LAUTERBACH
DEVELOPMENT TOOLS

Training Power Probe

Release 09.2023

MANUAL

Training Power Probe

Training Power Probe

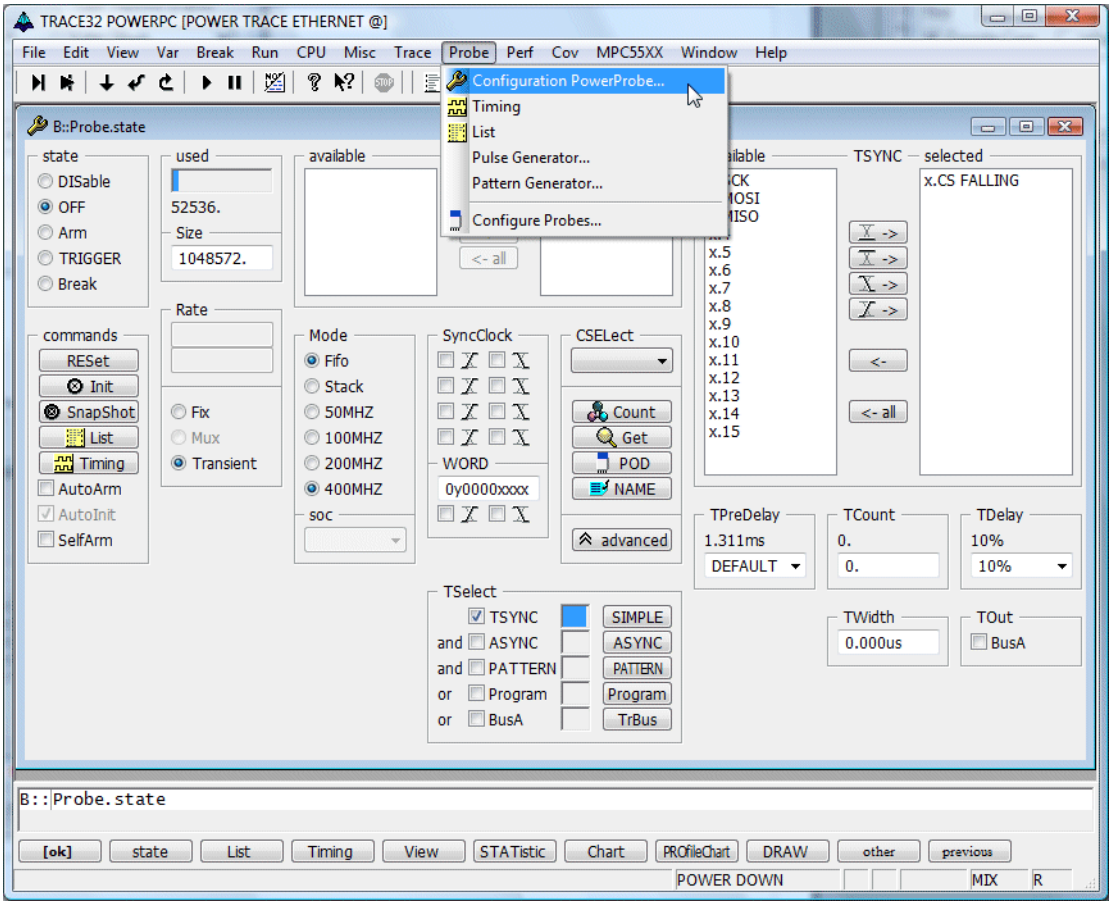
Version 10-Oct-2023

Basics

The PowerProbe is an accessory device for the TRACE32 debugger. It is optimally suited to record peripheral signals of up to 50 MHz.

The PowerProbe Configuration Window

To use the PowerProbe the first step is to open up the PowerProbe configuration window.

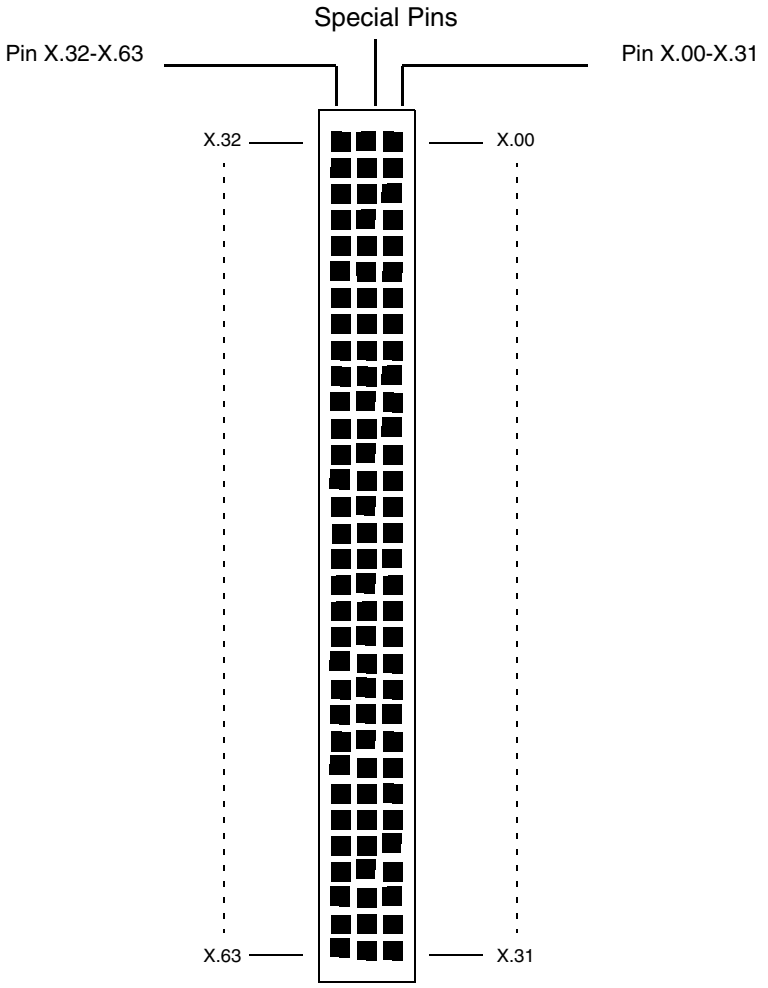


Probe.state	Display PowerProbe configuration window

The PowerProbe Connector

Now you have to physically connect some signals to the PowerProbe. The PowerProbe has 3 pin columns, each with 32 pins. The regular signal inputs are on the left and right column. The middle column contains special input and output pins.

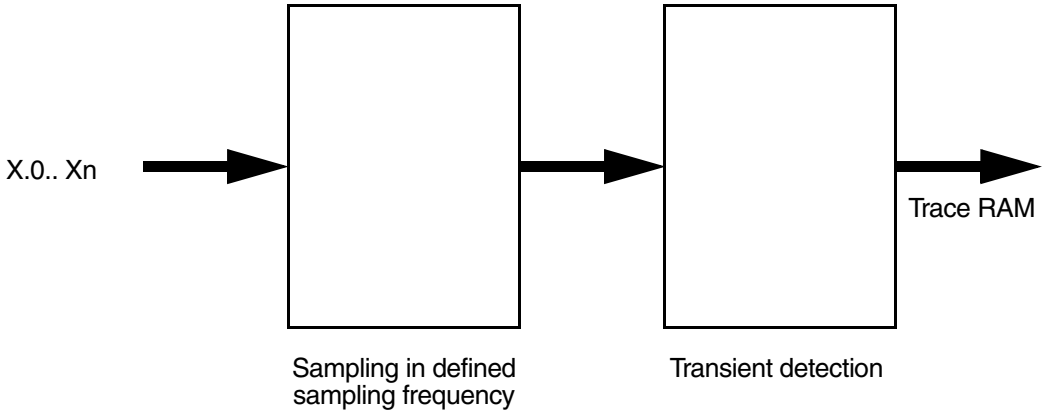
What is especially important, is to **connect the ground pins** on the middle column to a ground plane or pin on your target!



PowerProbe Input Connector

Standard Sampling Mode

The standard mode for the PowerProbe is Transient Mode.

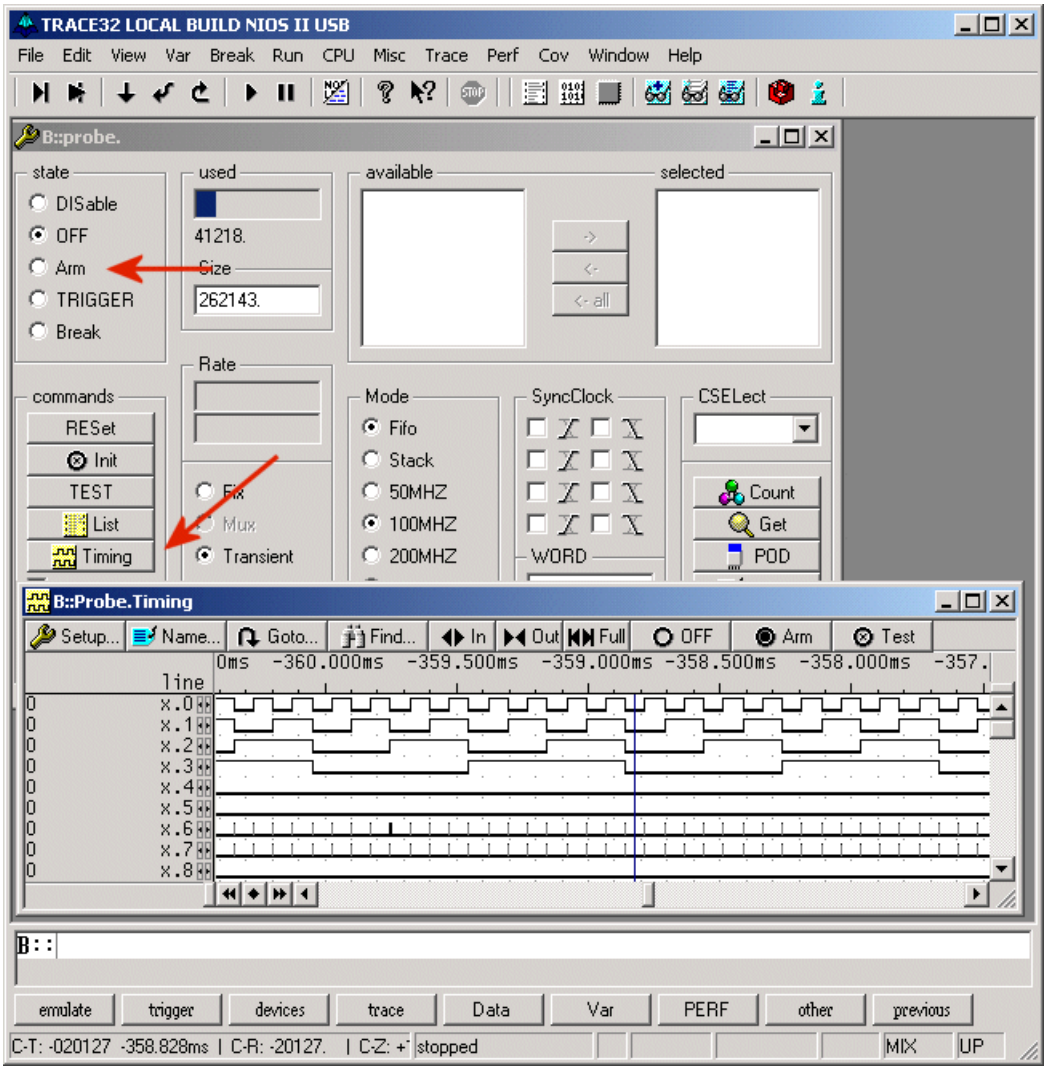


With the settings in the **Mode** box, you can change the sampling frequency of the PowerProbe. The default is **100 MHz** (100 MSamples/s). In this default mode all input pins are freely usable. When you select **200 MHz**, only input pins x.0-x.31 can be used. In **400 MHz** mode only input pins x.0-x.15 are available. In **50 MHz** mode, the PowerProbe lowers its sampling rate to 50 MSamples/s.

```
Probe.Mode 400MHz                    ; set the sampling frequency

Probe.Rate Transient                 ; select transient recording
```

