Meinberg Funkuhren

Meinberg Driver and API Concepts

Table of Contents

Meinberg Driver and API Concepts

Introduction

Meinberg is manufacturing a broad range of PCI cards and USB devices that can be used to provide computers with the accurate time.

There have been different types of **PCI cards** available for different PCI bus versions, e.g. PCI Express, the earlier 3.3V PCI/PCI-X interface, or the even older 5V legacy PCI, and different PCI interface chips have been used for these PCI bus variants. Similarly, different USB devices have been implemented according to different USB standards, e.g. v2.0 or v1.1).

Also, different types of time signal can be decoded by different device types, e.g.

- GNSS satellite signals (GPS, Glonass, Galileo)
- long wave signals (DCF-77, MSF, WWVB)
- IRIG and similar timecodes
- Precision Time Protocol (PTP/IEEE1588)

Thus the features and the level of accuracy provided by a specific card depends strongly on the type of input signal as well as on the the bus interface type and chipset. E.g. the time accuracy available to an application is usually much better with a PCI card than with an USB device, even though the accuracy inside both types of devices may be in the same range. This is simply because a PCI card can be accessed very much faster than a USB device.

Care has been taken that the software interface for a given feature is compatible across all devices which support this feature.

Meinberg's Policy for Driver Software Packages

Meinberg's policy for driver software packages is to

- provide only a **single driver package per supported operating system**, i.e., there's only one driver package for all Windows versions, one driver package for all Linux versions, etc.
- let the driver software be able to handle **all Meinberg bus-level clock models** that have been introduced before a driver was released, including all PCI cards, USB devices, and even legacy ISA cards, if possible.
- make sure that **API functions are source code compatible** across all supported operating systems, even for **32 bit and 64 bit versions** of an operating system.
- take care to **keep compatibility of existing API functions** across driver versions and device types, and introduce new API functions to support new features provided by new devices.

So new versions of the driver software packages may be required to support new types of clock devices, support new features provided by new devices, or meet the requirements of new versions of the supported operating systems. But in any case care will be taken not to break API compatibility for existing applications. Thus existing applications will still work with newer devices providing the same features, if only the driver software package is updated to support the new device.

All API calls provided by the driver packages are **thread-safe**, and thus can **safely be used in a multitasking environment**.

Driver Usage

Basically the driver package supports using bus-level Meinberg devices in different ways:

1. Under Unix-like systems the card can be used as a reference time source (refclock) for the NTP daemon (`ntpd` or `chronyd`) shipped with the OS to synchronize the system time with high accuracy.

2. On Windows there is a time service available which synchronizes the system time to the PCI or USB device.

3. In any case applications can access the device directly using some API functions. This is often used to read times directly, but can also be used for measurements with other hardware. For example, some Meinberg devices provide time capture inputs which can latch the internal time whenever a hardware slope is asserted to an input pin of the connector in the slot cover. The latched time stamps are stored in an on-board FIFO buffer and can be retrieved using an API call.

Basically the on-board microcontroller decodes the incoming timing signal (GPS/GLONASS/Galileo, IRIG, or whatever) autonomously, regardless whether some driver software is installed and loaded, or not.

The driver software is only used to let applications access the device, e.g. read current time and status, or read or write some configuration data.

Providing the Reference Time to NTP

Earlier versions of the driver package for Linux used an IRQ generated by the device to provide the NTP daemon with the reference time from the card. Newer versions of the driver code use a different approach which provides better accuracy, and is easier to port to other operating systems. How accurately `ntpd` or `chronyd` can discipline the system time depends strongly on how accurately it can determine the difference between the system time and the reference time at a given point in time.

The Old Approach

The original approach was to use the parse driver provided by `ntpd` which has originally been developed to read a time string via a serial port. Unlike the GPS receivers from most other manufacturers the external GPS receivers manufactured by Meinberg can send a time string once per second very close to the second changeover. Of course, if supported by the OS, the 1 PPS signal can additionally be used to account for latencies in the serial driver, etc. In any case, the resulting reference time stamp is always for a second changeover, i.e., the fractions of the reference time stamp are always 0.

The approach for Meinberg PCI card drivers was to let the card generate an IRQ whenever the second changeover occurs, and emulate a serial port for `ntpd` where the parse driver can wait for new data similar to the arrival of the serial time string.

For use with a serial device this is OK since the driver has to wait until device sends a string, and the application knows the string has been sent at the second changeover.

An application can not send a request to a serial device at an arbitrary point in time to read the current time of the serial device. The latencies introduced by sending the serial request, processing the request in the device, having the device send the reply, and receiving the reply would degrade the resulting accuracy so that this became useless for accurate timekeeping (BTW, this is exactly the problem `ntpd` tries to avoid when evaluating four time stamps from a polling event to an upstream NTP server).

If you have a bus-level device then an application can read the current time from the card at arbitrary points in time. On the other hand, the approach using IRQs at second changeovers has several disadvantages:

- If several drivers for several PCI cards share the same IRQ then the kernel has to call the IRQ handlers registered by each driver one after the other to let them check if their device has generated the IRQ. If other drivers have been loaded and thus have registered their IRQ handler before our driver, then there's a significant latency introduced until the time from our card is actually read.
- The effect is even worse since the parse driver works in a way that the corresponding system time stamp is taken by the parse driver, i.e. after our driver has read the time from the card and woken up the blocking read call from the parse driver.
- If there are several cards which generate an IRQ exactly at each second changeover then there are always several IRQs and accesses to the cards at nearly the same time.

The New Approach

A new approach has been implemented with a new API call to the kernel driver where the kernel driver reads both the reference time from the hardware device with high resolution and accuracy and the corresponding system time as close as possible after each other. Both time stamps are then returned to the caller which can do some plausibility checks and then feed the time stamp pair to

ntpd's shared memory driver.

The advantage of this approach is that there are no interrupt latencies anymore. If the target platform supports fast high resolution timers like the TSC on x86 then the kernel driver can take such counter values before and after reading the time stamps, so the calling application can determine how long it usually takes to read the time stamps. If the execution time is significantly longer than “usual” then the call may have been interrupted or preempted, and thus the application can discard the readings and simply retry once more, since the API function can be called at arbitrary points in time.

