支持语音识别的运动手环开发笔记

GitHub: Metro94
Email: flattiles@gmail.com

第一部分 项目简介

前言

项目原来的名称是"可训练的离线语音识别系统"，后来由于各种原因，决定将项目名称改为"支持语音控制的运动手环设计"，从原来的单语音识别库开发改为一个使用语音控制的完整的可穿戴设备解决方案，希望可以更好地展示双核的强劲实力。

项目亮点

- 在单片机上实现**离线语音识别**，可以做到**实时分析**并输出各个关键词出现的概率。
- 同时使用多个传感器，可以作为实用的**运动手环**设计。
- 使用**双核**合理分配任务，将语音识别等需要大量计算的任务交给 Cortex-M4 完成，Cortex-M0+主要对各外设和传感器进行管理。
- 引入**操作系统**（RTX v5），对外设资源的调用进行统一管理，大大增加了程序的可读性，同时保证了较高的效率。
- **超低功耗**，在"深度睡眠+开启语音活动识别"的时候电流仅为 **200 uA** 左右（电压 1.8 V，包括单片机和数字麦克风）。
- 工作时程序仅在 SRAM 上运行，在达到最高运行效率的同时降低 Flash 的运行功耗。

第二部分 语音识别原理和实现

语音识别原理

从狭义范围来说，语音识别仅包括从模拟信号到语音内容或者音素的过程，不包括后续的语义、语法分析。根据我的理解，语音识别的过程可以分为两个步骤，即**特征提取**和**模式识别**。
特征提取的目的就是将复杂的语音信号转化为容易分析的信号量，本质上是一种降维处理的方法。目前，比较通用的特征提取方法是 **MFCC**，即 Mel-Freuquency Cepstral Coefficient，这种方法根据人耳对频率和能量的感知特性进行了一定优化，在减小数据量的同时仍能准确地反应出语音信号的特征。
模式识别就是将语音特征在对应的语音模型上进行匹配，以发现语音信号所表示的内容。目前学术上已有多种可行的方法，最为经典的莫过于 GMM-HMM 方法，当然还有一些基于神经网络的方法可供研究。各种方法的特点如下：

- GMM-HMM 将语音信号转化为音素，再由音素来匹配单词。这种方法需要事先存储音素和字典，相对而言较为复杂（代码量较大），但模型参数增长速度较慢，适合中等词汇量的语音识别。
- 基于神经网络的方法则主要对 GMM 进行改造，使用神经网络来分析音素之间的联系，以替代基于经验的人工分析方法，代码量相对较小，但参数需要占用一定空间，且需要花费时间训练神经网络。
- 也有一些尝试是将模式识别的过程完全交由神经网络实现，代码量相对最小，但随着单词数量增加网络模型快速膨胀，仅适用于少量关键词识别的情形。

考虑到计算量、代码量、存储量等因素，前两种方法更适合在高性能嵌入式设备上实现（如树莓派），如要在单片机上实现语音识别，只能采用最后一个方案。好在，ARM 为我们提供了一种在嵌入式设备上实现语音识别（关键词识别）的方案，使用的就是最后一种方式。相关代码在 GitHub 有，见 ARM-software/ML-KWS-for-MCU。

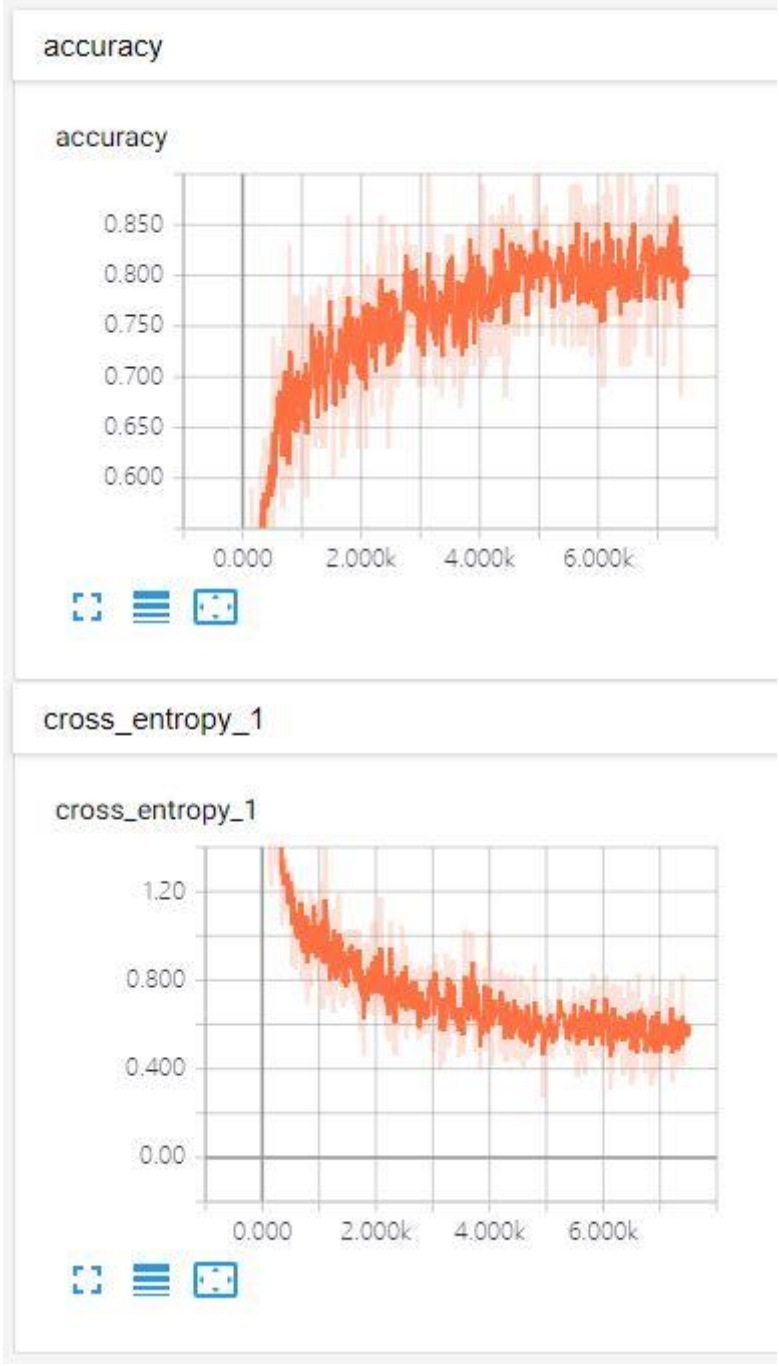
方案分析

ARM 为我们提供的方案如下：

1. 首先要做的还是 MFCC。MFCC 的过程占用的时间并不多，实时计算问题不大。
2. 然后将输出的 DCT 系数直接输入到训练好的网络中。使用 ARM 提供的 CMSIS-NN 库，推理代码可谓十分简洁，然而占用的时间比较长（只有最简单的 DNN 勉强满足实时性），且需要考虑精度损失。
3. 网络最后通过 softmax 生成各单词出现的概率，由此分析语音信号对应的单词。

LPC54114 的官方开发板是支持语音识别的（而且还特意推出了语音识别套件），但很可惜这部分的内容要签商业协议才能看到代码，所以只能看看网上的其它方案了。原本考虑的方案是 TIesr，但由于代码量太大难以在单片机上执行所以放弃了。正好，ARM 的方案不仅不需要太大的代码量（经测试不包括参数也就 10+ KB，可以放在 SRAMX 执行以达到最快速度，参数可以放在 Flash 中），而且还写好了相关工具，那就值得一试了。

在这其中，最麻烦的自然要数神经网络的训练了。ARM 的这个方案用的是 TensorFlow 的示例代码，并在其上加入了对量化的支持。而 TensorFlow 使用的数据库是由 Google 提供的 Speech Commands 数据库，目前的最新版本是 v0.02。这套数据库有一个缺点，就是只有 30 个英文单词，不过要添加其它关键词倒也十分方便，只需使用相关的命令即可。为了这次比赛，我自己录了一些中文单词的读音（直接使用开发板上的数字麦克风录取，并使用串口传到电脑上），并且自己进行训练。本人的渣渣低压 U 笔记本跑了一晚上才把十万级别的参数给跑完，效果还是可以的。



性能优化

考虑到单片机的性能，我们将输入的 PCM 音频限制为 16 位、8 kHz 采样率，这对于语音识别应用而言是足够的。由于有了 HWVAD 的加持，我们不需要时刻分析数字麦克风的语音数

据，只需在 HWVAD 的中断触发后进行采样即可。考虑到此次主要做关键词识别，我们规定一次采样时长为 800 ms（即 6400 次采样），在采样的同时做 MFCC 提取，并在采样结束后进行语音内容识别。另外，考虑到语音信号是前后关联的，每次进行信号分析的语音片段之间要有一定重叠，我们规定片段长度为 16 ms（即 128 次采样），片段重叠长度为 8 ms（即 64 次采样），这样一来共有 49 个语音片段，对应的 MFCC 为 13 个参数，也就是说神经网络的输入层共有(49*13)个参数。ARM 提供的工具（准确说他们大量采用了 TensorFlow 的示例代码）可以自动处理网络的参数个数（当然需要自行设置一些超参数），不需要我们关心。

在考虑好一些外部参数对性能造成的影响之后，不要忘了对代码本身进行优化。ARM 提供的代码是基于 mbed 的（采用 C++），其对于内存的使用过于“奔放”，且 MFCC 部分的代码处理效率较低。我花了点时间重写了所有代码（主要是 MFCC 部分），尽可能地引用 CMSIS-DSP 和 CMSIS-NN 这两个库来提高计算效率。经测试，MFCC 的处理时间小于一个语音片段，因此 MFCC 的分析可以实时完成；而 NN 部分则严重依赖所选用的网络模型。

第三部分 各传感器的原理和实现

MAX30102

MAX30102 可以用来测量**心率和血氧饱和度**，曾被用于三星的 S7 中（也是我的现役手机）。原来我以为 MAX30102 可以直接读出心率和血氧饱和度的数值，后来发现我还是太天真了。这个传感器说白了就是两个 LED（一个是 660 nm 红光，另一个是 880 nm 红外光）、一个可见光和红外光接收管、一个高精度 ADC 的组合，本质上和那种直接输出模拟信号的心率传感器区别不大。当然，MAX30102 还内置了温度传感器和一些实用的数字电路，可以对采样的过程进行精细控制。总的来说，MAX30102 的使用对于单片机来说和一个数字 ADC 没有太大区别，就是按时读取数据即可，当然内置的 FIFO 可以减少读取的次数，增加每次读取的数据个数，从而减小总线的占用率。

MAX30102 是怎么分析心率和血氧饱和度的呢？这主要依赖于血细胞对光的反射。具体原理可以参考这个帖子：MAX32630FTHR 板的学习（三）：MAX30102 的初步使用及心率、血氧的测量原理。由于我不是生医或者相关专业（大家猜猜我是什么专业的？），这里就不班门弄斧了。

在测量心率时，我们只需用到红光 LED。由于心跳信号有**周期性**，所以转化成电信号后依然具有周期性，我们只需分析出信号的最小周期就可以很容易地计算出心率了。这个过程通常有两种方法，一种是官方库使用的分析**波峰/波谷**，另一种则是网上看到的分析**信号相关性**（位移+互相关找出最小周期）。经过测试，后者对于采样个数较多的情况效果较差（因为信号可能发生干扰，从而破坏波形的相似性），因此我们主要采用前者。

测量心率的方法可以分为以下几步：

1. 截取一段信号，通常需要 4 s 长度，以达到更高的准确率。
2. 对信号进行适当的滤波处理。这里采用简单的滑动平均，可以获得不错的效果。
3. 去除“直流”信号。波形会因为外界干扰产生缓慢变化（可以理解为在心跳信号上叠加了低频率的伪“直流”信号），因此需要找出波形的“直流”分量并去除。这里可以采用高通滤波等方式去除，不过考虑信号的完整性和计算效率，这里使用**二次拟合**的方式来找出“直流”信号。
4. 找出峰值或谷值点。经过对信号的分析，MAX30102 输出的心跳信号的谷值点会更明确一些，因此使用谷值点来判定一次心跳的开始和结束。至于峰值或谷值点的判定则十分简单，只需找出某个“比前后若干个数据都更大/更小的点”即可。
5. 分析谷值点之间的距离，求出心率。有时候可能会漏掉一些谷值点，导致距离远大于实际周期值，这里应该做好相关判断。

