MTK GPS Logger Library User Manual

1.2

Release Date: 11/29/2006

Reported by: Andy Lee

1. Format of the logger data

There is a four bytes register (***FMT_REG***) that records the current log data formats. When the bit of some field is set, logger will log this specific data field. For example, if the bits of bit#0, bit#2 and bit#3 are set and the rest bits are kept 0. The record data will log three fields, UTC, LATITUDE and LOGTITUDE and the three fields will be logged by sequence. Because each field owns a specific data size, the total size of each record data will be the sum of the sizes of the selected fields.

Further, only after the formatting the SPI flash, the data can be logged to the SPI flash.MTK logger provides the dynamic log ging format changing mechanism that can allow users to changing the format at any time. Please note that according to the characteristics of the SPI flash that the formatting operation always takes seconds to finish.

By using the ***SETUP_LOG_CTL***, the host can choose a ***by second***, ***by distance*** or ***by speed*** logging mode. And further, a ***LOG_NOW*** argument can make the device record the data immediately.

After introduce how data is stored in the Logger, we will discuss how to fetch data out from the MTK LOGGER. A ***READ_LOG*** command is designed to ask the Logger to output the data. The device will use a ***LOG_DATA_OUTPUT*** to acknowledge this request. Once the data is read back from the device according to the ***FMT_REG*** value, host side can parse the data to get the logged information.

At present, MTK LOGGER supports two logging methods, one is **log till full** and another logging method is **overlapped logging**. Logging till full means the logger function will stop recording any data when the internal non-volatile memory is full. This logging method can be applied to the scenario that the user who doesn't want to lose any recorded data. The other method, overlapped logging is used to save the latest data. When the internal buffer of the MTK LOGGER is full, it will ring back and replace the fisrt writ in data (first in and first out). Please note that the replacement is by sector (64KB). For example, support that total size of the internal logger buffer is 0x100000. When the logger fill the last sector, sector #255, the recording will ring back to first sector and erase all the data that was previously stored in the sector #0.

2. Field Definition:

TABLE 1: FORMAT REGISTER DEFINITION

FMT_REG: FORMAT Register (4 bytes)							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
DSTA	**TRACK**	**SPEED**	**HEIGHT**	**LONGITUDE**	**LATITUDE**	**VALID**	**UTC**
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
AZI	**ELE**	**SID**	**NSAT**	**VDOP**	**HDOP**	**PDOP**	**DAGE**
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
RES	**RES**	**RES**	**RES**	**RES**	**MS**	**RCR**	**SNR**

31	30	29	28	27	26	25	24
RES	RES	RES	RES	RES	RES	RES	RES

TABLE 2: FORMAT FIELD DEFINITION

Remark: In Type field, U means unsigned, R means floating point, I means integer, 4 means 4 bytes…

FMT_REG bit	Name	Type	Note
0	**UTC**	I4	Value in time_t. It saves the time and date information when data recorded.
1	**VALID**	U2	BIT[0]: NO FIX or invalid BIT[1]: SPS mode BIT[2] : DGPS mode BIT[3] : PPS mode BIT[4] : RTK BIT[5] : FRTK BIT[6]: Estimated mode BIT[7]: Manual input mode BIT[8]: Simulator Mode
2	**LATITUDE**	R8	Value in degree. North latitude will be positive value and the south latitude will be a negative value
3	**LONGITUDE**	R8	Value in degree. East longitude will be positive value and the west latitude will be a negative value
4	**HEIGHT**	R4	Value in meters.(WGS-84)
5	**SPEED**	R4	Value in Km / hour.
6	**TRACK**	R4	Track value in degrees.
7	**DSTA**	U2	Differential GPS reference station ID.
8	**DAGE**	R4	Differential GPS correction data age
9	**PDOP**	U2	PDOP * 100
10	**HDOP**	U2	HDOP * 100
11	**VDOP**	U2	VDOP * 100
12	**NSAT**	U2	BIT[7:0] Number of satellites in view BIT[15:8] Number of satellites in use
13	**SID**	U4	BIT[7:0]➔ ID of satellite in view BIT[8]➔ SAT in use BIT[23:16]➔ Number of satellites in view
14	**ELE**	I2	Elevation angle in degree of the SID
15	**AZI**	U2	Azimuth angle in degree of the SID
16	**SNR**	U2	SNR of the SID
17	**RCR**	U2	Record Reason BIT[0]➔ Recorded by the time criteria BIT[1]➔ Record by the speed criteria BIT[2]➔ Record by the distance criteria BIT[3]➔ Record by Button BIT[15:4]➔Vendor usage
18	**MS**	U2	Milliseconds The milliseconds part of the current recording time. The second part of the current time should consult to the field of the UTC.

Because of variable length of the recorded data and the fixed sector size of the SPI flash, the recorded data might be placed into two sectors, but MTK LOGGER **does not** allow that happen. For example, if the current write-in address is located to the address 0x00fA0 of the SPI flash and the next length of the recorded data is 0x80, the recorded data might be recorded from 0x00fA0 to 0x001020. It is clear that the recorded data will be partitioned into two sectors (#1 and #2). However, in order to provide an **overlapped logging** mechanism, we need to erase some sectors to get new free buffer when buffer full that means sector erase might be used and this mechanism will allow no data to be separated into two sectors and if it did, that will cause an unable parsing problem. Because of that, some free space will be given up when the remainder space is not enough. Take the previous example for illustrating. The next write address shall be the in sector #2 (The NWA will be **0x01038**. About 0x38, please consult for the SECTOR PATTERN), and the space from 0x00fa0 to 0x00fff shall be preserved.

✓ Parsing Sample of the read out log data:

i. Please save the read out data NMEA as a binary file.

*EX: If received a data **NMEA** as **"$PMTK182,8,0,11223344…*<CR><LF>"***

*And, after transferring and saving it to a BYTE pointer **LOGData**, the pointer **LOGDat**a will look like:*

