Calibration 相关的基本概念

1.1 GSM 频带(GSM Band)

一个频带(Band)对应上行、下行两部份，上行即移动台发基站收，下行即基站发移动台收，在 900MHz 频带(GSM900)，上下行间隔 45MHz，在 1800MHz 频带(DCS)，上下行间隔 95MHz。一个频带占据一定的带宽，在带宽内划分了多个频道(Channel)，每个频道定义了一个对应的频道号(ARFCN – Absolute RF Channel Number)，频道间相互间隔 0.20MHz，现在比较常见的频带划分如下：

1.1.1 PGSM

下行：935.20MHz ~ 959.80MHz
上行：890.20MHz ~ 914.80MHz
其对应的频道号为 1 ~ 124，共 124 个频道，中心频道为 62；

1.1.2 EGSM

下行：925.20MHz ~ 959.80MHz
上行：880.20MHz ~ 914.80MHz
EGSM 对应频道号分为两个部份
925.20MHz ~ 934.80MHz 的频道号为 975 ~ 1023
935.00MHz ~ 959.80MHz 的频道号为 0 ~ 124
共 174 个频道，中心频道为 37，这里要注意，不存在两个中心频道，因为中心频道指的是频带中心频率所在的频道号，虽然 EGSM 频带定义的频道号不连续，但其使用的频率是连续的，从 925.20MHz 到 959.80MHz，其中心频率是 942.40MHz，此频率对应的频道号为 37；

1.1.3 DCS

下行：1805.20MHz ~ 1879.80MHz
上行：1710.20MHz ~ 1784.80MHz

其对应的频道号为 512 ~ 885，共 374 个频道，中心频道为 698；

1.1.4 PCS

下行：1930.20MHz ~ 1989.80MHz
上行：1850.20MHz ~ 1909.80MHz
其对应的频道号为 512 ~ 810，共 299 个频道，中心频率为 661；

1.1.5 RGSM

下行：921.20MHz ~ 959.80MHz
下行：876.20MHz ~ 914.80MHz
RGSM 对应频道号分为两个部份
921.20MHz ~ 934.80MHz 的频道号为 955 ~ 1023
935.00MHz ~ 959.80MHz 的频道号为 0 ~ 124
共 194 个频道，中心频道为 32；

1.1.6 GSM450

下行：460.60MHz ~ 467.40MHz
上行：415.60MHz ~ 422.40MHz
频道号为 259 ~ 293，共 35 个频道，中心频道 276；

1.1.7 GSM480

下行：489.00MHz ~ 495.80MHz
上行：444.00MHz ~ 450.80MHz
频道号为 306 ~ 340，共 35 个频道，中心频道 323；

1.1.8 GSM850

下行：869.20MHz ~ 893.80MHz
上行：824.20MHz ~ 848.80MHz
频道号为 128 ~ 251，共 124 个频道，中心频道 189；

1.1.9 GSM750

下行：777.20MHz ~ 791.80MHz
上行：732.20MHz ~ 746.80MHz
频道号为 438 ~ 511，共 74 个频道，中心频道 474

1.2 Power Level

GSM 的通讯协议中，定义了无线信号发送的功率等级(Power Level)，在实

际通讯过程中，移动台和基站的发送功率是可变的，移动台会根据实际情况来选择发送功率的大小。在 GSM 的协议中，不同频带功率等级的定义稍有不同。

表一、二、三分别为移动台在 GSM900、DCS1800 和 PCS1900 频带的功率等级的定义，在实际应用中，并不是所有定义了的功率等级都会实际使用到，比如，GSM900 通常只会使用 Level5 ~ Level19，DCS1800 频带只使用 Level0 ~ Level15，据体情况可以参考 GSM 标准的文档 GSM 05.05(Radio transmission and reception)。

表一

GSM 900

Power control level	Nominal Output power (dBm)	Tolerance (dB) for conditions	
		normal	extreme
0-2	39	±2	±2.5
3	37	±3	±4
4	35	±3	±4
5	33	±3	±4
6	31	±3	±4
7	29	±3	±4
8	27	±3	±4
9	25	±3	±4
10	23	±3	±4
11	21	±3	±4
12	19	±3	±4
13	17	±3	±4
14	15	±3	±4
15	13	±3	±4
16	11	±5	±6
17	9	±5	±6
18	7	±5	±6
19-31	5	±5	±6

表二

DCS 1 800

Power control level	Nominal Output power (dBm)	Tolerance (dB) for conditions	
		normal	extreme
29	36	±2	±2.5
30	34	±3	±4
31	32	±3	±4
0	30	±3	±4
1	28	±3	±4
2	26	±3	±4
3	24	±3	±4
4	22	±3	±4
5	20	±3	±4
6	18	±3	±4
7	16	±3	±4
8	14	±3	±4
9	12	±4	±5
10	10	±4	±5
11	8	±4	±5
12	6	±4	±5
13	4	±4	±5
14	2	±5	±6
15-28	0	±5	±6

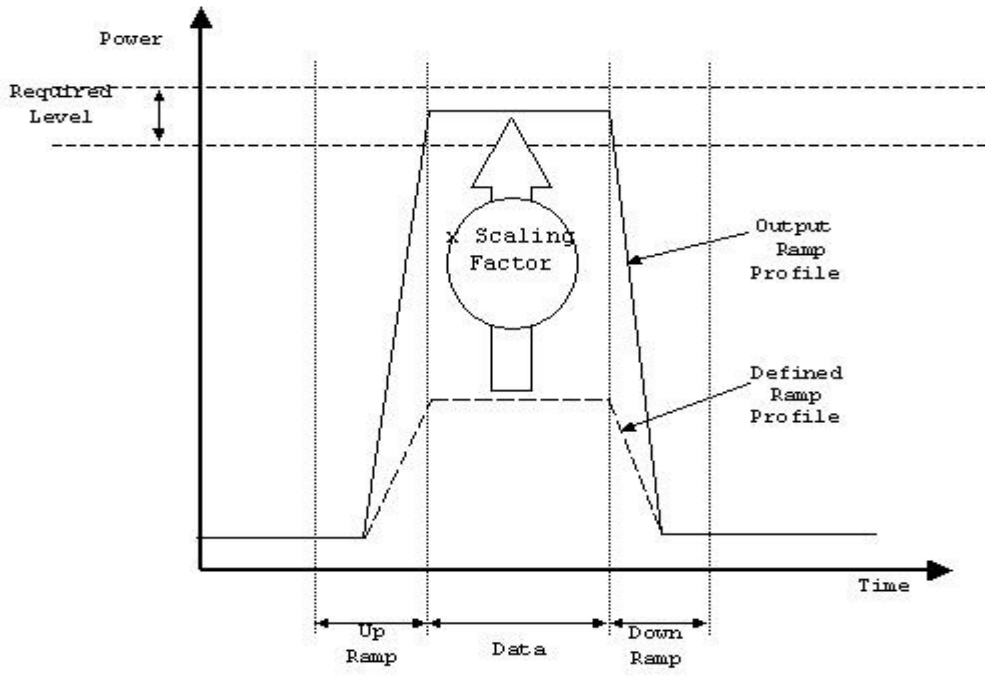
表三

PCS 1 900

Power Control Level	Output Power (dBm)	Tolerance (dB) for conditions	
		Normal	Extreme
22-29	Reserved	Reserved	Reserved
30	33	± 2 dB	± 2.5 dB
31	32	± 2 dB	± 2.5 dB
0	30	± 3 dB[1]	± 4 dB[1]
1	28	± 3 dB	± 4 dB
2	26	± 3 dB	± 4 dB
3	24	± 3 dB[1]	± 4 dB[1]
4	22	± 3 dB	± 4 dB
5	20	± 3 dB	± 4 dB
6	18	± 3 dB	± 4 dB
7	16	± 3 dB	± 4 dB
8	14	± 3 dB	± 4 dB
9	12	± 4 dB	± 5 dB
10	10	± 4 dB	± 5 dB
11	8	± 4 dB	± 5 dB
12	6	± 4 dB	± 5 dB
13	4	± 4 dB	± 5 dB
14	2	± 5 dB	± 6 dB
15	0	± 5 dB	± 6 dB
16-21	Reserved	Reserved	Reserved

Note 1: Tolerance for MS Power Classes 1 and 2 is ± 2 dB normal and ± 2.5 dB extreme at Power Control Levels 0 and 3 respectively.

1.3 Power Ramp



图一

Power Ramp 从字面上翻译为功率斜面，如图一所示，这个概念是用来反应

无线信号发送功率响应特性的。我们可以利用类似电平的上升沿和下降沿的概念来理解，电平跳变时，上升沿和下降沿不可能是绝对垂直的，而是有一个响应时间，比如一个非门，当输入电平由高变低时，输出电平由低变高用了 20ns，在这 20ns 中电平的变化就形成一个斜面。

功率斜面也分为 Up Ramp 和 Down Ramp，影响此斜面形状的参数包括反应时间、需要达到的功率等等因素。相同的时间要达到的功率等级越高，则此斜面越陡，同样的，如果是要达到相同的功率，那么反应时间越短，则斜面越陡。

功率斜面必须符合 GSM 规范的定义，图二所示为某移动台在综合测试仪上看到的测试结果，黄线为此移动台 Power Ramp 的实际值，绿色轮廓线为 GSM 规范定义值，如果黄线不超出绿色轮廓，则测试通过，如果黄线与绿线有交叉的部份，则测试失败。

(a)Up Ramp　　(b)Down Ramp

图二

1.4　DC Offset

DC Offset 译为直流偏差，当移动设备的无线信号接收机采用直接下变频方案（又称为零中频方案）时，就会存在直流偏差，它是零中频方案特有的一种干扰，是由自混频引起的。直流偏差会迭加在基带信号上，使得信噪比变差，可导致放大器饱和，无法放大有用信号。相关知识可以查阅通信射频电路参考书。

