Greta Oto Firmware Design

Jun Mo

Globsky Technology Inc.

2021/7/20

Copyright

1. Firmware architecture

The firmware is composed by four parts: the first part is baseband control, including satellite signal acquisition and tracking; the second part is position calculation, including navigation data decode, raw measurement calculation and receiver position/velocity calculation; the third part is user interface and input/output control; the fourth part is system support, including hardware abstract layer and OS/BSP. By running on a abstracted interface on hardware and low-level control, the firmware can run on real hardware or on a simulated environment (e.g. hardware C model or baseband behavior model).

1.1 Source code organization

The source codes of the firmware are put in different directories depend on their function:

`Abstract`: Hardware abstract layer and platform (OS/BSP) abstract layer
- `HWCtrl_Model.cpp`: Baseband hardware access functions
- `PlatformCtrl_Model.cpp`: OS/BSP abstracted API functions

`Baseband`: Baseband control, source code and header files are placed in /src and /inc
- `AEManager.c`: AE control functions
- `BBCommonFunc.c`: Common functions for baseband control
- `ChannelManager.c`: Channel management functions
- `ComposeOutput.c`: Functions to compose output message
- `FirmwarePortal.c`: firmware and system control API
- `InitSet.c`: PRN configuration arraies
- `TaskManager.c`: Task threads and API
- `TaskQueue.c`: Task queue management functions
- `TEManager.c`: TE control functions
- `TrackingLoop.c`: Tracking loop calculation and control functions
- `TrackingStage.c`: Tracking stage management functions

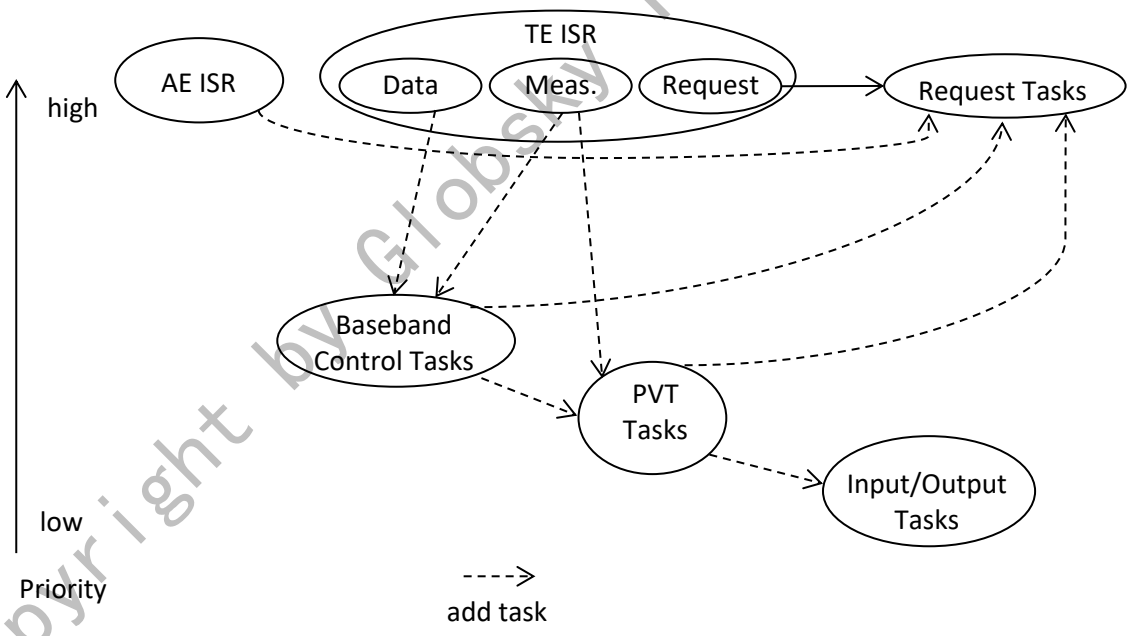
`Common`: Firmware common types and macros definition

`PVT`: Position fix
- `PvtBasicFunc.c`: PVT common functions
- `GlobalVar.c`: Instance of global variables
- `frontend`: PVT front-end for raw measurement calculation and navigation data decode
  - `GpsFrame.c`: GPS frame sync and navigation data decode
  - `BdsFrame.c`: BDS frame sync and navigation data decode
  - `MsrProc.c`: Raw measurement calculation
- `backend`:PVT back-end for position calculation
  - `Convert.c`: Functions for different data type conversion
  - `Matrix.c`: Functions for matrix calculation
  - `PvtKF.c`: Functions for Kalman filter calculation
  - `PvtLsq.c`: Functions for LSQ calculation
  - `PvtProc.c`: PVT position fix and related function API
  - `SatCoord.c`: Functions for satellite position and coordinate calculation
  - `SatManage.c`: Functions for satellite state and correction information calculation

1.2 Software module organization and relationship

After the software is initialized, the main operation is driven by the baseband interrupt. The ISR processes the AE interrupt and TE interrupt (including data ready interrupt, measurement interrupt and request interrupt). The AE interrupt occurs asynchronously with the tracking engine. When any of the TE interrupt happens, the TE will stop and wait for the software to send a command to resume operation. The ISR will do tasks that require immediate response, trigger other task threads of different priorities accordingly. The tasks that do not need immediate response will be done in the task threads. Since access to the baseband hardware (including modifying the contents of registers and the state buffer) needs to be performed when the tracking engine is in stop mode, baseband access requirements in all task threads need to add a task into request task queue and called in ISR.

Besides request task queue, there are 3 other task queues within different priority that were called in threads of different priority. The relationship between ISR and tasks is shown in the following figure:



The baseband control task mainly handles tasks related to baseband control, such as bit synchronization, navigation data decoding, loss-lock and reacquisition determination, etc. The loop filter and tracking loop control can be placed either in the ISR or in the baseband control task thread. The PVT task mainly handles navigation data frame synchronization, message decode, raw measurement calculation and position solution. At the same time, non-urgent baseband-related tasks (such as PPS control) can also be placed in the PVT task. The input and

output tasks mainly handle the composing of output messages, the parsing of input data and input commands, and other tasks that are not sensitive to delays.

Since it is assumed that the application processor does not support floating-point coprocessors, in principle, both interrupt handlers and baseband control tasks use fixed-point calculations to reduce processing time. This principle is also applicable to CPUs that has FPU coprocessors, because the task schedulers in some RTOS do not provide additional protection for the registers of the floating-point coprocessor in interrupt tasks, so floating-point operations are usually avoided in interrupt handlers.

1.3 Low-level support

The purpose of writing and encapsulating the low-level support modules for the baseband and PVT control firmware is to achieve the following goals: first, to improve cross-platform compatibility, including on actual hardware and in simulation environments, as well as in different application environments and OS/BSP support; second, to make the upper-layer software easier to write and more readable; third, to avoid access conflicts to variables and resources, including access conflicts to hardware and conflicts between tasks and threads. The following sections will describe low-level support modules in detail:

1.3.1 Functions to access GNSS related hardware

Hardware abstract layer is used to isolate the implementation method to access the GNSS related hardware. No matter the firmware runs on actual hardware or in a simulated environment (including C model or behavior model), it can access the hardware through a universal interface. The declaration of the functions is put in HWCtrl.h which can be included in upper-level source files. The implementation of these functions will be different and use different sources in corresponding platform. Following is the list of declaration of the functions:

- AttachBasebandISR(): This function is used to attach ISR function to baseband interrupt.
- AttachDebugFunc(): This function is used only for simulation platform.
- GetRegValue()/SetRegValue(): Functions to read/write GNSS baseband registers.
- LoadMemory()/SaveMemory(): Functions to read/write a continuous block of data from or to TE buffer. TE buffer can also be accessed using register access functions, but these two functions can use burst access and is more efficient.
- SetInpuFile(): For simulation environment only. Use to set input IF data file for C model or configuration file for behavior model.
- EnableRF(): To enable RF front-end. Calling to this function should be the last step of baseband initialization. On hardware platform, this function call will enable ADC sampling clock and start to input IF data. With the baseband hardware and software setup, valid ADC clock will trigger the baseband hardware to start and all software modules follows as interrupt generated. For simulation platform, this function will be implemented as a loop to process IF data or simulate according to behavior settings, then call ISR and/or call thread functions accordingly to process different tasks.

1.3.2 OS/BSP abstract layer

The main purpose of the OS/BSP abstraction layer is to isolate the access to the OS API and the access to the driver API. In this way, when porting between different OS and different system hardware platforms, the unified interface function can be replaced with different implementations to achieve the purpose of isolation. The declarations of the functions are placed in PlatformCtrl.h, and different function definition source files are used in different projects. Following is the list of declaration of the functions:

Functions related to OS/IPC are listed as following:

- EnableInt()/DisableInt(): Used to enable and disable interrupt globally.
- CreateThread(): Used to create a new thread. There should be at least 8 threads with different priorities reserved for GNSS firmware. The smaller priority indicator has higher priority. Because the firmware assumed to run infinitely until power off, so no delete thread function is needed.
- ENTER_CRITICAL/EXIT_CRITICAL: This function pair is used to solve resource access conflict between different threads. They are called at the beginning and end of atom operations. The implementation of such functions or macros can be either using mutex or enable/disable interrupt to prevent context switch by scheduler.
- EventCreate/EventSet()/EventWait(): These functions used to create an event, and then wait for such event to be triggered. Generally, the event will be set each time a new task added to a task queue. A thread do tasks will wait for corresponding event in an infinite loop.

Other functions in OS/BSP abstract layer include access standard input/output port, access non-volatile memory; access other auxiliary hardware like antenna status, thermometer, barometer, IMU etc.

1.4 Task queue and task management

Different tasks in one priority level have no priority difference with each other. In addition, the threads/ISR adding tasks and the threads/ISR doing tasks are usually asynchronous. There are many possible tasks to be executed at the same time. Therefore, tasks with the same priority level are organized into task queues, which are called in sequence by a thread on demand. Each task is stored in the following data structure:

```
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;
```

It contains the pointer to task function, the pointer to the storage location of the function parameters, and the size of the parameter buffer. Since the tasks are organized in the form of a linked list, it also contains a pointer to the next task in the linked list.

The above multiple tasks are managed by the task queue control structure. For the above task data, the task queue maintains two linked lists, namely the free linked list and the pending task linked list. At the same time, in order to effectively add new tasks to the end of the pending task linked list, there is also a pointer to the end of the pending task linked list.

Since the running of the task and the calling to add new tasks are asynchronous, the task adding function is not responsible to maintain the data in source buffer unchanged after the task is added. The data in source buffer will be copied to a circular queue.

The function to add new task will first gets an idle task item from the free list, assigns the content accordingly, and adds it to the end of the list of pending tasks. At the same time, parameter used by the task will copy to a circular buffer at place of the next continuous space that can hold the whole parameter buffer.

After a task is executed, the task release function will be called to put the task item in pending link list back to the free list, and release the corresponding parameter buffer in the circular buffer. The reason why a circular queue is used to store parameters is that new tasks may be added in the middle of the execution of the tasks to be executed, and the task data space released by the executed tasks can be reused. There will be space fragments when read and write pointers moving in the circular buffer, so after the pending link list is empty, the read and write pointers will be reset to the start of the circular buffer.

The task will be taken out from the pending link list one by one and the task function will be called. Then the task will be added back to empty link list until the pending link list is empty. The corresponding task thread will wait for event to be set again and repeat the above process.

Since the data of the corresponding task parameters will be copied to the task data space when adding a task, the general principle of parameter data (or data structure) is: if the parameter data may be modified later by the task adding function, or the parameter size is relatively small, then copy the data, otherwise copy the pointer.

The functions in task manager module provide the API to manage tasks. All tasks except those in the request queue are executed in threads. The tasks in request task queue are executed in the interrupt. Besides, some request tasks need to wait for a period of time before they can be executed. Therefore, it is necessary to be scheduled according to the following rules.