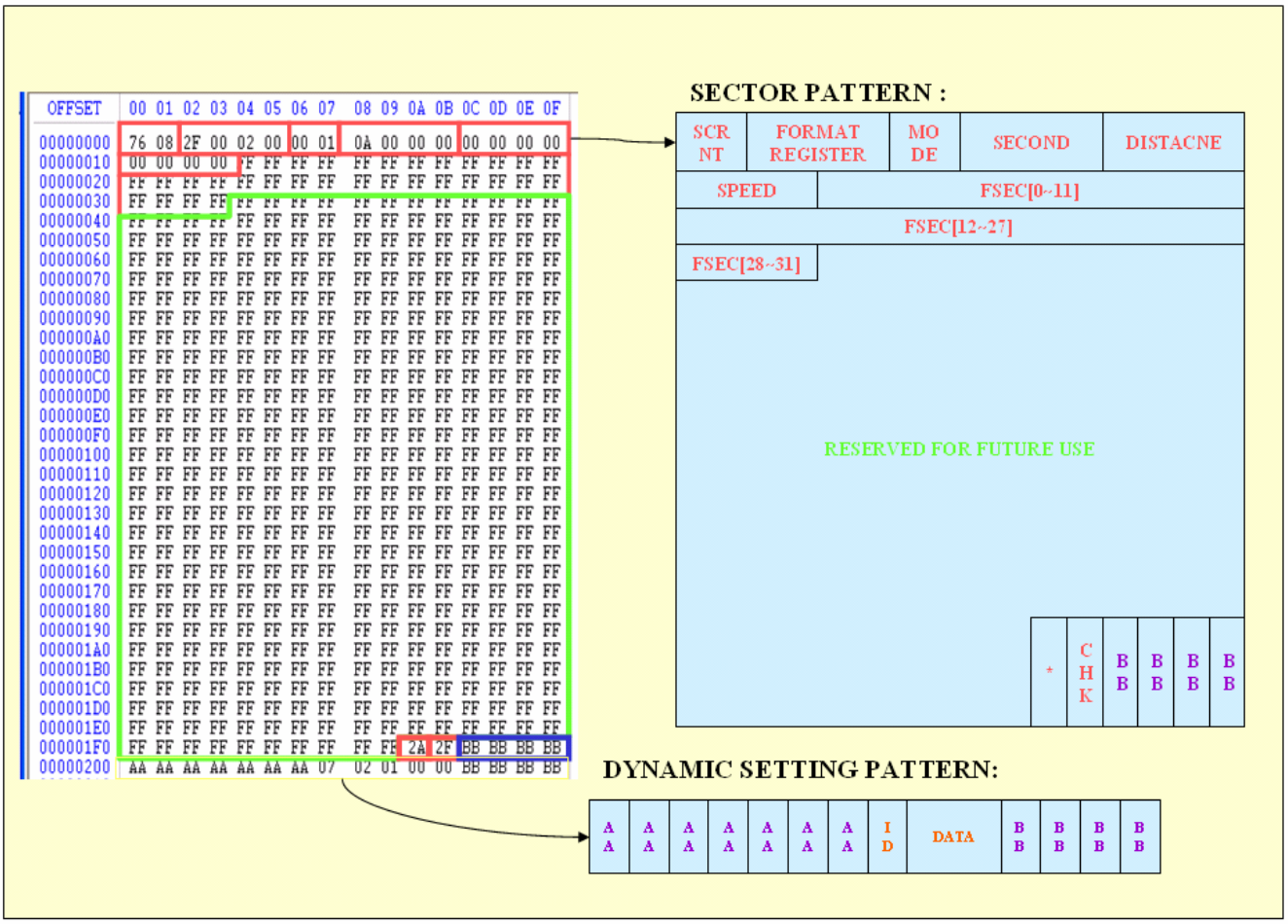
BYTE LOGData[] = {0x11, 0x22, 0x33, 0x44….}

ii. MTK Logger supports a dynamic setting changing. The changings includes format changing, auto log criteria changing and log mode changing. In order to achieve the dynamic changing function, two pattern types shall be introduiced.The first type is **SECTOR PATTERN**, it saves the information of the MTK LOGGER including the recording count of these sector and the format register data and mode data and the auto-log criteria data (SECOND, DISTANCE and SPEED) and a fail sector register data. The total size of the **SECTOR PATTERN** is 56 (**38h**) bytes and it resides the front of each sector. The MTK LOGGER will use this information to recover the logger setting in case the previous logged data can not be parsed out when the logger is running out of electricity. Please note that the software shall take care of the log mode and the format register. The log mode can help the software handle the logger. And the format register can be used to parsing the previous logged data.

The second pattern type is the **DYNAMIC SETTING PATTERN**. It can reside just after the SECTOR PATTERN data or any possible start of a new recorded data. That means when parsing, the parser shall first check if there is a dynamic format changing format. If not, the parser shall treat the following data as a recorded data. Please note that the format of DYNAMIC SETTING PATTERN will be started with a continous 7 "0xAA "and ended with 4 "0xBB". Please note that if the SCRNT is 0xFFFF that means the sector is currently writing and it dosen't mean the sector owns 0xFFFF record counts. The application program can parse the data or using the total count support by MTK LOGGER to calculate the current write count of this sector.

Further, these patterns are all written by the hardware of the MTK LOGGER, if the start of the SECTOR PATTERN does not end with the pattern, 0xBBBBBBBB. The MTK LOGGER wil assert the NEED_FORMAT_BIT, because the sector is not actually initialized. Sometimes if the logger is totally run out of power and the logger is just trying to initial one new sector might cause this happen.

The following diagram shows the format and an example of SECTOR PATTERN and the DYNAMIC SETTING PATTERN.



ID	DESCRIPTION	DATA SIZE
2	FORMAT REGISTER	4
3	0.1 SECONDS OF AUTO-LOG BY SECOND	4
4	0.1 METERS OF AUTO-LOG BY DISTANCE	4
5	0.1 KM/HR OF AUTO-LOG BY SPEED	4
6	RECORDING METHOD, OVP OR STP	2
7	MODE OF LOGGER, START, STOP	2

FIGURE 1: DYNAMIC SETTING CHANGING

iii. The current format register shall be all valid until encounter the next continuous 7 “0xAA” and end with 4 0xBB. For normal use, the MTK LOGGER will not generate such pattern, but for some special test sequence and criteria this pattern might be happened. The hardware of MTK LOGGER does not guarantee such kind test data. The software shall use some special setting logger to cover such test case.

3. NMEA Definition:

TABLE 3: NMEA COMMAND DEFINITION

$PMTK <LOG_API>,<CMD>,<ARG1>, <ARG2>,*<CHK><CR><LF>

Remark: D(decimal), H(hexadecimal)

LOG_API: PMTK182				
DEFINITION	**CMD**	**ARG1**	**ARG2**	**EXAMPLE**
SETUP_LOG_CTL	1	1: LOG NOW	H: RCD REASON	$PMTK182,1,1*<CHK><CR><LF>
		2: RCD FIELD	H: FMT_REG	$PMTK182,1,2,D*<CHK><CR><LF>
		3: BY SEC	D: 0.1 SEC (0: no use)	$PMTK182,1,3,1*<CHK><CR><LF>
		4: BY DIS	D: 0.1 METER (0: no use)	$PMTK182,1,4,20*<CHK><CR><LF>
		5: BY SPD	D: 0.1 KM/H (0: no use)	$PMTK182,1,5,30*<CHK><CR><LF>
		6: RCD METHOD	1: OVP 2: STP(dft)	$PMTK182,1,6,1*<CHK><CR><LF>
QUERY LOG STATUS	2	1: STATUS	N/A	$PMTK182,2,1*<CHK><CR><LF>
		2: FMT_REG		$PMTK182,2,2*<CHK><CR><LF>
		3: SEC		$PMTK182,2,3*<CHK><CR><LF>
		4: DIS		$PMTK182,2,4*<CHK><CR><LF>
		5: SPD		$PMTK182,2,5*<CHK><CR><LF>
		6: RCD METHOD		$PMTK182,2,6*<CHK><CR><LF>
		7: LOG STATUS		$PMTK182,2,7*<CHK><CR><LF>
		8: RCD ADDR		$PMTK182,2,8*<CHK><CR><LF>
		9: FLASH ID	D: RID_CMD	$PMTK182,2,9,9F*<CHK><CR><LF>
		10: RCD RCNT	N/A	$PMTK182,2,10*<CHK><CR><LF>
		11: RCD FSECTOR	N/A	$PMTK182,2,11*<CHK><CR><LF>
		12: VERSION	N/A	$PMTK182,2,12*<CHK><CR><LF>
RETURN LOG STATUS	3	1: SPI STATUS	1: RDY 2: BSY 3: FULL	$PMTK182,3,1,1*<CHK><CR><LF>
		2: FMT_REG	H: FMT_REG	$PMTK182,3,2,D*<CHK><CR><LF>
		3: SEC	D: SEC	$PMTK182,3,3,1*<CHK><CR><LF>
		4: DIS	D: METER	$PMTK182,3,4,20*<CHK><CR><LF>
		5: SPD	D: KM/H	$PMTK182,3,5,30*<CHK><CR><LF>
		6: RCD METHOD	1: OVP 2: STP	$PMTK182,3,6,1*<CHK><CR><LF>
		7: LOG STATUS	1: START LOG 2: STOP LOG	$PMTK182,3,7,1*<CHK><CR><LF>
		8: RCD ADDR	H: ADDR	$PMTK182,3,8,100*<CHK><CR><LF>
		9: FLASH ID	H: ID	$PMTK182,3,9,100*<CHK><CR><LF>
		10:RCD RCNT	H: RCNT	$PMTK182,3,10,0*<CHK><CR><LF>
		11: RCD FSECTOR	H: FAIL SECTOR	$PMTK182,3,11,FF~FF*<CHK><CR><LF>
		12: VERSION	D: VERSION*100	$PMTK182,3,12,100*<CHK><CR><LF>