Calibration 时会调整相应参数，将 DC Offset 的影响减到最小。

2　Calibration 概述

移动设备从产在线组装完毕之后，由于元器件以及工艺流程的本身存在的误差，使得每台移动设备的电气性能不尽相同，这可能对移动设备的通信质量产生

较大的影响。为了获得最佳的通信质量，每台移动设备在出厂之前都要进行校准，调整相关参数，这个过程就是 Calibration。

为了达到产线大批量量产的要求，就需要开发一个自动 Calibration 的软件，当把移动设备放到夹具上之后，Calibration 能自动进行。此软件的目标就是在最短的时间内校正 Calibration 测试项的参数，使其达到尽可能高的精确度，从而保证无线设备获得最好的通信质量。

3 Calibration 硬件及软件架构

3.1 硬件

Calibration 的硬件包括 PC、Target、Test Set 以及互相之间的连接线。

3.1.1 PC

PC 是主控端，当然性能越强劲越好。至少要能运行 Windows2000；要有一个 Comm 埠，负责与 Target 连接；一个 USB 埠，可以通过 USB-GPIB 转接卡与 Test Set 连接；

3.1.2 Target

Target 指的就是移动设备，在本文中，Target 指的就是 TTPCom 提供的 GSM Module，Calibration 要校正的就是 Target 的各项参数。它通过 Com 埠与 PC 相连，受 PC 控制，并通过射频线与 Test Set 相连。

3.1.3 Test Set

Test Set 即综合测试仪，在本文中，Test Set 指的是 Agilent 公司的 8960 系列综合测试仪，在 Calibration 中，Test Set 作为信号发生仪和功率测试仪，它通过 USB-GPIB 转接卡与 PC 相接，接受 PC 控制；通过射频线(RF Cable)与 Target 相连。

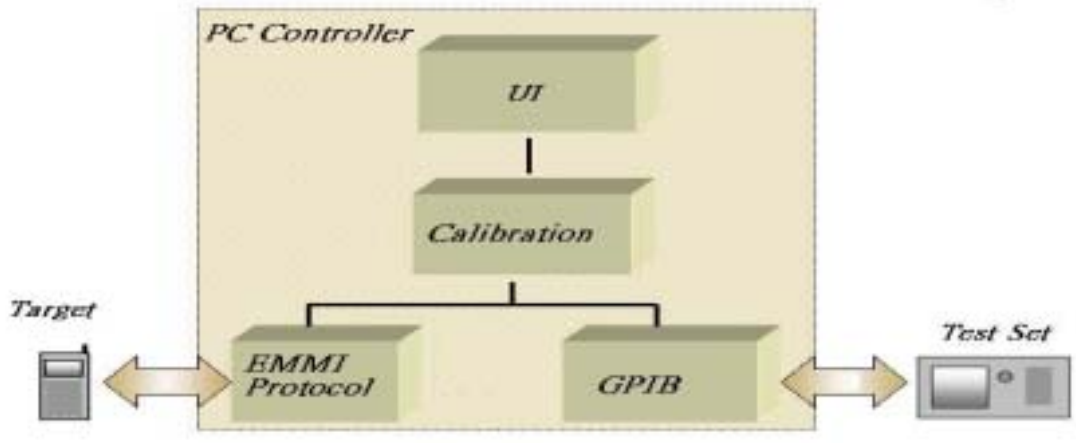
3.1.4 相互连接

Calibration 硬件之间的互相连接如下图所示

图三

3.2 软件

我们通常所说的 Calibration 软件指的是在 PC 端运行的程序，包括串口控制，GPIB 接口控制，EMMI 协议，Calibration 算法，以及程序外观接口。其框架图如图四所示



图四

3.2.1 串口控制

PC 通过 Com 埠与 Target 的 UART 相连，Com 的相关控制有 DLL 可以使用，

不需要我们自己重新编写，我们可以使用 NI 公司提供的 VISA 库实现串口控制，当然如果想自己编写也可以，请参考相关文档，这不在本文讨论范围之内。

对于 TTPCom 的 Module，通常支持多种波特率，使用 6.0 版本以上的代码，Module 的默认波特率为 115200，8 位数据位，无奇偶校验，1 位停止位，无流量控制，在公司现在的产品中，MPG、MPG+、KM276、KM289 等等，在 Calibration 时的串口通讯均采用此设置。

3.2.2 GPIB 界面

软件通过 GPIB 接口实现对 Test Set 的自动控制，大部分的测试仪都会支持 GPIB 接口，只是在具体的控制命令和控制流程上会稍有不同，我们现在使用的 Test Set 是 Agilent 公司的 8960 系列无线通讯综合测试仪，可以到下面的网址寻找相关数据 http://www.agilent.com/find/8960support/，以后我们还可能会使用其它品牌的综测仪，如 CMU200、MT8820 等等，关于综测仪具体的特性不在本文讨论范围之内，有兴趣的可以参考相关文档，在 Calibration 实现(5.2)中，我会主要讨论使用 GPIB 时需要注意的一些事项。

GPIB 埠的读写控制也是使用 NI 的 VISA 库来实现的，我们只是将控制命令通过调用库函数发送给 Test Set，在后面我会介绍 VISA 库的使用。

3.2.3 EMMI 协议

在硬件上 PC 与 Target 是通过串口相连接的，那么在软件上，PC 与 Target 之间是通过 EMMI 的接口相连接的，EMMI 的详细定义可参看 GSM04.05，但 TTPCom 对 EMMI 协议进行了简化，所以和 GSM04.05 略有不同，在 TTPCom 提供的文檔<<EMMI & Test Interface>>(u043.doc)以及<<Calibration Procedure Using CALDEV Signals In Layer1>>(cgpg051.doc)中都有较详细的描述，所以在 5.2 节中将不在重复，主要将讨论 EMMI 协议实现时需要注意的事项。

3.2.4 Calibration 过程

这是 Calibration 程序的核心部份，Calibration 主要包含以下几个项目：AFC(Auto Frequency Correction)，TPSF(Tx Power Scaling Factor)，TPFC(Tx Power Frequency Compensation)，AGC(Auto Gain Control)，DC Offset，RSSIC(RSSI Correction)，这在后面将有详细描述。

3.2.5 Calibration 用户接口

Calibration 的用户接口根据使用场合的不同分两种情况，一种是在试验室供

开发人员调试使用，另一种是供产在线使用。

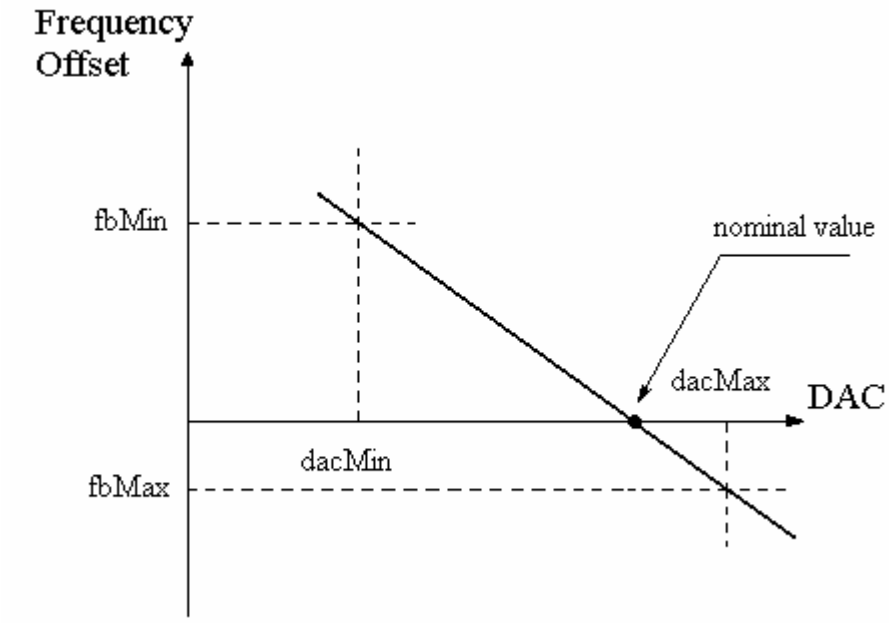
调试版本相对要复杂一些，此版本的目的在于得到尽量精确的结果，所以对于 Calibration 各项的校正会做的比较细致一些，比如允许的误差范围设置的小一点，各频带的每个频道都进行校正，并提供设置接口，让开发人员可以自己设置 Calibration 的各项参数。由于 Calibration 得到的校正值最后都是保存在 Target 的 NVRAM 中，所以最好还可以提供手动修改接口，让开发人员可以直接修改 Target NVRAM 中的数值。

产线版本相对于调试版本则显的粗糙一些，因为在产在线量产的时候，时间是重要因素，此版本的目的是在误差允许范围内，用最短时间完成 Calibration，以达到最高的工作效率。所以误差可以大一点(但必须在标准定义的范围之内)，也不必每个频道都进行校正，可以每次跳过几个频道，被跳过的频道可以采用插值法计算得出，在显示上只需要显示 Pass 还是 Failure。

4 Calibration 测试项

4.1 AFC(Auto Frequency Correction)

AFC 主要是为了保证 Target 的时钟频率和网络正确同步。我们知道 DAC(数模转换器)和 Frequency Offset(时钟频率偏移)有近似线性的关系，如下图



图五

DAC-Frequency Offset 曲线由两个要素决定，一个是基准值(nominal value)，一个是曲线的斜率(slope)。所谓的基准值指的就是当 Frequency Offset 值为零时 DAC 的值，如果还知道 Slope 的值，就可以根据任何一个 Frequency Offset 计算

出对应的 DAC 值了。

图五和 TTPCom 提供的文檔(cgpg051.doc)中所示稍有不同，cgpg051 中的曲线是正斜率的，而在实际调试的过程中发现，Frequency Offset 是随 DAC 的增大而减小的，所以应该是一条负斜率曲线。

DAC 的值域和所用的芯片有关，TTPCom 提供的 Module 的 DAC 是用一个 13 位的寄存器实现的，所以 DAC 的最小值为 0，最大值为 8191($2^{13}-1$)。

