

Greta Oto Firmware 设计说明



Jun Mo

Globsky Technology Inc.

2021/7/20

版权信息

本手册，以及与本手册相关的全部代码的版权属于本人所有。所有公开发布的内容仅限于个人和学术团体以学习的目的进行使用。本手册以及相关代码可以进行传播，但必须保留版权信息。未经本人书面允许，本手册以及所有的代码不能用于包括商业行为在内的任何盈利性目的。

本手册内容及相关代码在用于学习目的时，仅能按照原样提供（as is），不附带任何附加服务。另外，由于本手册的内容包含具体的工程实现，因此有可能涉及到已有专利中被保护的方法。由于以学习目的进行发布是非赢利性行为，因此不涉及专利侵权。但本人不对由于第三方私自将本手册的内容和相关代码用于商业目的所产生的侵权行为负责。

1. 软件整体架构

软件主要分成四部分：第一部分是基带控制，主要是卫星信号的捕获以及跟踪；第二部分是位置解算，主要是导航电文解码、原始观测量生成以及根据原始观测量计算接收机位置速度；第三部分是用户接口协议及输入输出；第四部分是底层支持，包括硬件和操作系统的抽象层。通过将访问底层基带硬件的操作抽象出来，使得 **firmware** 可以运行在真实的硬件上，也可以运行在 **C Model** 或者以 **SignalSim** 为基础构成的信号模拟平台上。

1.1 源代码组织

软件根据不同函数的功能组织在不同的目录下，不同目录的不同源代码对应的主要功能如下：

Abstract: 硬件抽象层和操作系统平台抽象层

HWCtrl_Model.cpp: 硬件控制实现在使用基带模型（C Model 或 SignalSim）时的实现

PlatformCtrl_Model.cpp: 操作系统 API 在使用基带模型时的实现

Baseband: 基带控制，源文件和头文件分别存放在 src 和 inc 目录下

AEManager.c: 捕获引擎控制相关函数

ChannelManager.c: 通道控制相关函数

ComposeOutput.c: 输出内容编写构成相关函数

FirmwarePortal.c: firmware 调用入口和与系统平台接口的函数

InitSet.c: 不同卫星 PRN 配置参数数组

TaskQueue.c: 任务队列控制及调用函数

TEManager.c: 跟踪引擎控制相关函数

Common: Firmware 公用的类型及宏定义

PVT: 位置解算，共用源文件和头文件分别存放在 src 和 inc 目录下

PvtBasicFunc.c: PVT 共用基础函数

GlobalVar.c: 全局变量实例化

frontend: PVT 前端，包括原始观测量计算、帧同步和导航电文解析

GpsFrame.c: GPS 帧同步和导航电文解析相关函数

MsrProc.c: 原始观测量计算相关函数

backend: PVT 后端，接收机位置解算

Convert.c: 不同格式数据的转换

Matrix.c: 矩阵运算相关函数

PvtLsq.c: PVT 最小二乘位置解算相关函数

PvtProc.c: PVT 接口及位置解算相关函数

SatCoord.c: 卫星位置计算及与卫星坐标相关的函数

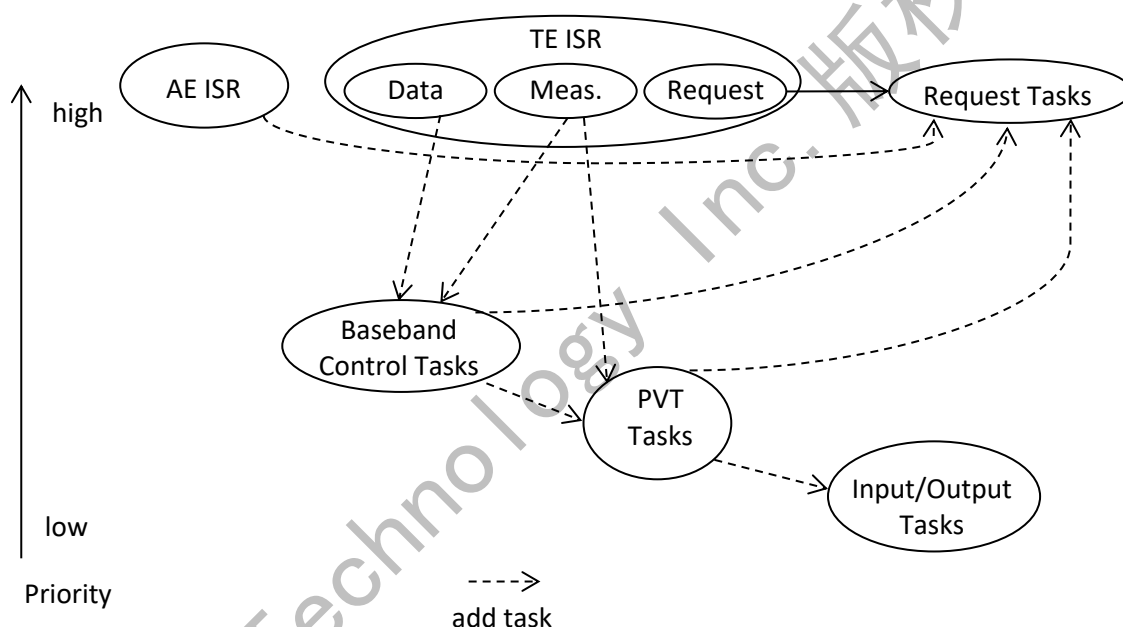
SatManage.c: 维护与卫星相关信息及改正量计算的函数

1.2 软件组织调用架构

软件在初始化完成后，主要的运行都是根据基带中断推动运行的。中断处理程序分别处理捕获引擎中断，跟踪引擎产生的 **data ready** 中断、**measurement** 中断和 **request** 中断。其中捕获引擎

的中断与跟踪引擎是异步发生的，跟踪引擎中断后跟踪引擎都会停止运行，等待软件发出恢复运行的命令。硬件中断处理程序会处理需要立即响应的任务，并且根据需求触发不同优先级的任务线程，并在线程中进行非紧急的任务。由于对基带硬件的访问（包括修改寄存器和 **State Buffer** 的内容）基本上都需要在跟踪引擎停止运行的时候进行，因此除中断任务以外，其他线程的任务完成后如果需要访问基带硬件，都需要设置标志或者将运行的任务加入任务队列并在 **Request** 中断中进行调用。

除了 **request** 任务外，其他任务一般分成三个不同优先级的层级。而 **request** 任务通常是为了处理需要与硬件同步的访问跟踪引擎硬件的操作，直接在 **request** 中断中进行调用。中断和任务的相互关系如下图所示：



基带控制任务主要处理与基带控制相关的任务，比如说比特同步、导航电文码流解码、通道失锁和重捕判断等。环路滤波可以放在中断里面，也可以放在基带控制任务线程中。**PVT** 任务主要处理导航电文帧同步和帧解析、原始观测量计算以及位置解算，同时非紧急的基带相关任务（如 **PPS** 控制）也可以放在 **PVT** 任务里面。输入输出任务主要处理输出内容的打印，输入数据和输入命令的解析等等对于延时不敏感的任务处理。

由于假定应用处理器不支持浮点协处理器，因此原则上中断处理程序和基带控制任务都采用定点计算以缩短处理时间。这样的设计对于包含浮点协处理器的 **CPU** 也适用，因为有些 **RTOS** 中的任务调度器（**Scheduler**）在中断任务中不对浮点协处理器的寄存器提供额外的保护，因此通常会避免在中断处理程序中进行浮点运算。

1.3 任务队列

一个优先级中不同的任务通常彼此之间没有优先级的区别，另外对于需要执行的任务的请求来源通常是异步的（如中断和操作系统线程之间），同时需要执行的任务可能性有很多，因此将具有同样优先等级的任务组织成任务队列，由一个线程根据需要依次调用。每一个任务由以下数据结构进行存储：

```
typedef struct tag_TASK_ITEM
{
    TaskFunction CallbackFunction;
    void *ParamAddr; // address of parameter in buffer (align to DWORD)
    int ParamSize; // size of parameter
    struct tag_TASK_ITEM *pNextItem; // pointer to next item in link list
} TASK_ITEM, *PTASK_ITEM;
```

其中包含了任务函数指针，调用参数存储位置的指针，调用参数 **Buffer** 的大小。由于任务以链表形成进行组织，因此还包含了指向链表中下一个任务的指针。

上述的多项任务由任务队列控制结构体进行管理。对于上述任务数据，任务队列维护两个链表，分别是空闲链表和待执行任务链表，同时为了方便将新任务添加到待执行任务链表的末尾，还有一个指向待执行任务链表末尾的指针。

由于任务的运行与任务的添加函数是异步运行的，因此任务添加函数在添加完成任务后，不负责继续维持添加的任务运行所需要的数据，取而代之的是将数据拷贝到一个循环队列中去。

任务添加函数首先从空闲列表取到一个空闲的任务项，将内容进行相应的赋值后添加到待执行任务列表的末尾。同时，对于任务所需要的数据会在一个循环 **Buffer** 中寻找下一块连续的可以放下任务数据的空间并且将内容拷贝进去。

任务释放函数会将待执行链表中的第一个任务项放回到空闲链表中，同时释放任务数据回到循环 **Buffer** 中。之所以通过一个读指针和一个写指针维护一个循环队列，是因为新任务有可能在待执行任务在执行过程中进行添加，此时已执行完的任务释放的任务数据空间就可以重复使用。

任务执行函数会从待执行链表中依次取出任务项调用执行，并在执行以后将任务项进行释放，直到待执行链表变空为止。相应的任务线程会等待一个信号量（通常由任务添加函数设置），然后调用任务执行函数，并重复上述流程。

由于添加任务时相应的任务参数的数据会拷贝到任务数据空间中，因此一般的参数数据（或数据结构）内容设计的原则是：如果相应的数据可能在后续被任务添加程序修改，或者数据量比较小，那么就拷贝数据，否则就拷贝指针。

2. 信号捕获流程

信号的粗捕获、精捕获、捕获成功的判断以及捕获转跟踪。目前仅实现了通过 AE 判断码相位和多普勒，待后续补充。

3. 部分相干累加结果的处理

在基带设计中，由于每一个 block 的数据的起始和结束时刻与不同卫星信号之间码周期的起始和结束位置不同，并且对应不同延迟的本地码的相干时间彼此错开半个码片，因此会出现在一个 block 的数据结果输出时，只有部分相关累加器的结果进行了输出。上述情况结合不同卫星多普勒造成的码周期起始时刻相对于 block 的起始时间一直在变化，形成了各种不同组合的情况。

