

The PsyScope USB Bbox module: a Software manual.

Second Draft, March 2007

Written by Luca L. Bonatti and Luca Filippin

Laboratory of Cognition and Development, Sissa/Isas, Trieste

Introduction	4
Installation	4
Quick Start	5
The Menus relevant to the USB Bbox	5
<i>Input Devices and the Inputs Window</i>	6
<i>Don't Use USBBbox</i>	6
<i>Reference Timer</i>	7
The Usb Bbox Stuff Menu item and its window	8
<i>The Mask</i>	10
<i>Port Direction and Logic</i>	12
<i>The Start Test & Configure Mode</i>	13
<i>Time Drift Estimation</i>	14
<i>Send Serial Out</i>	14
<i>The Voice Key Parameters</i>	16
The Delayed Trigger Voice Key and its settings	16
<i>The Minimum signal duration and Minimum silence duration parameters</i>	17
<i>Trigger Threshold</i>	20
<i>Primary and Secondary Gain</i>	20
<i>The peak level</i>	21
The Voice Key Tuning button and the DTVKTuner program	22
<i>Starting the DTVKtuner</i>	22
<i>Recording sound and trigger data to computer</i>	23
<i>Auto vs manual tuning procedure</i>	24
<i>The Manual Procedure</i>	24
<i>The Automatic Procedure</i>	24
<i>Tracking and inspecting the triggers</i>	26
<i>Ending the tuning procedure and going back to PsyScope.</i>	27
Designing your experiment: The USBBbox condition	27
<i>The Default Condition</i>	28
<i>Selecting restricted conditions</i>	28
<i>Selecting conditions depending on other input ports</i>	30
<i>The Serial In condition</i>	30
<i>Serialin special conditions</i>	32
<i>Conditions on other input ports</i>	32
Designing your experiment: The USBBboxDo action 34	

<i>MSKSET</i>	35
<i>P0SET, P2SET, PXSET</i>	35
<i>P0OR, P2OR, PXOR</i>	36
<i>P0AND, P2AND, PXAND</i>	36
<i>DIRSET</i>	36
<i>LOGSET</i>	37
<i>SEROUT</i>	37
<i>VCKSET</i>	37
The USBBox event type	38
Designing your experiment: Other initialization parameters	39
<i>The debounce period</i>	40
<i>Port Logic</i>	41
<i>Port Direction</i>	41
<i>Port2 Loopback Mode</i>	41
<i>Serial data parking</i>	42
Designing your experiment: Setting variables with Serialin data and USBBOX expressions	42
<i>GetKey</i>	43
<i>GetMsk</i>	43
<i>GetRTC</i>	43
<i>GetLog and GetDir</i>	44
<i>VCKGET</i>	44
<i>Composing USBBox predicates inside expressions</i>	45
<i>The SERIALIN assignment</i>	46
Analyzing your experiment: The Data File	46
<i>UBButtons</i>	47
<i>UBPorts</i>	47
<i>UBDrift</i>	47
<i>UBVoice and UBOptic</i>	48
<i>UBQueueLength</i>	48
<i>UBRelativeTS, UBAbsoluteTS, and UBSystemTS</i>	48
<i>Other Values</i>	50
<i>A final point: Reference timer and time values</i>	50
Miscellaneous issues	50

Introduction

PsyScope X offers support for an USB button box of novel conception, designed by ioLab (<http://www.iolab.co.uk/>). ioLab did the hardware, and we are not responsible for it. The Sissa LCD Lab wrote the software (main programmer: Luca Filippin).

ioLab will explain the hardware in a separate document. Here we will only refer to the minimum hardware details needed to explain how to program it for PsyScope X. However, if you want to understand how to use the USBBox correctly, you will need to understand some of the basics of its hardware.

An updated PDF version of this manual can be always downloaded [here](#).

Installation

We prefer that people know what and where software elements are installed, so we decided that you have to install the drivers manually.

You have to move the BBoxArchiver.kext, which you will find in the PsyScope X packages, into the /System/Library/Extensions folder. You have to have administrative privileges in order to do that. After restart, you should see the USBBox when you plug it into any USB port. No other operation is necessary.

You will need PsyScope X, at least version B36.

If you want to uninstall the USBBox software, simply trash the BBoxArchiver.kext.

Quick Start

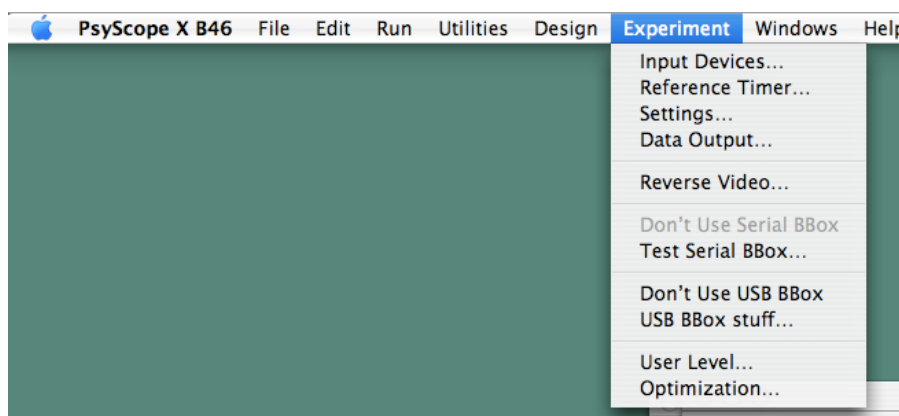
If you have no special needs, and you only want to receive input from the USB Bbox buttons, you only have to do two things

- [Enable the Usb Bbox as an Input Device](#)
- Write [Condition- Action](#) pairs in your events in which the UsbBbox is specified as a termination condition for the events.

If that's all what you want to do, you can stop here and start playing with the USB Bbox. Hopefully, the basic stuff should be easy to understand for everybody who has limited experience with PsyScope and/or its old serial CMU bbox. If you need or want to understand the functioning of the UsbBbox more in details, continue reading.

The Menus relevant to the USB Bbox

The first thing you have to do is to open a script. Without it, you cannot do anything with the USB Bbox. When a script is open, you will see three relevant entries under the "Experiment" menu:

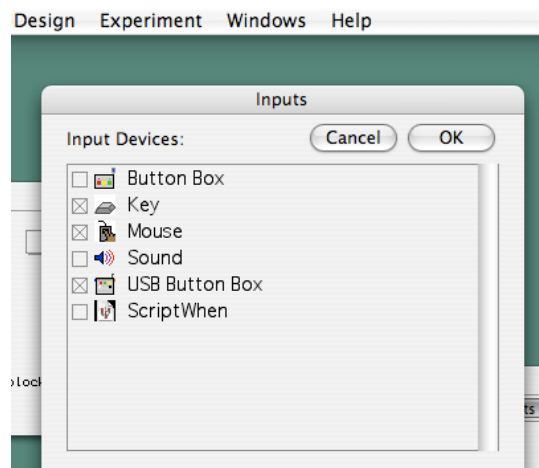


They are:

- The *USB BBox Stuff...* menu entry.
- The *Don't Use USB Bbox* menu entry.

- The *Reference Timer...* menu entry.

A fourth entry can be seen by choosing *Input Devices...*, which will make the following window appear:



We will take these windows in turn.

Input Devices and the Inputs Window

Let's start from the easy part. The *Inputs* window serves to enable the USBbbox as an input device. Without this feature checked, the USBbbox will be ignored at runtime. This works as the previous Serial Bbox, if you are familiar with it.

Don't Use USBbbox

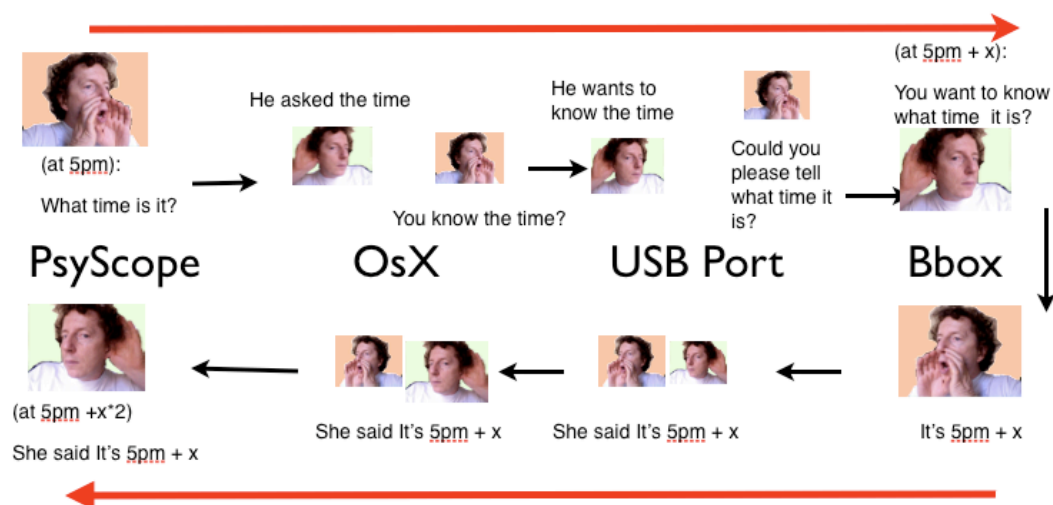
The *Don't Use USBbbox* window allows you to entirely design and run an experiment by ignoring all what concerns the USBbbox. In this way, you may prepare an experiment by saying, for example, that a certain events ends when something happens on the USBbbox OR on the keyboard, and the experiment will run in any case. This allows you to 1. write an experiment for the USBbbox without the USBbbox present, and 2. possibly write an experiment designed for various input and output devices, including the USBbbox, that will run in other ways when the USBbbox is not available. It's good practice to do that.

