VOIP 中 G.723.1 语音编码算法的 DSP 实现

姓　　名：李 章 林

年　　级：2001 级

专　　业：通信与信息系统

研究方向：无线通信

指导老师：吴岳教授

二零零四年 五月

摘要

本文描述了 G.723.1 双速率语音编码（信源部分）的原理和其在 C54xDSP 上实现的过程和方法。第一章为原理部分，根据 G.723.1 浮点程序详细、完整地描述了 G.723.1 信源编码的实现过程，它比 ITU-T 的原理描述更具体，比直接读 C 代码更容易理解，对研究 G.723.1 算法有一定的参考价值。第二章为实现过程部分，讨论了纯 C 代码在 54xDSP 上运行速度慢的原因，在讨论了几种优化方法后，采用了从关键代码入手的全汇编优化方案，结果证明采用该方法的优化结果和预期优化结果基本符合。优化结果基本达到了商用 G.723.1 算法的优化程度。该部分内容对移植类似 C 代码算法到 DSP 有一定参考价值。第三章为 54xDSP 汇编优化技术部分，讨论了以下内容：双字数据偶定位、函数堆栈结构设计、组合条件优化实现、硬件滤波器结构、整数小数除法和求余、OVERLY 技术、数据区堆栈等。具有创新性的技术和思想有：部分循环展开技术、并行指令使用技术、合理利用指针增减、AR0 作为循环次数、码本搜索优化实现、多位数移位的实现、由内自外的编程顺序、使用 PSHM 分配 ARx、使用硬件结构实现自增量求余等。该部分最后指出了一些编程容易出错的地方。最后一章算法应用部分介绍了两个实例：在实验箱上的实现实时语音播放和在 PC 机上的 IPPhone，主要讲述了 FIFO 环形缓冲区在语音实时采集播放中的应用。

关键字：G.723.1, 54xDSP, 优化技术, 语音编码

ABSTRCT

The thesis is about the theory of Dual Rate Speech Coding, G.723.1 (source coding part), and the procedure and method in its implementation on C54xDSP platform. The first chapter is theory description part, and we fully described the realization procedure of G.723.1 in detail, based on the floating point G.723.1 C source code. The description is more detail than the one given by ITU-T, and is easier to understand than reading the C code directly. So it may be useful in studying G.723.1. In the second chapter, we discussed the reason that causes the pure C code running so slowly. After discussing on some optimization methods, we adopted the method: fully using assembly language starting from the key code. It is proved that the optimization result basically coincide with the one predicted by this method. The result basically reached the optimization degree of commercial G.723.1 algorithm. This part may be somewhat useful as a reference when transferring similar C code algorithm to DSP. The third chapter is on 54xDSP assembly language optimization techniques, and we discussed The Even Alignment of Dual Word Data, Design of the Function Stack Framework, The Realization of Combined Conditions, Hardware Filter Structure, Integral and Fractional Division and Residue Modulus, OVERLY techniques, Stack for Data Segment etc. Some of the innovative techniques and idea are The Techniques of Partly Expanding Rotation, The Techniques in Using Parallel Instructions, The Proper Utilization of Increase or Decrease of Pointers, Use AR0 As a Rotation Counter, Optimum Realization of Codebook Searching, Realization of Extensive Shifting, From Inner To Outer Code Writing Order, Assignment of ARx Use PSHM, Hardware Realization of Modulating Self-Increase Value etc. In the end of this part, we gave some cases where mistakes may be made. In the last chapter, two examples, Realization of real-time speech recoding and playing on experiment box and IPPhone program running on PC, were given. Here, we mainly introduced the application of circular FIFO buffer in real-time speech recoding and playing.

Key words: G.723.1, 54xDSP, optimization techniques, speech coding.

目录

引言 1
第一章 G.723.1 算法原理 3
1.1 G.723.1 编解码系统整体分析 3
1.1.1 G.723.1 编码器和解码器外部接口特性 3
1.1.2 G.723.1 的编码延时 3
1.1.3 G.723.1 编码流程图 5
1.2 编码器 6
1.2.1 编码器的初始化 6
1.2.2 分帧(Framer) 7
1.2.3 高通滤波 7
1.2.4 LPC 参数的获取 8
1.2.5 LSP 量化器 10
1.2.6 数据移位 15
1.2.7 共振峰感觉加权滤波（Formant perceptual weighting filter） 15
1.2.8 开环基音预测（Pitch Estimate） 16
1.2.9 谐波噪声滤波器(Harmonic noise shaping) 16
1.2.10 LSP 解码 18
1.2.11 LSP 插值 20
1.2.12 LSP 参数至 LPC 参数的转化 20
1.2.13 联合滤波器的冲击响应 h(n)（combined filter） 22
1.2.14 联合滤波器的零输入响应的计算 24
1.2.15 自适应码本激励 25
1.2.16 固定码本激励 30
1.2.17 联合滤波器状态的更新 44
1.2.18 发送信息打包 44
1.3 解码器 49
1.3.1 解码器的初始化 49
1.3.2 接收信息解包 49
1.3.3 LSP 解码 49
1.3.4 LSP 插值和 LSP 参数至 LPC 参数的转化 49
1.3.5 当前帧的激励信号 e(n)的计算 49
1.3.6 基音后滤波参数的计算(如果 UsePf 等于 1) 50
1.3.7 新计算出来的激励信号的钳位 53
1.3.8 基音后滤波（如果 UsePf 等于 1） 53
1.3.9 LPC 合成滤波 53
1.3.10 共振峰后滤波（如果 UsePf 等于 1） 53
1.3.11 增益调节单元（如果 UsePf 等于 1） 55
1.3.12 语音信号的浮点表示到定点表示的转化 55
1.4 静音压缩方案（Annex A:SILENCE COMPRESSION SCHEME） 56
1.4.1 语音活动检测－－VAD(Voice Activity Detector) 56

1.4.2 CNG 算法 60
1.4.3 出错帧处理 67
第二章 G.723.1 算法在 54xDSP 上的实时实现 69
2.1 ITU-T 的 G.723.1 标准 C 代码算法的使用 69
2.1.1 ITU-T 的 G.723.1 算法标准及其 Annex 的内容 69
2.1.2 ITU-T G.723.1 标准算法 C 代码的编译运行 69
2.1.3 ITU-T G.723.1 算法程序的命令行参数 70
2.1.4 程序的正确性的验证及测试序列的使用 70
2.1.5 用户程序调用 G.723.1 核心算法的方法 74
2.2 纯 C 代码在 54xDSP 上的移植 78
2.2.1 DSP 程序读写 PC 机上的文件 79
2.2.2 为测试测试序列的程序修改 80
2.2.3 G.723.1 程序的存储空间分配 82
2.2.4 编译连接中的问题 85
2.2.5 G.723.1 的 DSP 程序运行正确性的验证 85
2.2.6 G.723.1 的 DSP 程序运行的速度 85
2.3 代码优化方法和过程 87
2.3.1 代码运行速度慢的原因 87
2.3.2 优化方法的讨论和尝试 88
2.3.3 从关键代码入手的全汇编优化方案 89
2.4 代码优化结果 95
第三章 G.723.1 算法中 C54xDSP 汇编程序优化技术 99
3.1 DSP 中的定点数表示法 99
3.2 54x 汇编语言的相关知识 99
3.2.1 运算标志位及在 G.723.1 中的设置 99
3.2.2 54xDSP 数据寻址方式的使用 101
3.2.3 流水线冲突及其解决方案 101
3.3 汇编程序的整体的考虑 102
3.3.4 伪指令在 G.723.1 中的使用 102
3.3.5 G.723.1 中双字数据的偶定位 103
3.3.6 函数参数传递及其堆栈设置 103
3.3.7 G.723.1 数据区设置 107
3.4 G.723.1 程序主要优化技术 108
3.4.1 部分循环展开技术 108
3.4.2 并行指令使用技术 110
3.4.3 合理利用指针增减 112
3.4.4 AR0 作为循环次数 114
3.4.5 码本搜索优化实现 115
3.4.6 组合条件的优化实现 118
3.4.7 修改程序以适应汇编 120
3.4.8 用硬件滤波器结构 120
3.4.9 通过堆栈的返回值传递方法 121
3.4.10 循环寻址结构的使用 121
3.5 G.723.1 中基本操作的实现 121

3.5.1 条件转移结构的比较 121
3.5.2 各种循环结构的比较 122
3.5.3 整数小数除法和求余 122
3.5.4 双字数据乘单字数据 124
3.5.5 单字运算操作 125
3.5.6 多位数移位和快速移位 125
3.5.7 数据拷贝 126
3.5.8 其它操作的快速实现 126
3.6 G.723.1 中代码优化的思路 126
3.6.1 提高程序速度的指导思想 126
3.6.2 节省代码量的方法 128
3.6.3 节省数据空间的方法 128
3.7 移植过程中易出错的地方 128
3.8 DSP 的 BUG 及解决方法 129
第四章 G.723.1 算法应用实例 131
4.1 G.723.1 在实验箱上实现实时语音播放 131
4.1.1 FIFO 数据缓冲区和程序结构 131
4.2 G.723.1 在 PC 机上实现 IPPhone 程序 132
4.3 G.723.1 的 DSP 嵌入式 IPPhone 的构思 133
结论 135
致谢 ..... **错误！未定义书签。**
参考文献 139

引言

VOIP(voice over IP)是当前的热门技术之一。实现 VOIP 可选择的协议有：H.323 协议组和 SIP(Session Initiation Protocol) 协议等。无论采用那种标准，语音压缩都是 VOIP 中的重要部分，图 1 是 H.323 协议组和 OSI 模型的关系[①]，从图中可知语音编码在 VOIP 中所处的地位。VOIP 语音编码首选方案是：G.723.1 和 G.729。

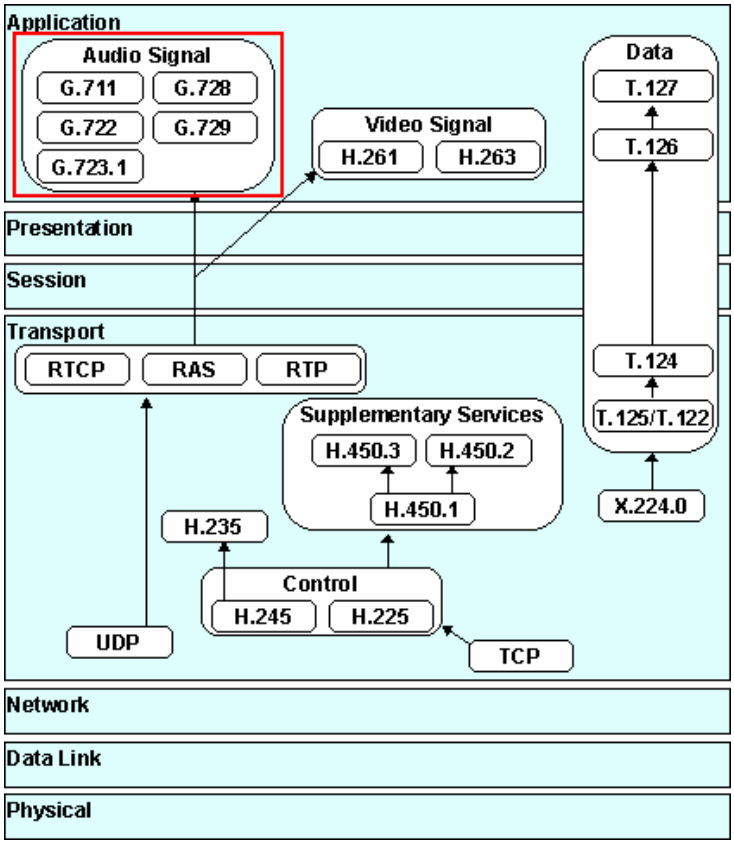

图 1. H.323 协议组和 OSI 模型的关系

目前，以 PC 机作为 VOIP 终端的应用方式已经有广泛的应用，以嵌入式系统作为 VOIP 终端的应用方式由于成本较高还没有得到广泛的应用，目前嵌入式方式应用较多的是企业 IP 电话机（enterprise IP phone）。嵌入式 VOIP 终端成本较高的一个原因是，VOIP 相关软件复杂，运算量大，特别是语音压缩算法，需要较大的运算量，为了运行复杂的软件就需要高成本的硬件支持。优化语音算法，就使得能够在低速，RAM 空间较小的低成本 DSP 上实现语音算法，从而降低成本。目前，已有商用的针对各类 DSP（包括 54xDSP）的 G.723.1 算法，但是它们都不是免费的，所以开发该算法可以获得一定程度的自主产权的 G.723.1 54xDSP 算法，为以后嵌入式 VOIP 设计做好准备。

另外已经有三届的师兄师姐在本论文中做过工作，能够完成该课题，能够使得他们的工作能够在实际中得到应用。另外从学术上讲，本课题的意义和目的是：

1、通过分析 G.723.1 算法的原理，完成 G.723.1 原理的详细描述文档。为进一步研究 G.723.1 算法和研究类似语音压缩算法提供参考。

2、研究 C 代码算法到 54xDSP 的通用步骤和方法。由于 G.723.1 算法是一个典型的 DSP 算法，研究它的 C 代码到 54xDSP 的移植，可研究 C 算法到 DSP 的移植的通用方法，这对移植类似 C 算法将有指导作用。

3、研究高效 54xDSP 汇编的相关技术。G.723.1 算法需要高效的 C54DSP 汇编技术。研究它的实现，同时可研究 C54DSP 高效汇编技术，从中得到的优化技术，可作为编写高效 54xDSP 汇编的参考。

我们首先通过阅读 G.723.1 浮点 C 代码从总体上把握 G.723.1 的算法实现原理，为优化做好准备，然后在 DSP 实验箱上进行代码优化。

第一章 G.723.1 算法原理

G.723.1 是 ITU 指定的标准，它有两种速率：5.3Kbit/s 和 6.3Kbit/s，5.3Kbit/s 和 6.3Kbit/s 能够在传输过程中动态切换，输入语音是 8KHz 的 16bit 线性 PCM 编码。通常语音编码分为波形编码和参数编码。波形编码是直接对信号波形编码；参数编码简单地讲就是：用一些脉冲通过有特定参数的滤波器就能够逼近所要的语音信号，所以对于某一段语音信号，只要计算它对应的脉冲和滤波器参数，然后传送这些参数，接收端根据这些参数可恢复出近似的语音信号。G.723.1 属于参数编码。G.723.1 中的两种速率采用不同的编码方案，对于 5.3Kbit/s 采用 ACELP（Algebraic Code-Excited Linear Prediction），对 6.3Kbit 采用 MP-MLQ(Multy-Pulse –Multy Level Quantization)。

本章内容的根据是：ITU-T G.723.1 的描述文档[②]和 ITU-T Annex A[③]，根据的代码是 G.723.1 浮点程序(Annex B[④])。本文的原理参考了前几届的论文，在原理描述上本文在前几届论文基础上增加了静音检测(AnnexA)，并修正了一些地方。本章尽量不涉及 C 代码中的具体内容，这样能在不阅读 C 代码的情况下了解 G.723.1 的实现过程，但是如果能结合 G.723.1 浮点程序阅读本章，将能更深入理解算法的实现原理。

由于 ITU-T 的标准写得比较简练，和具体实现还有一定的距离，而阅读 C 代码又需要较长的时间。本章能够加速研究者阅读 C 代码。

本章中符号的规定：[x]表示比 x 小的最大整数；dim(A)表示 A 矢量的维数。文档指 ITU-T 网站的 G.723.1 描述文档。

1.1 G.723.1 编解码系统整体分析

1.1.1 G.723.1 编码器和解码器外部接口特性

编码器和解码器外部接口特性如图 1－1 所示：

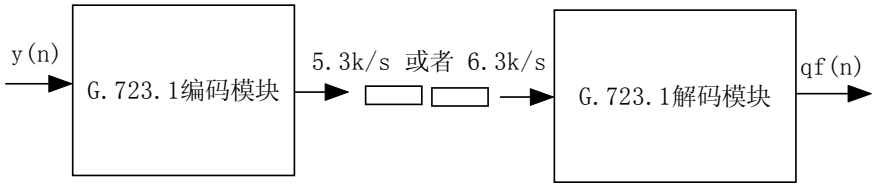

图1-1 G.723.1编码器和解码器外部接口

编码器的输入信号是 y(n)。模拟信号首先通过电话带通滤波器（G.712 建议）然后以 8000Hz 的速率采集，然后量化为 16bit 的线性 PCM 编码。16bit 的线性 PCM 编码用 16 位有符号数来表示采样点的幅度。

编码器将每 240 个样点作为一帧，编码以帧为单位处理。编码器的输出就是每帧语音信号的编码信息，每帧的编码信息以数据包的形式发送到解码器。

解码器的输入信息就是编码信息数据包，输出信息就是 8000Hz 采样速率、16bit 的线性 PCM 语音数据 qf（n）。

1.1.2 G.723.1 的编码延时

假设编码和解码的计算延时可以忽略；数据包的传输延时也可以忽略；在这种情况下 G.723.1 也有 37.5ms 的延时。

37.5ms 的延时包括两部分 30ms 和 7.5ms。

先看 30ms 部分。30ms 的延时是因为 G.723.1 编码一次需要读入 240 个点的数据，然后才进行编码。设第 t 帧第 n 个点到达编码器的时刻为 $CodInputT(t,n),0\le n\le 239$；第 t 帧第 n 个点在解码器输出的时刻为 $DecOutputT(t,n),0\le n\le 239$。语音输入和语音输出都是 8000Hz 的采样速率，有

$$DecOutputT(t,n)-DecOutputT(t,n-1)=1/8000$$

$$CodInputT(t,n)-CodInputT(t,n-1)=1/8000$$

总延时定义为

$$TotalDelay=DecOutputT(t,n)-CodInputT(t,n)$$

$$=DecOutputT(t,0)-CodInputT(t,0)$$

$$=DecOutputT(t,0)-CodInputT(t,239)+240/8000 \qquad (1.1)$$

由于编码器读取第 t 帧的第 239 点后才开始编码，所以

$$DecOutputT(t,0)-CodInputT(t,239)\ge 0$$

所以编码总延时大于等于 240/8000=30(ms)。

再看 7.5ms 的延时，7.5ms 延时产生的原因是编码时需要超前（look ahead）7.5ms，这是因为编码 n 时刻的点需要 n+60 时刻的点的数据，也就是需要知道 7.5ms 后的点。超前的引入是因为在计算 n 时刻的自相关时，需要 n 时刻的点的左右 60 个点的数据。为了实现超前，编码器不得不引入额外的 7.5ms 的延时。超前的具体的实现如图 1－2 所示：

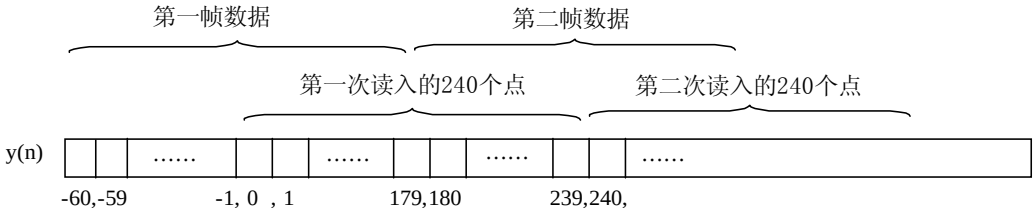

图1-2 编码器每次读入的数据和分帧的关系

图中点 180～239 虽然是第一次读入的数据，但是将作为第二帧数据，等到第二次读入 240 个点时才对点 180～239 编码。－60～－1 是编码器添加的 60 个全零的数据。7.5ms 延时使得第 t 帧开始编码的时刻并不是 $CodInputT(t,239)$ 而是 $CodInputT(t+1,59)$，所以有

$$DecOutputT(t,0)-CodInputT(t+1,59)\ge 0$$

$$\Rightarrow DecOutputT(t,0)-\left\{CodInputT(t,239)+60\times\frac{1}{8000}\right\}\ge 0$$

$$\Rightarrow DecOutputT(t,0) - CodInputT(t,239) \geq 7.5(ms) \qquad (1.2)$$

将(1.2)式代入（1.1）式可看出编码延时的两个组成部分

$$TotalDelay = DecOutputT(t,0) - CodInputT(t,239) + 240/8000 \geq 7.5(ms) + 30(ms)$$

1.1.3 G.723.1 编码流程图

图 1－3 为 G.723.1 的编码流程图。

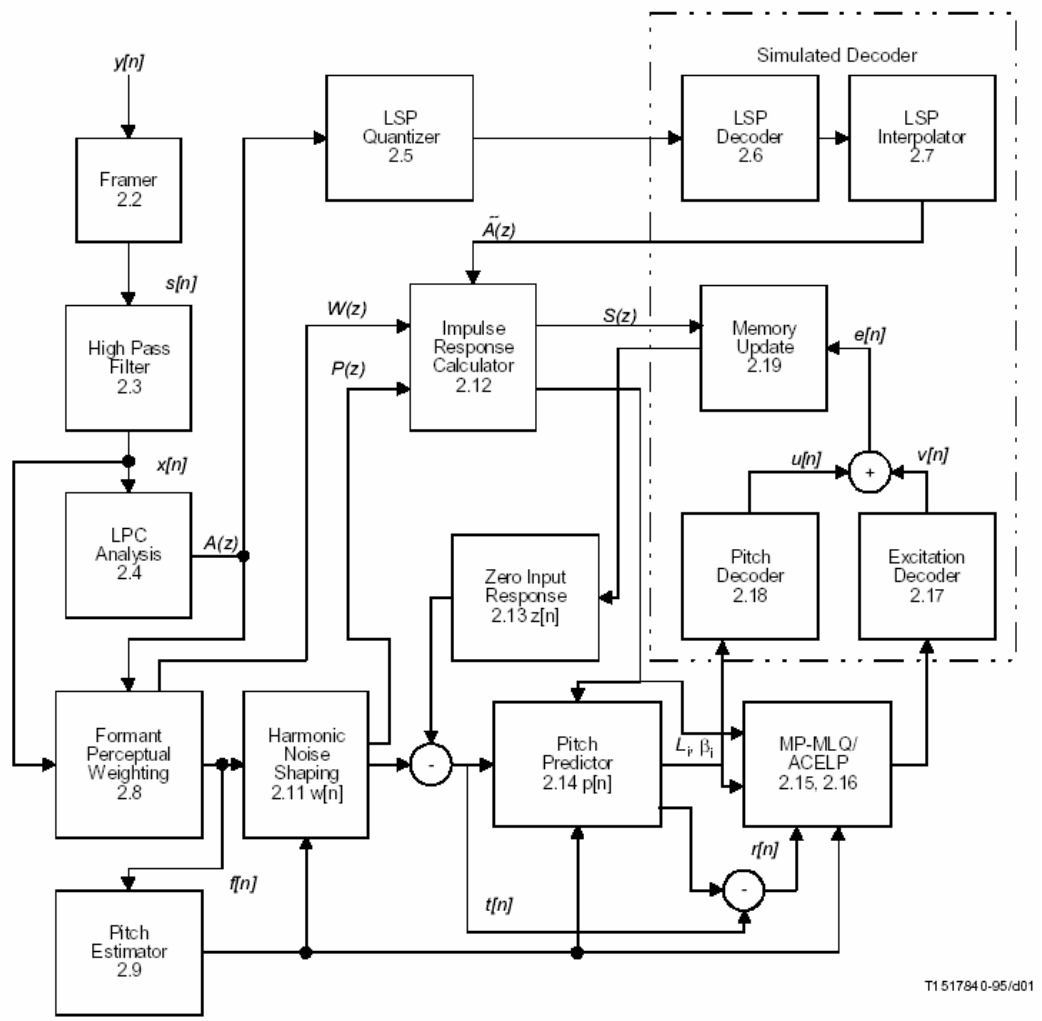

FIGURE 1/G.723.1

Block diagram of the speech coder – For each block the corresponding reference number is indicated

图 1-3 G.723.1 编码流程图

1.2 编码器

1.2.1 编码器的初始化

文件:CODER2. c[①]	函数: Init_Coder[②]	编码器初始化[③]

编码器在编码当前帧的数据时，需要用到前几帧的点和编码结果，必须将这些数据保存起来，我们称这些需要保存的变量为状态变量。状态变量需要初始化。各状态变量初始化情况如下：

（1） $\tilde{P}_{n-1}$：上一帧解码以后的 LSP 矢量。在 LSP 参数量化时使用；在 LSP 解码后更新。初始值为常矢量 P_{DC} ，即

$$\tilde{P}_{-1} \Leftarrow P_{DC}$$

（2） $Err[i], i = 0...4$：在计算自适应码本激励部分，用于计算码本搜索范围的 iTest 参数；在计算出当前子帧的自适应码本增益索引 Ic 和闭环基音周期 Li 以后更新。初始化值如下：

$$Err[i] = Err0, \mathrm{i} = 0...4$$

$$Err0 = 0.00000381464$$

（3） 其它状态变量都初始化为 0

1) $s_t[-1]$、$x_t[-1]$：高通滤波器状态变量。在高通滤波时使用，滤波完毕以后更新。
2) PreDate[n]：编码器处理上一帧时，读入编码器的 240 个语音采样点中的后 120 个点。PreDate[n]通过高通以后参与计算子帧的自相关系数；PreData[n]在数据移位部分更新为编码器此次读入的 240 个语音采样点中的后 120 个点。
3) $\mathrm{sinDet}[j], j = 0...14$：这 15 个数据记录最近 15 个子帧正弦特性，用于判断语音信号是否为正弦；sinDet 在计算了此子帧的 LPC 参数以后更新。
4) x[n]、fx[n]：感觉加权滤波器的状态变量，x[n]和 fx[n]各有 10 个点。在感觉加权滤波时使用，滤波完毕以后更新。
5) PrevWgt[n]：上一帧感觉加权滤波以后的后 145 个点。在开环基音预测时使用；计算完开环基音以后更新 PrevWgt[n]为此帧感觉加权滤波以后的后 145 个点。
6) RingFir[n]、PreErr[n]：联合滤波器的状态变量，保存了上一子帧的激励信号通过联合滤波器以后滤波器中保留的状态。它们用于计算联合滤波器的零输入响应；在自适应码本激励 u[n]和固定码本激励 v[n]计算出来以后，在联合滤波器保持上一子帧的保留状态的基础上，用激励信号 e[n]=u[n]+v[n]通过联合滤波器从而更新联合滤波器的状态。
7) PreExc[n]：上一帧计算的激励信号 e[n]的后 145 个点。自适应码本激励是用以

[①] 表示本小结的内容对应的 G.723.1 浮点程序所在的文件名。
[②] 表示本小结的内容对应的 G.723.1 浮点程序中的函数名。
[③] 表示本小结的内容对应的 G.723.1 浮点程序中的函数的主要功能。

前的激励重建的，在计算自适应码本激励时使用 PreExc[n]；在当前此帧的 e[n] 计算出来以后更新 PreExc[n]。

1.2.2 分帧(Framer)

设输入采样点序列（16bit 的线性 PCM 编码）为 $y[n]$。编码器每次读入 240 个点，记为 $s_t[n]_{n=0...239}$。帧的大小就是 240 个点，但是进行编码的帧并不是 $s_t[n]_{n=0...239}$，一个完整的帧由上一次读入的最后 60 个点和这次读入的 180 个点组成，所以第 t 帧的点为 $\{Frm_t[n]\}_{n=0...239} = \{s_{t-1}[180],\cdots,s_{t-1}[239],s_t[0],s_t[1],\cdots,s_t[179]\}$。

G.723.1 编码器和解码器都是以帧为处理单元来进行语音压缩和解压缩。编码器一次读取 240 个样点进行编码，解码器一次解码得到 240 个样点。

将第 t 帧 $\{Frm_t[n]\}_{n=0...239}$ 均分为 4 个子帧，分别为第 0、1、2、3 子帧，每个子帧有 60 个采样点。这里的第 0、1、2、3 子帧对应的数据为 $\{Frm_t[n]\}_{n=0..59}$、$\{Frm_t[n]\}_{n=60...119}$、$\{Frm_t[n]\}_{n=120...179}$、$\{Frm_t[n]\}_{n=180...239}$，也就是 $s_{t-1}[n]_{n=180...239}$、$s_t[n]_{n=0...59}$、$s_t[n]_{n=60...119}$、$s_t[n]_{n=120...179}$。

这就是说 G.723.1 编码有 60 个点的延迟时间，t 时刻读入的最后 60 个点，留在 t＋1 时刻处理。

1.2.3 高通滤波

文件:CODER2.c	函数: Rem_Dc	高通滤波

将一次读入的 240 个点：

$$s_t[n]_{n=0...239}$$

作为滤波器的输入。滤波输出信号为

$$x_t[n]_{n=0...239}$$

滤波器传输函数为：

$$H(z)=\frac{x(z)}{s(z)}=\frac{1-z^{-1}}{1-\frac{127}{128}z^{-1}} \tag{1.2.1}$$

时域系统函数：

$$x_t[n]=s_t[n]-s_t[n-1]+\frac{127}{128}x_t[n-1],n=0...239 \tag{1.2.2}$$

其中 n = -1 表示上一帧最后一个点（这是一个状态变量），如果 t 不等于 0 则：

$$s_t[-1]=s_{t-1}[239]$$

$$x_t[-1]=x_{t-1}[239]$$

如果 t=0，则 $s_t[-1]=0$， $x_t[-1]=0$。

1.2.4 LPC 参数的获取

文件:LPC2.c	函数: Comp_Lpc()	LPC 参数计算
文件:LPC2.c	函数: Durbin()	Levinson-Durbin 递推

对每个子帧求 10 阶的 LPC 参数 $a_{ij}, j=1,2...10, i=0...3$。LPC 参数将用来建立 LPC 合成滤波器。由 10 阶的 LPC 参数 $a_{ij}, j=1,2...10, i=0...3$ 建立的 LPC 合成滤波器定义如下

$$A_i(z)=\frac{1}{1-\sum_{j=1}^{10}a_{ij}z^{-j}}, 0\le i\le 3$$

1.2.4.1 子帧的 11 个自相关系数的求法

求自相关使用的数据为经过高通滤波以后的数据。求某一子帧的 LPC 参数首先需要知道该子帧的 11 个自相关系数 $R_i[j], j=0...10, i=0...3$。求第 i 子帧的 11 个自相关系数需要它的前后两个子帧的数据，即将 i-1、i、i+1 子帧组成一个 180 点的序列 $ThreeSubFrm_i$（注意，如果 i＝0，则第 i－1 子帧为前一帧的第 3 子帧；同理，如果 i＝3，，则 i＋1 表示下一帧的第 0 子帧）。正是这个原因，t 时刻还不能计算 $x_t[n]_{n=180...239}$ 的自相关和 LPC 参数，所以就有 60 个点的延迟。为了得到第 0 帧的数据 $x_{t-1}[n]_{n=180...239}$，编码器必须保留 t 时刻读入的最后 60 个点通过高通以后的数据 $x_{t-1}[n]_{n=180...239}$，为了计算 $x_{t-1}[n]_{n=180...239}$ 的自相关还需要保留 $x_{t-1}[n]_{n=120...179}$。所以 t－1 帧留给第 t 帧的数据是 $x_{t-1}[n]_{n=120...239}$，设这 120 个点保存在数组 PrevDat[120]中，编码器初始化时，PreDat[n]初始化为 0。自相关系数求法如下：

（1） 将 $ThreeSubFrm_i$ 和 180 点的汉明窗(Hamming window)相乘

$$ThreeSubFrmH_i[n]=ThreeSubFrm_i[n]\times HammiWindow[n], n=0...179$$

（2） 计算自相关

$$R_i[n]=\frac{1}{180\cdot 180}\sum_{j=0}^{179-n}ThreeSubFrmH_i[j]\times ThreeSubFrmH_i[j+n], n=0...10$$

（3）用 1025/1024 的白噪声系数（white noise correction factor）调整 R[0]

$$R_i[0]=R_i[0]\times[1+\frac{1}{1024}]$$

用二项式窗系数（binomial window coefficients）调整其它自相关系数

$$R_i[n]=R_i[n]\times Binormial[n], n\ne 0$$

1.2.4.2 用 Levision-Durbin 算法计算 LPC 参数

Levision-Durbin 算法用递推的方法由自相关系数计算 LPC 参数。自相关系数和 LPC 参数的关系由 Yule-Walker 方程

$$\begin{pmatrix} R[0] & R[1] & \cdots & R[p] \\ R[1] & R[0] & \cdots & R[p-1] \\ \vdots & \vdots & \ddots & \vdots \\ R[p] & R[p-1] & \cdots & R[0] \end{pmatrix} \cdot \begin{pmatrix} 1 \\ -a_1 \\ \vdots \\ -a_p \end{pmatrix} = \begin{pmatrix} E_p \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

决定。其中 $R[j], j = 0...p$ 是自相关系数，$a_j, j = 1...p$ 为 p 阶 LPC 参数。这里 $p = 10$。

下面用 Levision-Durbin 算法计算 Yule-Walker 方程的解。递推公式如下

（1） 初始化：i＝0，$E = R[0]$，$a_j = 0, j = 1,2...p$

（2） 计算

$$k \Leftarrow \frac{R[i+1] - \sum_{j=0}^{i-1} a_{j+1} \cdot R[i-j]}{E}$$

（注：其中当$i-1<0$时$\sum_{j=0}^{i-1} a_{j+1} \cdot R[i-j] = 0$）如果

$$|k| \geq 1$$

说明系统函数不稳定，则退出计算。

（3） 计算

$$a_j \Leftarrow \begin{cases} k & j = i+1 \\ a_j - k \cdot a_{i-j} & 1 \leq j \leq i \end{cases}$$

（4） 计算

$$E \Leftarrow (1-k^2) \cdot E$$

（5） i 增一，如果 i 等于 p 则退出计算，否则跳到（2）

用 $ThreeSubFrm_i$ 的自相关系数计算出来的 $a_j, j = 1...p$ 即为 i 子帧的 LPC 参数 $a_{ij}, j = 1,2...10, i = 0...3$

1.2.4.3 正弦检测

检测语音信号是否为正弦信号。这个检测结果将在计算自适应码本的 iTest 参数和 Annex A 的 VAD 算法中使用。检测是否为正弦信号的方法如下。

（1） 计算 $Pk2$

在 Levision-Durbin 算法中，如果检测到系统不稳定，即$|k| \geq 1$，则令

$$Pk2 = 0.99$$

否则令 $Pk2$ 为递推次数 i 等于 1 时的-k，即

$$Pk2 = -k, when \quad i = 1$$

对每一个子帧能够计算一个 $Pk2$ 值

（2） 更新 $\text{sinDet}[j], j = 0...14$ 并计算 sinD

sinDet[j]记录了最近 15 个子帧的 Pk2 值的情况用于计算正弦信号标志位 sinD。更新 $\text{sinDet}[j], j = 0...14$ 的方法是：

$$\text{sinDet}[j] = \text{sinDet}[j-1], j = 1...14$$

$$siDet[0] = \begin{cases} 1 & , Pk2 > 0.95 \\ 0 & , Pk2 \le 0.95 \end{cases} j = 0...15$$

如果 $siDet[j], j = 0...14$ 中等于 1 的个数大于等于 14 个，则认为信号是正弦信号，并设置 sinD＝1；否则为非正弦，设置 sinD＝0。

$\text{sinDet}[j], j = 0...14$ 也是编码器的状态变量，其初始化值为 0。

1.2.5 LSP 量化器

LSP 参数和 LPC 参数可用互相转化，但是 LSP 参数比较适合量化，所以将 LPC 参数转化为 LSP 参数，然后进行量化，解码端根据 LSP 参数反转化为 LPC 参数。这里只对每帧的第 3 子帧的 LPC 参数进行转化和量化。

1.2.5.1 LPC 参数至 LSP 参数的转化

将子帧的 10 个 LPC 参数 $\{a_j\}$ 转化为 10 个相应的 LSP 参数 $\{p'_j\}$ 的方法如下：

（1） 对 LPC 参数进行一个小的频带(7.5Hz)扩展

计算扩展以后的 LPC 参数

$$a'_{ij} = a_{ij} \cdot 0.994^j, j = 1,2...10$$

（2） LSP 参数和 LPC 参数的关系

LSP 参数和 LPC 参数通过 $P(z)$ 和 $Q(z)$ 多项式联系起来，它的原理如下：

设

$$P'(z) = 1 + z^{-11} - [(a'_1 + a'_{10})z^{-1} - (a'_2 + a'_9)z^{-2} - \cdots - (a'_{10} + a'_1)z^{-10}]$$

$$= A'(z) + z^{-11}A'(z^{-1})$$

设

$$Q'(z) = 1 - z^{-11} - [(a'_1 - a'_{10})z^{-1} - (a'_2 - a'_9)z^{-2} - \cdots - (a'_{10} - a'_1)z^{-10}]$$

$$= A'(z) - z^{-11}A'(z^{-1})$$

其中

$$A'(z)=1-a'_1z^{-1}-a'_2z^{-2}-\cdots-a'_{10}z^{-10}$$

可以证明方程

$$P(z)=\frac{P'(z)}{1+z^{-1}}=0$$

和

$$Q(z)=\frac{Q'(z)}{1+z^{-1}}=0$$

各有 5 对共轭根，且都在单位圆上，所以上式可以写作

$$P(z)=\prod_{i=1,3,5,7,9}(1+2q_iz^{-1}+z^{-2})=0$$

$$Q(z)=\prod_{i=2,4,6,8,10}(1+2q_iz^{-1}+z^{-2})=0$$

由两根之和的性质知，$-q_i$ 是以 z^{-1} 为变量的方程的第 i 个根的实部，而 q_i 则必然是以 z 为变量的方程的第 i 各根的实部设

$$q_i=\cos(\omega_i)$$

且

$$0<\omega_1<\omega_2<\cdots<\omega_{10}<\pi$$

则 ω_i 即为零点的圆频率，令

$$p'_j=256\frac{\omega_j}{\pi},j=1\ldots10$$

即为 LSP 参数。LSP 参数的物理意义是方程 $P(z)$ 和 $Q(z)$ 所在零点的圆频率。

（3） 计算 $P(z)$ 和 $Q(z)$ 多项式的系数

通过求 $P(z)$ 和 $Q(z)$ 多项式的系数然后通过求 $P(z)=0$ 和 $Q(z)=0$ 的根，来求 LSP 参数。设

$$P(z)=\sum_{i=0}^{10}f_P(i)z^{-i}$$

可以用长除法得到 $f_P(i)$，它是一个对称的多项式即 $f_P(i)=f_P(10-i)$，前 5 项的系数为

$$f_P(i)=-f_P(i-1)-(a'_i+a'_{11-i}),1\le i\le5$$

$$f_P(0)=1$$

设

$$Q(z)=\sum_{i=0}^{10} f_Q(i)z^{-i}$$

同理可得

$$f_Q(i)=f_Q(i-1)-(a_i'-a_{11-i}'),1\le i\le 5$$

$$f_Q(0)=1$$

（4） 求 $P(z)=0$ 和 $Q(z)=0$ 的根

首先计算方程的值的符号在哪两个之间有变化，然后通过一次牛顿弦截法，估计根。由于零点在单位圆上，可设

$$z=e^{j\omega}$$

可得

$$P(\omega)=e^{-j5\omega}[f_P(0)\cdot(e^{j5\omega}+e^{-j5\omega})+f_P(1)\cdot(e^{j4\omega}+e^{-j4\omega})+\cdots+f_P(5)\cdot e^{-j0\omega}]$$

$$=2e^{-j5\omega}\cdot[f_P(0)\cdot\cos 5\omega+f_P(1)\cdot\cos 4\omega+\cdots+\frac{f_P(5)}{2}]$$

同理可得

$$Q(\omega)=e^{-j5\omega}[f_Q(0)\cdot(e^{j5\omega}+e^{-j5\omega})+f_Q(1)\cdot(e^{j4\omega}+e^{-j4\omega})+\cdots+f_Q(5)\cdot e^{-j0\omega}]$$

$$=2e^{-j5\omega}\cdot[f_Q(0)\cdot\cos 5\omega+f_Q(1)\cdot\cos 4\omega+\cdots+\frac{f_Q(5)}{2}]$$

$P(\omega)$、$Q(\omega)$ 在 0～π 之间分别有 5 个根。它们有以下的特性

1） $$\omega_{i+1}-\omega_i\ge\frac{\pi}{256},1\le i\le 10$$

所以可以将 0～π 的单位圆分为 256 等分，然后比较相邻两等分的点的函数值符号是否变化，以判断它们之间是否存在零点。

2） $P(\omega)$ 和 $Q(\omega)$ 的零点位置交替出现，且 $P(\omega)$ 的第一个零点位置比 $Q(\omega)$ 的小。所以搜索时 ω 从 0 开始，先搜索 $P(\omega)$ 的零点，找到以后，再切换为搜索 $Q(\omega)$ 的零点，找到以后，再切换为搜索 $P(\omega)$ 的零点……。（注：ω 搜索的末点为 255/256×π，这根据程序分析所得）

用一次牛顿弦截法估计零点的位置以搜索 $P(\omega)$ 为例，设在

$$P(\omega_{i-1})\cdot P(\omega_i)<0$$

此次搜索到的一个 LSP 参数为：

$$p'_j = 256\frac{\omega_{i-1}}{\pi} + \frac{|P(\omega_{i-1})|}{|P(\omega_{i-1})| + |P(\omega_i)|}$$

如果此次搜索找不到 10 个零点，则用上一帧的 LSP 参数代替这一帧的 LSP 参数。

1.2.5.2 LSP 参数的量化

首先除去 LSP 矢量中的直流成分，然后将此帧的 LSP 矢量和上一帧的 LSP 矢量比较，将差值（残余误差矢量）编码，实现差分编码。然后对残余误差矢量量化。编码端和解码端都有一个量化表，量化的目的是寻找量化表中最接近残余误差矢量的元素，该元素在量化表中的索引将作为 LSP 参数量化的结果。方法如下：

（1） 计算对角加权矩阵W_n

$$W_n = \begin{pmatrix} W_{1,1} = \dfrac{1}{p'_2 - p'_1} & \cdots & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & W_{j,j} = \dfrac{1}{\min\{p'_j - p'_{j-1}, p'_{j+1} - p'_j\}} & \cdots & 0 \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & \cdots & W_{10,10} = \dfrac{1}{p'_{10} - p'_9} \end{pmatrix}$$

（2） 几个量的定义和计算

P'_n量化以后得到量化索引，解码端从量化索引可以计算解码以后的 LSP 矢量$\tilde{P}_n$， $\tilde{P}_n$ 近似等于P'_n。定义第 n 帧的解码以后的 LSP 矢量$\tilde{P}_n$

$$\tilde{P}_n = [\tilde{p}_0 \tilde{p}_1 \cdots \tilde{p}_{10}]$$

定义第 n 帧的 LSP 矢量P'_n

$$P'_n = [p'_0 p'_1 \cdots p'_{10}]$$

LSP 矢量P'_n和量化以后的 LSP 矢量$\tilde{P}_n$中都存在一个固定的 DC 分矢量，定义这个 DC 分矢量为P_{DC}

$$P_{DC} = [p_{DC0} p_{DC1} \cdots p_{DC10}]$$

用$b \cdot (\tilde{P}_{n-1} - P_{DC})$，其中 b 等于 12/32，来近似$P'_n - P_{DC}$，对它们的差值进行量化，定义残余误差矢量

$$e_n = [e_{n,1}, e_{n,2}, \cdots, e_{n,10}] = (P'_n - P_{DC}) - b \cdot (\tilde{P}_{n-1} - P_{DC})$$

