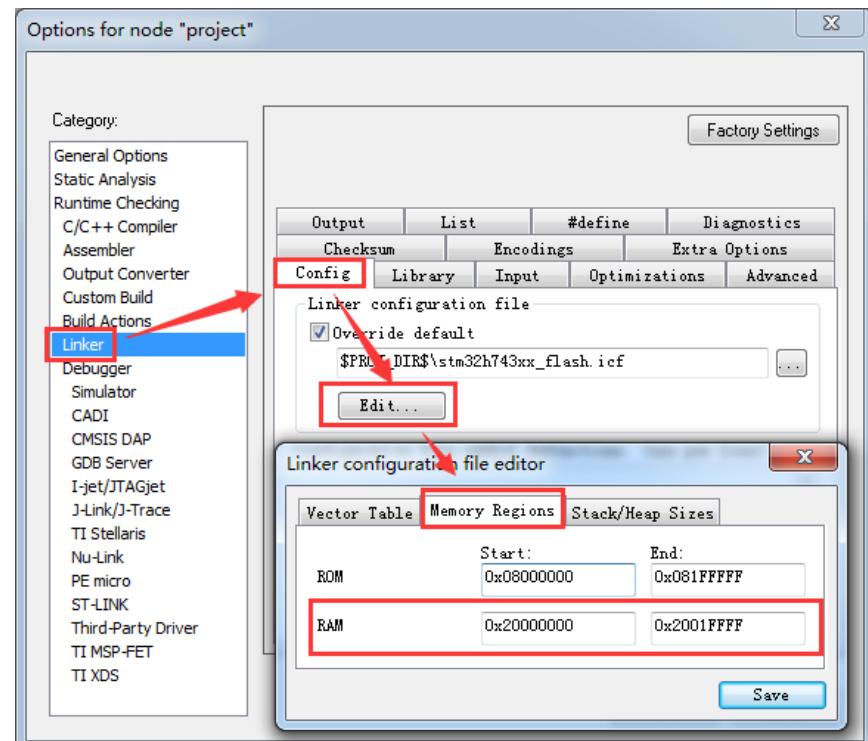
第 715 页 共 943 页



◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现:

```
/*  
*****
```



```
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
        - 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
```



```
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL MPU_ConfigRegion(&MPU_InitStruct);

    /*使能 MPU */
    HAL MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
        testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                      arm_sin_f32(2*3.1415926f*200*i/1000);
    }

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2); /* 翻转 LED2 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_fir_f32_bp();
                    break;
            }
        }
    }
}
```



```
default:  
    /* 其它的键值不处理 */  
    break;  
}  
}  
}
```

39.8 总结

本章节主要讲解了 FIR 滤波器的带通实现，同时一定要注意线性相位 FIR 滤波器的群延迟问题，详见本教程的第 41 章。



第40章 STM32H7 的 FIR 带阻滤波器实现（支持逐个数据的实时滤波）

本本章节讲解 FIR 带阻滤波器实现。

40.1 初学者重要提示

40.2 带阻滤波器介绍

40.3 FIR 滤波器介绍

40.4 Matlab 工具箱 filterDesigner 生成带阻滤波器 C 头文件

40.5 FIR 带阻滤波器设计

40.6 实验例程说明 (MDK)

40.7 实验例程说明 (IAR)

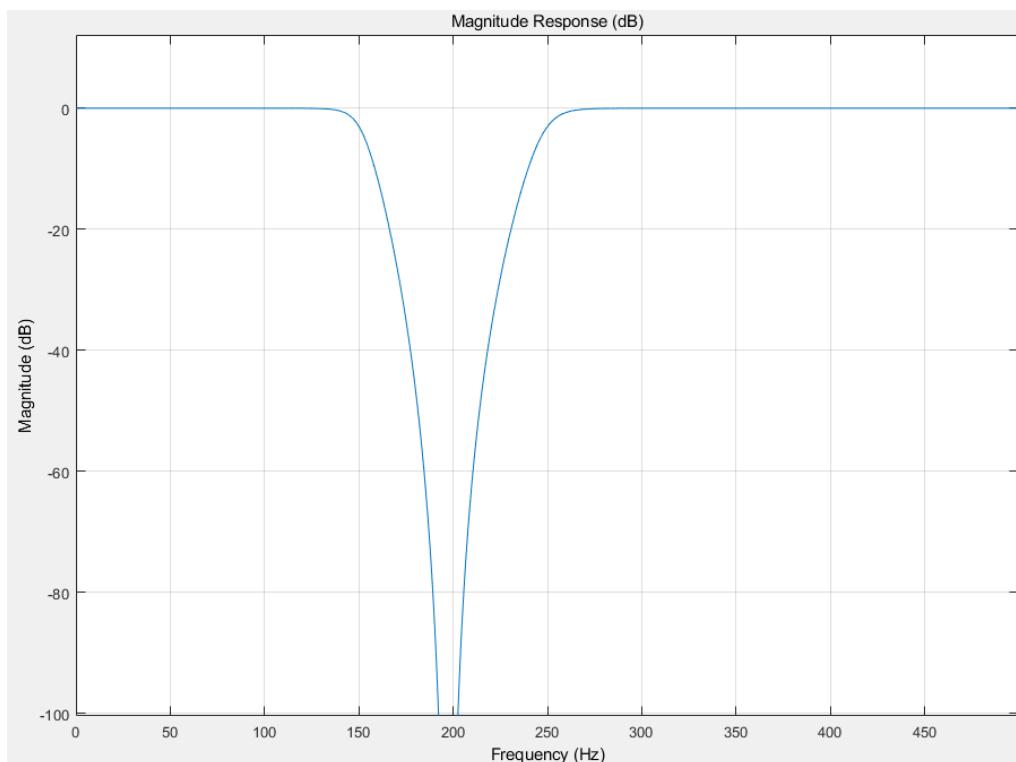
40.8 总结

40.1 初学者重要提示

- ◆ 本章节提供的带阻滤波器支持实时滤波，每次可以滤波一个数据，也可以多个数据，不限制大小。但要注意以下两点：
 - 所有数据是在同一个采样率下依次采集的数据。
 - 每次过滤数据个数一旦固定下来，运行中不可再修改。
- ◆ FIR 滤波器的群延迟是一个重要的知识点，详情在本教程第 41 章有详细说明。

40.2 带阻滤波器介绍

减弱一个范围内的频率信号通过，让范围之外的频率信号通过。比如混合信号含有 50Hz + 200Hz + 400Hz 信号，我们可通过带通滤波器，让 50Hz + 400Hz 信号通过，而阻止 200Hz 信号通过。



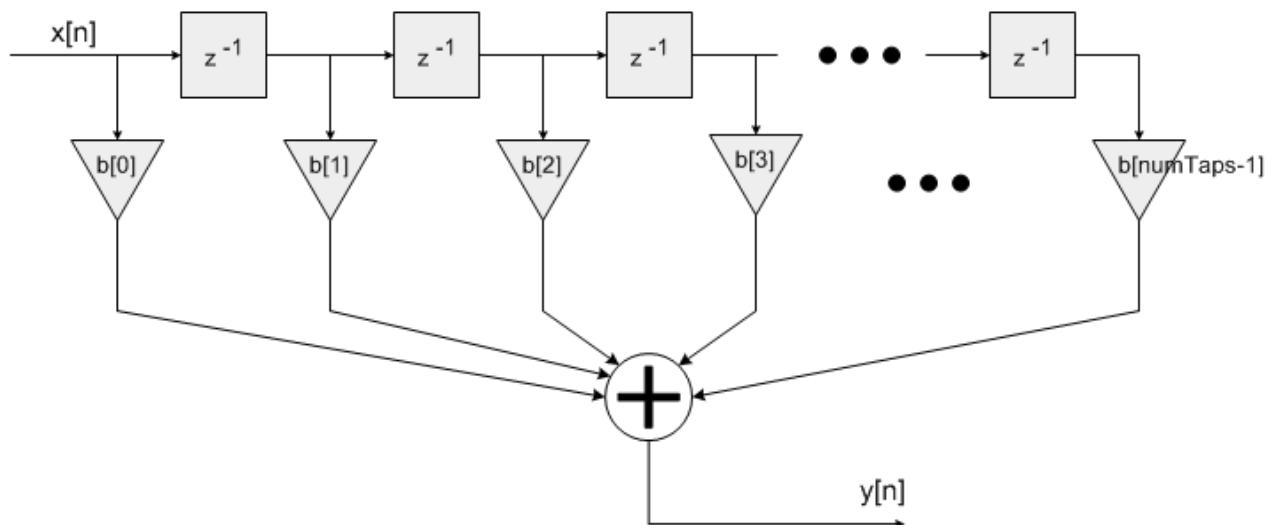
40.3 FIR 滤波器介绍

ARM 官方提供的 FIR 库支持 Q7, Q15, Q31 和浮点四种数据类型。其中 Q15 和 Q31 提供了快速算法版本。

FIR 滤波器的基本算法是一种乘法-累加 (MAC) 运行，输出表达式如下：

$$y[n] = b[0] * x[n] + b[1] * x[n-1] + b[2] * x[n-2] + \dots + b[\text{numTaps}-1] * x[n-\text{numTaps}+1]$$

结构图如下：



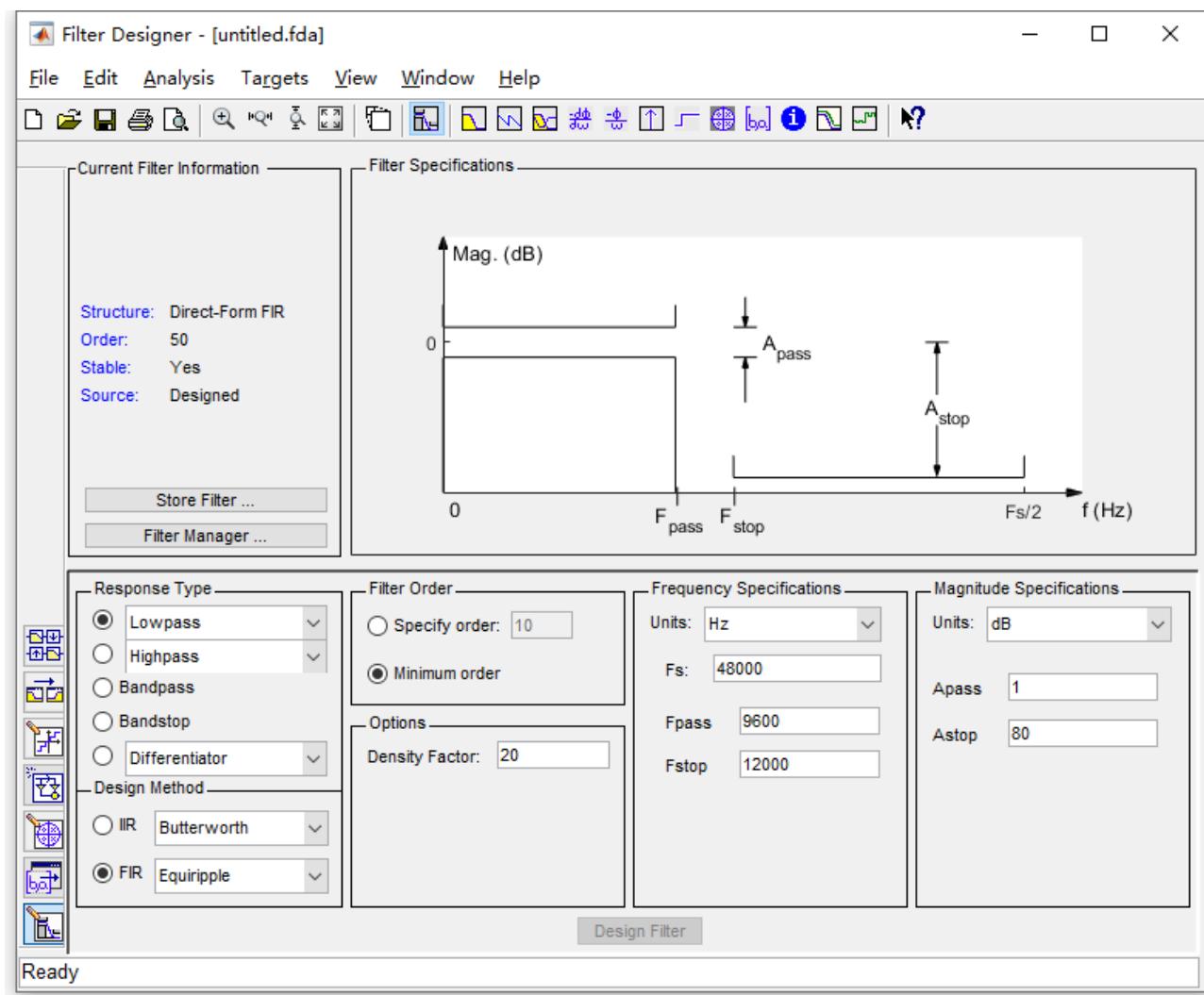
这种网络结构就是在 35.2.1 小节所讲的直接型结构。

40.4 Matlab 工具箱 filterDesigner 生成带阻滤波器 C 头文件

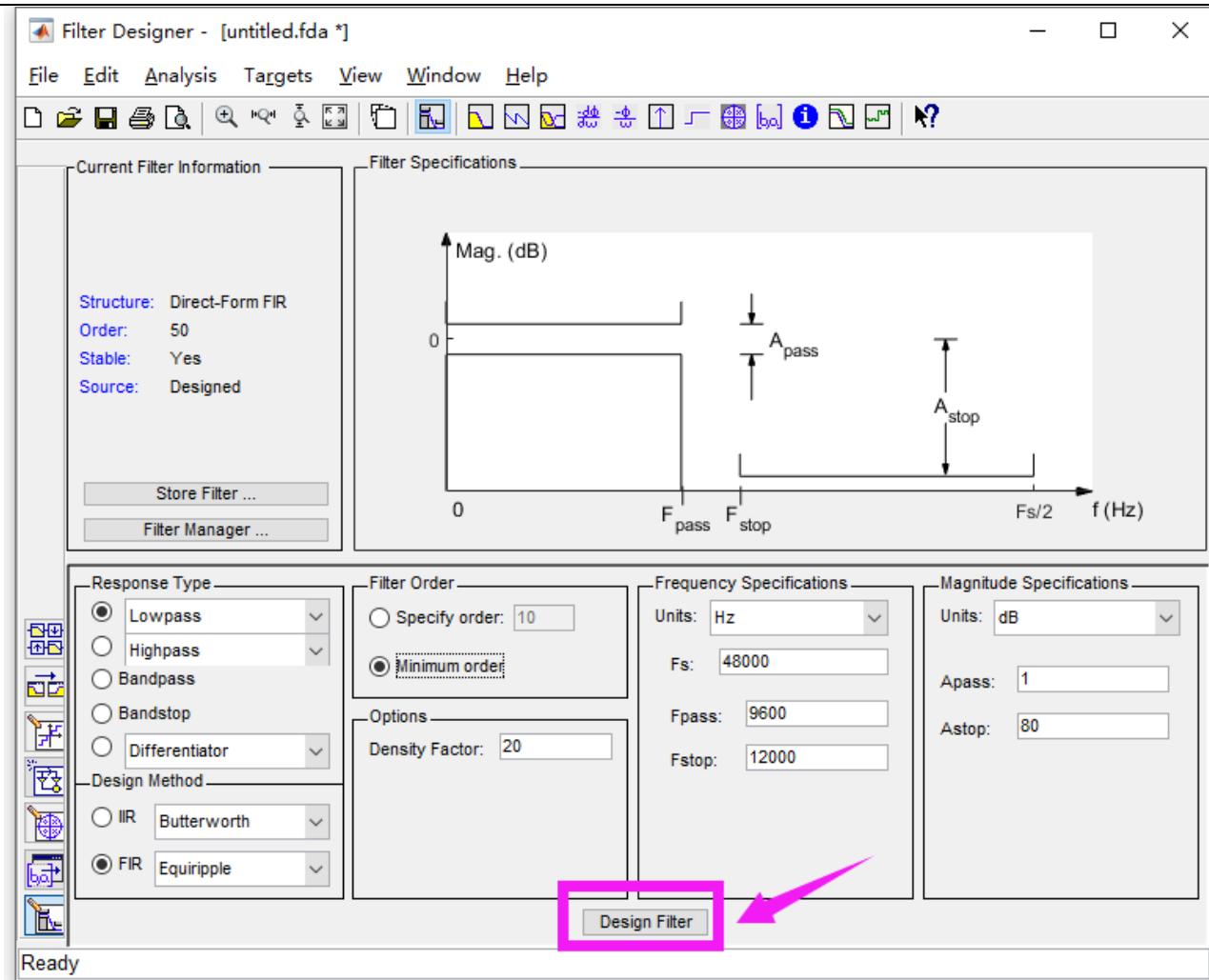
下面我们讲解下如何通过 filterDesigner 工具生成 C 头文件，也就是生成滤波器系数。首先在 matlab 的命窗口输入 filterDesigner 就能打开这个工具箱：



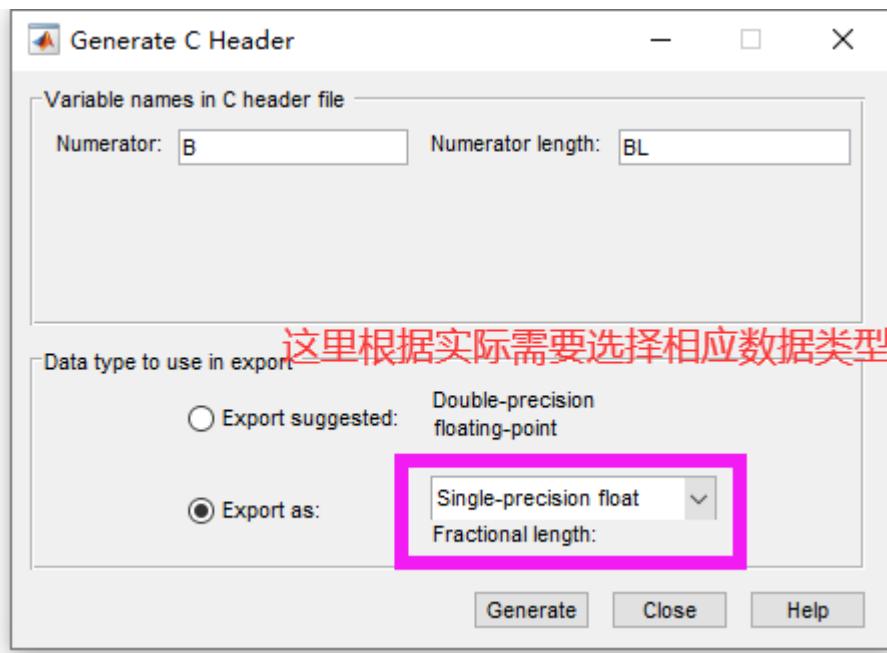
filterDesigner 界面打开效果如下：



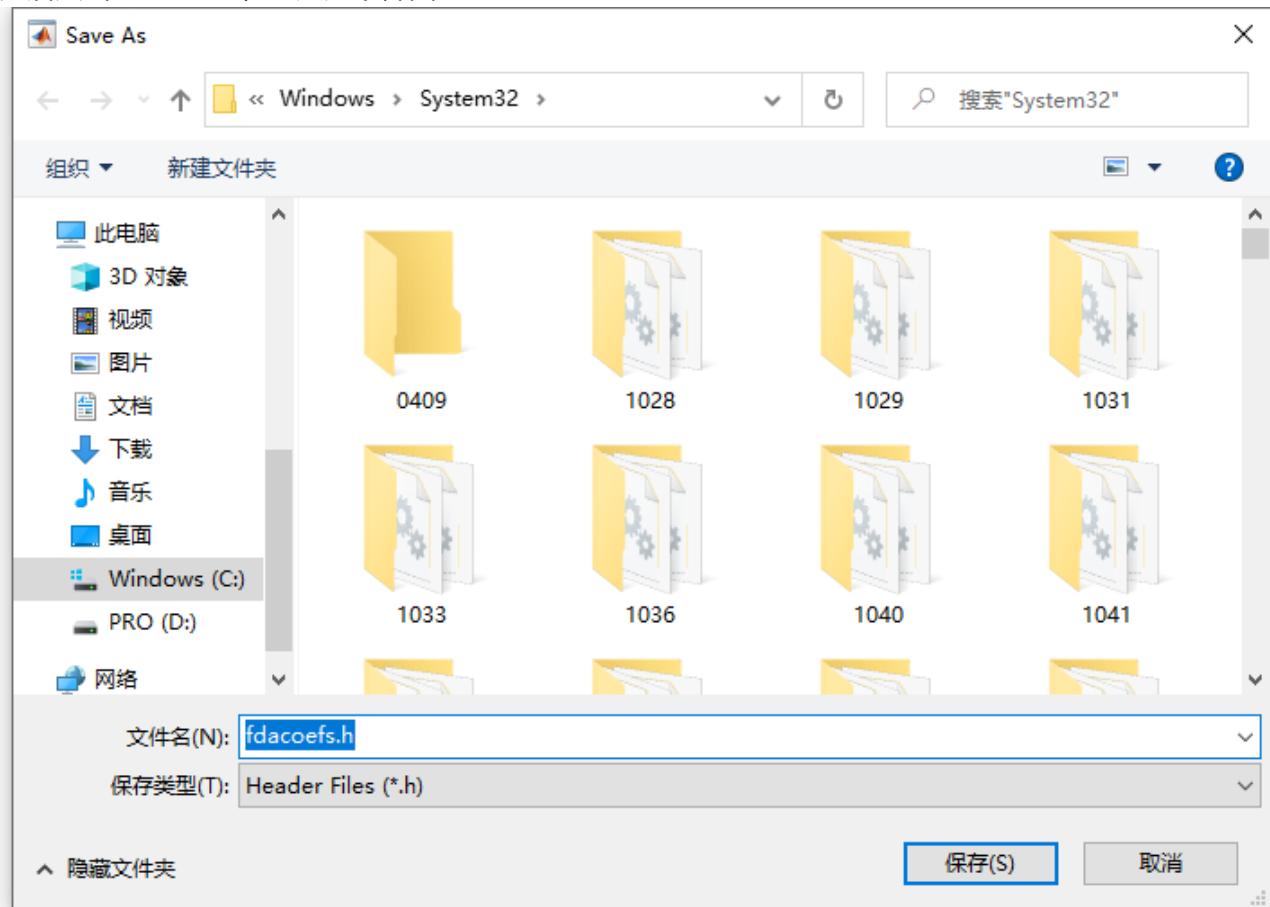
FIR 滤波器的低通，高通，带通，带阻滤波的设置会在后面逐个讲解，这里重点介绍设置后相应参数后如何生成滤波器系数。参数设置好以后点击如下按钮：



点击 Design Filter 按钮以后就生成了所需的滤波器系数，生成滤波器系数以后点击 filterDesigner 界面上的菜单 Targets->Generate C header ,打开后显示如下界面：



然后点击 Generate，生成如下界面：



再点击保存，并打开 fdatool.h 文件，可以看到生成的系数：

```
/*
 * Filter Coefficients (C Source) generated by the Filter Design and Analysis Tool
 * Generated by MATLAB(R) 9.4 and Signal Processing Toolbox 8.0.
 * Generated on: 20-Jul-2021 12:19:30
 */

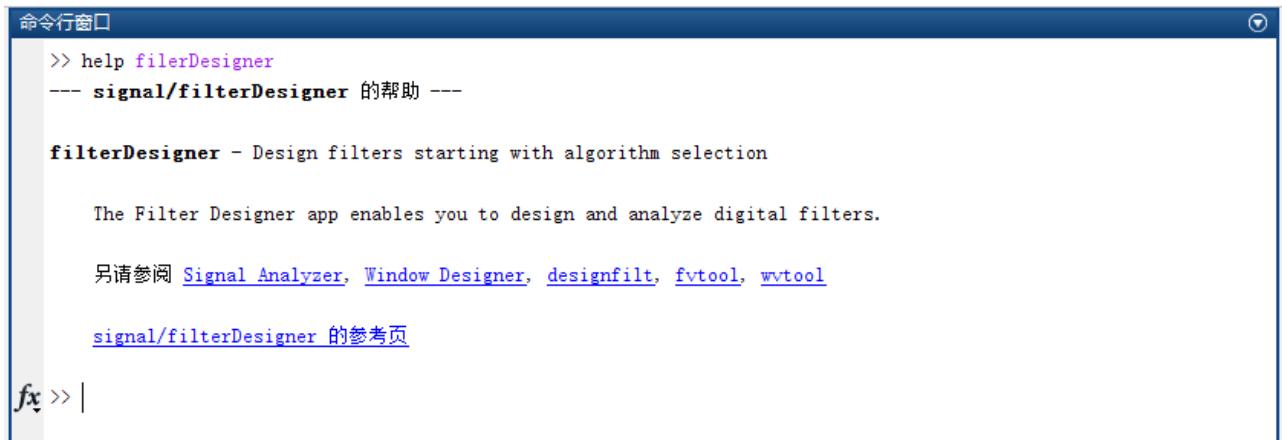
/*
 * Discrete-Time FIR Filter (real)
 * -----
 * Filter Structure : Direct-Form FIR
 * Filter Length   : 51
 * Stable          : Yes
 * Linear Phase    : Yes (Type 1)
 */

/* General type conversion for MATLAB generated C-code */
#include "tmwtypes.h"
/*
 * Expected path to tmwtypes.h
 * D:\Program Files\MATLAB\R2018a\extern\include\tmwtypes.h
 */
/*
 * Warning - Filter coefficients were truncated to fit specified data type.
 * The resulting response may not match generated theoretical response.
 * Use the Filter Design & Analysis Tool to design accurate
 * single-precision filter coefficients.
 */
const int BL = 51;
const real32_T B[51] = {
    -0.0009190982091, -0.00271769613, -0.002486952813, 0.003661438357, 0.0136509249,
    0.01735116541, 0.00766530633, -0.006554719061, -0.007696784101, 0.006105459295,
```



```
0.01387391612, 0.0003508617228, -0.01690892503, -0.008905642666, 0.01744112931,  
0.02074504457, -0.0122964941, -0.03424086422, -0.001034529647, 0.04779030383,  
0.02736303769, -0.05937951803, -0.08230702579, 0.06718690693, 0.3100151718,  
0.4300478697, 0.3100151718, 0.06718690693, -0.08230702579, -0.05937951803,  
0.02736303769, 0.04779030383, -0.001034529647, -0.03424086422, -0.0122964941,  
0.02074504457, 0.01744112931, -0.008905642666, -0.01690892503, 0.0003508617228,  
0.01387391612, 0.006105459295, -0.007696784101, -0.006554719061, 0.00766530633,  
0.01735116541, 0.0136509249, 0.003661438357, -0.002486952813, -0.00271769613,  
-0.0009190982091  
};
```

上面数组 B[51]中的数据就是滤波器系数。下面小节讲解如何使用 filterDesigner 配置 FIR 低通，高通，带通和带阻滤波。关于 Filter Designer 的其它用法，大家可以在 matlab 命令窗口中输入 help filterDesigner 打开帮助文档进行学习。



40.5 FIR 带通滤波器设计

本章使用的 FIR 滤波器函数是 arm_fir_f32。使用此函数可以设计 FIR 低通，高通，带通和带阻滤波器。

40.5.1 函数 arm_fir_init_f32

函数原型：

```
void arm_fir_init_f32(  
    arm_fir_instance_f32 * S,  
    uint16_t numTaps,  
    const float32_t * pCoeffs,  
    float32_t * pState,  
    uint32_t blockSize);
```

函数描述：

这个函数用于 FIR 初始化。

函数参数：

- ◆ 第 1 个参数是 arm_fir_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是滤波器系数的个数。
- ◆ 第 3 个参数是滤波器系数地址。
- ◆ 第 4 个参数是缓冲状态地址。



- ◆ 第 5 个参数是每次处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

注意事项：

结构体 arm_fir_instance_f32 的定义如下（在文件 arm_math.h 文件）：

```
typedef struct
{
    uint16_t numTaps;      /**< number of filter coefficients in the filter. */
    float32_t *pState;     /**< points to the state variable array. The array is of length */
                           numTaps+blockSize-1.
    float32_t *pCoeffs;    /**< points to the coefficient array. The array is of length numTaps. */
} arm_fir_instance_f32;
```

1. 参数 pCoeffs 指向滤波因数，滤波因数数组长度为 numTaps。但要注意 pCoeffs 指向的滤波因数应该按照如下的逆序进行排列：
 $\{b[numTaps-1], b[numTaps-2], b[N-2], \dots, b[1], b[0]\}$
但满足线性相位特性的 FIR 滤波器具有奇对称或者偶对称的系数，偶对称时逆序排列还是他本身。
2. pState 指向状态变量数组，这个数组用于函数内部计算数据的缓存。
3. blockSize 这个参数的大小没有特殊要求，最小可以每次处理 1 个数据，最大可以每次全部处理完。

40.5.2 函数 arm_fir_f32

函数原型：

```
void arm_fir_f32(
    const arm_fir_instance_f32 * S,
    const float32_t * pSrc,
    float32_t * pDst,
    uint32_t blockSize)
```

函数描述：

这个函数用于 FIR 滤波。

函数参数：

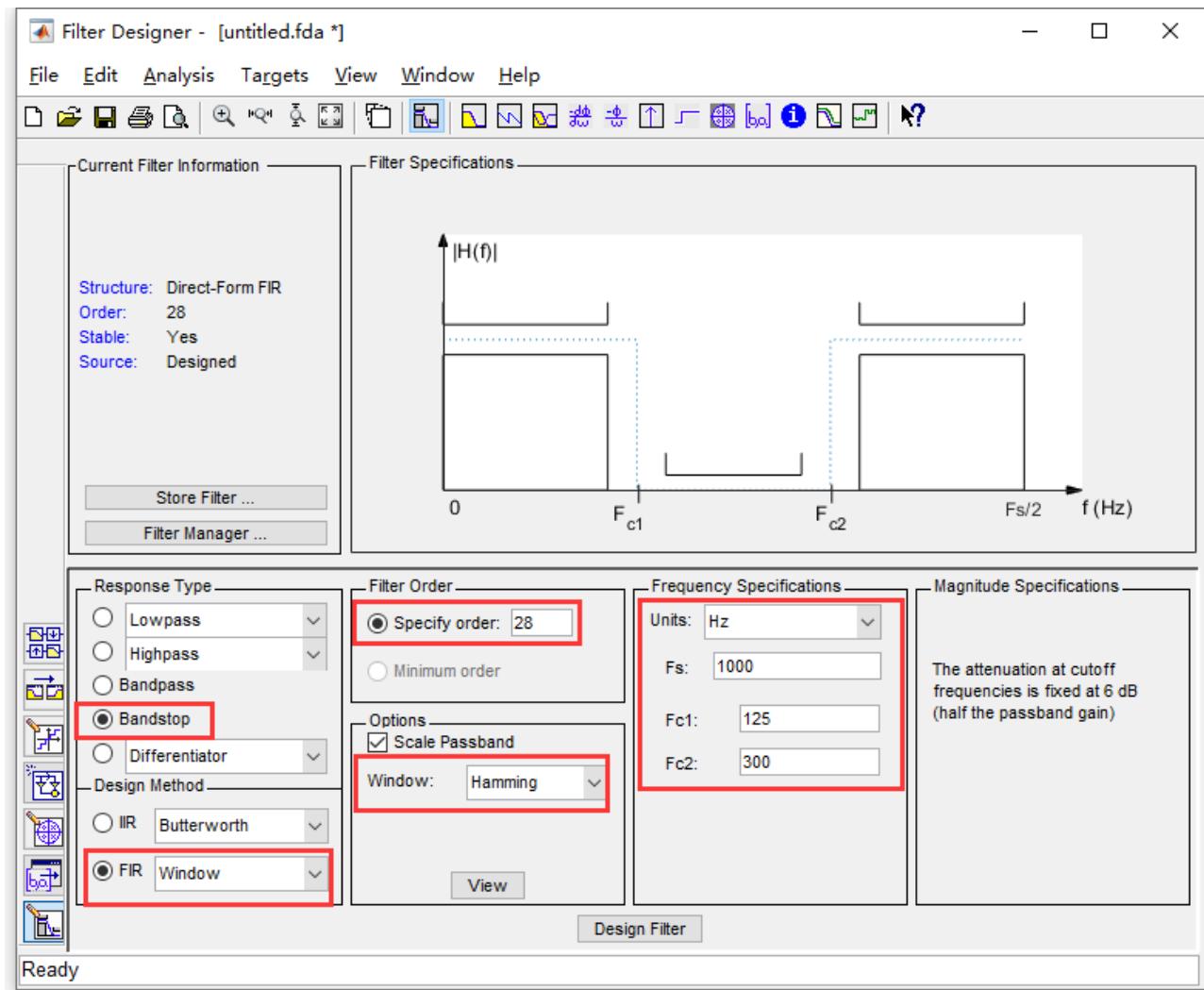
- ◆ 第 1 个参数是 arm_fir_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是源数据地址。
- ◆ 第 3 个参数是滤波后的数据地址。

第 4 个参数是每次调用处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

40.5.3 filterDesigner 获取低通滤波器系数

设计一个如下的例子：

信号由 50Hz 正弦波和 200Hz 正弦波组成，采样率 1Kbps，现设计一个带阻滤波器，截止频率 125Hz 和 300Hz，采样 1024 个数据，采用函数 fir1 进行设计（注意这个函数是基于窗口的方法设计 FIR 滤波，默认是 hamming 窗），滤波器阶数设置为 28。filterDesigner 的配置如下：



配置好带阻滤波器后，具体滤波器系数的生成大家参考本章第4小节的方法即可。

40.5.4 带阻滤波器实现

通过工具箱filterDesigner获得低通滤波器系数后在开发板上运行函数arm_fir_f32 来测试带阻滤波器的效果。

```
#define TEST_LENGTH_SAMPLES 1024 /* 采样点数 */
#define BLOCK_SIZE 1 /* 调用一次arm_fir_f32处理的采样点个数 */
#define NUM_TAPS 29 /* 滤波器系数个数 */

uint32_t blockSize = BLOCK_SIZE;
uint32_t numBlocks = TEST_LENGTH_SAMPLES/BLOCK_SIZE; /* 需要调用arm_fir_f32的次数 */

static float32_t testInput_f32_50Hz_200Hz[TEST_LENGTH_SAMPLES]; /* 采样点 */
static float32_t testOutput[TEST_LENGTH_SAMPLES]; /* 滤波后的输出 */
static float32_t firStateF32[BLOCK_SIZE + NUM_TAPS - 1]; /* 状态缓存，大小numTaps + blockSize - 1 */

/* 低通滤波器系数 通过fadtool获取 */
const float32_t firCoeffs32LP[NUM_TAPS] = {
    -0.001822523074f, -0.001587929321f, 1.226008847e-18f, 0.003697750857f, 0.008075430058f,
    0.008530221879f, -4.273456581e-18f, -0.01739769801f, -0.03414586186f, -0.03335915506f,
    8.073562366e-18f, 0.06763084233f, 0.1522061825f, 0.2229246944f, 0.2504960895f,
```



```
0.2229246944f,    0.1522061825f,    0.06763084233f,    8.073562366e-18f, -0.03335915506f,
-0.03414586186f, -0.01739769801f,   -4.273456581e-18f,  0.008530221879f,  0.008075430058f,
0.003697750857f, 1.226008847e-18f,   -0.001587929321f,  -0.001822523074f
};
```

```
/*
*****
* 函数名: arm_fir_f32_1p
* 功能说明: 调用函数arm_fir_f32_1p实现低通滤波器
* 形参: 无
* 返回值: 无
*****
*/
static void arm_fir_f32_1p(void)
{
    uint32_t i;
    arm_fir_instance_f32 S;
    float32_t *inputF32, *outputF32;

    /* 初始化输入输出缓存指针 */
    inputF32 = &testInput_f32_50Hz_200Hz[0];
    outputF32 = &testOutput[0];

    /* 初始化结构体S */
    arm_fir_init_f32(&S,
                      NUM_TAPS,
                      (float32_t *)&firCoeffs32LP[0],
                      &firStateF32[0],
                      blockSize);

    /* 实现FIR滤波, 这里每次处理1个点 */
    for(i=0; i < numBlocks; i++)
    {
        arm_fir_f32(&S, inputF32 + (i * blockSize), outputF32 + (i * blockSize), blockSize);
    }

    /* 打印滤波后结果 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        printf("%f, %f\r\n", testOutput[i], inputF32[i]);
    }
}
```

运行如上函数可以通过串口打印出函数arm_fir_f32滤波后的波形数据，下面通过Matlab绘制波形来对比Matlab计算的结果和ARM官方库计算的结果。

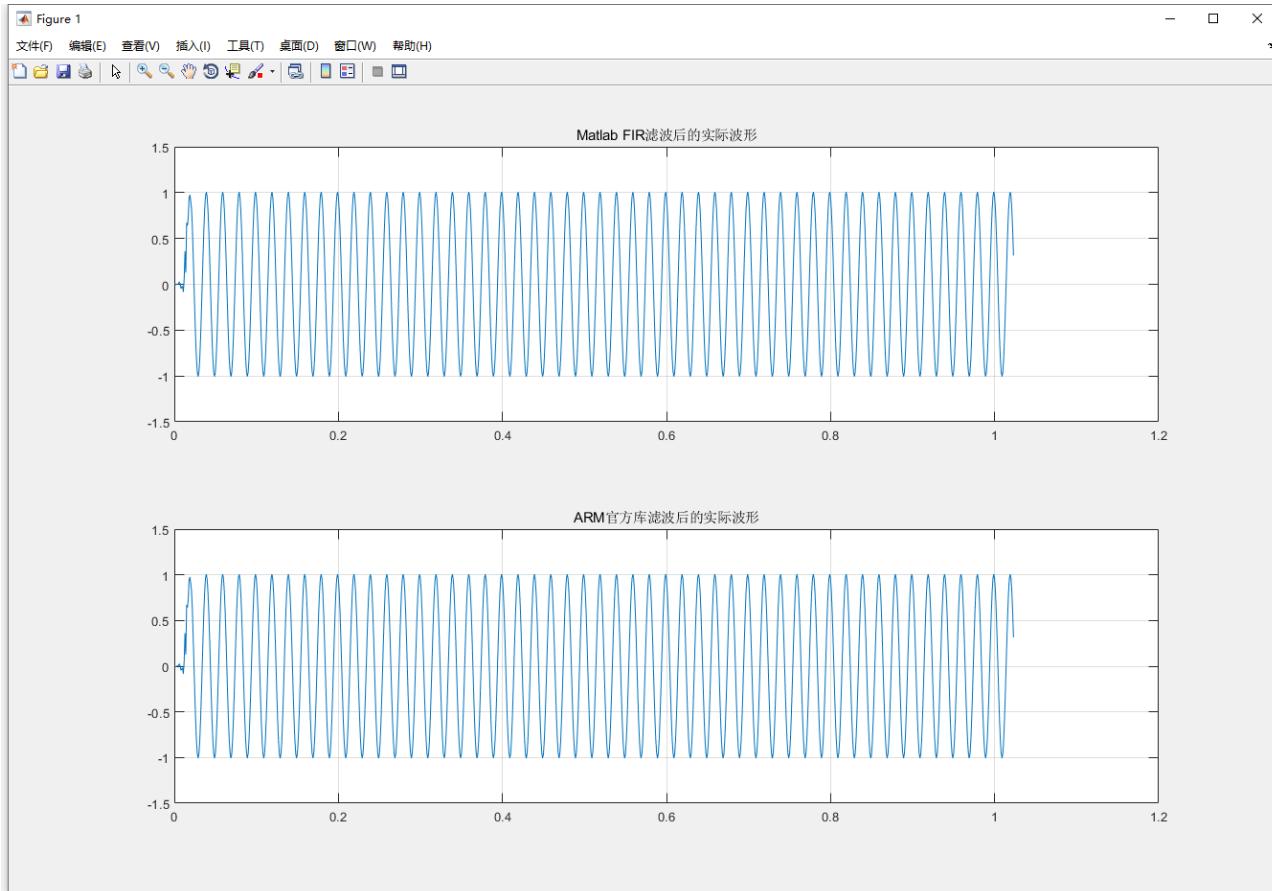
对比前需要先将串口打印出的一组数据加载到 Matlab 中，arm_fir_f32 的计算结果起名 sampledata，加载方法在前面的教程中已经讲解过，这里不做赘述了。Matlab 中运行的代码如下：

```
%%%%%
% FIR带阻滤波器设计
%%%%%
fs=1000;          %设置采样频率 1K
N=1024;           %采样点数
n=0:N-1;
t=n/fs;           %时间序列
f=n*fs/N;          %频率序列

x=sin(2*pi*50*t)+sin(2*pi*200*t);      %50Hz和200Hz正弦波混合
b=fir1(28, [125/500 300/500], 'stop');    %获得滤波器系数, 截止频率125Hz和300, 带阻滤波。
y=filter(b, 1, x);                        %获得滤波后的波形
subplot(211);
plot(t, y);
title('Matlab FIR滤波后的实际波形');
grid on;
```

```
subplot(212);
plot(t, sampledata); %绘制ARM官方库滤波后的波形。
title('ARM官方库滤波后的实际波形');
grid on;
```

Matlab 运行结果如下：



从上面的波形对比来看，matlab 和函数 arm_fir_f32 计算的结果基本是一致的。为了更好的说明滤波效果，下面从频域的角度来说明这个问题，Matlab 上面运行如下代码：

```
%*****
%          FIR带阻滤波器设计
%*****
fs=1000;           %设置采样频率 1K
N=1024;            %采样点数
n=0:N-1;
t=n/fs;             %时间序列
f=n*fs/N;           %频率序列

x=sin(2*pi*50*t)+sin(2*pi*200*t); %50Hz和200Hz正弦波混合
subplot(221);
plot(t, x); %绘制信号x的波形
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;

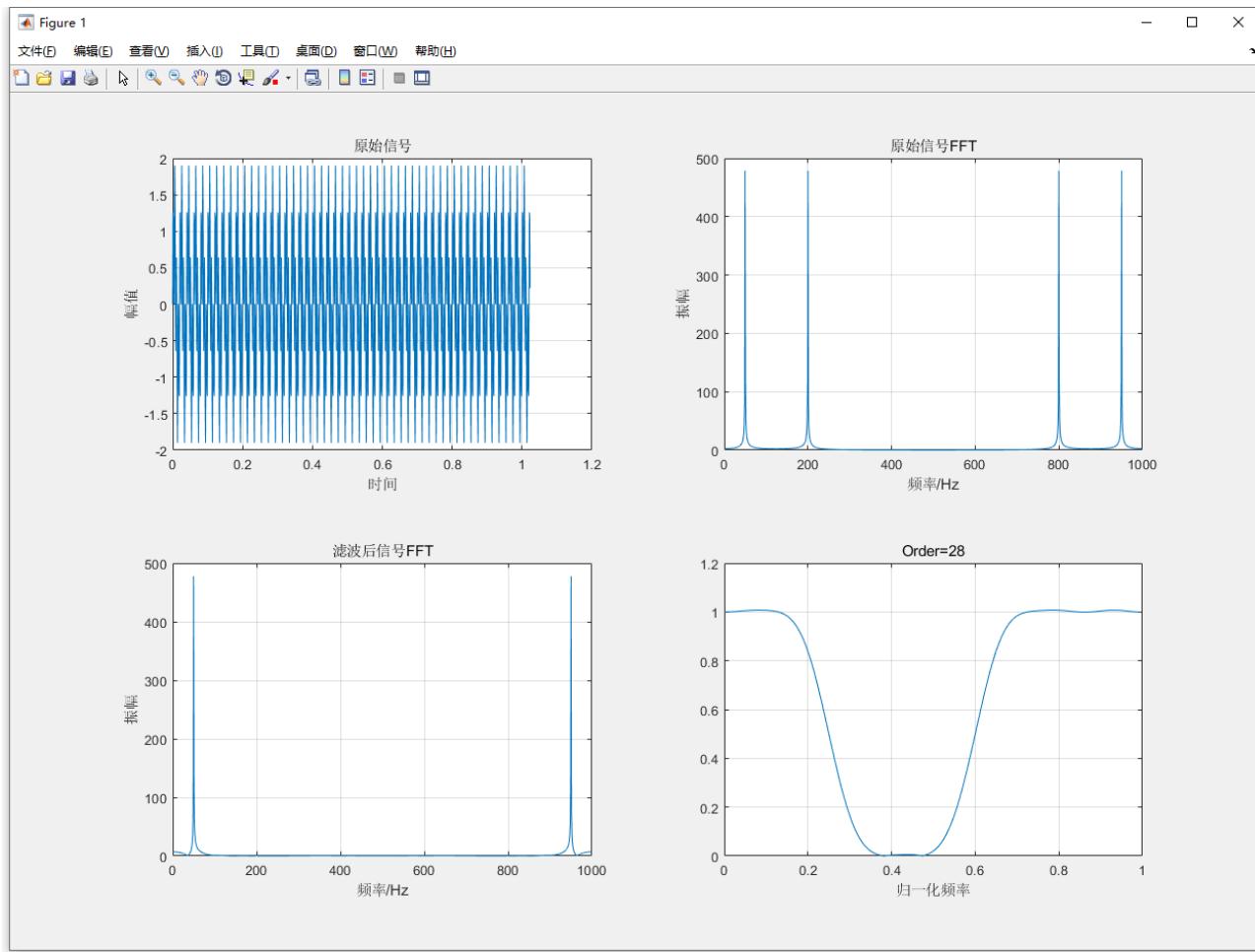
subplot(222);
y=fft(x, N); %对信号x做FFT
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号FFT');
```

```
grid on;

y3=fft(sampledata, N); %经过FIR滤波器后得到的信号做FFT
subplot(223);
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号FFT');
grid on;

b=fir1(28, [125/500 300/500], 'stop'); %获得滤波器系数，截止频率125Hz和300Hz，带阻滤波。
[H, F]=freqz(b, 1, 160); %通过fir1设计的FIR系统的频率响应
subplot(224);
plot(F/pi, abs(H)); %绘制幅频响应
xlabel('归一化频率');
title(['Order=', int2str(28)]);
grid on;
```

Matlab 显示效果如下:



上面波形变换前的 FFT 和变换后 FFT 可以看出，200Hz 的正弦波基本被滤除。

40.6 实验例程说明 (MDK)

配套例子：

V7-228_FIR 带阻滤波器设计(支持逐个数据的实时滤波)



实验目的：

- 学习 FIR 带阻滤波器的实现，支持实时滤波

实验内容：

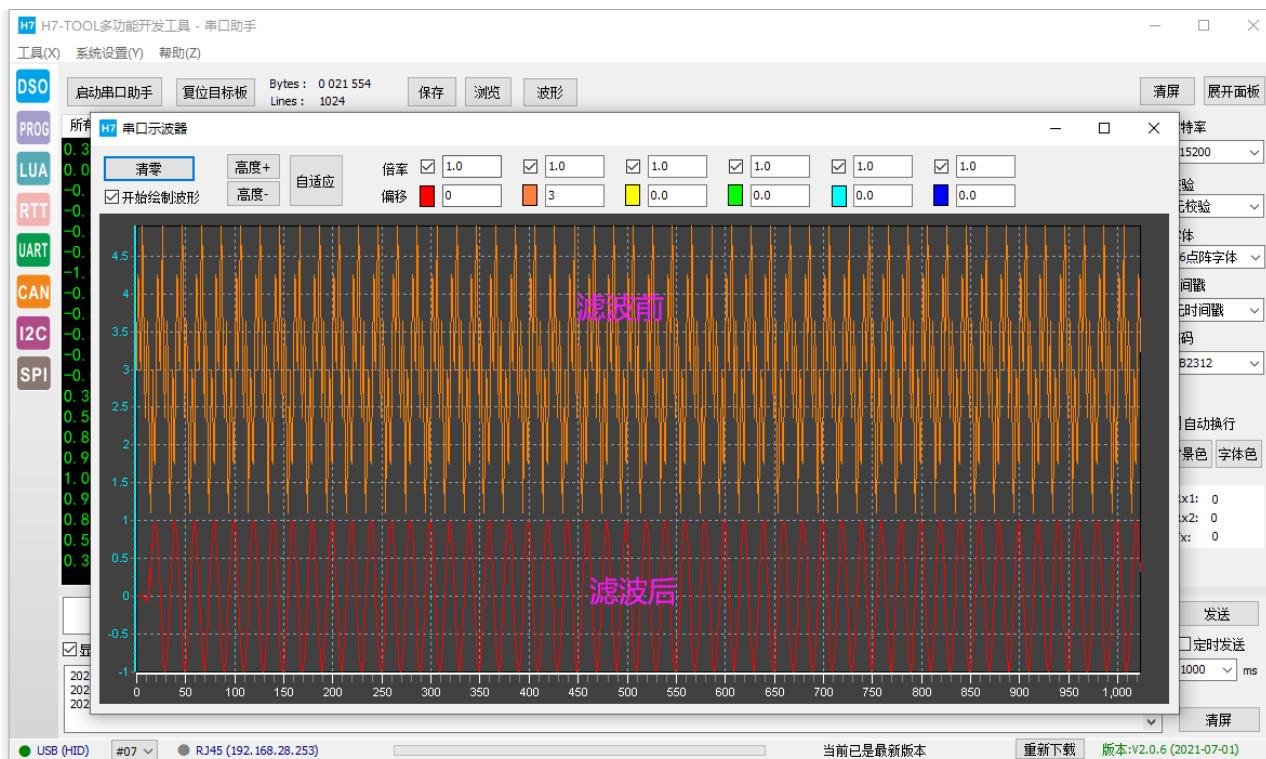
- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

使用 AC6 注意事项

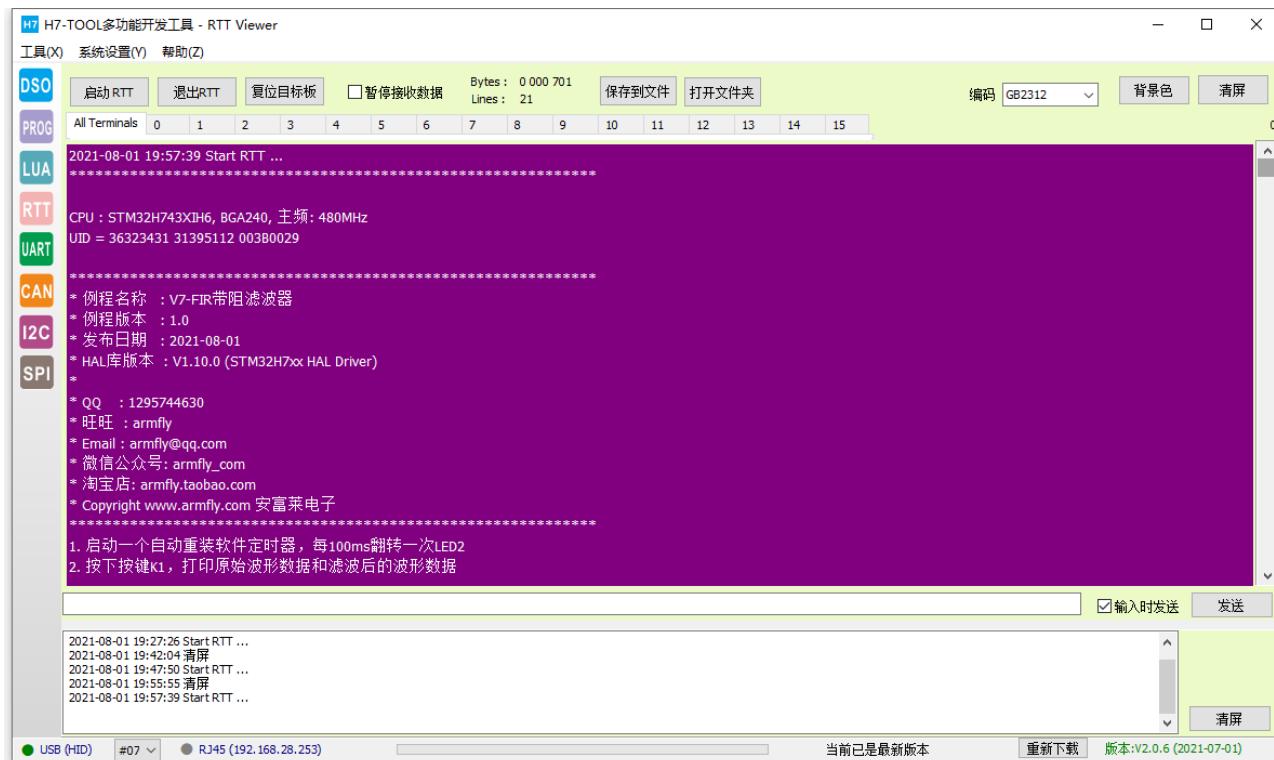
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

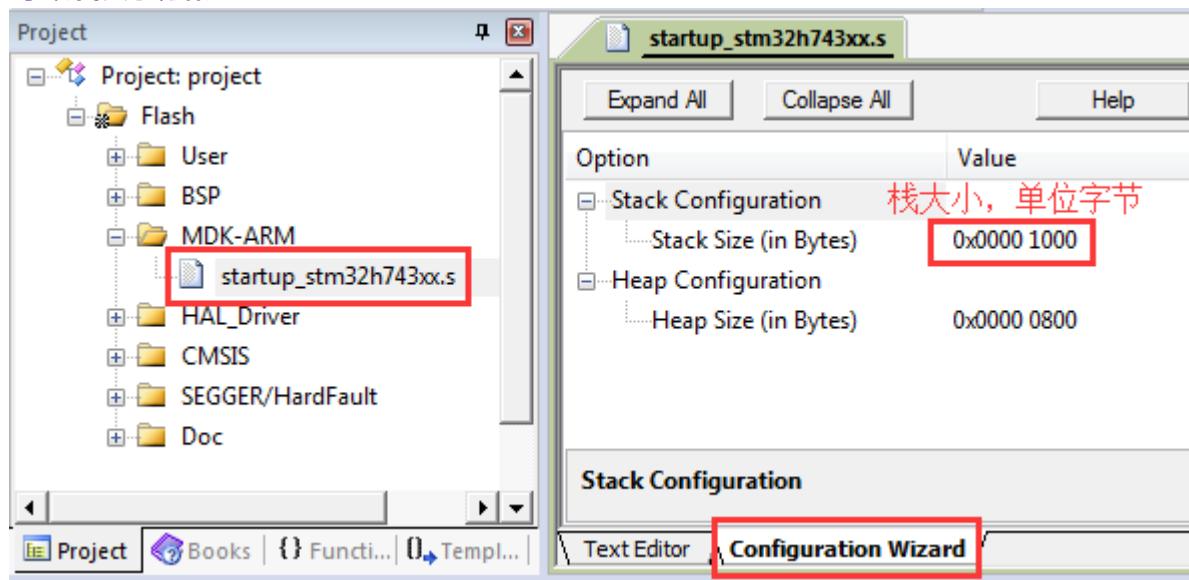


RTT 方式打印信息：

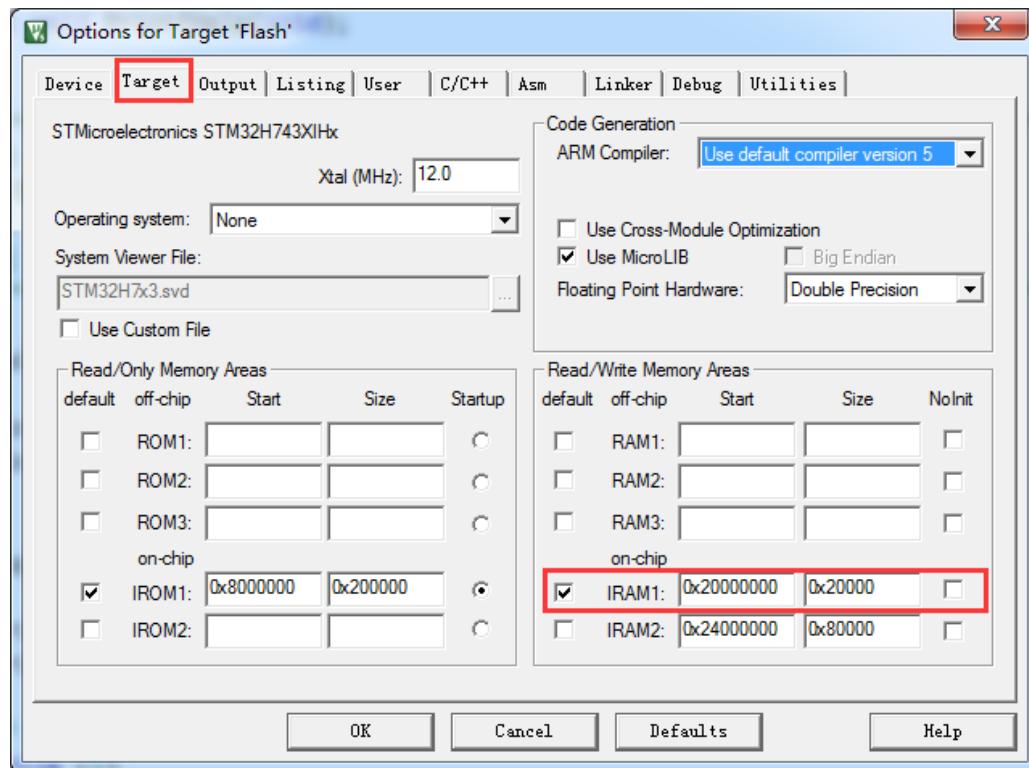


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
    testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                    arm_sin_f32(2*3.1415926f*200*i/1000);
}

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2); /* 翻转 LED2 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_fir_f32_bs();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
}
```

40.7 实验例程说明 (IAR)

配套例子：

V7-228_FIR 带阻滤波器设计(支持逐个数据的实时滤波)

实验目的：

1. 学习 FIR 带阻滤波器的实现，支持实时滤波

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

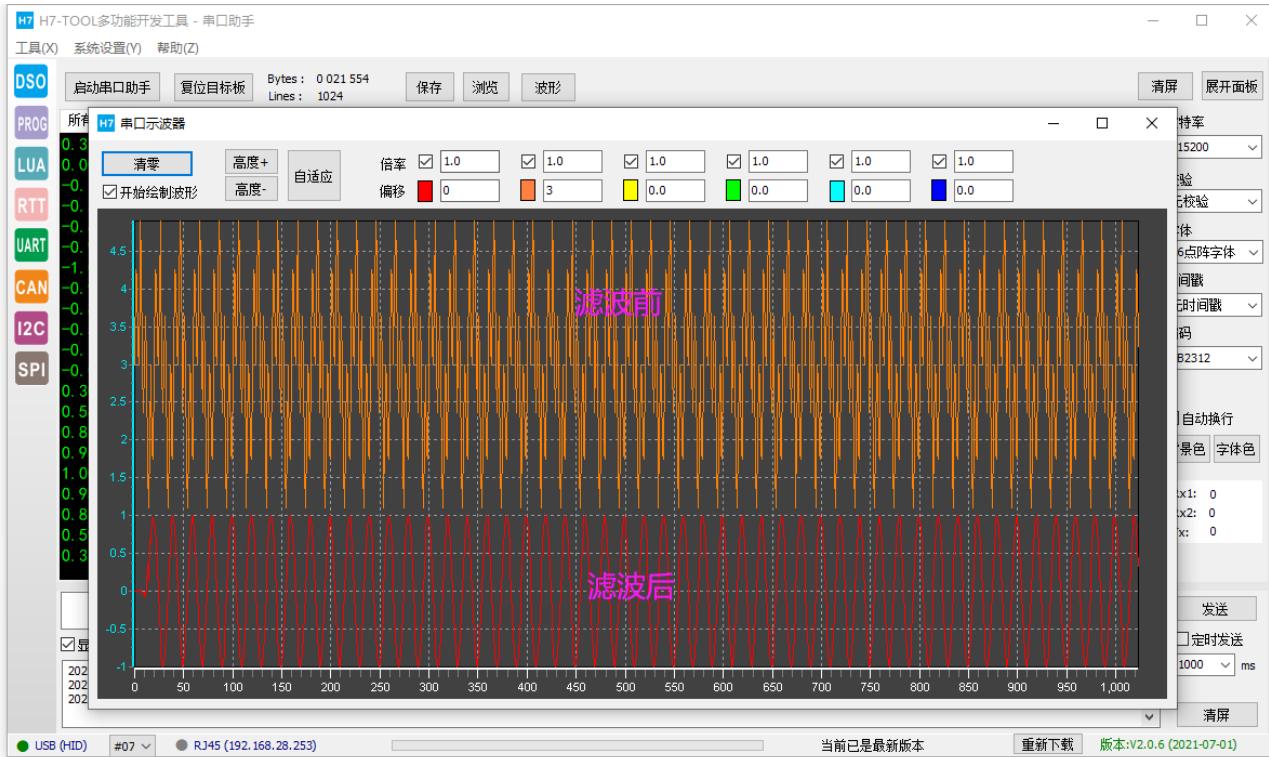
使用 AC6 注意事项



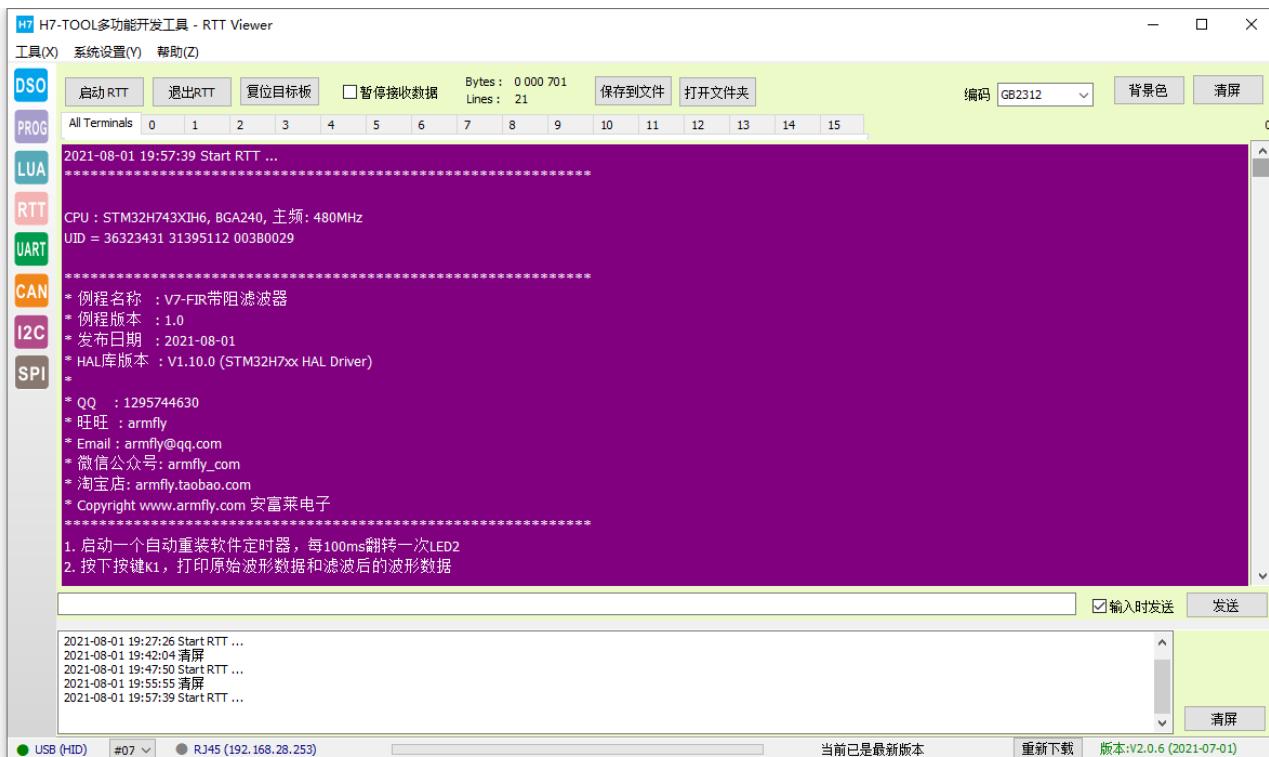
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。



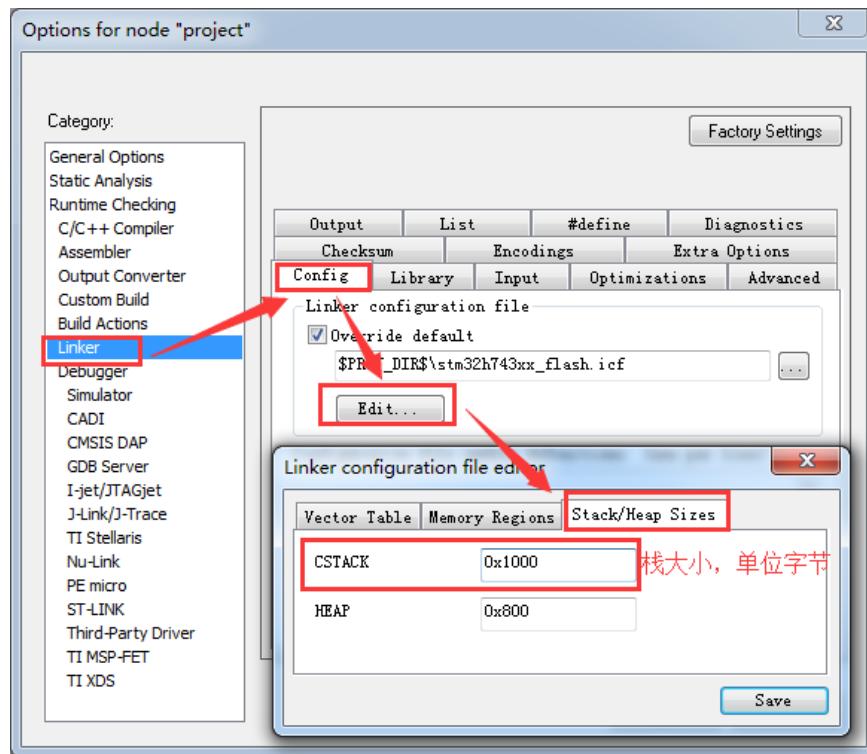
RTT 方式打印信息：



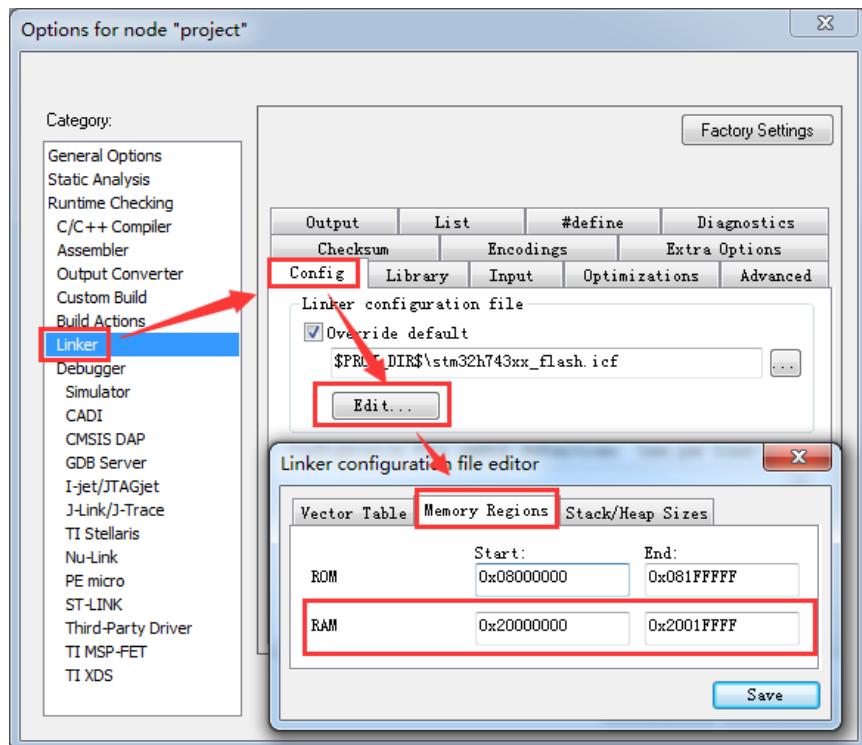
程序设计：



◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现:

```
/*  
*****
```



```
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
        - 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
```



```
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL MPU_ConfigRegion(&MPU_InitStruct);

    /*使能 MPU */
    HAL MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
        testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                      arm_sin_f32(2*3.1415926f*200*i/1000);
    }

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2); /* 翻转 LED2 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_fir_f32_bs();
                    break;
            }
        }
    }
}
```