In order to "reactivate" the USBBbox for the current script, you have to select the UsbBbox as input device.

Reference Timer

The USBBbox has a high precision internal timer that timestamps every event occurring in it. The Mac also has an internal timer. Which one should you use? Which one do we use?

Well, the right answer is: we use both. Here one needs to begin to understand some issues connected to the hardware. The USB port is very inefficient for fast reliable communication. In particular, communicating through the USB is limited by the speed at which the USB port allows communication to enter and to exit the computer. There are other passages, besides the bottleneck of the USB speed, and other intermediate passage could also add uncertainty. Minimally, the flow of a time request would involve the following passages (assuming that the clocks are perfectly synchronized, and that communication direction is immaterial, which is not the case):



Clearly, this is very inefficient. Because time requests in PsyScope occur very frequently, it is just impossible to base all the functioning of PsyScope on the

USBBbox external clock. So we don't do it. (Incidentally, even the old CMU Bbox was not doing it).

What is possible is to use the external timer as a reference timer. This is what the "Reference timer" menu allows you to do. If you select the internal timer, then all the time values will be those of the internal timer, but when an event occurs in the USBBbox, and comes in with its own timestamp, the timestamp will be reported to the internal clock's time. Vice-versa, if the chosen reference timer is the USBBbox timer, the timestamp information of the events will be left untouched, and all other time information recorded in the data file will be reported to the USBBbox clock. However, for any internal operations, in both cases, PsyScope keeps consulting the computer's internal clock, as this is the most efficient way to fix the time for preparing and scheduling the events and the actions occurring during an experiment.

A final issue has to do with clock synchronization. As the two clocks are never perfectly synchronized, just consulting the two clocks at the beginning of the experiment and doing the relevant adjustments is not enough (if you want very good precision across a long experiment). So we add a way to estimate the time drift for the two clocks, as we will explain below.

The *Usb Bbox Stuff* Menu item and its window

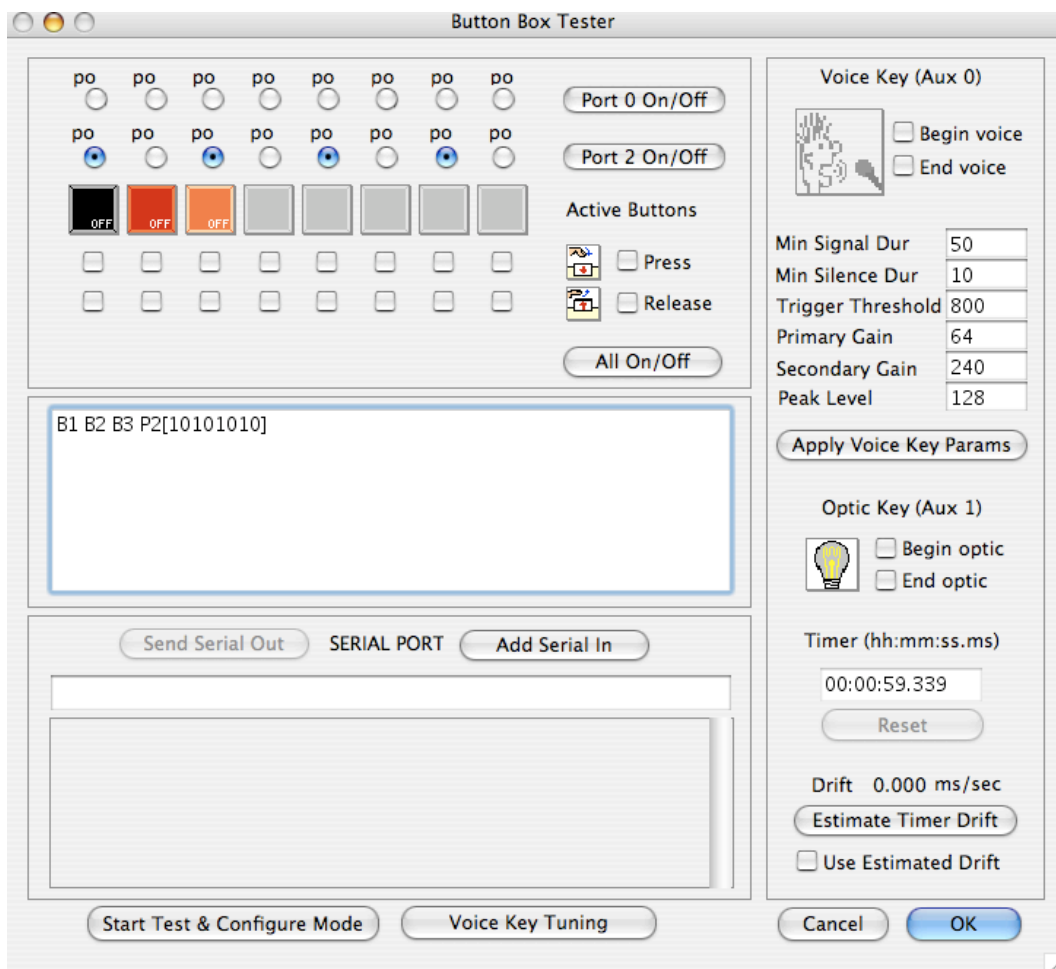
Briefly, the USBBbox is made of the following devices:

- Eight response buttons (which are really a port, Port1), working in "OR" mode. By default, buttons are all on.
- Two eight-bit ports (Port0 and Port2), which can be configured in their directions (as input or output ports) and in their logic (positive or negative). One of these ports, Port2, is connected to a row of LEDs, while the other can be directly accessed via one of the jacks on the back of the USBBbox. By default, both ports are configured with positive logic and as output devices. By default, these ports are off.
- A serial port, that can send and receive via serial communication. The serial port is configured as 9600 baud, 8 Data bit, No parity, 1 stop bit. You cannot change these parameters.
- Two AUX ports. By default, one is connected to the voice key circuit; The other

will be connected with an optic line, but for the moment is a generic input line. Both ports can be configured as interrupt trigger input ports.

- A sound in and a sound out jacks for a microphone or other sound input, and sound pass-through for recording the sound input via the USBbox.

When you select the *Usb BBbox Stuff...* Menu, you open a very rich panel, which allows you to configure all these devices, as well as to test them.



Let us take a look at the functions of the panel. A first function is to allow the user to define a mask. By default, a mask is already defined, so for simple experiments defining a mask is not necessary. However, it is important to understand what a mask is before continuing.

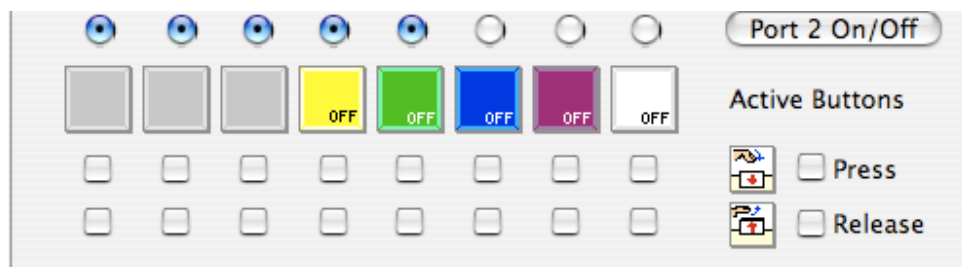
The Mask

A mask is used to either make active or, well, mask any of the subcomponents of the USBBbox.

In the default configuration, the USBBbox has all the buttons and all the LEDs active. Every other configuration must be set by defining a mask. A mask is tied to a script: the mask definition is saved directly in the *ExperimentDefinition* section of the script, which is what you see right at the top of it. If no mask is specified in the script, the program will assume that the default mask must be used.

Masked components behave as if they did not exist during the execution of an experiment. Any action on them is ignored by the USBBbox driver and never reaches the program. That is, if you have a condition such as "Press any button" (`UsbBbox[Any]` in the script) but, say, Button 2 is masked, nothing will happen if you press Button 2. In this way, you can easily customize an experiment by programming it in the most generic way, and then restricting the active buttons or devices via a mask.