The only drawback of this approach is that an additional simple daemon is required which calls the driver's API function periodically and feeds the results to a shared memory segment where it can be picked up by `ntpd` or `chronyd`. This is basically what the `gpsd` program does for external GPS receivers that send an NMEA time string, or use a proprietary binary protocol. The results seen by this approach are very much better than the results using the interrupt-driven parse driver, and, as mentioned above, no support for IRQs, no blocking reads, and no wait queues in the kernel driver are required.

Using Meinberg API Calls for PCI And USB Devices

API functions to access the device to read the current time, or read/write configuration information are exported by a module called **`mbgdevio` (Meinberg Device I/O)** which is provided by the driver package for the operating system. The driver package for Windows provides this module as a DLL (`mbgdevio.dll`), and current versions of the Linux driver package provide this module as shared object library (`libmbgdevio.so`).

The prototypes for the API functions exported by the `mbgdevio` module can be found in the header file `mbgdevio.h`, and the associated data structures are defined in the additional header files which are included automatically, so the API functions can be used natively by applications written in C or C++, or can be called from other programming environments using the proper wrappers or bindings.

Development for Linux

For Linux all the files required to build an own application are already installed with the driver package which is available as source code at our download page:

- https://www.meinbergglobal.com/english/sw/#linux

The Linux driver package can be used both with 32 bit and 64 bit versions of the Linux operating system, and once the package has been installed there are some shared object libraries available that can be used with 3rd party projects.

To use the basic API calls it is sufficient to add the following lines to a project `Makefile` to let the compiler find the header files, and link your application against the `mbgdevio` library:

```
CPPFLAGS += -I/usr/local/include/mbglib
LDFLAGS += -lmbgdevio
```

This makes it very easy to let simple applications use some Meinberg API functions from an own application written in C or C++.

However, if some extended functionality needs to be provided it may be necessary to add some more source code or object modules to the project.

Development for Windows

The driver package for Windows provides kernel drivers and DLLs for 32 bit and 64 bit versions of the Windows operating system.

Please note the location of the installed DLLs depends on whether the installed Windows version is 32 bit or 64 bit.

On 32 bit Windows there are only 32 bit DLLs:

```
C:\windows\system32  contains 32 bit versions of the DLLs
```

However, despite of the misleading naming convention, the **`system32`** folder always contains the ***native*** versions of the DLLs, so on 64 bit Windows it contains the 64 bit versions of the DLLs, and the 32 bit versions of the DLLs can be found in a different folder:

```
C:\windows\system32  contains 64 bit versions of the DLLs
C:\Windows\SysWOW64  contains 32 bit versions of the DLLs
```

This sounds pretty confusing since 64 bit DLLs are in a folder with '32' in its name, and 32 bit DLLs are in a folder with '64' in its name. However, WoW64 stands for `Windows-On-Windows 64-bit` which is the name of the Windows subsystem which allows running 32 bit applications on a 64 bit platform.

The correct DLLs are found automatically when a program starts, so there is no need to copy the DLLs to a different location when writing an own application.

However, for development of own C/C++ applications some **header files** are required which provide **function prototypes** and **definitions of associated data structures**, and **import libraries** associated with the DLLs. An import library tells the build environment which DLL (and also which version of a DLL, 32 or 64 bit) has to be loaded to be able to use an API function, and the function prototype in the header file tells the compiler how the function has to be called, and what the function returns.

Please note that the same header files can be used, but different import libraries for 32 bit and 64 bit

Windows versions are available in separate subdirectories, so care must be taken that applications are linked against the import libraries in the appropriate subdirectory.

An SDK is available which provides the header files and import libraries:

- http://www.meinbergglobal.com/english/sw/sdk.htm#windows

 But **please note** the code in the SDK is currently pretty old and needs to be updated, but nevertheless it is still valid, so it can still be used.

On the other hand, current versions of the header files and import libraries are shipped with the mbgtools for Windows package, which is also provided as source code.

Both packages provide an `mbglib` folder which contains the necessary files. The proposed way to use these files with an own project is to copy the `mbglib` folder and its subfolders to the own project, and add the relevant include search paths and import libraries to the project.

Add these paths to the header include file search path:

```
mbglib\common
mbglib\win32
```

and add ***one*** of these paths to the library search path:

```
mbglib\lib\bc     # for 32 bit applications with Borland Compilers
mbglib\lib\msc    # for 32 bit applications with Microsoft Compilers (Visual
Studio)
mbglib\lib64\msc  # for 64 bit applications with Microsoft Compilers (Visual
Studio)
```

The files in the mbglib folder should be left untouched to make future maintenance of the application easier by simply replacing the mbglib folder and its files by a newer version.

Opening a Device

The Meinberg driver software and API functions support several PCI and/or USB devices installed at the same time. This can be several instances of the same type of device, or several different device types. Software going to access a device directly first needs to open a specific device to get a device handle which can then be used with subsequent calls to address the device to be accessed.

Under Windows, device names are based on a unique ID number, so there is an API call

mbg_find_devices() which looks for all supported devices and builds an internal list which can then be accessed by an index.

Under Linux and FreeBSD devices have “human readable” names which are already distinguished by a subsequent index number, e.g. `/dev/mbgclock0`, `/dev/mbgclock1`, etc., so there is basically no need to look for all supported devices first, but calling mbg_find_devices() anyway does not hurt and increases the portability of the code across operating systems:

```
int devices_found = mbg_find_devices();

if ( devices_found == 0 )
{
  fprintf( stderr, “No supported device found\n” );
  exit( 1 );
}
```

After the call above has been executed there are 2 different ways to open a specific device. The first way calling mbg_open_device() simply uses the device index to open a specific device from the list of available devices. If only a single device is available the index is always 0, so this is suitable for most applications which only deal with a single device:

```
MBG_DEV_HANDLE dh;
int dev_idx = 0;

dh = mbg_open_device( dev_idx );

if ( dh == MBG_INVALID_DEV_HANDLE )
{
  fprintf( stderr, “Failed to open device #%i: %s\n”,
           dev_idx, strerror( errno ) );
  exit( 1 );
}
```

If several different types of device are installed an alternative way to open a device of a specific type is to use the different API call mbg_open_device_by_name():

```
const char dev_name[] = “GPS180PEX”;
MBG_DEV_HANDLE dh;

dh = mbg_open_device_by_name( dev_name, MBG_MATCH_MODEL );

if ( dh == MBG_INVALID_DEV_HANDLE )
{
  fprintf( stderr, “Failed to open device %s: %s\n”,
           dev_name, mbg_strerror( rc ) );
  exit( 1 );
```

```
}
```

The device handle returned by either of these functions can be used with all subsequent API calls for the same device, until the device is closed.