我们将e_n进行量化编码，实现差分编码，这比直接对P_n编码时编码精度要高。另外从这

里可以看出，不仅在解码器端需要 LSP 解码，编码器端也需要将刚刚量化的 LSP 参数解码以得到 $\tilde{P}_n$。

（3） 对 e_n 进行量化，搜索量化表找到量化索引

这一步将用到 e_n、W_n 来求量化索引 I。将 LSP 量化表中的元素 $\tilde{e}_n$、LSP 误差矢量 e_n 和加权矩阵 W_n 切分为 3 段形成 3 个子矢量，维数分别是 3、3、4。即定义

$$
\begin{aligned}
e_{subn,1} &= [e_{n,1}, e_{n,2}, e_{n,3}] \\
e_{subn,2} &= [e_{n,4}, e_{n,5}, e_{n,6}] \\
e_{subn,3} &= [e_{n,7}, e_{n,8}, e_{n,9}, e_{n,10}] \\
\tilde{e}_{subn,1} &= [\tilde{e}_1, \tilde{e}_2, \tilde{e}_3] \\
\tilde{e}_{subn,2} &= [\tilde{e}_4, \tilde{e}_5, \tilde{e}_6] \\
\tilde{e}_{subn,3} &= [\tilde{e}_7, \tilde{e}_8, \tilde{e}_9, \tilde{e}_{10}]
\end{aligned}
$$

$$
W_{subn,1} = \begin{pmatrix} W_{1,1} & 0 & 0 \\ 0 & W_{2,2} & 0 \\ 0 & 0 & W_{2,2} \end{pmatrix}, W_{subn,2} = \begin{pmatrix} W_{3,3} & 0 & 0 \\ 0 & W_{4,4} & 0 \\ 0 & 0 & W_{5,5} \end{pmatrix}
$$

$$
W_{subn,3} = \begin{pmatrix} W_{6,6} & 0 & 0 & 0 \\ 0 & W_{7,7} & 0 & 0 \\ 0 & 0 & W_{8,8} & 0 \\ 0 & 0 & 0 & W_{9,9} \end{pmatrix}
$$

其中

$$\tilde{e}_{subn,i}, i = 1,2,3$$

各有 256 种取值，组成三个具有 256 个元素（子矢量）的常数表

$$BandQntTable_i, i = 1,2,3$$

用 8 比特的数

$$I_i, i = 1,2,3$$

作为表的索引。量化的步骤就是在 $BandQntTable_i$ 中寻找用 I_i 索引的 $\tilde{e}_{subn,i_i}$，使得

$$E_{i_i} = (2 \cdot e_{subn,i} - \tilde{e}_{subn,i_i}) \cdot W_{subn,i} \cdot (\tilde{e}_{sub,i})^T$$

（注：这个公式和文档有出入）达到最大。最后，将找到的 3 个索引统一编码为 I，即

$$I = I_1 \cdot 2^{16} + I_2 \cdot 2^8 + I_3$$

1.2.6 数据移位

正如 1.2.2 节所述，一帧数据 $\{Frm_t[n]\}_{n=0...239}$ 并不是等于 t 时刻读入的 240 个点 $s_t[n]_{n=0...239}$，而是有 60 个点的延时，所以高通滤波以后的数据 $x[i]$ 也不是要处理的帧数据。数据移位的目的就是调整 $x[i]$，以及更新 PreData[]。X[i]的数据左移 60 个点，这样 x[i],i=0…239，就变为第 t 帧的数据。

$$x[i]=\begin{cases}\Pr eData[60+i] & ,i=0...59\\ x[i-60] & ,i=60...239\end{cases}$$

PreData[]更新为 x[ ]的最后 120 个点。

$$\Pr eData[i]=x[120+i],i=0...119$$

1.2.7 共振峰感觉加权滤波（Formant perceptual weighting filter）

对每一子帧构建共振峰感觉加权滤波器 $W_i(z),i=0...3$，将数据移位以后的x[i]通过该滤波器得到输出信号f[n]。$W_i(z)$ 为：

$$W_i(z)=\frac{1-\sum_{j=1}^{10}a_{i,j}z^{-j}r_1^j}{1-\sum_{j=1}^{10}a_{i,j}z^{-j}r_2^j},r_1=0.9,r_2=0.5$$

其中 $a_{i,j}$ 是每一子帧的未量化的 LPC 参数（量化的 LPC 参数是指用解码得到的 LSP 反求 LPC 参数得到的 LPC 参数）。感觉加权滤波器的输入信号为经过数据移位以后的语音信号 $\{x[n]\}_{n=0...239}$,将其分为 4 个子帧，分别通过各自的加权滤波器。感觉加权滤波器可以分为 FIR 和 IIR 两部分的串联。设 FIR 的输出为 $\{fx[n]\}_{n=0...239}$，IIR 的输出为 $\{f[n]\}_{n=0...239}$，则

$$fx[n]=x[n]-\sum_{j=1}^{10}a_{ij}r_1^j x[n-j]$$

$$f[n]=fx[n]+\sum_{j=1}^{10}a_{ij}r_2^j f[n-j]$$

各自的子帧使用其各自的感觉加权滤波器系数。从公式可知，当计算某一子帧的 f[n]需

要知道上一子帧通过滤波器以后 x[n]和 fx[n]的最后的 10 个点的数据，这也是编码器的状态变量，其初始化值为 0。

1.2.8 开环基音预测（Pitch Estimate）

开环基音预测就是估计语音信号的基音周期L_{OL}。由$\{f[n]\}_{n=0...239}$来计算基音周期，对每一帧计算两个基音周期，前两个子帧和后两个子帧分别计算一个基音周期。

计算开环基音周期的方法如下：

定义互相关准则

$$C_{OL}(j)=\frac{\left(\sum_{n=0}^{119} f[n]\cdot f[n-j]\right)^2}{\sum_{n=0}^{119} f[n-j]\cdot f[n-j]},18\le j\le 142$$

开环基音周期用如下方法寻找

(1) L_{OL}=18,j=18，$MaxC_{OL}$=0

(2) 计算$C_{OL}(j)$，如果同时满足以下两个条件

1) $\sum_{n=0}^{119} f[n]\cdot f[n-j]>0$，$\sum_{n=0}^{119} f[n-j]\cdot f[n-j]>0$

2) $MaxC_{OL}<C_{OL}(j)$且$L_{OL}-j<18$。或者$MaxC_{OL}<\frac{3}{4}C_{OL}(j)$

则$L_{OL}=j$，$MaxC_{OL}=C_{OL}(j)$

(3) j=j+1，如果$j\le 142$则转到(2)，否则结束。

说明：计算分母$E(j)=\sum_{n=0}^{119} f[n-j]\cdot f[n-j]$可以用如下的递推公式

$$E(j)=E(j-1)-(f[120-j])^2+(f[-j])^2$$

另外，从以上的公式中可知计算时用到上一帧的 f[n]的后 142 个点，设用 PrevWgt[ ]保存这些点的值。由于以后的计算（计算谐波噪声滤波器）需要用到上一帧的 f[n]的后 145 个点的数据，所以 PrevWgt[ ]实际保存 f[n]的后 145 个点的数据，它也是编码器的状态变量，初始化值为 0。

1.2.9 谐波噪声滤波器(Harmonic noise shaping)

为了提高语音的编码质量，将语音信号通过谐波噪声滤波器。对每一子帧计算一个谐波噪声滤波器，各子帧的 f[n]通过各自谐波噪声滤波器以后输出 w[n]。谐波滤波器定义如下

$$P_i(z)=1-\beta z^{-L}$$

1.2.9.1 最佳延时 L 的计算

设

$$C_{pw}(j)=\frac{(N(j))^2}{\sum_{n=0}^{59} f[n-j]\cdot f[n-j]},L_{OL}-3\le j\le L_{OL}+3$$

其中

$$N(j)=\sum_{n=0}^{59} f[n]\cdot f[n-j]$$

设最大的$C_{pw}(j)$定义为C_L，计算 L 的方法是

(1) $C_L=1$，$L=L_{OL}$, $j=L_{OL}-3$,LFound=FALSE

(2)如果$N(j)\le 0$或者$E(j)=\sum_{n=0}^{59} f[n-j]\cdot f[n-j]\le 0$跳到(4)，否则跳到（3）。

(3)如果$C_{pw}(j)>C_L$，则$L=j$，$C_L=C_{pw}(j)$，设置 LFound 标志为 TRUE

(4)j=j+1，如果$j\le L_{OL}+3$跳到(2)，否则结束。

说明：计算分母$E(j)=\sum_{n=0}^{59} f[n-j]\cdot f[n-j]\le 0$可以用如下的递推公式

$$E(j)=E(j-1)-(f[60-j])^2+(f[-j])^2$$

计算 L 时将用到此帧的 f[n]以前的 145 个点。

1.2.9.2 最佳增益 β 的计算

如果计算最佳延时 L 时得到的 LFound 标志为 FALSE 则 β 值为 0。否则按如下方式计算

$$\beta=\begin{cases}0.3125G_{opt} & ,-10\log_{10}(1-\frac{C_L}{E})\ge 2.0\\ 0.0 & -10\log_{10}(1-\frac{C_L}{E})<2.0\end{cases}$$

其中

$$E=\sum_{n=0}^{59}(f[n])^2$$

$$G_{opt}=\begin{cases}\frac{N(L)}{E(L)} & ,\frac{N(L)}{E(L)}\le 1\\ 1.0 & ,\frac{N(L)}{E(L)}>1\end{cases}$$

条件$-10\log_{10}(1-\frac{C_L}{E})\ge 2.0$可进一步化简为

$$1-\frac{C_L}{E}\le 10^{-\frac{2.0}{10}}\approx\frac{5}{8}$$

即近似于

$$(N(j))^2 > 0.375 \cdot E \cdot E(L)$$

1.2.9.3 谐波噪声滤波

各个子帧的通过 f[n]各自的谐波滤波器 $P_i(z)$ 得到 w[n]，在时域表现为

$$w[n] = f[n] - \beta f[n-L], 0 \le n \le 59$$

L 的最大值为 145，n－L 最小为-145，所以计算时需要用到此帧以前的 f[n]的 145 个点的数据。

1.2.10 LSP 解码

LSP 解码的任务就是从 LSP 量化索引 I 得到解码以后的 LSP 矢量 $\tilde{P}_n$。LSP 解码不仅存在于解码器端，而且在编码器端也要进行 LSP 解码，这是为了给 LSP 量化提供参数 $\tilde{P}_{n-1}$。

1.2.10.1 根据量化索引 I 计算 LSP 解码矢量 $\tilde{P}_n$

根据

$$I = I_1 \cdot 2^{16} + I_2 \cdot 2^8 + I_3$$

可以将 I 分解为

$$I_i, i = 1,2,3$$

在子矢量量化表

$$BandQntTable_i, i = 1,2,3$$

中用 I_i 索引可以得到量化以后的残余误差子矢量

$$\tilde{e}_{sub\,n,i} = BandQntTable_i[I_i], i = 1,2,3$$

则

$$\tilde{e}_n = [\tilde{e}_{n,1}, \tilde{e}_{n,2}, \tilde{e}_{n,3}]$$

类似于 LSP 量化时，残余误差矢量的计算公式有

$$\tilde{e}_n = (\tilde{P}_n - P_{DC}) - b \cdot (\tilde{P}_{n-1} - P_{DC})$$

即

$$\tilde{P}_n = (\tilde{e}_n + P_{DC}) + b \cdot (\tilde{P}_{n-1} - P_{DC})$$

1.2.10.2 LSP 解码矢量 $\tilde{P}_n$ 的稳定性测试和调整

设

$$\Delta_{\min} = 31.25Hz$$

对于采样频率为 8000 的数字系统 $\Delta_{\min}$ 频率大小的变化，对应圆频率的变化为

$$\frac{\Delta_{\min}}{8K/2}\pi$$

LSP 参数的值从 0～256 变化对应于圆频率从 0～π 变化，所以 $\Delta_{\min}$ 频率大小的变化对应 LSP 参数值的变化为

$$\Delta_{LSP\min} = 256 \cdot \frac{\frac{\Delta_{\min}}{8K/2}\pi}{\pi} = 2$$

$\tilde{P}_n$ 调整方法是：相邻 LSP 参数之间间隔不能太小，如果太小对其间隔进行扩展，如果多次扩展还是间隔太小，则使用上一帧的解码结果。具体步骤如下：

(1) 调整 $\tilde{p}_{n,1}$ 和 $\tilde{p}_{n,10}$：如果 $\tilde{p}_{n,1} < 3$ 则 $\tilde{p}_{n,1} = 3$；如果 $\tilde{p}_{n,10} > 252$ 则 $\tilde{p}_{n,10} = 252$

(2) 设置调整次数 AdustTimes 的初始值为 1

(3)扩展 $\tilde{p}_{n,i+1}, \tilde{p}_{n,i}, i = 1...9$ 之间的间隔：

(3.1)设 i＝1

(3.2)如果

$$\tilde{p}_{n,i+1} - \tilde{p}_{n,i} < \Delta_{LSP\min}, i = 1...9$$

则做如下调整

$$\tilde{p}_{n,i+1} \Leftarrow \tilde{p}_{n,i+1} + \frac{\Delta_{LSP\min} - (\tilde{p}_{n,i+1} - \tilde{p}_{n,i})}{2} = \frac{(\tilde{p}_{n,i+1} + \tilde{p}_{n,i})}{2} + \frac{\Delta_{LSP\min}}{2}$$

$$\tilde{p}_{n,i} \Leftarrow \tilde{p}_{n,i} - \frac{\Delta_{LSP\min} - (\tilde{p}_{n,i+1} - \tilde{p}_{n,i})}{2} = \frac{(\tilde{p}_{n,i+1} + \tilde{p}_{n,i})}{2} - \frac{\Delta_{LSP\min}}{2}$$

(3.3)如果 i<9，则 i 增一后跳到(3.1)，否则跳到(4)

（4）检测此次扩展以后，$\tilde{P}_n$ 是否已经满足稳定性要求：如果对所有的 i 都满足

$$\tilde{p}_{n,i+1} - \tilde{p}_{n,i} > \Delta_{LSP\min} - \frac{1}{32}, i = 1...9$$

则调整完毕，退出；否则进入（5）

（5）AdjustTimes 增 1，如果 AdjustTimes>9 则用 $\tilde{P}_{n-1}$ 代替 $\tilde{P}_n$ 后退出；否则跳到(3)

1.2.10.3 LSP 解码时的错误帧处理

在解码器端（不包括编码器端的 LSP 解码），如果由于传输的原因，接收编码信息出错，那么 LSP 解码将进行特殊的处理。编码信息出错有两个检测手段：一个是信道编码部分检测到 CRC 校验和错误，另一个是在解包编码信息时发现参数出现了不允许的值。当出现错误帧时：LSP 解码参数修改如下： $\Delta_{LSP\min}=4$；LSP 量化索引固定为 0，即 I＝0；计算残余误差矢量的固定预测值 $b=23/32$

1.2.11 LSP 插值

由于在一帧中只计算最后一个子帧的 LSP 参数，编码器端并没有将其它子帧的 LSP 参数传输给解码器端，所以解码器端必须用一种方法计算其它子帧的 LSP 参数。这里使用的方法是线性插值。LSP 插值的任务是用 LSP 解码得到的前后两帧的解码 LSP 矢量 $\tilde{P}_{n-1}$ 和 $\tilde{P}_n$ 插值得到各个子帧的 LSP 解码矢量。

定义第 n 帧的各个子帧的解码以后的 LSP 矢量为 $\tilde{P}_{n,i},i=0...3$。按照定义应该有 $\tilde{P}_{n,3}=\tilde{P}_n$。用线性插值法计算 $\tilde{P}_{n,i}$ 的公式如下

$$\tilde{P}_{n,i}=\begin{cases}0.75\tilde{P}_{n-1}+0.25\tilde{P}_n & i=0\\ 0.5\tilde{P}_{n-1}+0.5\tilde{P}_n & i=1\\ 0.25\tilde{P}_{n-1}+0.75\tilde{P}_n & i=2\\ \tilde{P}_n & i=3\end{cases}$$

1.2.12 LSP 参数至 LPC 参数的转化

LSP 参数转化为 LPC 参数的任务是将 LSP 插值得到的各个子帧的 LSP 参数 $\tilde{P}_{n,i}$ 反转化为量化后的 LPC 参数 $a_{ij},j=1,2...10,i=0...3$。方法如下：

（1） 计算 $\tilde{q}_i$

由于 LSP 参数

$$\tilde{p}_j=256\frac{\tilde{\omega}_j}{\pi},j=1...10$$

而

$$\tilde{q}_i=\cos(\tilde{\omega}_j)$$

现在要从 $\tilde{p}_j$ 计算 $\tilde{q}_i$，这里使用查 CosinTable 表和线性插值的方法计算

$$\tilde{q}_i=\text{CosinTable}[[\tilde{p}_i]]+(\text{CosinTable}[[\tilde{p}_i]+1]-\text{CosinTable}[[\tilde{p}_i]])\times(\tilde{p}_i-[\tilde{p}_i])$$

$[\tilde{p}_i]$ 表示取整数部分。CosinTable 表记录 512 个角的 cos 值，这些角是 0,1/512,2/512,…511/512。由于 LPC 转化为 LSP 参数时使用了线性插值的方法，所以这里用相反的插值过程可以完全恢复原来的值，即如果有 $\tilde{p}_j = p'_j$，则 $\tilde{q}_i = q_i$。

(2) 确定 $P(z)$ 和 $Q(z)$ 的系数和 $\tilde{q}_i$ 的关系

我们可以利用公式

$$P(z) = \prod_{i=1,3,5,7,9}(1 + 2q_i z^{-1} + z^{-2})$$

$$Q(z) = \prod_{i=2,4,6,8,10}(1 + 2q_i z^{-1} + z^{-2})$$

计算 $P(z)$ 和 $Q(z)$ 的系数。以 $P(z)$ 为例，设其因子为 $P_i(z), i = 1...5$，设 $2b_i = q_{i\times 2-1}, i = 1...5$

则

$$P(z) = \prod_{j=1}^{5}(1 + b_j z^{-1} + z^{-2})$$

设

$$P_{MULi}(z) = \prod_{j=1}^{i} P_j(z) = \prod_{j=1}^{i}(1 + b_j z^{-1} + z^{-2})$$

$$= 1 + k_{i,1} z^{-1} + k_{i,2} z^{-2} + \cdots + k_{i,2i-1} z^{-(2i-1)} + z^{-2i}, 1 \le i \le 5$$

容易证明 $P_{MUL_i}(z)$ 的系数是对称，所以只用计算 $k_{i,j}$ 中，$0 \le j \le i$ 的系数。$k_{5,j}, 0 \le j \le 5$ 即是最终要求的 $P(z)$ 的系数，以下将使用递推的方法计算

对任何 i，常数项即 j＝0 时都为 1，即

$$k_{i,0} = 1, 1 \le i \le 5$$

i=2 时

$$k_{2,1} = b_1 + b_2, \quad k_{2,2} = b_1 b_2 + 2$$

对于 $2 < i \le 5$

$$k_{i,j} = \begin{cases} k_{i-1,i-1} b_i + 2k_{i-1,i-2} & , j = i \\ k_{i-1,j-2} + k_{i-1,j-1} b_i + k_{i-1,j} & , 1 < j < i \\ k_{i-1,1} + b_i & , j = 1 \end{cases}$$

(3) 确定 LPC 参数 a_i 和 $P(z)$ 和 $Q(z)$ 的系数的关系

有公式

$$\tilde{A}_i(z) = \frac{P(z)(1 + z^{-1}) + Q(z)(1 - z^{-1})}{2}$$

设 $P(z)$ 和 $Q(z)$ 的系数为 p_i，q_i 则

$$\begin{aligned}\tilde{A}_i(z) &= \frac{1}{2}[(\sum_{i=0}^{5} p_i z^{-i} + \sum_{i=6}^{10} p_{10-i} z^{-i})(1+z^{-1}) + (\sum_{i=0}^{5} q_i z^{-i} + \sum_{i=6}^{10} q_{10-i} z^{-i})(1-z^{-1})] \\ &= \frac{1}{2}[(\sum_{i=0}^{5} p_i z^{-i} + \sum_{i=6}^{10} p_{10-i} z^{-i}) + (\sum_{i=1}^{6} p_{i-1} z^{-i} + \sum_{i=7}^{11} p_{11-i} z^{-i}) \\ &\quad + (\sum_{i=0}^{5} q_i z^{-i} + \sum_{i=6}^{10} q_{10-i} z^{-i}) - (\sum_{i=1}^{6} q_{i-1} z^{-i} + \sum_{i=7}^{11} q_{11-i} z^{-i})] \\ &= \frac{1}{2}[p_0 + q_0 + \sum_{i=1}^{5}(p_i + p_{i-1} + q_i - q_{i-1}) z^{-i} + (p_4 + p_5 + q_4 - q_5) z^{-6} \\ &\quad + \sum_{i=7}^{10}(p_{10-i} + p_{11-i} + q_{10-i} - q_{11-i}) z^{-i} + (p_0 - q_0) z^{-11}]\end{aligned}$$

由于 $p_0 = q_0 = 1$，上式化简为

$$\begin{aligned}&= \frac{1}{2}[p_0 + q_0 + \sum_{i=1}^{5}(p_i + p_{i-1} + q_i - q_{i-1}) z^{-i} + \sum_{i=6}^{10}(p_{11-i} + p_{10-i} - q_{11-i} + q_{10-i}) z^{-i}] \\ &= \frac{1}{2}[p_0 + q_0 + \sum_{i=1}^{5}(p_i + p_{i-1} + q_i - q_{i-1}) z^{-i} + \sum_{i=1}^{5}(p_i + p_{i-1} - q_i + q_{i-1}) z^{-(11-i)}]\end{aligned}$$

对照

$$\tilde{A}(z) = 1 - a_1 z^{-1} - a_2 z^{-2} - \cdots - a_{10} z^{-10}$$

可得 LPC 参数计算公式

$$a_i = \begin{cases} \dfrac{-p_{i-1} - p_i + q_{i-1} - q_i}{2} & 0 < i \le 5 \\ \dfrac{-p_{i-1} - p_i - q_{i-1} + q_i}{2} & 5 < i \le 10 \end{cases}$$

1.2.13 联合滤波器的冲击响应 h(n)（combined filter）

联合滤波器由 LPC 合成滤波器、共振峰感觉加权滤波器和谐波噪声滤波器级联组成。我们将计算出一个激励信号(excitation)，让激励通过联合滤波器，使得联合滤波器的输出信号接近语音信号。激励信号通过联合滤波器的输出就是等于激励和联合滤波器冲击响应的卷积。这就是计算联合滤波器冲击响应的目的。

1.2.13.1 联合滤波器的定义

对一个子帧计算一个联合滤波器参数。每一子帧的联合滤波器 $S_i(z)$ 由 LPC 合成滤波器（使用量化以后的 LPC 参数）、共振峰感觉加权滤波器和谐波噪声滤波器级联组成

$$S_i(z) = \tilde{A}_i(z) \cdot W_i(z) \cdot P_i(z), 0 \le i \le 3$$

其中

$$\tilde{A}_i(z) = \frac{1}{1 - \sum_{j=1}^{10} a_{ij} z^{-j}}, 0 \le i \le 3$$

$$W_i(z)=\frac{1-\sum_{j=1}^{10}a_{i,j}z^{-j}r_1^j}{1-\sum_{j=1}^{10}a_{i,j}z^{-j}r_2^j},r_1=0.9,r_2=0.5$$

$$P_i(z)=1-\beta z^{-L}$$

1.2.13.2 联合滤波器的冲击响应的计算

设联合滤波器的输入信号为冲击信号 $\delta(n)$

$$\delta(n)=\begin{cases}1 & n=0\\ 0 & n\neq 0\end{cases}$$

通过联合滤波器以后的信号为 $h(n)$，$0\le n<60$，这里只取冲击响应的前 60 个点。如图 1－4 所示：

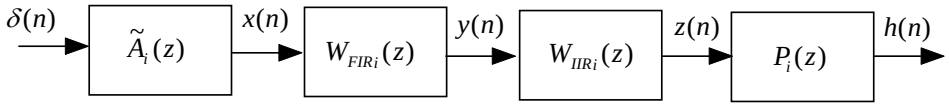

图 1-4 联合滤波器冲击响应的计算

计算步骤如下

（1） 滤波的状态（或者称为寄存器）有：$x(n)$ 当前时刻以前的 10 个点；$z(n)$ 当前时刻前的 145 个点。由于求冲击响应，滤波器的初始状态必须为零，所以首先需要将滤波器状态清零。

（2） 冲击信号通过 $\tilde{A}_i(z)$，即

$$x(n)=\sum_{j=1}^{10}a_{i,j}x(n-j)+\delta(n),n=0\cdots 59$$

可见需要用到 $x(n)$ 当前时刻以前的 10 个点的数据。必须保留这 10 个点，并随着 n 的增加更新这 10 个点的数据。

（3）通过 $W_{FIRi}(z)$，即

$$y(n)=x(n)-\sum_{j=1}^{10}a_{i,j}r_1^j x(n-j),n=0\cdots 59$$

同样需要用到 $x(n)$ 当前时刻以前的 10 个点的数据信息。

（4）通过 $W_{IIRi}(z)$，则

$$z(n) = y(n) + \sum_{j=1}^{10} a_{i,j} r_2^j z(n-j), n = 0 \cdots 59$$

需要用到 $z(n)$ 当前时刻以前的 10 个点的数据信息。必须保留这 10 个点，并随着 n 的增加更新这 10 个点的数据。

(5) 通过 $P_i(z)$，则

$$h(n) = z(n) - \beta z(n-L), n = 0 \cdots 59$$

L 的最大值为 145，所以需要保留 $z(n)$ 当前时刻以前的 145 个点的数据。

1.2.14 联合滤波器的零输入响应的计算

正如前所述，我们将用一激励信号通过联合滤波器，让联合滤波器的输出近似语音信号。我们对每一子帧计算一个联合滤波器参数和一个激励信号。具体的操作过程是这样的，当 i 子帧的激励通过 i 子帧联合滤波器以后，我们将联合滤波器的参数换成 i＋1 子帧的，激励也换成下 i＋1 子帧的激励，但是需要注意的是 i 子帧通过联合滤波器以后，在滤波器中保留了的状态变量（寄存器中保留的数据），这些状态变量将作为 i＋1 帧的联合滤波器的初始状态。这样 i＋1 子帧的联合滤波器的输出将包含两部分：i＋1 子帧的激励信号的响应；i＋1 子帧初始状态的响应。也就是零状态响应和零输入响应。

我们首先计算零输入响应 $\{z[n]\}_{n=0...59}$，然后从语音信号通过共振峰感觉加权滤波器和谐波噪声滤波器以后的信号 $\{w[n]\}_{n=0...59}$ 中减去 $\{z[n]\}_{n=0...59}$，得到 $\{t[n]\}_{n=0...59}$

$$t[n] = w[n] - z[n], n = 0...59$$

那么 $t[n]$ 将是激励响应（零状态响应）所要逼近的信号。以后将根据 $t[n]$ 来求这一激励。

$z[n]$ 的求法类似于计算联合滤波器的脉冲响应的的方法。参考图 1－4（联合滤波器冲击响应的计算），设 $\tilde{A}_i(z)$ 的输出为 xdl[n]，$W_i(z)$ 的输出为 zdl[n]。当前子帧的零输入状态为：RingFir[10]保存上一子帧的 xdl[n]的最后 10 个点，PreErr [145]保存上一子帧的 zdl[n]的最后 145 个点。RingFir[10]和 PreErr[145]是编码器的状态变量，编码其初始化时，初始化为 0。

零输入响应的计算步骤如下：

（1） 初始化。将 RingFir[10]的 10 个点赋给 xdl[]，PreErr[145]的 145 个点赋给 zdl[]。

（2） 通过 $\tilde{A}_i(z)$。输入信号为零。

（3） 通过 $W_{FIRi}(z)$。

（4） 通过 $W_{IIRi}(z)$。

（5） 通过 $P_i(z)$。得到 $z[n]$。

1.2.15 自适应码本激励

正如以上所述，可以用激励信号 e[n]通过联合滤波器来逼近 $\{t[n]\}_{n=0...59}$。激励信号 e[n]有可以分为两部分自适应码本激励 u[n]和固定码本激励 v[n]，有以下关系 e[n]=u[n]+v[n]。

本小节就是介绍自适应码本激励 u[n]的求法。为了得到 u[n]，必须得到两个参数自适应码本增益索引 Ic 和闭环基音周期 Li。利用闭环基音周期 Li 可以从以前（前几子帧）的激励 e[n-Li]计算重建激励 e'[n]。e'[n]用码本增益加权以后可以得到自适应码本激励 u[n]。

这里自适应是指，u[n]是用以前的激励 e[n]通过自适应码本增益调节以后得到，根据 $\{t[n]\}_{n=0...59}$ 调节自适应码本增益的大小，让 u[n]自适应于 $\{t[n]\}_{n=0...59}$

这里使用的自适应算法（预测方法）为五阶基音预测（fifth order pitch predictor），令阶数 PitchOrder=5，阶数的一半 PitchOrder2=2；

1.2.15.1 闭环基音周期的取值范围

设各子帧的闭环基音周期定义为 $L_i, i=0,1,2,3$。开环基音周期为： $L_{OLi}, i=0,1$，分别表示前两个子帧的开环基音周期和后两个子帧的开环基音周期。首先对 L_{OLi} 做一个调整得到 L_{OLAi}

$$L_{OLA_i}=\begin{cases}19 & ,L_{OLi}=18\\ L_{OLi} & ,18<L_{OLi}\le 140\\ 140 & ,L_{OLi}>140\end{cases}$$

闭环基音周期 L_i 在开环基音周期附近取值，奇子帧和偶子帧的 L_i 的取值范围不同，具体情况如下：

$$L_0 \in U_0=\{L_{OLA0}-1, L_{OLA0}, L_{OLA0}+1\}$$

$$L_1 \in U_1=\{L_0-1, L_0, L_0+1, L_0+2\}$$

$$L_2 \in U_2=\{L_{OLA1}-1, L_{OLA1}, L_{OLA1}+1\}$$

$$L_3 \in U_3=\{L_2-1, L_2, L_2+1, L_2+2\}$$

L_i 的最小值为 17，最大值为 143。Li 的取值个数也就是 Ui 的元素的个数，用 dim(Ui)表示。Ui 中的第 j 个元素用 Ui(j)表示，其中 j 大于等于 0 小于 dim(Ui)。

1.2.15.2 未量化的自适应码本增益 Vi(j)

Vi(j)表示第 i 子帧的闭环基音周期 L_i 取 Ui(j)时计算出来的未量化的自适应码本增益。Vi(j)是一个 20 维的矢量随着 L_i 取值的不同而不同。这一步的任务是：计算所有的 Vi(j), 0≤ j<dim(Ui)。计算步骤如下

（1） 初始化。 $j=0$

（2） Li=Ui(j)

（3）计算 Li 对应的重建激励 $e'[n]$。

设 $e[n]$ 表示激励信号，e[0]表示当前帧的激励的第一个点，现用以前的激励信号产生重建激励 $e'[n]$

$$e'[n]=\begin{cases} e[-L_i-PitchOrder2] & ,n=0 \\ e[-L_i-PitchOrder2+1] & ,n=1 \quad PitchOrder2=2 \\ e[-L_i+((n-PitchOrder2)\bmod L_i)] & ,2\le n\le 63 \end{cases} \quad (1.2.15.1)$$

（4）计算 $f_i(n)$。

定义 $f_i(n), i=0...(PitchOrder-1), n=0...59$， $f_i(n)$ 的计算公式如下

$$f_i(n)=\begin{cases} \sum_{j=0}^{n} e'(j+PitchOrder-1)\cdot h(n-j) & ,i=PitchOrder-1 \\ \begin{cases} e'(i) & ,n=0 \\ e'(i)\cdot h(n)+f_{i+1}(n-1) & ,1\le n\le 59 \end{cases} & ,0\le i<PitchOrder-1 \end{cases}$$

h(n)为联合滤波器冲击响应。

（5）计算 $t(n)$ 和 $f_i(n)$ 的互相关 C_i

$$C_i=\sum_{j=0}^{59} t(j)\cdot f_i(j), 0\le i\le PitchOrder-1$$

（6）计算 $f_i(n)$ 的能量 E_i

$$E_i=\frac{1}{2}\sum_{j=0}^{59} f_i^2(j), 0\le i\le PitchOrder-1$$

（7）计算 $f_i(n)$ 之间的相关性 $C_{I\,m,n}$

$$C_{I\,m,n}=\sum_{l=0}^{59} f_m(l)\cdot f_n(l), m\ne n, 0\le m\le PitchOrder-1, 0\le n\le PitchOrder-1$$

（8）得到 20 维矢量 Vi(j)

$$Vi(j)=[C_0,\cdots,C_4,E_0,\cdots,E_4,C_{I1,0},C_{I\,2,0},C_{I\,2,1},C_{I\,3,0},C_{I\,3,1},C_{I\,3,2}C_{I\,4,0},C_{I\,4,1},C_{I\,4,2},C_{I\,4,3}], 0\le i\le 4$$

（9）j 增一，如果 $j < \dim(U_i)$ 则跳到（2），否则退出。

1.2.15.3 闭环基音周期 Li 和码本索引 Ic 的选择

对于第 i 子帧，用以上的计算步骤得到了 Li 的 dim(Ui)个可能取值 Ui(j)，以及 Ui(j)对应的 Vi(j)。现在要从中选择一个最佳值。

选择的方法如下。编码器和解码器都有自适应码本增益表 AcbkGainTable，表中的元素为 20 维的矢量。搜索增益表，设表的索引为 m，将表中元素 AcbkGainTable[m]和 Vi(j),0≤j<dim(Ui)求互相关，记最大的互相关时的 j 和 m 为 j_{opt} 和 Ic，那么闭环基音周期 Li 的最佳值为 Ui(j_{opt})，自适应码本增益的量化值为 AcbkGainTable[Ic]，Ic 为自适应码本增益索引。

1.2.15.3.1 码本的选择(Code book)

在选择 Li 时，需要得到码本增益表 AcbkGainTable。G.723.1 中有两个自适应码本增益表：85 个元素的 AcbkGainTable085 和 127 个元素的 AcbkGainTable170。

使用 AcbkGainTable085 还是 AcbkGainTable170 作为 AcbkGainTable，要根据以下方法判断：对于 Li 的每个可能取值 Ui(j)都要进行一次码本选择。选择方法如图 1－5：

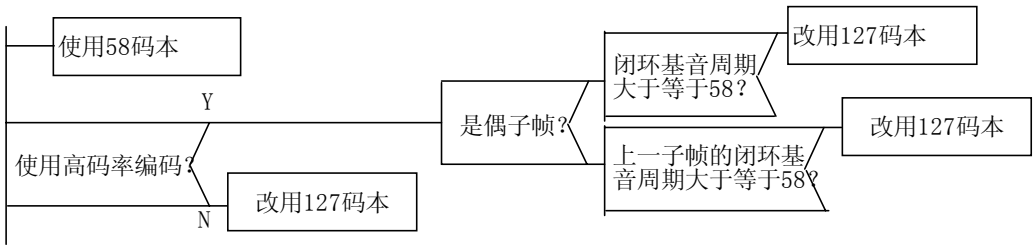

图 1-5 码本的选择（PAD图）

1.2.15.3.2 码本的搜索范围的计算

选择了码本以后我们不是搜索码本中的所有记录，码本索引 I_C 的取值区间为

$$[0, Bound)$$

Bound 的求法是：对于 85 码本

$$Bound = Min(51 + iTest \times 4, 85)$$

对于 127 码本

$$Bound = Min(93 + iTest \times 8, 170)$$

其中 *iTest* 的计算方法后面将讲述。

1.2.15.3.3 利用 Err[i]计算 iTest 参数

将 1～150 共 150 个点分为 5 个区间，每个区间 30 个点，如表 1－1：

表 1－1 区间分隔情况

区间号	0	1	2	3	4
包含的点	1～30	31～60	61～90	91～120	121～150

（2） 计算 zone1 和 zone2。设第 i 子帧的闭环基音周期的取值范围从为从 L_s 到 L_L 的闭区间（参考闭环基音周期的取值范围部分）。L_s 的最小值为 17，L_L 的最大值为 143。计算 L_L+3 所在的区间号，即

$$zone2=\left[\frac{L_L+3-1}{30}\right]$$

计算 L_s-60 所在的区间号：如果 $L_s-60-1\le 0$，则 $zone1=0$；否则

$$zone1=\left[\frac{L_s-60-1}{30}\right]$$

$zone1$ 和 $zone2$ 的可能值必定在[0，4]内间的整数。

（3） 计算 ErrMax。计算 Err[i],其中 i=zone1...zone2，内最大的 Err[i]令其为 ErrMax

$$ErrMax=\max\{Err[i]\},i=zone1,\cdots,zone2$$

如果 ErrMax<-1，则 ErrMax＝－1。

（4） 计算 iTest。

如果 $ErrMax>128$ 或者此帧的输入信号为正弦信号，即 sinD＝1

则

$$iTest=0$$

否则

$$iTest=128-ErrMax$$

1.2.15.3.4 闭环基音周期 Li 和码本索引 Ic 的选择

码本表中的元素是 20 维的矢量，设码本表中的第 m 个元素 AcbkGainTable[m]表示为

$$\text{AcbkGainTable[m]}=[\text{AcbkGainTable[m][0]},\text{AcbkGainTable[m][1]},\cdots,\text{AcbkGainTable[m][19]}]$$

对第 i 子帧搜索所有的 j 和 m，选择互相关最大的 j 和 m，步骤如下：

（1） j=0,CmjMax = 0,Ic=0。Li= L_{OPAi}

（2） 选择 j 对应的未量化的自适应码本增益 Vi(j)

（3） 根据 Ui(j)选择码本

（4） 计算码本边界 Bound 值

（5） m=0

（6） 计算互相关

$$Cmj = Vi(j) \bullet AcbkGainTable[m] = \sum_{l=1}^{20} AcbkGainTable[m][l] \cdot Vi(j)[l]$$

如果 Cmj>CmjMax，则 CmjMax = Cmj，Ic=m，Li=Ui(j)

（7） m 增一，如果 m 等于 bound 则跳到(8)，否则跳到（6）

（8） j 增一，如果 j 等于 dim(Ui)则搜索结束，否则跳到（2）

1.2.15.3.5 Err[i]的更新

对每一子帧，计算得到闭环基音周期 L_i 和自适应码本增益索引 Ic 以后需要更新 Err[i]，以便用于下一子帧的 iTest 参数的计算。其更新的方法如下所述。

（1）更新 $Err[i], i = 2,3,4$

$$Err[i] = Err[i-2], i = 2,3,4$$

（2）计算新的 $Err[0]$ 和 $Err[1]$

一阶增益表 tabgain 有两个选择：170 个记录的表 tabgain170 和 85 个记录的表 tabgain85。它们分别和自适应码本增益表 AcbkGainTable170 和 AcbkGainTable085 对应。tabgain170 和 tabgain85 的选择方法和码本的选择方法一样。

如果计算得到的第 i 帧的闭环基音周期 $L_i \le 30$，则

$$Err[1] \Leftarrow Err[0] \Leftarrow \text{tabgain[Ic]} \cdot Err[0] + Err0$$

$$Err0 = 0.00000381464$$

否则计算 L_i 所在的区间号（区间定义见表 1－1），计算区间号可用以下公式

$$zone = \left[\frac{L_i \cdot 1092}{32768} \right]$$

如果 L_i 在区间的分界线上，即

$$L_i = 30(zone + 1)$$

则

$$Err[0] \Leftarrow \text{tabgain[Ic]} \cdot Err[0] + Err0$$
$$Err[1] \Leftarrow \text{tabgain[Ic]} \cdot Err[1] + Err0$$

否则如果 $zone \neq 1$

$$Err[0] \Leftarrow \max\{\text{tabgain[Ic]} \cdot Err[j] + Err0\}, j = zone - 2, zone - 1$$
$$Err[1] \Leftarrow \max\{\text{tabgain[Ic]} \cdot Err[j] + Err0\}, j = zone - 2, zone - 1, zone$$

如果 $zone = 1$

$$Err[1] \Leftarrow Err[0] \Leftarrow \max\{\text{tabgain[Ic]} \cdot Err[j] + Err0\}, j = 0,1$$

（3）调整 $Err[0]$ 和 $Err[1]$

如果

$$Err[j] > 256$$

则

$$Err[j] = 256$$

这里 j 等于 0 或者 1。

1.2.15.4 自适应码本激励 u[n]的计算

对于求得的 L_i 用式（1.2.15.1）计算重建激励 $e'[n]$。

设第 i 子帧的自适应码本激励用

$$u[n], n = 0...59$$

表示，其计算方法是

$$u[n] = \sum_{j=0}^{4} e'(n+j) \cdot AcbkGainTable[Ic][j], n = 0...59$$

设基音预测信号 $p[n]$ 是 $u[n]$ 通过联合滤波器以后的响应，即

$$p[n] = \sum_{j=0}^{n} u[j] \cdot h[n-j], n = 0...59$$

从目标信号（target vector） $t[n]$ 中减去基音预测信号（contribution of thepitch predictor）

$p[n]$ 得到残余信号(residual signal) $r[n]$

$$r[n] = t[n] - p[n], n = 0...59$$

1.2.16 固定码本激励

t[n]中除去了自适应码本激励 u[n]的贡献得到了 r[n]，那么 r[n]将由固定码本激励信号 v[n]通过联合滤波器来逼近。根据使用的码率的不同（不同的帧可以选用不同的码率，取决于用户的设置），固定码本激励部分分为高码率激励（使用 MP-MLQ）和低码率激励（使用 ACELP）

1.2.16.1 高码率激励（MP-MLQ）

如果选用了 6.3Kbit/s 的传送速率，则使用高码率激励。

高码率激励的目标是用 $r'(n), n = 0...59$ 逼近 $r(n)$，其中 $r'(n)$ 为 $v(n)$ 通过联合滤波器以后的输出，定义

$$v(n) = G \cdot \sum_{k=0}^{M-1} \alpha_k \delta(n - m_k)$$

其中 G 表示固定码本增益， α_k 表示脉冲符号

$$\alpha_k \in \{1,-1\}$$

M 是脉冲的个数，对偶子帧 M 为 6，对奇子帧 M 为 5。脉冲位置 m_k 要么全为偶数，即

$$m_k \in \{n \mid n = 0,2,4\cdots 58\}, k = 0...M-1$$

要么全为奇数，即

$$m_k \in \{n \mid n = 1,3,5\cdots 59\}, k = 0...M-1$$

高码率激励的任务就是求最佳的 G 、 α_k 和 m_k

1.2.16.1.1 脉冲响应的调整

设高码率激励中使用的脉冲响应为 $h_A(n)$ 。当当前子帧对应的偶子帧的闭环基音周期大于等于 58 时，即

$$L_{i/2} \geq 58, i = 0,1,2,3$$

时

$$h_A(n) = h(n), n = 0...59$$

否则， $h_A(n)$ 有两种可能的取值

$$h_A(n) = h(n), n = 0...59$$

或者

$$h_A(n) = h_T(n), n = 0...59$$

其中 $h(n)$ 为 $\delta(n)$ 通过联合滤波器的响应。 $h_T(n)$ 是用脉冲串代替单脉冲通过联合滤波器后的响应。即，用周期的脉冲序列

$$\delta_T(n) = \sum_{k=0}^{\infty} \delta(n - kL_{i/2}), i = 0,1,2,3$$

代替 $\delta(n)$ 通过联合滤波器。其响应为

$$h_T(n) = \sum_{k=0}^{\infty} h(n - kL_{i/2}), i = 0,1,2,3, n = 0...59$$

$h_A(n)$ 到底选择 $h(n)$ 还是 $h_T(n)$ ，由两者中哪个更优决定，更优的判断条件是哪个 $h_A(n)$ 求得的最佳的增益 G 、脉冲符号 α_k 和脉冲位置 m_k 对应的误差

$$err^2(n) = [r(n) - G \cdot \sum_{k=0}^{M-1} \alpha_k h_A(n - m_k)]^2 = [r(n) - r'(n)]^2$$

