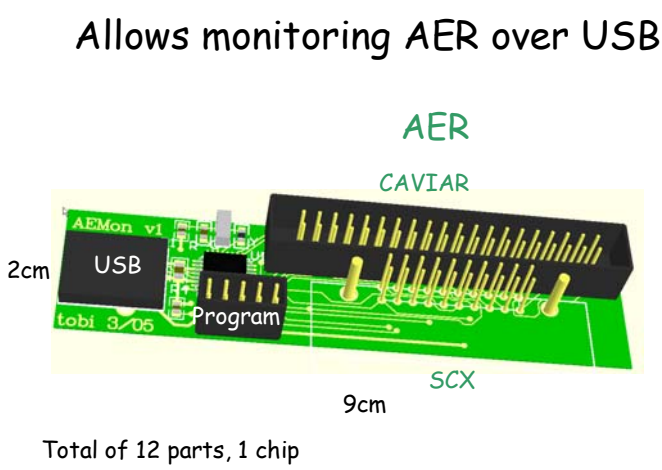
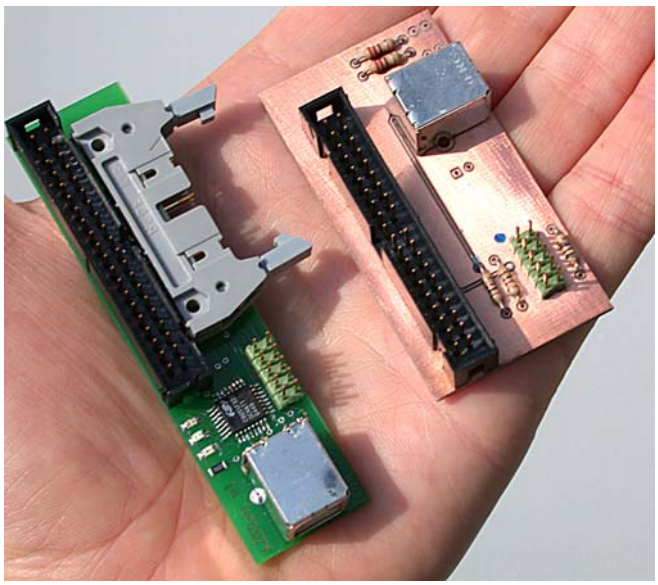
SimpleMonitorUSBXPress User Guide

Tobi Delbruck, tobi@ini.phys.ethz.ch



The SimpleMonitorUSBXPress is a single-chip bus-powered USB board that can capture address-event representation (AER) addresses and timestamps and send them to a Windows XP PC, where they can be directly acquired into matlab or into java programs.

Two versions of the PCB exist. The first is designed and built by Rafael Paz and Anton Civit at the University of Sevilla and the second by Tobi Delbruck at INI in Zurich. The firmware and host code is written by Delbruck. It is part of the EC 5th Framework project CAVIAR which is building an AER-based visual system.



The software and firmware of the board lives in the CAVIAR subversion repository (https://svn.ini.unizh.ch/repos/avlsi/CAVIAR) in the folder wp5\USBAER\SimpleMonitorUSBXPress

Features and Limitiations

Good and bad

- Good
  - Low cost (~20CHF). Easy to build.
  - Portable (bus powered)
  - Simple to debug
  - Windows, matlab or java
- Bad
  - Only monitors
  - Max rate ~100keps
  - Windows only

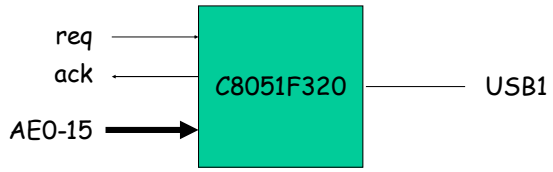
This device was developed for its simplicity, small size, low cost, and portability. The main applications are field capture of events and portable demonstrations. It is bus-powered and can therefore be taken out of the lab as long as the sender is also portable. It is also a Windows-based AER monitor.

It clearly has limitations compared with the Rome PCI-AER board and the Spanish PCI-AER and USB-AER boards. There are no capabilities besides event capture. For example you cannot sequence events out, you cannot remap events, you cannot capture from multiple senders, etc. Also it can only capture at a sustained rate of about 50-100k events/sec, and the timestamps will contain periodic jumps of about 900us while events are transferred over the USB bus.

Principle of operation

The device uses the Silicon Laboratories (http://www.silabs.com) C8051F320 USB 1 Microcontroller to capture AEs into internal RAM. A timer value is captured (on the request pin transition) along with the address for the timestamp value. When a full buffer is captured, it is sent to the host. (See Polling, advance transmission, and overruns on page 4 for more information.)

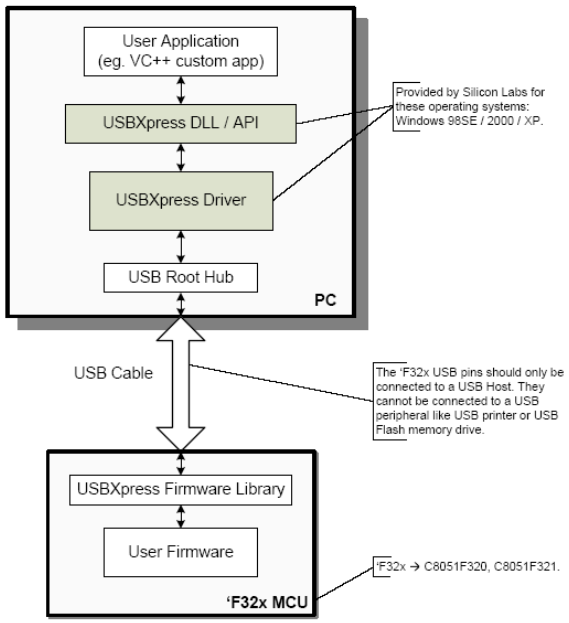
Firmware (simplified)




```
while(!req)
      if(>32ms since last xfer) sendEvents();
      storeEventAndHandshake();
      if(bufferFull) sendEvents();
```

The microcontroller uses Full speed (12Mbps) mode and USB Bulk Transfers to transfer the data. This is the same USB mode that USB disk drives use. It is designed for transfer of large amounts of data. There are two other USB1 modes—interrupt and isochrounous—but neither of these is more suitable. In reality none of the USB modes is ideal for transfer of large but unpredictable amounts of data. In USB Bulk mode, the host (the PC) requests data and the USB device sends it. Bulk mode was also chosen because it is the mode that is used by the Silicon Laboratories USBXPress framework, which supplies both firmware libraries and host driver and library software. USBXPress greatly simplifies USB development, but it does come with the disadvantages that it is not simple to use double buffering on the device side and consistent enumeration of multiple devices is also not simple.

data flow



Addresses and timestamps

16 bit addresses are returned along with 32 bit timestamps. It is up to the user to interpret the 16 address bits, e.g. as X and Y addresses of a 2D array.

The timestamp tick is 1 us. These timestamps are captured on the microcontroller using an internal 1 us clock and a 16 bit timer, allowing for 65 ms on the device. Timestamps are unwrapped in the host to 32 bit resolution providing for a maximum timestamp of about 4300 seconds. However, the largest *interspike* interval before an unwrapping loss is 65 ms. An interspike interval greater than 65 ms will be wrapped towards 0 seconds.