上述步骤都不大复杂，要说最难的可能是找出峰值或谷值点这一步了。最值点的窗口长度怎么取当然是一个问题，但是更值得思考的是算法的时间复杂度。我们知道，如果要找出 n 个数据中每 k 个连续数据的最值，只需要写一个二重循环即可，外层枚举窗口的第一个数据位置，内层枚举窗口内的每一个数据，这样做所需的时间复杂度为 O(kn)（用大 O 代表 Omicron，下同）。那么，有没有更快的方法呢？答案是使用优先级队列。在这里我们就不详细介绍优先级队列的原理了，只需知道使用优先级队列的时间复杂度可降至 O(n)，

和 k 无关。

计算血氧饱和度的方法也并不复杂。此时需要同时用到 RED 和 IR 两个 LED 的数据，计算的公式为$\frac{RED_{AC}/RED_{DC}}{IR_{AC}/IR_{DC}}$，还要经过函数$f(x) = 94.845 + 30.354x - 45.060x^2$才能得到最终结果（结果为百分数）。这里的原理我们不去深究，只需想办法求出 DC 和 AC 分量，而这在之前的拟合过程中已经求出来了。这里，我们定义相邻峰值和谷值点的高度差为 AC，而两点之间的值则为 DC，经过测试符合要求。对于正常人而言，一般的血氧饱和度在 95%以上。

考虑到算法本身的计算量不小，因此适当的优化是必须的。以下几点是我在算法中使用的优化：

1. 首先是对阈值的判断。在传感器前无障碍物时，读数通常是较低的（测试下来小于 1000），而在手指接触传感器时读数可达 100000 以上。这里我们选择 50000 作为阈值，若任意一次采样的读数小于阈值则认为没有检测到，算法不再进行处理。
2. 使用最小二乘法拟合的公式是$\mathrm{k} = (X^T X)^{-1} X^T y$，其中 X 为样本对应的 Vandermonde 矩阵。由于 X 的值是固定的，因此我们可以预先计算好 $A = (X^T X)^{-1} X^T$的值，这样每次就只需要计算*(多项式次数+1)*样本数*的乘法了。
3. 寻找峰值和谷值使用上面提到的优先级队列。注意使用优先级队列需要额外的 O(k) 空间，不过这对我们的应用来说只是九牛一毛。

BME280

BME280 是 Bosch 的一款传感器，支持温度、湿度和大气压的测量，精度应该属于还可以的水平。Bosch 出了不少相关的气体传感器，接口都是一样的，支持 I2C 和 SPI（部分产品没有）接口，相互移植还是非常方便的。

这个传感器本身的坑并不多，因为功能十分简单，就是测量这几个参数而已，不仅没有 FIFO（意味着每次新数据生成后都要及时读取），甚至连中断引脚都没有，免去了在单片机上编写 ISR 的额外工作。BME280 的主要难点在于数据补偿，传感器内置了一些补偿的参数，单片机需要使用这些参数计算补偿后的数据，好在官方也给了我们足够的测试代码。

BME280 共有两种工作模式（Sleep Mode 不计），分别是 Normal Mode 和 Forced Mode，在我看来就是对应连续采样和单次采样的区别。Normal Mode 适合需要连续获取数据的情况，而 Forced Mode 则可用于只需获取一组数据，或是由主机来控制采样速率的情况。不管使用哪种模式，都可以使用 BME280 内置的超采样和滤波器功能来过滤噪声，提高数据的有效性。

BME280 的补偿算法相对比较复杂，且湿度和气压的补偿算法需要参考补偿后的温度值。目前能在网上找到的两份代码来自 Bosch 和 SparkFun，两个代码的功能都是一样的。官方的算法有个缺点，就是使用定点数进行计算时的移位操作使用乘法而不是移位指令，这可能会降低效率。当然，官方的算法有不少关于精度的选项，这里我们考虑到 Cortex-M4 的

性能，在温度和湿度的补偿算法内部使用 int32_t，而在气压的补偿算法内部则使用 int64_t。

重新实现后的算法效率如下，可以看到比官方的算法要快一些（ARMCC -O3，使用标准库）：

	官方库	我的实现
温度	~46	**28**
气压	~1400	**349**
湿度	~83	**55**

在编写补偿算法的时候，我遇到了以下几个问题：

1. 结果值错误，后来发现是运算的过程中超了精度（两个 32 位数相乘后结果仍为 32 位，导致数据丢失），这里只要够细就能发现问题。
2. 另一个是关于 MicroLib 的问题。一开始我发现执行的周期数达到官方的两倍，并且打开编译优化选项后效果不明显。后来发现是因为打开了 MicroLib 导致的问题，把 Use MicroLib 关了之后执行周期立马就下来了。对于这种情况，如果不使用半主机等必须由 MicroLib 实现的功能，还是建议使用标准库来提计算速度。

最后是一张测试图（图见 1 楼实物图），可以在 OLED 上看到温湿度和大气压的数据，更新速度约为 8 sps，这对于温度监测应用已经绰绰有余。

CCS811

CCS811 可用于测量 eCO2 和 eTVOC，主要针对室内应用。虽然测量精度比不过专业仪器，但是用来玩玩还是可以的。网上对于 CCS811 的评价可谓是“怨声载道”，纷纷抱怨这个传感器的工作流程复杂、datasheet 不够清楚。不过呢，只要把握好它的工作原理，操作思路也就不难理解了。

CCS811 的奇葩之处，主要在于在传感器的内部集成了一个单片机。和传统的单片机相同的是，CCS811 的运行模式可以分为 Boot Mode 和 App Mode，前者对应 Bootloader，后者则是执行配置和测量任务的应用固件。CCS811 的启动流程如下：

1. 复位后进入 Boot Mode，不管是上电复位、硬复位（拉低 nRESET 引脚）还是软复位（写入 SW_RESET 寄存器），都会在一定时间后进入到 Boot Mode，其中上电复位所需的时间更长（20 ms VS 2 ms）。
2. 在 Boot Mode 中，可以对应用固件进行烧写，也可检测应用固件是否存在。此时不能执行跟传感器的配置和测量有关的操作。
3. 如果应用固件存在，应写入特定的寄存器，以从 Boot Mode 切换至 App Mode。这一过程需要 1 ms。
4. 应用固件启动，此时应先配置传感器，再开始进行测量。

可以看到，这就是一个完整的单片机程序启动流程。根据这个设计思路，CCS811 在 I2C 总线和中断引脚（nINT）之外，还另外设计了**硬复位**（nRESET）和**唤醒引脚**（nWAKE），前者不用多说（由于软复位的存在似乎也用不上），后者则是在进行 I2C 传输时必须拉低，并在传输过程中保持低电平。不过，nWAKE 引脚并不影响测量的流程，在不需要进行 I2C 传输时可将其拉高以减少电流消耗。

CCS811 的测量流程比较简单，就是设置好采样模式，在相应指示位置位或是 nINT 拉低（需开启中断功能）后便可读取数据。由于 CCS811 的内部已经处理好数据，主机端只需直接使用即可，这估计就是使用内部单片机的原因了。CCS811 的测量速度比较慢，最高只能达到 1 Hz（原始数据可以达到 4 Hz），且室内空气质量基本上在最低阈值以下，因此演示的效果并不是很好，只能吹口气然后观察数据了。

这里我们还有一个重点，就是 CCS811 可以输入温湿度数据以补偿测量结果。CCS811 的输入数据可以达到 16 位精度，而且 CCS811 不内置温湿度传感器（猜测内部加热装置可能会影响测量结果），因此我们需要用外部传感器来进行温湿度测量。正好，我们可以用 BME280 测量的数据来进行校准。在 BME280 中使用 Forced Mode 读取一组温湿度数据，然后写入到 CCS811 即可。

编写 CCS811 的驱动还是比较顺利的，没遇上什么问题，不过这是在前人踩了不少坑的前提下，如果让我看着 datasheet，一开始也会不知所云吧 233。测量效果图如下所示（图见 1 楼实物图）。

MPU-9250

MPU-9250 是一个九轴传感器，内置的 **DMP** 可以实现许多高级功能，这里我们使用的计步（即 **Pedometer**）就是直接使用 DMP 实现的，避免在主机上进行复杂的计算。不过呢，DMP 内置的计步功能并不能在检测到步行时产生中断信号，因此需要使用轮询的方式获取步数和步行时间，时间间隔以 0.5 s 到 1 s 为宜。

MPU-9250 已经提供了官方驱动库（即 Motion Driver 6.12），包含 DMP 的相关 API，不过这部分内容似乎并不是公开的，所以这里我就不放出代码了，相关代码可在 sparkfun/SparkFun_MPU-9250-DMP_Arduino_Library 中下载，其中的 src/util 文件夹下以 dmp 和 inv 开头的几个文件是官方库，其余的是 SparkFun 编写的二次封装库。移植的过程比较简单，因为官方使用了 define 来使用几个外部已定义函数（例如 I2C 的读写、delay 等），我们只要修改为实际使用的函数即可。

由于我们只使用计步功能，且不要求在使用过程中关闭（毕竟息屏的时候也要计数嘛），所以调用的过程就显得非常简单了，我这里是参考上面 repo 中名为 MPU9250_DMP_Pedometer.ino 的例子实现的，效果还是挺不错的。不过要注意的是，DMP 的算法是仅当**连续行走步数大于 7 步**（一说为 5 步）时才开始计步过程，如果小于该步数则不计入，以避免环境导致的误差。

VL53L0X

VL53L0X 是一个用激光来测距（即 ToF，Time-of-Flight）的传感器，精度可达±3%（高精度模式），最远距离为 2 m（长距离模式），可用于较远距离的姿势识别（此时需要多个传感器）。我们这里主要使用其精度高的特点，通过统计低于/高于阈值的数量实现计数过程，这在一些需要计算次数（如各种力量型训练）等地方是非常实用的。

VL53L0X 提供了官方驱动库，和 Motion Driver 一样已经把所有的 API 都封装好了，只需替换 I2C 的读写、delay 等相关操作即可。同样地，这里我参考了 adafruit/Adafruit_VL53L0X，不过这个例子只有单次测量模式，我在这之上添加了**连续测量**（即 Continuous ranging）模式的支持，并使用中断引脚通知主机一次测量完成。XSHUT 引脚可在不使用时拉低，以减小此时的功耗。

使用 VL53L0X 的驱动库有几个注意事项：

1. 如果 VDDIO 与 VDDA 相同或相近，则需要设置使用高压模式，方法是在 vl53l0x_api.c 文件或包括的头文件中定义 USE_I2C_2V8。
2. 驱动在测距时默认使用轮询模式，即保持查询状态寄存器直到测量完毕。如果要使用中断模式，则在设置 GPIO1 的基础上，还需要重新定义 VL53L0X_PollingDelay 函数，改成等待中断的方式。

第四部分 操作系统的使用思路

RTOS 简介

单片机需要使用操作系统吗？如果对操作系统的定义包括 RTOS（即 Run-Time Operating System，实时操作系统），那么答案显然是可行。单片机上使用操作系统主要解决两个问题：

1. 进程的运行和调度。简单来说，RTOS 解决了任务之间互不干涉和内核只有有限个之间的矛盾。RTOS 提供了一种解决问题的视角，即通过将不相关的任务以进程的方式区分并各自运行，以达到降低耦合性、提高内聚性的效果。如果合理划分进程，可以达到事半功倍的效果。
2. 系统级别的服务或资源提供。有一些服务或资源是全局性的，必须要有一个“统治地位”高于普通进程的“组织”来提供，而这正好是 RTOS 的工作。举个例子，操作系统中常见的互斥量（即 Mutex）和信号量（即 Semaphores）是进程使用有限资源时的必要工具，而这些内容的管理超出了进程的业务范围，因此必须有一个“管理员”级别的软件来专门处理此事。

对于 Cortex-M 系列内核，很多 RTOS 都支持在上面运行，从极简内核（如 RTX）到功能丰富（如 uCOS），可选项非常多。这里我强烈安利 ARM 官方推荐的 RTX，它有几个优点：

1. 内核占用空间小，SRAM 使用空间小（相对于 FreeRTOS 之类的操作系统）。
2. 任务切换速度快，支持“零延迟中断”，充分利用 ARMv7m 使用的特性（例如 SVC/PendSV 异常）。
3. 使用统一的 CMSIS-RTOS2 API，容易替换其它 RTOS 使用。

由于我们需要执行的任务对于 RTOS 来说并不复杂（主要是进程的调度、事件检测和互斥量/信号量操作），并不需要使用到复杂的协议栈等传统 RTOS 的优势项目，所以这里选择 RTX 自然是再好不过。接下来我们看看我们可以用 RTOS 解决哪些实际问题。