4.2 TPSF(Tx Power Scaling Factor)

对 TPSF 进行校正是为了让 Target 的发射功率能够满足 GSM05.05 中对各个功率等级的定义。TPSF 是与 Power Ramp 紧密相关的一个参数，根据图一所示，我们可以形象的了解 Scaling Factor 的含意。由于各种客观原因，导致 Target 的发送功率等级达不到 GSM 标准的要求，所以我们对 Scaling Factor 进行校正，然后将校正值保存在 Target 的 NVRAM 中，Target 在实际运行的时候，将从 NVRAM 中取得校正值对发送功率进行调整，这样就可以达到标准的要求了。

TPSF 的校正主要就是为了校正 Power Ramp，使其符合标准，大家可能会有一个疑问，在前面介绍 Power Ramp 的时候，说过 Power Ramp 还和反应时间等等因素有关，那为什么只对 Scaling Factor 进行校正？

其实在 TTPCom 提供的代码中，Power Ramp 是由两个因素决定的，一个是 Scaling Factor，另一个称之为 Ramp Profile。查看 TTPCom 的源代码，我们可以发现 Ramp Profile 定义的是一串数字，例如：

```
#define CAL_CFPOWERRAMP_GSM_LEVEL5      \
    70, 70, 70, 70, 70, 70, 75, 75, \
    75, 90,130,140,180,218,232,255, \
    255,242,218,185,140,130,90,80, \
    80,80,80,70,60,51,10,0
```

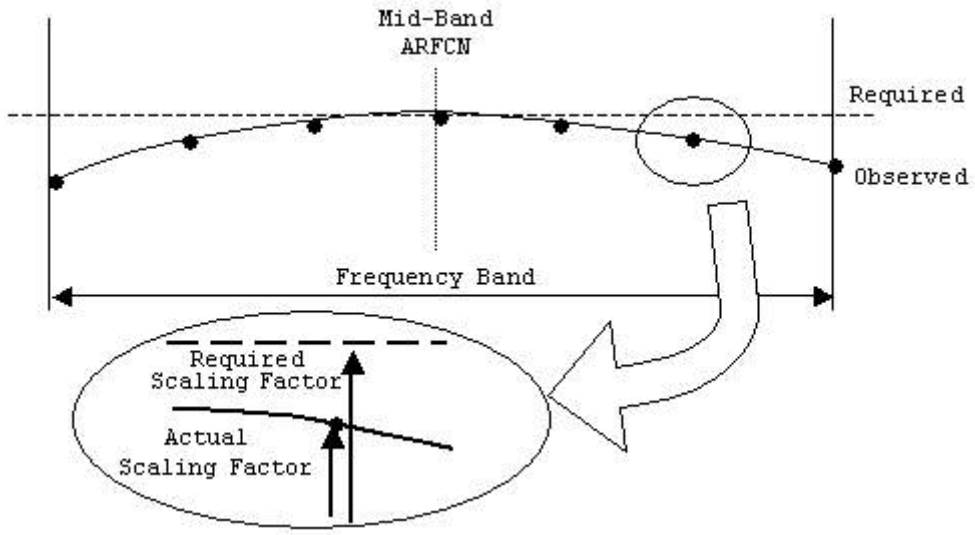
这是 GSM 频带 Power Control Level 5 的 Ramp Profile，相同的我们还可以找到各个 Power Control Level 的 Ramp Profile，以及其它频带各个 Power Control Level 的 Ramp Profile。仔细观察 Ramp Profile 的数值，对照 Power Ramp 的图形，你会发现其实 Ramp Profile 就是对 Power Ramp 的形状的描述，一共 32 个值，前 16 个值对应 Up Ramp，后 16 个值对应 Down Ramp。那为什么不对 Ramp Profile 进行校正呢？这主要出于两方面考虑，第一、校正过程比较麻烦，每个 Power Level 都有 32 个值要校正，数据量大，过程长，而且不太容易用程序去判断一个值是否校正成功；第二、TTPCom 并没有将此数值放入 NVRAM，而是固定在代码中的，每次修改 Ramp Profile 的值，必须重新编译才能起作用。如果以后能将 Ramp Profile 也保存在 NVRAM 中，对其进行校正也是有可能的。另外值得一提的是，最近 TTPCom 对产品进行了一些改进，硬件变化比较大，更换一些芯片，在更换了芯片之后，同一个频带各个功率等级均使用同一个 Ramp Profile，不同

频带的 Ramp Profile 稍有不同。

所以在和 Power Ramp 相关的两个因素中，我们目前只对 Scaling Factor 进行校正。

4.3 TPFC(Tx Power Frequency Compensation)

TPFC 是对发送功率进行频率补偿，一个频带占据一定的带宽，在其带宽内，每间隔 200kHz 划分为一个频道，随着频道的变化，发送功率会产生一些偏差，如图六所示。



图六

我们虽然进行了 TPSF 的校正，但当 Target 在使用不同的的频道时，可能某些频道的偏差会使得其发送功率无法达到 GSM05.05 的要求，所以我们必须对每个频道再进行 TPFC 的校正，使得 Target 的发送功率在整个带宽范围内都能达到 GSM05.05 的要求。

为了提高效率，在实际校正时，并不会每个频道都进行校正，通常都是隔 4 个 ARFCN 校正一下，跳过的 ARFCN 可以通过插值法近似的算出。

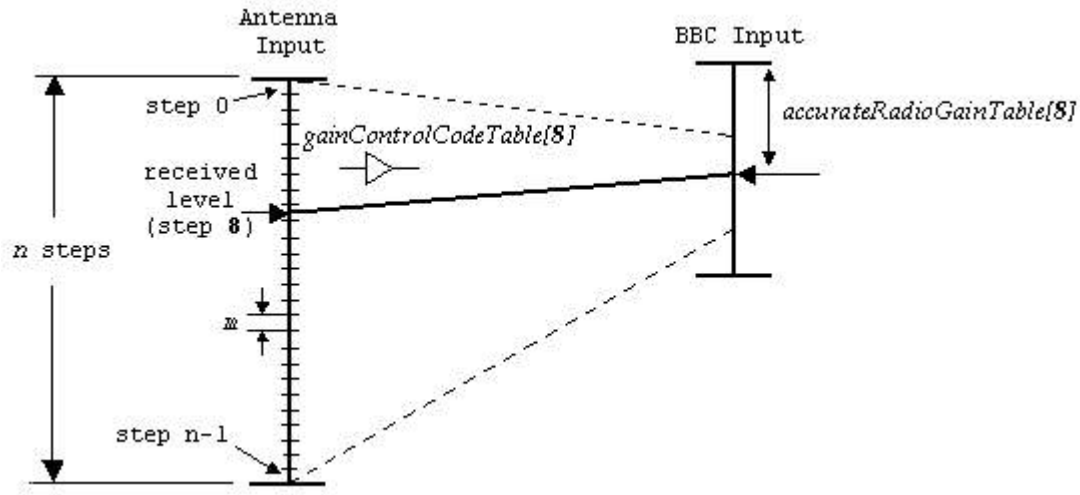
4.4 AGC(Auto Gain Control)

AGC 校正是为了保证输入信号保持在 BBC(Base Band Converter)可用范围之内。BBC 的响应范围比天线输入端的输入范围要小的多，所以我们必须将天线的输入信号映像到 BBC 的可用范围之内。

映像的方法是将天线的输入范围划分为 n 个值，每个值间隔为 m，我们称 m 为步进值(step size)，设计一个映像表 Gain Table，映像表包含 n 的单元，每个单元对应一个天线到 BBC 的映像值。

TTPCom 将 n 定义为 26(CF_RADIO_GAIN_TABLE_SIZE)，并且设计了两个

映像表，gainControlCodeTable 和 accurateRadioGainTable，在 TTPCom 的源代码中可以找到这两个表的默认值，gainControlCodeTable 的值用于设置 Radio 的 Gain Word，以使 BBC 获得最佳的输入，accurateRadioGainTable 用于计算 BBC 接收的精确值，如图七所示



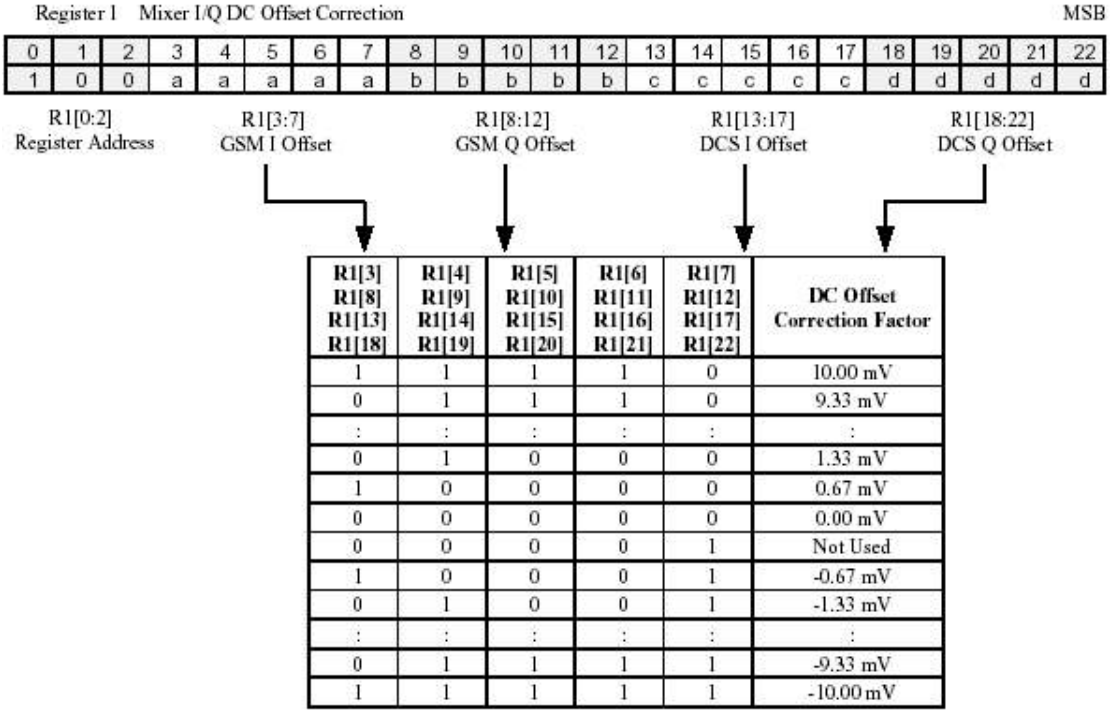
图七

在 TTPCom 的文文件中，把天线输入定为-10dBm ~ -110dBm，步进值为 4dBm，而实际 Calibration 的时候，我们使用的值是-21dBm ~ -96dBm，步进值为 3dBm，到底该使用哪个值呢？那就要根据 Target 中负责 Gain Control 的硬件性能来决定了，查阅硬件 Spec.可以得知，目前我们所使用的芯片组，当 Gain Control 寄存器的值增减 1 的时候，gain 变化为 3dBm，也就是说实际可获得的 gain 的值总是 3 的倍数，所以我们采用 3dBm 的步进值，而不用 4dBm，Gain Table 一共有 26 个值，那么输入可变范围就是 25*3= 75dBm，再根据所使用的天线的输入特性，我们将输入范围定为-21dBm ~ -96dBm。