```
default:  
    /* 其它的键值不处理 */  
    break;  
}  
}  
}
```

40.8 总结

本章节主要讲解了 FIR 滤波器的带阻实现，同时一定要注意线性相位 FIR 滤波器的群延迟问题，详见本教程的第 41 章。

第41章 FIR 滤波器的群延迟 (重要)

本章节为大家介绍 FIR 滤波器的群延迟问题。

41.1 FIR 滤波器介绍

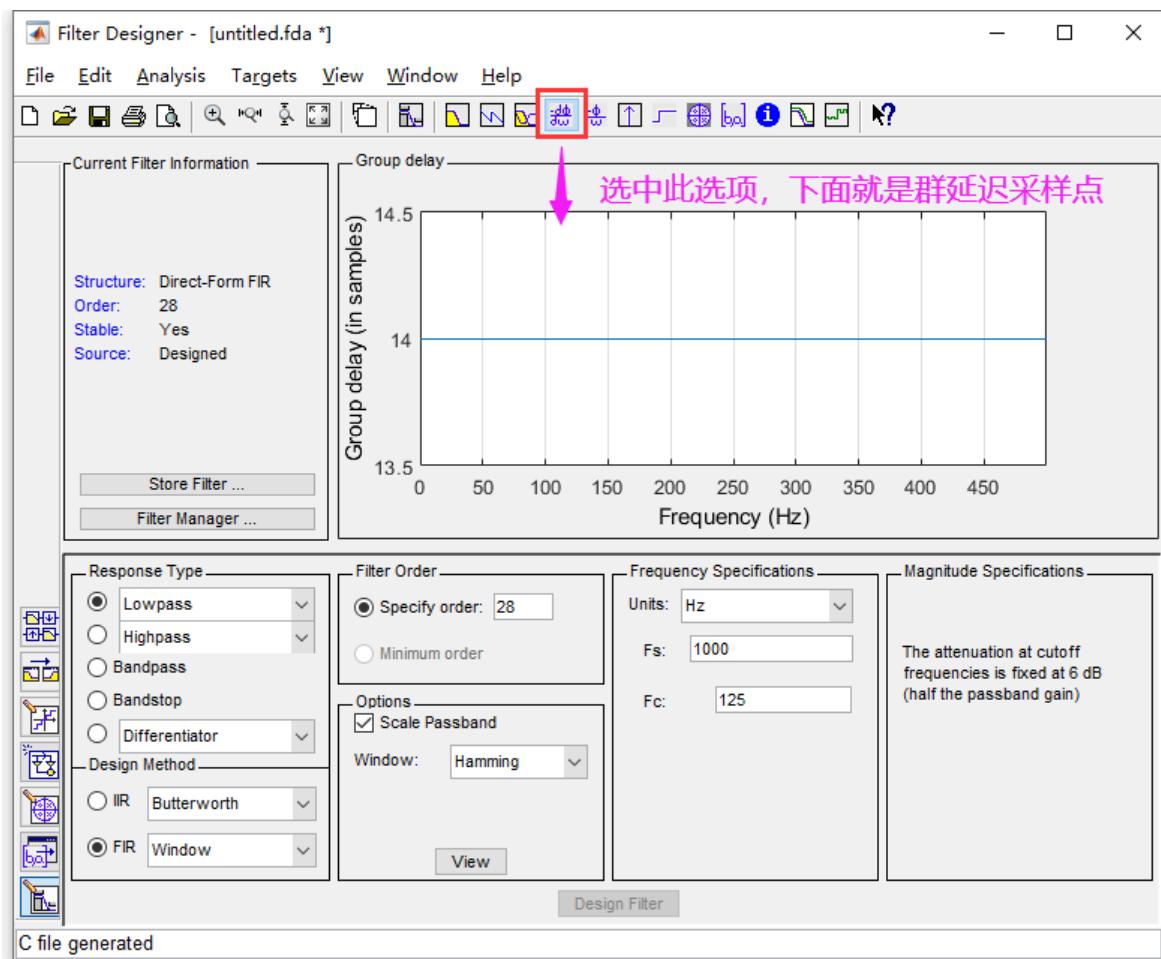
41.2 总结

41.1 FIR 滤波后的群延迟

波形经过 FIR 滤波器后，输出的波形会有一定的延迟。对于线性相位的 FIR，这个群延迟就是一个常数。但是实际应用中这个群延迟是多少呢？关于群延迟的数值，filterDesigner 工具箱会根据用户的配置计算好。

比如前面章节设计的 28 阶 FIR 高通，低通，带通和带阻滤波器的群延迟就是 14，反映在实际的采样值上就是滤波后输出数据的第 15 个才是实际滤波后的波形数据起始点。

下面是群延迟采样点的位置：



细心的读者可能发现全面做低通，高通，带通和带阻滤波后，输出的波形前面几个点感觉有问题，其实就是群延迟造成的。

为了更好的说明这个问题，下面再使用 Matlab 举一个低通和一个高通滤波的例子：信号由 50Hz 正弦波和 200Hz 正弦波组成，采样率 1Kbps，截止频率 125Hz，采样 320 个数据，采用函数 fir1 进行设计，滤波器阶数设置为 28。下面是低通滤波器的 Matlab 代码，将原始信号从第一个点开始显示，而滤波后的信号从群延迟后的第 15 个点开始显示：

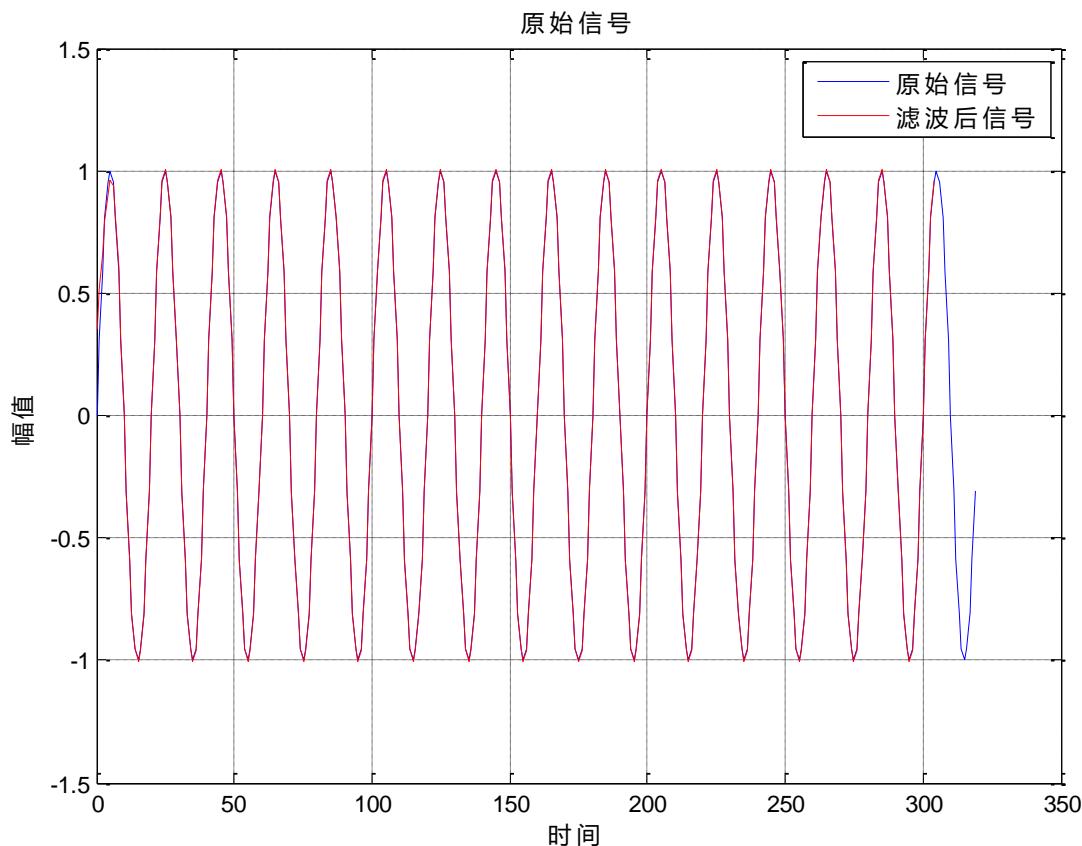
```
fs=1000; %设置采样频率 1K
N=320; %采样点数
n=0:N-1;
t=n/fs; %时间序列
f=n*fs/N; %频率序列

x1=sin(2*pi*50*t);
x2=sin(2*pi*200*t);
x=sin(2*pi*50*t)+sin(2*pi*200*t); %50Hz和200Hz正弦波混合

plot(n, x1, 'b'); %绘制信号x的波形
xlabel('时间');
ylabel('幅值');
title('原始信号和滤波后信号');
hold on;

b=fir1(28, 125/500); %获得滤波器系数，截止频率125Hz.
y=filter(b, 1, x);
plot(n(1:305), y(15:319), 'r');
legend('原始信号','滤波后信号');
grid on;
```

Matlab 的运行结果如下：



可以看出，显示波形基本重合，这个说明 14 个采样点的群延迟是正确的。下面同样使用上面的那个例子实现一个高通滤波器，截止频率是 125Hz，阶数同样设置为 28，将原始信号从第一个点开始显示，而滤波后的信号从群延迟后的第 15 个点开始显示，Matlab 运行代码如下：

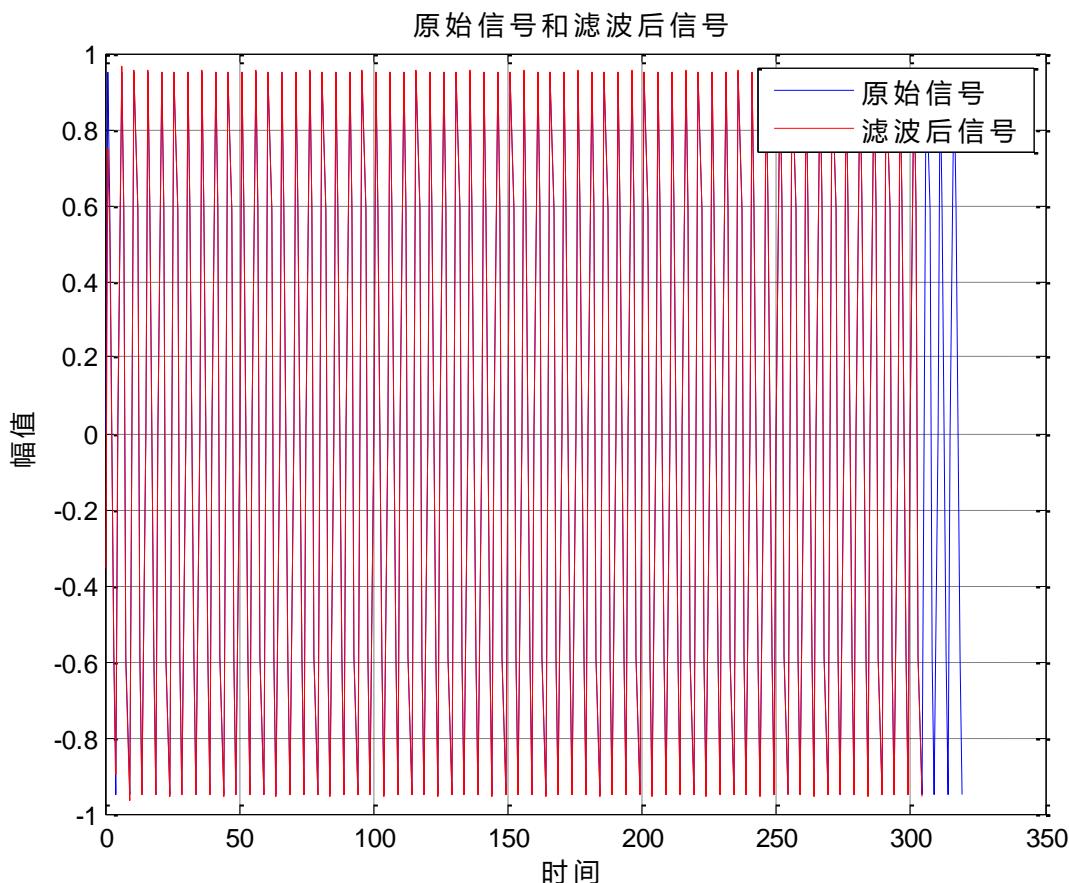
```
fs=1000; %设置采样频率 1K
N=320;
n=0:N-1;
t=n/fs; %时间序列
f=n*fs/N; %频率序列

x1=sin(2*pi*50*t);
x2=sin(2*pi*200*t);
x=sin(2*pi*50*t)+sin(2*pi*200*t); %50Hz和200Hz正弦波混合

plot(n, x2, 'b'); %绘制信号x的波形
xlabel('时间');
ylabel('幅值');
title('原始信号和滤波后信号');
hold on;

b=fir1(28, 125/500, 'high'); %获得滤波器系数，截止频率125Hz.
y=filter(b, 1, x);
plot(n(1:305), y(15:319), 'r');
legend('原始信号', '滤波后信号');
grid on;
```

Matlab 运行结果如下：



可以看出，显示波形基本重合，这个说明 14 个采样点的群延迟也是正确的。大家在使用 FIR 滤波器的时候一定要注意这个问题。



武汉安富莱电子有限公司

WWW.ARMBBS.CN

安富莱 STM32-V7 开发板数字信号处理教程

41.2 总结

本章节介绍的知识点比较重要，首次使用 FIR 容易在这个地方不理解。



第42章 IIR 无限冲击响应滤波器设计

IIR 滤波器涉及到的内容比较多，本章节主要进行了总结性的介绍，以帮助没有数字信号处理基础的读者能够有个整体的认识，有了整体的认识之后再去查阅相关资料可以到达事半功倍的效果。

42.1 基本概念

42.2 IIR 数字滤波器的基本网络结构

42.2 IIR 数字滤波器的设计方法

42.4 总结

42.1 基本概念

IIR 滤波器与 FIR 滤波器相比，具有相位特性差的缺点，但它的的结构简单、运算量小，具有经济、高效的特点，并且可以用较少的阶数获得很高的选择性。因此也得到了广泛应用。

目前 IIR 数字滤波器设计的最通用方法是借助于模拟滤波器的设计方法。模拟滤波器已经有了一套相当成熟的方法，它不但有完整的设计公式，而且还有较为完整的图表供查询，因此，充分利用这些已经有的资源会给数字滤波器的设计代理很大的方便。IIR 数字率变频器的设计步骤是：

1. 按一定的规则将给出的数字滤波器的技术指标转换为模拟低通率变频器的技术指标。
2. 根据转换后的技术指标设计模拟低通滤波器 $G(s)$ 。
3. 再按一定的规则将 $G(s)$ 转换成 $H(z)$ 。

若所设计的数字滤波器时低通的，那么上述设计工作可以结束，若所涉及的是高通，带通或者带阻滤波器，那么还有步骤 4。

4. 将高通、带通或带阻数字滤波器的技术指标先转化为低通模拟滤波器的技术指标，然后按上述步骤 2 设计出低通 $G(s)$ ，再将 $G(s)$ 转换为所需的 $H(z)$ 。

42.2 IIR 数字滤波器的基本网络结构

IIR 滤波器差分方程的一般表达式为：

$$y(n) = \sum_{i=0}^N b_i x(n-i) - \sum_{i=1}^M a_i y(n-i) \quad n \geq 0$$

$x(n)$: 输入序列, $y(n)$: 输出序列, a_i 、 b_i : 滤波器系数, N : 滤波器的阶数。

IIR 滤波器具有无限长度的单位脉冲响应，在结果上存在反馈回路，具有递归性，即 IIR 滤波器的输出不仅与输入有关，而且与过去的输出有关。

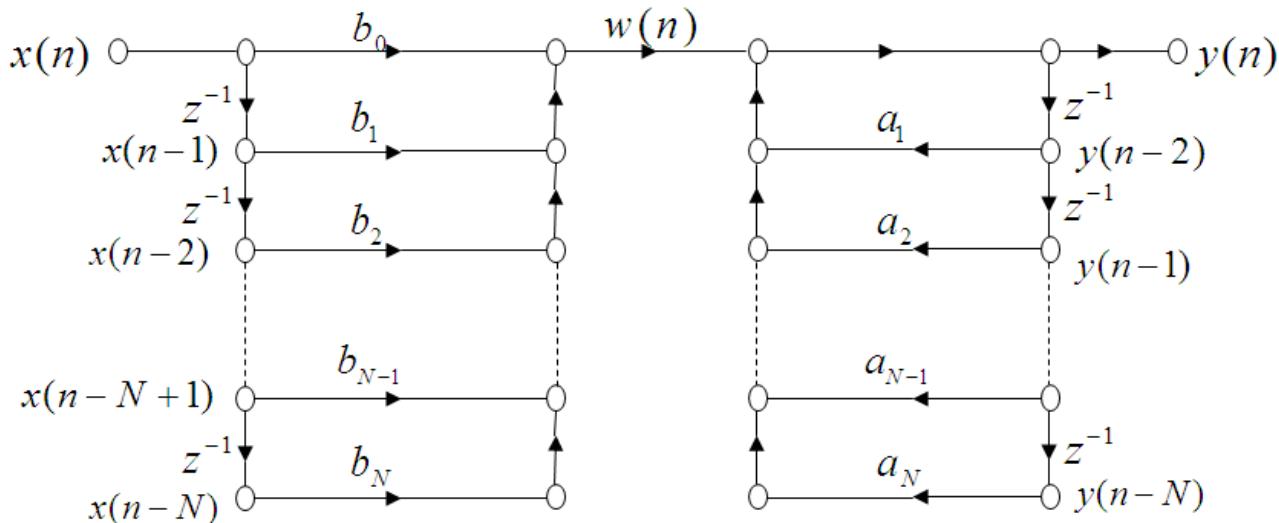
IIR 滤波器具有多种形式，主要有：直接型（也称直接 I 型）、标准型（也称直接 II 型）、变换型、级联型和并联型。

42.2.1 直接 I 型

系统函数：

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^M b_k \cdot z^{-k}}{1 - \sum_{k=1}^N a_k \cdot z^{-k}} = H_1(z) \cdot H_2(z)$$

下面是 N 阶 IIR 滤波器的直接 I 型流程图：



$$\text{其中: } H_1(z) = \sum_{k=0}^M b_k \cdot z^{-k} \quad H_2(z) = \frac{1}{1 - \sum_{k=1}^N a_k \cdot z^{-k}}$$

直接 I 型，先实现 $H_1(z)$ ，再实现 $H_2(z)$ 。

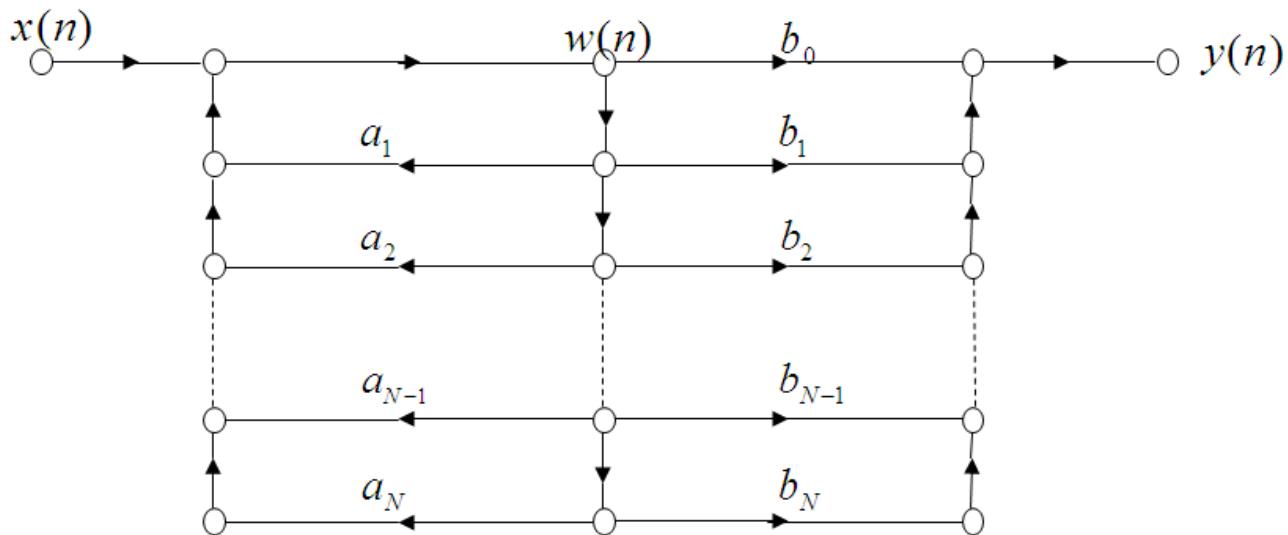
特点：先实现系统函数的零点，再实现极点；需要 $2N$ 个延迟期和 $2N$ 个乘法器。

42.2.2 直接 II 型

当 IIR 数字滤波器是线性非移变系统时，有：

$$H(z) = \frac{Y(z)}{X(z)} = H_2(z) \cdot H_1(z)$$

下面是 N 阶 IIR 滤波器的直接 II 型流程图：



特点：先实现系统函数的极点，再实现零点；需要 N 个延迟器和 $2N$ 个乘法器。

证明：由上图可知：

$$\begin{aligned} w(n) &= x(n) + a_1 w(n-1) + a_2 w(n-2) + \dots + a_{N-1} w(n-N) \\ y(n) &= b_0 x(n) + b_1 w(n-1) + b_2 w(n-2) + \dots + b_{N-1} w(n-N) \end{aligned}$$

上述方程两边同时进行 Z 变换，得：

$$\begin{aligned} W(z) &= X(z) + a_1 z^{-1} W(z) + a_2 z^{-2} W(z) + \dots + a_N z^{-N} W(z) \\ Y(z) &= b_0 W(z) + b_1 z^{-1} W(z) + b_2 z^{-2} W(z) + \dots + b_N z^{-N} W(z) \end{aligned}$$

整理上式得：

$$W(z)(1 - a_1 z^{-1} - a_2 z^{-2} - \dots - a_N z^{-N}) = X(z)$$

$$W(z)(b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_N z^{-N}) = Y(z)$$

所以直 II 型结构的系统函数为：

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^M b_k \cdot z^{-k}}{1 - \sum_{k=1}^N a_k \cdot z^{-k}} = H_2(z) \cdot H_1(z)$$

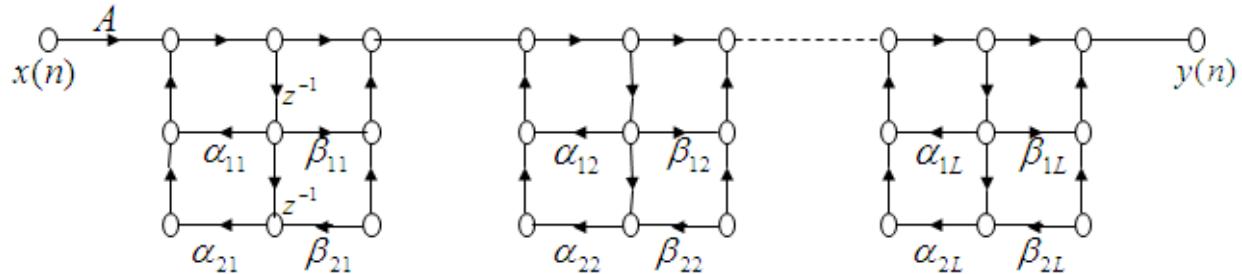
42.2.3 级联型

$$\text{系统函数: } H(z) = A \prod_{k=1}^L \frac{\beta_{0k} + \beta_{1k} z^{-1} + \beta_{2k} z^{-2}}{1 - \alpha_{1k} z^{-1} - \alpha_{2k} z^{-2}}$$

$$= A \prod_{k=1}^L H_k(z)$$

其中: $h(n)$ 为实系数,

$H_k(z) = \frac{\beta_{0k} + \beta_{1k}z^{-1} + \beta_{2k}z^{-2}}{1 - \alpha_{1k}z^{-1} - \alpha_{2k}z^{-2}}$ 称为滤波器的二阶基本节。



基本结构: 二阶基本节, “田字型” 结构。特点:

- 1、二阶基本节搭配灵活, 可调换次序;
- 2、可直接控制零极点;
- 3、存储器最少;
- 4、误差较大。

42.2.4并联型

系统函数: $H(z) = c_0 + \sum_{k=1}^P \frac{A_k}{1 - c_k z^{-1}} + \sum_{k=1}^Q \frac{\gamma_{0k} + \gamma_{1k} z^{-1}}{1 - \alpha_{1k} z^{-1} - \alpha_{2k} z^{-2}}$

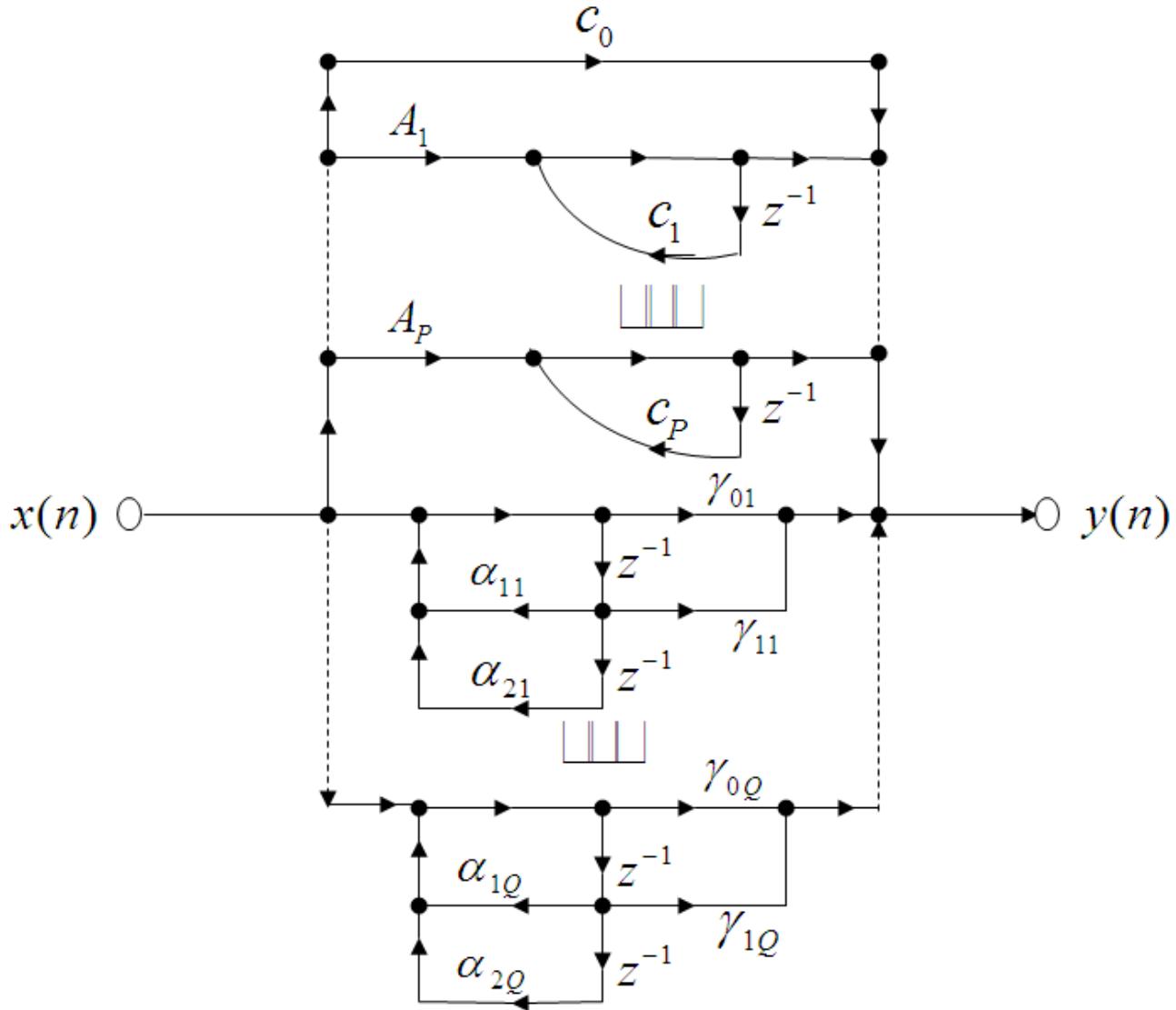
其中: $h(n)$ 为实系数,

$H_k(z) = \frac{\beta_{0k} + \beta_{1k}z^{-1} + \beta_{2k}z^{-2}}{1 - \alpha_{1k}z^{-1} - \alpha_{2k}z^{-2}}$ 称为滤波器的二阶基本节。

基本结构: 一阶基本节和二阶基本节。

- 特点: 1、可单独调整极点, 不能直接控制零点;
- 2、误差小, 各基本节的误差不相互影响;
 - 3、速度快。

IIR 滤波器并联结构图



42.3 IIR 数字滤波器的设计方法

IIR 滤波器的设计方法主要有以下三种：

1. 模拟低通滤波器 $G(s)$ 的设计方法及利用 $G(s)$ 设计模拟高通、带通及带阻滤波器 $H(s)$ 的方法。
2. 用冲击响应不变法设计 IIR 滤波器。
3. 用双线性 z 变换设计 IIR 滤波器

由于这三种方法涉及的内容都比较多，我们这里不做讨论了，大家有兴趣的可以查阅相关书籍资料进行了解，一般数字信号处理方法的书籍对这三种方法都有详细的讲解。

42.4 总结

本期教程主要对 IIR 滤波进行了总结性的介绍，每个知识点并没有进行详细的介绍，如果将这些知识



武汉安富莱电子有限公司

WWW.ARMBBS.CN

安富莱 STM32-V7 开发板数字信号处理教程

点也进行展开的话将占用大量的篇幅，而且大家不容易看懂。尽管这样，还是希望有兴趣的读者去查阅相关的书籍进行深入的了解，只有你对这些理论有了深入的理解，你的实际应用才能事半功倍。还是那句经常说的话：理论高度决定实践高度。



第43章 IIR 滤波器的 Matlab 设计

本章节讲解 IIR 滤波器的 Matlab 设计，主要包括巴特沃斯滤波器，切比雪夫 I 型和 II 型滤波器以及椭圆滤波器。

43.1 巴特沃斯滤波器的设计

43.2 切比雪夫滤波器的设计

43.3 椭圆滤波器的设计

43.4 总结

43.1 巴特沃斯滤波器的设计

43.1.1 butter 函数

功能：用于设计 Butterworth (巴特沃斯) 滤波器

语法： $[b, a] = \text{butter}(n, W_n)$;

说明：butter 函数可以设计低通、带通、高通和带阻数字滤波器，其特性可以使通带内的幅度响应最大限度地平坦，但会损失截止频率处的下降斜度，使幅度响应衰减较慢。

- $[b, a] = \text{butter}(n, W_n)$ 可以设计截止频率为 W_n 的 n 阶低通 butterworth 滤波器，其中截止频率 W_n 应满足 $0 \leq W_n \leq 1$ ， $W_n=1$ 相当于 $0.5f_s$ (采样频率)。当 $W_n = [W_1 W_2]$ 时，butter 函数产生一个 $2n$ 阶的数字带通滤波器，其通带为 $W_1 < W < W_2$ 。
- $[b, a] = \text{butter}(n, W_n, 'ftype')$ 可以设计高通或带阻滤波器。当 $\text{ftype}=\text{high}$ 时，可设计截止频率为 W_n 的高通滤波器；当 $\text{ftype}=\text{stop}$ 时，可设计带阻滤波器，此时 $W_n = [W_1 W_2]$ ，阻带为 $W_1 < W < W_2$ 。
使用 butter 函数设计滤波器，可以使通带内的幅度响应最大地平坦，但会损失截止频率处的下降斜度。因此，butter 函数主要用于设计通带平坦的数字滤波器。

43.1.2 buttord 函数

功能：用来选择 Butterworth 滤波器的阶数。

语法： $[n, W_n] = \text{buttord} (W_p, W_s, R_p, R_s)$;



说明：buttord 函数可以在给定滤波器性能的情况下，选择 Butterworth 数字滤波器的最小阶数，其中 W_p 和 W_s 分别是通带和阻带的截止频率，其值为 $0 \leq W_p$ (或 W_s) ≤ 1 ，当该值为 1 时表示 $0.5f_s$ (采样率)。 R_p 和 R_s 分别是通带和阻带区的波纹系数和衰减系数。

$[n, W_n] = \text{buttord} (W_p, W_s, R_p, R_s)$ 可以得到高通、带通和带阻滤波器的最小阶数 n 。

当 $W_p > W_s$ 时，为高通滤波器；当 W_p, W_s 为二元矢量时，若 $W_p < W_s$ ，则为带通和带阻滤波器，此时 W_n 也为二元矢量。

利用 buttord 函数可得到 Butterworth 数字滤波器的最小阶数 n ，并使通带 $(0, W_p)$ 内的波纹系数小于 R_p ，阻带 $(W_s, 1)$ 内衰减系数大于 R_s 。buttord 函数还可以得到截止频率 W_n ，再利用 butter 函数可产生满足指定性能的滤波器。

使用 butter 函数设计数字滤波器，可以使通带内的幅度响应最大限度地平坦，但在截止频率附件幅度响应衰减慢。如果期望幅度响应下降斜度大，衰减快，可使用 Elliptic(椭圆)或 Chebyshev (切比雪夫) 滤波器。

43.1.3 巴特沃斯低通滤波器设计

下面我们通过一个实例来讲解巴特沃斯低通滤波器的设计。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，将 200Hz 的正弦波当做噪声滤掉，下面通过函数 butter 设计一组低通滤波器系数，其阶数是 2，截止频率为 0.25 (也就是 125Hz)，采样率 1Kbps。Matlab 运行代码如下：

```
fs=1000;          %设置采样频率 1k
N=1024;          %采样点数
n=0:N-1;
t=0:1/fs:1-1/fs;    %时间序列
f=n*fs/N;          %频率序列

x1=sin(2*pi*50*t);    %信号
x2=sin(2*pi*200*t);    %噪声
x=x1+x2;            %信号混合

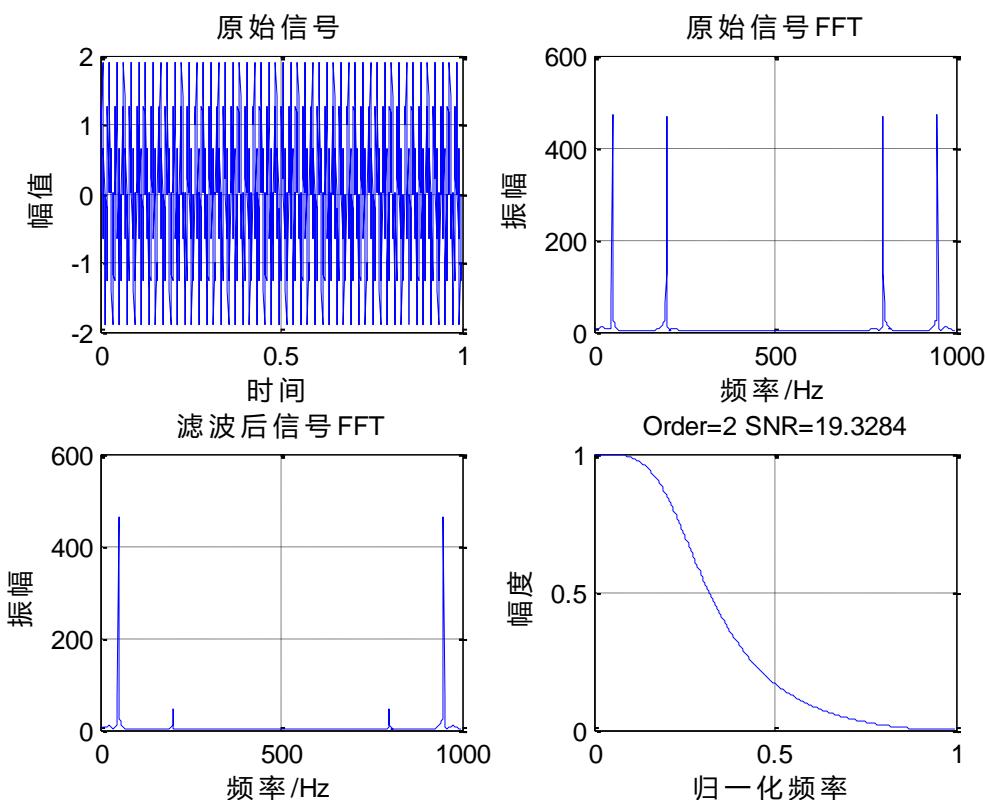
subplot(221);
plot(t, x);          %绘制原始信号
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;

subplot(222);
y=fft(x, N);        %绘制原始信号的幅频响应
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

subplot(223);
Wc=2*125/fs;        %设置截止频率 125Hz
[b, a]=butter(2, Wc); %获取 2 阶 IIR 滤波系数
```

```
% y2=filter(b, a, x);  
y2=filtfilt(b, a, x); %计算滤波后的波形 y2  
y3=fft(y2, N); %滤波后波形的幅频响应  
plot(f, abs(y3));  
xlabel('频率/Hz');  
ylabel('振幅');  
title('滤波后信号 FFT');  
grid on;  
  
[H, F]=freqz(b, a, 512);  
subplot(224);  
plot(F/pi, abs(H));  
xlabel('归一化频率'); %绘制绝对幅频响应  
ylabel('幅度');  
Ps=sum(x1.^2); %信号的总功率  
Pu=sum((y2-x1).^2); %剩余噪声的功率  
SNR=10*log10(Ps/Pu); %信噪比  
title(['Order=' int2str(2), ' SNR=' num2str(SNR)]);  
grid on;
```

运算 Matlab 结果如下：



从滤波的效果来看，2 阶的 IIR 滤波器能够达到将近 20 的信噪比，比使用 FIR 需要更少的阶数。



43.1.4 巴特沃斯高通滤波器设计

下面我们通过一个实例来讲解巴特沃斯高通滤波器的设计。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，将 50Hz 的正弦波当做噪声滤掉，下面通过函数 butter 设计一组高通滤波器系数，其阶数是 2，截止频率为 0.25（也就是 125Hz），采样率 1Kbps。Matlab 运行代码如下：

```
fs=1000; %设置采样频率 1k
N=1024; %采样点数
n=0:N-1;
t=0:1/fs:1-1/fs; %时间序列
f=n*fs/N; %频率序列

x1=sin(2*pi*50*t); %噪声
x2=sin(2*pi*200*t); %信号
x=x1+x2; %信号混合

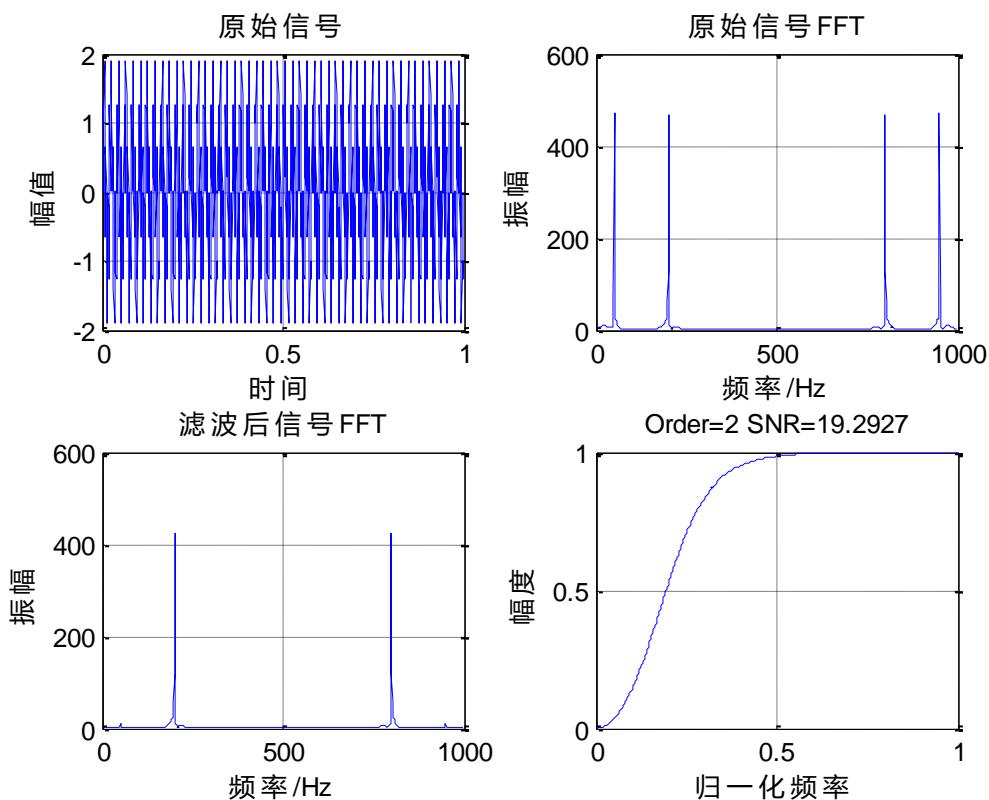
subplot(221);
plot(t, x); %绘制原始信号
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;

subplot(222);
y=fft(x, N); %绘制原始信号的幅频响应
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

subplot(223);
Wc=2*125/fs; %设置截止频率 125Hz
[b, a]=butter(2, Wc, 'high'); %获取 2 阶 IIR 滤波系数
% y2=filter(b, a, x);
y2=filtfilt(b, a, x); %计算滤波后的波形 y2
y3=fft(y2, N); %滤波后波形的幅频响应
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

[H, F]=freqz(b, a, 512);
subplot(224);
plot(F/pi, abs(H)); %绘制绝对幅频响应
xlabel('归一化频率');
ylabel('幅度');
Ps=sum(x2.^2); %信号的总功率
Pu=sum((y2-x2).^2); %剩余噪声的功率
SNR=10*log10(Ps/Pu); %信噪比
title(['Order=' int2str(2), ' SNR=' num2str(SNR)]);
grid on;
```

Matlab 运行结果如下：



从滤波的效果来看，2阶的IIR滤波器效果还是比较好的。

43.1.5 巴特沃斯带通滤波器设计

下面我们通过一个实例来讲解巴特沃斯带通滤波器的设计。原始信号是由50Hz正弦波和200Hz的正弦波组成，将50Hz的正弦波当做噪声滤掉，下面通过函数butter设计一组带通滤波器系数，其阶数是2，通带为125Hz到300Hz，采样率1Kbps。Matlab运行代码如下：

```

fs=1000; %设置采样频率 1k
N=1024; %采样点数
n=0:N-1;
t=0:1/fs:1-1/fs; %时间序列
f=n*fs/N; %频率序列

x1=sin(2*pi*50*t); %噪声
x2=sin(2*pi*200*t); %信号
x=x1+x2; %信号混合

subplot(221);
plot(t,x); %绘制原始信号
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;

subplot(222);

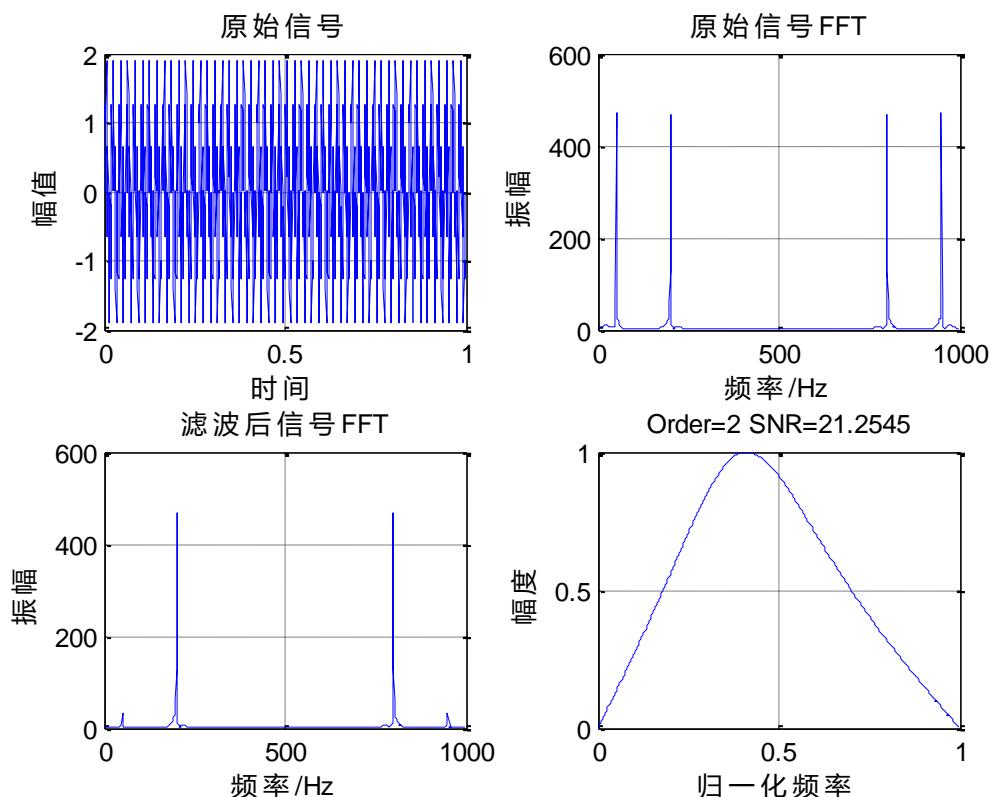
```

```
y=fft(x, N); %绘制原始信号的幅频响应
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

subplot(223);
Wn=[125*2 300*2]/fs; %设置通带 125Hz 到 300Hz
[b, a]=butter(1, Wn); %注意第一个参数虽然是 1, 但生成的却是 2 阶 IIR 滤波器系数
% y2=filter(b, a, x);
y2=filtfilt(b, a, x); %计算滤波后的波形 y2
y3=fft(y2, N); %滤波后波形的幅频响应
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

[H, F]=freqz(b, a, 512);
subplot(224);
plot(F/pi, abs(H)); %绘制绝对幅频响应
xlabel('归一化频率');
ylabel('幅度');
Ps=sum(x2.^2); %信号的总功率
Pu=sum((y2-x2).^2); %剩余噪声的功率
SNR=10*log10(Ps/Pu); %信噪比
title(['Order=', int2str(2), ' SNR=', num2str(SNR)]);
grid on;
```

Matlab 的计算结果如下：





从滤波的效果来看，2阶的 IIR 滤波器效果还是比较好的。

43.1.6 巴特沃斯带阻滤波器设计

下面我们通过一个实例来讲解巴特沃斯带阻滤波器的设计。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，将 200Hz 的正弦波当做噪声滤掉，下面通过函数 butter 设计一组带阻滤波器系数，其阶数是 2，阻带为 125Hz 到 300Hz，采样率 1Kbps。Matlab 运行代码如下：

```
fs=1000; %设置采样频率 1k
N=1024; %采样点数
n=0:N-1;
t=0:1/fs:1-1/fs; %时间序列
f=n*fs/N; %频率序列

x1=sin(2*pi*50*t); %信号
x2=sin(2*pi*200*t); %噪声
x=x1+x2; %信号混合

subplot(221);
plot(t,x); %绘制原始信号
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;

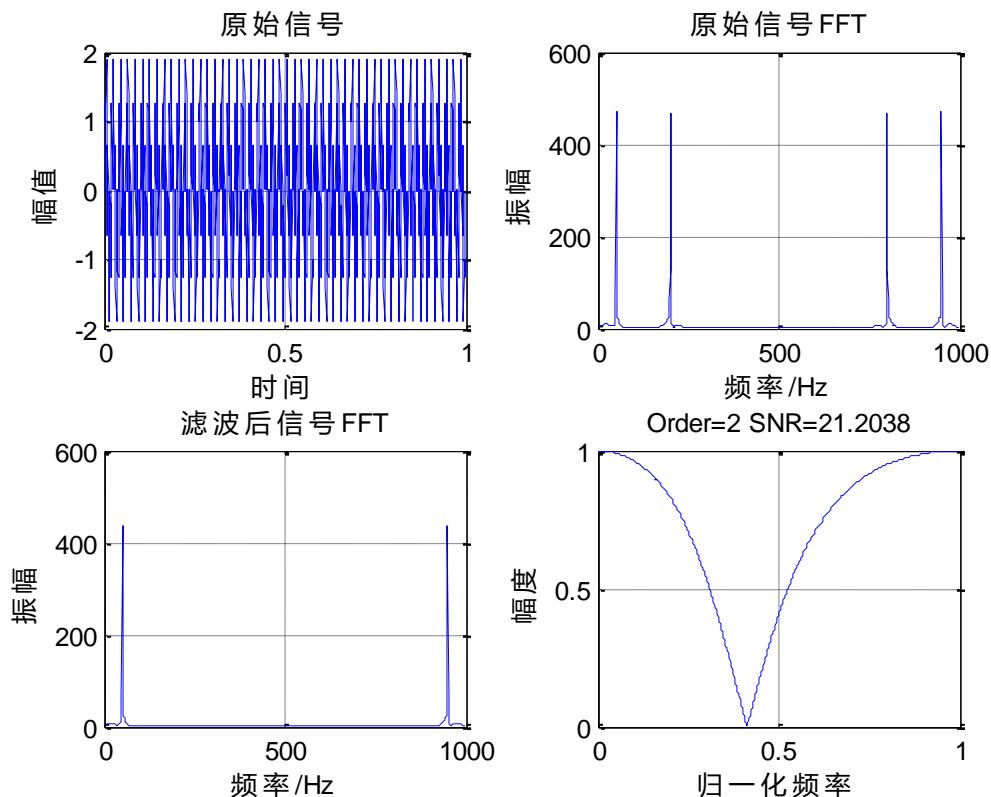
subplot(222);
y=fft(x,N); %绘制原始信号的幅频响应
plot(f,abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

subplot(223);
Wn=[125*2 300*2]/fs; %设置阻带 125Hz 到 300Hz
[b,a]=butter(1,Wn, 'stop'); %注意第一个参数虽然是 1，但生成的却是 2 阶 IIR 滤波器系数
% y2=filter(b,a,x);
y2=filtfilt(b,a,x); %计算滤波后的波形 y2
y3=fft(y2,N); %滤波后波形的幅频响应
plot(f,abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

[H,F]=freqz(b,a,512);
subplot(224);
plot(F/pi,abs(H)); %绘制绝对幅频响应
xlabel('归一化频率');
ylabel('幅度');
Ps=sum(x1.^2); %信号的总功率
Pu=sum((y2-x1).^2); %剩余噪声的功率
SNR=10*log10(Ps/Pu); %信噪比
```

```
title(['Order=' int2str(2), ' SNR=' num2str(SNR)]) ;  
grid on;
```

Matlab 运行结果如下：



从滤波的效果来看，2 阶的 IIR 滤波器效果还是比较好的。

43.2 切比雪夫滤波器的设计

切比雪夫 (Chebyshev) 滤波器分为 Chebyshev I 型和 Chebyshev II 型，分别具有通带等纹波和阻带等纹波性能。

43.2.1 cheby1 函数

功能：用来设计 Chebyshev (切比雪夫) I 型滤波器（通带等纹波）。

语法：[b,a] = cheby1(n, R_p, W_n);

[b,a] = cheby1(n, R_p, W_n, 'ftype');

说明：cheby1 函数可以设计低通，带通，高通和带阻 Chebyshev I 型数字滤波器，其通带内为等纹波，阻带内为单调。Chebyshev I 型滤波器的下降斜度比 Chebyshev II 型大，但其代价是在通带内纹波较大。

[b,a] = cheby1(n, R_p, W_n); 可以设计 n 阶低通 Chebyshev I 型数字滤波器，其中 R_p 用来确定通带内的纹波，W_n 为该滤波器的截止频率。



当 $W_n = [W_1, W_2]$ 时, cheby1 函数可产生一个 $2n$ 的数字带通滤波器, 其通带为 $W_1 < W < W_2$ 。

$[b,a] = \text{cheby1}(n, R_p, W_n, 'ftype')$; 可用来设计 n 阶高通或带阻滤波器, 其中 R_p 和 W_n 同上, $ftype$ 的定义与 butter 相同。

43.2.2 cheby1ord 函数

功能: 用来选择 Chebyshev I 型滤波器的阶数。

语法: $[n, W_n] = \text{cheb1ord} (W_p, W_s, R_p, R_s)$;

说明: cheb1ord 函数可以在给定滤波器性能的情况下, 选择 Chebyshev I 型数字滤波器的最小阶数, 其中 W_p 和 W_s 分别是通带和阻带的截止频率, 其值为 $0 \leq W_p$ (或 W_s) ≤ 1 。 R_p 和 R_s 分别是通带和阻带区的波纹系数。

$[n, W_n] = \text{cheb1ord} (W_p, W_s, R_p, R_s)$; 可以得到低通、高通、带通和带阻滤波器的最小阶数。

利用 cheblord 函数, 除了可以得到 Chebyshev I 型数字滤波器的最小阶数 n 外, 还可以得到截止频率 W_n , 再利用 cheby1 函数可产生满足指定性能的滤波器, 使滤波器通带($0, W_p$)内的纹波系数小于 R_p , 阻带($W_s, 1$)内衰减系数大于 R_s

43.2.3 cheby2 函数

功能: 用来设计 Chebyshev (切比雪夫) I 型滤波器 (通带等纹波)。

语法: $[b,a] = \text{cheby1}(n, R_p, W_n)$;

$[b,a] = \text{cheby1}(n, R_p, W_n, 'ftype')$;

说明: cheby2 函数与 cheby1 函数基本相同, 只是用 cheby2 函数所设计的滤波器, 其通带内为单调的, 阻带内为等波纹, 由 R_s 指定阻带内的波纹。

cheby2 函数可以设计低通, 带通, 高通和带阻 Chebyshev II 型数字滤波器。

43.2.4 cheby2ord 函数

功能: 用来选择 Chebyshev II 型滤波器的阶数。

语法: $[n, W_n] = \text{cheb2ord} (W_p, W_s, R_p, R_s)$;

说明: cheb2ord 函数与 cheb2 函数类似, 可以利用该函数确定 Chebyshev II 型数字滤波器的最小阶数 n 和截止频率 W_n 。

cheb2ord 函数和 cheb2 函数配合使用, 可设计出最低阶数的 Chebyshev II 型数字滤波器。



43.2.5 切比雪夫 I 型低通滤波器设计

下面我们通过一个实例来讲解切比雪夫 I 型低通滤波器的设计。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，将 200Hz 的正弦波当做噪声滤掉，下面通过函数 cheby1 设计一组低通滤波器系数，其阶数是 4，截止频率为 0.25（也就是 125Hz），采样率 1Kbps，通带波纹 1db。Matlab 运行代码如下：

```
fs=1000; %设置采样频率 1k
N=1024; %采样点数
n=0:N-1;
t=0:1/fs:1-1/fs; %时间序列
f=n*fs/N; %频率序列

x1=sin(2*pi*50*t); %信号
x2=sin(2*pi*200*t); %噪声
x=x1+x2; %信号混合

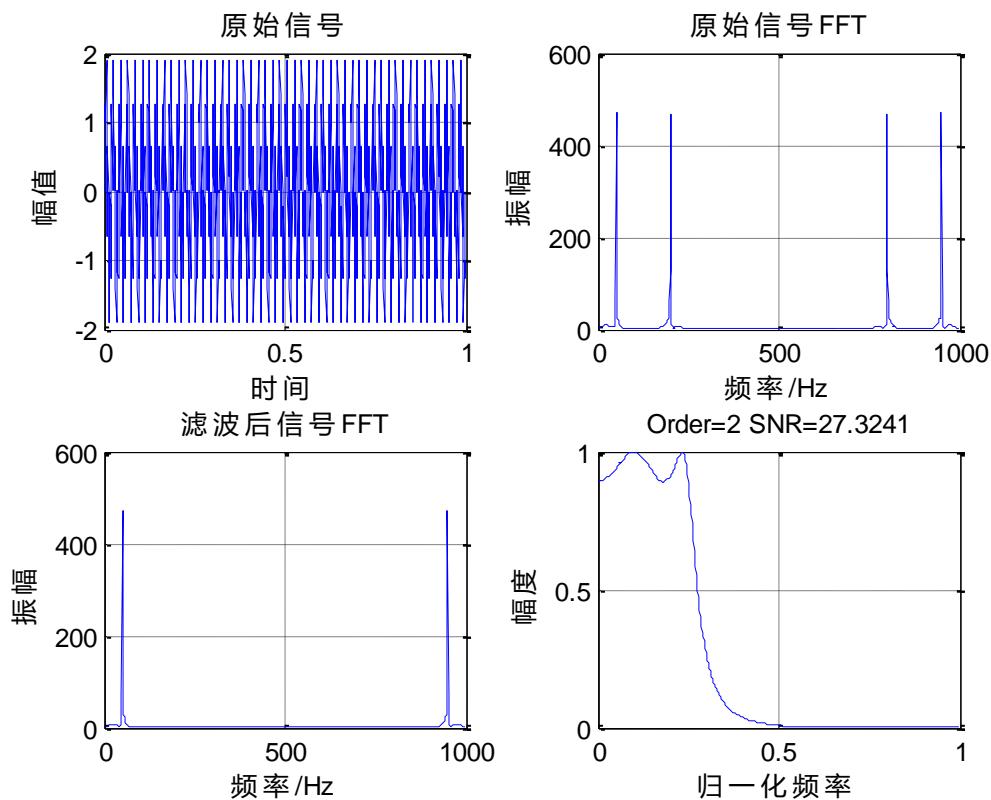
subplot(221);
plot(t, x); %绘制原始信号
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;

subplot(222);
y=fft(x, N); %绘制原始信号的幅频响应
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

subplot(223);
Wc=2*125/fs; %设置截止频率 125Hz
[b, a]=cheby1(4, 3, Wc); %获取 2 阶 IIR 滤波系数
% y2=filter(b, a, x);
y2=filtfilt(b, a, x); %计算滤波后的波形 y2
y3=fft(y2, N); %滤波后波形的幅频响应
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

[H, F]=freqz(b, a, 512);
subplot(224);
plot(F/pi, abs(H)); %绘制绝对幅频响应
xlabel('归一化频率');
ylabel('幅度');
Ps=sum(x1.^2); %信号的总功率
Pu=sum((y2-x1).^2); %剩余噪声的功率
SNR=10*log10(Ps/Pu); %信噪比
title(['Order=' int2str(2), ' SNR=' num2str(SNR)]);
grid on;
```

Matlab 计算结果如下：



从滤波的效果来看，4 阶的切比雪夫 I 型滤波效果还是比较好的。

43.2.6 切比雪夫 I 型高通滤波器设计

下面我们通过一个实例来讲解切比雪夫 I 型高通滤波器的设计。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，将 50Hz 的正弦波当做噪声滤掉，下面通过函数 cheby1 设计一组高通滤波器系数，其阶数是 2，截止频率为 0.25（也就是 125Hz），采样率 1Kbps，通带波纹 1db。Matlab 运行代码如下：

```
fs=1000;          %设置采样频率 1k
N=1024;          %采样点数
n=0:N-1;
t=0:1/fs:1-1/fs;    %时间序列
f=n*fs/N;          %频率序列

x1=sin(2*pi*50*t);    %噪声
x2=sin(2*pi*200*t);    %信号
x=x1+x2;            %信号混合

subplot(221);
plot(t,x);          %绘制原始信号
xlabel('时间');
ylabel('幅值');
title('原始信号');
```



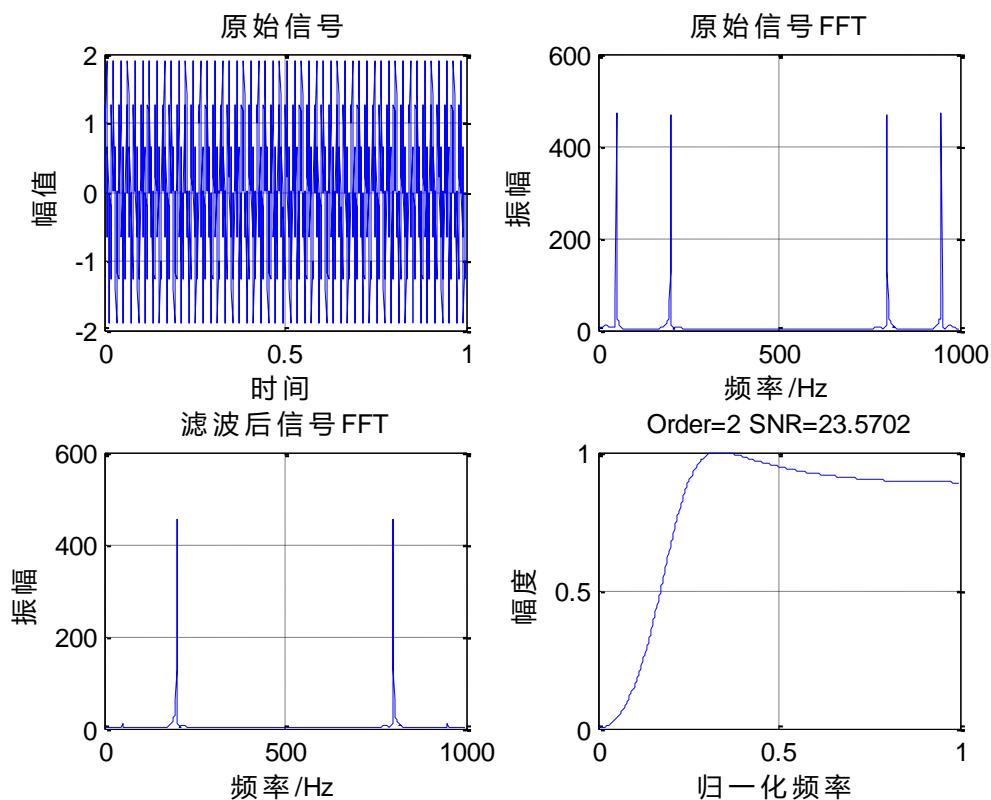
```
grid on;

subplot(222);
y=fft(x,N); %绘制原始信号的幅频响应
plot(f,abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

subplot(223);
Wc=2*125/fs; %设置截止频率 125Hz
[b,a]=cheby1(2, 1, Wc, 'high'); %获取 2 阶 IIR 滤波系数
% y2=filter(b,a,x);
y2=filtfilt(b,a,x); %计算滤波后的波形 y2
y3=fft(y2,N); %滤波后波形的幅频响应
plot(f,abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

[H,F]=freqz(b,a,512);
subplot(224);
plot(F/pi,abs(H)); %绘制绝对幅频响应
xlabel('归一化频率'); %绘制绝对幅频响应
ylabel('幅度');
Ps=sum(x2.^2); %信号的总功率
Pu=sum((y2-x2).^2); %剩余噪声的功率
SNR=10*log10(Ps/Pu); %信噪比
title(['Order=', int2str(2), ' SNR=', num2str(SNR)]);
grid on;
```

Matlab 运行结果如下：



43.2.7 切比雪夫 I型带通滤波器设计

下面我们通过一个实例来讲解切比雪夫 I型带通滤波器的设计。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，将 50Hz 的正弦波当做噪声滤掉，下面通过函数 cheby1 设计一组带通滤波器系数，其阶数是 2，通带为 125Hz 到 300Hz，采样率 1Kbps，通带纹波 1db。Matlab 运行代码如下：

```

fs=1000;          %设置采样频率 1k
N=1024;          %采样点数
n=0:N-1;
t=0:1/fs:1-1/fs;    %时间序列
f=n*fs/N;        %频率序列

x1=sin(2*pi*50*t);    %噪声
x2=sin(2*pi*200*t);    %信号
x=x1+x2;          %信号混合

subplot(221);
plot(t,x);        %绘制原始信号
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;

subplot(222);
y=fft(x,N);      %绘制原始信号的幅频响应
plot(f,abs(y));

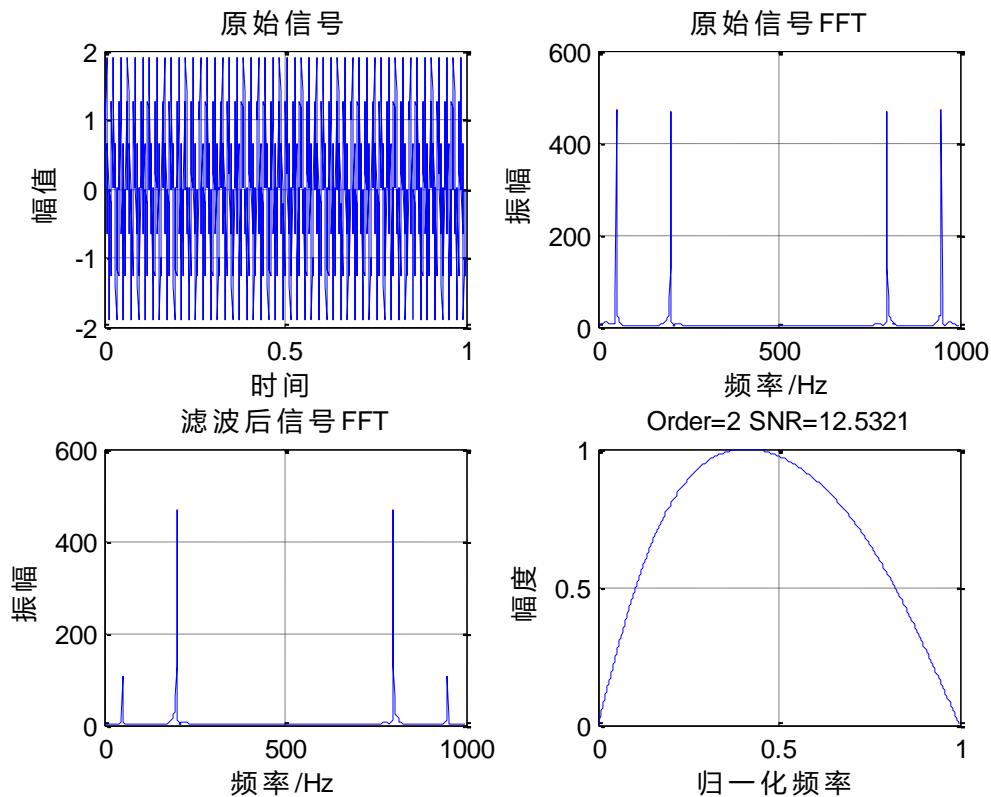
```

```
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

subplot(223);
Wn=[125*2 300*2]/fs; %设置通带 125Hz 到 300Hz
[b, a]=cheby1(1, 1, Wn); %注意第一个参数虽然是 1, 但生成的却是 2 阶 IIR 滤波器系数
% y2=filter(b, a, x);
y2=filtfilt(b, a, x); %计算滤波后的波形 y2
y3=fft(y2, N); %滤波后波形的幅频响应
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

[H, F]=freqz(b, a, 512);
subplot(224);
plot(F/pi, abs(H));
xlabel('归一化频率'); %绘制绝对幅频响应
ylabel('幅度');
Ps=sum(x2.^2); %信号的总功率
Pu=sum((y2-x2).^2); %剩余噪声的功率
SNR=10*log10(Ps/Pu); %信噪比
title(['Order=' int2str(2), ' SNR=' num2str(SNR)]);
grid on;
```

Matlab 的计算结果如下：





从滤波的效果来看，2阶的带通滤波器效果不够好，出现这种情况的时候，需要大家去重新的调节截止频率，滤波器阶数和通带波纹。

43.2.8 切比雪夫 I 型带阻滤波器设计

下面我们通过一个实例来讲解切比雪夫 I 型带阻滤波器的设计。原始信号是由 50Hz 正弦波和 200Hz 的正弦波组成，将 200Hz 的正弦波当做噪声滤掉，下面通过函数 cheby1 设计一组带阻滤波器系数，其阶数是 2，阻带为 125Hz 到 300Hz，采样率 1Kbps，通带波纹 1db。Matlab 运行代码如下：

```
fs=1000; %设置采样频率 1k
N=1024; %采样点数
n=0:N-1;
t=0:1/fs:1-1/fs; %时间序列
f=n*fs/N; %频率序列

x1=sin(2*pi*50*t); %信号
x2=sin(2*pi*200*t); %噪声
x=x1+x2; %信号混合

subplot(221);
plot(t, x); %绘制原始信号
xlabel('时间');
ylabel('幅值');
title('原始信号');
grid on;

subplot(222);
y=fft(x, N); %绘制原始信号的幅频响应
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号 FFT');
grid on;

subplot(223);
Wn=[125*2 300*2]/fs; %设置阻带 125Hz 到 300Hz
[b, a]=cheby1(1, 1, Wn, 'stop'); %注意第一个参数虽然是 1，但生成的却是 2 阶 IIR 滤波器系数
% y2=filter(b, a, x);
y2=filtfilt(b, a, x); %计算滤波后的波形 y2
y3=fft(y2, N); %滤波后波形的幅频响应
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('滤波后信号 FFT');
grid on;

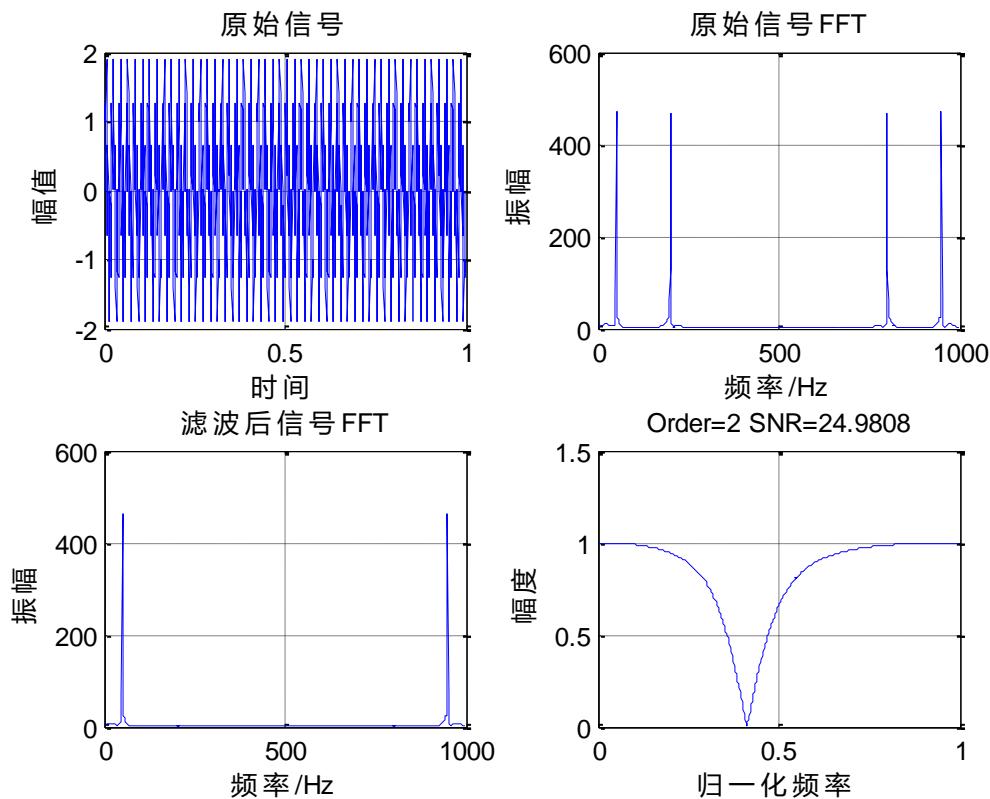
[H, F]=freqz(b, a, 512);
subplot(224);
plot(F/pi, abs(H)); %绘制绝对幅频响应
xlabel('归一化频率');
ylabel('幅度');
Ps=sum(x1.^2); %信号的总功率
Pu=sum((y2-x1).^2); %剩余噪声的功率
```