Closing the Device

Even though all device handles are closed automatically when an application terminates, it is good practice to let an application call mbg_close_device() close the device when there will be no more accesses:

```
mbg_close_device( &dh );
```

This also sets the device handle to `MBG_INVALID_DEV_HANDLE`.

Getting Basic Device Information

Right after a device has been opened, it is good practice to call the mbg_get_device_info() function to get some basic information on the device, including the device type, firmware version, etc. Eventually, this retrieved information has to be passed to some other extended API calls, but for some basic applications this is just optional:

```
PCPS_DEV dev_info;
int rc = mbg_get_device_info( dh, &dev_info );

if ( mbg_rc_is_error( rc ) )
{
  fprintf( stderr, “Failed to get device info from device: %s\n”,
mbg_strerror( rc ) );
  exit( 1 );
}
```

Checking if a Specific Feature is Supported

Current Feature Check Functions

Some features are only supported optionally, by particular types of devices, or by particular new

firmware versions where this feature has been implemented.

There is a group of functions available which can be used to check if a specific feature is supported. These functions return `MBG_SUCCESS` if the requested feature is supported, `MBG_ERR_NOT_SUPP_BY_DEV` if the feature is not supported, or some other error code if the function call failed to retrieve the requested information.

The example code below calls the mbg_chk_dev_has_synth() function to check if a programmable synthesizer is provided by the particular device:

```
int rc = mbg_chk_dev_has_synth( dh );

if ( mbg_rc_is_error( rc ) )
{
  fprintf( stderr, “No synthesizer provided: %s\n
           mbg_strerror( rc ) );
  exit( 1 );
}

// If we get here then the device provides a synthesizer
```

The availability of other features can be checked in a similar way by calling the appropriate function.

Deprecated Feature Check Functions

The calls above replace a couple of other API calls that had been introduced earlier, where each function provides a return code that needs to be checked, and only if the return code is `MBG_SUCCESS`, a variable has been updated with the requested information:

```
int supported = 0;
int rc = mbg_dev_has_synth( dh, &supported );

if ( rc != MBG_SUCCESS || !supported )
{
  fprintf( stderr, “No synthesizer provided.\n );
  exit( 1 );
}

// If we get here then the device provides a synthesizer
```

So after each call of such a function there are 2 return values that need to be checked to see if the requested feature is supported, or not.

The new functions have been introduced to make things easier, and the old check functions have

been marked deprecated even though they are still supported.

Deprecated Feature Check Macros

There are also some macros `_pcps_has_synth()` that had been introduced to check if a specific feature is supported, or not.

However, the code generated by these macros depends mostly on the definition of the macros in the header files used when compiling an application, so the result may not be appropriate for new devices which have been introduce after the header files have been published.

On the other hand, the current feature check functions always make a call into the device driver. Since the device driver has to be updated anyway to support a new device, it also knows which features are supported by the device, and can return the correct information.

So usage of these macros is discouraged, and the the current feature check functions should be used instead.

Getting the Time from a Device

There are different groups of API calls available which can be used to read the current time from a device. Each group of API calls has specific advantages and disadvantages.

A Legacy Call Returning Calendar Date and Time

The first series of Meinberg bus-level devices only supported an API call named mbg_get_time() to read the card's current date, time and status. The returned data structure (PCPS_TIME) contains the card's local date and time in broken down format, i.e. calendar date plus hours, minutes, seconds, and tenths of seconds, according to the time zone setting on the card.

At the time this was introduced this was appropriate for operating systems like MS-DOS which don't distinguish between local time and UTC, and the resolution of the system time was only about 55 ms, i.e. worse than the 10 ms resolution supported by this API call.

The broken-down date and time provided by this call requires much computation if e.g. a time difference needs to be determined. This API call is still supported by all PCI and USB devices, but it isn't widely used anymore.

High Resolution Time

Modern operating systems run on UTC internally, and the system time is kept in a linear format, e.g. the number of seconds since an epoch, plus fractions of a second (POSIX `time_t`), or the number of 100 ns units since an epoch (Windows `FILETIME`).

So some other functions have been introduced which return a linear time, and provide high resolution time stamps, which can be used very much easier in computations, e.g. do time comparisons, add offsets, etc.

There are standard functions available that can be used to convert the linear number of seconds into a calendar date and time.

Reading High Resolution Time and Status

The mbg_get_hr_time...() group of functions basically read a PCPS_HR_TIME structure which contains a PCPS_TIME_STAMP field plus some additional status information, e.g. if the card is currently synchronized, or not, a local time offset associated with the returned UTC time stamp, according to the time zone configured on the device, etc.

Even though most types of devices support these calls, there are some older devices out there which may not support this, so if in doubt the mbg_chk_dev_has_hr_time() function should be used to make sure these calls are indeed supported by the particular device.

These API functions always **require interaction with the on-board microcontroller**. However, there is some hardware support for these calls to yield highest accuracy, i.e. the on-board time is latched on entry of the call. Since the on-board microcontroller has to copy the latched time stamp plus some additional data to the output registers which can be read via the PCI bus, the overall execution time required to read the data is some tens of microseconds, or even longer if another access is in progress, compared to just a couple of microseconds required to just read a fast memory mapped time stamp that is provided by the current PCI Express cards.