START LOG	4	N/A	N/A	$PMTK182,4*<CHK><CR><LF>
STOP LOG	5	N/A	N/A	$PMTK182,5*<CHK><CR><LF>
FORMAT LOG	6	1: ALL	N/A	$PMTK182,6,1*<CHK><CR><LF>
		2: PARTIAL	H: ADDR(byte)	$PMTK182,6,2,10000*<CHK><CR><LF>
READ LOG	7	H: ADDR(byte)	H: LEN(byte) (even number)	$PMTK182,7,0,2*<CHK><CR><LF>
LOG DATA OUTPUT	8	H: ADDR(byte)	H: DATA	$PMTK182,8,0,55AA*<CHK><CR><LF>
INIT_LOG	9	N/A	N/A	$PMTK182,9*<CHK><CR><LF>
ENABLE_LOG	10	N/A	N/A	$PMTK182,10*<CHK><CR><LF>
DISABLE_LOG	11	N/A	N/A	$PMTK182,11*<CHK><CR><LF>
WRITE_LOG	12	H: ADDR(byte)	H: DATA (4 BYTES)	$PMTK182,12,0,55AA*<CHK><CR><LF>

4. Application Notes:

(1) LOG NOW:

There are four default recording reasons (RCD REASONs) defined in TABLE 1. By setting on the RCD_REASON in the format register. By the user-defiend vendor RCR REASON values, application program can use the LOG NOW to LOG varied time and location for some purpose. For example:

STEP 1: Create a vendor RCD REASON definition:

TABLE 4: EXAMPLE OF A VENDOR DEFINITION

RCD_REASON	TYPE	DEFINITION
BIT[0]	AUTO_LOG	SECONDS
BIT[1]	AUTO_LOG	AUTO_LOG, SPEED
BIT[2]	AUTO_LOG	AUTO_LOG, DISTANCE
BIT[3]	LOG_NOW	BUTTON
`BIT[4]	VENDOR	GAS STATION
BIT[5]	VENDOR	ELEMENTARY SCHOOL
BIT[6]	VENDOR	HOTEL
…	…	…
BIT[15]	VENDOR	SUPERMARKET

STEP 2: By issusing the LOG NOW command and setting a **RCD_REASON** field of the FORMAT REGISTER, we can record the specific geographical information we need.

(2) AUTO LOG:

There are thre kinds of AUTO LOG mode supported by the MTK LOGGER BY SEC (seconds), BY DIS (Distance) and BY SPD (Speed). The first step to start the MTK AUTO LOG function is to setup one of the above three criteria. If the value of the BY_SEC. BY_DIS and BY_SPD is 0, that means the current criterion is unused. Only when the set value is bigger then 0, it will be regarded as a valid setup.

The minimum logging time interval of theBY_SEC function will depend on execution Hz of MTK LOGGER default execution Hz is 1 Hz. If 1 Hz, the by second function can record the time inverval of 1 seconds, and if 5 Hz is available, the minimum time interval will be 0.2 seconds.

As defined in the TABLE 3. The unit of BY_SEC shall be set with the unit 0.1second, and the BY_DIS shall be set with the unit 0.1 KM and the BY_SPD value shall be set with the unit 0.1 KM/HR.

START_LOG is the command to trigger the AUTO LOG function. If the AUTO LOG function is started, it will check the current condition fit the setting criteria of SEC, DIS and SPD and decide whether to record a data or not.

STOP_LOG is the command to stop the AUTO LOG function.

Please note that if a logging data is progressing and the MTK LOGGER is powered off at the same time, it will cause some the current logged data damaged. The software might try to do some post operation to avoid such circumstances affecting the accuracy of the log data. Please note any updae to setting of START, STOP, BY_SEC, BY_DIS and BY_DIS will cause the MTK LOGGER write a DYNAMIC SETTING CHANGE PATTERN into the flash. Each update will spend 16 bytes data.

(3) Recording method (OVP and STP):

The default recording methos is overlapped logging method. The RCD method can be setup by the RCD_METHOD of SETUP_LOG_CTL and queried by the RCD METHOD of QUERY_LOG_STATUS.If there is no more size for recording when method is STP, and the user still issue new command to ask the MTK LOGGER to record data. A NMEA with a full status will be send to the host. Please note the update to the setting of the recording method will cause the MTK LOGGER write a DYNAMIC SETTING PATTERN into the flash. Each update will spend 16 bytes data. If there is not enough data size (16 bytes) for the MTK LOGGER to update a DYNAMIC SETTING PATTERN under the method STP, some command might return fail and a NMEA with the status FULL will be issued from the MTK LOGGER.

(4) SPI STATUS:

The SPI_STATUS is the status of the internal SPI flash of the MTK LOGGER. It owns three statuses, BSY, RDY and FULL. When the RCD methos is STP (STOP) and there is no more space for the new record or DYNAMIC SETTING PATTER, the MTK LOGGER will send a FULL status from a RETURN LOG STATUS and stop the auto-logging function.

(5) FORMAT LOG:

FORMAT LOG provides two function ALL and PARTIAL. The FORMAT ALL can reset the internal buffer to become all 0xFF. The PARTIAL formatting is used to format one of the sectors becoming all oxff. It is used in overlapped logging. The partial formatting command was designed for the engineering test only. Issuing a partial formatting command might cause the make the value of RCD ADDR and RCD RCNT become wrong. FORMAT ALL should be able to deal all the required use cases.

DO NOT issue commands except the QUERY_LOG_STATUS, STATUS (**$PMTK182,2,1**) , it might cause the formating operation failed. By the way, if the AUTO logging function is on (LOG_START) and the user issued a FORMAT_LOG command that will cause the AUTO logging function off. The default format register value is UTC, LON and LAT (0x0000000D). Please note that the FORMAT LOG command will cause the MTK LOGGER to write an SECTOR PATTERN to the sector # 0 and it will spend 28 bytes data of the internal non-volatile memory of MTK LOGGER.

(6) RCD INFORMATION:

The RCD ADDR is the address for the next write and RCD RCNT is the current total recorded count

After formatting, the initial RCD ADDR will be **0x00001A**. The first two bytes in each sector will be used as the total recording count of this sector. If the first two bytes are 0xFFFF that means this sector still has space to record data.