任务调度

根据程序的设计，我们需要在 RTOS 上运行这么几种任务：

1. **主应用程序**。主应用程序实际上就是控制运动手环程序执行流程和 GUI（不要瞧不起 OLED12864 哈）的进程。该进程会根据执行流程来运行各种功能对应的次级应用程序。例如，在运行时发出语音指令切换到心率测量时，语音分析程序会在后台运行，主程序接收分析结果并通知心率测量程序执行。
2. **次级应用程序**。每个次级应用程序都对应一个功能，按照这种思路可以去除进程之间的耦合（这由功能之间的正交性保证），而 IPC（即 Inter-Process Communication，任务间通信）则由 RTOS 完成。
3. **专用算法**。有些次级应用程序需要运行复杂的算法（最典型的莫过于语音识别和心率测量），因此在次级应用程序中直接执行这些算法可能会影响到原来的功能（例如在分析语音信号时新的语音信号因未被即时读取而丢失），这个时候就可以继续

将算法独立为进程，从而在运行复杂算法时可以在其中打断，并执行对实时性要求更高的 I/O 操作。

那么，这几种算法的优先级应该如何定义呢？很显然，优先级应该是**次级应用程序>专用算法>主应用程序**，这是因为：

1. 次级应用程序是操作 I/O 的直接对象，将其置为最高的优先级有助于即时处理任务。
2. 专用算法需要让位于次级应用程序，但优先级又要高于主应用程序，这是因为专用算法的运行结果需要传送给主程序进行最终处理（例如显示到屏幕上），主程序一般都是处于等待结果的状态中。
3. 主应用程序需要控制次级应用程序的运行，如果主程序的优先级过高，次级任务将不会执行。

通过上面的分析，任务调度的标准是很容易确立的。至于进程间通信，我们有这三个工具：

1. Thread Flags，主程序可以利用 Thread Flags 给次级程序发送一些指令（如退出进程），从而使得进程可以做出一些必要的响应（例如在退出进程前关闭相关传感器以节省功耗）。
2. Event Flags，主要传递从 ISR 到进程的信号。由于 ISR 中可以使用的 RTOS API 有限，因此 Event Flags 就成为了为数不多的工具了。很多情况下，我们需要在某个进程中等待来自传感器的中断信号，此时使用 osEventFlagsWait 函数可以在中断来临前将进程置于阻塞状态，从而允许其它进程执行。
3. 进程参数。RTOS API 中规定进程的函数类型为 void(* osThreadFunc_t)(void *argument)，这里的 argument 就可以拿来做文章，比如传递一个回调函数进去，既可以让次级程序的结果快速传递到主程序，也可以让主程序将结果处理的代码（如输出结果的代码）直接交由次级程序实现，且不用在次级程序中编写无关代码。

有了这些详细的设定，任务的调度应该不成问题，足以应对我们这个不算简单的应用了。

片上资源分配

相对于片外资源，片上资源往往会被多个进程同时使用，因此如何合理分配是最令人头痛的问题之一。好在，使用信号量可以非常简单地解决这一问题。

片上资源的使用可以分为两种，一种以总线协议为代表，每次只能有一个进程使用，多个进程同时使用会造成数据读写混乱（特别是像 I2C 这种多从机协议）；另一种则以 DMA 为代表，每个进程可以独立地调用 DMA，但可能有最高数量限制（当然这和 DMA 的设计有关，至少这里用的 LPC54114 不存在这个问题）。

当然，使用是一方面，如果要考虑时钟运行的话就又是另一个话题了。是否运行相应外设时钟的标准十分简单，就是判断正在使用片上资源的进程数量是否大于 0。这个工作正好

可以和控制权的分配绑定（没人控制正好对应不需要时钟的情况，反之亦然），使用的方法就是**信号量**。我们为信号量指定这样的使用方式：

1. 在启动 RTOS（即执行 osKernelStart 函数）之前，根据实际情况新建信号量并确定信号量的最大容量，其中 I2C 等模块为 1（表示同时只能有一个应用同时使用），DMA 等模块为 0xFFFF（表示有"无限"个应用同时使用，注意这里有个小坑，RTX 在内部会将容量值转换成 uint16_t 类型）。
2. 要获取某个模块的控制权时，需要先使能该模块对应的时钟。由于在获取模块的控制权后信号量被使用的容量肯定大于 0（至少有一个应用在使用），所以这里直接使能时钟是可以的。
3. 要释放某个模块的控制权时，可能在之后要关闭该模块对应的时钟。由于释放模块的控制权后使用该模块的应用数量可以是包括 0 的值，因此需要获取信号量的当前容量，并和最大容量对比得出已使用的数量。

但是，第 3 点会存在一个根本的问题——如果存在任务抢占行为，"对比信号量的容量"和"关闭时钟"可能不是连续执行的（尽管在代码是连续的，这通常意味着发生了中断或进程抢占），这会导致判断标准出错。一种好的解决方式是，在操作系统层面来解决这个问题——也就是采用 SVC 的方式。RTX 允许用户自定义 SVC 指令，这需要用户定义 osRtxUserSVC 变量，并通过特殊的语法告知编译器这里将产生 SVC 中断，从而生成正确的代码。具体的实现方式不再多提，大家可以看看代码。

低功耗模式

低功耗模式是我这个项目的一大卖点，因此在 RTOS 层面也要做好相关工作。

RTOS 对低功耗模式的实现主要是通过"空闲进程"的方式。空闲进程是整个 RTOS 中优先级最低的进程，只有当系统其它进程都不处于运行状态时才会进入。在此进程中，RTOS 可以只执行空循环（在任务状态发生改变时由于优先级最低会自动离开空闲进程），也可以做更多的事情——例如进入低功耗模式。RTX 默认的空闲进程只是一个无限的空循环，可以通过自定义 osRtxIdleThread 函数的方式来自定义空闲进程。

LPC54114 定义了多种低功耗模式，那么如何判断当前应该进入哪个模式呢？这个问题很大程度上取决于你的应用。例如，我们知道 LPC54114 的 I2C 在深度睡眠模式下只能作为从机使用，作为主机的话最多只能位于睡眠模式下（只有 CPU 停止工作），即使使用 DMA 传输也是一样；但是，一旦不使用 I2C 就可以进入深度睡眠模式的话，我们就需要在空闲进程中判断此时 I2C 是否在运行，而 I2C 是否在运行的判据我们在分配片上资源时已经做好了。由于空闲进程是由用户自定义的，所以这些操作都不成问题。

总结

RTOS 的价值主要体现在对"任务"和"资源"的高度抽象，它可以帮助我们解决很多本需要用大量代码解决的问题。但是，我们也应注意到 RTOS 对单片机资源的消耗，只有在确实需要时才使用。

第五部分 SoC 与开发板简介

LPC541xx 简介

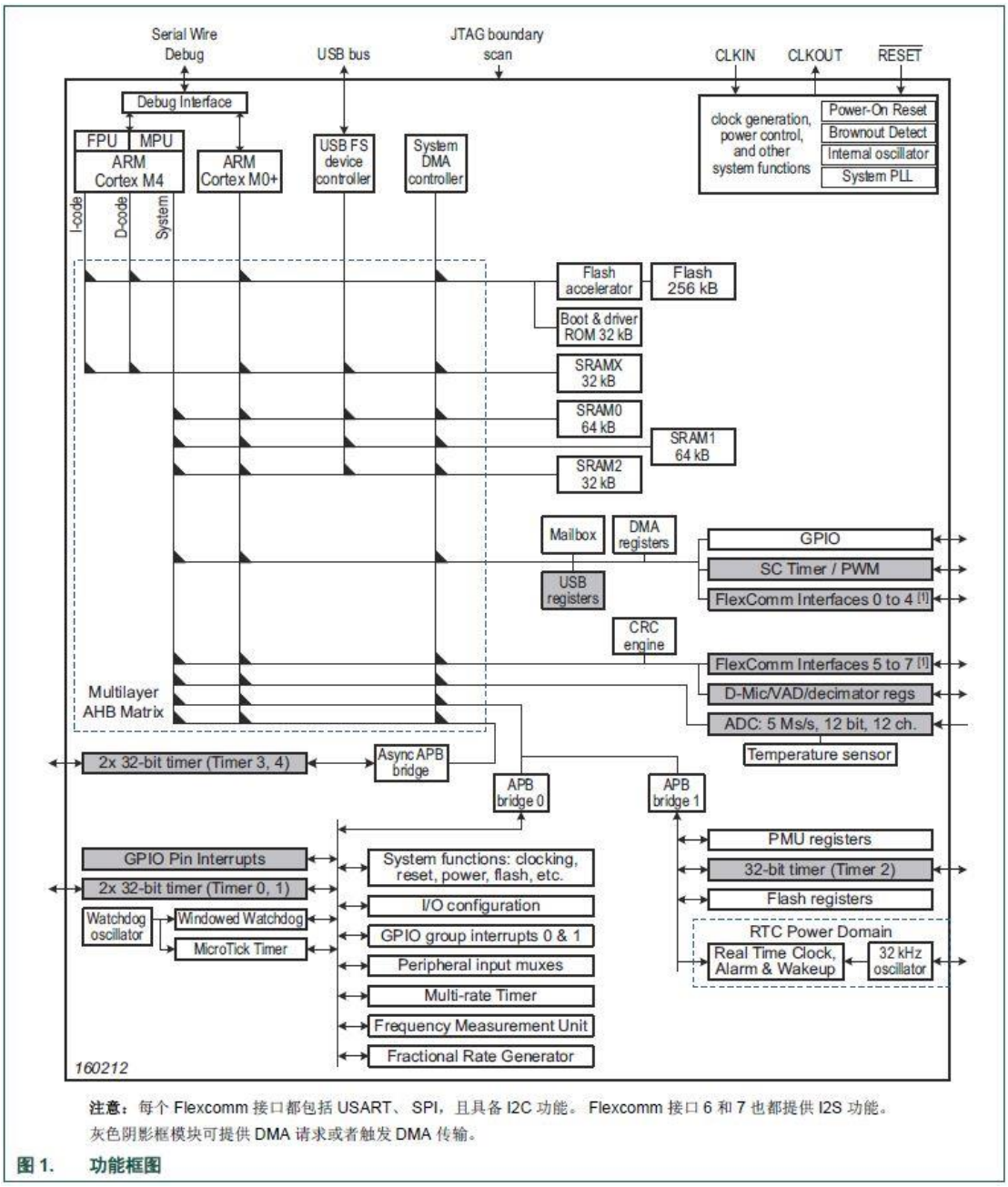
根据我的经验，要深入了解一个单片机的使用方式，必须要对性能、外设和系统架构有充分的认识。系统架构是容易被忽视的一个点，但它在一些系统设计中是非常重要的，例如 LPC437x 中的 Subsystem Cortex-M0 就与 SGPIO 直接连接，但访问大部分系统组件时则需要通过一个 AHB Bridge，这表明 Subsystem Cortex-M0 应该主要用于管理 SGPIO 的 I/O 操作。

首先来说说性能。LPC541xx 中只有一部分型号是双核的，且双核的型号都是 Cortex-M4F 与 Cortex-M0+的搭配，相对 LPC43xx 系列而言升级了小核（LPC43xx 使用 Cortex-M0），但时钟频率有所降低，最高为 100 MHz。如果不考虑小核，定位大致相当于大家熟知的 STM32F41x 系列。新增的 Cortex-M0+同样支持在 100 MHz 下运行，我们知道 Cortex-M0+的性能要弱于 Cortex-M4F，但运行时的功耗更低。

相对于双核的卖点，LPC541xx 的外设则显得有些平淡，像 USB FS、ADC、DMIC 等都算是这个档次的标配。能称得上是亮点的有以下几个：

1. HWVAD（基于硬件的语音活动检测）。这也是我这次参赛的主要方向之一。HWVAD 是一个容易被忽视、但实际上很有用的功能。相对于传统的检测声音电平的方式，HWVAD 使用了基于硬件的包络检测器，且能对信号和噪声分别进行检测，如果检测的结果乘上各自的增量满足一定的表达式则认为有语音活动。相较于其它方案，HWVAD 能够大大提升检测出的语音活动的有效性，同时功耗基本维持不变。
2. Flexcomm。其实 Flexcomm 就是综合了 USART、SPI、I2C 和 I2S 的一个模块，主要优势就是可以任意配置各种总线的数量，外加支持 PDM to I2S，也就是这样了。
3. ROM API。虽然 LPC541xx 貌似只开放了 USB 的 ROM API，但是就这一项节约的 Flash 空间确是实实在在的。
4. 电源管理。严格来说这并不算是外设（除了片上的调压器），但是 LPC541xx 的电源管理策略算是非常的简单好懂，睡眠模式、深度睡眠模式和深度掉电模式三个档次，电源和时钟想开啥关啥都在一个地方设置，不用对着参考手册看每个模式关闭了哪些外设，实际效果也相当不错。