You can write a mask definition directly in the script, or use the USBBbox panel to do it. In order to mask the devices via the panel, you click directly on their graphical representation in the panel. For example, let us suppose that we want to mask the three leftmost buttons and the three rightmost LEDs. You will click on the buttons and the LEDs so as to obtain the following configuration:

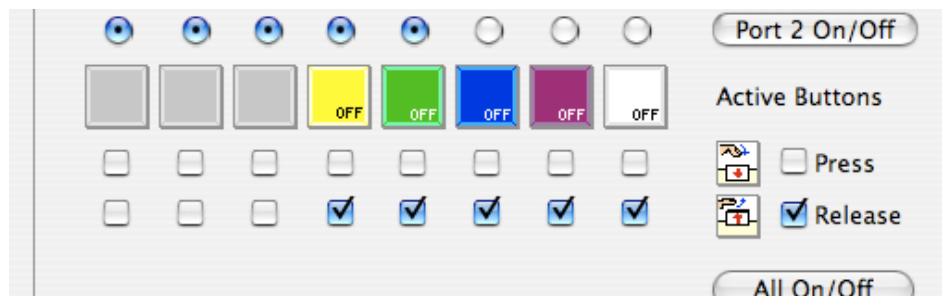


When you click the "OK" button, the program will write the following line in the *ExperimentDefinitions* section of the script:

```
USBBBoxInitialSettings: B4 B5 B6 B7 B8 "P2[11111000]"
```

and this configuration will be loaded at runtime, as well as the next time you open the USBBbox panel.

Notice that you also have some "Press" and "Release" options for the buttons. *They are not relevant* for a mask: you cannot "mask" a button only when it is pressed, or released. Nevertheless, should you make a mistake and use the "Press" and "Release" options in order to define a mask, the program will correct you and write only the relevant mask information in the *ExperimentDefinition* section. That is, should you happen to act directly on these options and obtain a graphical configuration such as this:



the program will still code that buttons 4-8 are active, *not* that they are active only when you release them. The initial settings for the script will still be:

```
USBBBoxInitialSettings: B4 B5 B6 B7 B8 "P2[11111000]"
```

Thus,

"Press" and "Release" options are irrelevant for mask definitions, like "Begin Voice" and "End Voice" or "Begin Optic" and "End Optic". Likewise, the "OFF" indication written on the buttons is irrelevant: "OFF" only signals that the button is not receiving input at the moment, while the color signals that the button is active and could receive input.

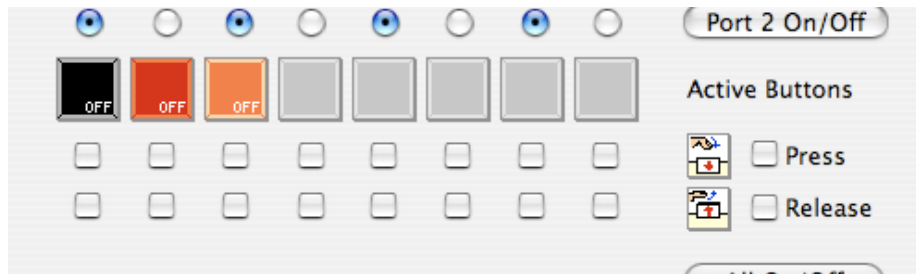
In order to avoid conflicts of USBBox initialization, when you define a mask via the USBBox Panel, you cannot directly edit the script. However, if, with the panel closed, you edit the script directly, for example by changing the current initial settings from

```
USBBBoxInitialSettings: B4 B5 B6 B7 B8 "P2[11111000]"
```

to

```
USBBBoxInitialSettings: B1 B2 B3 "P2[10101010]"
```

then next time you open the panel, these changes will be reflected in it, as well as in the USBBox directly. Thus, with this line saved in the initial settings, you will see this picture:

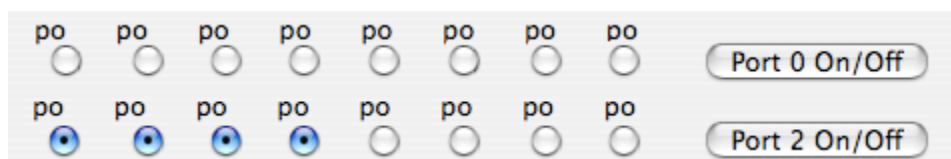


and you should see the corresponding LEDs 1, 3, 5, and 7 on in the USBBox.

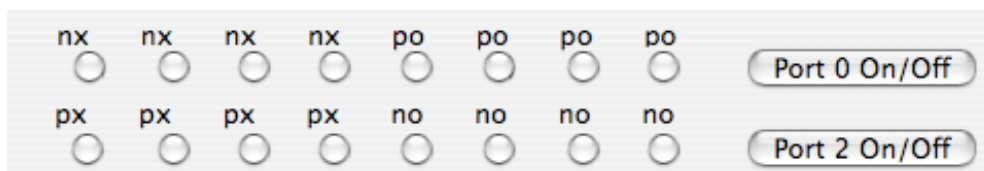
You mask or activate any other part of the USBBox in the same way. Thus, if you want to make the voice key active, press the voice key icon, and so on.

Port Direction and Logic

Before going on, we need to spend a couple of words about the funny characters on top of the radio buttons for Port 0 and Port 2:



As both the direction of the ports and their logic can be changed, we exposed these configurations on top of each radio button. As we were lazy, and didn't want to spend too much time on the graphical part, we did it that way. If you don't know what logic and direction are, then you can disregard them entirely. If you do, remember that *p* stands for *positive*, *n* stands for *negative*, *o* stands for output, and *x* stands for input. Thus in the above configuration, all ports are configured as output ports with positive logic. Instead, in the configuration below:



the first four lines of Port0 are input lines in negative logic and the other four are (normal) positive output lines, whereas the first four lines of Port2 are positive input lines and the other four are negative output lines.

Port logic and direction can only be changed by modifying the script directly, and are not configurable through the graphical interface. There are odd aspects of this topic that are not immediately intuitive, and you may have better things to do right now. So the best is if you forget what you just wrote and ignore the little letters on top of the ports, at least for now. But if you are really taken by the topic, the relevant syntax is explained [below](#).

Now we got close to the *Start Test & Configure Mode* button and we may as well take a look at what it does.

The Start Test & Configure Mode

On the bottom left part of the panel, you will find two buttons: the *Start Test & Configure Mode* and the *Voice Key Tuning* buttons.

For reasons tied to the hardware, you cannot be in *Test & Configure Mode* AND tune the voice key at the same time. If you are in *Test & Configure* mode and you press the *Configure Voice Key* button, the *Test & Configure* mode will automatically end and you will get access to the Voice Key Tuning facilities. When you are done, you will have to re-enter *Test & Configure* mode.

When you are out of the *Test & Configure* mode, you can set a mask but you cannot interact with the USBBbox. When you enter it, you get access to all the current open devices as defined by the current mask, and you can see if these devices respond appropriately. If you happen to need to modify the mask during test, you can do this, and when you exit from the test the new mask will be saved. This is what you can do in *Test & Configure* mode:

- Modify the mask if needed;
- Obtain visual feedback when you press the active buttons and when the Voice and Optic keys trigger;
- Reset the USBBbox clock;
- Estimate the time drift of the USBBbox clock;
- Send and receive data via the serial port.
- Manually modify the Voice Key Parameters.

Instead, in *Test & Configure* mode you cannot:

- Use the USBBbox Panel to set conditions and actions parameters in a script.

Many of the features available in *Test & Configure Mode* are self-explanatory. Let us thus only look at those which may appear a bit exotic.

Time Drift Estimation

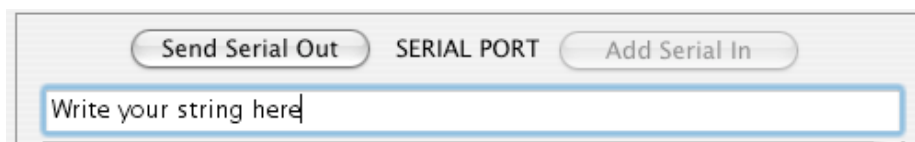
As we were writing above, the USBBox has a clock and the Mac has a clock. They are not synchronized, and there always is a drift between the two clocks. The drift is minimal, but for long experiments, it may be important to estimate it. To do that, we use a procedure that is far from being perfect, but may be sufficient. The procedure is activated by pushing the *Estimate Timer Drift* button. This will check the misalignment between the two clocks for 2 minutes, and will retain the misaligned value.

This test allows you to estimate the drift, not to know it exactly. In order to be faithful, the estimation must be done in the absence of any other event running (e.g., other programs, network activities) insofar as OS X allows you to do it.

You can decide to use this value to directly correct the reaction times in the experiment by clicking the corresponding checkbox. Because checking the USBBox timer takes itself time, the drift value cannot be updated frequently; however, at moments in which the program is not particularly busy, drift is checked again. The correction value you use (if you decide to do it) during the experiment will be the value given by the corrected drift estimation. The updated drift values are saved by default in the data file, so you can check whether they may potentially affect your experiment.