```

SNR=10*log10(Ps/Pu); %信噪比
title(['Order=' int2str(2), ' SNR=' num2str(SNR)]);
grid on;

```

Matlab 计算结果如下：



从滤波的效果来看，2 阶带阻滤波的效果较好。

43.3 椭圆滤波器的设计

43.3.1 ellip 函数

功能：用来设计 Elliptic(椭圆)型滤波器

语法：[b,a] = ellip(n, Rp, Rs, Wn);

[b,a] = ellip(n, Rp, Rs, Wn, 'ftype');

说明：ellip 函数与 cheby1、cheby2 函数类似，可以设计低通、高通、带通和带阻数字滤波器。参数 R_p 和 R_s 分别用来指定通带波纹和阻带波纹， W_n 指定滤波器的截止频率， n 为滤波器的阶数。

与 Butterworth 和 Chebyshev 滤波器相比，ellip 函数可以得到下降斜度更大、衰减更快的滤波器，但通带和阻带内均为等纹波。通常情况下，椭圆滤波器能以较低的阶数来实现指定的性能。

[b,a] = ellip(n, Rp, Rs, Wn); 可设计 n 阶低通或带通滤波器。当 $W_n=[W_1 \ W_2]$ 时，可设计带通滤波器。

[b,a] = ellip(n, Rp, Rs, Wn, 'ftype'); 可设计 n 阶高通或带阻滤波器。



当 $f\text{type}=\text{high}$ 时，可设计截止频率为 W_n 的高通滤波器。

当 $f\text{type}=\text{stop}$ 时，且 $W_n=[W_1 \ W_2]$ 时，可设计带阻滤波器，阻带为 $W_1 < W < W_2$ 。

43.3.2 ellipord 函数

功能：用来选择椭圆滤波器的阶数。

语法： $[n, W_n] = \text{cheb2ord } (W_p, W_s, R_p, R_s)$ ；

说明：ellipord 函数与 cheb1ord 函数类似，用于选择指定性能时的椭圆滤波器的最小阶数 n 和截止频率 W_n ，并与 ellip 函数配合可设计出最低阶数的椭圆滤波器。

43.3.3 椭圆滤波器设计

关于椭圆滤波器的使用，大家参考前面的切比雪夫滤波器设计即可，使用方法基本是类似的。但是由于椭圆滤波器要同时给我通带和阻带的纹波，所以要得到满足要求的滤波器系数要花些时间去做测试。

43.4 总结

本章节主要讲解了巴特沃斯，切比雪夫和椭圆滤波器的设计，如果想用好还需要大家多多做测试，并深入了解相关理论知识。



第44章 STM32H7 的 IIR 低通滤波器实现（支持

逐个数据的实时滤波）

本章节讲解 IIR 低通滤波器实现。

44.1 初学者重要提示

44.2 低通滤波器介绍

44.3 IIR 滤波器介绍

44.4 Matlab 工具箱 filterDesigner 生成低通滤波器 C 头文件

44.5 IIR 低通滤波器设计

44.6 实验例程说明 (MDK)

44.7 实验例程说明 (IAR)

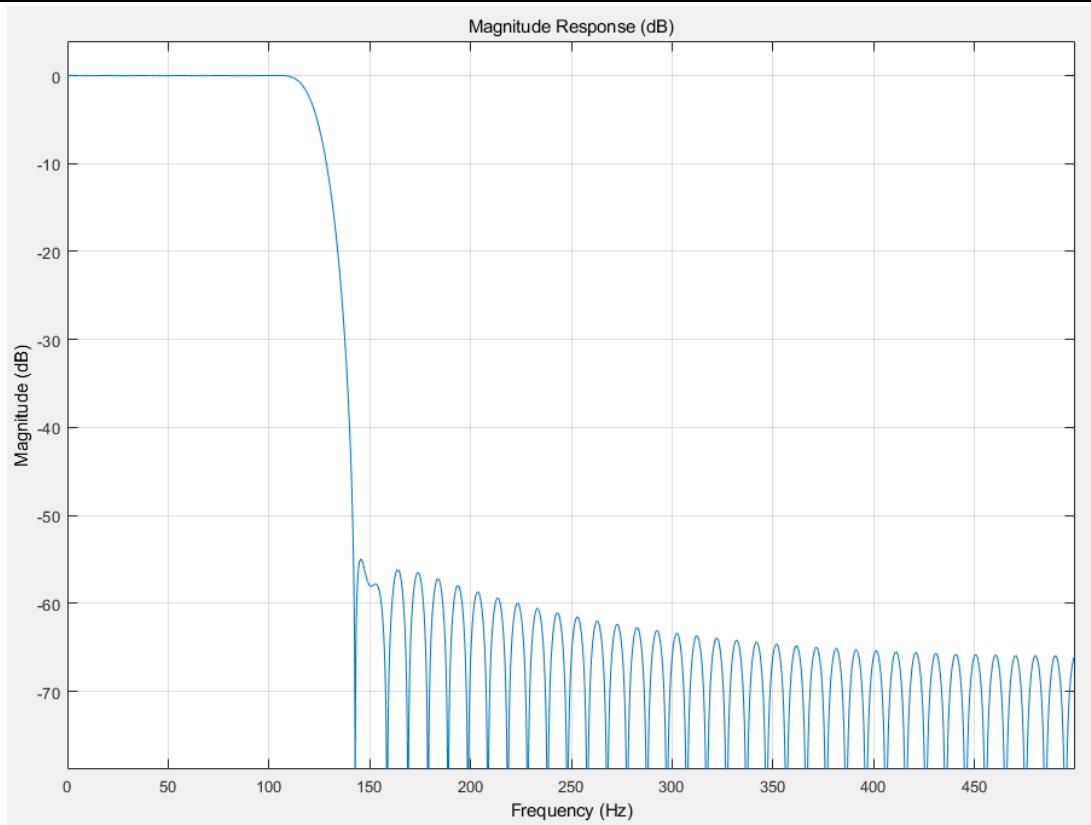
44.8 总结

44.1 初学者重要提示

- ◆ 本章节提供的低通滤波器支持实时滤波，每次可以滤波一个数据，也可以多个数据，不限制大小。
但要注意以下两点：
 - 所有数据是在同一个采样率下依次采集的数据。
 - 每次过滤数据个数一旦固定下来，运行中不可再修改。
- ◆ IIR 滤波器的群延迟是一个重要的知识点，详情在本教程第 41 章有详细说明。IIR 和 FIR 一样，也有群延迟问题。

44.2 低通滤波器介绍

允许低频信号通过，而减弱高于截止频率的信号通过。比如混合信号含有 50Hz + 200Hz 信号，我们可通过低通滤波器，过滤掉 200Hz 信号，让 50Hz 信号通过。



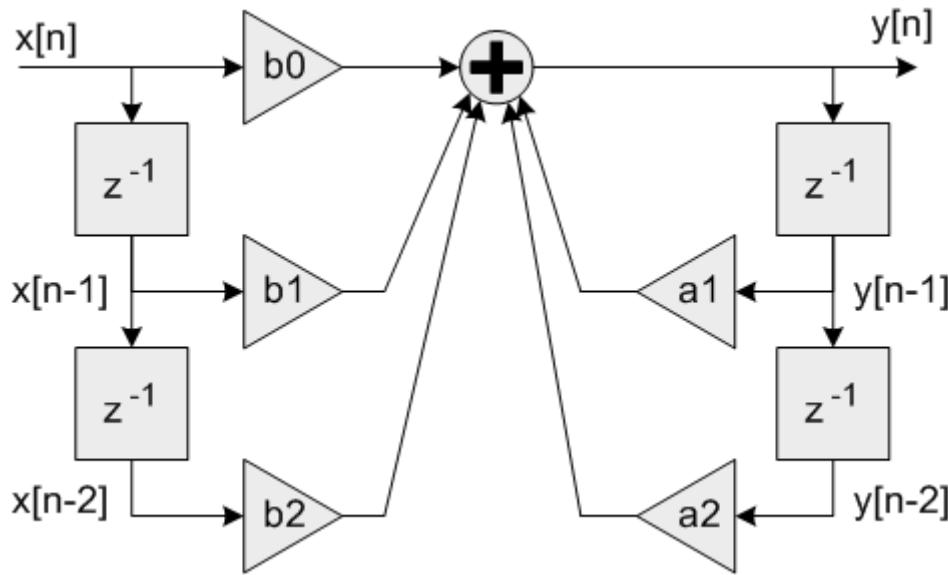
44.3 IIR 滤波器介绍

ARM 官方提供的直接 I 型 IIR 库支持 Q7, Q15, Q31 和浮点四种数据类型。其中 Q15 和 Q31 提供了快速版本。

直接 I 型 IIR 滤波器是基于二阶 Biquad 级联的方式来实现的。每个 Biquad 由一个二阶的滤波器组成：

$$y[n] = b_0 * x[n] + b_1 * x[n-1] + b_2 * x[n-2] + a_1 * y[n-1] + a_2 * y[n-2]$$

直接 I 型算法每个阶段需要 5 个系数和 4 个状态变量。

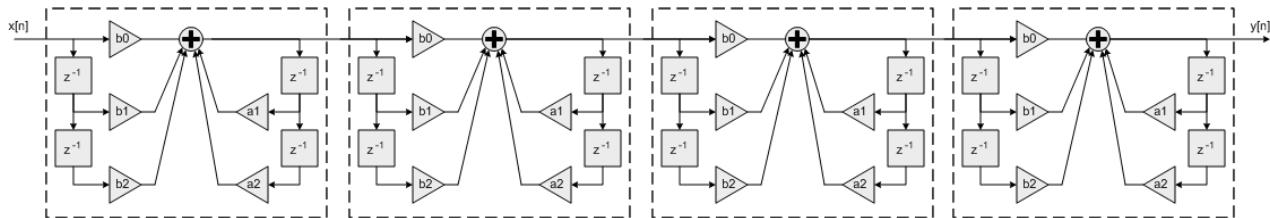


这里有一点要特别的注意，有些滤波器系数生成工具是采用的下面公式实现：

$$y[n] = b_0 * x[n] + b_1 * x[n-1] + b_2 * x[n-2] - a_1 * y[n-1] - a_2 * y[n-2]$$

比如 matlab 就是使用上面的公式实现的，所以在使用 fdatool 工具箱生成的 a 系数需要取反才能用于直接 I 型 IIR 滤波器的函数中。

高阶 IIR 滤波器的实现是采用二阶 Biquad 级联的方式来实现的。其中参数 numStages 就是用来做指定二阶 Biquad 的个数。比如 8 阶 IIR 滤波器就可以采用 numStages=4 个二阶 Biquad 来实现。



如果要实现 9 阶 IIR 滤波器就需要将 numStages=5，这时就需要其中一个 Biquad 配置成一阶滤波器(也就是 b2=0, a2=0)。

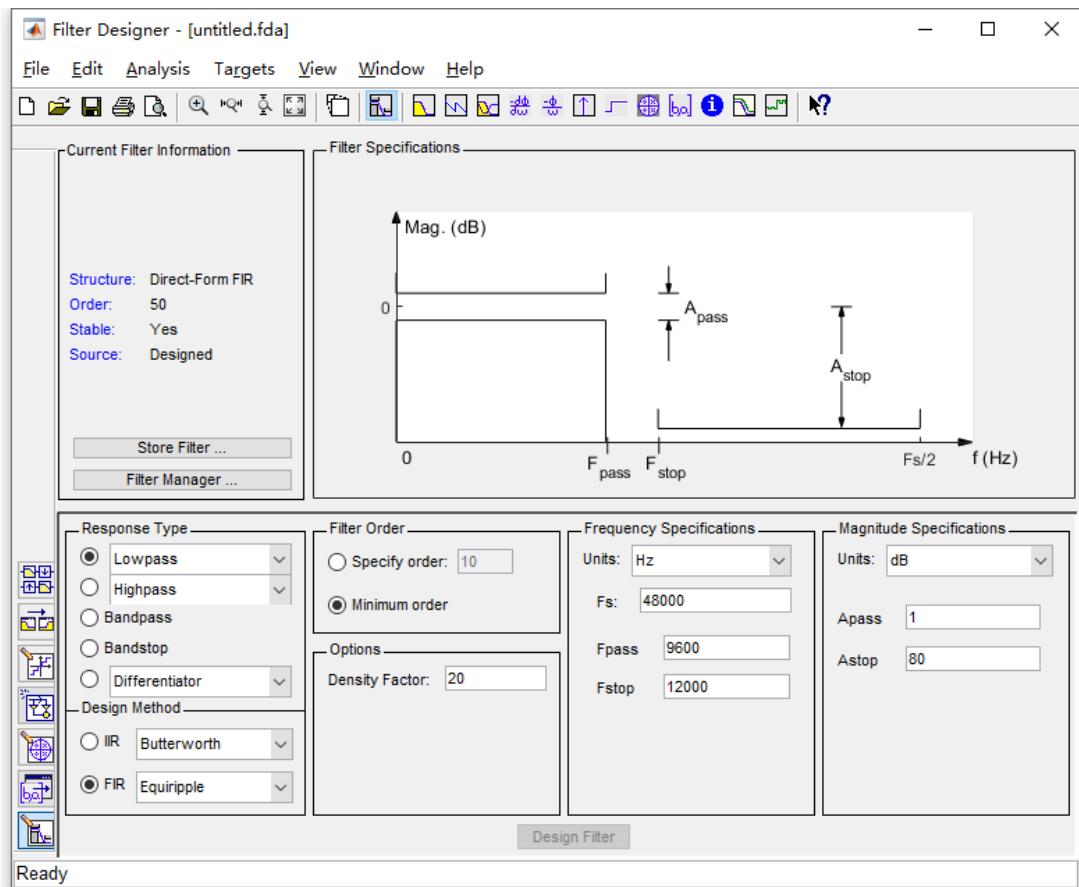
44.4 Matlab 工具箱 filterDesigner 生成 IIR 低通滤波器系数

前面介绍 FIR 滤波器的时候，我们讲解了如何使用 filterDesigner 生成 C 头文件，从而获得滤波器系数。这里不能再使用这种方法了，主要是因为通过 C 头文件获取的滤波器系数需要通过 ARM 官方的 IIR 函数调用多次才能获得滤波结果，所以我们这里换另外一种方法。

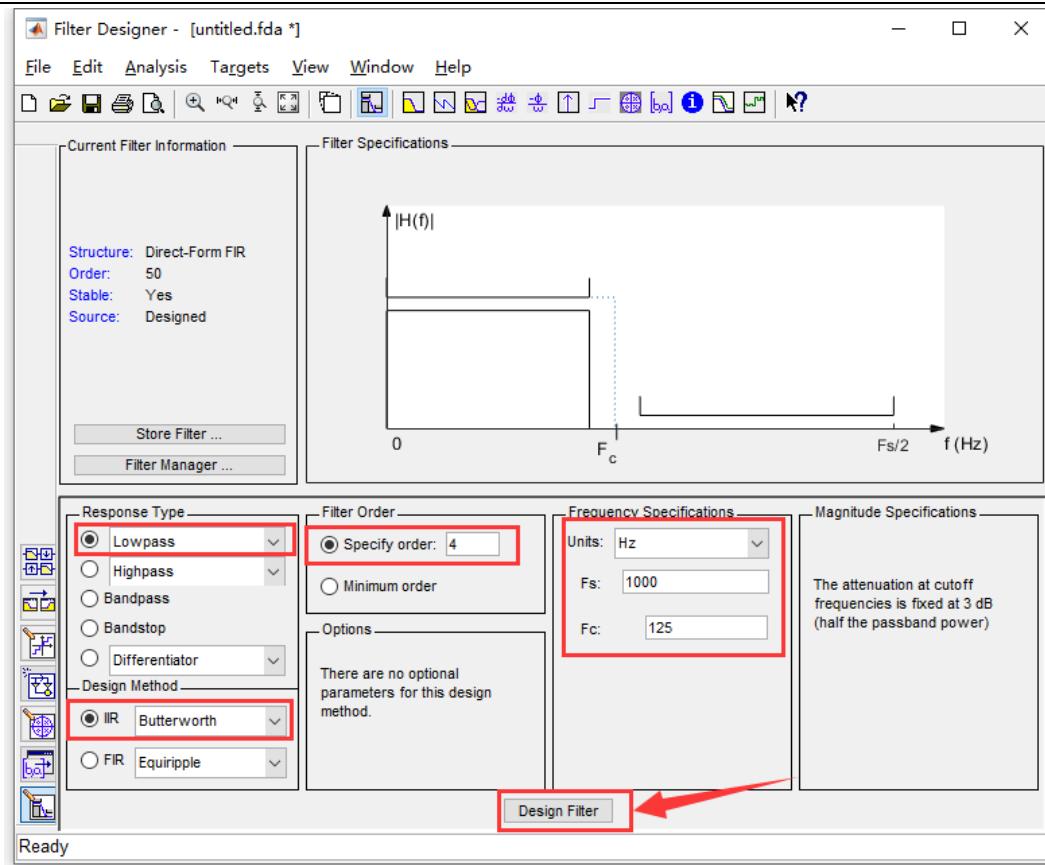
下面我们讲解如何通过 filterDesigner 工具箱生成滤波器系数。首先在 matlab 的命令窗口输入 filterDesigner 就能打开这个工具箱：

```
命令行窗口
>> filterDesigner
fx >>
```

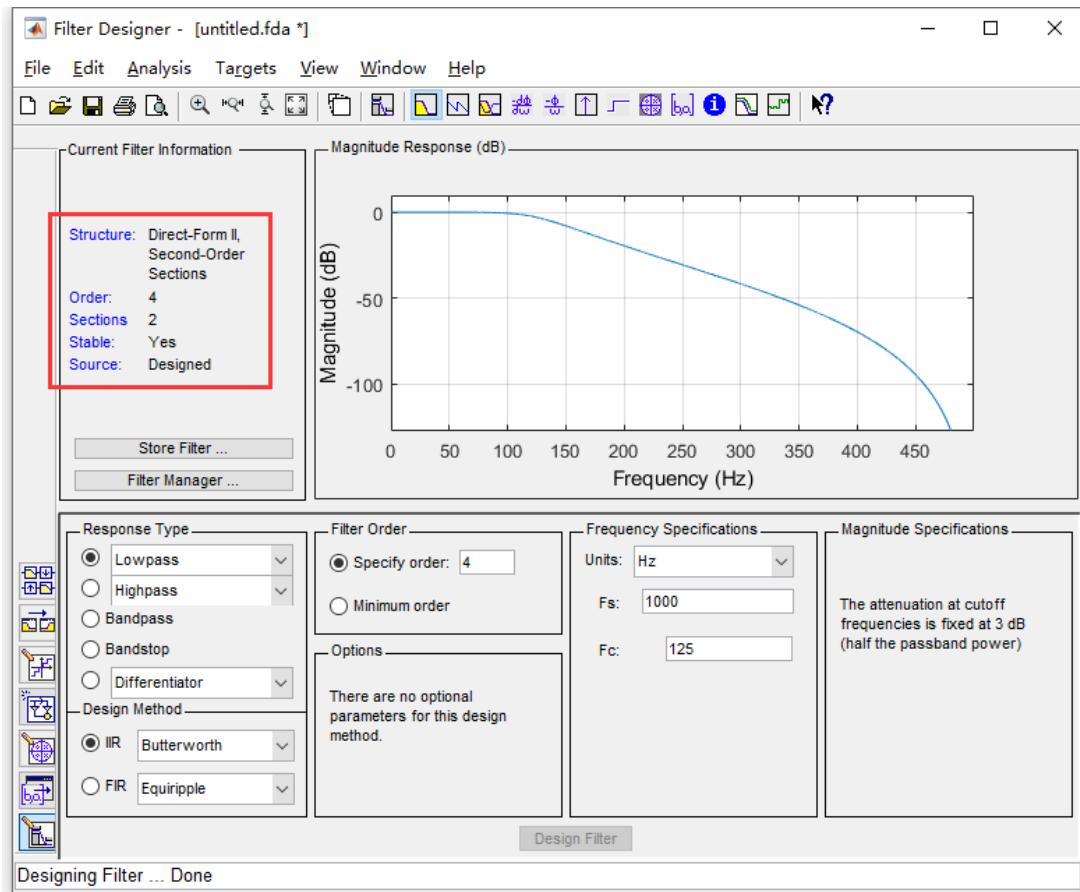
filterDesigner 界面打开效果如下：



IIR 滤波器的低通，高通，带通，带阻滤波的设置会在下面一一讲解，这里说一下设置后相应参数后如何生成滤波器系数。参数设置好以后点击如下按钮：

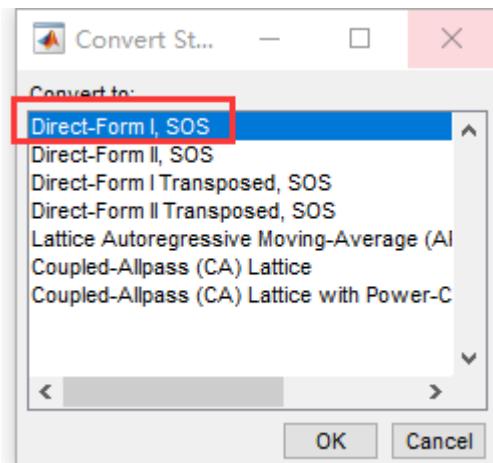


点击 Design Filter 之后，注意左上角生成的滤波器结构：

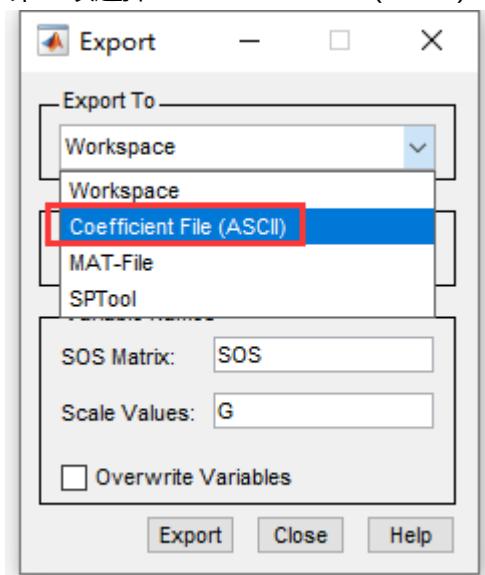


默认生成的 IIR 滤波器类型是 Direct-Form II, Second-Order Sections (直接 II 型，每个 Section 是一个二阶滤波器)。这里我们需要将其转换成 Direct-Form I, Second-Order Sections，因为本章使用的 IIR 滤波器函数是 Direct-Form I 的结构。

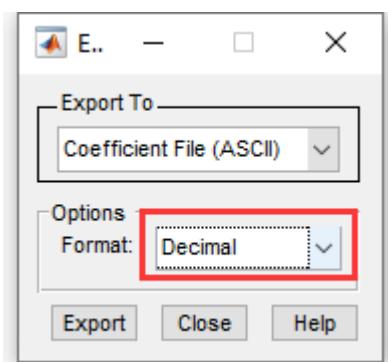
转换方法，点击 Edit->Convert Structure，界面如下，这里我们选择第一项，并点击 OK：



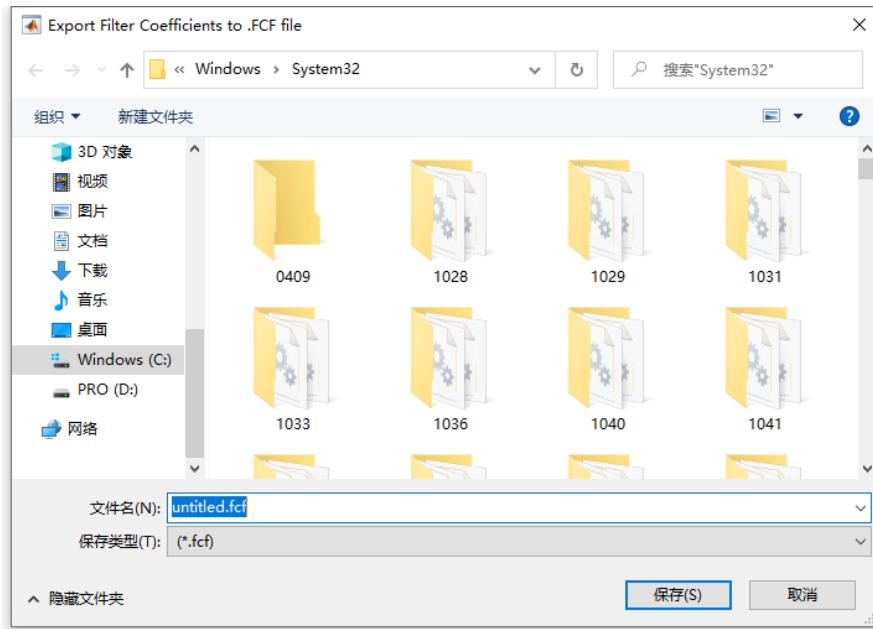
转换好以后再点击 File-Export，第一项选择 Coefficient File (ASCII)：



第一项选择好以后，第二项选择 Decimal：



两个选项都选择好以后，点击 Export 进行导出，导出后保存即可：



保存后 Matlab 会自动打开 `untitled.fcf` 文件，可以看到生成的系数：

```
% Generated by MATLAB(R) 9.4 and Signal Processing Toolbox 8.0.
% Generated on: 15-Aug-2021 15:21:07

% Coefficient Format: Decimal

% Discrete-Time IIR Filter (real)
%
% Filter Structure      : Direct-Form I, Second-Order Sections
% Number of Sections   : 2
% Stable                : Yes
% Linear Phase          : No

SOS Matrix:
1 2 1 1 -1.113029854163347875939381622476503253 0.574061915083954765748330828500911593437
1 2 1 1 -0.85539793277517017777113167394418269396 0.209715357756554754420363906319835223258

Scale Values:
0.115258015230151708574446445254579884931
0.08857935624534613028302487691689748317
```

由于前面选择的是 4 阶 IIR 滤波，生成的结果就是由两组二阶 IIR 滤波系数组成，系数的对应顺序如下：

```
SOS Matrix:
1 2 1 1 -1.113029854163347875939381622476503253 0.574061915083954765748330828500911593437
b0 b1 b2 a0           a1                               a2
1 2 1 1 -0.85539793277517017777113167394418269396 0.209715357756554754420363906319835223258
b0 b1 b2 a0           a1                               a2
```

注意，实际使用 ARM 官方的 IIR 函数调用的时候要将 `a1` 和 `a2` 取反。另外下面两组是每个二阶滤波器的增益，滤波后的结果要乘以这两个增益数值才是实际结果：

```
0.115258015230151708574446445254579884931
0.08857935624534613028302487691689748317
```

实际的滤波系数调用方法，看下面的例子即可。



44.5 IIR 低通滤波器设计

本章使用的 IIR 滤波器函数是 arm_biquad_cascade_df1_f32。使用此函数可以设计 IIR 低通，高通，带通和带阻滤波器

44.5.1 函数 arm_biquad_cascade_df1_init_f32

函数原型：

```
void arm_biquad_cascade_df1_init_f32(
    arm_biquad_casd_df1_inst_f32 * S,
    uint8_t numStages,
    const float32_t * pCoeffs,
    float32_t * pState)
```

函数描述：

这个函数用于 IIR 初始化。

函数参数：

- ◆ 第 1 个参数是 arm_biquad_casd_df1_inst_f32 类型结构体变量。
- ◆ 第 2 个参数是 2 阶滤波器的个数。
- ◆ 第 3 个参数是滤波器系数地址。
- ◆ 第 4 个参数是缓冲状态地址。

注意事项：

结构体 arm_biquad_casd_df1_inst_f32 的定义如下 (在文件 filtering_functions.h 文件) :

```
typedef struct
{
    uint32_t numStages;      /*< number of 2nd order stages in the filter. Overall order is 2*numStages. */
    float32_t *pState; /*< Points to the array of state coefficients. The array is of length 4*numStages. */
    const float32_t *pCoeffs; /*< Points to the array of coefficients. The array is of length 5*numStages */
} arm_biquad_casd_df1_inst_f32;
```

1. numStages 表示二阶滤波器的个数，总阶数是 2*numStages。
2. pState 指向状态变量数组，这个数组用于函数内部计算数据的缓存，总大小 4*numStages。
3. 参数 pCoeffs 指向滤波因数，滤波因数数组长度为 5*numStages。但要注意 pCoeffs 指向的滤波因数应该按照如下的逆序进行排列：

{b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ...}

先放第一个二阶 Biquad 系数，然后放第二个，以此类推。

44.5.2 函数 arm_biquad_cascade_df1_f32

函数定义如下：

```
void arm_biquad_cascade_df1_f32(
    const arm_biquad_casd_df1_inst_f32 * S,
    float32_t * pSrc,
```



```
float32_t * pDst,  
uint32_t blockSize)
```

函数描述：

这个函数用于 IIR 滤波。

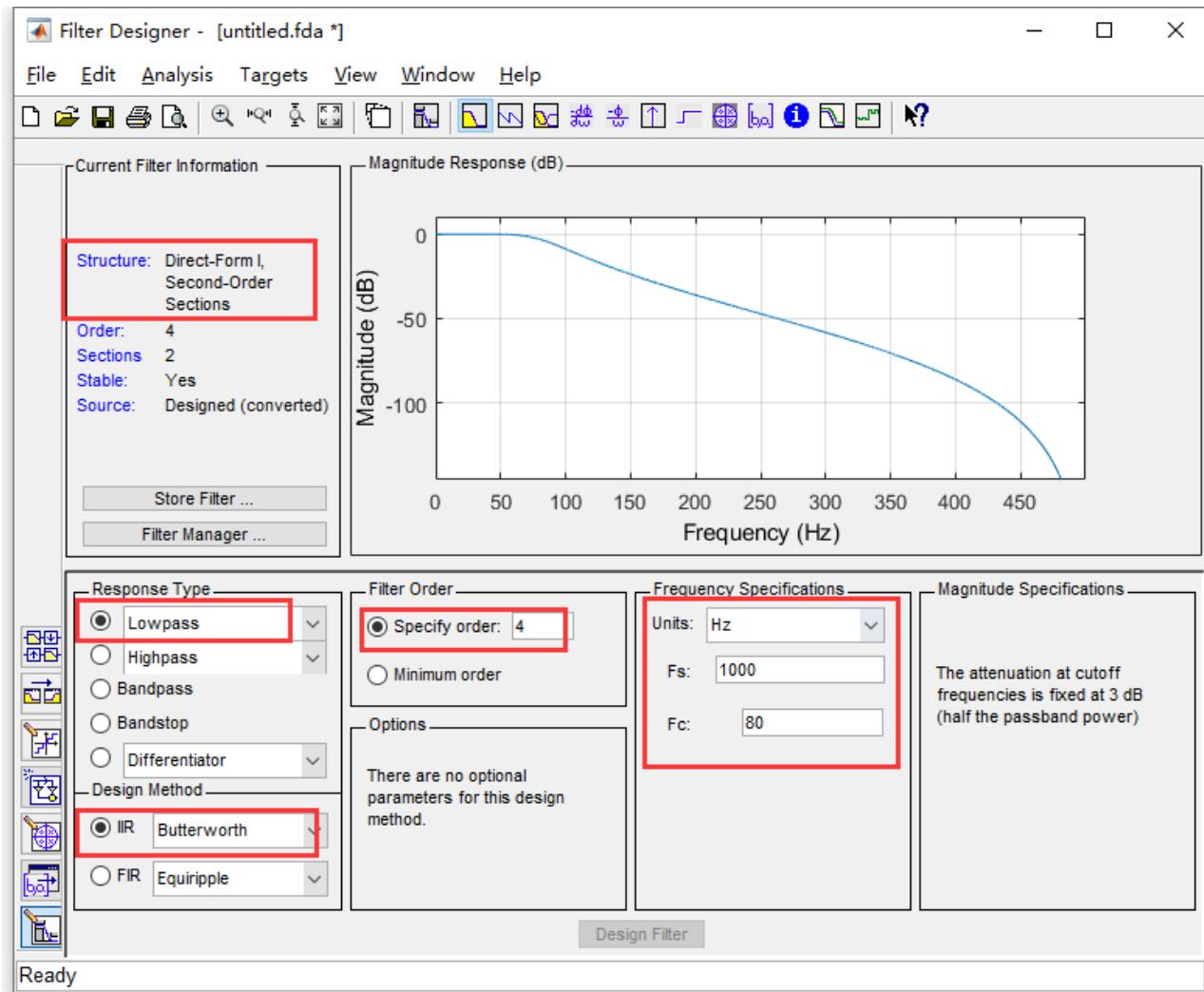
函数参数：

- ◆ 第 1 个参数是 arm_biquad_casd_df1_inst_f32 类型结构体变量。
- ◆ 第 2 个参数是源数据地址。
- ◆ 第 3 个参数是滤波后的数据地址。
- ◆ 第 4 个参数是每次调用处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

44.5.3 filterDesigner 获取低通滤波器系数

设计一个如下的例子：

信号由 50Hz 正弦波和 200Hz 正弦波组成，采样率 1Kbps，现设计一个巴特沃斯滤波器低通滤波器，采用直接 I 型，截止频率 80Hz，采样 400 个数据，滤波器阶数设置为 4。filterDesigner 的配置如下：





配置好低通滤波器后，具体滤波器系数的生成大家参考本章第4小节的方法即可。

44.5.4 低通滤波器实现

通过工具箱 filterDesigner 获得低通滤波器系数后在开发板上运行函数 arm_biquad_cascade_df1_f32 来测试低通滤波器的效果。

```
#define numStages 2           /* 2阶IIR滤波的个数 */
#define TEST_LENGTH_SAMPLES 400 /* 采样点数 */
#define BLOCK_SIZE 1           /* 调用一次arm_biquad_cascade_df1_f32处理的采样点个数 */

uint32_t blockSize = BLOCK_SIZE;
uint32_t numBlocks = TEST_LENGTH_SAMPLES/BLOCK_SIZE;           /* 需要调用arm_biquad_cascade_df1_f32的次数 */

static float32_t testInput_f32_50Hz_200Hz[TEST_LENGTH_SAMPLES]; /* 采样点 */
static float32_t testOutput[TEST_LENGTH_SAMPLES];                /* 滤波后的输出 */
static float32_t IIRStateF32[4*numStages];                   /* 状态缓存 */

/* 巴特沃斯低通滤波器系数 80Hz*/
const float32_t IIRCoeffs32LP[5*numStages] = {
    1.0f, 2.0f, 1.0f, 1.479798894397216679763573665695730596781f,
    -0.688676953053861784503908438637154176831f,
    1.0f, 2.0f, 1.0f, 1.212812092620218384908525877108331769705f,
    -0.384004162286553540894828984164632856846f
};

/*
*****
* 函数名: arm_iir_f32_lp
* 功能说明: 调用函数 arm_iir_f32_lp 实现低通滤波器
* 形参: 无
* 返回值: 无
*****
*/
static void arm_iir_f32_lp(void)
{
    uint32_t i;
    arm_biquad_cascade_df1_inst_f32 S;
    float32_t ScaleValue;
    float32_t *inputF32, *outputF32;

    /* 初始化输入输出缓存指针 */
    inputF32 = &testInput_f32_50Hz_200Hz[0];
    outputF32 = &testOutput[0];

    /* 初始化 */
    arm_biquad_cascade_df1_init_f32(&S, numStages, (float32_t *)&IIRCoeffs32LP[0],
                                    (float32_t *)&IIRStateF32[0]);

    /* 实现 IIR 滤波, 这里每次处理 1 个点 */
    for(i=0; i < numBlocks; i++)
    {
        arm_biquad_cascade_df1_f32(&S, inputF32 + (i * blockSize), outputF32 + (i * blockSize),
                                blockSize);
    }

    /* 放缩系数 */
```



```
ScaleValue = 0.052219514664161220673932461977528873831f * 0.042798017416583809813257488485760404728f ;
```

```
/* 打印滤波后结果 */
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    printf("%f, %f\r\n", testInput_f32_50Hz_200Hz[i], testOutput[i]*ScaleValue);
}
```

运行如上函数可以通过串口打印出函数arm_biquad_cascade_df1_f32滤波后的波形数据，下面通过Matlab绘制波形来对比Matlab计算的结果和ARM官方库计算的结果。

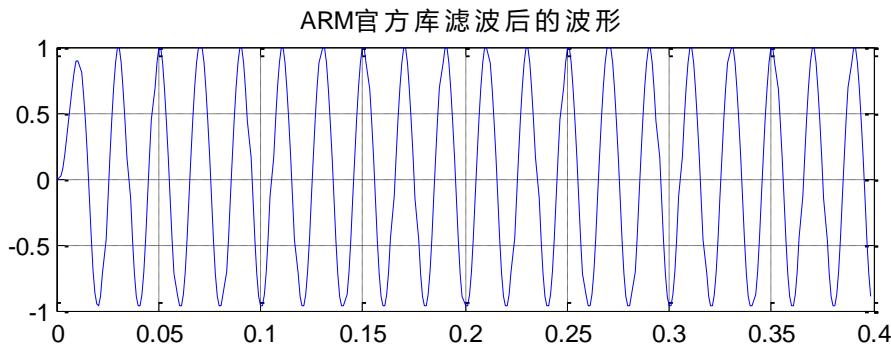
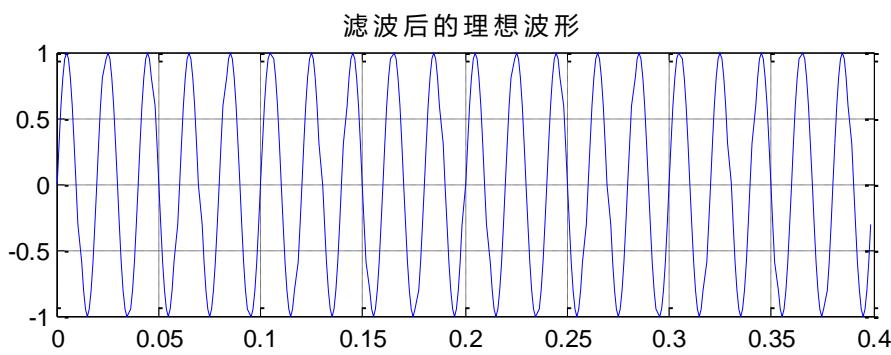
对比前需要先将串口打印出的一组数据加载到 Matlab 中， arm_biquad_cascade_df1_f32 的计算结果起名 sampledata，加载方法在[第 13 章 13.6 小结已经讲解](#)，这里不做赘述了。Matlab 中运行的代码如下：

```
fs=1000;          %设置采样频率 1K
N=400;           %采样点数
n=0:N-1;
t=n/fs;          %时间序列
f=n*fs/N;         %频率序列

x1=sin(2*pi*50*t);
x2=sin(2*pi*200*t);    %50Hz和200Hz正弦波
subplot(211);
plot(t, x1);
title('滤波后的理想波形');
grid on;

subplot(212);
plot(t, sampledata);
title('ARM官方库滤波后的波形');
grid on;
```

Matlab 计算结果如下：



从上面的波形对比来看，matlab 和函数 arm_biquad_cascade_df1_f32 计算的结果基本是一致的。为了更好的说明滤波效果，下面从频域的角度来说明这个问题，Matlab 上面运行如下代码：

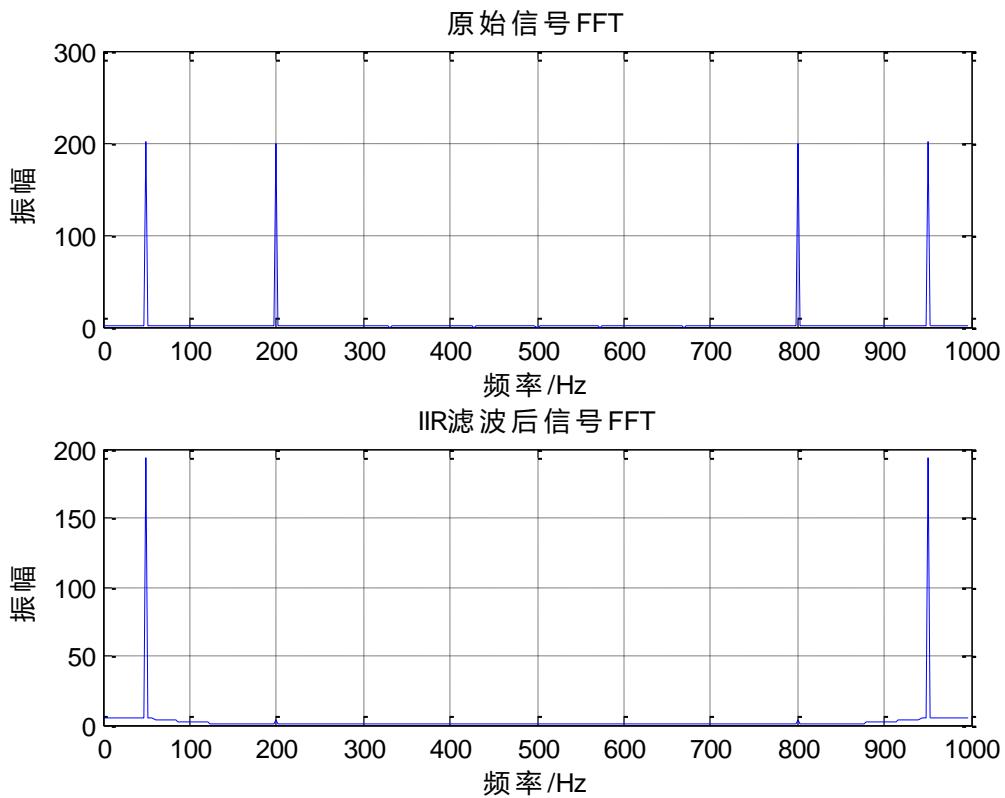
```
fs=1000; %设置采样频率 1K
N=400; %采样点数
n=0:N-1;
t=n/fs; %时间序列
f=n*fs/N; %频率序列

x = sin(2*pi*50*t) + sin(2*pi*200*t); %50Hz和200Hz正弦波合成

subplot(211);
y=fft(x, N); %对信号x做FFT
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号FFT');
grid on;

y3=fft(sampledata, N); %经过IIR滤波器后得到的信号做FFT
subplot(212);
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('IIR滤波后信号FFT');
grid on;
```

Matlab 计算结果如下：



上面波形变换前的 FFT 和变换后 FFT 可以看出，200Hz 的正弦波基本被滤除。



44.6 实验例程说明 (MDK)

配套例子：

V7-229_IIR 低通滤波器(支持逐点实时滤波)

实验目的：

1. 学习 IIR 低通滤波器的实现，支持实时滤波

实验内容：

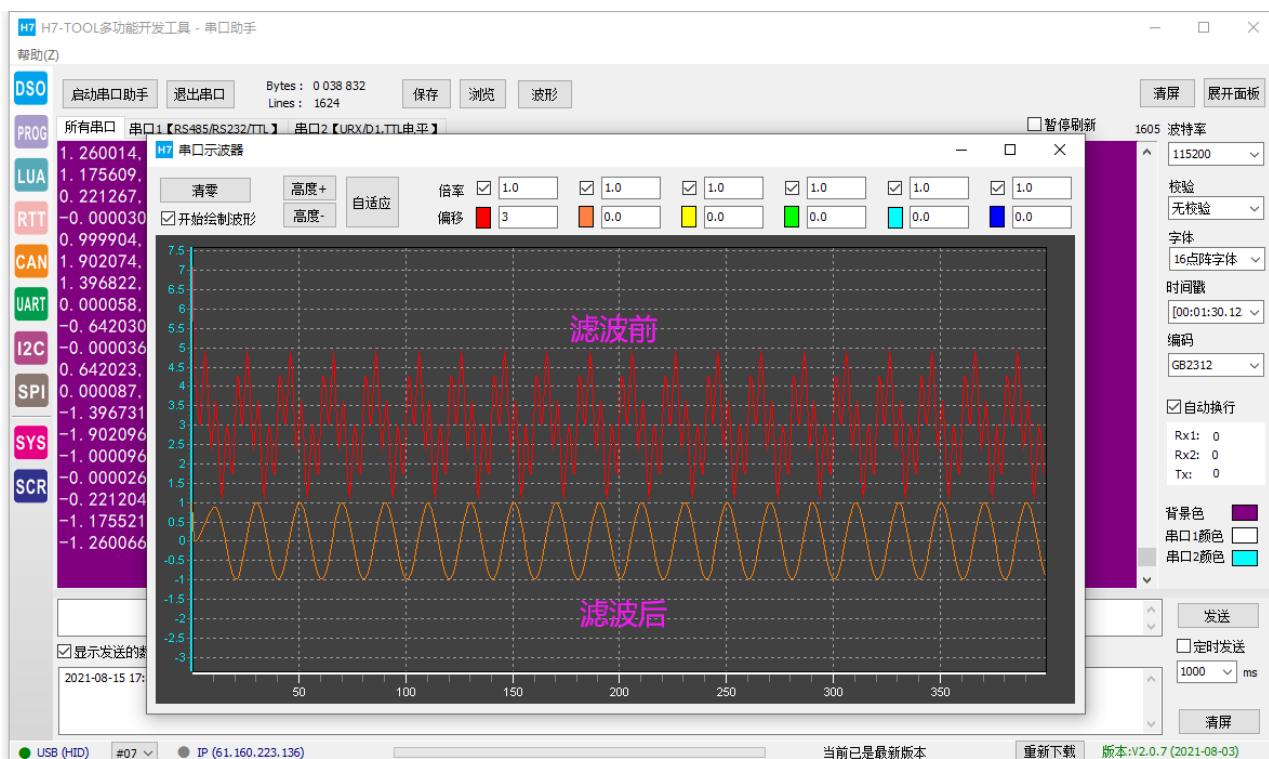
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

使用 AC6 注意事项

特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

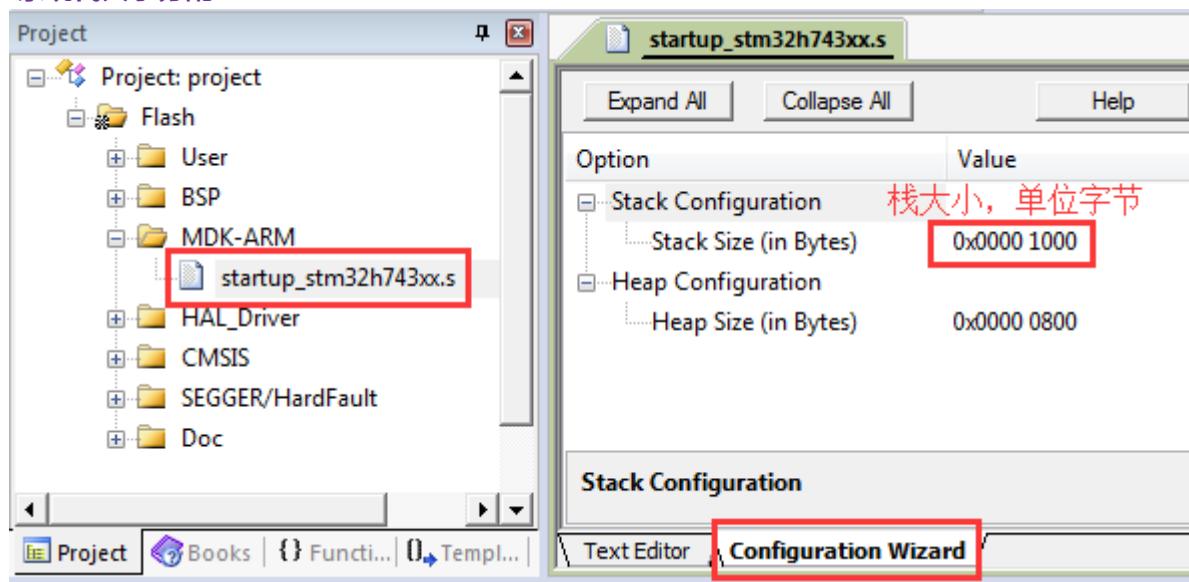


RTT 方式打印信息：

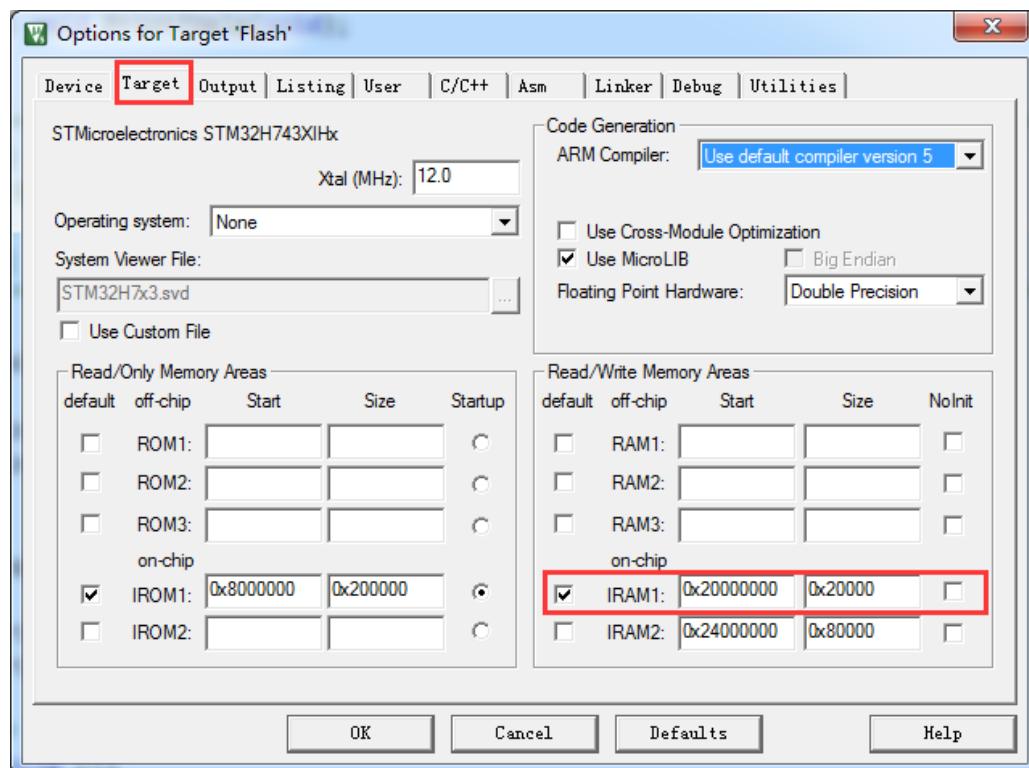


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
    testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                    arm_sin_f32(2*3.1415926f*200*i/1000);
}

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2); /* 翻转 LED 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_iir_f32_lp();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
}
```

44.7 实验例程说明 (IAR)

配套例子：

V7-229_IIR 低通滤波器(支持逐点实时滤波)

实验目的：

1. 学习 IIR 低通滤波器的实现，支持实时滤波

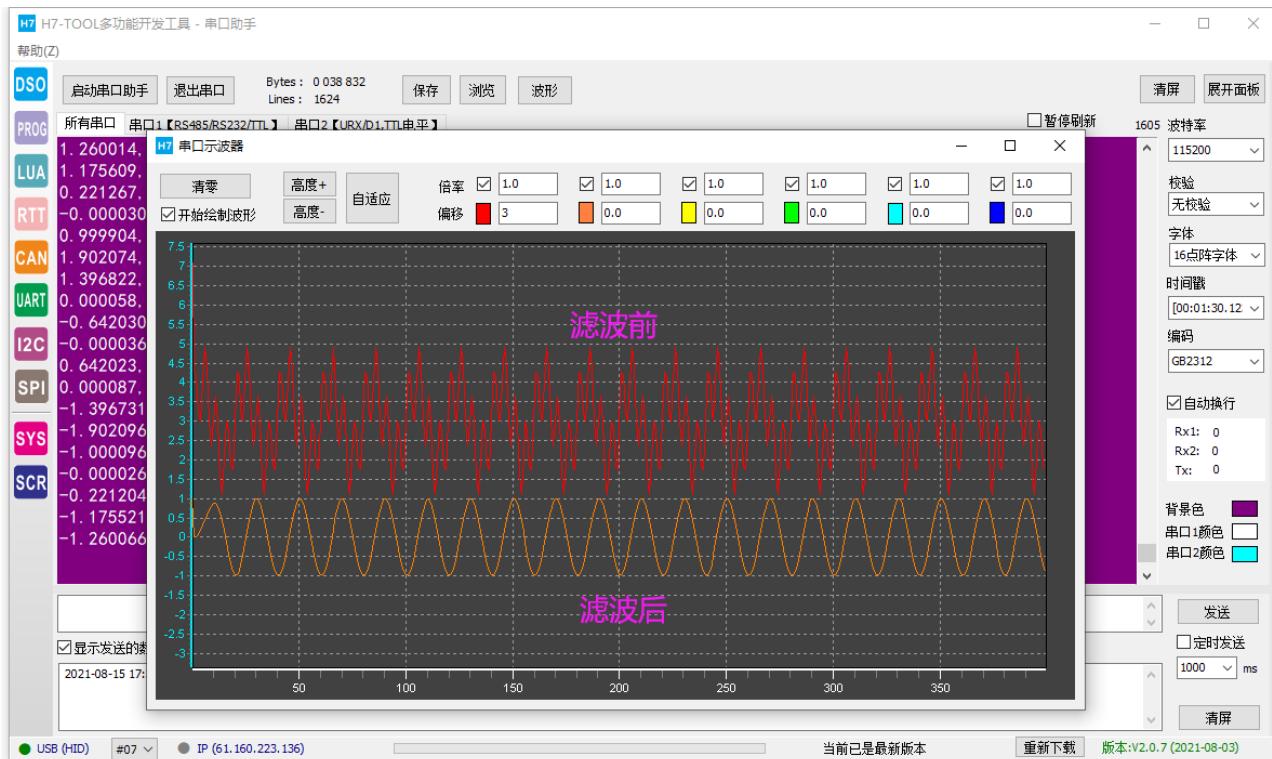
实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

上电后串口打印的信息：



波特率 115200，数据位 8，奇偶校验位无，停止位 1。

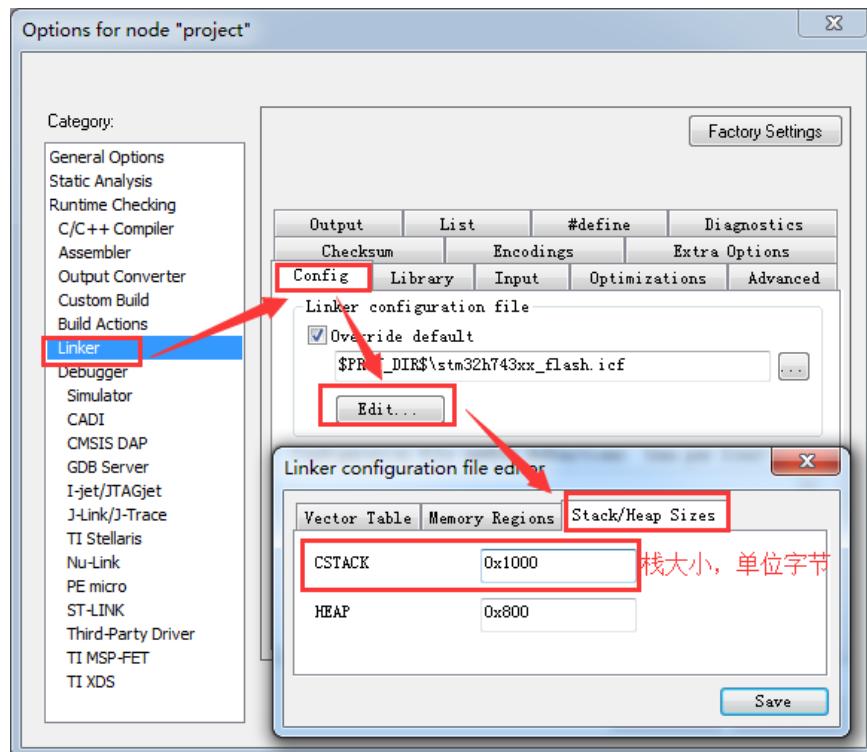


RTT 方式打印信息：

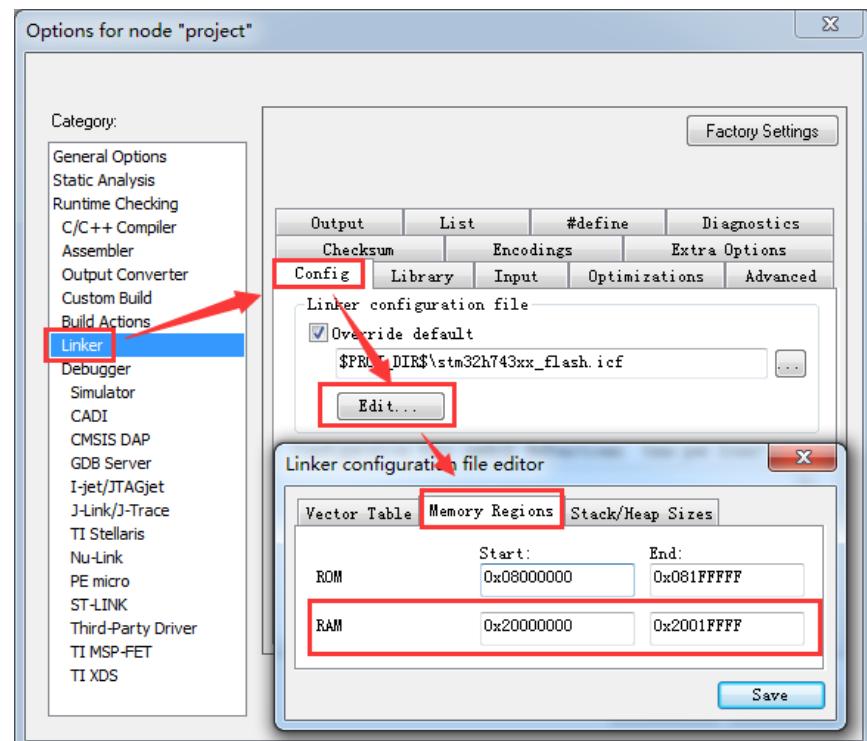


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
*****
```



```
/* 功能说明：初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形    参：无
* 返回值：无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
        - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
        - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名：MPU_Config
* 功能说明：配置 MPU
* 形    参：无
* 返回值：无
*****
*/
static void MPU_Config( void )
```



```
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /*使能 MPU */  
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);  
}  
  
/*  
*****  
* 函数名: CPU_CACHE_Enable  
* 功能说明: 使能 L1 Cache  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void CPU_CACHE_Enable(void)  
{  
    /* 使能 I-Cache */  
    SCB_EnableICache();  
  
    /* 使能 D-Cache */  
    SCB_EnableDCache();  
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
        testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                      arm_sin_f32(2*3.1415926f*200*i/1000);
    }

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2); /* 翻转 LED 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_iir_f32_lp();
                    break;
            }
        }
    }
}
```



```
default:  
    /* 其它的键值不处理 */  
    break;  
}  
}  
}
```

44.8 总结

本章节主要讲解了 IIR 滤波器的低通实现，同时一定要注意 IIR 滤波器的群延迟问题，详见本教程的第 41 章。



第45章 STM32H7 的 IIR 高通滤波器实现（支持逐个数据的实时滤波）

本章节讲解 IIR 高通滤波器实现。

45.1 初学者重要提示

45.2 高通滤波器介绍

45.3 IIR 滤波器介绍

45.4 Matlab 工具箱 filterDesigner 生成高通滤波器 C 头文件

45.5 IIR 高通滤波器设计

45.6 实验例程说明 (MDK)

45.7 实验例程说明 (IAR)

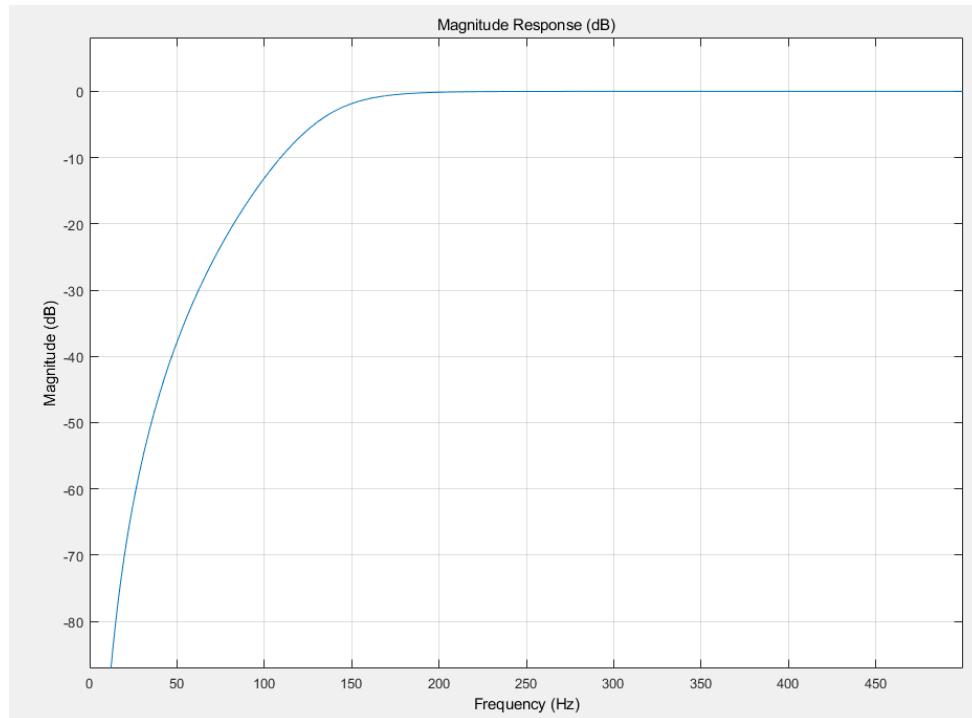
45.8 总结

45.1 初学者重要提示

- ◆ 本章节提供的高通滤波器支持实时滤波，每次可以滤波一个数据，也可以多个数据，不限制大小。
但要注意以下两点：
 - 所有数据是在同一个采样率下依次采集的数据。
 - 每次过滤数据个数一旦固定下来，运行中不可再修改。
- ◆ IIR 滤波器的群延迟是一个重要的知识点，详情在本教程第 41 章有详细说明。IIR 和 FIR 一样，也有群延迟问题。

45.2 高通滤波器介绍

允许高频信号通过，而减弱低于截止频率的信号通过。比如混合信号含有 50Hz + 200Hz 信号，我们可通过高通滤波器，过滤掉 50Hz 信号，让 200Hz 信号通过。



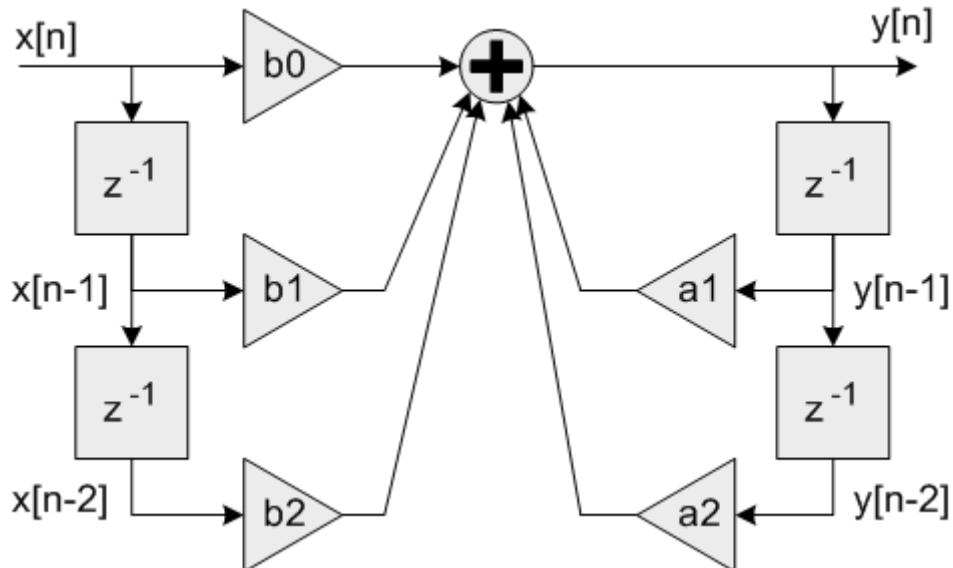
45.3 IIR 滤波器介绍

ARM 官方提供的直接 I 型 IIR 库支持 Q7, Q15, Q31 和浮点四种数据类型。其中 Q15 和 Q31 提供了快速版本。

直接 I 型 IIR 滤波器是基于二阶 Biquad 级联的方式来实现的。每个 Biquad 由一个二阶的滤波器组成：

$$y[n] = b_0 * x[n] + b_1 * x[n-1] + b_2 * x[n-2] + a_1 * y[n-1] + a_2 * y[n-2]$$

直接 I 型算法每个阶段需要 5 个系数和 4 个状态变量。

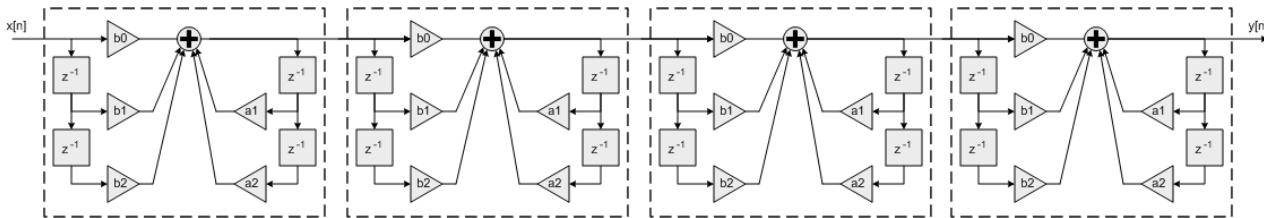


这里有一点要特别的注意，有些滤波器系数生成工具是采用的下面公式实现：

$$y[n] = b0 * x[n] + b1 * x[n-1] + b2 * x[n-2] - a1 * y[n-1] - a2 * y[n-2]$$

比如 matlab 就是使用上面的公式实现的，所以在使用 fdatool 工具箱生成的 a 系数需要取反才能用于直接 I 型 IIR 滤波器的函数中。

高阶 IIR 滤波器的实现是采用二阶 Biquad 级联的方式来实现的。其中参数 numStages 就是用来做指定二阶 Biquad 的个数。比如 8 阶 IIR 滤波器就可以采用 numStages=4 个二阶 Biquad 来实现。



如果要实现 9 阶 IIR 滤波器就需要将 numStages=5，这时就需要其中一个 Biquad 配置成一阶滤波器(也就是 $b2=0, a2=0$)。

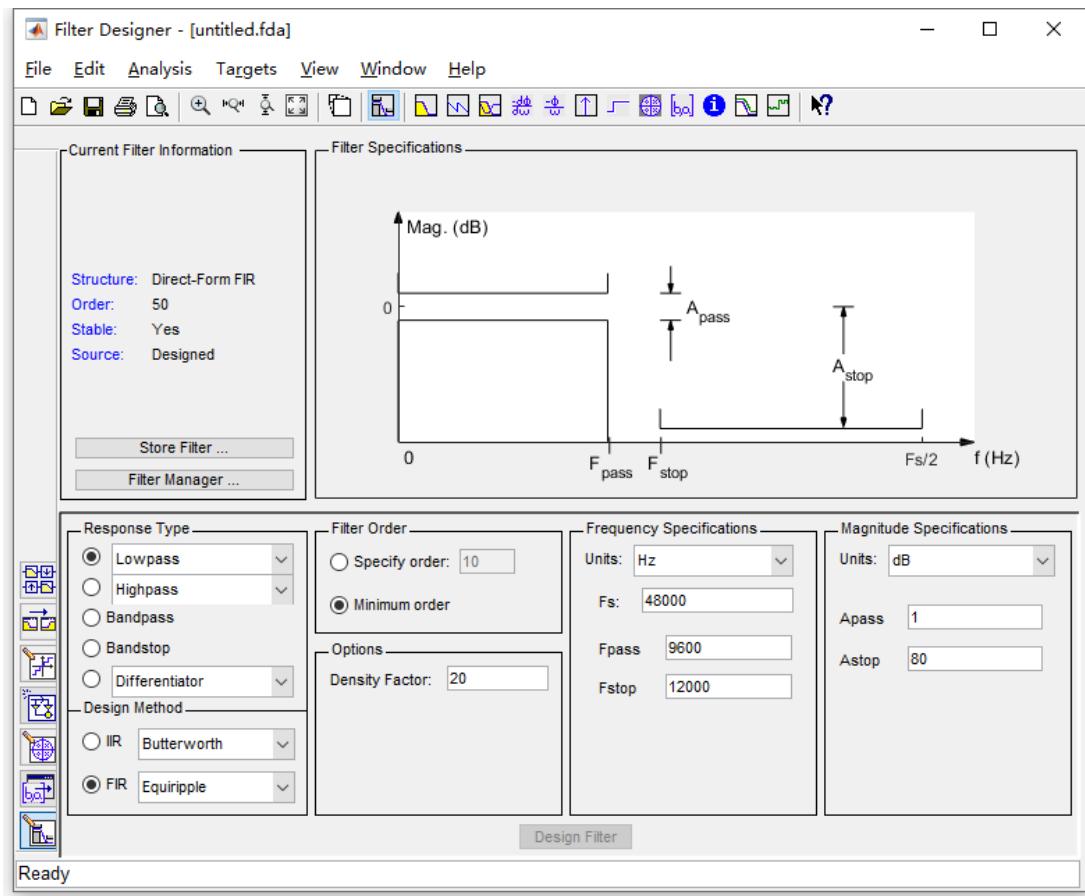
45.4 Matlab 工具箱 filterDesigner 生成 IIR 高通滤波器系数

前面介绍 FIR 滤波器的时候，我们讲解了如何使用 filterDesigner 生成 C 头文件，从而获得滤波器系数。这里不能再使用这种方法了，主要是因为通过 C 头文件获取的滤波器系数需要通过 ARM 官方的 IIR 函数调用多次才能获得滤波结果，所以我们这里换另外一种方法。

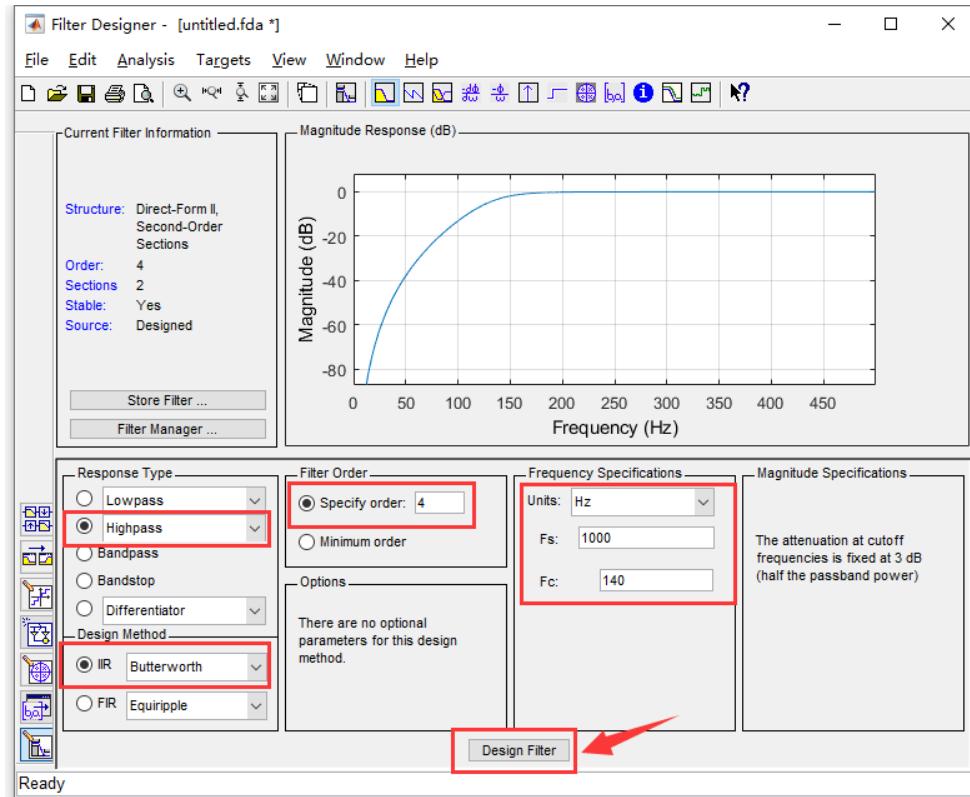
下面我们讲解如何通过 filterDesigner 工具箱生成滤波器系数。首先在 matlab 的命令窗口输入 filterDesigner 就能打开这个工具箱：



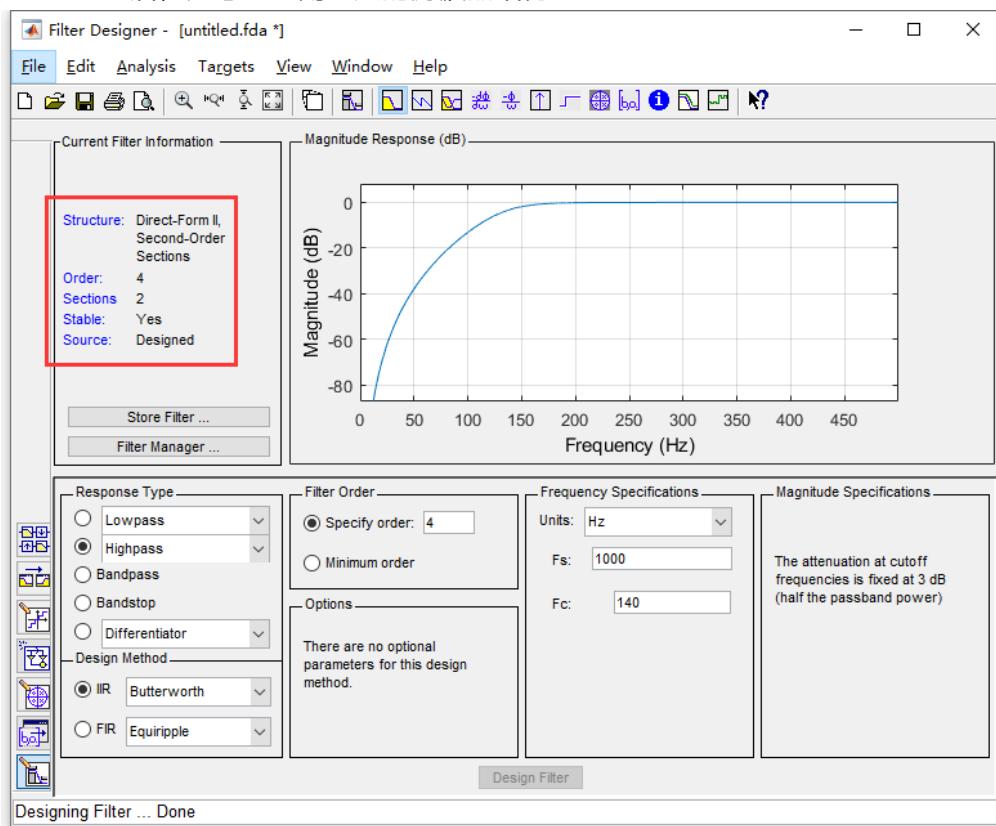
filterDesigner 界面打开效果如下：



IIR 滤波器的低通，高通，带通，带阻滤波的设置会在下面一一讲解，这里说一下设置后相应参数后如何生成滤波器系数。参数设置好以后点击如下按钮：

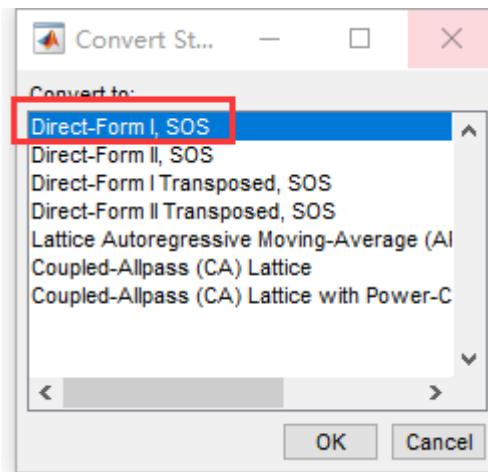


点击 Design Filter 之后，注意左上角生成的滤波器结构：

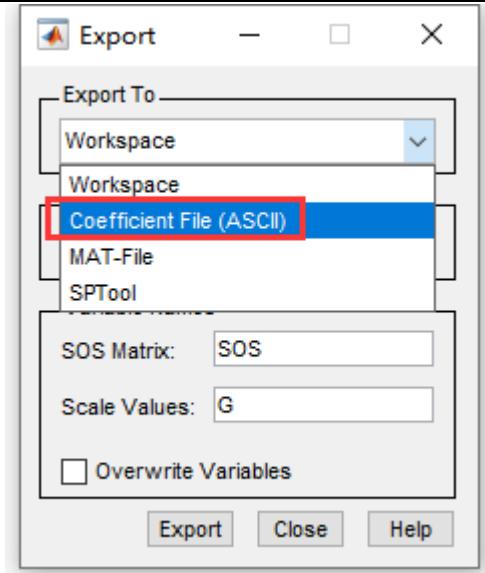


默认生成的 IIR 滤波器类型是 Direct-Form II, Second-Order Sections (直接 II 型，每个 Section 是一个二阶滤波器)。这里我们需要将其转换成 Direct-Form I, Second-Order Sections，因为本章使用的 IIR 滤波器函数是 Direct-Form I 的结构。

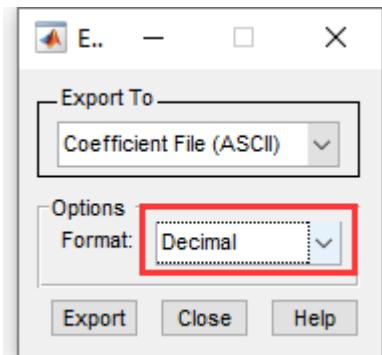
转换方法，点击 Edit->Convert Structure，界面如下，这里我们选择第一项，并点击 OK：



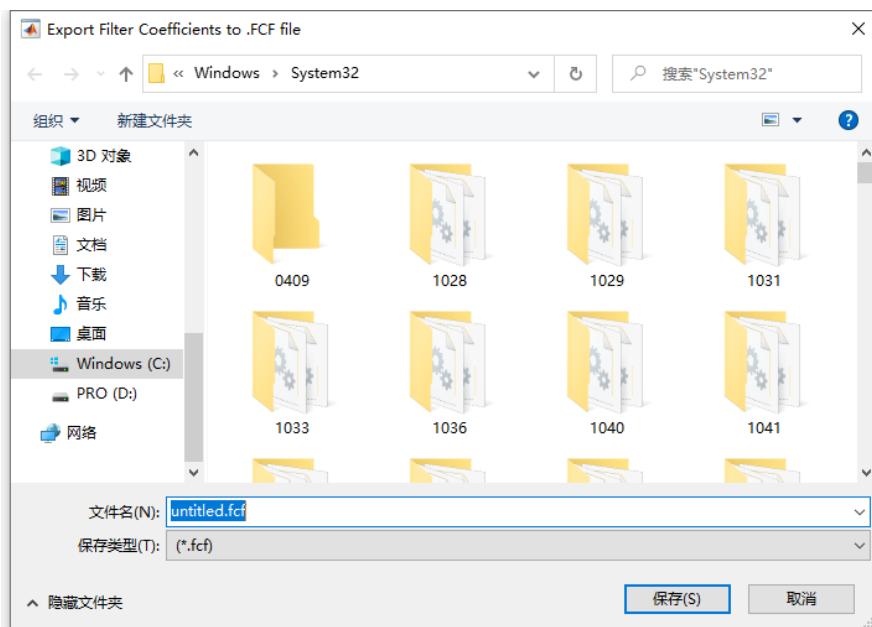
转换好以后再点击 File-Export，第一项选择 Coefficient File (ASCII)：



第一项选择好以后，第二项选择 Decimal：



两个选项都选择好以后，点击 Export 进行导出，导出后保存即可：



保存后 Matlab 会自动打开 untitled.fcf 文件，可以看到生成的系数：