Since the request task is mainly a task that needs to interact with the state buffer in baseband hardware, it needs to be completed when the hardware stops, so it is called in the interrupt. Since the request interrupt can be configured with a delay time, which is equivalent to a sleep() call, but there is only one request interrupt, it is necessary to manage tasks with different delays. When setting the request task, follow the following rules:

According to different delay requirements, the tasks that will be executed in the request interrupt are divided into two categories: one is the task that will be executed immediately, with task data, which can be added to the task queue. Typical tasks that updating the state buffer, adding and deleting channels, etc.; the other is the task that needs to wait for specific conditions to start, has no task data, and runs when the conditions are met. Typical tasks are to wait for the AE buffer to be filled to a certain length before configuring and starting capture.

Immediate tasks are added dynamically and directly added to the request task queue. When adding, the request count is set to 1, so that the request interrupt will be generated as soon as possible. Waiting tasks are added statically. The waiting conditions and the pointer to the execution function are configured during initialization, and the dynamic enable flag is used to indicate whether the task is valid. The flag is set when the task is enabled, and when the request count is 0, the request count register is set to the waiting time or a fixed time interval (such as 5ms). Every time a request interrupt occurs, if there is still a valid waiting task, the register is reset when the request count is 0 to ensure there will be next request interrupt. In this way, on the one hand, the immediate task can be executed as soon as possible, and on the other hand, the waiting task can be checked regularly until it is executed. Once the waiting task is executed after the conditions are met, the enable flag will be cleared.

1.5 State buffer content synchronization

The channel configuration and status of all channels, as well as related results, are stored in the state buffer RAM in baseband. These contents stored in the state buffer will be read and written by the CPU during the interrupt. In order to minimize the frequency of hardware access during the interrupt, the software maintains a state buffer content mirror in the system RAM which has higher access speed. After the interrupt starts, the state buffer content that needs to be synchronized is copied to the mirror buffer according to the type of interrupt, and the content that needs to be synchronized in the mirror buffer is written back to the state buffer before the interrupt ends.

To synchronize content from the state buffer to the mirror buffer, if it is a coherent sum interrupt, the coherent sum results (the contents of the last 8 DWORDs) are synchronized. If it is an measurement interrupt, the contents such as channel and counter status are synchronized (a total of 7 DWORDs from PRN_COUNT to MS_DATA).

If the channel configuration is updated during the interrupt processing, it is necessary to select which parts need to be synchronized based on the updated content. According to the update made during the interrupt processing, the synchronized content will select one or more of the following table:

Synchronize flag	Synchronize addresses	Scenario
ALL_DIRTY	0~15	Channel initialization

FREQ_DIRTY	0/1	Tracking loop update
CONFIG_DIRTY	2/3/4	Tracking stage switch
CODE_DIRTY	7/10/11/12	Set code phase

The synchronization read/write fulfills the need for most of the cases. The software uses memory copy to do the synchronization. For cases that need to do synchronize baseband state buffer contents not listed above, read/write directly on state buffer will be used instead.

2. Signal acquisition procedure

The procedure of coarse acquisition, fine acquisition and acquisition to tracking will be descripted in the future.

3. Process of partial coherent sum results

In the baseband design, the integration period is aligned to the local code period, which is different for different correlator. This will make it possible that at the end of data block only part of the correlators has coherent sum result instead of all of them. The code Doppler will change the time difference between the start of PRN code and the start of data block continuously. This means all possible scenarios need to be considered. Combined with different coherent number and overwrite protection flag, the scenarios will be complicated. So we have this whole chapter to describe all possible cases and process methods.

In the baseband hardware, only the current correlator index (actually the next correlator index to be output) and the correlation result to overwrite are recoded. The processing of different combinations is processed and judged by the baseband control software. The processing of these combinations is the most complicated part in the baseband processing. Incorrect processing will cause a series of problems such as data decoding errors and baseband observation calculation errors. In the future, it is also considered to optimize by changing the hardware timing to simplify the software logic of judgment.

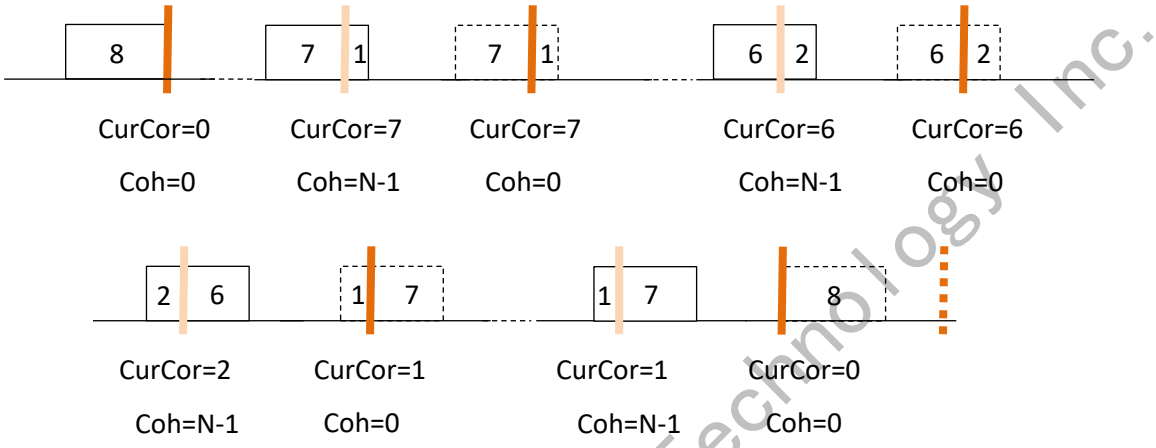
Before analyzing different combinations, the hardware implementation timing is introduced first. With the shifting of local codes with different code delays, the output of the correlation results lags by half chip between each other from Cor0 to Cor7. The change of register DumpCount, PrnCount and CodeSubPhase is synchronized with Cor0, and the increase of register CoherentCount is accumulated while outputting the correlation results of Cor7. Therefore, when using the above values for calculation, the time difference of changing above counter when partial results are output must be considered.

Under normal circumstances, when there is no partial correlation result output, the value of CurrentCor is 0, indicating that the next correlation accumulator to be output is Cor0. At the same time, when the coherent accumulation length is N, CoherentCount cycles from 0 to N-1. After the last correlator completes the last coherent accumulation, CoherentCount returns to 0, so when a normal coherent integration interrupt occurs, CoherentCount=0. When partial results are output, for example, the values of the first three coherent accumulators (Cor0, Cor1, Cor2) are output at a certain moment, then the next one to be output is Cor3, and CurrentCor is equal to 3 at this time. Therefore, by judging whether register CurrentCor is 0 can determine whether there is partial correlation result been output. The baseband processing software will always process when all 8 coherent results are output. If only the coherent accumulation results of part of the correlators are output, they will be recorded first and wait for the next interrupt.

The following analyzes the values reflected in the registers and the corresponding processing required in different situations based on the combination of the sign of the code Doppler (whether code Doppler is greater or less than the nominal value) and whether the coherent accumulation time is one.

3.1 Negative Doppler and coherent number not equals 1

In the case of negative code Doppler (the code Doppler is less than the nominal value), the code start position in the satellite signal gradually lags behind the local time. In this case, the relative relationship between the correlation accumulation value output and the end of a block changes as shown in the following figure:



In the figure above, the black solid line box indicates that the last coherent accumulation of the correlator has ended, and the black dotted line box indicates that a dump cycle of each correlator has ended, but it is not the last coherent accumulation. The brown vertical line indicates the edge of a data block. Solid line means a coherent sum interrupt has occurred and dotted line means there is no coherent sum interrupt. Within the solid brown lines, the dark brown lines indicate that there are 8 complete coherent accumulation results, and the light brown lines indicate that the coherent accumulation results are incomplete. The values of the register CurrentCor and CoherentCount is also shown when interrupt happens. It can be seen that under normal circumstances, all related accumulators have completed coherent integration, CurrentCor=0 and CoherentCount=0. However, due to the existence of partial accumulation, if not all coherent accumulators have completed coherent integration, a coherent accumulation interrupt will still occur at this time, but CurrentCor is not 0, and because the last related accumulator has not completed the last coherent integration, CoherentCount=N-1, where N represents the coherent sum number.

As the code start position gradually moves, CurrentCor will change from 0 to 7, then gradually decrease, and finally become 0. At the same time, each time the coherent sum completed, two consecutive interruptions will be given at an interval of 1 millisecond. One indicates the coherent result output of the first half of the coherent accumulator, at this time CoherentCount = N-1, and the other indicates the coherent result output of the second half of the coherent accumulator, at this time CoherentCount = 0. PendingCor is a variable recording how many coherent sum results are pending to be processed in the previous interrupt.

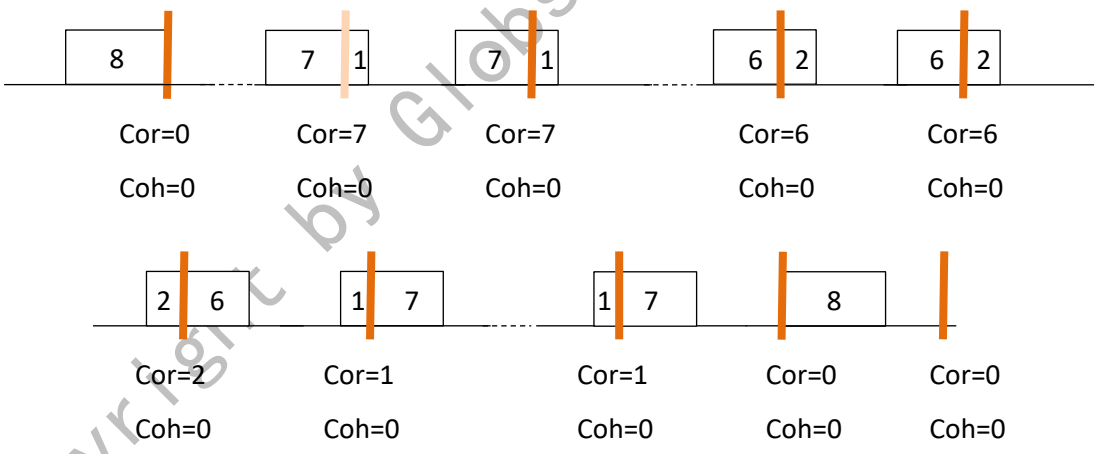
The following table gives the combination of different situation and corresponding operation:

CurrentCor	0 to 7	m to n; m, n ∈ [1,7]	1 to 0
PendingCor!=0	-	Complement the 8-PendingCor coherent sum result Data complete Set PendingCor=0	Complement the 8-PendingCor coherent sum result Data complete Set PendingCor=0
CurrentCor!=0 && CoherentCount==N-1	Save first 7 coherent sum results Set PendingCor=7	Save first n coherent sum results Set PendingCor=n	-

In the above table, - means such combination does not exist. Data complete means all 8 coherent sum result are ready to be processed.

3.2 Negative Doppler and coherent number equals 1

If coherent number is set to 1, the value of CoherentCount will be always 0, as illustrated in the following figure:



As illustrated in the above figure, interrupt will happen after processing of each data block. At the interrupt, part of the coherent sum results of previous millisecond need to be complement and part of the coherent sum result of next millisecond need to be saved. The combination is also shown in the following table:

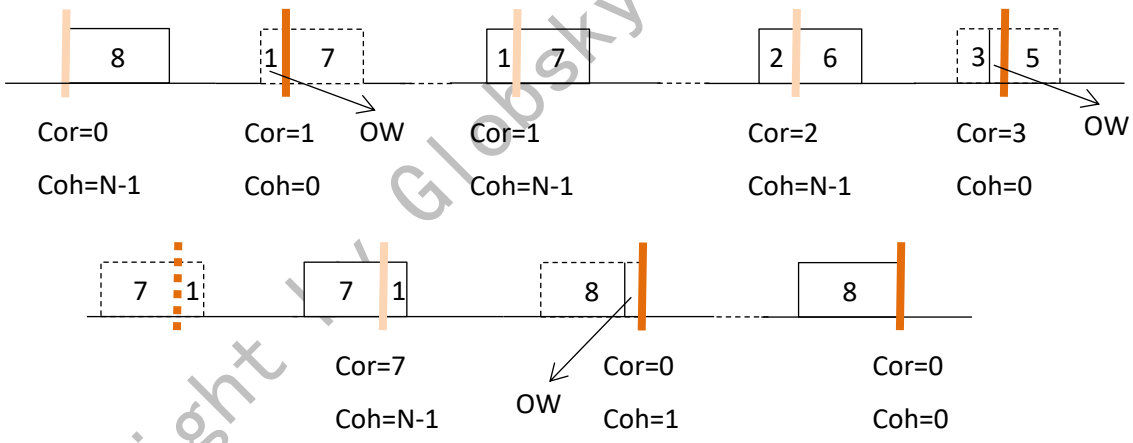
In order to compare with the cases when the coherent number is not 1, the table separates the steps of recording results and complementing results. It can be seen that PendingCor is set to m in the previous step. When N = 1, CoherentCount satisfies both equals 0 and equals N-1. When

CurrentCor changes from 0 to 7, PendingCor = 0. Therefore, the operations will be the same despite the value of coherent number.