4.5 DCO(DC Offset)

DC Offset 的校正是为了消除直流偏置，在 TTPCom 的文档中对此项校正未作介绍，要在硬件 Spec.中才能找到相关资料。通常将 DC Offset 向量化，可分为 I Offset 和 Q Offset，分别表示两个向量方向上的偏置，在校正的时候分别对 I/Q Offset 进行校正。

在 TTPCom 硬件改版之前，信号收发器使用的是 AD6523，AD6523 的内部的 Register1 是个 23 位寄存器，用于 DC Offset 的校正，如图八所示，GSM I/Q Offset 和 DCS I/Q Offset 共四个值，每个值占 5 个 bits，最高一个 bit 是符号位，四个值合并成一个 23 位的值。



R1[3] R1[8] R1[13] R1[18]	R1[4] R1[9] R1[14] R1[19]	R1[5] R1[10] R1[15] R1[20]	R1[6] R1[11] R1[16] R1[21]	R1[7] R1[12] R1[17] R1[22]	DC Offset Correction Factor
1	1	1	1	0	10.00 mV
0	1	1	1	0	9.33 mV
:	:	:	:	:	:
0	1	0	0	0	1.33 mV
1	0	0	0	0	0.67 mV
0	0	0	0	0	0.00 mV
0	0	0	0	1	Not Used
1	0	0	0	1	-0.67 mV
0	1	0	0	1	-1.33 mV
:	:	:	:	:	:
0	1	1	1	1	-9.33 mV
1	1	1	1	1	-10.00 mV

图八

在 TTPCom 硬件改版之后，AD6523 被 AD6539 所取代(AD6539 是一个集成化的芯片，它不光具有 AD6523 的收发器功能，还集成了串口，电源管理等功能，而且 AD6523 只支持 GSM 和 DCS 双频，而 AD6539 支持 GSM、DCS、PCS 以及 GSM850 四频)，AD6539 使用 Register2 保存 GSM I/Q Offset，Register3 保存 DCS I/Q Offset，与 AD6523 稍有不同。在软件上，使用 AD6523 时，DC Offset 使用一个 32 位的 Double Word 来表示，不过此 DW 只使用了低 23 位，而使用 AD6539 时，用了一个数据结构来表示 DC Offset，在 TTPCom 的源代码中可以找到如下定义

```
typedef struct DcOffsetValuesTag
{
    Int32    valGSM;
    Int32    valDCS;
    Int32    valPCS;
    Int32    valGSM850;
    Int32    valDcOffsetDAC2;
} DcOffsetValues;
```

其中每个值的 bit7 ~ bit14 表示对应频带的 I Offset，bit15 ~ bit22 表示对应频带的 Q Offset。不过有另外一个好消息，AD6539 有自动进行 DC Offset 校正的功能，只要开启了此功能，在每次开机的时候，AD6539 会自动校正 DC Offset，那么我们就可以把 DC Offset 的校正从 Calibration 中省略掉了。那么如何开启 DC Offset 自动校正功能呢？其实也是很简单，查看 AD6539 的 Spec，可以看到在 AD6539

的 Register0 中，bit8 是 BB_DAC_CAL_EN，bit9 是 ST_BB_CAL，只要把这两个 bits 设为 1 就行了，在头文件 ootvtypical.h 中我们可以找到一个宏

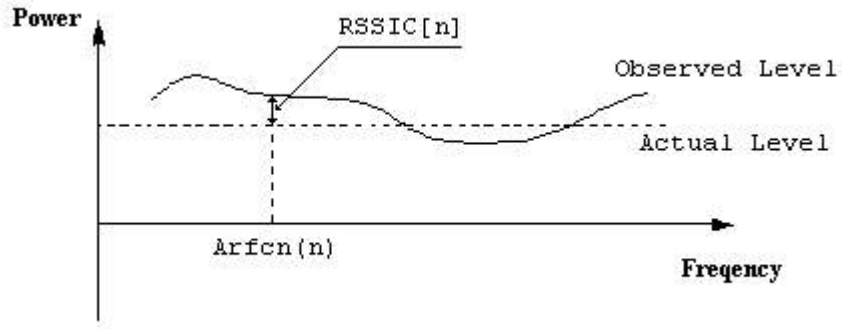
#define INITIAL_PROG_WORD_REG0_BB_FILTER_CAL　0x87DF80

在开机的时候，这个值会被用来初使化 Register0，只要保证这个值的 bit7 和 bit8 是 1，开机后就会开启 DC Offset 自动校正功能了。

我们现在就是使用了 AD6539 自动校正功能，所以在对使用了 AD6539 的项目进行 Calibration 时，DC Offset 是被屏避的。如果不使用自动校正功能，想要自己对 DC Offset 进行 Calibration，也是可以的，只要将 Register0 的 bit7 和 bit8 设为 0 即可。自行校正时，AD6539 和 AD6523 校正的流程和算法都是一样的，只是存储的数据结构稍有不同，这在 5.7 节中将详细论述。

4.6　RSSIC(RSSI Correction)

RSSIC 校正是为了保证 Target 在整个支持的频率带宽范围内接收信号的准确性。Target 在某个频带接收一个固定功率的信号时，不同的频道，所接收到的信号强度会稍有偏差，如图九所示



图九

通常在 Calibration 的程序中，我们给 Test Set 设置一个中等强度的发送功率值作为 Actual Level，比如-62dBm，然后测量在不同 Arfcn 上 Target 所接收到的信号强度，这个值在图九中称为 Observed Level，可以看到随着频率的改变，Observed Level 是一条不规则的曲线，在某个特定的 Arfcn(n)，测得的 Observed Level 与 Actual Level 的差值就是此频道的 RSSI Correction Value。

TTPCom 对应每个频带，都使用一个 SignedInt8 类型的数组来保存该频带每个 Arfcn 的 RSSIC value，查寻 TTPCom 的源代码，我们可以在 ootvcal.c 中找到这些表的默认值定义：

```
static const SignedInt8 gsmRssiCorrectionTable [NUM_PGSM_CHANNELS] ={ 0 } ;
static const SignedInt8 egsmRssiCorrectionTable [NUM_EXTRA_EGSM_CHANNELS] ={ 0 } ;
static const SignedInt8 dcsRssiCorrectionTable [NUM_DCS_CHANNELS] ={ 0 } ;
static const SignedInt8 pcsRssiCorrectionTable [NUM_PCS_CHANNELS] ={ 0 } ;
static const SignedInt8 gsm850RssiCorrectionTable [NUM_GSM850_CHANNELS] ={ 0 } ;
```

所有的默认值均设为零，每张表的元素个数为其对应频带的频道数，但egsmRssiCorrectionTable 稍有不同，如 1.1.2 节所述，在频带的概念上，EGSM 包含了 PGSM，所以在这张表中只包含了在 PGSM 基础上扩展的频道数，即 975 ~ 1023 频道以及 0 频道。

如图九所示，由于 Observed Level 是条不规则曲线，所以如果 Calibration 想要做的精确些，就应该对每个频道都进行校正，但这是以时间为代价的，所以 Calibration 的调试版可以每个频道都校正，但产线版本通常采用间隔的方法，比如每间隔 10 个频道校正一次，被跳过的这 10 个频道采用插值法计算得出。

5 Calibration 实现

在具体实现的讨论中，我们以下列环境为例

- PC：OS Win2000/NT，development tools VC++6.0
- Target：TTPCom GSM Module ver6.1
- Test Set：Agilent 8960 series wireless test set

5.1 VISA 库的使用

VISA 全称 Virtual Instrument Software Architecture，它是由 VXI*plug&play* 体系联盟提供的一种多方 I/O 软件标准。它为高级系统软件提供了一个可共享的基础层，我们使用的 NI-VISA 是由 National Instruments 公司提供的 VISA 解决方案。

图十

我们首先要安装 VISA 的驱动程序，如果你在公司的局域网内，你可以到目录\\inc-app\app\DRIVER\GPIB-NI4882\NI-VISA 找到安装程序，你也可以通过互联网到 http://www.ni.com/visa 网站上去下载最新的驱动。驱动程序安装好之后在你的开始菜单可以找到相关程序和 NI-VISA 的帮助文档，如图十所示。NI-VISA 提供的帮助文档对其使用有非常详细的描述，并提供了大量的源码可供参考，请仔细阅读这些文档。

如果操作系统 Window2000/NT 安装在 C:\WINNT，并且在安装 NI-VISA 时采用默认的安装路径，那么 NI-VISA 的动态链接库文件已经复制到你的系统目录(C:\WINNT\system32)中，同时在目录 C:\VXIpnp\WinNT\include 下可以找到两个 Head 檔 visatype.h 和 visa.h，visatype.h 定义了数据类型，visa.h 则包括了库函数原型的声明，在使用到 VISA 库的程序中需要包含这两个 Head 檔

```
#include <visatype.h>
#include <visa.h>
```

在目录 C:\VXIpnp\WinNT\lib\msc 下还有一个 lib 文件 visa32.lib，当你 Link 你的代码时会用到这个 lib，所以你必须把它也加入到你的 VC++工程中。