```
% Generated by MATLAB(R) 9.4 and Signal Processing Toolbox 8.0.  
% Generated on: 15-Aug-2021 20:38:33
```



```
% Coefficient Format: Decimal
% Discrete-Time IIR Filter (real)
% -----
% Filter Structure      : Direct-Form I, Second-Order Sections
% Number of Sections   : 2
% Stable                : Yes
% Linear Phase          : No

SOS Matrix:
1 -2 1 1 -0.98454301474115180070612041163258254528 0.544565360850816415627662081533344462514
1 -2 1 1 -0.744714477864321211519893495278665795922 0.168318873843973093595849377379636280239

Scale Values:
0.632277093897992026327870007662568241358
0.478258337927073562401147910350118763745
```

由于前面选择的是 4 阶 IIR 滤波，生成的结果就是由两组二阶 IIR 滤波系数组成，系数的对应顺序如下：

```
SOS Matrix:
1 2 1 1 -0.98454301474115180070612041163258254528 0.544565360850816415627662081533344462514
b0 b1 b2 a0           a1                               a2
2 2 1 1 -0.744714477864321211519893495278665795922 0.168318873843973093595849377379636280239
b0 b1 b2 a0           a1                               a2
```

注意，实际使用 ARM 官方的 IIR 函数调用的时候要将 a1 和 a2 取反。另外下面两组是每个二阶滤波器的增益，滤波后的结果要乘以这两个增益数值才是实际结果：

```
0.632277093897992026327870007662568241358
0.478258337927073562401147910350118763745
```

实际的滤波系数调用方法，看下面的例子即可。

45.5 IIR 高通滤波器设计

本章使用的 IIR 滤波器函数是 arm_biquad_cascade_df1_f32。使用此函数可以设计 IIR 低通，高通，带通和带阻滤波器

45.5.1 函数 arm_biquad_cascade_df1_init_f32

函数原型：

```
void arm_biquad_cascade_df1_init_f32(
    arm_biquad_cascade_df1_inst_f32 * S,
    uint8_t numStages,
    const float32_t * pCoeffs,
    float32_t * pState)
```

函数描述：

这个函数用于 IIR 初始化。

函数参数：

- ◆ 第 1 个参数是 arm_biquad_cascade_df1_inst_f32 类型结构体变量。
- ◆ 第 2 个参数是 2 阶滤波器的个数。
- ◆ 第 3 个参数是滤波器系数地址。
- ◆ 第 4 个参数是缓冲状态地址。

注意事项：



结构体 arm_biquad_casd_df1_inst_f32 的定义如下 (在文件 filtering_functions.h 文件) :

```
typedef struct
{
    uint32_t numStages;      /*< number of 2nd order stages in the filter. Overall order is 2*numStages. */
    float32_t *pState; /*< Points to the array of state coefficients. The array is of length 4*numStages. */
    const float32_t *pCoeffs; /*< Points to the array of coefficients. The array is of length 5*numStages */
} arm_biquad_casd_df1_inst_f32;
```

4. numStages 表示二阶滤波器的个数, 总阶数是 2*numStages。
5. pState 指向状态变量数组, 这个数组用于函数内部计算数据的缓存, 总大小 4*numStages。
6. 参数 pCoeffs 指向滤波因数, 滤波因数数组长度为 5*numStages。但要注意 pCoeffs 指向的滤波因数应该按照如下的逆序进行排列:
{b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ...}
先放第一个二阶 Biquad 系数, 然后放第二个, 以此类推。

45.5.2 函数 arm_biquad_cascade_df1_f32

函数定义如下:

```
void arm_biquad_cascade_df1_f32(
    const arm_biquad_casd_df1_inst_f32 * S,
    float32_t * pSrc,
    float32_t * pDst,
    uint32_t blockSize)
```

函数描述:

这个函数用于 IIR 滤波。

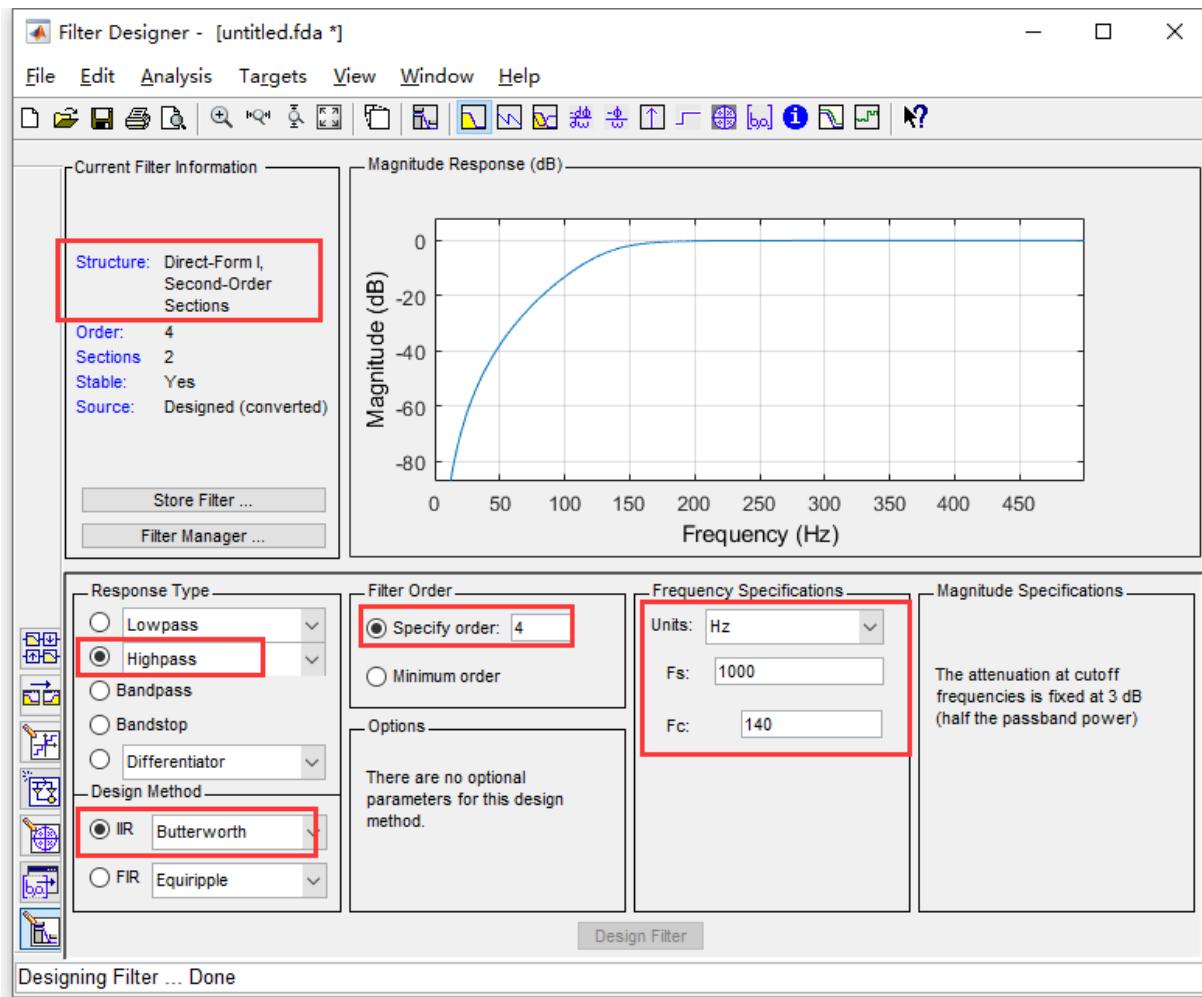
函数参数:

- ◆ 第 1 个参数是 arm_biquad_casd_df1_inst_f32 类型结构体变量。
- ◆ 第 2 个参数是源数据地址。
- ◆ 第 3 个参数是滤波后的数据地址。
- ◆ 第 4 个参数是每次调用处理的数据个数, 最小可以每次处理 1 个数据, 最大可以每次全部处理完。

45.5.3 filterDesigner 获取高通滤波器系数

设计一个如下的例子:

信号由 50Hz 正弦波和 200Hz 正弦波组成, 采样率 1Kbps, 现设计一个巴特沃斯滤波器高通滤波器, 采用直接 I 型, 截止频率 140Hz, 采样 400 个数据, 滤波器阶数设置为 4。filterDesigner 的配置如下:



配置好高通滤波器后，具体滤波器系数的生成大家参考本章第4小节的方法即可。

45.5.4 高通滤波器实现

通过工具箱 filterDesigner 获得高通滤波器系数后在开发板上运行函数 arm_biquad_cascade_df1_f32 来测试低通滤波器的效果。

```
#define numStages 2           /* 2阶IIR滤波的个数 */
#define TEST_LENGTH_SAMPLES 400 /* 采样点数 */
#define BLOCK_SIZE 1           /* 调用一次arm_biquad_cascade_df1_f32处理的采样点个数 */

uint32_t blockSize = BLOCK_SIZE;
uint32_t numBlocks = TEST_LENGTH_SAMPLES/BLOCK_SIZE;           /* 需要调用arm_biquad_cascade_df1_f32的次数 */

static float32_t testInput_f32_50Hz_200Hz[TEST_LENGTH_SAMPLES]; /* 采样点 */
static float32_t testOutput[TEST_LENGTH_SAMPLES];                /* 滤波后的输出 */
static float32_t IIRStateF32[4*numStages];                    /* 状态缓存 */

/* 巴特沃斯高通滤波器系数 140Hz */
const float32_t IIRCoeffs32HP[5*numStages] = {
    1.0f, -2.0f, 1.0f, 0.98454301474115180070612041163258254528f,
    -0.544565360850816415627662081533344462514f,
```



```
1.0f, -2.0f, 1.0f, 0.744714477864321211519893495278665795922f,
                                         -0.168318873843973093595849377379636280239
};

/*
***** 函数名: arm_iir_f32_hp
***** 功能说明: 调用函数 arm_iir_f32_hp 实现高通滤波器
***** 形参: 无
***** 返回值: 无
*****
*/
static void arm_iir_f32_hp(void)
{
    uint32_t i;
    arm_biquad_cascade_df1_inst_f32 S;
    float32_t ScaleValue;
    float32_t *inputF32, *outputF32;

    /* 初始化输入输出缓存指针 */
    inputF32 = &testInput_f32_50Hz_200Hz[0];
    outputF32 = &testOutput[0];

    /* 初始化 */
    arm_biquad_cascade_df1_init_f32(&S, numStages, (float32_t *)&IIRCoeffs32HP[0],
                                     (float32_t *)&IIRStateF32[0]);

    /* 实现 IIR 滤波, 这里每次处理 1 个点 */
    for(i=0; i < numBlocks; i++)
    {
        arm_biquad_cascade_df1_f32(&S, inputF32 + (i * blockSize), outputF32 + (i * blockSize),
                                   blockSize);
    }

    /* 放缩系数 */
    ScaleValue = 0.632277093897992026327870007662568241358f * 0.478258337927073562401147910350118763745f;

    /* 打印滤波后结果 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        printf("%f, %f\r\n", testInput_f32_50Hz_200Hz[i], testOutput[i]*ScaleValue);
    }
}
```

运行如上函数可以通过串口打印出函数arm_biquad_cascade_df1_f32滤波后的波形数据，下面通过Matlab绘制波形来对比Matlab计算的结果和ARM官方库计算的结果。

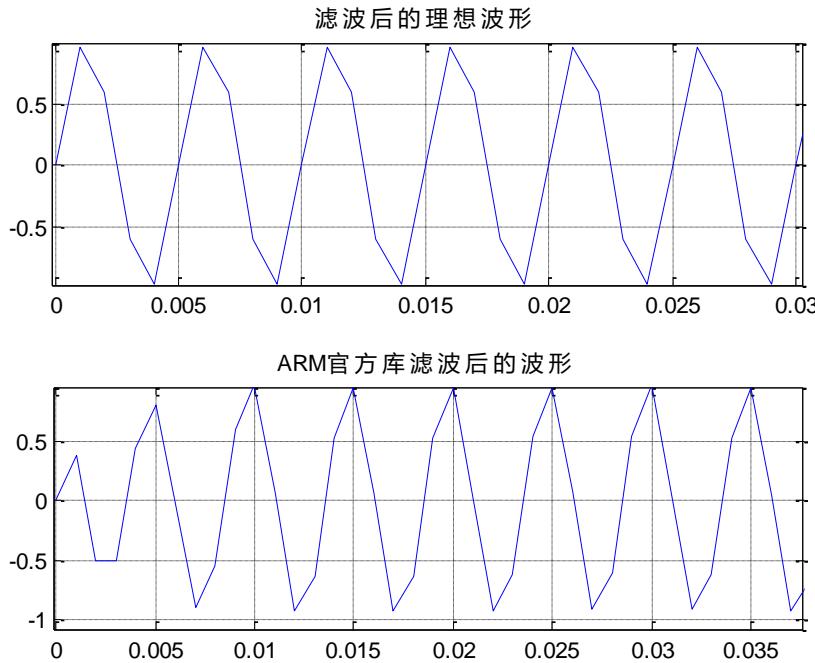
对比前需要先将串口打印出的一组数据加载到 Matlab 中，arm_biquad_cascade_df1_f32 的计算结果起名 sampledata，加载方法在[第 13 章 13.6 小结已经讲解](#)，这里不做赘述了。Matlab 中运行的代码如下：

```
fs=1000;          %设置采样频率 1K
N=400;           %采样点数
n=0:N-1;
t=n/fs;          %时间序列
f=n*fs/N;         %频率序列

x1=sin(2*pi*50*t);
x2=sin(2*pi*200*t);    %50Hz和200Hz正弦波
subplot(211);
plot(t, x2);
title('滤波后的理想波形');
grid on;
```

```
subplot(212);
plot(t, sampledata);
title('ARM官方库滤波后的波形');
grid on;
```

Matlab 计算结果如下：



从上面的波形对比来看，matlab 和函数 arm_biquad_cascade_df1_f32 计算的结果基本是一致的。为了更好的说明滤波效果，下面从频域的角度来说明这个问题，Matlab 上面运行如下代码：

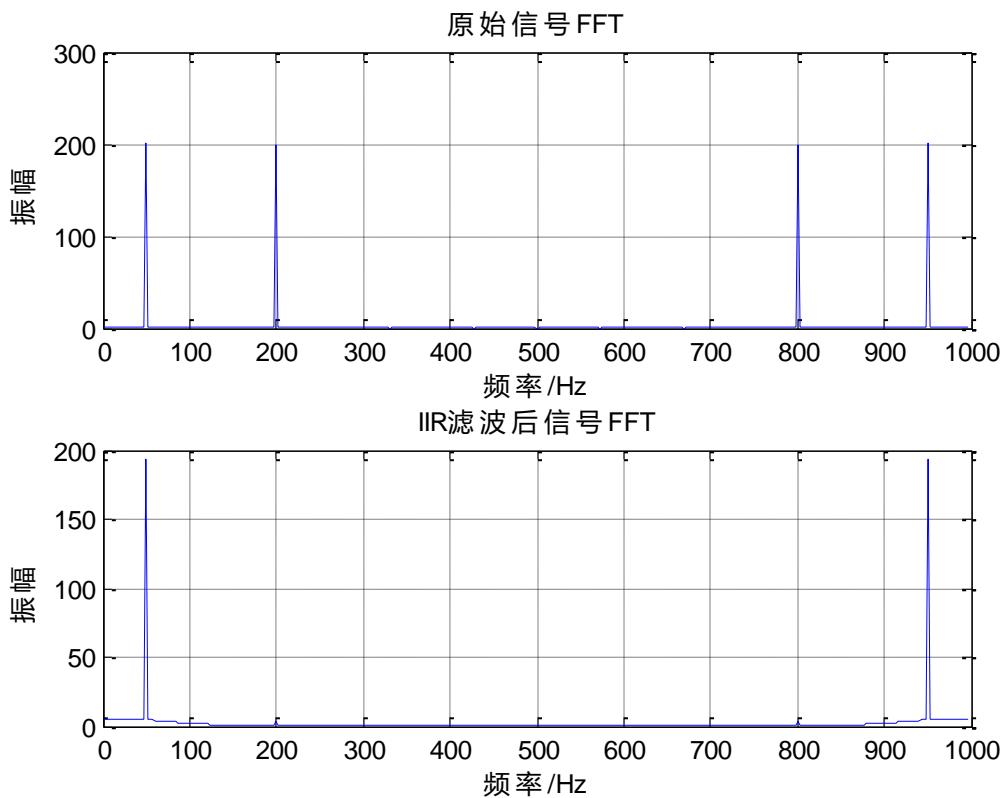
```
fs=1000; %设置采样频率 1K
N=400; %采样点数
n=0:N-1;
t=n/fs; %时间序列
f=n*fs/N; %频率序列

x = sin(2*pi*50*t) + sin(2*pi*200*t); %50Hz和200Hz正弦波合成

subplot(211);
y=fft(x, N); %对信号x做FFT
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号FFT');
grid on;

y3=fft(sampledata, N); %经过IIR滤波器后得到的信号做FFT
subplot(212);
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('IIR滤波后信号FFT');
grid on;
```

Matlab 计算结果如下：



上面波形变换前的 FFT 和变换后 FFT 可以看出，50Hz 的正弦波基本被滤除。

45.6 实验例程说明 (MDK)

配套例子：

V7-230_IIR 高通滤波器(支持逐点实时滤波)

实验目的：

1. 学习 IIR 高通滤波器的实现，支持实时滤波

实验内容：

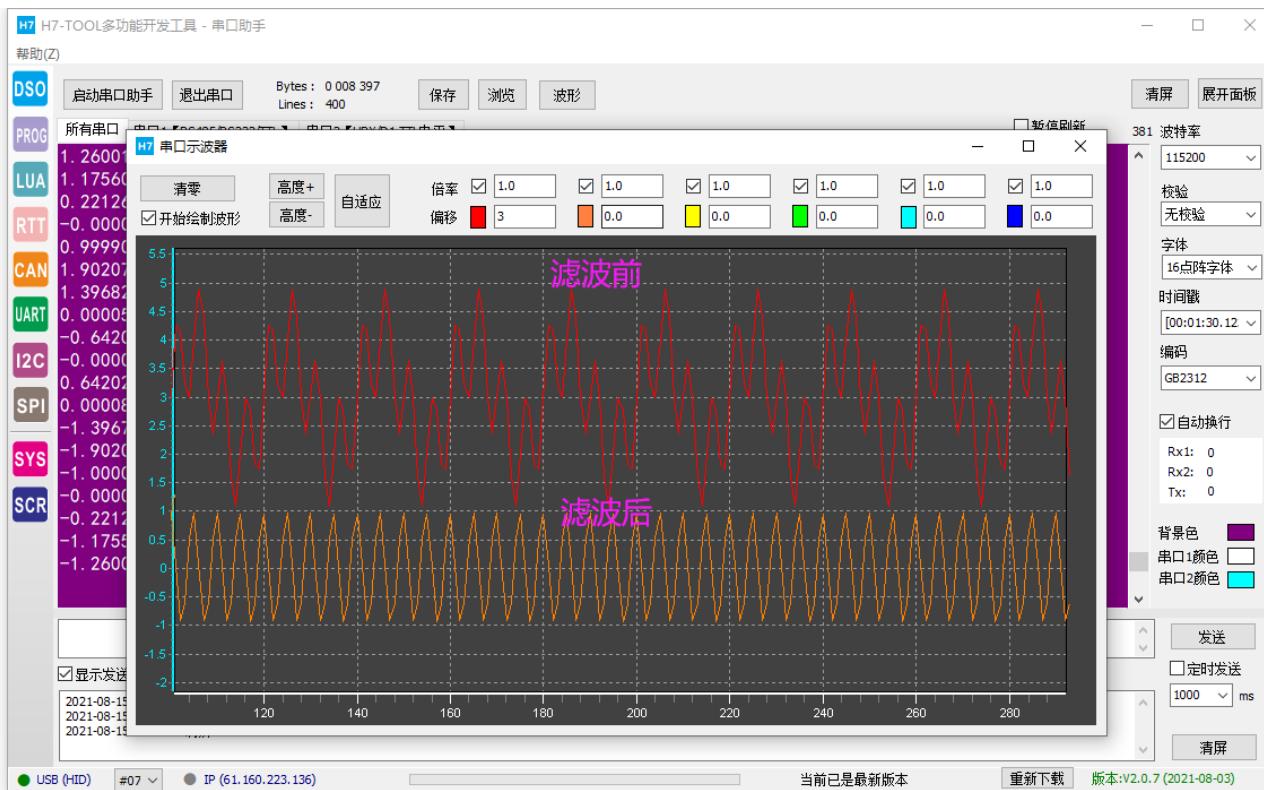
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

使用 AC6 注意事项

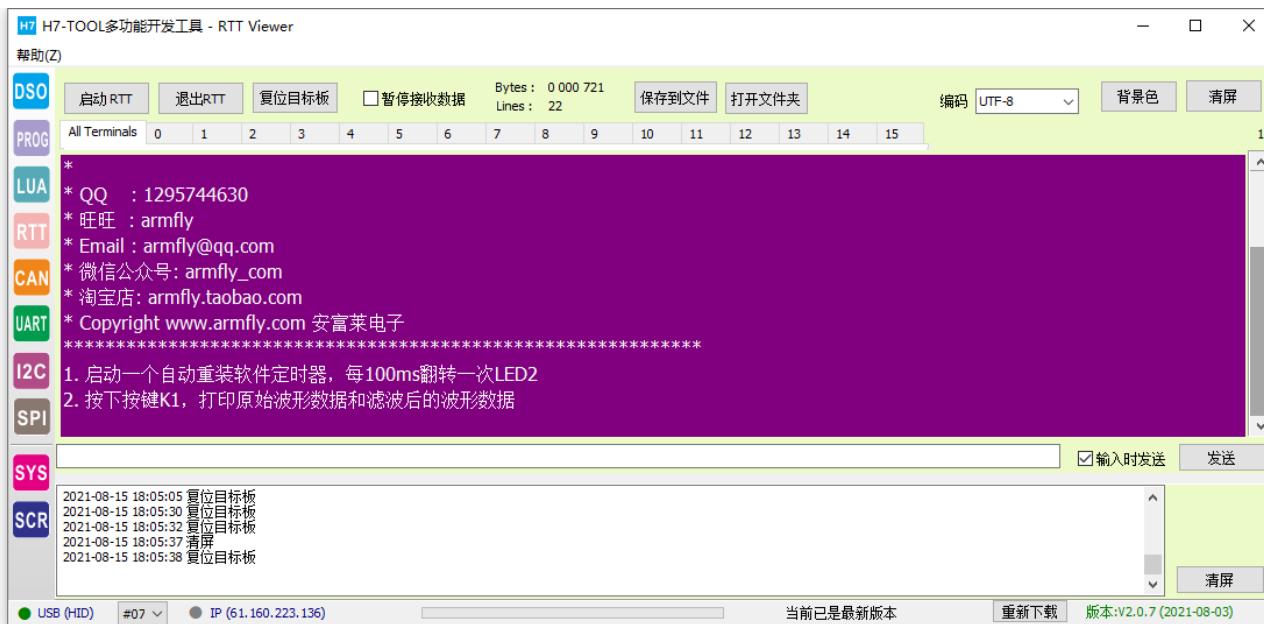
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

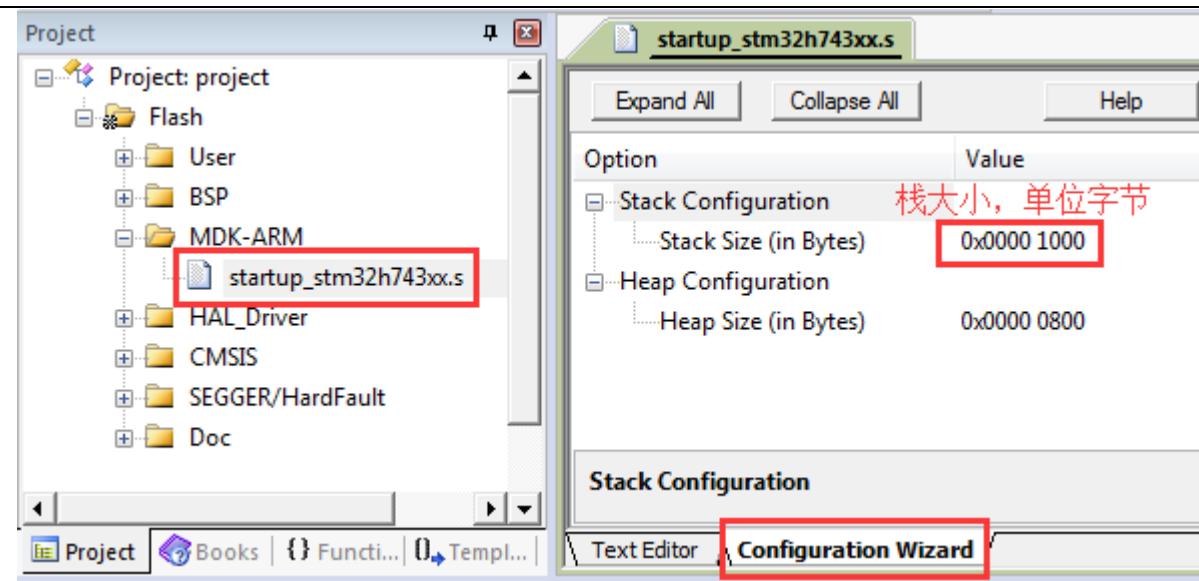


RTT 方式打印信息：

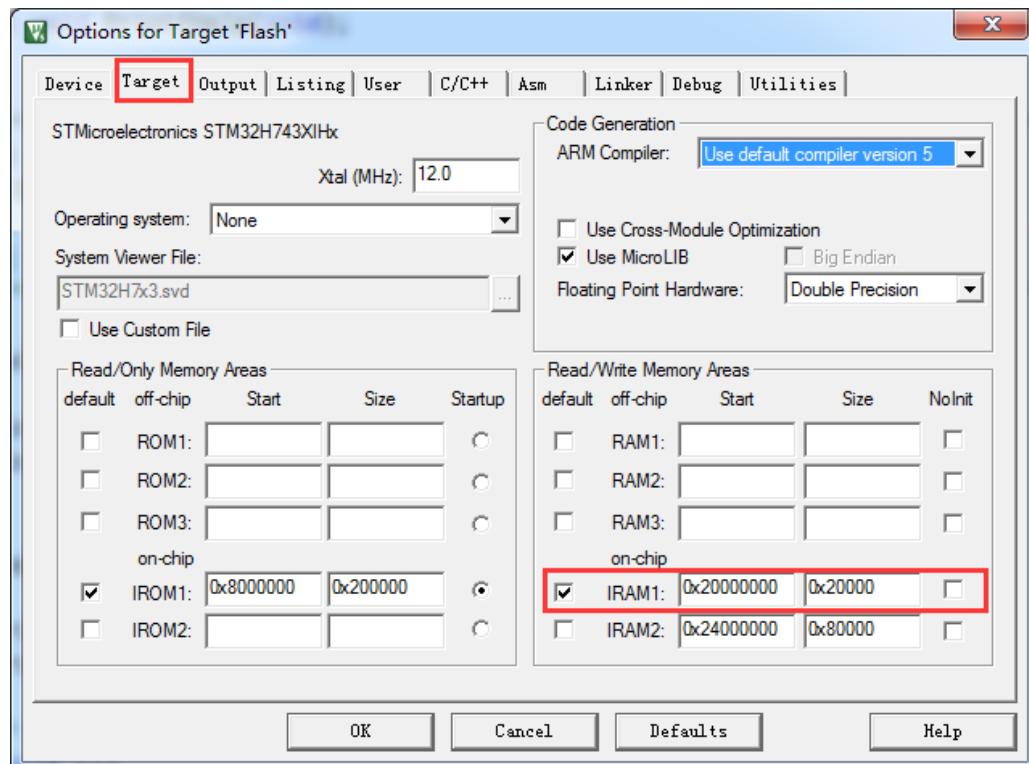


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现:

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
```



```
{  
    /* 配置 MPU */  
    MPU_Config();  
  
    /* 使能 L1 Cache */  
    CPU_CACHE_Enable();  
  
    /*  
     * STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：  
     * - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。  
     * - 设置 NVIC 优先级分组为 4。  
     */  
    HAL_Init();  
  
    /*  
     * 配置系统时钟到 400MHz  
     * - 切换使用 HSE。  
     * - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。  
     */  
    SystemClock_Config();  
  
    /*  
     * Event Recorder:  
     * - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。  
     * - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章  
     */  
#if Enable_EventRecorder == 1  
    /* 初始化 EventRecorder 并开启 */  
    EventRecorderInitialize(EventRecordAll, 1U);  
    EventRecorderStart();  
#endif  
  
    bsp_InitKey(); /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */  
    bsp_InitTimer(); /* 初始化滴答定时器 */  
    bsp_InitUart(); /* 初始化串口 */  
    bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */  
    bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();
```



```
/* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。



- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
***** */

int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
        testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                       arm_sin_f32(2*3.1415926f*200*i/1000);
    }

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2); /* 翻转 LED 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_iir_f32_hp();
                    break;

                default:
                    /* 其它的键值不处理 */
                    break;
            }
        }
    }
}
```

{
}

45.7 实验例程说明 (IAR)

配套例子：

V7-230_IIR 高通滤波器(支持逐点实时滤波)

实验目的：

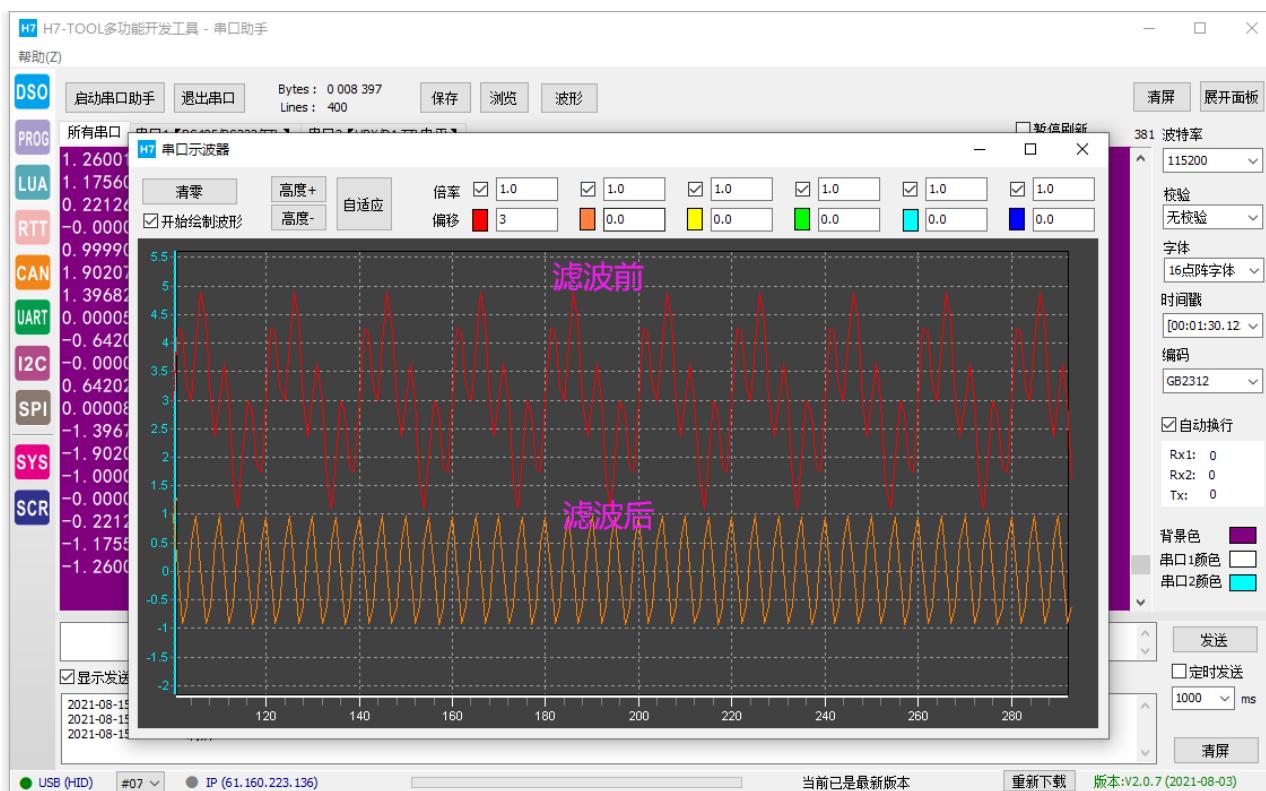
1. 学习 IIR 高通滤波器的实现，支持实时滤波

实验内容：

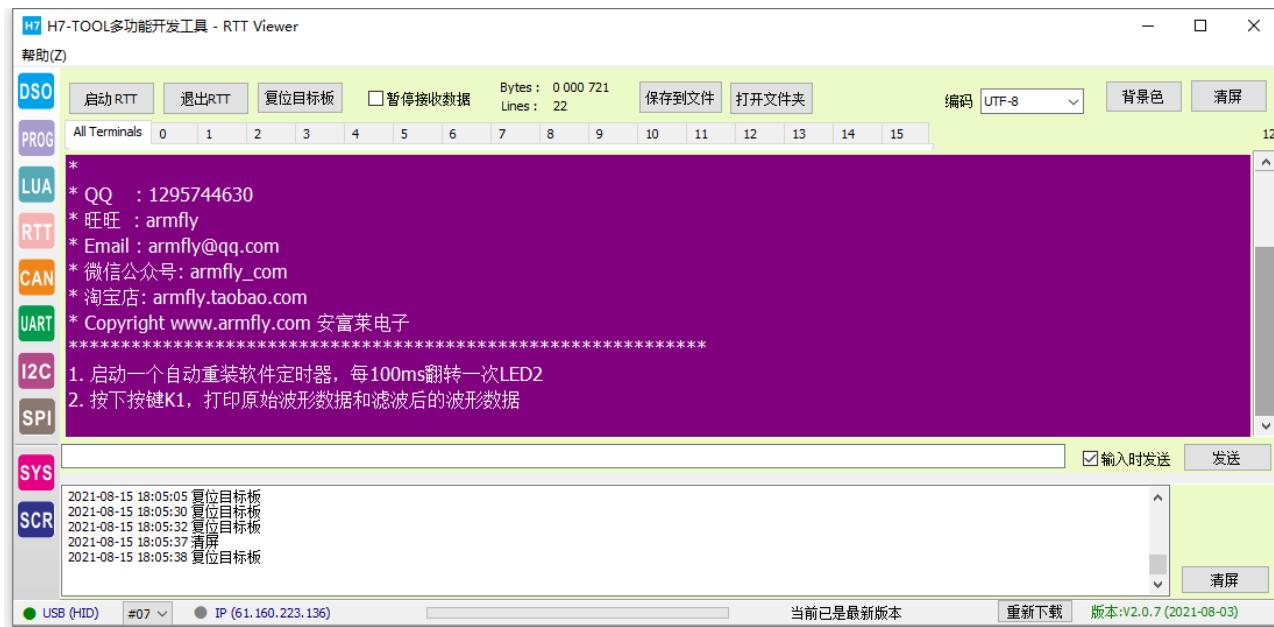
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

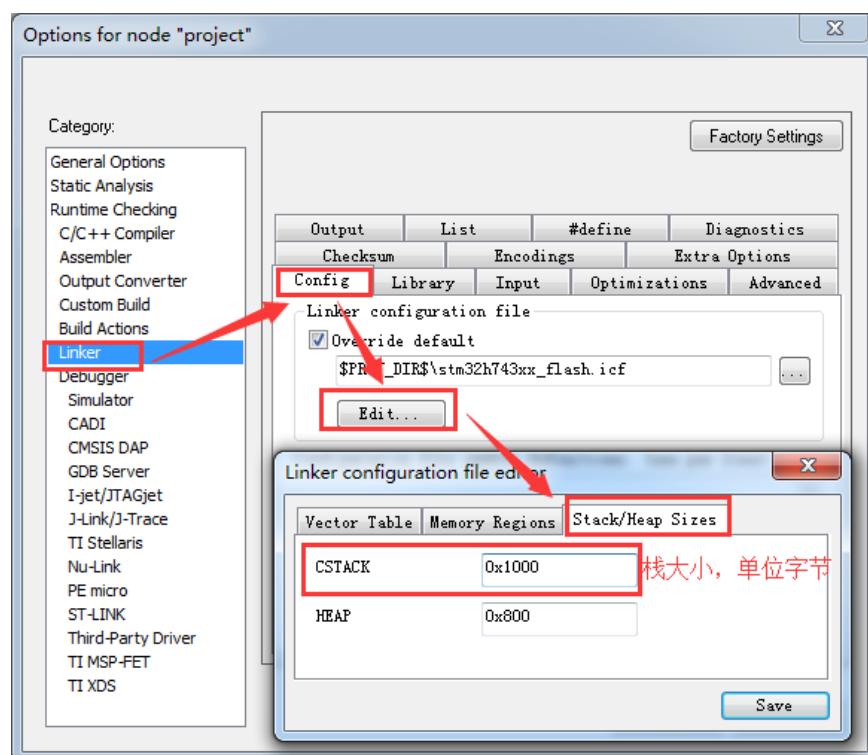


RTT 方式打印信息：

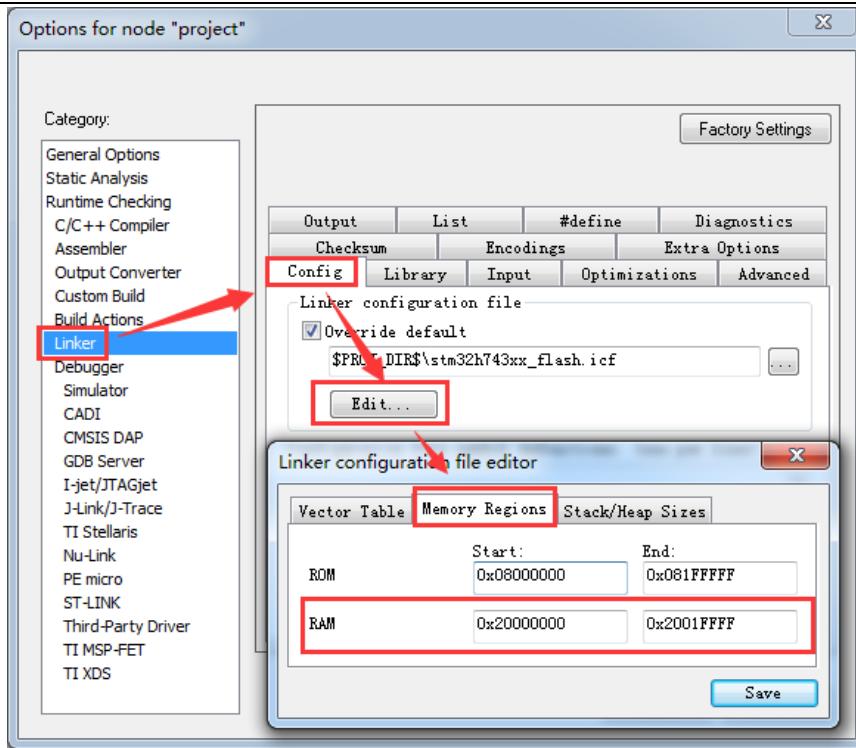


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*

```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec   = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波，采样率 1KHz */
    }
}
```



```
testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                arm_sin_f32(2*3.1415926f*200*i/1000);
}

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2); /* 翻转 LED 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_iir_f32_hp();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
}
```

45.8 总结

本章节主要讲解了 IIR 滤波器的高通实现，同时一定要注意 IIR 滤波器的群延迟问题，详见本教程的第 41 章。



第46章 STM32H7 的 IIR 带通滤波器实现（支持逐个数据的实时滤波）

本章节讲解 IIR 带通滤波器实现。

46.1 初学者重要提示

46.2 带通滤波器介绍

46.3 IIR 滤波器介绍

46.4 Matlab 工具箱 filterDesigner 生成带通滤波器 C 头文件

46.5 IIR 带通滤波器设计

46.6 实验例程说明 (MDK)

46.7 实验例程说明 (IAR)

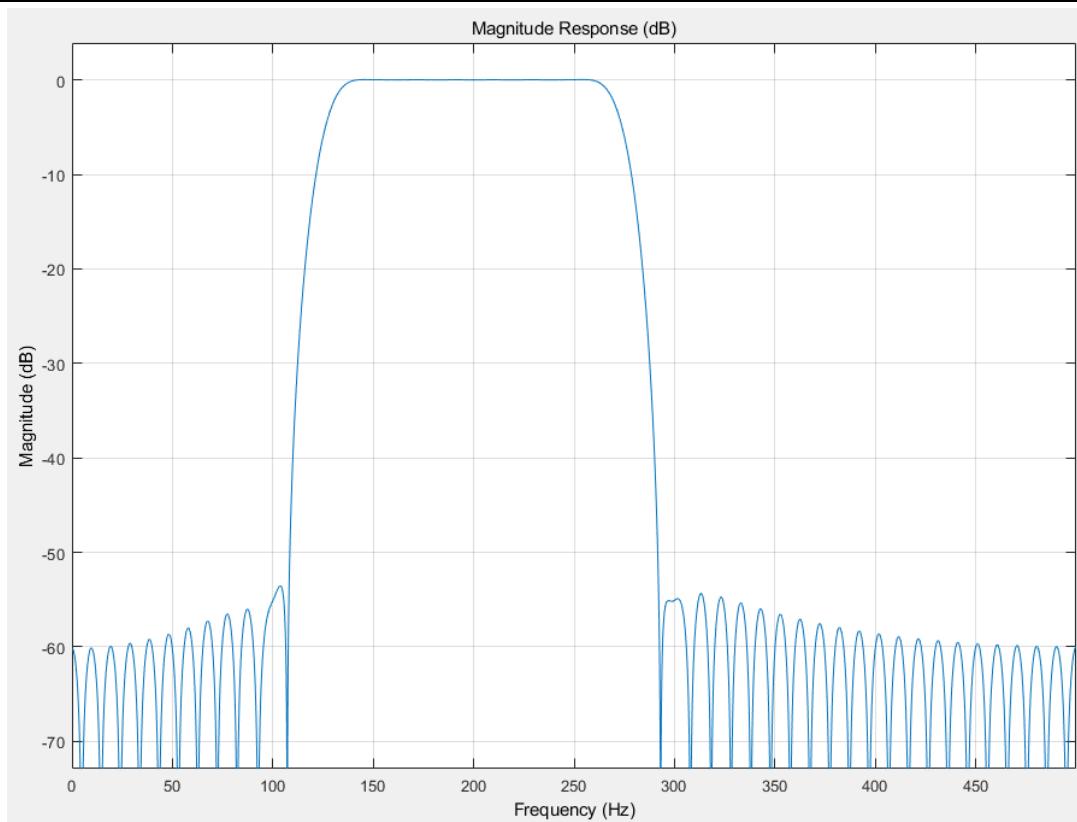
46.8 总结

46.1 初学者重要提示

- ◆ 本章节提供的带通滤波器支持实时滤波，每次可以滤波一个数据，也可以多个数据，不限制大小。但要注意以下两点：
 - 所有数据是在同一个采样率下依次采集的数据。
 - 每次过滤数据个数一旦固定下来，运行中不可再修改。
- ◆ FIR 滤波器的群延迟是一个重要的知识点，详情在本教程第 41 章有详细说明。IIR 和 FIR 一样，也有群延迟问题。

46.2 带通滤波器介绍

允许一个范围内的频率信号通过，而减弱范围之外频率的信号通过。比如混合信号含有 50Hz + 200Hz 信号，我们可通过带通滤波器，仅让 200Hz 信号通过。



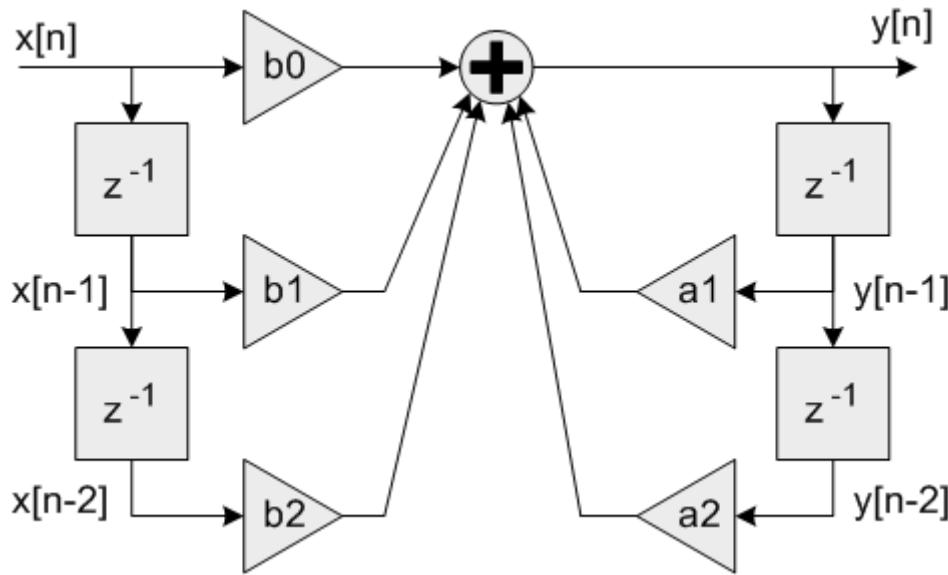
46.3 IIR 滤波器介绍

ARM 官方提供的直接 I 型 IIR 库支持 Q7, Q15, Q31 和浮点四种数据类型。其中 Q15 和 Q31 提供了快速版本。

直接 I 型 IIR 滤波器是基于二阶 Biquad 级联的方式来实现的。每个 Biquad 由一个二阶的滤波器组成：

$$y[n] = b_0 * x[n] + b_1 * x[n-1] + b_2 * x[n-2] + a_1 * y[n-1] + a_2 * y[n-2]$$

直接 I 型算法每个阶段需要 5 个系数和 4 个状态变量。

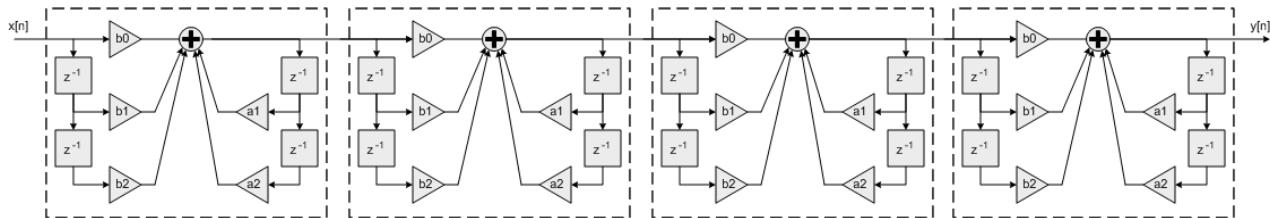


这里有一点要特别的注意，有些滤波器系数生成工具是采用的下面公式实现：

$$y[n] = b_0 * x[n] + b_1 * x[n-1] + b_2 * x[n-2] - a_1 * y[n-1] - a_2 * y[n-2]$$

比如 matlab 就是使用上面的公式实现的，所以在使用 fdatool 工具箱生成的 a 系数需要取反才能用于直接 I 型 IIR 滤波器的函数中。

高阶 IIR 滤波器的实现是采用二阶 Biquad 级联的方式来实现的。其中参数 numStages 就是用来做指定二阶 Biquad 的个数。比如 8 阶 IIR 滤波器就可以采用 numStages=4 个二阶 Biquad 来实现。



如果要实现 9 阶 IIR 滤波器就需要将 numStages=5，这时就需要其中一个 Biquad 配置成一阶滤波器(也就是 b2=0, a2=0)。

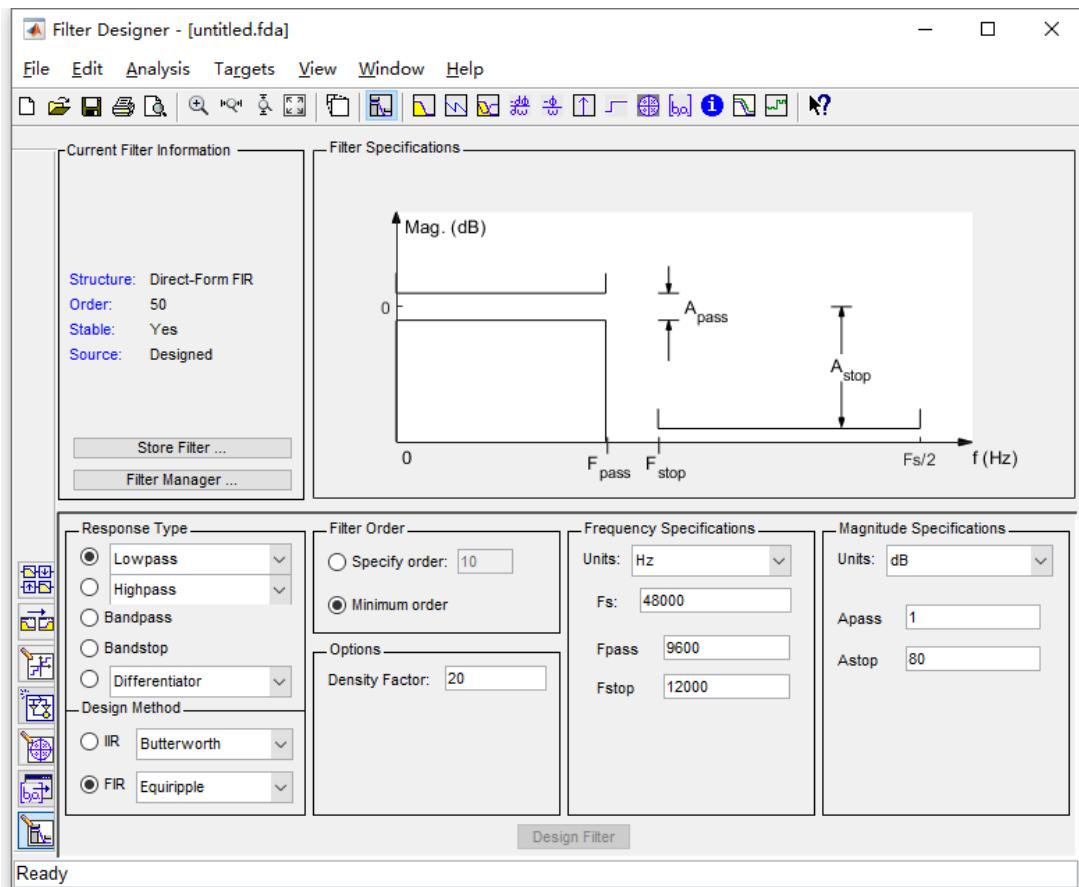
46.4 Matlab 工具箱 filterDesigner 生成 IIR 带通滤波器系数

前面介绍 FIR 滤波器的时候，我们讲解了如何使用 filterDesigner 生成 C 头文件，从而获得滤波器系数。这里不能再使用这种方法了，主要是因为通过 C 头文件获取的滤波器系数需要通过 ARM 官方的 IIR 函数调用多次才能获得滤波结果，所以我们这里换另外一种方法。

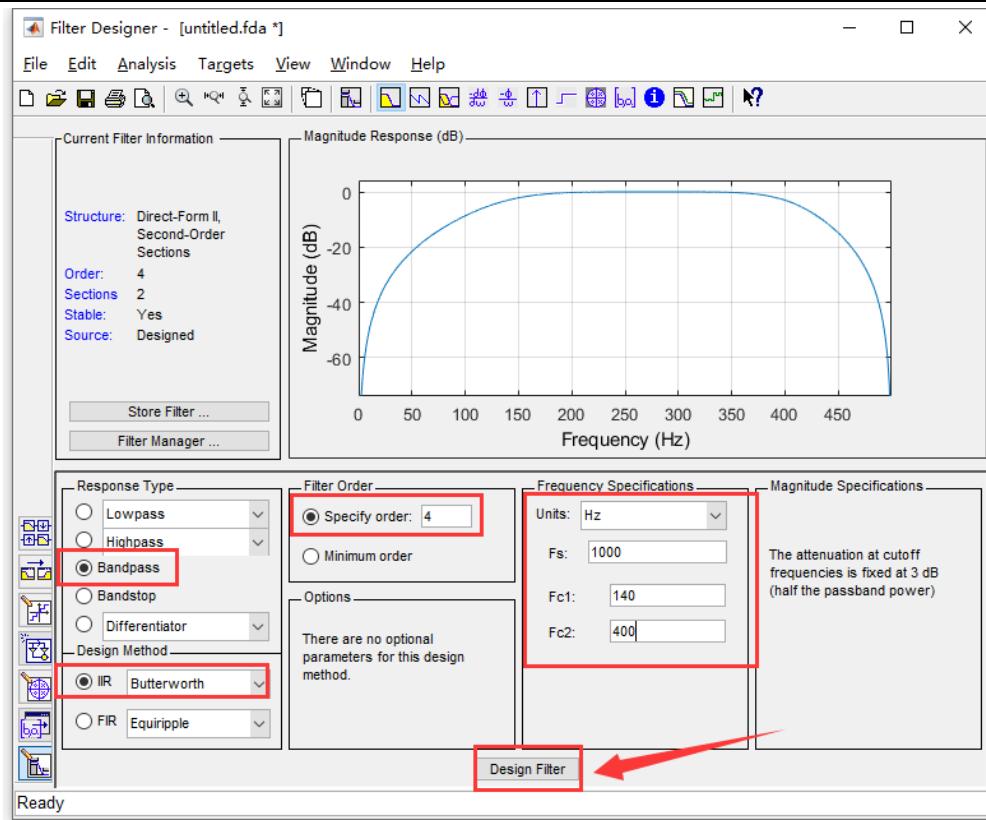
下面我们讲解如何通过 filterDesigner 工具箱生成滤波器系数。首先在 matlab 的命令窗口输入 filterDesigner 就能打开这个工具箱：



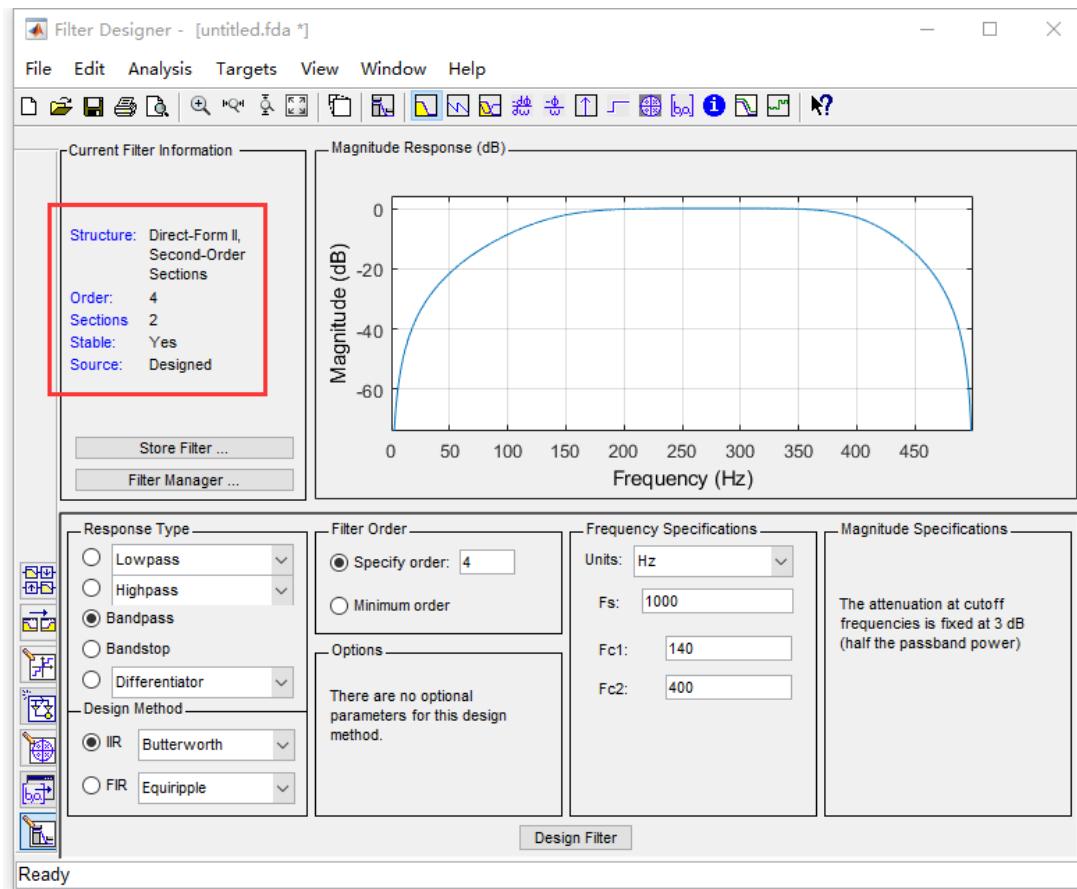
filterDesigner 界面打开效果如下：



IIR 滤波器的低通，高通，带通，带阻滤波的设置会在下面一一讲解，这里说一下设置后相应参数后如何生成滤波器系数。参数设置好以后点击如下按钮：

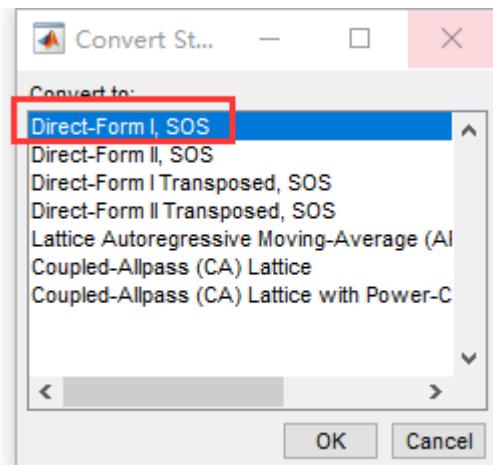


点击 Design Filter 之后，注意左上角生成的滤波器结构：

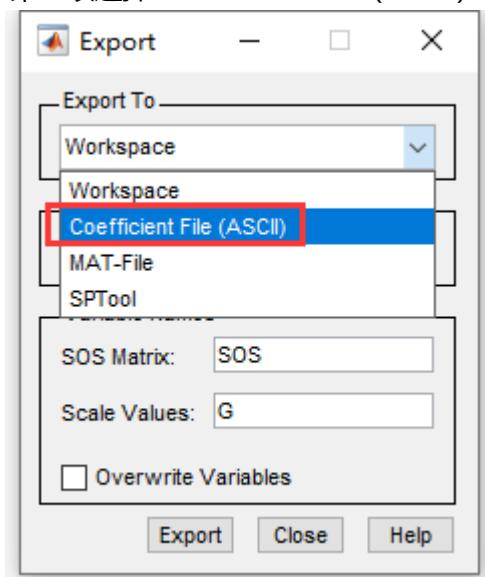


默认生成的 IIR 滤波器类型是 Direct-Form II, Second-Order Sections (直接 II 型，每个 Section 是一个二阶滤波器)。这里我们需要将其转换成 Direct-Form I, Second-Order Sections，因为本章使用的 IIR 滤波器函数是 Direct-Form I 的结构。