Also, **subsequent API calls can be blocked** by the on-board microcontroller. This is required to prevent the microcontroller from being overrun by API calls which would prevent the controller from decoding its input signal. The blocking interval can be a couple of microseconds for new types of card up to a few milliseconds for old PCI card models, depending on the type of card and the type of microcontroller on the card.

Since the time stamps are taken on entry but the copying to the output registers is delayed for the blocking interval, the returned time stamp is accurate for the entry of the API call but the call takes longer to execute until it returns.

This call is protected by a mutex semaphore used for any access requiring interaction with the on-board microcontroller, so an mbg_get_hr_time() call can block for a longer time than it normally needs to execute if a different API function reading or writing a large data structure is in progress from a different application or process.

Please note:

- The same blocking mechanism and mutex apply if configuration data is read from or written to a device.
- The exact access times also depend on the operating system and CPU speed.
- There are `mbg_get_…_cycles()` and `mbg_get_…_comp()` variants of the call which may have advantages over the standard call. Please see the chapter about latency compensation for details.

Reading Fast High Resolution Time Stamps Without Status

The mbg_get_fast_hr_timestamp...() functions are fast because they read a time stamp from a memory mapped register on the card. However, these calls return just a time stamp as PCPS_TIME_STAMP, but no additional status information. On the other hand, this API call **does not require interaction** with the on-board microcontroller, and thus does not block.

These calls are only supported by PCI Express cards. Older standard PCI cards and USB devices don't support this because their PCI interface circuits don't provide support for these functions. The mbg_chk_dev_has_fast_hr_timestamp() function can be used to determine if the particular device supports fast high resolution time stamps.

Memory mapped access has to be done in 2 subsequent read accesses to the board, so this is done inside the kernel driver which protects these accesses by a spinlock semaphore. This also makes sure the call is safe across different threads or applications. The spinlock is only held across the 2 read accesses, so a thread doesn't have to wait very long if another thread currently holds the spinlock.

Please note:

- The spinlock used by these functions doesn't interfere with the mutex used for functions requiring interaction with the on-board microcontroller.
- The exact access times also depend on the operating system and CPU speed. Also the type of chipset as well as the number and type of PCI bridges on the mainboard affect the total access time. See the chapter about access times for details.
- There are `mbg_get_…_cycles()` and `mbg_get_…_comp()` variants of the call which may have advantages over the standard call. Please see the chapter about latency compensation for details.

Compensating the API Call Latency

Both of the `mbg_get_hr_time…()` calls and the `mbg_get_fast_hr_timestamp…()` calls have variants called **`mbg_get_…_cycles()`**. These functions additionally pick up the PC's cycles counter

when they read the time stamp from the device, and return the card's time stamp plus the associated cycles value to the caller.

The calling process can read a cycles count `c1` *before* it calls one of the `.._cycles()` API functions. The kernel driver then reads another cycles count `c2` when the time stamp is read from the PCI card. The caller can then compute `(c2 - c1) / cf` to convert the difference of cycles counts to a time interval representing the latency, where `cf` is the cycles frequency. This also contains the time interval the API call had to wait for a semaphore as mentioned above.

The **`mbg_get_…_comp()`** calls do exactly what has been described above, i.e. read the cycles count, call the associated `.._cycles()` function, but in addition they compute the latency and subtract the latency from the time stamp read from the board. So the time stamps returned by the `.._comp()` calls reflect as good as possible the point in time when the API function has been called by the application.

In case of the `mbg_get_fast_hr_timestamp()` call it's worth thinking about using the standard call or the `.._comp()` variant.

Under Linux the cycles are read from the TSC counters which are provided by the individual CPU cores of the x86 and x86_64 architecture. This is very fast and works well if the counters in the individual cores are synchronized and always increment at the same clock speed.

This is usually the case with newest CPUs, but there are a number of older CPU types from Intel and AMD where the TSC counters may not be synchronized, or where the TSC clock is derived from the CPU clock and thus can be slowed down for power saving, e.g. if Intel's Speedstep or AMD's Cold'n'Quiet“ features are enabled in the PC BIOS.

Such limitations may lead to unrelated cycles counts returned by subsequent API calls, and thus the latency compensation may yield wrong results.

Using a different timer provided by the chipset on the mainboard is usually not a solution since accessing those timers is much slower than reading the TSC, and thus the execution time to read the timers is in the same range as the latency itself.

Access Times

In any case the access time to the device needs to be taken into account. Of course access via the PCI bus is much faster than via a USB connection, but there are still limitations by the PCI bus architecture.

Even in case of PCI there is a significant latency due to the tree architecture of the PCI busses on the mainboard. There is a variable number of PCI bridges on the mainboard between the CPU and a specific PCI slot into which a card has been installed, and the access time may vary depending on the type and number of PCI bridges involved in a transfer.

Even though the PCI Express bus has been designed for high data rates the access time for single read accesses is pretty bad. Typical access times to read a 64 bit time stamp may vary from 3 to 12

microseconds.

There are mainboards where the mean execution time required to read a time stamp even depends on the physical slot on the same mainboard into which the PCI card is installed, and varies from 5 to 8 microseconds, depending on the slot.

Since the PCI bus is formed as a tree, parts of the bus between two different slots and the CPU may be shared. This means if actually data is transferred from one slot to one CPU core when a different CPU core tries to read from a PCI card installed in a slot which shares the same bus then the 2nd access is delayed due to PCI bus arbitration.

This can't be controlled by a PCI card or its driver, and as a result single read accesses to a PCI card can be significantly slower than usual, even if the mean access time is in the low microsecond range.

The `mbgfasttstamp` utility program can be used to measure the access times, for example:

```
# mbgfasttstamp -n 15 -b /dev/mbgclock0

mbgfasttstamp v3.4.99 copyright Meinberg 2007-2013
GPS180PEX 029511026220 (FW 1.10, ASIC 8.05) at port 0xE000, irq 16
HR time 2013-11-18 11:49:28.0403599, latency: 0.2 us
HR time 2013-11-18 11:49:28.0403711 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0403744 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0403776 (+3.3 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0403808 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0403841 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0403872 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0403905 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0403936 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0404092 (+15.6 us), latency: 0.2 us
HR time 2013-11-18 11:49:28.0404124 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0404156 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0404188 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0404220 (+3.2 us), latency: 0.1 us
HR time 2013-11-18 11:49:28.0404253 (+3.2 us), latency: 0.1 us
```

The program is shipped with the Meinberg driver package for Linux, but a Windows version is also available by the mbgtools for Windows package.