Arm/Disarm the PowerProbe

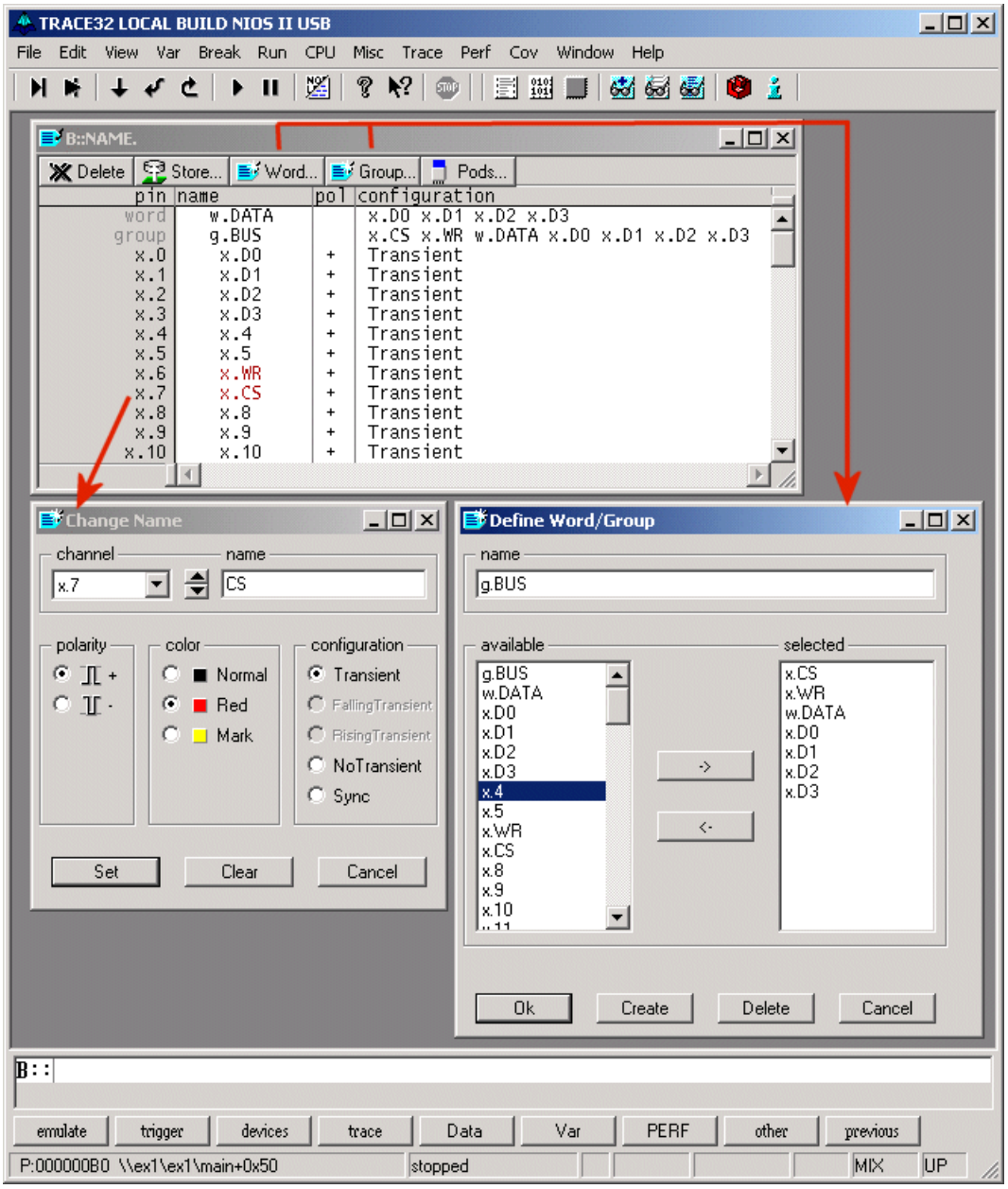
When your pins are physically connected to the PowerProbe, you can start a recording by clicking on the **Arm** button in the PowerProbe configuration window. To stop the recording you can click the **Off** button. You can view your recording by clicking on the **Timing** button.



Probe.Arm	Arm the PowerProbe
Probe.OFF	Disarm the PowerProbe
Probe.Timing	Display trace contents as timing diagram

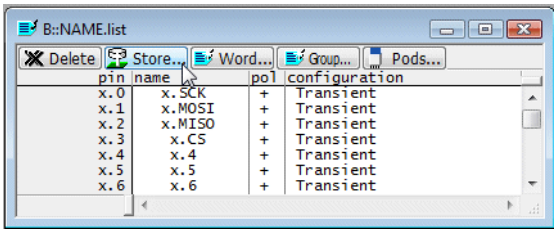
Signal Names

Now you can see your signals, but remembering the meanings of the signals is not very convenient. So first of all we will give the signals better names. This is done with the NAME.list window, which is reachable from the PowerProbe configuration window with the NAME button:

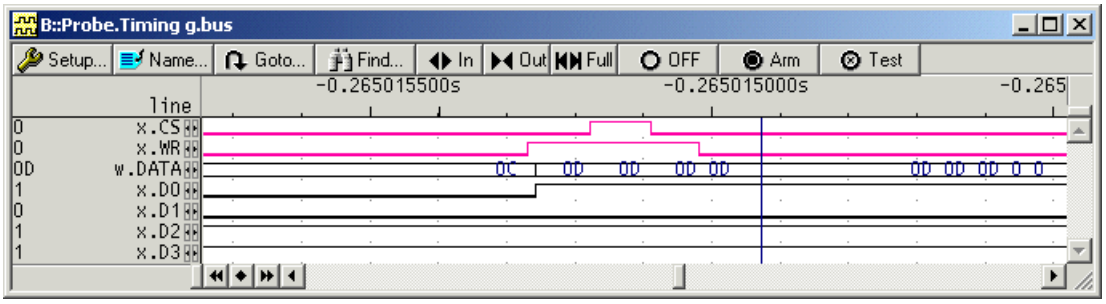


NAME.list	List channel names and attributes
NAME.Set	Assign name and attributes to a channel
NAME.Word *w.<name> <channel> <channel> …*	Group channels to a word

The configuration of the naming of the pins can be stored in a script. The TRACE32 software can create such a script for you: You just have to click on the **Store** button the **NAME.list** window.

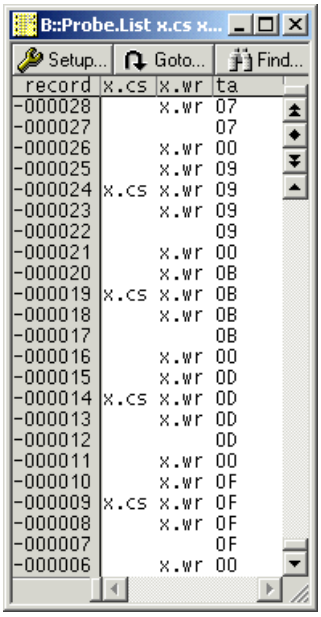


STOre *<file>* NAME	Create script for NAME settings.
ClipSTOre NAME	Create TRACE32 commands for the NAME settings and copy them to the clipboard.



In this example we have a bus on which there are 4 data pins grouped to a word, one active high ChipSelect pin and an active high Read/Write pin.

Another way to view your signal is the **Probe.List** window. This window is useful, if you are more interested in raw bus data than in timing wave forms:



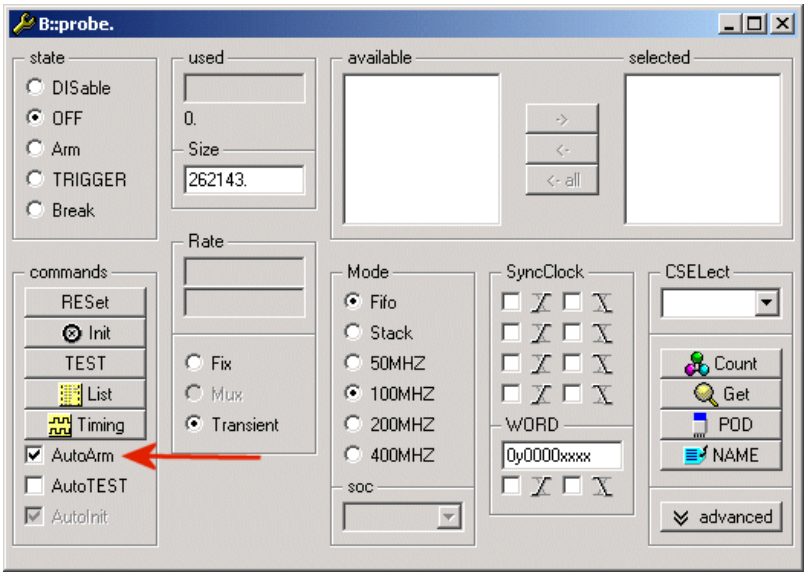
Probe.List	Display the PowerProbe trace contents textually.

One question which often comes up is how much data you can trace with the PowerProbe. The answer is not simple: When a signal is recorded in transient mode (the default), then only changes of the signal are recorded. So in the above recording one full bus cycle needs five records in the PowerProbe memory, when the timing is optimal. In the **Probe.List** window this transient recording becomes most obvious, because one line in the **Probe.List** window corresponds to exactly one record in the PowerProbe. Because the PowerProbe can store up to 256K records, you can record 52000 bus cycles in this example. It isn't really possible to tell how long the time period of a recording will be, because this is completely dependent on the frequency of your recorded signals.

Link the PowerProbe to the Application Debugging

When you are debugging an application, you usually want to start the PowerProbe recording, when your application is started and stop it, when your application stops. This is the default behavior of the PowerProbe, when the **AutoArm** option is enabled.

Probe.AutoArm ON	Link PowerProbe recording to the execution of the application program.



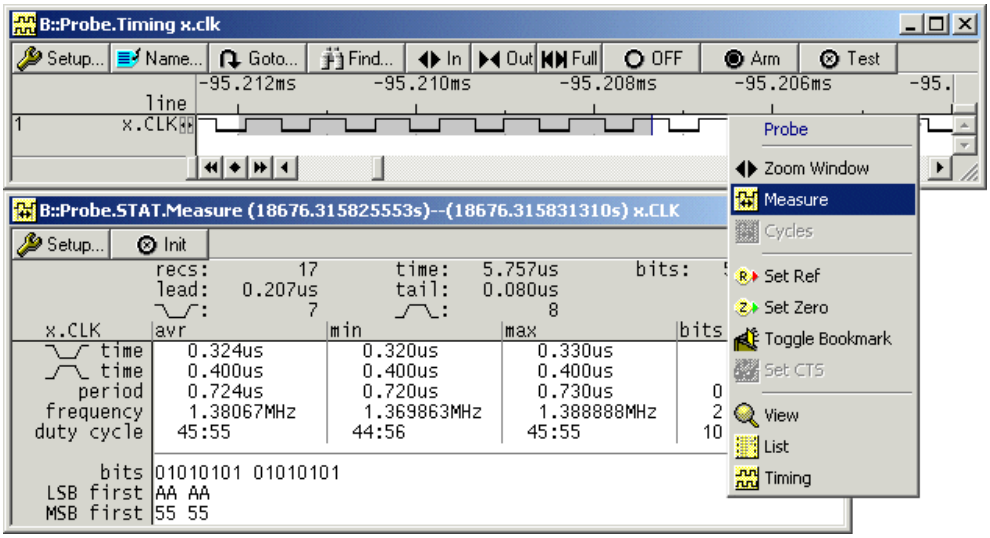
Fifo/Stack Mode

If you select **Stack** mode, the PowerProbe will stop the recording as soon as the trace memory is full. While this mode seems to be natural it actually is only useful when you want to see what happens at the start of a recording. In **Fifo** mode you will see the traced signals up to the point of time, where the PowerProbe was stopped. A more detailed explanation follows in the next chapter.