Because the device does not handshake during USB transfer, under heavy loading conditions you may observe periodic 860 us interspike intervals. The event after the large interval has been delayed by not being acknowledged from the sender.

Polling, advance transmission, and overruns

Under heavy load the device will transfer packets of 225 events. If you sit in a tight polling loop on the host, you will get either 0 or 225 events.

If you delay more on the host, you generally get packets that are multiples of 225 events.

If you wait too long, you will get 16k events. In this case the newest events have been lost, although you will still receive the events before the overrun occurred. You can detect an overrun by acquiring the events and then polling the overrun flag. Timestamp wrapping on the host is not reset after an overrun occurs, but you will have lost events and may have "lost time" on the device.

On the device side, there is a mechanism to transfer acquired events periodically, every 32ms, even if the transmission buffer is not full. Although this can reduces bus efficiency it doesn't matter because the event rate is low anyhow, and it delivers available events sooner and reduces host processing latency. Specifically, the device ensures that it does a USB transfer within 32ms of the last transfer if there are events to send. If the sender is sending lots of events then this advanced transfer never occurs because ordinary buffer-full transfers happen much more frequently in any case (e.g. every 2ms). But if the sender is generating a very low rate of events, then it is possible to receive even single events after a transfer.

Hardware and cabling

The hardware consists of a small PCB with only a single chip on it: the Silicon Laboratories (http://www.silabs.com) C8051F320 microcontroller. It is configured to receive word-parallel point to point or multisender AER, using active low request and acknowledge signals.

There is a USB device connector and two functionally equivalent AER connectors: a CAVIAR standard 40 pin 100mil double row header and a Rome 20 pin connector. The pin assignments on the CAVIAR connector are specified in the CAVIAR document "Consortiumstandards.pdf" in the repository folder CAVIAR/wp7 and are repeated here for reference:

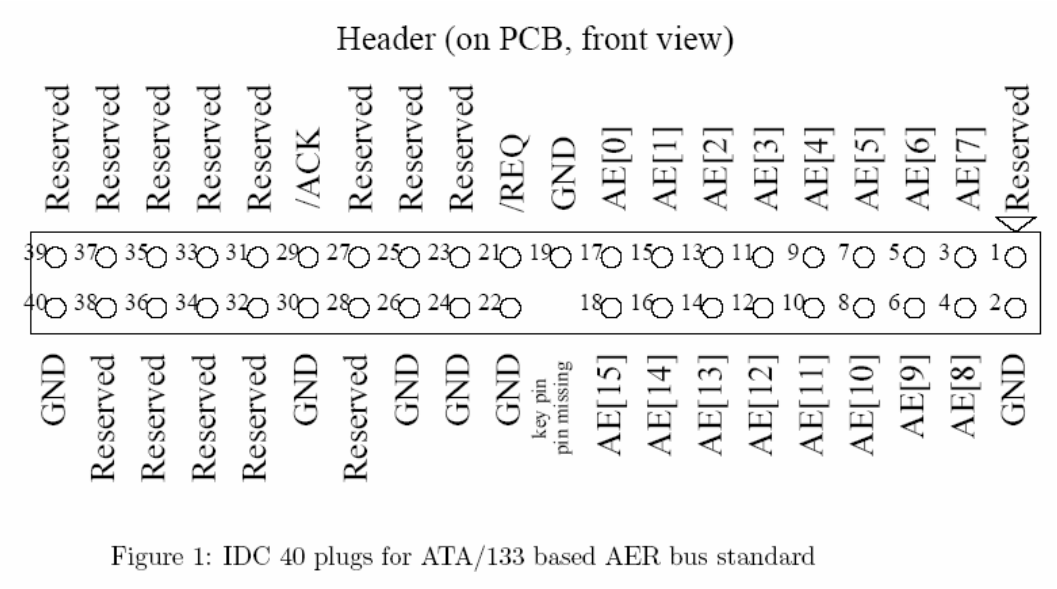
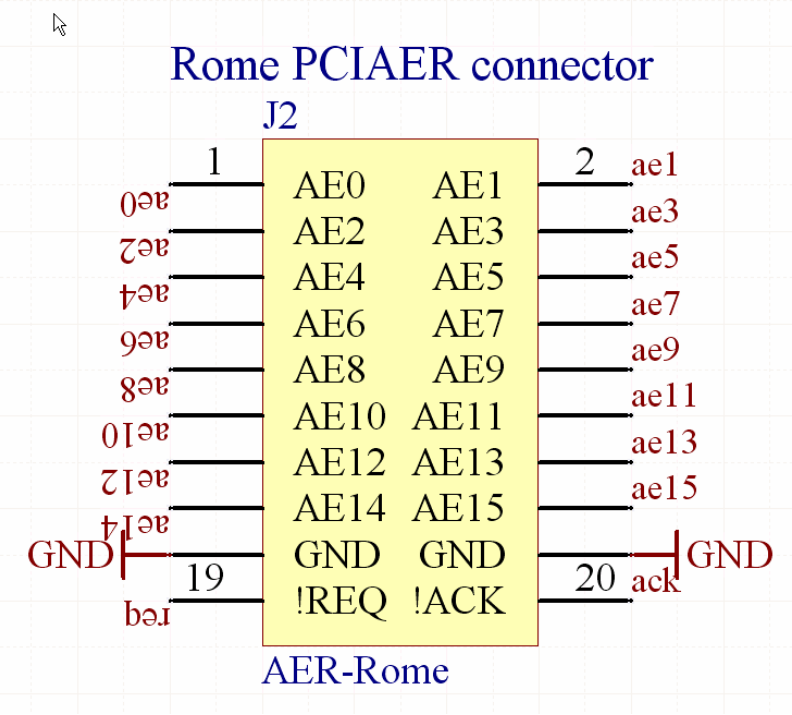


Figure 1: IDC 40 plugs for ATA/133 based AER bus standard

The other connector on the board is a 20 pin SCX Rome PCI-AER connector on it. That cabling follows



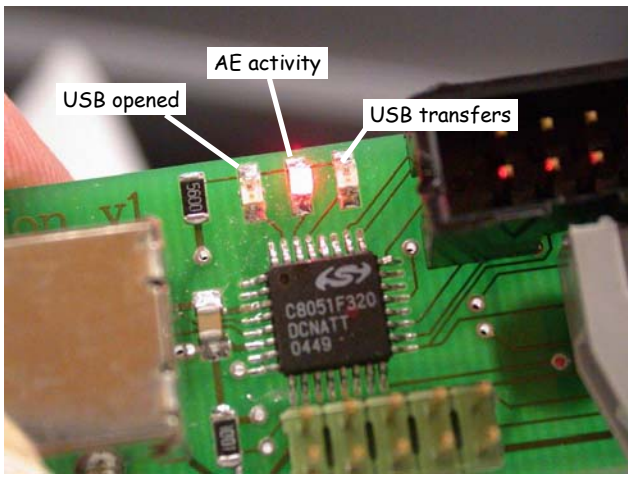
Handshake polarities and timing

As always for CAVIAR, *request and acknowledge signals are active low* and it is assumed that the chip is a point-to-point sender that makes data valid from the time request goes low until it goes low again.

The microcontroller reads the address a fraction of a microsecond after making acknowledge active (low), so multisender word parallel chips should also be handled correctly. See the plot on page 17 for the measured timing.

However, there could be timing problems with senders that do not keep the address lines valid after acknowledge from the receiver goes low. These chips may require revised firmware.