5.2 串口及 GPIB 控制

我们使用 NI-VISA 提供的库函数来控制串口及 GPIB 埠，在 NI-VISA 的帮助文档中都有详细说明，这里只讨论一下需要注意的事项。

在任何 VISA 操作之前，我们必须先初使化 VISA 系统，这通过调用下面的函数来实现，此函数总是第一个被调用的 VISA 函数

```
status = viOpenDefaultRM( &visDef ) ;
```

此函数可以得到一个 Session visDef(会话)，返回值 status 将告述你操作是否成功，当你的程序全部结束的时候，必须调用 viClose(visDef)来关闭此 Session。VISA 的所有函数的返回值都是 ViStatus 类型的值，用来表示该函数执行的状态，如果返回的是 VI_SUCCESS 表示成功执行，但要注意并不是只有 VI_SUCCESS 表示成功，以下宏定义都表示成功执行

```
#define VI_SUCCESS_EVENT_EN          (0x3FFF0002L)
#define VI_SUCCESS_EVENT_DIS         (0x3FFF0003L)
#define VI_SUCCESS_QUEUE_EMPTY       (0x3FFF0004L)
#define VI_SUCCESS_TERM_CHAR         (0x3FFF0005L)
#define VI_SUCCESS_MAX_CNT           (0x3FFF0006L)
#define VI_SUCCESS_DEV_NPRESENT      (0x3FFF007DL)
#define VI_SUCCESS_TRIG_MAPPED       (0x3FFF007EL)
#define VI_SUCCESS_QUEUE_NEMPTY      (0x3FFF0080L)
#define VI_SUCCESS_NCHAIN            (0x3FFF0098L)
#define VI_SUCCESS_NESTED_SHARED     (0x3FFF0099L)
```

```
#define VI_SUCCESS_NESTED_EXCLUSIVE  (0x3FFF009AL)
#define VI_SUCCESS_SYNC              (0x3FFF009BL)
```

举例来说，当我们调用函数

```
status = viRead( vis, buf, 10, &ret ) ;
```

来从 Session vis 读取 10 bytes 到 buf 中时，如果成功读取了 10 bytes，那么函数将会返回 VI_SUCCESS_MAX_CNT，而不是 VI_SUCCESS，这表示读出的数据长度达到指定的最大值，这时如果代码写成如下形式将是错误的

```
if( viRead(vis, buf, 10, &ret) != VI_SUCCESS )
    return VI_FALSE ;
```

为了保证函数的正确使用，在使用一个函数之前请先到 NI-VISA 的帮助文件中察看该函数的说明。

在初始化成功后就可以调用 viOpen()来开启相应的设备了，对于一个 GPIB 设备，通常使用首地址(Primary Address)和副地址(Secondary Address)加以标示，假定一个 GPIB 设备首地址为 14，副地址为 0，则如下调用

```
viOpen( visDef, “GPIB0::14::0::INSTR”, VI_NULL, 2000, &visGPIB ) ;
```

如果调用成功就可以获得一个 GPIB 设备的 Session visGPIB，然后就可以调用读写函数向此 GPIB 设备发送命令了。由于 VISA 库的使用，使得 GPIB 埠的控制相对简化很多，所以最需要了解的是测试仪的各项命令，测试仪的命令很多，像 Agilent 8960 系列，就有近百条命令，每个命令还可以带多项参数，从中选择我们所要使用的命令还是比较耗费时间的。同时考虑到以后还有可能使用其它测试仪，为了提高代码的易修改性，我们可以将使用的命令定义成宏，例如

```
#define TestSet_Reset( vis ) viPrintf( vis, "*RST\n" )
#define TestSet_GetIDN( vis, buf, len, pret ) \
    { \
        viPrintf( vis, "*IDN?\n" ) ; \
        viRead( vis, (buf), (len), (pret) ) ; \
    }
```

如果更换测试仪导致命令格式发生变化，只需要修改对应宏就可以了。

VISA 同样可以控制串口，只是开启串口设备稍微复杂一点，开启之后要设置串口的各项属性。假设我们要开启 Comm1，则如下调用

```
viOpen( visDef, “ASRL1::INSTR”, VI_NULL, 2000, &visComm ) ;
```

如果开启成功，接着就要调用 viSetAttribute()来设置 Session visComm 的波特率、数据位、奇偶位、停止位、流控等属性，需要设置的属性宏定义如下

```
#define VI_ATTR_ASRL_BAUD            (0x3FFF0021UL)
#define VI_ATTR_ASRL_DATA_BITS       (0x3FFF0022UL)
#define VI_ATTR_ASRL_PARITY          (0x3FFF0023UL)
#define VI_ATTR_ASRL_STOP_BITS       (0x3FFF0024UL)
#define VI_ATTR_ASRL_FLOW_CNTRL      (0x3FFF0025UL)
```

#define VI_ATTR_ASRL_END_IN (0x3FFF00B3UL)

上述属性中最后一项 VI_ATTR_ASRL_END_IN 比较容易被忽略，但却是非常重要的属性，因为在 NI-VISA 中定义了一个终止符，默认的终止符是‘\n’(0x0a)，当执行一个输入操作的时候，如果遇到这个终止符，那么这次操作就会自动结束。由于我们在串口上传输的是 EMMI 数据包，都是二进制数据，如果不修改此属性，那么每次调用 viRead()读取指定长度的数据的时候，如果数据中正好包含了二进制数 0x0a，将会导致操作中断，数据则无法完整读取，所以必须将此属性设为 VI_ASRL_END_NONE，如下所示

viSetAttribute(visComm, VI_ATTR_ASRL_END_IN, VI_ASRL_END_NONE) ;

对应的还有属性 VI_ATTR_ASRL_END_OUT，用于控制输出的终止符，不过由于输出终止符属性的默认设置就是 VI_ASRL_END_NONE，所以无须修改。

VISA 库提供了几个简单可靠的输入输出函数，最常用的 4 个输入输出还数如下所列：

ViStatus viRead(ViSession vi, ViPBuf buf, ViUInt32 cnt, ViPUInt32 retCnt);

ViStatus viWrite(ViSession vi, ViBuf buf, ViUInt32 cnt, ViPUInt32 retCnt);

ViStatus viPrintf(ViSession vi, ViString writeFmt, ...);

ViStatus viScanf(ViSession vi, ViString readFmt, ...);

其中 viRead()和 viWrite()是直接读写，viPrintf()和 viScanf()是格式化读写，具体如何使用请参考 NT-VISA 的帮助文档。

通常在读写操作之前会想要清除一下设备的输入输出，以保证读写数据的对应，我们可以看到 NI-VISA 提供了一个 viClear()函数

ViStatus viClear(ViSession vi);

但此函数功能有限，对于 GPIB 设备和符合 IEEE488.1 标准的设备可以使用此函数，但对于其它设备此函数是无效的，例如使用此函数想清除普通串口 Session，都会返回 VI_ERROR_INV_SETUP。那么用什么函数来清除串口 Session 呢？NI-VISA 提供了另外一个 viFlush()函数

ViStatus viFlush(ViSession vi, ViUInt16 mask);

如果将参数 mask 设为 VI_READ_BUF_DISCARD 可以清除输入 Buffer，如果设为 VI_WRITE_BUF_DISCARD 则清除输出 Buffer。

当一个 Session 不再使用的时候，必须调用 viClose()来结束这个 Session，释放其所占用的资源，所以 viClose()和 viOpen()必定是一一对应的，这一点一定要注意。

5.3 EMMI 协议

EMMI 协议在 TTPCom 的文文件中有较为详细的描述，请参考 3.2.3 节提到的两篇文档，在这里主要讨论一下在实际编程时需要注意的问题。

首先，我们需要注意一下编译环境的问题，TTPCom 的程序使用的是 ARM

的编译器，而 PC 上的控制程序的开发环境是 VC++，ARM 的编译器默认采用 4 字节对齐，所以我们也需要将 VC++的编译环境设为 4 字节对齐。这两个编译器还有一个很大的差别体现在对枚举类型的长度定义。ARM 的编译器根据枚举类型的值域来确定其长度，如表四所示

表四

值域	类型
0 < =x <256 且无负数	unsigned char(Int8)
-127<= x <128 且有负数	signed char(SignedInt8)
256<=x<65536 且无负数	unsigned short(Int16)
-32767<= x<32768 且有负数	signed short(SignedInt16)
Others value	signed int(Int32)

而 VC++把枚举类型统一定义为 int 类型(在 win2000 系统下就是 Int32)，这会导致什么问题呢？让我们来看一个例子：

我们要从 PC 端发送信号 SIG_CDMG_RX_CONTROL_REQ 到 Target 端，此信号的参数用数据结构 CalDevGsmRxControlReq 来保存，在 TTPCom 源码中，我们找到其结构定义为

```
typedef struct CalDevGsmRxControlTag
{
   Int16          commandRef;
   TaskId         taskId;
   RxControlMode  command;
   Int32          parameter;
   Int8           index;
} CalDevGsmRxControlReq;
```

此结构中的 taskId 的 command 都是枚举类型，定义如下

```
typedef enum TaskIdTag
{
#include <kisystsk.h>
   NUMBER_OF_TASKS,
   UNKNOWN_TASK_ID      = 0xFF,
   UNKNOWN_TASK         = UNKNOWN_TASK_ID
} TaskId;

typedef enum RxControlModeTag
{
      RX_POWER_UP_RADIO,
      RX_POWER_DOWN_RADIO,
      RX_RECEIVER_ON,
```

```
        RX_RECEIVER_OFF,
        RX_SEND_SYNTH_VALUE,
        RX_SET_STATIC_SYNTH_VALUE
    } RxControlMode;
```

可以看到，这两个枚举类型的值域都不会超过 255，所以在 Target 上，这两个枚举量被当作 Int8 来使用，而在 PC 上的 VC++环境中，这两个域都被当作 Int32 来使用，所以 CalDevGsmRxControlReq 的定义等同于下表所示