(7) STATE MACHINE:

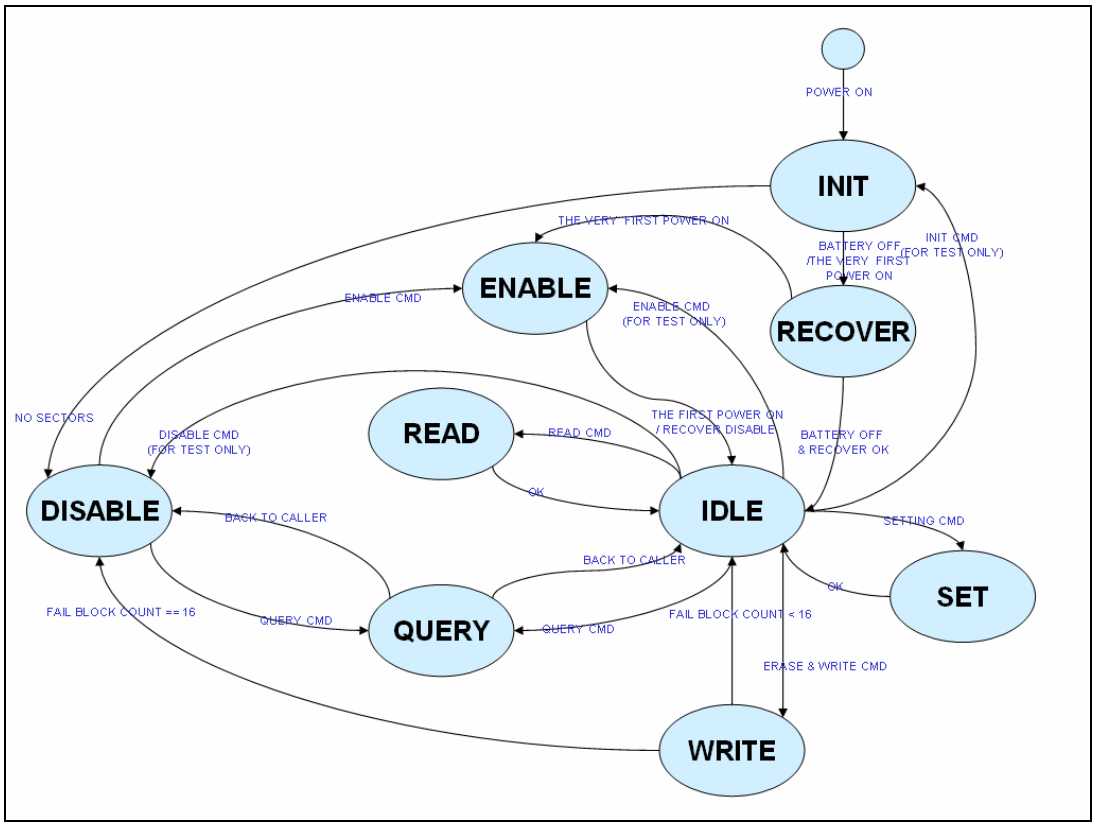


FIGURE 2: STATE CHART OF MTK LOGGER

ENABLE LOG is used for enable the logger function or a logger function which is not able to do any write command which is previously been disabled by a NMEA command. When a LOG disabled happened without issuing a DISABLE LOG command, it means the MTK LOGGER cannot finish some write operations. It maybe cause by the internal non-volatile memory reached its maximum write count (from 10000 to 100000) or some s the hardware suffers some physical damaged such as being dropped down. However, if the count of the damaged sectors is less then 16, the MTK LOGGER will mask it as a fail block (64KB) and save the information in the FSECTOR.The FSECTOR is an internal 16-bits register of MTK LOGGER. The application program can see if there is any block (64KB) that is masked by querying the FSECTOR register. (If Bit#0 asserted, it means the block#0 (64KB) is masked, and if bit1 is set, that means block#1 is masked etc...) An ENABLE command can reset the FSECTOR recording that means the user can use the ENABLE command to recover the masked sector and try to write it again. If all sector becoming unwritable, the MTK LOGGER will enter the disabled mode. Otherwise the FSECTOR will be set and return to IDLE mode.

In a DISABLE state, the commands accepted are only the QUERY type and because when the device entered the DISABLE State, we believe that the data in the MTK LOGGER might be untrustable, a NEED FORMAT bit will be asserted. Please note that you can not format the LOGGER in the DISABLE state, and an ENABLE command shall be inssued

before the FORMAT operation. INIT is used to ask the device checking its status is initialized. If the status is not initialized, the MTK LOGGER will set the SEC, SPD, and DIS to 0 and set the format reguister to 0x0D (UTC, LAT and LON) and setup the LOG STATUS to LOG_ENABLED and NEED FORMAT and the default RCD ADDR will be set to x00002.

If the MTK LOGGER is not enabled, the device will enable and set the LOGGER hardware setting to the initial values. Because of that, the MTK LOGGER can work normally. If the LOGGER hardware is damaged, the disable bit will be set and all the logger function except the QUERY will be disabled.

For some MTK LOGGER product battery off might happened some time. The MTK LOGGER will try to RECOVER the data stored in the internal non-volatile to keep users' precious data.

(8) LOG STATUS

Bit [1]: auto-log start bit

Bit [2]: log method bit

Bit [8]: log function enabled bit

Bit [9]: log function disabled bit

Bit [10]: logger need format bit

Bit [11]: logger full bit

For some special circumstances, such as when the battery is down and the MTK LOGGER is not able to recover the data, all user setting will be cleared, and the MTK LOGGER will be not able to known its previous status such as the previous write address, and the previous format setting. When power on, the MT3301 will first check the enabled bit, it is not enabled, the MT3301 will reset all settings to the default values and the logger need format bit will be asserted as well that means this logger system need a format operation and after formatting the logger need format bit will be negated. The application program shall always check the LOG STATUS and take the appropriate operations to help keep soundness of the logger function.

After logger function initialized, the **LOG STATUS** will be set as 0x0500 and the bit 8 is used for the initialization signature. After disabled and enabled, the Need format bit of the LOG STATUS will be asserted. The software shall take care of this scenario.

If the log method bit is assert that means the method is STP(STOP), otherwise the log method is OVP (overlapped).

(9) READ LOG:

DO NOT issue any commands during **reading** and **writing** and an abnormal power off during the above scenarios might cause the post-parsing operation failed.

If the requested length of the READ LOG command is lager than 0x800, the READ LOG command will divide the requested length into several LOG DATA OUTPUT NMEA with the maximum size 0x800 bytes.

For example:

If the AP asked a data with length 0x10000 and from address 0x200000 $PMTK182,7,20000,10000*<CHK><CR><LF>

The device will returns

$PMTK182,8,20000,XXX…XXXX*<CHK><CR><LF>

$PMTK182,8,20800,XXX…XXXX*<CHK><CR><LF>

……

$PMTK182,8,2F000,XXX…XXXX*<CHK><CR><LF>

$PMTK182,8,2F800,XXX…XXXX*<CHK><CR><LF>

Please note that, the read out length shall be even, and odd number is not allowable.

(10) QUERY ID:

Query FLASH ID can be used to the test of the MTK LOGGER function. This NMEA will pop up a 4 bytes ID data. RID CMD is the command OP code of read ID command of the SPI flash. The default value is 0x9F, and for some SPI flashes, the OP code may be 0xAB or 0x90.

(11) Zero satellites in view issue

The record length of MTK LOGGER is variable. The main reason is that the MTK LOGGER can support the satellites' information. However the satellites number is unpredictable. Because of that we shall preserve duplicated information, satellites in view (SAT_IN_VIEW). However, when the SAT_IN_VIEW is zero, the number of the satellites in view in the SID field (BIT [23:16]) will be set to 0x00. That means even there is no SAT_IN_VIEW, the MTK LOGGER will still generate a 4 bytes SID information that can prevent the ambigious parsing issue when there is no SAT_IN_VIEW.