LEDs



The first LED (closest to USB connector) signals USB connection status. It turns on when the device is opened from the host software.

The middle LED signals AE activity. It turns ON at chip request and off at end of handshake.

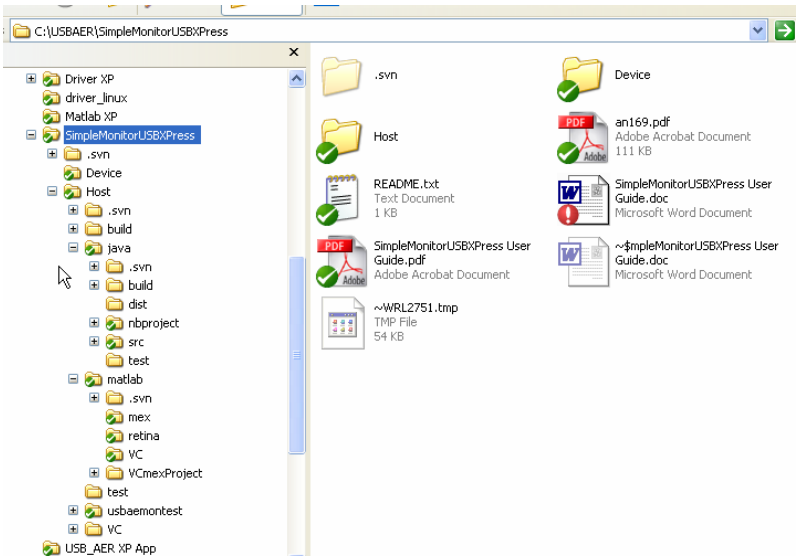
The last LED signals host transfers. It turns on during each USB transfer, e.g. every 225 events under high load or every 32ms under light loading.

USB inactive handling

While the device has not been opened by user software it handshakes with the sender. A handshake cycle requires about 1us. The middle LED still indicates AER activity.

Software installation

This software uses the Silicon Labs USBXPress device USB library and host driver which simplify USB programming. This software is only supplied for Windows. Here is a snapshot of the organization:



It consists of firmware (in folder Device) as an SiLabs IDE project for the C8051F320 controller, and software (in folder Host) that has a mex file for matlab to capture the address events and timestamps. There is also software in folder java for use in java programs that need to capture events.

Get working copy of CAVIAR repository

Use subversion to get a working copy of the CAVIAR repository. The repository URL is

```
https://svn.ini.unizh.ch/repos/alvsi/CAVIAR
```

SimpleMonitorUSBXPress is located in `CAVIAR/wp5/USBAER`. Checking out this folder gives you everything you need.

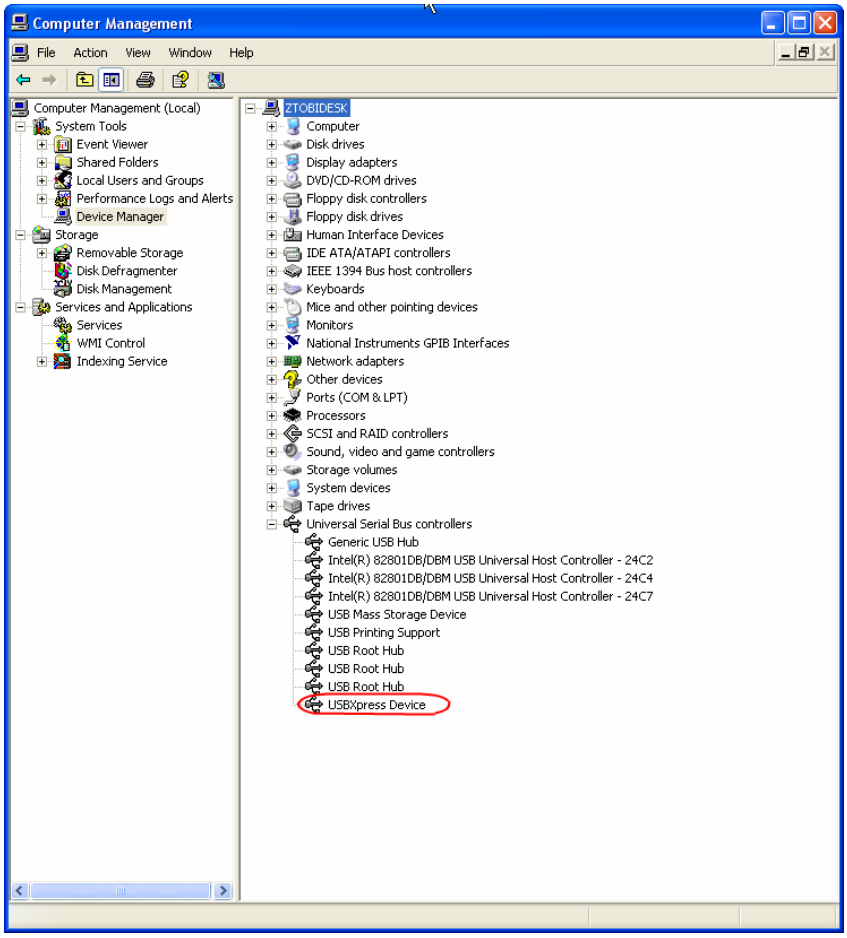
USBXPress Driver installation

1. Go to the folder SimpleMonitorUSBXPress\Host\Driver. Run `preinstaller.exe`, and answer "Continue anyhow" to any prompts about uncertified Windows drivers. You may need to browse for the location of the Driver folder. This step prepares your machine by copying driver files to locations known to Windows so that when you plug in the USB device the drivers can be located and installed.

2. *Alternatively,* download the USBXPress installer from http://www.silabs.com. You can try the direct link: USBXPress which may become outdated. The present code is based on USBXPress version 2.1. Run the USBXPress installer and install to c:\SiLabs. This will install the USBXPress package into a subfolder of the usual SILabs IDE installation, even if you don't have the entire IDE. You don't need the IDE unless you plan to modify the firmware on the device. Navigate to C:\SiLabs\MCU\USBXpress\Driver and run the PreInstaller.exe driver preinstaller. This prepares your machine by copying driver files to locations known to Windows so that when you plug in the USB device the drivers can be located and installed.