CurrentCor	0 -> 7	m -> n m, n ∈ [1,7]	1 -> 0
		Complement the 8-m coherent sum result Data complete	Complement the 7 coherent sum result Data complete
	Save first 7 coherent sum results Set PendingCor=7	Save first n coherent sum results Set PendingCor=n	

3.3 Positive Doppler and coherent number not equals 1

In the case of positive Doppler, the code start position in the satellite signal gradually advances relative to the local time. In this case, correlation result overwrite protection may occur. The relative relationship between the correlation sum result output and the end of a data block is shown in the following figure:



As the code start position gradually moves, CurrentCor will change from 0 to 1, then gradually increase, and finally become 0. At the same time, with the output of each 8 coherent sum results, two consecutive interruptions will be given at an interval of 1 millisecond. One indicates the coherent sum result output of the first half of the correlators, at this time CoherentCount=N-1, and the other one indicates the coherent sum result output of the second half of the correlator, at this time CoherentCount=0. After the last coherent accumulation is completed, if the index of the correlator in the next millisecond moves forward by 1, the correlation result overwrite protection flag will be set (the position marked OW in the figure). In addition, in the process of CurrentCor changing from 7 to 0, condition of CoherentCount=1 appeared once.

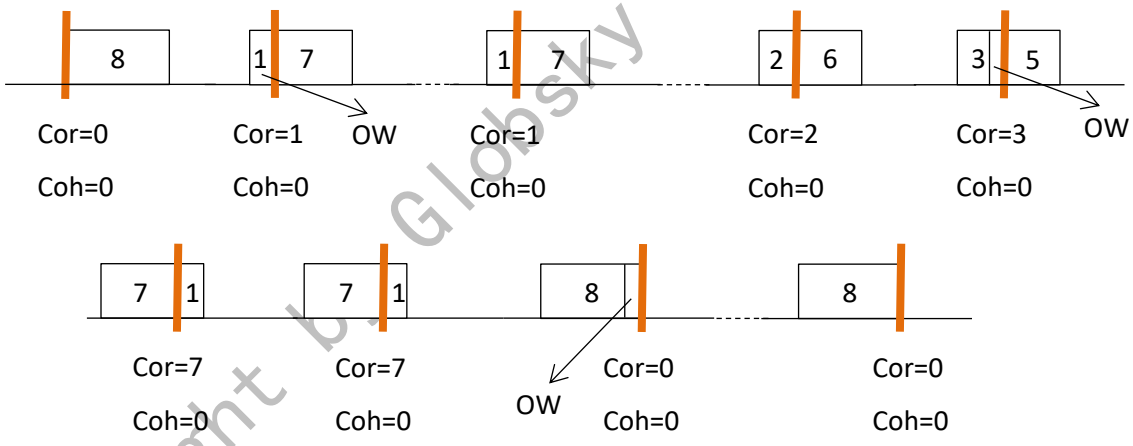
The following table gives the combination of different situation and corresponding operation:

CurrentCor	0 -> 1	m -> n m, n ∈ [1,7]	7 -> 0
PendingCor!=0	-	Complement the 8-PendingCor coherent sum result Data complete Set PendingCor=0	Complement the 8-PendingCor coherent sum result Data complete Set PendingCor=0
CurrentCor!=0 && CoherentCount==N-1	Save first 1 coherent sum results Set PendingCor=1	Save first n coherent sum results Set PendingCor=n	-

The above table has same operation as the tables for negative Doppler.

3.4 Positive Doppler and coherent number not equals 1

If coherent number is set to 1, the value of CoherentCount will be always 0, as illustrated in the following figure:



As illustrated in the above figure, interrupt will happen after processing of each data block. When CurrentCor changes from 7 to 0, there will be two set of complete 8 coherent sum results: one set comes from 7 pending results complemented by the first current result, the other set comes from latter 7 current results complemented by the one stored in overwrite protect register. The combination is also shown in the following table. Compare with the table of negative Doppler, table cells with orange background has extra operations while other cells have the same operation.

CurrentCor	0 -> 1	m -> n m, n ∈ [1,7]	7 -> 0
	Use 8 coherent sum results Data complete	Complement the 8-m coherent sum result Data complete	Complement the last coherent sum result Data complete
	Save first 1 coherent sum results Set PendingCor=1	Save first n coherent sum results Set PendingCor=n	Rest of the 7 coherent sum results and result in OW register Data complete

3.5 Operation to process partial coherent sum results

Base on above 4 scenarios, operations for different cases is listed below:

To save part of the results as pending: When CurrentCor!=0 and CoherentCount=N-1, the coherent sum result for first CurrentCor correlators need to be saved and set PendingCor equals CurrentCor.

To complement incomplete coherent results: When PendingCor not equals 0, the coherent sum result for the last 8-PendingCor correlators.

To determine whether data complete, following combinations of 4 variables/registers are used. In the following table P means PendingCor!=0, C means CurrentCor!= 0, A means CoherentCount= 0, N means CoherentNumber=1.

P/C A/N	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

T means data complete is true, F means data complete is false, C==1 means data complete when condition is true, X means combination is not possible (such as CoherentNumber=1 and CoherentCount!= 0). The above table shows data complete will be true except for the case of P/C combination of 0/1. And in this case, data complete is true when CoherentCount=0 and CurrentCor=1.

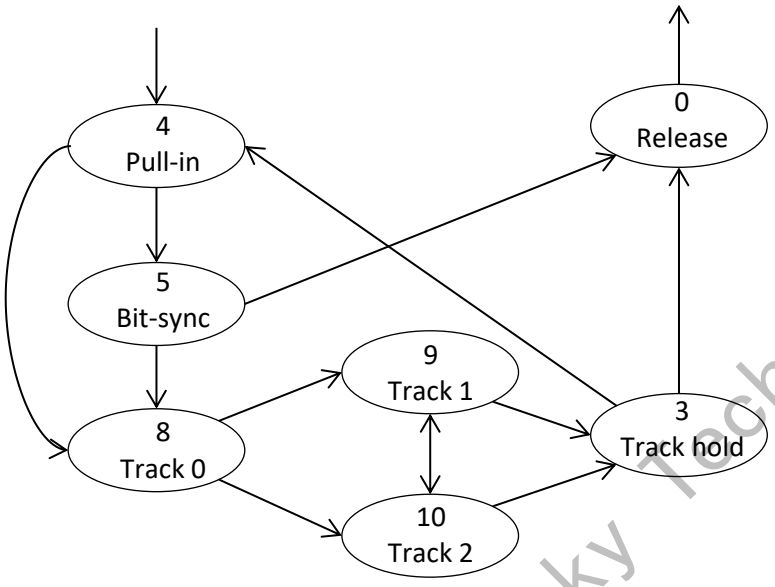
Cell with orange background means when PendingCor=7, two set of complete data need to be processed.

When OW flag set, it means the code Doppler is positive, current interrupt completes the coherent sum results, and CurrentCor increase by 1 (including from 7 to 0) so there is one extra correlation result. This one extra result must be the first correlation result during coherent period. So if CoherentNumber != 1, the correlation result in TE_OVERWRITE_PROTECT_VALUE need to be copied to the corresponding coherent sum position in state buffer. If CoherentNumber equals 1, this new correlation result should be treated as a new coherent sum result.

4. Signal tracking

4.1 Tracking stage and stage switch

The stages of signal tracking and switch between different stages are illustrated in following figure:



After a satellite that is not being tracked acquired by AE, an empty tracking channel is initialized using the acquired Doppler and code phase, and then enter the pull-in stage. The pull-in stage is then move forward to the bit-sync stage to begin the bit synchronization. If the bit synchronization fails, it will go to the release stage and the channel is released. If the bit synchronization succeeds, it will go into the track 0 stage to further have the tracking loop converge. Depend on the signal strength, it will either go into track 1 stage for strong signal or track 2 stage for weak signal. If the signal strength changes, track 1 and track 2 will also switch to each other. If signal loss is detected, the track hold state is entered, and waits for correlation peak to be detected again. If a correlation peak is detected, it will go to the pull-in stage again. If timeout occurs during tracking hold stage, it will go to release stage to release the channel. After the channel is released, it will be further determined whether to re-acquire the satellite that is not being tracked in the future.

For E1, B1C and L1C signal, because the data period is the same as code period, no bit synchronization is needed. The pull-in stage will go to track 0 stage directly.

The epoch of stage switch should align to the coherent sum period or bit edge. For L1C/A, this is done after bit synchronization. For other signals, the alignment is performed at channel initialization.

The following table gives the condition of stage switch for different signal (need to be optimized in the future):

	L1C/A	E1	B1C	L1C
Pull-in	To Bit-sync on timeout	To Track0 on timeout	To Track0 on timeout	To Track0 on timeout
Bit-sync	To Track0 if bit-sync succeed, otherwise to Release			
Track0	To Track1 if CN0>25, otherwise to Track2 on timeout	To Track1 after secondary code synchronization	To Track1 after secondary code synchronization	To Track1 after secondary code synchronization
Track1	To Track2 if CN0 is low			
Track2	To Track1 if CN0 is high			
Track hold				
Release				

4.2 Coherent number and shift bits

The hardware correlator and coherent sum results are 16bit width for both I and Q. To prevent overflow during correlation and coherent sum, pre-shift and post-shift bits need to be correctly configured. The suggested configuration value is calculated below:

Under the condition of optimal quantization for 4bit sign/mag format, with the gain of carrier Doppler mixed considered, the 1 millisecond noise amplitude will be around 625 for both I and Q (at sampling frequency of 4.113MHz). If maximum CN0 is 50, which means SNR is 20dB, the signal amplitude will be about $10 \times \sqrt{2} \times 625 \approx 8838$. So the 1 millisecond correlation result will not overflow even pre-shift bit set to 0.

If the maximum coherent sum period is 20ms, then the maximum coherent sum value will be about 22000 when post-shift bit configured at maximum value of 3. So the coherent sum results have enough margins using 16bit. If the coherent sum number is greater, pre-shift bit is suggested to set to 1 or larger value.

4.3 Bit synchronization and secondary code synchronization

Considering that different method may be adopted for bit synchronization, it is performed in baseband task instead of ISR. Each time the ISR collects 20 consecutive one millisecond correlation results, it sends the bit synchronization task to the baseband task queue together with the time tag aligned with the results. After the bit synchronization task is completed, the bit edge relative to the time tag is returned. Next ISR will adjust the coherent integration period starting from the bit synchronization position.

One simple bit synchronization method is now adopted. This method has fast synchronization speed on high power signal.

E1, B1C and L1C do not need bit synchronization. They will do secondary code synchronization instead. After the secondary code is synchronized, it can be removed by setting NH configuration and coherent time longer than PRN code period is possible.

4.4 Navigation data decode

After bit synchronization or secondary code synchronization, the navigation data decode begins. The symbol is quantized to 1bit for L1C/A, 4bit for E1 and 8bit for B1C and L1C to match the corresponding decode method.

For L1C/A, sign determination method is used for PLL tracking and bit toggle method is used for FLL tracking. Other signals will use PLL to track pilot channel and do data decode on data channel correlation results.

4.5 Phase, frequency and delay discrimination

The phase discrimination uses traditional arctan method implemented using CORDIC algorithm. The L1C/A uses 2-quandrant arctan and other signals use 4-quandrant arctan.