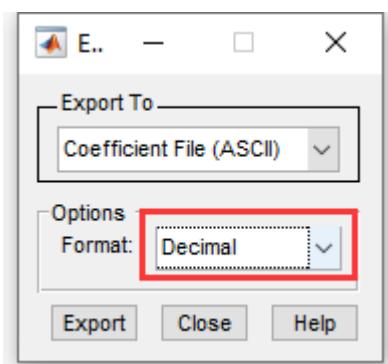
转换方法，点击 Edit->Convert Structure，界面如下，这里我们选择第一项，并点击 OK：



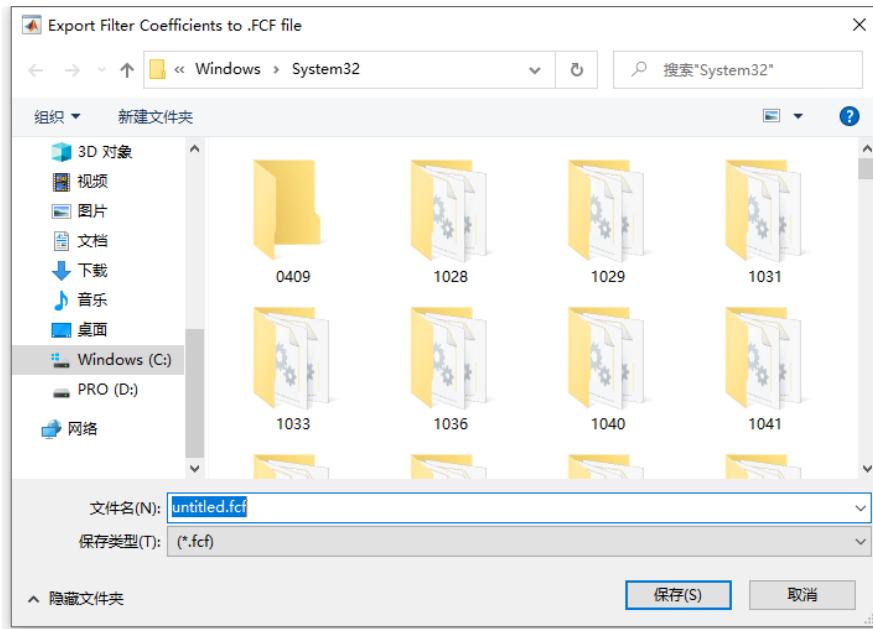
转换好以后再点击 File-Export，第一项选择 Coefficient File (ASCII)：



第一项选择好以后，第二项选择 Decimal：



两个选项都选择好以后，点击 Export 进行导出，导出后保存即可：



保存后 Matlab 会自动打开 `untitled.fcf` 文件，可以看到生成的系数：

```
% Generated by MATLAB(R) 9.4 and Signal Processing Toolbox 8.0.  
% Generated on: 15-Aug-2021 22:20:26  
  
% Coefficient Format: Decimal  
  
% Discrete-Time IIR Filter (real)  
% -----  
% Filter Structure : Direct-Form I, Second-Order Sections  
% Number of Sections : 2  
% Stable : Yes  
% Linear Phase : No  
  
SOS Matrix:  
1 0 -1 1 1.127651872054164616798743736580945551395 0.470013145087532668853214090631809085608  
1 0 -1 1 -0.774953058046049081397654845204669982195 0.367077500556684199750634434167295694351  
  
Scale Values:  
0.558156585760773649163013487850548699498  
0.558156585760773649163013487850548699498
```

由于前面选择的是 4 阶 IIR 滤波，生成的结果就是由两组二阶 IIR 滤波系数组成，系数的对应顺序如下：

```
SOS Matrix:  
1 2 1 1 1.127651872054164616798743736580945551395 0.470013145087532668853214090631809085608  
b0 b1 b2 a0 a1 a2  
3 2 1 1 -0.774953058046049081397654845204669982195 0.367077500556684199750634434167295694351  
b0 b1 b2 a0 a1 a2
```

注意，实际使用 ARM 官方的 IIR 函数调用的时候要将 `a1` 和 `a2` 取反。另外下面两组是每个二阶滤波器的

增益，滤波后的结果要乘以这两个增益数值才是实际结果：

```
0.558156585760773649163013487850548699498  
0.558156585760773649163013487850548699498
```

实际的滤波系数调用方法，看下面的例子即可。



46.5 IIR 带通滤波器设计

本章使用的 IIR 滤波器函数是 arm_biquad_cascade_df1_f32。使用此函数可以设计 IIR 低通，高通，带通和带阻滤波器

46.5.1 函数 arm_biquad_cascade_df1_init_f32

函数原型：

```
void arm_biquad_cascade_df1_init_f32(
    arm_biquad_casd_df1_inst_f32 * S,
    uint8_t numStages,
    const float32_t * pCoeffs,
    float32_t * pState)
```

函数描述：

这个函数用于 IIR 初始化。

函数参数：

- ◆ 第 1 个参数是 arm_biquad_casd_df1_inst_f32 类型结构体变量。
- ◆ 第 2 个参数是 2 阶滤波器的个数。
- ◆ 第 3 个参数是滤波器系数地址。
- ◆ 第 4 个参数是缓冲状态地址。

注意事项：

结构体 arm_biquad_casd_df1_inst_f32 的定义如下 (在文件 filtering_functions.h 文件) :

```
typedef struct
{
    uint32_t numStages;      /*< number of 2nd order stages in the filter. Overall order is 2*numStages. */
    float32_t *pState; /*< Points to the array of state coefficients. The array is of length 4*numStages. */
    const float32_t *pCoeffs; /*< Points to the array of coefficients. The array is of length 5*numStages */
} arm_biquad_casd_df1_inst_f32;
```

7. numStages 表示二阶滤波器的个数，总阶数是 2*numStages。
8. pState 指向状态变量数组，这个数组用于函数内部计算数据的缓存，总大小 4*numStages。
9. 参数 pCoeffs 指向滤波因数，滤波因数数组长度为 5*numStages。但要注意 pCoeffs 指向的滤波因数应该按照如下的逆序进行排列：

{b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ...}

先放第一个二阶 Biquad 系数，然后放第二个，以此类推。

46.5.2 函数 arm_biquad_cascade_df1_f32

函数定义如下：

```
void arm_biquad_cascade_df1_f32(
    const arm_biquad_casd_df1_inst_f32 * S,
    float32_t * pSrc,
```

```
float32_t * pDst,  
uint32_t blockSize)
```

函数描述：

这个函数用于 IIR 滤波。

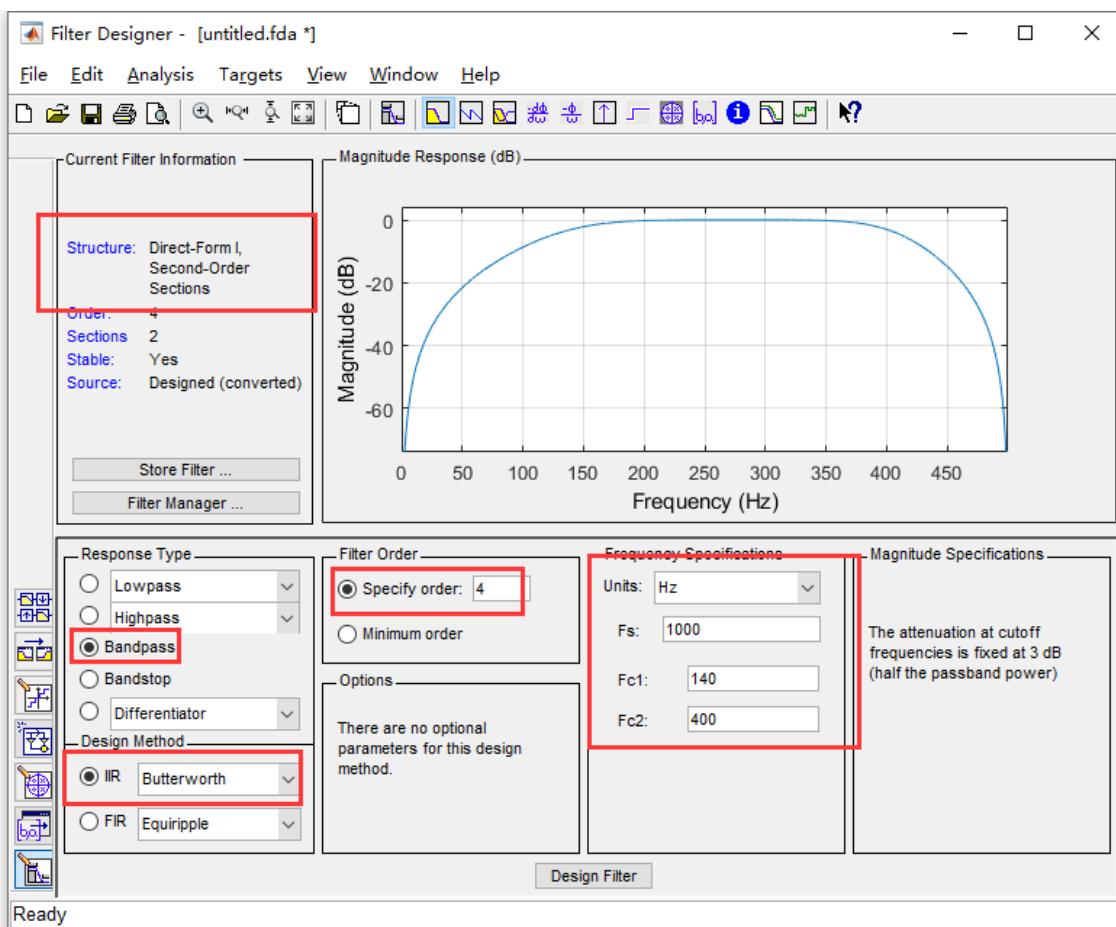
函数参数：

- ◆ 第 1 个参数是 arm_biquad_casd_df1_inst_f32 类型结构体变量。
- ◆ 第 2 个参数是源数据地址。
- ◆ 第 3 个参数是滤波后的数据地址。
- ◆ 第 4 个参数是每次调用处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

46.5.3 filterDesigner 获取带通滤波器系数

设计一个如下的例子：

信号由 50Hz 正弦波和 200Hz 正弦波组成，采样率 1Kbps，现设计一个巴特沃斯滤波器带通滤波器，采用直接 I 型，截止频率 140Hz 和 400，采样 400 个数据，滤波器阶数设置为 4。filterDesigner 的配置如下：



配置好带通滤波器后，具体滤波器系数的生成大家参考本章第 4 小节的方法即可。



46.5.4 带通滤波器实现

通过工具箱 filterDesigner 获得带通滤波器系数后在开发板上运行函数 arm_biquad_cascade_df1_f32 来测试低通滤波器的效果。

```
#define numStages 2           /* 2阶IIR滤波的个数 */
#define TEST_LENGTH_SAMPLES 400 /* 采样点数 */
#define BLOCK_SIZE 1           /* 调用一次arm_biquad_cascade_df1_f32处理的采样点个数 */

uint32_t blockSize = BLOCK_SIZE;
uint32_t numBlocks = TEST_LENGTH_SAMPLES/BLOCK_SIZE;           /* 需要调用 arm_biquad_cascade_df1_f32 的次数 */

static float32_t testInput_f32_50Hz_200Hz[TEST_LENGTH_SAMPLES]; /* 采样点 */
static float32_t testOutput[TEST_LENGTH_SAMPLES];                /* 滤波后的输出 */
static float32_t IIRStateF32[4*numStages];                    /* 状态缓存 */

/* 巴特沃斯带通滤波器系数 140Hz 400Hz*/
const float32_t IIRCoeffs32BP[5*numStages] = {
    1.0f, 0.0f, -1.0f, -1.127651872054164616798743736580945551395f,
    0.0f, 0.0f, 0.0f, -0.470013145087532668853214090631809085608f,
    1.0f, 0.0f, -1.0f, 0.774953058046049081397654845204669982195f,
    0.0f, 0.0f, 0.0f, -0.367077500556684199750634434167295694351f
};

/*
*****
* 函数名: arm_iir_f32_bp
* 功能说明: 调用函数 arm_iir_f32_bp 实现带通滤波器
* 形参: 无
* 返回值: 无
*****
*/
static void arm_iir_f32_bp(void)
{
    uint32_t i;
    arm_biquad_cascade_df1_inst_f32 S;
    float32_t ScaleValue;
    float32_t *inputF32, *outputF32;

    /* 初始化输入输出缓存指针 */
    inputF32 = &testInput_f32_50Hz_200Hz[0];
    outputF32 = &testOutput[0];

    /* 初始化 */
    arm_biquad_cascade_df1_init_f32(&S, numStages, (float32_t *)&IIRCoeffs32BP[0],
                                    (float32_t *)&IIRStateF32[0]);

    /* 实现 IIR 滤波, 这里每次处理 1 个点 */
    for(i=0; i < numBlocks; i++)
    {
        arm_biquad_cascade_df1_f32(&S, inputF32 + (i * blockSize), outputF32 + (i * blockSize),
                                   blockSize);
    }

    /* 放缩系数 */
    ScaleValue = 0.558156585760773649163013487850548699498f * 0.558156585760773649163013487850548699498f;

    /* 打印滤波后结果 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
```

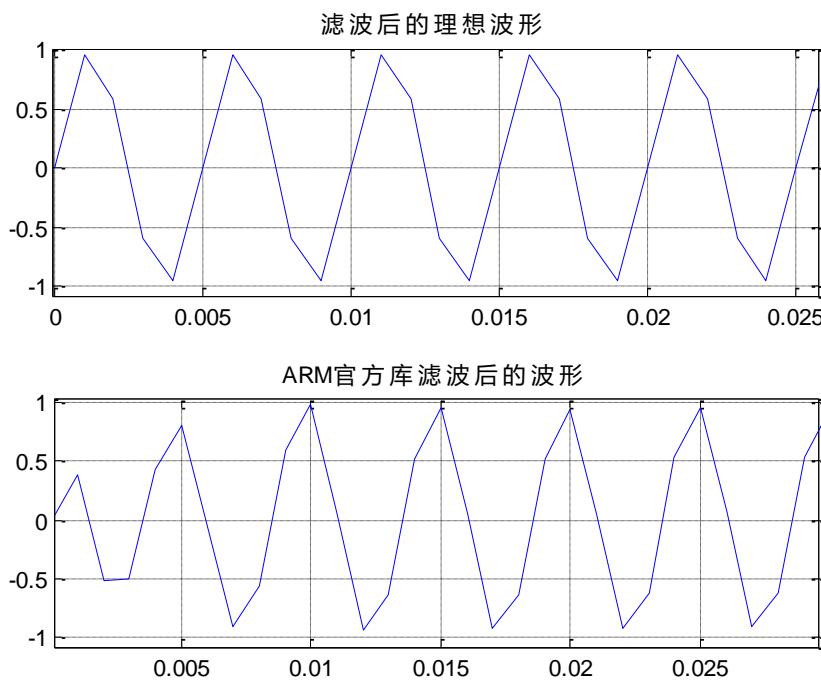
```
{  
    printf("%f, %f\r\n", testInput_f32_50Hz_200Hz[i], testOutput[i]*ScaleValue);  
}
```

运行如上函数可以通过串口打印出函数arm_biquad_cascade_df1_f32滤波后的波形数据，下面通过Matlab绘制波形来对比Matlab计算的结果和ARM官方库计算的结果。

对比前需要先将串口打印出的一组数据加载到 Matlab 中，arm_biquad_cascade_df1_f32 的计算结果起名 sampledata，加载方法在第 13 章 13.6 小结已经讲解，这里不做赘述了。Matlab 中运行的代码如下：

```
fs=1000;          %设置采样频率 1K  
N=400;           %采样点数  
n=0:N-1;  
t=n/fs;          %时间序列  
f=n*fs/N;        %频率序列  
  
x1=sin(2*pi*50*t);  
x2=sin(2*pi*200*t);    %50Hz和200Hz正弦波  
subplot(211);  
plot(t, x1);  
title('滤波后的理想波形');  
grid on;  
  
subplot(212);  
plot(t, sampledata);  
title('ARM官方库滤波后的波形');  
grid on;
```

Matlab 计算结果如下：



从上面的波形对比来看，matlab 和函数 arm_biquad_cascade_df1_f32 计算的结果基本是一致的。为了更好的说明滤波效果，下面从频域的角度来说明这个问题，Matlab 上面运行如下代码：

```
fs=1000;          %设置采样频率 1K
```

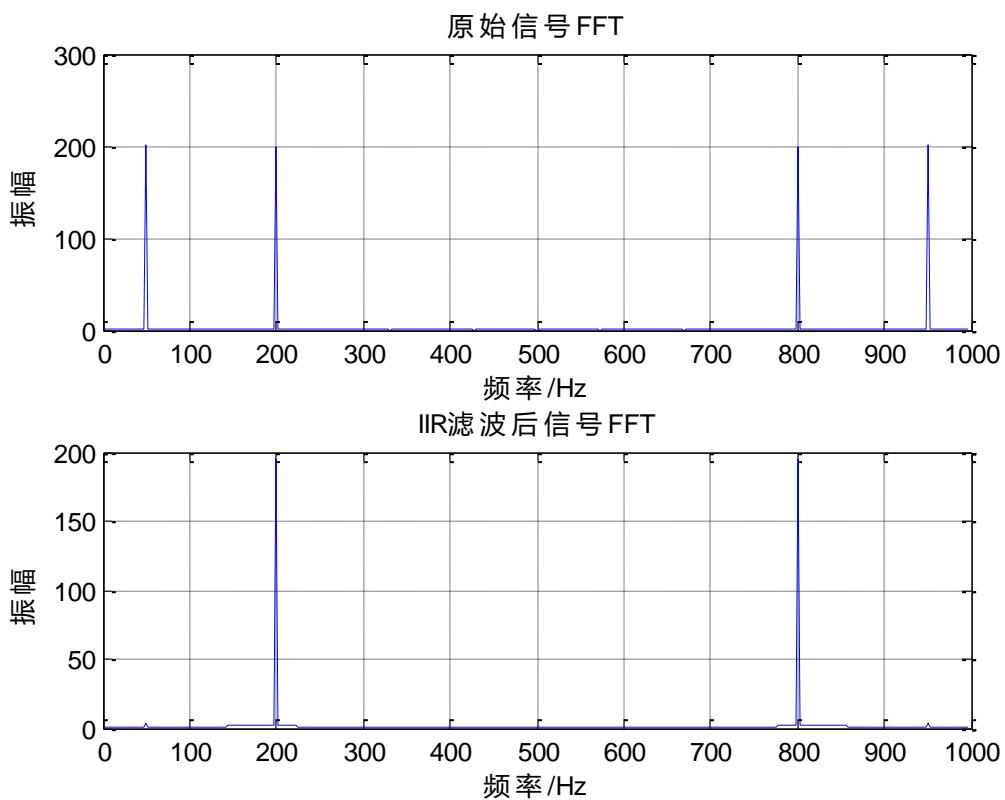
```
N=400; %采样点数
n=0:N-1;
t=n/fs; %时间序列
f=n*fs/N; %频率序列

x = sin(2*pi*50*t) + sin(2*pi*200*t); %50Hz和200Hz正弦波合成

subplot(211);
y=fft(x, N); %对信号x做FFT
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号FFT');
grid on;

y3=fft(sampledata, N); %经过IIR滤波器后得到的信号做FFT
subplot(212);
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('IIR滤波后信号FFT');
grid on;
```

Matlab 计算结果如下：



上面波形变换前的 FFT 和变换后 FFT 可以看出，50Hz 的正弦波基本被滤除。

46.6 实验例程说明 (MDK)

配套例子：



V7-231_IIR 带通滤波器(支持逐点实时滤波)

实验目的:

1. 学习 IIR 带通滤波器的实现，支持实时滤波

实验内容:

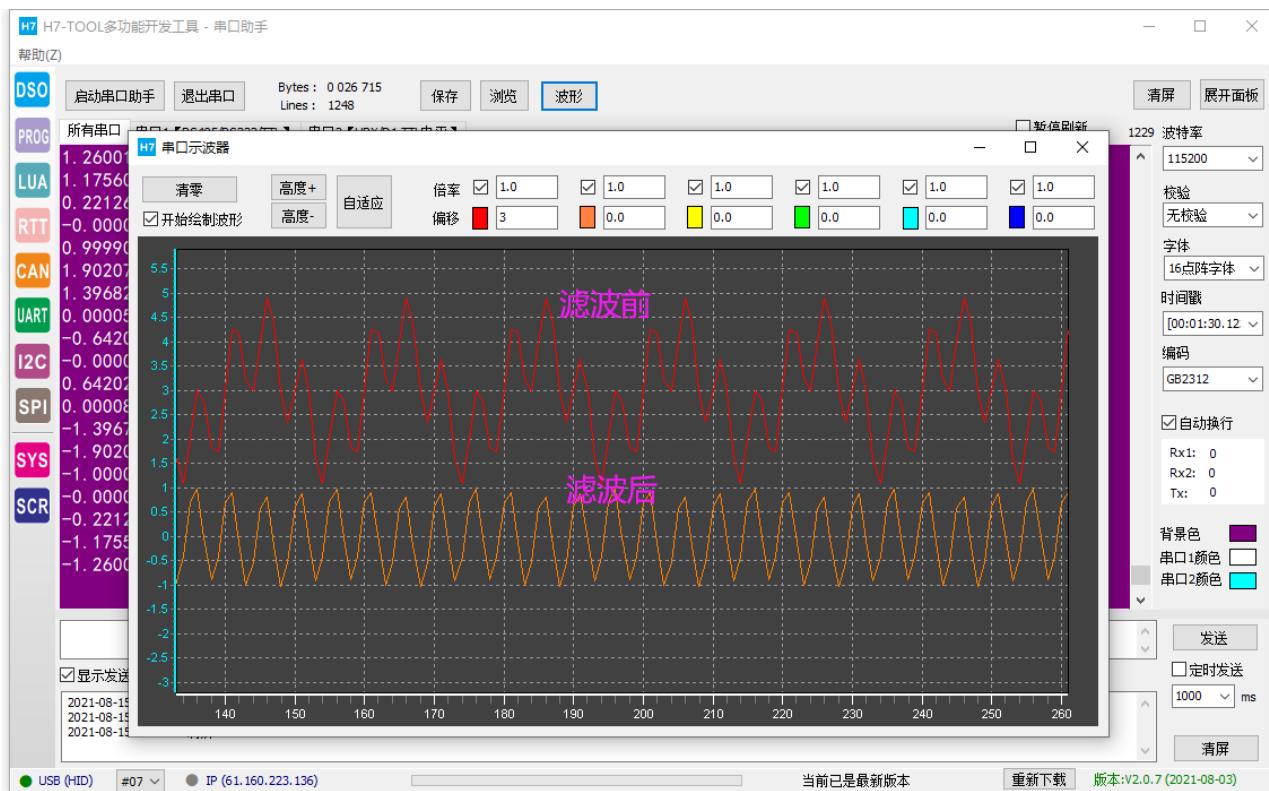
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

使用 AC6 注意事项

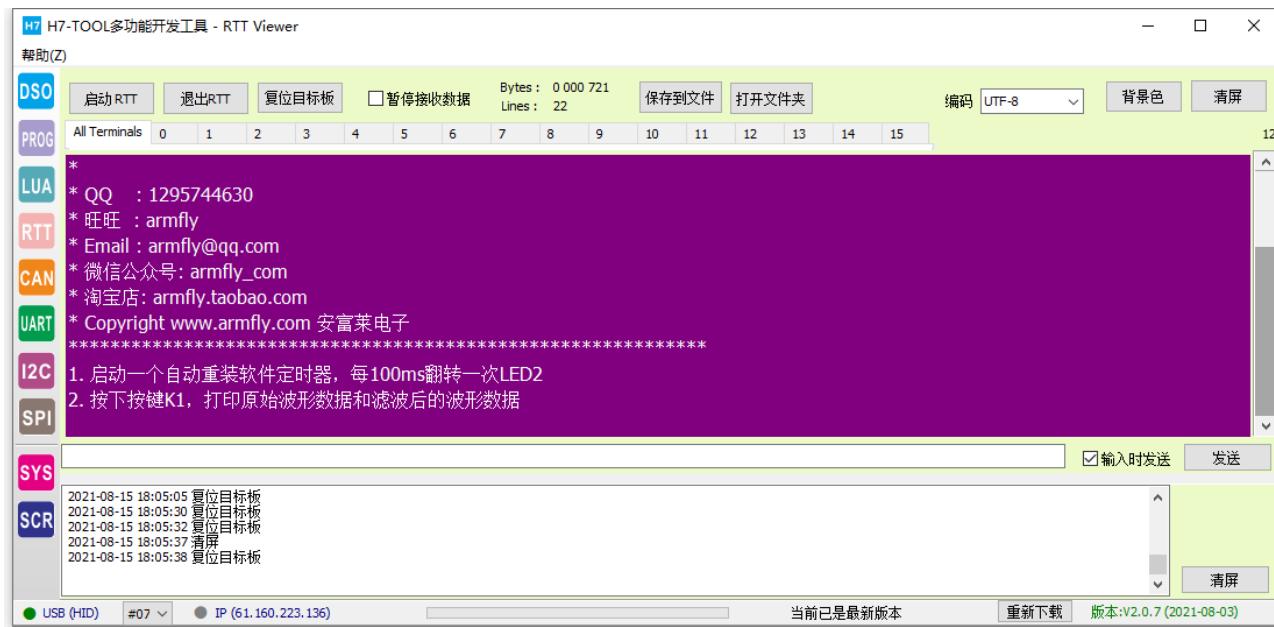
特别注意附件章节 C 的问题

上电后串口打印的信息:

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

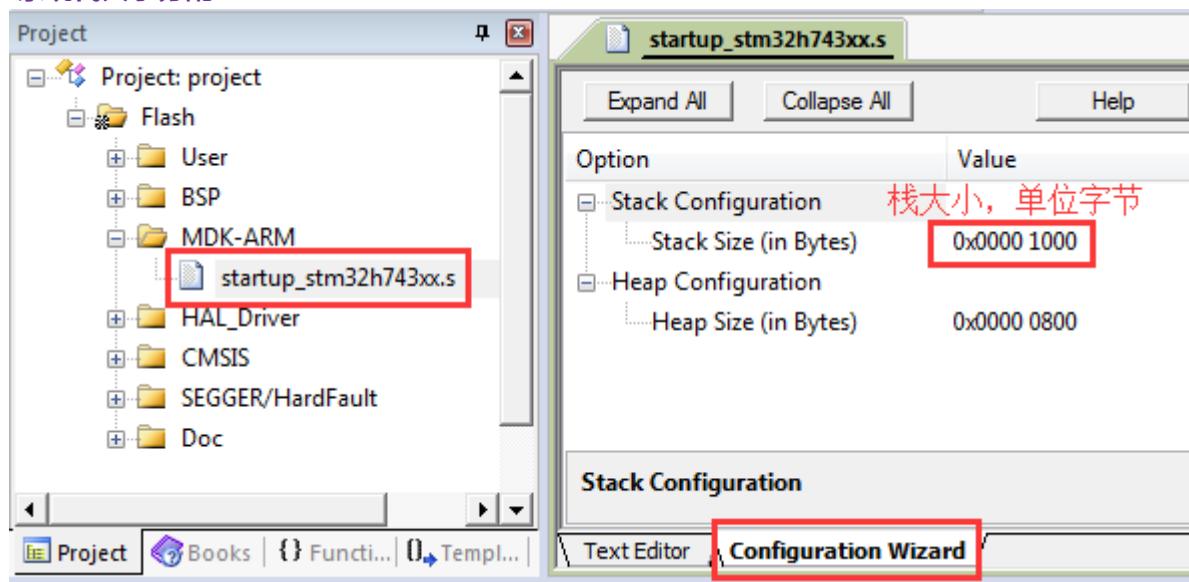


RTT 方式打印信息:

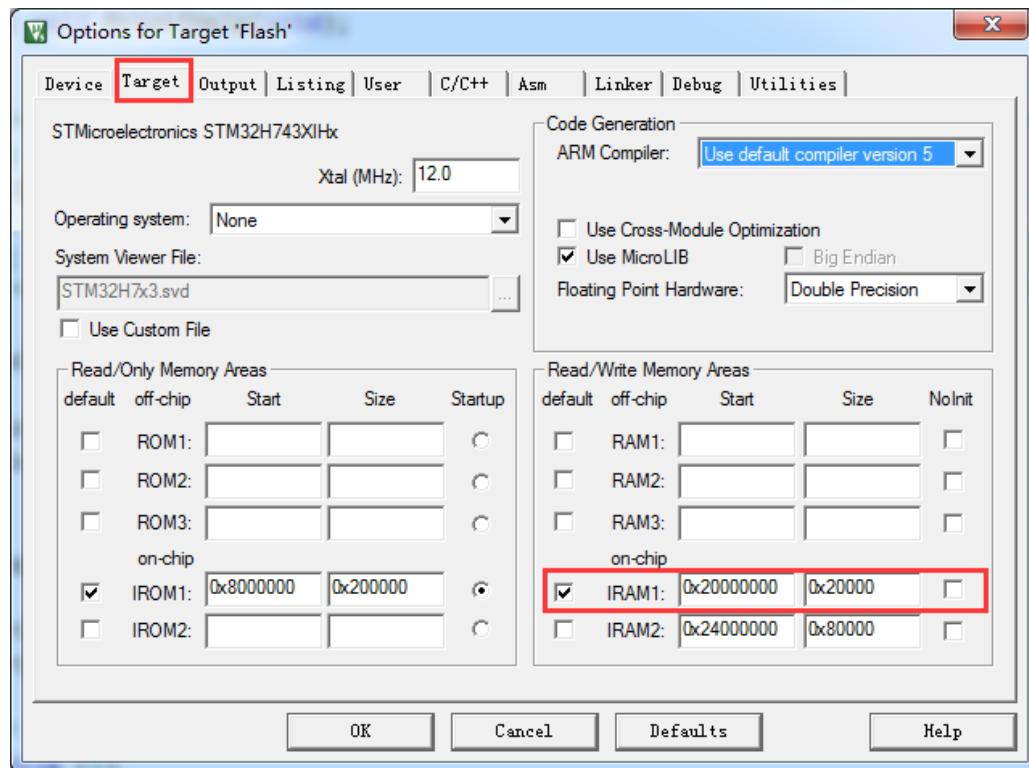


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */
```



```
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
    testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                    arm_sin_f32(2*3.1415926f*200*i/1000);
}

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2); /* 翻转 LED 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_iir_f32_bp();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
}
```

46.7 实验例程说明 (IAR)

配套例子：

V7-231_IIR 带通滤波器(支持逐点实时滤波)

实验目的：

1. 学习 IIR 带通滤波器的实现，支持实时滤波

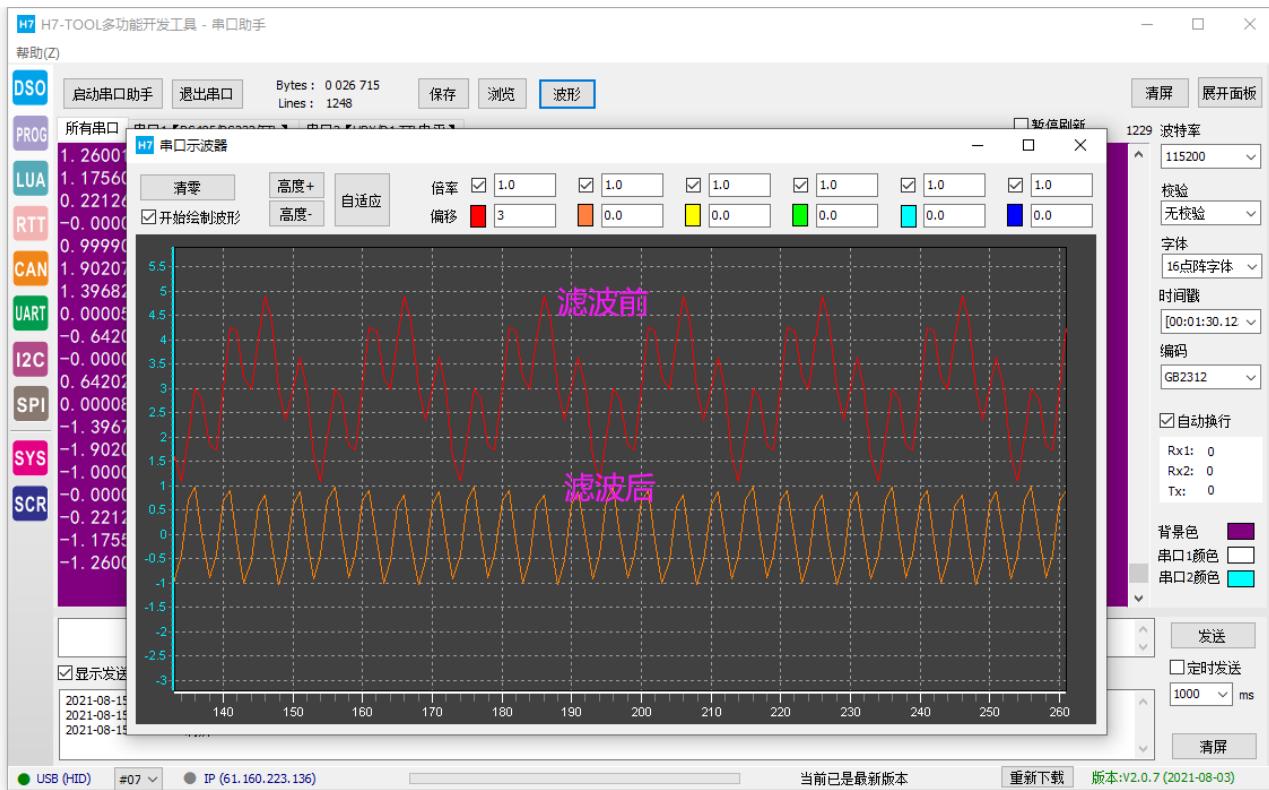
实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

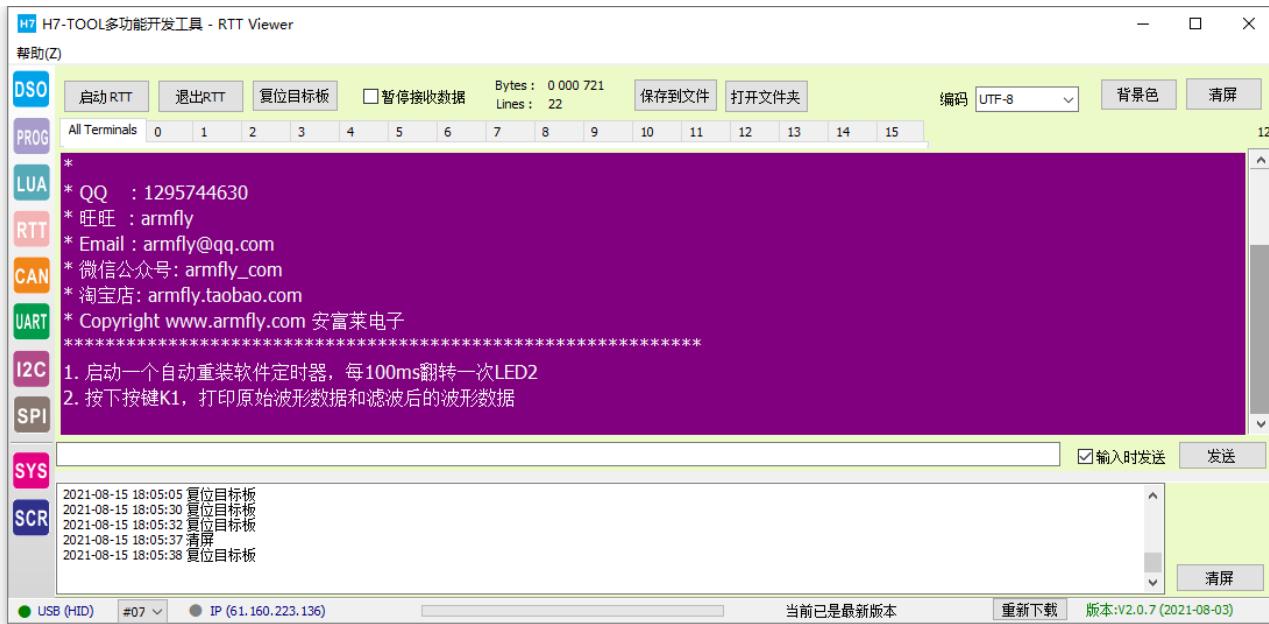
上电后串口打印的信息：



波特率 115200，数据位 8，奇偶校验位无，停止位 1。

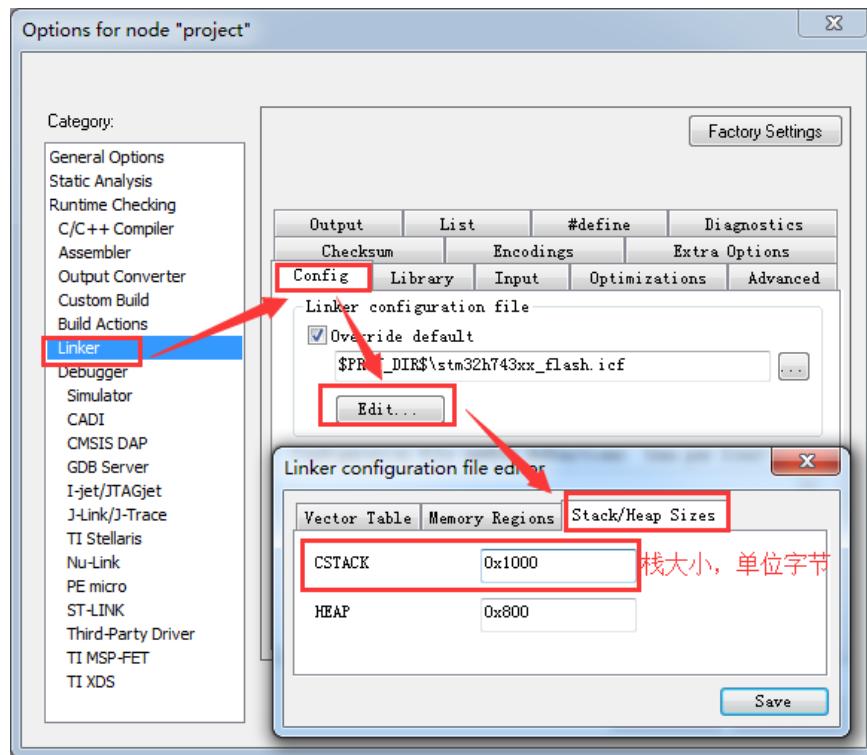


RTT 方式打印信息：

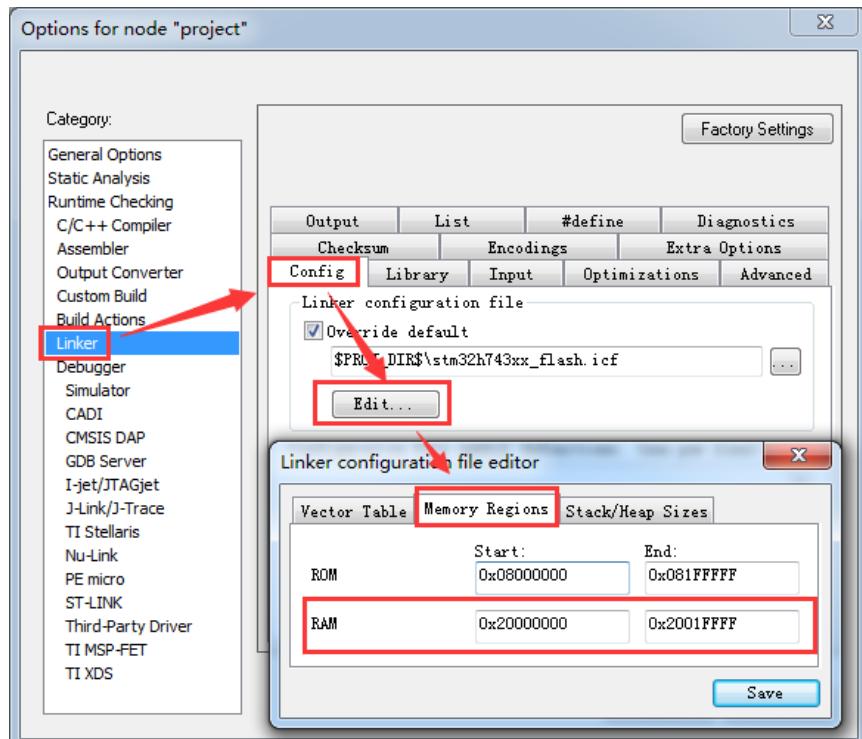


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
*****
```



```
/* 功能说明：初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形    参：无
* 返回值：无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
        - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
        Event Recorder:
        - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
        - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名：MPU_Config
* 功能说明：配置 MPU
* 形    参：无
* 返回值：无
*****
*/
static void MPU_Config( void )
```



```
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /*使能 MPU */  
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);  
}  
  
/*  
*****  
* 函数名: CPU_CACHE_Enable  
* 功能说明: 使能 L1 Cache  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void CPU_CACHE_Enable(void)  
{  
    /* 使能 I-Cache */  
    SCB_EnableICache();  
  
    /* 使能 D-Cache */  
    SCB_EnableDCache();  
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
        testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                      arm_sin_f32(2*3.1415926f*200*i/1000);
    }

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2); /* 翻转 LED 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_iir_f32_bp();
                    break;
            }
        }
    }
}
```



```
default:  
    /* 其它的键值不处理 */  
    break;  
}  
}  
}  
}
```

46.8 总结

本章节主要讲解了 IIR 滤波器的带通实现，同时一定要注意 IIR 滤波器的群延迟问题，详见本教程的第 41 章。



第47章 STM32H7 的 IIR 带阻滤波器实现（支持逐个数据的实时滤波）

本章节讲解 IIR 带阻滤波器实现。

47.1 初学者重要提示

47.2 带通滤波器介绍

47.3 IIR 滤波器介绍

47.4 Matlab 工具箱 filterDesigner 生成带通滤波器 C 头文件

47.5 IIR 带通滤波器设计

47.6 实验例程说明 (MDK)

47.7 实验例程说明 (IAR)

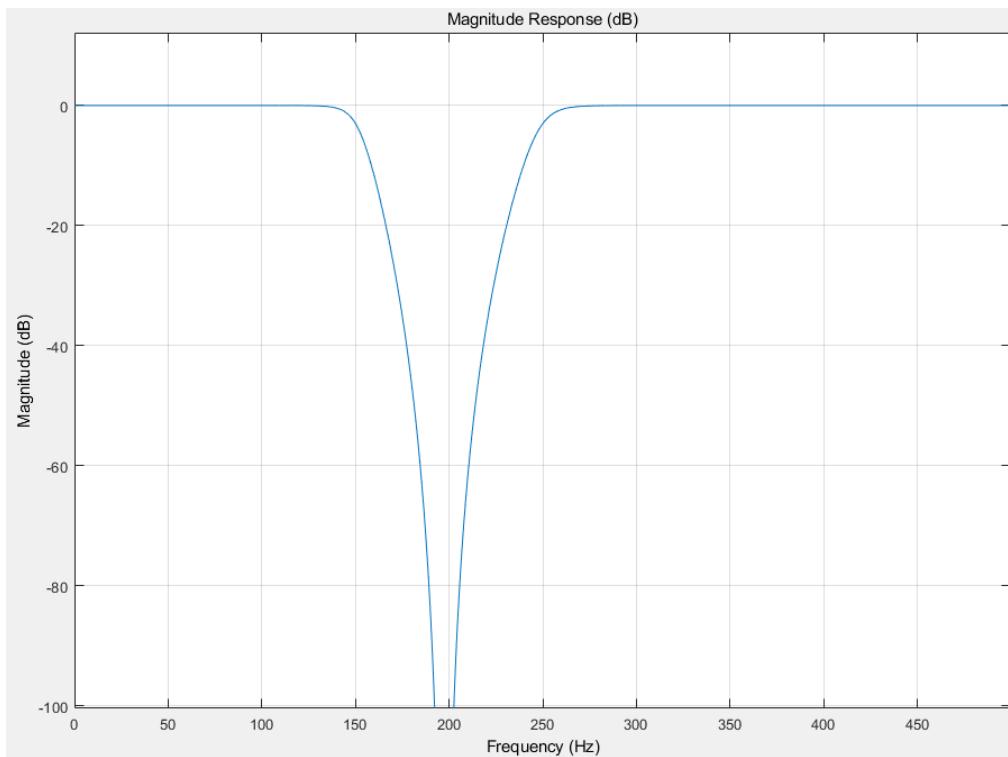
47.8 总结

47.1 初学者重要提示

- ◆ 本章节提供的带阻滤波器支持实时滤波，每次可以滤波一个数据，也可以多个数据，不限制大小。
但要注意以下两点：
 - 所有数据是在同一个采样率下依次采集的数据。
 - 每次过滤数据个数一旦固定下来，运行中不可再修改。
- ◆ FIR 滤波器的群延迟是一个重要的知识点，详情在本教程第 41 章有详细说明。IIR 和 FIR 一样，也有群延迟问题。

47.2 带阻滤波器介绍

减弱一个范围内的频率信号通过，让范围之外的频率信号通过。比如混合信号含有 50Hz + 200Hz + 400Hz 信号，我们可通过带通滤波器，让 50Hz + 400Hz 信号通过，而阻止 200Hz 信号通过。



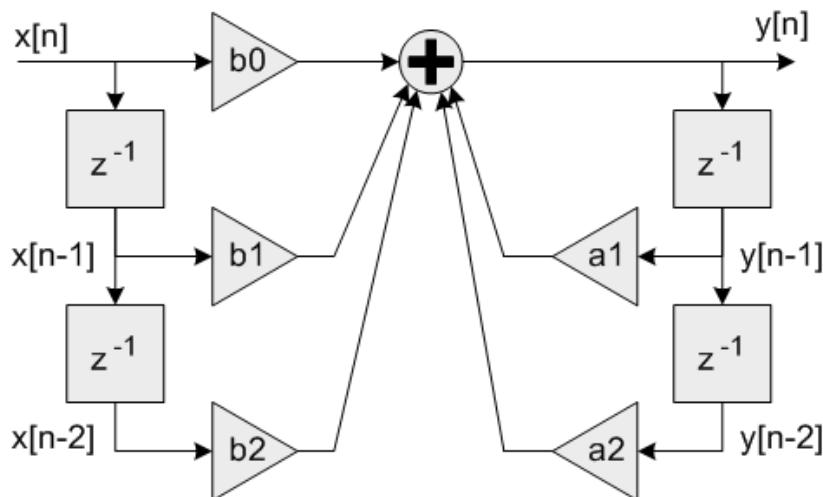
47.3 IIR 滤波器介绍

ARM 官方提供的直接 I 型 IIR 库支持 Q7, Q15, Q31 和浮点四种数据类型。其中 Q15 和 Q31 提供了快速版本。

直接 I 型 IIR 滤波器是基于二阶 Biquad 级联的方式来实现的。每个 Biquad 由一个二阶的滤波器组成：

$$y[n] = b_0 * x[n] + b_1 * x[n-1] + b_2 * x[n-2] + a_1 * y[n-1] + a_2 * y[n-2]$$

直接 I 型算法每个阶段需要 5 个系数和 4 个状态变量。



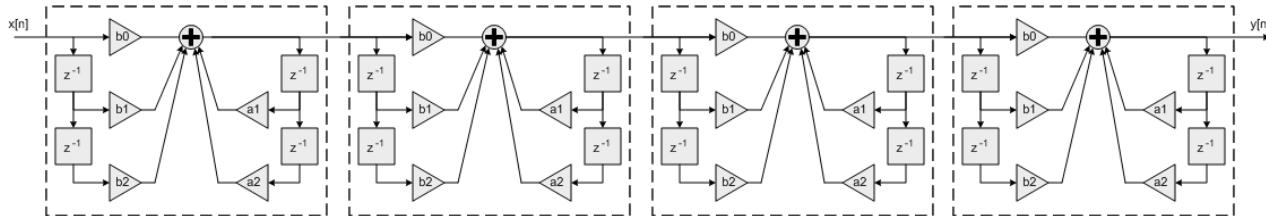
这里有一点要特别的注意，有些滤波器系数生成工具是采用的下面公式实现：



$$y[n] = b0 * x[n] + b1 * x[n-1] + b2 * x[n-2] - a1 * y[n-1] - a2 * y[n-2]$$

比如 matlab 就是使用上面的公式实现的，所以在使用 fdatool 工具箱生成的 a 系数需要取反才能用于直接 I 型 IIR 滤波器的函数中。

高阶 IIR 滤波器的实现是采用二阶 Biquad 级联的方式来实现的。其中参数 numStages 就是用来做指定二阶 Biquad 的个数。比如 8 阶 IIR 滤波器就可以采用 numStages=4 个二阶 Biquad 来实现。



如果要实现 9 阶 IIR 滤波器就需要将 numStages=5，这时就需要其中一个 Biquad 配置成一阶滤波器(也就是 $b2=0, a2=0$)。

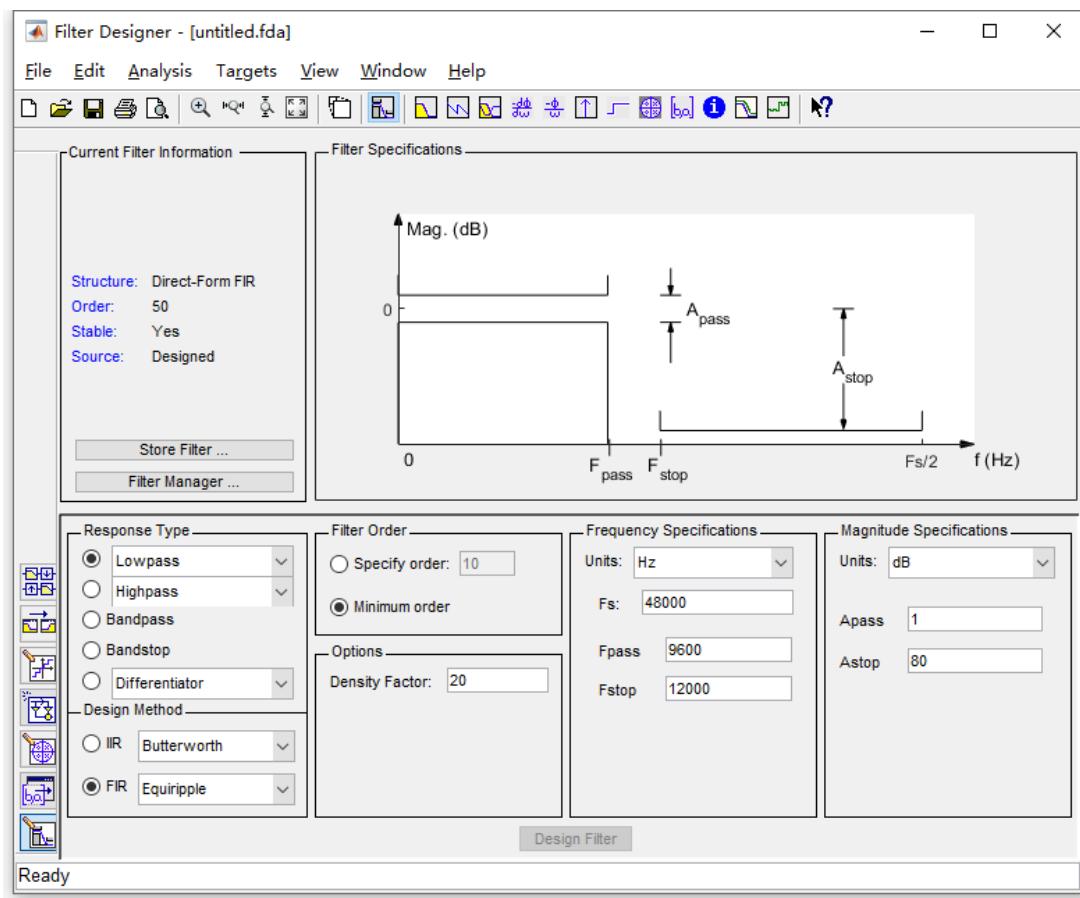
47.4 Matlab 工具箱 filterDesigner 生成 IIR 带阻滤波器系数

前面介绍 FIR 滤波器的时候，我们讲解了如何使用 filterDesigner 生成 C 头文件，从而获得滤波器系数。这里不能再使用这种方法了，主要是因为通过 C 头文件获取的滤波器系数需要通过 ARM 官方的 IIR 函数调用多次才能获得滤波结果，所以我们这里换另外一种方法。

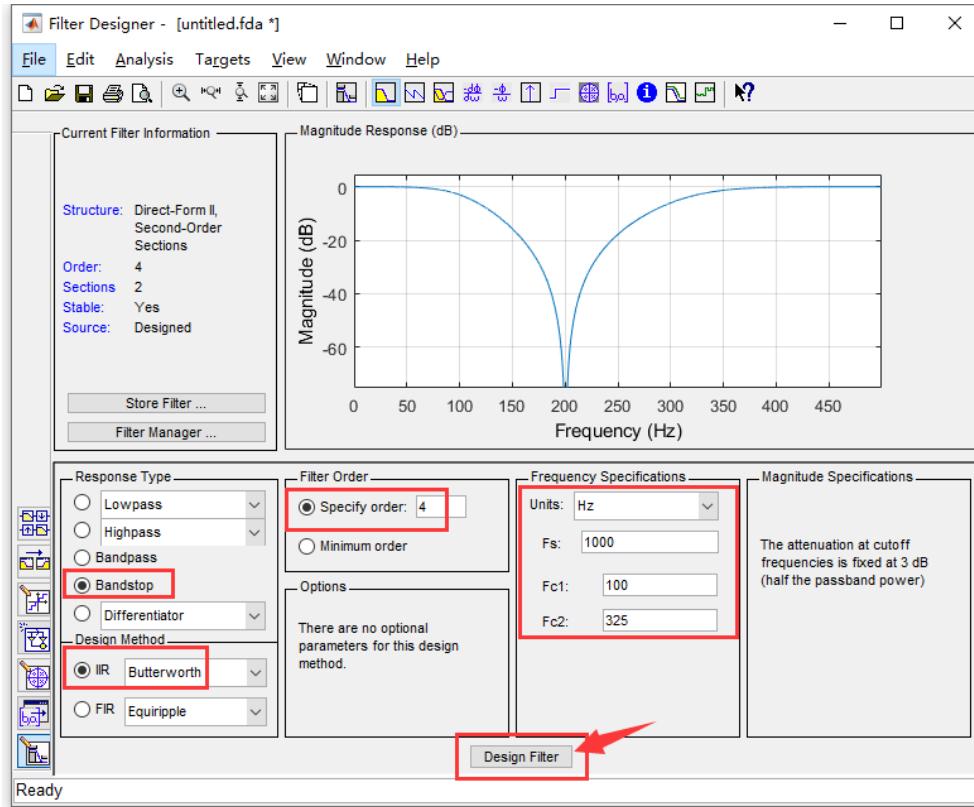
下面我们讲解如何通过 filterDesigner 工具箱生成滤波器系数。首先在 matlab 的命令窗口输入 filterDesigner 就能打开这个工具箱：



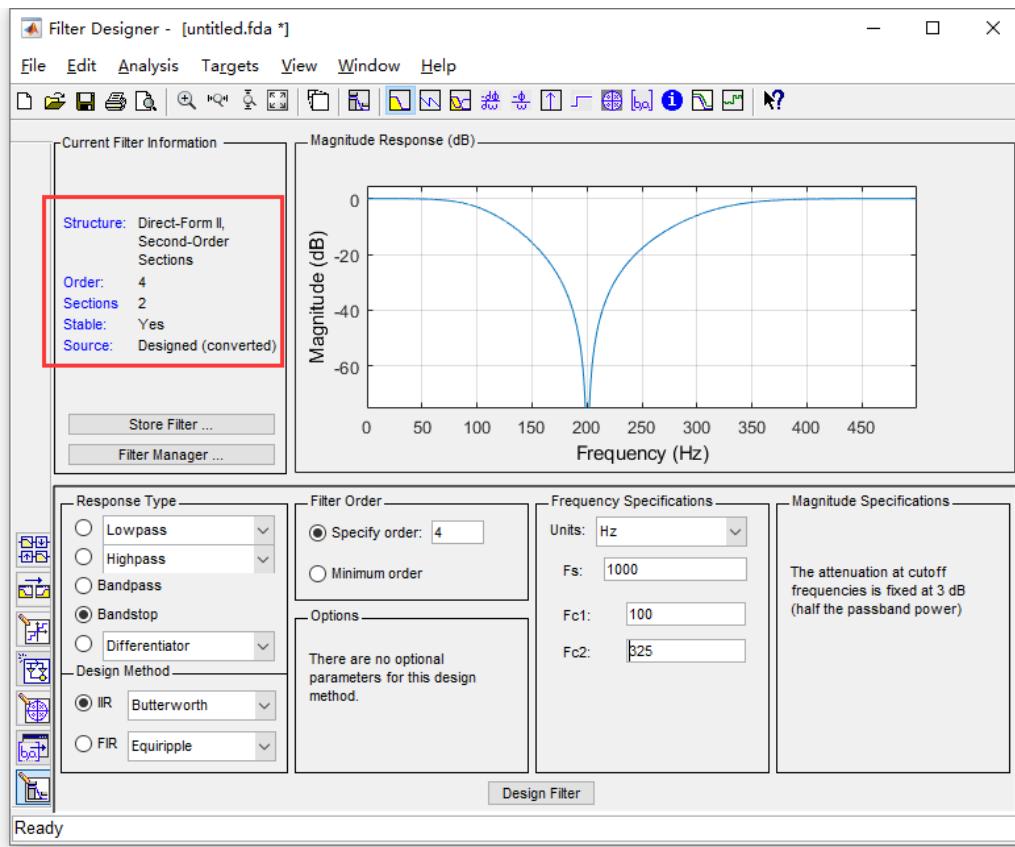
filterDesigner 界面打开效果如下：



IIR 滤波器的低通，高通，带通，带阻滤波的设置会在下面一一讲解，这里说一下设置后相应参数后如何生成滤波器系数。参数设置好以后点击如下按钮：

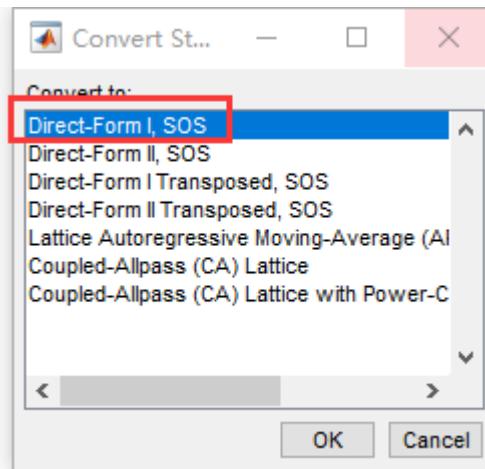


点击 Design Filter 之后，注意左上角生成的滤波器结构：

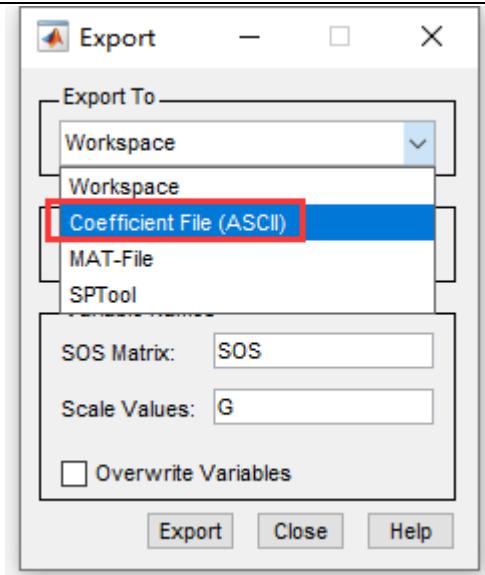


默认生成的 IIR 滤波器类型是 Direct-Form II, Second-Order Sections (直接 II 型，每个 Section 是一个二阶滤波器)。这里我们需要将其转换成 Direct-Form I, Second-Order Sections，因为本章使用的 IIR 滤波器函数是 Direct-Form I 的结构。

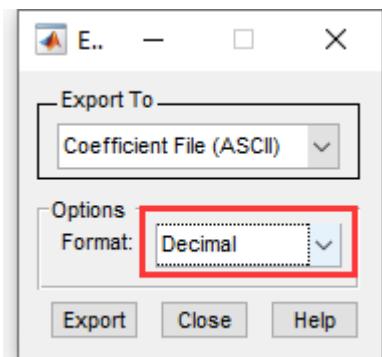
转换方法，点击 Edit->Convert Structure，界面如下，这里我们选择第一项，并点击 OK：



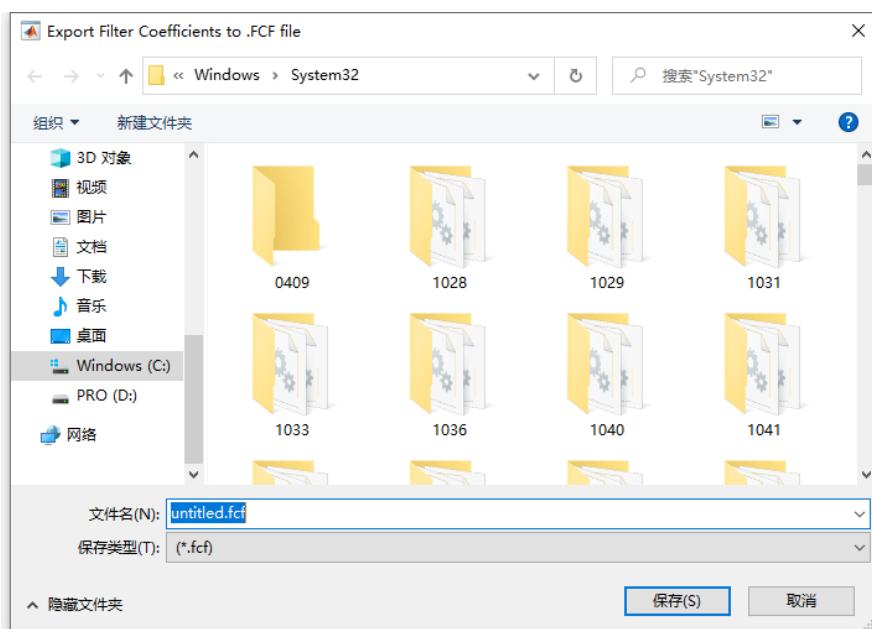
转换好以后再点击 File-Export，第一项选择 Coefficient File (ASCII)：



第一项选择好以后，第二项选择 Decimal：



两个选项都选择好以后，点击 Export 进行导出，导出后保存即可：



保存后 Matlab 会自动打开 untitled.fcf 文件，可以看到生成的系数：

```
% Generated by MATLAB(R) 9.4 and Signal Processing Toolbox 8.0.  
% Generated on: 15-Aug-2021 23:05:39
```



```
% Coefficient Format: Decimal
% Discrete-Time IIR Filter (real)
% -----
% Filter Structure      : Direct-Form I, Second-Order Sections
% Number of Sections   : 2
% Stable                : Yes
% Linear Phase          : No

SOS Matrix:
1 -0.614001926383350049576392848393879830837 1 1 -1.145142787949775309286337687808554619551
0.502980071467214684410862446384271606803
1 -0.614001926383350049576392848393879830837 1 1 0.474587046588418992598690238082781434059
0.35305199748708809837083322236139792949

Scale Values:
0.583479203143786984142593610158655792475
0.583479203143786984142593610158655792475
```

由于前面选择的是 4 阶 IIR 滤波，生成的结果就是由两组二阶 IIR 滤波系数组成，系数的对应顺序如下：

SOS Matrix:		
1 -0.614001926383350049576392848393879830837	1 1 -1.145142787949775309286337687808554619551	0.502980071467214684410862446384271606803
b0 b1	b2 a0 a1	a2
1 -0.614001926383350049576392848393879830837	1 1 0.474587046588418992598690238082781434059	0.35305199748708809837083322236139792949
b0 b1	b2 a0 a1	a2

注意，实际使用 ARM 官方的 IIR 函数调用的时候要将 a1 和 a2 取反。另外下面两组是每个二阶滤波器的增益，滤波后的结果要乘以这两个增益数值才是实际结果：

```
0.583479203143786984142593610158655792475
0.583479203143786984142593610158655792475
```

实际的滤波系数调用方法，看下面的例子即可。

47.5 IIR 带阻滤波器设计

本章使用的 IIR 滤波器函数是 `arm_biquad_cascade_df1_f32`。使用此函数可以设计 IIR 低通，高通，带通和带阻滤波器

47.5.1 函数 `arm_biquad_cascade_df1_init_f32`

函数原型：

```
void arm_biquad_cascade_df1_init_f32(
    arm_biquad_cascade_df1_inst_f32 * S,
    uint8_t numStages,
    const float32_t * pCoeffs,
    float32_t * pState)
```

函数描述：

这个函数用于 IIR 初始化。

函数参数：

- ◆ 第 1 个参数是 `arm_biquad_cascade_df1_inst_f32` 类型结构体变量。
- ◆ 第 2 个参数是 2 阶滤波器的个数。
- ◆ 第 3 个参数是滤波器系数地址。
- ◆ 第 4 个参数是缓冲状态地址。



注意事项：

结构体 arm_biquad_casd_df1_inst_f32 的定义如下 (在文件 filtering_functions.h 文件) :

```
typedef struct
{
    uint32_t numStages;      /*< number of 2nd order stages in the filter. Overall order is 2*numStages. */
    float32_t *pState; /*< Points to the array of state coefficients. The array is of length 4*numStages. */
    const float32_t *pCoeffs; /*< Points to the array of coefficients. The array is of length 5*numStages */
} arm_biquad_casd_df1_inst_f32;
```

10. numStages 表示二阶滤波器的个数，总阶数是 2*numStages。

11. pState 指向状态变量数组，这个数组用于函数内部计算数据的缓存，总大小 4*numStages。

12. 参数 pCoeffs 指向滤波因数，滤波因数数组长度为 5*numStages。但要注意 pCoeffs 指向的滤波因数应该按照如下的逆序进行排列：

{b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ...}

先放第一个二阶 Biquad 系数，然后放第二个，以此类推。

47.5.2 函数 arm_biquad_cascade_df1_f32

函数定义如下：

```
void arm_biquad_cascade_df1_f32(
    const arm_biquad_casd_df1_inst_f32 * S,
    float32_t * pSrc,
    float32_t * pDst,
    uint32_t blockSize)
```

函数描述：

这个函数用于 IIR 滤波。

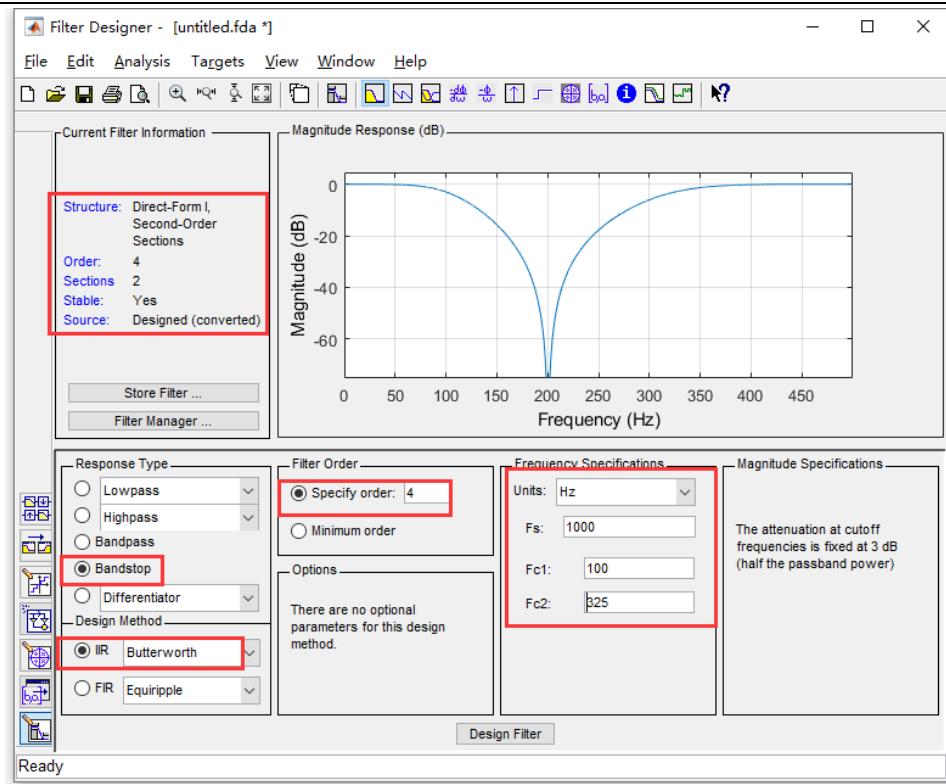
函数参数：

- ◆ 第 1 个参数是 arm_biquad_casd_df1_inst_f32 类型结构体变量。
- ◆ 第 2 个参数是源数据地址。
- ◆ 第 3 个参数是滤波后的数据地址。
- ◆ 第 4 个参数是每次调用处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

47.5.3 filterDesigner 获取带阻滤波器系数

设计一个如下的例子：

信号由 50Hz 正弦波和 200Hz 正弦波组成，采样率 1Kbps，现设计一个巴特沃斯滤波器带阻滤波器，采用直接 I 型，截止频率 100Hz 和 325Hz，采样 400 个数据，滤波器阶数设置为 4。filterDesigner 的配置如下：