Send Serial Out

The USBBox has a port for serial communication. In Test mode, you can verify that the serial communication works as it should. In order to send serial output, you write an ASCII string in the text field right below the buttons for the Serial Port:



When pressing the *Send Serial Out* button, the string is sent and you will see it appear in the window below:



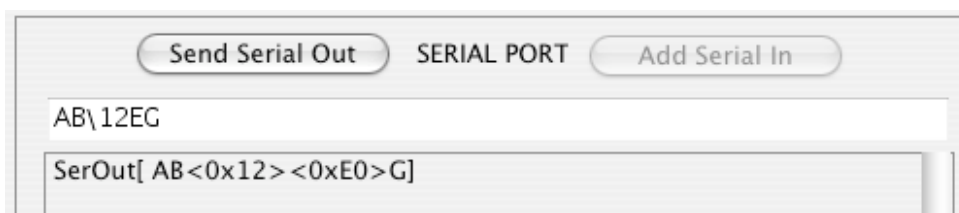
Notice how the string is cut, as there is a limit to the number of bits you can send in one pass.

The syntax for serial communication is pretty liberal. You can basically send any combination of ASCII characters or hexadecimal codes. Examples: “1”, “2”, “3”, “Hans”, “Maria”, “Luca loves pizza” etc.

Use ‘\’ (BACKSLASH) to toggle between ASCII and Hexadecimal out; the default starting mode is of course ASCII. When in HEX mode, any non hexadecimal character toggles the ASCII mode. That is, if you send a string like

AB\12EG

A and B are ASCII, then 12E is an hexadecimal, but G is again an ASCII character. Hence, you will send the following string:



Using hexadecimal, you can also send control characters. The parser also completes the lesser important half-byte. That is, \1 is equivalent to \10. To output the backslash character, use a double backslash (\\).

In the same window where you can see the result of a serial out, you can also see any serial response the USBbox may receive during the test period.

The Voice Key Parameters

This will be better understood after the next session. Suffice it to say that everything you can do in automatic mode as explained below can be manually done here by directly entering the values for the voice key. Importantly,

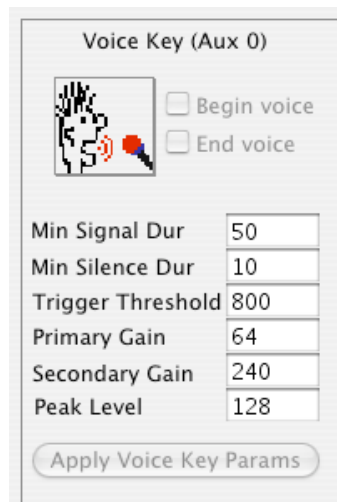
the voice key settings will not be saved inside the script. They are on purpose “volatile” values because they must be adjusted speaker by speaker. They will be saved directly inside the program and remain so any time you load it, until you change them again. It is assumed that you will adjust the voice key before running your experiment.

The Log file will save the voice key parameters set at the beginning of each experiment run, so that you can check their values if you need to. You can also change these values when an experiment is running. Furthermore, you can define a variable in PsyScope X, assign the voice key values to it during the experiment, and save the variable in the data file, so that you can keep track of what they were in the moment in which the voice key triggered.

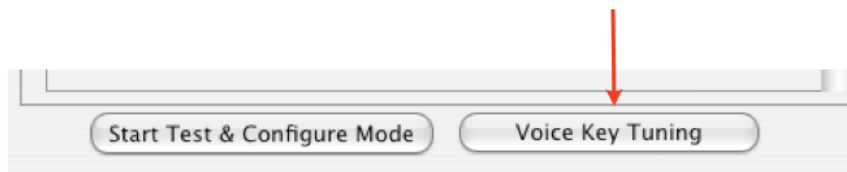
The Delayed Trigger Voice Key and its settings

The USBBox implements a version of the the delayed trigger voice key, as presented by Tyler, Tyler & Burnham (2005)¹, to which the reader is referred for a full understanding of how it works. This key is not a simple trigger that goes off whenever a signal is detected, but a more sophisticated key more suited to do experiments with human voice inputs. However, by setting its parameters appropriately, you may make it work like a traditional sound-triggered key. The relevant part of the window looks like this:

¹ Tyler, M. D., Tyler, L., & Burnham, D. K. (2005). The delayed trigger voice key: an improved analogue voice key for psycholinguistic research. *Behav Res Methods*, 37(1), 139-147.



Another relevant part of the window is the *Voice Key Tuning* button,



which we will describe below. Let us examine the voice key parameters first.

The Minimum signal duration and Minimum silence duration parameters

The main role of the voice key is to run psycholinguistics experiments in which the onset of participants' voice must be detected, although there are many other potential roles you could assign the key during an experiment. In order to detect when a person begins to speak, one has to fix how long a voice signal must be before it is considered speech (as opposed to, say, clacking the tongue while opening ones' mouth to speak), and how long the silence must be for it to be considered the end of speech (as opposed to, say, the silent period before explosion in a stop consonant).

The delayed trigger voice key aims at reducing misses and false alarms. One important aspect of the DTVK is that it is, precisely, delayed. If I want, say, that the minimum signal duration for a sound to be considered speech is 100 ms, then the VK must wait *at least* 100 ms before determining that that was speech and trigger the USBBox (the *at least* close is important; see later). Of course, the longer

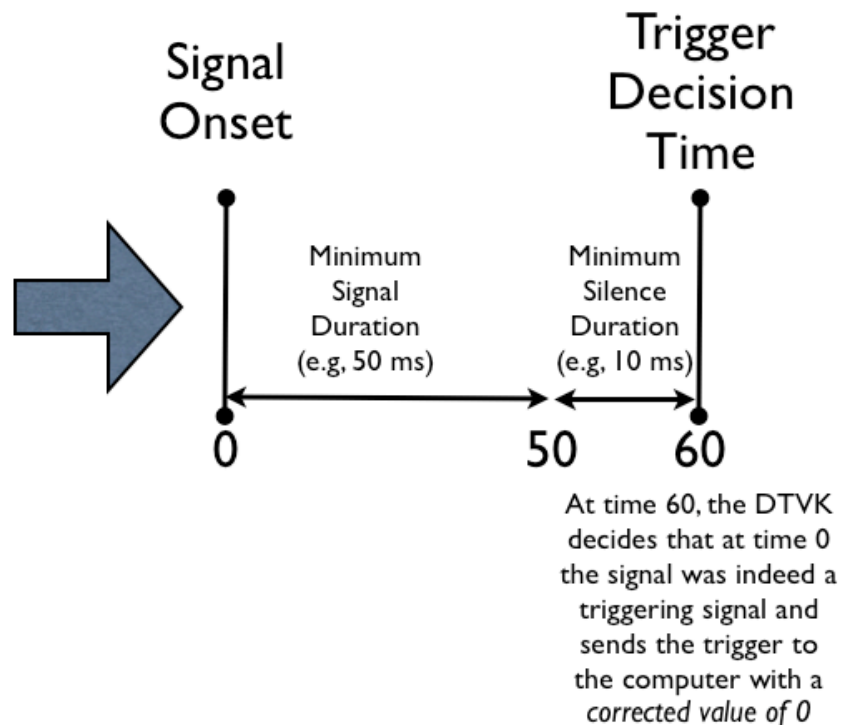
the signal duration for a sound to be considered speech, the later the onset of speech will be determined (because the key must wait at least for the chosen minimum signal duration before deciding that the sound was indeed speech), while the shorter we ask it to be, the higher the chances to trigger false alarms will be (because a click or a sound non related to speech may trigger the voice key).

A parallel argument holds for silence: sometimes silence *is* part of a speech signal -- for example, when you have a stop consonant. You may not want the voice key to go off when the voice pronounces the geminate "t" in "Attention", for example. In this case, you have to set a much longer minimum silence duration, as a silence in a geminate can easily last 100 ms or more. However, if you do that, you also have to know that the silence separation between the words you want to monitor must be larger than 100 ms; you may also have the side effect that accidental sounds followed by some silence will trigger the bbox. It's a balance you have to find yourself.

The above discussion means that, with the current implementation of the DTVK offered by the USBBox,

no action contingent to a voice key response can be faster than the minimum signal duration + the minimum silence duration you set for the voice key.

The following schema may clarify how these values affect the time of triggering:



For example, with a Minimum signal duration of 50 and a minimum silence duration of 10, assuming the ideal case that a continuous 50 ms above-threshold signal begins at time 0 and is followed by a continuous >10 ms silence (or below-threshold signal), the DTVK will trigger a “ON” signal exactly 60 ms after the signal passes above threshold. However, although the decision occurs at 60,

the timestamp for the signal onset will contain a back-corrected value report of 0. Thus, there is a delay, but you will *not* see this delay in the time fields of the data file.