(12) Fail Sector (FSEC)

The value of fail sector within sector pattern means some sectors are broken and couldn't use any more. Use 1 bit stands for the valid status of a sector. If the value of bit is 1 means the sector works well, otherwise the sector is broken. In order to support the size of serial flash to 128Mb, make use of 32 bytes to record the sector status The 20th byte of sector pattern records the fail information of sector#0~#7 and the 21th bytes of the sector pattern record the information of sector#7~#15, etc.

(13) Data check sum issue

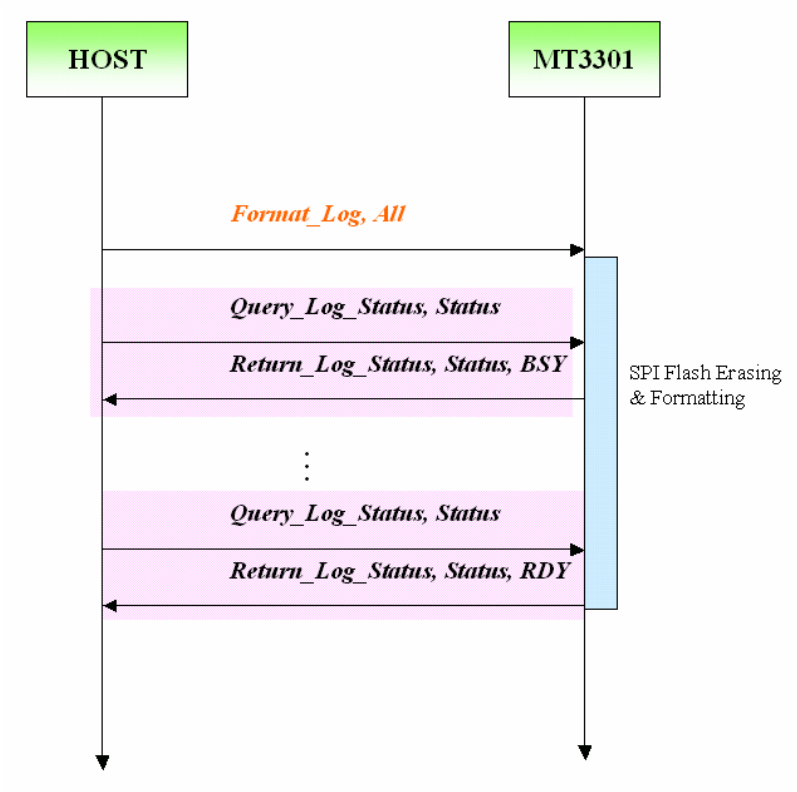
In order to make sure log data is valid, so each log data has its own checksum value. The value of checksum is calculated via following criteria.

```
GUCHAR GenerateCkSum(GUCHAR *pure_data, U2 size)
{
    GUCHAR chksum;
    I2 i;
    chksum = pure_data[0];
    for(i=1; i<size ; i++)
    chksum ^= pure_data[i];
    return chksum;
}
```

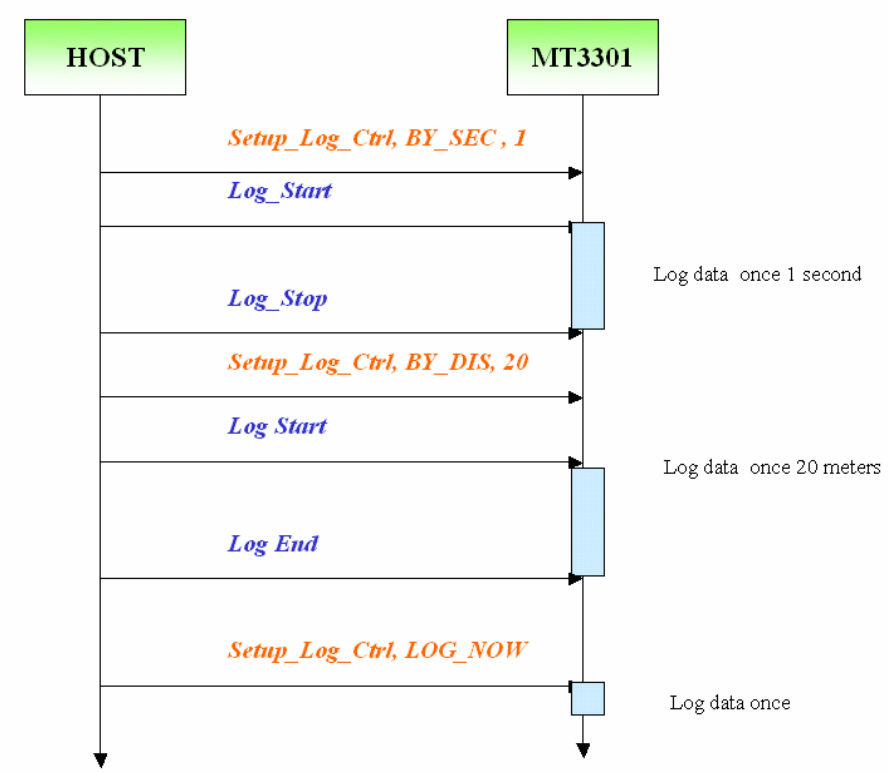
"pure_data" means the log data only without any checksum or any non-log data character, and size means the size of pure_data in bytes. The size of checksum is only 1 Byte. In order to identify data more easily, add 1 character('*', 0x2A) before the checksum byte.