配置好带通滤波器后，具体滤波器系数的生成大家参考本章第4小节的方法即可。

47.5.4 带阻滤波器实现

通过工具箱 filterDesigner 获得带阻滤波器系数后在开发板上运行函数 arm_biquad_cascade_df1_f32 来测试低通滤波器的效果。

```
#define numStages 2           /* 2阶IIR滤波的个数 */
#define TEST_LENGTH_SAMPLES 400 /* 采样点数 */
#define BLOCK_SIZE 1           /* 调用一次arm_biquad_cascade_df1_f32处理的采样点个数 */

uint32_t blockSize = BLOCK_SIZE;
uint32_t numBlocks = TEST_LENGTH_SAMPLES/BLOCK_SIZE;           /* 需要调用arm_biquad_cascade_df1_f32的次数 */
*/



static float32_t testInput_f32_50Hz_200Hz[TEST_LENGTH_SAMPLES]; /* 采样点 */
static float32_t testOutput[TEST_LENGTH_SAMPLES];                /* 滤波后的输出 */
static float32_t IIRStateF32[4*numStages];                     /* 状态缓存 */



/* 巴特沃斯带阻滤波器系数 100Hz 325Hz*/
const float32_t IIRCoeffs32BS[5*numStages] = {
    1.0f, -0.614001926383350049576392848393879830837f, 1.0f,
    1.45142787949775309286337687808554619551f, -0.502980071467214684410862446384271606803f,
    1.0f, -0.614001926383350049576392848393879830837f, 1.0f,
    -0.474587046588418992598690238082781434059f, -0.35305199748708809837083322236139792949f
};

/*
***** 函数名: arm_iir_f32_bs
```



```
* 功能说明：调用函数 arm_iir_f32_bs 实现带阻滤波器
* 形参：无
* 返回值：无
*****
*/
static void arm_iir_f32_bs(void)
{
    uint32_t i;
    arm_biquad_cascade_df1_inst_f32 S;
    float32_t ScaleValue;
    float32_t *inputF32, *outputF32;

    /* 初始化输入输出缓存指针 */
    inputF32 = &testInput_f32_50Hz_200Hz[0];
    outputF32 = &testOutput[0];

    /* 初始化 */
    arm_biquad_cascade_df1_init_f32(&S, numStages, (float32_t *)&IIRCoeffs32BS[0],
                                    (float32_t *)&IIRStateF32[0]);

    /* 实现 IIR 滤波，这里每次处理 1 个点 */
    for(i=0; i < numBlocks; i++)
    {
        arm_biquad_cascade_df1_f32(&S, inputF32 + (i * blockSize), outputF32 + (i * blockSize),
                                blockSize);
    }

    /* 放缩系数 */
    ScaleValue = 0.558156585760773649163013487850548699498f * 0.558156585760773649163013487850548699498f;

    /* 打印滤波后结果 */
    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        printf("%f, %f\r\n", testInput_f32_50Hz_200Hz[i], testOutput[i]*ScaleValue);
    }
}
```

运行如上函数可以通过串口打印出函数arm_biquad_cascade_df1_f32滤波后的波形数据，下面通过Matlab绘制波形来对比Matlab计算的结果和ARM官方库计算的结果。

对比前需要先将串口打印出的一组数据加载到 Matlab 中，arm_biquad_cascade_df1_f32 的计算结果起名 sampledata，加载方法在[第 13 章 13.6 小结已经讲解](#)，这里不做赘述了。Matlab 中运行的代码如下：

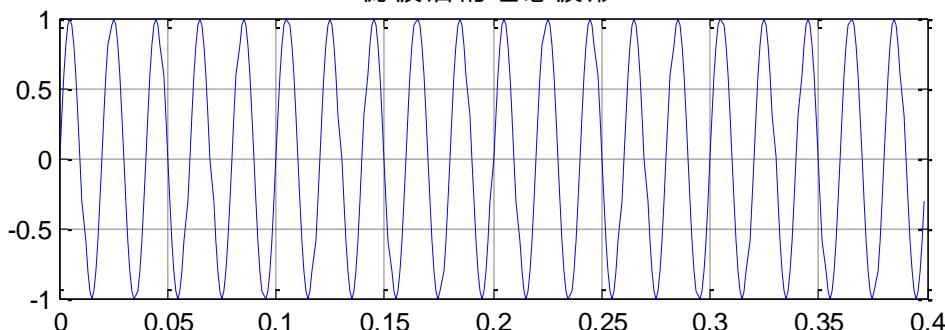
```
fs=1000;          %设置采样频率 1K
N=400;           %采样点数
n=0:N-1;
t=n/fs;          %时间序列
f=n*fs/N;         %频率序列

x1=sin(2*pi*50*t);
x2=sin(2*pi*200*t);      %50Hz和200Hz正弦波
subplot(211);
plot(t, x1);
title('滤波后的理想波形');
grid on;

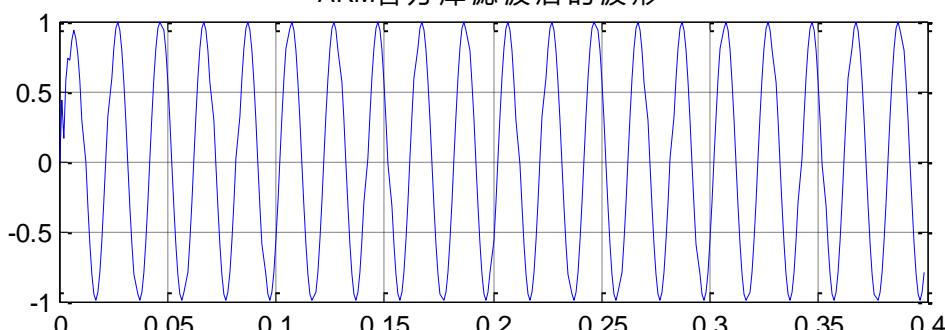
subplot(212);
plot(t, sampledata);
title('ARM官方库滤波后的波形');
grid on;
```

Matlab 计算结果如下：

滤波后的理想波形



ARM官方库滤波后的波形



从上面的波形对比来看，matlab 和函数 arm_biquad_cascade_df1_f32 计算的结果基本是一致的。为了更好的说明滤波效果，下面从频域的角度来说明这个问题，Matlab 上面运行如下代码：

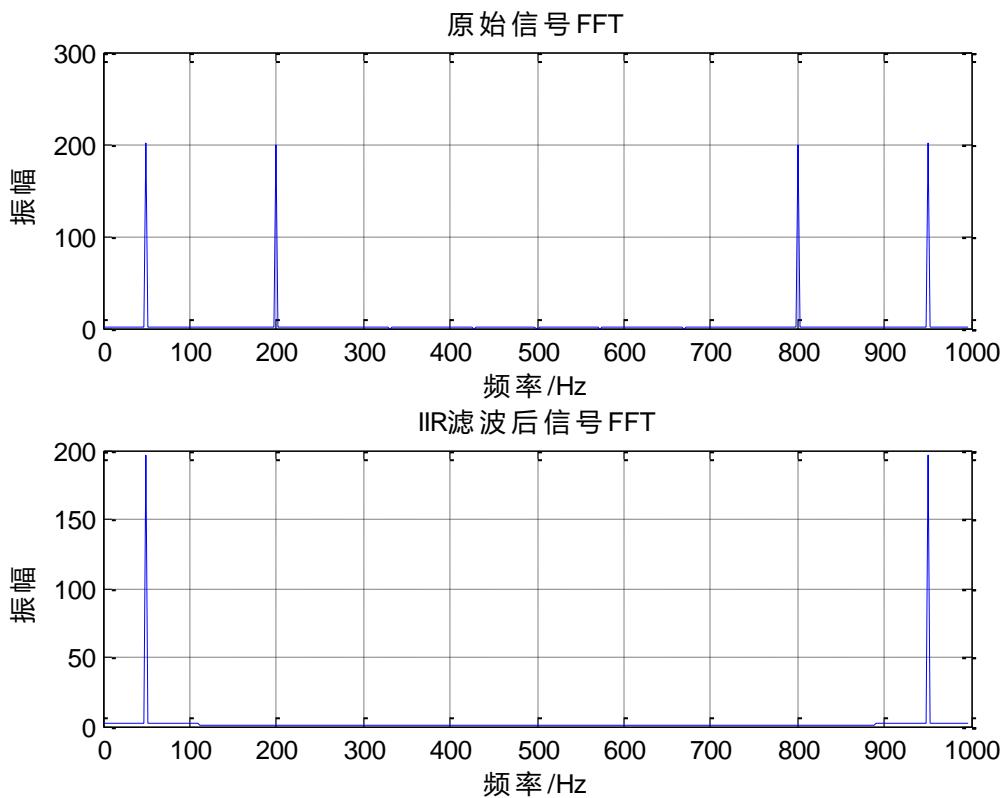
```
fs=1000; %设置采样频率 1K
N=400;
n=0:N-1;
t=n/fs; %时间序列
f=n*fs/N; %频率序列

x = sin(2*pi*50*t) + sin(2*pi*200*t); %50Hz和200Hz正弦波合成

subplot(211);
y=fft(x, N); %对信号x做FFT
plot(f, abs(y));
xlabel('频率/Hz');
ylabel('振幅');
title('原始信号FFT');
grid on;

y3=fft(sampledata, N); %经过IIR滤波器后得到的信号做FFT
subplot(212);
plot(f, abs(y3));
xlabel('频率/Hz');
ylabel('振幅');
title('IIR滤波后信号FFT');
grid on;
```

Matlab 计算结果如下：



上面波形变换前的 FFT 和变换后 FFT 可以看出，200Hz 的正弦波基本被滤除。

47.6 实验例程说明 (MDK)

配套例子：

V7-232_IIR 带阻滤波器(支持逐点实时滤波)

实验目的：

1. 学习 IIR 带阻滤波器的实现，支持实时滤波

实验内容：

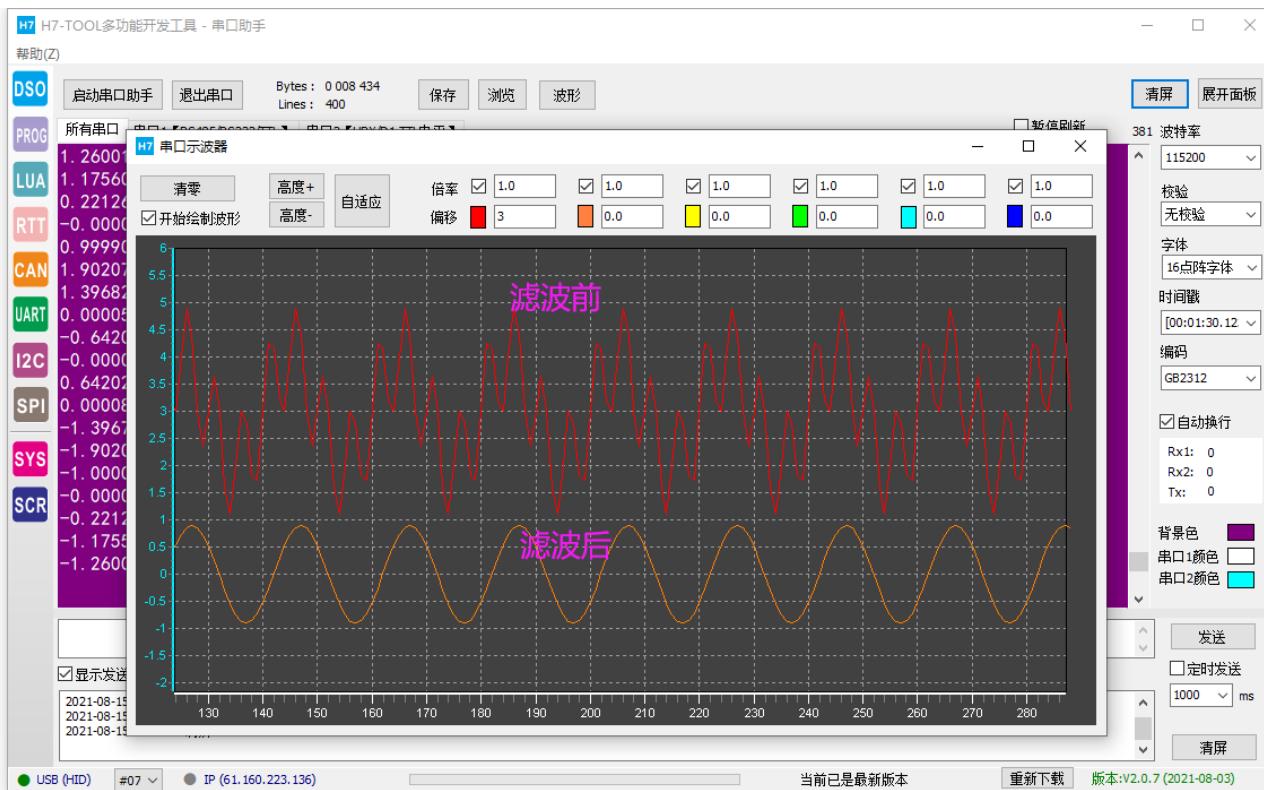
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

使用 AC6 注意事项

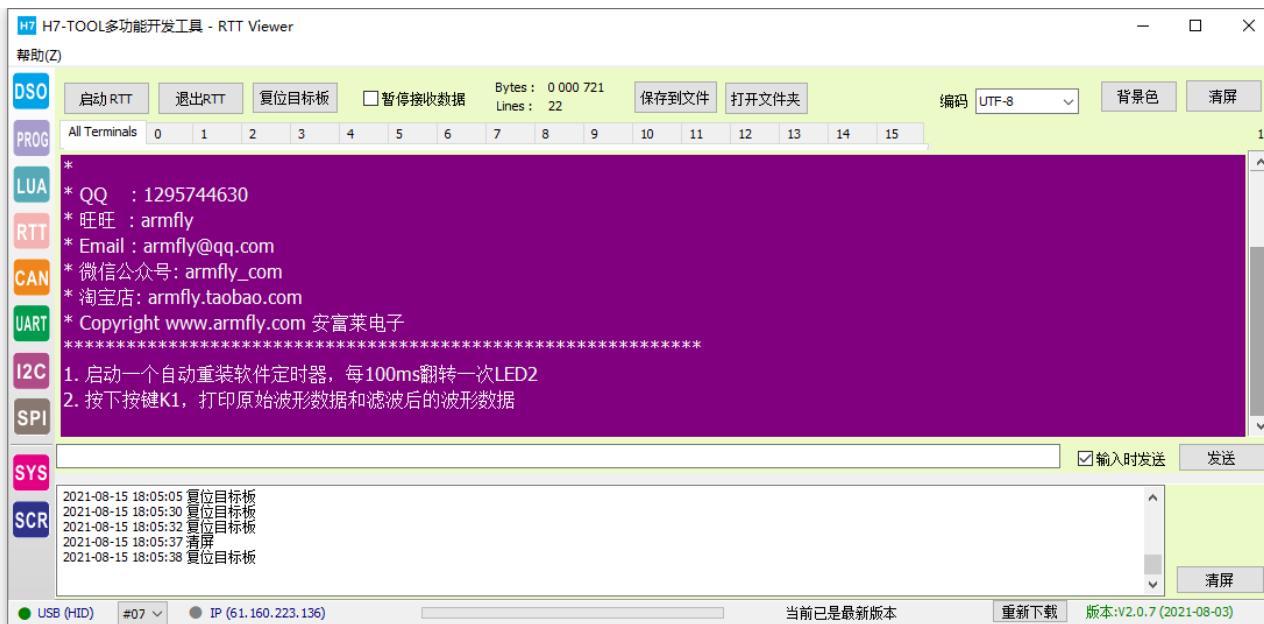
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

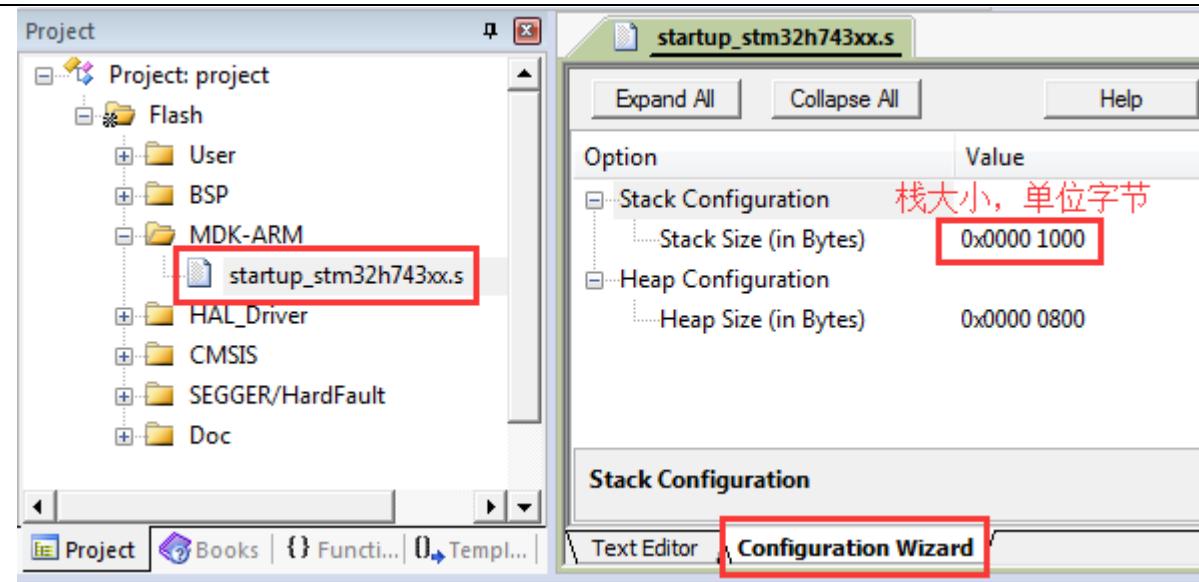


RTT 方式打印信息：

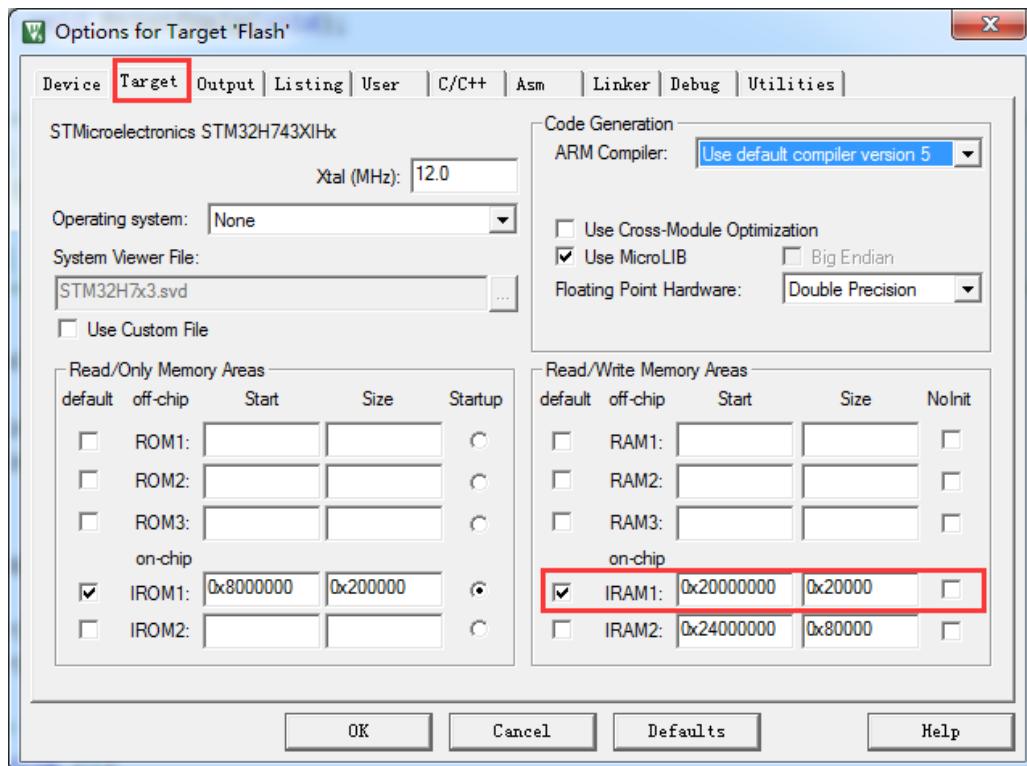


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现:

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
```



```
{  
    /* 配置 MPU */  
    MPU_Config();  
  
    /* 使能 L1 Cache */  
    CPU_CACHE_Enable();  
  
    /*  
     * STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：  
     * - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。  
     * - 设置 NVIC 优先级分组为 4。  
     */  
    HAL_Init();  
  
    /*  
     * 配置系统时钟到 400MHz  
     * - 切换使用 HSE。  
     * - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。  
     */  
    SystemClock_Config();  
  
    /*  
     * Event Recorder:  
     * - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。  
     * - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章  
     */  
#if Enable_EventRecorder == 1  
    /* 初始化 EventRecorder 并开启 */  
    EventRecorderInitialize(EventRecordAll, 1U);  
    EventRecorderStart();  
#endif  
  
    bsp_InitKey(); /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */  
    bsp_InitTimer(); /* 初始化滴答定时器 */  
    bsp_InitUart(); /* 初始化串口 */  
    bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */  
    bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();
```



```
/* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。



- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
***** */

int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
        testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                       arm_sin_f32(2*3.1415926f*200*i/1000);
    }

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2); /* 翻转 LED 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_iir_f32_bs();
                    break;

                default:
                    /* 其它的键值不处理 */
                    break;
            }
        }
    }
}
```

{
}

47.7 实验例程说明 (IAR)

配套例子：

V7-232_IIR 带阻滤波器(支持逐点实时滤波)

实验目的：

1. 学习 IIR 带阻滤波器的实现，支持实时滤波

实验内容：

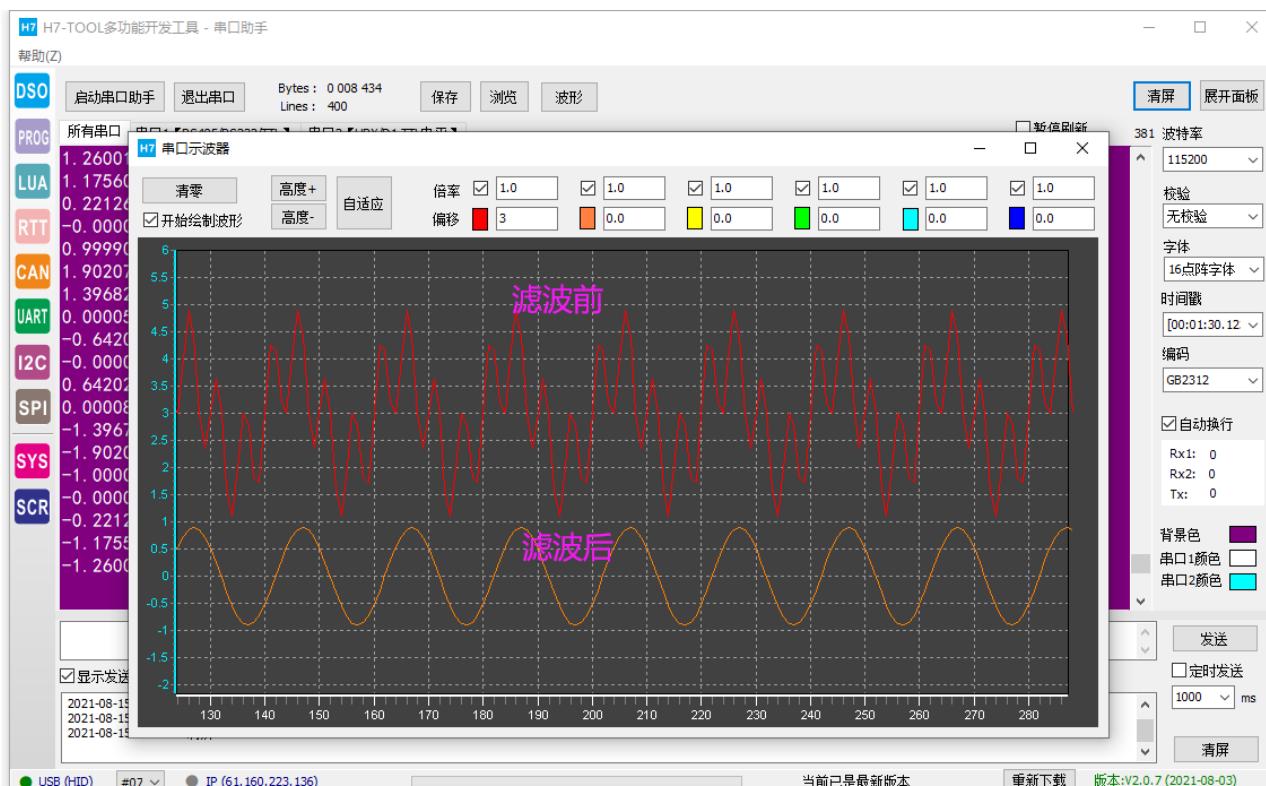
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印原始波形数据和滤波后的波形数据。

使用 AC6 注意事项

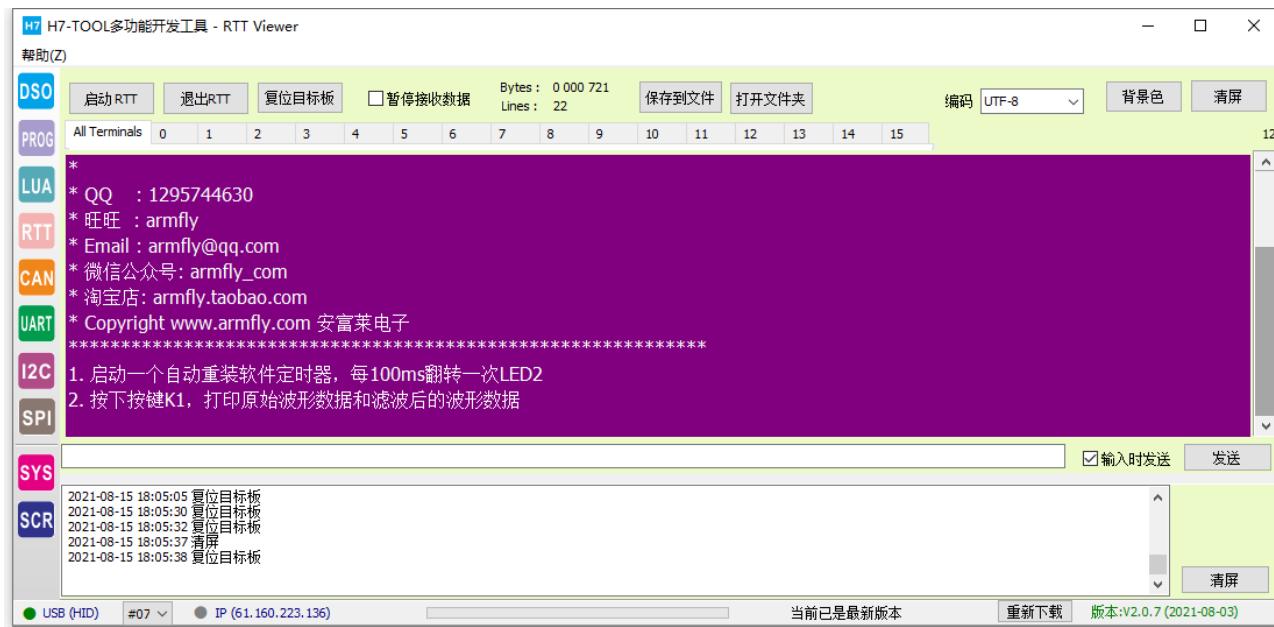
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

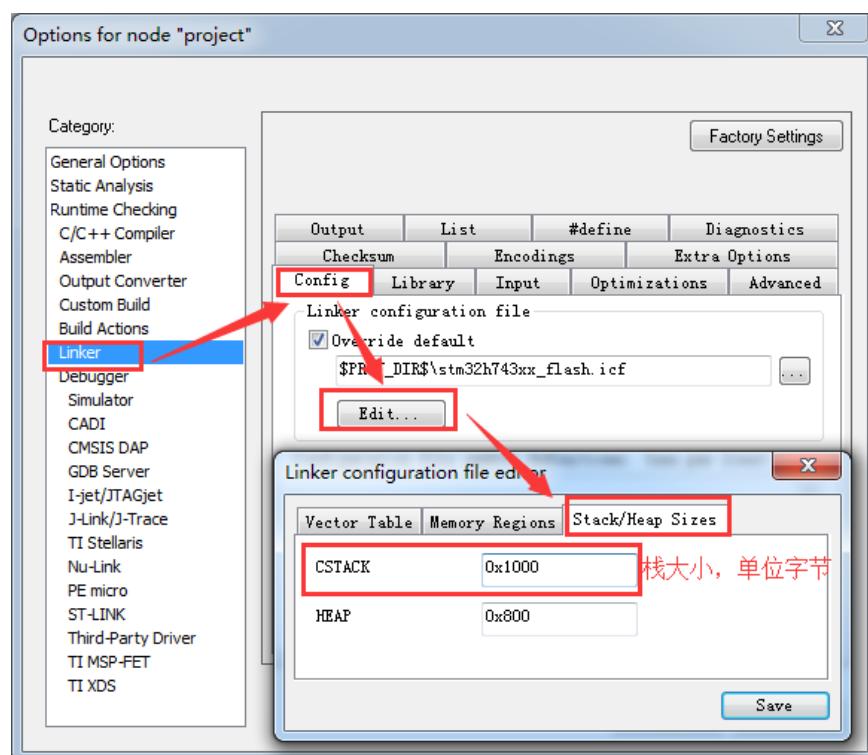


RTT 方式打印信息：

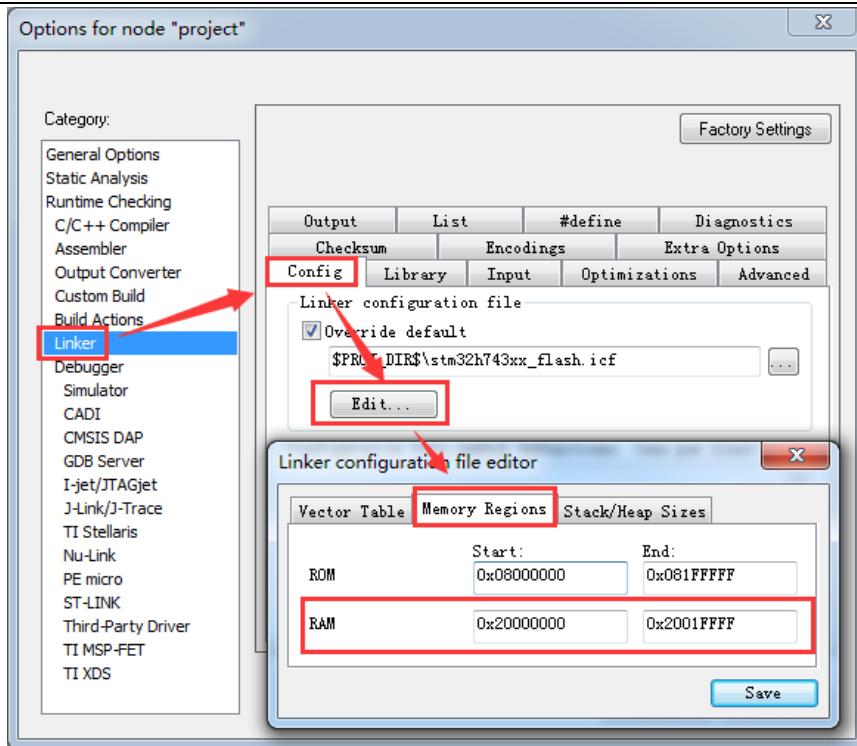


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec   = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印原始波形数据和滤波后的波形数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t uckKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();             /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波，采样率 1KHz */
    }
}
```



```
testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                arm_sin_f32(2*3.1415926f*200*i/1000);
}

bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2); /* 翻转 LED 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_iir_f32_bs();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
}
```

47.8 总结

本章节主要讲解了 IIR 滤波器的带阻实现，同时一定要注意 IIR 滤波器的群延迟问题，详见本教程的第 41 章。



第48章 STM32H7 的中值滤波器实现，适合噪

声和脉冲过滤（支持逐个数据的实时滤波）

本章节讲解中值滤波器实现，适用于噪声和脉冲的过滤。

48.1 初学者重要提示

48.2 中值滤波器介绍

48.3 中值滤波器原理

48.4 Matlab 中值滤波器实现

48.5 中值滤波器设计

48.6 实验例程说明 (MDK)

48.7 实验例程说明 (IAR)

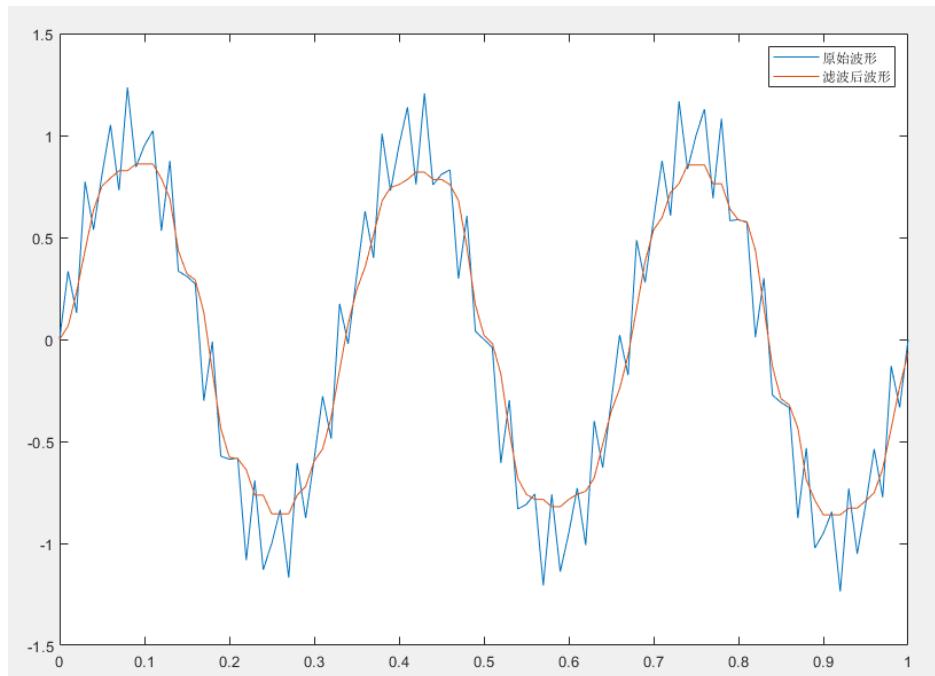
48.8 总结

48.1 初学者重要提示

- ◆ ARM DSP 库没有提供中值滤波器，所以本章的实现是根据中值滤波器原理做了两个函数，一个函数是一块数据的滤波器实现，另一个函数是实时的逐点滤波实现。

48.2 中值滤波器介绍

中值滤波器是一种非线性数字过滤技术，通常用于消除图像或信号中的噪声。中值滤波器在数字图像处理中被广泛使用。在信号处理中也有应用，通过丢弃所有可疑测量结果来抑制脉冲干扰。有几个输入数据，筛选器计算中值值。



48.3 中值滤波器原理

这里我们通过一个实例来理解中值滤波器。比如我们要对如下五个数据求中值：

$x = [14 \ 18 \ 16 \ 21 \ 11]$

我们将滤波阶数设置为 5，即 $y = \text{medfilt1}(x, 5)$ ，表示每 5 个采样值求一次中值。原理和实现如下：

函数是取 $x(k-2), x(k-1), x(k), x(k+1), x(k+2)$ 的中值作为输出 $y(k)$ 。对于 $y(1)$ ，只有 $x(1), x(2), x(3)$ 存在数值，之前的不存在，对于不存在的补 0。每 5 个数按从小到大排列后取中值有：

$y(1)$ 的计算是从 $[0 \ 0 \ 14 \ 16 \ 18]$ 中取中值是 14。

$y(2)$ 的计算是从 $[0 \ 14 \ 16 \ 18 \ 21]$ 中取中值是 16。

$y(3)$ 的计算是从 $[11 \ 14 \ 16 \ 18 \ 21]$ 中取中值是 16。

$y(4)$ 的计算是从 $[0 \ 11 \ 16 \ 18 \ 21]$ 中取中值是 16。

$y(5)$ 的计算是从 $[0 \ 0 \ 11 \ 16 \ 21]$ 中取中值是 11。

48.4 Matlab 中值滤波器实现

首先创建两个混合信号，便于更好测试滤波器效果。

混合信号 $\text{Mix_Signal_1} = \text{信号 Signal_Original_1} + \text{白噪声}$ 。

混合信号 $\text{Mix_Signal_2} = \text{信号 Signal_Original_2} + \text{白噪声}$ 。

```
Fs = 1000; %采样率
N = 1000; %采样点数
n = 0:N-1;
t = 0:1/Fs:1-1/Fs; %时间序列
Signal_Original_1 = sin(2*pi*10*t)+sin(2*pi*20*t)+sin(2*pi*30*t);
```



```
Noise_White_1 = [0.3*randn(1,500), rand(1,500)]; %前 500 点高斯分部白噪声, 后 500 点均匀分布白噪声
Mix_Signal_1 = Signal_Original_1 + Noise_White_1; %构造的混合信号

Signal_Original_2 = [zeros(1,100), 20*ones(1,20), -2*ones(1,30), 5*ones(1,80), -5*ones(1,30),
9*ones(1,140), -4*ones(1,40), 3*ones(1,220),
12*ones(1,100), 5*ones(1,20), 25*ones(1,30), 7 *ones(1,190)];

Noise_White_2 = 0.5*randn(1,1000); %高斯白噪声
Mix_Signal_2 = Signal_Original_2 + Noise_White_2; %构造的混合信号
```

滤波代码实现如下：

```
%*****
%
%          信号 Mix_Signal_1 和 Mix_Signal_2 分别作中值滤波
%
%*****
```

%混合信号 Mix_Signal_1 中值滤波

```
Signal_Filter=medfilt1(Mix_Signal_1,10);

subplot(4,1,1); %Mix_Signal_1 原始信号
plot(Mix_Signal_1);
axis([0,1000,-5,5]);
title('原始信号');

subplot(4,1,2); %Mix_Signal_1 中值滤波后信号
plot(Signal_Filter);
axis([0,1000,-5,5]);
title('中值滤波后的信号');
```

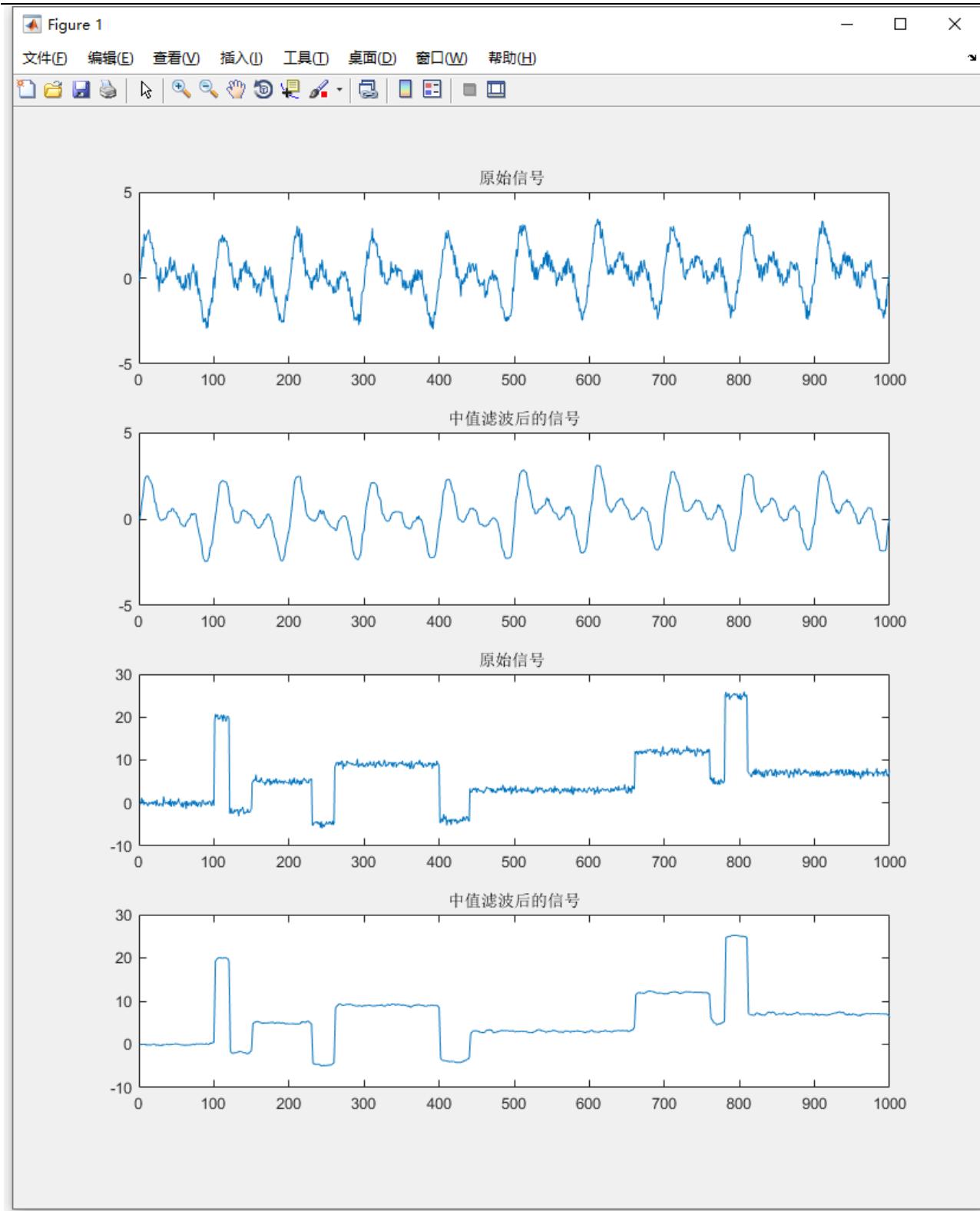
%混合信号 Mix_Signal_2 中值滤波

```
Signal_Filter=medfilt1(Mix_Signal_2,10);

subplot(4,1,3); %Mix_Signal_2 原始信号
plot(Mix_Signal_2);
axis([0,1000,-10,30]);
title('原始信号');

subplot(4,1,4); %Mix_Signal_2 中值滤波后信号
plot(Signal_Filter);
axis([0,1000,-10,30]);
title('中值滤波后的信号');
```

Matlab 运行效果：



48.5 中值滤波器设计

本章的实现是根据中值滤波器原理做了两个函数，一个函数是一块数据的滤波器实现，另一个函数是实时的逐点滤波实现。



48.5.1 函数 MidFilterBlock

函数原型:

```
void MidFilter(float32_t *pSrc, float32_t *pDst, uint32_t blockSize, uint32_t order)
```

函数描述:

这个函数用于一段数据的中值滤波。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是目的数据地址。
- ◆ 第 3 个参数是滤波数据个数，至少为 2。
- ◆ 第 4 个参数是滤波阶数，至少为 2。

48.5.2 函数 MidFilterRT

函数定义如下:

```
void MidFilterRT(float32_t *pSrc, float32_t *pDst, uint8_t ucFlag, uint32_t order)
```

函数描述:

这个函数用于逐个数据的实时滤波。

函数参数:

- ◆ 第 1 个参数是源数据地址。
- ◆ 第 2 个参数是目的数据地址。
- ◆ 第 3 个参数设置为 1 表示首次滤波，后面继续滤波，需将其设置为 0。
- ◆ 第 4 个参数是滤波阶数，至少为 2。

48.5.3 宏定义设置 (重要)

用到两个宏定义，大家根据自己的应用进行设置：

```
#define TEST_LENGTH_SAMPLES 1024      /* 采样点数 */  
#define MidFilterOrder 16             /* 滤波阶数 */
```

第 1 个宏定义：采样点数用于整块数据滤波，一次性滤波的点数。

第 2 个宏定义：设置滤波阶数。

48.5.4 整块数据中值滤波测试

适用于分段数据滤波，测试波形是由原始信号+高斯白噪声+均匀白噪声。

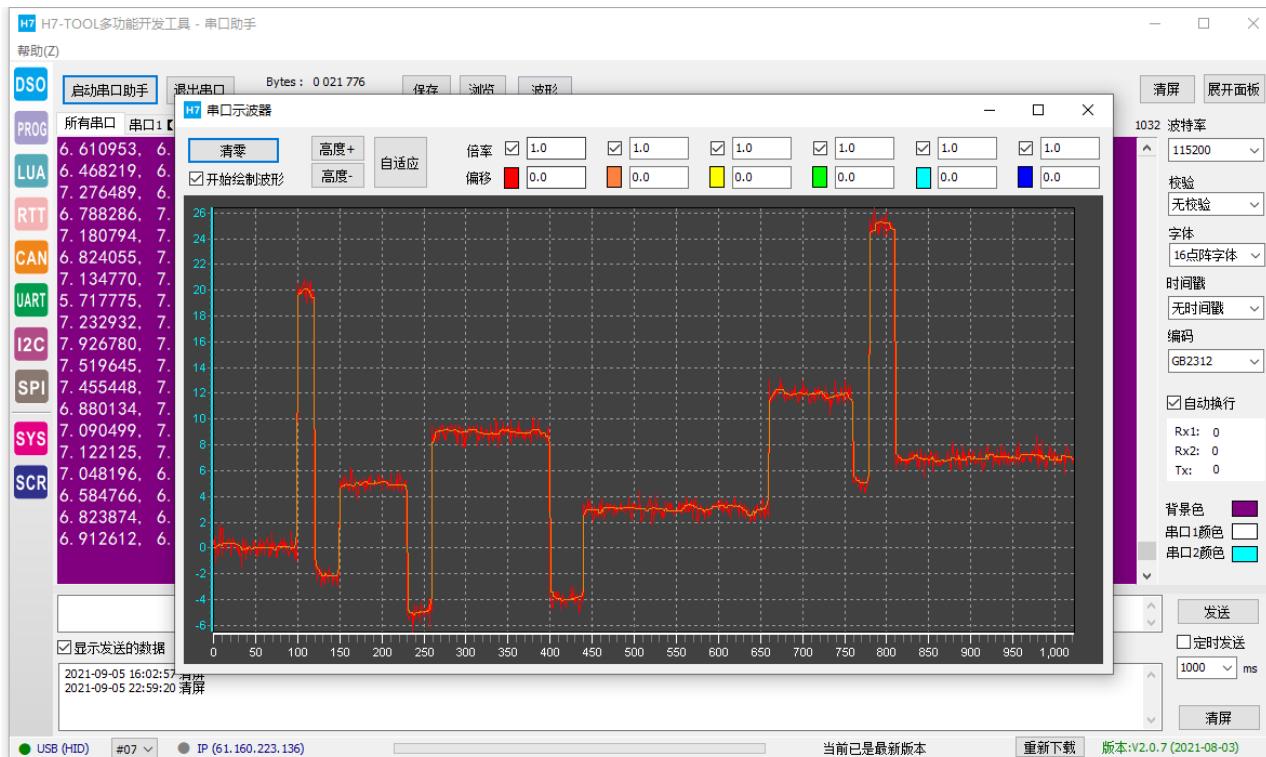
```
/*  
*****  
* 函数名: MidFilterBlockTest
```



```
/* 功能说明：整块数据滤波测试
* 形    参：无
* 返回 值：无
*****
*/
void MidFilterBlockTest(void)
{
    MidFilterBlock((float32_t *)&testdata[0], &DstDate[0], TEST_LENGTH_SAMPLES, MidFilterOrder);

    for(int i = 0; i < TEST_LENGTH_SAMPLES; i++)
    {
        printf("%f, %f\r\n", testdata[i], DstDate[i]);
    }
}
```

滤波器效果，红色是原始波形，杏黄色是滤波后效果：



48.5.5 逐个数据中值滤波测试 (支持实时滤波)

适用于逐个数据的实时滤波，测试波形是由原始信号+高斯白噪声+均匀白噪声。

```
/*
*****
* 函数名：MidFilterOneByOneTest
* 功能说明：逐个数据滤波测试
* 形    参：无
* 返回 值：无
*****
*/
void MidFilterOneByOneTest(void)
{
    float32_t *inputF32, *outputF32;
```



```
inputF32 = (float32_t *)&testdata[0];
outputF32 = &DstDate[0];

/* 从头开始, 先滤第 1 个数据 */
MidFilterRT(inputF32, outputF32, 1, MidFilterOrder);

/* 逐次滤波后续数据 */
for(int i = 1; i < TEST_LENGTH_SAMPLES; i++)
{
    MidFilterRT(inputF32 + i, outputF32 + i, 0, MidFilterOrder);

    for(int i = 0; i < TEST_LENGTH_SAMPLES; i++)
    {
        printf("%f, %f\r\n", testdata[i], DstDate[i]);
    }
}
```

滤波器效果，红色是原始波形，杏黄色是滤波后效果：



48.6 实验例程说明 (MDK)

配套例子：

V7-233_中值滤波器实现，适用于噪声和脉冲过滤(支持逐点实时滤波)

实验目的：

1. 学习中值滤波器。

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. K1 键按下，整块数据滤波测试。



3. K2 键按下，逐个数据滤波器测试。

使用 AC6 注意事项

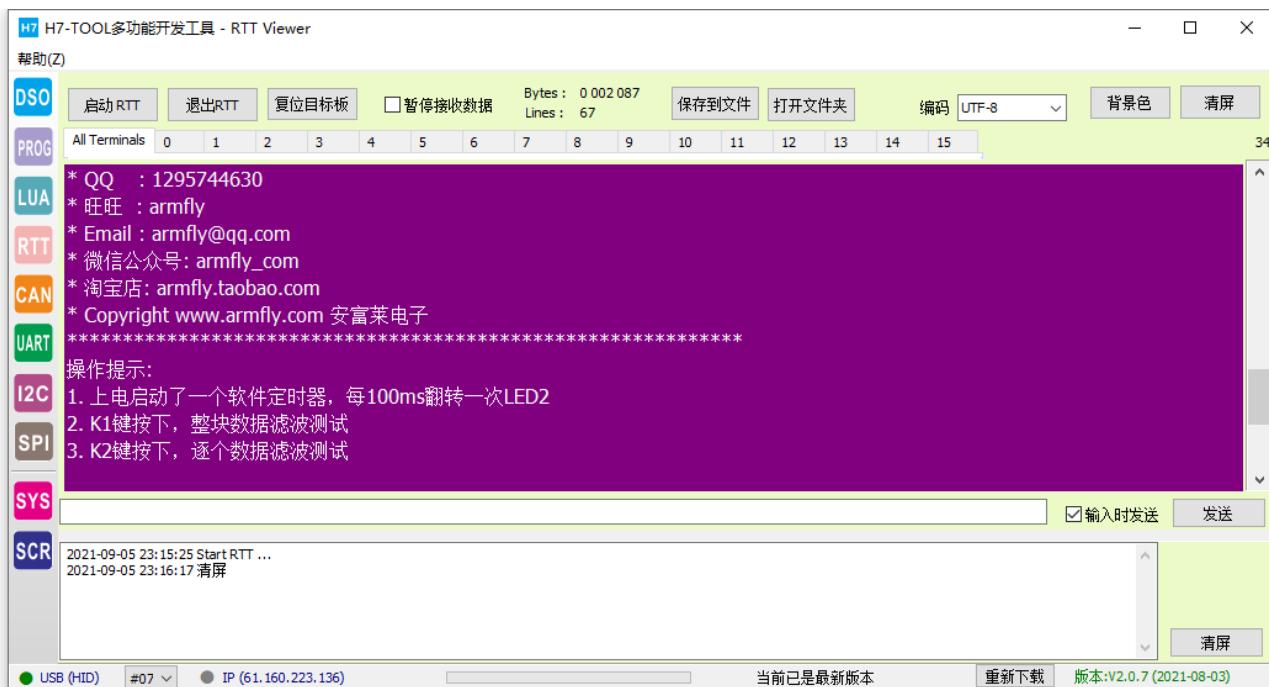
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

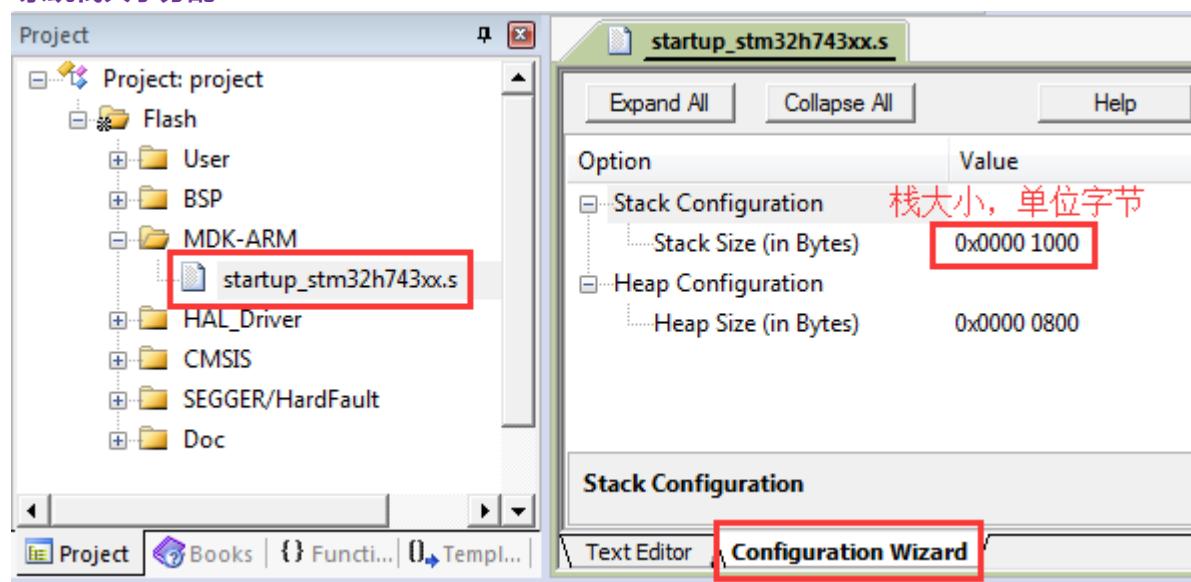


RTT 方式打印信息：

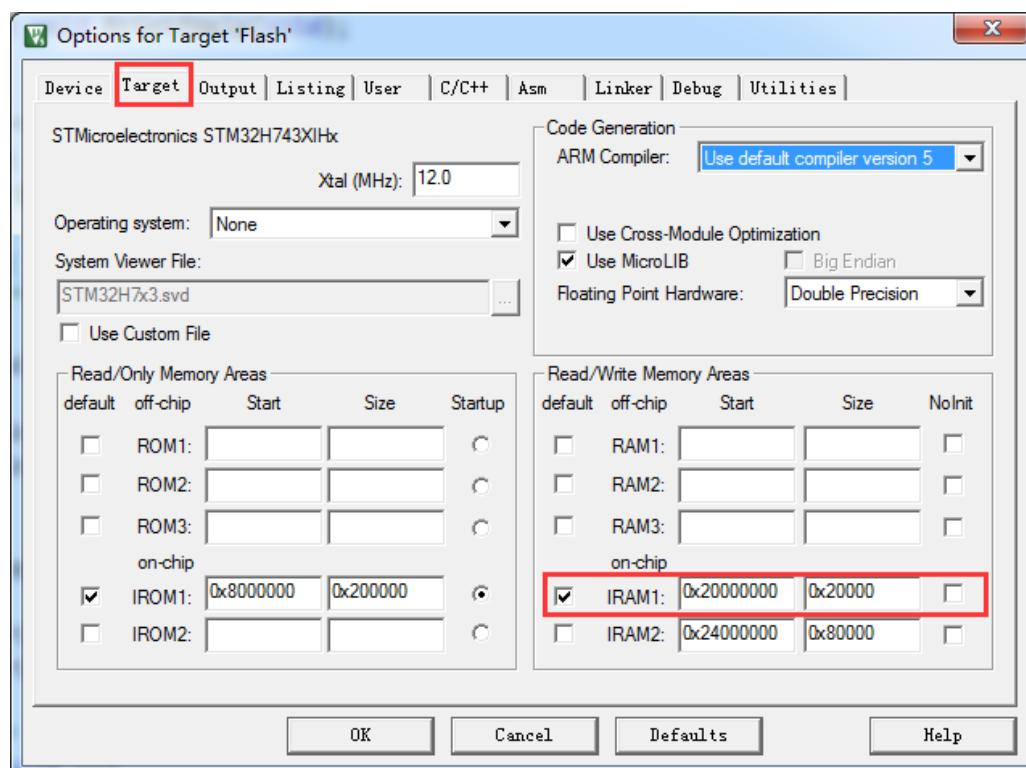


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现:

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
*****
```



```
/* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
     STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
     - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
     - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
     配置系统时钟到 400MHz
     - 切换使用 HSE。
     - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
     Event Recorder:
     - 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
     - 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;
```



```
/* 禁止 MPU */
HAL_MPU_Disable();

/* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number       = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec   = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number       = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec   = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:



主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- K1 键按下，整块数据滤波测试。
- K2 键按下，逐个数据滤波器测试。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2); /* 翻转 LED 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下, 整块数据滤波测试 */
                    MidFilterBlockTest();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下, 逐个数据滤波器测试 */
                    MidFilterOneByOneTest();
                    break;

                default:
                    /* 其它的键值不处理 */
                    break;
            }
        }
    }
}
```

{
}

48.7 实验例程说明 (IAR)

配套例子：

V7-233_中值滤波器实现，适用于噪声和脉冲过滤(支持逐点实时滤波)

实验目的：

1. 学习中值滤波器。

实验内容：

1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. K1 键按下，整块数据滤波测试。
3. K2 键按下，逐个数据滤波器测试。

使用 AC6 注意事项

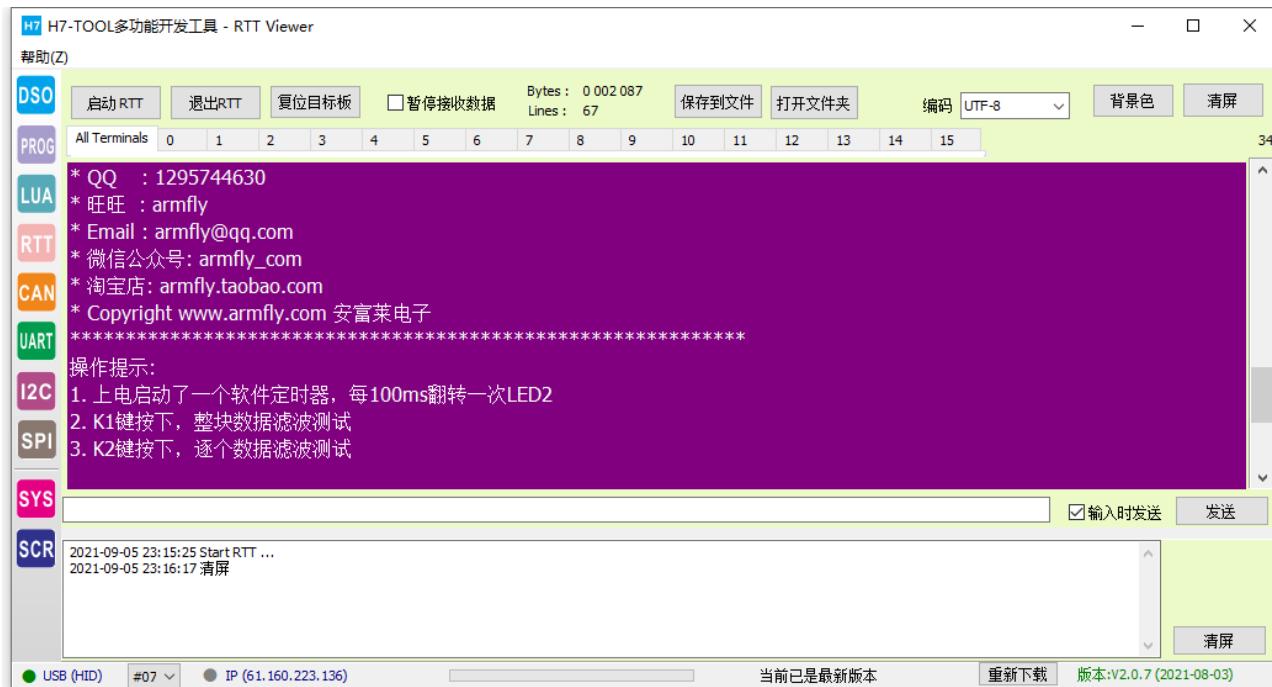
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

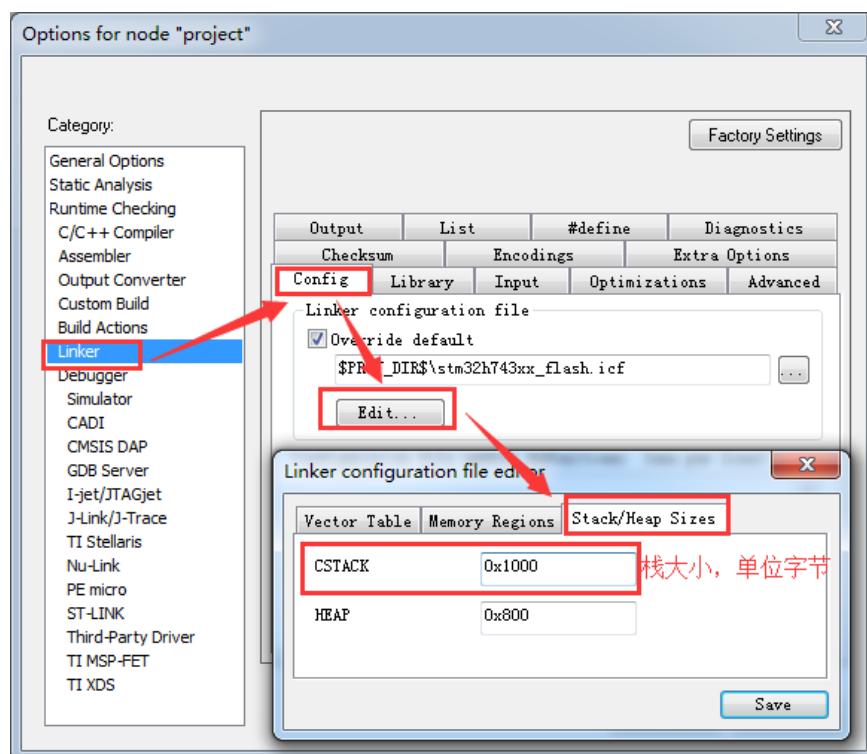


RTT 方式打印信息：

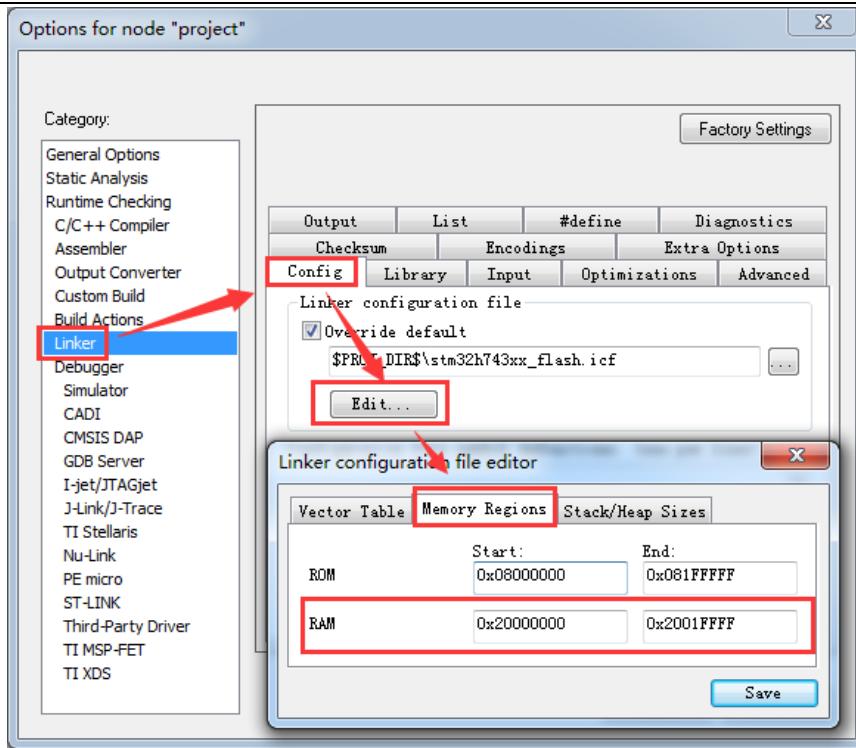


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- K1 键按下，整块数据滤波测试。
- K2 键按下，逐个数据滤波器测试。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();            /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */
}
```



```
/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2); /* 翻转 LED 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:           /* K1 键按下，整块数据滤波测试 */
                MidFilterBlockTest();
                break;

            case KEY_DOWN_K2:           /* K2 键按下，逐个数据滤波器测试 */
                MidFilterOneByOneTest();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
```

48.8 总结

本章节主要讲解了中值滤波器的实现，非常适合噪声滤除场景。



第49章 STM32H7 的自适应滤波器实现，无需

Matlab 生成系数（支持实时滤波）

本章节讲解 LMS 最小均方自适应滤波器实现，无需 Matlab 生成系数，可以自学习。

49.1 初学者重要提示

49.2 自适应滤波器介绍

49.3 LMS 最小均方自适应滤波器介绍

49.4 Matlab 自适应滤波器实现

49.5 自适应滤波器设计

49.6 实验例程说明 (MDK)

49.7 实验例程说明 (IAR)

49.8 总结

49.1 初学者重要提示

- ◆ ARM DSP 库提供了 LMS 最小均方自适应滤波和归一化最小均方自适应滤波器，推荐使用归一化方式，因为归一化方法的步长更容易设置。
- ◆ 自适应滤波器的滤波因数步长设置比较考究，详见本章教程第 5.3 小结。

49.2 自适应滤波器介绍

自适应滤波器能够根据输入信号自动调整滤波系数进行数字滤波。作为对比，非自适应滤波器有静态的滤波器系数，这些静态系数一起组成传递函数。

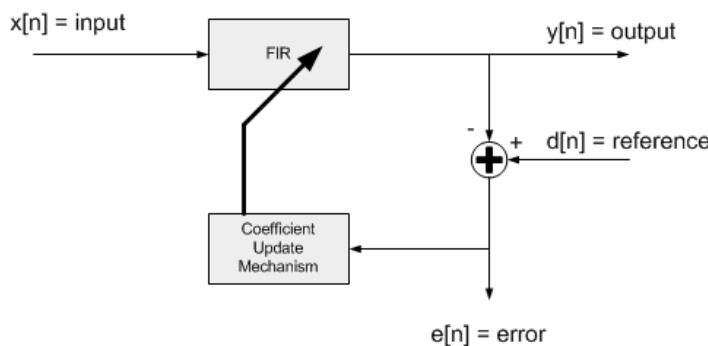
对于一些应用来说，由于事先并不知道需要进行操作的参数，例如一些噪声信号的特性，所以要求使用自适应的系数进行处理。在这种情况下，通常使用自适应滤波器，自适应滤波器使用反馈来调整滤波器系数以及频率响应。

随着处理器性能的增强，自适应滤波器的应用越来越常见，时至今日它们已经广泛地用于手机以及其它通信设备、数码录像机和数码照相机以及医疗监测设备中。

49.3 LMS 最小均方介绍

LMS 最小均方自适应滤波器能够"学习"未知的传输特性。LMS 滤波器使用梯度下降方法，根据瞬时错误信号更新滤波系数。自适应滤波器常用于通信系统、均衡器和降噪。

LMS 滤波器由以下两个部分组成。第一部分是 FIR 滤波器。第二部分是系数更新机制。LMS 滤波器具有两个输入信号。 $x[n]$ 是 FIR 滤波器输入，而参考输入 $d[n]$ 对应 FIR 滤波器的预期输出。更新 FIR 滤波器系数，以便 FIR 滤波器的输出与参考输入匹配。滤波器系数更新机制基于 FIR 滤波器输出和参考输入之间的差异。当滤波器调整时，"错误信号" $e[n]$ 倾向于为零。LMS 处理功能接受输入和参考输入信号，并生成滤波器输出和错误信号。



输出信号 $y[n]$ 由标准 FIR 滤波器计算：

$$y[n] = b[0] * x[n] + b[1] * x[n-1] + b[2] * x[n-2] + \dots + b[\text{numTaps}-1] * x[n-\text{numTaps}+1]$$

误差信号等于参考信号 $d[n]$ 和滤波器输出之间的差值：

$$e[n] = d[n] - y[n].$$

在计算每个样本的误差信号后，计算滤波器状态变量的瞬时能量：

$$E = x[n]^2 + x[n-1]^2 + \dots + x[n-\text{numTaps}+1]^2.$$

然后在逐个样本的基础上更新滤波器系数 $b[k]$ ：

$$b[k] = b[k] + e[n] * (\mu/E) * x[n-k], \text{对于 } k=0, 1, \dots, \text{numTaps}-1$$

其中 μ 是步长，并且控制系数收敛速度。在函数 `arm_lms_norm_init_f32` 中，`pCoeffs` 指向大小为 `numTaps` 的滤波器系数数组。系数按时间倒序存储：

$$\{b[\text{numTaps}-1], b[\text{numTaps}-2], b[\text{N}-2], \dots, b[1], b[0]\}$$

`pState` 指向一个大小为 `numTaps + blockSize - 1` 的状态数组。状态缓冲区中的样本按顺序存储：

$$\{x[n-\text{numTaps}+1], x[n-\text{numTaps}], x[n-\text{numTaps}-1], x[n-\text{numTaps}-2], \dots, x[0], x[1], \dots, x[\text{blockSize}-1]\}$$