Using the command line above, the tool reads 15 subsequent time stamps from the PCI card into a memory buffer in a fast loop, and after the loop has finished it prints the time stamps and differences. The time difference just reports the difference between the current time stamp and the previous time stamp, which is mainly the execution time required to read a single time stamp. The latency is the time from the beginning of the read call until the memory location is actually read in the kernel driver. The reason for the outlier in the example above can be due to a PCI bus arbitration, or due to hardware IRQ executed on the same core the test program. This shows another effect which may affect the resulting accuracy for an application.

Interpolating the Time from the PCI Card

If an application needs to retrieve accurate time stamps at a very high rate then PCI bus access can be the limiting factor which prevents the application from reading each time stamp directly from a PCI card.

An approach to workaround this limitation could be to read the time from the PCI card in relatively large intervals only, e.g. once per second, and then let the application extrapolate the time using values from some high resolution counter which can be accessed very fast.

In the Meinberg driver/API terminology this is referred to as *cycles* counter, and there are API calls which return a time stamp of the current time from the PCI card, plus an associated *cycles* count. An application can then start an own thread which periodically gets a time stamp from the card and an associated *cycles* counter value, and have other “worker” threads get the current *cycles* count only and convert it to real time using the last recent time stamp / *cycles* pair:

- read the current cycles count
- use the cycles count from the last recent time stamp cycles pair to determine the elapsed number of cycles after the pair has been taken
- add the resulting time offset to the last recent time stamp to extrapolate the current time

This is very fast since the main thread(s) don't need to access the PCI card directly, and just very few fast computations have to be made.

The extrapolation is also very accurate since if you don't use the clock frequency of the cycles counter but let the extra thread which updates the time stamp/cycles pair determine the cycles frequency based on the number of cycles which have elapsed between the time stamps returned by 2 subsequent updates.

Actually our Linux driver uses the CPU's TSC register for the cycles counter, which can be read very fast both from kernel space (inside the driver) and user space (from an application).

A possible drawback here is that you need to make sure you have modern CPU types installed on your mainboard. In the past there have been CPU types where the TSC counters on different CPU cores were not synchronized, in which case the extrapolation explained above could yield wrong results if different threads were executed on different CPU cores.

The Linux driver package contains an example program called `mbgxhrtime` which demonstrates how this can be implemented:

```
mbgxhrtime v3.4.99 copyright Meinberg 2008-2013
Initial process affinity mask: CPU0…CPU7
Process affinity mask set for CPU0 only
GPS180PEX 029511026220 (FW 1.10, ASIC 8.05) at port 0xE000, irq 16
Waiting until PC cycles frequency has been computed …
PC cycles freq: 3392.321462 MHz, default: 3392.262000 MHz
t: 2013-12-19 16:21:23.9403698 UTC+1:00h (0.281 us)
```

```
t: 2013-12-19 16:21:23.9404369 UTC+1:00h (0.109 us)
t: 2013-12-19 16:21:23.9404413 UTC+1:00h (0.054 us)
t: 2013-12-19 16:21:23.9404449 UTC+1:00h (0.052 us)
t: 2013-12-19 16:21:23.9404481 UTC+1:00h (0.054 us)
t: 2013-12-19 16:21:23.9404512 UTC+1:00h (0.052 us)
t: 2013-12-19 16:21:23.9404545 UTC+1:00h (0.054 us)
t: 2013-12-19 16:21:23.9404576 UTC+1:00h (0.054 us)
t: 2013-12-19 16:21:23.9404607 UTC+1:00h (0.053 us)
t: 2013-12-19 16:21:23.9404638 UTC+1:00h (0.054 us)
```

This example has been run on a Intel Core i7 CPU at 3.4 GHz, so the reported cycles frequency of 3392 MHz indicates that the TSC is used for cycles. As a consequence, times can be interpolated very fast. The example above also shows that the true cycles frequency computed by the application based on the accurate time stamps is differs from the default cycles frequency reported by the Linux kernel.

Under Windows the cycles counts are retrieved using the Windows **`QueryPerformanceCounter` (QPC)** function which can use one of the available timers in the CPU or on the mainboard. The `mbgxhrtime` program displays the cycles frequency at startup. On a Windows XP system the output could look like:

```
mbgxhrtime v3.4.99 copyright Meinberg 2008-2013
Initial process affinity mask: CPU0…CPU1
Process affinity mask set for CPU0 only
PZF180PEX 046411000030 (FW 1.07, ASIC 10.00) at port 0xC800, irq 17
Waiting until PC cycles frequency has been computed …
PC cycles freq: 3.570605 MHz, default: 3.579545 MHz
t: 2013-12-19 15:14:53.1607658 UTC (6.441 us)
t: 2013-12-19 15:14:53.1607868 UTC (6.441 us)
t: 2013-12-19 15:14:53.1608013 UTC (7.282 us)
t: 2013-12-19 15:14:53.1608170 UTC (6.441 us)
t: 2013-12-19 15:14:53.1608319 UTC (7.842 us)
t: 2013-12-19 15:14:53.1608484 UTC (6.441 us)
t: 2013-12-19 15:14:53.1608630 UTC (8.122 us)
t: 2013-12-19 15:14:53.1608795 UTC (6.441 us)
t: 2013-12-19 15:14:53.1608943 UTC (7.842 us)
```

The cycles frequency is about 3.58 MHz here, which indicates that the power management timer is used as cycles counter. The PM timer is located in the chipset and accessed via a peripheral bus, and thus it takes much longer to read the PM time count than to read the TSC count from a CPU register. As a consequence the differences between 2 interpolated time stamps on the Windows machine here are much larger than for the Linux system.

Depending on the Windows installation the QPC API could also be implemented using the HPET timer. In this case the reported frequency would be about 10 MHz.