Probe.Mode *<mode>*	Specify the sampling mode for the PowerProbe trace.

Measurement Statistics

Another useful basic feature is the capability to measure a signal. To do that you select the part of the signal you want to measure and use the context menu, which pops up if you right click on the selected part:



Trace.STATistic.Measure	Statistic information about a single channel

Postprocessing

If you don´t find time to analyse the information sampled by the PowerProbe, you can save the trace information and analyse it later.

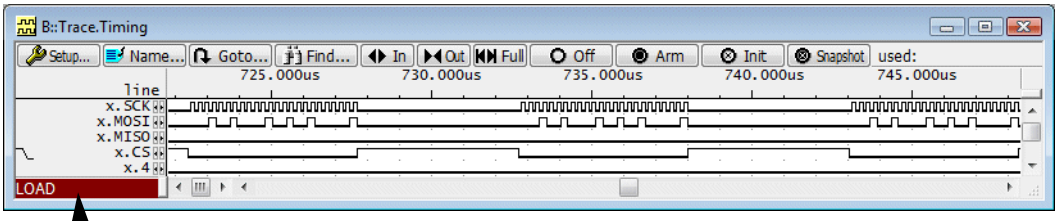
Probe.SAVE *<file>*	Save PowerProbe configuration and trace contents to a file.

There are two ways to postprocess the trace contents of the PowerProbe.

1. Postprocessing with a TRACE32 Instruction Set Simulator.
2. Regarding the saved contents a reference and compare it with other recordings.

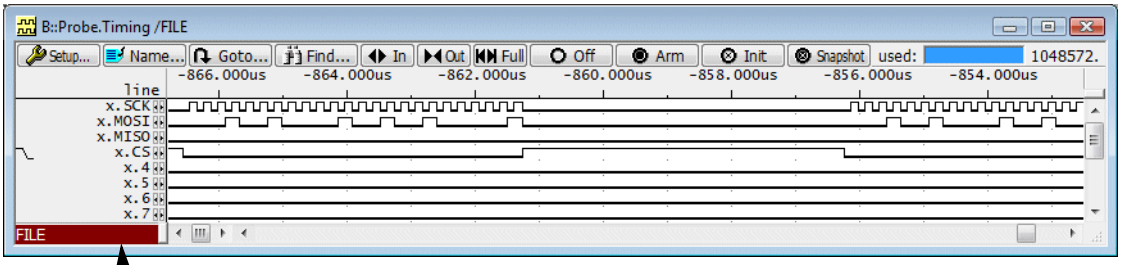
Postprocessing with a TRACE32 Instruction Set Simulator

```
; load trace contents and PowerProbe configuration
Trace.LOAD test1.ad /Config
; all Trace.<sub_cmd> apply to the loaded trace contents now

; display the trace contents as timing diagram
Trace.Timing
```



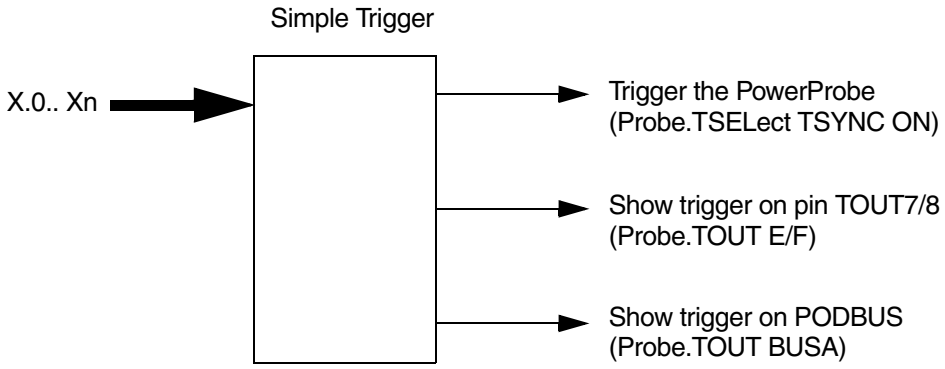
LOAD in red indicates that the displayed trace contents was loaded with the Trace.LOAD command

```
; load trace contents
Probe.FILE test1.ad /Config

; display the trace contents as timing diagram
Probe.Timing /FILE

; compare trace contents with file
Probe.ComPare , x.MOSI /FILE
Probe.ComPare , ALL    /FILE
```

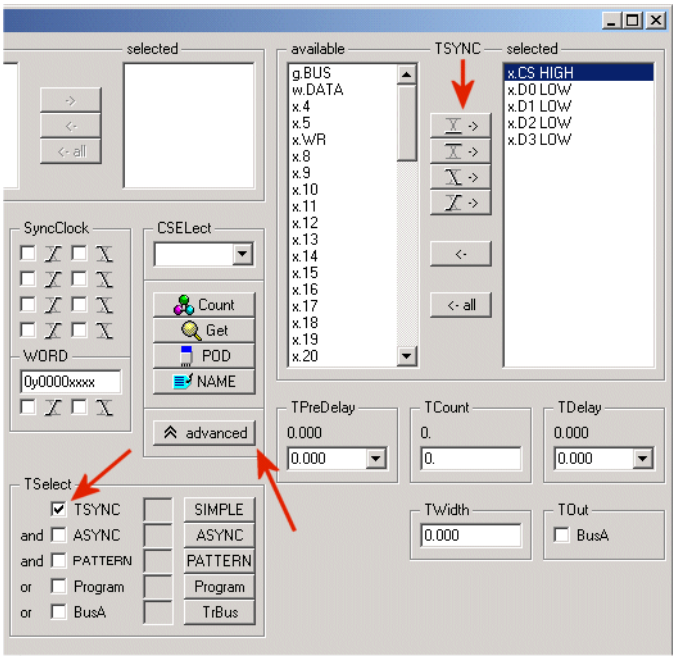


FILE in red indicates that the displayed trace contents was loaded with the Probe.FILE command

Simple Trigger



A standard scenario is that you don't want to look at a complete recording, but you want to see a specific event and the things which happened before and after this event. So it's necessary to define such an event in the PowerProbe:



The checkbox **TSYNC** in the lower left corner selects the so-called Simple Trigger as trigger source.

Probe.TSELect TSYNC ON	Select simple trigger as trigger source for the PowerProbe

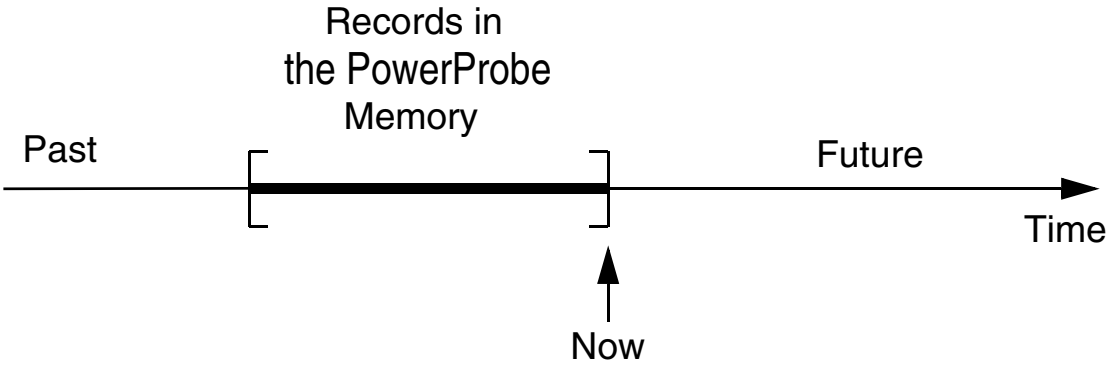
The window pane on the upper right defines the event on which the PowerProbe will trigger. So in the above example a trigger will be detected, when *x.CS* is high and *X.D0-D3* are low.

The settings in this window pane can be changed with the buttons left to it. With these buttons you can select if you want to trigger on a high or low level, or on a falling or rising edge of a signal.

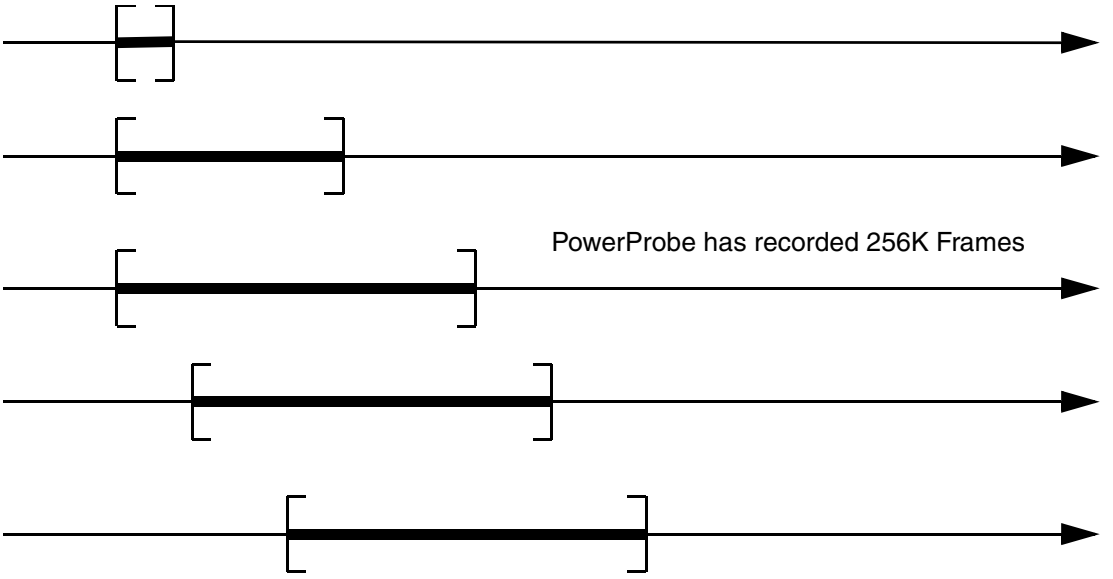
When you want to trigger on a certain event, you should use the PowerProbe in **Fifo** mode. This means that the PowerProbe will record data until it is stopped by an external event. Such an event is either a trigger as described above, or a manual stop of the PowerProbe (as for example when the application stops).