The fixed point CORDIC algorithm iterates 14 times to calculate result with enough accuracy. The output range is -32768 to 32767 to represent $-\pi$ to π. When adding or subtracting the angles, the overflow occurs on MSBs and 16LSB remains the range of $-\pi$ to π.

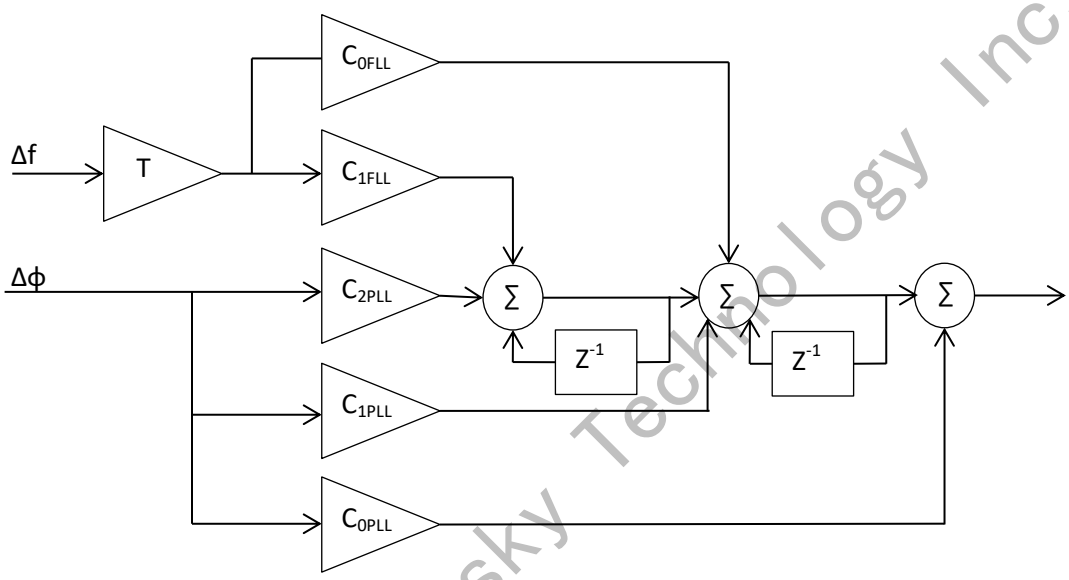
The frequency discrimination may use cross-dot method which is suitable for strong signal with large initial frequency bias, or use FFT method. The FFT method will do 8 point FFT on consecutive coherent sum results. If the number of results is less than 8, zero padding is applied. Interpolation is used on frequency bin with maximum amplitude and the amplitude at left and right bins. Either $\frac{\Delta f}{\Delta f_b} = \frac{\mathrm{L-R}}{\mathrm{2P-L-R}}$ or $\frac{\Delta f}{\Delta f_b} = \tan^{-1}\frac{\mathrm{L-R}}{\mathrm{2P-L-R}}$ can be used to do interpolation. The advantage of such equation is that the result is insensitive to the number of correlation results used in FFT. The discrimination result can be normalized using frequency bin interval Δf_b only.

The delay discrimination uses equation is $\mathrm{d} = \frac{\mathrm{E-L}}{\mathrm{2P-E-L}}$ which is insensitive to the shaped of the peak and correlator interval narrow factor.

In the above equations, all variables are amplitude. Square root calculation is used to calculate amplitude from signal power. Fixed point square root calculation method is described in Annex B.1.

4.6 Tracking loop and loop filter coefficients

The following figure shows a typical second-order FLL assisted third-order PLL loop filter



With given noise bandwidth Bn and update interval T, the loop filter coefficients can be calculated by following equations:

First order:

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

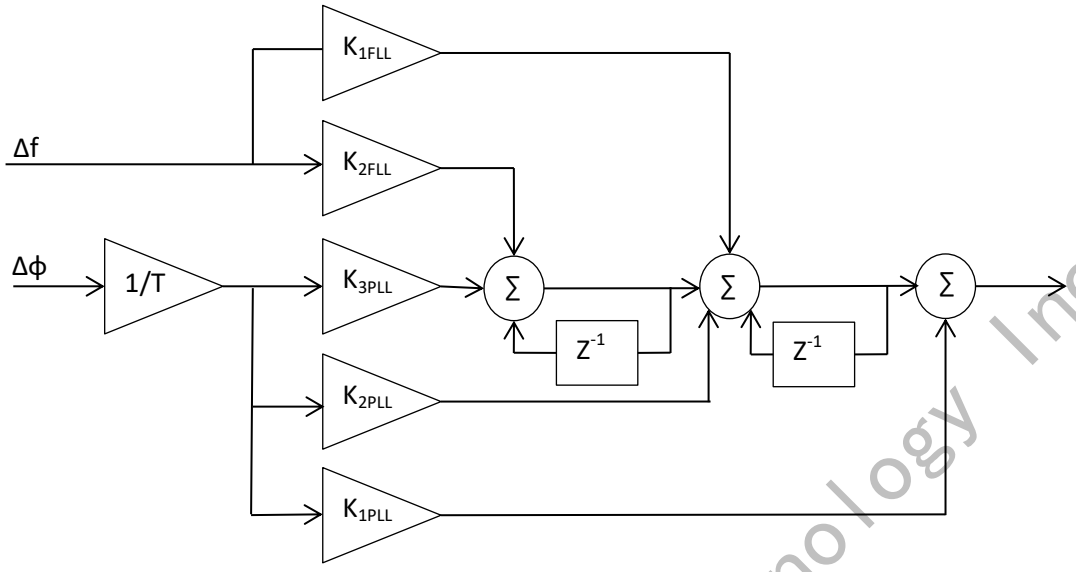
Second order:

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

Third order:

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

In order to facilitate the calculation of loop filter coefficients, the loop filter is changed to the following structure and calculation method:



The input frequency error and phase error are divided by T and combine this parameter into the loop filter coefficients equation:

First order:

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

Second order:

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

Third order:

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

It can be seen that the above coefficients are only depend on the product of the noise bandwidth Bn and the loop update interval T. Here BnT is a dimensionless quantity.

The above coefficients are calculated based on the principle of the continuous signal FLL/PLL. However, in a discrete system, the position of the poles in s plane is inconsistent to the poles in z plane. This will make the filtered result has error or even cause system instability especially for large BnT.

One solution is to use bilinear accumulation to replace direct accumulation. Another solution is to use the method described in “Controlled-Root Formulation for Digital Phase-Locked Loops” (by S. A. Stephens and J. B. Thomas). The following figure given by above paper can be used to calculate coefficients.

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			
$B_L T$	K_1	K_1	K_2	K_1	K_2	K_3	K_1	K_2	K_3	K_4
0.005	0.01976	0.01312	8.661e-05	0.01283	7.365e-05	1.590e-07	0.01165	6.831e-05	2.006e-07	2.951e-10
0.010	0.03918	0.02589	3.396e-04	0.02530	2.886e-04	1.242e-06	0.02299	2.673e-04	1.561e-06	4.572e-09
0.015	0.05822	0.03831	7.482e-04	0.03742	6.356e-04	4.084e-06	0.03399	5.879e-04	5.117e-06	2.237e-08
0.020	0.07689	0.05039	0.001302	0.04918	0.001106	9.429e-06	0.04468	0.001021	1.177e-05	6.827e-08
0.025	0.09520	0.06214	0.001992	0.06062	0.001691	1.794e-05	0.05506	0.001560	2.233e-05	1.610e-07
0.030	0.1132	0.07358	0.002810	0.07172	0.002383	3.020e-05	0.06516	0.002195	3.747e-05	3.225e-07
0.035	0.1308	0.08472	0.003746	0.08252	0.003175	4.674e-05	0.07496	0.002921	5.730e-05	5.774e-07
0.040	0.1481	0.09556	0.004793	0.09302	0.004060	6.800e-05	0.08450	0.003731	8.382e-05	9.522e-07
0.045	0.1651	0.1061	0.005944	0.1032	0.005032	9.438e-05	0.09377	0.004619	1.160e-04	1.474e-06
0.050	0.1818	0.1164	0.007191	0.1132	0.006085	1.262e-04	0.1028	0.005578	1.546e-04	2.173e-06
0.060	0.2143	0.1362	0.009950	0.1322	0.008410	2.075e-04	0.1201	0.007691	2.526e-04	4.218e-06
0.070	0.2456	0.1551	0.01302	0.1503	0.01099	3.137e-04	0.1365	0.01003	3.796e-04	7.321e-06
0.080	0.2758	0.1731	0.01637	0.1674	0.01380	4.460e-04	0.1521	0.01256	5.367e-04	1.171e-05
0.090	0.3051	0.1902	0.01995	0.1837	0.01679	6.055e-04	0.1669	0.01526	7.244e-04	1.761e-05
0.100	0.3333	0.2066	0.02372	0.1991	0.01995	7.924e-04	0.1809	0.01809	9.428e-04	2.523e-05
0.150		0.2788	0.04471	0.2657	0.03734	0.002135	0.2417	0.03356	0.002476	9.483e-05
0.200		0.3387	0.06740	0.3183	0.05595	0.004103	0.2899	0.04990	0.004650	2.270e-04
0.250		0.3912	0.09013	0.3606	0.07452	0.006583	0.3288	0.06604	0.007307	4.273e-04
0.300		0.4464	0.1108	0.3951	0.09238	0.009451	0.3606	0.08142	0.01030	6.937e-04
0.350				0.4241	0.1093	0.01259	0.3870	0.09580	0.01351	0.001020
0.400				0.4499	0.1253	0.01584	0.4093	0.1091	0.01683	0.001397
0.450							0.4282	0.1213	0.02021	0.001815
0.500							0.4446	0.1325	0.02359	0.002264
0.600							0.4726	0.1523	0.03034	0.003197

4.7 Iterated loop filter update method

The following equation gives the method to calculate frequency using phase error for third order PLL: $f = f_0 + k_1 \frac{\Delta\phi}{T} + k_2 \sum \frac{\Delta\phi}{T} + k_3 \sum\sum \frac{\Delta\phi}{T}$

Here f_0 is the initial frequency. If the order of the loop filter is less than 3, coefficients for higher order are 0. The above equation also applies for DLL with $\Delta\phi$ replaced by delay and the output is code frequency.

To simplify the calculation, we assume after the n-th update, we have the frequency $f_n = f_0 + k_1 \frac{\Delta\phi}{T} + k_2 \sum \frac{\Delta\phi}{T} + k_3 \sum\sum \frac{\Delta\phi}{T}$, then add an intermediate variable $f_n' = f_0 + k_2 \sum \frac{\Delta\phi}{T} + k_3 \sum\sum \frac{\Delta\phi}{T}$, we will get output frequency using $f_n = f_n' + k_1 \frac{\Delta\phi}{T}$ and iteration formula $f_n' = f_{n-1}' + k_2 \frac{\Delta\phi}{T} + k_3 \sum \frac{\Delta\phi}{T}$.

For second order FLL, the above equations are similar. The equation $f = f_0 + k_1 \sum \Delta f + k_2 \sum\sum \Delta f$ is replaced by its iteration form $f_n = f_{n-1} + k_1 \Delta f + k_2 \sum \Delta f$.

In the firmware, the loop filter coefficients and other gains are combined together. The coefficient k is calculated as $k = k_L \cdot k_C / k_D$, in which k_L, k_C and k_D are loop filter coefficient, frequency to FCW ratio and gain of discriminator respectively.

4.8 Fixed point tracking loop

In order to calculate the loop filter in fixed point, it is necessary to correctly select the scale factors of the loop parameters to provide sufficient quantization accuracy without overflow. Next, we will analyze loop filter coefficients determined by the range of B_nT in FLL/PLL/DLL, frequency to FCW ratio and discriminator gains one by one.

Generally, for stronger signals or in the pull-in stage, a shorter loop update period (integration time) is used, and a larger loop bandwidth is used. When entering stable tracking or tracking weak signals, the loop update period (integration time) is longer, but the loop bandwidth is smaller.

