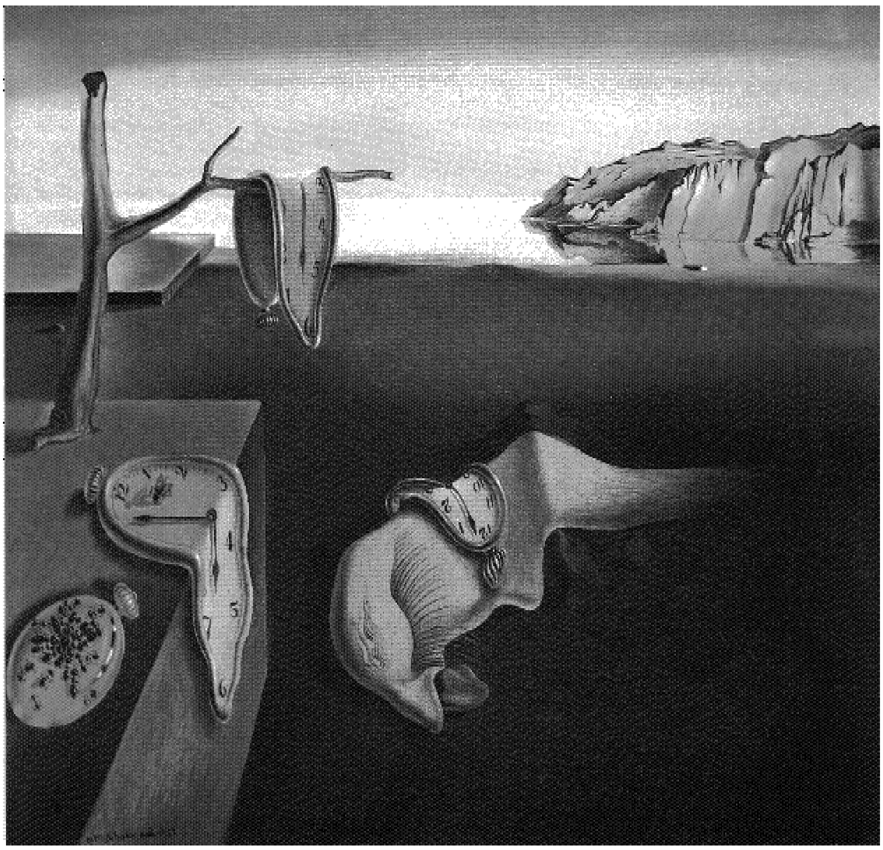
Institut für Automation
Abt. für Automatisierungssysteme

Technische Universität Wien

Projektbericht Nr. 183/1-97
October 1999

GPS Device-Driver Software Documentation

Martina Umlauft and Ulrich Schmid



Salvador Dali, "Die Beständigkeit der Erinnerung"

GPS Device-Driver Software Documentation

Martina Umlauft
MNr.: 9026329
Email: e9026329@stud1.tuwien.ac.at

Ulrich Schmid
Technische Universität Wien
Department of Automation
Treitlstraße 1, A-1040 Wien
Email: s@auto.tuwien.ac.at
Phone: 58801-18325

Revision: 1.18
March 15, 2000

Abstract

This report describes the *GPS Device-Driver* software[1] that makes available accurate time information from GPS receivers to pSOS$^+$-based target systems equipped with our *Network Time Interface* M-Module (NTI). Its generic design provides a common framework for different models[2] of GPS timing receivers, which makes it easy to add support of additional receivers. Owing to its reentrant design, the GPS-Driver can even handle multiple GPS receivers simultaneously. Application programmers are provided with a simple API, which guarantees reasonable control over the receiver but hides any model-specific handling. Utilizing the NTI's hardware timestamping capabilities, the GPS-Driver periodically calls a user-supplied callback function and delivers a sub-microsecond-accurate timestamp of the appropriate 1 pps signal transition. During operation, elaborate validity checking of the receivers 1 pps and RS232-output and flexible error handling is provided.

[1]This work is part of our SynUTC-project (*http://www.auto.tuwien.ac.at/Projects/SynUTC/*), which received funding from the Austrian Science Foundation (FWF) grant P10244-ÖMA, the OeNB "Jubiläumsfonds-Projekt" 6454, the BMfWV research contract Zl.601.577/2-IV/B/9/96, and the Austrian START programme Y41-MAT.

[2]The current revision supports the Motorola VP Oncore only.

Contents

1 Introduction

The GPS Device-Driver (*GPS-Driver* for short) has been designed for pSOS$^+$-based target systems, which are connected to one or more GPS timing receivers via RS232 interface, 1 pps signal and an optional[3] digital status signal according to Figure 1. The transition of the 1 pps signal announces the instant when some particular point-in-time (GPS-time or UTC) has been reached, the status signal indicates a valid/non-valid output. The RS232 data, which is usually supplied several 10 ms after the 1 pps, reveals the point-in-time of the last 1 pps transition, usually along with additional status information.

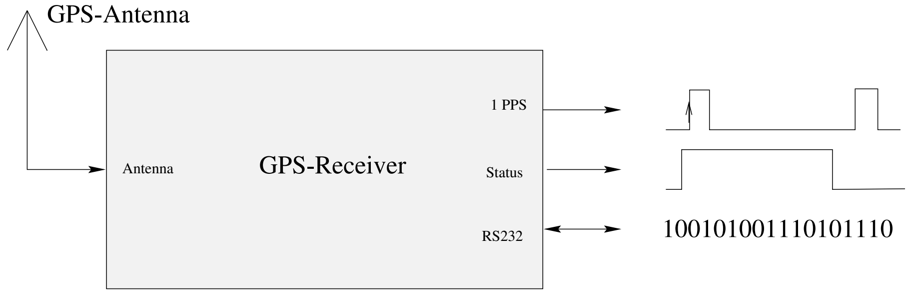


Figure 1: GPS-Receiver Interface

Whereas the RS232 interface can be connected to any standard pSOS$^+$serial channel, our custom *Network Time Interface* (NTI) M-Module [Man99] is required for interfacing the 1 pps and status output. A single NTI can service up to three GPS receivers via its three independent *GPS-Units* (GPUs). Each GPU can be programmed to timestamp incoming 1 pps transitions and optionally trigger an interrupt upon those events.

The GPS Device-Driver provides the application with an easy to use API (see Section 4) through which a supported GPS-Receiver can be controlled and time information retrieved. The driver takes care of all necessary initialization of the NTI M-Module, the RS232 interface and the Motorola VP Oncore. It then processes the 1 pps interrupts and RS232 messages. Special attention is paid to errors and faults (lost 1 pps pulses, lost or wrong RS232 messages) as wrong time announcements due to wrong 1 pps $\leftrightarrow$ RS232 message assignments would be in the 1 second (!) range.

The API has been designed with particular emphasis upon hiding receiver-specific handling. This goal has in fact been extended to the internal software architecture of the GPS-Driver, which consists of a *receiver-independent* and a *receiver-dependent* part (see Section 2). In order to support a certain model of a GPS receiver, all that needs to be done is to write the receiver-dependent part and to enter the required "linking information" in the receiver-independent one.

Still, the receiver-dependent part also contains an entire application that is not part of the GPS-Driver, but rather required for testing purposes: A *GPS-Emulator* (see Section 5.2), which emulates the particular GPS receiver at the interface level shown in Figure 1. A test cycle can be set up to simulate different sorts of faults and errors and their combinations, like lost or spurious 1 pps pulses, lost or garbled

[3]Unlike more expensive models, the Motorola VP Oncore does not have a digital status signal.

RS232 messages, wrong time information, etc. A hardware + software like this is needed, since real GPS receivers cannot usually be directed to produce such an erroneous output deliberately.

2 Architecture

2.1 General Overview

The GPS Device-Driver consists of two parts, namely, a receiver independent part and usually multiple receiver-dependent parts. Figure 2 shows the basic building blocks.

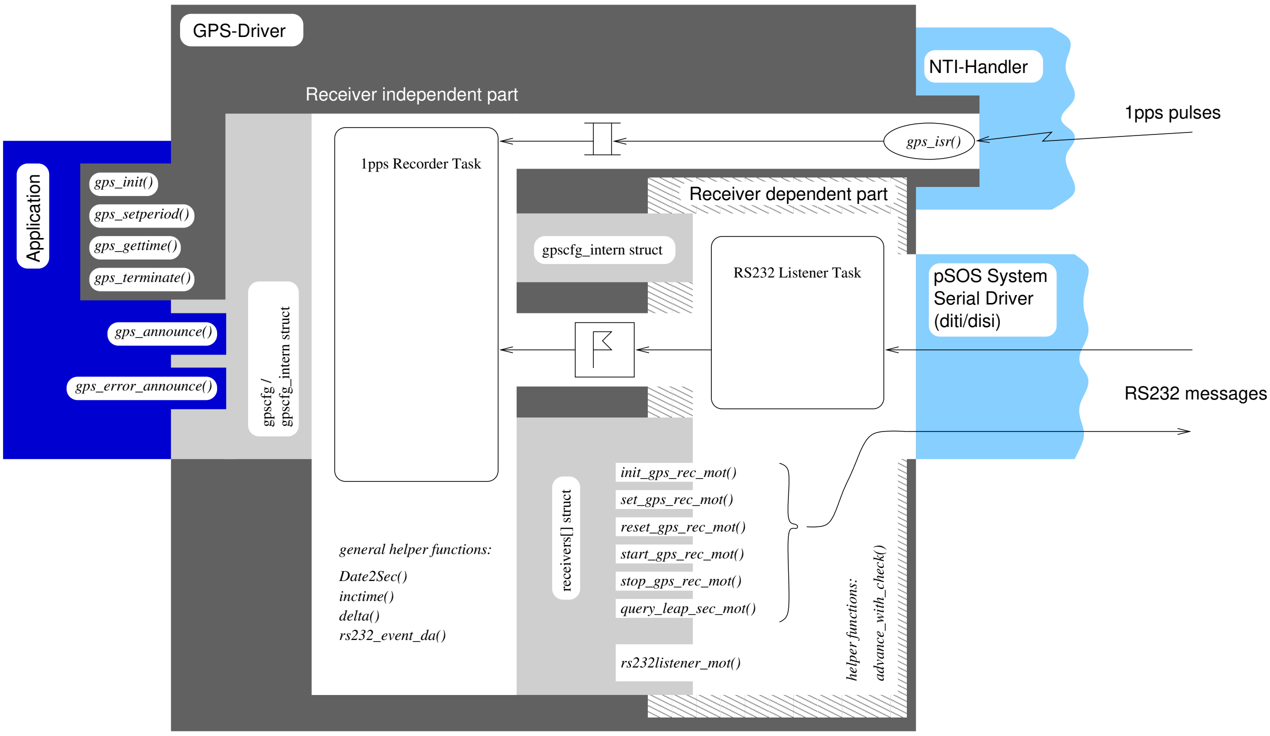


Figure 2: GPS Device-Driver architecture

The GPS-Driver as a whole is connected to the outside world by means of its API (see Section 4), the low-level NTI-Handler [SM99], and the pSOS$^+$ serial driver (diti/disi). Internally, one can distinguish three parts:

- The *receiver-independent part* consists of the 1 pps-Recorder task, *gps_isr()*, API functions and several helper functions.
- The *receiver-dependent part* consists of the RS232-Listener task, receiver specific functions and helper functions.
- The struct *receivers[]* is used as a "directory" of supported receivers, providing the required linking information for pasting together all parts. Moreover, there is a major data struct *gpscfg_intern* that contains all the information shared between receiver-dependent and -independent part.