更小。如果采用脉冲串调整 $Trn = 1$ ，否则 $Trn = 0$ ， Trn 是高码率激励编码信息的一部分。

1.2.16.1.2 脉冲响应$h_A(n)$的自相关系数$R_h(n)$的计算

方法如下：

$$R_h(n)=\sum_{k=0}^{59-n}h_A(k)\cdot h_A(n+k),n=0...59$$

1.2.16.1.3 冲击响应$h_A(n)$和残余信号$r(n)$的互相关$d(n)$的计算

方法如下：

$$d(n)=\sum_{k=0}^{59-n}h_A(k)\cdot r(n+k),n=0...59$$

1.2.16.1.4 最佳的增益G、脉冲符号α_k和脉冲位置m_k的搜索

方法如下：

（1） 首先选择$h_A(n)=h(n),n=0...59$

（2） 奇偶位置分别搜索

脉冲位置要么全部为奇数位置要么全部为偶数位置。搜索最佳位置，需要分别进行全奇位置搜索和在全偶位置搜索。设用$\sigma=0$表示是偶位置搜索；$\sigma=1$表示是奇位置搜索。于是奇位置搜索和偶位置搜索的脉冲位置统一表示为

$$n=2\cdot k+\sigma,k=0...29$$

（3） 对于每个σ，计算第一个脉冲的位置m_0，m_0是互相关最大的位置，即

$$d(m_0)=\max\{d(n)\},n=2\cdot k+\sigma,k=0...29$$

（4） 利用$d(m_0)$求$\tilde{G}_{\max}$

定义

$$G_{\max}=\frac{d(m_0)}{R_h(0)}$$

$G_{\max}$的量化值即为$\tilde{G}_{\max}$，$G_{\max}$的量化值已经被存于表 FcbkGainTable 中。

FcbkGainTable 是一个有 24 个元素的表，表中的元素从小到大排列，相邻元素之间基本上按照 3.2dB 的大小增加，而具体的取值，目前不知道计算公式，可能是从实验得到的。量化的方法是，使式

$$\left|FcbkGainTable[j]\cdot R_h(0)-d(m_0)\right|,j=2,3,4...22$$

取得最小值的 j 定义为$j_{\min}$，则$\tilde{G}_{\max}$定义为

$$\tilde{G}_{\max}=FcbkGainTable[j_{\min}]$$

（5） 利用$\tilde{G}_{\max}$确定G的取值范围

G 的取值范围是

$$\left[\tilde{G}_{\max}-3.2dB,\tilde{G}_{\max}+6.4dB\right]$$

即取值可以为

$$FcbkGainTable[j], j\in\{j_{\min}-1, j_{\min}, j_{\min}+1, j_{\min}+2\}$$

批注 [d1]: G 的范围修改为 -3.2db~+6.4db

（6） 对于G的每个取值，寻找剩余的 M－1 个脉冲的位置和符号

寻找的过程为

1）初始化：设有$\{d_r(n)\}_{n=0...59}$，初始时，

$$d_r(n)=d(n), n=2\cdot k+\sigma, k=0...29$$

设有当前所求的脉冲是第 i 个脉冲，初始时设 i=0

2）第 i 个脉冲脉冲的符号

$$\alpha_i=\begin{cases}1 & ,d_r(m_i)\ge 0\\ -1 & ,d_r(m_i)<0\end{cases}$$

3）i 增一。如果 i＝M，则退出，否则进入 4）

4）更新$d_r(n)$

$$d_r(n)\Leftarrow d_r(n)-G\cdot\alpha_{i-1}\cdot R_h(|n-m_{i-1}|), n=2\cdot k+\sigma, k=0...29, n\ne m_l, l<i$$

5）第 i 个脉冲的位置m_i是使得

$$|d_r(n)|, n=2\cdot k+\sigma, k=0...29, n\ne m_l, l<i$$

取得最大值的 n，m_i不能取已经取过脉冲的位置。

6）跳到 2）

（7） 对于每个σ和每个G比较

$$err^2(n)=[r(n)-G\cdot\sum_{k=0}^{M-1}\alpha_k h_A(n-m_k)]^2=[r(n)-r'(n)]^2=r^2(n)-2\cdot r(n)\cdot r'(n)+r'(n)$$

由于$r^2(n)$相同，实际上只用比较$-2\cdot r(n)\cdot r'(n)+r'(n)$，使得$err^2(n)$取得最小值的$\sigma$和$G$以及由$\sigma$和$G$求得的$\alpha_k$和$m_k$即为高码率激励要求的参数。

（8） 选择$h_A(n)=h_T(n), n=0...59$，跳到（2）目的是比较$h_A(n)$选择$h(n)$和$h_T(n)$那个$err^2(n)$更小。若$h_T(n)$的$err^2(n)$更小，则$Trn=0$。最小的$err^2(n)$对应的σ、G、α_k和m_k被选用。令$G=FcbkGainTable[I_F]$，I_F就是固定码本增益索引。

1.2.16.1.5 高码率激励的脉冲位置和符号的编码

（1） 脉冲位置的编码

脉冲位置使用$\binom{30}{M}$组合编码，其中对偶子帧 M＝6，对于奇子帧 M＝5。因为脉冲位置为全奇或者全偶，所以是在总共的 30 个可能中选择 M 个位置，可能的组合总个数为

$$\binom{30}{M}=\begin{cases}593775=0\text{x}90\text{F}6\text{F} & ,\text{M}=6\\ 142506=0\text{x}22\text{CAA} & ,\text{M}=5\end{cases}$$

分别用 20bit 和 18bit 编码。如何用 20bit 或 18bit 数映射到$\binom{30}{M}$种组合呢？

有 CombinatorialTable 表是一个 6×30 的二维表。其中的元素是

$$\text{CombinatorialTable[i][j]}=\begin{cases}\binom{29-\text{j}}{5-\text{i}} & ,29-\text{j}\ge 5-\text{i}\\ 0 & ,29-\text{j}<5-\text{i}\end{cases}$$

设在 30 个位置中，脉冲位置为 p_i 有

$$0\le p_0<p_1<\cdots<p_{M-1}\le 29$$

为了方便表示设 $p_{-1}=-1$，则脉冲位置编码为

$$C_{pos}=\sum_{i=0}^{M-1}\sum_{j=p_{i-1}+1}^{p_i-1}combinatorialTable[6-M+i][j]$$

注，如果 $p_{i-1}+1>p_i-1$则$\sum_{j=p_{i-1}+1}^{p_i-1}\binom{29-j}{5-i}=0$

下面证明以上编码方法的有效性：

1）证明$C_{pos}<\binom{30}{M}$

以偶子帧（M＝6）为例

$$C_{pos}=\sum_{i=0}^{5}\sum_{j=p_{i-1}+1}^{p_i-1}combinatorialTable[i][j]=\sum_{i=0}^{5}\sum_{j=p_{i-1}+1}^{p_i-1}\binom{29-j}{5-i}$$

$$=\sum_{i=0}^{4}\sum_{j=p_{i-1}+1}^{p_i-1}\binom{29-j}{5-i}+\sum_{i=5}^{5}\sum_{j=p_4+1}^{p_5-1}\binom{29-j}{0}\qquad(1.2.16.1)$$

根据组合数的性质

$$\binom{M+1}{n+1}=\binom{M}{n}+\binom{M-1}{n}+\cdots+\binom{n}{n}$$

有

$$\sum_{i=5}^{5}\sum_{j=p_4+1}^{p_5-1}\binom{29-j}{0}<\sum_{j=p_4+1}^{29}\binom{29-j}{0}=\binom{29-p_4}{1}$$

代入式(1.2.16.1)得

$$C_{pos}<\sum_{i=0}^{3}\sum_{j=p_{i-1}+1}^{p_i-1}\binom{29-j}{5-i}+\sum_{j=p_3+1}^{p_4}\binom{29-j}{1}$$

同理可得

$$\sum_{j=p_3+1}^{p_4}\binom{29-j}{1}\le\sum_{j=p_3+1}^{29-1}\binom{29-j}{1}=\binom{29-p_3}{2}$$

又

$$C_{pos}=\sum_{i=0}^{3}\sum_{j=p_{i-1}+1}^{p_i-1}\binom{29-j}{5-i}+\sum_{i=4}^{5}\sum_{j=p_{i-1}+1}^{p_i-1}\binom{29-j}{5-i}$$

所以

$$\sum_{i=4}^{5}\sum_{j=p_{i-1}+1}^{p_i-1}\binom{29-j}{5-i}<\binom{29-p_3}{2}$$

由此可得通式

$$\sum_{i=k}^{5}\sum_{j=p_{i-1}+1}^{p_i-1}\binom{29-j}{5-i}<\binom{29-p_{k-1}}{6-k},k=0...5 \qquad (1.2.16.2)$$

当 k＝0 时，可得

$$C_{pos}<\binom{30}{6}$$

对于奇子帧（M＝5）（1.2.16.2）式变为

$$\sum_{i=k}^{4}\sum_{j=p_{i-1}+1}^{p_i-1}\binom{29-j}{5-1-i}<\binom{29-p_{k-1}}{6-1-k},k=0...4$$

当 k＝0 时，可得

$$C_{pos}<\binom{30}{5}$$

总之

$$\sum_{i=k}^{M-1}\sum_{j=p_{i-1}+1}^{p_i-1}\binom{29-j}{5-(6-M)-i}<\binom{29-p_{k-1}}{6-(6-M)-k},k=0...M-1 \qquad (1.2.16.3)$$

$$C_{pos}<\binom{30}{M}$$

2）解码时能够唯一地解码得到原位置信息 $p_i,i=0...M-1$

分析求C_{pos}的过程的式(1.2.16.1)，其求解过程说明如下：如图 1－6 所示，脉冲位置p_i将 0～29 分为 M＋1 个开区间（不包括脉冲位置所在的点），其中左边 0～M－1 区间中的所有点都对应一个$C_{posij} = combinatorialTable[i][j]$，i 和 j 的取值如图 1－6 所示。有

$$C_{pos} = \sum C_{posij}$$

式(1.2.16.3)可以进一步理解为：脉冲 i 所在位置p_i若对应值$combinatorialTable[i][p_i]$则$combinatorialTable[i][p_i]$，将大于$p_i$右边所有点的$C_{posij}$之和。

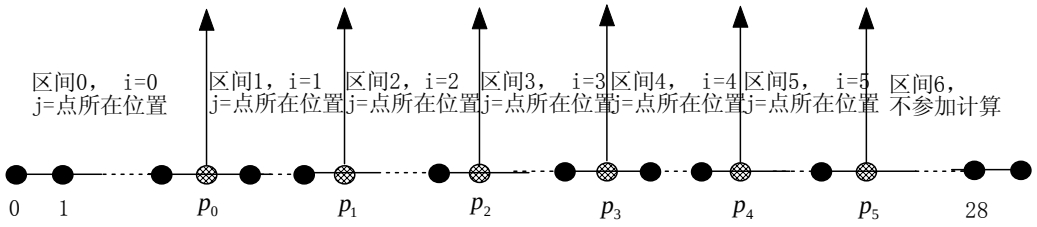

图 1-6 脉冲位置编码示意图（以偶子帧为例）

当已经知道了C_{pos}以后，如何唯一地确定脉冲的位置？其计算步骤如下：

1）初始时，设区间号 i＝0，位置号 j＝0，C＝0

2）$C \Leftarrow C + combinatorialTable[i][j]$

3）如果$C > C_{pos}$，则$p_i \Leftarrow j$，$C \Leftarrow C - combinatorialTable[i][j]$，i 增一。证明如下：如果$j < p_i$则必然有$C \le C_{pos}$；当 j 等于$p_i$根据式(1.2.16.3)$combinatorialTable[i][p_i]$大于$p_i$右边所有点的$C_{posij}$之和，而 C 加上$p_i$右边所有点的$C_{posij}$即是$C_{pos}$。所以当 j 增加时，第一次出现$C > C_{pos}$的点一定为$p_i$。

4）如果 i＝M 则退出，否则 j 增一，跳到 2）

（2） 脉冲符号编码

脉冲符号用 Mbit 编码，1 对应正脉冲，0 对应负脉冲，并且比特位越高对应的脉冲越靠近图 1－6 的右方。

1.2.16.2 低码率激励（ACELP）

如果选用了 5.3Kbit/s 的传送速率，则使用低码率激励。

低码率激励的目的是寻找激励信号$v(n), n = 0...59$使得它通过联合滤波器以后的响应$y(n), n = 0...59$，接近$r(n)$，设

$$v(n)=\tilde{G}\cdot\sum_{i=0}^{3}S_{Opti}\delta(n-m_{Opti}),n=0...59$$

可见激励信号由四个脉冲信号组成，其中$\tilde{G}$为固定码本增益，S_{Opti}为各个脉冲的符号，m_{Opti}为各个脉冲的位置。四个脉冲的可选位置如表 1—2 所示

表 1—2 脉冲位置选择

脉冲号	脉冲位置	脉冲的可能位置							
1	m0	0	8	16	24	32	40	48	56
2	m1	2	10	18	26	34	42	50	58
3	m2	4	12	20	28	36	44	52	(60)
4	m3	6	14	22	30	32	46	54	(62)

表中带括号的“（60）”和“（62）”，是为了统一计算起见添加的，也就是要将原 60 个位置扩充为 64 个位置。另外这个表格是偶位置表，脉冲的位置也可以为奇，但是 4 个脉冲只能全奇或者全偶。从表中可知，每个脉冲可选 8 个偶位置。如何选择脉冲位置，有以下判断方法：求脉冲位置，使得

$$\frac{C^2}{\varepsilon}=\frac{(d^TV)}{V^T\Phi V}$$

达到最大值。其中，设

$$H=\begin{pmatrix}h_A(0) & 0 & \cdots & 0\\ h_A(1) & h_A(0) & \cdots & 0\\ \vdots & \vdots & \ddots & \vdots\\ h_A(59) & h_A(58) & \cdots & h_A(0)\end{pmatrix}$$

其中$h_A(n)$是调整以后的联合滤波器冲击响应（如何调整后面将述及）

$$d=H^Tr=\begin{pmatrix}\sum_{j=0}^{59}r(j)\cdot h_A(j)\\ \sum_{j=0}^{58}r(1+j)\cdot h_A(j)\\ \vdots\\ \sum_{j=0}^{0}r(59+j)\cdot h_A(j)\end{pmatrix}$$

$$\Phi=H^TH$$

$$V=\begin{pmatrix}Cod(0)\\ Cod(1)\\ \vdots\\ Cod(59)\end{pmatrix}$$

由于V中只有 4 个元素不为 0，设四个脉冲位置分别为m_0、m_1、m_2、m_3，用$sg(m_i)$表示脉冲在m_i位置时应取的符号。则

$$C = (d^T V)$$
$$= sg(m_0)\cdot d(m_0 + shift) + sg(m_1)\cdot d(m_1 + shift) + sg(m_2)\cdot d(m_2 + shift) + sg(m_3)\cdot d(m_3 + shift)$$

其中 shift 用 0 和 1 表示采用全偶位置还是全奇位置。

$$\varepsilon = V^T \Phi V = \Phi(m_0, m_0)$$
$$+ \Phi(m_1, m_1) + 2sg(m_0)\cdot sg(m_1)\cdot \Phi(m_0, m_1)$$
$$+ \Phi(m_2, m_2) + 2[sg(m_0)\cdot sg(m_2)\cdot \Phi(m_0, m_2) + sg(m_1)\cdot sg(m_2)\cdot \Phi(m_1, m_2)]$$
$$+ \Phi(m_3, m_3) + 2[sg(m_0)\cdot sg(m_3)\cdot \Phi(m_0, m_3) + sg(m_1)\cdot sg(m_3)\cdot \Phi(m_1, m_3) + sg(m_2)\cdot sg(m_3)\cdot \Phi(m_2, m_3)]$$

其中 $\Phi(m_i, m_j)$ 是矩阵 Φ 的元素。这里没有 shift，也就是说奇位置的 ε 用偶位置代替。

求低码率激励的目的就是求 $\tilde{G}$、S_{Opti}、m_{Opti}、shift。

1.2.16.2.1 一阶基音预测参数的计算

一阶基音预测将用于调整冲击响应。一阶基音预测参数包括基音周期 T_0 和增益 gt，由下式求得

$$T_0 = L_i + epsi170(I_c), i = 0,1,2,3$$

$$gt = gain170[I_c]$$

其中 I_c 为已经求得的自适应码本索引，epsi170 是一个有 170 个元素的表，表中的元素表示基音周期值小量的波动值，gain170 是一个有 170 个元素的表，表中存放一阶基音增益值。

1.2.16.2.2 冲击响应的调整

将一阶基音预测的影响加入到联合滤波器的冲击响应 $h(n)$ 中得到 $h_A(n)$

$$h_A(n)\begin{cases} h(n) + h(n - T_0) & gt < 58 \\ h(n) & gt \geq 58 \end{cases}, n = 0...59$$

其中当 $n < 0$ 时，$h(n) = 0$

1.2.16.2.3 冲击响应 $h_A(n)$ 的自相关矩阵 Φ 的计算

计算调整以后的冲击响应 $h_A(n)$ 的自相关矩阵 Φ。Φ 为（60×60）的角对称矩阵，其上三角矩阵中的元素表示为 $\Phi(i, j), j \geq i$。不是对所有的 i，j 的取值都计算，只计算

$$\Phi(m_\alpha, m_\beta), \alpha = 0...3, \beta = 0...3, \beta \geq \alpha$$

$\beta \geq \alpha$ 是因为计算 ε 的公式中 $\beta \geq \alpha$。所以最后需要计算的值为 $\Phi(m_0, m_0)$（8 个），$\Phi(m_1, m_1)$（8 个），$\Phi(m_2, m_2)$（8 个），$\Phi(m_3, m_3)$（8 个），$\Phi(m_0, m_1)$（64 个），$\Phi(m_0, m_2)$（64 个），$\Phi(m_0, m_3)$（64 个），$\Phi(m_1, m_2)$（64 个），$\Phi(m_1, m_3)$（64 个），$\Phi(m_2, m_3)$（64 个）。

$\Phi(i, j)$ 的计算公式为

$$\Phi(i, j) = \sum_{n=j}^{59} h_A[n-i] \cdot h_A[n-j], j \geq i$$

为了简化计算注意使用各个元素之间的累加继承性关系。

1.2.16.2.4 冲击响应 $h_A(n)$ 和残余信号 $r(n)$ 的互相关矩阵 d 的计算

公式如下：

$$d(n) = \sum_{j=0}^{59-n} r(n+j) \cdot h_A(j), n = 0...59$$

1.2.16.2.5 最佳的脉冲位置及其符号的搜索

方法如下：

（1） 在 $d(n)$ 后补充 4 个点，该 4 个点的值全为 0，$d(n)$ 的点数增加到 64 个

（2） 脉冲位置和脉冲符号的关系

当脉冲的位置固定以后，它的符号也就相应确定了。定义 j 位置对应的脉冲符号

$$s(j) \in \{1, -1\}, j = 0...31$$

如果

$$d(2j) + d(2j+1) \geq 0$$

则

$$s(j) = 1$$

否则

$$s(j) = -1$$

为了以后计算方便，定义带符号的互相关 $d'(i)$

$$d'(i) = s(\left[\frac{i}{2}\right]) \times d(i), i = 0...63$$

（3） 计算门限

门限值将作为以后搜索的判定条件，如果要求的值小于门限则没有必要进一步搜索。

门限求法如下：设

$$M_{Odd0} = \max\{d'(8j)\}, j = 0...7$$

$$M_{Odd1} = \max\{d'(8j+2)\}, j = 0...7$$

$$M_{Odd2} = \max\{d'(8j+4)\}, j = 0...7$$

$$M_{Odd} = M_{Odd0} + M_{Odd1} + M_{Odd2}$$

$$M_{OddMean} = \frac{1}{8}\sum_{j=0}^{7} d'(8j) + d'(8j+2) + d'(8j+4)$$

则

$$thr_3 = M_{OddMean} + \frac{(M_{Odd} - M_{OddMean})}{2}$$

又设

$$M_{Even0} = \max\{d'(8j+1)\}, j = 0...7$$

$$M_{Even1} = \max\{d'(8j+3)\}, j = 0...7$$

$$M_{Even2} = \max\{d'(8j+5)\}, j = 0...7$$

$$M_{Even} = M_{Even0} + M_{Even1} + M_{Even2}$$

$$M_{EvenMean} = \frac{1}{8}\sum_{j=0}^{7} d'(8j+1) + d'(8j+3) + d'(8j+5)$$

则

$$\max_1 = M_{EvenMean} + \frac{(M_{Even} - M_{EvenMean})}{2}$$

最后门限值为

$$thres = \max\{thr_3, \max_1\}$$

（4） 为了方便计算定义带符号的相关矩阵Φ'

只用修改$\Phi(m_0, m_1)$（64 个），$\Phi(m_0, m_2)$（64 个），$\Phi(m_0, m_3)$（64 个），$\Phi(m_1, m_2)$（64 个），$\Phi(m_1, m_3)$（64 个），$\Phi(m_2, m_3)$（64 个）

$$\Phi'(i, j) = s(\frac{i}{2}) \cdot s(\frac{j}{2}) \cdot \Phi(i, j), i = m_\alpha, j = m_\beta, \alpha \neq \beta$$

（5） 搜索最佳的脉冲位置

搜索步骤如下

1) 初始化：

四个脉冲的最佳位置

$$m_{Opt0}=0, m_{Opt1}=2, m_{Opt2}=4, m_{Opt3}=6$$

初始时最佳脉冲位置全部为偶位置，即位置标志 $shift_{Opt}=0$。

最佳脉冲位置时的互相关以及能量

$$C_{Opt}=0, \varepsilon_{Opt}=1.0$$

另外设 $time=120+extra$

2）搜索“前三个脉冲”的位置 m_0、 m_1、 m_2

定义
前三个脉冲的能量

$$\varepsilon' = \Phi'(m_0,m_0) + \Phi'(m_1,m_1)+2\Phi'(m_0,m_1) + \Phi'(m_2,m_2)+2[\Phi'(m_0,m_2)+\Phi'(m_1,m_2)]$$

如果前三个脉冲位置的组合得到的

$$\varepsilon' > thres$$

则进入 3），否则继续寻找其它组合，直到搜索全部可能后退出

3）搜索第四个脉冲的位置 m_3

如果

$$d'(m_0)+d'(m_1)+d'(m_2) < d'(m_0+1)+d'(m_1+1)+d'(m_2+1)$$

脉冲位置采用奇位置 $shift=1$ 否则采用偶位置 $shift=0$

有

$$C = d'(m_0+shift)+d'(m_1+shift)+d'(m_2+shift)+d'(m_3+shift)$$

定义四脉冲总能量

$$\varepsilon = \Phi'(m_0,m_0) + \Phi'(m_1,m_1)+2\Phi'(m_0,m_1) + \Phi'(m_2,m_2)+2[\Phi'(m_0,m_2)+\Phi'(m_1,m_2)] + \Phi'(m_3,m_3)+2[\Phi'(m_0,m_3)+\Phi'(m_1,m_3)+\Phi'(m_2,m_3)]$$

如果

$$\frac{C^2}{\varepsilon} > \frac{C_{Opt}^2}{\varepsilon_{Opt}}$$

首先，更新最优脉冲参数

$$m_{Opt0} \Leftarrow m_0, m_{Opt1} \Leftarrow m_1, m_{Opt2} \Leftarrow m_2, m_{Opt3} \Leftarrow m_3$$

$$shift_{Opt} \Leftarrow shift, C_{Opt} \Leftarrow C, \varepsilon_{Opt} \Leftarrow \varepsilon$$

然后，将“搜索次数”time 减一，如果

$$time \le 0$$

则退出。

跳到 2）

time 的引入是为了控制每次的搜索次数。以上搜索步骤中出现一个量 *extra*，它表示了上一子帧搜索次数的剩余，将这剩余的次数留给本子帧搜索用（将本次搜索的最大值增加 *extra*）。*extra* 在每帧语音编码前初始化为 120。在每子帧的“最佳的脉冲位置”搜索完毕以后更新为 $extra = time$，表示本次搜索的剩余次数，以留给下一子帧使用。

（6） 记录脉冲符号

搜索到了四个脉冲的最佳位置以后，需要记录脉冲的符号

$$S_{Opt0} = s(\frac{m_{Opt0}}{2}), S_{Opt1} = s(\frac{m_{Opt1}}{2}), S_{Opt2} = s(\frac{m_{Opt2}}{2}), S_{Opt3} = s(\frac{m_{Opt3}}{2})$$

（7） 计算四个脉冲信号通过联合滤波器的响应 $y(n)$

$$y(n) = S_{Opt0} \cdot h_A(n - m_{Opt0}) + S_{Opt1} \cdot h_A(n - m_{Opt1})$$

其中 $h_A(n)$ 是调整以后的联合滤波器脉冲响应，当 n<0 时 $h_A(n) = 0$。如果第三个脉冲的位置有效，即 $m_{Opt2} < 60$，则在 $y(n)$ 中增加第三个脉冲的贡献

$$y(n) \Leftarrow y(n) + S_{Opt2} \cdot h_A(n - m_{Opt2})$$

如果第四个脉冲的位置有效，即 $m_{Opt3} < 60$，则在 $y(n)$ 中增加第四个脉冲的作用

$$y(n) \Leftarrow y(n) + S_{Opt3} \cdot h_A(n - m_{Opt3})$$

（8） 对低码率激励信号参数编码

1) 格点 $shift_{Opt}$，也就是使用奇位置还是偶位置，需要 1bit 编码

2) 脉冲符号 S_{Opt0}、S_{Opt1}、S_{Opt2}、S_{Opt3} 需要 4bit 编码，编码方式：用 1 表示正脉冲，用 0 表示负脉冲，比特从低到高分别表示 S_{Opt0}、S_{Opt1}、S_{Opt2}、S_{Opt3}。

3) 脉冲位置 m_{Opt3}、m_{Opt2}、m_{Opt1}、m_{Opt0} 需要用 12bit 编码，编码方式：$\frac{m_{Opti}}{8}, i = 0,1,2,3$

占用 3bit，对于 m_{Opti} 总共只有 8 种选择，如果知道了 $\frac{m_{Opti}}{8}$ 就可以确定 m_{Opti} 的值，4 个 $\frac{m_{Opti}}{8}$ 共需 12bit，4 个 $\frac{m_{Opti}}{8}$ 的排列顺序从高到低分别是 $\frac{m_{Opt3}}{8}$、$\frac{m_{Opt2}}{8}$、$\frac{m_{Opt1}}{8}$、$\frac{m_{Opt0}}{8}$

1.2.16.2.6 固定码本增益的计算

和高码率一样，固定码本增益 G 通过表 FcbkGainTalbe 量化，也就是寻找最佳的 I_F 使得 FcbkGainTalbe[I_F]接近 G。方法为：

（1）如果

$$\sum_{n=0}^{59} r(n)\cdot y(n) \le 0$$

其中 $r(n)$ 为残余信号， $y(n)$ 为低码率激励四个脉冲信号通过联合滤波器的响应，则 $I_F = 0$，量化以后的码本增益 $\tilde{G} = FcbkGainTable[I_F]$。

否则进入（2）

（2）如果

$$\sum_{n=0}^{59} y(n)\cdot y(n) = 0$$

码本增益量化以前的值

$$G = 0$$

否则

$$G = \frac{\sum_{n=0}^{59} r(n)\cdot y(n)}{\sum_{n=0}^{59} y(n)\cdot y(n)}$$

搜索 FcbkGainTable 表，使得 $FcbkGainTable[I_F]$ 最接近 G（两者之差的绝对值最小），则量化以后的码本增益 $\tilde{G} = FcbkGainTable[I_F]$，码本索引为 I_F

1.2.16.2.7 最终的低码率激励信号 $v(n), n = 0...59$ 的获得

方法如下：

（1） 由四个非零脉冲组成的低码率激励信号为：

$$v(n)=\begin{cases} S_{Opt0}\cdot\tilde{\tilde{G}} & n=m_{Opt0} \\ S_{Opt1}\cdot\tilde{\tilde{G}} & n=m_{Opt1} \\ S_{Opt2}\cdot\tilde{\tilde{G}} & n=m_{Opt2},if \quad m_{Opt2}<60 \\ S_{Opt3}\cdot\tilde{\tilde{G}} & n=m_{Opt3},if \quad m_{Opt3}<60 \\ 0 & else \end{cases}$$

注意如果脉冲位置（主要对第三个脉冲和第四个脉冲来说）大于等于 60，则该脉冲为无效脉冲，不加入 $v(n)$ 中。但是编码信息传送到解码器端时，仍然传送这个脉冲的信息。解码时无效位置的脉冲不会在激励信号 v[n]中出现。

（2） 对激励信号进行一阶基音修正

如果一阶基音预测周期

$$T_0<58$$

进行如下调整

$$v(n)\Leftarrow v(n)+gt\cdot Cod(n-T_0),n=T_0\cdots 59$$

其中 gt 为一阶基音预测增益。其中 $n-T_0$ 小于 0 的点，$Cod(n-T_0)=0$，所以不必进行调整。

1.2.17 联合滤波器状态的更新

至此 G.723.1 主算法所要传送的参数已经全部计算出来。这里将更新联合滤波器的状态 RingFir[n]、PreErr[n]和激励 PreExc[n]。

（1） 更新 PreExc[n]。为下一子帧重建激励做好准备。更新的方法是：PreExc 中的点左移一个子帧，即 PreExc[i]=PreExc[i+60]，i=0…(145-60-1)。PreExc 的后 60 个点更新为此子帧的激励信号 e[n]，即 PreExc[145-60+i]=e[i]=u[i]+v[i],i=0…59。限制新增加的点 PreExc[i],i=(145-60)…(145-1)的幅度：如果幅度小于-32767.5 则令其为-32768.0，如果幅度大于 32766.5 令其为 32767.0。

（2） 更新联合滤波器状态。更新联合滤波器状态为当前子帧的激励通过以后的状态，这个状态将用于计算下一子帧的联合滤波器零输入响应。更新方法如下：参考图 1－4（联合滤波器冲击响应的计算），设 $\tilde{A}_i(z)$ 的输出为 xdl[n]，$W_i(z)$ 的输出为 zdl[n]。

将上一子帧的联合滤波器状态 RingFir[n]、PreErr [n]赋给 xdl[n]和 zdl[n]作为此次滤波的初始状态，即 xdl[i]=RingFir[i],i=0…9，zdl[i]=PreErr[i],i=0…144。联合滤波器的输入信号为此子帧的激励信号 e[n],n=0…59。参考“联合滤波器冲击响应的计算”部分的计算方法，让 e[n]通过联合滤波器。然后，将 RingFir[i]和 PreErr[i]更新为 e[n]通过滤波器以后的联合滤波器状态，即 RingFir[i]=xdl[i],i=0…9，PreErr[i]=zdl[i],i=0…144。

1.2.18 发送信息打包

发送信息打包的任务是将编码器得到的某帧的编码信息打包成数据包，以便传送到解码端。G.723.1 和其 Annex A 将帧分为三种类型：活动帧、静音帧和不传送帧。以上

讲述的都是活动帧的编码方法，关于静音帧和不传送帧的检测和编码方法在 Annex A 中描述。打包后的数据包图 1－7、图 1－8、图 1－9、图 1－10 所示（其中的符号命名参考了 C 代码中的变量名）：

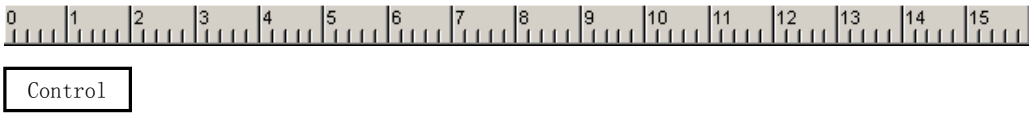

Figure1-7. No Imfomation Transmited Packet

Figure1-8. Sid Packet

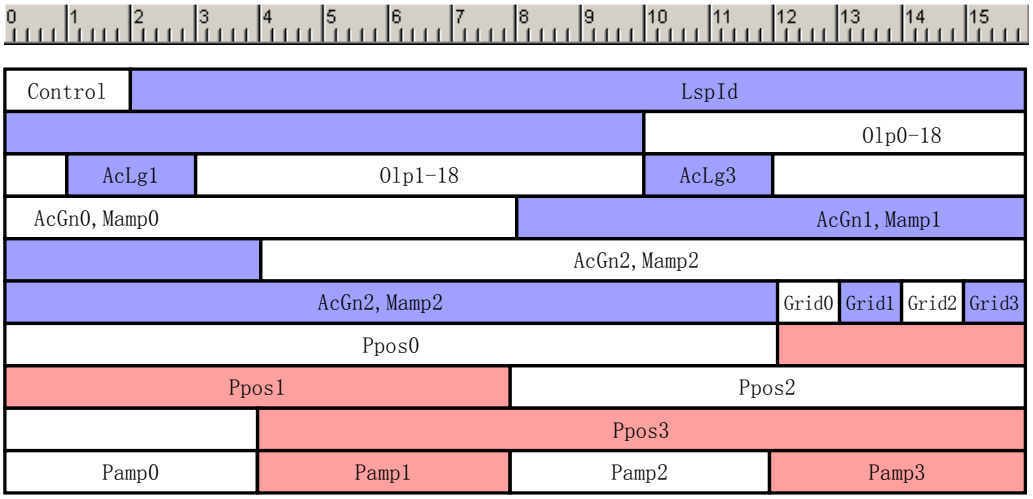

Figure1-10. 5.3K/s Packet

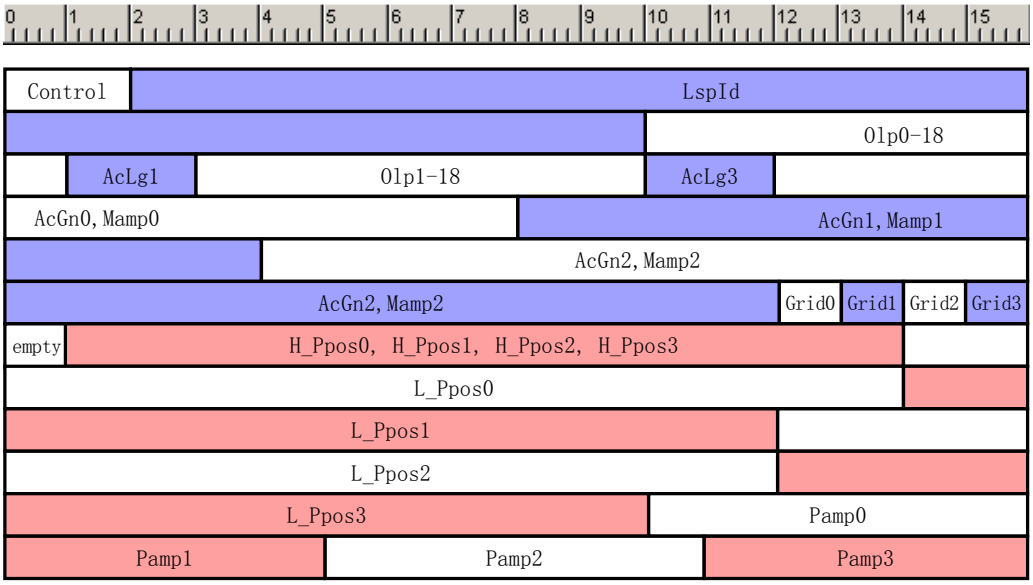

Figure1-11. 6.3K/s Packet

不管是那种帧数据包开头都有 2bit 的控制信息，表 1－3 为所有帧的共同部分：

表 1－3 所有的帧的共同部分

比特分配	该部分的功能
0～1	控制信息（本帧的码率和帧类型）（2bit）

其中：

- 0～1：控制信息（本帧的码率和帧类型）（2bit）

表 1－4 为控制信息类型：

表 1－4 控制信息类型

帧类型(Ftyp)	码率	Bit1,0	含义
1	Rate63	00	活动帧（高码率编码）
1	Rate53	01	活动帧（低码率编码）
2	X	10	静音帧
0	X	11	不传送帧

1.2.18.1 活动帧共同部分数据包格式

表 1－5 为高码率和低码率编码共同部分：

表 1－5 高码率和低码率编码共同部分

比特分配	该部分的功能
2～25	整帧的 LSP 索引（24bit）
26～32	子帧 0 的闭环基音周期减 18（7bit）
33～34	子帧 1 的闭环基音周期和子帧 0 的闭环基音周期的关系（2bit）
35～41	子帧 2 的闭环基音周期减 18（7bit）
42～43	子帧 3 的闭环基音周期和子帧 2 的闭环基音周期的关系（2bit）
44～55	子帧 0 的两个码本增益的联合编码（12bit）
56～67	子帧 1 的两个码本增益的联合编码（12bit）

68～79	子帧 2 的两个码本增益的联合编码（12bit）
80～91	子帧 3 的两个码本增益的联合编码（12bit）
92	子帧 0 奇偶格点（采用奇位置还是偶位置）（1bit）
93	子帧 1 奇偶格点（采用奇位置还是偶位置）（1bit）
94	子帧 2 奇偶格点（采用奇位置还是偶位置）（1bit）
95	子帧 3 奇偶格点（采用奇位置还是偶位置）（1bit）

其中：

- 2～25：整帧的 LSP 索引（24bit）

从 LSP 索引可以求出 LPC 参数。因为 10 阶的 LSP 参数被分成三个子矢量，每个子矢量有 256 种选择，所以共用 24bit。

- 26～43：各子帧的闭环基音信息（18bit）

第 0、2 子帧的闭环基音周期的取值范围是 18～141。以第 0 子帧为例，“子帧 0 的闭环基音周期减 18”取值范围为 0～123，用 7bit 可编码。以第 1 子帧为例，“子帧 1 的闭环基音周期和子帧 0 的闭环基音周期的关系”＝“子帧 1 的闭环基音周期” －“子帧 0 的闭环基音周期” ＋1。由于“子帧 1 的闭环基音周期和子帧 0 的闭环基音周期的关系”只能取值为 0、1、2、3，所以可以用 2bit 编码。

- 44～91：各子帧的两个码本增益的联合编码（48bit）

两个码本增益指自适应码本增益 Ic 和固定码本增益 I_F。对于固定码本激励，不管是低码率还是高码率，码本增益的索引有 24 种取值，需 5bit 编码。自适应码本增益的索引最大为 169，需 8bit 编码。本来共需 13bit 编码，为了减少 bit 使用数，将“自适应码本增益索引”×24＋“固定码本增益索引”编码，其最大值为 169×24＋23＝4079=0xFEF，可以用 12bit 编码。

查看编码过程可以发现：使用自适应码本的 85 码本的条件是，高码率编码且“偶子帧”（注：这里的“偶子帧”是指：对偶子帧来说是其自身，对于奇子帧来说是其上一子帧）闭环基音周期小于 58；在固定码本激励中，有可能采用脉冲串调整的条件是，高码率编码且“偶子帧”闭环基音周期小于 58。这就是说只有在使用 85 码本时才有可能使用脉冲串调整。当使用 85 码本时，自适应码本增益的索引的最大值为 57，联合编码最大值为 57×24＋23＝1391＝0x56F，这时只使用 11bit，而其最高 bit（第 12bit）正好留出来表示是否使用脉冲串调整（1 表示采用，0 表示不采用）。

- 92～95：各子帧奇偶格点标志位（4bit）

不管是采用高码率还是低码率激励，都有脉冲是使用全奇位置还是全偶位置的问题。如果是 1 表示采用全奇位置，否则表示采用全偶位置。四个子帧共 4bit。

1.2.18.2 活动帧中高码率部分数据包格式

表 1－6 为高码率部分数据包格式：

表 1－6 高码率部分

比特分配	该部分的功能
96	保留位（1bit）
97～109	4 个子帧的脉冲位置编码的高 4 位的联合编码（13bit）
110～125	子帧 0 的低 16 位脉冲位置编码（16bit）
126～139	子帧 1 的低 14 位脉冲位置编码（14bit）
140～155	子帧 2 的低 16 位脉冲位置编码（16bit）
156～169	子帧 3 的低 14 位脉冲位置编码（14bit）

170～175	子帧 0 的脉冲符号编码（6bit）
176～180	子帧 1 的脉冲符号编码（5bit）
181～186	子帧 2 的脉冲符号编码（6bit）
187～191	子帧 3 的脉冲符号编码（5bit）

其中：

- 96：保留位（1bit）
  不使用时写入 0。
- 97～109：4 个子帧的脉冲位置编码的高 4 位的联合编码（13bit）

为了节约 bit 使用量，将 4 个子帧的脉冲位置编码的高 4 位提取出来进行联合编码。以上已经讲述，对于高码率激励脉冲位置使用组合数编码。对偶子帧脉冲位置编码最大数为 0x90F6F，高 4 位为 9。对奇子帧脉冲位置编码最大数为 0x22CAA，高 4 位为 8。设 4 个子帧的高 4 位分别为 a0、a1、a2、a3。其中 a3 的最大值为 8，所以可以用 a2×9＋a3 来对 a2、a3 联合编码。同理可用 a0×9＋a1 对 a0、a1 联合编码。a2×9＋a3 的最大值为 89，所以可以用（a0×9＋a1）×90＋（a2×9＋a3）对 a0、a1、a2、a3 联合编码。

- 110～169：4 个子帧脉冲位置编码（剩余部分）（60bit）
- 170～191：4 个子帧脉冲符号编码（22bit）

1.2.18.3 活动帧中低码率部分数据包格式

表 1－7 为低码率部分数据包格式：

表 1－7 低码率部分

比特分配	该部分的功能
96～107	子帧 0 的脉冲位置编码（12bit）
108～119	子帧 1 的脉冲位置编码（12bit）
120～131	子帧 2 的脉冲位置编码（12bit）
132～143	子帧 3 的脉冲位置编码（12bit）
144～147	子帧 0 的脉冲符号编码（4bit）
148～151	子帧 1 的脉冲符号编码（4bit）
152～155	子帧 2 的脉冲符号编码（4bit）
156～159	子帧 3 的脉冲符号编码（4bit）

其中：

- 96～143：各个子帧的脉冲位置编码（48bit）

每个子帧的脉冲位置用 12bit 编码。见 1.14.2.4。

- 144～159：各个子帧的脉冲符号编码（16bit）

1.2.18.4 静音帧的数据包格式

表 1－8 为静音帧数据包格式：

表 1－8 静音帧

比特分配	该部分的功能
2～25	整帧的 LSP 索引（24bit）
26～31	静音帧增益索引（子帧 0 的固定码本增益）（6bit）

1.2.18.5 不传送帧的数据包格式

不传送帧数据包除了 2bit 控制信息，没有其它数据。

1.3 解码器

1.3.1 解码器的初始化

初始化步骤如下：

（1） 和编码器一样，量化以后的 LSP 矢量 $\tilde{P}_n$ 初始化值为常数 P_{DC} ，即

$$\tilde{P}_{-1} \Leftarrow P_{DC}$$

（2）增益调节单元的 g(n)的初始化，即

$$g(-1) = 1.0$$

（3）其它参数都初始化为 0

1.3.2 接收信息解包

解码端接收到编码信息数据包以后要从数据包中提取一帧语音的编码信息。解包的方法就是打包的逆过程。

在这部分算法中，还需要检测出数据包是否有传输过程中引入的错误。如果检测到错误帧设置 Crc=1。错误的检测包括两个方法：

1）信道编码部分发现此帧数据包校验和错误。

2）解包过程中发现解包出来的编码参数超出了取值范围。例如闭环基音周期减 18 的值应该小于等于 123。自适应码本索引 Ic 应该小于相应码本（85 码本或者 170 码本）的最大索引。

1.3.3 LSP 解码

根据解包得到的 LSP 索引 I，和上一帧的解码以后的 LSP 矢量 $\tilde{P}_{n-1}$，以及帧出错信息 Crc，可以得到当前帧解码以后的 LSP 矢量 $\tilde{P}_n$。LSP 解码的方法在编码器部分已经说明。

1.3.4 LSP 插值和 LSP 参数至 LPC 参数的转化

使用 $\tilde{P}_n$ 和 $\tilde{P}_{n-1}$ 进行 LSP 插值，以得到 4 个子帧的 LSP 参数，然后将 LSP 参数反转化为 LPC 参数，这里得到的 LPC 参数称之为量化以后的 LPC 参数 $a_{ij}, j = 1,2...10, i = 0...3$。LSP 插值和 LSP 参数转化为量化为 LPC 参数的方法在编码器部分已经说明。

1.3.5 当前帧的激励信号 e(n)的计算

恢复当前帧（四个子帧）的激励信号 e(n)。对于每个子帧，根据自适应码本增益索引 Ic、闭环基音周期 Li 和以前的激励 PreExc，可以得到自适应码本激励 u[n]；根据固定码本增益索引 I_F 和脉冲位置、符号、奇偶格点信息（高码率还有是否使用脉冲串调整的信息）可以得到固定码本激励 v[n]。e(n)=u(n)+v(n),n=0…239。

1.3.6 基音后滤波参数的计算(如果 UsePf 等于 1)

为了改善合成语音信号的质量，让激励信号 e(n),n=0…239 通过基音后滤波器。是否采用基音后滤波是用户可以选择的，如果用户选择后滤波则设置 UsePf=1。

将 e(n),n=0…239 分为 4 个子帧设为 ei(n),n=0…59,i=0…3，对每一子帧计算对应的基音后滤波器 Fposti,i=0…3。基音后滤波时，ei(n)通过 Fposti 得基音后滤波以后的激励 ppfi(n),i=0…3。