In the `mbgxhrtime` example program the updater thread calls the `mbg_get_fast_hr_timestamp_cycles()` functions in periodic intervals, e.g. once per second, and

updates the time stamp/cycles pair. The update, i.e. copying the new pair over the old one, is done inside a critical section (i.e. protected by a semaphore) to make sure the data pair used by the worker threads is always consistent.

As an improvement for highest accuracy the updater thread could read the TSC or QPC before and after calling `mbg_get_fast_hr_timestamp_cycles()`, so it could check how long it has actually taken to execute this call. If this took longer than “usual” then access to the PCI card may have been delayed by a hardware IRQ, or a PCI bus arbitration, and the call could be repeated if this is the case. However, this is actually not implemented in the functions provided by the shared object libraries / DLLs.

Using the Time Capture Inputs

Common Hints

Some Meinberg devices, PCI cards as well as some standalone devices, provide time capture inputs which can be used to time stamp external hardware trigger slopes.

Whenever a trigger slope is detected at one of the inputs the device saves the current on-board time as a capture event in an on-board FIFO buffer which can hold up to about 600 events, depending on the specific device. The capture event structure also includes the channel number at which the slope was detected, usually 0 or 1.

Capture events can be retrieved from the FIFO in 2 different ways:

- If a device provides a serial port and supports the Meinberg Capture String type then the device can be configured to send an ASCII string automatically via the serial port as soon as a capture event has been detected.
- If a device is a PCI card then capture events can be retrieved via the PCI bus by using the appropriate API calls provided by the driver software package. This is much faster than sending and receiveing time capture events via a serial port, and doesn't require any external cabling.

Whenever a capture event is retrieved from the FIFO buffer ***it is removed from the buffer***, so if a capture event has already been sent as string via the serial port it can't be retrieved anymore via the PCI bus using the API calls.

Most PCI cards with time capture inputs also provide one or two serial ports, so if time capture events are to be retrieved via the PCI bus it should be made sure that **none of the card's serial ports** has been configured to send a **time capture string automatically**. One of the serial ports and the time capture inputs as well as some signal outputs are available via the 9 pin D-type connector in the card's slot cover.

In order to use the time capture inputs the associated DIP switches of the card have to be set to the “ON” position to wire the pin from the connector in the slot cover to the internal capturing circuitry. By default all switches are in the OFF position. See the card's PDF manual for details. Please take care to use 5V logic levels only. Higher voltages may damage the card.

If the capture events are to be retrieved by an application running on the machine where a PCI card is installed then it's most appropriate to read the capture events from the FIFO via the PCI bus. This is fast, and the data structures are easy to evaluate.

Most devices can only capture events on a falling trigger slope, except the PTP270PEX card. See the next chapter.

Specific Information for the PTP270PEX Card

Please note the PTP270PEX card provides no serial port, even though it has 9 pin D-type connector in its slot cover, so capture events can be retrieved from this card only via the PCI bus, using the API calls.

Most devices can only capture events on a falling trigger slope, and this is not configurable. However, the PTP270PEX card captures on the rising edge of an input signal by default, but for each of its 2 capture channels the capture slope can be configured.

Since this is very specific to the PTP270PEX card there is unfortunately no API call which can be used change this configuration. Instead, you have to open an SSH session to the Linux system running on the PTP270PEX card, login, and type:

```
nano /config/tsu_config
```

Locate the following lines at the bottom of the file:

```
UCAP0 inverted: 0
UCAP1 inverted: 0
```

If you want one or both inputs to capture on the falling edge set the appropriate value to '1'. Then press `Ctrl+X`, `Y`, <Enter> to exit the nano editor.

Then power cycle the PC and the card to let the card reboot and make the changes become effective. The changed settings are saved persistently on the card, so this change has to be done only once.

Handling Time Capture Events via the PCI Bus

Reading capture events via the PCI bus is very much faster than sending and receiveing time capture events via a serial port, and it doesn't require any external cabling. The Meinberg driver software supports the following API calls to deal with the time capture events via the PCI bus:

- `mbg_clr_ucap_buff()` Clear the FIFO on the card.
- `mbg_get_ucap_entries()` Return the current and maximum number of entries of the FIFO

buffer.
- `mbg_get_ucap_event()` Retrieve an event. The event is removed from the FIFO buffer. If the buffer has been empty then the retrieved time stamp is 0.00000.

A common way to read all available capture events as fast as possible is:

```
// Read out events from the FIFO and wait
// for new events if the FIFO is empty.
for (;;)
{
  PCPS_HR_TIME ucap_event;
  rc = mbg_get_ucap_event( dh, &ucap_event );
  if ( mbg_rc_is_error( rc ) )
    break; // an error has occurred

  // If a capture event has been read then it
  // has been removed from the card's FIFO buffer.
  // If the time stamp is not 0 then a new capture
  // event has been retrieved.
  if ( ucap_event.tstamp.sec || ucap_event.tstamp.frac )
  {
    show_ucap_event( &ucap_event );
    continue; // immediately read next event
  }
  usleep( USLEEP_INTV );  // sleep, then try again
}
```

Capture events are retrieved as `PCPS_HR_TIME` structure containing the time stamp (UTC), the channel number, and some status flags, e.g. `PCPS_UCAP_OVERRUN` if capture events occurred faster than they could be saved, or `PCPS_UCAP_BUFFER_FULL` if no space was left in the FIFO buffer. How to evaluate this structure is explained in one of the next chapters.

Getting Time Capture Events via a Serial Port

If time capture events are to be sent via a serial port rather than be read via the PCI bus then it should be made sure that only one of the physical ports provided by the device has been configured to send the capture string.

Usually the easiest way is to let a string be sent automatically whenever a trigger slope has occurred and thus an event has been generated. To achieve this the selected port has to be configured to output the “Meinberg Capture” string format, and the “Mode” for this port has to be set to “automatically”.

If the device provides more than one physical port then all other ports should be configured for a different string format. Otherwise capture events could also be sent via a different port and get lost

for the application waiting for a capture string sent via the selected port.