Probe.TSYNC.SELect *<channel>* **Low | High | Falling | Rising**

To understand the rest of the settings of the Simple Trigger it is necessary to understand how the **Fifo** mode works. You can visualize the data which is traced by the PowerProbe as a sliding window on a time line:

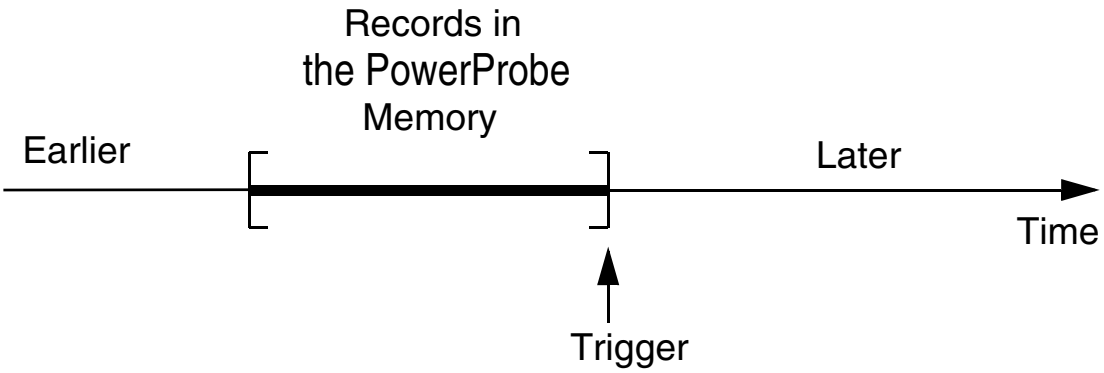


Now you can visualize a recording in the following way:

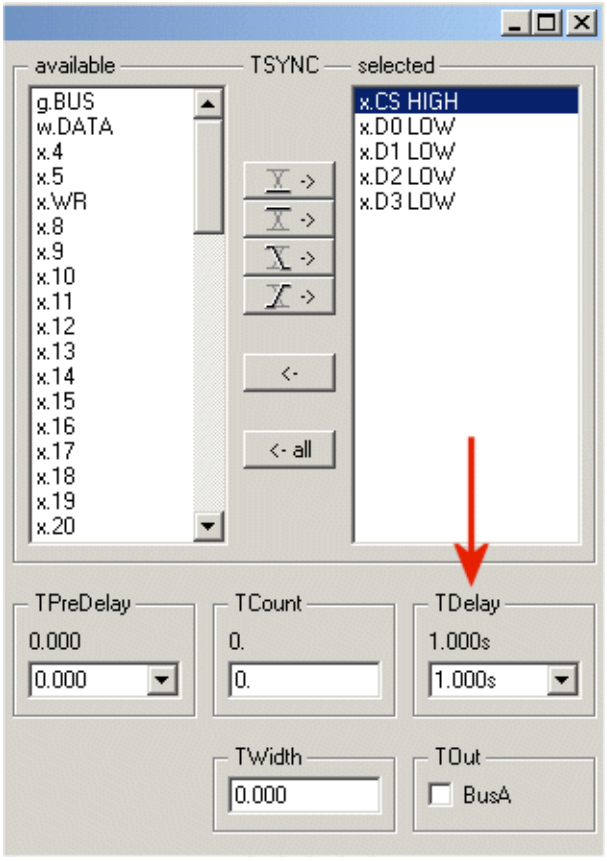


So at first the trace memory starts to fill up and after the trace memory is full, the window starts to slide.

When a trigger event happens, the PowerProbe can break immediately. In this case the data in the PowerProbe will look like this:

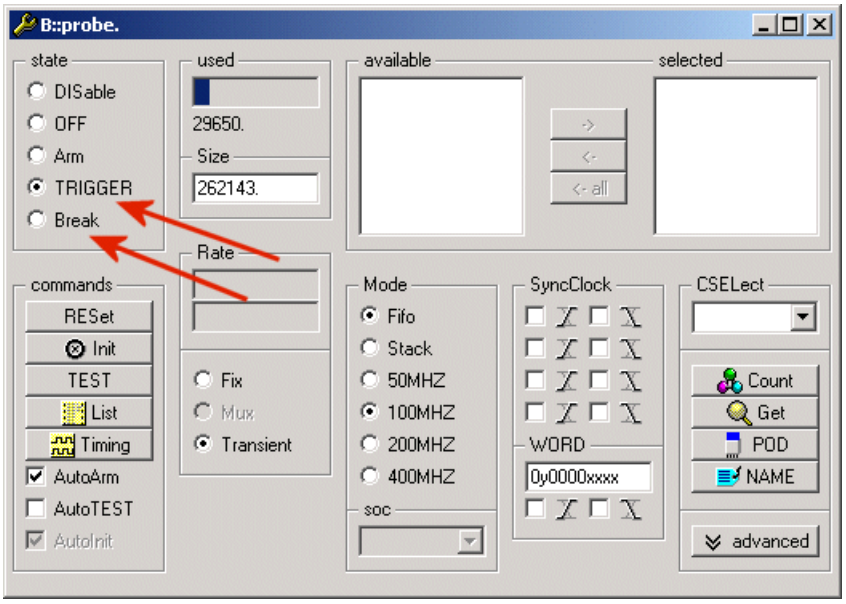


In this case you can see what happened before the trigger, but you can't see what happened after the trigger. To make this possible, you tell the PowerProbe to delay the break of the recording after a trigger has been detected. This TriggerDelay (**TDelay**) can be set in the advanced part of the **Probe** window:

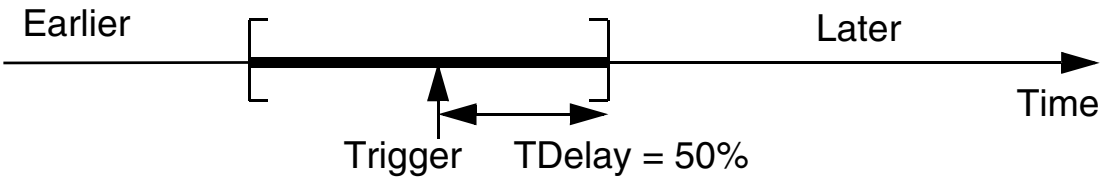


Probe.TDelay *<percentage>* Specify trigger delay

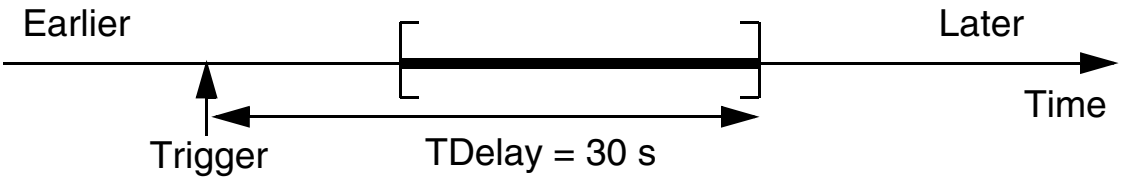
As soon as the trigger is detected, the PowerProbe will switch into the **TRIGGER** state. While the PowerProbe is in the **TRIGGER** state, the TriggerDelay starts to count down, until it reaches zero. When the TriggerDelay reaches zero, the PowerProbe enters the **Break** state and the recording is stopped:



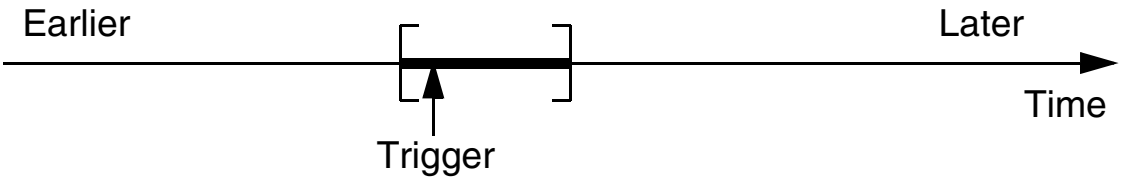
You can specify the TriggerDelay in two different units: Either you specify the TriggerDelay as an absolute time constant, or you specify the TriggerDelay as a percentage of the number of trace records the trace memory can hold. As mentioned before, the PowerProbe usually records in transient mode. So when you specify the TriggerDelay as a percentage, you can't tell how many nanoseconds the PowerProbe will continue to record data, because you don't know how many records are recorded per second:



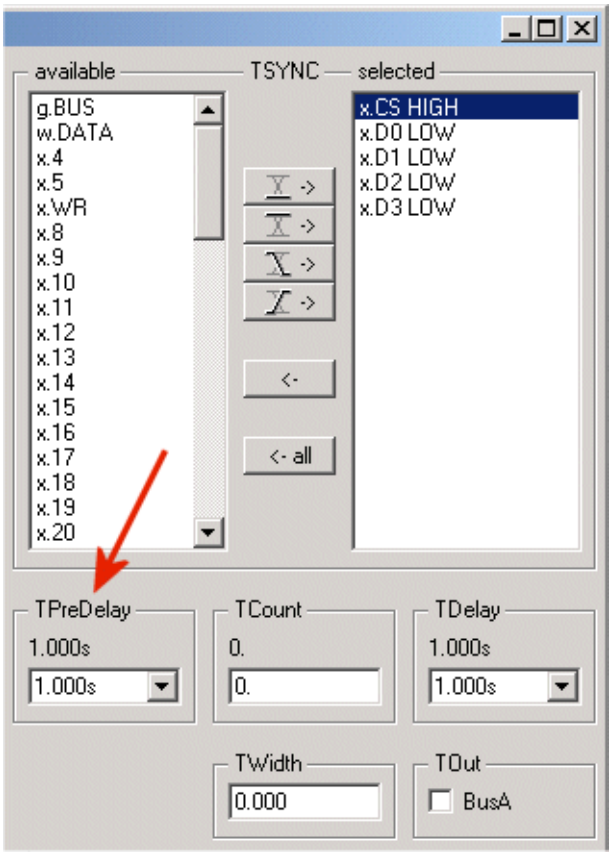
If you specify the TriggerDelay as an absolute time constant, you can't tell how many records the PowerProbe will record before the recording is breaked. If you use a very long TriggerDelay, the point of time on which the PowerProbe triggered can slide out of the record window; in this case all records which you can see in the PowerProbe trace memory will refer to a point of time after the trigger happened:



If an event happens repeatedly, it is possible that the trigger activates immediately after the trace starts recording:

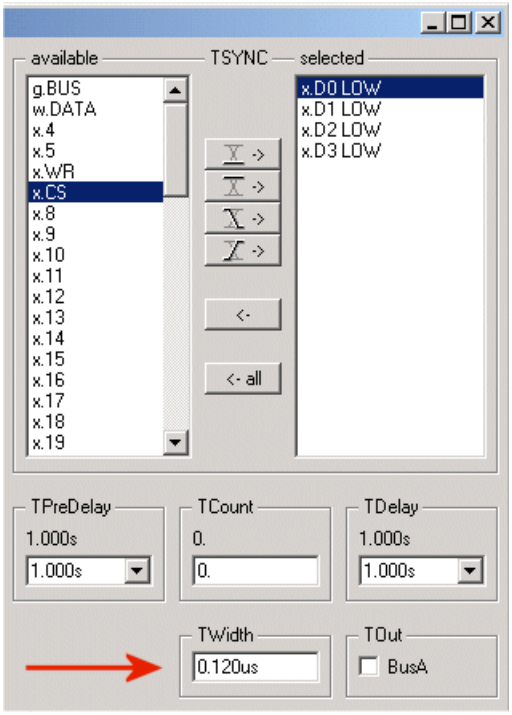


If you want to avoid this scenario, you can tell the PowerProbe how long the recording has to be active, before the trigger detection is activated. This setting is called TriggerPreDelay (**TPreDelay**) and can be changed in the trigger window:



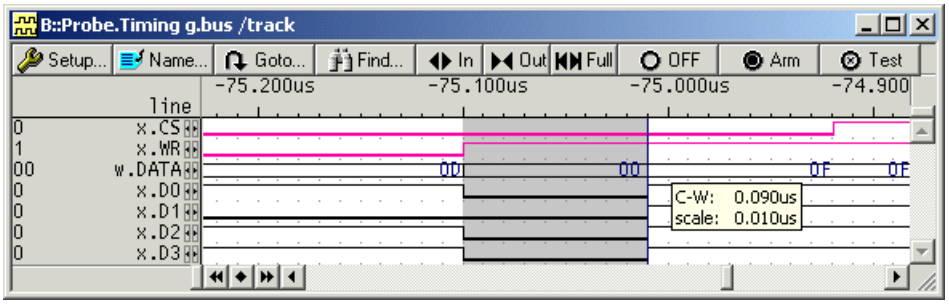
Probe.TPreDelay *<percentage>*	Specify how much of the trace buffer should be filled before the trigger becomes active