在基带硬件中，仅仅记录了当前相关器索引（实际上是下一个将要输出的相关器索引）和进行了相关值覆盖保护。对于不同的组合情况的处理都由基带控制软件进行处理和判断。这些组合情况的处理是基带处理中逻辑判断最复杂的部分，而不正确的处理会造成数据解码错误、基带观测测量计算错误等一系列的问题，因此用单独的章节进行介绍。未来也考虑通过改变硬件时序进行优化，以简化逻辑判断。

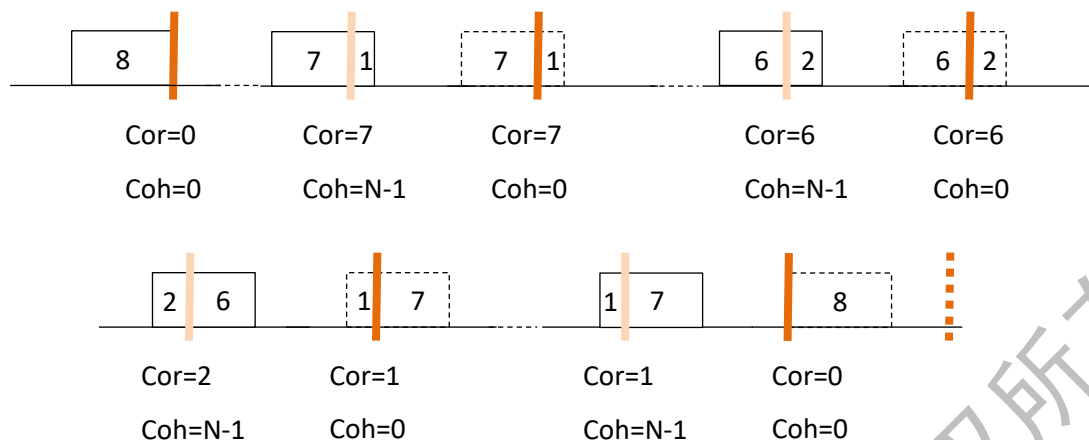
在对不同的组合情况进行分析前，先对硬件的实现时序进行介绍。随着不同码延迟的本地码的推移，相关结果的输出从 Cor0 开始到 Cor7 依次落后半个码片。无论是 DumpCount、PrnCount 还是 CodeSubPhase 的变化都是和 Cor0 是同步的，而 CoherentCount 的增加是在输出 Cor7 的相关结果的同时累加，因此采用上述值进行计算的时候，一定考虑部分结果输出时的计数器变化延迟。

在正常情况下没有部分相关结果输出时，CurrentCor 的值是 0，表示下一个将要输出的相关累加器是 Cor0。同时，当相干累加长度为 N 的时候，CoherentCount 是从 0 到 N-1 进行循环。最后一个相关器完成最后一次相干累加后，CoherentCount 返回 0，因此正常的相干积分中断发生时，CoherentCount=0。当输出了部分结果的时候，比如说在某一个时刻输出了前 3 个相关累加器的值（Cor0、Cor1、Cor2），则下一个将要输出的是 Cor3，此时 CurrentCor 等于 3。因此判断 CurrentCor 是否为 0 就可以判断是否有部分相关结果输出。基带处理软件总是在全部的 8 个相干结果都输出完成的时候才会进行处理，如果仅有部分相关器的相干累加结果输出，则会先记录下来，待下一毫秒中断

下面就根据多普勒的符号以及相干累加时间是否为 1 的组合，分析不同情况下反映在寄存器上的值以及需要进行的相应处理。

3.1 负多普勒相干累加不为 1

在负多普勒的情况下，卫星信号中的码起始位置相对于本地时间在逐渐落后。这种情况下，相关累加值输出和一个 block 结束的相对关系变化如下图所示：



上图中，黑色的实线框表示各个相关器最后一次相干累加结束，黑色的虚线框表示各个相关器的一个 Dump 周期结束，但不是最后一次累加。棕色的竖线表示一个数据 block 处理结束，实线表示发生了相干累加中断，其中深棕色标识有齐全的 8 个相干累加结果，浅棕色标识相干累加结果不完全。虚线表示未发生相干累加中断。数字表示在中断时刻已经完成输出和未完成输出的相关累加器数目。可以看到，正常情况下全部的相关累加器都完成了相干积分，此时 $\text{CurrentCor}=0$ 且 $\text{CoherentCount}=0$ 。但由于存在部分累加和的情况，如果不是全部的相关累加器完成相干积分，此时仍然会发生相干累加中断，但是 CurrentCor 不为 0，且因为最后一个相关累加器还没有完成最后一次相干积分，因此 $\text{CoherentCount}=\text{N}-1$ ，这里 N 代表相干累加长度。

随着码起始位置的逐渐推移， CurrentCor 会从 0 变成 7，然后逐渐减小，最后再变成 0。同时，每一次相干累加完成会间隔 1 毫秒连续给出两次中断，一次标识着前半相关累加器的相干结果输出，此时 $\text{CoherentCount}=\text{N}-1$ ，另一次标识着后半相关累加器的相干结果输出，此时 $\text{CoherentCount}=0$ 。 PendingCor 记录了上一次中断时有多少记录待处理的相干累加结果。

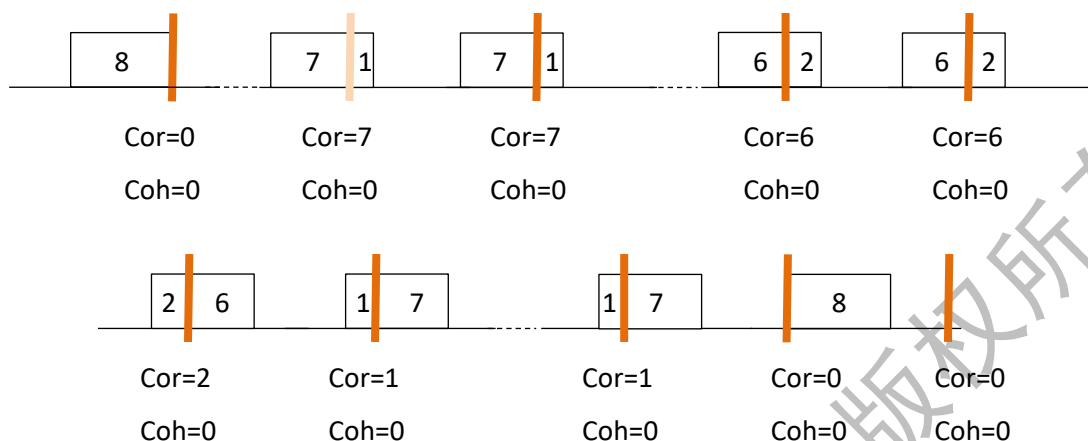
下表列出了上述情况下需要进行的处理：

Cor	0 -> 7	m -> n m, n ∈ [1,7]	1 -> 0
PendingCor!=0	-	补足后 8-PendingCor 个相干累加结果 数据齐全 设置 PendingCor=0	补足后 8-PendingCor 个相干累加结果 数据齐全 设置 PendingCor=0
CurrentCor!=0 && CoherentCount==N-1	记录前 7 个相干累加结果 设置 PendingCor=7	记录前 n 个相干累加结果 设置 PendingCor=n	-

表中“-”表示不会出现的情况。

3.2 负多普勒相干累加等于 1

如果设置相干积分长度为 1，则 CoherentCount 会始终为 0，则出现如下图的情况。



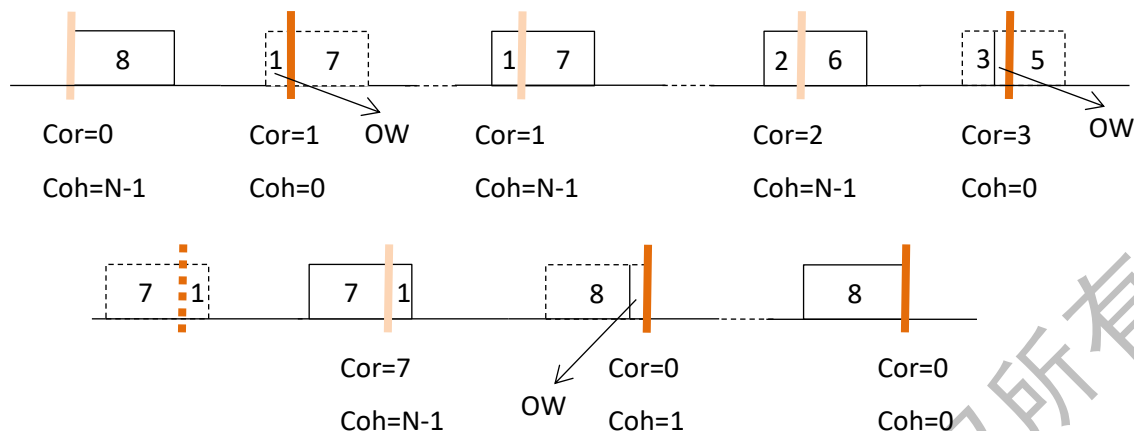
按照上图所示，每毫秒都会产生相干累加中断，在中断时的积分结果的后一部分是前一毫秒的结果，前一部分是后一毫秒的结果，因此既需要补足之前的积分结果，也需要记录新的积分结果。在每次中断中根据不同的情况，需要进行的处理如下表：

Cor	0 -> 7	$m \rightarrow n, m, n \in [1, 7]$	1 -> 0
		补足后 8-m 个相干累加结果 数据齐全	补足后 7 个相干累加结果 数据齐全
	记录前 7 个相干累加结果 设置 PendingCor=7	记录前 n 个相干累加结果 设置 PendingCor=n	

为了跟相干累加长度不为 1 的时候进行对照，表中把记录数据和补足数据的步骤分开。可以看到，前一次记录的 PendingCor=m。由于 N=1，因此同时满足 Coh=0 和 Coh=N-1。当 CurrentCor 从 0 变化到 7 的时候，PendingCor=0。因此相干累加长度为 1 时候的操作也可以与相干累加长度不为 1 时的操作进行对应。