基音后滤波时，不仅需要当前帧解码得到的激励信号 e(n),n=0…239，而且需要上一帧的激励信号的后 145 个点即 e(n),n=－145…－1，这 145 个点也就是保存在 PreExc 中的 145 个点。

基音后滤波公式如下：

$$ppf_i(n) = g_p \cdot [e_i(n) + g_{ltp} \cdot g \cdot e_i(n)]$$

其中 g_p 、g_{ltp} 、g 、M_{sel} 为要求的第 i 子帧的基音后滤波器 Fposti 的参数。M_{sel} 为激励信号互相关最大位置，可以是前向互相关也可以是后向互相关，到底选用哪个，需要对两者进行比较。

1.3.6.1 激励信号后向互相关最大位置 M_b 和对应的互相关值 C_b 的计算

设解码得到的闭环基音周期为 $L_i, i = 0,1$，设 M 的搜索范围是

$$L_i - 3 \le M \le L_i + 3$$

则求第 j 子帧的 M_b 的方法是，使得后向互相关

$$C = \sum_{n=60j}^{60j+59} e(n) \cdot e(n - M)$$

取得最大值的 M 就是 M_b，对应的 C 为后向互相关最大值 C_b。如果所有的 C 都小于等于 0，$M_b = 0$

1.3.6.2 激励信号前向互相关最大位置 M_f 和对应的互相关值 C_f 的计算

M 的搜索范围如前所述。求第 j 子帧的 M_f 的方法类似于计算激励信号后向互相关最大位置 M_b 的方法，只是这里需要判断点的位置是否越过帧边界。具体步骤如下：

1） 初始化：$M_f = 0$，前向互相关最大值 $C_f = 0$，搜索位置 $M = L_i - 3$

2） 判断 $60 \times (j+1) + M$（注：$60 \times j + 59 + M$ 是计算前向互相关用到的最靠右的点）是否小于等于帧边界 240，即

$$60 \times (j+1) + M \le 240$$

不成立跳到 3），否则计算前向互相关

$$C=\sum_{n=60j}^{60j+59}e(n)\cdot e(n+M)$$

如果

$$C>C_f$$

则

$$M_f \Leftarrow M\ ,\ C_f \Leftarrow C$$

3）M 增一。如果 $M>L_i+3$，则退出，否则跳到 2）

和后向互相关有一样，如果前向互相关都小于 0，则 $M_f=0$。

1.3.6.3 后向和前向互相关值的比较与选择

从求解步骤可以知道如果 M_b 等于 0，则表明最大后向互相关小于等于 0；如果 M_f 等于 0，则表明最大前向互相关小于等于 0。后向和前向互相关值 M_b、M_f 有四种可能的组合

（1）$M_b=0,M_f=0$，则后向和前向互相关都不被采用，不使用基音后滤波。

（2）$M_b>0,M_f=0$，后向互相关被采用，$C\Leftarrow C_b$，$D\Leftarrow D_b$，$M_{sel}\Leftarrow -M_b$

（3）$M_b=0,M_f>0$，前向互相关被采用，$C\Leftarrow C_f$，$D\Leftarrow D_f$，$M_{sel}\Leftarrow M_f$

（4）$M_b>0,M_f>0$，需要进行进一步的比较，比较的方法如下：

设

$$D_b=\sum_{n=60j}^{60j+59}e(n-M_b)\cdot e(n-M_b)$$

$$D_f=\sum_{n=60j}^{60j+59}e(n+M_f)\cdot e(n+M_f)$$

如果

$$\frac{C_b^2}{D_b}>\frac{C_f^2}{D_f}$$

采用后向互相关，$C\Leftarrow C_b$，$D\Leftarrow D_b$，$M_{sel}\Leftarrow -M_b$；否则采用前向互相关，$C\Leftarrow C_f$，$D\Leftarrow D_f$，$M_{sel}\Leftarrow M_f$。

1.3.6.4 基音后滤波器参数的计算

第 j 子帧的激励信号的能量为

$$T_{en}=\sum_{n=60j}^{60j+59}e(n)\cdot e(n)$$

要求“预测增益”大于 1.25DB 即：

$$-10\log_{10}(1-\frac{C^2}{D\cdot T_{en}})>1.25$$

$$\Rightarrow \frac{C^2}{D\cdot T_{en}}>1-10^{-\frac{1.25}{10}}\approx\frac{1}{4}$$

如果上式不满足，则不需要进行基音后滤波，即设

$$g_p=1, g=0$$

否则做如下计算：

（1）g 的计算方法如下

如果

$$C\geq D$$

则

$$g=1$$

否则

$$g=\frac{C}{D}$$

（2）g_{ltp} 的计算方法如下

使用高码率

$$g_{ltp}=0.1875$$

使用低码率

$$g_{ltp}=0.25$$

（3）g_p 的计算方法如下

g_p 的计算公式为

$$g_p=\sqrt{\frac{\sum_{n=0}^{59}e^2(n)}{\sum_{n=0}^{59}(ppf'(n))^2}}=\sqrt{\frac{\sum_{n=0}^{59}e^2(n)}{\sum_{n=0}^{59}(e(n)+g_{ltp}\cdot g\cdot e(n+M_{sel}))^2}}$$

根号中的分数的分子即为 T_{en}，分母

$$\sum_{n=0}^{59}(e(n)+g_{ltp}\cdot g\cdot e(n+M_{sel}))^2$$

$$=\sum_{n=0}^{59}e^2(n)+2\cdot g_{ltp}\cdot g\sum_{n=0}^{59}e(n)\cdot e(n+M_{sel})+(g_{ltp}\cdot g)^2\sum_{n=0}^{59}e^2(n+M_{sel})$$

$$= T_{en} + 2 \cdot g_{ltp} \cdot g \cdot C + (g_{ltp} \cdot g)^2 \cdot D$$

为了保险起见在计算开方以前测试该分母是否大于 0，如果不成立

$$g_p = 0$$

否则

$$g_p = \sqrt{\frac{T_{en}}{T_{en} + 2 \cdot g_{ltp} \cdot g \cdot C + (g_{ltp} \cdot g)^2 \cdot D}}$$

1.3.7 新计算出来的激励信号的钳位

$e(n)$ 中 $n = 0...239$ 的 240 个点是新计算出来的联合激励信号。钳位的目的是将激励信号限制在闭区间[–32768.0,32767.0]之中，钳位的方法如下

$$e(n) = \begin{cases} -32768.0 & e(n) < -32767.5 \\ e(n) & others \\ 32767.0 & e(n) > 32766.5 \end{cases}, n = 0...239$$

可以看到钳位时采用了四舍五入的方法。

是否需要基音后滤波，对新计算出来的激励信号进行钳位都是必需的。如果需要基音后滤波，则在通过基因后滤波以前（计算基音后滤波参数以后），进行钳位。

1.3.8 基音后滤波（如果 UsePf 等于 1）

对每一子帧做基音后滤波器处理

$$ppf_i(n) = g_p \cdot [e_i(n) + g_{ltp} \cdot g \cdot e_i(n)], i = 0...3$$

1.3.9 LPC 合成滤波

LPC 合成滤波对于每一子帧处理一次。LPC 合成滤波器的输入信号是经过基音后滤波的联合激励信号 $ppf(n), n = 0...59$（如果不使用基音后滤波就是钳位以后的 e(n)信号），输出是 LPC 合成语音信号 $Sy(n), n = 0...59$，系统函数是

$$Sy(n) = \sum_{j=1}^{10} a_{i,j} \cdot Sy(n-j) + ppf(n), n = 0 \cdots 59$$

$a_{i,j}$ 就是解码得到的 LPC 参数。从公式中可以看出，计算此子帧的 $Sy(n)$ 时，需要上一子帧的最后 10 个点的 $Sy(n)$。

1.3.10 共振峰后滤波（如果 UsePf 等于 1）

共振峰后滤波也是用户可选的，如果使用设置 UsePf=1。共振峰后滤波对每一子帧

处理一次。共振峰后滤波器的输入信号为 LPC 合成滤波器的输出信号 $Sy(n), n = 0...59$，输出信号为 $pf(n), n = 0...59$，系统函数为

$$F(z) = \frac{1 - \sum_{j=1}^{10} a_{i,j} \lambda_1^j z^{-j}}{1 - \sum_{j=1}^{10} a_{i,j} \lambda_2^j z^{-j}} (1 - 0.25 k_1 z^{-1})$$

其中 $a_{i,j}$ 为第 i 子帧的第 j 个 LPC 参数，$\lambda_1 = 0.65$，$\lambda_2 = 0.75$，k_1 用以下方法得到

$$k_1 = \frac{3}{4} k_{1Old} + \frac{1}{4} k$$

其中 k_{1Old} 为前一子帧的 k_1，k 的求法如下：当 $\sum_{n=0}^{59} Sy(n) \cdot Sy(n) = 0$ 时

$$k = 0$$

否则

$$k = \left(\frac{\sum_{n=1}^{59} Sy(n) \cdot Sy(n-1)}{\sum_{n=0}^{59} Sy(n) \cdot Sy(n)} \right)$$

共振峰后滤波，可以认为由三个滤波器串联组成，如图 1－13

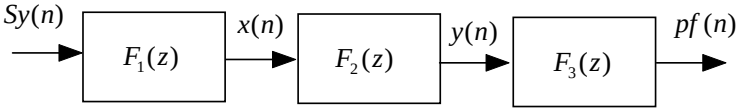

图　1－13 共振峰后滤波器

其中

（1）

$$F_1(z) = 1 - \sum_{j=1}^{10} a_{i,j} \lambda_1^j z^{-j}$$

其时域表达式为

$$x(n) = Sy(n) - \sum_{j=1}^{10} a_{i,j} \lambda_1^j Sy(n-j)$$

计算 $x(n)$ 需要前一子帧的 $Sy(n)$ 的最后 10 个点的数据。

（2）

$$F_2(z) = \frac{1}{1 - \sum_{j=1}^{10} a_{i,j} \lambda_2^j z^{-j}}$$

其时域表达式为

$$y(n) = x(n) + \sum_{j=1}^{10} a_{i,j} \lambda_2^j y(n-j)$$

计算 $x(n)$ 需要前一子帧的 $y(n)$ 的最后 10 个点的数据

（3）

$$F_3(z) = 1 - 0.25k_1 z^{-1}$$

其时域表达式为

$$pf(n) = y(n) - 0.25k_1 y(n-1)$$

计算 $x(n)$ 需要前一子帧的 $y(n)$ 的最后 1 个点的数据

总之共需保留的状态变量有 $Sy(n)$ 的前 10 个点的数据， $y(n)$ 的前 10 个点的数据。

1.3.11 增益调节单元（如果 UsePf 等于 1）

增益调节单元也是用户可选的，如果使用需设置 UsePf=1。增益调节单元用于调节 $pf(n)$ 的幅度，设调节以后的信号为 $q(n)$。调节所用公式为

$$q(n) = pf(n) \cdot g(n) \cdot (1+\alpha), n = 0...59$$

其中 $\alpha = \frac{1}{16}$， $g(n)$ 的计算方法如下

$$g(n) = (1-\alpha) \cdot g(n-1) + \alpha \cdot g_s$$

为了计算 $g(n)$，必须保存一个状态变量，即 $g(n)$ 的前一个点的值 $g(n-1)$，式中 g_s 的求法如下：如果

$$\sum_{n=0}^{59} pf^2(n) = 0$$

则

$$g_s = 1$$

否则

$$g_s = \sqrt{\frac{\sum_{n=0}^{59} Sy^2(n)}{\sum_{n=0}^{59} pf^2(n)}}$$

1.3.12 语音信号的浮点表示到定点表示的转化

由于计算过程采用了浮点方式，需要将计算的结果 $q(n)$ 转化为定点方式表示。设转化以后的信号为 $qf(n)$，转化的方法如下。

$$qf(n) = \begin{cases} 32767 & q(n) > 32766.5 \\ [q(n)+0.5] & 32766.5 \ge q(n) \ge 0 \\ [q(n)-0.5] & -32767.5 \le q(n) < 0 \\ -32768 & q(n) < -32767.5 \end{cases}$$

1.4 静音压缩方案（Annex A:SILENCE COMPRESSION SCHEME）

1.4.1 语音活动检测－－VAD(Voice Activity Detector)

语音活动检测的任务是检测当前帧为活动帧还是非活动帧，如果判定为非活动帧设置则 $Ftyp_t=1$，否则让 $Ftyp_t\neq 1$。$Ftyp_t$ 将用于进一步帧类型的判断。$Ftyp_t$ 的可能取值和意义是：1 活动帧，0 不传送帧，2 静音帧

1.4.1.1 VAD 初始化

文件:VAD.c	函数: Init_Vad	VAD 部分初始化

VAD 需要初始化的变量和初始化值如下：

Hangover 帧数：Hcnt＝3

连续语音帧数：Vcnt＝0

逆滤波器参数：$\{a_{no}[j]\}_{j=1...10}=0$

能量：$Enr_{-1}=1024$

噪声电平(Noise Level)：$Nlev_{-1}=1024$

使能标志(Adaptive enable flag)：$Aen_{-1}=0$

$Polp[i],i=0...3$：

$$Polp=\begin{cases}1 & ,i=0,1\\ 60 & i=0,2\end{cases}$$

1.4.1.2 语音活动检测

文件:VAD.c	函数: Comp_Vad	语音活动检测

1.4.1.2.1 使能标志(Adaptive enable flag) Aen_t 的计算

第 t 帧的使能标志为 Aen_t，Aen_t 将影响后面的噪声电平 $Nlev_t$ 的计算。根据信号是否为浊音或者正弦信号，Aen_t 的计算方式不同。Aen_t 的取值范围是区间[0,6]内的整数。

（1）清音(unvoiced)/浊音(voiced)判断

根据 Polp[i]计算 pc，如果 pc＝4 则判定为浊音帧。

在静音检测时，$Polp[i],i=0...3$ 中的数据是：$Polp[0]$：前第 t-2 帧的第一个开环基音周期；$Polp[1]$：第 t-2 帧的第二个开环基音周期；$Polp[2]$：第 t-1 帧的第一个开环基音

周期；*Polp*[3]：第 t-1 帧的第二个开环基音周期。*Polp*[2]，*Polp*[3] 需要在 G.723.1 代码中计算出开环基音周期时设置；*Polp*[0]，*Polp*[1]，则用 *Polp*[2]，*Polp*[3] 更新。

得到 $Polp[i], i = 0...3$ 中的最小值 $L_{OL}^{\min}$，即

$$L_{OL}^{\min} = \min\{Polp[i]\}_{i=0...3}$$

pc 的计算方法是：看有多少个 *Polp*[*i*] 的值落在 $L_{OL}^{\min}$ 的整数倍的附近。具体计算方法如下：

1）初始化。 *pc* ＝0，i＝0

2）设 j＝1

3）如果

$$\left|Polp[i] - j \cdot L_{OL}^{\min}\right| \le 3$$

则

$$pc \Leftarrow pc + 1$$

4）j 增一，如果 j 等于 9 则进入 5），否则跳到 3）

5）i 增一，如果 i 等于 4 则退出，否则跳到 2）

（2）正弦信号检测

正弦信号检测在 G.723.1 主算法计算 LPC 参数时已经计算。如果 $\text{sinD} = 1$ 则为正弦信号。

（3）计算使能标志

设当前帧 t 的使能标志为 Aen_t，前一帧为 Aen_{t-1}，Aen_t 的计算方法如下

$$Aen_t = \begin{cases} Aen_{t-1} + 2 & , pc = 4 \quad or \quad siDet[15] = 1 \\ Aen_{t-1} - 1 & , otherwise \end{cases}$$

（4）钳位使能标志

如果 $Aen_t > 6$ 则 $Aen_t = 6$；如果 $Aen_t < 0$ 则 $Aen_t = 0$

1.4.1.2.2 逆滤波(Inverse filtering)后语音信号的能量 Enr_t

Enr_t 将用于和门限 *Thr* 比较，以判定是否为活动帧，同时它用于计算噪声电平 $Nlev_t$。Enr_t 的计算方法如下：设语音信号 $\{s[n]\}_{n=60...239}$（注：$\{s[n]\}_{n=60...239}$ 也就是 $\{Frm_t[n]\}_{n=0...179}$ 即当前帧的前三子帧）通过一个 FIR 逆滤波器 $A_{no}(z)$ 得到 $e_t'[n]$

$$e_t'[n] = s[n] + \sum_{j=1}^{10} a_{no}[j] \cdot s[n-j], n = 60...239$$

其中 $\{a_{no}[j]\}_{j=1...10}$ 是 $A_{no}(z)$ 的参数，由 CNG 模块计算。

计算 $e_t'[n], n = 60...239$ 的能量

$$Enr_t = 0.5 \cdot \frac{1}{180} \sum_{n-60}^{239} (e_t'[n])^2$$

（注：以上公式中，程序中有乘以 0.5 而文档中没有乘以 0.5）

1.4.1.2.3 噪声电平(Noise Level) $Nlev_t$ 的计算

噪声电平用于计算门限值 Thr。设第 t 帧的噪声电平为 $Nlev_t$，前一帧的噪声电平为 $Nlev_{t-1}$。

1）如果 $Nlev_{t-1} > Enr_{t-1}$，则需要将噪声电平被裁减(clipped)，即

$$Nlev_t = \begin{cases} 0.25 \cdot Nlev_{t-1} + 0.75 Enr_{t-1} & , Nlev_{t-1} > Enr_{t-1} \\ Nlev_{t-1} & , otherwise \end{cases}$$

2)根据 Aen_t 调整噪声电平

$$Nlev_t \Leftarrow \begin{cases} \frac{33}{32} \cdot Nlev_t & , Aen_t = 0 \\ \frac{2047}{2048} Nlev_t & , otherwise \end{cases}$$

3）钳位 $Nlev_t$ 在 $[128,131071]$ 之间，即：如果 $Nlev_t > 131071$，$Nlev_t = 131071$；如果 $Nlev_t < 128$，$Nlev_t = 128$

1.4.1.2.4 门限值(Threshhold)的计算

门限值将用于 Vad 的判决。将 $Nlev_t$ 用指数和尾数表示出来

$$Nlev_t = Frac \cdot 2^{bexp}, 0.5 \le Frac < 1$$

根据 $Nlev_t$ 的取值范围可以知道 bexp 的取值范围是[8,17]。设 $Frac$ 用二进制表示为

$$Frac = (0.1 b_1 b_2 b_3 b_4 b_5 b_6 ...)_b = 0.5 + \sum_{j=2}^{\infty} b_{j-1} (0.5)^j$$

其中 b_j 为 0 或者 1，表示对应位的数值。

令

$$temp = (0.b_1 b_2 b_3 b_4 b_5 b_6)_b$$

另外设 $ScfTab$ 表格如下

ScfTab[11] = { 9170.0, 9170.0, 9170.0, 9170.0, 10289.0, 11544.0, 12953.0, 14533.0, 16306.0, 18296.0, 20529.0 };

则门限计算公式如下：

$$Thr = \frac{1}{4096}\left[(1-temp)\cdot ScfTab[18-\text{bexp}] + temp\cdot ScfTab[17-\text{bexp}]\right]\cdot Nlev_t$$

（注：这部分程序和文档说明的不太一样，是否一致没有验证过。文档中门限值计算方法如下：

$$Thr = \begin{cases} 5.012 & ,Nlev = 128 \\ 10^{0.7-0.05\log_2\frac{Nlev}{128}} & ,128 < Nlev < 16384 \\ 2.239 & ,Nlev \geq 16384 \end{cases}$$

）

1.4.1.2.5 Vad 判决

$$Vad_t = \begin{cases} 1 & Enr_t > Thr \\ 0 & Enr_t \leq Thr \end{cases}$$

（注：文档中是大于等于时判为 1）

$Vad_t = 1$表示是语音帧，否则是静音帧。

1.4.1.2.6 Hangover 帧数 Hcnt 的计算

当检测到静音帧时，并不是立即从语音活动状态转化到语音非活动状态，而是当出现较多连续的静音帧以后（此时 Hcnt 减小）才判定为非活动状态。

（1）连续静音帧数 Vcnt 的变化：

首先 Vcnt 值限定为[0,3]内的整数。

当$Vad_t = 1$时，Vcnt＋＋；否则 Vcnt－－

（2）Hangover 帧数 Hcnt 的变化：

首先 Hcnt 值限定为大于 0 的整数。

如果$Vad_t = 1$时，Hcnt＋＋；

连续没有语音帧（Vcnt 等于 0）时，Hcnt－－；

当出现连续语音帧（Vcnt 大于等于 2）时，Hcnt 无条件设置为 6；

1.4.1.2.7 语音活动检测

如果 Hcnt 大于 0，判定为语音活动状态。否则，如果$Vad_t = 1$则为活动状态，$Vad_t = 0$则为非活动状态。

1.4.2 CNG 算法

1.4.2.1CNG 编码部分

只有帧类型 $Ftyp_t \neq 0$（非活动帧时）才采用以下的 CNG 算法

1.4.2.1.1 CNG 编码部分初始化

文件:CODCNG.c	函数: Init_Cod_Cng

量化以后的码本增益 $\tilde{G}_{-1} = 0$

和自相关 $R^k[j] = 0, k = -1...-3, j = 0...10$

SID－LPC 参数 $a_{sid}[j] = 0, j = 1...10$

上一帧类型 $Ftyp_{-1} = 1$

随机种子 $RandSeed = 12345$

1.4.2.1.2 四子帧自相关之和的计算

文件:CODCNG.c	函数: Update_Acf

四子帧自相关之和 $R^t[j], j = 0...10$ 用于计算残余能量 E_t。在 G.723.1 的主算法中求 LPC 参数时，已经计算出四个子帧的自相关 $R_i[j], i = 0...3, j = 0...10$。定义第 t 帧自相关和为 $R^t[j], j = 0...10$。

$$R^t[j] = \sum_{i=0}^{3} R_i[j], j = 0...10$$

1.4.2.1.3 残余能量(Residual Enery) E_t 和当前帧 CNG－LPC 滤波器 $A_t(z)$ 的计算

文件:CODCNG.c	函数: Cod_Cng

若在 Levision-Durbin 算法中，输入量（即 11 个自相关）采用 $R^t[j]$，则此时：

（1） Levision-Durbin 在递推过程的最后一步（即 i 等于 9 时）计算出来的 E 定义为 E_t。

（2） 计算出来的 LPC 参数 $a_t[j], j = 1...10$，组成 CNG 的 LPC 滤波器 $A_t(z)$

1.4.2.1.4 能量和(Energy Sum) $\overline{E}$ 的计算和码本增益 G_t 的量化

文件:UTILCNG.c	函数: Qua_SidGain

定义最近的k_E帧的能量和为$\overline{E}$

$$\overline{E} = \sum_{i=0}^{k_E-1} E_{t-i}$$

其中k_E的计算方法如下：如果此帧为第一个非活动帧（即，上一帧为活动帧），则$k_E = 1$，否则k_E增一，但是最大为 3。码本增益的平方的计算公式为

$$G_t^2 = fact[k_E] \cdot \overline{E}, k_E = 0...3$$

其中 fact[4] = {(F)0.008333, (F)61.6815, (F)30.84075, (F)20.6145};
（注：fact 的计算方法还没有弄清，按照文档给定的方法计算的结果和此结果不同）

以下量化G_t，产生码本增益量化索引$GInd_t$。量化G_t的过程中都使用G_t^2做为比较对象，以避免使用开方。G_t的量化采用6bit伪对数（pseudo-logarithmic）量化，G_t的取值被分为三个区间（区间0，区间1和区间2）。区间特性如表1－9：

表 1－9 Gt 量化分区表

	区间 0	区间 1	区间 2
区间范围	0～31	32～95	96～351
等分数（等分号数目）	16 等分	16 等分	32 等分
每等分大小	2	4	8

为了说明方便定义：

区间的起点：　　base[3] = {(F)0.0, (F)32.0, (F)96.0};
下一区间的起点的平方：　　bseg[3] = {(F)1024., (F)9216., (F)115617.};
区间等分数为：　　frag[3] = {16,16,32}

（注：这里最后一个值为(F)115617，和文档说法有出入）

量化索引$GInd_t$的存储格式。$GInd_t$共需 6bit。设G_t所处的区间号为 Seg，等分号为 j。则$GInd_t = Seg << 4 + j$。解码时若发现$GInd_t$最高位为 1，则必然是第 2 区间，$GInd_t$的低 5 位为 j；否则低 4 为 j；

量化的步骤如下：

（1）确定G_t所属区间（Seg）：

如果G^2大于等于 bseg[2]，则用最大值量化，即$GInd_t = 0x3F$。量化结束。

否则，如果大于等于 bseg[1]，则说明处在区间 2 中，Seg＝2。
否则，如果大于等于 bseg[0]，则说明处在区间 1 中，Seg＝1。
否则，在区间 0 中，Seg＝0。

（2）用折半查找法查找G_t在 Seg 区间的哪个等分附近，记该等分为 j

查找方法如下：

1）初始化：开始时处于中间等分上，j＝frag[seg]/2；查找次数 i＝0；折半查找增量 k＝j/2

2）Seg 区间 j 等分点的值 temp 用下式计算， $temp = base[seg] + j \cdot 2^{seg+1}$

3) 如果 G_t 大于等分点数值 temp（实际比较 G^2 和 temp×temp），则 j = j + k；否则 j = j – k。

4）更新折半查找增量 k=k/2。查找次数 i 增一，如果 i 等于 $\log_2 frag[seg]$ 则退出，否则跳到 2）

（3）四舍五入，使结果进一步精确

经过以上的折半查找 G_t 将处在 seg 区间的 j 等分点附近，但是并不知道是在该等分点的左边还是右边，并且不知道 j-1、j、j+1 中的哪个等分点更加精确。以下进一步确定

如果 j 等分点在 G_t 的左边（即 $temp^2 - G^2 \le 0$）：则 G 处于 j 和 j＋1 之间，比较 $base[seg] + j \cdot 2^{seg+1}$ 和 $base[seg] + (j+1) \cdot 2^{seg+1}$ 哪个更接近 G^2。如果 j 更接近，码本增益索引 $GInd_t = seg << 4 + j$，否则 $GInd_t = seg << 4 + j + 1$

如果 j 等分点在 G_t 的右边：则 G 处于 j－1 和 j 之间，比较 $base[seg] + j \cdot 2^{seg+1}$ 和 $base[seg] + (j-1) \cdot 2^{seg+1}$ 哪个更接近 G^2。如果 j 更接近，码本增益索引 $GInd_t = seg << 4 + j$，否则 $GInd_t = seg << 4 + j - 1$

1.4.2.1.5 当前帧类型 $Ftyp_t$ 的计算

文件:CODCNG.c　　函数: Cod_Cng

由于 CNG 算法只在此帧为非活动帧时被采用。计算当前帧的类型的任务就是要区别是静音帧（SID 帧）还是不传送帧。

判断为 SID 帧的方法是：

如果此帧为第一个非活动帧（即，上一帧为活动帧）则此帧设置为 SID 帧 $Ftyp_t = 2$。

否则，需要进一步判断，只要符合以下条件的任何一个，则设置此帧为 SID 帧

（1）如果当前 CNG－LPC 滤波器 $A_t(z)$ 和 SID 滤波器 $A_{sid}(z)$ 之间有很大的差别

$A_t(z)$ 已经在上面计算出来，$A_{sid}(z)$ 也是一个 LPC 滤波器，其 LPC 参数通过最近几帧的自相关系数的平均计算，表示 LPC 参数的平均情况。$A_t(z)$ 和 $A_{sid}(z)$ 之间的差别的大小是通过比较 $R_a^{t-1}[i]$ 和 $R^t[i]$ 得到的。

$$\sum_{i=0}^{10} R_a^{t-1}[i]\cdot R^t[i] \geq E_t \cdot thr1, thr1 = 1.2136$$

则认为相差很大，否则相差不大。其中的 $R_a^{t-1}[i]$ 是同 $A_{sid}(z)$ 相关的参数。$R_a^{t-1}[i]$ 和 $A_{sid}(z)$ 的计算方法见后面。

（2）如果当前帧的码本增益量化索引 $GInd_t$ 和上一 SID 帧的码本增益量化索引 $GInd_{sid1}$ 相差较大。判断条件如下：$|GInd_t - GInd_{t-1}| > 3$ 则认为相差较大。

1.4.2.1.6 SID 帧参数的计算

如果以上步骤设定此帧类型为 SID 帧，则进行以下的算法

（1） 计算后平均(post average)LPC 滤波器 $\overline{A}_p(z)$ 的 LPC 参数计算

文件:CODCNG.c	函数: ComputePastAvFilter

设当前帧为第 t 帧，计算 11 个平均自相关参数

$$\overline{R}_p[j] = \sum_{k=t-3}^{t-1} R^k[j], j = 0...10$$

其中 $R^k[j]$ 为第 k 帧的自相关和。使用输入参数为 $\overline{R}_p[j]$ 的 Levision-Durbin 算法得到的 LPC 参数 $\overline{a}_p[j], j = 1...10$ 即为 $\overline{A}_p(z)$ 的 LPC 参数

（2） 更新 VAD 算法中的逆滤波器参数
当使能标志 Aent＝1 时更新

$$a_{no}[j] = \overline{a}_p[j], j = 1...10$$

否则不更新。

（3） 计算 $A_{sid}(z)$ 和 $R_a^t[i]$

计算 $R_a^t[i]$ 时需要 $A_{sid}(z)$ 的 10 个 LPC 参数 $a_{sid}[j], j = 1...10$。$A_{sid}(z)$ 的 LPC 参数 $a_{sid}[j]$ 的取值有两种选择：后平均(post average)LPC 滤波器 $\overline{A}_p(z)$ 的 LPC 参数 $\overline{a}_p[j]$ 或者当前帧 CNG－LPC 滤波器 $A_t(z)$ 的 LPC 参数 $a_t[j]$。$R_a^t[i]$ 计算公式如下：

$$R_a[0] = 1 + \sum_{j=1}^{10} a_{sid}[j]$$

$$R_a[1] = 2\cdot\left[R_a[0] - a_{sid}[i] + \sum_{j=0}^{10-1-1}\left(a_{sid}[j+1]\cdot a_{sid}[j+i+1]\right)\right]$$

$$R_a[i] = 2\cdot\left[\frac{R_a[i-1]}{2} - a_{sid}[i] + \sum_{j=0}^{10-i-1}\left(a_{sid}[j+1]\cdot a_{sid}[j+i+1]\right)\right], i = 2...10$$

（注：以上公式和程序描述相符但和文档的说明不一样）

$a_{sid}[j]$ 的选择用以下方法判断：

首先假定 $a_{sid}[j]=\bar{a}_p[j], j=1...10$（即 $A_{sid}(z)$ 选择 $\overline{A}_p(z)$），然后计算 $R_a^t[i]$，如果 $A_t(z)$ 和 $A_{sid}(z)$ 的参数相差不大（比较方法见前面），则选择结束。

否则， $a_{sid}[j]=a_t[j], j=1...10$（即 $A_{sid}(z)$ 选择 $A_t(z)$），重新计算 $R_a^t[i]$。

（4） SID－LPC 滤波器的 LPC 参数对应的 *LspId*

将 $A_{sid}(z)$ 的 LPC 参数 $a_{sid}[j]$ 转化为 LSP 参数，得到 SID－LSP 矢量 P_{sid}^t。转化的方法和 G.723.1 主算法中的方法一样。转化时，要用到上一帧量化以后的 LSP 矢量，这里用主算法中解码以后的矢量 $\widetilde{P}_{t-1}$ 代替。

量化 SID－LSP 矢量 P_{sid}^t 得到 LSP 量化索引 *SidLspId*。量化的方法和 G.723.1 主算法中的方法一样。这里用到的上一帧量化以后的 LSP 矢量，也用 $\widetilde{P}_{t-1}$ 代替。

从 *LspId* 得到量化以后的 LSP 矢量 $\widetilde{P}_{sid}^t$，这里用到的上一帧量化以后的 LSP 矢量，也用 $\widetilde{P}_{t-1}$ 代替。反量化时用到的校验和 Crc，令其为 0。

（5） 计算 SID 帧的“量化后的码本增益” $\widetilde{G}_{sid}$

文件:CODCNG.c 函数: Dec_SidGain

更新上一 SID 帧的码本增益量化索引 $GInd_{sid1}$， $GInd_{sid1}=GInd_t$

计算 $\widetilde{G}_{sid}$ 需要逆转化码本增益索引 $GInd_t$。转化过程如下：

先得到 $\widetilde{G}_{sid}$ 所处的区间 seg， $seg=GInd_t<<4$。如果 seg 等于 3，说明 $\widetilde{G}_{sid}$ 处在第 2 区，并且 j（第 2 区的 j 需要 5bit 表示）的最高位是 1，修改 seg＝2。

然后得到 $\widetilde{G}_{sid}$ 所处的等分 j， $j=GInd_t-seg<<4$

计算 $\widetilde{G}_{sid}$， $\widetilde{G}_{sid}=base[seg]+j\cdot 2^{seg+1}$

1.4.2.1.7 CNG 激励的计算

（1） 计算目标激励增益(target excitation gain) $\widetilde{G}_t$

如果上一帧为活动帧，则

$$\widetilde{G}_t=\widetilde{G}_{sid}$$

否则

$$\tilde{G}_t = \frac{7}{8}\tilde{G}_{t-1} + \frac{1}{8}\tilde{G}_{sid}$$

说明如下：如果上一帧为活动帧，则说明当前帧是这个静默期的第一个非活动帧（必定为 SID 帧），则$\tilde{G}_{t-1}$没有意义，所以采用第一个式子计算；$\tilde{G}_{sid}$在 SID 帧参数计算部分，已经计算出来。

（2） 随机数产生方案

计算 CNG 激励时，需要产生[0,np1)区间内的随机整数 Rand，其中 np1 是可以设定的参数。这里产生随机数的方法是：

$$RandSeed \Leftarrow (RandSeed \cdot 521 + 259)\,\&\,0\text{x}0000\text{FFFF}$$

$$Rand = \frac{RandSeed\,\&\,0\text{x}7\text{FFF}}{0\text{x}8000} \cdot np1$$

其中 *RandSeed* 在每个静默期开始前初始化为 12345

（3）CNG 激励计算

激励分为点数为 120 的前后两块来处理。CNG 采用高码率激励的形式，但是激励的参数不是计算得到，而是随机产生。

1）随机产生以下参数：

- 闭环基音周期：前两子帧的闭环基音周期和后两子帧的闭环基音周期是[123, 143]的随机数
- 自适应码本增益索引：四个子帧的自适应码本增益索引都是[1,50]的随机数。
- 闭环基音周期：四个子帧的闭环基音周期分别为 Lop1、Lop1－1、Lop2、Lop2+2，其中 Lop1 和 Lop2 表示当前帧的前后两个开环基音周期。
- 奇偶格点选择：用 offset[4]表示四帧的激励脉冲起点。则 offset[0]和 offset[1]随机地选择 1 和 0，offset[0]和 offset[1]随机地选择 61 和 60
- 脉冲符号选择：偶数帧有 6 个脉冲，奇数帧有 5 个脉冲，所以每块激励有 11 个脉冲。随机的产生这些脉冲的符号，即随机的选择 1 和－1。
- 脉冲位置：随机的选择 4 子帧的脉冲位置，即选择[0,29]的随机数

2）用参数计算激励

首先计算自适应码本激励 u[n]。用到的参数有，闭环基音周期、自适应码本增益、和以前的 145 个点的激励信号 PrevExc。由于这里的闭环基音周期大于 123，所以只用到 PrevExc 中 145-123 个点的数据，所以可以同时用 PrevExc 计算两个子帧（也就是说这里计算激励是 120 个点一起计算而不是 60 个点一起计算），然后更新 PreExc。

对于每块 120 点的激励信号

$$e[n] = u[n] + Gf \cdot v[n], n = 0...119$$

其中的Gf 计算如下：

使得

$$\left| \frac{1}{120} \sum_{n=0}^{119} (u[n] + Gf \cdot v[n])^2 - \tilde{G}_t^2 \right|$$

取最小值的 Gf，即为要求的 Gf。

定义方程

$C(x)=x^2+2bx+c$， $b=\frac{\sum_{n=0}^{119}u[n]v[n]}{\sum_{n=0}^{119}v^2[n]}$， $c=\frac{\sum_{n=0}^{119}u^2[n]-120\tilde{G}_t^2}{\sum_{n=0}^{119}v^2[n]}$，其中 $\sum_{n=0}^{119}v^2[n]=11$

如果判别式：$\Delta=b^2-c$ 小于等于 0，则 Gf＝b；

否则计算方程的根 Gf 等于绝对值小的那个根。即 $\left|-b+\sqrt{\Delta}\right|>\left|-b-\sqrt{\Delta}\right|$，则 $Gf=-b-\sqrt{\Delta}$，否则 $Gf=-b+\sqrt{\Delta}$

限定 Gf 在[-5000,5000]之间。

限定激励值的范围在[-32768,32767]内。

（3）更新激励

一块 120 点的激励信号计算完毕以后要更新以前激励 PreExc，即 PreExc 左移 120 点，空出的 120 点用刚刚计算出来的激励 e(n)代入。

1.4.2.1.8 量化后的LPC参数的计算和上一帧量化后的LSP矢量 $\tilde{P}_{t-1}$ 的更新

对于 SID 帧和静音帧都做以下的处理。

从 LSP 矢量 P_{sid}^t 计算量化以后的 LPC 参数，其中用到上一帧量化以后的 LSP 矢量 $\tilde{P}_{t-1}$。

1.4.2.1.9 数据更新

（1）Err[i]的更新。对每一子帧更新一次。和 G.723.1 的主算法的更新方法一样

（2）更新联合滤波器状态

1.4.2.1.10 发送信息打包

见 G.723.1 主算法的发送信息打包部分。

更新 $\tilde{P}_{t-1}$ 即， $\tilde{P}_{t-1}=P_{sid}^t$

1.4.2.2 CNG 解码部分

1.4.2.2.1 CNG 解码部分初始化

$Ftyp_{-1}=1$

$G_{sid}=0$

上一帧量化解码以后的 LSP 矢量 $\tilde{P}_{-1}=P_{DC}$

$RandSeed3=12345$

文件:DECODCNG.c	函数: Regen

1.4.2.2.2 CNG 解码

方法如下：

（1） 使用接收到的信息更新 $\tilde{G}_t$ 和 $\tilde{P}_{sid}$

如果接收到的帧是 SID 帧则

解码码本增益索引 $GInd_t$ 得到量化以后的码本增益 $\tilde{G}_t$；利用上一帧解码以后的 LSP 矢量和 LSP 索引 *LspId* 得到当前帧量化以后 LSP 矢量 $\tilde{P}_{sid}$

如果是不传送帧，则 $\tilde{G}_t$ 和 $\tilde{P}_{sid}$ 的值保持不变

（2） 计算激励信号，使用参数 $\tilde{G}_t$，PrevExc，解码的随机种子 RandSeed，编码信息。

（3） 计算量化以后的 LPC 参数，使用参数 $\tilde{P}_{sid}$ 和 $\tilde{P}_{t-1}$。得到 LPC 参数和激励以后，以后的处理和 G.723.1 主算法中的算法一样。

（4） 更新 $\tilde{P}_{t-1} = \tilde{P}_{sid}$

1.4.3 出错帧处理

出错帧的定义：信道编码部分检测到校验和错误；编码信息解码时出现不允许的参数。当解码器发现错帧时：如果上一帧为活动帧，则当前帧为活动帧；否则当前帧为不传送帧。

帧丢失处理：如果 Ftypt=0 而 Ftypt-1=1，这是不允许出现的情况，说明有帧丢失，$\tilde{G}_{sid}$ 的计算方法变为：设 $k_E = 0$，将上一帧解码得到的码本增益 $\tilde{G}_{sid}$ 量化（$k_E = 0$ 是专门为出错帧设计的量化参数），量化的方法在 CNG 的“码本增益的量化”一节已经讲述。然后将量化结果反量化得到此帧的 $\tilde{G}_{sid}$。

用 ECount 累计连续出错的活动帧数：如果出现一个出错的活动帧，ECount 增一；如果出现一个没有出错的活动帧，Ecount 复位为 0。Ecount 累加的最大值为 3。

如果 Ecount 不等于 0，则 InterGain 的索引为当前帧后两子帧的固定码本增益索引的平均；否则，InterGain<=InterGain × 0.75。

在 Ecount 等于 0 的情况下，解码参数正常求出；否则使用激励的插值产生。

1.4.3.1 清音帧或浊音帧的判定

清音帧和浊音帧的判定结果用于激励的插值产生。清音帧和浊音帧的判定只在 Ecount 不等于 0 的情况下进行，判定结果产生一个状态变量 $M_{\max}$。当在帧出错进行激励插值时，$M_{\max}$ 表示上一语音帧的浊音周期。$M_{\max}$ 的计算方法如下：

（1） 未钳位以前的激励信号 e(n),n=0…239 的自相关最大值的位置 $M_{\max}$

设解码得到后两子帧的开环基音周期为 L_1，则 M 的搜索范围是

$$L_1 - 3 \le M \le L_1 + 3$$

设自适应码本和固定码本激励为 $e(n)$，其中 $n = 0...239$ 为此帧的联合激励，$n = -145...-1$ 为以前联合激励。求使得

$$C = \sum_{n=120}^{239} e(n) \cdot e(n - M)$$

取得最大值的 M 和对应的 C，令其为 $M_{\max}$ 和 $C_{\max}$。如果所有的 C 都小于等于 0，则 $M_{\max} = L_1$ 和 $C_{\max} = 0$

（2） 联合激励信号的能量

$$T_{en} = \sum_{n=120}^{239} e(n) \cdot e(n)$$

（3） 联合激励信号自相关最大位置的能量

$$D = \sum_{n=120}^{239} e(n - M_{\max}) \cdot e(n - M_{\max})$$

（4） 判定为浊音帧的条件

$$-10\log_{10}(1 - \frac{C^2}{D \cdot T_{en}}) > 0.58$$

$$\Rightarrow 1 - \frac{C^2}{D \cdot T_{en}} < 10^{-\frac{0.58}{10}} \approx 0.875$$

$$\Rightarrow \frac{1}{8} D \cdot T_{en} - C^2 < 0$$

则判定为浊音帧，令其周期为 $M_{\max}$。否则判定为清音帧令其周期为 0。

1.4.3.2 激励的插值产生

如果 Ecount 大于等于 3 则当前激励和 PreExc 都设置为 0。否则
如果前一个没有出错的活动帧为浊音帧：

设浊音周期为 $M_{\max}$，则激励信号为激励信号左移 $M_{\max}$ 个点的激励信号的 0.75 倍，

即 $e(n) = 0.75 \cdot e(n - M_{\max})$

否则若为清音：随机产生激励信号

$$RandSeed3 \Leftarrow (RandSeed3 \cdot 521 + 259) \,\&\, 0x0000FFFF$$

$$e[n] = InterGain \cdot \frac{RandSeed3}{32768}, n = 0...239$$

其中， $RandSeed3$ 为一个随机种子。以前的激励 PreExc[n],n=0…144，全部清除为 0。

第二章 G.723.1 算法在 54xDSP 上的实时实现

G.723.1 算法是国际电信组织(International Telecommunication Union 简称 ITU)的电信标准(Telecom Standardization 简称 ITU-T)的一部分。ITU 提供用标准 C 代码实现的 G.723.1 算法，但是由于它运行速度慢，无法在 54xDSP 上实时实现。这一章讲述如何修改 ITU-T 的 G.723.1 标准 C 代码，从而达到在 54xDSP 上实时实现的要求。

2.1 ITU-T 的 G.723.1 标准 C 代码算法的使用

ITU-TG.7231.算法标准的内容有两部分：①文档，文档的内容是原理的文字描述；②代码，代码是用标准 C 编写的算法程序。代码是 G.723.1 算法标准不可分割的部分，它补充了文字描述的不足，并且它用代码实例程序证明了 G.723.1 算法是可实现的，是 G.723.1 算法实现的一个实例。