2.2 Overview of Operation

There are basically three phases in the lifetime of the GPS-Driver:

1. An instance of the driver is created by the application via *gps_init()* (see Section 4). It sets up the internal configuration structure according to the given information (like receiver type, RS232 device number, NTI slot and GPU, etc.), initializes the GPS-Receiver and starts the RS232-Listener and 1 pps-Recorder tasks.

   After successful initialization, the application can activate the supply of time information by calling *gps_setperiod()*. Note that it is possible to re-initialize an already existing instance of a GPS-Driver at any time, e.g., to enforce a different configuration.

2. During normal operation, the GPS-Driver gathers 1 pps transitions and RS232 messages. If they pass a number of sanity checks, the user-supplied callback to *gps_announce()* is called at the requested instants. It delivers the local NTI-timestamp and the GPS-time of the responsible 1 pps pulse to the application. In *immediate announce mode*, this is done before the RS232 message of the triggering 1 pps pulse arrived. If *post announce mode* has been selected, *gps_announce()* is called after reception of the message.

   If the GPS-Driver detects errors in the timing relation of the 1 pps transitions and/or in the RS232 messages, it enters an elaborate error handling and recovery procedure. It distinguishes both soft and hard errors, which are announced to the application via the callback function *gps_error_announce()*. Further details can be found in Section 3.

3. If an instance of the GPS-Driver is no longer needed, it can be terminated by calling *gps_terminate()*.

As the GPS-Driver is reentrant, it is of course possible to start several different instances simultaneously.

2.3 GPS Device-Driver Taskmap

The taskmap of the GPS Device-Driver is shown in Figure 3. Note that it contains both the receiver-dependent and -independent part.

There are three different threads of control involved, which perform the following steps:

Interrupts signaling 1 pps transitions

1. Every 1 pps pulse triggers an NTI-Interrupt.
2. *gps_isr()* writes the UTCSU timestamp in *timex* format into the 1 pps queue for this slot and gpu.

RS232-Listener-task processing RS232 data

1. RS232 data is parsed and preprocessed by the RS232-Listener task.
2. The RS232-Listener writes RS232 time, last UTCSU timestamp, a valid tag and a leap second pending field[4] to *RS232dat* (see Section 6.2).
3. The 1 pps-Recorder is notified of new data in *RS232dat* via the RS232-Event.

[4]In this version of the driver UTC mode is not implemented, therefore the leap second field is not written by the RS232-Listener task and ignored by the 1 pps-Recorder task.

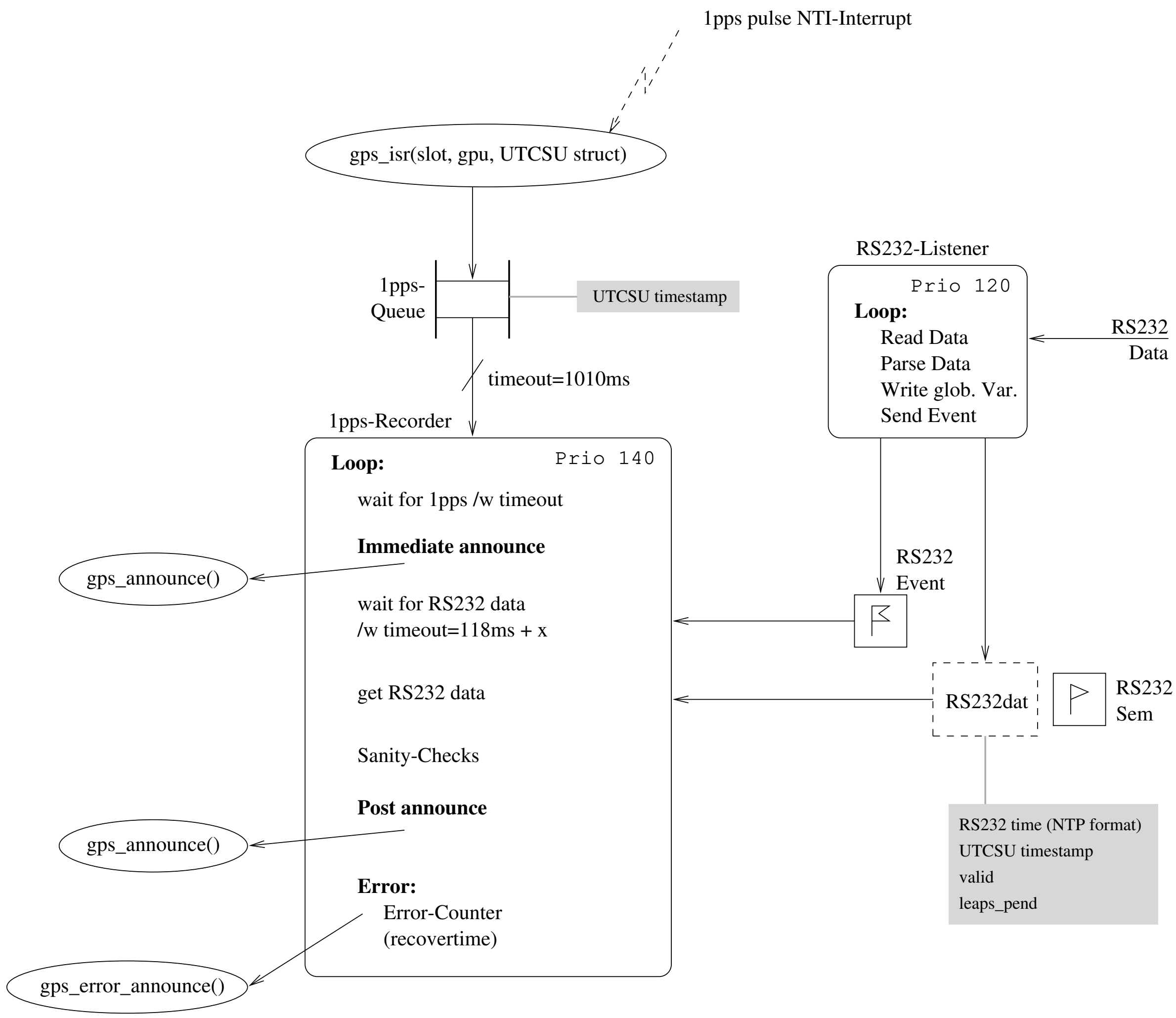


Figure 3: GPS Device-Driver taskmap

Recorder-task processing messages from 1 pps-queue

1. The 1 pps-Recorder task waits at the 1 pps queue for 1 pps pulses with timeout (see Section 4.5).
2. To measure RS232 timeout, an event timer is set up. In case of timeout the event *RS232TOUT_EV* is received before the event *RS232EVENT* from the RS232-Listener task.
3. The 1 pps timestamp is tested for validity (see Section 3.3).
4. The driver can be in one of two modes: immediate announce and post announce (see Item 7 below). If the driver is in immediate announce mode and not in hard error mode it now calls *gps_announce()*. Because RS232 data is not available yet the driver increments the last RS232 time information by one second and announces this value.
5. The task waits for RS232 data with timeout as explained above. Timeout values can be found in Section 4.5.
6. RS232 data is tested for validity (see Section 3.3).
7. If the driver is in post announce mode and not in hard error mode it now calls *gps_announce()*.

8. If an error occurred and the driver is not already in hard error mode, it now calls *gps_error_announce()*, increases the error counters and retriggers *recovertime*.
9. The task checks whether its configuration (*recovertime*, ...) has changed, performs various cleanups (reading the timeout event in case there was no timeout, advancing its history, ...). If the driver is in hard error mode and has received *recovertime* seconds worth of good 1 pps pulses and RS232 messages, it resumes normal operation mode.

The driver supports handling of soft errors (eg. temporary loss of RS232 messages), see Section 3.4.

2.4 GPS-Emulator

The GPS-Emulator assumes a dedicated target system incorporating an NTI, the APPDUTY output of which is used to emulate the receivers 1 pps output. It is connected to the GPS-Driver's target via an RS232 cable and a special "inter-NTI" connector cable. The task map of GPS-Emulator is shown in Figure 4.

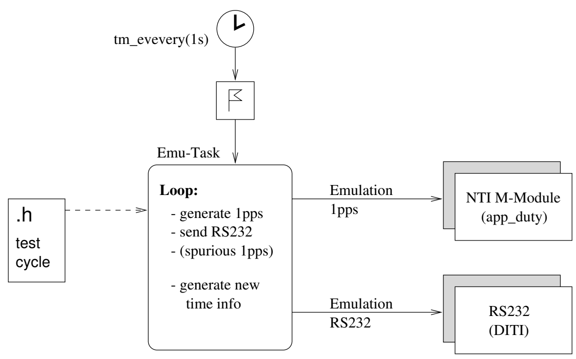


Figure 4: GPS-Emulator taskmap

Every second, the GPS-Emulator sends 1 pps pulses and receiver-specific RS232 messages according to the test cycle specified (see the more detailed description of *testdaten_mot.h* in Section 5.2). In conjunction with the simple demo application described in Section 5.1, this allows to record the behavior of the GPS-Driver in presence of particular testcases. Therefore, using an advanced comparison tool like Emacs ediff in conjunction with an earlier record, it is easy to re-run a set of testcases after changing the GPS-Driver to verify correct execution.