Target	PC
typedef struct CalDevGsmRxControlTag { Int16 commandRef; **Int8 taskId;** **Int8 command;** Int32 parameter; Int8 index; } CalDevGsmRxControlReq;	typedef struct CalDevGsmRxControlTag { Int16 commandRef; **Int32 taskId;** **Int32 command;** Int32 parameter; Int8 index; } CalDevGsmRxControlReq;

黑体部份标出了两个编译环境所造成的差别。

如果我们在 PC 端定义了一个此结构类型的变量 A

```
CalDevGsmRxControlReq    A ;
```

在 Target 端定义同样的结构变量 B

```
CalDevGsmRxControlReq    B ;
```

结构变量 A 在 PC 端经过初使化，再封装到 EMMI 协议的数据包中，通过串口传送到 Target 端，Target 按照 EMMI 协议解析数据包，把结构 A 提取出来，复制到 Target 定义的结构变量 B 中，这类似于进行了一个 memory copy 的动作

```
memcpy( B, A, sizeof(CalDevGsmRxControlReq) ) ;
```

但由于 A 和 B 内部数据类型不统一，就会导致数据发生偏移，造成的结果就是 B.taskId != A.taskId，B.command !=A.command，以及后面的域 parameter，index 全都对不上了。那么几乎所有包含了枚举变量的结构，从 A 传到 B，或从 B 传到 A 之后，就变得无法辨识了。

由于不方便修改 Target 端的源代码，所以要解决这个问题，在编写 PC 端程序的时候就要注意，把包含枚举变量的结构改写成符合 Target 结构的类型，在上例中，把 PC 端的数据结构定义做如下修改就可以了

```
typedef struct CalDevGsmRxControlTag
{
    Int16               commandRef;
//  TaskId              taskId;
    Int8                taskId ;
//  RxControlMode       command;
```

```
    Int8                command ;
    Int32               parameter;
    Int8                index;
} CalDevGsmRxControlReq;
```

根据实际经验，在 Target 源代码中所以涉及的枚举类型，除了 SignalId (其定义在 pssignal.h 中)是 Int16 的以外，其余的都是 Int8 类型的。

另外一个问题是超长数据的问题，这个问题在两个地方都会遇到，一个是在 EMMI 的协议部份，根据 EMMI 的 Spec，一个 EMMI 数据包最多只能包含 255 bytes 的数据，如果有超过 255 bytes 的数据，就要分成两个 EMMI 数据包传送，可以参考 TTPCom 提供的文檔<<EMMI&TEST INTERFACE>>(u043.doc)，里面有详细的解决方案。另一个是超长信号的问题，举例说明，当我们要往 NVRAM 写数据时，我们会发送信号 SIG_L1AL_NVRAM_WRITE_CAL_REQ 给 Target，此信号所带数据结构定义如下

```
typedef struct L1AlNvramWriteCalReqTag
{
    TaskId    taskId;
    Int16     commandRef ;
    Int8      nvControl ;
    Int8      calDataId ;
    NvData    nvData;
}L1AlNvramWriteCalReq;
```

要写入 NVRAM 的数据存放在 nvData，nvData 的数据结构定义如下

```
typedef struct NvDataTag
{
    DataLength    numBytes;
    Int8          *dataBuffer_p;
    Int8          data[NVDATASIZE];
    DataLength    offset;
}NvData;
```

在这个结构中，numBytes 标示本信号所带数据长度；Int8 型的数组 data 存放数据，长度定义为 256(#define NVDATASIZE 256)，所以一个写信号只能捎带 256 个 bytes 的数据，数据量超过 256 bytes 就要分成两个信号传输；offset 标示本信号所带数据在整个数据中的位置，例如有 300 bytes 数据要写入 NVRAM，超出 data 的长度，那么肯定要分成两个信号写入，如果第一个信号先传输 200 bytes，那么第一个信号中 numBytes=200，data[0] ~ data[199]=数据，offset=0；第二个信号传输剩余的 100 bytes，那么第二个信号中 numBytes=100，data[0]~data[99]=数据，offset=200。

我们来看一个实际应用的例子，已知 DCS 频带的 RSSIC 数据的长度为 374

bytes，保存在数组 dcsRssiCorrectionTable 中，其定义为

SignedInt8 dcsRssiCorrectionTable[NUM_DCS_CHANNELS] ;

如果我们需要将这个数组从 PC 端传到 Target 端，并写入 Target 的 NVRAM 中，需要几个 EMMI 数据包才能完成此项操作？答案是 4 个。首先我们需要使用信号 SIG_L1AL_NVRAM_WRITE_CAL_REQ 来实现写入 NVRAM 的动作，一个写信号最多容纳 256 个数据，所以需要 2 个信号，如下定义两个信号

L1AlNvramWriteCalReq stData1 ;

L1AlNvramWriteCalReq stData2 ;

第一个信号传送 256 bytes，第二个信号传送剩余的 118 bytes，如下赋值

```
stData1.tasked = EMMI_LOW_PRI_TASK_ID ;
stData1.commandRef = 0x1213 ;
stData1.nvControl = NV_CONT_EXT_DATA ;
stData1.calDataId = CAL_CFDCSRSSICORRECTIONTABLE ;
stData1.nvData.numBytes = 256 ;
stData1.nvData.dataBuffer_p = 0 ;
memcpy( stData1.nvData.data, dcsRssiCorrectionTable, NVDATASIZE ) ;
stData1.nvData.offset = 0 ;

stData2.tasked = EMMI_LOW_PRI_TASK_ID ;
stData2.commandRef = 0x1213 ;
stData2.nvControl = NV_CONT_EXT_DATA ;
stData2.calDataId = CAL_CFDCSRSSICORRECTIONTABLE ;
stData2.nvData.numBytes = 118 ;
stData2.nvData.dataBuffer_p = 0 ;
memcpy( stData2.nvData.data, &dcsRssiCorrectionTable[256], 118 ) ;
stData2.nvData.offset = 256 ;
```

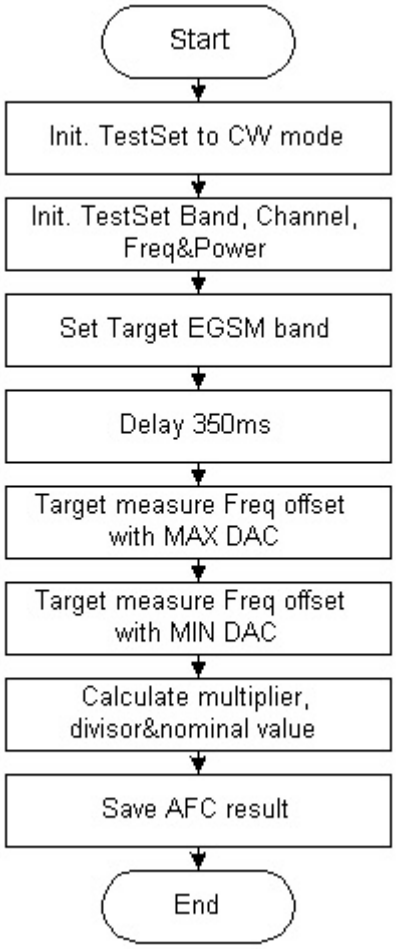
红色部份标示出两个信号不同的地方。PC 和 Target 之间是以按 EMMI 协议连接的，所以信号赋值完毕后，需要将每个信号封装到 EMMI 数据包中，我们可以看到，stData1 和 stData2 的大小是固定的，都是 sizeof(L1AlNvramWriteCalReq)，但这个值超出了一个 EMMI 数据包的容量，于是每个信号都会被分割成 2 个 EMMI 数据包，所以总共需要 4 个 EMMI 数据包才能完成全部数据的传送。

5.4 AFC 算法

AFC 的算法相对简单一些，它跟频带没有什么关系，所以不用对每个频带都进行校正。我们先选择两个 DAC 的值，将这两个值分别设置为 Target 的当前值，然后分别测量在这两个 DAC 下的 Frequency Offset，对照 4.1 节的图五，将 AFC 转化为数学模型就是以 DAC 为坐标 x 轴，Frequency Offset 为 y 轴，已知 x

轴两点 x_1 和 x_2，通过测量得到对应的 y_1 和 y_2，根据一条直线上已知两点(x_1, y_1)和(x_2, y_2)计算直线斜率以及 y = 0 时 x 的值。这里的 x_1 和 x_2 的取值一般是根据经验值得到的，首先它肯定不会超过 DAC 的最大值 2^{13}，其次，这两个值最好选择在基点(x,0)的两侧，同时如果将 x_2 和 x_1 的差值设为 1000 将可以简化计算，通常采用 4500 和 5500 分别作为 x_1 和 x_2，因为基点通常在 5100~5200 左右。

图十一是 AFC 的流程图



图十一

当计算完毕后，AFC 的值保存在一个数据结构中，在 TTPCom 提供的源码中我们可以找到它的定义

```
typedef struct AfcDacConfigurationTag
{
   SignedInt32       multiplier;
   SignedInt32       divisor;
   Int16             nominalValue;       /* DAC value */
   Int16             minRadioInput;      /* DAC value */
   Int16             maxRadioInput;      /* DAC value */
} AfcDacConfiguration;
```

由于采用 4 字节的对齐方式，所以 sizeof(AfcDacConfiguration)会多出来 2 bytes，所以在 PC 上多定义一项 Int16 dummy，如下所示，dummy 这个值不起任何实际作用，仅仅只是为了数据结构对齐

```
typedef struct AfcDacConfigurationTag
{
   SignedInt32      multiplier;
   SignedInt32      divisor;
   Int16            nominalValue;
   Int16            minRadioInput;
   Int16            maxRadioInput;
   Int16            dummy;
} AfcDacConfiguration;
```