2.1.1 ITU-T 的 G.723.1 算法标准及其 Annex 的内容

从 ITU-T 的网站可以获得 G.723.1 标准的相关资料[5]，主要内容如表 2－1：

表 2－1 ITU 网站关于 G..723.1 的资料

资料名	内容
G.Imp723.1/G.729/G.723.1 (02/02)	包含文件 T-REC-G.Imp723.1-200202-I!!MSW-E.doc，内容是纠正 G.723.1 文档中的一个错误和 G.729 的某些错误。
G.723.1 (03/96)	包含 G.723.1 描述文档：ITU_T Recommendation G.723.1.pdf，内容是 G.723.1 原理的描述。
G.723.1 Annex A (11/96)	Annex A：g723_1aa.pdf，内容是描述 G.723.1 的静音压缩方案。这个资料中还包含 G.723.1 的定点算法 C 代码（包括静音压缩）和 G.723.1 的测试序列。
G.723.1 Annex B (11/96)	用浮点运算描述 G.732.1： g723_1ab.pdf 。这个资料中包含 G.723.1 的浮点代码和该代码的测试序列。
G.723.1 Annex C (11/96)	G.723.1 的信道编码（在无线应用中使用）。包含信道编码的代码和测试序列。
G.Imp723.1/G.723.1 (10/02)	包含文件 T-REC-G.Imp723.1-200210-I!!MSW-E.doc，内容是纠正静音压缩的文档描述部分和代码部分的几个错误。

2.1.2 ITU-T G.723.1 标准算法 C 代码的编译运行

现在我们有两个 C 代码：G.723.1 定点程序和 G.723.1 浮点程序。

在研究算法之前最好能够看看算法运行的效果。由于两个程序都是用标准 C 而且都是调用 C 标准库函数，所以它们都可以在 windows 操作系统下编译运行。

使用 windows 下的 visual C++6.0 编译的方法是：新建一个空的 Win32 Console Application 类型的工程，工程名为 lbccodec；将 G.723.1 的相关文件添加到工程中；然

后就可以编译了；编译得到 lbccodec.exe 程序。

Lbccodec.exe 程序的功能是：如果用于编码，它从文件中读取语音数据，将编码结果写入文件；如果用于解码时，它从文件读取编码数据，将解码得到的语音写入文件。

2.1.3 ITU-T G.723.1 算法程序的命令行参数

现在我们得到了 lbccodec.exe 可执行程序，lbccodec.exe 的命令行格式为：

lbccodec [options] inputfile outputfile

其中的 option 支持的参数及参数的意义如表 2－2、表 2－3、表 2－4 所示：

表 2－2 G.723.1 算法程序命令行参数表（编解码模式设置）

参数	含义
-b	编码＋解码：对输入文件编码然后立即解码，将解码得到语音写入输出文件。
-c	编码：对输入文件数据编码。
-d	解码：对输入文件数据解码。

编码模式默认为-b。

表 2－3 G.723.1 算法程序命令行参数表（码率设置）

参数	含义
-r63	高码率。
-r53	低码率。
-rfilename	码率从文件 filename 中读取。只在 Cod 或者 Both 模式该参数才有效。

码率设置默认为-r63。码率设置对于 both 或者 cod 模式才有效，dec 模式的码率从编码信息中读取。

表 2－4 G.723.1 算法程序命令行参数表（其它设置）

参数	含义
-v	使用静音压缩功能。
-Noh	不使用高通滤波器。
-Nop	不使用基音后滤波器。
-ffilename	使用 Crc 校验文件，每帧解码开始时，从 filename 中读取一个 16bit 的数据，作为该帧的 Crc 校验和，校验和在解码过程中被使用。只有在 Dec 模式该参数才有效。
-n	关闭运行时输出信息到屏幕。
-Rnum	表示 num 帧以后编解码器重新初始化一次（不使用该参数时，Num 为 0，表示永远不会重新初始化）。

2.1.4 程序的正确性的验证及测试序列的使用

验证程序运行的正确性主要有两种方法：人耳主观判断、测试测试序列。人耳主观判断是先录制一段语音然后用 lbccodec.exe 编码语音，然后再解码，从主观上比较解码得到的语音和原始语音的差别。

2.1.4.1 程序输入语音文件格式

为了测试编解码实际语音信号，我们可以用 windows 的“录音机”程序录制一段 wav 格式的语音。那么首先要确认录制的语音是否符合 lbccodec 程序语音输入格式的要

求。

从 ITU 的 G.732.1 的描述知道，输入信号首先通过 G.712 规定的电话带通滤波器，然后用 8000Hz 采样率采样，然后将采样点转化为 16bit 的线性 PCM 编码。可见只要将“录音机”程序参数设置为 8K、16bit 线性 PCM、单声道即可。

为了进一步了解 16bit 线性 PCM 的数据格式做以下分析：

（1） 线性 PCM 编码的 0 电压用 0x80 表示（无符号类型）还是用 0x00（有符号类型）表示呢？windows 的“录音机”参数设置为 8K，16bit 的线性 PCM、单声道，在基本没有声音的情况下录制，发现数据为 00 和 FF 居多，可见 0 电压表示为 0x00。

（2） 16bit 数据的存放格式是怎样的？代码中用 fread()函数读取 16bit 语音数据，根据 fread()函数的特点可知，16bit 数的高 8bit 存放在文件的高地址，低 8bit 存放在低地址。即 little-endian 格式。

2.1.4.2 实际语音信号编解码效果的测试

可以发现“录音机”程序录制的 wav 语音文件格式符合 lbccodec 程序的输入文件格式要求。设录制好的语音文件为 my.wav，测试步骤如下：

（1） 将 my.wav 用 hex 编辑器（例如 UltraEdit32 软件）打开。

（2） 将 100H 位置开始的 240×2×50 个字节的数据拷贝到新建的空文件中。不拷贝前面的 100H 是因为 wav 文件有一个文件头格式，开始的数据并不是语音。

（3） 将新文件存盘为 my.tin 文件。

（4） 用 lbccodec –r63 –c my.tin my.pco 和 lbccodec –r63 –d my.pco my.pou 编码和解码语音。

（5） 用 hex 编辑器打开 my.pou 文件，用其内容替换到 my.wav 的 100H 开始的 240×2×50 的数据，并另存为 my2.wav。

（6） 用媒体播放器播放 my.wav 和 my2.wav，比较两者的主观差别。

2.1.4.3 使用测试序列验证程序的正确性

主观测试不能发现细微差别，这时可用测试序列来检测。所谓测试序列是专门设计用来检验算法运行正确性的一些输入文件，用 lbccodec 程序编码（或者解码）这些文件，然后将输出文件和所给的正确文件比较，如果两者相符则说明编解码无误。

为了全面的测试程序，测试序列包含多种类型，例如 path 类型序列可以测试程序中的条件转移判断是否正确；over 类型序列可以测试程序对数据溢出处理是否正确。

ITU-T 有两个测试序列：G.723.1 测试序列、AnnexA 的 vad-cng 测试序列。G.723.1 测试序列测试 6.3K 和 5.3K 的编码和解码，AnnexA 的 vad-cng 测试序列测试 Vad 和 cng。它们分别存放在 AnnexA 资料的 G723anAe\TESTSEQ 目录和 G723anAe 目录下。

Cmpcode 和 checksnr 程序：cmpcode 程序能够比较两个编码结果是否相同，若两者不同还能指出是哪一帧不同；checksnr 程序用于将解码后的语音和原始语音比较，得到两者的信噪比。这两个程序的源代码在 AnnexB 的 G723anBe 目录下。

2.1.4.3.1 G.723.1 测试序列

G.723.1 测试序列文件的文件扩展名和对应含义如表 2－5：

表 2－5 G.723.1 测试序列文件扩展名的含义

扩展名	含义
.TIN	编码输入文件
.PCO	用户编码程序编码后的输出文件

.RCO	用于和.PCO 比较的正确编码结果存放的文件
.TCO	解码输入文件
.POU	用户解码程序解码输入语音后的输出文件
.ROU	用于和.POU 比较的正确解码结果存放的文件
.CRC	和同文件名的解码输入文件配套的校验和文件

将测试序列、lbccodec 程序、cmpcode 程序、checksnr 程序拷贝到同一个目录下，进行测试的步骤如下：

（1） 使用 lbccodec 编解码测试所有序列。步骤如下：

制作一个名为 CodDecode.bat 的批处理文件。其内容如下：

```
%1 -c -r63     PATHC63H.TIN PATHC63H.PCO
%1 -c -r53 -Noh PATHC53.TIN  PATHC53.PCO
%1 -c -r63 -Noh OVERC63.TIN  OVERC63.PCO
%1 -c -r53     OVERC53H.TIN OVERC53H.PCO
%1 -c -r63 -Noh CODEC63.TIN  CODEC63.PCO
%1 -c -r53 -Noh INEQC53.TIN  INEQC53.PCO
%1 -c -r63     TAMEC63H.TIN TAMEC63H.PCO

%1 -r53 -d -Nop          INEQD53.TCO  INEQD53.POU
%1 -r53 -d -Nop          OVERD53.TCO  OVERD53.POU
%1 -r63 -d              OVERD63P.TCO OVERD63P.POU
%1 -r53 -d -Nop          PATHD53.TCO  PATHD53.POU
%1 -r63 -d -fPATHD63P.CRC PATHD63P.TCO PATHD63P.POU
%1 -r63 -d -fTAMED63P.CRC TAMED63P.TCO TAMED63P.POU
```

在命令下输入 CodDecode lbccodec 回车即可。

（2） 验证编解码的正确性。步骤如下：

制作一个名为 compare.bat 的批处理文件。其内容如下：

```
@echo off
Rem compare sequence of G.723.1
Rem coder part
cmpcode PATHC63H.RCO PATHC63H.PCO
cmpcode PATHC53.RCO  PATHC53.PCO
cmpcode OVERC63.RCO  OVERC63.PCO
cmpcode OVERC53H.RCO OVERC53H.PCO
cmpcode CODEC63.RCO  CODEC63.PCO
cmpcode INEQC53.RCO  INEQC53.PCO
cmpcode TAMEC63H.RCO TAMEC63H.PCO

Rem decoer part
checksnr INEQD53.POU  INEQD53.POU
checksnr OVERD53.ROU  OVERD53.POU
checksnr OVERD63P.ROU OVERD63P.POU
checksnr PATHD53.ROU  PATHD53.POU
checksnr PATHD63P.ROU PATHD63P.POU
checksnr TAMED63P.ROU TAMED63P.POU
```

在命令行下输入 compare 回车，屏幕上出现类似如下的输出结果：

```
WARNING: file size is not a multiple of 10 frames
WARNING: file size is not a multiple of 10 frames
100.00%
WARNING: file size is not a multiple of 10 frames
WARNING: file size is not a multiple of 10 frames
100.00%
100.00%
WARNING: file size is not a multiple of 10 frames
WARNING: file size is not a multiple of 10 frames
100.00%
WARNING: file size is not a multiple of 10 frames
WARNING: file size is not a multiple of 10 frames
100.00%
WARNING: file size is not a multiple of 10 frames
WARNING: file size is not a multiple of 10 frames
100.00%
100.00%
SNR PASSED(INEQD53.POU): infinity >= 1000.00
SNR PASSED(OVERD53.POU): infinity >= 1000.00
SNR PASSED(OVERD63P.POU): infinity >= 1000.00
SNR PASSED(PATHD53.POU): infinity >= 1000.00
SNR PASSED(PATHD63P.POU): infinity >= 1000.00
SNR PASSED(TAMED63P.POU): infinity >= 1000.00
```

2.1.4.3.2 Annex A 的 Vad-cng 测试序列

进行测试的步骤如下：

（1） 使用算法编解码测试所有序列。步骤如下：

制作一个名为 VAD_CNG.bat 的批处理文件。其内容如下：

```
%1 -c -r63 -v dtx63.tin dtx63.xco
%1 -c -r53 -v dtx53mix.tin dtx53.xco
%1 -c -rdtxmix.rat -v dtx53mix.tin dtxmix.xco

%1 -d dtx63.rco dtx63.xou
%1 -d dtx53.rco dtx53.xou
%1 -d dtxmix.rco dtxmix.xou
%1 -d -fdtx63e.crc dtx63e.tco dtx63e.xou
%1 -d dtx63b.tco dtx63b.xou
```

在命令下输入 VAD_CNG lbccodec 回车即可。

（2） 验证编解码的正确性。步骤如下：

制作一个名为 compare_vad_cng.bat 的批处理文件。其内容如下：

```
@echo off
Rem compare VAD CNG sequence of G.723.1
Rem coder part
cmpcode dtx63.xco dtx63.rco
```

```
cmpcode dtx53.xco dtx53.rco
cmpcode dtxmix.xco dtxmix.rco

Rem decoer part
checksnr dtx63.xou  dtx63.rou
checksnr dtx53.xou  dtx53.rou
checksnr dtxmix.xou dtxmix.rou
checksnr dtx63e.xou dtx63e.rou
checksnr dtx63b.xou dtx63b.rou
```

在命令行下输入 compare_vad_cng 回车

2.1.5 用户程序调用 G.723.1 核心算法的方法

以上主要介绍了 ITU-T 提供的算法程序的外部功能并验证算法运行的正确性。接下来我们就要考虑使用这些代码，如何将这些代码集成到我们自己的程序中，如何调用这些算法？在实际的系统中语音信号是一个数据流的输入，而并不是从文件中读取，为了在用户程序中方便地使用 G.723.1 算法，需要提取 G.723.1 的核心算法并设计外部接口。

2.1.5.1 核心算法外部接口函数

这里的核心算法指的是不论在任何应用中都必须保留的代码部分。G.732.1 的核心算法外部接口函数有：

（1） 初始化函数：Init_Coder，Init_Decod，Init_Vad，Init_Cod_Cng，Init_Dec_Cng

（2） 编码和解码主函数：Coder，Decod

（3） reset_max_time 函数，如果是用低码率编码每帧编码以前要调用该函数。

2.1.5.2 核心算法外部接口数据

（1）外部程序需要给核心算法提供的参数及参数值的含义见表 2－6

表 2－6 G.723.1 核心算法参数及值的含义

参数	对应的变量	可能的取值
码率	WrkRate	Rate53，Rate63
静音压缩开关	UseVx	False，Ture
是否使用高通滤波（编码时）	UseHp	False，Ture
是否使用基音后滤波（解码时）	UsePf	False，Ture

（2）Coder 函数参数和返回值

函数原型：Flag　Coder(Word16 *DataBuff, char *Vout)

DataBuff：输入量，一帧 240 个采样点的输入数据流。

Vout：输出量，一帧编码以后的信息，最大为 24 字节，根据开头的 2bit 的帧类型和控制信息可以知道编码后信息的大小。表 2－7 为编码信息类型和长度：

表 2－7 编码信息类型和长度

Bit1,0	编码信息帧类型	信息帧长度	所需字节数
00	正常帧（高码率编码）	192bit	24
01	正常帧（低码率编码）	160bit	20
10	静音帧	32bit	4
11	不传送帧	2bit	1

（3）Decod 函数参数和返回值

函数原型：Flag　　Decod(Word16 *DataBuff, char *Vinp, Word16 Crc)
Vinp：输入量，一帧的编码信息。
Crc：输入量，此帧编码信息的 Crc 校验和。如果不使用校验和该参数为 0。
DataBuff：输出量，解码后的语音信息。

2.1.5,3 核心算法的基本调用流程

（1）设置参数

```
WrkRate UseVx UseHp UsePf
```

（2）初始化

```
Init_Coder();
Init_Decod();
    if (UseVx) {
        Init_Vad();
        Init_Cod_Cng();
    }
Init_Dec_Cng();
```

（3）如果要进行编码，方法如下：
1）读取一帧的数据。
2）如果有必要修改参数设置。
3）调用 Coder（）压缩此帧，并获得编码以后的信息。
（4）如果要进行解码，方法如下：
1）读取一帧编码以后的数据。
2）如果有必要修改参数设置。
3）设置 Crc。
4）调用 DeCod（）解压缩，并获得解码以后的语音数据。

2.1.5.4 ITU-T G.723.1 中非核心算法的代码

非核心算法代码如果没有使用可以去掉，它包括：
（1） util_lbc.c 中的 Read_lbc，Write_lbc，Line_Wr，Line_Rd。这几个函数用于从文件读取数据或将数据写入文件。在用 ADDA 采集语音时，不用这些函数。
（2） LBCCODEC.c：该文件中的 main 函数调用了核心算法接口函数，在使用中，这个文件需要做大改动。

2.1.5.5 核心算法接口函数设计实例

先去掉非核心算法代码，然后将我们设计的 LBCCODEC.C 文件代替原来的 LBCCODEC.C 文件。在这个设计实例中，我们在新 LBCCODEC.C 文件中设计了 6 个 G.723.1 核心算法接口函数，如表 2－8：

表 2－8 核心算法接口函数的使用方法

函数名	功能	参数	使用方法
G7231Init	G.723.1 模块总初始化	无	在使用 G.723.1 模块以前初始化。程序中再次使用 G.723.1 算法时，不必再调用此函数。
G7231SetParam	设置 G.723.1	setWrkRate：0 表示	在 G7231Init 函数后调

	编解码参数。这相当于 ITU-TG.723.1 算法程序命令行参数中的 option 的功能。	6.3K 编码，1 表示 5.3K 编码。 setUsePf：解码时是否使用后滤波 setUseHp：解码时是否使用高通滤波。 setUseVx：是否采用静音检测。 setReinitSize：相当于命令行方式下的-Rnum 参数。	用此函数。为将要开始的编码和解码设置参数。
G7231SysInit	根据新参数初始化 G.723.1 模块。	无	当调用 G7231SetParam 设置好参数以后，需要调用该函数进行一次初始化。
G7231Coder	编码一帧语音。	DataBuff：一帧语音缓冲的开始地址(大小为 16bit×240)。 Line：一帧编码结果存放的起始地址(大小为 8bit×24)	调用该函数可以得到一帧语音的编码结果。
G7231Decoder	解码一帧语音。	DataBuff：解码得到的一帧语音数据存放的起始地址(大小为 16bit×240)。 Line：一帧语音的编码(大小最大为 8bit×24) Crc：该一帧编码的 Crc 校验和。这是一个输入量，如果没有校验和，可用 0 代替。	调用该函数可以得到一帧语音的解码结果。
LineSize	返回 Line 的有效数据大小，Line 是某一帧的编码结果。	FirstByte：Line 的第一个 byte，是输入参数。	任何时候可调用该函数。

代码如下：

```
enum  Wmode   WrkMode = Both;
enum  Crate   WrkRate = Rate63;

Flag  UseHp = True;
Flag  UsePf = True;
Flag  UseVx = False;
Flag  UsePr = True;
```

```
int   ReinitSize = 0;
int CodeFrCnt = 0;
int DecodeFrCnt = 0;

void G7231Init()
{
    WrkMode = Both;
    WrkRate = Rate63;
    UseHp = True;
    UsePf = True;
    UseVx = False;
    UsePr = True;
    ReinitSize = 0;
    CodeFrCnt = 0;
    DecodeFrCnt = 0;
    G7231SysInit();
}
void   G7231SetParam(Flag   setWrkRate,Flag   setUseHp,Flag   setUsePf,Flag
setUseVx,int setReinitSize)
{
    if(setWrkRate == True)
        WrkRate = Rate63;
    else
        WrkRate = Rate53;
    UseHp   = setUseHp;
    UsePf   = setUsePf;
    UseVx   = setUseVx;
    ReinitSize = setReinitSize;
}
void G7231SysInit()
{
    Init_Coder();
    Init_Decod();
    if (UseVx)
    {
        Init_Vad();
        Init_Cod_Cng();
    }
    Init_Dec_Cng();
}
void G7231Coder(Word16 * DataBuff, char * Line)
{
    if (WrkRate == Rate53) reset_max_time();
    Coder(DataBuff, Line);
    CodeFrCnt++;
```

```
    if (ReinitSize > 0 && CodeFrCnt == ReinitSize)
    {
        G7231SysInit();
        CodeFrCnt = 0;
    }
}
void G7231Decoder(Word16 * DataBuff, char * Line, Word16 Crc)
{
    Decod(DataBuff, Line, Crc); // Crc == 0
    DecodeFrCnt++;
    if (ReinitSize > 0 && DecodeFrCnt == ReinitSize)
    {
        G7231SysInit();
        DecodeFrCnt = 0;
    }
}
int LineSize(char FirstByte)
{
    int Size;
    switch (FirstByte & 0x03)
    {
        case 0 :
            Size = 24;          break;
        case 1 :
            Size = 20;          break;
        case 2 :
            Size = 4;          break;
        default :
            Size = 1;         break;
    }
    return Size;
}
```

2.2 纯 C 代码在 54xDSP 上的移植

这部分的内容介绍如何将 ITU-T G.723.1 标准算法以纯 C 代码方式移植到 54xDSP 上。这里采用的 54xDSP 实验板为：闻亭(Wintech)公司的 TE54XUSB。实验板的 DSP 型号为：320VC5410PGE。开发工具是：TI 的 CCS2.0。这一步移植有两个目的：

1、在 54xDSP 上正确运行程序。由于在 54xDSP 上运行程序，需要涉及到存储区分配、如何读输入语音数据等一些问题，这和 windows PC 机上运行程序相比需要解决一些新的问题。解决这些问题，让程序在 54xDSP 上正确运行，将为后面的优化打下基础。

2、测试纯 C 文件在 54xDSP 上运行的速度。了解所需优化的程度。

在将 G.723.1 程序移植到 54xDSP 上前，首先需要考虑一个问题：如何验证 G.723.1 程序在上 54xDSP 上运行的正确性？

前面介绍了验证程序正确性的方法是通过编解码实际语音信号和测试测试序列。我们上面所用的这两种方法都要求 lbccodec 程序读取 PC 机硬盘上的语音文件或测试序列文件。为了仍采用上面的验证正确性的方法，需要解决两个问题：

1、DSP 程序在实验箱上运行，它如何读写 PC 机上的文件？

2、 PC 机上验证程序时，在命令行上输入 lbccodec [options] inputfile outputfile 来执行程序。这里命令行参数 inputfile 和 ouputfile 作为程序中的 main 函数的参数传递到程序内部。而在 DSP 上运行程序是通过点击 CCS 中的 RUN 命令或者上电复位，所以 main 函数参数无法传递。

2.2.1 DSP 程序读写 PC 机上的文件

在使用 DSP 接仿真器(Emulator)的情况下，DSP 上运行的程序可以直接读写 PC 机硬盘上的文件。实际上文件数据通过连接它们的 USB 口线（或者 JTAG 线）传递。这个功能是在 CCS 的 RTDX(Real-Time Data Exchange)技术支持下实现的。

DSP 程序读写 PC 机上的文件主要有两个方法：探测点(Probe Point)®和 File I/O 函数调用。File I/O 函数使用比较方便，下面介绍它的使用方法。

File I/O 函数的使用和标准 C 的 File I/O 函数使用方式基本一样。使用 FileI/O 函数时，首先要包含 stdio.h 头文件，接下来就可以使用 fopen、fread、fwrite、fclose 函数，这些和标准 C 一样，使用中最主要的区别是：默认时，DSP 程序读写文件的路径在程序所在工程目录下的 debug 目录。

2.2.1.1 读写 16bit 数的问题

DSP 的 RAM 最小存储单位是一个字（16bit），int 型变量大小是 16bit，于是 CCS2.0 中用 C 语言的 sizeof(int)得到的值是 1，而不象 VC6.0 中得到 2。在 DSP 和其它处理器系统（例如 PC 机）交换数据时，这个差异需要注意。例如，这个差异将会引起使用 File I/O 函数读写 16bit 数时的问题。

假设现在要从文件中读取一个 16bit 数放到 int 型变量 tmp 中，设文件的内容为

```
0x0000: 12 34 00 00
```

若用以下语句：

```
fread(&tmp,2,1,fp);  //读一个 2byte 的数据
```

或者

```
fread(&tmp,1,2,fp);  //读两个 1byte 的数据
```

则希望执行以后 tmp 的值为 0x3412。但是实际上却不是，设 tmp 在 DSP 的 RAM 中的地址为 0x1000，则执行该语句后 DSP RAM 中的数据是：

```
0x1000: 0012 0034
```

tmp 的值是 0x12。这就是说 DSP 中使用 fread 函数读取文件数据时，每次只能读取 8bit，而不能读 16bit。如果想读取 16bit 的数据，只能分别读取高 8bit 和低 8bit，然后拼装在一起。将一个 16bit 数读入 DataBuff[i]的程序如下：

```
if(fread(&tmp,1,1,fp) == 0)
   break;
DataBuff[i] = tmp;
if(fread(&tmp,1,1,fp) == 0)
   break;
DataBuff[i] += tmp<<8;
```

使用 fwrite 写 16bit 数据也会出现类似问题。将一个 16bit 数 DataBuff[i]写入文件的

程序如下：

```
fwrite(&DataBuff[i],1,1,fp);
tmp = DataBuff[i]>>8;
fwrite(&tmp,1,1,fp);
```

2.2.1.2 Ftell 函数返回值溢出问题

标准 C 库函数没有提供 GetFileLength()之类的函数来获得文件长度。Ftell 函数可以返回当前文件指针所在位置，所以获得文件长度可以先让文件指针定位在文件尾，然后调用 ftell 函数的方法来实现，例如：

```
fseek( *Ifp, 0L, SEEK_END ) ;   //文件指针定位在文件尾
Flen = ftell( *Ifp ) ;          //返回文件大小
```

在 DSP 中的 ftell 函数返回值有 65536 溢出问题，也就是说实际返回值是 16bit 表示的，即返回值是 65536 对实际文件大小求余的结果。例如：文件实际长度 489,540 字节，ftell 为返回值为 30788，两者差了 0x70000。

2.2.2 为测试测试序列的程序修改

为了解决文件读写和无法测试测试序列的问题，这里采用修改程序的方法。修改程序需要用到 2.1.5.5 节“用户程序调用 G.723.1 核心算法的方法”和 2.2.1 节“DSP 程序读写 PC 机上的文件”的相关知识点。

修改方法如下：在程序中定义两个结构体数组

```
struct tag_CodeParm CodeParm[CODE_FILE_NUM]=
    {
        {Rate63,True  ,"PATHC63H.TIN"   ,"PATHC63H.PCO"   },
        {Rate53,False ,"PATHC53.TIN"    ,"PATHC53.PCO"    },
        {Rate63,False ,"OVERC63.TIN"    ,"OVERC63.PCO"    },
        {Rate53,True  ,"OVERC53H.TIN"   ,"OVERC53H.PCO"   },
        {Rate63,False ,"CODEC63.TIN"    ,"CODEC63.PCO"    },
        {Rate53,False ,"INEQC53.TIN"    ,"INEQC53.PCO"    },
        {Rate63,True  ,"TAMEC63H.TIN"   ,"TAMEC63H.PCO"   }
    };
struct tag_DecodeParm DecodeParm[DECODE_FILE_NUM]=
    {
        {Rate53,False ,NULL      ,"INEQD53.TCO"    ,"INEQD53.POU"    },
        {Rate53,False ,NULL      ,"OVERD53.TCO"    ,"OVERD53.POU"    },
        {Rate63,True  ,NULL      ,"OVERD63P.TCO"   ,"OVERD63P.POU"   },
        {Rate53,False ,NULL      ,"PATHD53.TCO"    ,"PATHD53.POU"    },
    {Rate63,True,"PATHD63P.CRC","PATHD63P.TCO"    ,"PATHD63P.POU"   },
    {Rate63,True  ,"TAMED63P.CRC","TAMED63P.TCO"  ,"TAMED63P.POU"   }
    };
```

两个结构体数组分别定义了编码测试序列的输入输出文件名、参数和解码测试序列的输入输出文件名、参数。在 DSP 程序中，对于这个结构体数组中的每条信息，用 G7231SetParam 接口函数设置参数，用 fread 函数读取输入文件内容，用 G7231Coder 函数编码（或者用 G7231Decoder 函数解码），然后将结果用 fwrite 写到文件中。

程序如下：

```
/* ----------- 使用测试序列来测试编码解码的正确性 ----------- */
for(RotateCount=0; RotateCount < CODE_FILE_NUM;  RotateCount++)
{
    G7231Init();
    if(CodeParm[RotateCount].WrkRate == Rate63)
        G7231SetParam(True,CodeParm[RotateCount].UseHp,True,False,0);
    else
        G7231SetParam(False,CodeParm[RotateCount].UseHp,True,False,0);
    fpIn = fopen(CodeParm[RotateCount].FileIn,"rb");
    fpOut = fopen(CodeParm[RotateCount].FileOut,"wb");
    while(ReadWord16(DataBuff,Frame,fpIn) == Frame)
    {
        G7231Coder(DataBuff,Line);
        fwrite(Line,1,LineSize(*Line),fpOut);
    }
    fclose(fpIn);
    fclose(fpOut);
}
/* 解码编码的结果，用于测试解码的准确性 */
for(RotateCount=0; RotateCount < DECODE_FILE_NUM;    RotateCount++)
{
    G7231Init();
    if(DecodeParm[RotateCount].WrkRate == Rate63)
        G7231SetParam(True,True,DecodeParm[RotateCount].UsePf,False,0);
    else
        G7231SetParam(False,True,DecodeParm[RotateCount].UsePf,False,0);
    fpIn = fopen(DecodeParm[RotateCount].FileIn,"rb");
    fpOut = fopen(DecodeParm[RotateCount].FileOut,"wb");
    if(DecodeParm[RotateCount].FileCrc != NULL)
        fpCrc = fopen(DecodeParm[RotateCount].FileCrc,"rb");
    else
    {
        Crc = 0;
        fpCrc = NULL;
    }
    while(fread(Line,1,1,fpIn))
    {
        fread(Line+1,1,LineSize(*Line)-1,fpIn);
        if(fpCrc != NULL)
            ReadWord16(&Crc,1,fpCrc);
        G7231Decoder(DataBuff,Line,Crc);
        WriteWord16(DataBuff,Frame,fpOut);
    }
    fclose(fpIn);
    fclose(fpOut);
```

```
    if(fpCrc != NULL)
        fclose(fpCrc);
}
```

这样运行该程序就相当于运行 2.1.4.3 节测试测试序列时的“CodDecode lbccodec”批处理命令。最后还要用 compare 批处理命令比较编解码的正确性。

2.2.3 G.723.1 程序的存储空间分配

2.2.3.1 TMS320VC5410 的存储空间分布特点

图 2－1 是 TMS320VC5410 的存储空间分布图⑦

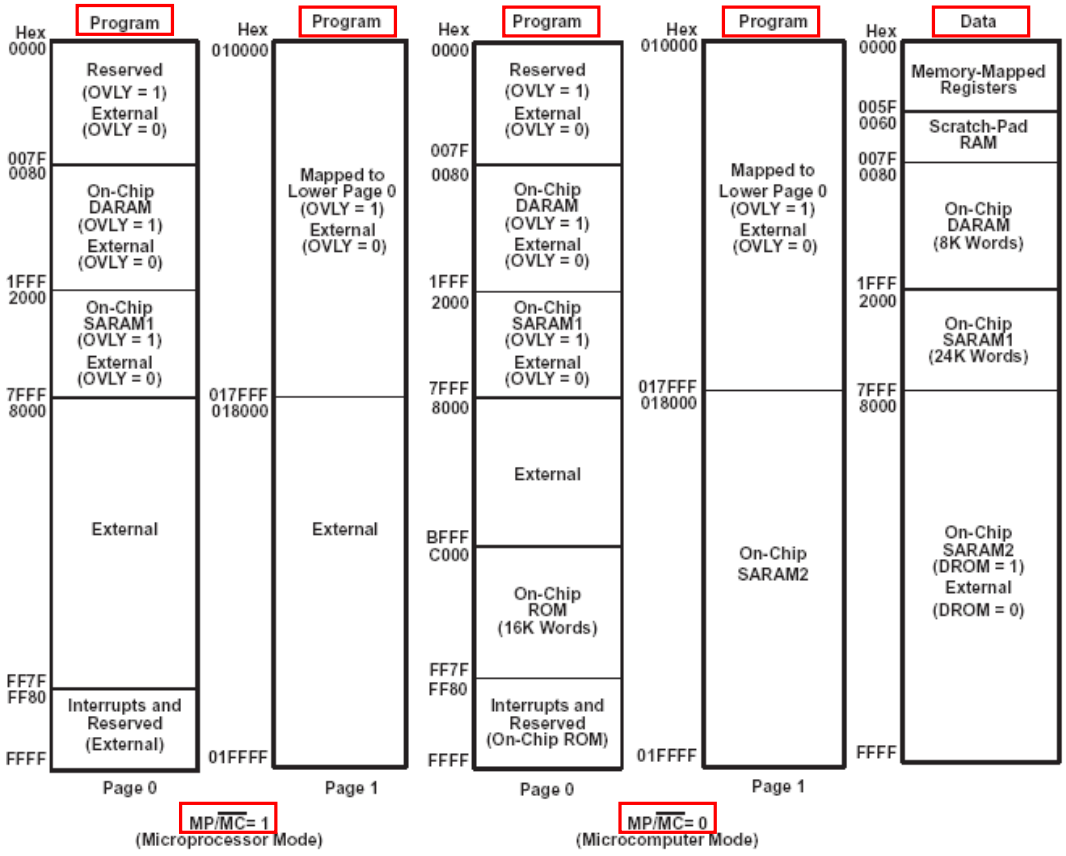

Figure2-1. **Memory Map**

一、程序区：

1、MP 和 MC 模式的区别如表 2－9 所示：

表 2－9 MP 和 MC 模式的区别

有区别的地址范围	MP	MC
C000-FFFF（共 16K）	External	On-Chip ROM
18000-1FFFF	External	SARAM2

（32K）

说明：MC 模式下使内部 ROM 使能。其中的 FF80 地址为上电复位中断向量所在地址。在 MC 模式下复位中断和起始程序使用内部 ROM 中的程序，该程序包含 bootloader，能够将用户程序加载到内部 RAM 中，然后跳到用户程序执行。所以如果想使用 bootloader 加载存储在外部 FLASH 中的程序并运行应该使用 MC 方式；如果想通过仿真器下载，并在 CCS 中仿真，那么需要使用 MP 方式。另外在 C000-FF80 之间的 ROM 存放了一些如三角函数表等常用数据，在 MC 模式下可以直接访问这些数据。

另外 MC 方式的 18000-1FFFF 地址范围是 SRAM2。

2、OVLY=0 和 OVLY=1 的区别如表 1－10 所示：

表 2－10 OVLY=0 和 OVLY=1 的区别

有区别的地址范围	OVLY=0	OVLY=1
0000-7FFF（共 32K）	External	和 Data 区共用
x0000-x7FFF（x 表示页号，x 从 1 到 127）	External	和 Data 区共用

说明：OVLY＝1 时，表示采用覆盖方式，覆盖指两方面：程序区低 32K 和数据区低 32K 共用；程序区的 1－127 页低 32K 也和数据区低 32K 共用。

OVLY＝1，使得程序和数据共享一块 RAM，这能使 RAM 资源分配更合理。

二、数据区：

DROM=0 和 DROM=1 的区别见表 2－11

表 2－11 DROM=0 和 DROM=1 的区别

有区别的地址范围	DROM =0	DROM =1
7FFF-FFFF（共 32K）	External	使用 DRAM2

说明：OVLY 使得程序区和数据区共用一块 RAM，而 DROM 则相反，使得数据区能够使用程序区的 RAM。DRAM2 在 MC=0 时是地址范围为 18000-1FFFF 的程序区，如果设置 DROM=1 同时 MC=0 则 7FFF-FFFF 地址范围的数据区和 18000-1FFFF 地址范围的程序区使用同一块 RAM。表 2－12 为 5410 低 32K 数据区结构：

表 2－12 5410 低 32K 数据区结构

地址范围	名称	特点
0000-005F	Memory-mapped register	该部分是 CPU 寄存器所在空间
0060-0080	Scratch-Pad RAM	该部分虽然不是 CPU 寄存器，但是它同样可以用 Memory-mapped register 的寻址方式（例如使用 STM、STLM、LDM 指令寻址）寻址。
0080-1FFF	DARAM	双操作数 RAM。DARAM 和 SRAM 相比，DARM 能够允许在单周期内同时读写，而不会有流水线冲突。所以将需要双操作数访问的数据放到 DARAM 区域。

2000-7FFF	SRAM1	单操作数 RAM。

2.2.3.2 TMS320VC5410 的存储空间配置

上面提到决定存储空间配置的主要是三个量 MP/MC、OVLY 和 DROM。该三个量分别对应 PMST（Processor Mode Status Register）寄存器的三个 bit 位，见图 2－2：

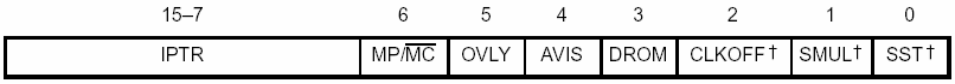

图 2－2 PMST 寄存器结构

复位的时候OVLY=1，DROM=0，MP/MC的值在复位的时候从MP/MC引脚读取。在程序的运行过程中可以修改MP/MC、OVLY、DROM的值。

2.2.3.3 TE54xUSB 实验箱的存储空间配置

MP/MC：TE54xUSB 实验箱的 DSP 的 MP/MC 引脚和跳线 JP3 连接。JP3 短接 MP/MC 引脚为高电平，复位时 DSP 处于 MP 模式；JP3 不短接 MP/MC 引脚为低电平，复位时 DSP 处于 MC 模式。

外部程序空间：实验箱的外部程序空间接了一个 64K 的 RAM（型号 CY7C1021BV33），若某段程序空间为 Exteral，则指该外部程序存储器。

外部数据空间：实验箱的外部数据空间接了一个 128K 的 FLASH（型号 SST29LE010），这 128K 的 FLASH 被分为 4 页，通过 DSP 的 IO 口可以选择 4 页中的一页。被选择的一页，也就是 32K 的 FLASH 映射到数据区的 8000-FFFF。所以当 DROM=0 时，8000-FFFF 地址范围数据区对应外部 FLASH。

2.2.3.4 G7321 程序的存储空间配置及其 CMD 文件内容

首先看一下纯 C 代码的 G7321 程序的程序数据空间大小。让程序编译的时候输出 map 文件（通过设置 CCS 中 build option/ linker/ map filename），从 map 文件可知程序中各个段的大小，见表 2－13：

表 2－13 纯 C 的 G7231 程序所需空间大小

段名	大小	说明
.text	77f6H	代码段
.cinit	2618H	需要初始化的数组所在的段（包括 G.723.1 中的大量表）
数据段，堆栈等	3800H	

为了提高运行速度最好能够将程序加载到内部 RAM 中运行，G7231 总共所需空间为 D60EH，可以全部放到片内 RAM 中。另外程序空间比数据空间大。为此需要使用 MC 模式，以利用 18000-1FFFF 的片内程序空间；取 OVLY=1，DROM=0。所以设置 PMST 寄存器的内容为 0x124。注意实验箱 JP3 跳线仍然不短接，因为程序是在连接仿真器的方式下运行，虽然复位时为 MP 模式，但是可以通过修改 PMST 寄存器的内容设置为 MC 模式。接仿真器是，可以通过修改 c5410.gel 文件的 PMST_VAL 的值来修改 PMST 寄存器的初始值（需要重新编译运行）。

.text 段定位到 18000 开始的程序空间。数据段需要定位到 DRAM，所以定位到 0800 地址开始的数据空间。段定位的方法有两种：visual link 和 text link。Text link 也就是使

用 CMD 文件段定位的方法，这里采用 CMD 文件的方法。G7321 程序的 CMD 文件内容如下：

```
MEMORY  {
        PAGE 0:
               PROG1:  org = 18000h     len = 08000h
               PROG2:  org = 5800h      len = 2800h
        PAGE 1:
               DARAM:  org = 80h        len = 5780h
         }
SECTIONS{
        .text    : >       PROG1   PAGE 0
        .cinit   : >       PROG2   PAGE 0
        .switch  : >       PROG2   PAGE 0
        .data    : >       DARAM   PAGE 1
        .bss     : >       DARAM   PAGE 1
        .const   : >       DARAM   PAGE 1
        .system  : >       DARAM   PAGE 1
        .stack   : >       DARAM   PAGE 1
         }
```

2.2.4 编译连接中的问题

用 CCS 新建一个工程，工程名为 G7231，将相关程序文件拷贝到该工程目录下。将相关文件中的.c 文件和 CMD 文件添加到工程中。在 CCS 菜单中选择编译命令。编译器提示的错误和修改方法如下：

1、编译提示类型定义没有找到，这是因为 typedef.h 条件编译选项中没有 CCS 编译器的选项。将 typedef.h 文件中的类型定义改为：

```
typedef  long  int   Word32;
typedef  short int   Word16;
typedef  short int   Flag;
```

2、添加 rts.lib。在 C 语言编写的 DSP 中经常要用到 rts.lib 库中的函数。添加 ti\c5400\cgtools\lib\rts.lib 到工程中。

2.2.5 G.723.1 的 DSP 程序运行正确性的验证

•虽然这里验证的是纯 C 代码的 G7231，但是优化以后的程序也用该方法验证。前面已经为测试测试序列修改好了代码，这里只要运行就可以了，步骤如下：

1、确保所有的需要输入的语音文件和测试序列文件已经拷贝到 Debug 目录下

2、在 CCS 中编译运行 G7231 程序

3、在 PC 机上运行 compare 批处理文件，检查编解码结果的正确性。

2.2.6 G.723.1 的 DSP 程序运行的速度

2.2.6.1 CCS profile 的使用

CCS 的 profile 功能能够测试某个函数(function)或者某段程序(range)运行需要多少个指令周期(CPU clock)。某次 profile 的测试结果如图 2－3：

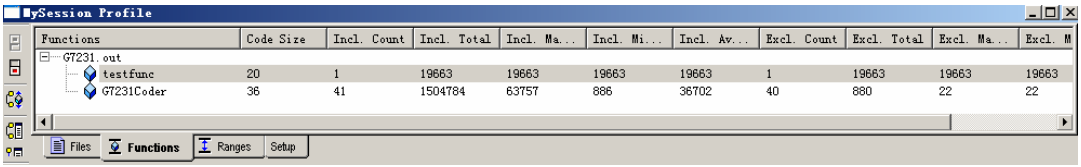

图 2－3 CCS profile 窗口实例

profile 窗口中的数据的含义：Code Size 表示测试区代码量大小；Incl. Count 表示测试区被执行的次数;Incl. Total 表示程序执行测试区代码消耗的总时间（单位指令周期）；Incl. maximum 表示每执行一次测试区代码需要消耗的时间中最多的一次。Incl. Average 表示平均执行一次测试区代码需要消耗的时间。Incl.和 Excl.的区别是：前者要包含测试区内子函数运行的时间，后者不包含。

Profile 窗口中显示的数据是否可信？经过测试发现，当执行测试代码所需时间少于 65536 时是正确的。但是当时间超过 65535 时则不对，显示的数据是真实数据对 65536 求余的结果。

注：曾经出现过一次能够在超过 65535 时也能正确显示的情况，除此至外再没有出现过。为什么会出现显示不对的情况，目前尚不知道。

由于 G7231 程序的一次编码所需指令周期大于 65536，所以 profile 不能用于测试 G7231 程序的运行时间。

2.2.6.2 TMS320VC5410 运行在最高速度的设置

如果不用 profile 测试速度。可以用秒表测试编码一次的时间。在测试之前需要确认一件事情，就是 TMS320VC5410 DSP 确实运行在了它所能运行的最高速度－－100MHz。

5410DSP 的倍频数是可调的。实验箱外部晶振为 20MHz，所以需要 5 倍频，也就是将 4 写入 CLKMD 寄存器的 PLLMUL 位®。设置的具体方法参考 SPRU131(CPU and Peripherals)。另外虽然程序中没有访问外部程序空间和数据空间，这里也将外部数据访问等待时间设置为 0。这可通过向 SWWSR 和 SWCR 寄存器写入 0 实现。设置程序如下：

```
        STM     #0, SWWSR
        STM     #0, SWCR

        ;设置 PLL
        STM #0b,CLKMD
WaitPLL:
        LDM     CLKMD, A
        AND     #1,     A
        BC      WaitPLL,   ANEQ
        STM     #0,        CLKMD
        STM     #4287H,    CLKMD
        RPT     #1000
            NOP
```

这样一来 5410 运行在了最高速度上，为了检测是否是 100MHz，采用以下程序：

```
        while(1){
        wait();
        twinkle _led();}
```

wait 函数用汇编编写，可以精确计算它的指令数，估计为 40627200（个 cpu clock），而实际测试灯闪烁为每 60 秒 140 次。以此计算出每个 cpu clock 的周期为 (60/140) /40627200＝1.0548879e-8,时钟频率 94796800Hz 约 94MHz。这在误差范围内可以认为是运行在了 100MHz。