3 Sanity Checks and Error Handling

There are many sources of error, which might affect any part of the timing input. According to [HS97], there may be missing, wrong, and spurious 1 pps transitions both due to GPS receiver and interfacing problems. Moreover, there may be a wealth of errors in the RS232 data, ranging from communication problems over receiver firmware bugs up to errors in the GPS space and control segment, cf. [Dan97]. Therefore, an elaborate and robust error checking and recovery scheme is very important.

3.1 RS232 Faults

RS232 messages are preprocessed by the RS232-Listener task; only when a complete message with correct checksum is received, data is passed on to the 1 pps-Recorder. Apart from complete loss of a message we can classify 3 primary sources of low level RS232 errors (exemplified by the Motorola VP Oncore):

- Faulty start of packet (see Figure 5) with destroyed @@-Command-Id marker.
- Faulty end of packet (see Figure 6) with destroyed checksum and end marker.
- Fault in the middle of a packet(see Figure 7) with destroyed data.

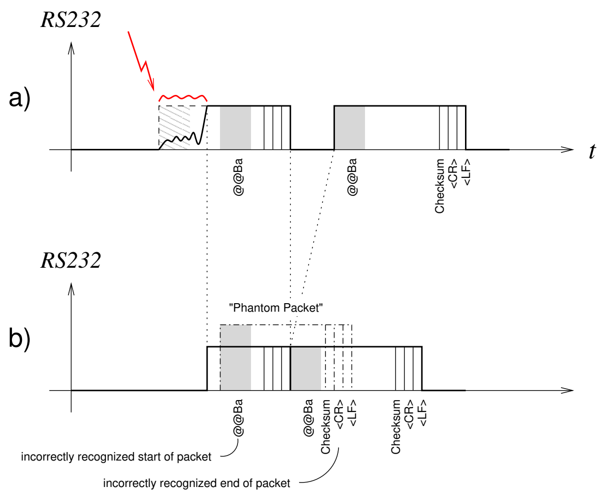


Figure 5: Faulty start of packet:
a) *timing,*
b) *order of received bytes*

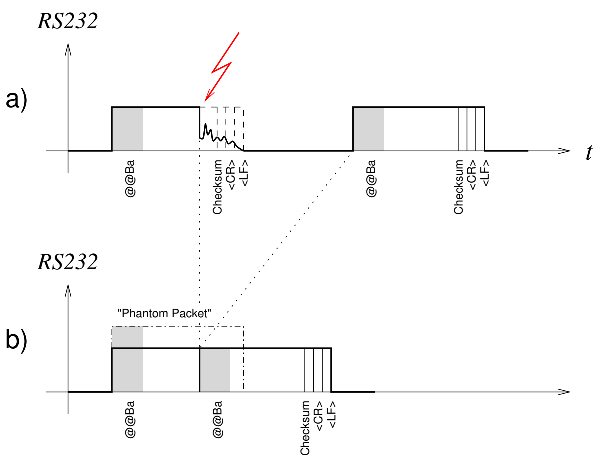


Figure 6: Faulty end of packet:
a) *timing,*
b) *order of received bytes*

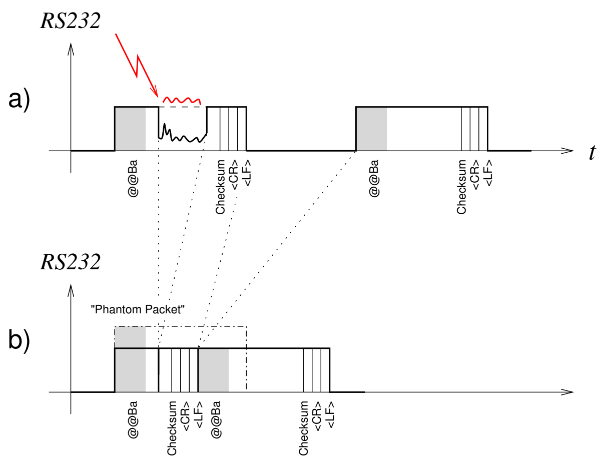


Figure 7: Fault in the middle of a packet:
a) *timing,*
b) *order of received bytes*

In all three cases the low level parser might recognize "phantom packets" if a packet start or end marker is found inside the data area. These faults can not be recognized by the parser in the RS232-Listener and must be caught by the 1 pps-Recorder.

3.2 1024 Week Rollover

As an example of a receiver-induced failure, consider that the satellites in the GPS system use a 29 bit register (see Figure 8) to count weeks and time of week since *Jan.* 6^{th} *1980, 00:00:00 UTC*. 10 bits of this register are reserved for the week number. Therefore, every 1024 weeks the GPS system experiences a week rollover (week 1023 is followed by week 0). The last such rollover occurred on *Aug.* 22^{nd} *1999, 00:00:00 UTC*.

WEEK Number	Time Of Week
10 bit	19 bit

Figure 8: 29 bit register aboard the satellites.

Receivers which are not 1024 week rollover compliant could experience problems but according to [Mot98] the Motorola VP Oncore has been designed and tested for 1024 week rollover.

3.3 Sanity Checks

The 1 pps-Recorder performs the following sanity checks to find out whether the received signal is to be considered valid or faulty. A valid signal is shown in Figure 9, examples of faulty signals are shown in Figures 10 and 11.

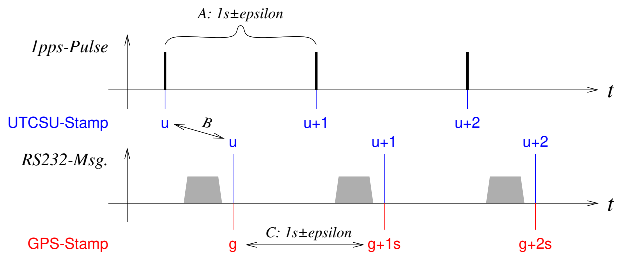


Figure 9: Valid signals from Motorola VP Oncore.

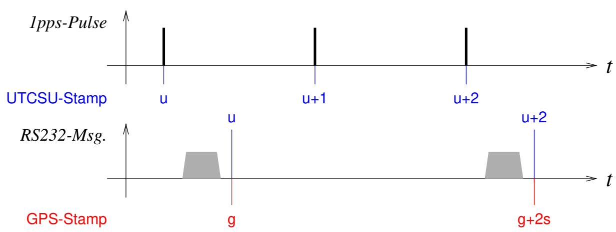


Figure 10: Lost RS232 messages.

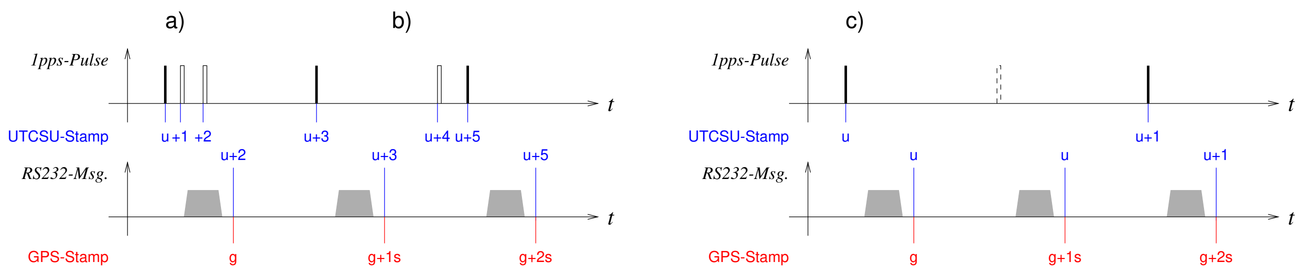


Figure 11: **a), b)** *Spurious 1 pps pulses.* **c)** *Lost 1 pps pulses.*

A signal is considered to be valid if the following assertions hold:

A two adjacent 1 pps pulses are $1 \pm \epsilon$ seconds apart.

B the UTCSU timestamp on the RS232 message equals the UTCSU timestamp of the last 1 pps pulse.

C GPS time information of two adjacent RS232 messages is $1 \pm \epsilon$ seconds apart.

D RS232 message status is valid.

3.4 Errorhandling

The GPS Device-Driver handles faults according to their severity.

1. (Least severe) lost RS232 message. Either the RS232 message was really lost or suppressed by the RS232-Listener due to garbage, wrong checksum, ...

   In case of a single (or few, see Section 4.7) lost RS232 messages, the driver will call *gps_announce()* with a calculated time value and set *warn* to TRUE.

   If more messages are lost the driver calls *gps_error_announce()*, and goes into hard error mode.

2. Temporary loss of 1 pps pulse. In this case the driver calls *gps_error_announce()* every second with an errorcode indicating weak 1 pps. When a valid 1 pps pulse is received the driver stops announcing errors immediately.

   If more than a few (see Section 4.7) 1 pps pulses are lost the driver goes into hard error mode.

3. (Most severe) hard error mode is also entered in any of the following situations:

   - time difference between two adjacent 1 pps pulses wrong (except case 2 applies)
   - time difference between announced time information of two adjacent RS232 messages wrong (except case 1 applies)
   - a fault was announced by the GPS-Receiver (message tagged as invalid)
   - loss of both 1 pps pulse *and* RS232 message

   The driver calls *gps_error_announce()* once when it enters hard error mode and stays silent until it leaves hard error mode again. The driver will stay in hard error mode until *recovertime* seconds worth of good 1 pps pulses and RS232 messages were received.

4 Application Interface

All API relevant declarations and #defines can be found in the file *gpsdrv_api.h*.