3. Plug in the board. It should show up in a popup first briefly as "USBAER" and then immediately afterwards as "USBXPress Device". You will now be prompted to allow installation of an uncertified Windows driver; answer "Continue anyhow" to any prompts. Also later in the Device Manager the device will appear as "USBXPress Device". It can be

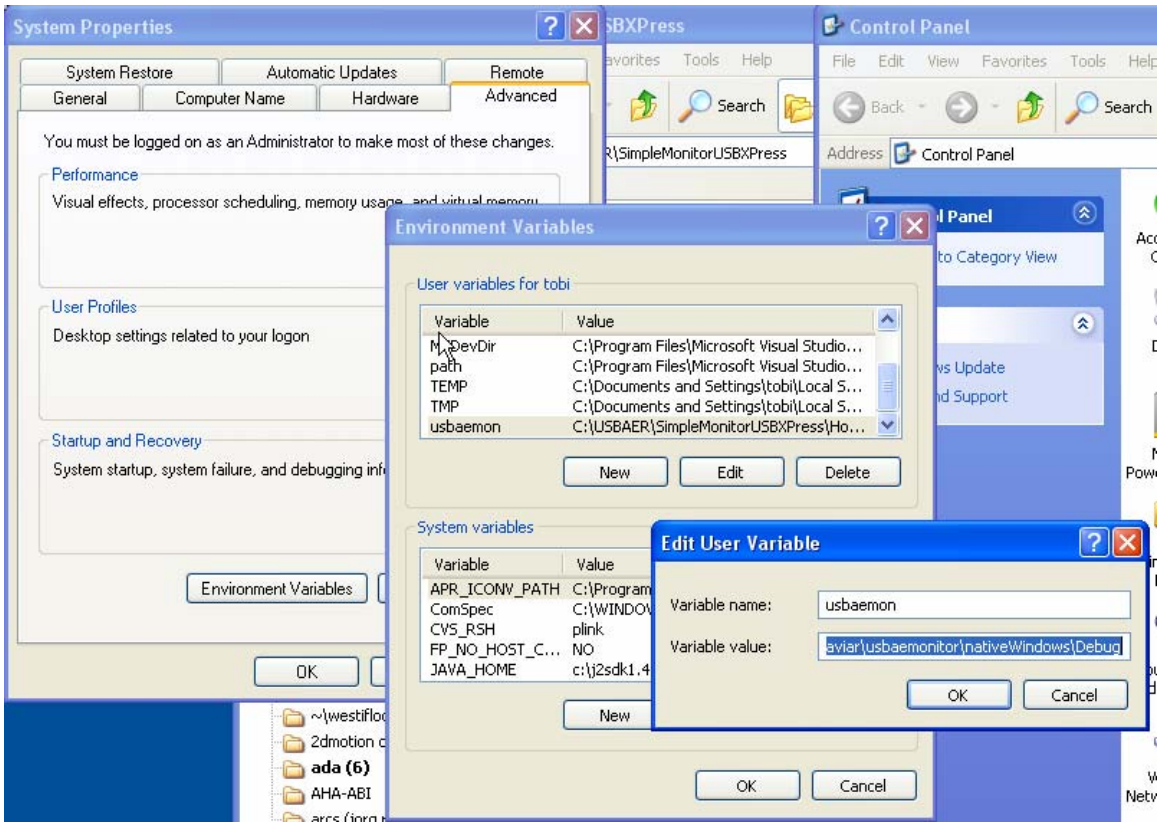
seen in the Windows Device Manager (right click My Computer and select Manage) as shown here



DLL and Mex PATH setting

You have a local working copy of the CAVIAR repository and thus have the DLLs and other software. You have also installed the USBXPress driver. But Windows must also be able to locate the DLLs to access the driver.

1. **Set Windows PATH environment variable.** In order for executables under windows to be able to locate DLLs (dynamically linked libraries) they need to be somewhere on the Windows `PATH`, like for example in the folder you start a program from. You can view your Windows `PATH` by opening a cmd window and typing `set`. Certain folders like `%SystemRoot%` and `%SystemRoot%\system32` are always on the `PATH`, (these will usually be `C:\Windows` and `C:\Windows\system32`) so you can just put the DLLs there, but that is bad if there are updates; you would need to manually copy the files there. It is probably better to put the folders that the DLLs live in on the `PATH`. Do this, albeit painfully, using the control panel System/Advanced/Environment Variables. Add the paths to the folders with the DLLs to the PATH. If `SiUSBXp.dll` cannot be located you will get a mex file error although matlab will find `usbaemon` and shows its path. What you can do is define a new variable and use it in either the Windows User or System `PATH`, as shown here



1. **Set Matlab PATH.** For Matlab use, add the folder SimpleMonitorUSBXPress\Host\matlab\mex to your Matlab PATH. You can do this from the `File/Set Path...` menu item. Then Matlab will be able locate the mex file usbaemon.dll. But that is not enough; usbaemon.dll also needs to be able to find the SiUSBXp.dll. That's why you also need to add the mex folder to your Windows `PATH` environment variable.

2. **For java use**, add the folder, e.g. `C:\USBAER\SimpleMonitorUSBXPress\Host\java\src\ch\unizh\ini\caviar\usbaemonitor\nativeWindows` to your `PATH` so that java can locate both the `USBAEMonitor.dll` and `SiUSBXp.dll` files.

3. **Test your installation.** After connecting the board you can see if its working without an AER sender just by typing `usbaemon` to a Matlab prompt. You should get a silent return. Now usbaemon is loaded and the driver is trying to capture events. If you have a sender (see following), try capturing events with `addr=usbaemon.` If this works everything should be functional.

Toubleshooting installation

1. **Mex file won't run.** If Matlab complains that the usbaemon DLL is invalid, specifically, reports that "*The specified module could not be found*," and you have verified that your Matlab and Windows `PATH` environment settings are correct, it could be that you have a mex DLL incompatibility or, more likely, you are missing some magic Windows DLLs that are installed

along with Microsoft Visual C++. Included in the `SimpleMonitorUSBXPress\Host\matlab\mex` folder are two DLLs `msvcrtd.dll` and `msjava.dll` that were missing on one system. If you have set your Windows PATH environment variable to correctly to point to this folder Windows will find these DLLs here. We have observed this with a mixture of Matlab 6.5 and Matlab 7.x installations. One way to fix this is to rebuild the DLL. You will need to install Microsoft Visual C++ (we are using MSVC 6). You can very easily rebuild the mex file by running the Matlab script `buildusbaemon`, which is located in the folder `C:\USBAER\SimpleMonitorUSBXPress\Host\matlab\mex`. This script silently rebuilds `usbaemon.dll`.
Another way to find out what DLLs are missing is to run the very nice utility that is in the `depends21_x86.zip` archive. This is also in the `SimpleMonitorUSBXPress\Host\matlab\mex` folder. Then if you can locate these missing DLLs on another machine you can copy them to the mex folder.

Using the software

You can use matlab or java to capture events to user software. The operation is similar: You open the device once, and then poll at your convenience for available events. When finished, you close the device.

Using matlab

AE are captured by a matlab mex function usbaemon. Here is the help file for usbaemon:

```
[addr,ts,overrun]=usbaemon
  collects address-event data from Sevilla mini USB AER board
  addr are uint16 addresses.
  ts are uint32 timestamps in microseconds, which have been unwrapped from uint16 values
  received from USB SI labs C8051F320 controller device.
  overrun is double flag that is set to 1 when there is a driver overrun
  (>16k events since last call) and 0 otherwise. If overrun is not an output parameter then warning
  messages are printed on driver overrun.

  The events are buffered in the SiLabs USB host driver, which
    has a 64kB buffer, enough to hold 16kEvents.

  The timestamps wrap at 32 bits*1us=4295 sec=71 minutes. They are
  reset when the device is opened.

  If there is a buffer overrun (>16k events since last read) 16k events are
  returned and you have probably lost the newest events.

  To plot the addresses vs time:
  [addr,ts]=usbaemon;
  plot(ts,addr);

  The device buffer holds 900 bytes=900/4=225 events. Transfer of the USB
  buffer requires about 860us, during which time the device does not
  acknowledge a request. Thus with high activity you will notice periodic jumps of about 850
  in the timestamps; these are the USB transfers.

  Handshake time from req active to ack inactive while acquiring events is measured to be about 4.3us.
  There are approx 860 us pauses every buffer's worth of events for USB transfer.
  Therefore max theoretical xfer rate is about 225 events
  in 4.7us*225events+860us = 225 events/1.8ms= 117k events/sec.
  This rate is reduced somewhat by transmitter's ability to
```

generate a new request after acknowledge goes inactive.
Measured rate with tmpdiff11 retina (not tuned for max output rate) is about 50k events/sec. Peak rate is (during a frame) 117k events/sec