3.3 正多普勒相干累加不为 1

在正多普勒的情况下，卫星信号中的码起始位置相对于本地时间在逐渐超前。这种情况下，有可能出现相关值覆盖保护。相关累加值输出和一个 block 结束的相对关系变化如下图所示：



随着码起始位置的逐渐推移，CurrentCor 会从 0 变成 1，然后逐渐增大，最后再变成 0。同时，每一次相干累加完成会间隔 1 毫秒连续给出两次中断，一次标识着前半相干累加器的相干结果输出，此时 CoherentCount=N-1，另一次标识着后半相干累加器的相干结果输出，此时 CoherentCount=0。当完成最后一次相干累加后，如果下一毫秒的积分相关器位置向前移了 1，将会出现相关值覆盖保护（即图中标识 OW 的位置）。另外，在 CurrentCor 从 7 到 0 的过程中，出现了一次 CoherentCount=1 的情况。

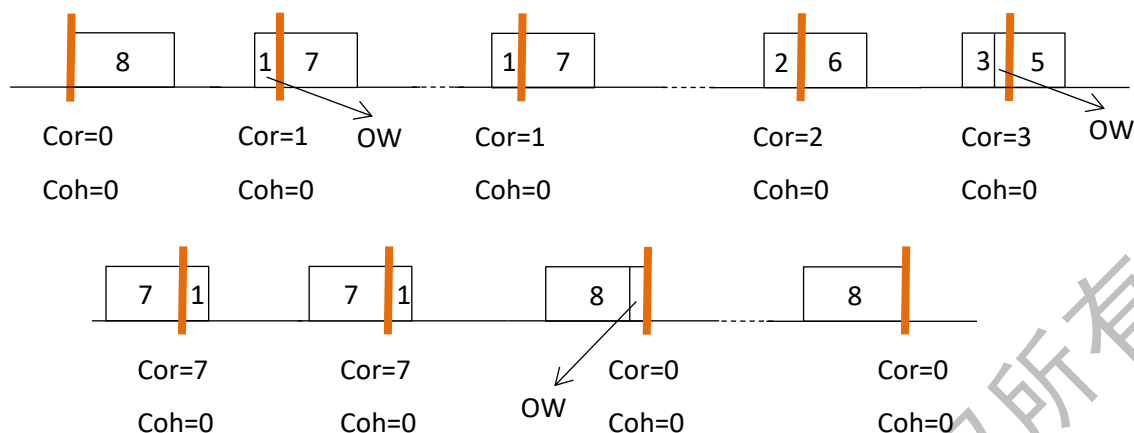
下表列出了上述情况下需要进行的处理：

Cor	0 -> 1	m -> n m, n ∈ [1,7]	7 -> 0
PendingCor!=0	-	补足后 8-PendingCor 个相干累加结果 数据齐全 设置 PendingCor=0	补足后 8-PendingCor 个相干累加结果 数据齐全 设置 PendingCor=0
CurrentCor!=00 && CoherentCount==N-1	记录前 1 个相干累加结果 设置 PendingCor=1	记录前 n 个相干累加结果 设置 PendingCor=n	-

上表中对相应情况的处理与负多普勒的情况时一样的。

3.4 正多普勒相干累加等于 1

如果设置相干积分长度为 1，则 CoherentCount 会始终为 0，则出现如下图的情况。



在上图中，每毫秒都会产生中断，其中当 **CurrentCor** 从 7 变成 0 的时候，会有两次完整的 8 个相关结果出现，一次是将上次中断记录的 7 个补足最后一个，另一次是前 7 个加上储存在相关值覆盖保护寄存器中的值拼成 8 个。相应的处理如下：

Cor	0 -> 1	$m \rightarrow n, m, n \in [1, 7]$	7 -> 0
	使用 8 个相干累加结果 数据齐全	补足后 8-m 个相干累加结果 数据齐全	补足后 1 个相干累加结果 数据齐全
	记录前 1 个相干累加结果 设置 PendingCor=1	记录前 n 个相干累加结果 设置 PendingCor=n	前 7 个相干加 OW 的值 数据齐全

上表中的处理与负多普勒的情况相比，在橙色的表格单元中进行了额外的处理，其他部分的处理方法是一样的。

3.5 部分相关结果的处理逻辑

根据上述情况，我们可以确定以下的若干处理逻辑：

记录部分数据：当 **CurrentCor**!=0 且 **CoherentCount**=N-1 的时候，需要记录前 **CurrentCor** 个相干累加结果，并设置 **PendingCor** 的值为 **CurrentCor**。当 **N**=1 或者 **CurrentCor** 等于 0 的时候，也可以按照同样的逻辑处理。

补足相干累加结果：当 **PendingCor** 不为 0 的时候，需要将后 8-PendingCor 个相干累加结果补足在之前记录的相干累加结果之后，并设置 **PendingCor** 的值为 0。当 **PendingCor** 为 0，且数据齐全时（此时无论 **CurrentCor** 是否为 0，都表示有一组完整的相干结果产生），此时 8-pendingCor 的值为 8，因此与 **PendingCor** 不为 0 的处理逻辑是一样的。

对于数据齐全的判断，需要根据不同的情况进行组合。需要考虑的条件包括 **PendingCor**，也就是上次一相干积分中断后记录的部分相关值是否为 0；**CurrentCor** 是否为 0；相干累加长度是否

设置为 0；当前相干累加值是否为 0。下面的表格给出了在以上条件的各种组合下数据是否齐全的判断：

A/N \ P/C	00	01	11	10
00	X	0	T	T
01	X	X	X	X
11	T	C==1	T	T
10	T	C==1	T	T

上图中，P 表示 PendingCor 不为 0，C 表示 CurrentCor 不为 0，A 表示 CoherentCount 不为 0，N 表示 CoherentNumber 为 1。表格内部 T 表示数据齐全为真，F 表示数据齐全为假，X 表示不会出现的组合，C==1 表示仅当满足 CurrentCor 等于 1 时为真。可以看到，当 P/C 的组合为 01 之外的情况，数据都是齐全的。当 P/C 组合为 01 时，满足 CoherentCount 为 0 且 CurrentCor 为 1 时数据也是齐全的。另外，图中橙色的组合中，当 PendingCor=7 时，有两组完整的相干累加结果。

发生 OW 的情况是多普勒为正，当前中断是最后一次相干累加完成，且额外多输出了一个相关器 相关结果。因此，多输出的这一个结果一定是相干累加中的第一个结果。如果 CoherentNum!=1，则需要将 **TE_OVERWRITE_PROTECT_VALUE** 的内容写入 state buffer 中相应的相干累加结果的位置（地址在寄存器 **TE_OVERWRITE_PROTECT_ADDR** 中）；如果 CoherentNum==1，则表示是一个新的相干累加结果，则需要将其作为最后的一个记录的相干累加结果。

4. 信号的跟踪

4.1 信号跟踪状态及状态转换

目前仅实现了 pull-in 到比特同步到跟踪的简单步骤，待后续补充。

4.2 比特同步和导航电文解码

考虑到比特同步的判断可能采用比较复杂的方法，同时比特同步判断不属于需要紧急完成的任务，因此没有必要在中断处理程序中完成。每次处理相干累加的中断程序收集 20 个 1 毫秒的相关数据后将比特同步任务发送给基带任务队列。同时发送的还有和数据同步的时间标签。比特同步任务完成后，根据时间标签返回相对于时间标签的比特同步位置。后续的中断处理程序可以在忽略比特同步任务延迟的情况下正常调整以比特同步位置为起始的相干积分周期。

比特同步暂时采用最简单的策略：判断 20 个毫秒间隔上的翻转次数，当某一个位置的翻转次数超过 5 次，且在全部翻转中的占比超过一半，即判定比特同步成功。

4.3 鉴相、鉴频和延迟鉴别器

鉴相采用的是传统的反正切的鉴相器。反正切运算采用定点的 CORDIC 算法。对于 GPS L1C/A 信号，采用二象限的反正切，对于导频信号，采用四象限的反正切。

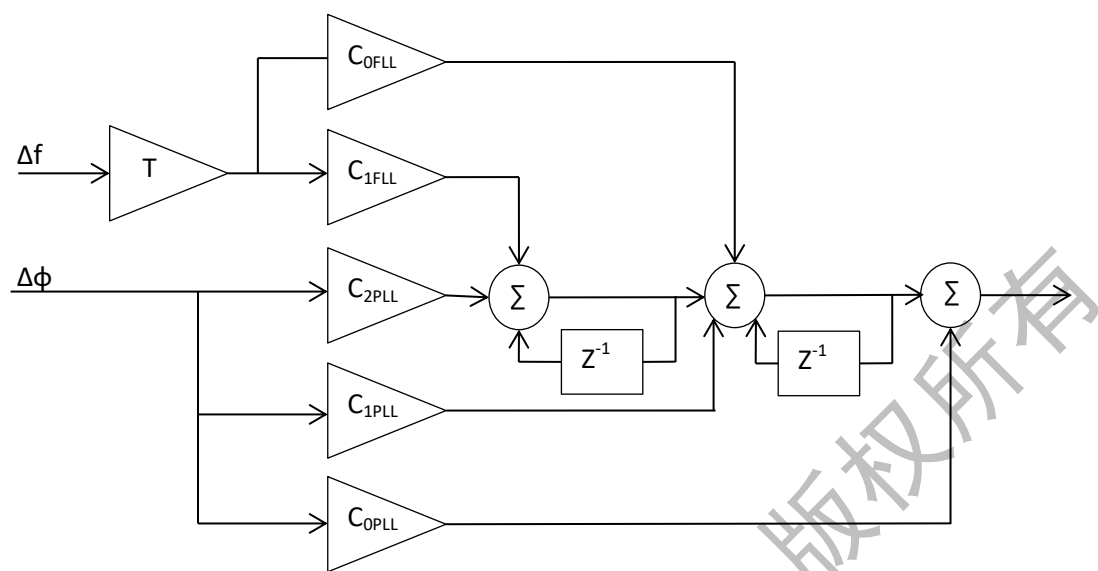
CORDIC 算法采用 14 次旋转运算就以及足够精确。反正切的输出为 -32768~32767 对应 $-\pi \sim \pi$ 的范围，这样当需要对角度进行加减的时候，整周的部分会自动溢出到高 16 比特，或者说低 16 比特依然是对应 $-\pi \sim \pi$ 的范围。