The baud rate and framing can be set according to the requirement of the application which waits to read the capture event strings. Please keep in mind that the baud rate affects the maximum capture rate for continuous operation. The lower the baud rate the more time it takes to send a string for an event, and if the trigger events occur at a higher rate then the FIFO buffer fills over time until it finally overflows.

Capture events can be read very much faster via the PCI bus by using the appropriate API calls, if a device supports this. So the card can handle a higher continuous rate of trigger events if those API calls are used.

No Meinberg-specific API calls have to be used to read a capture string via the serial port. The string is just an ASCII text string, so an application just has to open the serial port, read a line of text and parse the string to decode the date and time of the capture event.

A description of the capture string format can be found here:

- http://www.meinbergglobal.com/english/specs/capstr.htm

Converting Time Stamps to Calendar Date and Time

The **`PCPS_TIME_STAMP`** structure's `sec` field contains the seconds since 1970 similar to the POSIX `time_t` value which is returned by the standard C library function `time()`. This simplifies computations without having to care about ranges of seconds, minutes, hours, etc.

Accordingly, the `sec` field can be broken down to human readable calender date and time using the standard `gmtime()` function provided by the standard C library.

However, please keep in mind that the seconds field from the `PCPS_TIME_STAMP` structure is always only 32 bits wide, while the `time_t` type supported by some programming environments is 64 bit wide to avoid an overflow that happens with the 32 bit signed `time_t` type in January 2038.

```
PCPS_HR_TIME ucap;
time_t t;
struct tm tm;
mbg_get_ucap_event( dh, &ucap );
// first assign to a time_t type variable since time_t may be 32 or 64 bit,
// depending on the OS type and/or version of the runtime library which
// comes with the build environment
t = ucap.tstamp.sec;

// eventually do an epoch conversion here to support dates after January
2038.

if ( we_want_local_time )
```

```
  t += ucap.utc_offs;

// convert ot broken down date and time
tm = *gmtime( &t );
```

The `tm` structure also contains a field `tm_yday` which returns the day of year.

The function `localtime()` should not be used here since it accounts for the time zone settings and local time offset of the operating system at the time when the time stamp or capture event is evaluated, but not the local time offset at the moment when the trigger slope was recorded, or the time stamp was read from a device.

Instead, the UTC offset returned with the `PCPS_HR_TIME` structure can be used as shown in the example above.

The `frac` field of the `PCPS_TIME_STAMP` structure doesn't provide the number of microseconds or nanoseconds but instead contains the ***binary fractions*** of the time stamp, i.e.:

```
0x00000000 = 0.00000000 s
0x80000000 = 0.50000000 s
0xFFFFFFFF = 0.9999999999999999… s
```

The advantages of using binary fractions instead of a number of microseconds or nanoseconds are:

- Easy computations without having to check ranges (0 to 1000000 microseconds, or 0 to 1000000000 nanoseconds), i.e. seconds must be incremented when fractions just roll over from 0xFFFFFFFF to 0x00000000.
- Always same high resolution 2^-32 s =~ 0.23 ps for the least significant bit (resolution, not accuracy of a time stamp!)

The following function can be used to convert the binary fractions to a number of milliseconds or microseconds, according to the application's requirements:

```
uint32_t frac_sec_from_bin( uint32_t b, uint32_t scale )
{
  return (uint32_t) ( (PCPS_HRT_FRAC_CONVERSION_TYPE) b * scale
                      / PCPS_HRT_BIN_FRAC_SCALE );
}

// frac_sec_from_bin
long milliseconds = frac_sec_from_bin( ucap.tstamp.frac, 1000 );
long microseconds = frac_sec_from_bin( ucap.tstamp.frac, 1000000 );
```

Please note that for **printing the fractions** e.g. using `printf()` the **format string** for the fractions must specify the field width corresponding to the resolution used for the conversion as shown above, e.g. ”`%06li`“ for microseconds:

```
// print date/time in ISO format:
printf( "%04i-%02i-%02i %02i:%02i:%02i.%06li",
        tm.tm_year + 1900, tm.tm_mon + 1; tm.tm_mday,
        tm.tm_hour, tm.tm_min, tm→tm_sec,
        (long) microseconds );
```

Accordingly, `"%03li"` has to be used for milliseconds:

```
printf( ".%03li", (long) milliseconds );
```

Using the format

```
printf( ".%li", (long) milliseconds );
```

would print `".1"` instead of `".001"` if milliseconds == 1, which is obviously wrong since this means 0.1s instead of 0.001s.

The same applies to microseconds accordingly, if `".li"` was used instead of `".%06li"`.

Enabling of Output Signals

Some of the output signals provided by Meinberg devices can be configured to be enabled always immediately after the device has been powered up, or only after the device has synchronized to its reference time signal.

The associated data structure is called `ENABLE_FLAGS`, and the current settings are stored in non-volatile memory on the device. The header file `gpsdefs.h` defines the structure and the associated constants to be written to the structure's data fields.

For example to enable the synthesizer output (if provided) immediately after power-up:

```
ENABLE_FLAGS ef;
int rc;

// read current settings
rc = mbg_get_gps_enable_flags( dh, &ef );
// be sure rc == MBG_SUCCESS

ef.serial = EF_OFF;   // off until synchronized
ef.synth = EF_SYNTH;  // always enabled

// write new settings
rc = mbg_set_gps_enable_flags( dh, &ef );
// be sure rc == MBG_SUCCESS
```

Please note each of the fields can only be set to some specific values defined in file `gpsdefs.h`.

The Programmable Synthesizer

Some PCI cards provide a programmable frequency synthesizer the frequency of which can be determined or set via API calls. The following API calls are available, and `mbg_dev_has_synth()` should be called first to check if a synthesizer is provided by the particular device. See also chapter ...

```
mbg_dev_has_synth()    determine if a synthesizer is provided
mbg_get_synth()        get current settings
mbg_set_synth()        write new settings
mbg_get_synth_state()  get the state of the synthesizer
```

Setting the Synthesizer Output

Please note the synthesizer output can be configured to be enabled only after the device has synchronized to its time source, or always after power-up. See here for details.