5. Scenarios:

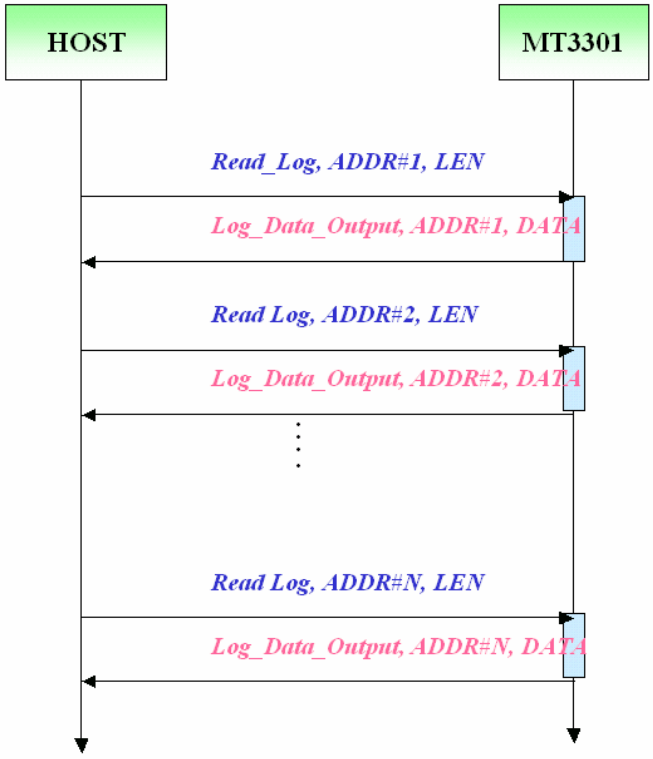
+Log Format Scenario



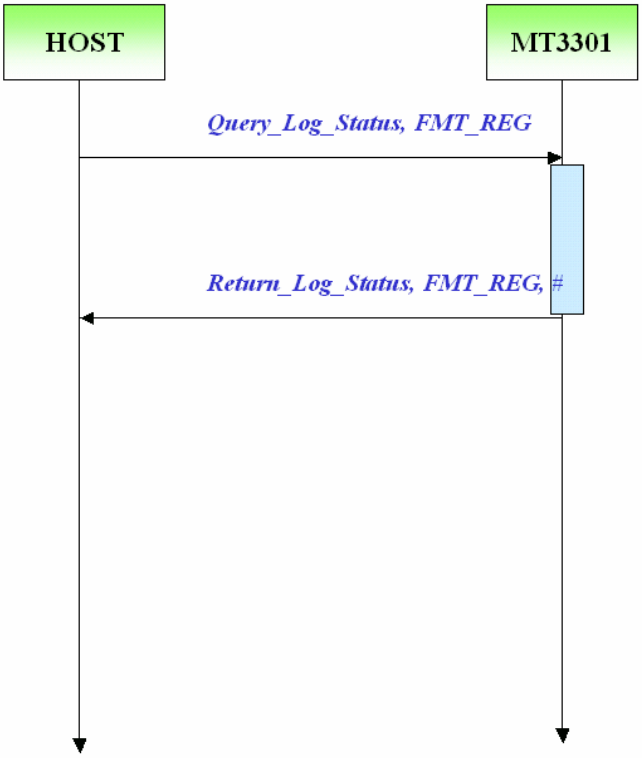
+Log Control Setup



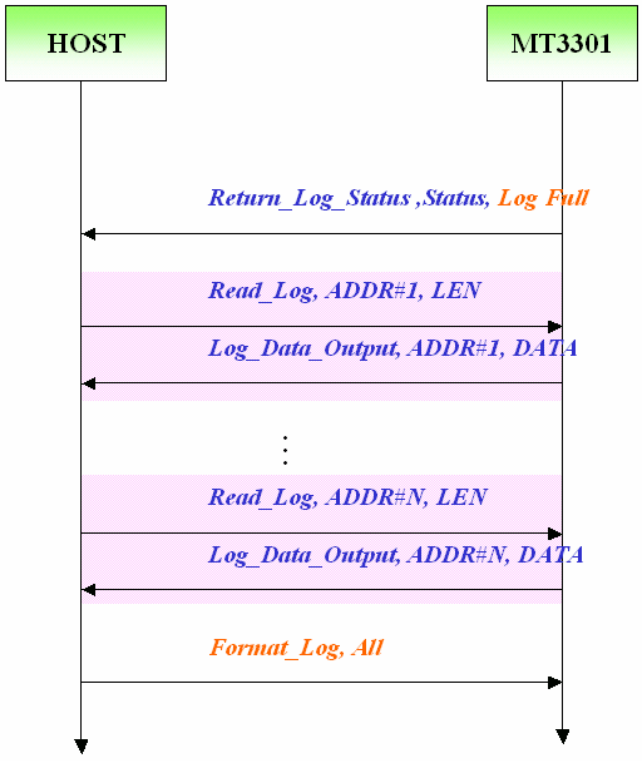
+Read Scenario



+Query FMT_REG



+Log Full Control



6. **Sample Parser Source**

```
/*****************************************************************************
*[File] *         GParser.CPP
*[Version] *      v0.96
*[Revision Date] * 2006-09-25
*[Author] *       JL Juang, jl juang@mtk.com.tw, +886-3-567-0766 EX. 26447
*[Description] *   Sample code for parsing binary data from the MTK Logger Ver. 0.99E
*[Copyright] *    Copyright (C) 2005 MediaTek Incorporation. All Rights Reserved
*****************************************************************************/
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "GParser.h"

int main(int argc, char* argv[])
{
     FILE *in;
     FILE *out;
     int i;
     // print information
     printf("+-----------------------------------------------------------+\r\n");
     printf("| [File]      GParser.EXE                                   |\r\n");
     printf("| [Author]    JL Juang, jl juang@mtk.com.tw, 2005-08-16     |\r\n");
     printf("| [Copyright]  Copyright (C) 2006 MediaTek Incorporation.    |\r\n");
     printf("|            All Rights Reserved                            |\r\n");
     printf("+-----------------------------------------------------------+\r\n");

     if(argc<2)
     {
       printf("USAGE: GPARSER IN FILE NAME [OUT FILE] \r\n");
       printf("EXAMPLE: GPARSER LOG.BIN  \r\n");
       return 0;
     }
   // open file
     in = fopen(  argv[1], "rb");
     if(argv[2] != NULL)
       out = fopen(argv[2], "w+");
     else
     out = fopen("LOG.TXT", "w+");
     if(in == NULL)
     {
     printf("FAIL TO  OPEN INPUT FILE !!!!\r\n");
       fcloseall();
     return 0;
     }
     if(out == NULL)
     {
       printf("FAIL TO  OPEN INPUT FILE !!!!\r\n");
       fcloseall();
       return 0;
     }


    for(i=0; i<LOG SECTOR TOTAL/LOG SECTOR SIZE;i++)
    {
       printf("Parsing sector #%-3d.......Parsing\r", i);
       ParseSector(i, in, out);
       printf("Parsing sector #%-3d......OK     \r", i);
    }
    fclose(out);
    fclose(in);
      return 0;
}

unsigned long ParseSector(int iSec, FILE *in,  FILE *out)
{
   UC LogBuf[LOG SECTOR SIZE + 0x800];
   U4 idx = 0;
   U2 u2Size = 0;
   U4 u4Cnt = 0;
   UC iInView = 0;
   // Setting values
   U4 u4FMTREG = 0;
   U2 u2SRCNT = 0;
   U2 u2RCDMET = 0;
   U2 u2RCDMOD = 0;
   U4 u4SEC = 0;
   U4 u4DIS = 0;
   U4 u4SPD = 0;
   UC ucFSEC[32];
```

```
U4 u4BBBB = 0;
// flag
UC ucSETCHG = 0;
char strBuf[MAX STRBUF SIZE];
// Read data
if(fseek(in,(long)iSec*LOG SECTOR SIZE, SEEK SET)!=  0)
   return 0;
if(fread(LogBuf, LOG SECTOR SIZE, 1, in) == 0)
  return 0;
idx = 0;
///////////////restore setting ////////////////////////
// STEP 1: read recording counts in this sector
memcpy(&u2SRCNT, &LogBuf[idx], sizeof(u2SRCNT));
idx += sizeof(u2SRCNT);
// STEP 2: load FMT REG
memcpy(&u4FMTREG, &LogBuf[idx], sizeof(u4FMTREG));
idx += sizeof(u4FMTREG);
// STEP 3: rcd mode
memcpy(&u2RCDMOD, &LogBuf[idx], sizeof(u2RCDMOD));
idx += sizeof(u2RCDMOD);
// STEP 4: SEC mode
memcpy(&u4SEC, &LogBuf[idx], sizeof(u4SEC));
idx += sizeof(u4SEC);
// STEP 5: DIS mode
memcpy(&u4DIS, &LogBuf[idx], sizeof(u4DIS));
idx += sizeof(u4DIS);
// STEP 6: SPD mode
memcpy(&u4SPD, &LogBuf[idx], sizeof(u4SPD));
idx += sizeof(u4SPD);
// STEP 7: FSEC mode
{
   int i;
   for(i=0 ; i<32 ; i++)
     ucFSEC[i] = LogBuf[idx+i]
   idx += sizeof(ucFSEC);
}
// STEP 8: End pattern
Idx = 0x200 - sizeof(u4BBBB);
memcpy(&u4BBBB, &LogBuf[idx], sizeof(u4BBBB));
idx +=  sizeof(u4BBBB);
if(u4BBBB != 0xBBBBBBBB)
{
   // invalid pattern
  return u4Cnt;
}
else
{
  int i;

  fprintf(out, "#####################################################\r\n");
  fprintf(out, "# SECTOR #      :    %-8d                #\r\n",iSec);
  fprintf(out, "# SECTOR COUNT   :    %-8x                 #\r\n",u2SRCNT);
  fprintf(out, "# FORMAT REGISTER:    %-8lx                  #\r\n",u4FMTREG );
  if(u2RCDMOD & 0x04)
  {
    fprintf(out, "# RCD METHOD     :    %-8s                  #\r\n","STOP");
  }
  else
  {
    fprintf(out, "# RCD METHOD     :    %-8s                  #\r\n","OVP");
  }
  fprintf(out, "# RCD MODE       :    %-8x                #\r\n",u2RCDMOD);
  fprintf(out, "# SEC MODE       :    %-8d                #\r\n",u4SEC);
  fprintf(out, "# DIS MODE       :    %-8d                #\r\n",u4DIS);
  fprintf(out, "# SPD MODE       :    %-8d                #\r\n",u4SPD);

  fprintf(out, "# FSEC MODE      :    ");
  for(i=0; i<32 ; i++)
    fprintf(out, "%-2x ", ucFSEC[i]);
  fprintf(out, "#\r\n");

  fprintf(out, "#####################################################\r\n");
}
/////////////////////////////////////////////////////
while(1)
{
   u2Size = 0;
   // clear buffer
   strcpy(strBuf,"");
   // parse
   ucSETCHG = true;
   // -------------------------------------------
```

```
      // check pattern:
      // -------------------------------------------
      //   AA AA AA AA - AA AA AA ID
      //   DD DD DD DD - BB BB BB BB
      // -------------------------------------------
      //  0 ~ 6 : AA AA AA AA AA AA AA ==> intitial pattern
      //      7 : ID ==> ID of the Setting
      //  8 ~ 11: DD DD DD DD ==> Data of the setting
      // 12 ~ 15: BB BB BB BB ==> End pattern
      // -------------------------------------------
    for(int i = 0;i<16;i++)
    {
      if((LogBuf[idx+i] != 0xAA) && i < 7)
        ucSETCHG = 0;
      if((i>11) &&(LogBuf[idx+i] != 0xBB))
        ucSETCHG = false;
    }
    if(ucSETCHG)
    {
      if(LogBuf[idx+7] == RCD FIELD)
      {
        memcpy(&u4FMTREG, &LogBuf[idx+8], 4);
        fprintf(out, "<CHANGE FORMAT : %08lxh >\r\n", u4FMTREG);
      }
      else if(LogBuf[idx+7] == BY SEC)
      {
        memcpy(&u4SEC, &LogBuf[idx+8], 4);
        fprintf(out, "<CHANGE SEC : %08f >\r\n", u4SEC/10.0);
      }
      else if(LogBuf[idx+7] == BY DIS)
      {
        memcpy(&u4DIS, &LogBuf[idx+8], 4);
        fprintf(out, "<CHANGE DIS : %08f >\r\n", u4DIS/10.0);
      }
      else if(LogBuf[idx+7] == BY SPD)
      {
        memcpy(&u4SPD, &LogBuf[idx+8], 4);
        fprintf(out, "<CHANGE SPD : %08f >\r\n", u4SPD/10.0);
      }
      else if(LogBuf[idx+7] == RCD METHOD)
      {
        memcpy(&u2RCDMET, &LogBuf[idx+8], 2);
        fprintf(out, "<CHANGE METHOD : %04xh >\r\n", u2RCDMET);
      }
      else if(LogBuf[idx+7] == LOG STA)
      {
        memcpy(&u2RCDMOD, &LogBuf[idx+8], 2);
        fprintf(out, "<CHANGE MOD : %04xh >\r\n", u2RCDMOD);
      }
      idx += 16;
      continue;
    }
    if(u4FMTREG & FMT UTC)
    {
       time t t1;
      sprintf(strBuf+strlen(strBuf), "%10s:    ", "UTC");
      memcpy(&t1, &LogBuf[idx+u2Size], sizeof(time t));
      if(ctime(&t1) == NULL)
      {
           sprintf(strBuf+strlen(strBuf), "%10s", "NO TIME DATA\r\n");
           break;
      }
      else
      {
        sprintf(strBuf+strlen(strBuf), "%10s", ctime((time t*)&LogBuf[idx+u2Size]));
      }
      u2Size += 4;
    }
    if(u4FMTREG & FMT VAL)
    {
      sprintf(strBuf+strlen(strBuf), "%10s:    ", "VAL");
      if(LogBuf[idx+u2Size] & FMT VAL FIX) sprintf(strBuf+strlen(strBuf), "[FIX]");
      if(LogBuf[idx+u2Size] & FMT VAL SPS) sprintf(strBuf+strlen(strBuf), " [SPS]");
      if(LogBuf[idx+u2Size] & FMT VAL DGPS) sprintf(strBuf+strlen(strBuf), " [DGPS]");
      if(LogBuf[idx+u2Size] & FMT VAL EST) sprintf(strBuf+strlen(strBuf), " [EST]");
        if(LogBuf[idx+u2Size] == 0x00)
        sprintf(strBuf+strlen(strBuf), "[NO FIX]");
        sprintf(strBuf+strlen(strBuf), "\r\n");
        u2Size += 2;
    }
    if(u4FMTREG & FMT LAT)
    {
```

```
        double r8Tmp;
      sprintf(strBuf+strlen(strBuf), "%10s:    ", "LAT");
      memcpy(&r8Tmp, &LogBuf[idx+u2Size], 8);
      sprintf(strBuf+strlen(strBuf), "%-9.6f",r8Tmp);
      sprintf(strBuf+strlen(strBuf), "\r\n");
      u2Size += 8;
    }
    if(u4FMTREG & FMT LON)
    {
       double r8Tmp;
      sprintf(strBuf+strlen(strBuf), "%10s:    ", "LON");
      memcpy(&r8Tmp, &LogBuf[idx+u2Size], 8);
      sprintf(strBuf+strlen(strBuf), "%-9.6f",r8Tmp);
      sprintf(strBuf+strlen(strBuf), "\r\n");
      u2Size += 8;
    }
    if(u4FMTREG & FMT HGT)
    {
      float r4Tmp;
        sprintf(strBuf+strlen(strBuf), "%10s:    ", "HGT");
      memcpy(&r4Tmp, &LogBuf[idx+u2Size], 4);
      sprintf(strBuf+strlen(strBuf), "%-9.2f",r4Tmp);
      sprintf(strBuf+strlen(strBuf), "\r\n");
      u2Size += 4;
      }
      if(u4FMTREG & FMT SPD)
      {
        float r4Tmp;
        sprintf(strBuf+strlen(strBuf), "%10s:    ", "SPD");
        memcpy(&r4Tmp, &LogBuf[idx+u2Size], 4);
        sprintf(strBuf+strlen(strBuf), "%-9.2f",r4Tmp);
        sprintf(strBuf+strlen(strBuf), "\r\n");
        u2Size += 4;
      }
      if(u4FMTREG & FMT TRK)
      {
        float r4Tmp;
        sprintf(strBuf+strlen(strBuf), "%10s:    ", "TRK");
        memcpy(&r4Tmp, &LogBuf[idx+u2Size], 4);
        sprintf(strBuf+strlen(strBuf), "%-9.2f",r4Tmp);
        sprintf(strBuf+strlen(strBuf), "\r\n");
        u2Size += 4;
      }
      if(u4FMTREG & FMT DSTA)
      {
        U2 u2Tmp;
        sprintf(strBuf+strlen(strBuf), "%10s:    ", "DSTA");
        memcpy(&u2Tmp, &LogBuf[idx+u2Size], 2);
        sprintf(strBuf+strlen(strBuf), "%x",u2Tmp);
        sprintf(strBuf+strlen(strBuf), "\r\n");
        u2Size += 2;
      }
      if(u4FMTREG & FMT DAGE)
      {
        float r4Tmp;
        sprintf(strBuf+strlen(strBuf), "%10s:    ","DAGE");
        memcpy(&r4Tmp, &LogBuf[idx+u2Size], 4);
        sprintf(strBuf+strlen(strBuf), "%-9.2f",r4Tmp);
        sprintf(strBuf+strlen(strBuf), "\r\n");
        u2Size += 4;
       }
     if(u4FMTREG & FMT PDOP)
      {
         U2 u2Tmp;
       sprintf(strBuf+strlen(strBuf), "%10s:    ", "PDOP");
       memcpy(&u2Tmp, &LogBuf[idx+u2Size], 2);
       sprintf(strBuf+strlen(strBuf), "%-9.2f",(float)u2Tmp/100.0);
       sprintf(strBuf+strlen(strBuf), "\r\n");
       u2Size += 2;
      }