鉴频有两种方式：一种是通过点叉积的方式，即将两次相干积分的结果分别求点积和叉积然后再求二者的反正切。点叉积适合比较强的信号或者初始频差比较大的情况。另外一种方式是 FFT 的方式，即将若干个相干积分结果做 FFT 运算（如相干积分个数少于 FFT 点数则在后面补 0），然后对峰值的频格（Frequency Bin）和其左右两个频格上的能量（或幅度）进行插值运算得到相应的频差。插值运算采用的公式是 $\frac{L-R}{2P-L-R}$ ，或者为了使鉴频曲线变得更加平直，采用 $\tan^{-1} \frac{L-R}{2P-L-R}$ 。采用上述公式的好处是，鉴频结果相对于参与 FFT 的相干积分的个数不敏感，并且当频偏为半个频格（即 $P=L$ 或 $P=R$ ）的时候，鉴频的结果就是 ± 1 ，不需要额外的归一化。

延迟鉴别器采用类似的方法，公式采用 $\frac{E-L}{2P-E-L}$ ，同样具有鉴频结果相对于相关器间隔和相关峰形状不敏感，只需要根据相关器间隔进行简单归一化等优点。

4.4 跟踪环路和环路滤波系数选择

下图显示了一个典型的二阶 FLL 辅助三阶 PLL 的环路滤波器。



上述环路的参数，是根据给定的噪声带宽 B_n 以及环路更新间隔计算的。计算的公式如下：

First order:

$$\omega_n = B_n/0.25 \quad c_0 = \omega_n$$

Second order:

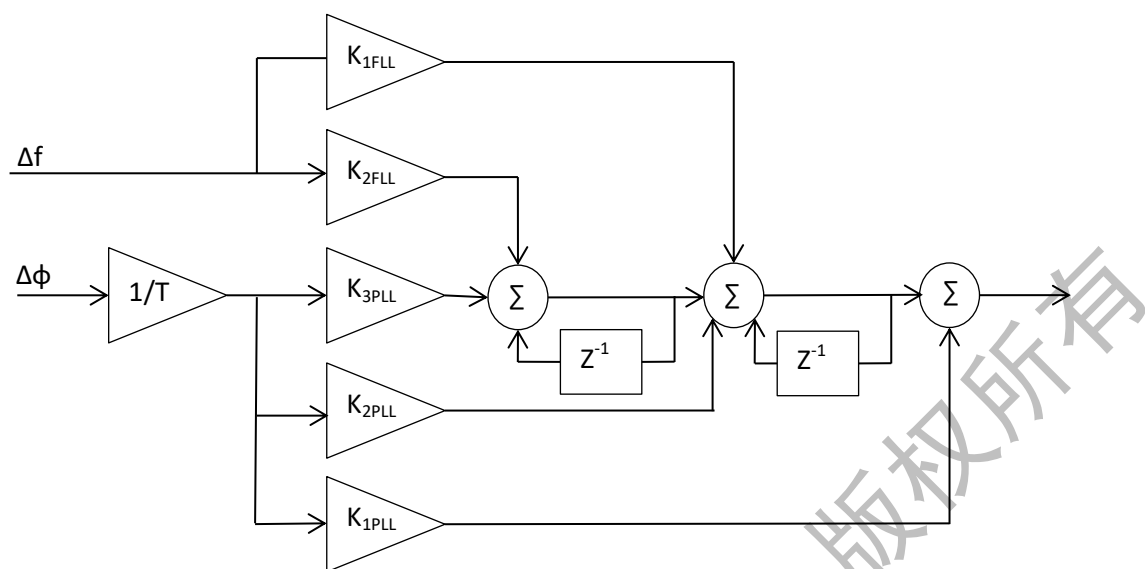
$$\omega_n = B_n/0.53 \quad c_0 = 1.414\omega_n \quad c_1 = \omega_n^2 T$$

Third order:

$$\omega_n = B_n/0.7845 \quad c_0 = 2.4\omega_n \quad c_1 = 1.1\omega_n^2 T \quad c_2 = \omega_n^3 T^2$$

上式中的 T 是环路更新间隔。

为了方便环路参数的计算，将环路改成以下的结构和计算方法：



由于输入的频率差和相位差都除一个 T 的比例因子，因此该比例因子合并到后续的环路参数中。所有，有如下的对应关系：

First order:

$$\omega_n = B_n/0.25 \quad k_1 = c_0 T = \frac{1}{0.25} B_n T$$

Second order:

$$\omega_n = B_n/0.53 \quad k_1 = c_0 T = \frac{1.414}{0.53} B_n T \quad k_2 = c_1 T = \frac{1}{0.53^2} (B_n T)^2$$

Third order:

$$\omega_n = B_n/0.7845 \quad k_1 = c_0 T = \frac{2.4}{0.7845} B_n T \quad k_2 = c_1 T = \frac{1.1}{0.7845^2} (B_n T)^2 \quad k_3 = c_2 T = \frac{1}{0.7845^3} (B_n T)^3$$

可以看出，上述参数都是和噪声带宽 B_n 以及环路更新时间的乘积 $B_n T$ 有关。这里 $B_n T$ 是一个无量纲的量。

上述的参数计算是根据连续信号 FLL/PLL 的系统响应进行计算的，但是在离散系统中，由于 s 平面的极点映射到的 z 平面的极点形成的系统响应不一致，所以上述计算得到的参数用于离散系统 PLL 的时候会造成滤波响应与期望不完全一致，甚至会造成系统不稳定。

一个解决的方法是对于上述的延时累加改成双线性累加。另外一个解决方法是参照论文《Controlled-Root Formulation for Digital Phase-Locked Loops》(by S. A. Stephens and J. B. Thomas) 中提供的参数进行计算。下图给出了论文中通过计算得到的参数表格，可以根据 $B_n T$ 的值查表直接得到相应的环路参数。

TABLE VIII
Loop-Filter Constants for DPLL With Rate-Only Feedback and Standard-Underdamped Response

No computation delay									
	1st order		2nd order		3rd order			4th order	
$B_n T$	K_1		K_1	K_2	K_1	K_2	K_3	K_1	K_2
0.005	0.01976		0.01312	8.661e-05	0.01283	7.385e-05	1.590e-07	0.01165	6.831e-05
0.010	0.03918		0.02589	3.398e-04	0.02580	2.886e-04	1.242e-06	0.02299	2.673e-04
0.015	0.05822		0.03831	7.482e-04	0.03742	6.356e-04	4.084e-06	0.03399	5.879e-04
0.020	0.07689		0.05039	0.001302	0.04918	0.001106	9.429e-06	0.04468	0.001021
0.025	0.09520		0.06214	0.001992	0.06062	0.001691	1.794e-05	0.05506	0.001560
0.030									
0.030	0.1132		0.07358	0.002810	0.07172	0.002383	3.020e-05	0.06516	0.002195
0.035	0.1308		0.08472	0.003746	0.08252	0.003175	4.674e-05	0.07496	0.002921
0.040	0.1481		0.09558	0.004793	0.09302	0.004060	6.800e-05	0.08450	0.003731
0.045	0.1651		0.1061	0.005944	0.1032	0.005032	9.438e-05	0.09377	0.004619
0.050	0.1818		0.1164	0.007191	0.1132	0.006085	1.262e-04	0.1028	0.005578
0.060									
0.060	0.2143		0.1362	0.009950	0.1322	0.008410	2.075e-04	0.1201	0.007691
0.070	0.2456		0.1551	0.01302	0.1503	0.01099	3.137e-04	0.1365	0.01003
0.080	0.2758		0.1731	0.01637	0.1674	0.01380	4.460e-04	0.1521	0.01256
0.090	0.3051		0.1902	0.01995	0.1837	0.01679	6.055e-04	0.1669	0.01526
0.100	0.3333		0.2066	0.02372	0.1991	0.01995	7.924e-04	0.1809	0.01809
0.150			0.2788	0.04471	0.2657	0.03734	0.002135	0.2417	0.03356
0.200			0.3387	0.06740	0.3183	0.05595	0.004103	0.2899	0.04990
0.250			0.3912	0.09013	0.3606	0.07452	0.006583	0.3288	0.06604
0.300			0.4464	0.1108	0.3951	0.09238	0.009451	0.3606	0.08142
0.350					0.4241	0.1093	0.01259	0.3870	0.09580
0.400					0.4499	0.1253	0.01584	0.4093	0.1091
0.450								0.4282	0.1213
0.500								0.4446	0.1325
0.600								0.4726	0.1523
									0.001683
									0.001397
									0.001815
									0.002264
									0.003197

可以看到，随着 $B_n T$ 的增加，环路变得不稳定，需要更高的环路滤波器的阶数。

4.5 环路滤波器更新算法

根据环路滤波器的计算公式，对于三阶 PLL，输入的相位差 $\Delta\phi$ 和输出的控制频率 f 之间的关系是 $f = f_0 + k_1 \frac{\Delta\phi}{T} + k_2 \sum \frac{\Delta\phi}{T} + k_3 \sum \sum \frac{\Delta\phi}{T}$

其中 f_0 为 PLL 运行开始时的初始频率设置。如果阶数小于 3 阶，则更高次的系数为 0。

对于 DLL，上述公式依旧成立，只是 $\Delta\phi$ 变成以码片（或 1/2 码片）表示的码相位差，而 f 则表示本地码频率（或两倍码频率）。

由此可以得到，对于 PLL，第 n 次更新以后的频率为 $f_n = f_0 + k_1 \frac{\Delta\phi}{T} + k_2 \sum \frac{\Delta\phi}{T} + k_3 \sum \sum \frac{\Delta\phi}{T}$

设 $f'_n = f_0 + k_2 \sum \frac{\Delta\phi}{T} + k_3 \sum \sum \frac{\Delta\phi}{T}$

则可以得到 $f'_n = f'_{n-1} + k_2 \frac{\Delta\phi}{T} + k_3 \sum \frac{\Delta\phi}{T}$ ，控制频率为 $f_n = f'_n + k_1 \frac{\Delta\phi}{T}$