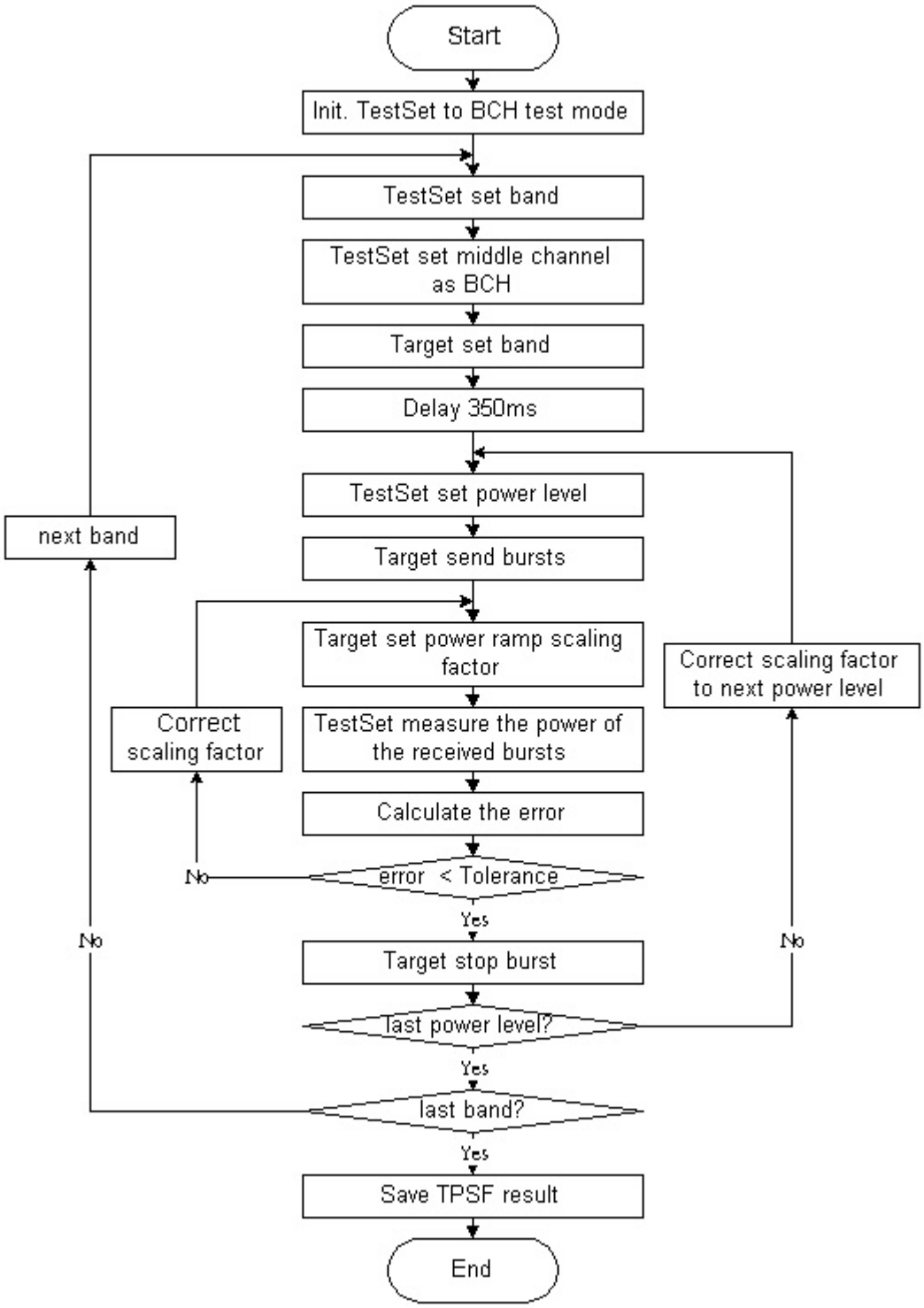
当硬件确定之后，minRadioInput 和 maxRadioInput 也就确定了，在这里这两个值分别为 0 和 2^{13}-1，multiplier 跟 x_1 和 x_2 的选择相关，x_1 和 x_2 确定后，multiplier 也就确定了 multiplier = (x_2-x_1)*FEEDBACK_GAIN，dummy 不起作用，固定的为 0，以上提到的 4 个值可看作常数。divisor 和 nominalValue 这两个值是通过计算得出的，divisor 对应为斜率，divisor=Slope*FEEDBACK_GAIN，nominalValue 则是基点值，nominalValue = $x_1 + y_1$*(multiplier/divisor)。

AFC 计算完毕后，将结果作为信号 SIG_L1AL_NVRAM_WRITE_CAL_REQ 的参数发送给 Target，Target 会将结果保存在 NVRAM 中。

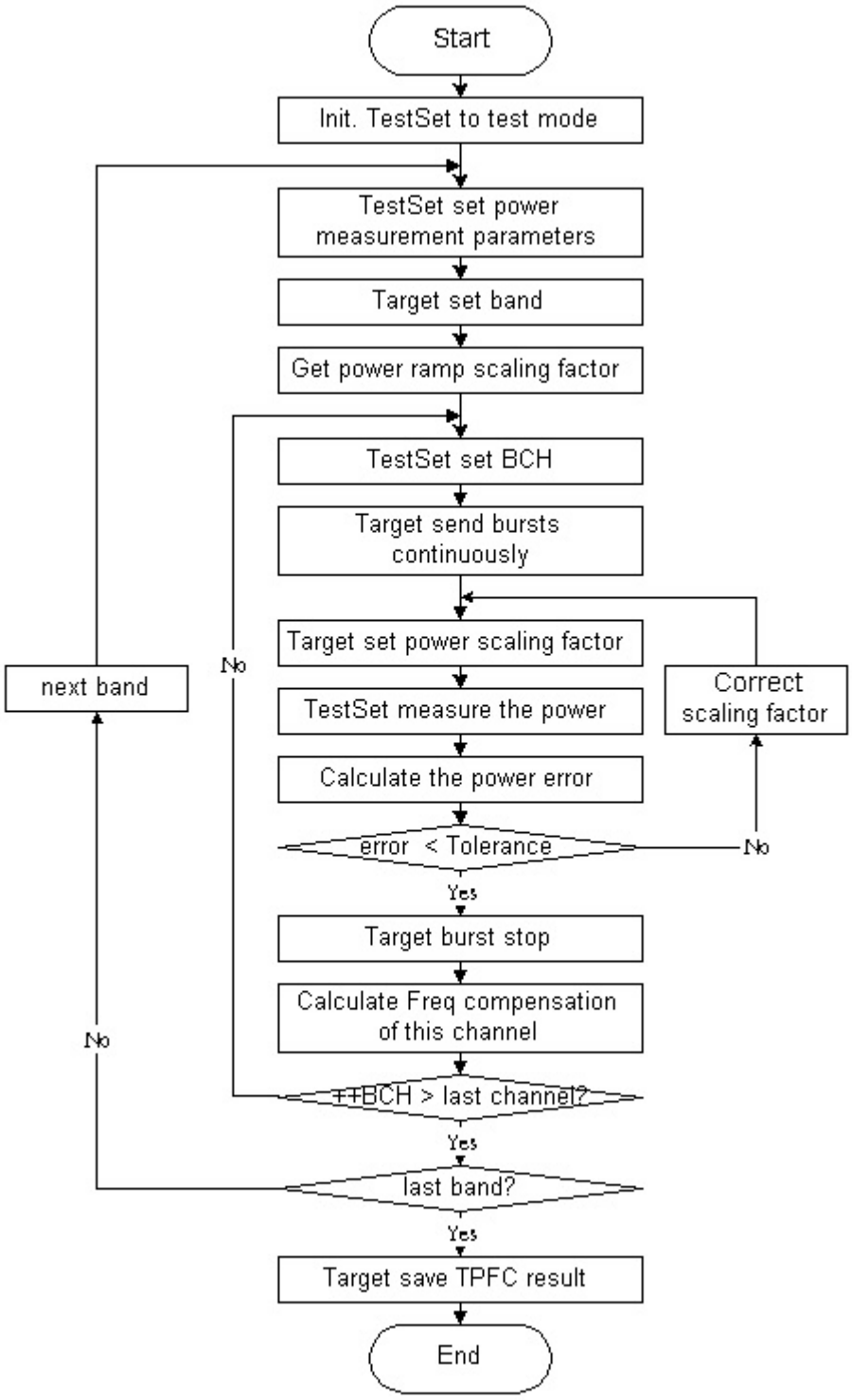
5.5 TPSF 算法

TPSF 的值是和频带相关的，所以我们需要对每个频带进行校正，对于每个频带，选择其中心频道进行校正。校正的过程是由 Target 设置好自身的发送功率等级，然后通过 BCH 信道连续发送 bursts，TestSet 则负责测量接收到的 bursts 的功率，看其值是否能达到 GSM05.05 的要求。