4.1 Functions provided by the GPS Device-Driver

- **int gps_init(short slot, short gpu, const gpscfg *cfg, int reset, INT32 *id)**

  Installs and initializes or reinitializes the GPS-Driver according to configuration information in *cfg*. If *reset* is set to TRUE, a reset is sent to the GPS-Receiver via the function registered in *receivers[].reset()*. Returns a "handle" *id* (address of driver internal data structure). Must be called from task context in supervisory mode.

- **int gps_setperiod(INT32 id, timex gpstime, int period)**

  Activates driver calls to *gps_announce()* every k**period*, $k \geq 0$ seconds after *gpstime* is reached. The fractional part of *gpstime* is ignored. *period* is specified in seconds; a value of 0 meaning "never".

  Calling *gps_init()* initializes *period* to 0, which means that *gps_announce()* is never called unless the application makes a subsequent call to *gps_setperiod()*. Can be called from ISR or task context.

- **int gps_gettime(INT32 id, timex *gpstime, timex *utcsutime)**

  Returns time information received from the GPS-Receiver via RS232 in *gpstime* and the UTCSU timestamp of the last 1 pps signal in *utcsutime*. Only *utcsutime* has a valid 8 bit checksum, the checksum in *gpstime* is zero. Can be called from ISR or task context.

- **int gps_terminate(INT32 id)**

  The GPS-Driver corresponding to *id* is stopped and deinstalled. Must be called from task context in supervisory mode.

4.2 Functions driver applications must implement

The following functions must be implemented by applications using the driver. These functions must be registered with the driver via *gps_init()* and should be as short as possible as they block the driver. If these functions are too slow, the driver might miss 1 pps events. These functions are called in a task context (not ISR context) by the GPS-Driver.

- **int gps_announce(short slot, short gpu, INT32 id, timex *gpstime, timex *utcsutime, short warn)**

  Announces new time information to the application. In case of a soft error (lost RS232 message) *warn* is set to TRUE, otherwise FALSE.

  The driver calls *gps_announce()* immediately after the 1 pps pulse was recognized if it runs in immediate announce mode (bit 1 of *gpscfg.opmode* was set to 0). Since RS232 time information is not available at this time the driver reports the last time information received plus 1 second. In post announce mode the driver waits until it has received time information via RS232 before it calls *gps_announce()*.

- **int gps_error_announce(short slot, short gpu, INT32 id, INT32 errorcode)**

  Announces an error (wrong or lost 1 pps pulses or too many lost RS232 messages) to the application (see also Section 3).

  In case of a hard error (wrong time difference between two 1 pps pulses, too many lost 1 pps pulses, too many lost RS232 messages, error announced by the GPS receiver) *gps_error_announce()* is called once, then the receiver stays silent until it leaves error mode again.

  In case of a temporary loss of the 1 pps signal —but while RS232 messages are correct— *gps_error_announce()* is called every second until the maximum number of lost 1 pps pulses is reached whereupon the receiver goes into error mode, announces a hard error and stays silent as long as it is in error mode.

Errors are always announced at the end of a measurement epoch, i.e., after 1 pps *and* RS232 signals are (or in case of timeout should have been) received.

4.3 Configuration data

```
typedef struct {
                        /* Application hook: Announce GPS-Zeitinfo */
    int (* gps_announce)(short slot, short gpu, INT32 id,
 timex *gpstime, timex *utcsutime, short warn);

                        /* Application hook: Announce GPS-Error */
    int (* gps_error_announce)(short slot, short gpu, INT32 id,
       INT32 errorcode);


    /*-- Receiver settings --*/
    int rectype;        /* Receiver Type:
                         * Motorola VP Oncore ... 0
                         */
    INT32 siodev; /* pSOS Major/Minor Device Number RS232 */


    /*-- GPS settings --*/
    INT32 cabledelay;   /* Delay (ns) caused by length of antenna cable */
    int fixposmode;     /* TRUE: Fixed Position Mode, FALSE: Auto. Pos.-Fix */
    INT32 latitude;     /* Position of Antenna (N-S), f. fixed Pos. Mode */
    INT32 longitude;    /* Position of Antenna (W-O), f. fixed Pos. Mode */
    INT32 height;       /* Position of Antenna (Height), f. fixed Pos. Mode */


    /*-- Driver settings --*/
    INT16 opmode;       /* Bit 0: 0/1 ... Output GPS-Time/UTC */
                        /* Bit 1: 0/1 ... Immediate/Post-RS232 Announce */
    int recovertime;    /* Recovertime (in seconds) after GPS-Error
                         * before gps_announce is called the next time
                         */

    /*-- Error counters --*/
    const int cs_err;   /* Message Errors: Checksum Errors */
    const int lost_msgs;/*                 Lost Messages   */
    const int msg_err;  /*                 Message Error   */

    const int lost_1pps;/* 1pps Errors:    Lost 1pps Pulses */
    const int _1pps_err;/*                 Timestamp wrong  */

} gpscfg;
```

If *fixposmode* is set to TRUE, *latitude*, *longitude* and *height* must contain valid values in receiver specific format (see [Mot94]). If *fixposmode* is set to FALSE their values are ignored.

Error counters are reset to 0 upon start of an instance of the driver and count up for as long as that instance runs. Reinitializing the driver instance with *gps_init()* does not reset the error counters.

The following #defines are available to help in filling out the *gpscfg* structure:

```
#define RECT_MOTOROLA  0          /* Receiver type */

#define OUTP_GPS  0x0000          /* opmode: GPS time */
#define OUTP_UTC  0x0001          /* opmode: UTC time */
#define IMME_ANN  0x0000          /* opmode: immediate announce */
#define POST_ANN  0x0002          /* opmode: post announce */
```

4.4 Time format

```
typedef struct {
    INT32 rms;   /* Checksum (8 bit) + Higher-Part NTP-Zeit (NTP-Bit 8...31) */
    INT32 rts;   /* Lower-Part NTP-Zeit (NTP-Bit -24...7) */
} timex;
```

Time information originating from the RS232 messages does not contain a valid checksum; therefore the checksum byte is set to zero by the driver.

4.5 Timeout

Timeout for 1 pps pulses is set to 1010 ms and can be changed by setting the number of pSOS$^+$-ticks in *gpsdrv_api.h/_1PPSTIMEOUT* in (see also Section 4.7). RS232 timeout is receiver dependent and must be set in pSOS$^+$-ticks in *gpsdrv_api.c/receivers.rs232timeout*.

For the Motorola VP Oncore RS232 timeout is calculated as follows:

$$118\ ms + x = \sum \begin{cases} 0 - 50\ ms & : \text{Start of RS232 data transfer} \\ \approx 68\ ms & : \text{Duration data transfer @@Ba-Response} \\ x & : \text{Overhead for preprocessing and pSOS}^+ \end{cases} \qquad (1)$$

In this Version of the GPS-Driver RS232 timeout is set to 300 ms.

4.6 Error- and Return-Codes

Available errorcode #defines:

```
#define GPSERR_NOMEM 1  /* not enough memory for config structure */
#define GPSERR_NONTI 2  /* no NTI board found */
#define GPSERR_NIMPL 4  /* feature not implemented eg. UTC-mode */
```

Available returncode #defines:

```
#define GPSOK_INIT 0
```

A translation from error/warn code numbers to plain text for the error and warn codes input to *gps_announce()* and *gps_error_announce()* can be found in the following global:

```
char * err_names[] = { "ok",
                       "weak RS232",
                       "weak 1pps",
                       "error"
};
```

4.7 Controlling Driver behavior

The following options can be set at compile time (in *gpsdrv_api.h*).

```
#define _1PPSTIMEOUT  101          /* timeout for 1pps pulse [ticks] */

#define MAXSKIPPED    2            /* how many skipped 1pps in a row? */
#define MAXRS232FAIL  3            /* how many lost rs232 msgs in a row*/

#define EPSILON       0x00000FFF   /* new time = old + (1s +- EPSILON) */
                                   /* [ntp format] */
```

$MAXSKIPPED \geq 0$ is the number of 1 pps pulses, $MAXRS232FAIL \geq 0$ is the number of RS232 messages that may be lost before the driver switches into hard error mode.

EPSILON is the lower part of a *timex* format struct; the example above indicates a value of $15 * 16^{-4} + 15 * 16^{-5} + 15 * 16^{-6}s = 244\mu s$. Two 1 pps pulses must be $1s \pm \epsilon$ apart to be considered correct.

5 Applications

5.1 Using the Demo Application

The simple demo application supplied with the GPS-Driver package can be started by calling startgps from its directory. It first asks for the desired receiver type and the number of the GPU $(0 \ldots 2)$ used and then starts executing. Its behavior as well as the remaining configuration parameters are controlled by setting the appropriate #defines in *testappl.c*:

```
#define ALOOPS  0                    /* how many application loops?
                                      * 0 ... for ever
                                      */
#define SLOT  0                      /* set slot here */
#define SIO_PORT_GPS   1             /* SIO-Port for GPS-Reciever */


#undef DUAL                          /* use 2 GPS receivers? */
#define SLOT2 0                      /* slot for DUAL */
#define SIO_PORT_GPS2  2             /* SIO-Port for 2nd GPS-Reciever */
```

Additional Remarks:

- If ALOOPS is set to a nonzero value the application will quit the loop after this number of announcements, call *gps_gettime()* and terminate the driver. This is useful for capturing fixed-length testcases in conjunction with the GPS-Emulator.
- If DUAL is defined, the application will install two instances of the driver. To use this feature the application must be compiled to redirect stdout to a netterm[5] because the CPU board has only two serial ports.
- Debug output for the application is turned on or off by defining or undefining APPLTEST in *gpsdrv_debug.h*.

[5]This is done by activating the commented-out compilation of root.c in makefile. Note, however, that debug output must be set to a minimum otherwise netterm output will slow the driver to such a degree that it won't be able to keep up with its timing requirements!

5.2 Using the GPS-Emulator / Test Cycle Setup

Each supported GPS receiver should be supported by a GPS-Emulator capable of emulating the following signals and faults:

- any receiver specific RS232 message (actually any stream of RS232 data): correct messages, garbled, wrong or invalid messages
- a series of properly timestamped response messages (in case of the Motorola VP Oncore: @@-Ba response messages)
- lost RS232 messages
- correct 1 pps pulses
- lost 1 pps pulses
- spurious 1 pps pulses
- any combination of the above

Note that the basic functionality of the GPS-Emulator is the same for all receivers. However, both its simplicity and message-dependency suggest different implementations instead of a generic one with receiver-dependent plug-ins.