对于二阶 FLL，上述公式则变为 $f = f_0 + k_1 \sum \Delta f + k_2 \sum \sum \Delta f$

同样，第 n 次更新的频率和上次更新的频率之间关系为： $f_n = f_{n-1} + k_1 \Delta f + k_2 \sum \Delta f$

这样，控制频率只需要根据上一次更新的值进行迭代更新就可以了。

实际进行更新的时候，使用的增益系数 $k = k_L \cdot k_C / k_D$ ，其中 k_L 、 k_C 和 k_D 分别为环路滤波系数，控制频率到 NCO 频率控制字的转换系数以及鉴别器增益。在对环路滤波进行定点化的时候，会将上述增益一并考虑进去。

4.6 环路滤波器的定点化

为了以定点计算环路滤波，需要正确选择环路参数的比例因子，以在不至于溢出的情况下提供足够的量化精度。为此，需要计算上述参数的选择范围。

下面对 FLL/PLL/DLL 的 $B_n T$ 分别进行分析。

一般来说，对于比较强的信号或者在 pull-in 阶段，会采用比较短的环路更新周期（积分时间），同时采用比较大的环路带宽，而在进入稳定跟踪或者跟踪弱信号时，环路更新周期（积分时间）比较长，但是环路带宽会比较小。

对于 FLL，下面给出几个典型的积分时间

Coh	FFT	Non-coh	Bn (Hz)	T (ms)	BnT	Mode
1	2	5	25	10	0.25	Wideband pull-in (cross-dot)
1	5	8	8	40	0.32	Narrowband pull-in
4	5	16	0.8	320	0.256	weak data signal tracking
10	5	6	0.8	300	0.24	weak pilot signal tracking
1	2	1	5	2	0.01	Minimum possible BnT

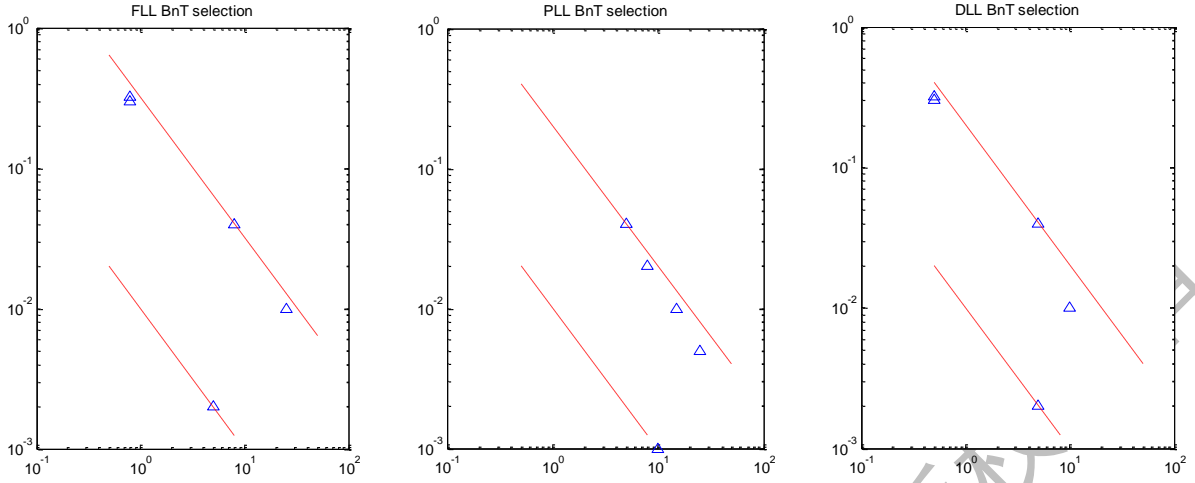
对于 PLL，下面给出几个典型的积分时间

Coh	FFT	Non-coh	Bn (Hz)	T (ms)	BnT	Mode
5	-	-	25	1	0.125	Wideband pull-in
10	-	-	15	5	0.075	Narrowband pull-in
20	-	-	8	20	0.16	weak data signal tracking
40	-	-	5	40	0.2	weak pilot signal tracking
1	-	-	10	1	0.01	Minimum possible BnT

对于 DLL，下面给出几个典型的积分时间

Coh	FFT	Non-coh	Bn (Hz)	T (ms)	BnT	Mode
1	2	5	10	10	0.1	Wideband pull-in (cross-dot)
1	5	8	5	40	0.2	Narrowband pull-in
4	5	16	0.5	320	0.16	weak data signal tracking
10	5	6	0.5	300	0.15	weak pilot signal tracking
1	2	1	5	2	0.01	Minimum possible BnT

显示在下图中：



上图中三个对应下限的红线都是 0.01，FLL/PLL/DLL 对应上限的红线分别是 0.32、0.2 和 0.2。

根据 BnT 的上限和下限，可以从表中查到环路系数的范围。

载波和码的频率控制字增益由采样率确定，如果采样率为 f_s ，则相应的频率控制字增益 $k_C = \frac{2^{32}}{f_s}$ 。采样率最小为 4 倍码速率，因此根据采样率的不同 k_C 的最大值大约为 1050。

鉴相器是通过 CORDIC 算法计算反正切得到的角度，由于需要乘以 $1/T$ 折算到 Hz 为单位，因此角度以周角为单位，如间隔 $T=20\text{ms}$ 时角度差为 $1/10$ 周，则 $\Delta f = \frac{\Delta\phi}{T} = \frac{0.1}{0.02} = 5\text{Hz}$ 。由于 CORDIC 的输出为 2^{16} 表示一个周角，则鉴相器增益 $k_D = 2^{16}T_c$ ，也就是鉴相器输出 $P = 2^{16}T_c \cdot \Delta f$ 。这里 T_c 是相干积分时间。

鉴频器根据鉴频的方式不同，计算方式也不同。当进行点叉积鉴别的时候，可以得到 $\Delta f = \frac{1}{T_c} \tan^{-1} \frac{S_1 \times S_2}{S_1 \cdot S_2}$ 。同样，通过 CORDIC 方法输出的角度增益为 2^{16} ，因此鉴相器增益 $k_D = 2^{16}T_c$ ，也就是鉴频器输出 $F = 2^{16}T_c \cdot \Delta f$ 。当进行 FFT 鉴频的时候，一般通过 8 点 FFT 进行计算。此时，8 个频格（frequency bin）之间总的频率范围为 $\frac{1}{T_c}$ ，则每一个频格之间的频率差距为 $\frac{1}{8T_c}$ 。如果鉴频器采用的差值鉴频公式为 $\tan^{-1} \frac{L-R}{2P-L-R}$ ，其中 L 和 R 分别代表峰值左右的两个频格的信号幅度。则频率相对于中心频格最大偏离 $\pm 1/2$ 频格的时候，有 $P=L$ 或者 $P=R$ ，因此 $\frac{L-R}{2P-L-R}$ 的范围在 ± 1 之间，相应反正切的输出范围在 $\pm 1/8$ 周，也就是 ± 8192 。因此可以得到一个频格对应的鉴频输出范围为 16384，相应的鉴频器增益为 $16384 \times 8T_c = 2^{17}T_c$ 。为了与其他的鉴频鉴相的增益统一，可以将输出右移一位，同样可以得到 $k_D = 2^{16}T_c$ 。需要注意的是如果最大频点不在 0 频率对应的频格上，则每偏一个频格需要补偿 8192（将反正切结果右移一位后）。

延迟鉴别器采用的鉴别公式为 $\frac{E-L}{2P-E-L}$ ，其中 E 和 L 分别代表超前和滞后相关器的信号幅度，则当偏离峰值相关器 $\pm 1/2$ 相关器间隔范围的时候，上述公式的输出的范围为 ± 1 。也就是说当相关器间隔为 $1/2$ 码片， $1/4$ 码片和 $1/8$ 码片的时候，鉴别器增益分别为 2、4、8 倍的半码片。为保

证输出有效数字位数和用整数除法进行鉴别器输出，分子 $E-L$ 在做除法前分别左移 13、12、11 位，使得鉴别器增益变成 2^{14} 。如果最大峰值不在峰值相关器（Correlator 4）上，则每偏一个相关器，需要相应补偿 2^{14} 、 2^{13} 、 2^{12} 。延迟鉴别器输出到环路滤波器的时候，还需要除以环路更新周期，也就是总的积分时间 T 。因此延迟鉴别器输出增益 $k_D = 2^{14}T$ 。

当计算得到 k_C 和 k_D 的值，并确定 k_L 的范围以后，就可以计算环路增益系数 k 并确定环路增益的计算方法了。一般来说，大部分 PLL 会采用两阶或是三阶，FLL 会采用两阶，DLL 会采用两阶或者带载波辅助的一阶。

首先计算 FLL/PLL 的环路更新系数。由于 $k = k_L \cdot k_C / k_D$ ，因此带入 k_C 和 k_D 后可以得到 $k = k_L \cdot \frac{2^{32}}{f_s} \cdot \frac{1}{2^{16}T_c} = k_L \cdot \frac{2^{16}}{f_s T_c}$ 。这里 $f_s T_c$ 表示的是每毫秒的采样点数目乘以相干累加的毫秒数，范围大约在 4000 到 200000，或者 $2^{12} \sim 2^{18}$ 之间。参考从环路参数表中查到的 k_L 值，可以判断 k_1 的范围大约在 2^3 到 2^4 之间。 k_2 的范围大约在 2 到 2^7 之间。考虑到一般应用场景动态较大时环路带宽通常比较宽，而动态较小时或者积分时间较长，或者不需要 3 阶环路，因此 BnT 在 0.05 以下可以不考虑 3 阶环路。由此得到 k_3 的范围大约在 2^8 到 2^{12} 之间。由于上述 k 值不能用整数表示，因此需要以一定的比例因子扩大。

首先，鉴相器的输出在 -32768 到 32767 之间，可以最大乘以 2^{16} 不会溢出，因此 k_1 可以有 2^{13} 的增益，即 $K_1 = 2^{13}k_1$ 。同样 k_2 可以有 2^{15} 的增益，即 $K_2 = 2^{15}k_2$ 。由于 k_3 需要乘鉴频器输出的累加值，以累加值可能扩大 2^9 计算，那么累加值可以乘以 2^7 不溢出，也就是可以有 2^{15} 的增益，即 $K_3 = 2^{15}k_3$ 。最终得到输出给 NCO 的频率控制字为 $W = W_n + (K_1 \cdot P_n) \gg 13$ ， $W_n = W_{n-1} + (K_2 \cdot P_n + K_3 \cdot ACC_n) \gg 15$ ， $ACC_n = ACC_{n-1} + P_n$ 。其中 P_n 为进行第 n 次更新的鉴相器输出， ACC_n 为前 n 次鉴相器输出的累加值。