LPC541xx 的系统架构相对而言也比较简单，相关的文档解释得比较清楚了。不过奇葩的是，LPC5410x 和 LPC5411x 的系统架构有很大区别，甚至连内存空间的分配都不一致，严重怀疑这是拿两个不同系列的单片机改的，相互之间的代码也无法共用。考虑到我们这次使用的是 LPC54114，这里只介绍 LPC5411x 的系统架构。



注意：每个 Flexcomm 接口都包括 USART、SPI，且具备 I2C 功能。Flexcomm 接口 6 和 7 也都提供 I2S 功能。
灰色阴影框模块可提供 DMA 请求或者触发 DMA 传输。

图 1. 功能框图

上图是一张系统架构图。我们按照不同的划分方式进行介绍。

总线

同所有 ARM 单片机一样，LPC5411x 同时具有 AHB 总线和 APB 总线。

AHB 总线是 SoC 的“高速通道”，连接的通常是 CPU 和一些高速模块。这里可以看到，两个核心的地位是完全等同的，且 USB 和 DMA 也可作为 AHB 总线的主机，以方便发起数据传输。比较有趣的是 SRAMX 的连接方式和 Flash/ROM 相同，都是接到了代码区域（即 0x00000000-0x20000000），这使得使用了哈佛体系架构的 Cortex-M4F 可以在最高速度下运行，甚至比在 Flash 还快(Flash 需要考虑延迟和预取的问题)，但对于使用冯·诺依曼体系架构的 Cortex-M0 而言没有影响。因此，在主时钟速度高于 12 MHz 且程序不大的情况下，Cortex-M4F 运行

SRAMX 中的程序可获得最高的性能；至于 Cortex-M0+，则放在 SRAM 中执行也会比 Flash 更快，但是否放在 SRAMX 则没有影响。

APB 总线则分为两个，分别是同步和异步，其中前者使用总线时钟，而后者可以使用其它时钟，并通过 APB 总线同步单元进行同步。异步总线的好处是主时钟的运行状态和频率对异步总线时钟没有影响。

需要注意的是，LPC5411x 允许对各个模块的复位和时钟信号进行控制，利用好时钟信号的开关可以使芯片的功耗更低。

时钟

时钟对于外设来说是必需的吗？答案是未必：对于 IOCON 和 INPUT MUX 之类的模块，通常只要配置一次即可，且配置后不需要再对寄存器进行读写，那么便可以在配置后关闭相应总线时钟即可。以上例子清楚地表明了总线时钟的作用：只有在需要对外设进行读写时才需要时钟，若关闭相关总线时钟则无法修改但可保持现有状态。这对其它时钟来说也是一样的。
当然，除了开关时钟以外，还有一种用法是调整时钟频率。我们知道，频率和功耗一般是正相关的关系，更高的频率通常带来更高的功耗，但同时也意味着更强的处理能力，这就需要通过测试找到一个平衡点了。对频率的调节可以通过选择时钟源和 PLL/分频器来完成。

时钟源

LPC5411X 的时钟源较为丰富，涵盖了低速（看门狗振荡器和 RTC 振荡器）到高速（FRO 和 PLL）等选项，并且除了主时钟树（见下文“时钟树”部分）之外还有一些特殊的应用，我们来细细分析。

FRO 12 MHz (fro_12m)
该时钟属于片内振荡器(IRC)，且为 CPU 和总线的默认时钟，猜测从下文提到的 fro_hf 分频而来。根据参考手册，SYSCON 模块的时钟源也是 fro_12m。
FRO 48 MHz/96 MHz (fro_hf)
该时钟属于片内振荡器，可选 48MHz 或 96MHz，出厂时经过校正，按数据手册精度可达±1%，足以用于 USB 的时钟。
Watchdog oscillator (wdt_clk)
也就是所谓的看门狗振荡器，也是片内振荡器，频率可在 0.4 MHz 到 3.05 MHz（未分频）之间选择。精度比较差（按数据手册为±40%），但是功耗特别低。wdt_clk 是 Micro-tick 和 WWDT 的唯一输入时钟，且可用于 DMIC，在测量频率后应该还是个挺好用的时钟。
RTC oscillator (32k_clk)
搞过 RTC 的同学都知道 32.768 kHz 时钟的重要性。32.768 kHz 时钟需要外接晶体才能产生，估计是因为体积太大不好集成。该时钟主要用于 RTC 模块（见下文“时钟树”部分），但也有一个特别的应用是 UART 在异步模式下作为从机检测输入信号。PLL 可以将 32k_clk 作为输入源。
Clock IN (clk_in)
输入时钟，注意只能接有源晶振（类似于 STM32 的 Bypass 模式），因为片上没有振荡器，不能接无源晶体。估计这个时钟只在需要和其他器件同步时才用得到了。
PLL (pll_clk)

PLL 大家应该都很熟悉了，参考文档也说得很详细，这里就不多提了。需要注意的是 PLL 的功耗也是相当可观的，如无必要可减少使用。

Flexcomm/DMIC clock (mclk_in)

专用于 Flexcomm/DMIC 模块的外部时钟，没啥好说的。

时钟树

关于时钟树的部分我觉得没啥好说的，贴两个图让大家看看就好。注意第二个图是 RTC 的时钟树，详细地解释了 32.768 kHz 时钟的各种用途。

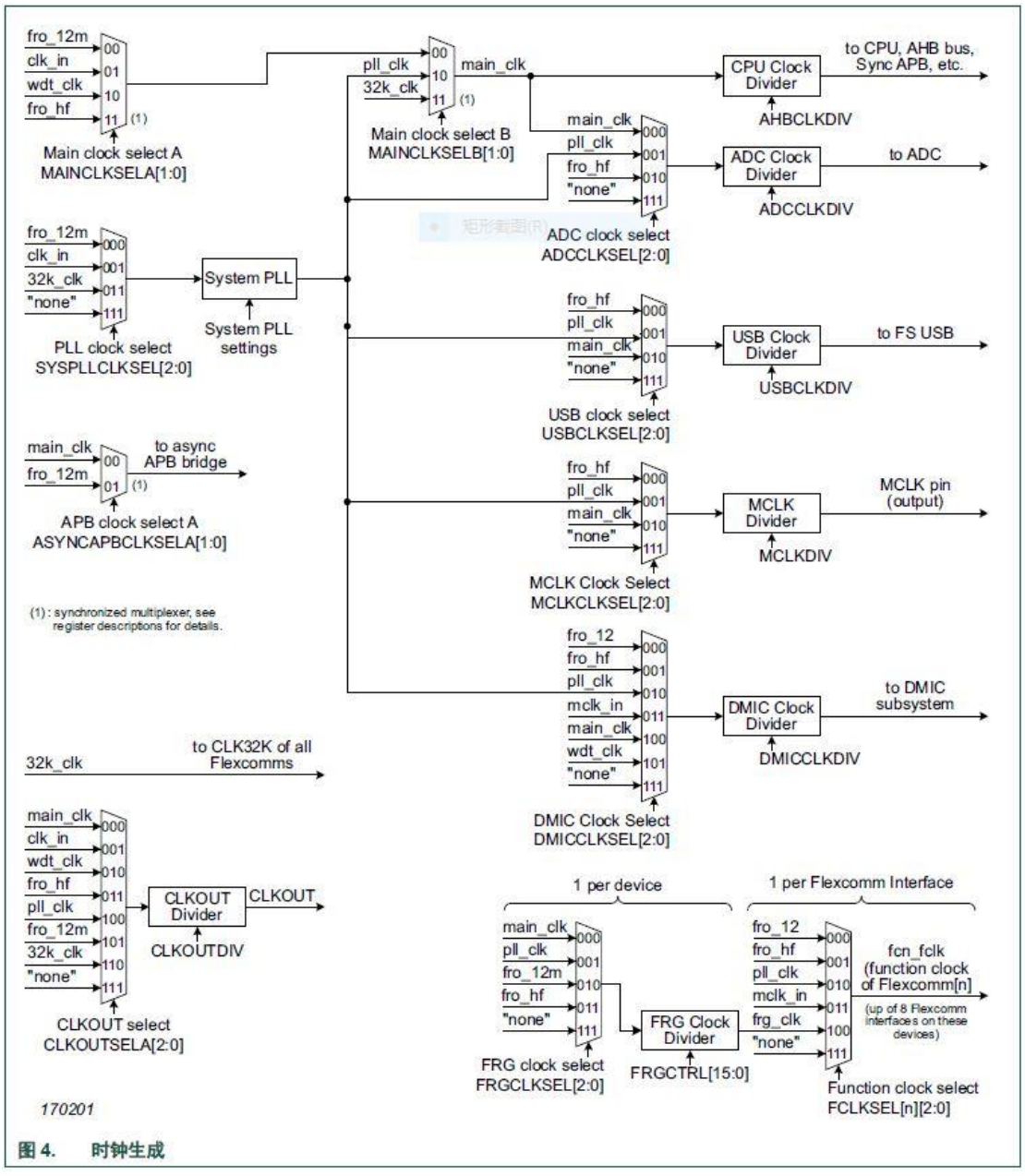


图 4. 时钟生成

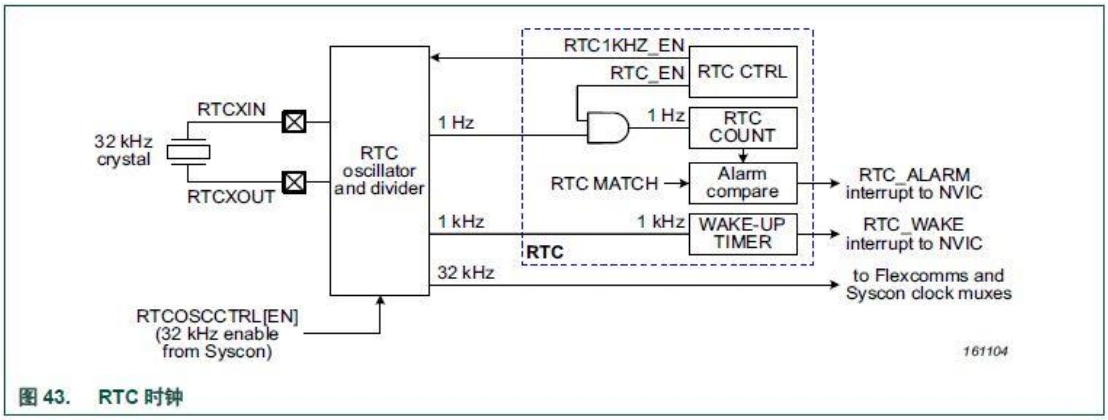


图 43. RTC 时钟

时钟域

对于每个模块来说，时钟域基本可以分为两个部分，一部分与总线同步，另一部分（如果有）则与一个决定工作或传输速度的时钟同步，无论是纯数字外设（例如 Flexcomm 和 DMIC）还是模拟外设（例如 USB 和 ADC）都存在这种双时钟域的情况。这种情况通常不需要太担心，外设内部会自动完成时钟域的同步工作的。

总线时钟开关

这个功能在前面已经提到过了，就是对每个总线上的外设时钟进行控制，对于不用修改或访问的外设可以关闭时钟以节省功耗。顺带一提，总线上还有一个复位开关，使用方法是类似的。

其它选项

LPC5411X 还包含了其它和时钟有关的选项，但有些比较隐蔽，因此在这里整理一下，做个笔记。

Flash 时钟和延时

Flash 的读取延迟通常是比较大的，对于 Flash 时钟频率较高的情况，可能需要多个周期才能访问到 Flash 的数据，因此需要根据 Flash 频率调整对应的读取延时。引入读取延时会导致每 MHz 的性能有所下降，这是由于预取的时间变长、指令执行效率降低的缘故。

HWWAKE

即硬件唤醒寄存器，用于控制深度睡眠模式（见下文“低功耗模式”）下的总线时钟状态（这里的唤醒仅针对总线时钟）。其实蛮好理解的，就是临时打开时钟进行传输而已。不过，参考手册并没有说明时钟是否能在进入深度睡眠模式前关闭，也不清楚打开 FORCEWAKE 位时从 CPU 是否可以访问在 PDSLEEPCFG0 寄存器中关闭的外设，这个可能需要进行试验才知道了。