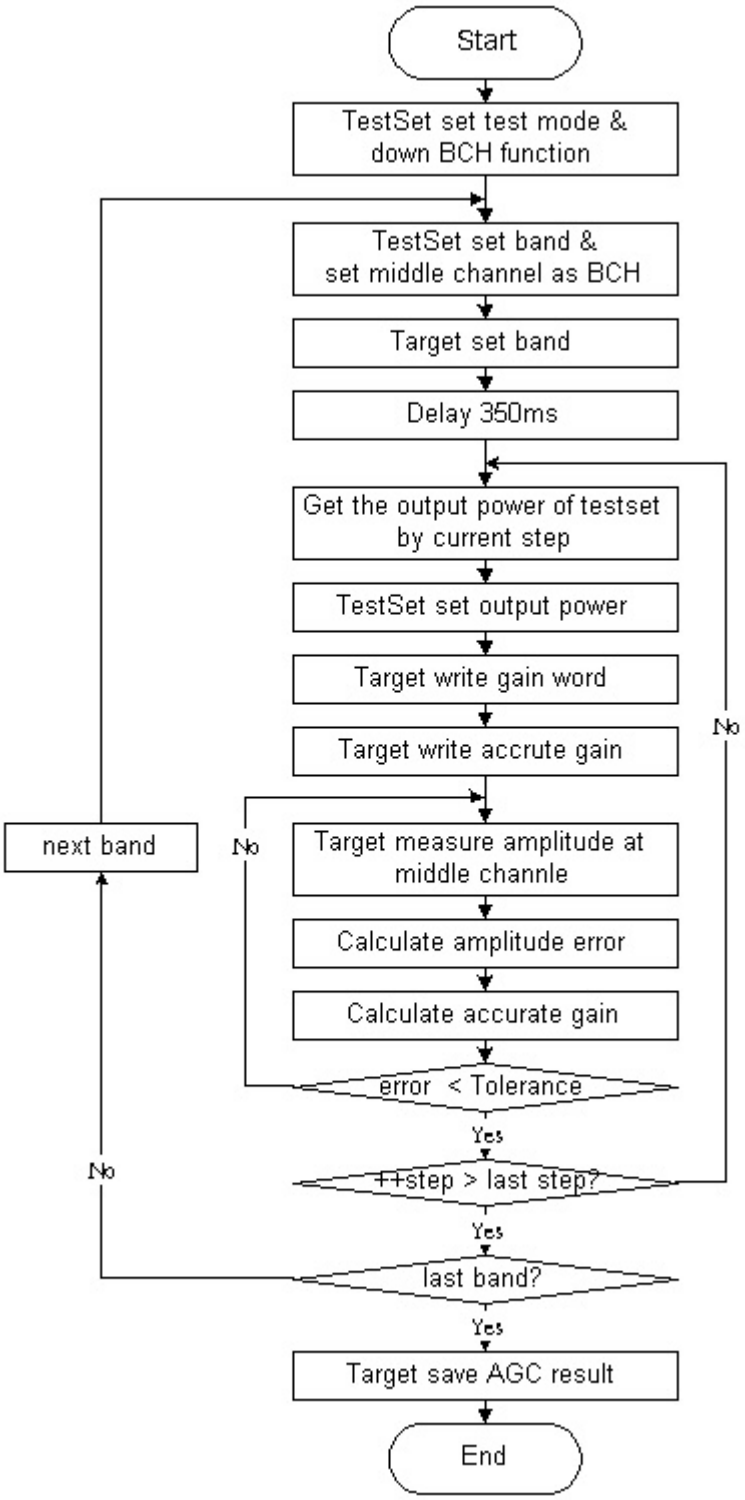
通过信号 SIG_CDMG_BURST_REQ 可以控制 burst 的发送和停止，用此信号将 Target 设为连续发送 bursts，然后使用信号 SIG_CDMG_RAMP_SCALE_REQ 来设置 Power ramp scaling factor，scaling factor 设置好了之后，便可以通过 TestSet 测试接收到的 bursts 的功率，如果接收的功率值不在误差范围之内


Start
Init. TestSet to BCH test mode
TestSet set band
TestSet set middle channel as BCH
Target set band
Delay 350ms
TestSet set power level
Target send bursts
Target set power ramp scaling factor
TestSet measure the power of the received bursts
Calculate the error
error < Tolerance
No
Yes
Correct scaling factor
Correct scaling factor to next power level
Target stop burst
last power level?
No
Yes
last band?
No
Yes
next band
Save TPSF result
End

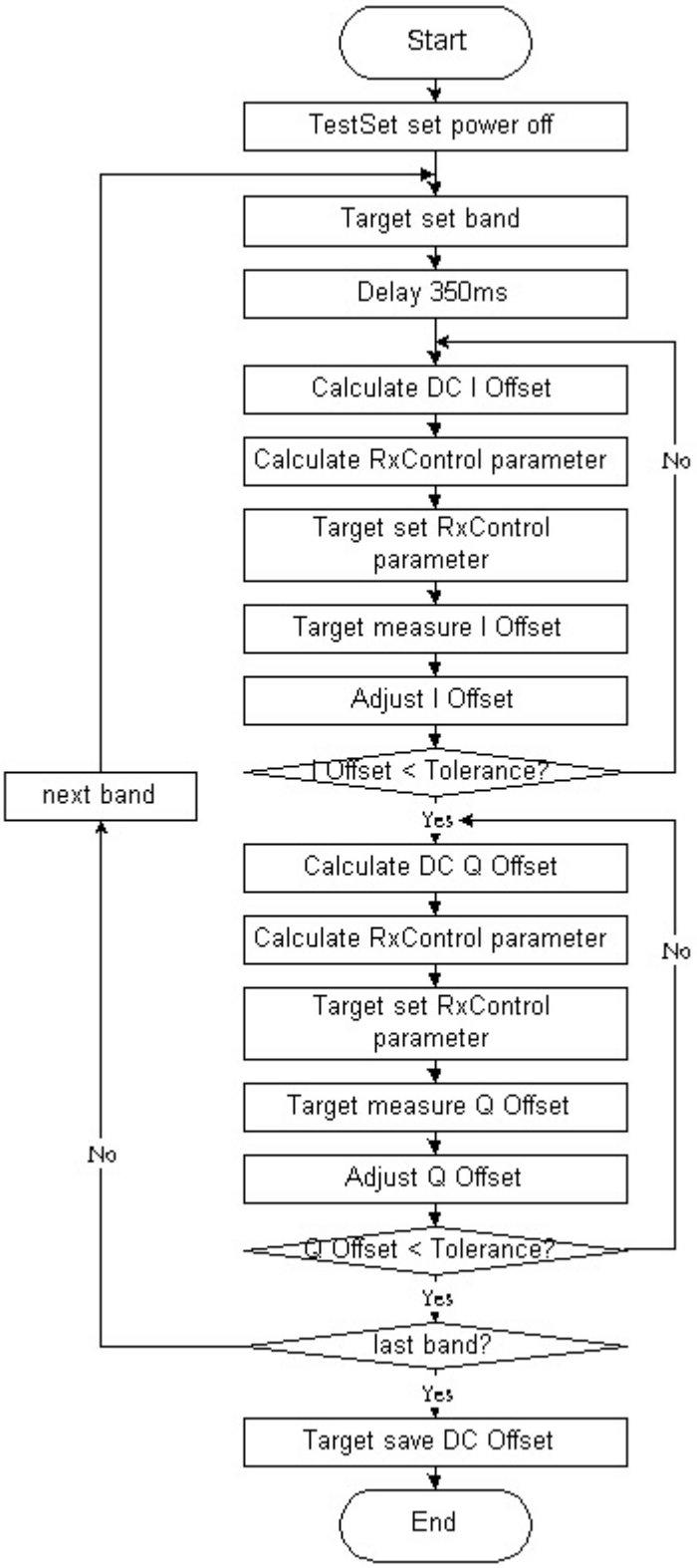
5.6 TPFC 算法


Start
Init. TestSet to test mode
TestSet set power measurement parameters
Target set band
Get power ramp scaling factor
TestSet set BCH
Target send bursts continuously
Target set power scaling factor
TestSet measure the power
Calculate the power error
error < Tolerance
No
Correct scaling factor
Yes
Target burst stop
Calculate Freq compensation of this channel
++BCH > last channel?
No
Yes
last band?
No
next band
Yes
Target save TPFC result
End

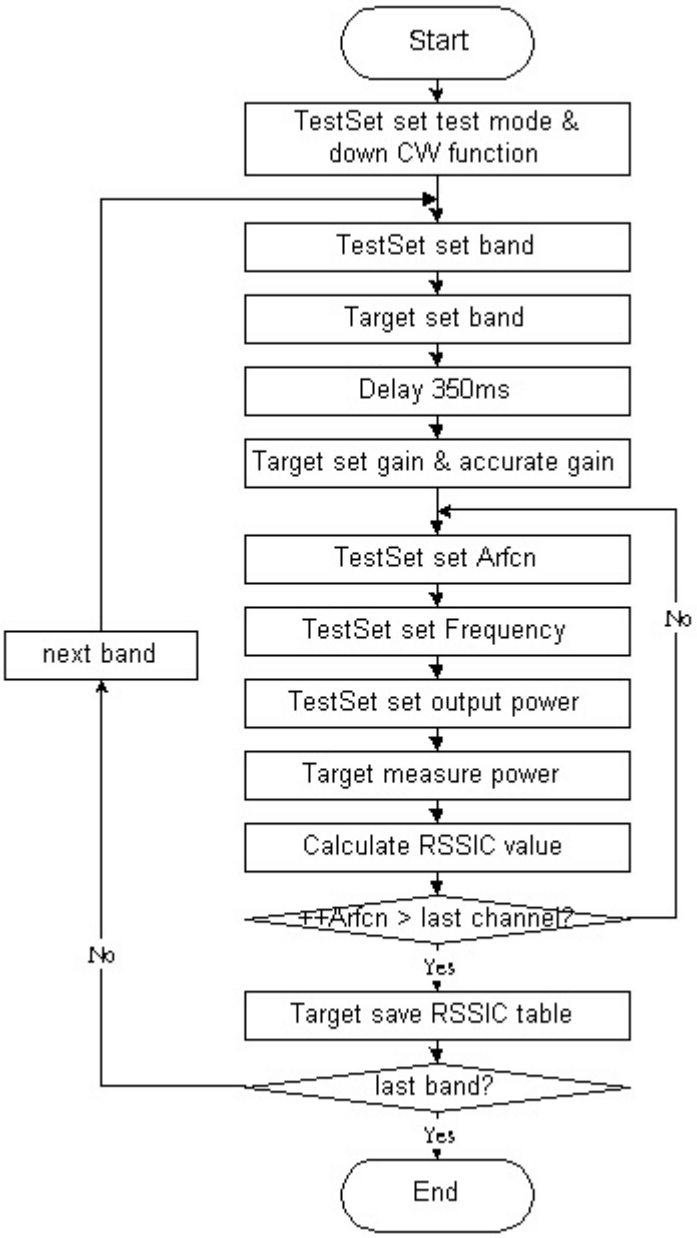
5.7 AGC 算法


Start
TestSet set test mode & down BCH function
TestSet set band & set middle channel as BCH
Target set band
Delay 350ms
Get the output power of testset by current step
TestSet set output power
Target write gain word
Target write accrute gain
Target measure amplitude at middle channle
Calculate amplitude error
Calculate accurate gain
error < Tolerance
No
Yes
++step > last step?
No
Yes
last band?
No
Yes
next band
Target save AGC result
End

5.8 DC Offset 算法



5.9 RSSIC 算法



6 参考文档