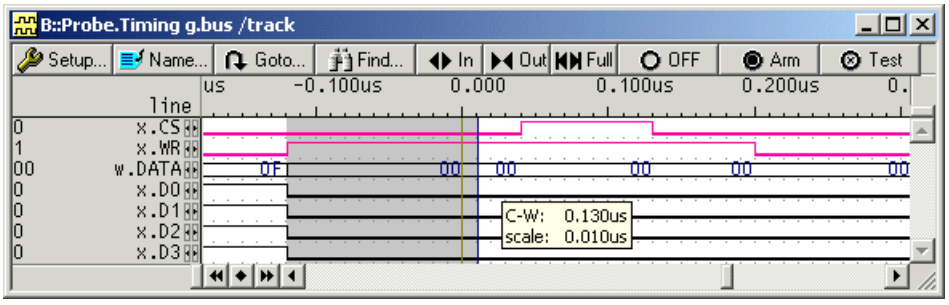
The input signals, which are recorded by the PowerProbe, will have some skew. That means that even signals, which should change on exactly the same moment won't change exactly in the same moment. So if for example a data bus changes its value, it's possible that the PowerProbe "sees" this change. In this case you get a record in which the recorded data is from an in-between state. If you want to trigger on such a signal source, you don't want this unstable states to interfere with your trigger detection. The solution is the **TWidth** setting in the trigger window: It defines how long a trigger state has to be stable, before a trigger is detected:



Probe.TWidth *<time>*	Define how long a trigger state has to be stable, before the trigger is detected.

In the first case the trigger condition was ignored, because *w.Data* was zero for only 90 ns.
In the second case the trigger was activated after *w.Data* stayed zero for 120 ns:

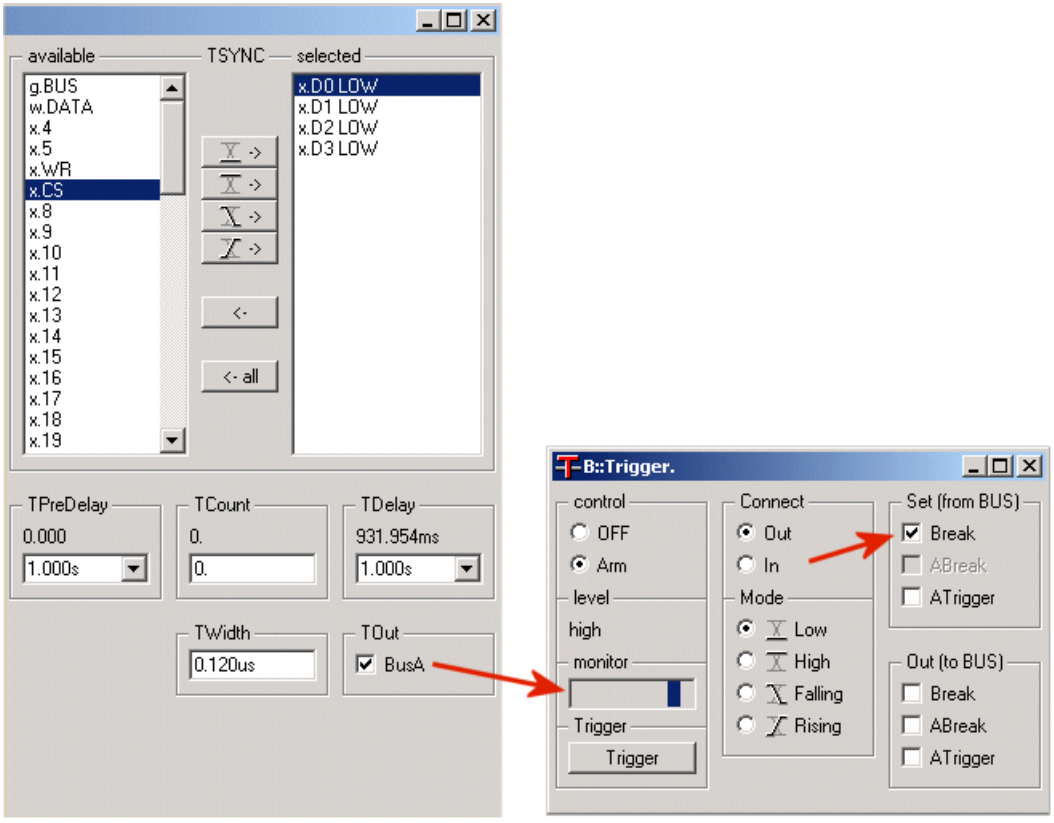




The remaining two unexplained settings are **TCount** and the **BusA** check box in the **TOut** box.
With **TCount** (TriggerCount) you can define how often the trigger condition has to occur, before the trigger is activated.

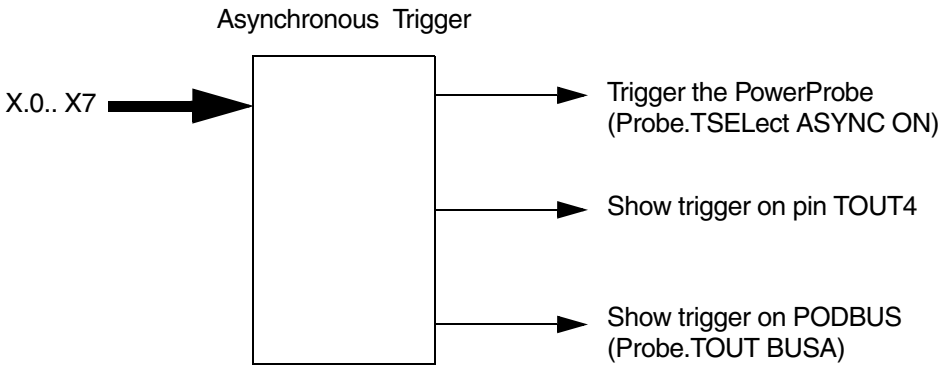
Probe.TCount *<number>*	Specify trigger counter

When the **BusA** check box is enabled, the PowerProbe will send out a trigger on the PodBus, as soon as the trigger is activated. This PodBusTrigger can be used for example to break the application program:

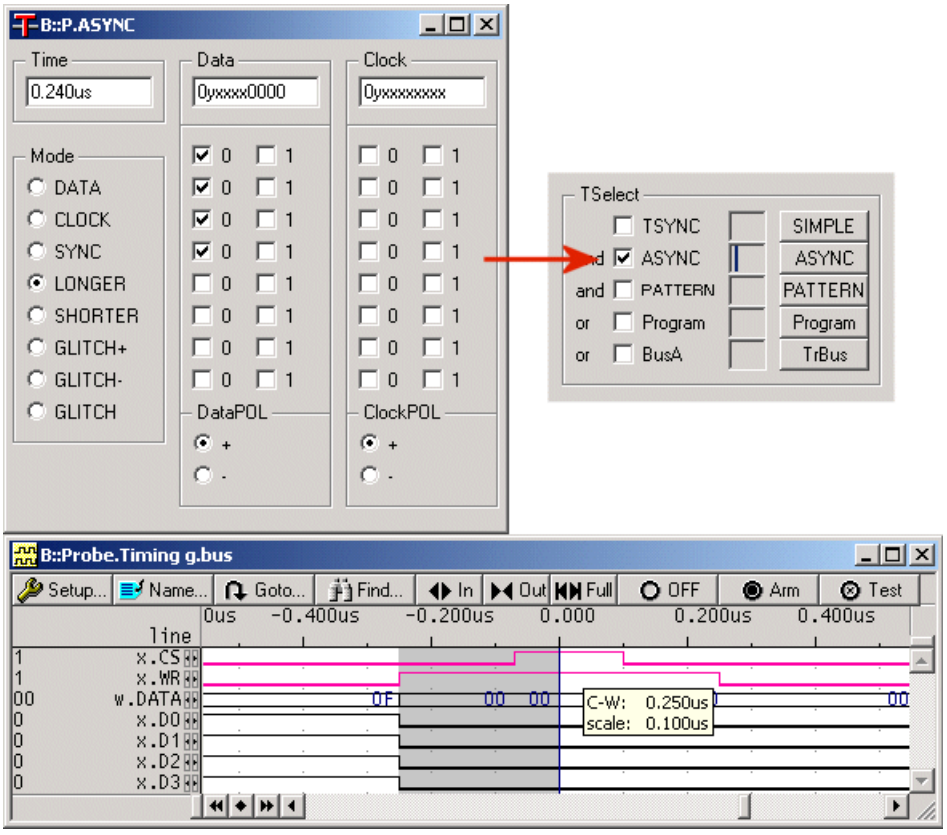



```
Probe.TOut BUSA ON                      ; output trigger to POBDBUS trigger
                                        ; bus

TrBus.Set Break ON                      ; connect the PODBUS trigger bus to
                                        ; the program execution
```

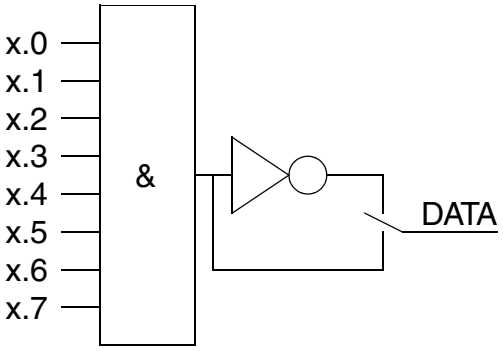
Asynchronous Trigger

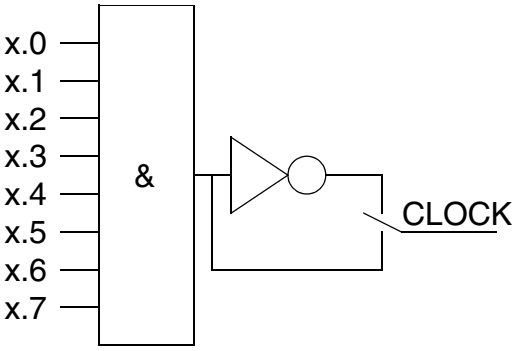


Additional to the Simple Trigger, there is also a so-called Asynchronous Trigger. It can be configured by clicking on the **ASYNC** button in the advanced part of the **Probe** window:



The input pins for this trigger are **x.0-x.7**. Internally the Async Trigger Unit generates two signals, which are called *DATA* and *CLOCK*. These signals are active, when the specified pattern is recognized on the input pins. For both signals it can be specified if they are active high or active low; this is selected with the **DataPOL** and **ClockPOL** settings. So the schematic of the Async Trigger Unit looks like this:


x.0
x.1
x.2
x.3
x.4
x.5
x.6
x.7
&
DATA


x.0
x.1
x.2
x.3
x.4
x.5
x.6
x.7
&
CLOCK

The Async Trigger Output is generated by using this two internal signals. The Mode box defines how the two signals are used. The following modes are possible:

DATA	Trigger when the **Data** pattern matches, the **Clock** pattern is ignored.
CLOCK	Trigger when **Clock** pattern matches, the **Data** pattern is ignored.
SYNC	Sample the **Data** signals with the **Clock** signal is active; trigger when the sampled **Data** pattern matches.
LONGER	Trigger when the **Data** pattern is longer active than the specified time.
SHORTER	Trigger when the **Data** pattern is shorter active than the specified time.
Glitch+	Trigger when the **Data** pattern is active for under 5 ns (positive Glitch).
Glitch-	Trigger when the **Data** pattern is inactive for under 5 ns (negative Glitch).
Glitch	Trigger when the **Data** pattern has a positive or negative Glitch