The emulator is controlled by an include file (like *testdaten_mot.h* for the Motorola VP Oncore), which specifies all test data and the test cycle. The GPS-Emulator arms an event timer (tm_evevery) for 1 second intervals and runs in a continuous loop through the test cycle. In each iteration, a correct 1 pps pulse, an RS232 message and a spurious 1 pps pulse can be created. To appear lifelike, correct 1 pps pulses end after the RS232 message is sent (see e.g. Figure 15).

RS232 response messages can be stamped with current time information (so one does not have to create hundreds of messages with different timestamps manually to perform a continuous test). To achieve this, the emulator gets the current pSOS^{+}time and writes the corresponding hours, minutes, seconds and fractional seconds into the RS232 message stream. If required, The correct checksum for the message is of course also caluculated.

All RS232 messages that can be used in a test cycle are stored in the array *test_msgs[]* which is of type *test_msg* shown below.

```
typedef struct {

    unsigned char msg[100];    /* Command String */
    int length;                /* actual length of above */
    int calc_cs;               /* calculate checksum or keep
                                * the one from the string ?
                                */
    int stamp;                 /* generate a timestamp ?
                                * (also calcs checksum)
                                */
} test_msg;
```

msg can be any stream of bytes, a correct message or garbage on the line.

length specifies the actual lenght of the message. To simulate an ommitted message, set the length field to 0.

calc_cs specifies whether the checksum should be calculated automatically. Setting it to FALSE allows for simulation of messages with wrong checksums.

stamp specifies if a timestamp should be generated and inserted into the message.

The RS232 test cycle is specified by filling out the global array *test_control[]*. The indices of the messages in *test_msgs[]* must be set up here in the order these messages are to be sent. In the example shown below the emulator will first send the message stored in *test_msg[1]*, then *test_msg[3]*, then start the cycle over again.

```
int test_control[TC_MAX] = {

    1,                         /* 1st Message: test_msg[1] */
    3                          /* 2nd Message: test_msg[3] */

};
```

The 1 pps test cycle is specified by filling out the global array *test_1pps[]* shown below.

```
int test_1pps[TP_MAX] = {

    1,                         /* correct 1pps */
    0,                         /* lost 1pps */
    2,                         /* spurious 1pps */
    3                          /* spurious 1pps with 1 tick offset */

};
```

A value of 1 means that a correct 1 pps pulse, 0 a lost pulse, 2 a correct pulse and an additional spurious pulse is sent. Values greater than 2 mean that the spuriuos pulse is sent $n-2$ pSOS$^+$-ticks late.

It should be possible to use the GPS-Emulator in conjunction with the demo application of Subsection 5.1. The one for Motorola VP Oncore, for example, can be started from the directory containing the demo application by means of *startemu_mot*. Note carefully that the NTI of the emulator must have the APPDUTY-output jumper set in order to produce 1 pps pulses.

In order to gather the GPS-Driver's behavior in presence of the testcases, the output printed on the terminal should be saved in a file (cut and paste). It can then be compared with a valid output via tools like Emacs ediff. Note that in order to collect fixed-size shots, one should set ALOOP in *appltest.c* to the number of testcases.

6 Implementation Receiver-Independent Part

6.1 Global Conventions

- All functions returning *int* values return 0 for OK and Errorcodes > 0
- The NTI and GPU of the receiver is identified through two input parameters:

  short slot (0 ... MAX_SLOT): Slot of the NTI-M-Module on the Carrier-Board
  short gpu (0 ... MAX_GPU): Number of the GPU on this board
- All #defines and typedefs which are relevant for the API can be found in the include file *gpsdrv_api.h*.

6.2 Internal Configuration data

Internally the driver uses an extended version of the *gpscfg* structure introduced in Section 4. The structure *RS232dat* is used for communication between the RS232-Listener task and the 1 pps-Recorder task (see also Section 2.3).

```
typedef struct {
                        /* Application hook: Announce GPS-Zeitinfo */
    int (* gps_announce)(short slot, short gpu, INT32 id,
 timex *gpstime, timex *utcsutime, short warn);

                        /* Application hook: Announce GPS-Error */
    int (* gps_error_announce)(short slot, short gpu, INT32 id,
       INT32 errorcode);


    /*-- Receiver settings --*/
    int rectype;        /* Receiver Type:
                         * Motorola VP Oncore ... 0
                         */
    INT32 siodev; /* pSOS Major/Minor Device Number RS232 */


    /*-- GPS settings --*/
    INT32 cabledelay;   /* Delay (ns) caused by length of antenna cable */
    int fixposmode;     /* TRUE: Fixed Position Mode, FALSE: Auto. Pos.-Fix */
    INT32 latitude;     /* Position (N-S), f. fixed Pos. Mode */
    INT32 longitude;    /* Position (W-O), f. fixed Pos. Mode */
    INT32 height;       /* Position (Height), f. fixed Pos. Mode */


    /*-- Driver settings --*/
    INT16 opmode;       /* Bit 0: 0/1 ... Output GPS-Time/UTC */
                        /* Bit 1: 0/1 ... Immediate/Post-RS232 Announce */
    int recovertime;    /* Recovertime (in seconds) after GPS-Error
                         * before gps_announce is called next time
                         */

    /*-- Error counters --*/
    int cs_err;         /* Message Errors: Checksum Errors */
    int lost_msgs;      /*                 Lost Messages   */
    int msg_err;        /*                 Message Error   */

    int lost_1pps;      /* 1pps Errors:    Lost 1pps Pulses */
    int _1pps_err;      /*                 Timestamp wrong  */


    /*----- end API visible ------------------------------------------------*/


    /*-- Driver settings via api functions --*/
    short slot;         /* gps_init: slot of NTI-Board */
    short gpu;          /* gps_init: gpu of Receiver [0..2] */

    timex per_gpst;     /* gps_setperiod: Starting time */
    int period;         /* gps_setperiod: Period of announce */

    INT32 terminate;    /* gps_terminate: kill yourself */


    /*-- Driver return values --*/
    timex gpstime;      /* gps_gettime: GPS-Time */
    timex utcsutime;    /* gps_gettime: UTCSU-Time */


    /*-- Driver internals --*/
    INT32 recorder_tid; /* Stuff for 1pps-Recorder task */
    INT32 recorder_args[4];

    INT32 listener_tid; /* Stuff for RS232-Listener task */
    INT32 listener_args[4];

    struct {            /* Comm. RS232-Listener -> 1pps-Recorder */
        timex data;     /* Time received from GPS */
        timex stamp;    /* UTCSU timestamp recorded at the last 1pps */
        int valid;      /* no fault announced by receiver */
        int leaps_pend; /* leap second pending: (ignored in GPS-Time mode)
                         * 0 ... no leap sec pending
                         * 1 ... addition of 1 sec
                         * 2 ... substraction of 1 sec
                         */
    } RS232dat;

    INT32 smid_rs232;   /* Semaphor protecting RS232dat */

    short dead1pps;     /* 1pps-Recorder has terminated */
    short deadrs232;    /* RS232-Listener has terminated */

} gpscfg_intern;
```

6.3 1 pps-Recorder

The 1 pps-Recorder implements the state machine shown in Figure 12. Signal validity is determined as shown in Section 3. To achieve correct behavior, all calls to *gps_error_announce()* are made at the end of the measurement epoch — otherwise the driver would incorrectly announce weak 1 pps state before knowing whether the following RS232 message was valid when in immediate announce mode.

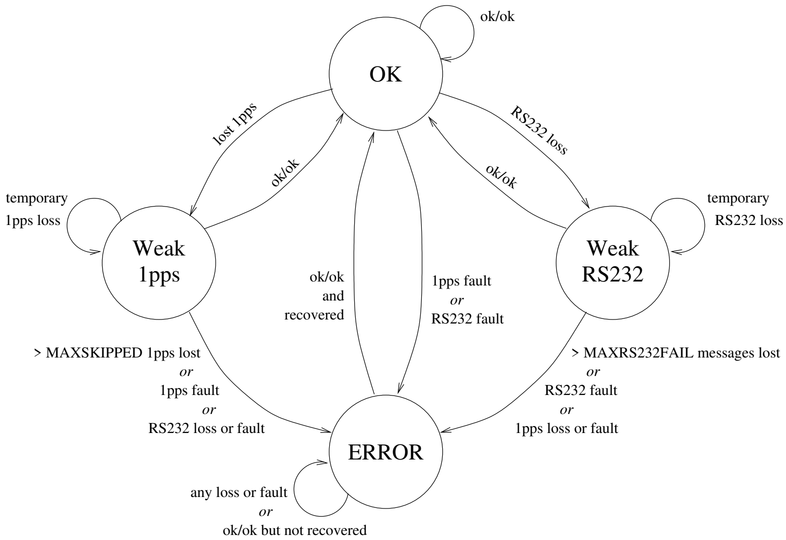


Figure 12: State machine of 1 pps-Recorder

6.4 GPS-Driver Package - Files, Functions, Directory Structure

The GPS-Driver package consists of two separate pSOS$^+$-applications; a demo application using the driver and the GPS-Emulator application. The GPS Device-Driver directory structure is shown in Figure 13.