2.2.6.3 G.723.1 程序运行速度的测试

由于已知 6.3K 的编码速度是编解码中最慢的。这里只测试 6.3K 的编码速度。首先制作一个含有 15 帧的语音文件。在程序中将 15 帧的语音数据都读入 RAM 中。然后对 15 帧语音数据反复编码 TEST_ROTATION_NUM 次，即运行下面代码。

```
extinguish_led();
for(RotateCount = 0; RotateCount < TEST_ROTATION_NUM; RotateCount++)
{
    G7231Init();
    G7231SetParam(True,True,True,False,0);
    for(i=0, pLine = Line; i < TEST_FRM_NUM; i++, pLine += 24)
        G7231Decoder(DataBuff,pLine,0);
}
light_led();
```

用秒表测量运行这部分代码所需的时间：先让程序运行到（使用 CCS 的 run to curse 功能）第一行代码处，然后同时启动秒表和运行程序。当程序运行到最后一行（或者 led 灯点亮时）处时，记下时间。

编码一帧的时间为 time/TEST_ROTATION_NUM/15。测试得到编码 30 帧所需时间为 14.53s。如果需要实现实时编码，按照 G.723.1 的要求至少需要在 30×30ms=0.9s 内编码完毕，所以速度所需提高的倍数至少为 14.53/0.9=16.14 倍。通过编码时间大致计算 MIPS（执行一段程序时运行的指令周期总数）的方法是：举例说明，100MIPS 如果在 0.9s 内正好编码完 30 帧，则编码运算量为 100MIPS，由于现在需 14.53s，所以为 100*14.53/0.9=1614MIPS。

2.3 代码优化方法和过程

从纯 C 代码的移植知道程序的速度至少需要提高 16.14 倍。这里讨论如何提高速度。

2.3.1 代码运行速度慢的原因

ITU-T 的 G.723.1 代码有两个版本，定点程序和浮点程序。定点程序用定点表示数据，采用定点运算；浮点程序则用浮点表示数据，采用浮点运算。定点程序和浮点程序流程是一样的，只是基本运算操作不一样。这里的基本操作包括加、减、乘、除和移位。G.723.1 算法要求这些基本运算包含溢出控制。例如

1、16bit 最大有符号数 0x7FFF 加 1 以后还是最大数 0x7FFF。如果在 C 语言中用 0x7FFF+1 来实现那么得到的结果是 0x8000，而 0x8000 则是最小的负数，不符合要求。

2、16bit 最小有符号数 0x8000 左移一位后还是 0x8000。如果在 C 语言中用 0x8000<<1 实现得到的是 0。

为了进行溢出控制定点程序和浮点程序采用了不同的方法。定点程序使用函数调用方法（简称这类函数为 BASEOP 函数）。例如 16bit 加法用 add 函数实现，在 add 函数中进行溢出处理。浮点程序则直接用运算符实现运算，这是因为浮点表示法的数值表示范

围很大，对于 16bit 线性 PCM 表示的语音数据，溢出不太可能，所以不用进行溢出控制。

以 16bit 加法为例：

```
定点实现：add(a,b)  //a,b 是 Word16 型变量
浮点实现：a+b       //a,b 是 float 型变量
```

假设实现 add(a,b)需要 20 条指令。在带有浮点协处理器的 CPU 上实现 a+b 可以用一条 fadd 指令实现。所以在 PC 机上运行浮点程序比定点程序要快得多。

和带有浮点协处理器的 CPU 相反，由于 C5410DSP 没有浮点协处理器，所以不能用一条指令实现浮点加法，一般通过函数调用实现。但是 C5410 却有带溢出控制的 16bit 加法，所以 add(a,b)函数就可以用单条 ADD 指令实现。

所以浮点程序适合在 PC 机 CPU 上运行，定点程序适合在 DSP 上运行，这是由它们的 CPU 硬件结构决定的。

由于 C 语言不能用一个运算符描述带溢出控制的运算，只能用大段函数描述，这就是纯 C 代码运行速度慢的关键所在。

2.3.2 优化方法的讨论和尝试

从上面的讨论知道优化的主要手段就是用 DSP 的单条指令代替程序中的 BASOP 函数调用。采用 BASOP 函数和单指令相比主要增加代码有两部分：

1、第一部分：出栈入栈操作。调用 BASEOP 函数之前需要将参数压入堆栈，BASEOP 函数再从堆栈取出参数。

2、第二部分：BASOP 函数内部执行溢出控制所消耗的代码。

由于考虑到将所有调用 BASEOP 的代码改为单条指令基本上意味着要用全汇编实现，全汇编实现工作量较大，所以首先考虑了其它的优化方法。

一、 内联 BASOP 函数

C＋＋语言的 inline 特性可以将函数设置为内联函数，内联函数将函数内代码直接嵌入到调用处，这节省了参数压栈和出栈的操作，能够节省运行时间。但是由于直接嵌入将使代码量大大提高，由于本身代码量已经较大，而且这种方法不能解决第二部分增加的代码，所以不采用此方法。

二、 优化 BASOP 函数

不使用全汇编实现的一个折中的办法是只优化 BASOP 函数。例如在 add 函数中的溢出控制的代码可以用一条带溢出控制的 ADD 指令实现。这种方法可以解决第二部增加的代码，但是却不能解决第一部分增加的代码。

该方法优化以后的速度为：30 帧 6.3K 编码所需 5.94 秒，提高了约 3 倍。但是没有达到 16.14 倍的要求。

三、 C 编译器的优化选项

CCS 的 C 编译器支持多级优化，优化后 C 代码的执行效率更高。选择-O3 级优化，测试优化后的代码速度提高了约 1/14。可见 C 编译器的优化功能不能达到所需的优化目的。

四、 优化算法逻辑

另外的一种优化思路是着眼于优化 G.723.1 的算法逻辑。它从算法原理入手，根据原理提出更简单的实现方法，从而减少运算量。这种方法是使用全汇编优化的一个补充，但是它不能够代替全汇编优化，因为它所能提高的运算速度有限，而且常以减少算法的精度（不能 100％通过测试序列）为代价[⑨]。

无论是方法一还是方法二都不能同时解决两部分增加的代码。所以只有采用全汇编方案才能达到所需优化的程度。

2.3.3 从关键代码入手的全汇编优化方案

如果采用全汇编的优化方案，那么可以先估计一下工作量。根据已经优化成功的 G.723.1 算法的代码量，可以估计工作量。SPIRIT 公司（http://www.spiritcorp.com/g723.html）优化的代码量为 11.5Kwords，如果按照一条指令 1word 的大致估计，需要 11776 条指令，也就是说 1 万多行的代码量。

考虑到工作量较大，可以采用从关键代码入手的方法（该方法不是我首次提出）。所谓关键代码就是占总运算量比重较大的函数（或者某部分代码）。这部分代码一般是一个多重循环体，例如每重循环为 10 次的 4 重循环，其第 4 重循环内部指令需要执行 10×10×10×10＝10000 次。所以，如果能将第 4 重循环内部指令周期减少一个，总运行时间减少 10000 个指令周期。优化关键代码的“性价比”是很高的。

2.3.3.1 关键代码的寻找

关键代码可以利用工具来寻找。例如使用 CCS 的 profile 工具和 Microsoft Visual C++6.0（后面简称 VC）的 profile 工具。通过测试某个模块（例如 6.3 编码模块）中的各个函数运行所需的时间，然后通过对比，可以知道该模块中那些函数是关键代码。

使用 CCS 的 profile 可以测试函数的运行时间，但是它有两个问题：1、上面提到的“65536 溢出问题”。2、测试时间很长。在打开 profile 的情况下运行程序，6.3K 每帧编码时间超过 4 个小时。

VC 的 profile 工具没有以上的两个问题，但是有一点要注意：由于它测试的代码是运行在 PC 机上，从汇编指令级来看，VC 编译的程序和在 CCS 中编译的程序是不一样的，所以它的测试结果和在 DSP 上运行的真实情况不完全相符。由于我们只是想得到大概的关键代码分布情况，所以可以暂时不考虑这个误差。

2.3.3.1.1 VC++6.0 profile 的使用

在使用 profile 之前需要在 project setting/Link/General 中开启 Enable profiling 和 Generate mapfile；在 project setting/Link/Debug 中填写 map 文件存放地址[10]。然后重新编译。选择 compile/profile…/Custom，然后打开 profile 批处理文件，点击 OK 开始测试。

2.3.3.1.2 G.723.1 运算量分布情况

2.3.3.1.2.1 6.3K 编码运算量分布

profile 批处理文件的内容如下：

```
PREP /FT /OM /EXCALL /INC Exc_lbc.obj /INC util_lbc.obj /INC lpc.obj /INC lsp.obj
/INC cod_cng.obj /INC coder.obj /OI Dspcode.pbi /OT Dspcode.pbt %1
if errorlevel == 1 goto done
PROFILE     /I     dspcode.pbi     /O     dspcode.pbo          %1     -r63
-c ..\testseq\my.bin ..\testseq\mydspcode.lin
if errorlevel == 1 goto done
PREP /IO dspcode.pbo /IT dspcode.pbt /OT dspcode.pbt
if errorlevel == 1 goto done
PLIST /ST dspcode.pbt
:done
```

测试结果如下（Func+Child 表示函数执行时间包含该函数内子函数的执行时间）：

Func Time	%	Func+Child Time	%	Hit Count	Function
1263.927	37.0	1264.586	37.0	268	_Find_Best (exc_lbc.obj)
1086.093	31.8	1104.533	32.3	200	_Find_Acbk (exc_lbc.obj)
366.940	10.7	366.940	10.7	100	_Estim_Pitch (exc_lbc.obj)
98.784	2.9	98.784	2.9	200	_Sub_Ring (lpc.obj)
96.680	2.8	96.680	2.8	200	_Comp_Ir (lpc.obj)
94.554	2.8	94.554	2.8	200	_Upd_Ring (lpc.obj)
90.179	2.6	121.831	3.6	50	_Comp_Lpc (lpc.obj)
87.417	2.6	87.417	2.6	50	_Lsp_Svq (lsp.obj)
59.720	1.7	59.720	1.7	50	_Error_Wght (lpc.obj)
42.377	1.2	42.377	1.2	200	_Comp_Pw (exc_lbc.obj)
33.500	1.0	34.259	1.0	400	_Decod_Acbk (exc_lbc.obj)
19.961	0.6	19.961	0.6	200	_Durbin (lpc.obj)
19.132	0.6	19.132	0.6	50	_AtoLsp (lsp.obj)
15.899	0.5	15.899	0.5	250	_Vec_Norm (util_lbc.obj)
11.520	0.3	11.520	0.3	50	_Rem_Dc (util_lbc.obj)
6.424	0.2	6.424	0.2	200	_LsptoA (lsp.obj)
5.641	0.2	5.641	0.2	200	_Filt_Pw (exc_lbc.obj)
5.410	0.2	3414.418	100.0	50	_Coder (coder.obj)
2.052	0.1	2.052	0.1	1100	_Get_Rez (exc_lbc.obj)
1.592	0.0	89.009	2.6	50	_Lsp_Qnt (lsp.obj)
1.387	0.0	7.811	0.2	50	_Lsp_Int (lsp.obj)
1.198	0.0	1.198	0.0	50	_Update_Acf (cod_cng.obj)
0.980	0.0	0.980	0.0	50	_Lsp_Inq (lsp.obj)
0.880	0.0	0.880	0.0	50	_Wght_Lpc (lpc.obj)
0.797	0.0	0.797	0.0	85	_Gen_Trn (exc_lbc.obj)
0.598	0.0	0.598	0.0	200	_Fcbk_Pack (exc_lbc.obj)
0.310	0.0	0.310	0.0	50	_Mem_Shift (util_lbc.obj)
0.294	0.0	0.294	0.0	50	_Read_lbc (util_lbc.obj)
0.204	0.0	0.329	0.0	50	_Line_Pack (util_lbc.obj)
0.138	0.0	1265.460	37.1	200	_Find_Fcbk (exc_lbc.obj)
0.125	0.0	0.125	0.0	950	_Par2Ser (util_lbc.obj)
0.045	0.0	0.045	0.0	50	_Line_Wr (util_lbc.obj)
0.003	0.0	0.003	0.0	1	_Init_Coder (coder.obj)

2.3.3.1.2.2 5.3K 编码运算量分布

profile 批处理文件的内容和 6.3K 编码的 profile 批处理文件相同，测试结果如下：

Func Time	%	Func+Child Time	%	Hit Count	Function
1179.879	43.0	1198.976	43.7	200	_Find_Acbk (exc_lbc.obj)
369.986	13.5	369.986	13.5	100	_Estim_Pitch (exc_lbc.obj)
274.829	10.0	274.829	10.0	200	_D4i64_LBC (exc_lbc.obj)

```
 98.154   3.6      98.154   3.6     200 _Sub_Ring (lpc.obj)
 96.818   3.5      96.818   3.5     200 _Comp_Ir (lpc.obj)
 95.155   3.5      95.155   3.5     200 _Upd_Ring (lpc.obj)
 90.067   3.3     122.257   4.5      50 _Comp_Lpc (lpc.obj)
 87.902   3.2      87.902   3.2      50 _Lsp_Svq (lsp.obj)
 86.750   3.2      86.750   3.2     200 _Cor_h_X (exc_lbc.obj)
 62.333   2.3      62.333   2.3      50 _Read_lbc (util_lbc.obj)
 60.053   2.2      60.053   2.2      50 _Error_Wght (lpc.obj)
 47.588   1.7      47.588   1.7     200 _Cor_h (exc_lbc.obj)
 42.274   1.5      42.274   1.5     200 _Comp_Pw (exc_lbc.obj)
 34.068   1.2      34.830   1.3     400 _Decod_Acbk (exc_lbc.obj)
 20.507   0.7      20.507   0.7     200 _Durbin (lpc.obj)
 19.626   0.7      19.626   0.7      50 _AtoLsp (lsp.obj)
 18.837   0.7      18.837   0.7      50 _Rem_Dc (util_lbc.obj)
 16.221   0.6      16.221   0.6     250 _Vec_Norm (util_lbc.obj)
  8.354   0.3       8.354   0.3     200 _G_code (exc_lbc.obj)
  8.165   0.3       8.165   0.3     200 _LsptoA (lsp.obj)
  5.991   0.2    2679.199  97.7      50 _Coder (coder.obj)
  5.767   0.2       5.767   0.2     200 _Filt_Pw (exc_lbc.obj)
  3.289   0.1     420.810  15.3     200 _ACELP_LBC_code (exc_lbc.obj)
  2.088   0.1       2.088   0.1    1100 _Get_Rez (exc_lbc.obj)
  1.595   0.1      89.497   3.3      50 _Lsp_Qnt (lsp.obj)
  1.453   0.1       9.618   0.4      50 _Lsp_Int (lsp.obj)
  1.199   0.0       1.199   0.0      50 _Update_Acf (cod_cng.obj)
  0.982   0.0       0.982   0.0      50 _Lsp_Inq (lsp.obj)
  0.886   0.0       0.886   0.0      50 _Wght_Lpc (lpc.obj)
  0.315   0.0       0.315   0.0      50 _Mem_Shift (util_lbc.obj)
  0.170   0.0       0.316   0.0      50 _Line_Pack (util_lbc.obj)
  0.146   0.0       0.146   0.0     900 _Par2Ser (util_lbc.obj)
  0.056   0.0     420.895  15.4     200 _Find_Fcbk (exc_lbc.obj)
  0.044   0.0       0.044   0.0      50 _Line_Wr (util_lbc.obj)
  0.029   0.0       0.029   0.0     200 _search_T0 (exc_lbc.obj)
  0.005   0.0       0.005   0.0      50 _reset_max_time (exc_lbc.obj)
  0.003   0.0       0.003   0.0       1 _Init_Coder (coder.obj)
```

2.3.3.1.2.3 解码运算量分布

profile 批处理文件的内容：

```
PREP /FT /OM /EXCALL /INC decod.OBJ /INC exc_lbc.OBJ /INC dec_cng.OBJ /INC
util_lbc.OBJ /INC lsp.OBJ
/INC lpc.OBJ /OI Dspcode.pbi /OT Dspcode.pbt %1
if errorlevel == 1 goto done
PROFILE     /I     dspcode.pbi     /O     dspcode.pbo          %1     -r63
-d ..\testseq\MY.PCO ..\testseq\MY.POU
if errorlevel == 1 goto done
PREP /IO dspcode.pbo /IT dspcode.pbt /OT dspcode.pbt
```

```
if errorlevel == 1 goto done
PLIST /ST dspcode.pbt
:done
```

测试结果如下：

```
       Func         Func+Child          Hit
       Time   %        Time     %      Count  Function
-------------------------------------------------------
     73.376  30.8      76.795  32.2      200 _Spf (lpc.obj)
     36.286  15.2      36.286  15.2      200 _Synt (lpc.obj)
     19.731   8.3      19.731   8.3      200 _Find_B (exc_lbc.obj)
     17.159   7.2      22.482   9.4      200 _Scale (util_lbc.obj)
     17.121   7.2      17.504   7.3      200 _Decod_Acbk (exc_lbc.obj)
     12.329   5.2      17.777   7.5       50 _Comp_Info (exc_lbc.obj)
     11.695   4.9      45.400  19.0      200 _Comp_Lpf (exc_lbc.obj)
     11.503   4.8      11.503   4.8      200 _Find_F (exc_lbc.obj)
      8.867   3.7       8.867   3.7      250 _Vec_Norm (util_lbc.obj)
      6.368   2.7       6.368   2.7      200 _LsptoA (lsp.obj)
      6.147   2.6       6.147   2.6      200 _Filt_Lpf (exc_lbc.obj)
      5.231   2.2     238.105  99.8       50 _Decod (decod.obj)
      3.580   1.5       3.580   1.5      200 _Comp_En (util_lbc.obj)
      2.927   1.2       2.927   1.2      328 _Sqrt_lbc (util_lbc.obj)
      1.400   0.6       7.768   3.3       50 _Lsp_Int (lsp.obj)
      1.377   0.6       1.377   0.6       50 _Lsp_Inq (lsp.obj)
      1.287   0.5       2.471   1.0      200 _Get_Ind (exc_lbc.obj)
      0.565   0.2       0.762   0.3      200 _Fcbk_Unpk (exc_lbc.obj)
      0.437   0.2       0.576   0.2       50 _Line_Unpk (util_lbc.obj)
      0.384   0.2       0.384   0.2      200 _Get_Rez (exc_lbc.obj)
      0.284   0.1       0.284   0.1       50 _Write_lbc (util_lbc.obj)
      0.196   0.1       0.196   0.1       17 _Gen_Trn (exc_lbc.obj)
      0.139   0.1       0.139   0.1      950 _Ser2Par (util_lbc.obj)
      0.094   0.0       0.094   0.0       51 _Line_Rd (util_lbc.obj)
      0.002   0.0       0.002   0.0        1 _Init_Decod (decod.obj)
      0.000   0.0       0.000   0.0        1 _Init_Dec_Cng (dec_cng.obj)
```

2.3.3.1.2.4 运算量分布分析

以 6.3K 编码为例。_Find_Best 占用总运行时间的 37.0%，_Find_Acbk 占用 32.3%，_Estim_Pitch 占用 10.7%，只这三个函数就占用 80%的总运行时间。这三个函数是关键代码。所以改写 C 为汇编的时候从_Find_Best 函数开始，按照以上列表从上向下依次改写。当程序速度达到要求时，可停止优化。以此以最小的工作量实现所需的优化。

2.3.3.2 优化过程中的调试方法

寻找了关键代码以后确定了代码各个函数修改的先后次序。调试方法将决定以何种方式来修改这些函数。

由于编写汇编程序比 C 程序更容易出错，如何能够快速找到程序中的错误（BUG），

是很重要的问题。从实际编写过程可知，调试所需的工作量是编写的两三倍。

2.3.3.2.1 缩小错误代码的区域

一个简单的事实是：如果能够确定程序的 BUG 一定在某 10 行代码内，那么你很容易发现 BUG 在哪里，如果只知道在某 1000 行代码内，那么就要麻烦很多。所以，错误代码区域越小越容易找到错误所在。

以上的事实说明，不能在全部修改完所有函数以后再测试程序是否正确，那样很难找到 BUG 所在。实际中采用以下措施：

一、 采用逐步将 C 程序替换为汇编程序的“蚕食方法”。

首先我们确认了纯 C 代码的正确性（通过测试测试序列），然后我们将用汇编编写好的_Find_Best 函数替换 C 语言编写的 Find_Best 函数。如果汇编的_Find_Best 函数和 C 的 Find_Best 有相同的功能，则一定能够通过测试序列的测试，我们用此方法来保证_Find_Best 函数的正确性。如果不能通过测试，那么说明 BUG 一定存在于_Find_Best 函数内。只有当通过了测试序列以后，才开始下一个函数的修改，同样采用该方法，直到最后所有的函数都用汇编代替，于是得到了一个正确的 G7231 全汇编程序。以此我们将错误代码区域限制在一个函数内。

二、 采用边写边调试的方法。

第一个措施虽然减小了错误代码区域，但是对于比较大的函数，区域仍然过大。能否做到每编写约 10 行代码就验证这 10 行代码的正确性，以进一步减小区域？

这就遇到一个问题，以_Find_Best 函数改写为例，既然_Find_Best 函数才编写了一部分（剩余的部分不能用 C 代替，因为在同一个函数内 C 和汇编的混合不好实现），那又如何运行整个 G7231 程序测试测试序列？参考如下编写了一部分的_Find_Best 函数

```
;***    if ( Olp < (Word16) (SubFrLen-2) ) {
;***        Temp.UseTrn = (Word16) 1 ;
    LD      Olp, B
    SUB     #(SubFrLen-2), B
    BCD     Not_Update_Impulse, BGEQ
    STM     #0, BK
    ST      #1, Temp+BESTDEF.UseTrn      ;编写到这条语句，后面还有代码未编写
```

虽然不能测试测试序列，但是只有部分_Find_Best 的 G7231 程序能够编译和运行，只是编码结果不对。但是有一点可以确认，如果让程序只运行到新编写的最后一条代码处(代码中第 7 行)，所有变量值（指代码中的变量的值）和运行流程都应该是正确的，若有误则说明新编的 10 行代码有误。实际中，每编写约 10 行代码，马上编译运行到信编写代最后一行处，通过查看这 10 行代码执行结果（变量的值和运行流程）来判断这 10 行代码是否编写正确。虽然这不能找出所有 BUG，但是能快速找出大部分的 BUG。

2.3.3.2.2 对比程序运行结果

2.3.3.2.1 节第二个措施提到查看变量的值和运行流程。G7231 程序输入数据是语音数据，程序运行过程中的变量值保存语音数据的处理结果，直接观察变量值时，不能判断这些变量值是否正确。

幸好我们有能够在 PC 机上运行的 G7231 C 代码程序。有这么一个事实，如果 DSP 上的程序和 PC 上的程序都是正确的，那么当两个程序运行到相同的位置时，所有变量值都是相同的。

在 PC 机上可以使用 VC 的 Debug 功能将程序运行到指定的代码处；在 DSP 调试中

可以用 CCS 的 Debug 功能运行到相同的代码处，然后比较两个 Debug 的变量窗口中显示的变量值是否相同。如果变量值相同可继续用单步运行程序，直到单步运行了某条语句时，两者显示的变量值不同，则可以确定刚刚运行的语句编写错误。

用这个方法可以将 BUG 定位到一条语句。这个也是在 G7231 汇编调试过程采用的主要方法。

2.3.3.2.3 编码 cnt 帧以后再开始比较结果

直接采用措施二描述的方法，不能得到很好的效果。因为 G.723.1 第一帧开始 60 个点数据一定为 0。让程序第一次运行到新编代码最后一行，实际正第一帧编码，所以各个变量值几乎都是 0。对 0 做乘除运算结果还为 0，这样即使程序中乘数（或除数）不对结果也是 0。这就不能达到寻找 BUG 的目的。

为此我们采用编码 cnt 帧以后再比较结果的方法(cnt 的值一般取 10 即可)。由于_Find_Best 函数还只编写一部分，无法先编码 cnt 帧。为此前 cnt 帧编码仍使用修改前 C 的 Find_Best 函数；将汇编编写的_Find_Best 函数暂时命名为_Find_Best2，只对第 cnt 帧编码使用_Find_Best2。

实现的方法是，在 DSP 程序中改写：

```
Find_Best( &Best, Dpnt, ImpResp, Srate, (Word16) SubFrLen ) ;
```

为

```
if(cnt==10)           //cnt 为全局变量初始化为 0
//在 debug 方式下运行到下面语句
Find_Best2( &Best, Dpnt, ImpResp, Srate, (Word16) SubFrLen ) ;
else
Find_Best( &Best, Dpnt, ImpResp, Srate, (Word16) SubFrLen ) ;
cnt++;
```

在 VC 程序中改写

```
Find_Best( &Best, Dpnt, ImpResp, Srate, (Word16) SubFrLen ) ;
```

为

```
if(cnt==10)           //cnt 为全局变量初始化为 0
   cnt=10;            //在 debug 方式下运行到此语句
cnt++;
Find_Best( &Best, Dpnt, ImpResp, Srate, (Word16) SubFrLen ) ;
```

在 CCS 中运行到第三行代码，在 VC 中运行到第二行代码，此时，两程序都已经编码了 cnt 帧，之后再运行到新编最后一行代码处，这时各个变量值一般将不再为 0。

2.3.3.2.4 使用 cmpcode_detail 和 fcmp 工具定位出错帧

有时编解码结果中只有少数几帧出错。这时我们首先要找出是哪一帧出错。对于编码结果，使用我们在 cmpcode 程序基础上编写了 cmpcode_detail 程序，它能够发现哪些帧编码错误。对于解码结果，使用我们编写的 fcmp 工具，能够找到第一个解码错误数据对应的帧号。

当找到出错帧以后，设置 cnt 的值为出错帧号，使用“编码 cnt 帧以后再开始比较结果”的方法，可调试第 cnt 帧编码中的问题。这样可以方便地从编码出错帧开始调试，用“对比程序运行结果”的方法找到此帧编码出错的原因。

2.4 代码优化结果

我们从平均速度、代码量、所需数据空间来衡量优化的结果，见表 2－14、表 2－15、表 2－16、表 2－17、表 2－18。

平均速度：打开高通滤波和后滤波选项，使用包含 15 帧的 my.wav 测试，内容为“新年快乐”，使用秒表测试编解码时间：

表 2－14 G7231 优化结果（速度）

	编解码 3000 帧所需时间(s)	MIPS
6.3K 编码	18.32	20.36
5.3K 编码	15.68	17.42
5.3K+VAD 编码	6.24	6.93
6.3K+VAD 编码	6.83	7.59
6.3K 解码	1.733	1.93

表 2－15 G7231 优化结果（代码量）

模块名	大小（word，16 进制表示）
COD_CNGa.obj	00000216
CODERa.obj	000001e2
DEC_CNGa.obj	0000006c
DECODa.obj	00000197
EXC_LBCa.obj	00000de0
LBCCODEC.obj	000000ba
Lpca.obj	0000032a
LSPa.obj	00000359
TAMEa.obj	000000e9
Util_cnga.obj	00000256
UTIL_LBCa.obj	0000038b
VADa.obj	000000 fa
共计	22DC （约 8.71K）

表 2－16 G7231 优化结果（G.723.1 中的表）

TAB_LBC.obj	00002324
TAB_LBCa.obj	00000200
共计	2524（约 9.29K）

表 2－17 G7231 优化结果（实现一路编码时所需数据区大小）

EXC_LBCa.obj	000005d8
COD_CNGa.obj	00000020
DECODa.obj	00000222
LSPa.obj	00000050
Lpca.obj	0000034b
VAD.obj	00000016
DECOD.obj	000000bf

CODER.obj	0000026c
COD_CNG.obj	00000059
DEC_CNG.obj	0000000e
LBCCODEC.obj	00000009
共计	F66（3.85K）

表 2－18 G7231 优化结果（程序和数据总共所需空间）

总空所需空间	5766（21.85K）

性能对比：这里和 Spirit（http://www.spiritcorp.com/g723.html）公司的产品做一下比较，见表 2－19。由于测量和计算的方法不同，这里只是做个参考，以确定哪些部分还有优化的余地。

表 2－19 Spirit 公司 54xDSP 上的 G.723.1 算法性能

Algorithm		Peak MIPS	Program Memory (KWords)	Const Memory (KWords)	Dynamic Memory (KWords)	Scratch Memory (KWords)
Encoder	6.3 kbps	19.2	11.5	9.5	0.95	1.28
	5.3 kbps	19.9				
Decoder	6.3 kbps	2.3				
	5.3 kbps	2.3				
Encoder + Decoder	6.3 kbps	21.5				
	5.3 kbps	22.2				

图 2－4 是编码优化的效果图：

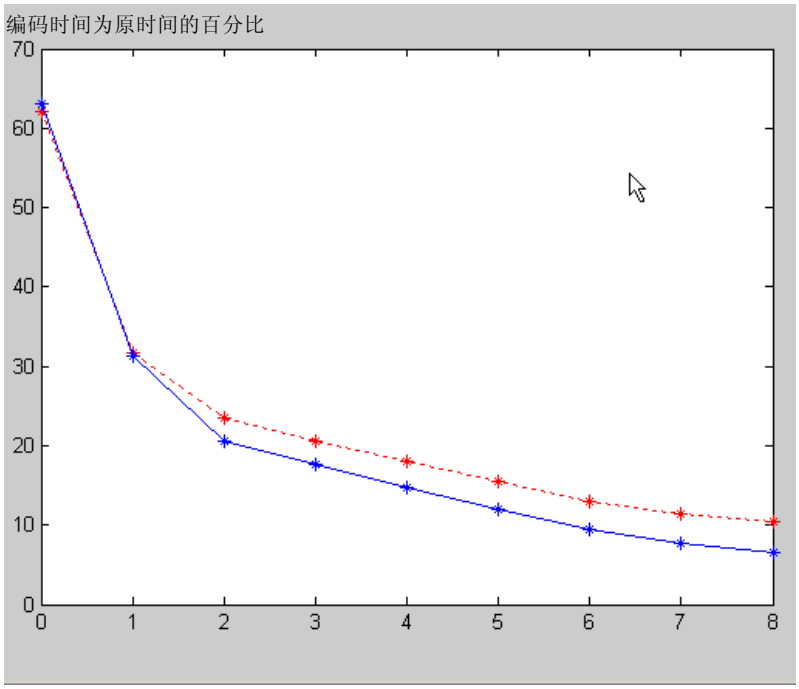

图2－4. 函数优化效果图（红线为实际情况，蓝线为预期值）

图中蓝线为预期值，它是根据 VC 的 profile 的结果计算的；红线为实际值，它是在各个函数优化后实际测量的。从该图可以说明实际优化的值基本符合预期值。可见用关键代码入手的方法是有效的。

第三章 G.723.1 算法中 C54xDSP 汇编程序优化技术

3.1 DSP 中的定点数表示法

DSP 中，有符号表示法使用 2 的补码表示法。对于整数，小数点放在最低位之后。对于小数，在 G7231 程序中遇到的小数都是在(-1,1)范围内，所以这里指的小数属于(-1,1)，小数一般有两种表示法 Q15 表示法和 Q31 表示法。Q15 表示法适用于 16bit 表示的小数，小数点在第 16bit 和第 15bit 之间。Q31 表示法适用于 32bit 表示的小数，小数点在第 32bit 和第 31bit 之间。例如 Q15 表示的最大正小数（近似为 1）为 0x7FFF，最小负小数（近似为－1）为 0x8000。Q32 表示为 0x7FFF FFFF 和 0x8000 0000。

小数乘法和整数乘法不同。两小数用整数乘法相乘以后需要再左移 1 位才得正确结果，如果结果用 Q31 表示，取结果的高 32bit；如果用 Q15 表示，取结果的高 16bit。例如 0x7FFF × 0x7FFF = 3FFF0001<<1= 7FFE0002，可以看到两个近似为 1 的数相乘也应该得近似为 1 的小数。

小数除法和整数除法不同，这将在后面介绍。

3.2 54x 汇编语言的相关知识

54x 汇编语言使用前首先需要了解：寄存器、指令、运算标志位、寻址方式。

3.2.1 运算标志位及在 G.723.1 中的设置

运算标志位应该是 DSP 汇编中很具特色的一部分，体现了 DSP 运算的特殊性。运算标志位在 ST1 和 PMST 寄存器中设置。参见图 3－1、图 3－2：

15	14	13	12	11	10	9	8	7	6	5	4–0
BRAF	CPL	XF	HM	INTM	0	OVM	SXM	C16	FRCT	CMPT	ASM

图 3－1 ST1 寄存器结构图

15–7	6	5	4	3	2	1	0
IPTR	MP/$\overline{MC}$	OVLY	AVIS	DROM	CLKOFF†	SMUL†	SST†

图 3－2 PMST 寄存器结构图

标志位用法说明如下：

1、OVM

溢出模式位。54xDSP 的硬件溢出控制功能主要通过 OVM 实现。OVM 的意义是，OVM＝1 时，当一条指令运行完了以后累加器中的值不能保持在溢出状态，如果处于溢出状态，则自动设置在最大或者最小值。即：对于有符号数（即 SXM＝1 时），如果累加器的值大于 00 7FFF FFFF 则设置为 00 7FFF FFFF；小于 FF 8000 0000 则设置为 FF 8000 0000。对于无符号数（SXM＝0 时），控制数值在 0 到 00 FFFF FFFF 之间。

一般溢出控制不会有误。因为累加器留出了 8 位的空余 bit（累加器是 40bit 的）进

行符号控制，对加减乘都能进行有效的控制。但是要注意移位操作时的溢出控制失效问题。见下例：

```
LD     #8000H,16, A
SFTA   A          15
```

第一条语句以后 A 累加器的内容是 FF 8000 0000。左移 15 位则变为 00 0000 0000，由于 OVM=1，所以需要对 A 的内容饱和处理，这时发现 A 的值为 0，饱和处理以后还是 0，出现错误。所以得出结论：为了防止溢出控制失效，左移位数不能大于 8 位。

在 G.723.1 中默认设置 OVM＝1。

2、FRCT

小数模式位。FRCT＝1 时表示进行小数乘法，DSP 在任何乘操作之后自动将结果左移 1 位，以实现定点小数乘法，包括乘操作的指令有 MPY、MAC、MAS 等。FRCT＝0 时，进行整数乘法。

在 G.723.1 中乘法以小数乘法占大部分，所以设置 FRCT＝1。

3、SXM

符号扩展位。SXM＝1 时，对应于有符号数处理，数据加载到 ALU（算术逻辑单元）之前先进行符号扩展，例如 0xFFFF 加载到累加器后变为 0xFF FFFF FFFF；累加器加 16bit 数 0xFFFF 相当于加 0xFF FFFF FFFF。如果 SXM＝1，累加器加 16bit 数 0xFFFF 相当于加 0x00 0000 FFFF。同时 SXM 影响 OVM 的溢出控制。注意：乘法操作都认为操作数为有符号数，不受 SXM 标志位影响。

G.723.1 中默认数据为有符号数，所以设置 SXM＝1。

4、C16

双 16 位/双精度算术模式位。C16＝0 时为双精度模式，这时双字算术指令 DADD、DSUB 等进行 32bit 加减运算。C16＝1 时，DADD、DSUB 等指令进行双 16 位运算。

G.723.1 中经常用到 32bit 加减运算，所以设置 C16＝0。

5、SMUL

乘法饱和位。SMUL 针对乘累加和乘累减操作有效，比如 MAC 和 MAS[11]指令。实际测得对 SQURA 和 SQURS 指令也有效。SMUL＝1，使得小数乘累加（乘累减）指令在加(减)之前先对乘的结果做一次溢出控制。只有在 FRCT＝1 且 OVM＝1 的情况下 SMUL＝1 才有意义。例如：

```
ST     #8000H,    tmp
LD     #(-1), A
MAC    tmp,#8000H,    A
```

SMUL＝0，FRCT＝1，OVM＝1 时，8000*8000 等于 00 8000 0000，注意不能认为时 FF 8000 0000，因为会进行符号扩展。A 中原来的值为－1，所以结果为 00 7FFF FFF。

SMUL＝1，FRCT＝1，OVM＝1 时，8000*8000 等于 00 8000 0000，然后进行溢出控制得 00 7FFF FFFF。A 中原来的值为－1，所以结果为 00 7FFF FFE。

这就是说设置 SMUL＝0，反而比 SMUL＝1，结果更精确。但是在 G.723.1 代码中用 C 语言描述的 L_mac()函数却是和 SMUL＝1 情况一样的，所以这里必须设置 SMUL＝1。

6、SST

存储饱和位。对 STH,STL, STLM, DST, ST||ADD, ST||LD, ST||MACR[R], ST||MAS[R], ST||MPY, 和 ST||SUB.指令有效。但是测试发现对条件存储指令 SACCD 指令也有效。SST 的使用和 SMUL 有类似之处，SST＝0 时的存储指令的操作步骤是：先按照指令要求移位，然后存储；SST＝1 时 DSP 会在移位和存储之间加入饱和处理的步骤。注意存储指

令并不改变累加器的内容。如下例：

```
LD      #7FFFH,         A
STH     A,1,            tmp
```

SST＝1 时，存储步骤是，A 累加器的值左移一位得 00 FFFE 0000，饱和处理为 00 7FFF FFFF，然后取高 16bit 存入到 tmp，结果为 7FFF。在 SST＝0 时，结果为 FFFE。

在 G.723.1 中有类似以下的语句

```
tmp = shl(tmp,1)
```

用汇编实现时为

```
LD      tmp,16,     A
STH     A,1,        tmp
```

由于 shl()函数是带饱和处理的，为了两者等价必须采用存储饱和位。设置 SST＝1。

3.2.2 54xDSP 数据寻址方式的使用

DSP 数据寻址方式有：直接寻址、*(1k)寻址、单操作数间接寻址、双操作数间接寻址、MMR 寻址、堆栈寻址、立即寻址和累加器寻址。

其中堆栈寻址就是 PUSH 和 POP 操作，累加器寻址只在 READA 和 WRITEA 等少数指令中用到，立即寻址就是将数据写在指令中的寻址方式。MMR 寻址，只在 LDM、STM、STLM 等指令中用于访问低 128 字节 RAM 时使用。所以经常用到，而且需要灵活使用的是：直接寻址、*(1k)寻址、单操作数间接寻址、双操作数间接寻址。

直接寻址是最快捷的方式，G7231 中使用基于 SP 的直接寻址，寻址的数据一般为函数局部变量。间接寻址通过辅助寄存器的值作为指针寻址，有多种指针增减变化。双操作数间接寻址只能使用 AR2-AR5，指针增减有 4 种变化。如果某变量通过指针访问次数较少，则不用将指针放入 ARx，直接用*(1k)寻址。

3.2.3 流水线冲突及其解决方案

DSP 执行指令是流水线结构，如果下条指令需要用到上条指令的执行结果，而此时上条指令还没有计算出结果时，将产生流水线冲突。检查流水线冲突的重要方法是打开编译器的流水线冲突警告 Project/Compiler/Diagnostics/Warn on Pipeline，它能够检测出大部分的流水线冲突。当产生流水线冲突时，或者程序运行出错，或者 CPU 自动加入延迟，加入延迟后降低了程序速度。熟悉流水线冲突是编写汇编程序所不可缺少的知识，具体内容参考参考文献[12]。

3.2.3.1 解决流水线冲突的“将错就错”法

1、用“将错就错法”解决 SP 流水线冲突，看如下代码：

```
CALLD       _func1          ;SP == 1000H
LD          0H,     A       ;SP == 0FFFH,该指令是基于 SP 的直接寻址
```

由于 CALLD 指令执行后 SP 指令已经减一，第二条指令不再访问 1000H+0H 地址，而是 0FFFH+0H。解决办法：一个方法是将第二条指令放到 CALLD 之前，但这时没有利用 CALLD 的延迟时间，第二个方式是将错就错，将程序修改如下：

```
CALLD       _func1          ;SP == 1000H
LD          1H,     A       ;SP == 0FFFH,访问的地址是 0FFFH+1=1000H
```

PSHD、PSHM、POPD、POPM 指令的流水线冲突也可用第二种方法解决。

2、用“将错就错法”使用 XC 指令，看如下代码：

```
LD      tmp,    A
LD      #1,     A
```

```
NOP
XC     1,     AEQ
```

这里根据 tmp 的值是否为 0，条件执行 XC 指令后的语句。其中第二条指令离 XC 指令小于 2 个指令周期，所以它不影响 XC 的判断，所以可以在第二条指令处改变 A 的值。

3.2.3.2 “将错就错”法使用注意事项

当程序中使用中断时，将不能使用“将错就错法”。以 3.2.3.1 节的例 2 为例，如果在第二条指令和 XC 指令之间产生了中断，则第二条指令离 XC 指令的间隔超过 2 个指令周期，使得第二条指令对 XC 指令的判断有影响，从而使得将错就错法失效。

3.3 汇编程序的整体的考虑

3.3.4 伪指令在 G.723.1 中的使用

- .struct 伪指令[13]

G7231 程序中有很多结构体。.struct 可以用于定义结构体。使用.struct 的好处是，减少编写复杂度；使用.struct 定义结构体后可以实现结构体内双字变量的偶定位。

- .asg 伪指令

.asg 伪指令能够将一个数或者字符串，指定给一个另一个字符串，例如

```
.asg      0, i
```

在程序中出现的“i”会在编译时全部替换为 0。.asg 伪指令使得无意义的数值和寄存器名可以用有意义的字符串代替，这对程序的易读性和可维护性很有好处。G7231 程序中主要在以下几方面使用了.asg 伪指令：

一、 定义变量

1、函数内局部变量。例如

```
.asg   0, i
```

此后可以用

```
ST  #0,    i
```

方式访问变量。

2、汇编中增加的局部变量

这些变量在修改前的 C 代码中没有，在改为汇编时，为了提高程序效率而添加这些变量。为了防止混淆规定这类变量名前加前缀“m_”。

二、 命名辅助寄存器（以下简称 ARx）

1、给 ARx 一个有意义的名字

为了和变量名区别，名字前加“AR_”前缀，例如：

```
.asg   AR2,   AR_Imr
STM    #Imr,  AR_Imr
```

有时需要在 ARx 的分配上作调整，这时只要修改.asg 定义就可以了。

2、防止 ARx 的混用

在程序中，往往某些 ARx 保存着中间结果，所以暂时不能作其它使用，这就要求程序员记住所有当前不能使用的 ARx ，这增加程序编写难度。在我们的程序中，没有保存中间结果的 ARx，我们在循环体开始处或者函数开始处将其定义为：

```
.asg   AR4,   AR_tmp1
.asg   AR5,   AR_tmp2
.asg   AR7,   AR_tmp3
```

使用这些 ARx 前，再次定义为：

```
.asg   AR_tmp1,  AR_AcGn
```

这样程序员只用记住 AR_tmp1、AR_tmp2 等，而不用记具体是哪个 ARx。

3.3.5 G.723.1 中双字数据的偶定位

DSP 读取或存储双字数据（32bit 数据）的例子如下

```
DLD *(0100H), A
DLD *(0101H), A
```

高 16bit 一定从给定的地址 LmemAddr 读取（第一条指令为 100H，第二条指令为 101H），低 16bit 从 LmemAddr 和 1 异或后的地址读取（第一条指令为 101H，第二条指令为 100H）。这样为了确保：双字数据的高 16bit 放在 LmemAddr 地址，低 16bit 放在 LmemAddr+1 地址，就要求 LmemAddr 为偶数，这就是双字数据的偶定位问题。

为了将所有双字数据定位在偶地址我们的程序中采用了以下措施：

1、局部变量定位在偶地址。程序的局部变量存储在堆栈中用基于 SP 的直接寻址访问，例如若 Acc0 为双字数据，我们用.asg offset, Acc0 定义了 Acc0 相对于 SP 的偏移，然后就可以用 DLD Acc0,A 形式加载 Acc0，这时要求 SP+offset 为一个偶数。这通过保证 SP 为偶数和 offset 为偶数来实现。Offset 是我们自己定义的，容易设置为偶数，为了保证 SP 为偶数我们做以下的处理：①程序执行 CALL 进入子函数后 SP 为奇数。②在函数的开头用 FRAME PcPos 改变 SP 的值，并且保证 PcPos 的值为奇数。其中第二点是程序员容易实现的，那么如何保证第一点？为了保证 CALL 以后 SP 为奇数必须保证 CALL 之前 SP 为偶数，因为 CALL 指令使得 SP 减一。这就是可以递推下去，最后只要保证总程序的第一个函数（对于 C 语言为 c_int00 函数）中 SP 为偶数即可。实时上 C 编译器确实能够保证在 c_int00 函数中 SP 为偶数[14]。