It is usually more convenient to use the Simple Trigger; but the Async Trigger has one advantage if it is used as external trigger source: It's faster (see next chapter).

```
Probe.ASYNC.view                        ; display configuration window for
                                        ; the asynchronous trigger

Probe.ASYNC.Data 0yxxxx0000             ; specifiy pattern for X.0..X.3

Probe.ASYNC.Mode LONGER                 ; an asynchronous trigger is
                                        ; generated if the pattern is
                                        ; valid LONGER then the specified
                                        ; time

Probe.ASYNC.Time 120ns

Probe.TSELect ASYNC ON                  ; allow the asynchronous trigger
                                        ; to stop the PowerProbe recording
```

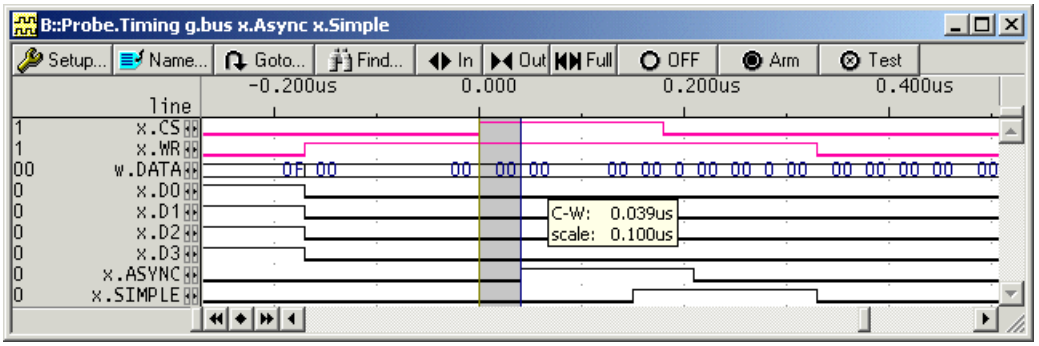
Trigger Outputs

Sometimes it's useful to generate external trigger signals. The PowerProbe has several output pins, which will become active when a trigger is detected. This output pins are all located on the middle column of the PowerProbe.

The output of the Async Trigger is directly available on output pin **TOUT4**.

The output of the Simple Trigger is available on output pins **TOUT7** and **TOUT8**. This two outputs can be controlled with the command **Probe.TOut E** and **Probe.TOut F**. For both outputs it can be selected if they are enabled, and if they are active high or low.

The advantage of the Async trigger output is that it's a lot faster than the Simple Trigger outputs:



The output pins **TOUT0** to **TOUT3** are also trigger outputs which can be activated by the Complex Trigger Unit (see chapter Complex Trigger Unit Introduction). With the commands **Probe.TOut A** to **Probe.TOut D** you can configure if these output pins are active high or active low.

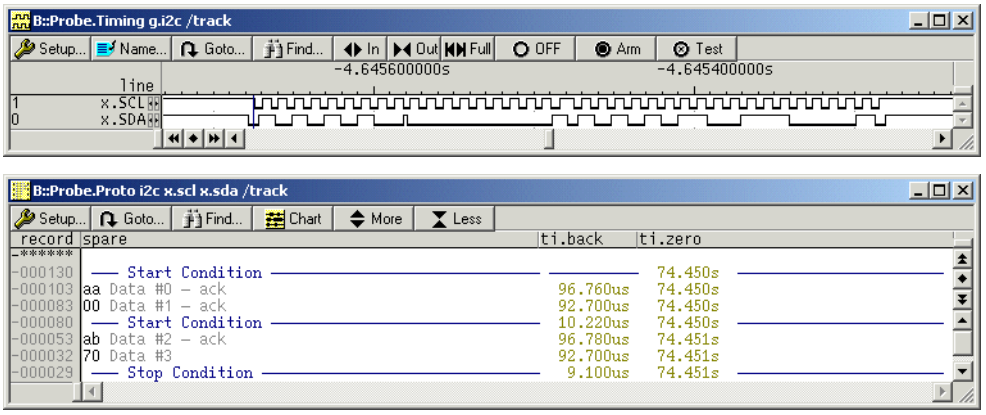
Protocol Analysis

The PowerProbe offers the possibility to analyze data which was recorded according to a given protocol. This protocol analysis can be extended by the user to support any arbitrary protocol. To do this the user has to provide a protocol specific DLL (Dynamic Link Library). If you are interested in this customized protocol analysis please refer to **"Protocol Analyzer Application Note"** (protocol_app.pdf).

For the following common serial protocols the PowerProbe already has built-in support:

JTAG	IEEE 1194.1 serial boundary scan test protocol
CAN	CAN bus protocol
USB	USB 1.1 protocol (USB 2.0 is NOT supported)
I2C	I2C two wire serial bus protocol
ASYNC	Asynchronous serial protocols (like RS232)

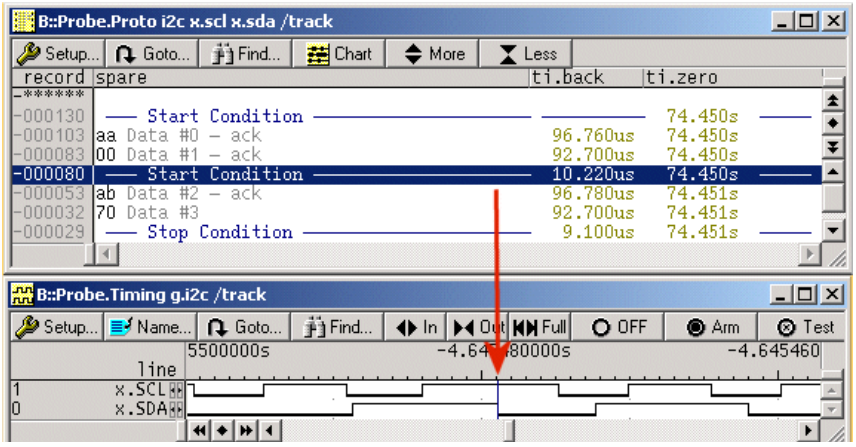
Here is an example of an I2C protocol analysis:



```
; JTAG <tck> <tms> <tdi> <tdo> <trst> <initstate>
; when the sampling is started the JTAG state machine is in state
; run-test/idle

Probe.PROTOcol.List JTAG x.8 x.9 x.4 x.12 x.14 Run-Test/Idle

; CAN <canline> <frequency> DEFault | ALL
; the frequency is defined in Hz
Probe.PROTOcol.List CAN x.7 1.MHZ DEFault

; USB <+signal> <-signal>

Probe.PROTOcol.List USB X.17 X.18

; I2C <scl> <sda>

Probe.PROTOcol.List USB X.17 X.18

; asynchronous communication interface
; ASYNC <asyline> <frequency> +|- <parity> <length> <stopbit>


Probe.PROTOcol.List ASYNC X.6 3600. + EVEN 7 1STOP STRING
Probe.PROTOcol.List ASYNC X.5 2400. - NONE 5 2STOP CHAR

; special protocols
; TRACE32 offers a API that allows to use special, customer specific
; protocols

Probe.PROTOcol.List protojtag.dll X.4 X.12 X.14

; examples for special protocols are provided in the TRACE32 system
; directory under ~~\demo\proto
```

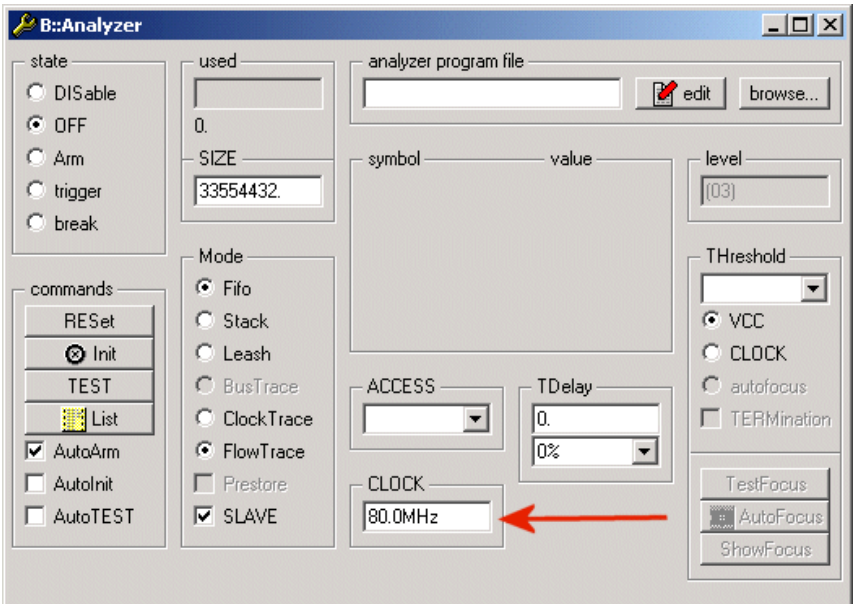
Track Option

Usually you want to correlate your windows; for example you want your timing window to be time correlated to your protocol analysis window. That means that you want to see the physical signal levels to a given analyzed protocol part. This is possible by adding the **/Track** option to the **Probe.Timing** and the **Probe.Proto** window. All windows which have been opened with the **/Track** option are time correlated; that means if you move to some point of time in a window which has timestamped data, all the windows which have the **/Track** option enabled will jump to the same point of time.

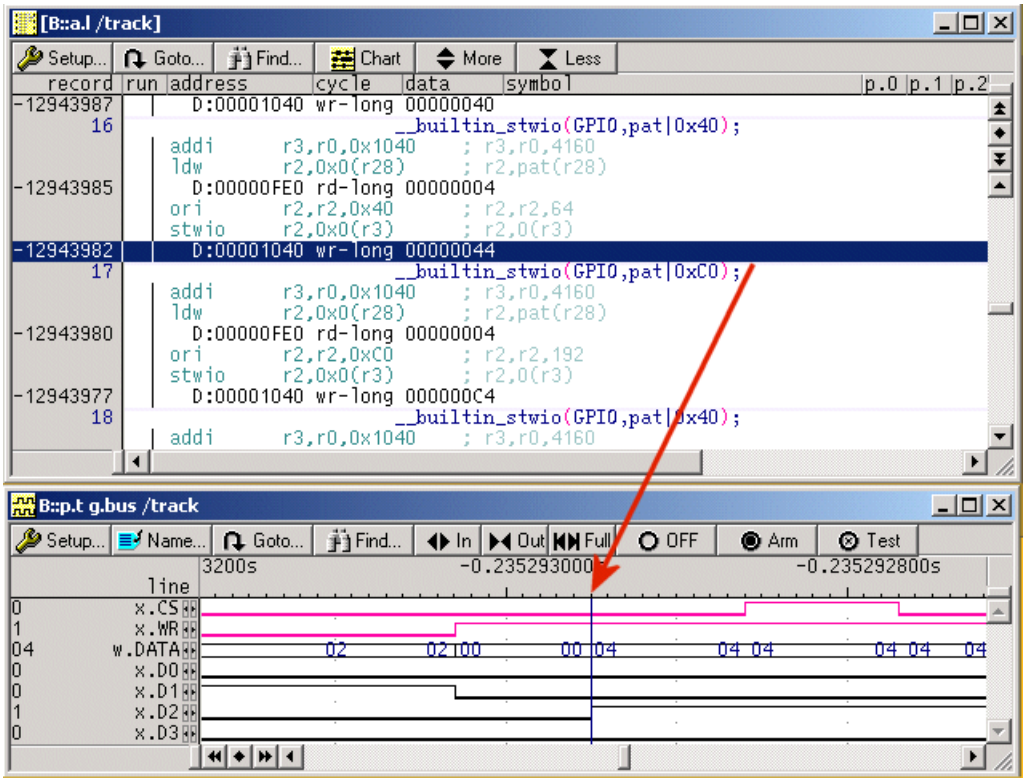


The **/Track** option also works to time correlate the trace data of the PowerProbe to the trace data of the PowerTrace. So with the **/Track** option you can correlate your program flow and data trace to the recording of your physical signals. There is some problem with this tracking: program flow and data traces are usually not very accurate and there usually is quite a big offset between the execution of an instruction and the generation of some corresponding trace data.