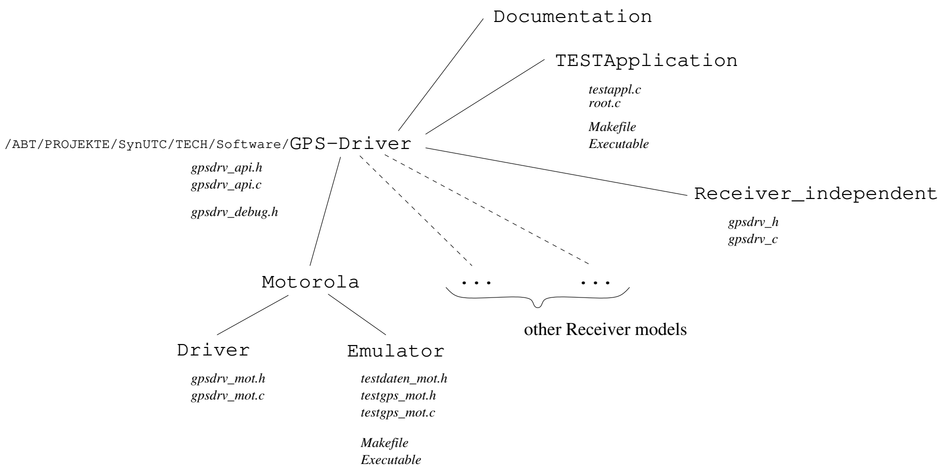


Figure 13: GPS Device-Driver directory structure

GPS-Driver

GPS-Driver API `gpsdrv_api.h`, `gpsdrv_api.c`

is included by Applikation (`testappl.c`), GPS-Driver(`gpsdrv.h`, `gpsdrv.c`, `gpsdrv_mot.h`, `gpsdrv_mot.c`), GPS-Emulator(`testgps_mot.c`)

is linked with Applikation, GPS-Driver, GPS-Emulator

contents

- gpscfg-Struct
- defines for opmode, rectype, error codes, driver behavior
- declaration of API-Functions gps_init(), gps_setperiod(), gps_gettime(), gps_terminate(), gps_announce() and gps_error_announce()
- receivers[] array

receiver-independent part `gpsdrv.h`, `gpsdrv.c`

includes `gpsdrv_api.h`, `gpsdrv_debug.h`

is included by GPS-Driver

is linked with Demo application, GPS-Emulator

contents

- gpscfg_intern struct
- implementation of API-Functions gps_init(), gps_setperiod(), gps_gettime(), gps_terminate()
- _1ppsRecorder() task
- driver internal functions Date2Sec(), inctime(), delta(), rs232_event_da(), gps_isr(), ci_inta(), ci_intt()

receiver-dependent part (for Motorola VP Oncore) `gpsdrv_mot.h`, `gpsdrv_mot.c`

includes `gpsdrv_api.h`, `gpsdrv.h`

is included by GPS-Driver

is linked with Demo application, GPS-Emulator

contents

- defines MOT_CHANNEL_ERROR, MOT_REC_STATUS_ERROR
- rs232listener() task
- motorola specific functions `init_gps_rec_mot()`, `reset_gps_rec_mot()`, `stop_gps_rec_mot()`, `set_gps_rec_mot()`, `query_leap_sec_mot()`

Demo application (executable in `GPS-Driver/TESTApplication`) `testappl.c`

includes `gpsdrv_api.h`, `gpsdrv_debug.h`

must link `gpsdrv.o`, `gpsdrv_mot.o`

contents

- main() of demo application
- implementation gps_announce(), gps_error_announce()
- defines to control the application: SLOT, GPU, DUAL, ALOOPS

GPS-Emulator (executable in `GPS-Driver/Motorola/Emulator`)

for receiver model Motorola VP Oncore

Definition of test cycle `testdaten_mot.h`

includes `gpsdrv_mot.h`

is included by GPS-Emulator only

contents

- global variables test_msgs[], test_control[], test_1pps[]
- defines for test cycle
- init_test_msgs()

Emulator `testgps_mot.h`, `testgps_mot.c`

includes `gpsdrv_api.h`, `testdaten_mot.h`, `gpsdrv_debug.h`

is included by GPS-Emulator

is linked with GPS-Driver API (to get access to receivers[])

contents

- defines SIO_PORT_EMU, PWIDTH
- GPS-Emulator main()

Debug settings for the whole package `gpsdrv_debug.h`

includes -
is included by GPS-Driver, GPS-Emulator, Applikation
contents Debug defines for the whole package.

6.5 Other Implementation Details

kc_iojtable[SC_DEV_SERIAL] in *root.c* is changed to allow for calling IOCTRL of the DITI serial driver instead of the console driver IOCTL calls (see listing in Appendix A).

7 Implementation Receiver-Dependent Part Motorola VP Oncore

7.1 Receiver settings

Receiver settings must be entered into the global array *receivers[]* of type *rec_settings* (in *gpsdrv_api.c*), which is indexed by the receiver type.

```
typedef struct {

    INT32 sioflags_i;   /* DITI Input: HW/SW Flow Control, EOL Chars,...*/
    INT32 sioflags_o;   /* DITI Output: EOL Definition, ... */
    INT32 sioflags_c;   /* DITI Control: Baud rate/Start/Stopbits, Parity,...*/
    INT32 sioflags_l;   /* DITI Line Discipline */
    int sio_timeout;    /* number of ticks timeout [VTIME] for diti driver */
    int rs232timeout;   /* timeout for RS232 event [PSOS ticks] */

    char  lowtohigh;    /* TRUE:  rising edge active 1~pps,
                         * FALSE: falling edge active 1~pps
                         */
    char  statusok; /* Status-Signal (0/1) for OK (= no GPS error)
                         * -1... ignore
                         */
    /*---------- Functions -----------*/
                                      /* RS232 Listener Task */
    void (* rs232listener)(INT32 arg0, INT32 arg1, INT32 arg2, INT32 arg3);
    int (* init)(INT32 id);           /* initialize receiver */
    int (* reset)(INT32 id);          /* reset receiver */
    int (* start)(INT32 id);          /* start receiver output */
    int (* stop)(INT32 id);           /* stop receiver output */
    int (* set)(INT32 old_id,         /* set receiver settings */
              const gpscfg * new,     /* old_id = NULL -> force set */
              int all);               /* set everything ? */
    int (* query_leap_sec)(INT32 id);  /* query pending leap second */

} rec_settings;
```

All implementations of receiver dependent parts must provide the following functions which must be registered in *receivers[]*:

- **int rs232listener(INT32 arg0, INT32 arg1 INT32 arg2, INT32 arg3)**
  Function containing the RS232-Listener task. This task must listen on the RS232 port specified in *gpscfg_intern* and fill out the *RS232dat* structure. A handle to *gpscfg_intern* is passed to the task in *arg0*.
- **int init(INT32 id)**
  Initializes the GPS-Receiver to the required protocol mode. In case of the Motorola VP Oncore, the receiver is set to Motorola binary protocol.

- **int reset(INT32 id)**

  Sends a reset command to the receiver. In case of the Motorola VP Oncore, the @@Cf command (SET TO DEFAULTS, see [Mot94], p. 6.96) is used. The application can force a receiver reset by calling *gps_init()* with *reset* = TRUE. Note that the Motorola VP Oncore will lose its almanach data when reset and re-fetching the almanach is time consuming.

- **int start(INT32 id)**

  Sets the receiver to continuosly output its time information message every second. An @@Ba response message ([Mot94], p.6.64) is requested from the Motorola VP Oncore.

- **int stop(INT32 id)**

  Tells the receiver to stop continuos output of the time information message. Some receivers, like the Motorola VP Oncore will send one more (@@Ba) message as an answer before turning silent.

- **int set(INT32 old_id, const gpscfg * new, int all)**

  Sends all messages necessary to set the receiver after a reset. Also, this function is used to change the settings for cabledelay and fixed position mode (latitude, longitude and height) when re-initializing the GPS-Driver.

  In this implementation, if neither *old_id* is NULL nor *all* is TRUE, the function registered as *set()* will only send the following commands to the Motorola VP Oncore if their values in *new* are different from their values in *old*:

  @@Az Cabledelay, [Mot94], p. 6.52

  @@At Position hold option, [Mot94], p. 6.40

  @@Ad Latitude, [Mot94], p. 6.8

  @@Ae Longitude, [Mot94], p. 6.10

  @@Af Height, [Mot94], p. 6.12

  If either *old_id* is NULL or *all* is TRUE the above messages will always be sent to the Motorola VP Oncore. In addition, the following messages will also be sent:

  @@Cg Position fix, [Mot94], p. 6.98

  @@Aw UTC Time Correction, [Mot94], p. 6.46

  @@Aj xDOP type, [Mot94], p. 6.21

- **int query_leap_sec(INT32 id)**

  For receivers that don't supply leap second information in their time information message this function must send an appropriate request to the receiver. Since this version of the GPS-Driver doesn't support UTC time mode this function is empty.

7.2 Protocol Details

For Communication with the Motorola VP Oncore the Motorola binary format (see Figure 14 and [Mot94], p. 5.1) is utilized. This format is only available in a 9600,8,N,1 setting. According to [Höc96], p. 71 the data in the data sequence is little endian encoded.

Protocol timing of the Position/Status/Data Output Message (@@Ba-Response message) is shown in Figure 15 (see also [Mot94], p. 5.16 and 5.20).