2、双字数组定位在偶地址。一般我们在程序中将数组定位在数据区，为了将双字数组定位在偶地址在定义双字数组之前使用.align 2 伪指令将标号(lable)定位在偶地址。

3、结构体双字成员的偶定位。由于我们使用.struct 伪指令定义结构体所以如果能够保证结构体变量定位在偶地址，其所有内部双字成员也定位在偶地址。为了将结构体变量定位在偶地址①如果结构体变量定位在堆栈中则采用措施一的方法。②如果定位在数据区，则要使用.align 2 伪指令。

4、双字函数参数的偶定位。函数参数通过堆栈传递所以和措施一类似，实例如下：void Lsp_Inq(Word16 *Lsp, Word16 *PrevLsp, Word32 LspId, Word16 Crc)。其中 LspId 为双字参数。如果将 PrevLsp 放到 SP+0 中，将 LspId 放到 SP+1 和 SP+2 中，则 LspId 没有满足偶定位要求。所以一般做以下的规定（这和 CCS 的 C 编译器的规定是一样的）：如果参数为 32bit 数，则定位到下一个（向高地址方向）偶地址。所以 LspId 存储到 SP+2 和 SP+3。

3.3.6 函数参数传递及其堆栈设置

3.3.6.1 混合汇编及其参数传递规则

汇编和 C 混合时，从 C 函数进入汇编函数时需要保存 AR1、AR6、AR7，因为 C 函数中可能正在使用这几个辅助寄存器。

函数参数从右到左依次压入堆栈（对于双字参数需要偶定位），第一个参数通过 A

累加器传递。当函数参数中出现结构体变量时，参数传递的是结构体变量的地址。

函数返回值放在 A 中传递回来。如果返回值是一个结构体，此时稍有不同，例如

```
PWDEF Comp_Pw( Word16 *Dpnt, Word16 Start, Word16 Olp )
```

这个函数返回一个 PWDEF 结构体变量。这时上面函数原型定义被编译器改为：

```
void Comp_Pw(PWDEF *Pw, Word16 *Dpnt, Word16 Start, Word16 Olp )
```

也就是说，主调函数将一个结构体变量的指针作为第一个参数传递给被调函数，在被调函数中通过 Pw 指针修改结构体变量的内容，这和返回一个结构体变量能实现同样的功能。

3.3.6.2 预留参数空间的参数传递方式

函数调用时，在主调函数中一般需要以下步骤：

1、在函数调用以前将参数按照顺序压入堆栈。

2、执行调用指令（例如 CALL）调用函数。

3、函数返回以后平衡堆栈。

以下面函数调用为例

```
func1( 0, 1,tmp*2 ) ;
```

用以上方法实现为：

```
ST      #0, tmpval
PSHD    tmpval
LD      tmp,1, A
NOP
PSHM    AL
LD      #0,     A
CALL    func1
FRAME   2
NOP
```

由于在 54xDSP 中只能将 Smem（Smem 指用单操作数寻址 RAM）或 MMR（指存储器映射寄存器）压入堆栈。所以当立即数作为函数参数的时候，需要首先将其放到 Smem 中，然后压入堆栈（需 3 个 clock）；也可以采用另一种方法，例如：

```
LD      #0,     A
NOP
PSHM    AL
```

将立即数加载到 A，然后将 AL 压入堆栈，但是用 MMR 方式访问 AL 时，有一个流水线冲突，需要一个 NOP，所以还是需要 3 个 clock。至于第 3 个参数 tmp*2，虽然 tmp 在 Smem 中，但是需要经过算术运算，不能直接用 PSHD tmp 指令。此外，CALL 指令之后需要用 FRAME 指令平衡堆栈。一般情况下 FRAME 后还要加一个 NOP，因为 FRAME 指令对 SP 有一个流水线冲突

所以在 54xDSP 中采用一般的参数传递方式所需指令数较多。这里借鉴了 CCS 编译中采用的函数调用方法：预留参数空间的方法。上例在使用该方法实现如下：

```
func_caller:
        FRAME   -MaxParmNum                     ;预留空间
        ……
        ST      #0,     0H                      ;基于 SP 的直接寻址
        LD      tmp,1, A
        STL     A,      1H
```

```
LD      #0,     A
CALL    func1
```

假设这段代码在 func_caller 函数中，首先观察一下 func_caller 函数内部子函数中最多函数参数个数是多少，如果是 func1(0, 1,tmp*2)的参数最多则最多参数是 3 个，那么 MaxParmNum=3-1，MaxParmNum 表示了 func_caller 函数给其子函数预留的参数空间。在 func_caller 函数的开始处用 FRAME –MaxParmNum 指令预留空间。程序中第 4 行用直接寻址方式直接将参数 1 放入堆栈中，此时不需要使用压栈操作。CALL 后面不再有堆栈平衡调整。这种参数调用方式一般能够减少指令数。

3.3.6.3 函数堆栈结构设计

以设计_Find_Best 函数的堆栈为例。当执行了 CALL _Find_Best 指令以后，则进入 _Find_Best 函数，进入子函数后的堆栈情况如图 3－3 左半部分所示。在_Find_Best 函数的开始处我们使用 FRAME –PcPos 指令，将 SP 减 PcPos。新增的可用堆栈如图 3－3 右半部分所示：

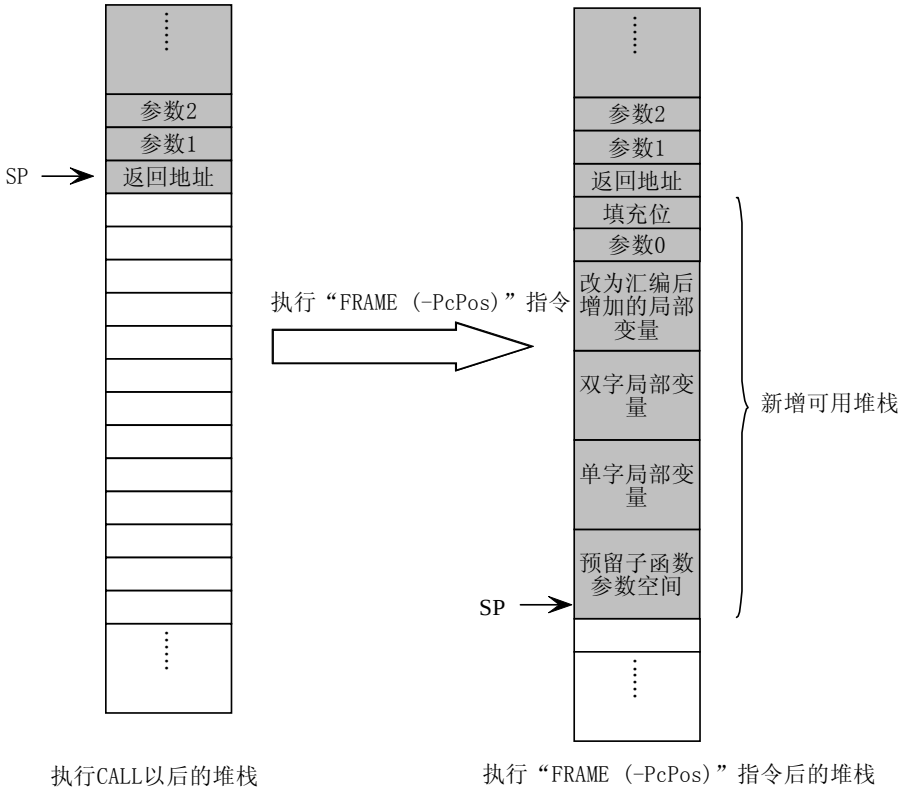

图 3－3 函数堆栈结构设计示意图

实际程序中我们用 asg 伪指令将数赋给字符串，这样可以方便地用基于 SP 的直接寻址访问堆栈中的变量。

C 语言的 Find_Best 函数部分内容如下：

```
void  Find_Best( BESTDEF *Best, Word16 *Tv, Word16 *ImpResp, Word16 Np,
Word16 Olp )
{
```

```
int   i,j  ;
BESTDEF  Temp  ;
Word32   Acc0,Acc1 ;
Word32   ErrBlk[SubFrLen] ;
Word16   Imr[SubFrLen]  ;
```

根据 Find_Best 的参数和局部变量，我们设计的_Find_Best 函数堆栈用汇编程序表示如下：

```
;数据区分配情况
          .data
Imr:      .space 16*SubFrLen
;下面是 32bit 的数组，将其定位在偶地址
          .align 2
ErrBlk:       .space 16*(SubFrLen*2)
;堆栈分配情况
   ;预留子函数参数
   .asg   0,  ParmLeft1
   .asg   1,  ParmLeft2
;单字局部变量
   .asg   2,  i
   .asg   3,  j
;双字局部变量必须定位在偶地址
   .asg   4,  Acc0          ;32bit
   .asg   6,  Acc1          ;32bit
   .asg   8,  Temp          ;含有 32bit 数的结构体,Temp 大小是 17
;汇编中增加的局部变量
   .asg   26, m_ErrBlkPlusk
   .asg   28, m_RND_val
;该函数第一个参数
   .asg   29, Best
;为了保证下面的 PcPos 为奇数地址，填充一个字节
   .asg   30, padding
;返回地址所在位置，要求 PcPos 为奇数地址
   .asg   49  ,PcPos
;该函数的其它参数，从左到右依次为:
   .asg   (PcPos+1), Tv
   .asg   (PcPos+2), ImpResp
   .asg   (PcPos+3), Np
   .asg   (PcPos+4), Olp
;下面是函数代码
   .text
_Find_Best:
       FRAME    -(PcPos)      ;堆栈指针 SP 减 PcPos
       NOP
       STL    A, Best         ;保存第一个参数
```

堆栈中各部分的作用如表 3－1：

表 3－1 函数堆栈结构中各部分的功能

堆栈中的各个部分	作用
预留子函数参数空间	子函数调用时，参数通过这个空间传递。
单字局部变量	将局部变量设置在堆栈中有一个好处：这样可以用基于 SP 的直接寻址访问变量，直接寻址是最简单快速的寻址方式。
双字局部变量	双字局部变量需要定位在偶地址。这包括含有双字成员变量的结构体。
改为汇编后增加的局部变量	为了提高汇编程序的效率，有时需要增加一些变量，这些变量在 C 程序中是没有的。
参数 0	这是函数的第一个参数，它通过 A 累加器传递到给函数，但是 A 累加器在函数中需要使用，所以将用 STL A, Best（Best 是第一个参数的名称）指令将参数 0 存储到堆栈中。这样在函数内部访问参数 0 和访问函数其它参数方法相同。
填充位	由于偶定位问题，PcPos 必须是一个奇数，如果是偶数需要填充一位。
返回地址	函数返回地址。返回地址是在主调函数执行 CALL 指令时写入堆栈的。
参数 1、参数 2……	这是主调函数中在 CALL 指令之前写入堆栈的函数参数。

3.3.7 G.723.1 数据区设置

前面的例子中的局部变量

```
Word32  ErrBlk[SubFrLen] ;
Word16  Imr[SubFrLen]  ;
```

没有设置在堆栈中，我们将局部变量数组设置在数据区(参考上例代码)。这是因为：

1、 最主要的原因：数字信号处理类程序需要处理大量数据，这些数据通常放在数组中。处理数据时常使用双操作数指令，但是 DSP 的堆栈区有时设置在 SRAM 中，所以将数组定位在堆栈中，将可能造成流水线冲突，这将大大影响程序速度。为此将数组定位在.data 段，在 CMD 文件中，我们将.data 段定位在 DRAM 中。

2、 基于 SP 的直接寻址的最大空间是 SP～SP+127。而数组占用空间通常较大，无法容纳在寻址范围内。

3、 容易得到数组头指针。例如 Imr 定位在数据区中时，将 Imr 数组头指针放入 AR1 寄存器的操作是：

```
STM    #Imr,  AR1
```

如果用.asg 定义在堆栈中，获得头指针的方法是

```
MVMM       SP,    AR1
MAR        *+AR1(Imr)
```

数组定位在数据区的缺点是始终占用一部分 RAM 空间，将使整个程序的数据空间增大。

3.4 G.723.1 程序主要优化技术

在 54xDSP 中经常需要调整语句的顺序，有时需要将后面的指令提到前面去。调整语句顺序的主要目的是：①用有用指令替换程序中为了防止流水线冲突引入的 NOP 指令②将特定的指令放到一起，以便使用并行指令。调整指令顺序是优化技术中的重要部分，这里主要有以下方法：

1、部分循环展开法（参考 3.4.1 节“部分循环展开法” 中第 2 处 NOP 的消除方法）
2、先预减一个值的方法（参考 3.4.1 节“部分循环展开法”中第 1 处 NOP 的消除方法）
3、将一个循环视为两个并行执行的循环（参考 3.4.2 节“并行指令使用技术”）

3.4.1 部分循环展开技术

我们知道循环体是消耗指令主要地方，减少循环内部指令尤为重要。部分循环展开法，是在 G7231 程序中经常使用的优化手段，能够减少循环体内 NOP 指令、有利于利用并行指令等好处，当第一次或最后一次循环需要特殊处理时也可以使用该方法。

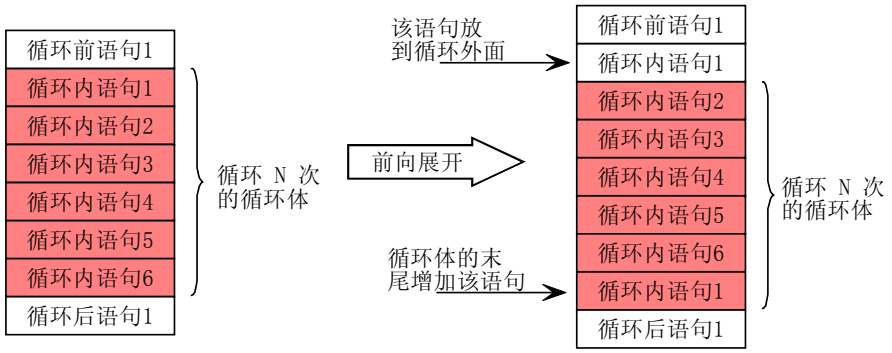

图3－4 前向部分循环展开（前向展开一条语句）示意图

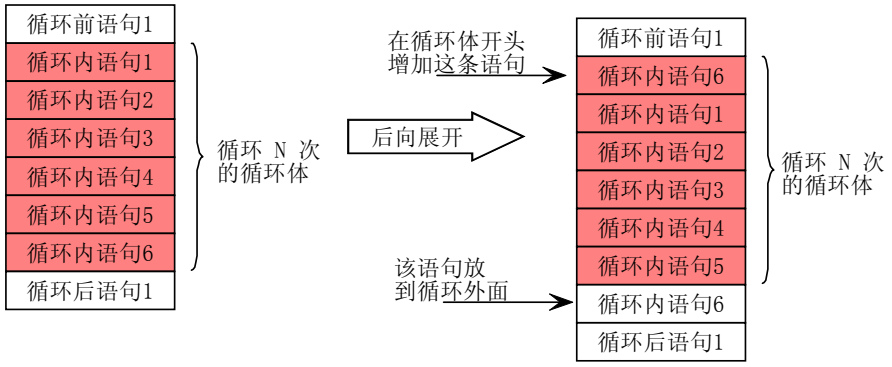

图3－5 后向部分循环展开（后向展开一条语句）示意图

部分循环展开法分为前向部分循环展开和后向部分循环展开，分别如图 3－4 和图 3－5 所示。以前向展开为例，其主要思想是将循环体内前几条语句放到循环体外，然后再在循环体的末尾增加这几条语句，如图 3－4 所示。这样程序的执行结果只有一点不同：在最后一次循环中，需要多执行一次“在循环体的末尾增加的几条语句”。看上去它需多执行了一些语句，但是它使得循环体内部语句顺序能够调换，为压缩循环内部指

令创造了条件。

3.4.1.1 减少循环体内 NOP 指令

参考如下代码：

```
          STM    #(D4_p_sign),     AR_p_sign_i0
D4_modify_rr:
          STM    #((SubFrLen/2)/(STEP/2)-1),     BRC ;循环体内语句 1
          RPTBD      D4_modify_rr_inner
          MVMM       AR_p_sign_1,      AR_p_sign_i1
          NOP                                    ;第 1 处 NOP
             MPY     *AR_p_sign_i0,    *AR_p_sign_i1+,       A
             ……
          MAR    *AR_p_sign_i0+0                    ;循环体内语句 M-3
          BANED      D4_modify_rr,     *AR_cnt-
          NOP                                    ;第 2 处 NOP
          NOP                                    ;循环体内语句 M
```

第 2 处 NOP 的消除。该处 NOP 语句如果能用有用语句替换就好了，但是如果不变换顺序，没有语句能放到此处。这时使用前向部分循环展开，将“循环体内语句 1”放到循环体外，再用该语句替换这里的 NOP 语句。

第 1 处 NOP 的消除。将“循环体内语句 M-3”放到第一处 NOP 处。如果不加额外处理，直接替换将改变程序逻辑，“循环体内语句 M-3”用于将 AR_p_sign_i0 指针增加 AR0 的值，将该语句提前以后为了使得循环体内部 AR_p_sign_i0 的值保持不变，需要将 AR_p_sign_i0 的初值预先减去 AR0 的值。改后程序如下：

```
          MAR    AR_p_sign_i0-0                     ;预先减去 AR0 的值
          STM    #((SubFrLen/2)/(STEP/2)-1),     BRC ;前向部分循环展开
          D4_modify_rr:
          RPTBD  D4_modify_rr_inner
             MVMM   AR_p_sign_1,  AR_p_sign_i1
             MAR    *AR_p_sign_i0+0                    ;替换了一个 NOP
             MPY    *AR_p_sign_i0,    *AR_p_sign_i1+,       A
             ……
          BANZD  D4_modify_rr,     *AR_cnt-
          STM    #((SubFrLen/2)/(STEP/2)-1),         BRC;替换了这两个 NOP
```

共节省周期数：(2+1)×循环次数-3

3.4.1.2 有利于利用并行指令

有时需要调整循环内指令的顺序才能利用并行指令。例子参考 3.4.2 节“并行指令使用技术”。

3.4.1.3 第一或最后一次循环需要特殊处理时的情形

参考以下 C 代码：

```
          for ( j = 0 ; j < LpcOrder ; j ++ )
              Acc0 = L_msu( Acc0, Dpnt[i-j-1], VadStat.NLpc[j] ) ;
          Tm0 = round ( Acc0 ) ;
```

一般做法：

```
RPT    #(LpcOrder-1)
   MAS    *AR_Dpnt_i_j_1-,  *AR_VadStat_Nlpc_j+, A
RND A
```

由于最后一次循环的 MAS 和 RND 指令可以用 MASR 指令代替，为了合并这两条语句，将最后一次循环单独处理。

```
RPT    #(LpcOrder-2)                          ;循环次数少一次
MAS    *AR_Dpnt_i_j_1-,  *AR_VadStat_Nlpc_j+, A
MASR   *AR_Dpnt_i_j_1-,  *AR_VadStat_Nlpc_j+      ;合并以后的语句
```

这种情况的部分循环展开还经常和 3.4.3 节的“合理利用指针增减”配合起来使用。

3.4.2 并行指令使用技术

使用并行指令可用一条指令可以代替两条指令的功能。但是只有在特定的某两条语句，凑在一起时，才能合并这两条语句，所以在实际编程中能合并的情况出现较少。这里我们介绍几种方法能够创造一些条件使得特定的语句能够凑在一起，从而可被并行指令替换。

并行指令使用有 3 种情况：①LD 和 ST 的合并② ST 和算术指令的合并③LD 和算术指令的合并。其中只有第 3 种情况要求指令中两个累加器必须不同。

3.4.2.1 将一个循环视为两个并行执行的循环

参考如下程序：

```
for ( j = 0 ; j < SubFrLen ; j ++ ) {
   Dpnt[j] = shl( Dpnt[j], (Word16) 1 ) ;
   Dpnt[j] = add( Dpnt[j], AcbkCont[j] ) ;
}
```

一般地，可以用

```
    MVDK      Dpnt,            AR_Dpnt_j
    STM       #(SubFrLen-1),   BRC
    RPTBD     Deocd_AddAdaptive
    STM       #Decod_AcbkCont, AR_AcbkCont_j
       LD        *AR_Dpnt_j,16,   A
       SFTA      A,               1
       ADD    *AR_AcbkCont_j+,16, A
Deocd_AddAdaptive:
       STH       A,               *AR_Dpnt_j+
```

来实现。从以上的代码中看不出能够使用并行指令的地方。拆分这个循环为两个并行执行的循环。姑且称之为 i 循环和 j 循环。

```
for ( j = 0 ; j < SubFrLen/2 ; j ++ ) {
   Dpnt[j] = shl( Dpnt[j], (Word16) 1 ) ;
   Dpnt[j] = add( Dpnt[j], AcbkCont[j] ) ;
}
for ( j = SubFrLen/2+1 ; j < SubFrLen ; j ++ ) {
   Dpnt[j] = shl( Dpnt[j], (Word16) 1 ) ;
   Dpnt[j] = add( Dpnt[j], AcbkCont[j] ) ;
}
```

“并行”是指这两个循环没有先后次序问题，所以可以被并行处理，它们是相互独立的。可以看到如果能够有并行指令同时运行这两个循环体，则指令周期数将减半。由于两个循环的循环次数相同，又可以合并这两个循环：如下：

```
for (i = 0, j = SubFrLen/2+1 ; j < SubFrLen ; j ++ ,i++) {
   Dpnt[i] = shl( Dpnt[i], (Word16) 1 ) ;
   Dpnt[i] = add( Dpnt[i], AcbkCont[i] ) ;

   Dpnt[j] = shl( Dpnt[j], (Word16) 1 ) ;
   Dpnt[j] = add( Dpnt[j], AcbkCont[j] ) ;
}
```

其汇编代码如下：

```
    MVDK      Dpnt,             AR_Dpnt_i
    STM       #Decod_AcbkCont,  AR_AcbkCont_i
    LD        #(SubFrLen/2),    A
    ADD       Dpnt,             A
    STLM      A,                AR_Dpnt_j
    STM       #(SubFrLen/2-1),     BRC
    RPTBD     Deocd_AddAdaptive
    STM       #(Decod_AcbkCont+SubFrLen/2),AR_AcbkCont_j
       LD        *AR_Dpnt_i,16,    A
       SFTA      A,                1
       ADD       *AR_AcbkCont_i+,16,  A
       STH       A,                *AR_Dpnt_i+

       LD        *AR_Dpnt_j,16,    B
       SFTA      B,                1
       ADD       *AR_AcbkCont_j+,16,  B
Deocd_AddAdaptive:
       STH       B,                *AR_Dpnt_j+
```

其中原 i 循环体的语句使用 AR_Dpnt_i、AR_AcbkCont_i、累加器 A；原 j 循环的语句使用 AR_Dpnt_j、AR_AcbkCont_j、累加器 B，这样 i 循环部分和 j 循环部分也是各不相关。这时你可以将 i 循环和 j 循环语句交织在一起，在保证 i 循环和 j 循环自身语句顺序不变的情况下，可以任意调整它们之间的顺序。当你发现其中的某两条语句可以组合为并行语句时，可将它们放在一起。由于 i、j 循环分别使用 A 和 B 累加器，这样使得需要使用不同累加器的并行指令也可以被使用，增大了合并的概率。该例子不需调整语序，可合并的语句为：

```
    STH    A,            *AR_Dpnt_j+
    LD     *AR_Dpnt_j,16,    B
```

合并为

```
    ST     A,  *AR_Dpnt_i+   ||  LD  *AR_Dpnt_j,   B
```

3.4.2.2 部分循环展开法

使用“视为两个并行执行的循环”的方法，能够任意调整 i、j 循环语句之间的先后次序，但是不能变化自身的语句顺序。部分循环展开法，使得能在一定程度上调整 i，j

循环自身的语句次序，从而能够配合“视为两个并行执行的循环”的方法进一步使用并行指令。例如上面的汇编代码中的

```
STH    B,            *AR_Dpnt_j+
```

如果能够放到

```
ADD *AR_AcbkCont_i+,16,  A
```

的后面，则可合并为

```
ST     B,  *AR_Dpnt_j+   ||  ADD *AR_AcbkCont_i+,A
```

但是这需要改变 j 循环自身的语序，采用后向部分循环展开，将语句：

```
ST     B,  *AR_Dpnt_j+
```

提到前面，可达到这个目的。展开以后的代码如下：

```
   MVDK   Dpnt,             AR_Dpnt_i
   STM    #Decod_AcbkCont,  AR_AcbkCont_i
   LD     #(SubFrLen/2-1),  A             ;AR_Dpnt_j 指针预减一
   ADD    Dpnt,          A
   STLM   A,             AR_Dpnt_j
   STM    #(SubFrLen/2-1),  BRC
   LD     *AR_Dpnt_j,16,       B          ;B 预加载*AR_Dpnt_j 的原内容
   RPTBD  Deocd_AddAdaptive
   STM    #(Decod_AcbkCont+SubFrLen/2),AR_AcbkCont_j
      LD     *AR_Dpnt_i,16,    A
      SFTA   A,             1
      ST     B,  *AR_Dpnt_j+   ||  ADD *AR_AcbkCont_i+,A        ;并行指令 1
      ST     A,  *AR_Dpnt_i+   ||  LD  *AR_Dpnt_j,   B          ;并行指令 2
      SFTA   B,             1
Deocd_AddAdaptive:
      ADD    *AR_AcbkCont_j+,16,  B
   STH    B,      *AR_Dpnt_j+              ;后向部分循环展开
```

说明：如果没有预处理，第一次运行到并行指令 1 时，AR_Dpnt_j 指针增一后指向 Dpnt[SubFrLen/2+1]，而实际要求指向 Dpnt[SubFrLen/2]，为此先将 AR_Dpnt_j 指针预减 1。这样一来 B 的值会存储到 Dpnt[SubFrLen/2-1]，这是不希望的操作，为此首先 B 预加载 AR_Dpnt_j 原来的内容，执行存储时是存储原内容，所以不受影响。（注：上面程序由于 ST||ADD 指令本身有问题，B 的内容会加载到 A 中去，ST||ADD 指令运行不正常）。

3.4.2.3 使用并行指令加载 T

有时需要先将乘操作的一个操作数加载到 T。将数据加载 T 可以和存储操作合并为并行指令，例如：

```
    ST     B,  *AR_IirCoef+  ||  LD  *AR_Lpc+,  T
```

这是唯一一条不需要两个累加器作为操作数的指令，任何存储指令都可以被扩展为这个并行指令，使用限制比较少，所以在 G7231 程序中，多处使用该并行指令。

3.4.3 合理利用指针增减

G7231 中常用指针增减有*ARx+,*ARx-,*ARx+0,*ARx-0,*ARx(1k),*+ARx(1k)，*+ARx 及它们的环形增减(带%的增减)。

*ARx+0 和*ARx-0，比用*+ARx(1k)要少一个指令周期，所以尽量用前者代替后者。*ARx(1k)主要用于访问结构体的元素。*+ARx 能够做到先增加 ARx 的值然后将数据存储到 ARx 指向的单元，有时经常需要这种操作。

54xDSP 汇编中专门用于改变 ARx 的指令为 MAR 指令，但是任何包含有通过 ARx 间接寻址的指令都能够附带地改变 ARx 的值，如果能够合理利用这一特性能够减少指令数和节省 ARx 的使用。

3.4.3.1 使用 DLD 增减 2

前面提到用*ARx+0 和*ARx-0 代替*+ARx(1k)的使用，但是*ARx+0 和*ARx-0 需要设置 AR0 的值，在循环体中有时需要大小不一样的增量，例如某些 ARx 需要增 2，某些需要增 4。这时就需要频繁设置 AR0，降低了效率，这时不得已要使用*+ARx(1k)。其实当增减量为 2 时，可以巧妙利用 DLD 指令。

DLD 指令是双字加载指令，双字操作指令的*ARx+和*ARx-是增减 2，而不是增减 1。所以

```
MAR *+ARx(2)
```

可替换为单周期指令

```
DLD *ARx+, A
```

3.4.3.2 合理利用指针增减操作

前面提到在包含有 ARx 间接寻址的指令中增减 ARx，可以避免使用 MAR 指令专门用于指针增减，不仅如此，如果合理利用还可以节省 ARx 的使用，避免在在循环内部再次初始化 ARx 等好处，可以较大地提高程序效率。

合理利用指针增减的关键点是，在当前指令使用 ARx 的同时修改 ARx 的值，这样为下面的指令使用 ARx 做好准备。

3.4.3.2.1 节省 ARx 的使用

设一个循环内部的部分代码如下：

```
MACSU  *AR_Acc1-, *AR_Pk,    A   ;低位乘法，之后 AR_Acc1 指向
SFTA   A,     -15
MPY    *AR_Pk,    *AR_Acc1+, B   ;高位乘法，之后 AR_Acc1 指向低字节部分
```

Acc1 是一个双字数据，先用 AR_Acc1 指向其低 16bit，然后循环中使用*AR_Acc1-和*AR_Acc1+在低 16bit 和高 16bit 之间切换。这避免了分别用 1 个 ARx 指令指令高 16bit 和低 16bit 的情况。

3.4.3.2.2 避免赋初值

设一个循环内部的部分代码如下：

```
for ( j = 0 ; j < LpcOrder ; j ++ )
   Acc0 = L_mac( Acc0, QntLpc[j], FirDl[j] ) ;
……
for ( j = 0 ; j < LpcOrder ; j ++ )
   Acc0 = L_msu( Acc0, PerLpc[j], FirDl[j] ) ;
……
FirDl[0] = round( Acc1 ) ;
```

汇编实现如下，注意其中 AR_FirDl 指针的使用。

```
RPTZ   A, #(LpcOrder-1-1)
```

```
    MAC    *AR_QntLpc+,  *AR_FirDl+,   A
MAC    *AR_QntLpc,   *AR_FirDl, A          ;后向部分循环展开

MAS    *AR_FirDl-,   *AR_PerLpc_Fir_first,   A
RPTZ   B,  #(LpcOrder-1-1)
    MACD   *AR_FirDl-,   #(PerLpc_AntiOrder + LpcOrder +1),    B
STH    A,     *+AR_FirDl
```

第一个内循环采用后向部分循环展开，最后一次循环提出循环外并采用*AR_FirDl 增减方式，而不是*AR_FirDl+，这样第一个内循环之后 AR_FirDl 指向了 FirDl[LpcOrder-1]。第二个循环则不采用 C 代码中指针增的方式，而是倒序，这样就正好可以利用第一个循环后 AR_FirDl 的值。第二个循环结束以后 AR_FirDl 指向 FirDl[-1]，为了给 FirDl[0]赋值，采用*+AR_FirDl 的寻址方式，此后指针重新指向 FirDl[0]，从而为下次运行到第一个循环处又做好了准备。

如果不合理利用指针增量，在每个内循环前都要用 MVMM 或这 MVDK 之类的语句给 AR_FirDl 设置初值。

3.4.4 AR0 作为循环次数

在求卷积等操作中，经常有以下的运算

```
for ( l = SubFrLen-1 ; l >= 0 ; l -- ) {
   Acc0 = (Word32) 0 ;
   for ( j = 0 ; j <= l ; j ++ )
      Acc0 = L_mac( Acc0, OccPos[j], Imr[l-j] ) ;
   Acc0 = L_shl( Acc0, (Word16) 2 ) ;
   OccPos[l] = extract_h( Acc0 ) ;
}
```

一般的做法是：

```
    STM    #(Imr+SubFrLen-1),       AR_Imr_l
    STM    #(OccPos+SubFrLen-1),       AR_OccPos_l
    STM    #(OccPos),           AR_OccPos
    STM    #(SubFrLen-1),BRC
    RPTBD  ResponseEnd
    ST     #(SubFrLen-1),m_InnerLoopCnt          ;内循环的次数
        MVMM   AR_Imr_l,  AR_Imr_l_j
        MVMM   AR_OccPos, AR_OccPos_j
        LD     #0,A
        RPT    m_InnerLoopCnt
            MAC    * AR_OccPos_j+,* AR_Imr_l_j-,A
        STH    A,2,* AR_OccPos_l-
        ADDM   #(-1), m_InnerLoopCnt
ResponseEnd:
        MAR    *pImr_l-
```

该程序中将二重循环指针 AR_OccPos_j 和 AR_Imr_l_j 的初始值分别保存在 AR_OccPos 和 AR_Imr_l 中，二重循环开始前用 MVMM 指令赋值，由于二重循环次数是个变数，所以二重循环次数需要保存在变量 m_InnerLoopCnt 中，并且在一重循环末

递减 m_InnerLoopCnt。

由于用 ADDM 指令递减 m_InnerLoopCnt 需要二个指令周期，递减操作最快的操作就是使用*ARx-，能否将内循环次数保存在 AR0 中，采用 MAR *AR0-递减呢？这就是 AR0 作为循环次数的方法。

使用该方法后，第 10 行指令应变为“RPT *(AR0)”，这是采用*(1k)方式寻址 AR0 的内容，所以这条指令增加一个指令周期，MAR *AR0-少一个指令周期，总的程序时间并没有优化。但是采用该方法，将会带来一个好处。采用该方法时，例子中二重循环循环的次数为 AR0+1 次，而二重循环使得 AR_OccPos_j 和 AR_Imr_l_j 各自增减总量为 AR0+1。所以可以用 AR_OccPos_j-0 和 AR_Imr_l_j+0 方式可将 AR_OccPos_j 和 AR_Imr_l_j 设置为二重循环开始时的值，从而不需要用 MVMM 指令再次赋初值。程序如下：

```
          STM    #(Imr+SubFrLen-1),       AR_Imr_l
          STM    #(OccPos),           AR_OccPos
          STM    #(SubFrLen-2),BRC
          RPTBD      ResponseEnd
          STM    #(SubFrLen-2),           AR0
             LD     #0,                  A
             RPT    *(AR0)
                MAC    *AR_OccPos+,  *AR_Imr_l-,A
             MAC    *AR_OccPos, *AR_Imr_l+0%,A          ;后向部分循环展开
             STH        A,2,   *AR_OccPos-
             MAR    *AR_OccPos-0
ResponseEnd:
             MAR    *AR0-
          ;最后一次循环需单独处理
          LD         #0,        A
          MPY    *AR_OccPos,   *AR_Imr_l, A
          STH        A,2,       *AR_OccPos
```

程序采用后向部分循环展开将最后一次循环单独处理，并使用 AR_Imr_l+0%，从而将 AR_Imr_l 设置为下次循环需要的初始值；使用*AR_OccPos 为下条语句使用 AR_OccPos 做好准备。本例使用“AR0 作为循环次数”，节省 2*(SubFrLen-2)个指令周期（SubFrLen-2 为一重循环次数）。

3.4.5 码本搜索优化实现

码本搜索是在 G7231 程序中经常出现的一种结构，占用较大的运算量。码本搜索其实就是在数组 array[SIZE]中搜索最大（或最小）值 array[Indx]，并且记录此时对应的索引 Indx。典型 C 代码如下：

```
          Acc1 = 0;
          for(i=0; i<SIZE; i++)
          {
             Acc0 = array[i];
             if(Acc0 > Acc1)          ; 第 4 行
             {
                 Indx = i;            ; 第 6 行
                   Acc1 = Acc0;
```

```
        }
    }
```

其中“第 4 行”有 4 种变化分别为①Acc0 > Acc1（极大值）② Acc0<Acc1（极小值）③ Acc0>=Acc1（最大值）④Acc0<=Acc1（最小值）。

3.4.5.1 实现方法一

以求最大值的索引为例，首先一个问题是如何判断是最大值。这里采用 MAX 指令。MAX 指令能够取 A 和 B 累加器中较大者放到目的累加器中。MAX 指令根据执行结果设置 C 标志位。设置方法为，如果 A>B，取 A 的值放到目的累加器，C=0；否则，取 B 的值放到目的累加器，C=1。我们若采用 MAX B 指令，则如果 C=0，则说明找到了更大的值。求最小值的时，使用 MIN 指令，C 标志位设置方法为，A<B，C=0；否则 C=1。

根据条件存储索引。循环计数寄存器 BRC 在每次循环中减一，所以根据 BRC 可以推得当前索引 i（SIZE-1-BRC=i）。利用 SRCCD 指令可以根据条件决定是否存储 BRC 的当前值到 Xmem（和 Smem 对应，Xmem 指用双操作数寻址 RAM），判断和存储总共只需 1 个指令周期。这里用 SRCCD 存储 BRC 值，然后通过 Xmem 中的值计算最佳索引 Indx。但是使用 SRCCD 要注意流水线冲突，BRC 的减一操作在循环最后一条指令的解码阶段被执行，这要求 SRCCD 指令到循环末尾指令之间要有 3 个指令周期。如果在 SRCCD 之后加三个 NOP，则效率变低。一个可能想到的办法是不使用 NOP，而用修改计算公式 SIZE-1-BRC=i 为 SIZE-1-BRC+1=i 的方法，因为既然 BRC 是减一以后的值，使用 BRC+1 能恢复到原来的值。但是由于这种方法会使最后一次循环的 BRC 和倒数最后一次循环的 BRC 值都为 0，而无法恢复 i 的真实值。所以，这里使用前向部分循环展开法来消除 SRCCD 后的三个 NOP。

由于 SRCCD 指令不能判断 C 标志，即不能用 C 或者 NC 作为条件，所以下面采用相减的办法产生 SRCCD 需要判断的条件，即 A-B=0 时说明 Acc0>=Acc1。

以求最大值索引（“Acc0>=Acc1”）情况为例，码本搜索程序如下：

```
        LD      #0,         B
        STM     #(SIZE-1), BRC
        MVDK    array, AR_array
        LD      *AR_array+,    A          ; 部分循环展开
        RPT BD  RotationEnd
        MAX     B                         ; 部分循环展开
        SUB     B,          A             ; 部分循环展开
            SRCCD   *AR_Indx,  AEQ        ; 离循环末有 3 条指令
            LD      *AR_array+,    A      ; 部分循环展开
            MAX     B                     ; 部分循环展开
    RotationEnd:
            SUB     B,          A         ; 部分循环展开
        LD      #(SIZE-1), A
        SUB     *AR_Indx,          A      ;最后索引在 A 中
```

该方法，循环内部指令周期数为 4。由于采用“部分循环展开”，最后一次循环多运行了三条指令，这三条指令可能改变 B 的值，这样求得的 B 可能不再是最值，除非保证 array[SIZE]的值不可能比 array 中的其它值都大，使得最后一次循环不可能改变 B 的值，这可以在 array[SIZE]中存放最小负数来实现。

如果是求最小值（“Acc0<=Acc1”），则只要将以上代码的 MAX 改为 MIN 即可。

极大值（Acc0>Acc1）和极小值（Acc0<Acc1）无法用该方法实现，因为方法一程序中的 A-B=0，说明了 Acc0>=Acc1，但是并不知道是 Acc0==Acc1 还是 Acc0>Acc1。

3.4.5.2 实现方法二

由于方法一不能适用于极大、极小值的情况，另外为了求得最值需要需要额外处理等问题，提出方法二。方法二不再有上述问题，但是循环内部指令周期数为 5。方法二，不再使用相减产生判断条件，使用 ADDC 指令将 C（NC）判断条件转化为 ANEQ(AEQ)条件。以求最大值“Acc0>=Acc1”情况为例，码本搜索程序如下：

```
        LD      #0,         A
        ST      #0,         m_Zero          ; 增加一句
        STM     #(SIZE-1), BRC
        MVDK    array, AR_array
        LD      *AR_array+,    B
        RPT BD  RotationEnd
        MAX     A
        LD      #0,            B
            ADDC    m_Zero,    B           ;B==0 表示 C=0，表明 B>=A
            SRCCD   *AR_Indx,  BNEQ        ;离循环末有 3 条指令
            LD      *AR_array+,B
            MAX     A
    RotationEnd:
            LD      #0,        B
        LD      #(SIZE-1),     A
        SUB     *AR_Indx,      A
```

说明如下：当 Acc0>=Acc1 时，也就是 B>=A 时，认为找到一个最大值，此时 MAX A 指令取 B 放到目的累加器，C=1，ADDC 以后 B 的值为 1，所以使用 BNEQ 作为存储索引的条件。对于 Acc0>Acc1 类型搜索，需要调换 A 累加器和 B 累加器的位置，并且使用 AEQ 作为存储条件。对于求最小值的搜索需要使用 MIN 指令。四种类型的使用总结如表 3－2：

表 3－2 四类码本搜索类型对应方法二程序变化

搜索类型	累加器使用	产生判断的语句	找到最值时的 C 标志位	存储条件
最大值：Acc0>=Acc1	A 存放 Acc1 B 存放 Acc0	MAX A	C=1	BNEQ
最小值：Acc0<=Acc1	A 存放 Acc1 B 存放 Acc0	MIN A	C=1	BNEQ
极大值：Acc0>Acc1	B 存放 Acc1 A 存放 Acc0	MAX B	C=0	AEQ
极小值：Acc0<Acc1	B 存放 Acc1 A 存放 Acc0	MIN A	C=0	AEQ

3.4.5.3 实现方法三

由于方法一和方法二使用 SRCCD 指令，所以一般需要部分循环展开。如果使用 XC 指令代替 SRCCD 指令则不必使用部分循环展开。方法三将 i 存放在 AR_i 辅助寄存器中，

在循环中自增 AR_i，当找到一个最值时，用 XC 指令条件执行 MVMM 指令来存储 AR_i 到 AR_Indx 中。用方法三求最大值“Acc0>=Acc1”的代码如下：

```
        LD      #0,         A
        STM #(SIZE-1), BRC
        STM #(-1),          AR_i
        RPT BD  RotationEnd
        MVDK    array, AR_array
            LD      *AR_array+,    B
            MAX     A
            NOP
            MAR     AR_i+
            XC      1,  C
    RotationEnd:
                MVMM        AR_i,  AR_Indx
```

由于 XC 指令要求判断条件在至少前 2 个周期产生，所以在 MAX 指令和 XC 指令之间需要加入 2 个周期的指令，这里还剩余一个 NOP 没有被替换。这样它的循环内部指令周期数为 6。该方法的好处是，不用部分循环展开，所以没有部分循环展开最后一次循环额外执行的指令，并且求得的 AR_Indx 直接是索引，不用公式转化。所以在能够替换循环中的 NOP 的情况下（此时循环内部指令可降为 5），可考虑使用方法三代替方法二。

求极大值、极小值、最小值的索引需要稍微修改一下程序，修改的方法同方法二，见表 3－2。

3.4.5.4 三种方法的比较

三种实现方法各有优缺点，在不同的场合可适当选择使用。表 3－3 为三种方法的比较：

表 3－3 三种码本搜索方法的比较

方法	搜索类型	循环内部指令数	是否求得索引	是否求得最值	部分循环展开开销	用公式转化 Indx 的开销	使用辅助寄存器个数
方法一	最大值、最小值	4	是	需要额外处理	3	2	2
方法二	最大值、最小值、极大值、极小值	5	是	是	3	2	2
方法三	最大值、最小值、极大值、极小值	6（替换 NOP 后为 5）	是	是	0	0	3

3.4.6 组合条件的优化实现

C 语言中用&&和||符号组合多个条件。例如：

```
if(Bindx == (Word16) 0) && (Findx == (Word16) 0)
```

汇编中一般可以用如下方法实现：

```
LD      Bindx,      A
BC      NotExcuteIf,  ANEQ        ;不等于 0 时，跳过 If 需要指令的语句
```

```
LD      Findx,     A
BC      NotExcuteIf,  ANEQ
```

但是该方法需要 10 个指令周期。下面讨论优化方法的实现。

1、利用 AND 和 OR 指令

该方法适用于“==0”和“!=0”类条件的组合，以上程序实现方法如下：

```
LD      Bindx,     A
OR      Findx,     A
BC      NotExcuteIf,  ANEQ
```

使用 OR 将两个跳转组合为一个，只用 6 个指令周期。AND 的使用方法类似。

2、利用 XC 指令

该方法适用于任何条件的组合。例如

```
if(Bindx == 2) && (Findx <3)){......}
else{......}
```

的实现如下：

```
CMPM Bindx, #2
LD       Findx, B
SUB      #3,    B
XC       1,     NTC
  LD     #0,        B       ;如果 Bindx == 2，强制让 Findx>=3
BC       DoElse,    BGEQ              ;如果 Findx>=3，则跳过 If
; DoIf
DoElse:
```