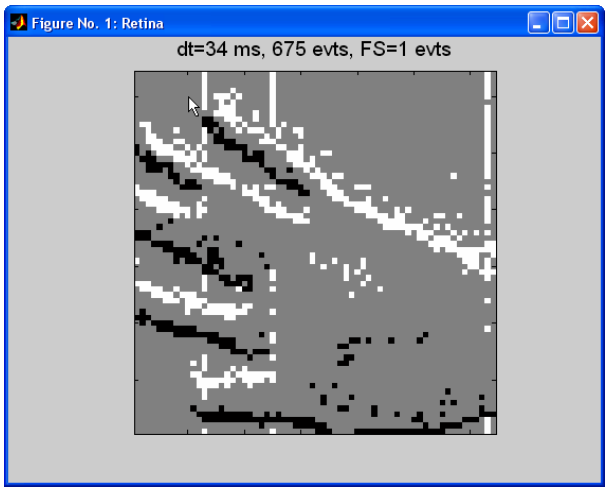
If USBXPress DLL is not loaded, it will be.
First call opens device, which remains open.
To reset and reopen device, clear mex or clear usbaemon.

When the device is not opened, then it just handshakes with AE transmitter, without storing events. Then the cycle time is about 1us/event.

The LEDs indicate activity. The yellow LED goes high on req and low on ack, and the red LED toggles for each USB transfer and is off otherwise, so it can be used to tell if the device driver has been opened.

usbaemon.dll needs to be on the matlab path to be recognized as a mex-function, and SiUSBLib.dll need to be somewhere on the Windows PATH. Otherwise you will get an error that the mex file can be loaded by not intialized.

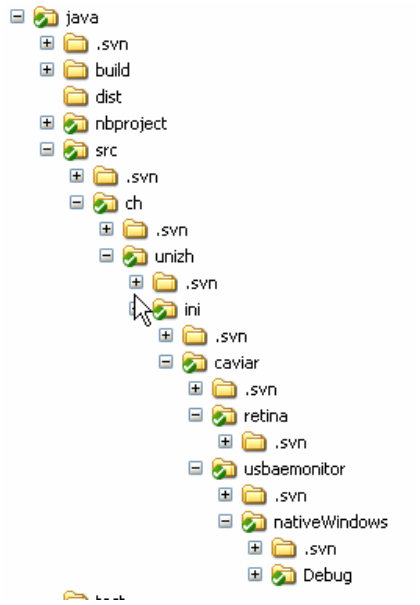
The folder retina has a reference m-file showLiveRetina.m that demonstrates use of usbaemon. A typical snapshot of output from this function looks like this:



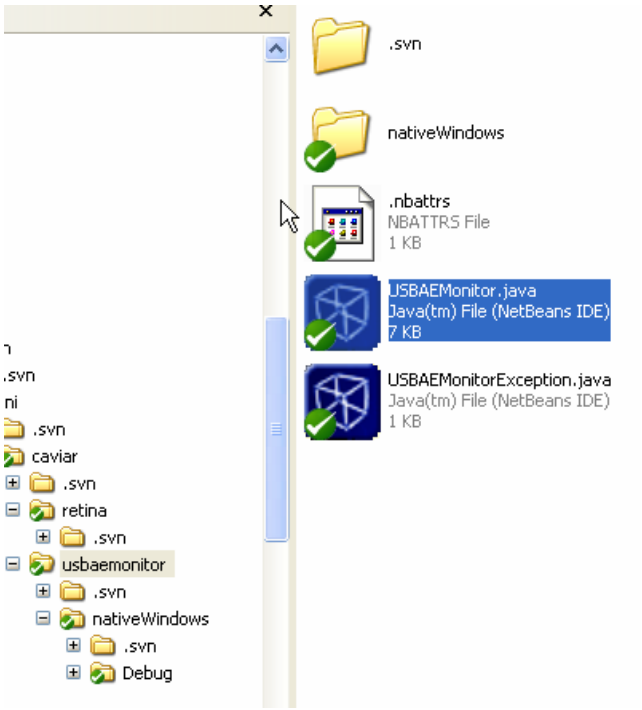
`showLiveRetina.m` demonstrates how to make an application hot-pluggable by using try/catch blocks, so that the USB cable plugged in or out at any time.

Using java

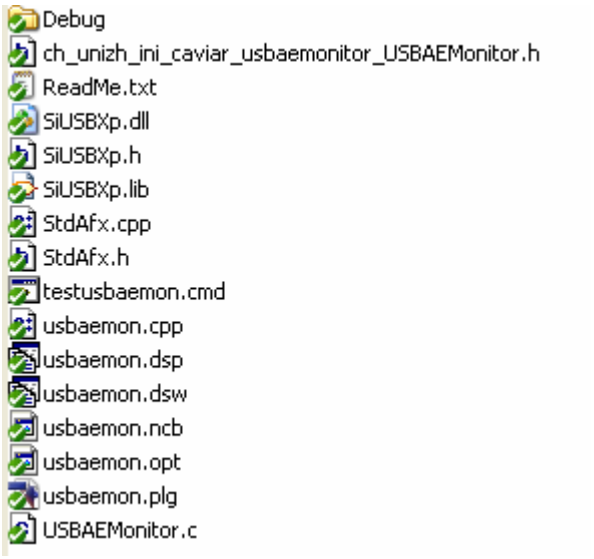
The java files are located in the java folder:



There is a package `ch.unizh.ini.caviar.usbaemonitor` in the jar file `java\dist\usbaemon.jar` that has the class `USBAEMonitor` that accesses the USB board:



The folder `nativeWindows` (shown below) is the source for the DLL `USBAEMonitor.dll` that implements the native methods in the java class `USBAEMonitor`. It too uses `SiUSBXp.dll`:

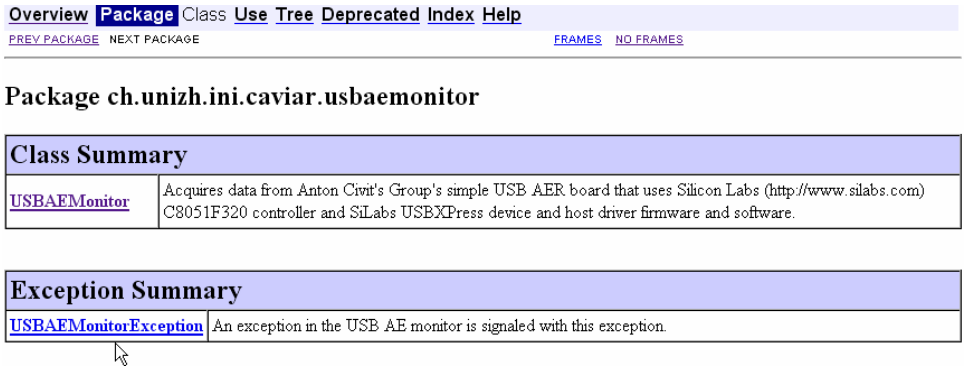


Example java application

There is a very preliminary java application to display the retina output in the package `ch.unizh.ini.caviar.retina`.