表 94. 硬件唤醒控制寄存器（HWWAKE，主 syscon：偏移量 0x780）位描述

位	符号	说明	复位值
0	FORCEWAKE	强制外设时钟在深度睡眠模式保持启动。 为 1 时，阻止外设时钟在 CPU 进入深度睡眠模式时进入关闭状态。这可让协处理器在主 CPU 关闭时继续操作。	0x0
1	FCWAKE	唤醒 Flexcomm 接口。 为 1 时，任意 Flexcomm 接口 FIFO 达到其自身 TXLVL 指定的电平时，都会导致外设时钟在相关状态有效时被临时唤醒。	0x0
2	WAKEDMIC	唤醒数字麦克风。 为 1 时，数字麦克风输入 FIFO 达到任一通道的 TRIGLVL 指定的电平时，都会导致外设时钟在相关状态有效被临时唤醒并且。	0x0
3	WAKEDMA	唤醒 DMA。 为 1 时，DMA 繁忙会造成外设时钟在 DMA 完成前始终保持运行。这通常与位 1 和 / 或 2 配合使用，用于防止外设时钟在唤醒的原因被清除，但 DMA 完成相关活动之前关闭。	0x0
31:4	-	保留。未定义读取数据，只应写入 0。	-

时钟分频

分频的选项很容易理解，就是把高速时钟变成相关的低速时钟，除了特定需求（如控制 Flexcomm 的传输速度）之外，也可以是功耗控制的选项。0x40000300 开始的一些寄存器就是用来改变分频系数的，好好看参考手册就知道了。

SRAM 的自动时钟门控

LPC5411X 有一个比较有趣的功能，就是在 16 个总线时钟没有访问后关闭相应 SRAM 的时钟，代价是下次访问需要额外的 1 个总线时钟来重新打开 SRAM 时钟。该功能默认是使能的，按参考手册的说法，关闭该功能会使系统获得 1%到 2%的性能提升，但提升的幅度显然与代码的关系是比较大的。

电源

相对于时钟，电源的作用更好理解：上电就照常工作，断电就停止工作，此时无法保持状态。因此，我们通常只在完全不需要用到某个外设或不需要保持外设状态的时候才关闭外设对应的电源。

电源域

按参考手册的分法，电源域可以分为主电源域（CPU、总线和绝大部分外设）与始终上电的电源域（仅有 RTC 和 PMU），而我觉得前者还可再细分为数字电源域和模拟电源域（包括 SRAM、Flash、USB 等，见下文“模拟电源域”）。

数字电源域

目前没有看到可以显式关闭的选项，或许关闭总线时钟后的功耗就可以忽略不计了？不过模拟部分的功耗确实是通常更高。

模拟电源域

LPC5411X 对于模拟模块的定义可能是比较特别的，不过相关的配置选项都在 PDRUNCFG0 和 PDSLEEPRUNCFG0 中展示了（虽然我有理由相信参考手册还是藏了一些模块没有注明），其中后者用于深度睡眠模式，而前者用于其它模式（有趣的是深度掉电模式也用了前者，见下文“低功耗模式”）。

表 85. 功率配置寄存器 (PDRUNCFG0，主 syscon：偏移量 0x610）位描述

位	符号	说明	复位值
3:0	-	保留	-
4	PDEN_FRO	FRO 振荡器。0 = 已上电；1 = 已掉电。	0x0
5	-	保留	-
6	PDEN_TS	温度传感器。0 = 已上电；1 = 已掉电。	0x1
7	PDEN_BOD_RST	掉电检测复位。0 = 已上电；1 = 已掉电。	0x0
8	PDEN_BOD_INTR	掉电检测中断。0 = 已上电。1 = 已掉电。	0x1
9	-	保留	-
10	PDEN_ADC0	ADC。0 = 已上电；1 = 已掉电。	0x1
12:11	-	保留	-
13	PDEN_SRAM0	SRAM0。0 = 已上电；1 = 已掉电。	0x0
14	PDEN_SRAM1	SRAM1。0 = 已上电；1 = 已掉电。	0x0
15	PDEN_SRAM2	SRAM2。0 = 已上电；1 = 已掉电。	0x0
16	PDEN_SRAMX	SRAMX。0 = 已上电；1 = 已掉电。	0x0
17	PDEN_ROM	ROM。0 = 已上电；1 = 已掉电。	0x0
18	-	保留	-
19	PDEN_VDDA	要使 ADC 工作，必须使能 ADC 的 VDDA。另请参见位 23。0 = 已上电；1 = 已掉电。	0x1
20	PDEN_WDT_OSC	看门狗振荡器。0 = 已上电；1 = 已掉电。	0x1
21	PDEN_USB_PHY	USB 引脚接口。0 = 已上电；1 = 已掉电。	0x1
22	PDEN_SYS_PLL	系统 PLL。0 = 已上电；1 = 已掉电。	0x1
23	PDEN_VREFP	要使 ADC 工作，必须使能 ADC 的 Vrefp。另请参见位 19。0 = 已上电；1 = 已掉电。	0x1
31:24	-	保留	-

始终上电的电源域

该电源域与主电源域是分开的，为的就是让 RTC 模块能在其它模块都关闭的时候还能照常工作。当然，如果用不到 RTC 的话，也是可以将 RTC 的电源关闭的。

内部稳压器(Regulator)

LPC5411X 的参考手册一直在强调 PMU 对于低功耗的作用，不过却很少提及 PMU 的原理。在阅读电源的源码库时，我发现了内部稳压器的存在，而这个模块并没有出现在参考手册里，可以说是十分有趣了。

简单来说，内部稳压器的作用自然是为片上各个模块提供合适的电压。我们前面提到过频率对功耗的影响，因此内部稳压器需要根据系统频率进行电压输出的微调。其实这个模块并不新鲜，STM32 中也有类似的功能，只不过设置的选项更透明一些罢了。

低功耗特性

终于说到重点了。LPC5411X 关于功耗的控制选项还是比较丰富的，虽然没有 STM32L 系列这么变态（STM32L4 系列的低功耗可以有七八种状态），但是用起来要舒服得多。当然，用得好的前提是合理的配置。

低功耗模式

低功耗模式可以在两个层次上定义，即 CPU（核心）和 SoC（芯片）。

CPU

我们知道，在 Cortex-M 系列单片机中一共定义了两种低功耗模式，分别是睡眠模式和深度睡眠模式，两者的区别主要在供电上。

睡眠模式主要做的就是关闭 CPU 上运行的大部分时钟以节省功耗。当然，有些时钟还是要

保持运行的，例如 SysTick 和 NVIC。由于仅仅是关闭了时钟而没有切断电源，所以降低的主要是动态功耗部分，对静态功耗没有影响，但是唤醒速度也是最快的。需要将 SLEEPDEEP 位置为 0。
深度睡眠模式采取的策略更为激进，通过切断 CPU 的大部分电源以节省功耗。为了保持 CPU 的运行状态，一些寄存器和逻辑电路将不会被切断电源。在深度睡眠模式下，NVIC 的工作将移至 WIC 进行，后者可在前者被断电后继续保持运行，并在中断来临时唤醒整个 CPU。深度睡眠模式进一步削减了静态功耗，代价是更长的唤醒时间。需要将 SLEEPDEEP 位置为 1。

SoC
LPC5411X 的低功耗模式共有三种，分别是睡眠模式、深度睡眠模式和深度掉电模式。
睡眠模式对应 CPU 的睡眠模式，除了 CPU 的时钟被停止之外，其它外设通常保持运行，包括可以用作总线和可以作为主机运行的 DMA、USB 等外设。睡眠模式的外设设置和工作模式下没有区别。
深度睡眠模式对应 CPU 的深度睡眠模式，进一步关闭了总线和大部分数字外设的运行时钟，并可以选择性地关闭模拟外设的电源。深度睡眠模式需要配置的寄存器包括 PDSLEEPCFGx 和 STARTERx，前者用于配置深度睡眠模式下的模拟外设的电源（睡眠模式只和 PDRUNCFGx 有关），后者则像是配置 WIC（按照标准，WIC 不需要显式配置，但 STARTERx 寄存器只影响深度睡眠模式下的中断配置，此处存疑）。
深度掉电模式则是功耗最低的模式，通过关闭除 RTC 和 PMU 的电源以最大程度地降低功耗，代价是唤醒后无法恢复到先前状态，只能复位后重新执行代码。该模式的原理在参考手册上基本没怎么提及，后来在 LPCOpen 的电源管理 API 源代码中找到了线索，其实就是通过直接操作 PDRUNCFGx 寄存器的方式来关闭电源的，可谓是简单粗暴，这可能也是不提供详细说明的原因之一。

低功耗策略
所谓的低功耗策略是时钟、电源和低功耗模式的配置组合。在不同的情况下，低功耗策略是不同的。

CPU 工作时
在 CPU 需要正常工作时，显然不适合使用任何低功耗模式。要降低系统功耗，就必须在以下几个方面下手：

1. 关闭不需要使用的电源和时钟。前者可通过 PDRUNCFGx 寄存器完成，而后者则与 AHBCLKCTRLx、ASYNCAPBCLKCTRL 寄存器有关。
2. 合理选择系统各部分模块的时钟。需要时使用分频器也是不错的选择。
3. 使用内部稳压器设置合适的工作电压。可以调用 *Chip_POWER_SetVoltage*（使用 LPCOpen）或 *POWER_SetVoltageForFreq*（使用 SDK）来设置。

CPU 休息时
在 CPU 可以暂停工作时，便可以考虑进入到低功耗模式。关于低功耗模式的选择可以这样考虑：

1. 如果只需要 RTC 保持运行且不需要暂存任何数据或配置，那么深度掉电模式是不二之选。深度掉电模式只适用于需要直接关闭整个 SoC 或者 RTC 可以满足要求的情况。
2. 如果 CPU 在一段时间内没有动作，仅需要被动地接收来自外界的信号或数据，不需要进行复杂的控制和处理，那么可以尝试使用深度睡眠模式。深度睡眠模式下基本只能接收而不能处理或发送数据，对于单片机而言适合作为类似于从机的方式运行，可以配置在 FIFO 达到一定状态才唤醒 SoC，也可让 DMA 和总线在必要时保持工作。
3. 如果 CPU 在操作过程中需要等待外界信号或者数据，但是等待的时间较短（如使用 Mailbox 中的互斥量）或需要进行一些传输/发送的操作（如通过 I2S 输出音频信号），则使用睡眠模式会比较合适。

除此之外，还有一些关于降低系统功耗的建议：

1. 尽可能使用 FRO 12 MHz/48 MHz 作为深度睡眠模式下的时钟，特别是 FRO 12 MHz。根据参考手册，芯片内部已对该时钟在低功耗模式下进行了优化，调用 *Chip_POWER_SetLowPowerVoltage*（使用 LPCOpen）或 *POWER_SetLowPowerVoltageForFreq*（使用 SDK）可以有针对性地优化低功耗模式下的电源效率。
2. 合理地使用 HWWAKE 寄存器。这可以在深度睡眠模式下允许临时进行总线的数据传输操作。由于翻译的原因，中文版的参考手册说得不是很清楚，建议参考英文版的参考手册。
3. 在睡眠模式下，可以将不用的引脚至于一个固定的状态以减小功耗，但在深度睡眠模式下无效。深度睡眠模式下的 GPIO 已经断电，应该不会产生功耗。
4. 在 USB 部分看到一种说法，在深度掉电模式下必须要将 DP 和 DM 引脚进行外部上拉或下拉的方式，也就是不允许处于浮动模式。目前尚不清楚不使用 USB（PDSLEEPCFG0 的 PDEN_USB_PHY 为 1）时是否也要遵循这一模式，也不知道是否会和 USB 标准有冲突。
5. 我们知道可以在 SRAM 中执行程序。从应用笔记中可以看出，对于在 Flash 中进入深度睡眠模式和在 SRAM 中进入深度睡眠模式这两种情况，待机功耗应该是大致相同的，但是在从深度睡眠模式中恢复时后者的时间要小于前者，因此尽可能在 SRAM 中执行进入深度睡眠模式的代码。LPCOpen 和 SDK 中的电源管理 API 就是这么做的。

双核的原理和应用思路

在开始阅读本章前，需要对 LPC54114 的双核机制有一定了解。强烈建议阅读官方的 LPC54114 双核使用指南中文版，这篇文章能让读者对 LPC43XX/LPC541XX 系列双核单片机的理解更加深刻。

引入多核的原因

我们知道，多核 CPU 的组织方式可以分为两种，一种称为 SMP（对称多处理，Symmetric Multiprocessing），另一种被成为 AMP（非对称多处理，Asymmetric Multiprocessing）。

前者是目前高性能 CPU 中多核的主要使用方式，移植过 Linux 内核的同学应该有所耳闻；而后者则特别适合分配一些高实时性、低延迟或和低功耗相关的任务，典型例子是一些嵌入式 CPU（如 Rockchip RK3399）中集成的 PMU（使用 Cortex-M0 核心）。可以说，这代表了多核的两种不同思路。

SMP 的引入主要是为了突破 CPU 频率的瓶颈。在集成电路中，时钟频率通常是有限的，且功耗随时钟频率呈指数级增长，因此在时钟频率达到一定程度后，便会由于功耗和温度的迅速上升而难以进一步提升，这就是我们说的"功耗墙"和"温度墙"的来源。这种时候，就有必要部署多个 CPU，以达到"1+1>2"的效果。