      if(u4FMTREG & FMT HDOP)
      {
         U2 u2Tmp;
       sprintf(strBuf+strlen(strBuf), "%10s:    ", "HDOP");
       memcpy(&u2Tmp, &LogBuf[idx+u2Size], 2);
       sprintf(strBuf+strlen(strBuf), "%-9.2f",(float)u2Tmp/100.0);
       sprintf(strBuf+strlen(strBuf), "\r\n");
       u2Size += 2;
      }
      if(u4FMTREG & FMT VDOP)
      {
         U2 u2Tmp;
```

```
        sprintf(strBuf+strlen(strBuf), "%10s:    ","VDOP");
        memcpy(&u2Tmp, &LogBuf[idx+u2Size], 2);
        sprintf(strBuf+strlen(strBuf), "%-9.2f", (float)u2Tmp/100.0);
        sprintf(strBuf+strlen(strBuf), "\r\n");
        u2Size += 2;
      }
      if(u4FMTREG & FMT NSAT)
      {
        sprintf(strBuf+strlen(strBuf), "%10s:    ", "IN VIEW");
        iInView =LogBuf[idx+u2Size];
        sprintf(strBuf+strlen(strBuf), "%d\r\n", iInView);
        sprintf(strBuf+strlen(strBuf), "%10s:    ", "IN USE");
        sprintf(strBuf+strlen(strBuf), "%d",LogBuf[idx+u2Size+1]);
        sprintf(strBuf+strlen(strBuf), "\r\n");
        u2Size += 2;
      }
     ////////////////////////////
      if(u4FMTREG & FMT SID)
      {
        iInView = LogBuf[idx+u2Size+2];
        if(iInView == 0xff)
        {
          iInView = 0;
        }
       for(int i=0;i<iInView;i++)
          {
          // SID part
            unsigned char uSID = LogBuf[idx+u2Size];
        sprintf(strBuf+strlen(strBuf), "    --------------------------------
\r\n");
        sprintf(strBuf+strlen(strBuf), "  %10s#", "SID");
          sprintf(strBuf+strlen(strBuf), " %02d", uSID);
          // in use check
          if(LogBuf[idx+u2Size+1] & 0x01)
            sprintf(strBuf+strlen(strBuf), "%s"," [IN USE]");
          sprintf(strBuf+strlen(strBuf), "\r\n");
        u2Size += 4;
          // ELE
          if(u4FMTREG & FMT ELE)
          {
          signed char  i1Tmp;
            sprintf(strBuf+strlen(strBuf), "    %10s:    ", "ELE");
            memcpy(&i1Tmp, &LogBuf[idx+u2Size], 1);
            sprintf(strBuf+strlen(strBuf), "%02d", i1Tmp);
            sprintf(strBuf+strlen(strBuf), "\r\n");
            u2Size += 2;
          }
            //AZI
        if(u4FMTREG & FMT AZI)
           {
             short i2Tmp;
             sprintf(strBuf+strlen(strBuf), "    %10s:    ", "AZI");
             memcpy(&i2Tmp, &LogBuf[idx+u2Size], 2);
             sprintf(strBuf+strlen(strBuf), "%02d", i2Tmp);
             sprintf(strBuf+strlen(strBuf), "\r\n");
             u2Size += 2;
           }
           //SNR
           if(u4FMTREG & FMT SNR)
           {
             unsigned short u2Tmp;
             sprintf(strBuf+strlen(strBuf), "    %10s:    ","SNR");
             memcpy(&u2Tmp, &LogBuf[idx+u2Size], 2);
             sprintf(strBuf+strlen(strBuf), "%02d", u2Tmp);
             sprintf(strBuf+strlen(strBuf), "\r\n");
             u2Size += 2;
           }
         }
       if(iInView == 0)
          {
         // Empty SID
            sprintf(strBuf+strlen(strBuf), "%10s:    ", "SID");
            sprintf(strBuf+strlen(strBuf), "%s", "NO SAT IN VIEW");
            sprintf(strBuf+strlen(strBuf), "\r\n");
            u2Size+= 4;
          }
      }
        if(u4FMTREG & FMT RCR)
        {
           unsigned short u2RCR;
          sprintf(strBuf+strlen(strBuf), "%10s:    ","RCR");
           memcpy(&u2RCR, &LogBuf[idx+u2Size], 2);
```

```
            sprintf(strBuf+strlen(strBuf), "%x",u2RCR);

            if(u2RCR & FMT RCR SEC)
            {
              sprintf(strBuf+strlen(strBuf), " [SEC]");
            }
          if(u2RCR & FMT RCR SPD)
            {
              sprintf(strBuf+strlen(strBuf), " [SPD]");
            }
            if(u2RCR & FMT RCR DIS)
            {
              sprintf(strBuf+strlen(strBuf), " [DIS]");
            }
           if(u2RCR & FMT RCR LN)
            {
              sprintf(strBuf+strlen(strBuf), " [BTN]");
            }
            sprintf(strBuf+strlen(strBuf), "\r\n");
            u2Size += 2;
          }
        if(u4FMTREG & FMT MS)
        {
            unsigned short u2Tmp;
          sprintf(strBuf+strlen(strBuf), "%10s:     ","MS");
            memcpy(&u2Tmp, &LogBuf[idx+u2Size], 2);
            sprintf(strBuf+strlen(strBuf), "%d", u2Tmp);
            sprintf(strBuf+strlen(strBuf), "\r\n");
            u2Size += 2;
        }


        //check sum
        u2Size += 2;

          ////////////////////////////////
          if((idx+u2Size)>LOG SECTOR SIZE)
        {
           break;
        }
          else
          {
          int i;
            for(i=0;i<u2Size;i++)
          {
               if(LogBuf[idx+i] != 0xff)
               break;
          }
            if(i==u2Size)
          {
            break;
              }

          if(Checksum Verify())
          {
             u4Cnt++;
             fprintf(out, "(%d)=====================================================\r\n", u4Cnt);
             fprintf(out, "%s",strBuf);
          }

          idx += u2Size;

        }
        if(idx >= LOG SECTOR SIZE)
            break;
      }
      return u4Cnt;
}
```