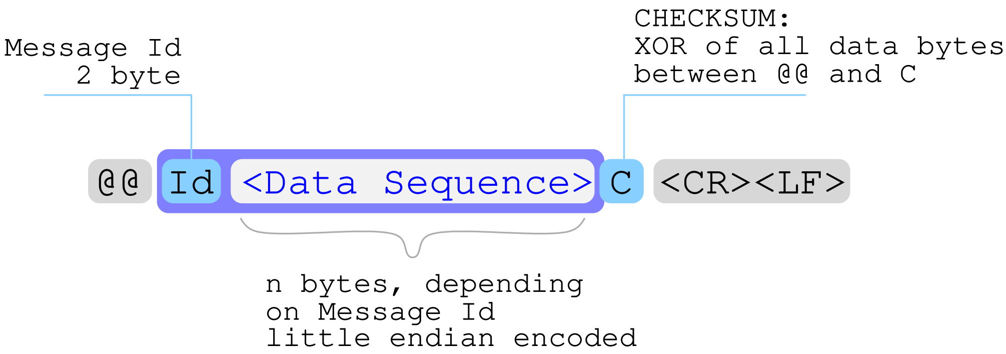


Figure 14: Motorola binary format

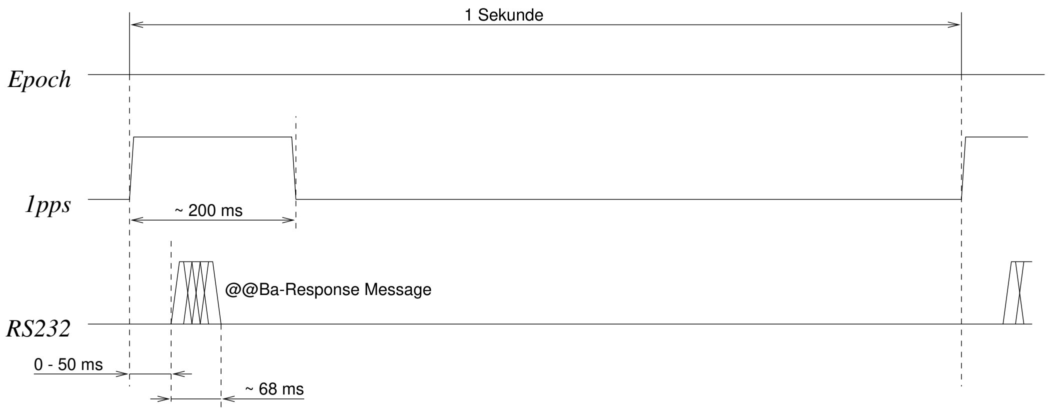


Figure 15: Protocol timing Motorola binary protocol

Valid status of a @@Ba-response message is determined by the RS232-Listener by checking the 6 receiver channel status flags and the receiver status message byte. The bit values of these bytes are defined in *gpsdrv_mot.h*.

```
#define MOT_CHANNEL_ERROR 0x11          /* Channel Status Flag in @@Ba msg */
                                        /* (bytes 43, 47, 51, 55, 59, 63) in
                                         * case of error (see p.6.64 ff):
                                         *
                                         *   - Sat reported unhealthy
                                         *   - Parity Error
                                         *
                                         * don't set "Sat reported inaccurate"
                                         * here - all sats are reported
                                         * inaccurate (probably because of
                                         * selective availability)
                                         */
#define MOT_REC_STATUS_ERROR 0x43       /* Receiver Status Message byte in */
                                        /* @@Ba msg (byte 64) in case of error:
                                         *
                                         *   - Poor geometry
                                         *   - Insufficient visible sats
                                         *   - Bad Almanac
                                         */
```

7.3 RS232-Listener / Command Parser

The RS232-Listener uses the state machine shown in Figure 16 to parse received messages from the Motorola VP Oncore. The parser works with a look ahead until the end of the packet (checksum-<CR><LF> sequence). Data is buffered in a ringbuffer; the parser works directly on the ringbuffer.

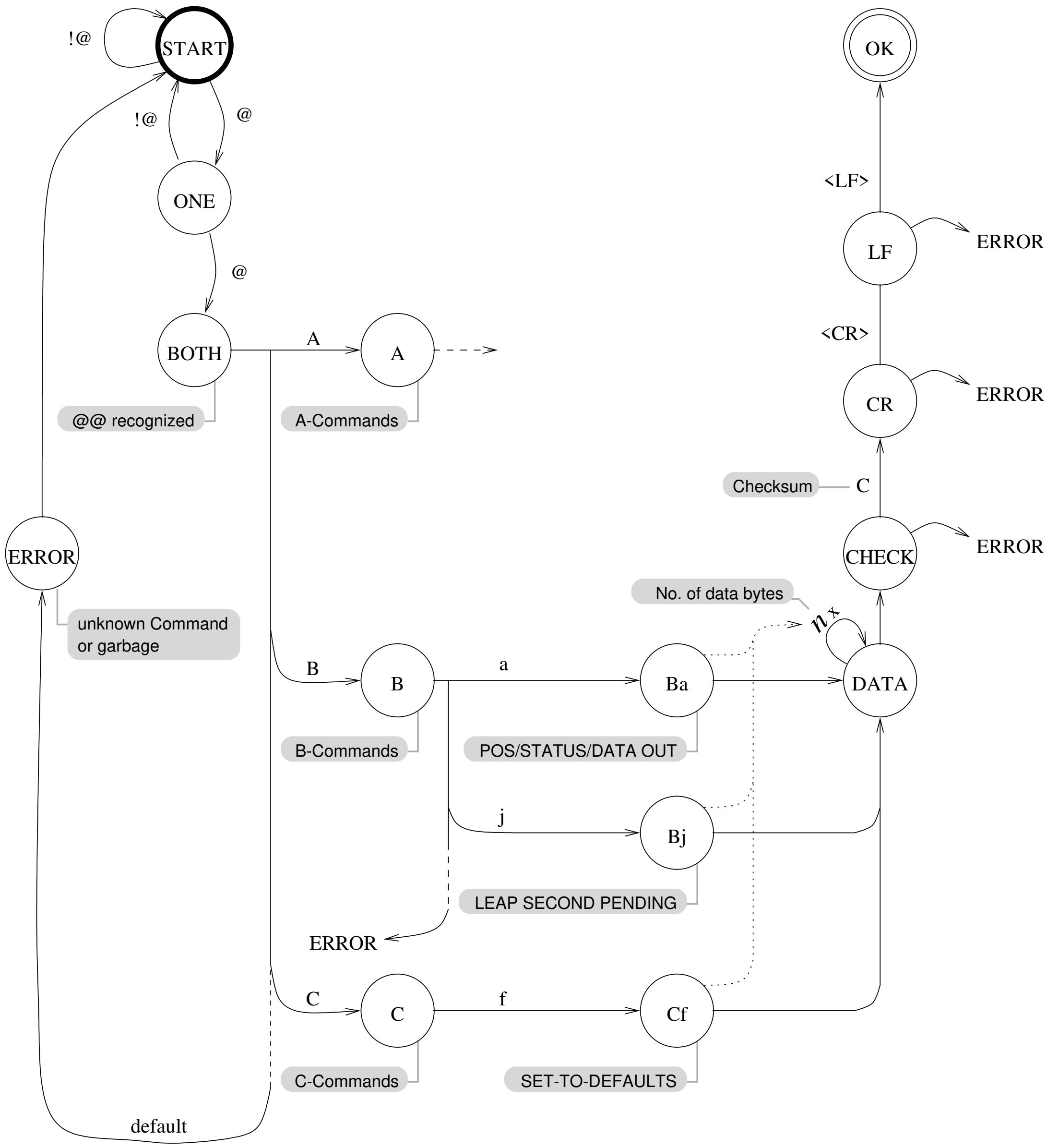


Figure 16: State machine of Motorola command parser

8 Open Issues and Further Work

8.1 UTC-Mode

This version of the driver does not support UTC mode because it does not recognize and anticipate leap seconds. If *gps_init()* is called with an *opmode* equal to UTC *gps_init()* returns an errorcode of GPSERR_NIMPL.

Unfortunately the command protocol of the Motorola VP Oncore does not simply provide leap second status in it's @@Ba-Response message ("this was a leap second") but rather requires one to request information about pending leap seconds explicitly. According to [Höc96] the GPS system operators will make leap second information available to users "some weeks" before the leap second occurs (which can happen on December 31st, June 30th, March 31st or September 30th).

UTC mode could be implemented by querying the receiver some time (hours? days?) before a possible occurrence of a leap second and then using the receiver's reply to anticipate the leap second.

8.2 Serial Interface Driver Problems

When the driver initializes the serial interface while the receiver sends RS232 messages to the target the first message received is garbled by the serial driver. The reason for this problem is unknown.

8.3 Implementing Support for other Receiver Models

Support for other receiver models can easyly be implemented by (re-)writing the receiver-dependent part (see Section 6.4) of the driver containing the RS232-Listener and receiver specific functions, expanding *receivers[]* (in file *gpsdrv_api.c*) accordingly and linking the new receiver-dependent part with the rest of the driver.

9 RS-232 Listener / Command Parser

The $ STATUS message has the following format:

$STATUS, week, tow, time_ref, data_valid_status, num_track_svs, have_GPS_database, fault_status xx CR LF

Example:

```
$STATUS,726,49769588,GPS,DATA INVALID,TRACKING 4 SVs, GPS DB COMPLETE,
FAULT 0000*4F<CR><LF>
```

The implementation for the parser uses a ringbuffer. Instead of operating dirctly on the ringbuffer, the ringbuffer is implemented in a separate file and organized by this structure:

```
typedef struct
{
  INT32 siodev;
  char rbuffer[RINGBUFFER_LENGTH];
  char* pWCursor;
  char* pRCursor;
  char* pStart;
  char* pEnd;
  rbuffer_struct;
}
```

With this structure and the following functions the ringbuffer can be accessed comfortly:

- `void init_rs232(rbuffer_struct* pbuf, INT32 siodev0);`
  This function resets the ringbuffer to it's initial state.
- `char read_char(rbuffer_struct* pbuf);`
  This function reads a character from the ringbuffer and, if the ringbuffer is empty, calls *read_rs232* to fill it.
- `int read_rs232(rbuffer_struct* pbuf);`
  Fills the buffer with new data from the rs232 port. Normally it is not neccessary to call this function, if *read_char* is used this is done automatically.
- `void unread_char(rbuffer_struct* pbuf);`
  Allows to make the last *read_char* undone. Should only called once after a read.

The incoming string is parsed for the correct label, the week, the time of the week, if the receiver is in GPS mode, if the data is valid and if the checksum is okay. Satellite tracking and database status are not checked.

If one incorrect character is detected the parser continues it's work with searching for the starting label.