The following table gives typical integration time and bandwidth for FLL:

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

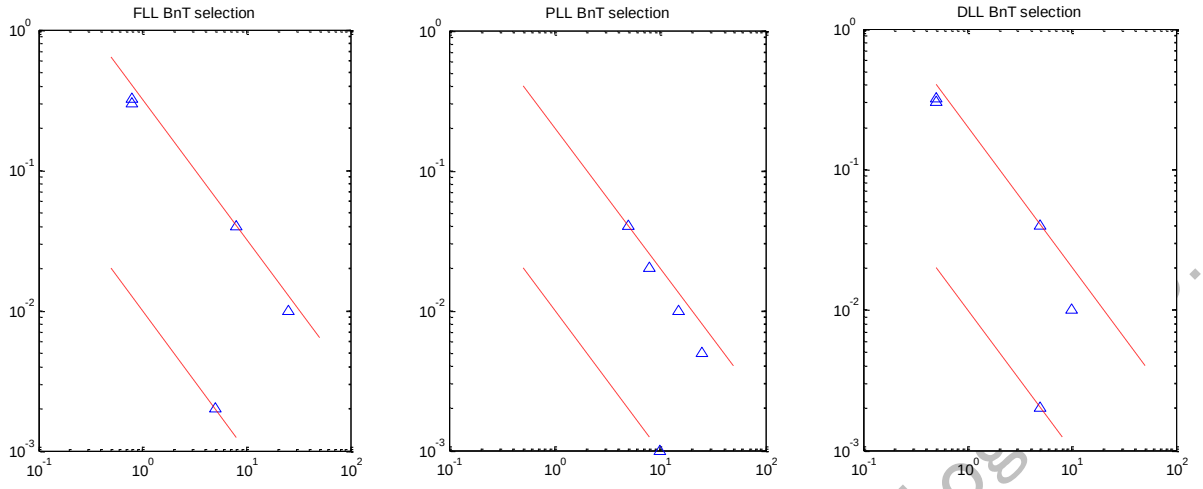
The following table gives typical integration time and bandwidth for PLL:

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

The following table gives typical integration time and bandwidth for DLL:

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

The parameters are illustrated in the following figures:



In above figures, red lines corresponding to the lower limits are all 0.01, and the red lines corresponding to the upper limits of FLL/PLL/DLL are 0.32, 0.2, and 0.2 respectively. Based on the upper and lower limits of B_nT, the range of the loop coefficient can be found in the table.

The FCW gain of the carrier and code is determined by the sampling rate. If the sampling rate is f_s, the corresponding frequency control word gain $k_C = \frac{2^{32}}{f_s}$. In this design, sampling rate is about 4 times the code rate, so the value of k_C is about 1050.

The phase discriminator calculates the angle of the arc tangent by the CORDIC algorithm. Since it needs to be multiplied by 1/T to convert to Hz, the unit of angle is cycles. For example, when the interval T=20ms, the angle difference is 1/10 cycle, then the frequency sent to loop filter is $\Delta f = \frac{\Delta\phi}{T} = \frac{0.1}{0.02} = 5\text{Hz}$. The output of CORDIC uses 216 to represents one cycle, so the phase detector has gain $k_D = 2^{16}T_c$, in which T_c is the coherent integration time.

The frequency discriminator has different gains for dot-cross product discrimination and FFT discriminator. For dot-cross product discriminator, we have $\Delta f = \frac{1}{T_c}\tan^{-1}\frac{S_1 \times S_2}{S_1 \cdot S_2}$. Similarly, the angle gain output by the CORDIC method is 216, so the phase detector gain $k_D = 2^{16}T_c$. For FFT discrimination, it is generally calculated by 8-point FFT. At this time, the total frequency range between the 8 frequency bins is $\frac{1}{T_c}$, and the frequency difference between each frequency bin is $\frac{1}{8T_c}$. If the difference discrimination formula adopted by the frequency discriminator is $\tan^{-1}\frac{L-R}{2P-L-R}$, in which L and R represent the signal amplitudes of the two frequency bins on the left and right of the peak, respectively, then the maximum frequency offset from the center frequency bin is ±1/2 bin when P=L or P=R. Therefore, the range of $\frac{L-R}{2P-L-R}$ is between ±1, and the corresponding inverse tangent output range is ±1/8 cycle which is ±8192. Therefore, the demodulator output range corresponding to one frequency bin is 16384, and the corresponding demodulator gain is $16384 \times 8T_c = 2^{17}T_c$. In order to unify the gains with other frequency and phase demodulators, the output can be right shifted by one bit to get the same gain $k_D = 2^{16}T_c$. It

should be noted that if the maximum frequency bin is not at the center frequency, 8192 needs to be compensated for each deviation of one bin.

The delay discriminator uses $\frac{E-L}{2P-E-L}$ to calculate the delay, in which E, P and L represent the signal amplitudes of the early, prompt and late correlators respectively. When the range of the correlator interval deviates from the peak correlator is ±1/2 correlator interval, the output range of the above formula is ±1. That is to say, when the correlator interval is 1/2 chip, 1/4 chip and 1/8 chip, the discriminator gain is 2, 4 and 8 times the half chip respectively. It should be noted that the unit used in delay discriminator output is half chip, corresponding to the code NCO nominal frequency at 2 times of code rate. In order to ensure the output of the effective digits and the use of integer division for the discriminator output, the numerator E-L is shifted left by 13, 12 and 11 bits respectively for different interval ration before the division. Therefor the discriminator gain becomes 2^{14}. If the maximum peak is not on the desired peak correlator (Correlator 4), each deviation from a correlator requires corresponding compensation of 2^{14}, 2^{13} and 2^{12} respectively. When the delay discriminator outputs to the loop filter, it also needs to be divided by the loop update period or the overall accumulation time T. Therefore, the delay discriminator output gain $k_{\mathrm{D}} = 2^{14}T$.

Based on above analysis, the overall loop filter coefficients k is determined with calculated k_{C}, k_{D} and the range of k_{L}. In most of case, we will use second or third order PLL, second order FLL and second order DLL or first order DLL with carrier aiding.

To calculate the loop filter coefficients of FLL/PLL $k = k_{\mathrm{L}} \cdot k_{\mathrm{C}}/k_{\mathrm{D}}$, by substituting k_{C} and k_{D} we get $k = k_{\mathrm{L}} \cdot \frac{2^{32}}{f_s} \cdot \frac{1}{2^{16}T_c} = k_{\mathrm{L}} \cdot \frac{2^{16}}{f_s T_c}$. Here $f_s T_c$ represents the number of sampling points within coherent integration period, which ranges from about 4000 to 200000 or 2^{12}~2^{18}. Referring to the k_{L} value found in the loop parameter table, it can be judged that the range of k_1 is about 2^3 to 2^{-4}. The range of k_2 is about 2 to 2^{-7}. Considering that the loop bandwidth is usually wider when the dynamics are large in most scenarios, and the integration time is longer when the dynamics are small, or the third-order loop is not required. So we ignore the case of using third order PLL when $B_n T$ is below 0.05. Therefor the range of k_3 is about 2^{-8} to 2^{-12}. The above k values cannot be expressed by integer values, they needs to be enlarged with certain scaling factor to get scaled coefficients K.

Considering the output of the phase discriminator ranges from -32768 to 32767 and will not overflow with maximum gain of 2^{16}, so k_1 is determined to enlarged by 2^{13}, which means $K_1 = 2^{13}k_1$ will be less than 2^{16}. Similarly, k_2 is determined to enlarged by 2^{15} to ensure the value to be less than 2^{16}, which means $K_2 = 2^{15}k_2$. Since k_3 is the coefficients of accumulated phases and the accumulation is assumed to be less than 2^9 cycles, then the accumulated value will not overflow when multiply with a value less than 2^7. So k_3 can be enlarged by 2^{15} to have $K_3 = 2^{15}k_3$ be less than 2^7.

Finally, the FCW can be calculated using $W = W_n + (K_1 \cdot P_n) \gg 13$ with the n-th phase discriminator fixed point output be P_n. And the formula to calculate W_n iteratively is $W_n = W_{n-1} + (K_2 \cdot P_n + K_3 \cdot ACC_n) \gg 15$, in which $ACC_n = ACC_{n-1} + P_n$.

If $f_s = 4.113MHz$, $T_c = 20ms$ and $B_n = 7.5Hz$, and using third order PLL, we have BnT=0.15, $f_sT_c = 82260$. Then we can calculate $K_1 = 0.2657 \cdot \frac{2^{16}}{f_sT_c} \cdot 2^{13} = 1734$, $K_2 = 0.03734 \cdot \frac{2^{16}}{f_sT_c} \cdot 2^{15} = 975$ and $K_3 = 0.002135 \cdot \frac{2^{16}}{f_sT_c} \cdot 2^{15} = 56$.

The MATLAB program in Annex A.1 gives an example of comparing floating point and fixed point results of a PLL loop filter.

The calculation of the output range, discriminator gain, and coefficient range of the FLL discriminator is similar to that of the PLL and will not be repeated here. The method for calculating the frequency control word is $W_n = W_{n-1} + (K_1 \cdot P_n + K_2 \cdot ACC_n/4) \gg 13$, $ACC_n = ACC_{n-1} + P_n$ with the n-th phase discriminator fixed point output be P_n. The MATLAB program in Annex A.2 gives the calculation process of the FLL fixed-point FCW with a 1ms coherent integration time and a 20Hz bandwidth dot-cross product discriminator.

For DLL, by substituting k_C and k_D, we get $k = k_L \cdot \frac{2^{32}}{f_s} \cdot \frac{1}{2^{14}T} = k_L \cdot \frac{2^{18}}{f_sT}$. Since the range of f_sT is about to between 2^{13} to 2^{21}, the range of k_1 will be within 2^1 to 2^{-7}, the range of k_2 will be within 2^{-3} to 2^{-10}. The gain of k_1 and k_2 are both set to 2^{15}, then the FCW to code NCO is $W = W_0 + (K_1 \cdot D_n + K_2 \cdot ACC_n) \gg 15$, $ACC_n = ACC_{n-1} + D_n$ with the n-th phase discriminator fixed point output be D_n and initial code NCO FCW value be W_0. The MATLAB program in Annex A.3 gives an example of comparing floating point and fixed point results of a DLL loop filter using 5ms coherent integration, 8 times non-coherent integration and 2Hz bandwidth.

To simplify the calculation of enlarged coefficients, a lookup table of k_n enlarged $\frac{2^{33}}{f_s}$ at different B_nT is established. For PLL and FLL, $K_1 = 2^{13}k_1 = (k_{L1} \cdot \frac{2^{33}}{f_s}) \cdot \frac{1}{16T_c}$, $K_2 = 2^{15}k_2 = (k_{L2} \cdot \frac{2^{33}}{f_s}) \cdot \frac{1}{4T_c}$ and $K_3 = 2^{15}k_3 = (k_{L3} \cdot \frac{2^{33}}{f_s}) \cdot \frac{1}{4T_c}$. For DLL, $K_1 = 2^{15}k_1 = (k_{L1} \cdot \frac{2^{33}}{f_s}) \cdot \frac{1}{T}$, $K_2 = 2^{15}k_2 = (k_{L2} \cdot \frac{2^{33}}{f_s}) \cdot \frac{1}{T}$.

4.9 Calculating C/N0

The method for calculating C/N0 is to calculate the ratio between the signal energy and the noise floor both normalized to 1ms. The distribution of the signal energy and the noise energy is introduced below.

The calculation of the noise floor is automatically performed by the hardware. According to the hardware function, the smoothed noise floor value $NF = \sigma\sqrt{\frac{\pi}{2}}$ read from the register can be

obtained, where σ is the noise variance of either I or Q. So the overall noise energy within single sideband is $N_0 = \sigma^2 = 2 \cdot NF^2/\pi$.

Since the samples entering the noise floor calculation unit are the down-converted data output by the first logical channel with pre-shift, so the noise floor value itself is related to the pre-shift setting. It is recommended that all channels use the same pre-shift setting, so that no conversion is required when calculating C/N0. At the same time, since the initial noise floor value configured by the hardware noise floor calculation unit is calculated based on the pre-shift with value 0 under the optimal quantization of white noise, if other pre-shift value is set, it is recommended to set the initial noise floor value during initialization to reduce the convergence time.