以采样率 $f_s = 4.113\text{MHz}$ ， $T_c = 20\text{ms}$ ， $B_n = 7.5\text{Hz}$ 的 3 阶环为例，得到 $BnT=0.15$ ， $f_s T_c = 82260$ 。此时可以得到 $K_1 = 0.2657 \cdot \frac{2^{16}}{f_s T_c} \cdot 2^{13} = 1734$ ， $K_2 = 0.03734 \cdot \frac{2^{16}}{f_s T_c} \cdot 2^{15} = 975$ ， $K_3 = 0.002135 \cdot \frac{2^{16}}{f_s T_c} \cdot 2^{15} = 56$ 。

Annex A.1 的 Matlab 程序说明了定点频率控制字的计算 PLL 过程。

FLL 鉴频器的输出范围、鉴别器增益、系数范围计算与 PLL 类似，这里不再赘述。计算频率控制字的方法为 $W_n = W_{n-1} + (K_1 \cdot P_n + K_2 \cdot ACC_n / 4) \gg 13$ ， $ACC_n = ACC_{n-1} + P_n$ 。其中 P_n 为进行第 n 次更新的鉴频器输出， ACC_n 为前 n 次鉴频器输出的累加值。Annex A.2 的 Matlab 程序以 1ms 相干累加长度，20Hz 带宽的点叉积鉴频计算 FLL 定点频率控制字的过程。

对于 DLL，也是用类似的方法计算。首先带入 k_C 和 k_D 到公式 $k = k_L \cdot k_C / k_D$ 后可以得到 $k = k_L \cdot \frac{2^{32}}{f_s} \cdot \frac{1}{2^{14}T} = k_L \cdot \frac{2^{18}}{f_s T}$ 。由于 $f_s T$ 的范围大致在 2^{13} 到 2^{21} 之间，因此可以判断 k_1 的范围大约在 2^1 到 2^7 之间。 k_2 的范围大约在 2^3 到 2^{10} 之间。可以设定 k_1 和 k_2 的增益都是 2^{15} ，即可得到输出给 NCO 的频率控制字为 $W = W_0 + (K_1 \cdot D_n + K_2 \cdot ACC_n) \gg 15$ ， $ACC_n = ACC_{n-1} + D_n$ 。其中 D_n 为进行第 n 次更新的鉴相器输出， ACC_n 为前 n 次鉴相器输出的累加值， W_0 为码 NCO 频率控制字标

称值。Annex A.3 的 Matlab 程序以 5ms 相干累加，8 次非相干累加，2Hz 的滤波器带宽计算 DLL 定点频率控制字的过程。

Globsky Technology Inc. 版权所有

5. 观测量计算和帧同步

5.1 基带观测量

原始观测量计算的基础是基带观测量，也就是基带的各个计数器在观测时刻的计数值。基带观测量通常涉及到以下的值，这些值大部分由基带寄存器提取或基带控制软件维护：

- ✓ 载波频率控制字
- ✓ 载波 NCO，表示本地载波的当前相位
- ✓ 载波整周计数
- ✓ 码频率控制字
- ✓ (半) 码片计数，如果一个调制比特超过一个码周期，还要加上调制比特内的码周期个数
- ✓ 码 NCO，表示一个半码片内的计数值
- ✓ 跟踪状态，锁定标志等

其中半码片计数和码 NCO 是用于计算信号发射时刻的关键数据。由于信号跟踪采用通过本地码对齐信号的方式，因此会将本地码的生成时刻作为观测历元上的信号发射时刻。此时，信号的发射时刻由以下几部分组成：TOW（即一周内的子帧计数）、帧内的比特计数、一个比特内的半码片计数、一个半码片内的码 NCO 计数。发射时刻就是以上几部分的和。前两者的计数值在帧同步的章节进行介绍，半码片内的码 NCO 计数直接是码 NCO 寄存器的值。

一个码循环周期内，半码片的计数值为两倍的码计数（PrnCount 计数值）加上 CodeSubPhase。如果相干积分时间超出一个码循环周期，则还需要加上相干累加次数。这里需要注意的是，无论是 DumpCount 还是 PrnCount 的增加都是跟 Cor0 对齐的，但是相干累加次数的增加是跟 Cor7 对齐的，所以如果 CurrentCor 不是 0，表示寄存器中的相干累加次数与 Cor0 对应的相干累加次数差 1，因此需要加 1 进行补偿。另外，这样计算得到的本地码是对应 Cor0 的本地码，需要额外减掉 4 个半码片得到对应 Cor4 的本地码。

5.2 本地时间的维护

在介绍原始观测量计算之前，先介绍本地时间的维护。由于钟差的存在，本地时间可以不是精确的观测时刻，而是人为定义的接近观测时刻的某一个时刻。为方便起见，本地时间可以对齐到整毫秒、整 20 毫秒甚至是整秒。

本地时间的初始值可以从帧同步中获得的 TOW 值进行初始化。当帧同步完成后，通过 TOW 以及比特计数和码相位，可以得到卫星信号的发送时刻，在此时刻的基础上加上卫星信号的平均传播时间（对于 GPS 大约在 75 毫秒左右）就可以粗略估计出本地时。将粗略估计的本地时对齐到某一个整数，就可以以此值初始化本地时。

本地时间的推算是根据基带处理的时序进行的。一般来说，基带 TE FIFO 设置的一个 block 的数据长度为标称的 1 毫秒，而 Measurement Count 是依据处理的 block 数据的个数来计数的，因此 Measurement Number 即定义的按照标称值计算的两次观测时间（或者称历元）之间的时间长

度。本地时间的维护就是在本地的毫秒计数值上加上了历元间隔，实现了按照射频采样钟计算的本地时的维护。

本地时维护结合 PVT 解算，需要涉及到两个历元间隔，一个是实际的历元间隔，即实际的 Measurement Number，另外一个为标称的历元的间隔。本地时的推算采用标称的历元间隔。通过调整这两个历元间隔的值，可以实现观测时刻整秒对齐和钟差调整。

首先介绍观测时刻整秒对齐。一般接收机位置的输出都要求在时间上对齐整秒（对于位置输出频率高于 1Hz 的情况，需要其中的一个时刻对齐整秒），而由于接收机启动时间的不确定性，一开始的观测时刻可能与整秒相差甚远，因此需要调整观测时刻到尽量接近整秒。此时，通过将 Measurement Number 调整到一个较小的值。此时，实际历元间隔和标称历元间隔都是使用同一个值。由于 Measurement Number 小于 1000，因此可以将观测时刻提前，由此调整本地时到整秒时刻。为了避免 measurement 中断间隔过小，如果需要调整得比较多，可以将 Measurement Number 的值限制在 800~1000 之间，通过多次调整实现整秒的对齐。在实际历元间隔和标称历元间隔使用同一个值的情况下，由于本地时和实际的观测时刻都改变了同样的大小，因此伪距是没有跳变的，因此钟差也不会变化。

钟差的来源一个是在本地时初始化的时候存在的既有时间差，另外一个来自于本地钟飘的累积。除了将本地时刻调整到整秒外，通常也需要将接收机的本地钟差变得尽量小。当通过 PVT 计算得到本地钟差以后，将钟差取整，可以得到钟差调整的毫秒数。将 Measurement Number 取值 1000 加钟差调整毫秒数进行一次调整，同时取实际历元间隔为 Measurement Number，标称历元间隔维持 1000 不变，即可完成钟差调整。此时，由于本地时增加为标称历元间隔，因此可以维持整秒时刻不变，而实际观测时刻由于调整了 Measurement Number，因此调整到与本地时相差 1 毫秒以内。在此情况下，由于钟差发生了变化，因此对应伪距也出现了跳变。

5.3 原始观测量的计算

原始观测量主要包括伪距，即时多普勒和载波相位（或积分多普勒）。其中伪距的计算比较简单，原始伪距计算的公式就是接收机时间减信号发射时间再乘以光速： $\rho = (T_r - T_t) \cdot c$ 。

即时多普勒为本地载波频率控制字折算的本地载波频率减去标称中频。

积分多普勒（ADR）相对来说比较复杂。首先，两个历元间多普勒的积分值可以表示为 $\Delta\Phi = \int_{t_1}^{t_2} f_D dt = \int_{t_1}^{t_2} (f_{LO} - f_{IF}) dt = \int_{t_1}^{t_2} f_{LO} \cdot dt - \int_{t_1}^{t_2} f_{IF} \cdot dt$ ，其中 f_{LO} 表示本地载波频率， f_{IF} 表示标称中频。折算到载波 NCO 的计算就可以得到 $\Delta\Phi = N_2 + \frac{NCO_2}{2^{32}} - (N_1 + \frac{NCO_1}{2^{32}}) - N_{IF}$ 。其中 N_1 和 N_2 分别表示两个时刻载波整周计数值， NCO_1 和 NCO_2 分别表示两个时刻载波 NCO 的值， N_{IF} 表示在时间间隔内的标称中频计数（通常为整数）。

由于 ADR 本身和伪距相对应，正的多普勒对应伪距减小，也就是 ADR 的变化和多普勒的积分值是相反的，因此两个观测量历元上的 ADR 的差值 $\hat{\Phi}_2 - \hat{\Phi}_1 = -\Delta\Phi$ 。因此，可以定义某一时刻的积分多普勒 $\hat{\Phi}_0 = \hat{N}_0 - \frac{NCO_0}{2^{32}}$ ，其中初值 \hat{N}_0 可以根据伪距折算某一个整数值，然后通过计算

$\hat{N}_k = \hat{N}_{k-1} - [(N_k - N_{k-1}) - N_{IF}]$ 得到历元 k 上的整周计数值 \hat{N}_k ，再计算得到历元 k 上的积分多普勒 $\hat{\Phi}_k = \hat{N}_k - \frac{NCO_k}{2^{32}}$ 。由此计算得到的积分多普勒可以满足 $\hat{\Phi}_k - \hat{\Phi}_{k-1} = -\Delta\Phi$ 。