10 Implementation Receiver-Dependent Part Stellar Absolute Time 100 A/B

10.1 Receiver settings

Receiver settings must be entered into the global array *receivers[]* of type *rec_settings* (in *gpsdrv_api.c*), which is indexed by the receiver type.

```
typedef struct {

    INT32 sioflags_i;   /* DITI Input: HW/SW Flow Control, EOL Chars,...*/
    INT32 sioflags_o;   /* DITI Output: EOL Definition, ... */
    INT32 sioflags_c;   /* DITI Control: Baud rate/Start/Stopbits, Parity,...*/
    INT32 sioflags_l;   /* DITI Line Discipline */
    int sio_timeout;    /* number of ticks timeout [VTIME] for diti driver */
    int rs232timeout;   /* timeout for RS232 event [PSOS ticks] */

    char  lowtohigh;    /* TRUE:  rising edge active 1~pps,
                         * FALSE: falling edge active 1~pps
                         */
    char  statusok; /* Status-Signal (0/1) for OK (= no GPS error)
                         * -1... ignore
                         */
    /*---------- Functions -----------*/
                                      /* RS232 Listener Task */
    void (* rs232listener)(INT32 arg0, INT32 arg1, INT32 arg2, INT32 arg3);
    int (* init)(INT32 id);           /* initialize receiver */
    int (* reset)(INT32 id);          /* reset receiver */
    int (* start)(INT32 id);          /* start receiver output */
    int (* stop)(INT32 id);           /* stop receiver output */
    int (* set)(INT32 old_id,         /* set receiver settings */
              const gpscfg * new,     /* old_id = NULL -> force set */
              int all);               /* set everything ? */
    int (* query_leap_sec)(INT32 id); /* query pending leap second */
)
} rec_settings;
```

All implementations of receiver dependent parts must provide the following functions which must be registered in *receivers[]*:

- **int rs232listener(INT32 arg0, INT32 arg1 INT32 arg2, INT32 arg3)**
  Function containing the RS232-Listener task. This task must listen on the RS232 port specified in *gpscfg_intern* and fill out the *RS232dat* structure. A handle to *gpscfg_intern* is passed to the task in *arg0*. In case of the Stellar Absolute Time 100 A/B the task listens for the $ STATUS message.

- **int init(INT32 id)**
  Initializes the GPS-Receiver to the required protocol mode. In case of the Stellar Absolute Time 100 A/B, the communication parameters are set up by the RS232 command.

- **int reset(INT32 id)**
  Sends a reset command to the receiver. For the Stellar Absolute Time 100 A/B the CLEARNVM command is used.

- **int start(INT32 id)**
  Sets the receiver to continuosly output its time information message every second. In case of the Stellar Absolute Time 100 A/B the LOG STATUS command is used.

- **int stop(INT32 id)**

  Tells the receiver to stop continuos output of the time information In case of the Stellar Absolute Time 100 A/B the UNLOG STATUS command is used.

- **int set(INT32 old_id, const gpscfg * new, int all)**

  Sends all messages necessary to set the receiver after a reset. Also, this function is used to change the settings for cabledelay and fixed position mode (latitude, longitude and height) when re-initializing the GPS-Driver. The cabeldelay has to be in nanoseconds,the latitude and logitude in milliseconds and the height in meters.

  In this implementation, if neither *old_id* is NULL nor *all* is TRUE, this function will only send the following command to the Stellar Absolute Time 100 A/B if one of the values in *new* are different from their values in *old*. The latitude, longitude and height are set by the MODE TTWL command if not in fixed position mode and by the MODE TT command in the other case. The cabledelay is set by the BIASCAL command (p. 12).

- **int query_leap_sec(INT32 id)**

  For receivers that don't supply leap second information in their time information message this function must send an appropriate request to the receiver. Since this version of the GPS-Driver doesn't support UTC time mode this function is empty.

10.2 Protocol Details

All command are set and received as ASCII characters. Every command starts with it's command identifier, followed by the command data and is terminated with a carriage return. The commands are not case sensitive. The output of the Stellar Absolute Time 100 A/B receiver is formated as follows:

- Begin with a $ (dollar) character.
- Next is the name if the log data item, (here: STATUS).
- Next is the comma
- Next are the data fields (if applicable) separated by commas;
- Next is an (asterisk) character followed by two hexadecimal characters for the message checksum. The checksum is the XOR of all preceeding characters except for the $ character and the character.
- End with a CR/LF character pair.

11 Implementation Receiver-Dependent Part Trimble SVeeSix CM2

11.1 Receiver settings

Receiver settings must be entered into the global array *receivers[]* of type *rec_settings* (in *gpsdrv_api.c*), which is indexed by the receiver type.

```
typedef struct {

    INT32 sioflags_i;   /* DITI Input: HW/SW Flow Control, EOL Chars,...*/
    INT32 sioflags_o;   /* DITI Output: EOL Definition, ... */
    INT32 sioflags_c;   /* DITI Control: Baud rate/Start/Stopbits, Parity,...*/
    INT32 sioflags_l;   /* DITI Line Discipline */
    int sio_timeout;    /* number of ticks timeout [VTIME] for diti driver */
    int rs232timeout;   /* timeout for RS232 event [PSOS ticks] */

    char  lowtohigh;    /* TRUE:  rising edge active 1~pps,
                         * FALSE: falling edge active 1~pps
                         */
    char  statusok; /* Status-Signal (0/1) for OK (= no GPS error)
                         * -1... ignore
                         */
    /*---------- Functions -----------*/
                                       /* RS232 Listener Task */
    void (* rs232listener)(INT32 arg0, INT32 arg1, INT32 arg2, INT32 arg3);
    int (* init)(INT32 id);            /* initialize receiver */
    int (* reset)(INT32 id);           /* reset receiver */
    int (* start)(INT32 id);           /* start receiver output */
    int (* stop)(INT32 id);            /* stop receiver output */
    int (* set)(INT32 old_id,          /* set receiver settings */
              const gpscfg * new,      /* old_id = NULL -> force set */
              int all);                /* set everything ? */
    int (* query_leap_sec)(INT32 id);  /* query pending leap second */
)
} rec_settings;
```

All implementations of receiver dependent parts must provide the following functions which must be registered in *receivers[]*:

- **int rs232listener(INT32 arg0, INT32 arg1 INT32 arg2, INT32 arg3)**

  Function containing the RS232-Listener task. This task must listen on the RS232 port specified in *gpscfg_intern* and fill out the *RS232dat* structure. A handle to *gpscfg_intern* is passed to the task in *arg0*. In case of the Stellar Absolute Time 100 A/B the task listens for the $ STATUS message.

- **int init(INT32 id)**

  Initializes the GPS-Receiver to the required protocol mode. In case of the Stellar Absolute Time 100 A/B, the communication parameters are set up by the RS232 command.

- **int reset(INT32 id)**

  Sends a reset command to the receiver. For the Stellar Absolute Time 100 A/B the CLEARNVM command is used.

- **int start(INT32 id)**

  Sets the receiver to continuosly output its time information message every second. In case of the Stellar Absolute Time 100 A/B the LOG STATUS command is used.

- **int stop(INT32 id)**

  Tells the receiver to stop continuos output of the time information In case of the Stellar Absolute Time 100 A/B the UNLOG STATUS command is used.

- **int set(INT32 old_id, const gpscfg * new, int all)**

  Sends all messages necessary to set the receiver after a reset. Also, this function is used to change the settings for cabledelay and fixed position mode (latitude, longitude and height) when re-initializing the GPS-Driver. The cabeldelay has to be in nanoseconds,the latitude and logitude in milliseconds and the height in meters.

  In this implementation, if neither *old_id* is NULL nor *all* is TRUE, this function will only send the following command to the Stellar Absolute Time 100 A/B if one of the values in *new* are different from their values in *old*. The latitude, longitude and height are set by the MODE TTWL command if not in fixed position mode and by the MODE TT command in the other case. The cabledelay is set by the BIASCAL command (p. 12).

- **int query_leap_sec(INT32 id)**

  For receivers that don't supply leap second information in their time information message this function must send an appropriate request to the receiver. Since this version of the GPS-Driver doesn't support UTC time mode this function is empty.

11.2 Protocol Details

All command are set and received as ASCII characters. Every command starts with it's command identifier, followed by the command data and is terminated with a carriage return. The commands are not case sensitive. The output of the Stellar Absolute Time 100 A/B receiver is formated as follows:

- Begin with a $ (dollar) character.
- Next is the name if the log data item, (here: STATUS).
- Next is the comma
- Next are the data fields (if applicable) separated by commas;
- Next is an (asterisk) character followed by two hexadecimal characters for the message checksum. The checksum is the XOR of all preceeding characters except for the $ character and the character.
- End with a CR/LF character pair.

References

[Dan97] Peter H. Dana. Global Positioning System (GPS) time dissemination for real-time applications. *Real-Time Systems*, 12(1):9–40, January 1997.

[Höc96] Dieter Höchtl. *Verfügbarkeitsuntersuchung von GPS-Satellitenempfängern.* Diplomarbeit, Technische Universität Wien, Department of Automation, September 1996.

[HS97] Dieter Höchtl and Ulrich Schmid. Long-term evaluation of GPS timing receiver failures. In *Proceedings of the 29th IEEE Precise Time and Time Interval Systems and Application Meeting (PTTI'97)*, pages 165–180, Long Beach, California, December 1997.

[Man99] Thomas Mandl. *Network Time Interface Benutzerhandbuch.* Diplomarbeit, Technische Universität Wien, Department of Automation, February 1999. (in German).

[Mot94] Motorola, Inc. *Oncore User's Guide*, June 1994. Revision 6.1, For Use with ONCORE Receiver Board Firmware Release 4.X, 5.X and 6.X.

[Mot98] Motorola, Inc. Oncore technical application note - 1999 and 2000 rollovers, February 1998. Revision 2.4, PDF-document can be downloaded from `http://www.mot.com/ACCES/GPS/products/index.html`.

[SM99] Ulrich Schmid and Thomas Mandl. Implementation of the NTI Device-Handler. Technical Report 183/1-86, Department of Automation, TU Vienna, January 1999.

A Listings

Attatched are the following listings:

File	Number of pages
gpsdrv_api.h	3
gpsdrv_api.c	1
gpsdrv_debug.h	1
gpsdrv.h	2
gpsdrv.c	8
gpsdrv_mot.h	2
gpsdrv_mot.c	9
gpsdrv_mag.h	0
gpsdrv_mag.c	0
gpsdrv_stel.h	0
gpsdrv_stel.c	0
gpsdrv_svee.h	0
gpsdrv_svee.c	0
testappl.c	4
root.c	5
testdaten_mot.h	4
testgps_mot.h	1
testgps_mot.c	3
testdaten_mag.h	0
testgps_mag.h	0
testgps_mag.c	0
testdaten_stel.h	0
testgps_stel.h	0
testgps_stel.c	0
testdaten_svee.h	0
testgps_svee.h	1
testgps_svee.c	0
capture.txt	2