一般 XC 指令比 BC 指令要高效，这里将前一个 BC 指令用 XC 指令代替，指令数为 11。

3、使用多条件组合指令

该方法适用于一个条件为等（或不等）另一个条件为大小比较的组合。上例实现为：

```
CMPM  Bindx, #Rate63
LD       Findx, A
SUB      #3,    A
BC       DoIf,   TC, NC
;DoElse
DoIf:
```

指令数为 9。该方法使用了 BC 指令的多条件跳转，只有两个条件都符合才跳转，所以它是一个与的关系，而这里给出的条件是两个条件相与，所以跳转的目的地址为 If 模块，不跳转时进入 Else 模块。如果组合条件为或的关系，例如 if(Bindx == 2) || (Findx <3))则先改为与的关系，即将原条件取非变为(Bindx!=2)&&(Findx>=3)，此时 BC 跳转的目的地址为 Else 模块，不跳转进入 If 模块。等于和不等于用 TC 和 NTC 判断，大小用用相减后的 C 标志位判断，判断方法如下表 3－4：

表 3－4 各类情况下使用减法比较大小使用方法

判断条件	使用的减法顺序	条件符合时 C 的值
value>= #1k	Value-#1k	C=1
Value<= #1k	#1k - Value	C=1
Value> #1k	#1k - Value	C=0
Value< #1k	Value-#1k	C=0

注：SUB 指令，借位的时候设置 C＝0，否则 C＝1。(src) – (Smem) << 16 的形式影响 C 标志位的方式和其它指令不同，只有在借位的时候设置为 0，不借位时不影响 C。

3.4.7 修改程序以适应汇编

3.4.7.1 重新组织内存结构

1、让变量存储空间连续分布以使用 MVDD 指令。如下例：

```
dst1 = scr1;
dst2 = scr2;
dst3 = scr3;
```

为了实现上述语句，汇编中要么使用 6 个辅助寄存器分别指向这 6 个变量，然后使用 MVDD 指令；要么只能使用 LD scr1,A 和 STL A,dst1 两条指令来实现数据的拷贝。为了解决这个问题，我们在汇编中将 dst1,dst2,dst3 连续存放在内存中，scr1,scr2,scr3 连续存放。此时可以用

```
MVDD  *AR_scr1Addr+,    *AR_dst1Addr+
MVDD  *AR_scr1Addr+,    *AR_dst1Addr+
MVDD  *AR_scr1Addr+,    *AR_dst1Addr+
```

这三条单指令来实现。而且只用两个 ARx。

2、设计变量存储结构使更有利于指针增减。在 G7231 的 AtoLsp 函数中，数组 p 和 q 的元素是交替存放在数组 Spq 中，这时需要指针增 2(使用*+ARx(2))来访问每个 p 元素和 q 元素，为了使用更高效的*ARx+方式，我们重新分配数据结构，在 Spq 中，先存放 p 元素然后存放 q 元素。

3.4.7.2 语句的合并

C 语言的多条语句可能被合并为一条 DSP 汇编指令。例如，MPY＋ADD 可合并位 MAC，例如：

```
for(i=0;i<10;i++){
  Acc0 = L_mult( Tv[j], FltBuf[i][j] ) ;
  Acc1 = L_add( Acc1, L_shr( Acc0, (Word16) 1 ) ) ;}
```

循环内两条语句可用 FRCT=0 的单条 MAC 指令代替（注意当两个乘操作数为 0x8000*0x8000 时这个代替将不等价）。将语句合并时要注意等价性，例如以上语句如果循环内部使用 FRCT=1 的 MAC，循环结束后使用 SFTA A,-1 实现右移的方法会使得合并不等价，因为这时循环内部的累加器值放大了一倍，可能使累加器溢出。

3.4.8 用硬件滤波器结构

这里主要指 FIR 和 IIR 滤波器结构。FIR 或 IIR 滤波器每产生一个输出，滤波器主要有两个操作：①采样数据乘以滤波系数后累加（减）②滤波器中数据后移一个单元（Delay 操作）。C 语言中实现如下：

```
for ( j = 0 ; j < LpcOrder ; j ++ )
   Acc0 = L_msu( Acc0, FirCoef[j], DecStat.PostFirDl[j] ) ;
for ( j = LpcOrder-1 ; j > 0 ; j -- )
   DecStat.PostFirDl[j] = DecStat.PostFirDl[j-1] ;
DecStat.PostFirDl[0] = Tv[i] ;
```

C54x 的 MACD 指令能够同时实现乘累加和数据后移， MACD 的使用如下：

```
MACD  Smem,pmad,src
```

我们让滤波数据放在 Smem 中，滤波系数放在 pmad 中，pmand 是程序空间的一个地址，当 OVLY＝1 时，也可以是低 32K 数据空间的地址。MACD 的执行操作为

```
Src = src + Smem*Pmem, T<－Sem, (Sem+1)<－Smem
```

可以看到它执行乘累加，并且 Sem 中的数据向高地址移动一位。当 MACD 用 RPT 指令循环时，它变为单周期指令，并且 pmand 会自动增一。由于 Smem 数据向高地址移动，所以滤波时为了防止覆盖，需要先利用 Smem 中高地址的数据，所以要使用*ARx-。但是由于 pmand 只能增加，所以系数只能反序存放（FirCoef[0]在高地址）。以上程序实现如下：

```
MAS      *AR_PostFirDl_j-, *AR_FirCoef_0,    A
RPTZ     B,     #(LpcOrder-1-1)
  MACD   *AR_PostFirDl_j-, #(FirCoef+1), B
SUB      B,         A
LD       *AR_Tv_i,  B
STL      B,         *AR_PostFirDl_0
```

说明：①对最后一个数据 DecStat.PostFirDl[LpcOrder-1]，不需要使用(Sem+1)<－Smem 移位操作。当然也可以使用，但是这时要保证 DecStat.PostFirDl[LpcOrder]内存单元不是有用数据单元，可以被任意修改。②MACD 指令执行累加，对于需要累减的情况（例如此例），先在 B 累加器中累加，然后 B 累加器减 A 累加器即可。③有时候系数 FirCoef 是一个变量，也就是说，不一定采用固定的滤波系数，但是 MACD 指令的 pmand 要求是一个固定址（立即数表示）。解决办法是先将系数拷贝到 pmand 指向的地址空间，然后使用 MACD 指令，当然这需要额外的指令，这需要看一下是否值得这么做，多数情况下是值得的，因为使用 MACD 指令比 MAC 和 DELAY 的组合实现滤波器结构几乎快一倍。

3.4.9 通过堆栈的返回值传递方法

标准 C 函数只能有一个返回值，如果有多个返回值时，一般要通过函数参数将变量的指针传递给被调函数，在被调函数中通过指针修改这个变量，以达到返回多个值的目的。但是这种方法实现起来需要较多的指令。

在汇编函数中可以通过寄存器或者全局变量来传递返回值，但是这要占用寄存器和 RAM 资源，我们的程序中采用了通过堆栈传递返回值的方法。既然可以通过堆栈将主调函数的参数传递给被调函数，当然也可以将被调函数的返回值传递给主调函数。

3.4.10 循环寻址结构的使用

在AtoLsp中，有一个512个字的cos表，为了获得的cos表的值C语言中用类似CosineTable[i%CosineTableSize]的方式求得，CosineTable[i%CosineTableSize]在汇编中可用环形（Circular Address）寻址方便实现[15]。环形寻址时要求CosineTable数组定位在大于CosineTableSize的2^N边界，BK寄存器加载CosineTableSize。

3.5 G.723.1 中基本操作的实现

3.5.1 条件转移结构的比较

一般条件转移用 BC 指令实现，但是 BC 指令需要 4 个周期（条件真时），即使使用 BD 指令也需两个指令周期。所以 C54DSP 汇编中要慎用 BC 和 BCD 指令。

对于没有 Else 的 If 模块，如果 if 模块内为少数几条指令时，使用 XC 比 BC 要高效。

如果 if 模块中为单条存储语句，可用条件存储指令实现。BCD 和 XC 的使用效率比较如表 3－5：

表 3－5 BCD 和 XC 指令的比较

If 模块内汇编指令数	执行 if 模块的概率	使用 BCD 指令还是 XC
1	/	使用 XC 指令
2	小于 0.5	使用 BCD 指令
2	大于等于 0.5	使用 XC 指令
大于等于 3	/	使用 BCD 指令

对于有 Else 的 If 模块。如果出现类似以下结构 if(flag==1)Acc0=1;else Acc0=0;经常改为 Acc0=0;if(flag==1)Acc0=1;结构，这样就可以使用 XC 指令。

另外在安排 If 和 Else 模块时，哪个模块放在 BC 指令的跳转目的也有讲究。因为跳转需 4 个指令周期，不跳需 2 个指令周期，如果 If 模块被执行的概率较高，则需要将 BC 指令的跳转目的设置为 Else 模块。这可从统计上减少周期数。

对于 switch-case 结构常使用 CMPM 和 BC 组合实现。

3.5.2 各种循环结构的比较

实现循环体有 4 种方法分别为 RPT(RPTZ)、RPTB(RPTBD)、BANZ(BANZD)、BC(BCD)。它们效率逐渐降低，一般越内层的循环使用越高效的循环结构。它们的性能对比如表 3－6：

表 3－6 各类循环结构的比较

类型	执行整个循环体所需额外指令数	每循环一次所需额外指令数	额外限制
RPT,RPTZ	2	0	循环内只能有一条语句。不能使用*(1k)寻址方式
RPTB,RPTBD	4,不能替换两个 NOP 时为 6	0	循环内部再嵌套 RPTB 时，需要额外处理
BANZ,BANZD	2+2*SIZE，不能替换两个 NOP 时为 2+4*SIZE	2，不能替换两个 NOP 时为 4	使用了一个寄存器
BC，BCD 使用 ARx 方式（将循环计数量放到 ARx）	2+4*SIZE，不能替换两个 NOP 时为 2+5*SIZE	4，不能替换两个 NOP 时为 5	使用了一个寄存器
BC，BCD（将循环计数量放到 Smem 中）	2+5*SIZE	5	无

3.5.3 整数小数除法和求余

C54x 汇编中 SUBC 指令实现除法，SUBC 的语法为“SUBC Smem, src”SUBC 指令功能如下：

```
(src) - ((Smem) <<15) →ALU output
If ALU output≥ 0
Then  ((ALU output) << 1) + 1 →src
```

```
Else  (src) << 1 → src
```

SUBC 相当于手动做除法，并将结果从 src 的最右端移入 src。

3.5.3.1 小数除法

这里除法的要求是这样的：被除数 Num 是属于[0,1）的小数，Den 是属于(0,1)的小数，并且为了保证商小于 1，Num<Den。如果 Num 或者 Den 是一个负数，先将其取反，然后将所得的商取反即可。

32bit 小数除 16bit 小数除的程序如下（设 Num 已经加载到 A 和 B 中）：

```
ABS        A
RPT        #(15-1)
    SUBC          Den,          A
AND    m_data_7FFF,  A
XC         1,         BLT     ;判断是否为负数
    NEG    A                  ;负数则取反
```

根据 SUBC 指令功能，Den<<15 后和 A 比较，所以第一次 SUBC 得到的 bit 是商的最高位，商最高位放在 A 的第 15bit，所以共需要左移 15bit，SUBC 需执行 15 次。之后，A 第 16bit 是 SUBC 循环之前的 A 的第 1bit，这里需要屏蔽这一位和 A 的高 16bit，这可通过将 A 和 0x7FFF 相与实现。最后如果被除数是负数则取反。

16bit 小数除以 16bit 小数的方法和以上方法类似，只是需要将 Num 加载到 A 的高 16bit，并保证 A 低 16bit 为 0。（设 Num 已经加载到 A 和 B 高 16bit）：

```
ABS        A
RPT        #(15-1)
    SUBC   Den,          A
AND    m_data_7FFF,  A
XC         1,         BLT     ;判断是否为负数
    NEG    A                  ;负数则取反
```

虽然此时，SUBC 执行 15 次以后，A 第 16bit 必定为 0，但是还是需要用 0x7FFF 屏蔽，因为 A 高 16bit 并不一定为 0，从而影响后面的 NEG 指令的结果。

3.5.3.2 整数除法

1、16bit 正整数的除法和求余。

将被除数加载到 A 的低 16bit，然后用一下方法：

```
RPT   #(16-1)
  SUBC   Den,   A
```

执行后，商在 AL 中，余数在 AH 中

2、32bit 正整数的除法和求余。请参考有关文献[16]

3.5.3.3 求余

求余有的地方也称为求模。C 语言中用“%”符号来求余。以下介绍几种求余方法，要根据情况选用简单的求余方法。

1、利用除法求余。参考“整数除法”部分，这种方法对于 16bit 数求余需 16 个指令周期；32bit 数对 16bit 求余需要更多的指令周期。

2、16bit 数或者 32bit 数对 2 的幂次方数的求余。这种类型的求余可以用“与”操作代替。例如 Num 对 8 求余，就是 Num&8。

3、自增量的求余。例如

```
for(i=0,i<4* CosineTableSize ;i++)
  Acc0+=CosineTable[i%CosineTableSize];
```

这时可以用环形寻址实现 CosineTable[i%CosineTableSize];部分。请看另一种情况：

```
for ( i = 0 ; i < SubFrLen+ClPitchOrd/2 ; i ++ )
   Tv[ClPitchOrd/2+i] = PrevExc[PitchMax - (int)Lag + i%(int)Lag] ;
```

此例和上例的区别是：上例 CosineTable 是一个常量（数组头指针），此例 PrevExc 为一个变量（指针）；CosineTableSize 为常数，Lag 为变量。上例实现环形寻址时要求 CosineTable 数组定位在大于 CosineTableSize 的 2^N 边界；由于 PrevExc 为变量，且 Lag 为变量，所以不好实现环形寻址。为此我们只好先求其中的 i% Lag，这里 i 是一个自增量，而且每次自增的量不超过 Lag（自增量为 1）。所以在 C 中我们可以用如下逻辑实现求余：

```
if(i>=Lag)i -= Lag;
```

这避免了用除法求余，在 DSP 汇编中可以巧妙地利用环形寻址实现求余。我们假想一个定位在 0 地址大小为 Lag 的数组，当对这个数组进行环形寻址时，正好是实现了 i%Lag 的操作。所以，在寻址 PrevExc[PitchMax - (int)Lag + i%(int)Lag]时，将 PrevExc+PitchMax - (int)Lag 存放在 AR3 中，将 i%(int)Lag 存放在 AR0 中。BK 寄存器中存放 Lag，AR0 初始化为 0。这样循环中先运行以下语句：

```
MVMM    AR3,   AR4
MAR     *AR4+0
MAR     *AR0+%            ;此句实现 i%Lag
```

然后用*AR4，即可寻址。

3.5.4 双字数据乘单字数据

也就是 32bit 数据（设为 Acc1）乘以 16bit 数据（设为 y），这里只讨论 Acc1 和 y 为有符号数的情况。设 Acc1 高 16bit 为 H_Acc1，低 16bit 为 L_Acc1。

3.5.4.1 小数乘法

32bit×16bit 小数乘法实现是， (L_Acc1*y)>>15+(H_Acc1*y)<<1。其中“*”表示整数乘法（FRCT＝0），其中 L_Acc1*y 中 L_Acc1 为一个无符号数，y 则为有符号数，这时一般只能利用 MACSU 指令实现，它的两个乘操作数中，第一个为无符号数，第二个为有符号数。

1、一般情况代码如下：

```
LD      #0,       A
RSBX  FRCT
MACSU *AR_Acc1-, *AR_y, A          ;L_Acc1*y
SFTA  A,         -15
SSBX  FRCT
MAC   *AR_y, *AR_Acc1+, A          ;+(H_Acc1*y)<<1
```

我们在 G7231 程序中默认 FRCT=1，所以需要设置 FRCT=0，然后重设置为 1。需要注意的是 FRCT=0 的(L_Acc1*y)>>15，不能用 FRCT=1 的(L_Acc1*y)>>16 实现，因为有溢出问题。

2、y 为正数情况。例如 y 为一个常正数的情况。此时因为两个操作数都可视为无符号数，所以可以用 MPYU 指令代替 MACSU 指令，这种方法一般比第一种方法指令数少。

```
STLM  A,            T          ;设 Acc1 已经存放在 A 中
RSBX  FRCT
        MPYU   y,             B          ; L_Acc1*y
SFTA  B,            -15
SSBX  FRCT
MACA  y,            B          ;+(H_Acc1*y)<<1
```

3.5.4.2 整数乘法

1、只讨论 y 为正数情况。例如，Acc1 * 90，Acc1 为 32bit 整数。设开始时，FRCT=0：

```
ST     #90,       y       ;设 Acc1 在 A
MPYA   y                  ;中 B = H_Acc1*y
STLM   A,         T       ;T = L_Acc1
SFTA   B,         8
MPYU   y,         A       ;L_Acc1*y
ADD B,8,          A       ;没有溢出控制
```

一般整数乘法不需要溢出控制（在 G7231 中是这样），所以代码最后一条语句没有进行溢出控制，如果结果超过 32bit 数的表示范围则结果不对（实际上进行溢出控制后结果也是不对的）。

3.5.5 单字运算操作

G7231 程序中有很多的单字（16bit 数）的运算。如果要对单字运算进行溢出控制，需要在累加器的高 16bit 中做运算，例如单字求反、求绝对值，需要加载到累加器高 16bit 然后利用 NEG 或 ABS 指令。其中常使用“LD Smem,16, scr”,“SUB Smem,16, scr”,“STH scr, Smem”等支持高 16bit 操作的指令。

3.5.6 多位数移位和快速移位

1、多位数移位

移位范围最大的指令为 NORM 指令。移位 TS 次，TS 有效范围[-16,31]，TS 就是 T 寄存器的低 6bit。我们特意测试了当 TS 在[-17,-32]范围时的情况，它的实际移位为 TS+16，虽然移位次数不对，但是这个特性将在多位数移位中被利用；当 TS 不在 6bit 表示范围内时，只取低 6bit。当需要移位次数超过[-16,32]时，NORM 指令不能一次完成移位操作。例如：

```
Sen = L_shl( Acc1, (Word16)(2*Exp - 4) ) ;
```

其中 Exp 的范围是[-13,3]，所以左移次数范围[-30,2]，超过 NORM 指令范围。注意，本例不能使用多次移位法实现，例如用如下方法实现：

```
LD    Exp,        T
NORM  A
NORM  A
SFTA  A,          -4
```

这是因为，当 Exp 为正数时，该方法出现先左移后右移的情况，这和一次性移位结果不同。例如 A=0x4000 0000，Exp=1,左移一位以后为 0x7FFF FFFF（溢出控制），再左移一位也是 0x7FFF FFFF，右移 4 位后为 0x07FF FFFF，这和想要的结果不同。得出以下结论：一次性移位不能用左移和右移多次来实现。实际中利用 TS 在[-17,-32]范围时的移位特性巧妙实现移位。即当 TS 处在[-17,-32]时只要再左移-16 位即可。

```
LD           Exp,1, B
```

```
SUB     #4,     B
STLM    B,      T
SUB     #17,        B
NORM    A
XC      1,      BLEQ            ;当 TS 处在[-17,-32]执行下面语句
SFTA    A,      -16             ;已经左移 TS+16 位，需要再左移-16 位。
```

3、快速移位法

常用左移代替右移来快速实现移位，例如 h[i] = shr(h[i], 1);的实现可用

```
LD      *AR_h,-1,  A                ;双指令
STL     A,         *AR_h+
```

使用左移代替右移实现为：

```
LD      *AR_h,(16-1), A             ;单指令
STH     A,         *AR_h+
```

3.5.7 数据拷贝

1、数据区的数据拷贝常使用 MVDD、MVDK 和 MVKD。后面两条指令在向（从）固定地址拷贝数据时比 MVDD 优越，因为它只使用一个辅助寄存器。

2、双字数组拷贝。双字操作一般用双字指令，但是它的拷贝不使用 DLD 和 DST 的组合而使用 MVDD。拷贝一个双字数据 DLD+DST 需要 3 个指令周期；MVDD 只需 2 个。

3.5.8 其它操作的快速实现

1、 Sem 移位以后加到累加器。指令“ADD Smem[,SHIFT],src[,dst]”需 2 周期指令，“ADD Xmem[,SHFT],src”和“ADD Smem,TS, src”需 1 周期指令。在需要快速指令的地方，若 AR2-AR5 有空余，可以使用第 2 条指令；若没有空余，可以先设置好 TS，使用第 3 条指令。

2、 Smem 加#K。其中#K 属于[0,255]，采用

```
LD      #K,     A       ;单指令
ADD i,      A           ;单指令
```

不应使用

```
LD      i,      A       ;单指令
ADD #K,     A           ;双指令
```

3.6 G.723.1 中代码优化的思路

优化一般主要从三方面考虑：速度、程序空间大小和数据空间大小。然而速度和程序空间、数据空间的优化是互相抵触的。我们在 G7231 中采取了：提高速度侧重在循环内部；减少代码量侧重在循环外部的指导思想。在循环内部常常以增加代码量为代价提高速度；在循环外部常以增加少量指令周期为代价压缩代码量。

3.6.1 提高程序速度的指导思想

为了提高速度，要注意以下几点：使用带延时的调用、跳转、返回指令；防止流水线冲突；选择高效的指令；以及本文前面介绍的一些方法（直接寻址、使用环形寻址、并行指令、使用 MIN、MAX、XC 等特殊指令等）。

由于提高速度主要是减少循环内部指令周期数，除了以上的方法外，这里主要针对

循环体介绍循环体优化方法。循环体优化的一个重要思路是：将循环内部的事情放到循环外部来做。为了循环体内部能够使用单周期指令，循环外部需做很多的准备，例如，1、使用 ARx+0 代替+ARx(1k)，这就需要设置 AR0 的值。2、“码本搜索优化实现”中在循环外使用公式修正得到的码本索引。

3.6.1.1 将循环内部的立即数替换为 Smem

由于一般有立即数的指令都是 2 周期的，如果将立即数替换为 Smem 则变为 1 周期指令。例如循环内部 2 周期指令 ADD #1, A[17]可用单周期指令 ADD m_data_1, A 代替，这需要在函数堆栈区的“改写为汇编以后增加的变量”部分增加一个字（即 m_data_1），并在循环外将 1 写入 m_data_1 中。

3.6.1.2 循环内部指针的初始化

指针的初始化如果用 STM 或者 MVDK 是比较慢的。1、循环内部的指针初始化可使用 MVMM 指令，这就需要事先将初始化值保存在另一个 ARx 中，所以它需要两个 ARx 辅助寄存器。2、使用“合理利用指针增减”的方法，避免初始化。

3.6.1.3 ARx 及 CPU 资源分配

ARx 及 CPU 资源（主要指寄存器）在汇编中是很宝贵的，多使用一个 ARx，有时能少用很多指令。

3.6.1.3.1 由内自外的编程顺序

当有多重嵌套循环时，首先应该注重减少最内层的循环的指令数：①在各种循环结构中，应该让最高效的循环结构用于最内层的循环。②提高程序速度，有时必须占用 ARx 和 CPU 资源（主要指寄存器），在最内层循环需要 ARx 和 CPU 资源时应尽量先满足最内层循环。为此我们提出了“由内自外的编程顺序”。一般编写程序的顺序是从上向下编写，对于 C54 汇编的多重循环我们建议采用先从最内层循环开始编写的编程顺序，这将带来以下好处：

1、 可以考虑先让最内层循环使用最高效的循环结构、ARx 和 CPU 资源，然后再逐层照顾外层循环的使用。
2、 内层循环可以尽量考虑使用最快速的指令，而快速指令往往需要在循环外做某些准备，所以只有先编写循环内代码，然后才能知道循环外需要做哪些事情。

3.6.1.3.2 使用 PSHM 和 POPM 保存 ARx

假如最内层循环使用了全部的 8 个 ARx 寄存器，但是外层循环必须使用间接寻址，也就是说必须使用 ARx，这时如果最内层循环放弃使用一个 ARx，则有可能使指令周期数大大增加，这就形成一个两难的境地。以二重循环为例，处理方法如下：可以让一重循环使用一个 ARx，当进入二重循环前使用 PSHM ARx 指令保存 ARx，这时二重循环内可使用这个 ARx，退出二重循环后使用 POPM ARx 恢复 ARx 的值。

保存和恢复 ARx 在一重循环中增加了 2 个指令周期，但是这往往是值得的。注意，使用 PSHM 以后改变了 SP，通过基于 SP 的直接寻址时，需要改变偏移量。

该方法虽然将增加程序的复杂度，但是减少的指令周期数往往相当可观。G7231 的 5.3K 编码的 D4i64_LBC 函数就是采用这种方法。

3.6.2 节省代码量的方法

主要在循环外考虑减少代码量，减少代码量的方法有：1、将相同功能的代码改写为函数。2、将重复的操作改写为循环体。这两个方法主要是编写 C 代码时考虑，将 C 改写为汇编时，出现的地方不多。C 改写为汇编时，主要减少代码的一个情况是：将多处需要计算的变量先保存起来。例如 Find_Best 函数中多处用到 Imr+k，在程序中可以增加一个 m_ImrPlusk 局部变量，将 Imr+k 保存起来，以后只用读取 m_ImrPlusk 即可。

3.6.3 节省数据空间的方法

G7231 中数据空间主要用于分配局部变量中的数组、以及 G.723.1 的各种表。

3.6.3.1 OVERLAY 技术

这里的 OVERLAY 技术源于 51 单片机 C 编译器 KeilC（http://www.keil.com）的 OVERLAY 技术。OVERLAY 技术就是说：如果两个函数处于函数调用树的不同分枝，那么它们的局部变量可以相互覆盖。

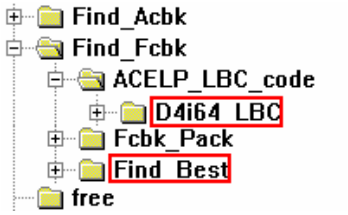

图3－6. 函数调用树示例

如图 3－6 所示为 G7231 函数调用树的一部分，可以看到 D4i64_LBC 和 Find_Best 函数处在不同的分枝。它们的局部变量数组可以分配在同一个数据区中。我们在程序中的做法是只为 Find_Best 开辟数据区，设其中一个数组变量为 Imr。Dri64_LBC 中有一个数组为 p_sign。则用.asg Imr,p_sign 将 p_sign 和 Imr 绑定，访问 p_sign 时实际访问的是 Imr 数据区。

使用 OVERLAY 时要注意：共用同一块数据区的函数都应该在不同的函数调用树分枝上。使用 OVERLAY 技术，可以减少相当的数据区。

3.6.3.2 数据区堆栈

由于 CCS 汇编编译器不支持自动识别能 OVERLAY 的函数，手动设置 OVERLAY 容易出错，并且不能找到所有能够 OVERLAY 的函数。这时可以在数据区也设置一个堆栈，姑且称之为“数据区堆栈”，当然也要有一个变量作为数据区堆栈指针，假设为 DATA_SP。当进入某函数时，根据该函数需要的数据空间大小分配一块数据区（通过增加 DATA_SP 实现），退出函数后回收分配的数据区（通过恢复 DATA_SP 为原值实现）。这种方法实际也是实现了 OVERLAY，而且能最大限度节省数据空间，但使得程序复杂化，所以 G7231 没有采用这种方法。

3.7 移植过程中易出错的地方

- 32bit 数寻址。使用 ARx+0 寻址时，要注意如果是对 32bit 的数组寻址，AR0 的值是数组索引乘以 2。
- ARx 保存负数时的问题。例如有如下语句：

```
ST      #(-1), AR1      ;AR1=0xFFFF
LDM AR1,    A           ;A = 00 0000 FFFF 即 65536
```

执行后 A 的值不是-1。DSP 认为 ARx 的值是无符号数，将 ARx 加载到累加器，累加器的值始终大于等于 0。

- 循环块最后一条指令为双字指令的处理。使用 RPTB 指令循环时，若循环块最后一条指令为双字指令则可将 lable 定位在循环后，然后用 RPTB lable-1 指令。
- 累加器的符号位。累加器 A（或者 B）的符号位是 A 的第 40bit，而不是 A 的第 32bit。所以 C＝1 时，“ROR A”指令将 C 移入 A 的第 32bit，此时不能说明 A 小于 0。
- 防止 AND 屏蔽符号位。如果用 AND #0FFFFH,16, A 来屏蔽 A 的低 16bit，则要注意，它同时屏蔽了 A 的第 40 到 33bit。正确的方法是：LD #0FFFFH,16, B；AND B, A
- 注意 Round(Acc1)的第 15 位。在 G7231 中经常有 negate(round(Acc0)) ;和 shl(round(Acc1), (Word16) 1)之类的语句。但是 negate(round(Acc0)) ;不能用 RND A 和 NEG A 实现，因为 RND 以后 A 的低 16bit 不是 0，使得 NEG 出错。为此一般采用屏蔽的方法：RND A；LD #0FFFFH, B；AND B, A；NEG A
- CALLD 后的指令，如下面指令

```
CALLD   Vec_Norm2
LD      #TmpVect,  A
LD      #0,        B
```

第二条指令根据#TmpVect 的大小可能为一个字或者双字指令。当为单字指令时，第四条指令在调用前执行，否则在调用后执行。为了防止这种二义性，要避免在延迟指令后使用 LD #K, A 等指令数不定的指令。

- 未知循环次数时使用 RPT 或 RPTB。有如下代码

```
for(i=0;i<j;i++)
        tmp[i]=0;
```

汇编实现如下：

```
    RPT     j
        STL     A,      *AR_tmp+
```

以上的代码不等价，当 j 等于 0 时，C 代码不执行循环，汇编代码执行 65536 次循环。

3.8 DSP 的 BUG 及解决方法

- RND 指令的使用

默认编译器不能识别 RND 指令，这时必须在 CCS 的 build option 中的 prossessor version 中写入 5410。但是写入 5410 时出现编译出错，提示“>>Version flag ignored 410”。实际中我们用写入 549 代替。

另外有中断的程序中不能使用 RND 指令，因为 RND 指令会修改 IFR 寄存器[18]。G7231 程序中用 SUB m_RND_val, A 代替 RND A。其中 m_RND_val 预先加载 0x8000。需要注意的是，这里不是加上 8000H，这时因为当 SXM=1 时，加 8000H 相当于加上负数 FFFF 8000。

- 循环体中读写操作数的冲突，及解决办法。

```
RPTBD   VN_NormAll
MVDK    Vect,   AR_Vect
   MPYA    *AR_Vect
```

```
VN_NormAll:
   STL    B,-4,  *AR_Vect+
```

如例子所示：如果在循环体的末尾用 ARx 写辅助寄存器指向的内容，而在循环的开头用寄存器读取数据，则有时会产生冲突。解决的方法是采用前向部分循环展开，防止在循环体的开头读寄存器。

- DST 指令不能被 RPT 循环。解决办法使用 RPTB 指令实现循环。
- ST||ADD 并行指令运行不正常，scr 的内容会加载到 dst 中去。
- CCS 编译器无法提示编译出错的一种情况为：STL B,1 *AR2。显然，该句“1”后面少了一个逗号，但是编译器有时对这种情况不会提示出错，但是编译结果是错误的。

第四章 G.723.1 算法应用实例

4.1 G.723.1 在实验箱上实现实时语音播放

实验箱采用的 AD/DA 器件为 TLC320AD50C（下简称 AD50），DSP 通过 McBSP0 和 AD50 通信。McBSP0 的同步信号由 AD50 提供，设置每帧为 1word，每个 word 为 16bit[19]。AD50 外部晶振约为 8M，设置分频数为 1024，采用 15bit 模式[20]。可以使用 PC 机的音频输出信号作为 AD50 的采样信号。

需要注意的是：由于 AD50 采用 15bit 模式，所以发送给 AD50 的语音数据的最低位必须为 0；在本实验箱上使用 AD50 时，不能随意读写 DSP 的 IO 口，因为读写 IO 口将改变 AD50 的 FC 引脚的电平。

4.1.1 FIFO 数据缓冲区和程序结构

我们设计的程序的功能是：将 AD 采集的语音数据压缩，然后马上解压缩送给 DA。实现语音实时播放时要考虑 G.723.1 压缩程序和语音采集播放程序之间的关系。语音采集和播放不能停顿，在我们的程序中将语音采集和播放放在中断服务程序中；主程序进行语音压缩和解压缩。所以程序中相当于有两个进程：ADDA 中断服务进程和主程序进程。这两个进程通过大小为 240×2word（两帧）的 WaveInBuff 和 WaveOutBuff 两个环形 FIFO 交换数据，其中 WaveInBuff 用于存放采集的语音数据，WaveOutBuff 用于存放播放的语音数据。两各进程之间，通过四个指针 WaveInBuffStart、WaveInBuffEnd、WaveOutBuffStart、WaveOutBuffEnd 传递信息。见图 4－1：

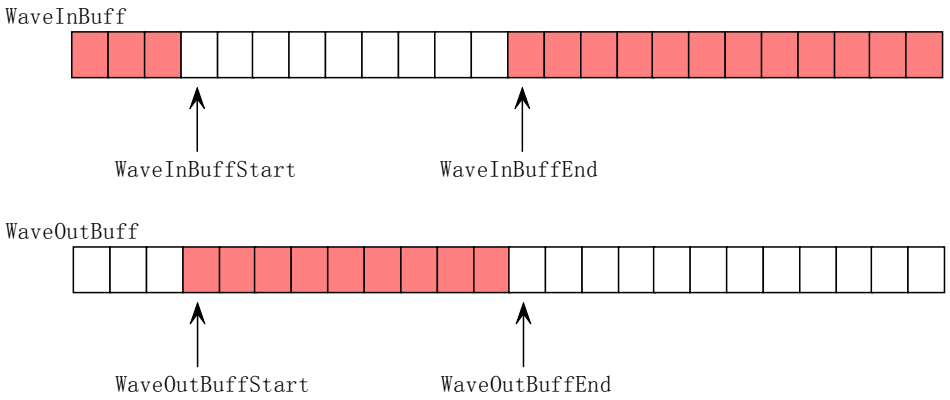

图 4－1 FIFO 环形缓冲区结构示意图

初始时指针都指向缓冲区的开始地址：

```
WaveInBuffStart = WaveInBuff;        WaveInBuffEnd = WaveInBuff;
WaveOutBuffStart = WaveOutBuff;      WaveOutBuffEnd    = WaveOutBuff;
```

- 对于 ADDA 中断服务进程：

当 AD50 采集一个数据以后，引发中断。在中断中，服务程序将数据放入 WaveInBuffStart 指向的单元，并使 WaveInBuffStart 指针循环增一。同时如果发现 WaveOutBuffStart 不等于 WaveOutBuffEnd 则说明输出缓冲区中有数据，服务程序将 WaveOutBuffStart 指向的单元的数据，送给 AD50，进行 DA 转化，并使 WaveOutBuffStart

指针循环增一。因为 AD50 采集数据是 1/8K(s)采集一次，所以 DSP 也是每隔 1/8K(s)给 AD50 一个数据，播放的语音也是 8KHz。

- 对于主程序进程：

主程序中反复检测是否 WaveInBuffStart 超前 WaveInBuffEnd 超过 240 个点，如果是说明一帧的语音数据已经准备好，主程序将调用 G7231Coder 函数对从 WaveInBuffEnd 开始的 240 个点的语音数据编码。然后将 WaveInBuffEnd 循环增加 240。然后主进程将编码结果用 G7231Decoder 函数解码，解码的数据放到 WaveOutBuffEnd 开始的 240 个 word 的缓冲区中。然后将 WaveOutBuffEnd 环形增加 240。然后又反复检测是否 WaveInBuffStart 超前 WaveInBuffEnd 超过 240 个点……。

为了实现实时性要求，要求编码一帧的时间小于 30ms，这样 ADDA 中断服务进程采集的数据才不会覆盖正在编码的语音数据。图 4－2 为编码一帧数据后的 WaveInBuff 的数据更新情况：

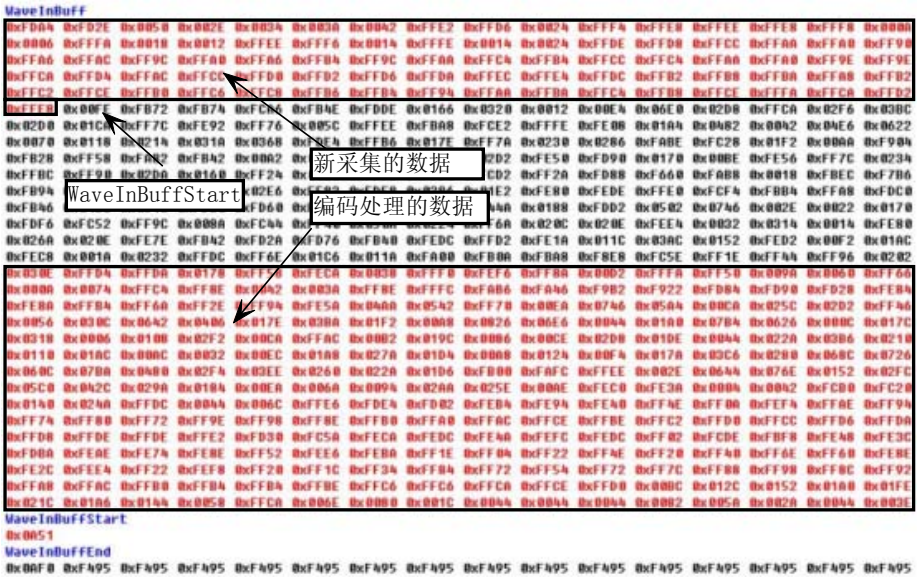

图 4－2 输入缓冲区数据更新示意图

红色区域（用方框框住的区域）表示编码前后改变的数据，它包括两部分：编码处理的数据和新采集的数据。从图中可知在编码一帧的时间内，ADDA 中断服务程序才新采集了约 1/3 帧的数据，所以不会覆盖正在编码的数据。

4.2 G.723.1 在 PC 机上实现 IPPhone 程序

该 IPPhone 程序主要用于验证 G.723.1 算法在网络环境下，通过 TCP/IP 传输语音的效果。采用 VC++6.0 编写。该程序主要包括语音采集和播放模块、TCP/IP 网络接口模块、G.723.1 算法模块。

语音采集和播放模块采用 windows 的 WaveIn 类和 WaveOut 类 API 函数编写[21]，TCP/IP 部分采用 CAsyncSocket 和 CSocket 类编写[22]，由于在 PC 机上实现，所以采用 ITU-T 的浮点 G.723.1 C 代码程序。程序结构和“实验箱上实时语音播放”程序类似。图 4－3 为程序界面：

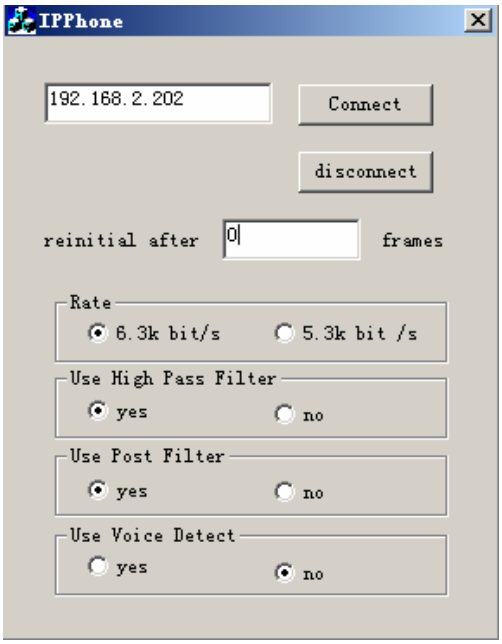

图 4－3 IPPhone 程序界面

该 IPPhone 程序在校园网内（实验室和宿舍之间）传送语音时，能实现实时语音传送，语音质量较好。

4.3 G.723.1 的 DSP 嵌入式 IPPhone 的构思

DSP 上的实时语音播放程序和 PC 机上 IPPone 程序为设计在 DSP 上的嵌入式 IPPhone 程序做好了准备。后者和前两者的主要差异是嵌入式系统的网络部分。嵌入式系统上的 TCP/IP 可以使用 ZLIP2.0，该软件包源代码可以从 http://www.zlgmcu.com 网站下载，该软件包具有较小的代码量，在 54xDSP 上编译的代码量为 4.7K，所需数据空间为 4k。这样 TCP/IP 加 G.723.1 总程序空间和数据空间可小于 32K，能够在 5409 等 DSP 上实现。

结论

论文所作的工作和成果是：

1、 本论文在前几届论文的基础上完成 G.723.1 算法在 54xDSP 上的实现，优化程度达到了一般商用 G.723.154xDSP 算法的程度。

2、 本文通过阅读 G.7231.浮点 C 代码，详细描述了 G.723.1 的实现步骤。它是目前我们能够获得的对 G.723.1 算法实现步骤最完整细致的文字描述，对研究 G.723.1 算法和类似语音算法有一定的参考价值。

3、 通过将 G.723.1 的定点 C 代码移植到 54xDSP，说明了采用从关键代码入手的全汇编优化方法的具体步骤和调试方法，从优化结果来看，和预期的结果基本相符，这为移植类似 C 代码到 54xDSP 提供了参考实例。

4、 在优化 54xDSP 汇编代码过程中，本文提出了多种优化技术，并用例子说明了，这些方法和技术的使用场合。这些方法可应用于编写高效的 54xDSP。

今后还需要做的工作有以下几部分：

1、 本论文所完成的 G..723.1 C54x 算法只能提供一路语音编解码，若需实现多路编解码需要给每一路分配一个编码状态 CodeStat 结构体和解码状态 DecodStat 结构体，这需要在程序上做一些改动。

2、 进一步压缩数据空间。从优化的结果来看数据空间比同类 G.723.1 算法要大一些。压缩数据空间可以从两方面着手：①使用本文提到的数据区堆栈方法分配数据区。②将 G.723.1 的表放在片外数据空间，在使用的时候再拷贝到内部数据空间，这可以节省约 9K 的数据空间，效果应该很明显。但是这需要增加拷贝数据所需的时间。

3、 制作嵌入式 IPPhone 的硬件电路，编写嵌入式 IPPhone 的软件。

参考文献

① http://www.protocols.com/pbook/h323.htm#H263
② ITU-T Recommendation G.723.1 DUAL RATE SPEECH CODER FOR MULTIMEDIA COMMUNICATIONS TRANSMITTING AT 5.3 AND 6.3 kbits/s,http://www.itu.int
③ ITU-T Recommendation G.723.1 Annex A,http://www.itu.int
④ ITU-T Recommendation G.723.1 Annex B,http://www.itu.int
⑤ http://www.itu.int/rec/recommendation.asp?type=folders&lang=e&parent=T-REC-G.723.1
⑥ 《DSP 集成开发和应用实例》4.4 节，电子工业出版社，2002 年 6 月，张雄伟等
⑦ TMS320VC5410 Fixed-Point Digital Signal Processor Data Manual, http://www.ti.com
⑧ SPRU132:CPU and Peripherals, p304 ,http://www.ti.com
⑨ 杨树堂，周敬利，余胜生，《G.723.1 语音编码器算法的聚类优化策略及其应用》，通信学报，2001 年 02 期
⑩ MSDN Library – July 2000，http://www.microsoft.com
11 SPRU131: CPU and Peripherals, p95, http://www.ti.com
12 SPRU131: CPU and Peripherals, p93, http://www.ti.com
13 SPRU102: Assembly Language Tools User's Guide, http://www.ti.com
14 Allocating the Frame and Using the 32-bit Memory Read Instructions, TMS320C54x Code Composer Studio Help ,http://www.ti.com
15 SPRU131: CPU and Peripherals, p130, http://www.ti.com
16 SPRU173: Applications Guide, p144, http://www.ti.com
17 SPRU172:Mnemonic Instruction Set, http://www.ti.com
18 SPRZ177a: TMS320VC5410 Digital Signal Processor Silicon Errata, p9,http://www.ti.com
19 SPRU302: Enhanced Peripherals, http://www.ti.com
20 TLC320AD50C/TLC320AD52C Data Manual, http://www.ti.com
21 MSDN Library – July 2000，Platform SDK/Grahpics and Multimedia Services /Windows Multimedia/Multimedia Reference/Multimedia Functions，http://www.microsoft.com
22 MSDN Library – July 2000，Visula C++ Documentation/Reference/Microsoft Foundation Class Library and Templates/Microsoft Foundation ClasssLibrary/Class Library Reference/CSocket，http://www.microsoft.com