Even if a continuous above-threshold signal is longer than the Minimum Signal Duration, the DTVK will *still* wait Minimum Signal Duration + Minimum Silence Duration in order to send an “ON” trigger, and hence will be reported with that delay. Also an “OFF” trigger after an “ON” will be delayed, but only of the value of the minimum silence duration. OFF triggers will be corrected in the timestamp reports likewise.

You have to keep this in mind when designing your experiment. This being said, in most cases psycholinguistic experiments only ask to detect the onset of speech, and not to trigger actions conditional to that detection. Hence, in most cases the delay can be tolerated. Minimum signal and silence durations can go from 1 ms to 255 ms.

Reasonable minimum signal duration and silence durations for the DTKV may be 50 and 10 for the human voice. This should guarantee a good accuracy for voice onset detection, at a price of triggering the Voice key many times during a single speech episode. Start from these values and change them according to your needs. These are also the default values in the firmware of the USBBox.

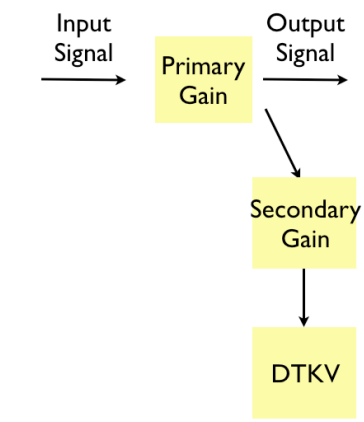
Because by default when you collect a reaction time PsyScope only collects the *first* trigger of any input device, repeated triggers of the voice key during the same speech episode should not pose a problem.

Trigger Threshold

We just saw that for the DTKV to work, one has to set a *minimum signal duration* and a *minimum silence duration*. But determining that a signal is a signal also requires that a minimum signal thresholds must be set for the timers to begin counting signal. A trigger threshold can go from 1 to 900. These are arbitrary values, and the USBBox comes pre-configured with a high threshold of 800. You generally don't need to touch this value.

Primary and Secondary Gain

Part of the working of the DTKV depends on boosting the signal in such a way that the differences between below and above threshold signals are maximized. The USBBox contemplates two separate gains, arranged as per the schema below:



The primary gain is a first-pass amplification that increases the signal strength like a normal gain control would do. This amplified signal is passed to the sound output if sound passthrough is active: this is what you will record if you ever record the USBBox sound output. The secondary gain is a further signal boost that is passed directly to the DTKV, and is only used by it. It would not make sense to record this twice boosted signal, as the purpose of double boosting is to distort it for increasing the efficiency of voice onset detection.

Primary and secondary gains range from 1 to 255; default values are, respectively, 64 and 240. You have to experiment with your microphone, as these values may change drastically from one microphone to another. Generally, you will want to use the tuning procedure described below. But as a general rule,

you should tend to minimize the primary gain and maximize the secondary gain. However, the optimal values depend on your microphone, the speaker, and your computer.

The peak level

Peak level cannot be adjusted manually. It is automatically determined by the calibration procedure described below, and it is only shown in the graphical interface for your knowledge.

The Voice Key Tuning button and the DTKVTuner program

When you press the *Voice Key Tuning* button, you actually start a program (called *DTKVTuner*) that takes you out of the Configuration Window and out of PsyScope, into the Unix environment.

Remember that in order for the program to fully work, you *must* activate the voice key in the mask by clicking onto the voice key icon, Without the voice key active in the mask, no trigger will be taken.

With the DTKVTuner, you can

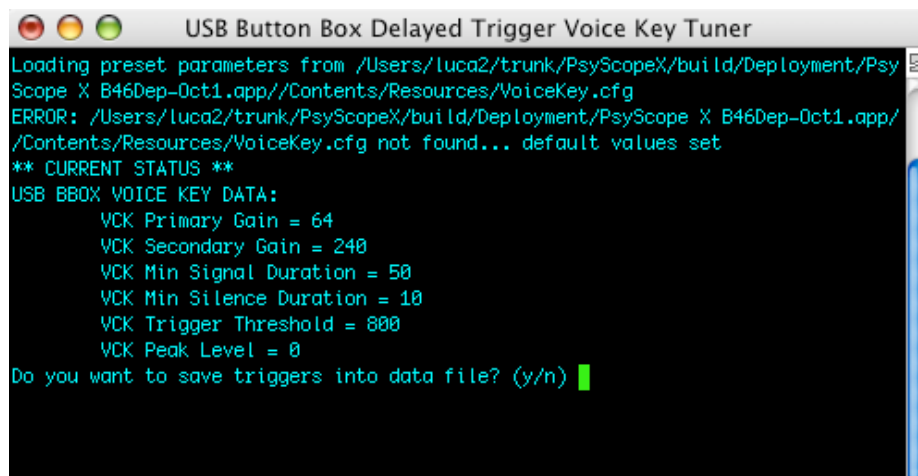
- manually calibrate the Voice key;
- access a procedure to automatically calibrate the Voice Key parameters;
- test the quality of the input to the DTKV by recording the sound output from the USBBox
- see when the DTKV triggers when it receives signal.

We assume you have a microphone connected to the sound input jack (the red plug on the back of the USBBox. The green one is for output. This is the convention for the Ugly Other Computers Besides Macs) and that you also have connected a male-male audio cable from the USBBox to the line in of your computer if you want to record. Remember:

if you want to record the USBBox sound output, you have to connect the sound output from the USBBox to the line in *and* you have to set the Sound Preferences of your computer to record from the line in. Try setting the *Sound input volume* to around 70% and check the sound quality before relying on these recordings.

Starting the DTKVTuner

After pressing the *Voice Key Tuning* button, PsyScope enters a waiting mode, and the control is passed to the Unix terminal. A new terminal window opens and you see the following screen:



```
USB Button Box Delayed Trigger Voice Key Tuner
Loading preset parameters from /Users/luca2/trunk/PsyScopeX/build/Deployment/Psy
Scope X B46Dep-Oct1.app/Contents/Resources/VoiceKey.cfg
ERROR: /Users/luca2/trunk/PsyScopeX/build/Deployment/PsyScope X B46Dep-Oct1.app/
/Contents/Resources/VoiceKey.cfg not found... default values set
** CURRENT STATUS **
USB BBOX VOICE KEY DATA:
    VCK Primary Gain = 64
    VCK Secondary Gain = 240
    VCK Min Signal Duration = 50
    VCK Min Silence Duration = 10
    VCK Trigger Threshold = 800
    VCK Peak Level = 0
Do you want to save triggers into data file? (y/n) █
```

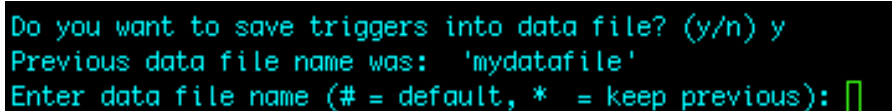
The first time you run the program, no already set value exists, and therefore the program loads the default values in the UsbBbox. These are shown in the terminal window. The question you are prompted asks whether you want to save both the sound input to the USBBbox onto the computer, as an .AIFF mono sound file, and a text file containing the trigger values. If you want to check how accurate the voice key is, you are warmly suggested to answer “yes” (y, lowercase).

Recording sound and trigger data to computer

If you were wise and answered yes, you are prompted for a file name. If none is specified, the default name (DTKV_Triggers and DTKV_Triggers.aiff) will be used

By default, the files will be saved into your home directory, which is the default directory for the Unix terminal, unless you specified otherwise.

If a file already exists (which will occur if you calibrate repeatedly), you are asked whether you want to append the files or create new ones:



```
Do you want to save triggers into data file? (y/n) y
Previous data file name was: 'mydatafile'
Enter data file name (# = default, * = keep previous): █
```

and again, you can use the default file (with '#') , or append the existing file (with '*'), or else create a new file.

Auto vs manual tuning procedure

After this step, the program asks you whether you want to use an automatic or a manual tuning procedure.

This choice only concerns the primary and secondary gain parameters, as you have to decide yourself the values for the minimum signal, minimum silence duration and the trigger threshold.

No matter whether you select the manual ('m' lowercase) or automatic procedure ('a' lowercase), you are first asked whether you want to modify the permanent parameters (that is, the minimum signal, minimum silence duration and the trigger threshold). You can do that or skip this according to your needs.

The Manual Procedure

In the manual procedure you can then modify the secondary and primary gains by entering new values (from 0 to 255) *in that order*.

```
Select (a)uto or (m)anual tuning of gains & zero level (a/m) m
(k)eeep or (m)odify permanent params? k
(k)eeep or (m)odify gains params? m
Enter II_gain [0-255] I_gain[0-255]
-> 240 16

240 16 Right? (y/n) █
```

When you confirm, the new set of values is displayed, and the algorithm asks you to enter a carriage return to set the zero level. After that setting, the manual and the automatic procedures proceed in the same way. So let us take first a look at the automatic procedure.