而 AMP 则是解决实时性问题的一把利剑。在 ARM 的定位中，Cortex-A 系列中的 A 代表 Application，特点是高性能、高延迟，而高延迟是为了支持更多操作系统特性所做的妥协；而 Cortex-R 系列中的 R 代表 Real-time，Cortex-M 系列中的 M 代表 Microcontroller，两者对中断的响应更快，更适合处理高实时性的任务。其中，Cortex-R 系列虽然默默无闻，但其实它作为实时处理单元已被广泛应用于各种芯片，包括无线网卡、硬盘等；而 Cortex-M 系列则由于其较低的购买和开发成本，更是渗透到了电子产业的每个角落。除此之外，Cortex-M 系列单片机还具有很低的运行功耗，这使得它可被用作"始终"的处理器进行执行，一种有趣的应用就是进入到高性能 CPU 中作为电源管理单元使用。

单片机和多核

随着集成电路的发展，芯片产量不断提升、性能更加强劲，而价格却在不断降低，终于单片机也迎来了多核化的一天。Cortex-M 系列作为单片机的性能代表，自然成为了多核单片机的第一梯队。那么，为什么单片机需要多核心呢？

我们知道，单片机并不是性能驱动型的应用，因此对于多核单片机的应用主要以 AMP 为主。在这种情况下，单纯的运算量并不是瓶颈，问题在于运算时间、I/O 操作、功耗、成本等方面，这是无法通过更换更高端的 CPU 来解决的。在我看来，只要用对了地方，多核单片机一定有其用武之地。

说到多核单片机，我最喜欢举的例子就是 Sensor Hub（笑）。Sensor Hub 也就是传感器中枢的意思，即使用一个芯片管理众多的传感器。开发过传感器应用的同学都知道，传感器通常是"低速传输协议+中断"的方式与主机进行交互。而这套操作的难点有以下两点：

1. 传输协议通常是不规则的，且对总线的占用时间很长。由于各种传感器的定义互不相同，且都使用了较为简单的传输协议（如 I2C、SPI、UART 等），因此对传感器的访问通常比较复杂（比如说写入特定值后读取数据），无法使用 DMA 来自动传输数据。另外，有些协议会极大地占用系统总线和 I/O 总线，一个极端的例子是 1-Wire 协议，大家熟悉的 DS18B20 的读写时间可达 ms 级，且复杂的传输协议导致在传输过程中主机必须连续工作，无法切换至其它任务。
2. 中断的响应对实时性要求很高。有时候，我们必须在给定的时间内对中断进行处理，否则可能会出现 FIFO 溢出、数据值已改变等情况，而且也不利于快速处理传感器的数据。

因此，现在有不少产品开始使用一颗独立的单片机甚至是 FPGA 来处理传感器的数据，并

将处理后的数据发送至主机。这样做固然可以解决时间上的冲突，但还是会有以下的问题：

1. 成本。多个香炉多个鬼，一旦多用了一块芯片，反应到成本上就又是一次不小的改变。在做产品时不仅要考虑芯片本身的成本，还要考虑使用这个芯片给软件、外围电路、测试等方面带来的额外成本，这往往是决定产品生死的重点。
2. 性能。有些传感器需要大量的运算资源进行处理，典型的应用有图像、音频和运动传感器等，为了保证运算的实时性，必须使用不同的运算资源分别进行传感器的读写和数据处理，否则会对效率产生很大影响。
3. 功耗。前面提到数据的读写和处理是分开进行的，但是有些数据处理并不应该交给主机完成，而是就地进行计算，这主要是考虑到功耗的影响。举个例子，如果我们使用单片机和运动传感器开发了一个计步器，如果能在单片机内将运动传感器所记录的数据融合并计算出是否正在运动，便可避免主机参与到数据处理的过程中来，从而可将主机置于低功耗状态，直到接收到来自计步器的中断再唤醒主机即可。

因此，对于 Sensor Hub，我们发现一个核心的单片机来说似乎有些不够用了。那就再加一个核心？这也就是我们容易想到的双核方案了。不过，在多核单片机出来之前，更流行的 AMP 是 CPU+DSP 的方式，这种方式更多地出现在需要高性能的场合，通过 DSP 来弥补 CPU 在计算密集型应用的不足之处。多核单片机的性能自然是比不过前者的，但胜在成本、功耗等方面，而这对于传感器密集型应用往往是决定性的要素。现在，市面上已有的多核单片机主要包括 NXP LPC43xx 系列、LPC541xx 系列和 Cypress 的 PSOC 6 系列，这些单片机都在各自领域发挥着重要的作用。

LPC541xx 的双核机制

前面我们已经介绍过了 LPC541xx 的双核特性。那么，这两个核心应该如何使用，才能取长补短，充分发挥其作用呢？以下几个方面是我们会考虑的内容。

主核与从核

对主从核的分配要根据需要来选择。通常的做法是将 Cortex-M4F 作为主核，而 Cortex-M0+ 则作为从核，这适合于更注重性能的情况。不过，要是项目并不复杂且通常有较低的 CPU 占用率，那么将 Cortex-M0+作为主核也是可以的，Cortex-M4F 可以在有必要的时候再唤醒，以执行 Cortex-M0+无法胜任的工作。

存储空间

对存储空间的分配是搭建双核工程时不得不考虑的内容，这包括 Flash 和 SRAM 两部分。我们知道，同时访问同一块空间时，总线会根据仲裁结果决定让其中一个主机的操作首先进行，而另一个主机则会等待直到总线分配下一个主机，这会影响系统的速度。因此，除了两个核心共用的部分代码和数据，其它代码和数据理应放在不同的 Flash 和 SRAM 中。当然，放在哪个地方也是有一些制约因素的，主要是 Flash 在高速时钟的执行效率和 SRAMX 的总线连接这两点需要慎重考虑。

低功耗特性

在 LPC541xx 中，有两个关于核心选择的选项，一个是 MASTERCPU，用来控制主从核，主核无法被复位或断开时钟，但是从核可以；另一个是 POWERCPU，用来选择控制电源的核

心，被选中的核心在进入深度睡眠模式时可以关闭设备上的各个模块，而另一个核心的睡眠模式则只对本身有效。一般来说，这两个选项应该要相同，因为从核可被看作是主核的外设，主核对从核的运行状态进行控制要更为合理。其实，低功耗特性的选择就是主从核选择的延伸。

双核间交互

对于有些应用，双核之间并不总是各自为政，而是存在相互之间的交流。LPC541xx 相对于 LPC43xx 而言添加了 Mailbox 模块，支持双核中断和互斥量（即 mutex）。

双核中断可以用中断的方式提醒另一个核心。Mailbox 对于每个核心支持多达 32 位的信号，只要有任意位信号为 1，则会唤醒相应的核心，使用起来非常简单，可惜 32 位还是有些不够用。

互斥量则是 ARMv7-M 指令集中对于排他访问的延伸。如果说排他访问的应用范围主要是核内异常和中断的检测，那么 Mailbox 中的互斥量则是核间同时访问的检测。ARM 使用的排他访问模型通常是读-写-判断三步走的，读操作后会设置相关的信号量，写操作则会检测信号量是否改变，通过对写操作的结果来判断排他访问是否成功。Mailbox 对于互斥量的处理则是遵循读-清空-判断-写的方式，虽然与排他访问模型略有不同，但都能达到相同的效果。虽然互斥量只有一位，但该位信号可定义为是否处于空闲状态，通过允许其中一个核心排他地控制该位状态的方式来表示对共享资源的占有。

开发板简介

这块开发板我就不打算再做太多介绍了，一方面是评测的帖子已经够多了，另一方面则是因为开发板上的资源我只使用到了数字麦克风。总的来说，开发板的外设还是相对比较丰富的，特别适合作为语音相关任务的实现；但是，这些外设占用了不少引脚，导致我在外接传感器和功耗评测上遇到了一定困难，希望以后开发板的设计能够更多地考虑到这些因素。

第六部分 赛题与设计思路

本次的赛题是与双核设计有关的内容，因此如何利用好双核就成为了重点。另外，SoC 上的外设也是值得考虑的对象。我们先来看看有什么值得利用的资源吧。

计划使用的资源

双核

我的设计其实是 Sensor Hub 的一种延伸。考虑到传感器通常具有较低的时钟频率，12 MHz 的 Cortex-M0+就可以很好地满足要求，而 Cortex-M4F 仅在处理语音识别等需要大量运算的场合才用得到，因此 Cortex-M0+作为主核是更为合适的。这种结构其实类似于 MCU+DSP，因其平均功耗较低而十分适合传感器应用。我们来分配一下双核各自的任务。

Cortex-M0+：

1. 处理各种总线的信号传输。特别是对于 I2C 这样的多从机总线，务必要保证同时只有一个任务在执行，这可以通过为各个总线创建并执行唯一的 handler 来实现。
2. 为每个传感器对应的任务分配独立的进程，并允许其对数据进行简单处理。不同的传感器可能会同时处理数据，在可行的情况下应该允许进程穿插执行，这就需要操作系统的协调了。
3. 在任务有必要时可以发起对 Cortex-M4F 的请求，并且可以等待处理结果以决定下一步的操作。请求是通过 Mailbox 完成的，在这个过程中一定要小心处理临界区可能带来的问题。
4. 控制各个模块的运行和设备的低功耗模式。由于 Cortex-M0+是主核，因此有义务对于片上各个模块的运行状态进行控制，在必要时可以开启或关闭从核。在关闭从核时钟时，一定要确保从核已进入睡眠状态，否则可能会导致程序执行出现问题。

Cortex-M4F：

1. 处理来自 Cortex-M0+的请求。由于 Cortex-M4F 不负责外设管理，因此只需要接收来自 Cortex-M0+的中断请求，并在运算完成后作出回应。
2. 如有必要，可以使用操作系统来协调多个任务的执行，以降低简单任务的等待时间。

存储空间

根据我们的测算，需要在 Cortex-M4F 上执行的程序不会超过 SRAMX 的容量，同时 Cortex-M0+也应在 SRAM 上执行以降低低功耗模式下的功耗和唤醒的等待时间，这就需要对空间有合理安排才行。

Flash：

1. 首先，两个核心的程序一开始都要放在 Flash，然后再分别复制到不同内存。但是呢，由于语音识别中需要使用大量的参数，这些参数无法在 SRAM 中储存，因此应该将参数存到 Flash 中，并在开始语音识别前打开 Flash 的开关，这样就满足使用的需要。
2. 为了方便以后修改参数，我们有必要将其放在给定的位置，并在固定的位置设置合适的标记，这样便可调用 IAP API 对参数进行运行时的修改，即控制了成本又方便调用。

SRAM：

1. Cortex-M4F 的程序使用 SRAMX 作为代码空间。这是由 SRAMX 的连接决定的，没有别的选择。
2. Cortex-M4F 的程序使用 SRAM0 作为数据空间。Cortex-M4F 一共有三条总线，将代码和数据分开可以提高其运行效率。
3. Cortex-M0+的程序使用 SRAM1 作为代码和数据空间。对于 Cortex-M0+，64 KB 应该足够使用了，而且单总线的设计使得这样的设计并不会影响其效率。
4. Cortex-M4F 和 Cortex-M0+共用 SRAM2 作为共享的数据空间。数据交换可以放在一块单独的 SRAM 空间中，以防止在处理非共享数据时另一个核心的访问对性能造成的影响。

DMIC

由于我们的应用是语音识别，因此 DMIC 及其内部的 HWVAD 应当总是处于运行状态，DMIC 可以在深度睡眠模式下运行的特性正好符合我们的需要。考虑到 FRO 振荡器的功耗较高，我们应允许在低功耗模式下使用低精度的看门狗振荡器，并在发生语音活动后更改为更高精度的 FRO 振荡器，同时开启 DMIC 的 DMA 传输。

DMA

DMA 的用法就不再多说了，需要注意的是 DMA 本身可以在深度睡眠模式下运行，但只有在其它外设也可以在深度睡眠模式下运行的情况下才可以这么做。

Flexcomm

至于系统中的 I2C、SPI、UART 等总线，可以归到 Flexcomm 一类中。由于运行在主机模式，这些总线都需要 CPU 的参与才能开始传输，并且在传输过程中不能进入到深度睡眠模式，这是令人比较遗憾的一点，不过也在一定程度上简化了设计。除了调用 DMA 的时机需要仔细考虑之外，其它应该都是比较简单的。