Since the calculation method of signal energy is related to factors such as integration time, if the signal energy within 1ms is S_0, then the total energy obtained by the final non-coherent accumulation is $P = \frac{(S_0 \cdot n_c^2 + N_0 \cdot n_c) \cdot n_n}{2^{2B+F}}$. Within which n_c is the number of coherent integration time (including the product of the number of coherent accumulations and the number of FFT points), and n_n is the number of non-coherent integration time, B is the sum of the number of bits of pre-shift and post-shift, F is 6 for 8-point FFT (amplitude right shift of 3 bits corresponds to right shift 6 bits on energy) and is 0 without FFT.

In the above formula, the signal energy proportional to the square of the number of coherent accumulations and proportional to the number of non-coherent accumulations; the noise energy is proportional to both the number of coherent accumulations and the number of non-coherent accumulations.

Since S_0 and N_0 are the signal energies normalized to 1ms, which mean $\frac{S_0}{N_0}$ is the signal-to-noise ratio under 1kHz bandwidth, the formula to calculate C/N0 is $\mathrm{CN0} = 30 + 10\lg\frac{S_0}{N_0}$. We will first calculate $N = \frac{N_0 \cdot n_c \cdot n_n}{2^{2B+F}}$ with the value N_0, then calculate $S = \frac{S_0 \cdot n_c^2 \cdot n_n}{2^{2B+F}} = P - N$, and finally get $\frac{S_0}{N_0} = \frac{S}{n_c \cdot N}$.

Annex B.2 gives a method of fixed point logarithm calculation.

4.10 Lock indicator

The PLL, FLL and DLL each have their own lock indicator ranging from 0 to 100. The larger the lock indicator means the more stable the signal is locked. The lock indicator is calculated using the output of discriminator: the discriminator output is first right shifted to get a normalized adjustment value A, and the lock indicator is updated according to the value of A with the following table.

A	Change to Indicator
0	+6
1	+4

A	Change to Indicator
2~3	+2
4~7	+1
>=8	-A/8

After update, the lock indicator is further clipped within range of 0 to 100.

5. Raw measurement calculation and frame synchronization

5.1 Baseband measurement

The raw measurement is calculated based on the baseband observation, that is, the count value of each baseband counter at the observation time. The baseband observation usually involves the following values, most of which are extracted by the baseband register or maintained by the baseband control software:

- ✓ FCW of local carrier as instant frequency of local carrier
- ✓ Carrier NCO as the phase of local carrier
- ✓ Cycle count of local carrier
- ✓ FCW of code
- ✓ Counter of (half) code chips, including count of code periods within one navigation bit
- ✓ Code NCO as fraction of code chips
- ✓ Tracking state, lock indicator etc.

Among them, the half-chip count and code NCO are key data for calculating the signal transmission time. Since signal tracking uses the method of aligning the signal with the local code, the generation time of the local code will be used as the signal transmission time on the observation epoch instead. At this time, the transmission time of the signal is composed of the following parts: TOW (which is the subframe count within a week), the bit count within the frame, the half-chip count within a bit, and the code NCO count within a half-chip. The transmission time is the sum of the above parts. The counter values of the first two are introduced in the section on frame synchronization, and the code NCO count within the half-chip is directly the value of the code NCO register.

In one code period, the count value of a half-code chip is twice the code count (PrnCount count value) plus CodeSubPhase. If the coherent integration time exceeds one code cycle, the number of coherent accumulations needs to be added. It should be noted here that the increment of both DumpCount and PrnCount is aligned with Cor0, but the increment of the number of coherent accumulations is aligned with Cor7, so if CurrentCor is not 0, it means that the number of coherent accumulations in the CoherentCount register is one less than the number of coherent accumulations corresponding to Cor0, so 1 needs to be added for compensation. In addition, the local code calculated in this way is the local code corresponding to Cor0, and 4 half-code chips need to be subtracted to obtain the local code phase corresponding to Cor4.

5.2 Local time calculation and maintenance

Before introducing the calculation of raw measurements, we first introduce the maintenance of local time. Due to the existence of clock errors, local time may not be the exact observation time, but a time close to the observation time that is artificially defined. For convenience, local time can be aligned to whole milliseconds, whole 20 milliseconds, or even whole seconds.

The initial value of the local time can be initialized from the TOW value obtained from the frame synchronization. When the frame synchronization is completed, the transmission time of the satellite signal can be obtained through TOW, bit count and code phase. On the basis of this time, the average propagation time of the satellite signal (about 75 milliseconds for GPS) will be added to roughly estimate the local time. Align the roughly estimated local time to a certain integer value, and then initialize the local time with this value.

The calculation of the local time is based on the timing of baseband processing. Generally speaking, the data length of a block set by the baseband TE FIFO is nominally 1 millisecond, and the Measurement Count is counted according to the number of processed block data. Therefore, the Measurement Number is defined as the time length between two observation times (or epochs) calculated according to the nominal value. The maintenance of the local time is to add the epoch interval to the local millisecond count value, so as to realize the maintenance of the local time calculated according to the RF sampling clock.

The local time maintenance combined with the PVT solution involves two epoch intervals, one is the actual epoch interval which is the actual value set in the MeasurementNumber register, and the other is the nominal epoch interval. The local time is calculated using the nominal epoch interval. By adjusting the values of these two epoch intervals, the whole second alignment of the observation time and the clock error adjustment can be achieved.

First, let's introduce the whole second alignment of the observation time. Generally, the output of the receiver position is required to be aligned to the whole second in time (for the case where the position output frequency is higher than 1Hz, one of the moments needs to be aligned to the whole second). However, due to the uncertainty of the receiver startup time, the initial observation time may be far from the whole second boundary, so the observation time needs to be adjusted to be as close to the whole second as possible. Since the register MeasurementNumber is limited to be within 10bits, this alignment is performed by adjusting the MeasurementNumber to a value smaller than 1000 to make the observation time to be advanced. At this time, the actual epoch interval and the nominal epoch interval use the same value. In order to avoid the measurement interval being too small, the value of the MeasurementNumber is usually clipped to between 800 and 1000. The process of above adjustment can be performed multiple times to achieve larger adjustment value. With the actual epoch interval and the nominal epoch interval using the same value, the local time and the actual observation time changed by the same amount, which means there is no jump on the pseudorange, and the clock error will not change.

The source of the clock error is the estimation error when the local time is initialized, and the other comes from the accumulation of local clock drift. In addition to adjusting the local time to whole seconds, it is usually necessary to make the local clock error of the receiver as small as possible. After the local clock error is obtained through PVT calculation, the clock error is rounded to the nearest millisecond to obtain the number of milliseconds for the clock error adjustment. The MeasurementNumber is adjusted by adding the value of 1000 to the number of milliseconds for the clock error adjustment. At the same time, the actual epoch interval is taken

as the value set into MeasurementNumber, and the nominal epoch interval remains unchanged at 1000. Since the local time is increased to the nominal epoch interval, the updated local time remain aligned to the boundary of whole second, and the actual observation time is adjusted to be within 1 millisecond of the local time due to the adjustment of the MeasurementNumber. In this case, since the clock error has changed, the corresponding pseudorange has also jumped.

5.3 Calculation of raw measurements

The raw measurements of one satellite usually include at least the following values: the pseudorange, instant Doppler and carrier phase (or represent as accumulative Doppler with whole carrier cycle count included).

With the signal transmission time obtained, the unadjusted pseudorange can be calculated simply by multiply the light speed with the difference of local time and transmission time: $\rho = (T_r - T_t) \cdot c$.

The instant Doppler is the local carrier frequency minus the nominal IF.

The calculation of accumulated Doppler range (ADR) is relatively more complicate. First, the integration of Doppler between two epochs can be expressed as $\Delta\Phi = \int_{t_1}^{t_2} f_D dt = \int_{t_1}^{t_2} (f_{LO} - f_{IF}) dt = \int_{t_1}^{t_2} f_{LO} \cdot dt - \int_{t_1}^{t_2} f_{IF} \cdot dt$, within which f_{LO} is the local carrier frequency and f_{IF} is nominal IF. Because the local carrier frequency is integrated into cycle count and carrier NCO, so the integration value can be further expressed as $\Delta\Phi = N_2 + \frac{NCO_2}{2^{32}} - (N_1 + \frac{NCO_1}{2^{32}}) - N_{IF}$. Here N_1 and N_2 are cycle count at the previous epoch and the current epoch; NCO_1 and NCO_2 are NCO value of these two epochs respectively; N_{IF} is the nominal IF multiply time interval (usually be an integer value if nominal IF is integer kHz and time interval is integer milliseconds).

The ADR is usually corresponds to the pseudorange, and positive Doppler reduces the pseudorange, so the ADR difference between two epochs is defined as opposite value of integrated Doppler: $\widehat{\Phi}_2 - \widehat{\Phi}_1 = -\Delta\Phi$. ADR is initialized by the value $\widehat{\Phi}_0 = \widehat{N}_0 - \frac{NCO_0}{2^{32}}$ in which $\widehat{N}_0$ is selected to have $\widehat{\Phi}_0$ be close to the pseudorange at that epoch. The ADR of following epochs are calculated by first iteratively calculating $\widehat{N}_k = \widehat{N}_{k-1} - [(N_k - N_{k-1}) - N_{IF}]$, and then used $\widehat{\Phi}_k = \widehat{N}_k - \frac{NCO_k}{2^{32}}$ to get ADR of epoch k.

5.4 Frame synchronization and frame decode for signals with pilot channel

For satellite signals with pilot channels, the first step of frame synchronization is completed in the baseband. At this time, data decoding function will first decode the modulation data of the pilot channel and align the secondary code of the pilot channel according to the modulation data of the pilot channel. At this time, the data modulation on the pilot channel can be stripped by setting the enable NH, the phase is aligned and a four-quadrant phase detector is used. Data decoding uses the second code generator to decode the navigation message of the data channel.

For B1C and L1C, the repetition period of the navigation channel message is 1800 bits. The synchronization of the navigation message will first slide the most recently received 16-bit data in the 1800-bit secondary code. When the matching position is found, the next 16-bit data is used for confirmation. The above search and confirmation steps are completed in the baseband control task queue.

After completing the synchronization of the secondary code, the subsequent decoded message of the data channel has a known position of the current data in the message frame, so the frame parsing program can store the received navigation message in the corresponding position to piece together a complete frame. The frame parsing task and the position solution are run in the same task queue. In each observation interruption, the frame parsing task is first added, and then the observation quantity calculation and position solution tasks are added. Since the tasks in the same task queue are run serially in the order in which they are added, the parsing of the navigation message will be completed before the PVT, that is, the message parsing task will update the ephemeris and other parameters earlier than the PVT, and will not be modified during the PVT operation.

For the navigation messages of B1C and L1C, after piecing together a complete frame, the message parsing task first decodes subframe 1, and then deinterleaves subframes 2 and 3. If hardware LDPC decoding is required, the task needs to be suspended and wait for the hardware to run (if the LDPC decoding time is long, it is necessary to consider better task arrangement to reduce the software waiting time).

5.5 GPS LNAV frame synchronization and frame decode

Compared with other satellite signals with pilot channels, the frame synchronization on GPS L1C/A signals is relatively more complicated. For satellite systems with pilot signals, since the data modulated frame is synchronized with the Secondary Code on the pilot channel, the frame start position of the data channel can be determined by synchronizing the Secondary Code, and then parsing it. The LNAV navigation message of GPS L1C/A code can only be synchronized by parsing the navigation message.