The Automatic Procedure

In the automatic procedure, the program chooses the secondary and primary gains by for you by estimating the signal strength and the background noise. To estimate the signal, you are first asked to speak at a normal level with a continuous sound. For simplicity, we ask you to say "PAAAA" until the programs tell you to stop:


```

Select (a)uto or (m)anual tuning of gains & zero level (a/m) a
(k)eep or (m)odify permanent params? k
USB BBOX VOICE KEY DATA:
    VCK Primary Gain = 16
    VCK Secondary Gain = 240
    VCK Min Signal Duration = 50
    VCK Min Silence Duration = 10
    VCK Trigger Threshold = 800
    VCK Peak Level = 127
Say 'PAAAAA...' for several seconds and while doing it press <enter>
DO NOT STOP BEFORE THE PROGRAM SAYS SO... █

```

Remember to *first* begin speaking and *then* hitting <enter>.

When the program has determined the primary gain, it will tell you

```

Say 'PAAAAA...' for several seconds and while doing it press <enter>
DO NOT STOP BEFORE THE PROGRAM SAYS SO...
WAIT: Starting self tuning of gains & zero level...

Calibrating PRIMARY GAIN...
    STEP 01  primary gain = 128 peak level = 189
    STEP 02  primary gain = 64 peak level = 157
    STEP 03  primary gain = 32 peak level = 141  ----> GOT IT!

```

and will ask you to stay silent in order to assess background noise:

```

Now, stay SILENT and ...
Press <enter> without making any sound...

Calibrating SECONDARY GAIN...
    STEP 01  secondary gain = 240  ----> GOT IT! Peak level 128
Calibration completed

```

After this, the newly determined set of parameters is shown, and you can begin tracking the triggers:

```

USB BBOX VOICE KEY DATA:
    VCK Primary Gain = 32
    VCK Secondary Gain = 240
    VCK Min Signal Duration = 50
    VCK Min Silence Duration = 10
    VCK Trigger Threshold = 800
    VCK Peak Level = 128
Press <enter> to reset timer & start tracking TRIGGERS... █

```

Tracking and inspecting the triggers

When you hit <enter>, the program will first play a long beep. This is done to set a first clear trigger, as well as to have a clear sound recorded into your sound file. After the beep is finished, you can start speaking, and you should see the ON and OFF triggers being written to the terminal:

```
Press <enter> to reset timer & start tracking TRIGGERS...
Time to get timer reset 2 ms.
Press <enter> to STOP tracking...
Start recording NOW...
Sound START call took 105 ms (sound frame 512)
SET SYSTEM REFERENCE TIME NOW: 315 ms after bbox RTC reset
BBOX RTC reset took 2 ms
Sound FINISHED. Duration 386 ms
Sound CLOSE call took 25 ms
GOT TRIGGER type D -- 1594 ms (BBOX timer) -- MAC timer: 1594
GOT TRIGGER type U -- 1688 ms (BBOX timer) -- MAC timer: 1689
GOT TRIGGER type D -- 2130 ms (BBOX timer) -- MAC timer: 2131
GOT TRIGGER type U -- 2200 ms (BBOX timer) -- MAC timer: 2201
GOT TRIGGER type D -- 2657 ms (BBOX timer) -- MAC timer: 2657
GOT TRIGGER type U -- 2714 ms (BBOX timer) -- MAC timer: 2715
GOT TRIGGER type D -- 3105 ms (BBOX timer) -- MAC timer: 3105
GOT TRIGGER type U -- 3188 ms (BBOX timer) -- MAC timer: 3189
GOT TRIGGER type D -- 4457 ms (BBOX timer) -- MAC timer: 4457
GOT TRIGGER type U -- 4503 ms (BBOX timer) -- MAC timer: 4503
GOT TRIGGER type D -- 4880 ms (BBOX timer) -- MAC timer: 4880
GOT TRIGGER type U -- 4964 ms (BBOX timer) -- MAC timer: 4964
```

In analogy with a button, type *D* triggers are *down* triggers (as if you were pressing a button), hence “ON” signals, and type *U* triggers are *up* triggers (as if you were releasing the button), hence “OFF” triggers. This recording will continue until you press <enter> again. These values will be saved in a text file, which will look like this:

```
AUTOMATIC tunings of gain & zero level selected
USB BBOX VOICE KEY DATA:
  VCK Primary Gain = 32
  VCK Secondary Gain = 240
  VCK Min Signal Duration = 50
  VCK Min Silence Duration = 10
  VCK Trigger Threshold = 800
  VCK Peak Level = 128

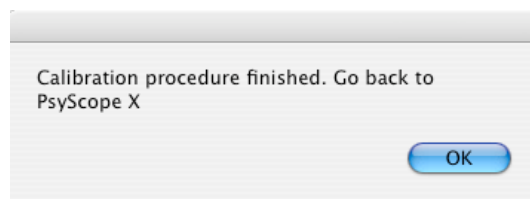
Type  BBOX timer      MAC timer (since beep start)
ON    1594            1594
OFF   1688            1689
ON    2130            2131
OFF   2200            2201
ON    2657            2657
OFF   2714            2715
ON    3105            3105
OFF   3188            3189
ON    4457            4457
OFF   4503            4503
ON    4880            4880
OFF   4964            4964
```

You can then check your text file and your sound file, put into correspondence the trigger times with the sound signal, and figure out if the trigger accuracy is sufficient for your needs. This can be rather easily achieved by scripting a program like PRAAT (<http://www.fon.hum.uva.nl/praat/>).

Ending the tuning procedure and going back to PsyScope.

You can repeat the tuning procedure if you are not happy with it as many times as you please. When you are finished, you must decide whether to save the last tuned parameters. If you do, when you go back to PsyScope, you should see these values immediately reflected in the *UsbBbox Stuff...* window, and these parameters will be used during your experiment. These parameters will be also saved into the Log file when you run the experiment.

If you decide not to save the parameters, the old parameters will remain active. In either cases, a dialog box will tell you that the procedure is finished:



Clicking “OK” will close the DTVKTuner, but will not take you back automatically to PsyScope: you have to make the program active manually.

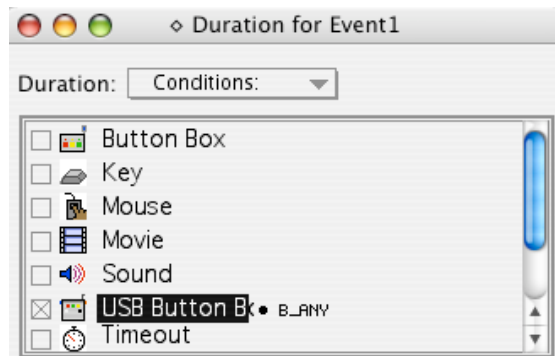
We can now begin looking at how to design a real experiment. As in all other aspects of PsyScope, input/output devices interact with the program via *Conditions* and *Actions*. We will analyze these commands separately.

Designing your experiment: The USBBbox condition

Event attributes and event actions have now access to a new set of conditions relative to the UsbBBox, along the lines of the old CMU Serial Bbox. Only, there are much more possibilities given by the UsbBBox.

The Default Condition

When you specify that an event terminates with a UsbBbox input, you select the USBBbox in a standard condition window:



By default, clicking on the UsbBbox icon will select any USBBbox input line available *given the current mask*. That is, the condition written in the script will be:

`USBBBox[B_ANY]`

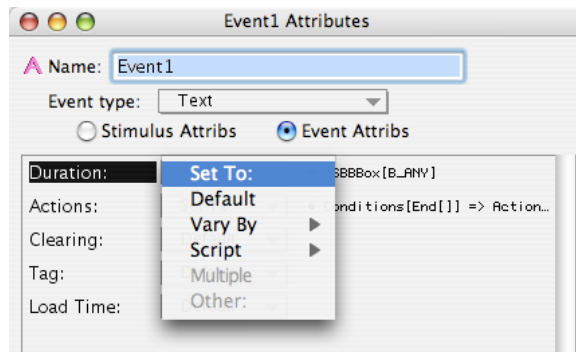
Any really means *any given the mask*: if the voice key is active, a voice trigger will terminate an event whose duration is `USBBBox[B_ANY]`. If instead, say, Button 4 is masked, nothing will happen when button 4 is pressed even if the event terminates with `USBBBox[B_ANY]`.

Selecting restricted conditions

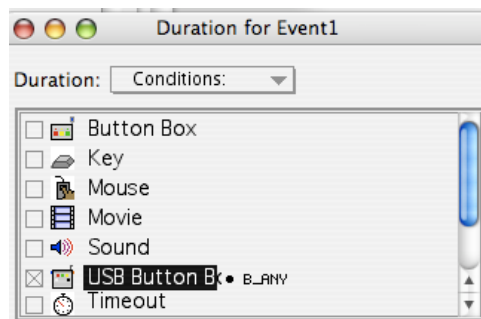
If the `USBBBox[B_ANY]` condition is too wide for your needs, you can click on the UsbBbox line of the *Conditions* window, and you will open again the USBBbox window we have already discussed.