5.4 GPS 的帧同步和帧解析

相比于其他有导频通道的卫星信号，GPS 的帧同步相对来说会更复杂。对于有导频信号的卫星系统，由于数据调制的帧与导频通道上的 Secondary Code 是同步的，因此通过同步 Secondary Code，可以基本确定数据通道的帧起始位置，然后再进行解析。而 GPS L1C/A 码的 LNAV 导航电文只能通过解析导航电文的方式进行同步。

GPS 的 LNAV 电文是以 WORD 的方式构成，每一个 30bit 的 WORD 再加上前一个 WORD 的最后两个比特构成一个可以进行校验的单元，再通过 10 个 WORD 构成一个子帧。每一个 WORD 的数据构成和校验方式如下图所示（IS-GPS-200K 128 页 Table 20-XIV）：

D_1	=	$d_1 \oplus D_{30}^*$
D_2	=	$d_2 \oplus D_{30}^*$
D_3	=	$d_3 \oplus D_{30}^*$
•		•
•		•
•		•
•		•
D_{24}	=	$d_{24} \oplus D_{30}^*$
D_{25}	=	$D_{29}^* \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_5 \oplus d_6 \oplus d_{10} \oplus d_{11} \oplus d_{12} \oplus d_{13} \oplus d_{14} \oplus d_{17} \oplus d_{18} \oplus d_{20} \oplus d_{23}$
D_{26}	=	$D_{30}^* \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_6 \oplus d_7 \oplus d_{11} \oplus d_{12} \oplus d_{13} \oplus d_{14} \oplus d_{15} \oplus d_{18} \oplus d_{19} \oplus d_{21} \oplus d_{24}$
D_{27}	=	$D_{29}^* \oplus d_1 \oplus d_3 \oplus d_4 \oplus d_5 \oplus d_7 \oplus d_8 \oplus d_{12} \oplus d_{13} \oplus d_{14} \oplus d_{15} \oplus d_{16} \oplus d_{19} \oplus d_{20} \oplus d_{22}$
D_{28}	=	$D_{30}^* \oplus d_2 \oplus d_4 \oplus d_5 \oplus d_6 \oplus d_8 \oplus d_9 \oplus d_{13} \oplus d_{14} \oplus d_{15} \oplus d_{16} \oplus d_{17} \oplus d_{20} \oplus d_{21} \oplus d_{23}$
D_{29}	=	$D_{30}^* \oplus d_1 \oplus d_3 \oplus d_5 \oplus d_6 \oplus d_7 \oplus d_9 \oplus d_{10} \oplus d_{14} \oplus d_{15} \oplus d_{16} \oplus d_{17} \oplus d_{18} \oplus d_{21} \oplus d_{22} \oplus d_{24}$
D_{30}	=	$D_{29}^* \oplus d_3 \oplus d_5 \oplus d_6 \oplus d_8 \oplus d_9 \oplus d_{10} \oplus d_{11} \oplus d_{13} \oplus d_{15} \oplus d_{19} \oplus d_{22} \oplus d_{23} \oplus d_{24}$
Where		
d_1, d_2, \dots, d_{24} are the source data bits;		
the symbol \star is used to identify the last 2 bits of the previous word of the subframe;		
$D_{25}, D_{26}, \dots, D_{30}$ are the computed parity bits;		
$D_1, D_2, \dots, D_{29}, D_{30}$ are the bits transmitted by the SV;		
\oplus is the "modulo-2" or "exclusive-or" operation.		

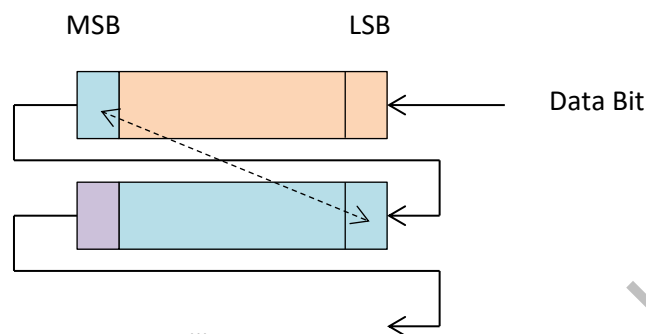
上述 D_1 到 D_{30} 为实际发送的比特，而由于跟踪采用 Costas 环，因此实际接收的比特流可能是发送比特取反。从以上奇偶校验的公式可以发现如下的特点：

- ✓ 奇偶校验总共覆盖范围为 32 个的比特

- ✓ 比特流反相（即 D_{29}^* 、 D_{30}^* 、 D_1 到 D_{30} 全部取反）并不会影响校验的结果
- ✓ 比特流反相仍然可以恢复正常的导航电文
- ✓ 每一个 WORD 的内容会影响校验比特 D_{30} ，由此会影响下一个的 WORD 发送内容

根据上述的第二和第三个特点，对 WORD 进行校验已经解析导航电文的步骤可以和判断码流极性的步骤独立开。同时根据第四个特点，导航电文设计的时候特别把 WORD2 和 WORD10 的最后 2 比特强制置零，这样可以保证 WORD1 的 TOW 的改变不会影响 WORD3~WORD10 的调制内容，同时每一个子帧的发送内容不会受到前一个子帧内容的影响。

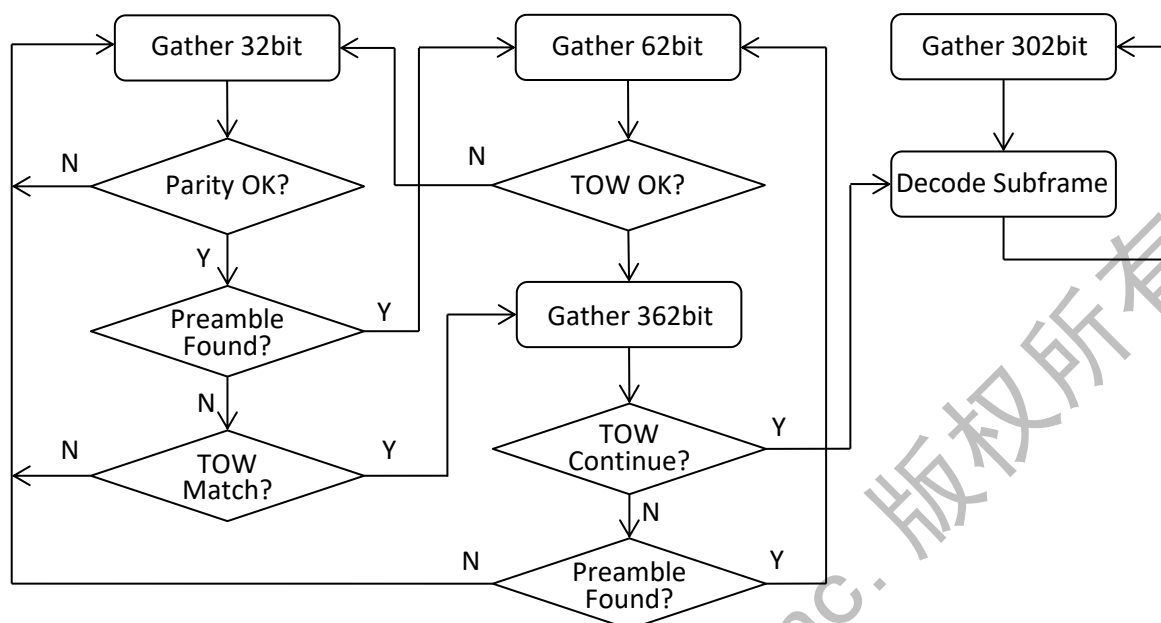
根据上述的第一个特点，输入的比特流数据按照以下方式组织：



比特流的存储是按照 32 比特的 DWORD 数组来组织的，上述每一行的方形代表一个 DWORD，从上到下分别表示数组下标从小到大。当输入新的数据比特的時候，从数字地址 0 的最低比特移入，同时每一个 DWORD 从高位移出的比特进入到数字下一个 DWORD 的低位。另外，从高位移出中的高位指 Bit29 而不是 Bit31。这样，每一个比特在进入同一个 DWORD 的 Bit30 和 Bit31 的同时，也进入了下一个 DWORD 的 Bit0 和 Bit1。这样的方式保证了每一个 DWORD 的最高 2 比特和下一个 DWORD 的最低 2 比特是同样的。

按照这样的存储方式，当一个 DWORD 内 Bit29~Bit0 存储的数据分别为导航电文一个 WORD 的 $D_1 \sim D_{30}$ 时，Bit30 的内容为 D_{30}^* ，Bit31 的内容是 D_{29}^* 。这样，就可以对一个 DWORD 的内容进行奇偶校验的判断了。

对上述比特流进行奇偶校验，如果有连续的几个 DWORD 通过了校验，可以基本认为 WORD 的边沿已经确定。但是为了实现帧同步，还需要校验同步字（Preamble），也就是在 DWORD 的 Bit29~Bit22 寻找 10001011 或 01110100 的模式。在 GPS 的 LNAV 导航电文中，Preamble 仅有 8 比特，所以有可能出现导航电文中某一个 WORD 的高 8 比特出现同样模式的情况，此时就会出现帧同步错误。而接收机又需要尽快进行帧同步，以获取 TOW 后完成本地时间的计算和原始观测量的计算。为了兼顾效率和帧同步的可靠性，帧同步的处理步骤按照以下的方式进行：



首先对齐 **WORD**，也就是累积 32 个比特，然后进行奇偶校验。如果校验不通过，则移入一个新的比特重新进行校验。如果校验通过，则检查同步字（**Preamble**）以同步到 **TLM**。这里为了减少匹配到错误位置的概率，同步字包含了上一个子帧的最后两比特，也就是检查 0010001011 这 10 个比特或其反相的序列。如果有比较准确的本地时间，还可以直接比对 **TOW** 以同步到 **HOW**。当同步字校验通过后，则积累 62 个比特，新增加的 30 个比特将包含 **HOW**，也就是从中可以解析出 **TOW**，此时就可以完成本地时间的计算和原始观测量的计算了。再下一步是继续积累到 362 个比特，也就是连续 12 个 **WORD** 的内容。此时这 12 个 **WORD** 中应该包含一个完整的子帧以及下一个子帧的 **TLM** 和 **HOW**。此时可以检验 **TLM** 中的 **Preamble** 和两个 **HOW** 中的 **TOW** 是否连续。如果检验通过，则确认帧同步已经完成，后续就是解析子帧和积累下一个子帧的数据。如果检验失败，则在后续的 **WORD** 中查找 **Preamble**，然后根据查找的结果确定下一个步骤。