注意，状态缓冲区的长度超过了滤波器系数数组的长度 `blockSize-1` 个样本。

49.4 Matlab 自适应滤波器实现

首先创建两个混合信号，便于更好测试滤波器效果。



混合信号 Mix_Signal_1 = 信号 Signal_Original_1+白噪声。

混合信号 Mix_Signal_2 = 信号 Signal_Original_2+白噪声。

```
Fs = 1000; %采样率
N = 1000; %采样点数
n = 0:N-1;
t = 0:1/Fs:1-1/Fs; %时间序列
Signal_Original_1 = sin(2*pi*10*t)+sin(2*pi*20*t)+sin(2*pi*30*t);
Noise_White_1 = [0.3*randn(1, 500), rand(1, 500)]; %前 500 点高斯分部白噪声，后 500 点均匀分布白噪声
Mix_Signal_1 = Signal_Original_1 + Noise_White_1; %构造的混合信号

Signal_Original_2 = [zeros(1, 100), 20*ones(1, 20), -2*ones(1, 30), 5*ones(1, 80), -5*ones(1, 30),
9*ones(1, 140), -4*ones(1, 40), 3*ones(1, 220),
12*ones(1, 100), 5*ones(1, 20), 25*ones(1, 30), 7*ones(1, 190)];

Noise_White_2 = 0.5*randn(1, 1000); %高斯白噪声
Mix_Signal_2 = Signal_Original_2 + Noise_White_2; %构造的混合信号
```

滤波代码实现如下：

```
%*****
%
% 信号 Mix_Signal_1 和 Mix_Signal_2 分别作自适应滤波
%
%*****
%
%混合信号 Mix_Signal_1 自适应滤波
N=1000; %输入信号抽样点数 N
k=100; %时域抽头 LMS 算法滤波器阶数
u=0.001; %步长因子

%设置初值
yn_1=zeros(1, N); %output signal
yn_1(1:k)=Mix_Signal_1(1:k); %将输入信号 SignalAddNoise 的前 k 个值作为输出 yn_1 的前 k 个值
w=zeros(1, k); %设置抽头加权初值
e=zeros(1, N); %误差信号

%用 LMS 算法迭代滤波
for i=(k+1):N
    XN=Mix_Signal_1((i-k+1):(i));
    yn_1(i)=w*XN';
    e(i)=Signal_Original_1(i)-yn_1(i);
    w=w+2*u*e(i)*XN;
end

subplot(4, 1, 1);
plot(Mix_Signal_1); %Mix_Signal_1 原始信号
axis([k+1, 1000, -4, 4]);
title('原始信号');

subplot(4, 1, 2);
plot(yn_1); %Mix_Signal_1 自适应滤波后信号
axis([k+1, 1000, -4, 4]);
title('自适应滤波后信号');

%混合信号 Mix_Signal_2 自适应滤波
N=1000; %输入信号抽样点数 N
k=500; %时域抽头 LMS 算法滤波器阶数
u=0.000011; %步长因子
```

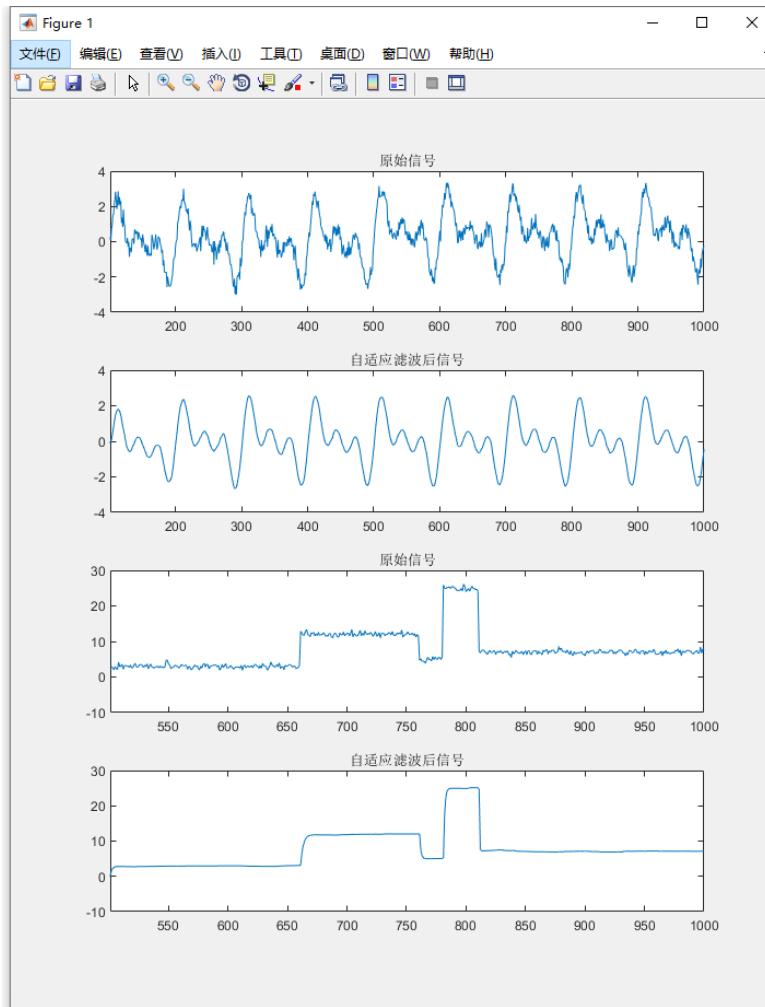
```
%设置初值
yn_1=zeros(1,N);
yn_1(1:k)=Mix_Signal_2(1:k);
w=zeros(1,k);
e=zeros(1,N);

%用 LMS 算法迭代滤波
for i=(k+1):N
    XN=Mix_Signal_2((i-k+1):(i));
    yn_1(i)=w*XN';
    e(i)=Signal_Original_2(i)-yn_1(i);
    w=w+2*u*e(i)*XN;
end

subplot(4, 1, 3);
plot(Mix_Signal_2); %Mix_Signal_1 原始信号
axis([k+1, 1000, -10, 30]);
title('原始信号');

subplot(4, 1, 4);
plot(yn_1); %Mix_Signal_1 自适应滤波后信号
axis([k+1, 1000, -10, 30]);
title('自适应滤波后信号');
```

Matlab 运行效果：





49.5 自适应器设计

自适应滤波器的主要通过下面两个函数实现，支持逐点实时滤波。

49.5.1 函数 arm_lms_norm_init_f32

函数原型：

```
void arm_lms_norm_init_f32(
    arm_lms_norm_instance_f32 * S,
    uint16_t numTaps,
    float32_t * pCoeffs,
    float32_t * pState,
    float32_t mu,
    uint32_t blockSize);
```

函数描述：

此函数用于自适应滤波器初始化。

函数参数：

- ◆ 第 1 个参数是 arm_lms_norm_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是滤波因数个数。
- ◆ 第 3 个参数是滤波因数地址。
- ◆ 第 4 个参数指向状态变量数组，这个数组用于函数内部计算数据的缓存。
- ◆ 第 5 个参数用于设置滤波因数更新的步长。
- ◆ 第 6 个参数是每次处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

49.5.2 函数 arm_lms_norm_f32

函数原型：

```
void arm_lms_norm_f32(
    arm_lms_norm_instance_f32 * S,
    const float32_t * pSrc,
    float32_t * pRef,
    float32_t * pOut,
    float32_t * pErr,
    uint32_t blockSize);
```

函数描述：

此函数用于自适应滤波器实时滤波。

函数参数：

- ◆ 第 1 个参数是 arm_lms_norm_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是源数据地址。
- ◆ 第 3 个参数是参考数据地址，需要用户提供想要逼近的波形效果。
- ◆ 第 4 个参数是输出数据地址。
- ◆ 第 5 个参数是误差数据地址。
- ◆ 第 6 个参数是每次处理的数据个数，最小可以每次处理 1 个数据，最大可以每次全部处理完。

**注意事项：**

结构体 arm_lms_norm_instance_f32 的定义如下 (在文件 arm_math.h 文件) :

```
typedef struct
{
    uint16_t numTaps;      /**< number of coefficients in the filter. */
    float32_t *pState;     /**< points to the state variable array. The array is of length
                           numTaps+blockSize-1. */
    float32_t *pCoeffs;    /**< points to the coefficient array. The array is of length numTaps. */
    float32_t mu;          /**< step size that control filter coefficient updates. */
    float32_t energy;     /**< saves previous frame energy. */
    float32_t x0;          /**< saves previous input sample. */
} arm_lms_norm_instance_f32;
```

1. 参数 numTaps 用于设置滤波因数个数。
2. pState 指向状态变量数组，这个数组用于函数内部计算数据的缓存。
3. 参数 pCoeffs 指向滤波因数，滤波因数数组长度为 numTaps。但要注意 pCoeffs 指向的滤波因数应该按照如下的逆序进行排列：
 $\{b[numTaps-1], b[numTaps-2], b[N-2], \dots, b[1], b[0]\}$
但满足线性相位特性的 FIR 滤波器具有奇对称或者偶对称的系数，偶对称时逆序排列还是他本身。
4. 参数 mu 用于设置滤波因数更新的步长。
5. blockSize 这个参数的大小没有特殊要求，最小可以每次处理 1 个数据，最大可以每次全部处理完。

49.5.3 滤除 200Hz 正弦波测试（含不同步长测试，重要）

原始波形200Hz + 50Hz正弦波，滤除200Hz正弦波测试：

```
/*
*****
* 函数名: arm_lms_f32_test1
* 功能说明: 原始波形 200Hz + 50Hz 正弦波, 滤除 200Hz 正弦波。
* 形参: 无
* 返回值: 无
*****
*/
void arm_lms_f32_test1(void)
{
    uint32_t i;
    float32_t *inputF32, *outputF32, *inputREF, *outputERR;
    arm_lms_norm_instance_f32 lmsS={0};

    for(i=0; i<TEST_LENGTH_SAMPLES; i++)
    {
        /* 50Hz 正弦波+200Hz 正弦波, 采样率 1KHz */
        testInput_f32_50Hz_200Hz[i] = arm_sin_f32(2*3.1415926f*50*i/1000) +
                                       arm_sin_f32(2*3.1415926f*200*i/1000);
        testInput_f32_REF[i] = arm_sin_f32(2*3.1415926f*50*i/1000);
    }

    /* 如果是实时性的滤波, 仅需清零一次 */
    memset(lmsCoeffs32, 0, sizeof(lmsCoeffs32));
    memset(lmsStateF32, 0, sizeof(lmsStateF32));
}
```



```
/* 初始化输入输出缓存指针 */
inputF32 = (float32_t *)&testInput_f32_50Hz_200Hz[0]; /* 原始波形 */
outputF32 = (float32_t *)&testOutput[0]; /* 滤波后输出波形 */
inputREF = (float32_t *)&testInput_f32_REF[0]; /* 参考波形 */
outputERR = (float32_t *)&test_f32_ERR[0]; /* 误差数据 */

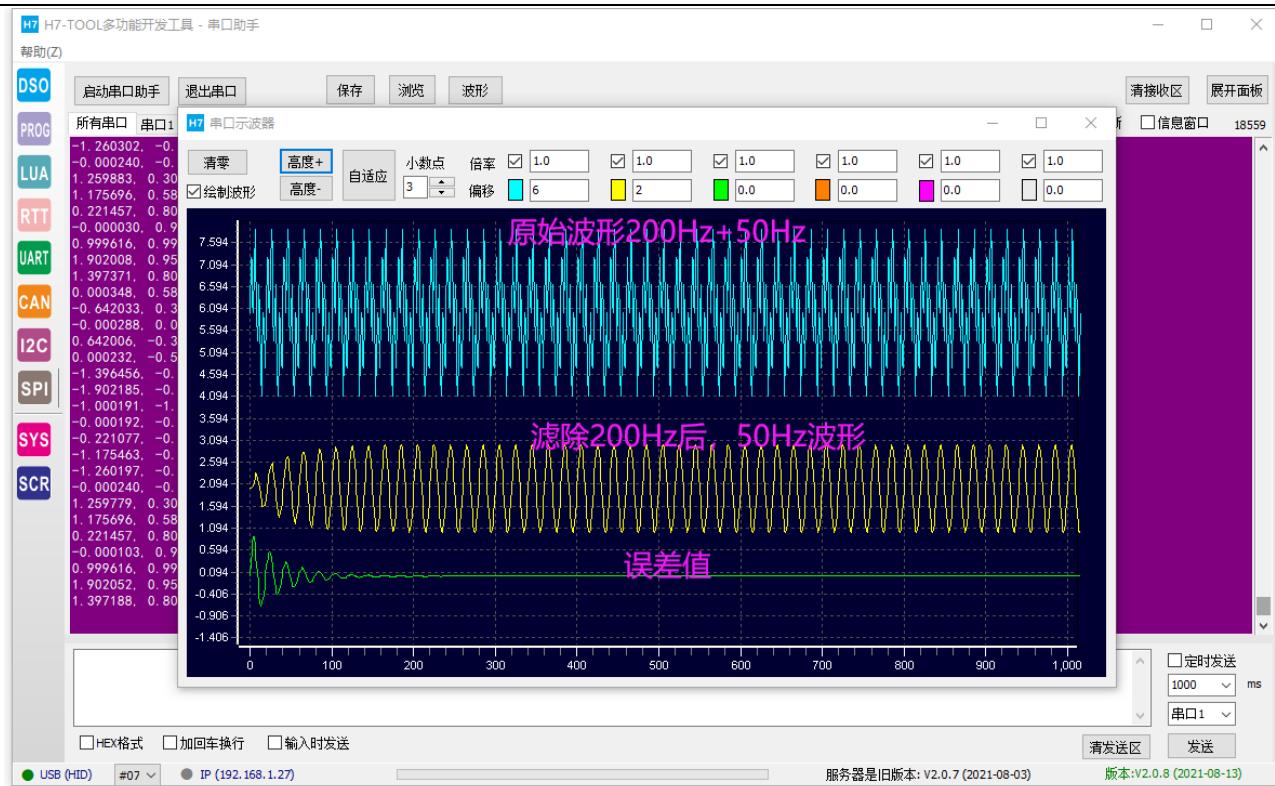
/* 归一化LMS初始化 */
arm_lms_norm_init_f32 (&lmsS, /* LMS结构体 */
                       NUM_TAPS, /* 滤波器系数个数 */
                       (float32_t *)&lmsCoeffs32[0], /* 滤波 */
                       &lmsStateF32[0], /* 滤波器系数 */
                       0.1, /* 步长 */
                       blockSize); /* 处理的数据个数 */

/* 实现LMS自适应滤波，这里每次处理1个点 */
for(i=0; i < numBlocks; i++)
{
    arm_lms_norm_f32(&lmsS, /* LMS结构体 */
                      inputF32 + (i * blockSize), /* 输入数据 */
                      inputREF + (i * blockSize), /* 输出数据 */
                      outputF32 + (i * blockSize), /* 参考数据 */
                      outputERR + (i * blockSize), /* 误差数据 */
                      blockSize); /* 处理的数据个数 */

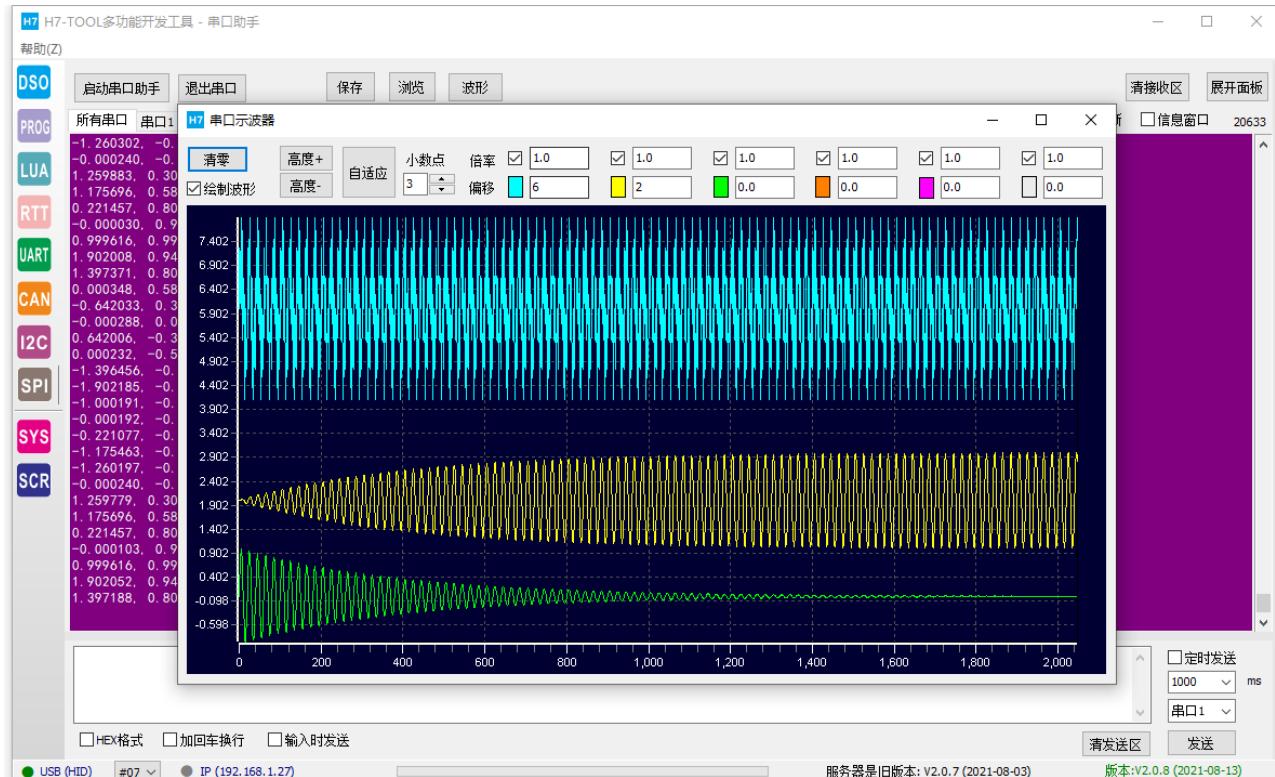
}

/* 打印滤波后结果 */
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    printf("%f, %f, %f\r\n", testInput_f32_50Hz_200Hz[i], outputF32[i], test_f32_ERR[i]);
}
```

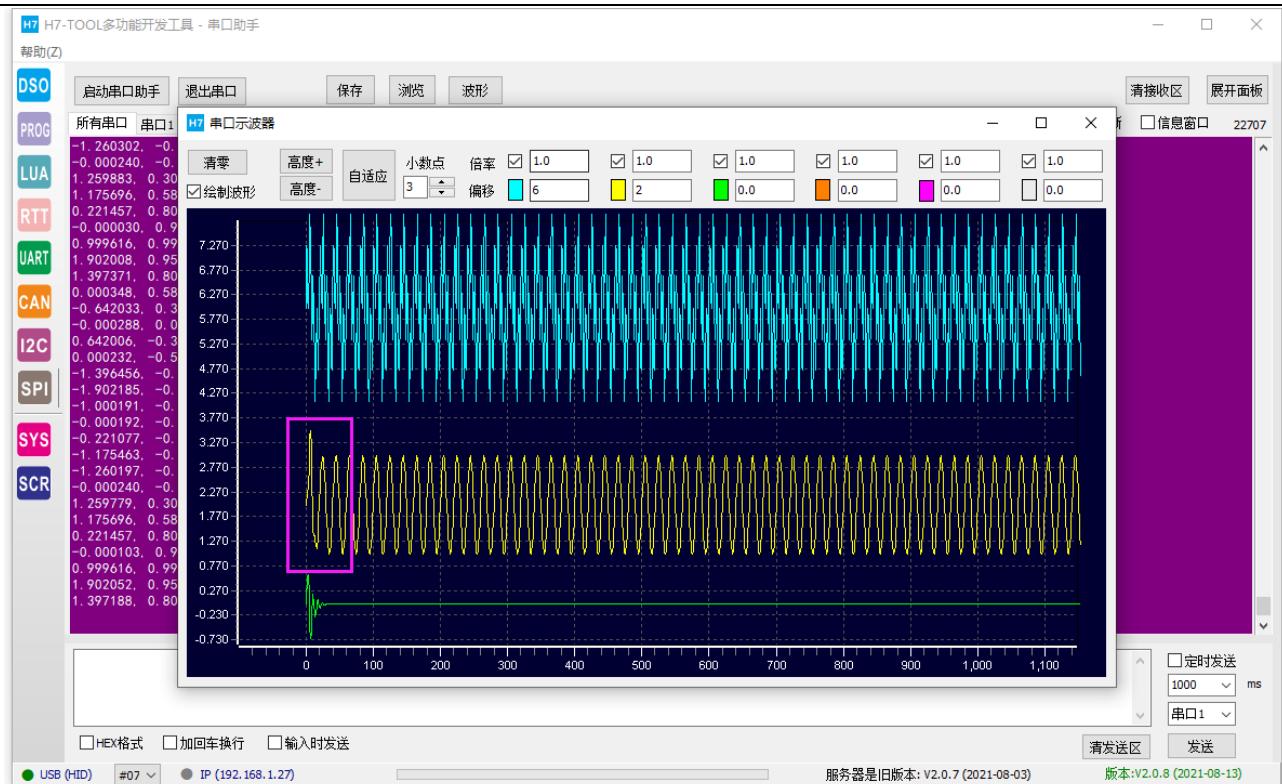
下面是滤波因数步长调节为0.1时的滤波器效果，浅蓝色是原始波形，黄色是滤波后波形，绿色是误差值：



步长为0.01时效果，可以看到逼近参考波形的速度较慢：



步长为1的效果，逼近参考波形的速度更快，前面的波形由一段陡增。



关于步长，没有特别好的方式直接锁定那种步长大小更合适，一般的处理思路是按照10倍关系先锁定范围，比如先测试步长为1, 0.1, 0.001等来测试，然后进一步设置一个合适的值。

49.5.4 滤除白噪声测试（一）

原始波形由任意波形+ 高斯分布白噪声 + 均匀分布白噪声组成，滤除高斯分布白噪声 + 均匀分布白噪声。

```
/*
*****
* 函数名: arm_lms_f32_test2
* 功能说明: 原始波形由任意波形+ 高斯分布白噪声 + 均匀分布白噪声组成，滤除高斯分布白噪声 + 均匀分布白噪声。
* 形参: 无
* 返回值: 无
*****
*/
void arm_lms_f32_test2(void)
{
    uint32_t i;
    arm_lms_norm_instance_f32 lmsS;
    float32_t *inputF32, *outputF32, *inputREF, *outputERR;

    /* 如果是实时性的滤波，仅需清零一次 */
    memset(lmsCoeffs32, 0, sizeof(lmsCoeffs32));
    memset(lmsStateF32, 0, sizeof(lmsStateF32));

    /* 初始化输入输出缓存指针 */
    inputF32 = (float32_t *)&MixData[0];      /* 原始波形 */
    outputF32 = (float32_t *)&testOutput[0];   /* 滤波后输出波形 */
    inputREF = (float32_t *)&OriginalData[0]; /* 参考波形 */
    outputERR = (float32_t *)&test_f32_ERR[0]; /* 误差数据 */
```



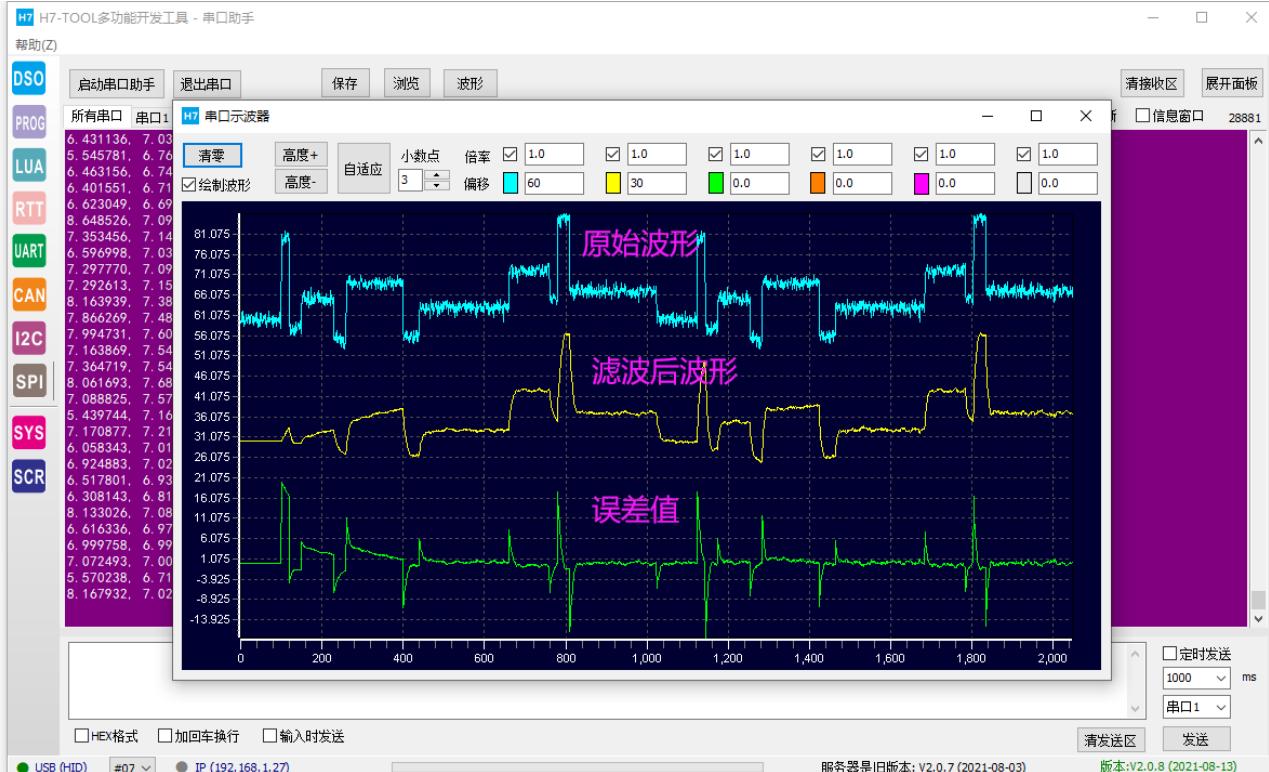
```
/* 归一化 LMS 初始化 */
arm_lms_norm_init_f32 (&lmsS,
    NUM_TAPS,
    (float32_t *)&lmsCoeffs32[0], /* 滤波 */
    &lmsStateF32[0],
    0.01, /* 步长 */
    blockSize); /* 处理的数据个数 */

/* 实现 LMS 自适应滤波，这里每次处理 1 个点 */
for(i=0; i < numBlocks; i++)
{
    arm_lms_norm_f32(&lmsS, /* LMS结构体 */
        inputF32 + (i * blockSize), /* 输入数据 */
        inputREF + (i * blockSize), /* 输出数据 */
        outputF32 + (i * blockSize), /* 参考数据 */
        outputERR + (i * blockSize), /* 误差数据 */
        blockSize); /* 处理的数据个数 */
}

/* 打印滤波后结果 */
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    printf("%f, %f, %f\r\n", MixData[i], outputF32[i], test_f32_ERR[i]);
}

}
```

下面是滤波因数步长调节为0.01时的滤波器效果，浅蓝色是原始波形，黄色是滤波后波形，绿色是误差值：



这个波形做了两个周期，前1024点和后1024点，后面1024点滤除白噪声的效果已经比较好，而前1024点的前半段一直在逼近我们设置的参考波形中。另外从误差值波形中，我们可以看到原始波形跳变的



地方，误差值也会有一个跳变，然后向0趋近。这是自适应滤波器特性决定的，不断的调节滤波器系数中。

我们再来看下将滤波因数步长调节为0.1时的效果：



可以看到逼近速度很快，但是逼近效果一般，也就是白噪声的滤除效果一般。

49.5.5 滤除白噪声测试（二）

原始波形10Hz正弦波 + 20Hz正弦波 + 30Hz正弦波 + 高斯分布白噪声 + 均匀分布白噪声。滤除高斯分布白噪声 + 均匀分布白噪声。

```
/*
*****
* 函数名: arm_lms_f32_test3
* 功能说明: 10Hz 正弦波 + 20Hz 正弦波 + 30Hz 正弦波 + 高斯分布白噪声 + 均匀分布白噪声，滤除高斯分布白噪声 + 均匀分布白噪声
* 形参: 无
* 返回值: 无
*****
*/
void arm_lms_f32_test3(void)
{
    uint32_t i;
    arm_lms_norm_instance_f32 lmsS;
    float32_t *inputF32, *outputF32, *inputREF, *outputERR;

    /* 如果是实时性的滤波，仅需清零一次 */
    memset(lmsCoeffs32, 0, sizeof(lmsCoeffs32));
    memset(lmsStateF32, 0, sizeof(lmsStateF32));
```



```
/* 初始化输入输出缓存指针 */
inputF32 = (float32_t *)&MixData1[0];           /* 原始波形 */
outputF32 = (float32_t *)&testOutput[0];        /* 滤波后输出波形 */
inputREF = (float32_t *)&OriginalData1[0];      /* 参考波形 */
outputERR = (float32_t *)&test_f32_ERR[0];       /* 误差数据 */

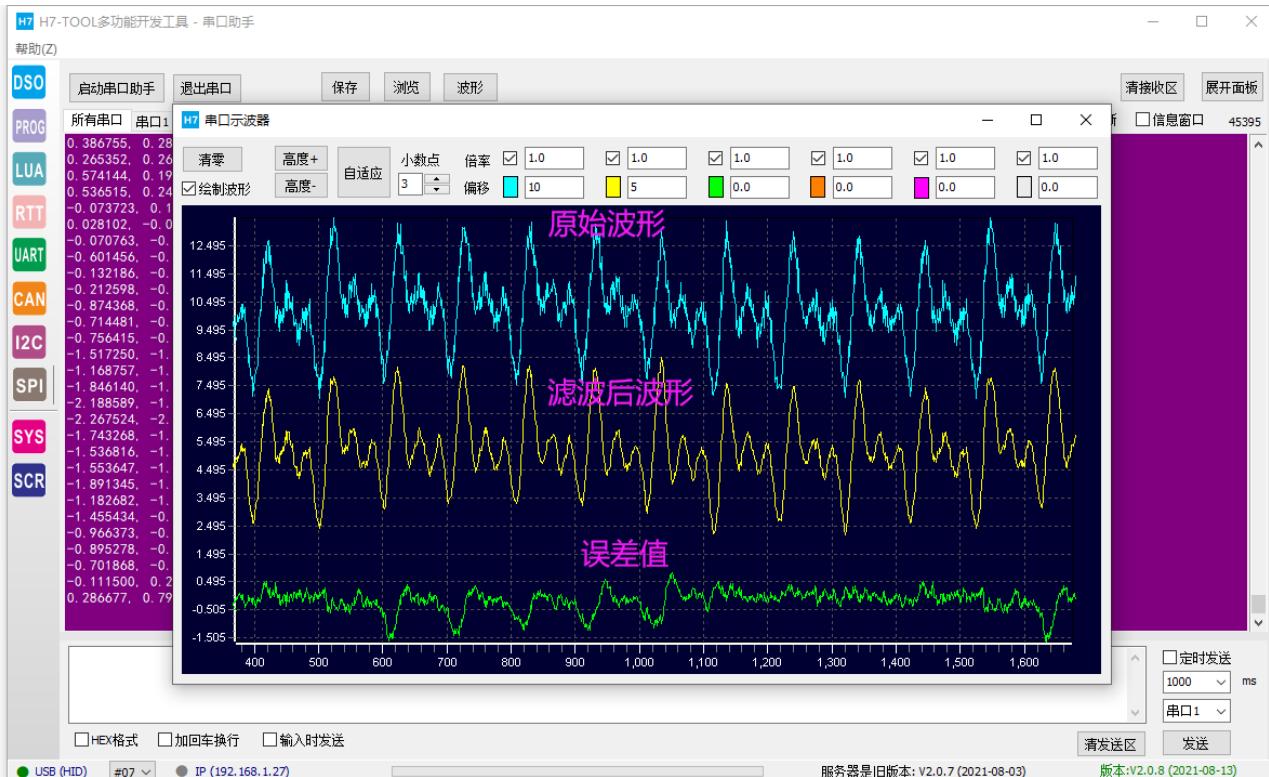
/* 归一化 LMS 初始化 */
arm_lms_norm_init_f32 (&lmsS,                  /* LMS结构体 */
                       NUM_TAPS,             /* 滤波器系数个数 */
                       (float32_t *)&lmsCoeffs32[0], /* 滤波 */
                       &lmsStateF32[0],       /* 滤波器系数 */
                       0.1,                  /* 步长 */
                       blockSize);           /* 处理的数据个数 */

/* 实现 LMS 自适应滤波，这里每次处理 1 个点 */
for(i=0; i < numBlocks; i++)
{
    arm_lms_norm_f32(&lmsS, /* LMS结构体 */
                      inputF32 + (i * blockSize), /* 输入数据 */
                      inputREF + (i * blockSize), /* 输出数据 */
                      outputF32 + (i * blockSize), /* 参考数据 */
                      outputERR + (i * blockSize), /* 误差数据 */
                      blockSize);               /* 处理的数据个数 */
}

/* 打印滤波后结果 */
for(i=0; i<TEST_LENGTH_SAMPLES; i++)
{
    printf("%f, %f, %f\r\n", MixData1[i], outputF32[i], test_f32_ERR[i]);
}

}
```

下面是滤波因数步长调节为0.1时的滤波器效果，浅蓝色是原始波形，黄色是滤波后波形，绿色是误差值：





49.6 实验例程说明 (MDK)

配套例子：

V7-234_自适应滤波器实现，无需 Matlab 生成系数（支持实时滤波）

实验目的：

1. 学习 LMS 最小均方滤波器。

实验内容：

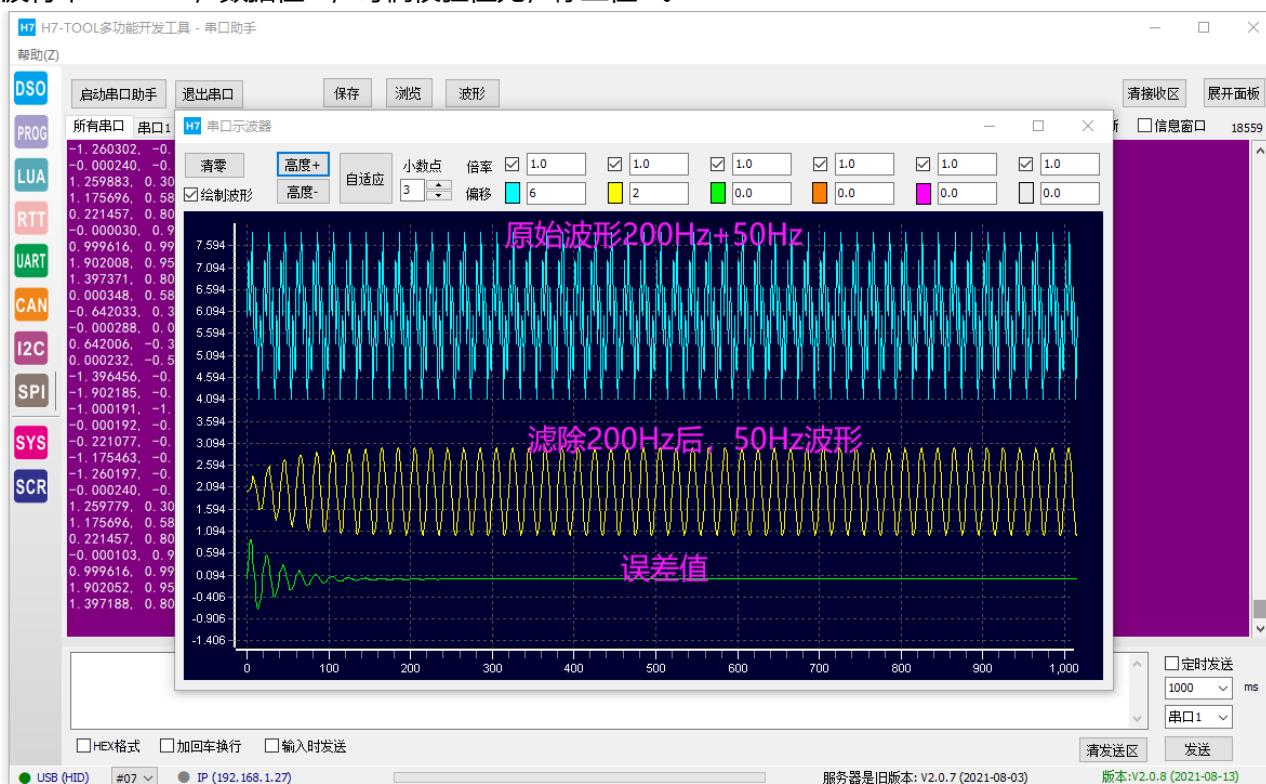
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印测试波形 1 和滤波后的波形数据。
3. 按下按键 K2，打印测试波形 2 和滤波后的波形数据。
4. 按下按键 K3，打印测试波形 3 和滤波后的波形数据。

使用 AC6 注意事项

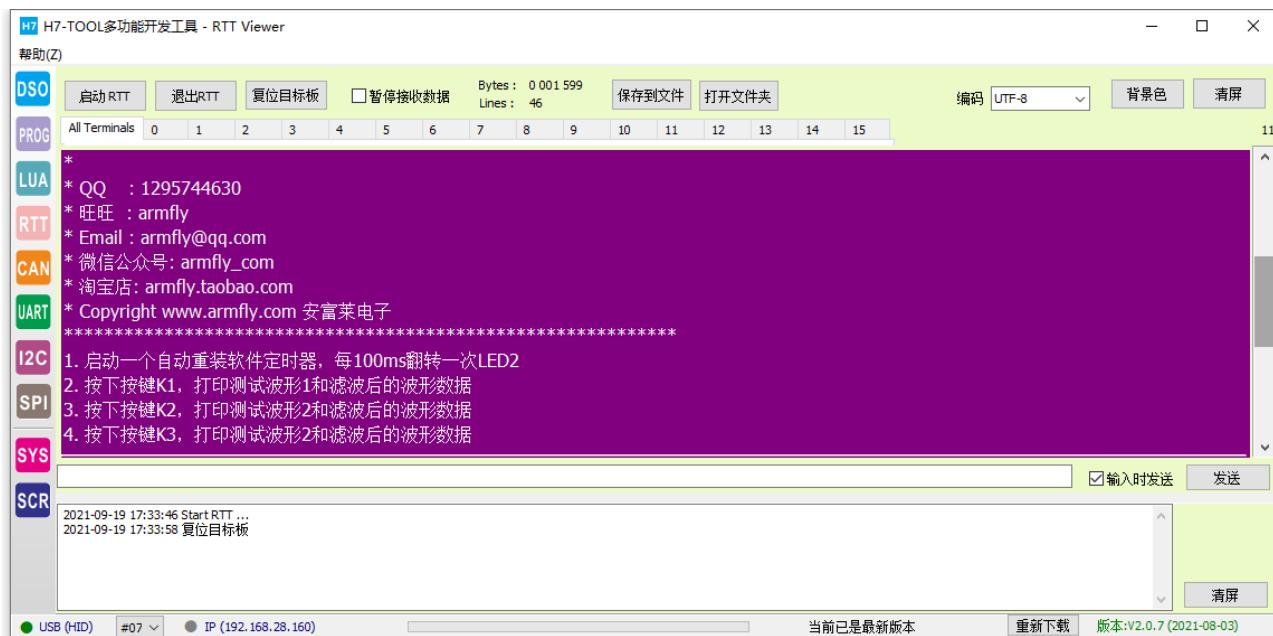
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

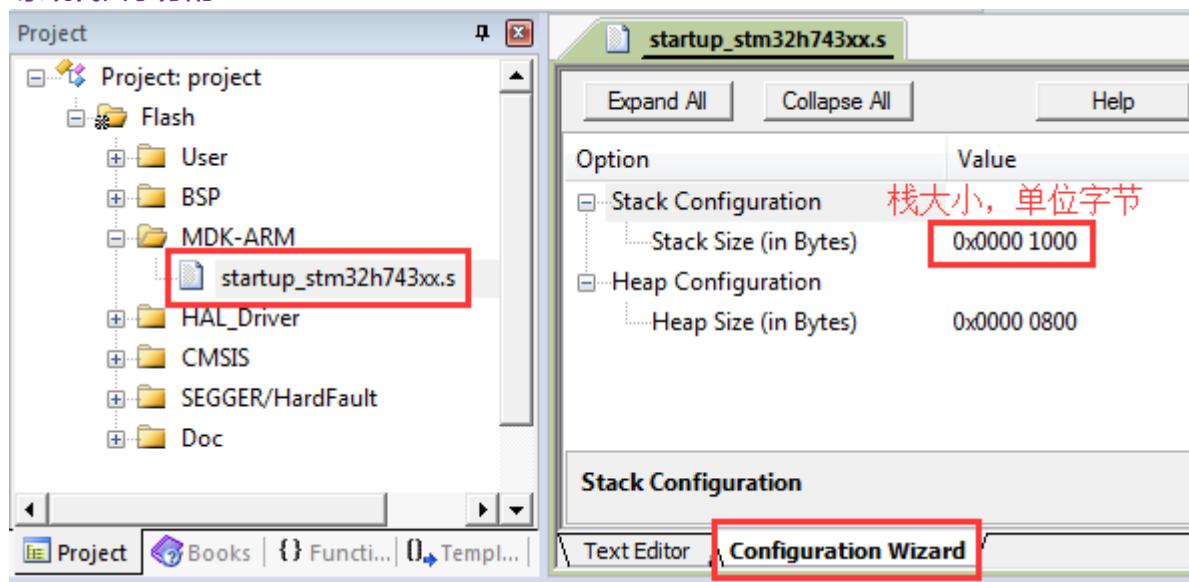


RTT 方式打印信息：

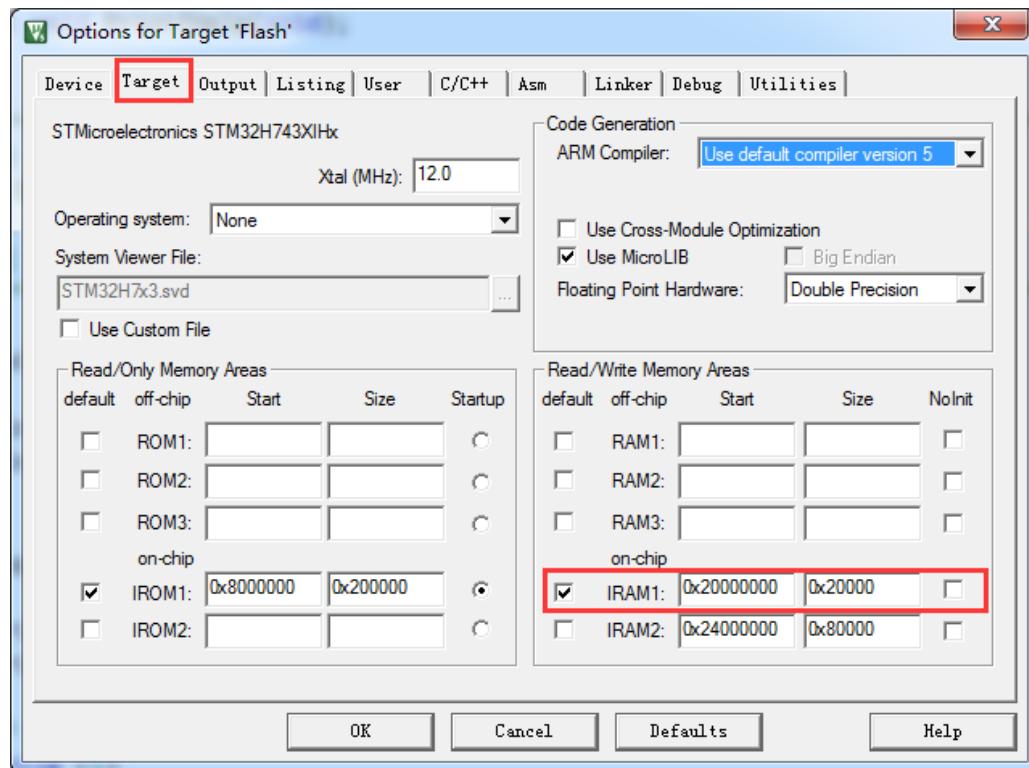


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
        STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
        - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
        - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
        配置系统时钟到 400MHz
        - 切换使用 HSE。
        - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
/*
Event Recorder:
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章
*/
#endif

#if Enable_EventRecorder == 1
/* 初始化 EventRecorder 并开启 */
EventRecorderInitialize(EventRecordAll, 1U);
EventRecorderStart();
#endif

bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */
bsp_InitTimer(); /* 初始化滴答定时器 */
bsp_InitUart(); /* 初始化串口 */
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */
bsp_InitLed(); /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*
*****
* 函数名: MPU_Config
* 功能说明: 配置 MPU
* 形参: 无
* 返回值: 无
*****
*/
static void MPU_Config( void )
{
    MPU_Region_InitTypeDef MPU_InitStruct;

    /* 禁止 MPU */
    HAL_MPU_Disable();

    /* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x24000000;
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
    MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
    MPU_InitStruct.SubRegionDisable = 0x00;
    MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

    HAL_MPU_ConfigRegion(&MPU_InitStruct);

    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
    MPU_InitStruct.BaseAddress = 0x60000000;
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
```



```
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable     = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable      = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number           = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField    = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印测试波形 1 和滤波后的波形数据。
- 按下按键 K2，打印测试波形 2 和滤波后的波形数据。
- 按下按键 K3，打印测试波形 3 和滤波后的波形数据。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();            /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */
}
```



```
PrintfHelp(); /* 打印操作提示信息 */
```

```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

/* 进入主程序循环体 */
while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2); /* 翻转 LED 的状态 */
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值，无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1:          /* K1 键按下 */
                arm_lms_f32_test1();
                break;

            case KEY_DOWN_K2:          /* K2 键按下 */
                arm_lms_f32_test2();
                break;

            case KEY_DOWN_K3:          /* K3 键按下 */
                arm_lms_f32_test3();
                break;

            default:
                /* 其它的键值不处理 */
                break;
        }
    }
}
```

49.7 实验例程说明 (IAR)

配套例子：

V7-234_自适应滤波器实现，无需 Matlab 生成系数（支持实时滤波）

实验目的：

1. 学习 LMS 最小均方滤波器。

实验内容：

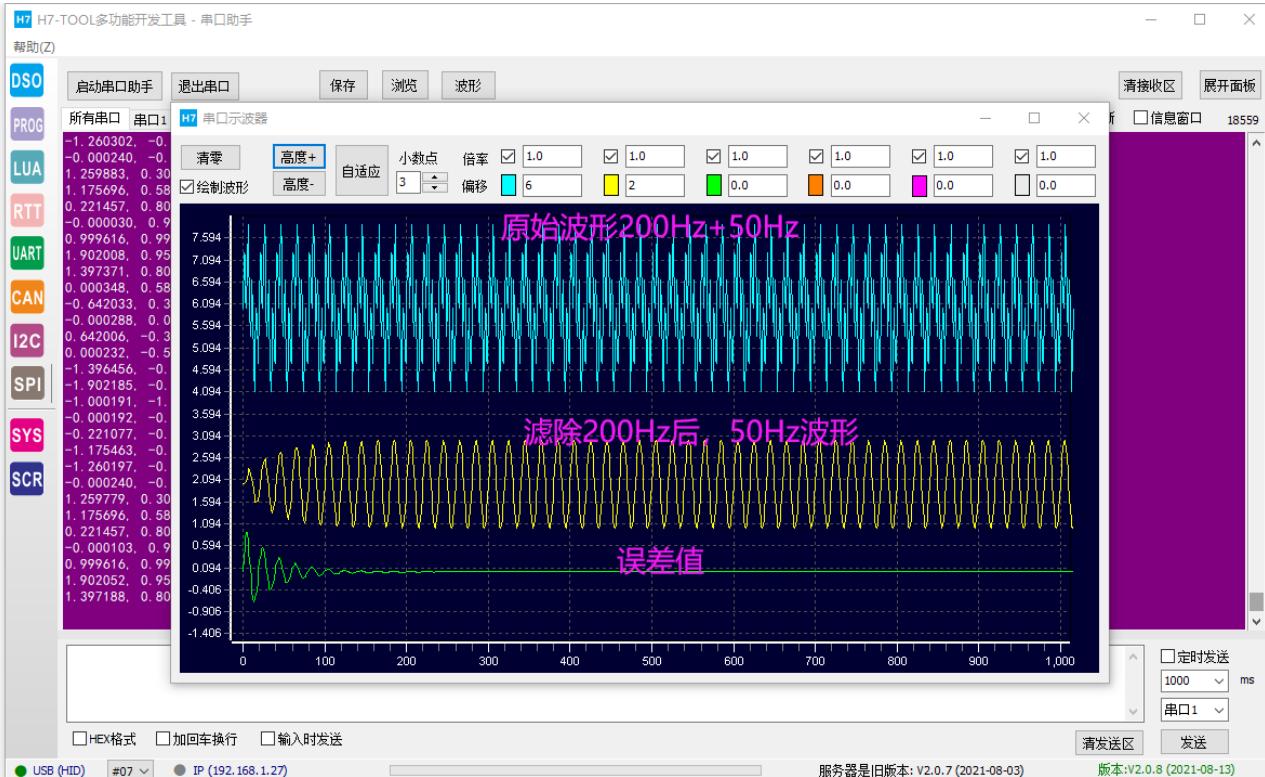
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. 按下按键 K1，打印测试波形 1 和滤波后的波形数据。



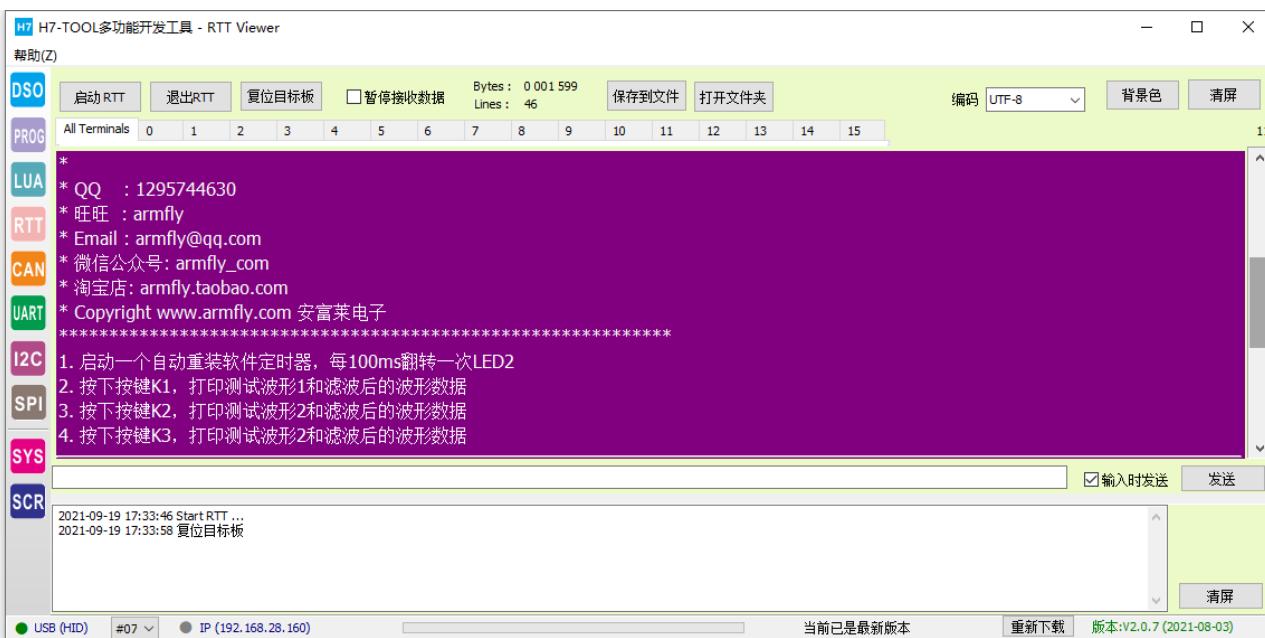
3. 按下按键 K2，打印测试波形 2 和滤波后的波形数据。
4. 按下按键 K3，打印测试波形 3 和滤波后的波形数据。

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

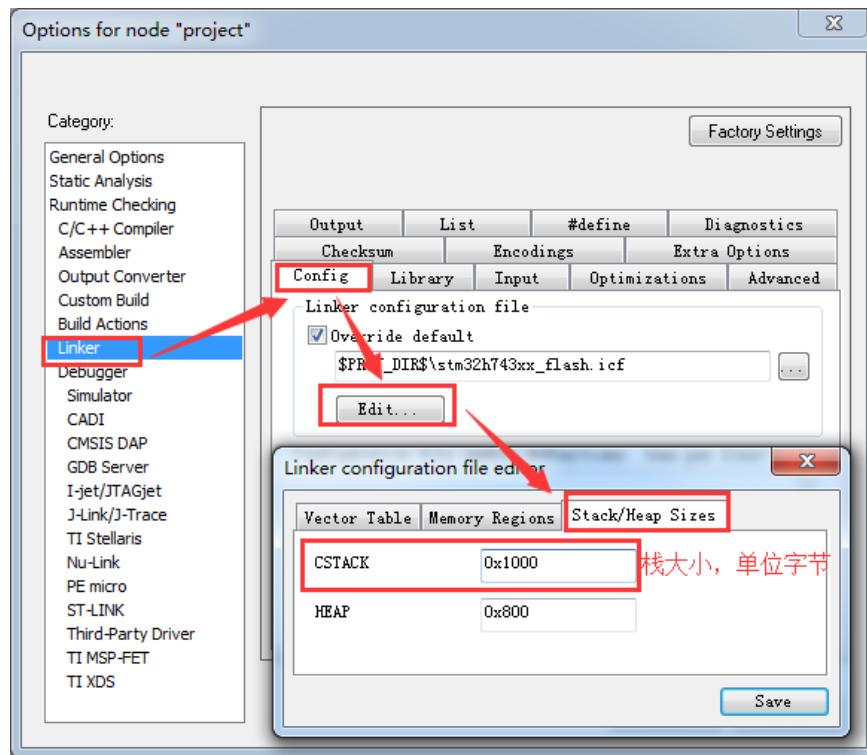


RTT 方式打印信息：

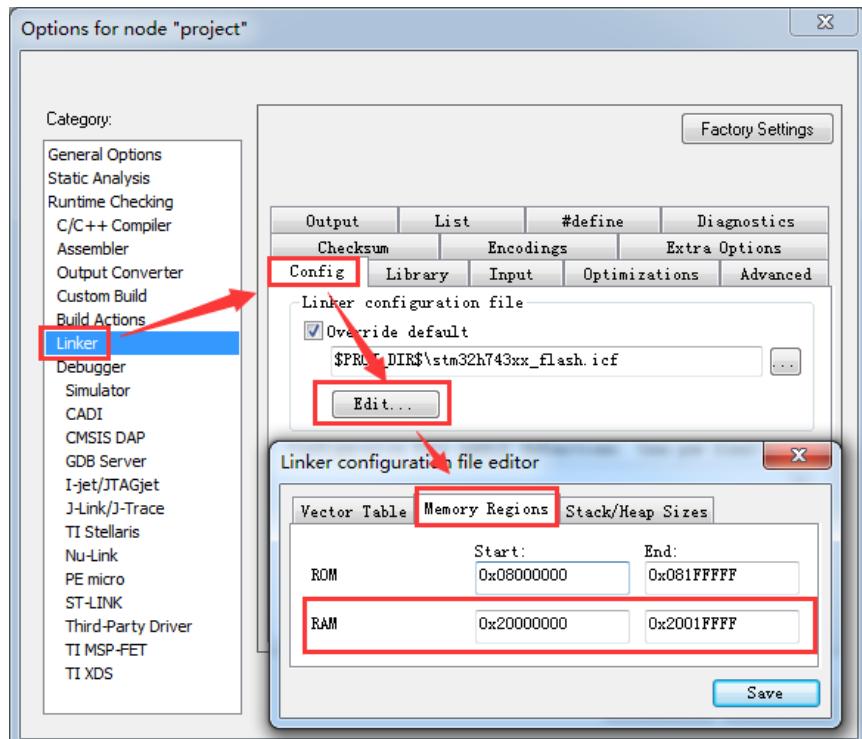


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
*****
```



```
/* 功能说明：初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形    参：无
* 返回值：无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
     STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：
     - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。
     - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
     配置系统时钟到 400MHz
     - 切换使用 HSE。
     - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。
    */
    SystemClock_Config();

    /*
     Event Recorder:
     - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。
     - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章
    */
#if Enable_EventRecorder == 1
    /* 初始化 EventRecorder 并开启 */
    EventRecorderInitialize(EventRecordAll, 1U);
    EventRecorderStart();
#endif

    bsp_InitKey();      /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */
    bsp_InitTimer();   /* 初始化滴答定时器 */
    bsp_InitUart();    /* 初始化串口 */
    bsp_InitExtIO();   /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */
    bsp_InitLed();     /* 初始化 LED */
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*
*****
* 函数名：MPU_Config
* 功能说明：配置 MPU
* 形    参：无
* 返回值：无
*****
*/
static void MPU_Config( void )
```



```
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec   = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER1;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec   = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /*使能 MPU */  
    HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);  
}  
  
/*  
*****  
* 函数名: CPU_CACHE_Enable  
* 功能说明: 使能 L1 Cache  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void CPU_CACHE_Enable(void)  
{  
    /* 使能 I-Cache */  
    SCB_EnableICache();  
  
    /* 使能 D-Cache */  
    SCB_EnableDCache();  
}
```



◆ 主功能：

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- 按下按键 K1，打印测试波形 1 和滤波后的波形数据。
- 按下按键 K2，打印测试波形 2 和滤波后的波形数据。
- 按下按键 K3，打印测试波形 3 和滤波后的波形数据。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****/
int main(void)
{
    uint8_t ucKeyCode;      /* 按键代码 */
    uint16_t i;

    bsp_Init();           /* 硬件初始化 */
    PrintfLogo(); /* 打印例程信息到串口 1 */

    PrintfHelp(); /* 打印操作提示信息 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 进入主程序循环体 */
    while (1)
    {
        bsp_Idle();          /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        if (bsp_CheckTimer(0)) /* 判断定时器超时时间 */
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2); /* 翻转 LED 的状态 */
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1:           /* K1 键按下 */
                    arm_lms_f32_test1();
                    break;

                case KEY_DOWN_K2:           /* K2 键按下 */
                    arm_lms_f32_test2();
                    break;

                case KEY_DOWN_K3:           /* K3 键按下 */
                    arm_lms_f32_test3();
                    break;
            }
        }
    }
}
```



```
        break;

    default:
        /* 其它的键值不处理 */
        break;
    }

}

}
```

49.8总结

本章节主要讲解了自适应滤波器，功能比较强劲，熟练应用需要多做测试。



第50章 STM32H7 的样条插补实现，波形拟合

丝滑顺畅

本章节讲解样条插补，主要用于波形拟合，平滑过渡。

50.1 初学者重要提示

50.2 样条插补介绍

50.3 样条插补实现

50.4 实验例程说明（MDK）

50.5 实验例程说明（IAR）

50.6 总结

50.1 初学者重要提示

- ◆ DSP 库支持了样条插补，双线性插补和线性插补，我们这里主要介绍样条插补的实现。

50.2 样条插补介绍

在数学学科数值分析中，样条是一种特殊的函数，由多项式分段定义。样条的英语单词 spline 来源于可变形的样条工具，那是一种在造船和工程制图时用来画出光滑形状的工具。在中国大陆，早期曾经被称做“齿函数”。后来因为工程学术语中“放样”一词而得名。在插值问题中，样条插值通常比多项式插值好用。用低阶的样条插值能产生和高阶的多项式插值类似的效果，并且可以避免被称为龙格现象的数值不稳定的出现。并且低阶的样条插值还具有“保凸”的重要性质。在计算机科学的计算机辅助设计和计算机图形学中，样条通常是指分段定义的多项式参数曲线。由于样条构造简单，使用方便，拟合准确，并能近似曲线拟合和交互式曲线设计中复杂的形状，样条是这些领域中曲线的常用表示方法。

50.3 样条插补实现

样条插补主要通过下面两个函数实现。

50.3.1 函数 arm_spline_init_f32

函数原型：

```
void arm_spline_init_f32(
```



```
arm_spline_instance_f32 * S,
arm_spline_type type,
const float32_t * x,
const float32_t * y,
uint32_t n,
float32_t * coeffs,
float32_t * tempBuffer)
```

函数描述：

此函数用于样条函数初始化。

函数参数：

- ◆ 第 1 个参数是 arm_spline_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是样条类型选择：
 - ARM_SPLINE_NATURAL 表自然样条。
 - ARM_SPLINE_PARABOLIC_RUNOUT 表示抛物线样条。
- ◆ 第 3 个参数是原始数据 x 轴坐标值。
- ◆ 第 4 个参数是原始数据 y 轴坐标值。
- ◆ 第 5 个参数是原始数据个数。
- ◆ 第 6 个参数是插补因数缓存。
- ◆ 第 7 个参数是临时缓冲。

注意事项：

- ◆ x 轴坐标数据必须是递增方式。
- ◆ 第 6 个参数插补因数缓存大小问题，如果原始数据个数是 n，那么插补因数个数必须要大于等于 $3*(n-1)$ 。
- ◆ 第 7 个参数临时缓冲大小问题，如果原始数据个数是 n，那么临时缓冲大小必须大于等于 $2*n - 1$

50.3.2 函数 arm_spline_f32

函数原型：

```
void arm_spline_f32(
    arm_spline_instance_f32 * S,
    const float32_t * xq,
    float32_t * pDst,
    uint32_t blockSize)
```

函数描述：

此函数用于样条插补实现。

函数参数：

- ◆ 第 1 个参数是 arm_spline_instance_f32 类型结构体变量。
- ◆ 第 2 个参数是插补后的 x 轴坐标值，需要用户指定，注意坐标值一定是递增的。
- ◆ 第 3 个参数是经过插补计算后输出的 y 轴数值
- ◆ 第 4 个参数是数据输出个数



50.3.3 使用样条插补函数的关键点

样条插补的主要作用是使得波形更加平滑。比如一帧是128点，步大小是8个像素，我们可以通过插补实现步长为1，1024点的波形，本质是你的总步长大小不能变，我们这里都是1024，这个不能变，在这个基础上做插补，效果就出来了。

这个认识非常重要，否则无法正常使用插补算法。

50.3.4 自然样条插补测试

样条测试代码的实现如下：

```
#define INPUT_TEST_LENGTH_SAMPLES    128 /* 输入数据个数 */
#define OUT_TEST_LENGTH_SAMPLES      1024 /* 输出数据个数 */

#define SpineTab OUT_TEST_LENGTH_SAMPLES/INPUT_TEST_LENGTH_SAMPLES /* 插补末尾的 8 个坐标值不使用 */

float32_t xn[INPUT_TEST_LENGTH_SAMPLES]; /* 输入数据 x 轴坐标 */
float32_t yn[INPUT_TEST_LENGTH_SAMPLES]; /* 输入数据 y 轴坐标 */

float32_t coeffs[3*(INPUT_TEST_LENGTH_SAMPLES - 1)]; /* 插补系数缓冲 */
float32_t tempBuffer[2 * INPUT_TEST_LENGTH_SAMPLES - 1]; /* 插补临时缓冲 */

float32_t xpos[OUT_TEST_LENGTH_SAMPLES]; /* 插补计算后 X 轴坐标值 */
float32_t ypos[OUT_TEST_LENGTH_SAMPLES]; /* 插补计算后 Y 轴数值 */

/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint32_t i;
    uint32_t idx2;
    uint8_t ucKeyCode;
    arm_spline_instance_f32 S;

    bsp_Init(); /* 硬件初始化 */
    PrintfLogo(); /* 打印例程名称和版本等信息 */
    PrintfHelp(); /* 打印操作提示 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 原始 x 轴数值和 y 轴数值 */
    for(i=0; i<INPUT_TEST_LENGTH_SAMPLES; i++)
    {
```



```
xn[i] = i*SpineTab;
yn[i] = 1 + cos(2*3.1415926*50*i/256 + 3.1415926/3);
}

/* 插补后 X 轴坐标值, 这个是需要用户设置的 */
for(i=0; i<OUT_TEST_LENGTH_SAMPLES; i++)
{
    xpos[i] = i;
}

while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1: /* K1 键按下, 自然样条插补 */
                /* 样条初始化 */
                arm_spline_init_f32(&S,
                                    ARM_SPLINE_NATURAL ,
                                    xn,
                                    yn,
                                    INPUT_TEST_LENGTH_SAMPLES,
                                    coeffs,
                                    tempBuffer);

                /* 样条计算 */
                arm_spline_f32 (&S,
                                xpos,
                                ypos,
                                OUT_TEST_LENGTH_SAMPLES);

                /* 打印输出输出 */
                idx2 = 0;
                for (i = 0; i < OUT_TEST_LENGTH_SAMPLES-SpineTab; i++)
                {
                    if ((i % SpineTab) == 0)
                    {
                        printf("%f,%f\r\n", ypos[i], yn[idx2++]);
                    }
                    else
                    {
                        printf("%f,\r\n", ypos[i]);
                    }
                }
                break;

            default:
        }
    }
}
```

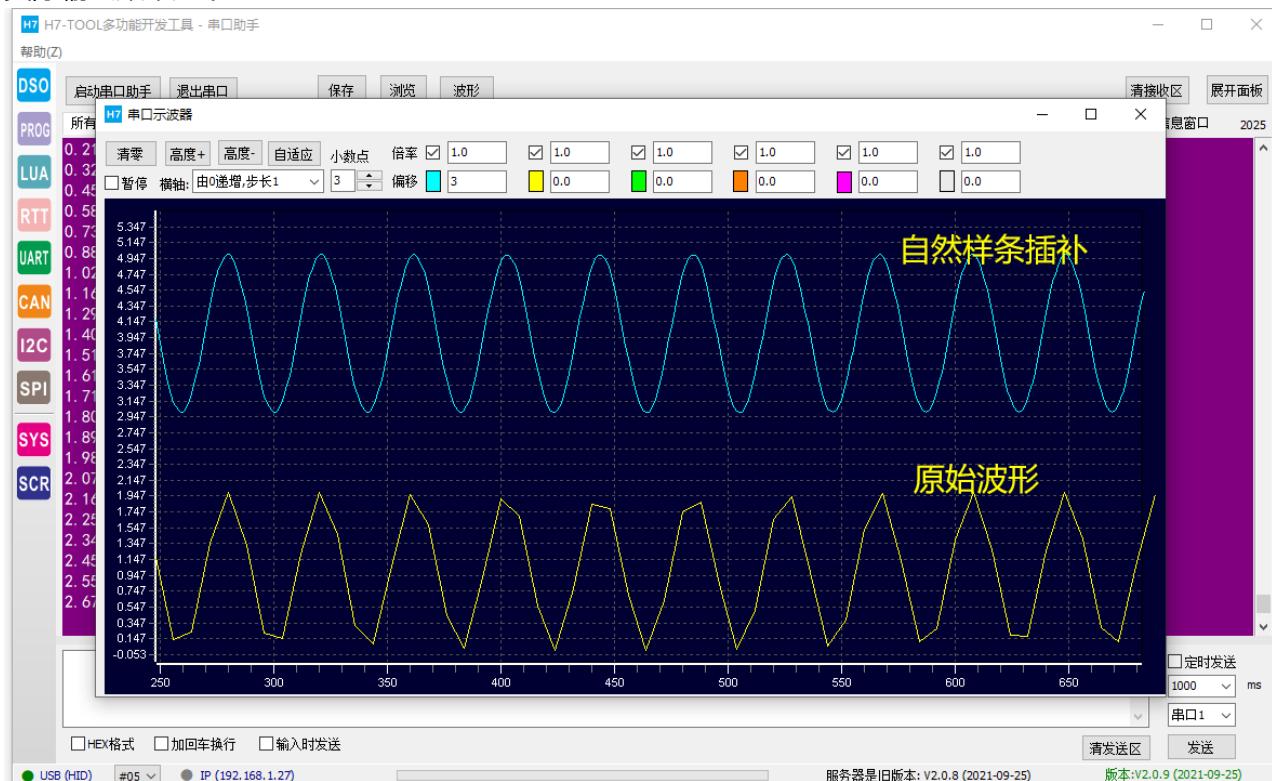


```
/* 其它的键值不处理 */
break;
}
}
}
```

代码里面的几个关键地方：

- ◆ 原始坐标数组xn和yn是128组，而我们通过插补生成的是1024组xnpos和ynpos，其中1024组的xnpos需要用户设置初值，这点不能忽略。
- ◆ 函数arm_spline_init_f32用于样条函数初始化，这里特别注意，此函数主要是对原始数据的操作。
自然样条插补用的ARM_SPLINE_NATURAL。
- ◆ 函数arm_spline_f32用于样条函数计算。

实际输出效果如下：



50.3.5 抛物线样条插补测试

样条测试代码的实现如下：

```
#define INPUT_TEST_LENGTH_SAMPLES      128 /* 输入数据个数 */
#define OUT_TEST_LENGTH_SAMPLES        1024 /* 输出数据个数 */

#define SpineTab OUT_TEST_LENGTH_SAMPLES/INPUT_TEST_LENGTH_SAMPLES /* 插补末尾的 8 个坐标值不使用 */

float32_t xn[INPUT_TEST_LENGTH_SAMPLES]; /* 输入数据 x 轴坐标 */
```



```
float32_t yn[INPUT_TEST_LENGTH_SAMPLES]; /* 输入数据 y 轴坐标 */

float32_t coeffs[3*(INPUT_TEST_LENGTH_SAMPLES - 1)]; /* 插补系数缓冲 */
float32_t tempBuffer[2 * INPUT_TEST_LENGTH_SAMPLES - 1]; /* 插补临时缓冲 */

float32_t xnpos[OUT_TEST_LENGTH_SAMPLES]; /* 插补计算后 X 轴坐标值 */
float32_t ynpos[OUT_TEST_LENGTH_SAMPLES]; /* 插补计算后 Y 轴数值 */