However, this time the window has a different role. Instead of allowing you to select a mask, it allows you to write the relevant conditions in the script. Thus, you do not operate on the button icons only, but also on the *Press and Release*, the *Begin Voice*, *End Voice*, or the *Begin Optic*, *End Optic* checkboxes.

Thus, suppose you want to end an event when the subject presses Button 3 or Button 4, releases Button 5, does anything on Button 6, or else stops speaking. You would select the *Duration* option of the *Event Attribs*:



then open the *Condition* window and double-click on the USBBox part:



and when the USBBox window open, deselect the irrelevant buttons, select Button 3 and 4, (or equivalently, mark the *Press* checkbox of Button 3 or 4 -- in the example, we do both things) the *Release* checkbox of Button 5, both checkboxes of Button 6, and mark the *End Voice* checkbox of the Voice Key after having selected it:

po po po po po po po po

po po po po po po po po

Port 0 On/Off

Port 2 On/Off

Active Buttons

Min Signal Dur 50

Min Silence Dur 10

Trigger Threshold 800

Primary Gain 32

Secondary Gain 240

Peak Level 130

Apply Voice Key Params

B2_DOWN B3 B4_UP B5_BOTH VOICE_END

Notice that while you are doing that, the text field (which you cannot modify) writes the expression that it will finally be written in the script, for your inspection. Thus, when you press OK, you will find this duration condition in the script:

```
Event1::
  EventType: Text
  Duration: USBBox[ B2_DOWN B3 B4_UP B5_BOTH VOICE_END ]
```

If you scripted the script directly, and then opened the USBBox window, you would find the same button configuration shown above, as the window reflects what it finds in the scripts or else scripts what is reflected in the graphical interface. You will have guessed that *B* stands obviously for Button, *VOICE* for the Voice key (Aux0), and, surprise, *OPTIC* for the optical in (Aux1).

Notice that, as the example above shows, *B2_DOWN* is equivalent to *B2*. Likewise, *VOICE* is equivalent to *VOICE_START* and *OPTIC* to *OPTIC_START*.

Selecting conditions depending on other input ports

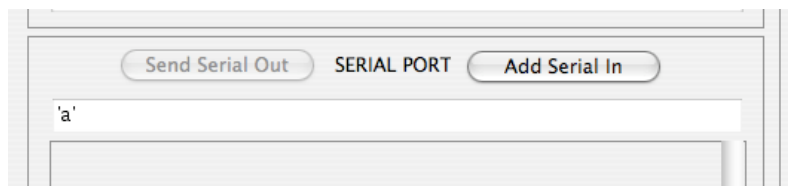
The Serial In condition

While setting Conditions, the USBBox window has another slight change connected to the serial port. If you now inspect the buttons related to it, you will see that the *Send Serial Out* button is grayed, and the *Add Serial In* button is not. With that, you can add conditions related to incoming serial communication.

To do that, you have to use the same text field below the buttons that was used to test serial communication. However,

serial in data must be written within single quotes.

Thus, to write a condition that triggers when the serial port receives an 'a' character, you have to write:



then press the *Add Serial In* button. This will be reflected in the window in the relevant text inspection field



as well as in the script when you OK the changes, with the following syntax:

```
Event1::  
  EventType: Text  
  Duration: USBBBox[ "SERIALIN[ 'a' ]" ]
```

Notice the double quotes surrounding the *SERIALIN* clause, and the single quotes surrounding the ASCII character.

This is not the most straightforward syntax, but for reasons connected to the PsyScope legacy code doing otherwise would have taken an unjustified effort. Inside a *SERIALIN* condition you can write data with the same syntax we explained in the [Send Serial Out](#) section. Thus

```
Conditions[ "SERIALIN['Hello']" ]
```

is well formed.

Notice, however, that because the single and double quotes have special meanings, they also need to be preceded by a backslash. Thus, if you want a *SERIALIN* condition triggered by, say, *Hello* followed by a carriage return and then followed by the single quote, you will write:

```
Conditions[ "SERIALIN['Hello\13\']" ]
```

Notice the double single quote: one preceded by the backslash, interpreted literally, and the other as a closure of the data structure.

Serialin special conditions

The data coming from the serial port are bufferized. Thus, also conditions on the size of the buffer can be written. So the condition

```
Conditions[ "SERIALIN[=20]" ]
```

is satisfied whenever the input bytes are exactly 20. Likewise, the condition

```
Conditions[ "SERIALIN[<30]" ]
```

is satisfied so long as the input bytes are less than 30, and the condition

```
Conditions[ "SERIALIN[>10]" ]
```

is satisfied any time the input bytes are more than 10.

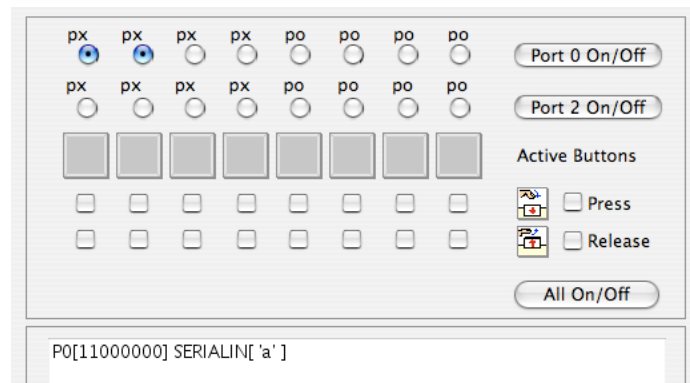
Finally, the condition

```
Conditions[ "SERIALIN[]" ]
```

(where nothing is written inside the inside square brackets) is legitimate and satisfied by *any* input from the serial port.

Conditions on other input ports

Any port that is enabled as input port can be used and scripted in a *Condition*. Thus, if the first four lines of Port0 are enabled as input lines, you can write a condition triggered by input coming to, say, the first two lines by clicking the first two radio buttons:



You can see from the *px* string that they are input lines (with positive logic).²

This condition will write the following clause in the script:

Event1::

EventType: Text

Duration: USBBBox["P0[11000000]" "SERIALIN['a']"]

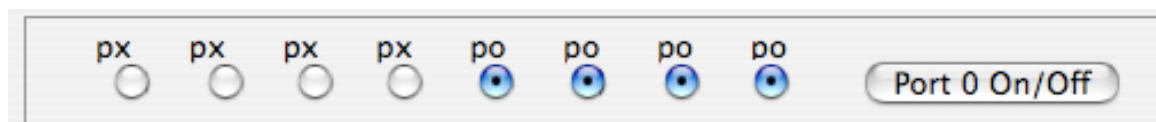
Again, notice that the *P0[11000000]* part is written between double quotes.

Conditions on Ports P0 and P2 are matching conditions: they trigger when all the string is verified.

That is, a condition such as

Conditions[USBBBox["P0[11100111]" "P2[10101010]"]] =>

² In fact, if you switched again in *Test* mode, you would see that you *could not* select the *x* (input) ports by clicking their radio buttons, whereas the *o* ports (outputs) would be highlighted if active. That is, you would see this:



assuming that the script enables the first four lines as input and the last four as output, with this *ExperimentDefinition* clauses:

USBBBoxInitialSettings:"P0[00001111]"

USBBBoxInitialPortsDir: "P0[11110000]"

will trigger iff

- (the status of Port 0 is 1 AND 1 AND 1 AND 0 AND 0 AND 1 AND 1 AND 1)

or

- (the status of Port 1 is 1 AND 0 AND 1 AND 0 AND 1 AND 0 AND 1 AND 0)

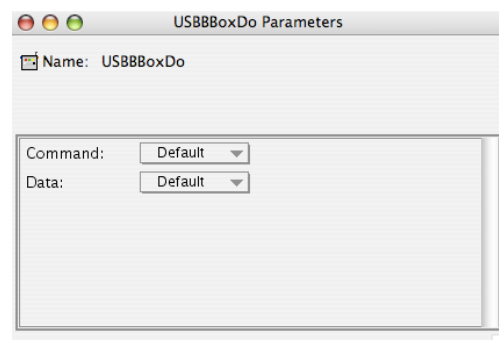
Analogous considerations hold for the AUX0 and AUX1 ports, when enabled as input ports.

A caveat: remember that when when Port 2 Loopback is active, the direction of the single lines you may have set for Port 2 will be ignored.

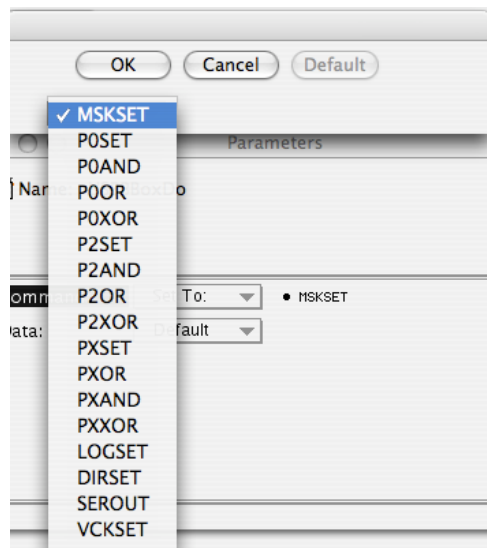
Designing your experiment: The USBBoxDo action

When you define a *Condition-Action* pair, the *USBBoxDo* action allows you to modify practically any aspect of the new USBBox. Let us take a look at them and their syntax.