Javadoc

Documentation for the java classes is located in `USBAER/SimpleMonitorUSBXPress/Host/java/dist/javadoc/index.html`. If you view this HTML you will see something like this



Here is a snapshot of the `USBAEMonitor` class javadoc

ch.unizh.ini.caviar.usbaemonitor

Class USBAEMonitor

```
java.lang.Object
  └ ch.unizh.ini.caviar.usbaemonitor.USBAEMonitor
```

```
public class USBAEMonitor
extends java.lang.Object
```

Acquires data from Anton Civit's Group's simple USB AER board that uses Silicon Labs (http://www.silabs.com) C8051F320 controller and SiLabs USBXPress device and host driver firmware and software.
The DLL's USBAEMonitor.dll and SiUSBXp.dll must be accessible for windows programs. Generally, this means they must be somewhere on the PATH, for example, in `WINNT\system32`, or they can be in the directory the program is started. It is generally simplest to just put the folders on the PATH variable.
Events are captured as 16 bit addresses and 32 bit timestamps with 1us tick. To use this class, construct an instance of `USBAEMonitor`, then `open()` it. Each time you want to capture available events, call `acquireAvailableEventsFromDriver()`, which returns the number of events grabbed. Access these events with `getAddresses()` and `getTimestamps()`, which each create new java arrays of the captured data. `overrunOccured()` can be used to see if there was a driver overrun.
See the main() method for an example of use.

Constructor Summary

USBAEMonitor()	Creates a new instance of USBAEMon

Method Summary

int	**acquireAvailableEventsFromDriver**() Gets available events from driver.
int	**close**() Closes the device and frees the internal device handle.
short[]	**getAddresses**() Returns a newly constructed array of addresses
int	**getNumEventsAcquired**() Returns the number of events acquired by the last call to `acquireAvailableEventsFromDriver()`
int[]	**getTimestamps**() Returns a newly constructed array of timestamps that have a 1us tick
static void	**main**(java.lang.String[] args) Tests event acquisition.
int	**open**() Opens the device driver and gets a handle to the device which is internally maintained.
boolean	**overrunOccured**() Is true if an overrun occured in the driver (>16k events) the last time `acquireAvailableEventsFromDriver()` was called.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

USBAEMonitor

```
public USBAEMonitor()
```

Creates a new instance of USBAEMon

Method Detail

open

```
public int open()
    throws USBAEMonitorException
```

Opens the device driver and gets a handle to the device which is internally maintained. Timestamps are reset to 0 and buffers are flushed. To reset timestamps, close() and reopen the device.

Returns:
0 if always.

Throws:
USBAEMonitorException - if there is a problem.

See Also:
close()

---

acquireAvailableEventsFromDriver

```
public int acquireAvailableEventsFromDriver()
                                    throws USBAEMonitorException
```

Gets available events from driver. Call this before calling getAddresses() or getTimestamps(). USBAEMonitorException is thrown if there is an error. overrunOccured() will be true if these was an overrun of the host USBXPress driver buffers (>16k events).

Returns:
number of events acquired. If this is zero there is no point in getting the events, because there are none.

Throws:
USBAEMonitorException - .

---

getNumEventsAcquired

```
public int getNumEventsAcquired()
```

Returns the number of events acquired by the last call to acquireAvailableEventsFromDriver()

Returns:
number of events acquired

---

getAddresses

```
public short[] getAddresses()
```

Returns a newly constructed array of addresses

Returns:
array of addresses

---

getTimestamps

```
public int[] getTimestamps()
```

Returns a newly constructed array of timestamps that have a 1us tick

Returns:
the timestamps

---

overrunOccured

```
public boolean overrunOccured()
```

Is true if an overrun occured in the driver (>16k events) the last time acquireAvailableEventsFromDriver() was called. This flag is cleared by the next acquireAvailableEventsFromDriver(). If there is an overrun, the events grabbed are the most ancient; events after the overrun are discarded. The timestamps continue on but will probably be lagged behind what they should be.

Returns:
true if there was an overrun.

---

close

```
public int close()
```

Closes the device and frees the internal device handle. Never throws an exception.

Returns:

0 if successful

main

```
public static void main(java.lang.String[] args)
```

Tests event acquisition.
Parameters:
`args` - the command line arguments

skip-navbar_bottom

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES All Classes
SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Testing the USBAEMonitor class

There is a cmd file that tests the java implementation of USBAEMonitor. This test script is here:

```
C:\USBAER\SimpleMonitorUSBXPress\Host\java\src\ch\unizh\ini\caviar\usbaemonitor\nativeWindow
s\testusbaemon.cmd
```

It contains the following:

```
set CP=C:\USBAER\SimpleMonitorUSBXPress\Host\java\classes
set
LIBPATH=C:\USBAER\SimpleMonitorUSBXPress\Host\java\src\ch\unizh\ini\caviar\usbaemonitor\nati
veWindows\Debug;C:\USBAER\SimpleMonitorUSBXPress\Host\java\src\ch\unizh\ini\caviar\usbaemoni
tor\nativeWindows
cd
C:\USBAER\SimpleMonitorUSBXPress\Host\java\src\ch\unizh\ini\caviar\usbaemonitor\nativeWindow
s
java -cp %CP% -Dload.library.path=%LIBPATH% ch.unizh.ini.caviar.usbaemonitor.USBAEMonitor
pause
```

The output checks every second for events and prints the number of events, the x and y addresses decoded for the retina, over "overrun" if there was a buffer overrun:

```
USBAEMonitor: loaded dynamic link library USBAEMonitor.dll
USBAEMonitor.c: USB AE Mon product string: USBXpress Device
no events
no events
no events
overrun 16384 events: 63,63,32930 32,8,47608 11,9,74944
overrun 16384 events: 42,2,2580576 41,0,2580582 43,11,2580586 43,
1146 events: 63,37,3447348 62,37,3447354 60,37,3447358 53,37,3447362
no events
no events
overrun 16384 events: 46,32,3740020 63,63,3769436 59,23,3783206
106 events: 53,54,5792026 51,54,5792030 47,54,5792036 56,55,5792042
no events
no events
no events
no events
no events
USBAEMonitor.c: SI_Close successful
closed
```

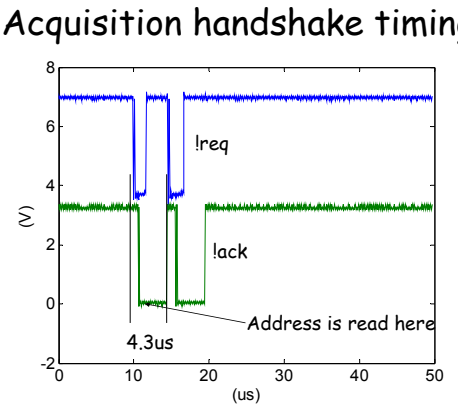
Performance

USB1 has basic data rate of 12 Mbps (million bits/sec), so in principle a system should be able to achieve a raw data transfer rate of about 12 Mbps/10=1.2 MBps (accounting for a bit of overhead). If each event requires 4 bytes consisting of a 16 bit address and a 16 bit timestamp, then about 1.2MB/4=400 k events/sec (keps) is the absolute limit for a USB1-based system. In fact pumping 900 bytes into the USB bus from the device requires about 860 us, a rate of about 1 MBps, although this must be decreased a bit by the fact that the call to Block_Write returns as soon as possible, before the bytes have actually been written to the bus.