/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint32_t i;
    uint32_t idx2;
    uint8_t ucKeyCode;
    arm_spline_instance_f32 S;

    bsp_Init(); /* 硬件初始化 */
    PrintfLogo(); /* 打印例程名称和版本等信息 */
    PrintfHelp(); /* 打印操作提示 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 原始 x 轴数值和 y 轴数值 */
    for(i=0; i<INPUT_TEST_LENGTH_SAMPLES; i++)
    {
        xn[i] = i*SpineTab;
        yn[i] = 1 + cos(2*3.1415926*50*i/256 + 3.1415926/3);
    }

    /* 插补后 X 轴坐标值, 这个是需要用户设置的 */
    for(i=0; i<OUT_TEST_LENGTH_SAMPLES; i++)
    {
        xnpos[i] = i;
    }

    while (1)
    {
        bsp_Idle(); /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {

```



```
switch (ucKeyCode)
{
    case KEY_DOWN_K2:           /* K2 键按下，抛物线样条插补 */
        /* 样条初始化 */
        arm_spline_init_f32(&S,
                             ARM_SPLINE_PARABOLIC_RUNOUT ,
                             xn,
                             yn,
                             INPUT_TEST_LENGTH_SAMPLES,
                             coeffs,
                             tempBuffer);

        /* 样条计算 */
        arm_spline_f32    (&S,
                            &xnpos,
                            &ynpos,
                            OUT_TEST_LENGTH_SAMPLES);

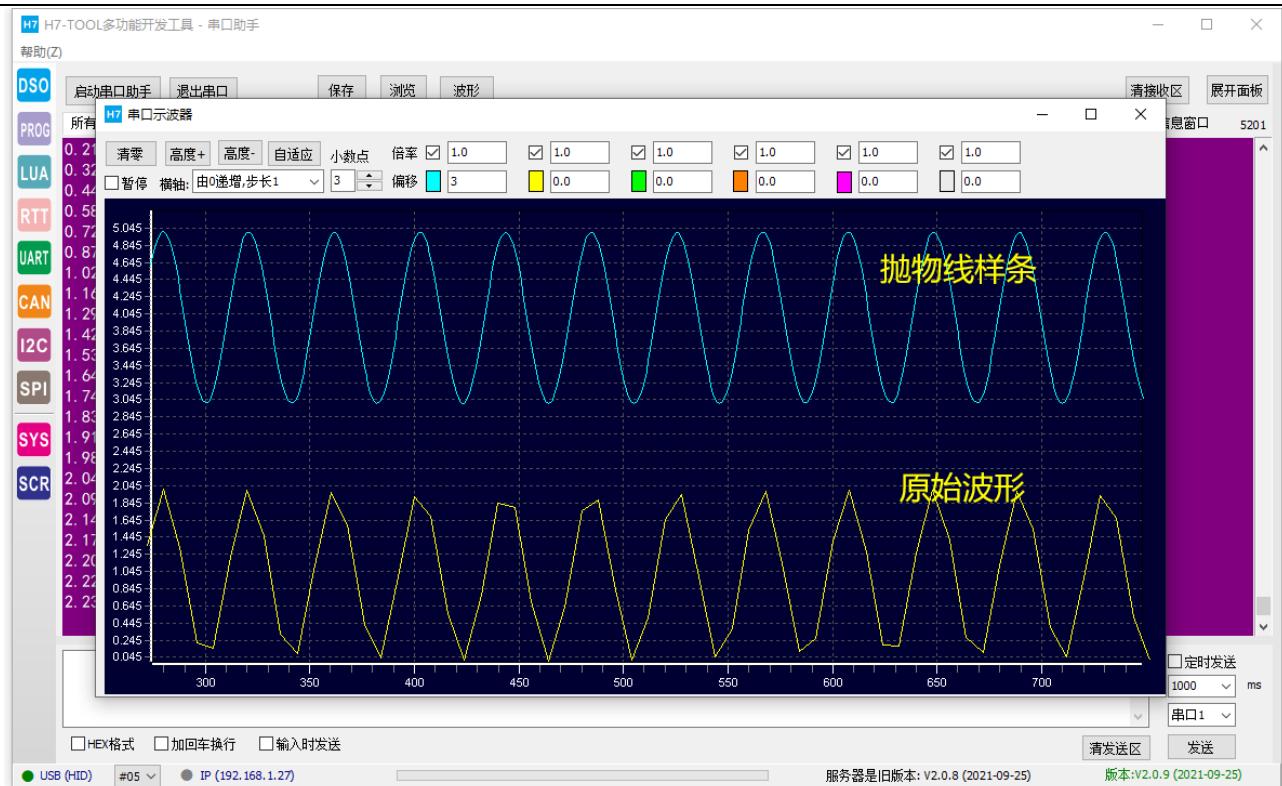
        /* 打印输出输出 */
        idx2 = 0;
        for (i = 0; i < OUT_TEST_LENGTH_SAMPLES-SpineTab; i++)
        {
            if ((i % SpineTab) == 0)
            {
                printf("%f,%f\r\n", ynpos[i], yn[idx2++]);
            }
            else
            {
                printf("%f,\r\n", ynpos[i]);
            }
        }
        break;

    default:
        /* 其它的键值不处理 */
        break;
    }
}
}
```

代码里面的几个关键地方：

- ◆ 原始坐标数组xn和yn是128组，而我们通过插补生成的是1024组xnpos和ynpos，其中1024组的xnpos需要用户设置初值，这点不能忽略。
- ◆ 函数arm_spline_init_f32用于样条函数初始化，这里特别注意，此函数主要是对原始数据的操作。
抛物线样条插补用的ARM_SPLINE_PARABOLIC_RUNOUT。
- ◆ 函数arm_spline_f32用于样条函数计算。

实际输出效果如下：



50.4 实验例程说明 (MDK)

配套例子：

V7-235_样条插补，波形拟合丝滑顺畅

实验目的：

1. 学习样条插补的实现。

实验内容：

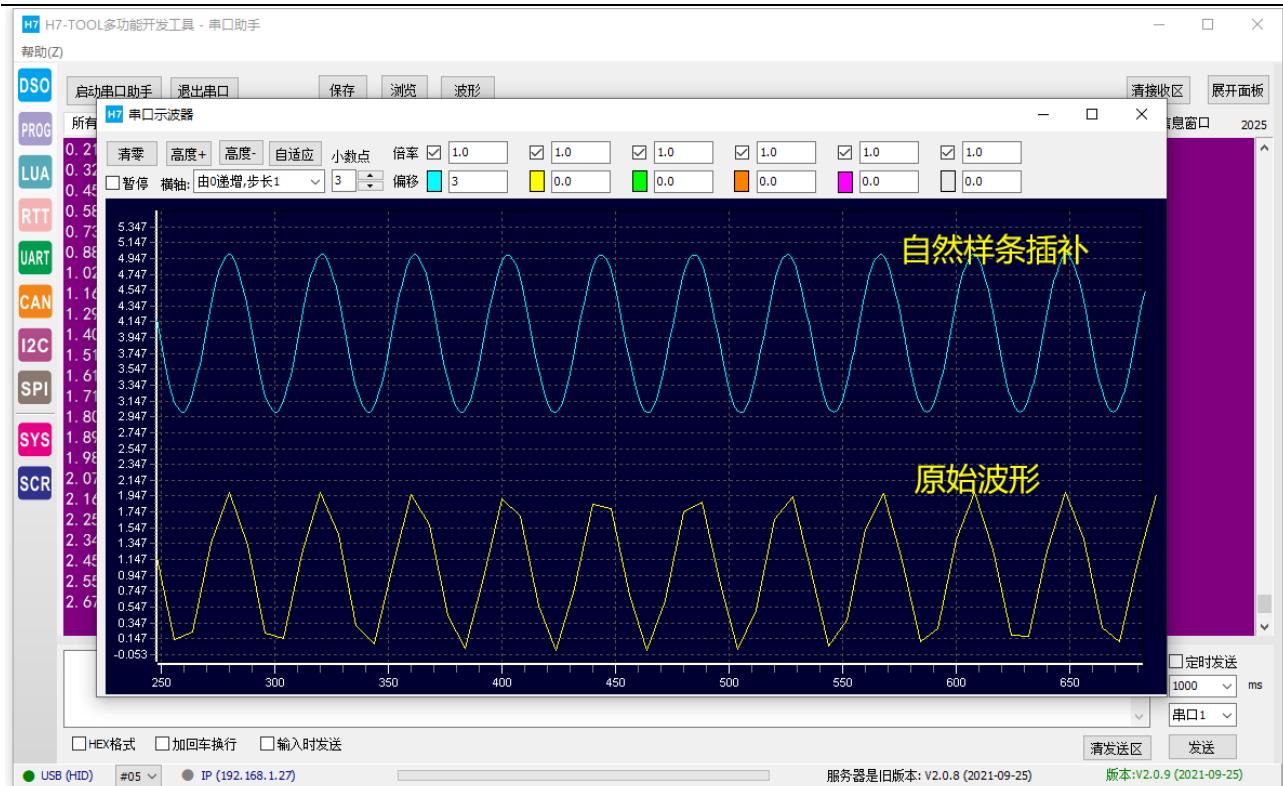
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. K1 键按下，自然样条插补测试。
3. K2 键按下，抛物线样插补测试。

使用 AC6 注意事项

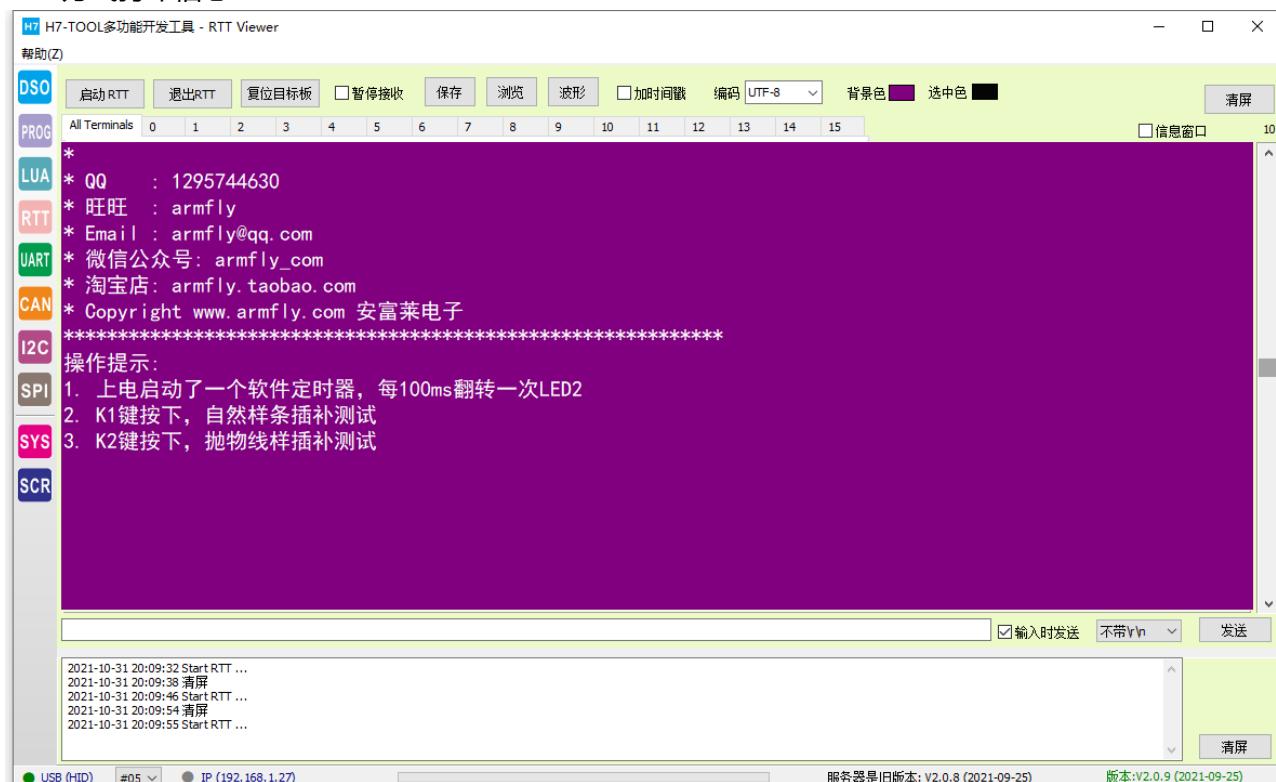
特别注意附件章节 C 的问题

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

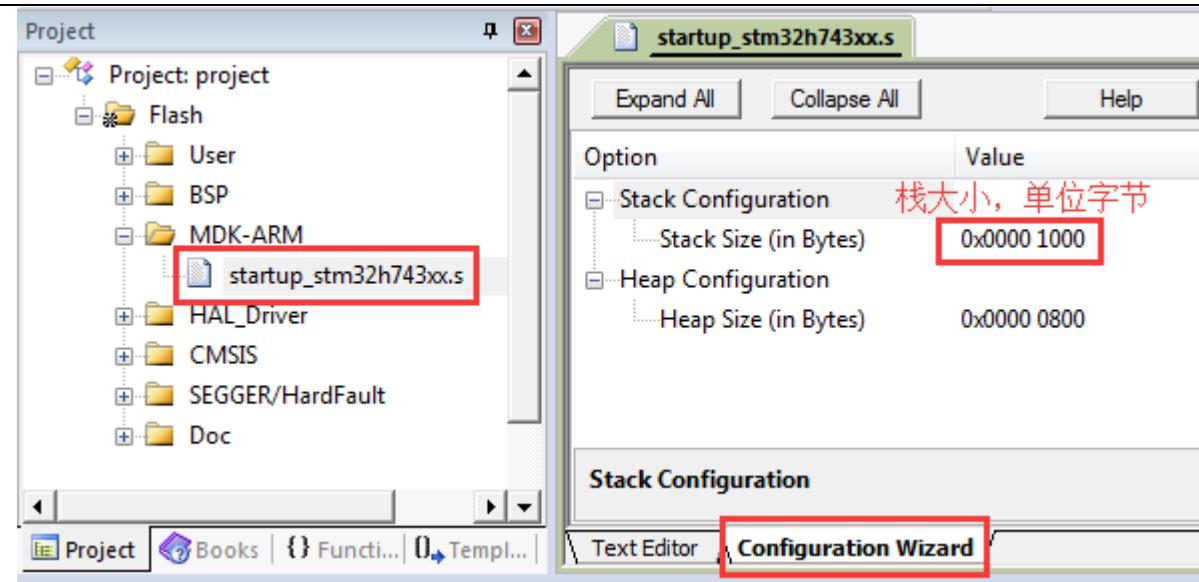


RTT 方式打印信息：

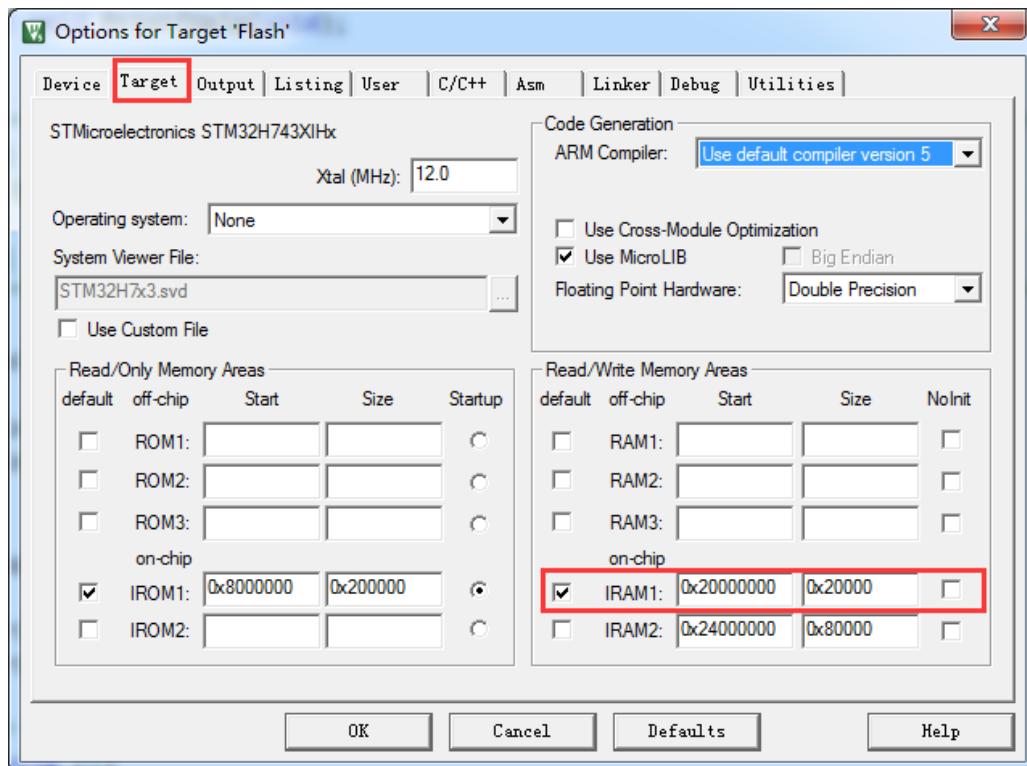


程序设计：

◆ 系统栈大小分配：



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
```



```
{  
    /* 配置 MPU */  
    MPU_Config();  
  
    /* 使能 L1 Cache */  
    CPU_CACHE_Enable();  
  
    /*  
     * STM32H7xx HAL 库初始化，此时系统用的还是 H7 自带的 64MHz，HSI 时钟：  
     * - 调用函数 HAL_InitTick，初始化滴答时钟中断 1ms。  
     * - 设置 NVIC 优先级分组为 4。  
     */  
    HAL_Init();  
  
    /*  
     * 配置系统时钟到 400MHz  
     * - 切换使用 HSE。  
     * - 此函数会更新全局变量 SystemCoreClock，并重新配置 HAL_InitTick。  
     */  
    SystemClock_Config();  
  
    /*  
     * Event Recorder:  
     * - 可用于代码执行时间测量，MDK5.25 及其以上版本才支持，IAR 不支持。  
     * - 默认不开启，如果要使能此选项，务必看 V7 开发板用户手册第 8 章  
     */  
#if Enable_EventRecorder == 1  
    /* 初始化 EventRecorder 并开启 */  
    EventRecorderInitialize(EventRecordAll, 1U);  
    EventRecorderStart();  
#endif  
  
    bsp_InitKey(); /* 按键初始化，要放在滴答定时器之前，因为按钮检测是通过滴答定时器扫描 */  
    bsp_InitTimer(); /* 初始化滴答定时器 */  
    bsp_InitUart(); /* 初始化串口 */  
    bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO。必须在 bsp_InitLed() 前执行 */  
    bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置：

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区（本例子未用到 AXI SRAM），FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();
```



```
/* 配置 AXI SRAM 的 MPU 属性为关闭读 Cache 和写 Cache */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x24000000;
MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */
MPU_InitStruct.Enable      = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x60000000;
MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;
MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;
MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number      = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。



- K1 键按下，自然样条插补测试。
- K2 键按下，抛物线样插补测试。

```
/*
***** 函数名: main
***** 功能说明: c 程序入口
***** 形参: 无
***** 返回值: 错误代码(无需处理)
*****
int main(void)
{
    uint32_t i;
    uint32_t idx2;
    uint8_t ucKeyCode;
    arm_spline_instance_f32 S;

    bsp_Init(); /* 硬件初始化 */
    PrintfLogo(); /* 打印例程名称和版本等信息 */
    PrintfHelp(); /* 打印操作提示 */

    bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */

    /* 原始 x 轴数值和 y 轴数值 */
    for(i=0; i<INPUT_TEST_LENGTH_SAMPLES; i++)
    {
        xn[i] = i*SpineTab;
        yn[i] = 1 + cos(2*3.1415926*50*i/256 + 3.1415926/3);
    }

    /* 插补后 X 轴坐标值, 这个是需要用户设置的 */
    for(i=0; i<OUT_TEST_LENGTH_SAMPLES; i++)
    {
        xpos[i] = i;
    }

    while (1)
    {
        bsp_Idle(); /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

        /* 判断定时器超时时间 */
        if (bsp_CheckTimer(0))
        {
            /* 每隔 100ms 进来一次 */
            bsp_LedToggle(2);
        }

        ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
        if (ucKeyCode != KEY_NONE)
        {
            switch (ucKeyCode)
            {
                case KEY_DOWN_K1: /* K1 键按下, 自然样条插补 */
                    /* 样条初始化 */
            }
        }
    }
}
```



```
arm_spline_init_f32(&S,
                     ARM_SPLINE_NATURAL ,
                     xn,
                     yn,
                     INPUT_TEST_LENGTH_SAMPLES,
                     coeffs,
                     tempBuffer);

/* 样条计算 */
arm_spline_f32    (&S,
                    xnpos,
                    ynpos,
                    OUT_TEST_LENGTH_SAMPLES);

/* 打印输出输出 */
idx2 = 0;
for (i = 0; i < OUT_TEST_LENGTH_SAMPLES-SpineTab; i++)
{
    if ((i % SpineTab) == 0)
    {
        printf("%f,%f\r\n", ynpos[i], yn[idx2++]);
    }
    else
    {
        printf("%f,\r\n", ynpos[i]);
    }
}
break;

case KEY_DOWN_K2:           /* K2 键按下，抛物线样条插补 */
/* 样条初始化 */
arm_spline_init_f32(&S,
                     ARM_SPLINE_PARABOLIC_RUNOUT ,
                     xn,
                     yn,
                     INPUT_TEST_LENGTH_SAMPLES,
                     coeffs,
                     tempBuffer);

/* 样条计算 */
arm_spline_f32    (&S,
                    xnpos,
                    ynpos,
                    OUT_TEST_LENGTH_SAMPLES);

/* 打印输出输出 */
idx2 = 0;
for (i = 0; i < OUT_TEST_LENGTH_SAMPLES-SpineTab; i++)
{
    if ((i % SpineTab) == 0)
    {
        printf("%f,%f\r\n", ynpos[i], yn[idx2++]);
    }
    else
    {
        printf("%f,\r\n", ynpos[i]);
    }
}
```



```
        break;

    default:
        /* 其它的键值不处理 */
        break;
    }
}
}
```

50.5 实验例程说明 (IAR)

配套例子：

V7-235_样条插补，波形拟合丝滑顺畅

实验目的：

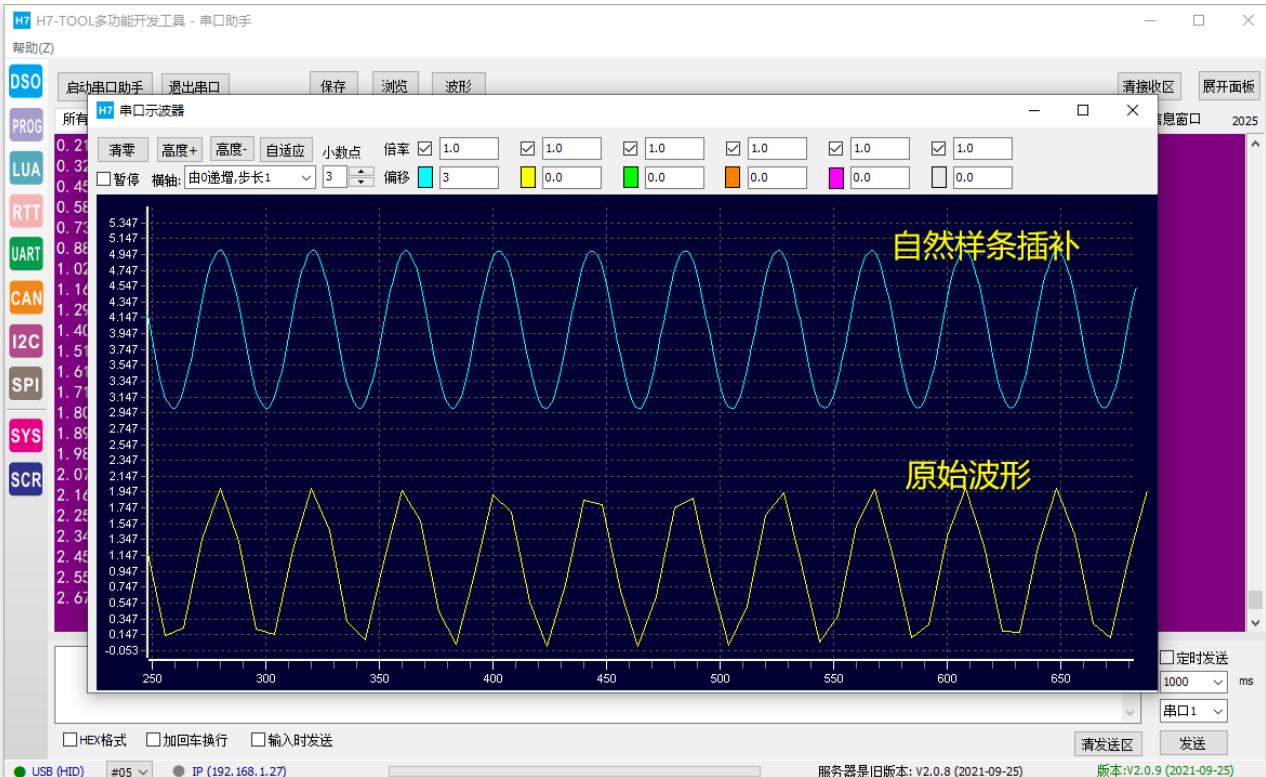
1. 学习样条插补的实现。

实验内容：

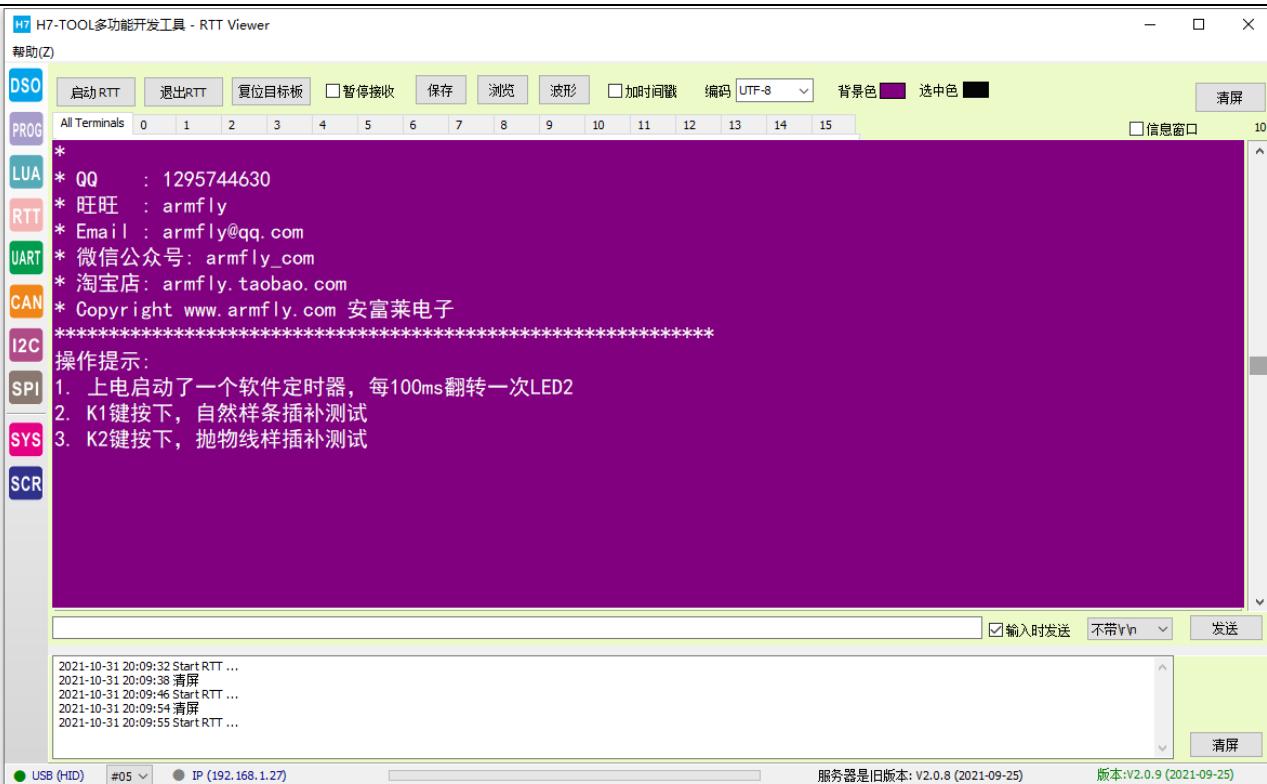
1. 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
2. K1 键按下，自然样条插补测试。
3. K2 键按下，抛物线样插补测试。

上电后串口打印的信息：

波特率 115200，数据位 8，奇偶校验位无，停止位 1。

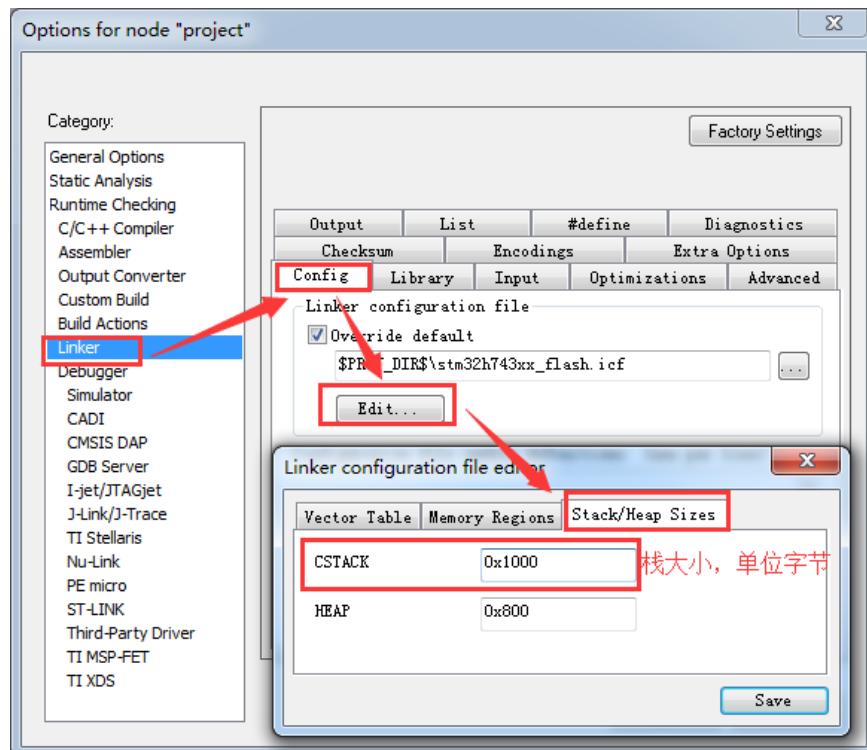


RTT 方式打印信息：

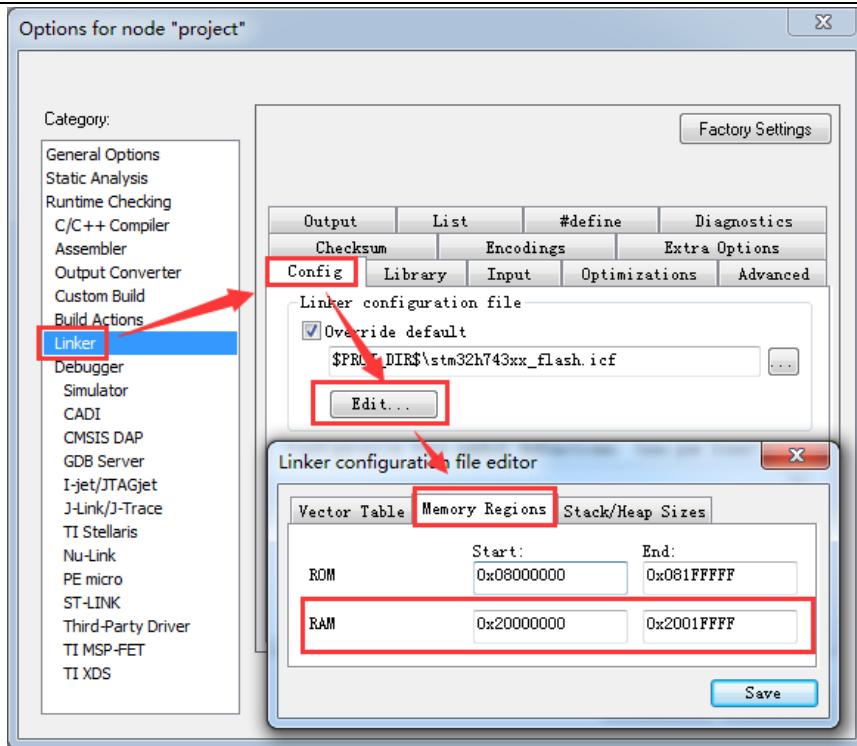


程序设计:

◆ 系统栈大小分配:



◆ RAM 空间用的 DTCM:



◆ 硬件外设初始化

硬件外设的初始化是在 bsp.c 文件实现：

```
/*
*****
* 函数名: bsp_Init
* 功能说明: 初始化所有的硬件设备。该函数配置 CPU 寄存器和外设的寄存器并初始化一些全局变量。只需要调用一次
* 形参: 无
* 返回值: 无
*****
*/
void bsp_Init(void)
{
    /* 配置 MPU */
    MPU_Config();

    /* 使能 L1 Cache */
    CPU_CACHE_Enable();

    /*
    STM32H7xx HAL 库初始化, 此时系统用的还是 H7 自带的 64MHz, HSI 时钟:
    - 调用函数 HAL_InitTick, 初始化滴答时钟中断 1ms。
    - 设置 NVIC 优先级分组为 4。
    */
    HAL_Init();

    /*
    配置系统时钟到 400MHz
    - 切换使用 HSE。
    - 此函数会更新全局变量 SystemCoreClock, 并重新配置 HAL_InitTick。
    */
    SystemClock_Config();
}
```



```
Event Recorder:  
- 可用于代码执行时间测量, MDK5.25 及其以上版本才支持, IAR 不支持。  
- 默认不开启, 如果要使能此选项, 务必看 V7 开发板用户手册第 8 章  
*/  
#if Enable_EventRecorder == 1  
/* 初始化 EventRecorder 并开启 */  
EventRecorderInitialize(EventRecordAll, 1U);  
EventRecorderStart();  
#endif  
  
bsp_InitKey(); /* 按键初始化, 要放在滴答定时器之前, 因为按钮检测是通过滴答定时器扫描 */  
bsp_InitTimer(); /* 初始化滴答定时器 */  
bsp_InitUart(); /* 初始化串口 */  
bsp_InitExtIO(); /* 初始化 FMC 总线 74HC574 扩展 IO. 必须在 bsp_InitLed() 前执行 */  
bsp_InitLed(); /* 初始化 LED */  
}
```

◆ MPU 配置和 Cache 配置:

数据 Cache 和指令 Cache 都开启。配置了 AXI SRAM 区 (本例子未用到 AXI SRAM) , FMC 的扩展 IO 区。

```
/*  
*****  
* 函数名: MPU_Config  
* 功能说明: 配置 MPU  
* 形参: 无  
* 返回值: 无  
*****  
*/  
static void MPU_Config( void )  
{  
    MPU_Region_InitTypeDef MPU_InitStruct;  
  
    /* 禁止 MPU */  
    HAL_MPU_Disable();  
  
    /* 配置 AXI SRAM 的 MPU 属性为 Write back, Read allocate, Write allocate */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x24000000;  
    MPU_InitStruct.Size        = MPU_REGION_SIZE_512KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;  
    MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;  
    MPU_InitStruct.IsShareable = MPU_ACCESS_NOT_SHAREABLE;  
    MPU_InitStruct.Number      = MPU_REGION_NUMBER0;  
    MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL1;  
    MPU_InitStruct.SubRegionDisable = 0x00;  
    MPU_InitStruct.DisableExec   = MPU_INSTRUCTION_ACCESS_ENABLE;  
  
    HAL_MPU_ConfigRegion(&MPU_InitStruct);  
  
    /* 配置 FMC 扩展 IO 的 MPU 属性为 Device 或者 Strongly Ordered */  
    MPU_InitStruct.Enable      = MPU_REGION_ENABLE;  
    MPU_InitStruct.BaseAddress = 0x60000000;  
    MPU_InitStruct.Size        = ARM_MPU_REGION_SIZE_64KB;  
    MPU_InitStruct.AccessPermission = MPU_REGION_FULL_ACCESS;  
    MPU_InitStruct.IsBufferable = MPU_ACCESS_BUFFERABLE;
```



```
MPU_InitStruct.IsCacheable      = MPU_ACCESS_NOT_CACHEABLE;
MPU_InitStruct.IsShareable       = MPU_ACCESS_NOT_SHAREABLE;
MPU_InitStruct.Number            = MPU_REGION_NUMBER1;
MPU_InitStruct.TypeExtField     = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec      = MPU_INSTRUCTION_ACCESS_ENABLE;

HAL_MPU_ConfigRegion(&MPU_InitStruct);

/*使能 MPU */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);
}

/*
*****
* 函数名: CPU_CACHE_Enable
* 功能说明: 使能 L1 Cache
* 形参: 无
* 返回值: 无
*****
*/
static void CPU_CACHE_Enable(void)
{
    /* 使能 I-Cache */
    SCB_EnableICache();

    /* 使能 D-Cache */
    SCB_EnableDCache();
}
```

◆ 主功能:

主程序实现如下操作：

- 启动一个自动重装软件定时器，每 100ms 翻转一次 LED2。
- K1 键按下，自然样条插补测试。
- K2 键按下，抛物线样条插补测试。

```
/*
*****
* 函数名: main
* 功能说明: c 程序入口
* 形参: 无
* 返回值: 错误代码(无需处理)
*****
*/
int main(void)
{
    uint32_t i;
    uint32_t idx2;
    uint8_t uckKeyCode;
    arm_spline_instance_f32 S;

    bsp_Init(); /* 硬件初始化 */
    PrintfLogo(); /* 打印例程名称和版本等信息 */
    PrintfHelp(); /* 打印操作提示 */
}
```



```
bsp_StartAutoTimer(0, 100); /* 启动 1 个 100ms 的自动重装的定时器 */
```

```
/* 原始 x 轴数值和 y 轴数值 */
for(i=0; i<INPUT_TEST_LENGTH_SAMPLES; i++)
{
    xn[i] = i*SpineTab;
    yn[i] = 1 + cos(2*3.1415926*50*i/256 + 3.1415926/3);
}

/* 插补后 X 轴坐标值, 这个是需要用户设置的 */
for(i=0; i<OUT_TEST_LENGTH_SAMPLES; i++)
{
    xpos[i] = i;
}

while (1)
{
    bsp_Idle();           /* 这个函数在 bsp.c 文件。用户可以修改这个函数实现 CPU 休眠和喂狗 */

    /* 判断定时器超时时间 */
    if (bsp_CheckTimer(0))
    {
        /* 每隔 100ms 进来一次 */
        bsp_LedToggle(2);
    }

    ucKeyCode = bsp_GetKey(); /* 读取键值, 无键按下时返回 KEY_NONE = 0 */
    if (ucKeyCode != KEY_NONE)
    {
        switch (ucKeyCode)
        {
            case KEY_DOWN_K1: /* K1 键按下, 自然样条插补 */
                /* 样条初始化 */
                arm_spline_init_f32(&S,
                    ARM_SPLINE_NATURAL ,
                    xn,
                    yn,
                    INPUT_TEST_LENGTH_SAMPLES,
                    coeffs,
                    tempBuffer);

                /* 样条计算 */
                arm_spline_f32 (&S,
                    &xpos,
                    &ypos,
                    OUT_TEST_LENGTH_SAMPLES);
            }

            /* 打印输出输出 */
            idx2 = 0;
            for (i = 0; i < OUT_TEST_LENGTH_SAMPLES-SpineTab; i++)
            {
                if ((i % SpineTab) == 0)
                {
                    printf("%f,%f\r\n", ypos[i], yn[idx2++]);
                }
                else
                {

```



```
        printf("%f, \r\n", ynpos[i]);
    }
}

break;

case KEY_DOWN_K2:           /* K2 键按下，抛物线样条插补 */
/* 样条初始化 */
arm_spline_init_f32(&S,
                     ARM_SPLINE_PARABOLIC_RUNOUT ,
                     xn,
                     yn,
                     INPUT_TEST_LENGTH_SAMPLES,
                     coeffs,
                     tempBuffer);

/* 样条计算 */
arm_spline_f32 (&S,
                 xnpos,
                 ynpos,
                 OUT_TEST_LENGTH_SAMPLES);

/* 打印输出输出 */
idx2 = 0;
for (i = 0; i < OUT_TEST_LENGTH_SAMPLES-SpineTab; i++)
{
    if ((i % SpineTab) == 0)
    {
        printf("%f,%f\r\n", ynpos[i], yn[idx2++]);
    }
    else
    {
        printf("%f, \r\n", ynpos[i]);
    }
}
break;

default:
/* 其它的键值不处理 */
break;
}
}
}
```

50.6 总结

本章节主要讲解了样条插补的实现，实际项目比较实用，有兴趣可以深入源码了解。



第51章 附件 A---Cortex-M7, A8, A9, A15

与 ADI 的 BlackFin 以及 SHARC 的 DSP 性能

PK

说明：

- 1、通过此贴让我们对 M4 和 M7 的 DSP 性能有个全面的认识。
- 2、测试数据来源于 DSP Concepts，对于这家公司的名字，大家可能比较陌生。我们现在用的 CMSIS-DSP 软件就是由 ARM 委托这家公司设计的，是一家比较厉害的嵌入式音频 DSP 解决方案开发商。
- 3、硬件测试平台：

M4 使用一款 204MHz 的芯片（估计是 LPC43XX 系列）。

M7 使用 Atmel 的 amV71。

A8 使用 TI 的 AM335x

A9 使用 TI 的 OMAP4430

A15 使用 TI 的 OMAP5432

DSP 芯片：

Blackfin 53x，支持 16 位定点的 DSP

Blackfin 70x，支持 16 位和 32 位定点的 DSP

SHARC 21489，支持 32 位定点和 32/40 位浮点的 DSP

M 核的 DSP 处理单元与专业 DSP 的区别：

	Cortex-M3	Cortex-M4	Cortex-M7	True DSP
Single cycle MAC		Fixed-point only	Fixed and floating-point	Y
Floating-point		Y	Y	Y
Fractional and saturating math		Y	Y	Y
SIMD operations		Y	Y	Y
Load and store in parallel with math			Y (1)	Y (2)
Zero overhead loops			Y	Y
Accumulator with guard bits				Y
Circular and bit-reversed addressing				Y

一、FIR 滤波器测试



- 1、分别测试了 5, 10, 20, 50 和 100 阶 FIR 滤波，采样点数 256 个。
- 2、测试结果的单位是时钟周期个数。
- 3、全部采用浮点测试，而 BlackFin 采用的定点 Q31，测试软件使用 DSP Concepts 的 Audio Weaver。

Num Taps	Cortex-M4	Cortex-M7	Cortex-A8	Cortex-A9	Cortex-A15	Blackfin 53x	Blackfin 70x	SHARC 21489
5	6743	4282	6467	6315	2673			1954
10	9871	6268	9793	9245	5142			2473
20	15650	9938	13598	14338	5031			3777
50	35801	22734	29310	32799	10267	27404	14456	7677
100	67833	43074	53913	62145	15525			14210

Cycles Per Sample Per Tap, 每阶每个采样点的时钟周期个数。

Cortex-M4	Cortex-M7	Cortex-A8	Cortex-A9	Cortex-A15	Blackfin 53x	Blackfin 70x	SHARC 21489
2.65	1.68	2.11	2.43	0.61	2.14	1.13	0.56

结论：

SHARC21489 性能最强，M7 的 FIR 性能高于 A8 和 A9，低于 A15。

二、IIR 滤波器 (Biquad 级联)

- 1、分别测试了 1 个，4 个，8 个和 12 个 IIR 的 Biquad 级联个数，采样点数 256 个。
- 2、测试结果的单位是时钟周期个数。
- 3、测试软件使用 DSP Concepts 的 Audio Weaver

Num Stages	Cortex-M4	Cortex-M7	Cortex-A8	Cortex-A9	Cortex-A15	Blackfin BF53x	Blackfin BF70x	SHARC 21489
1	4480	2912	4867	4326	2439	4650	3338	1455
4	16700	10855	16712	17750	9040			5405
8	32900	21385	33354	32933	17825			10650
12	49100	31915	49274	50243	26664			15958

Cycles Per Sample Per Stage, 每个 Biquad 每个采样点的时钟周期个数。



Cortex-M4	Cortex-M7	Cortex-A8	Cortex-A9	Cortex-A15	Blackfin BF53x	Blackfin BF70x	SHARC 21489
15.98	10.39	16.04	16.36	8.68	18.16	13.04	5.19

结论：

SHARC21489 性能最强，M7 的 IIR 性能高于 A8 和 A9，低于 A15。

三、FFT 测试：

1、分别测试了 64 点, 128 点, 256 点, 512 点和 1024 点 FFT。

2、测试结果的单位是时钟周期个数。

Length	Cortex-M4	Cortex-M7	Cortex-A8	Cortex-A9	Cortex-A15	Blackfin BF53x	Blackfin BF70x	SHARC 21489
64	3709	2297	3773	3358	2264	2200	1526	783
128	9811	6018	6384	5682	3830	5249	3431	1334
256	21575	13881	11114	9891	6668	11744	7611	2542
512	37813	25551	21852	19448	13111	27385	17084	5189
1024	96630	63484	50738	45157	30443	60216	37568	10972

结论：

SHARC21489 性能最强，M7 的 FFT 低于 A8, A9 和 A15。

四、上面测试都是基于时钟周期的，考虑到主频的影响，结论如下

Cortex-M4: 204 MHz

Cortex-M7: 300 MHz

Cortex-A8: 1 GHz

Cortex-A9: 1 GHz

Cortex-A15: 1.5 GHz

Blackfin 53x: 700 MHz

Blackfin BF70x: 400 MHz

SHARC: 450 MHz



	Cortex-M4	Cortex-M7	Cortex-A8	Cortex-A9	Cortex-A15	Blackfin 53x	Blackfin 70x	SHARC 21489
FIR	0.09	0.22	0.59	0.51	3.05	0.40	0.44	1.00
Biquad	0.07	0.19	0.41	0.62	1.46	0.23	0.18	1.00
FFT	0.05	0.11	0.48	0.54	1.20	0.28	0.26	1.00

考虑测试的 M7 是用的 300MHz 的 V71，如果换成现在的 H7，主频可以搞到 480，那么 FIR 和 IIR 性能几乎媲美 1GHz 的 A8。

五、更详细的测试：

Audio Module	Cortex M4F	Cortex-M7	Cortex A8	Cortex A9	Cortex A15	SHARC 21489	Description
Abs	1810	1030	1342	929	426	327	Absolute value
Adder	1790	990	1639	1167	418	502	Adder
AGCAttackRelease	4840	6403	28196	5759	7727	2173	Attack / release envelop follower
AGCLimiterCore	64200	58614	37987	20919	15727	6071	Peak limiter
AGCMultiplier	2840	1515	1573	1138	834	866	Multiplier used with limiter
Biquad	3810	3045	4876	5452	2452	1458	Single biquad (non cascade) Single channel
Biquad	9010	6157	4549	5329	2269	1501	Single biquad (non cascade) Stereo
BiquadCascade	4480	3821	4867	4326	2439	1455	Biquad (1 stage) Single channel
BiquadCascade	16700	14433	16712	17750	9040	5405	Biquad (4 stages) Single channel
BiquadCascade	32900	28592	33354	32933	17825	10650	Biquad (8 stages) Single channel
BiquadCascade	49100	42745	49274	50243	26664	15958	Biquad (12 stages) Single channel
BiquadCascadeDelay			15113	9499	7149		Biquad (4 stages) Single channel. Optimized for NEON. Cortex-A only
BiquadCascadeDelay			29409	19435	13854		Biquad (8 stages) Single channel. Optimized for NEON. Cortex-A only
BiquadCascade	10700	9085	8896	8439	4833	1486	Biquad (1 stage) Stereo
BiquadCascade	40500	34760	32299	35808	17878	5511	Biquad (4 stages) Stereo
BiquadCascade	80800	68979	65239	70521	35540	10829	Biquad (8 stages) Stereo
BiquadCascade	120000	103191	97872	102630	53092	16203	Biquad (12 stages) Stereo
BiquadCascadeHP	7400	4306	29394	4900	4964	1763	High precision biquad (1 stage) Single channel
BiquadCascadeHP	29500	16169	122194	18897	19765	6870	High precision biquad (4 stages) Single channel
BiquadCascadeHP	58200	31862	242732	39145	39300	13755	High precision biquad (8 stages) Single channel
BiquadCascadeHP	87200	47543	364257	58063	58963	20478	High precision biquad (12 stages) Single channel
BiquadCascadeHP	14800	8220	29704	10251	10270	1784	High precision biquad (1 stage) Stereo
BiquadCascadeHP	58700	31789	119663	39159	39420	6965	High precision biquad (4 stages) Stereo
BiquadCascadeHP	117000	63069	238114	78845	78774	13943	High precision biquad (8 stages) Stereo
BiquadCascadeHP	174000	94312	359908	119134	118435	20880	High precision biquad (12 stages) Stereo
BlockStatistics	3480	3528	10547	1667	2904	878	Computes statistics across an entire block
ButterworthFilter	4500	3837	4753	5085	2471	1470	2nd order Butterworth filter
ButterworthFilter	16700	14533	16985	16667	9136	5426	10th order Butterworth filter



Audio Module	Cortex M4F	Cortex-M7	Cortex A8	Cortex A9	Cortex A15	Sharc 21489	Description
ClipAsym	3640	6133	1417	992	335	352	Asymmetric clip
Copier	1330	815	1288	794	301	353	Simple copier
Db20Approx	15300	12620	5535	4503	1437	10477	Approximation to $20^{\log_{10}(x)}$
Deinterleave	2640	1349	2600	1452	1682	1180	Deterinterleave stereo data to mono
Delay	3190	1553	4267	3491	2834	8042	Sample based time delay. External memory.
FIRDecimator	21000	12829	154318	29085	21837	4911	Polyphase decimator
FIRInterpolator	36700	22762	260068	76009	65943	16560	Polyphase interpolator
FIR	8080	4828	6467	6315	2673	1904	FIR (5-pt)
FIR	12900	8261	9793	9245	5142	2423	FIR (10-pt)
FIR	20700	13517	13598	14338	5031	3727	FIR (20-pt)
FIR	47600	30881	29310	32799	10267	7627	FIR (50-pt)
FIR	90800	58651	53913	62145	15525	14160	FIR (100-pt)
FloatToFract32	5080	3543	1358	940	314	365	Convert float to fract32 with clipping
Fract32ToFloat	1940	1679	1354	915	322	346	Convert fract32 to float
Interleave	2700	1771	2927	1617	1704	1268	Interleaves mono data to stereo
MaxAbs	1820	1126	931	914	446	352	Computes maximum absolute value across channels
Meter	2860	3804	824	539	312	1422	Standard level meter
Mixer	4810	3290	15001	4367	1868	1276	Mixer without smoothing
MixerSmoothed	1550	1109	2488	1620	979	443	Mixer with smoothing
MultiplexorFade	1410	785	1106	842	300	409	Multiplexor with crossfading
Multiplexor	1360	705	1187	823	292	371	Multiplexor without crossfading
Multiplier	4140	2145	2206	1960	817	785	Multiplier
MuteSmoothed	2420	1899	11950	2223	890	897	Mute with smoothing
NullSink	21	45	38	22	25	30	Discards samples
RMS	1680	1340	1245	786	289	487	Computes RMS across a block of samples
Router	1430	805	877	687	277	431	Channel router
ScaleOffset	2090	1323	15286	2724	857	339	Scales and adds a fixed offset
Scaler	1350	951	1545	979	453	363	Linear scaler without smoothing
ScalerSmoothed	2460	1942	12298	3376	896	941	Linear scaler with smoothing
ScalerN	1350	974	1560	980	469	616	N channel scaler without smoothing
ScalerNSmoothed	2360	1875	13148	2233	974	880	N channel scaler with smoothing
SineGen	2720	2372	22082	2971	3929	1688	Sine wave generator
Source	1420	816	1165	834	323	1595	Basic source
Sqrt	8590	8895	26044	6777	3846	10513	Square root
SubblockStatistics	6270	4580	22512	10702	6707	5959	Computes statistics on smaller blocks
Subtract	1790	1171	1497	1150	453	492	Subtracts two signals
Undb20Approx	31100	35765	9446	6849	3893	13079	Approximation to $10^{\log_{10}(x/20)}$
VolumeControl	3860	3040	5145	4316	2370	1502	Volume control with loudness compensation



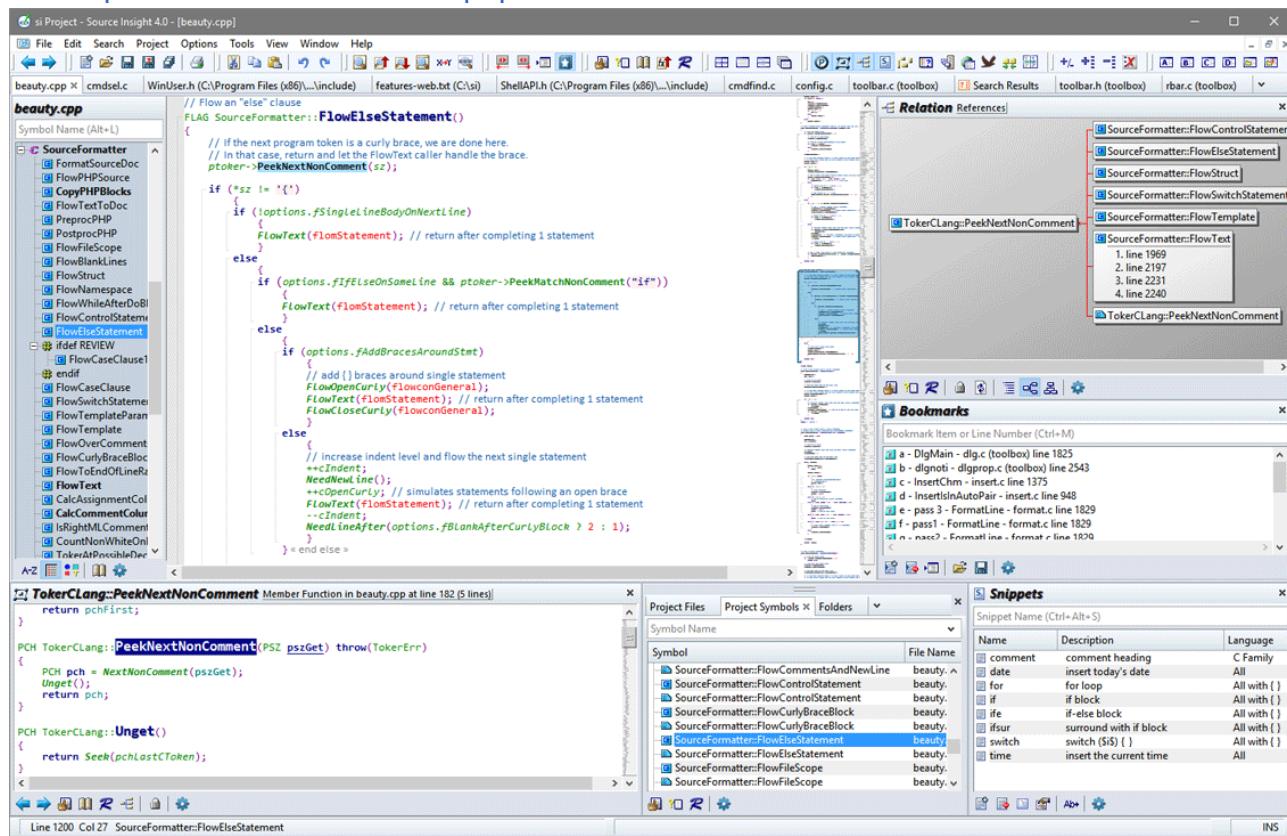
第52章 附件 B---Source Insight 源代码浏览和

编辑器

Source Insight 是一款面向项目开发的程序编辑器和代码浏览器。相对于一般的软件开发环境，Source Insight 能够更直观，更方便地浏览和编辑代码。

Source Insight4.0 安装包和注册，含早年做的无声入门视频：

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=82497>。

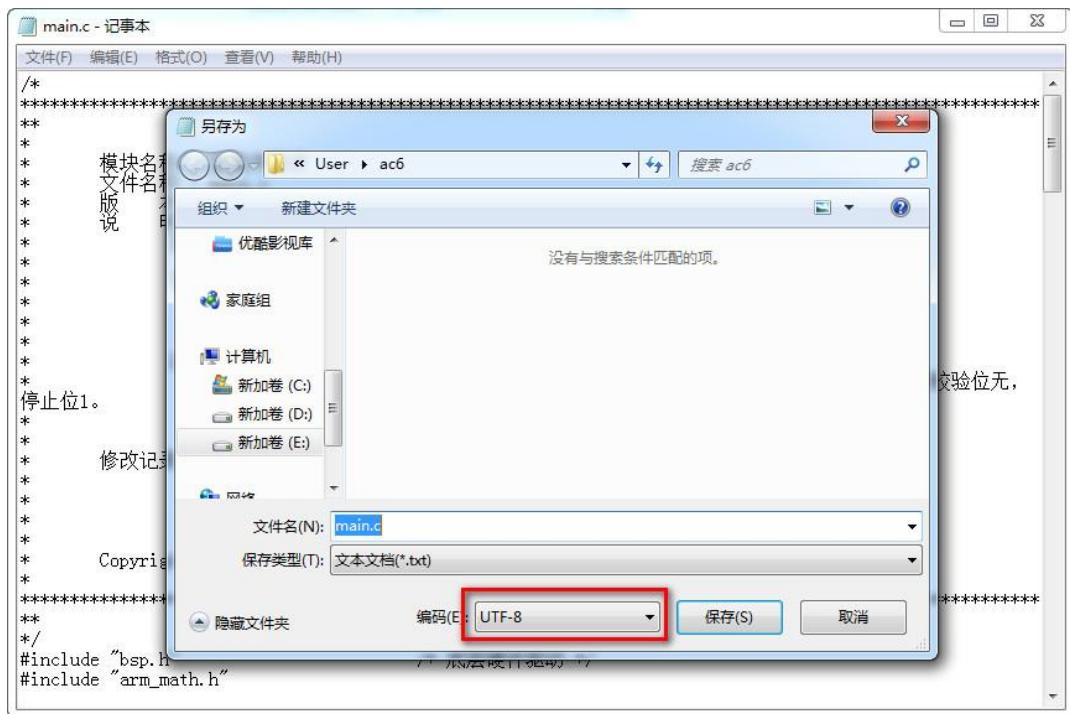


补充一个知识点：

同时测试，VS Code, Si, Sublime Text3 和 NotePad++，添加了 N 种插件后，那个编辑代码更溜
<http://www.armbbs.cn/forum.php?mod=viewthread&tid=24001>。

第53章 附件 C---MDK AC6 的汉字编码问题

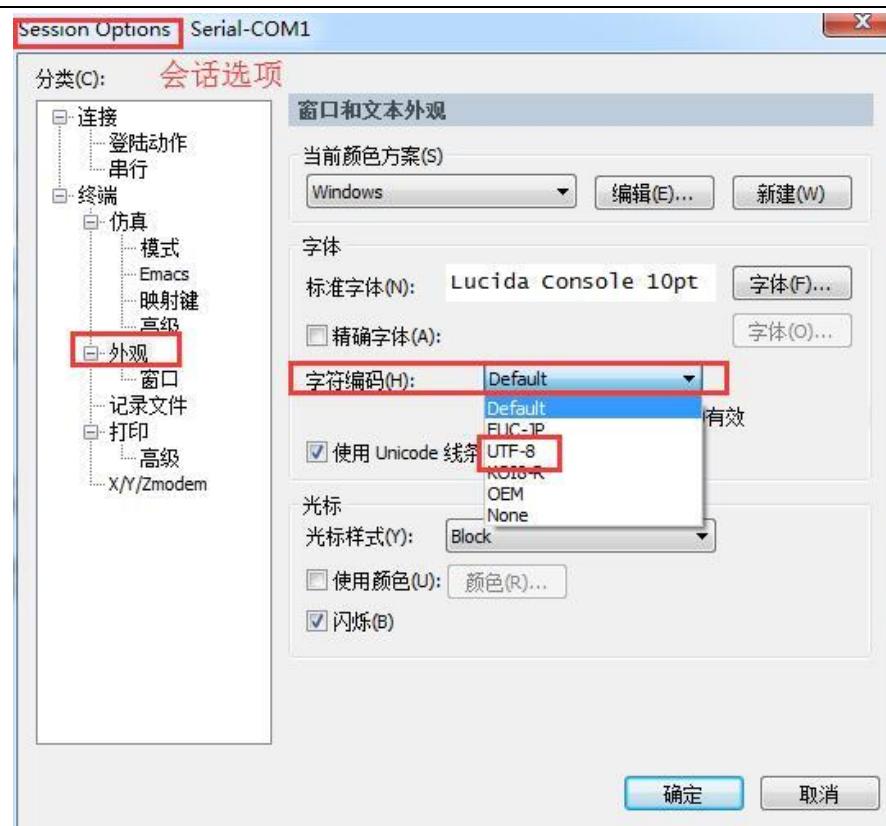
MDK 的 AC6 工程代码如果有源文件是 GBK 编码，而且使用汉字，MDK 编译时会报错，需要用记事本打开使用汉字的源码文件，另存为 UTF-8。比如 main.c 文件的串口打印函数 printf 用到了汉字，那么就需要做如下修改：



然后再重新编译就不会报错了。同时，串口打印时，使用的串口助手要支持 UTF-8，推荐用 SecureCRT，
下载地址：<http://www.armbbs.cn/forum.php?mod=viewthread&tid=91718>。

设置如下：







第54章 附件 D---安富莱 C 语言编程规范

后面会为大家专门整理出一个文档，当前已经整理发布到论坛：

安富莱 C 语言编码规范 1--文件与目录

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=85595>

安富莱 C 语言编码规范 2--排版

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=85596>

安富莱 C 语言编码规范 3 --注释

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=85709>

安富莱 C 语言编码规范 4 --可读性

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=85710>

安富莱 C 语言编码规范 5--变量、结构、常量、宏

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=85724>

安富莱 C 语言编码规范 6--函数

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=85725>



第55章 附件 E---MDK AC6 相比 AC5 的优势

ARM Compiler 6 我们简称 AC6, ARM Compiler 5 简称 AC5。

下面通过三个方面对 AC6 的优势做个介绍:

一、比较 IAR, MDK 的 AC5 和 AC6 以及 Embedded Studio 的 CLANG 和 GCC 编译 HAL 库性能。

详情看此贴了解

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=93102>。

二、KEIL 推出的 RTX5 安全认证和新的 C 标准库安全版本都是基于 AC6 编译。

详情可以看此贴了解: RTX5 的汽车级, 工业级, 医疗和铁路安全认证已经通过, 证书已颁发

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=94752>。

三、根据 ARM 官方的时间记录, AC6 是在 2015 年发布的。推出到现在已经快三年了, 各方面都有了比较大的发展。

◆ 首先是安全认证, 编译器也是有安全认证的, 下面是 AC5 和 AC6 的对比。

	Arm Compiler 5	Arm Compiler 6
Safety standards		
ISO 26262 (automotive)	✓	✓
IEC 61508 (industrial)	✓	✓
EN 50128 (railway)		✓
IEC 62304 (medical)		✓ ¹
Integrity level		
ASIL	Up to ASIL D	Any Safety Integrity Level
SIL	Up to SIL 3	

¹ Suitably validated for use in safety-related development

◆ AC5 和 AC6 的综合 PK, 根据这个比较, AC6 的综合性能提升了 14.9%。

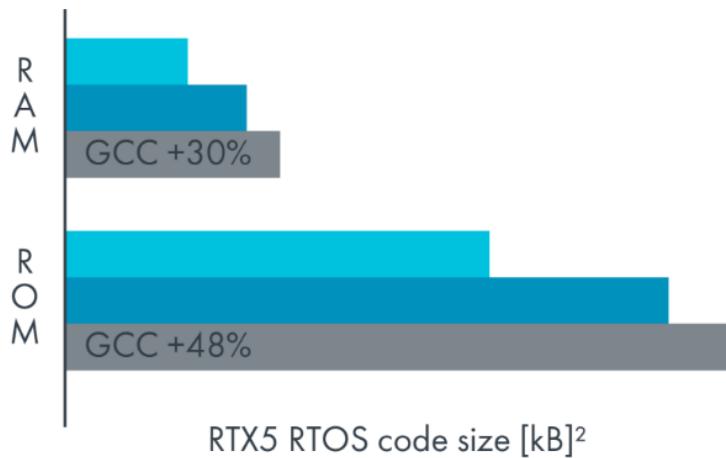
事实上, 在一些大型工业和汽车代码体系中, 性能比上一代增长超过 30%。这是针对复杂, 真实的嵌入式代码进行全面优化的结果, 而不是专注于某个芯片测试。

14.9%

Average performance gain*

across tested automotive, consumer, networking and telecom
workloads on Cortex-M7, compared to Arm Compiler 5.06

- ◆ AC6, AC5 和 GCC 生成代码量大小的比较。



- Arm 6.6 Oz LTO ■ Arm 6.6 OI ■ GCC 5u2 Os

更详细些，下面是使用了常用的标准代码做测试比较，结果如下：

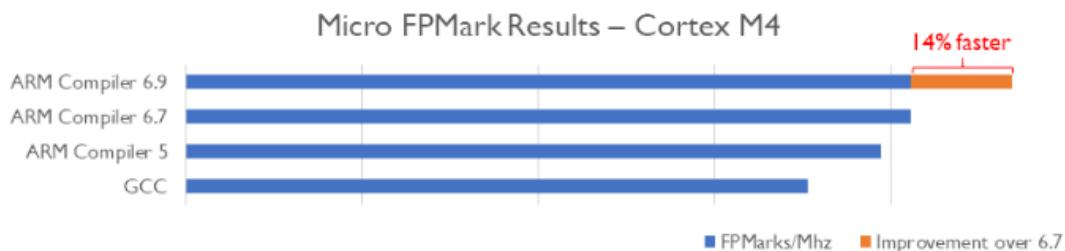
Software program	GCC code size	ARM Compiler 6 code size
EEMBC auto	80,874	37,115
Dhrystone	45,480	7,908
Sort	54,800	8,808
Coremark	53,640	20,188

下面这个是 GCC 使能优化库 newlib-nano, ARM Compiler 6 使能 microlib 对比：

Software program	GCC code size	ARM Compiler 6 code size
EEMBC auto	39,498	22,483
Dhrystone	12,590	4,404
Sort	14,044	3,412
Coremark	19,928	10,822

对比可以看出 ARM Compiler 6 编译器的优化效果非常明显。

- ◆ 随着 AC6 的不断升级，性能也一代比一代强。



The execution performance of generated code improved by an average of 10% compared to Version 6.7. Execution of FPMark for example improved by 14% on Cortex-M4 targets (see illustration above).



第56章 附件 F---DSP 库, MDK5 的 AC5,

AC6, IAR 和 SES 的三角函数性能

详情见此贴:

【性能测评】DSP 库, MDK5 的 AC5, AC6, IAR 和 Embedded Studio 的三角函数性能

<http://www.armbbs.cn/forum.php?mod=viewthread&tid=95455>



第57章 文档更新记录

版本	更改说明	作者	发布日期
V0.1	初版。 2019-08-31发布前7章。	白永斌 (Eric2013)	2019-08-31 (103页)
V0.2	更新两个章节。	白永斌 (Eric2013)	2019-09-08 (132页)
V0.3	更新一个章节。	白永斌 (Eric2013)	2019-09-14 (161页)
V0.4	更新一个章节。	白永斌 (Eric2013)	2019-09-29 (206页)
V0.5	更新一个章节。	白永斌 (Eric2013)	2019-10-06 (263页)
V0.6	更新一个章节。	白永斌 (Eric2013)	2019-10-20 (285页)
V0.7	更新一个章节。	白永斌 (Eric2013)	2019-10-26 (308页)
V0.8	更新一个章节和一个附加章节F。	白永斌 (Eric2013)	2019-11-02 (329页)
V0.9	更新两个章节。	白永斌 (Eric2013)	2019-12-29 (366页)
V1.0	更新两个章节。	白永斌 (Eric2013)	2020-02-16 (402页)



V1.1	更新两个章节。	白永斌 (Eric2013)	2020-03-22 (448页)
V1.2	更新两个章节。	白永斌 (Eric2013)	2020-03-29 (482页)
V1.3	更新两个章节。	白永斌 (Eric2013)	2020-04-19 (500页)
V1.4	更新两个章节。	白永斌 (Eric2013)	2020-05-17 (532页)
V1.5	更新一个章节。	白永斌 (Eric2013)	2021-04-25 (541页)
V1.6	更新一个章节。	白永斌 (Eric2013)	2021-05-16 (561页)
V1.7	更新一个章节。	白永斌 (Eric2013)	2021-05-23 (580页)
V1.8	更新一个章节。	白永斌 (Eric2013)	2021-05-30 (598页)
V1.9	更新一个章节。	白永斌 (Eric2013)	2021-06-07 (616页)
V2.0	更新一个章节。	白永斌 (Eric2013)	2021-06-27 (636页)
V2.1	更新三个章节。	白永斌 (Eric2013)	2021-07-11 (666页)
V2.2	更新五个章节。	白永斌 (Eric2013)	2021-08-01 (764页)



V2.3	更新两个章节。	白永斌 (Eric2013)	2021-08-09 (786页)
V2.4	更新四个章节。	白永斌 (Eric2013)	2021-08-15 (880页)
V2.5	更新一个章节。	白永斌 (Eric2013)	2021-09-06 (898页)
V2.6	更新一个章节。	白永斌 (Eric2013)	2021-09-19 (922页)
V2.7	更新一个章节。	白永斌 (Eric2013)	2021-11-01 (943页)