Mailbox

Mailbox 用来在双核间传递信息，一共可以传递 32 位。考虑到 SRAM 的位宽远小于 32 位，这里我们使用高八位作为信息的识别码（表明是哪种类型的信息），其余位数则用来表示指针相对于 0x20000000 的偏移量或其它只需用到 24 位的数据。考虑到信息传递的有效性，我们规定在一方使用 Mailbox 发送信息时，另一方在收到信息后必须清空相应寄存器，发送方会等待相应寄存器清空后才认为信息传递完毕。

RTC

这个模块其实不是必须使用的，但是如果在操作系统中需要用到定时一类的任务，且允许在等待过程中进入睡眠模式，则可以使用 RTC 来计算睡眠的时间。在睡眠模式下，Systick Timer 可能停止执行，因此 RTX5 支持自定义空闲进程函数，以使用其它模块来代替 Systick，此时操作系统的内核可以暂时挂起，在被唤醒时更新 Systick Timer 并恢复内核执行。

程序流程

由于两个内核在同时运行，因此程序流程可能相对复杂且互有联系。

程序可以分为以下三个阶段。

第一阶段：初始化

除了 CPU 各种的初始化之外，大部分初始化都由 Cortex-M4F 完成，这一方面是由于 Cortex-M4F 在上电时总是作为主核运行，另一方面也考虑到了 Cortex-M4F 在效率和代码量的优势。

不考虑 CPU 内部寄存器和 PPB 总线上的 SCB，初始化过程可以分为以下几步：

1. 设置初始状态的电源、时钟和引脚状态。这一步骤最好在一开始就执行，目的是降低系统功耗，并且提升程序的执行速度。
2. 将打算在 SRAMX 上执行的程序复制到对应位置。这个过程通常已由链接器准备好相关代码。
3. 初始化相关外设和中断。这个没啥好说的，该做啥就做啥呗。
4. 将 Cortex-M0+要执行的程序和数据复制到对应位置，设置好从核的栈地址和复位向量，并复位 Cortex-M0+。在复位完成后，Cortex-M0+应该开始执行。由于复制的数据量并不算小，我们考虑使用 DMA 来复制，下同。
5. Cortex-M4F 跳转到 SRAMX 中执行，将电源控制核心设为 Cortex-M0+，并进入到低功耗模式中，此时 Cortex-M4F 的初始化过程结束。
6. Cortex-M0+从 Flash 启动，并跳转到 SRAM2 中继续执行。同样地，Cortex-M0+也要设置自己的中断。
7. Cortex-M0+将自己设为主核，并且在等到 Cortex-M4F 进入到低功耗模式后关闭其时钟。在这一步可以选择将系统时钟调低以降低功耗，但一定要记得更新 System Timer。
8. Cortex-M0+初始化操作系统，打开中断开关，然后启动操作系统。如果没有活跃的任务，Cortex-M0+也将进入低功耗模式，此时 Cortex-M4F 的初始化过程结束。

第二阶段：主核单独运行

这一过程主要是不涉及从核时的运行，套路一般都是这样的：

1. 系统被某个引脚中断唤醒，表明某个外设有情况。在相应引脚的 ISR 中，可以通过信号量的方式唤醒相关进程。
2. 在退出中断后，主核进入到相应进程中开始执行。

3. 如果有使用 DMA 的需求，则需在代码中调用，并且通过信号量等待 DMA 通知传输完成。
4. 如果没有其它任务，则在进程挂起后进入到低功耗模式中。

当然，这里还需考虑将 VAD 作为一个单独的进程来运行，目的是不管在哪个进程中，都可以在 HWVAD 中断到来时切换到 VAD 进程，进而完成语音识别的任务。

第三阶段：主核和从核同时运行

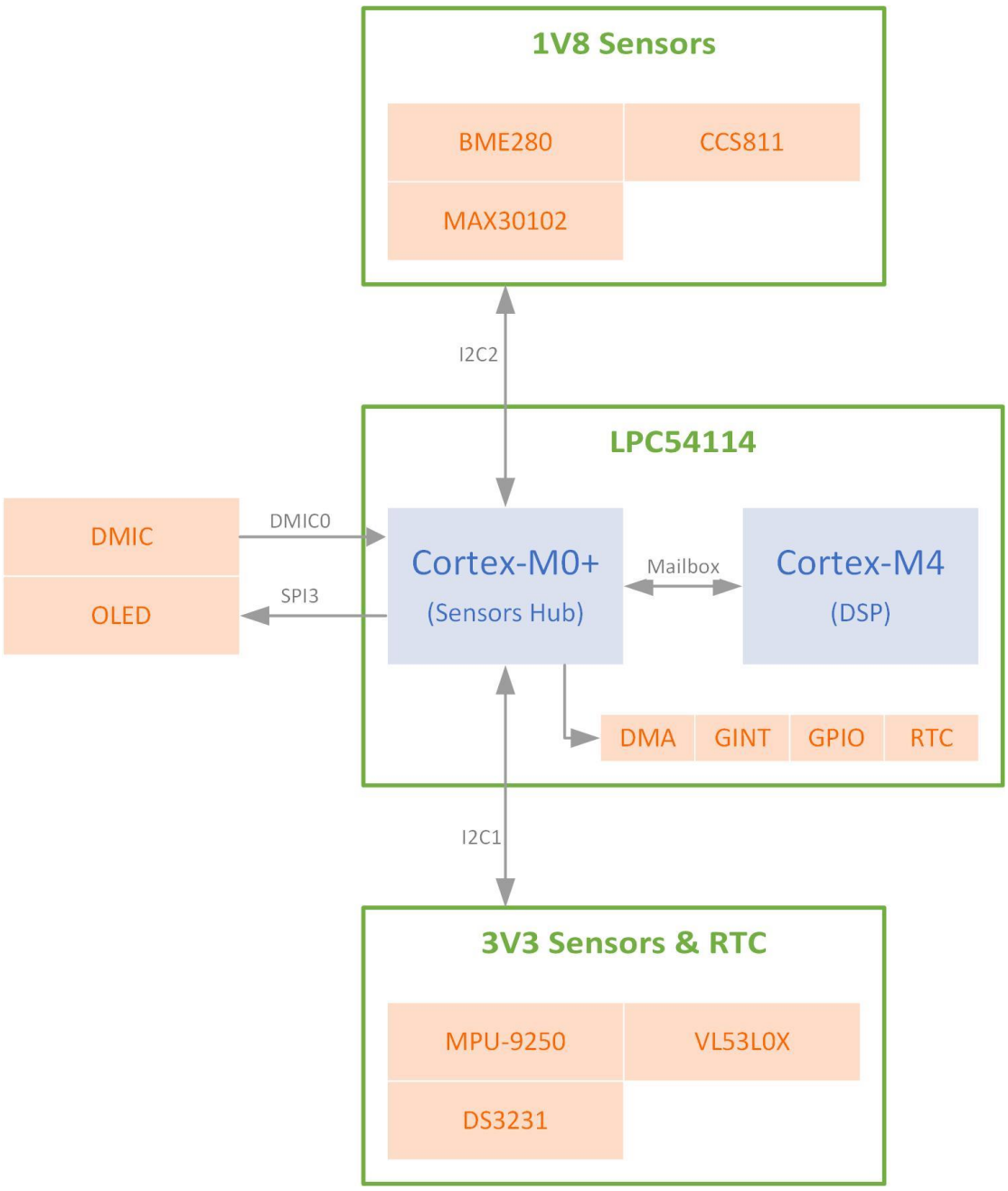
一旦有需要从核运行的场合，我们就需要将其唤醒。具体操作如下：

1. 确认特定中断有效后，将主时钟的频率调高，使能从核的时钟，并通过 Mailbox 将其唤醒。在此之后，主核可以选择继续处理相关事务或是进入睡眠模式，但一定要确保不关闭总线时钟，以免影响从核的运行。
2. 从核响应来自主核的中断，并根据其传递的参数完成相应操作。
3. 在主核分配的任务完成后，从核通过 Mailbox 通知主核。如果没有活跃的任务，从核将进入到低功耗模式。
4. 主核接收来自从核的信息，并在从核空闲时等待其进入睡眠模式后关闭其时钟，之后调低主时钟频率。
5. 如果没有其它任务，则进入到低功耗模式中。

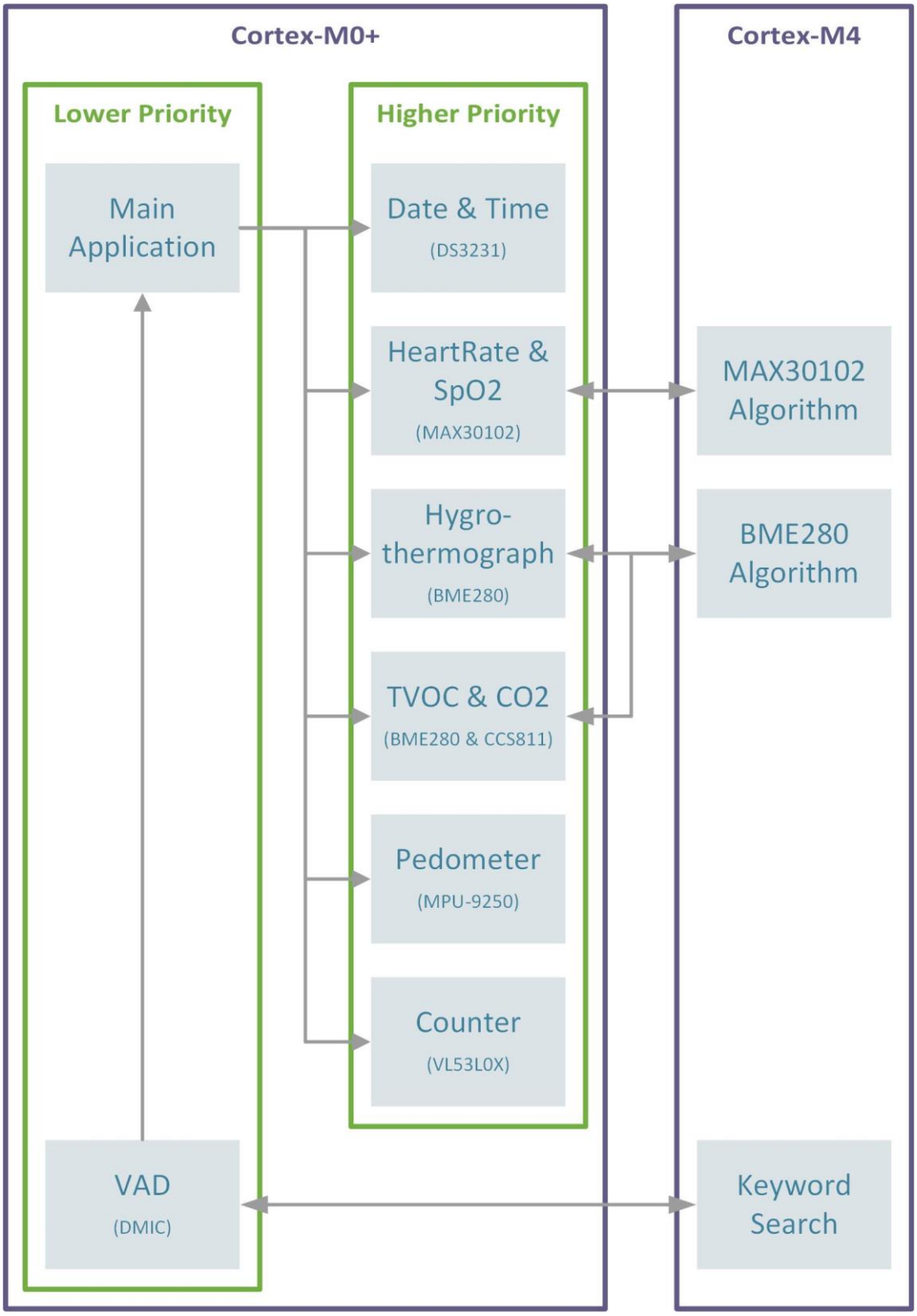
需要注意的是，从核的运行会使用到 Flash（因为参数量太大，无法全部复制到 SRAM 中），因此需要在从核运行时打开 Flash。