The synthesizer frequency can be set with 4 digits to determine the frequency in 0.1 Hz units, plus a range value used as multiplier. The related data structures and definitions can be found in file `gpsdefs.h`, mainly:

```
// Synthesizer configuration parameters
typedef struct
{
  int16_t freq;    ///< four digits used; scale: 0.1 Hz; e.g. 1234 → 123.4
Hz
  int16_t range;   ///< scale factor for freq; 0..::MAX_SYNTH_RANGE
  int16_t phase;   ///< -::MAX_SYNTH_PHASE..+::MAX_SYNTH_PHASE; >0 → pulses
later

} SYNTH;
```

The API call `mbg_get_synth()` retrieves this structure with the current settings, and the API call `mbg_set_synth()` writes this structure with updated settings to the device.

The example code below sets up a frequency of 100 Hz with 180° phase shift:

```
SYNTH synth;
int rc;

synth.freq = 1000;    // 100.0 base value
```

```
synth.range = 0;      // multiplied by 1e0
synth.phase = 1800;   // 0.1° units → 180.0°

rc = mbg_set_synth( dh, &synth );
// On success, rc should be MBG_SUCCESS
```

Checking the Synthesizer State

The synthesizer state can be checked using the function `get_synth_state()` which fills up a `SYNTH_STATE` structure. The state field of this structure contains a number represented by one of the following symbols:

```
SYNTH_DISABLED   // disbled by cfg, i.e. freq == 0.0
SYNTH_OFF        // not enabled after power-up
SYNTH_FREE       // enabled, but not synchronized
SYNTH_DRIFTING   // has initially been sync'd, but now running free
SYNTH_SYNC       // fully synchronized
```

For detailed information please see the comments in file `gpsdefs.h`.

Calling API Functions from Other Kernel Modules

In newer versions of the Meinberg driver packages the kernel modules also export some functions which can safely be called from other kernel modules:

```
mbgclock_default_get_fast_hr_timestamp()
mbgclock_default_get_fast_hr_timestamp_cycles()
mbgclock_default_clr_ucap_buff()
mbgclock_default_get_ucap_entries()
mbgclock_default_get_ucap_event()
```

see the `mbgclock.h` file from the driver package or SDK for the full prototypes.

Why is the Driver not Shipped with Open Source Systems?

In the past there have been several discussions about the pro's and con's of the Meinberg driver concept, i.e., providing the driver as a standalone product vs. adding the driver code to the source tree of a specific OS, e.g. Linux or *BSD.

Here are some points:

* Adding a driver to the source code tree / code base of a specific OS usually also requires the formal coding style for the OS is being used, so you often can not commit the same source modules to different OS source trees.

- As a consequence of the above, the same set of modifications has to be applied to several similar source code modules for several operating systems. Using our library approach, we have to implement these modifications only once and then simply pull the updated files into the driver sources for all supported OSs.
- If the driver code comes as a part of the OS, you usually have to upgrade the OS to get a newer version of the driver which supports e.g. new device types or new features. With the Meinberg approach the OS-specific code is such that it can be used in the same way under several OS versions. For example, the same driver package for Windows can be installed on all Windows versions from Windows NT up to Windows 11 or Server 2022. The Linux driver package can be used with all 2.6.x and 3.x, 4.x, 5.x and 6.x kernels, and the kernel API changes between 2.6.0 and the latest kernel releases are covered in the OS-specific parts of the code.

The advantage of the above is that even customers who have stuck with an older version of the OS for some reasons can install a brand new PCI card model, and install a driver which works with the new card on an older OS.

The advantage of easier maintenance using a driver library also covers the API for user space applications. The API functions do not only provide ways to read the current time from a card. There are also API functions which query whether a specific feature is supported by a card, or which can and have to be used to configure the card according to the card's possibilities and requirement of the target system.

For example, if you install one of our IRIG receiver cards you have to configure the UTC offset of the incoming IRIG signal first, otherwise the card will never synchronize to any input signal. This is to prevent the system time from being unintentionally set to a wrong time or date immediately after installation, since most commonly used IRIG signal frame types don't provide information whether they carry UTC, or some local time. See IRIG Time Code Basics.

Unsupported Operating Systems

Meinberg driver implementations are using a set of library-like source modules to implement the basic functions of the driver, i.e. to detect and support all features provided by a particular card.

These pieces of code are mostly source code compatible across all supported operating systems, which actually includes Windows, Linux, FreeBSD, NetBSD as well as older OS types like MS-DOS, Novell NetWare, etc. Supported hardware platforms are actually x86/x86_64, Sparc64, Itanium IA64, and ARM.

The driver code library supports plug and play operating systems like Linux and Windows where the OS kernel scans the PCI bus, then loads the associated driver and calls the driver's probe routine for

each particular PCI device it has found, as well as old systems like Windows NT where the driver is loaded on demand and then scans the PCI bus by itself to see if any device is found which is supported by the driver.

Similarly, the library supports target systems where hardware access can only be done in kernel space while normal applications usually run in user space, calling kernel functions via IOCTL. On the other hand it also supports targets where an application can access the hardware directly, for example in plain old MS-DOS.

Which way is used for a particular target system is simply controlled by some #define statements. To support a specific target OS with kernel drivers a skeleton driver for the target OS is required which provides the glue functions to the kernel, e.g., implements a probe routine for a PCI device which then calls our library probe routine to check the device, implements an IOCTL routine which then calls the ioctl handler which is defined as inline function in our library, etc.

Also it must be known how to define proper IOCTL codes from a set of enumerated numbers, how spinlock and mutex functions are implemented in kernel space, etc.

For highest accuracy and latency compensation it would be good to know if there is a high resolution timer API available for the target platform and/or operating system, similar to the TSC on Intel x86/x86_64, which could also be called from a user space application, similar to the RDTSC inline asm instruction on x86/x86_64.

Endianess conversion is already handled in the common library code, so the existing driver library code can be used on little endian systems, e.g. x86/x86_64, as well as on big endian systems, e.g. Sparc64.

— Martin Burnicki martin.burnicki@meinberg.de, last updated 2023-11-17