子帧的解析会首先判断子帧号，如果是子帧 4、5，则解析星历、UTC 参数等。如果是子帧 1、2、3，则收集全部 3 个子帧后，判断各自的 **IODC** 和 **IODE** 是否一致，然后进行解析。由于 **LNAV** 的星历信息中不包含卫星号，为了防止由于互相关造成解析到错误卫星星历，在存在有效的历书时，还可以额外进行星历和历书的一致性检查。主要判断升交点赤经以及参考时刻卫星相对于升交点的位置两个参数是否接近。

6. 位置解算

为了减少数据的拷贝和搬移，由于观测量计算和位置解算作为 PVT 的前端和后端运行在同一个线程中，所以两者共享同样的全局变量，包括星历、历书、卫星状态信息、接收机定位结果以及位置解算核心数据等。这些全局变量在 `GlobalVar.h` 中定义，并在 `GlobalVar.c` 中例化。

位置解算的步骤如下：首先，对原始观测量进行筛选，将观测量有效且有有效星历的卫星放入指针数组，并且同一卫星系统的观测量放在一起；第二步是对于筛选出的卫星计算卫星相应的信息，包括通过发射时刻计算卫星位置，根据卫星位置和接收机位置计算卫星仰角和方位角。同时与星历相关的卫星钟差改正和相对论改正也在这一步完成；第三步是对原始观测量进行过滤，包括剔除重复的观测量、剔除仰角和 `CNO` 小于阈值的观测量等；第四步是计算卫星的对流层、电离层改正，并且合并钟差及相对论改正后对原始伪距进行修正，得到修正后的伪距；第五步是根据观测量确定定位方式，如 2D 或 3D 的最小二乘定位、Kalman 滤波定位或 5 星定位等；第六步是根据定位方式调用相应的位置计算函数进行位置解算；最后将解算的结果赋值给接收机定位结果并设置相应的标志。

为了兼容 Kalman 滤波的运算流程以及兼容不同个数的卫星系统，位置解算结果在位置解算核心数据结构中按照三维的速度、钟飘、三维的位置、各个不同卫星系统的钟差这样的顺序排列，这样在增加更多的卫星系统的时候，就可以向后扩展钟差的个数而不会影响到前面数据排列的顺序。

6.1 最小二乘计算

最小二乘由公式

$$\Delta \rho = \mathbf{H} \cdot \Delta \mathbf{x}$$

得到

$$\Delta \mathbf{x} = (\mathbf{H}^T \mathbf{H})^{-1} \mathbf{H}^T \cdot \Delta \rho$$

首先计算列矢量 $\mathbf{H}^T \cdot \Delta \rho$ ，然后计算 $(\mathbf{H}^T \mathbf{H})^{-1}$ ，再进行相乘得到位置误差 $\Delta \mathbf{x}$ 。由于 $\mathbf{H}^T \mathbf{H}$ 是对称正定阵，因此只需要计算其下三角部分，然后通过 Cholesky 分解为三角阵进行求逆后再进行相乘得到 $(\mathbf{H}^T \mathbf{H})^{-1}$ 。

对于位置解算需要进行多次迭代，计算完位置以后速度计算可以采用同样的 \mathbf{H} 矩阵进行运算，不需要再进行迭代。

7. 系统和硬件抽象平台接口

在调试和验证 **Firmware** 的时候，可以在实际的硬件平台上运行，也可以在 PC 仿真环境中运行。为了让 **Firmware** 在运行和调试的时候不依赖于具体的底层功能，因此会将底层抽象出来，提供统一的调用接口，但是对应不同的平台提供接口函数的不同实现。根据需要抽象的功能不同，分为硬件抽象层和系统抽象层两类。其中硬件抽象层主要是提供硬件逻辑（捕获单元、跟踪单元等、外设等）的抽象，系统抽象层主要是提供系统功能（线程调度、IPC 等）的抽象。

硬件抽象层提供的接口函数的定义在 **HWCtrl.h** 中，提供了基带逻辑总线读写、AE Buffer 和 TE Buffer 的读写、外设控制等等，而具体的函数实现在与具体的平台相对应的.c 或.cpp 文件中，根据需要运行的平台不同，在工程中添加相应实现的源代码文件。

系统抽象层提供的接口函数定义在 **PlatformCtrl.h** 中，提供了线程控制、中断控制、IPC 等功能的接口，同样具体的函数实现在与特定平台和 OS 相对应的.c 或.cpp 文件中。

通过抽象层，**main()**函数在 C Model 平台上只需要以下简单的方式构造就可以运行起来：

```
#include <stdio.h>
#include <math.h>
#include <string.h>

#include "RegAddress.h"
#include "InitSet.h"
#include "HWCtrl.h"
extern "C" {
#include "FirmwarePortal.h"
}

void main()
{
    SetInputFile("../..\\..\\..\\matlab\\signal_src\\sim_signal_L1CA.bin");

    FirmwareInitialize();
    EnableRF();
}
```

函数 **SetInputFile()**指定仿真对象（在实际硬件上没有实际作用）。

函数 **FirmwareInitialize()**进行 **firmware** 的初始化，包括创建相应的线程和信号量，注册中断处理函数以及初始化变量等。

函数 **EnableRF()**使能射频前端，产生中频采样信号和采样钟从而启动基带硬件并触发相应的中断处理函数。而在仿真平台上会运行底层仿真功能。

同样的，在 **SignalSim** 平台上，也只需要指定仿真配置文件，就可以运行预设的场景。**main()**函数所在的文件如下所示：

```
#include <stdio.h>
#include <math.h>
```

```
#include <string.h>

#include "HWCtrl.h"
extern "C" {
#include "FirmwarePortal.h"
}

void main()
{
    SetInputFile("test_obs2.xml");

    FirmwareInitialize();
    EnableRF();
}
```

Annex A.1 PLL 频率控制字定点计算示例代码

```
% settings
T=0.02; Bn=7.5; BnT=Bn*T;
% coefficients
k1=0.2657; k2=0.03734; k3=0.002135;
% initial Doppler frequency
f0=120;
% first round
dp1=0.1; df1=dp1/T;
acc=df1; acc2=df1;
f1=f0+k1*df1+k2*acc+k3*acc2;
% second round
dp2=0.12; df2=dp2/T;
acc=acc+df2; acc2=acc2+acc;
f2=f0+k1*df2+k2*acc+k3*acc2;
% third round
dp3=0.075; df3=dp3/T;
acc=acc+df3; acc2=acc2+acc;
f3=f0+k1*df3+k2*acc+k3*acc2;
% convert to frequency control word
fs=4.113e6;
w=[f1*2^32/fs f2*2^32/fs f3*2^32/fs];
% fixed point
% coefficients
kk1=1734; kk2=975; kk3=56;
% discriminator output
p1=dp1*65536; p2=dp2*65536; p3=dp3*65536;
% first round
fc0=f0*2^32/fs;
accp=p1; w1=fc0+floor((kk2*p1+kk3*accp)/2^15);
fc1=w1+floor((kk1*p1)/2^13);
% second round
accp=accp+p2; w2=w1+floor((kk2*p2+kk3*accp)/2^15);
fc2=w2+floor((kk1*p2)/2^13);
% third round
accp=accp+p3; w3=w2+floor((kk2*p3+kk3*accp)/2^15);
fc3=w3+floor((kk1*p3)/2^13);
% compare floating point w with fixed point result [fc1 fc2 fc3]
```

Annex A.2 FLL 频率控制字定点计算示例代码

```
% settings
T=0.002; Bn=20; BnT=Bn*T;
T=0.001; % coherent length
% coefficients
k1=0.09556; k2=0.004793;
% initial Doppler frequency
f0=120;
% first round
df1=12;
acc=df1; acc2=df1;
f1=f0+k1*acc+k2*acc2;
% second round
```

```

df2=14;
acc=acc+df2;acc2=acc2+acc;
f2=f0+k1*acc+k2*acc2;
% third round
df3=9;
acc=acc+df3;acc2=acc2+acc;
f3=f0+k1*acc+k2*acc2;
% convert to frequency control word
fs=4.113e6;
w=[f1*2^32/fs f2*2^32/fs f3*2^32/fs];
% fixed point
% coefficients
kk1=12473;kk2=2503;
% discriminator output
f1=df1*65536*T;f2=df2*65536*T;f3=df3*65536*T;
fc0=f0*2^32/fs;
% first round
accf=f1;fc1=fc0+floor((kk1*f1+kk2*accf/4)/2^13);
% second round
accf=accf+f2;fc2=fc1+floor((kk1*f2+kk2*accf/4)/2^13);
% third round
accf=accf+f3;fc3=fc2+floor((kk1*f3+kk2*accf/4)/2^13);
% compare floating point w with fixed point result [fc1 fc2 fc3]

```

Annex A.3 DLL 频率控制字定点计算示例代码

```

% settings
T=0.04; Bn=2; BnT=Bn*T;
% coefficients
k1=0.1731; k2=0.01637;
% initial Doppler frequency
c0=2.046e6;
% first round
d1=0.2; df1=d1/T;
acc=df1;
w1=c0+k1*df1+k2*acc;
% second round
d2=0.15; df2=d2/T;
acc=acc+df2;
w2=c0+k1*df2+k2*acc;
% third round
d3=0.12; df3=d3/T;
acc=acc+df3;
w3=c0+k1*df3+k2*acc;
% convert to frequency control word
fs=4.113e6;
w=[w1*2^32/fs w2*2^32/fs w3*2^32/fs];
% fixed point
% coefficients
kk1=9038;kk2=855;
% discriminator output
di1=d1*2^14;di2=d2*2^14;di3=d3*2^14;
fc0=c0*2^32/fs;
% first round

```

```
accd=di1;fc1=fc0+floor((kk1*di1+kk2*accd)/2^15);  
% second round  
accd=accd+di2;fc2=fc0+floor((kk1*di2+kk2*accd)/2^15);  
% third round  
accd=accd+di3;fc3=fc0+floor((kk1*di3+kk2*accd)/2^15);  
% compare floating point w with fixed point result [fc1 fc2 fc3]
```