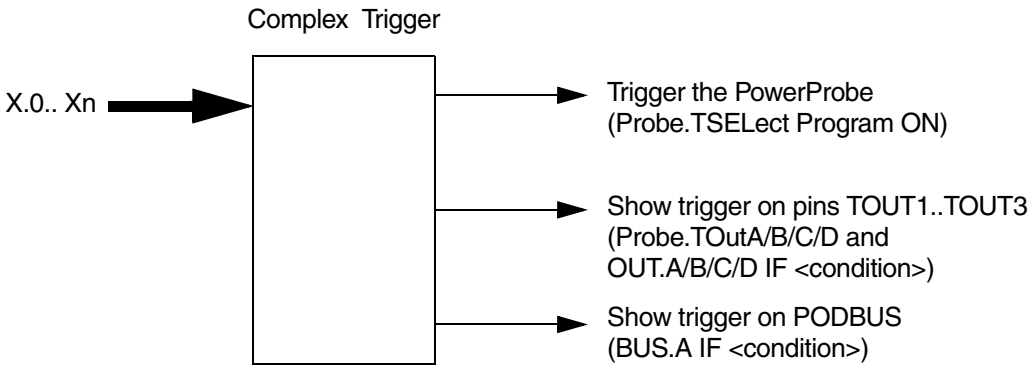
To make the time correlation as good as possible, the PowerTrace analyzer needs to know the frequency of your target, to compensate for the time offset introduced by the generation of flow or data trace. This frequency can be set in the **Analyzer** Window:



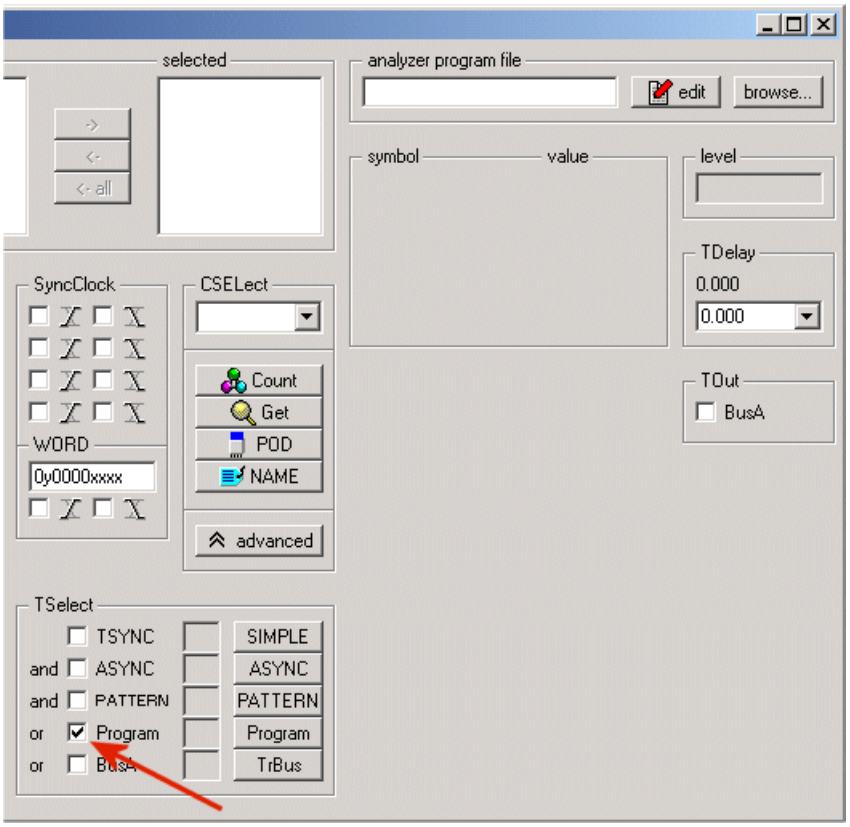
Here is an example. The write to the address 0x1040 coincides with the change of *w.Data* to 04:



Complex Trigger Introduction



Like the PowerTrace, the PowerProbe also has a Complex Trigger Unit (CTU). To use the CTU you have to activate it by selecting the **Program** Checkbox in the advanced **Probe** window:



As with the PowerTrace you have to write a Trigger Program to define the actions of the CTU.

The most important declaration for the PowerProbe is the definition of a data selector. A data selector defined a pattern on the PowerProbe input pins.

SELECTOR *<name> <pin> <value>*

Here is a simple example:

```
SELECTOR dataIsZero w.data 0 x.wr 1 x.cs 1

                        ; Define an input selector which is called
                        ; dataIsZero. This selector will become true,
                        ; when w.data==0 and x.wr==1 and x.cs==1

Trigger.TRACE IF dataIsZero

                        ; Trigger PowerProbe when dataIsZero becomes true
```

So basically a Trigger Program has two parts:

In the first part all the resources which you want to use are defined, by giving them names. (In the above example a selector is defined with the name *dataIsZero*).

In the second part you define action / condition pairs. Each action will only be executed if the corresponding condition is met. (In the above example the action *trigger.trace* is executed if *dataIsZero* is true).

In a Complex Trigger Program for the PowerProbe you can define up to 8 data selectors, so you can react on 8 different patterns. The Complex Trigger contains 3 counters, which can be used to count occurrences of conditions or to measure time. Counters can also be used in conditions: A counter will become “true” when the counter reaches a user-defined value:

```
SELECTOR dataIsF w.data 0xF x.wr 1 x.cs 1

                        ; Selector definition

EVENTCOUNTER cnt 10.    ; define counter cnt which will count up to 10

Counter.Increment cnt if dataIsF

                        ; increment cnt if selector dataIsF is true

Trigger.TRACE IF cnt    ; Trigger PowerProbe when counter cnt reaches
                        ; limit.
```

Both of the above examples can also be handled with the Simple Trigger. Here is another example which cannot be handled with the Simple Trigger:

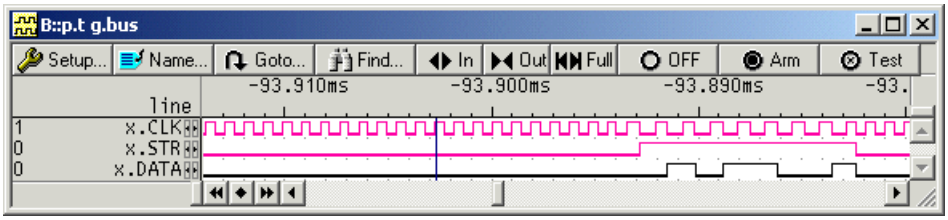
```
SELECTOR dataIsF W.Data 0xF x.wr 1 x.cs 1

                        ; Selector definitions

SELECTOR dataIsA W.Data 0xA x.wr 1 x.cs 1

EVENTCOUNTER cnt 10.    ; counter definition

Counter.Increment cnt IF dataIsF

Trigger.TRACE IF cnt&&dataIsA

                        ; Trigger PowerProbe when counter reaches limit
                        ; and selector dataIsA becomes true.
```

There is also the possibility to stimulate external trigger outputs with the CTU. With the complex trigger program commands *OUT.A* to *OUT.D* you can activate the output pins **TOUT0** to **TOUT3**, which are located on the middle pin column of the PowerProbe. With the commandline commands **Probe.Tout A** to **Probe.Tout D** you can select if the output pins are active high or active low.

One of the most useful features of the CTU is the ability to act as a trace filter. It is a common case, that one of the signals you are recording is a clock signal. In that case the trace memory of the PowerProbe will rapidly fill up, because a clock is always transient. (As you may remember: The PowerProbe records signal changes in transient mode.)

With the CTU it is possible to filter out clock cycles in which nothing interesting is happening. With such a filter you can use the memory of the PowerProbe much more efficiently and thus the real trace depth grows a lot.

Here is an example of such a filter application:
Assume that we want to trace a simple serial protocol, which has a free-running clock. To indicate that a data transfer is in progress, there is a strobe signal, which is high when data is transferred. So the traced data will look like this:

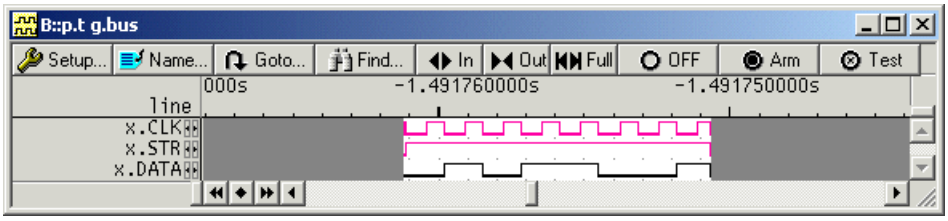


As you can see there are many clock transitions on which no data is transferred.

Now we use the following trigger program:

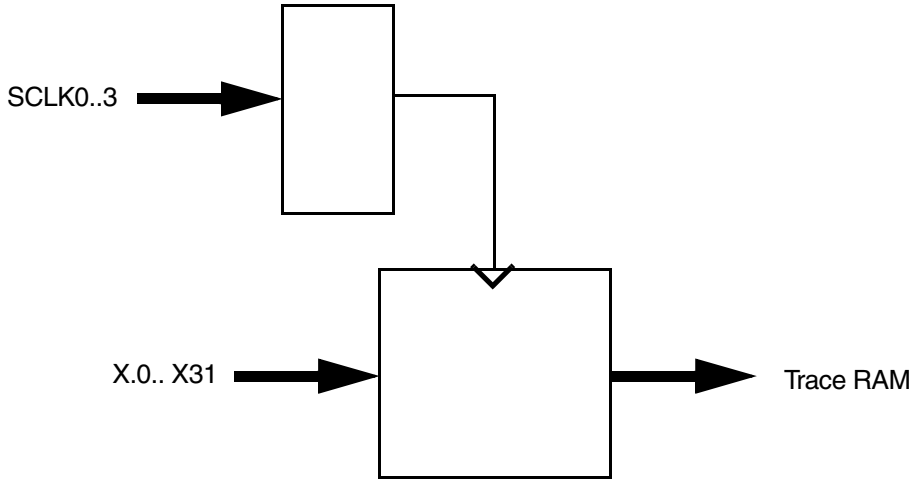
```
SELECTOR shiftIsActive x.STR 1    ; Selector definitions


Sample.Enable IF shiftIsActive    ; Only enable tracing if the
                                  ; shiftIsActive selector is true
```

With this CTU program, the PowerProbe will only store trace data if the **x.STR** signal is high. So all the idle clock cycles are filtered out. The result looks like this:



As you can see, the recording is only enabled while the **x.STR** signal is high.