GPS's LNAV message is composed of WORDs. Each 30-bit WORD plus the last two bits of the previous WORD constitutes a unit that can be verified, and then 10 WORDs constitute a subframe. The data structure and verification method of each WORD are shown in the figure below (IS-GPS-200K page 128 Table 20-XIV):

D_1	=	$d_1 \oplus D_{30}^{\star}$
D_2	=	$d_2 \oplus D_{30}^{\star}$
D_3	=	$d_3 \oplus D_{30}^{\star}$
•		•
•		•
•		•
•		•
D_{24}	=	$d_{24} \oplus D_{30}^{\star}$
D_{25}	=	$D_{29}^{\star} \oplus d_1 \oplus d_2 \oplus d_3 \oplus d_5 \oplus d_6 \oplus d_{10} \oplus d_{11} \oplus d_{12} \oplus d_{13} \oplus d_{14} \oplus d_{17} \oplus d_{18} \oplus d_{20} \oplus d_{23}$
D_{26}	=	$D_{30}^{\star} \oplus d_2 \oplus d_3 \oplus d_4 \oplus d_6 \oplus d_7 \oplus d_{11} \oplus d_{12} \oplus d_{13} \oplus d_{14} \oplus d_{15} \oplus d_{18} \oplus d_{19} \oplus d_{21} \oplus d_{24}$
D_{27}	=	$D_{29}^{\star} \oplus d_1 \oplus d_3 \oplus d_4 \oplus d_5 \oplus d_7 \oplus d_8 \oplus d_{12} \oplus d_{13} \oplus d_{14} \oplus d_{15} \oplus d_{16} \oplus d_{19} \oplus d_{20} \oplus d_{22}$
D_{28}	=	$D_{30}^{\star} \oplus d_2 \oplus d_4 \oplus d_5 \oplus d_6 \oplus d_8 \oplus d_9 \oplus d_{13} \oplus d_{14} \oplus d_{15} \oplus d_{16} \oplus d_{17} \oplus d_{20} \oplus d_{21} \oplus d_{23}$
D_{29}	=	$D_{30}^{\star} \oplus d_1 \oplus d_3 \oplus d_5 \oplus d_6 \oplus d_7 \oplus d_9 \oplus d_{10} \oplus d_{14} \oplus d_{15} \oplus d_{16} \oplus d_{17} \oplus d_{18} \oplus d_{21} \oplus d_{22} \oplus d_{24}$
D_{30}	=	$D_{29}^{\star} \oplus d_3 \oplus d_5 \oplus d_6 \oplus d_8 \oplus d_9 \oplus d_{10} \oplus d_{11} \oplus d_{13} \oplus d_{15} \oplus d_{19} \oplus d_{22} \oplus d_{23} \oplus d_{24}$

Where

$d_1, d_2, ..., d_{24}$ are the source data bits;

the symbol ★ is used to identify the last 2 bits of the previous word of the subframe;

$D_{25}, D_{26}, ..., D_{30}$ are the computed parity bits;

$D_1, D_2, ..., D_{29}, D_{30}$ are the bits transmitted by the SV;

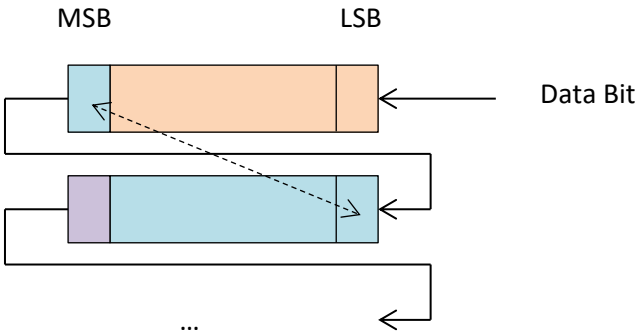
⊕ is the "modulo-2" or "exclusive-or" operation.

In the above figure, D_1 to D_{30} are actual bits that being transmitted. As the carrier tracking loop uses Costas loop, it is possible that received bit stream is opposite. The parity check method shown in above figure has following characteristics:

- ✓ The parity check involves totally 32bits
- ✓ An opposite bit stream will have same parity check result (pass or not pass)
- ✓ Navigation bits d_1 to d_{24} can still be recovered from the opposite bit stream
- ✓ Contents of each WORD will affect the value of D_{30} and consequently affect the transmitted bits in the next WORD

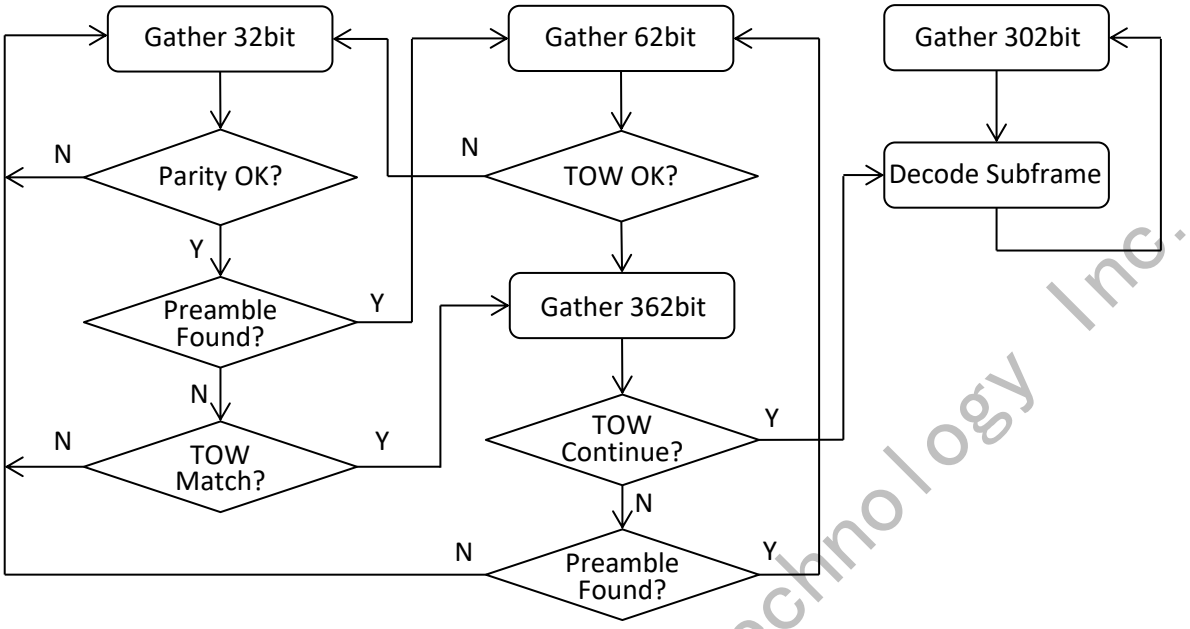
According to the second and third characteristics mentioned above, the steps of verifying WORD and parsing the navigation message can be independent to the step of determining the code stream polarity. At the same time, according to the fourth characteristic, the last two bits of WORD2 and WORD10 are forced to be zero when designing the navigation message, so as to ensure that the changing value of TOW in WORD1 will not affect the transmitted bits of WORD3~WORD10, and the transmitted bits content of each subframe will not be affected by the content of the previous subframe.

According to the first characteristic mentioned above, the input bit stream data is organized in the following way:



The storage of the bit stream is organized according to a 32-bit DWORD array. The squares in each row above represent a DWORD, and from top to bottom they represent the array with ascend index. When a new data bit is input, the lowest bit of digital address 0 is shifted in, and at the same time, each bit shifted out from the high bit of the DWORD enters the low bit of the next DWORD. In addition, the high bit shifted out from the high bit refers to bit29 instead of bit31. In this way, after shift bit29 goes to both bit30 of the same DWORD and bit0 of the next DWORD. This method ensures that the highest 2 bits of each DWORD are the same as the lowest 2 bits of the next DWORD and when 30LSB of each DWORD aligns with a 30bit WORD of a navigation message, the 2MSB holds D_{29} and D_{30} of the previous WORD. Then parity check can be performed using the contents of one 32bit DWORD.

Perform a parity check on the above bit stream. If several consecutive DWORDs pass the check, it can be considered that the edge of the WORD has been determined. However, in order to achieve frame synchronization, it is also necessary to find the preamble, that is, to find the pattern of 10001011 or 01110100 in bit29 to bit22 of the DWORD (a more strict check will add additional two 0 bits of the previous WORD at position bit31 and bit30. In the GPS LNAV navigation message, the preamble has only 8 bits, so it is possible that the same pattern appears in the first 8 bits of a WORD in the navigation message, and a frame synchronization error will occur. The receiver needs to perform frame synchronization as soon as possible to complete the calculation of local time and raw measurement after obtaining TOW. In order to take into account both efficiency and reliability of frame synchronization, the frame synchronization processing steps are carried out as follows:



First, align the WORD, that is, accumulate 32 bits, and then perform a parity check. If the check fails, move in a new bit and check again. If the check passes, find the preamble to synchronize to TLM. Here, in order to reduce the probability of matching the wrong position, the synchronization word contains the last two bits of the previous subframe, that is, check the 10 bits 0010001011 or their inverse sequence. If there is a more accurate local time, you can also directly compare TOW to synchronize to HOW. When the synchronization word check passes, 62 bits are accumulated, and the newly added 30 bits will contain HOW, that is, TOW can be parsed from it. At this time, the calculation of local time and the calculation of original observations can be completed. The next step is to continue to accumulate to 362 bits, that is, the content of 12 consecutive WORDs. At this time, these 12 WORDs should contain a complete subframe and the TLM and HOW of the next subframe. At this time, you can check whether the preamble in the TLM and the TOW in the two HOWs are continuous. If the test passes, it is confirmed that the frame synchronization has been completed, and the subsequent step is to parse the subframe and accumulate the data of the next subframe. If the test fails, the preamble is searched in the subsequent WORD, and the next step is determined based on the search result.

The analysis of the subframe will first determine the subframe number. If it is subframe 4 or 5, the almanac, UTC parameters, etc. will be analyzed. If it is subframe 1, 2, or 3, after collecting all three subframes, determine whether their respective IODC and IODE are consistent, and then perform the message decoding. Since the LNAV ephemeris information does not contain the satellite number, in order to prevent the wrong satellite ephemeris from being parsed due to cross-correlation, an additional consistency check of the ephemeris and almanac can be performed when a valid almanac exists. It mainly determines whether the two parameters of the right ascension of the ascending node and the position of the satellite relative to the ascending node at the reference time are close.

6. Position fix

In order to reduce data copying and moving, since raw measurement calculation and position fix run in the same thread as the front-end and back-end of PVT, they share the same global variables, including ephemeris, almanac, satellite status information, receiver positioning results, and core data of position solution. These global variables are defined in GlobalVar.h and instantiated in GlobalVar.c.

The steps of position solution are as follows: First, the raw measurements are screened, and the satellites with valid measurements and valid ephemeris are placed in the pointer array, and the measurements of the same satellite system are placed together; the second step is to calculate the corresponding information of the screened satellites, including calculating the satellite position at the transmitting epoch, and calculating the satellite elevation and azimuth according to the satellite position and the receiver position. At the same time, the satellite clock correction and relativistic correction related to the ephemeris are also completed in this step; the third step is to filter the original observations, including removing repeated measurements, removing measurements with elevation angles and CN0 less than the threshold, etc.; the fourth step is to calculate the satellite's troposphere and ionosphere corrections, and correct the original pseudorange after combining the clock error and relativistic correction to obtain the corrected pseudorange; the fifth step is to determine the positioning method based on the measurements, such as 2D or 3D least square positioning, Kalman filter positioning or 5-star positioning, etc.; the sixth step is to call the corresponding position calculation function according to the positioning method for position solution; finally, the solution result is assigned to the receiver positioning result and the corresponding flag is set.

In order to be compatible with the Kalman filter operation process and compatible with different numbers of satellite systems, the position solution results are arranged in the core data structure of the position solution in the order of three-dimensional speed, clock drift, three-dimensional position, and clock error of each different satellite system. In this way, when more satellite systems are added, the number of clock errors can be expanded backward without affecting the order of the previous data arrangement.

6.1 Least square positioning

According to the least square measurement equation

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

The position error can be calculated with

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

First, the vector $\mathbf{H}^{\mathrm{T}} \cdot \Delta\rho$ is calculated, and then calculate square matrix $(\mathbf{H}^{\mathrm{T}}\mathbf{H})^{-1}$, then multiply the matrix to the vector to get position error (and clock error as well). Because matrix $\mathbf{H}^{\mathrm{T}}\mathbf{H}$ is a symmetric matrix, so Cholesky decomposition is used to calculate its inverse matrix.

Multiple iterations may be required for final position solution. After the position is calculated, the velocity calculation can be performed using the same H matrix without the need for further iterations.