Measured performance of the present system peaks at about 100keps when using a synthetic source (a function generator) to generate "events" at the minimum handshake cycle time. When recording from the transient retina, which does not always immediately generate a new event, sustained performance in matlab is about 50keps. Since each event is transferred with 4 bytes of information (16 address bits and 16 timestamps bits) this represents a transfer rate of 200kBps, which is about 13% of the raw 12Mbps USB1 rate. It is clear that more performance should be achievable but this would probably require custom driver development that uses the full double-buffering USB FIFO capability both on controller and host side. In addition, the event capture time of 4.3 us on the device limits the raw event capture rate to about 230 keps.

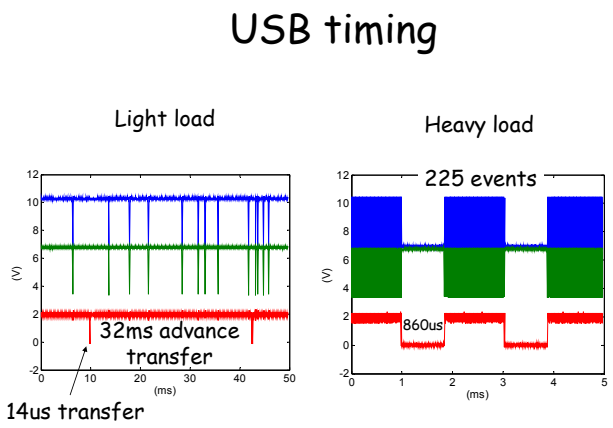
Handshake timing

Timestamps are assigned with 1 us resolution in the microcontroller but each handshake and each handshake/acquisition cycle requires about 4.3 us.



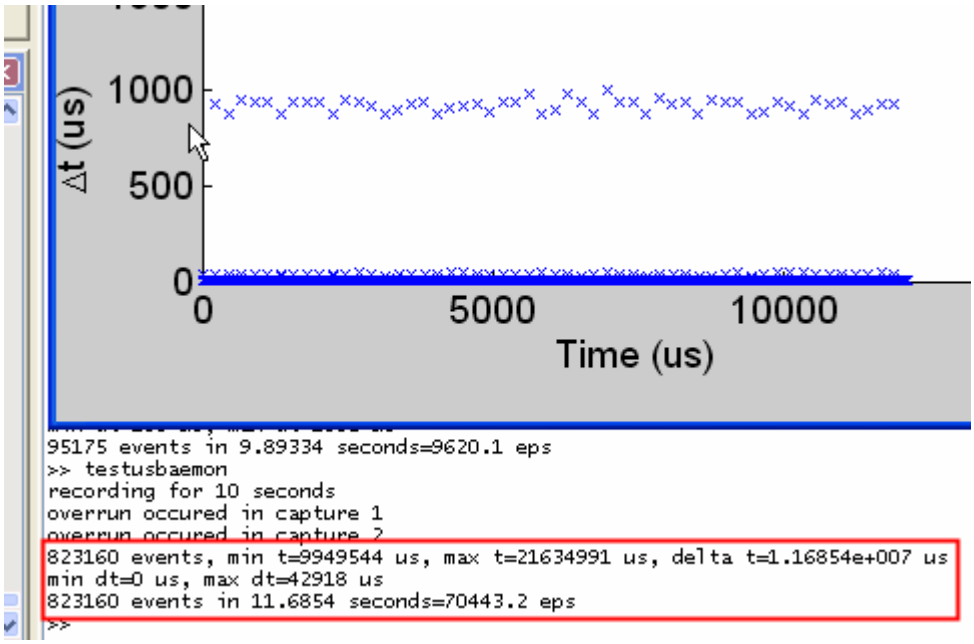
Frame transfer timing

Every 225 events (900 bytes), there can be a pause of about 860us in the timestamps that comes from the time that the microcontroller is doing a USB transfer. During this time it refuses to acknowledge a sender request. Under light load, advance transfers of acquired events require a call that takes under 20 us when there are only 2-3 events to be sent.



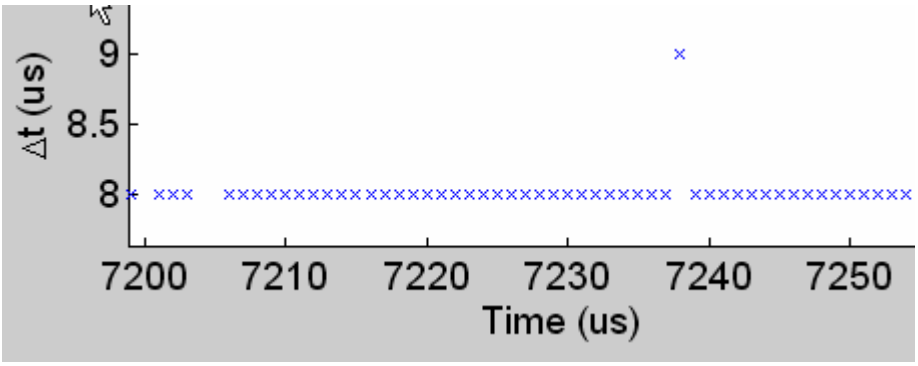
Maximum acquisition rates

An experiment using a function generator to generate "requests" in open-loop to the device showed the following results with an "interspike" interval (pulse frequency) of 8us, corresponding to a sender rate of 125keps.



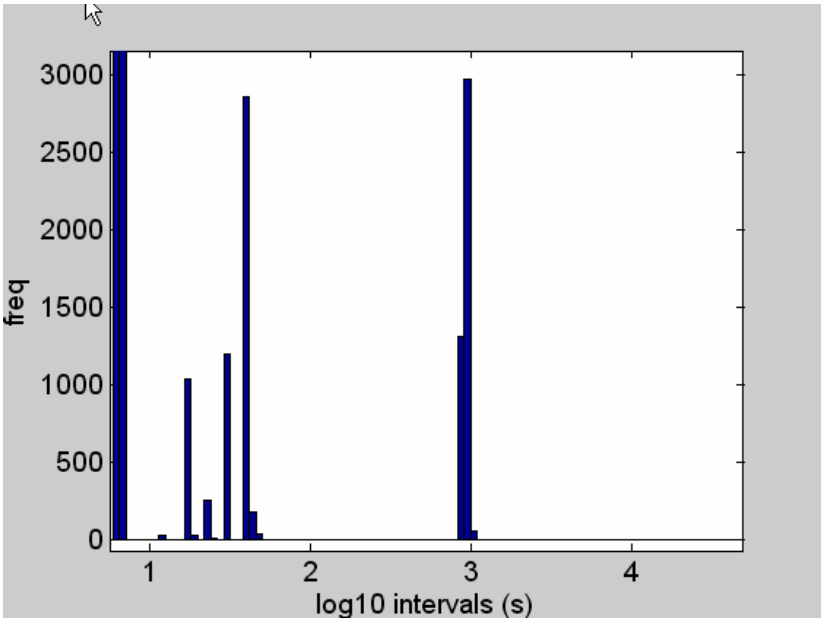
The matlab usbaemon captured at a rate of 70kHz with periodic jumps in the interspike interval corresponding to USB transfers.

Zooming up on the interspike intervals above shows the following



Most "events" are captured correctly with 8 us interval but quantization sometimes causes jumps to other intervals.