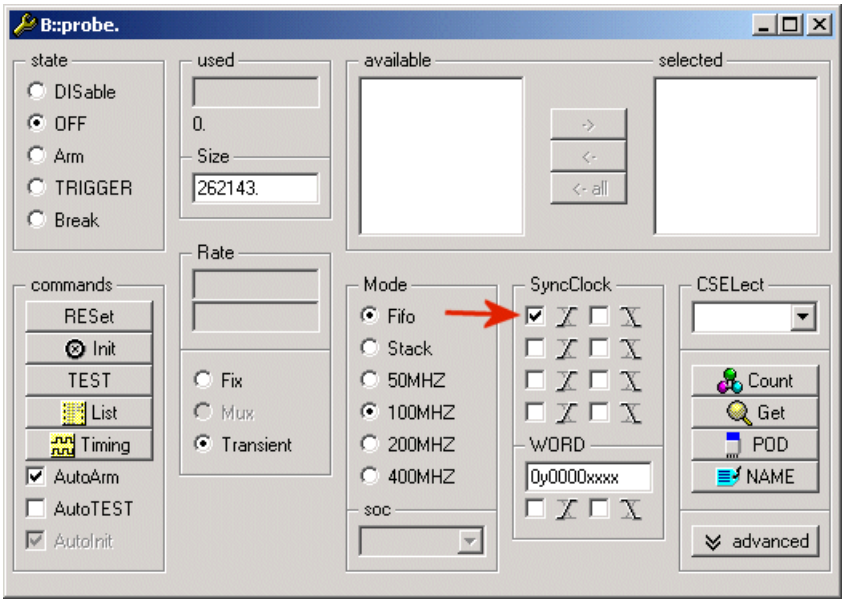
Synchronous Recording



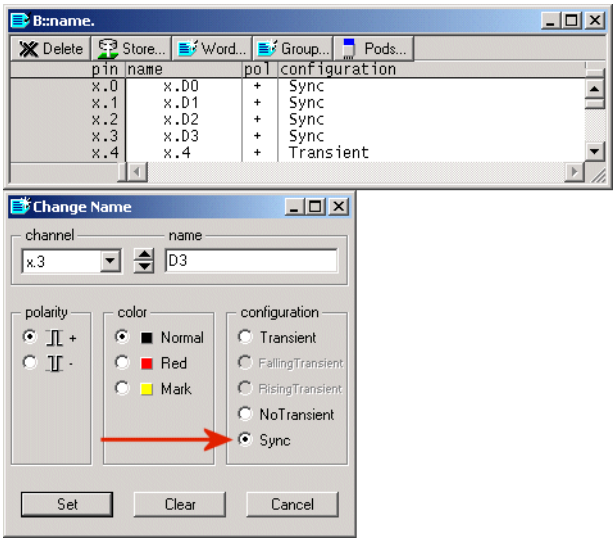
Sampling with SCLKx

The PowerProbe also offers the possibility to sample your data with an external clock. This clock has to be connected to one of the **SCLK0**,**SCLK1**,**SCLK2** or **SCLK3** pins on the middle pin column of the PowerProbe. Only input pins x.0 - x.31 offer synchronous sampling, so you have to connect your data signals to one of those input pins. In the **Probe** window you can select in the **SyncClock** box which input clock pin is used and if you want to sample your data on the rising or falling edge of your clock:

```
Probe.SyncClock SCLK0 Rising
```

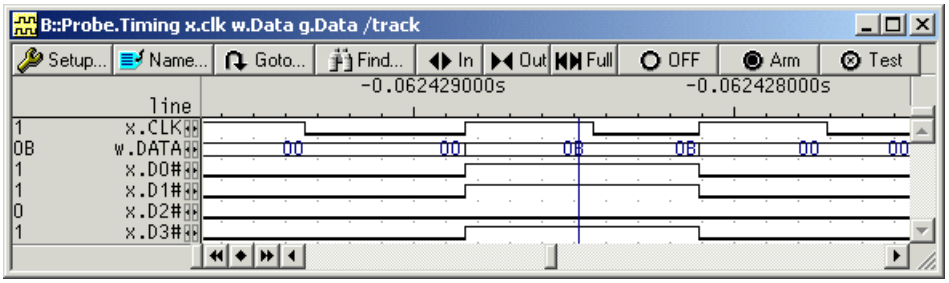


The next step is to configure your input pins to be recorded synchronously with the selected clock. This is done via the **Name** window. You can choose for all pins from x.0 to x.31 if the pin is recorded synchronously or not:




```
NAME.Set X.4 X.D3 + Sync
```

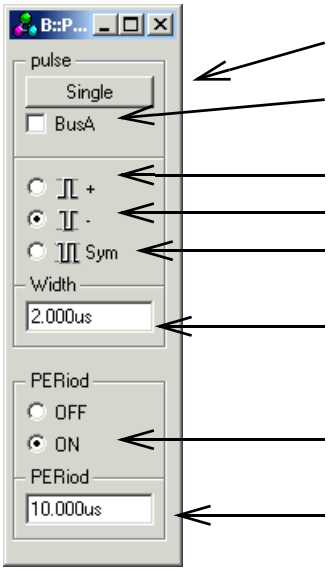
Now your recording will look like this:



Because the data is sampled on the rising edge of x.CLK the x.D inputs change on the rising edge of the sampling clock. The PowerProbe will use one trace record for each external clock cycle. So when you sample synchronously, the memory of the PowerProbe will fill up rapidly. If you have idle clock cycles (clock cycles which don't transfer meaningful data), you can use the CTU to increase your record length dramatically by filtering out these idle cycles.

Pulse Generator

If you need to generate a pulse for example to stimulate some behavior of your application, you can use the Pulse Generator which is integrated in the PowerProbe. The Pulse Generator has an output level of 3.3V. It can either generate Pulses on request, or it can send out a continuos PWM (pulse-width modulation) clock signal. To open up the control window of the Pulse Generator you have to enter the command **Pulse**:



Click on this button to generate a single Pulse

BusA means: Generate a single Pulse, when a PodBus Trigger is received.

High Active Pulse

Low Active Pulse

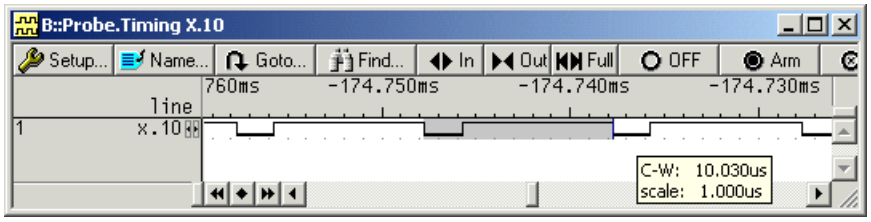
Only for periodic signal: Use 50/50 duty cycle

Pulse Width. (Width of active period.)

Generate periodic signal

Period length, for periodic signal

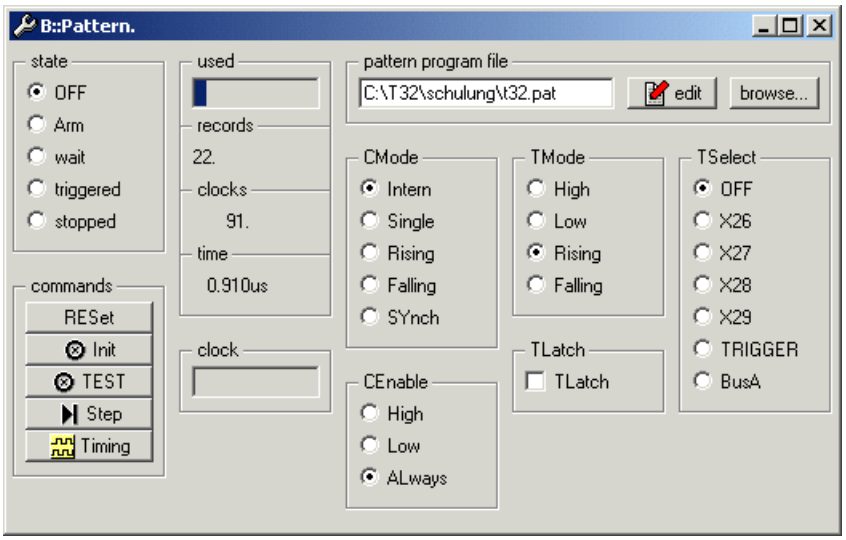
Pulse, generated by this settings



The output pin of the Pulse Generator is on the **middle** column of the PowerProbe and is labelled **TOUT5.**

Pattern Generator

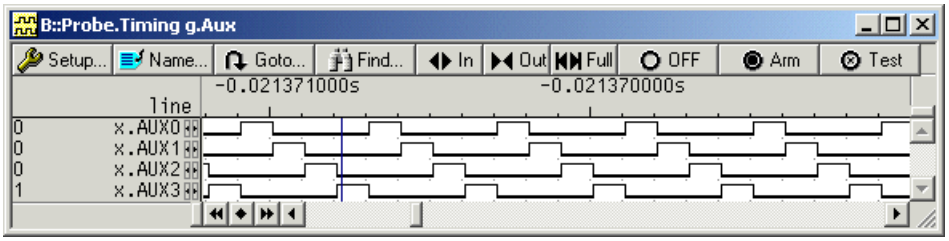
If you want to generate more complex outputs, you can use the built-in Pattern Generator of the PowerProbe. To use the Pattern Generator you first of all have to open up the control window by entering the command **Pattern** into the command line:



The Pattern generator has to be programmed via a pattern program file. Here is an example of such a file:

```
set 0x001                ; Set a pattern for output pins (takes 10 ns)
delay 90ns               ; Wait for 90ns (takes 90 ns)
set 0x002
delay 90ns               …
set 0x004
delay 90ns
set 0x008
delay 60ns               ; Wait for 60ns (takes 60 ns)
restart                  ; Go back to start (takes 30 ns)
```

The Pattern Generator controls the output pins **AUX0**-**AUX8**. This pins are locate on the middle pin column of the PowerProbe. The above program produces the following pattern on the pins **AUX0**-**AUX3**:



To repeat a pattern sequence, you can use the **RePeaT** command in your pattern program:

```
RePeaT 5.               ; Repeat the next block 5 times.
(
     set 0x001
     delay 90ns
     set 0x000
     delay 90ns
)
```

The pattern generator uses an internal 100 Mhz clock as default. If you want to use an external clock, you have to connect this external clock to input pin x.24. Additionally you have to select in the **CMode** box, if you want to output the data on the **rising** or **falling** edge of the external clock.

You can also provide an external clock enable signal for your external clock by connecting the enable signal to input pin x.25. If you want to use such an external clock enable pin, you have to select in the **CEnable** box, if the enable signal is active **high** or active **low**.

The second method to use an external clock is to connect the external clock to one of the SCLK0, SCLK1, SCLK2 or SCLK3 pins on the middle row, and then configure a synchronous clock in the **Probe** Window (see Synchronous Recording chapter). For this method you have to select **SYnch** in the **CMode** box.

The last clocking option is to single step your pattern. For this method you have to select **single** in the **CMode** box. If you start your pattern generator, you can then use the **Step** button to send a single clock cycle to the Pattern Generator.

The Pattern Generator also offers the possibility to wait on a trigger signal. This is done by using a *wait* commando in your pattern program:

```
set 0x001
wait                    ; The wait commando waits until a trigger is
set 0x002               ; received.
wait
set 0x004
wait
set 0x008
wait
restart
```

In the **TSelect** box, you have to select which trigger source is used. You can use the input pins X.26--X.29 as input trigger pins, or you can use the Complex Trigger as trigger source, or you can use the PodBus Trigger.

If you want to use an input pin as trigger source, then you have to select in the **TMode** box, if you want to trigger on **high**, **low**, **rising** edge or **falling** edge of the input signal.

If the **TLatch** checkbox is selected, the trigger will be held active until the Pattern Generator reaches a *wait* statement. The *wait* statement will then clear the trigger. Without the **TLatch** checkbox the Pattern Generator will ignore triggers when no *wait* statement is active.