6.2 Kalman filter

The Kalman filter positioning uses a constant velocity model, and the state vector $X = [v_x\ v_x\ v_x\ \dot{t}\ x\ y\ z\ \delta t_{GPS}\ \delta t_{BDS}\ \delta t_{Galileo}]^T$, so the state transition matrix is:

$$\mathbf{\Phi} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \Delta t & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \Delta t & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \Delta t & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta t & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & \Delta t & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \Delta t & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

According to the dynamic of the receiver, the matrix of the state noise can be presented as

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_p \Delta t & \mathbf{0} & \frac{1}{2}\mathbf{Q}_p \Delta t^2 & \mathbf{0} \\ \mathbf{0} & \mathrm{Q}_f \Delta t & \mathbf{0} & \frac{1}{2}\mathbf{Q}_f \Delta t^2 \\ \frac{1}{2}\mathbf{Q}_p \Delta t^2 & \mathbf{0} & \frac{1}{3}\mathbf{Q}_p \Delta t^3 & \mathbf{0} \\ \mathbf{0} & \frac{1}{2}\mathbf{Q}_f \Delta t^2 & \mathbf{0} & \frac{1}{3}\mathbf{Q}_f \Delta t^3 \end{bmatrix}$$, in which $\mathbf{Q}_p = \begin{bmatrix} Q_{xx} & Q_{xy} & Q_{xz} \\ Q_{xy} & Q_{yy} & Q_{yz} \\ Q_{xz} & Q_{yz} & Q_{zz} \end{bmatrix}$ are three dimensional dynamic noise model and can be calculated from horizontal and vertical dynamics of the receiver. Q_f is the error from receiver clock jitter.

Based on above matrixes, the prediction equations of Kalman filter are:

$$\mathbf{X}_n^- = \mathbf{\Phi}\mathbf{X}_{n-1}^+$$

$$\mathbf{P}_n^- = \mathbf{\Phi}\mathbf{P}_{n-1}^+\mathbf{\Phi}^T + \mathbf{Q}$$

Because the matrix $\mathbf{\Phi}$ is a sparse matrix most of the elements has value 1, so the update of P matrix can be greatly simplified.

The update of Kalman uses sequential update method. Only one pseudorange or Doppler measurement is used each time to avoid do matrix inversion. For different satellite system, the pseudorange observation matrix is $\mathbf{H} = [0\ 0\ 0\ 0\ r_x\ r_y\ r_z\ \delta_{GPS}\ \delta_{BDS}\ \delta_{Galileo}]$, in which r_x,r_y and r_z are unified line of sight vector, $\delta_{GPS}/\delta_{BDS}/\delta_{Galileo}$ has value of 1 for corresponding system, otherwise 0. The Doppler observation matrix is $\mathbf{H} = [r_x\ r_y\ r_z\ 1\ 0\ 0\ 0\ 0\ 0\ 0]$.

One each update, innovation $\mathrm{d} = \mathrm{z} - \mathbf{HX}$ is calculated at first. The innovation is a scalar for sequential update. And then the Kalman filter gain **K**, update **P** matrix and state vector **X** is the calculated using:

$$\mathbf{K}_n = \mathbf{P}_n^- \mathbf{H}^{\mathrm{T}} (\mathbf{H}\mathbf{P}_n^- \mathbf{H}^{\mathrm{T}} + r)^{-1}$$

$$\mathbf{P}_n^+ = \mathbf{P}_n^- - \mathbf{K}_n \mathbf{H}\mathbf{P}_n^-$$

$$\mathbf{X}_n^+ = \mathbf{X}_n^- + \mathbf{K}_n \mathrm{d}$$

The Kalman filter gain K is also a scalar and division is used to calculate its inversion value.

7. Hardware simulation environment and behavior model

The debug and verification of the firmware can be either performed on real hardware platform or under PC environment. In order to have firmware runs independent to the implementation of the low-level baseband, the interfaces to access baseband and OS/BSP have be abstracted.

For the case of running firmware with the input of IF data file on PC platform, baseband hardware C model is used. By using the abstract layer, the function main() simply call function SetInputFile() to set the IF data file as input, and then calls function FirmwareInitialize() to do initialization including creating threads and semaphores, register ISR and assign initial value to variables etc.

Finally calls EnableRF() which initial and enable RF front-end on hardware platform and feed IF data to C model on simulation platform.

```
#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();
}
```

Similarly, a behavior model based on SignalSim project provides the same baseband hardware abstract layer interface. The firmware can run much faster on this platform, usually dozen times faster than hardware platform. The main function of the project using behavior is also most the same as C model platform with the only difference that it assigns a scenario file as input instead of IF data file. The main() function is illustrated below:

```
#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 Floating point PLL filter vs. fixed point filter

```
% 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 Floating point FLL filter vs. fixed point filter

```
% 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 Floating point DLL filter vs. fixed point filter

```
% 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]
```

Annex B.1 Fixed point square root calculation

Calculate the square root of a fixed-point number. The result of the square root is also a fixed point. Return the largest integer that is not greater than the square root of the input data. The calculation method is as follows:

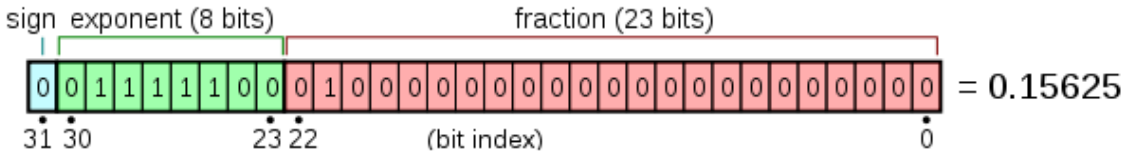
The square root is completed by searching bit by bit. First, find the largest number that is not greater than the square root result among the integer powers of 2 as the initial value of the square root. For example, if you take the square root of 8562, the result is 2^6=64 (64^2<8562<128^2). This value can be calculated using the function __builtin_clz(). Then calculate the difference between the two D=8562-64^2=4466. Next, judge the next bit one by one iteratively, that is, from the bit with weight 2^5 to the bit with weight 2^0.

If the current iteration value is R and the data to do square root is S, to determine next bit k is 1 or 0 is performed by comparing S with $(R + 2^k)^2$. If the former is greater, the k-th bit is 1, vice versa. Because $(R + 2^k)^2 = R^2 + R \cdot 2^{k+1} + 2^{2k}$, so $S - (R + 2^k)^2 = D - (R \ll (k + 1) + 1 \ll 2k)$ in which $D = S - R$. Which means the comparison is performed between D and $R \ll (k + 1) + 1 \ll 2k$. If the former is larger, than the k-th bit is 1 and D is subtracted by the latter one, otherwise the k-th bit is 0.

The iteration start from the bit following the MSB to the last bit. Each iteration only need integer add/sub and shift operation.

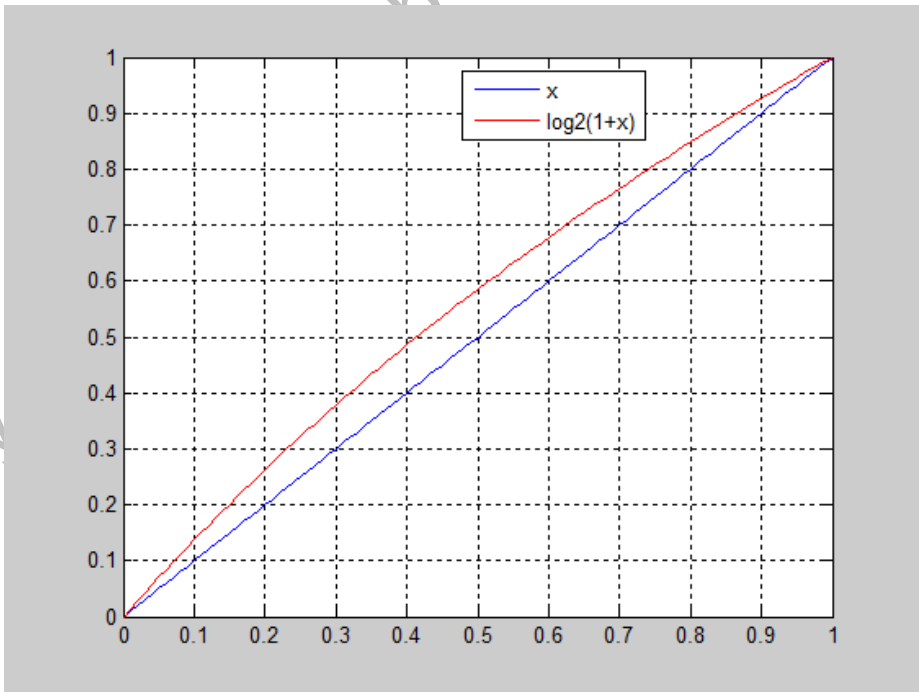
Annex B.2 Fixed point logarithm

To calculate the logarithm with base 10, you can first calculate the logarithm with base 2, and then multiply the result by lg2. To calculate the logarithm with base 2, you can use floating point conversion. First, let's introduce the floating point representation of IEEE 754. For a 32-bit float type number, it is represented as follows:

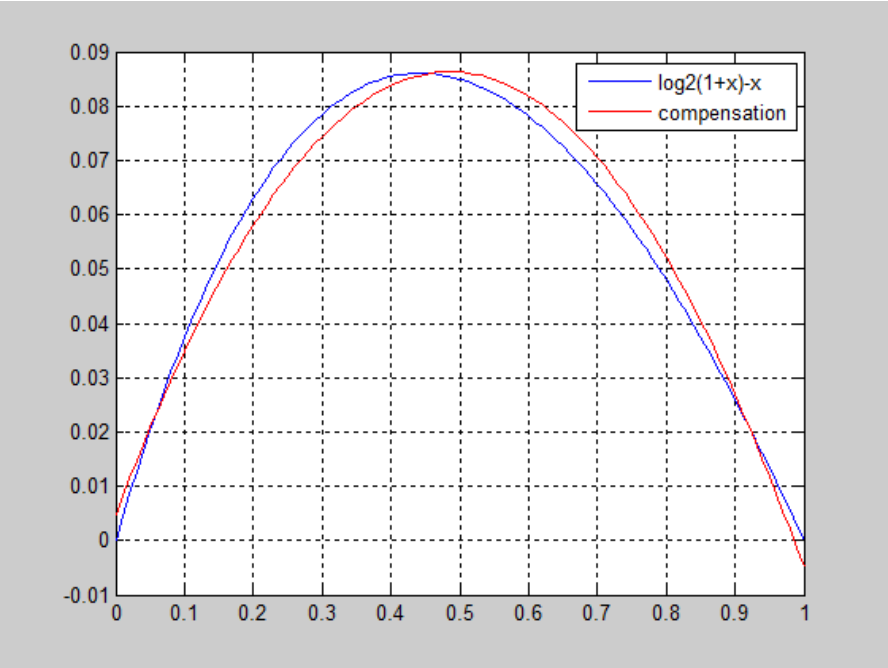


The value is divided into 3 segments: 1bit sign, 8 bit exponent and 23 bit fraction. The floating point value is expressed as $D = (-1)^S \cdot 2^{e-127} \cdot (1 + \frac{f}{2^{23}})$. Because only positive value has valid logarithm, so the sign bit can be ignored. Then we have $log_2 D = (e - 127) + log_2\left(1 + \frac{f}{2^{23}}\right) = (e - 127) + log_2\left(1 + \frac{f}{2^{23}}\right) \approx (e - 127) + \frac{f}{2^{23}}$. Because an integer value with the same 32bit pattern has the value $N = e \cdot 2^{23} + f$, so we have $log_2 D = \frac{N - 127 \cdot 2^{23}}{2^{23}}$, or treated $N - (127 \ll 23)$ as an decimal value while putting decimal point between bit22 and bit23.

The above approximate formula has error which comes from using x as an approximate value of $log_2(1 + x)$. The following figure shows the difference:



The largest difference is about 0.09, or 0.26dB when convert to base 10. High accuracy can be achieved by using quadratic function to do compensation. The error and the corrections are illustrated below:



With this correction, the largest absolute error decrease to 0.005 or 0.015dB, this is accurate enough for most cases.

The determine of exponent and fraction value can also use __builtin_clz() to avoid calling floating point related functions.