至于程序在运行阶段的具体流程，涉及到 RTOS 的调度，这里不再赘述，相信看了上一部分内容的读者应该可以理解。

硬件和软件架构图



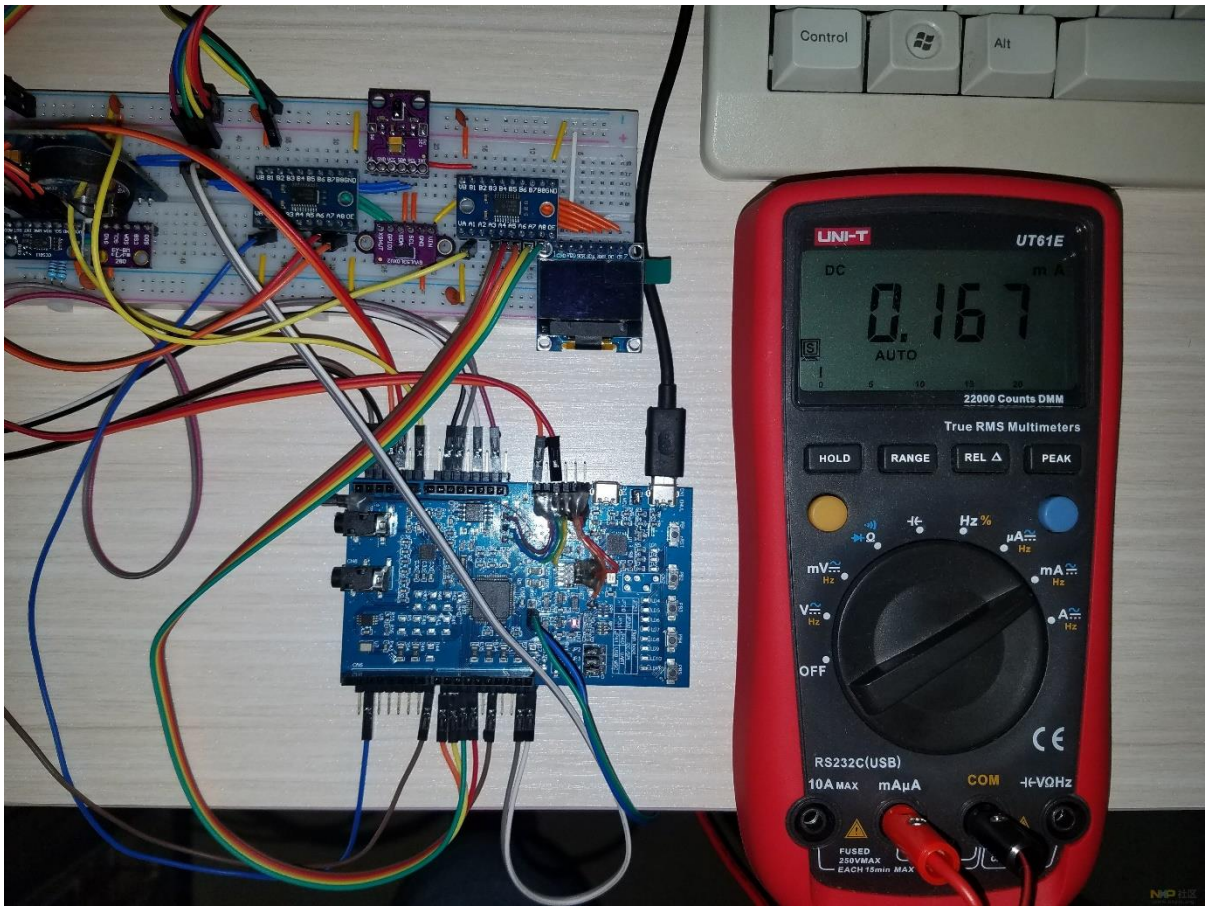
这里的硬件架构图忽略了总线以外的 I/O 连接，只显示了几条重要的总线。Cortex-M0+作为 Sensors Hub 使用，管理各个传感器，而 Cortex-M4F 则作为纯粹的 DSP 使用，不与外设进行交互。



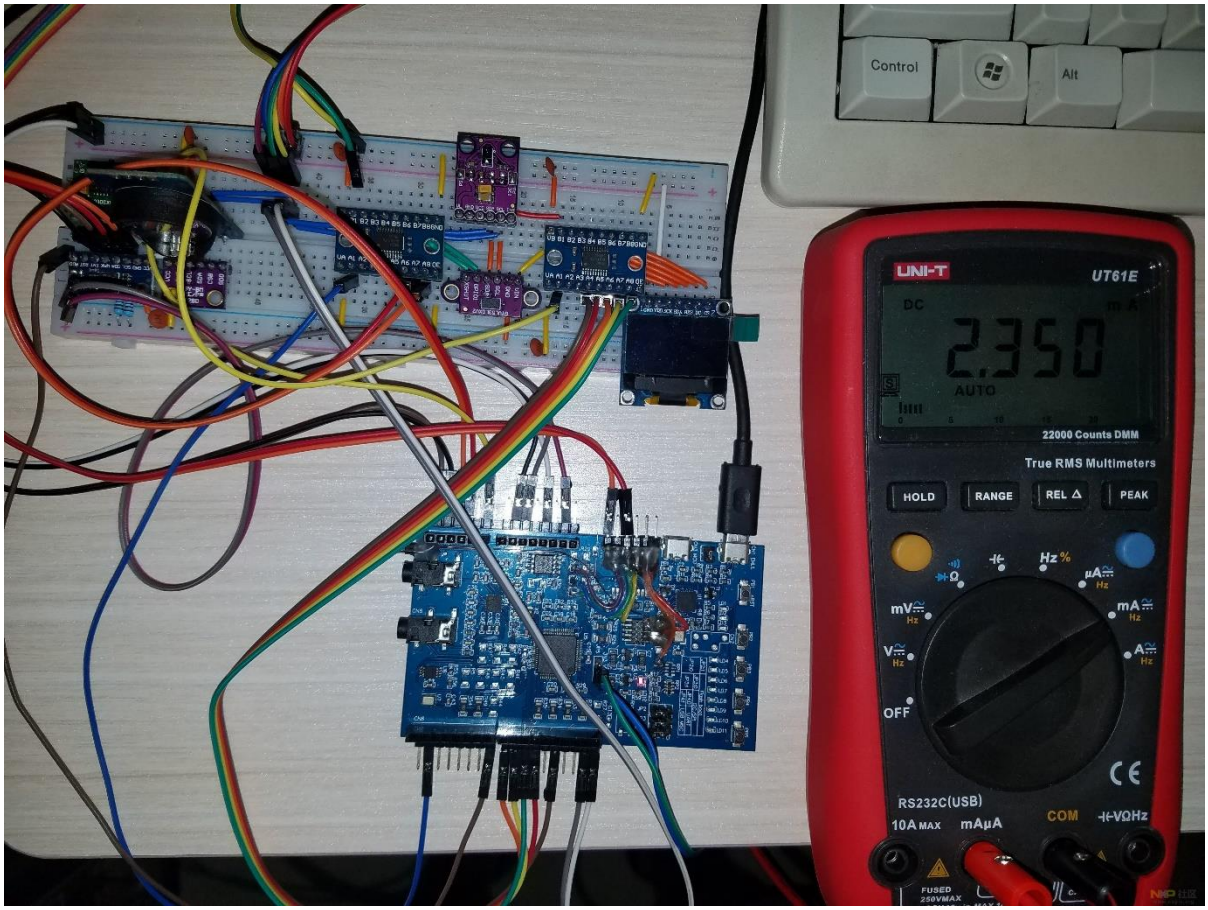
软件架构图则更多地表达了进程之间的依赖关系。可以看到，在 Cortex-M0+一侧，主进程和 VAD 进程在较低的优先级，而一些具体的功能进程则在较高的优先级，这样做的好处我们已经在前面说过了。在 Cortex-M4F 一侧，进程是和 Cortex-M0+一侧的某些进程对应的，这需要在 Mailbox 中断内进行处理。

功耗测量

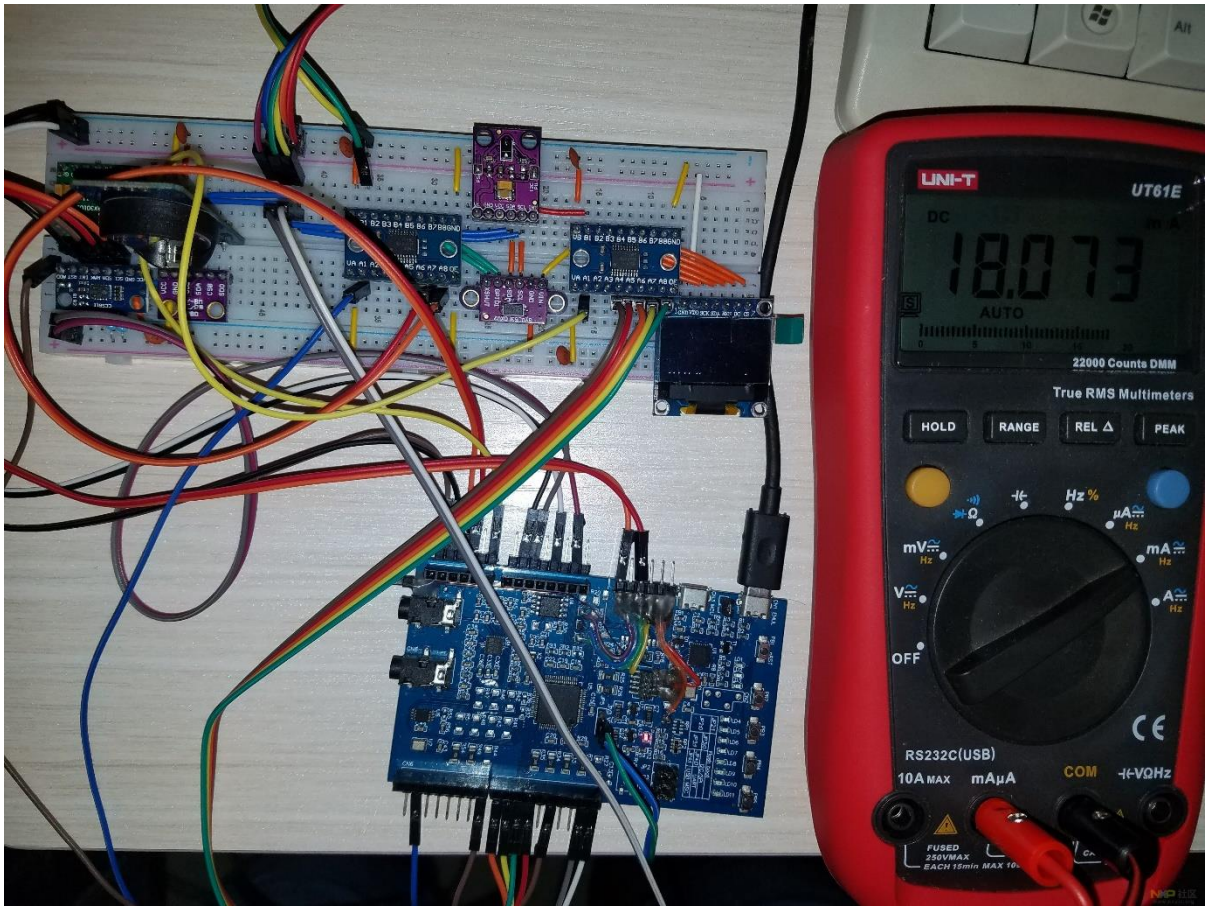
由于外接了不少传感器，因此进行准确的功耗测量是比较困难的。这里我们只测量三种情况，分别是进入深度睡眠模式时、只有主核运行时（系统时钟为 12 MHz）和主从核一起运行时（系统时钟为 96 MHz），测量时断开外设的连接，MCU 电压为 1.8 V。



进入深度睡眠模式时，此时只有 HWVAD、看门狗振荡器和数字麦克风在运行。



只有主核运行时，由于不需要进行复杂的处理，系统时钟可以运行在 12 MHz 以节省功耗。



主核与从核一起运行时，为了尽快处理完数据，系统时钟应当运行在 96 MHz。

由于任务不同,运行时间也不同,要衡量各个核心消耗的电流是比较困难的,不过作为参考，数据手册上注明了在 96 MHz 下 Cortex-M4F 的功耗为 **9.9 mA**,Cortex-M0+的功耗为 **8.0 mA**，这说明同频率下两个核心消耗的功耗是差不多的，且与频率成正比。当然，还要考虑到执行时间对功耗的影响，在有些情况下 Cortex-M4F 可以用更快的执行速度来弥补功耗的差距。

通过简单的计算，深度睡眠模式下的功耗仅为全速运行时的功耗的 **0.9%**，即使是在只有简单传感器操作时，功耗也只有全速运行的 **13%**，这足以弥补多核带来的额外功耗。因此可以说，这个项目利用双核可以达到非常好的低功耗效果。

第七部分 总结

由于时间的原因，有部分功能并没有实现（主要是 Cortex-M4F 在 SRAMX 上运行这一块，由于避免了 Flash 的延迟，这样可以加快运行的速度），但是演示还是可以达到预期效果的。一个人用一个多月的时间来开发一个完整的项目还是有些过分（笑），不过在这个过程中我也学到了很多。希望以后多多举办这样的比赛，也希望 NXP 可以推出更多实用的单片机，为我们的创意提供展示的平台。