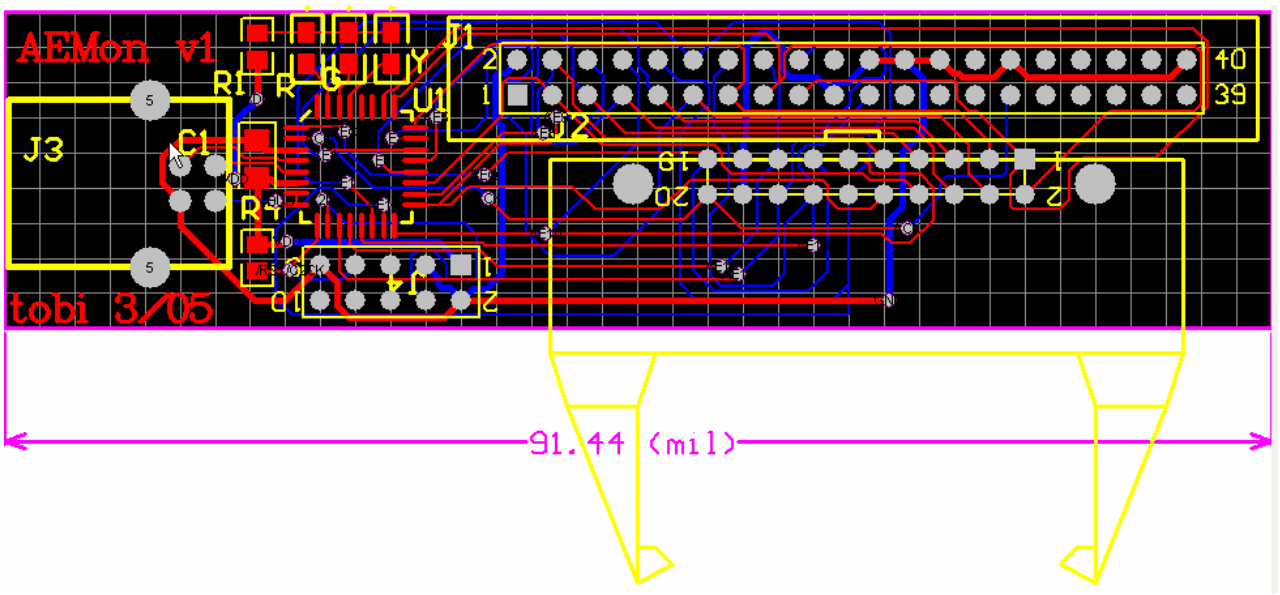
Increasing the "frequency of events" (the function generator frequency) to 166kHz (period 6us) increases the average capture rate to about 97kHz, corresponding to the absolute maximum rate. A histogram of the captured log10 timestamps looks like the following

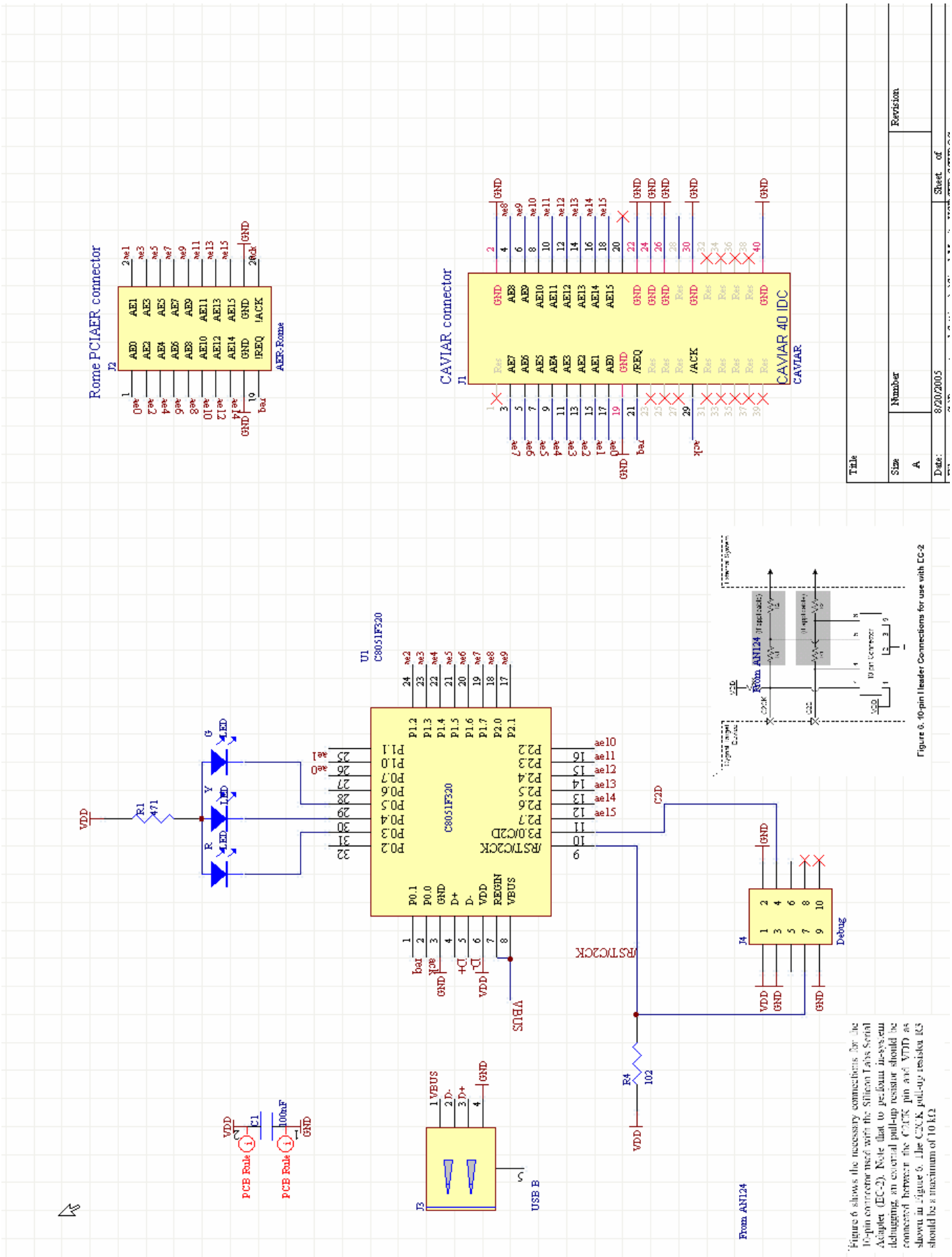


(The time scale should read "us" and not "s" in this plot.) Nearly all of the 1M points captured in 10 s are at 6 u. A few percent of the timestamps are a few multiplies of this value, and a larger population at 1 ms corresponding to the USB transfers. There are still a very few interspike intervals at 65 ms that are of unknown origin.

Board schematics and layout

The PCB design is an Altium Protel DXP design; it is stored in the folder PCB. Following are the board layout and schematic.


AEMon v1
J3
C1
R1
R
C
Y
U1
J1
J2
40
39
20
1
tobi 3/05
91.44 (mil)


Rome PCIAER connector
CAVIAR connector
CAVIAR 40 IDC
C8051F320
Debug
USB B
From AN124
Figure 6. 40-pin Header Connections for use with EC-2