Selecting the *USBBoxDo* action opens the following window:



and the *Command* menu allows you to access to a pulldown menu with the available choices:



MSKSET

MSKSET allows you to change the masking state of the asynchronous ports of the USBBox during the experiment. These ports are Port 1 (the buttons) and the AUX port. The AUX port comprises the optic and the voice keys, but in fact is a port like any others, containing 8 bits. Thus, this command takes 16 bits as an argument, the first 8 of which correspond to the eight buttons, and the other 8 corresponding to the AUX lines.

Notice that for hardware reasons, the two AUX lines corresponding to the voice and the optic keys are, unintuitively, the lines 3 and 4 of the AUX port. Thus the action:

```
Action[ MSKSET 1111111100110000 ]
```

would make all the eight button active, as well as the voice and optic keys.

P0SET, P2SET, PXSET

These commands set port P0 or P2, or both together. They take a 8 bit sequence as input (16 bits for *PXSET*). Thus,

```
Action[ USBBoxDo[ P0SET 10101010 ] ]
```

sets lines 1, 3, 5, and 7 to ON on Port 0 and

```
Action[ USBBoxDo[ PXAND 101010101111111 ] ]
```

sets lines 1, 3, 5, and 7 to ON on Port 0 and all lines of Port 2 to ON. These commands work in conjunction with the logic and direction of the ports, which are set independently.

P0OR, P2OR, PXOR

These commands work exactly as the commands above, but use an OR logic to set the ports.

P0AND, P2AND, PXAND

Same as above, with AND logic.

DIRSET

This command sets the port directions (as input or output), line per line, and take a 16 bit argument, 8 per port. Thus

```
Action[ USBBoxDo[ DIRSET 100000011111111 ] ]
```

sets Line 1 of Port 0 as output line and the rest as output lines, and sets all the lines of Port 2 as output lines.

For Port2 (the buttons), *DIRSET* can be used also to activate port loopback. The loopback may allow direct hardware feedback to button presses, by switching the LEDs on or off when the buttons are pressed. In this case, you have to pass two parameters: one is the 16 bit string as above, and the other is one of the three parameters

```
P2NoLoopback
```

```
P2LoopbackSwitchOnLedOn
```

```
P2LoopbackSwitchOffLedOn
```

whose meaning is self explanatory Thus, an action such as:

```
EventActions: Conditions[ Start[] ] => Actions[ USBBoxDo[ DIRSET  
"100000011111111 P2LoopbackSwitchOnLedOn" ] ]
```

will activate loopback from that moment on, and will cause the LEDs to switch on when the buttons are pressed. Of course, you can obtain the same behavior by scripting the LEDs' behaviors, but the loopback mode will allow you to obtain a more immediate response, as the switching is directly handled by the bbox and does not need to pass through the OS and the software.

LOGSET

This command sets the port logic (as positive or negative), line per line, and take a 16 bit argument, 8 per port. Thus with

```
Action[ USBBBoxDo[ LOGSET 101010101111111 ] ]
```

lines 1, 3, 5, and 7 of Port 0 work in positive logic and the even lines of Port 0 work in negative logic, whereas all lines of Port 2 work in positive logic. As a default, all lines of both ports work in positive logic.

SEROUT

This command sends any data out via the serial port. Thus

```
Action[ USBBBoxDo[ SEROUT Pippo is \0Athe best ] ]
```

sends the (unquoted) string "Pippo is\nthe best". The syntax for serial communication is explained in the [Send Serial Out](#) section.

VCKSET

This command can change the voice key parameters during the experiment. It takes the parameters available to tune the voice key as explained [above](#). Thus, as you would expect, the parameters it can take are

- *MinSigDur* for Minimum Signal Duration;
- *MinSilDur* for Minimum Silence Duration;
- *TrigThreshold* for Trigger Threshold;
- *PriGain* for Primary Gain;
- *SecGain* for Secondary Gain; and
- *MicPass* for Microphone PassThrough on or off.

Thus, the following are well formed actions:

```
Action[ USBBoxDo[ VCKSET "MinSigDur(20) MinSilDur(100) PriGain(100)" ] ]
```

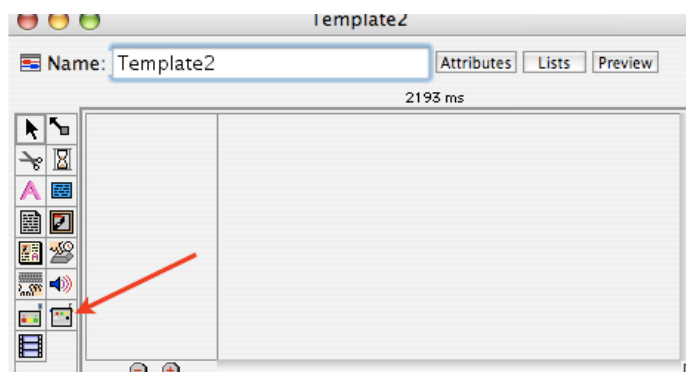
or

```
Action[ USBBoxDo[ VCKSET "TrigThreshold(600)" ] ]
```

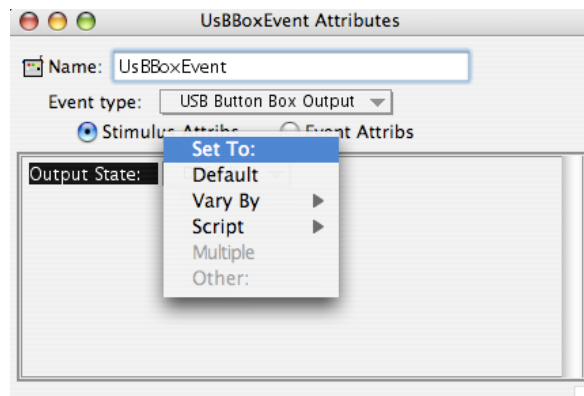
or any other combination you may want to use. By carefully using these parameters, it is possible to adapt the voice key to the speaker. For example, as speakers modify their voice during the experiment by getting softer or louder, it is possible to design events that, adding on certain predefined keys, change the primary gain "on the fly" during the experiment. But this requires some practice with the behavior of the Voice Key.

The USBbox event type

Just as PsyScope 9 provided a BBox event type for the CMU Button Box, we have maintained the possibility of introducing an USBbox event type accessible from a template window:



This is essentially a shorthand to define output states of the ports 0 and 2. Thus adding an USBbox event and setting its stimulus, via the graphical interface just as you would do for any other stimulus, thus:



will open the USBBox window. From this, you will only be able to access the part relevant to this stimulus -- the upper left part allowing you to fix Port0 and Port 2. Clicking on every other part of the window will have no effect. Setting the stimulus will write the stimulus definition in the script with the by now known syntax, contained in double quotes. For example, a USBBox event may look like this:

```
UsBBoxEvent::
  EventType: USBBox
  Duration: 100
  Stimulus: "P0[00001111]" "P2[00001111]"
```

and the USBBox window that will open if you want to modify the definition of the stimulus for this event will reflect any change you may want to script directly within the definition.

Thus essentially running a USBBox event is equivalent to using a condition-action pair within any event, in which the action contains a *USBBoxDo* command with a *P0SET*, *P2SET* OR *PXSET* CLAUSE. Notice that even for USBBox events, the stimulus definition is affected by the logic and direction of the ports.

Designing your experiment: Other initialization parameters

There are other aspects of the USBBox that you can customize for your experiment, which are only accessible by direct scripting the *ExperimentDefinitions* section of your script.

The debounce period

Any time a button is pressed or released (or any other line goes on or off for that matter), a decision must be made as to how short a press/release event must be for it to count as a press/release event. The spring itself of a button can oscillate, thus creating a train of press/release events when the subject only pressed one. Thus, a *debounce period* is defined, which is basically a refractory period starting after a change in state is detected for a button, within which any other state change is disregarded.

By default, the button press debounce time is set to 20 milliseconds, and the release debounce time to 5 milliseconds. The same values are set for the Optic key line (INT1), whereas the voice key parameters are set to 0 and entirely under the control of the DTVK tuning procedure.

For special needs, you may want to change the default debounce periods, by adding a line like this in the *ExperimentDefinitions* section:

```
USBBoxDebouncePeriods: "B_ANY(10)"
```

This would reduce the release debounce period for any key to 10 milliseconds. The following predicates are supported:

```
B_ANY() B_ANY_UP() B5() B5_UP() VOICE_START() VOICE_END()  
OPTIC_START() OPTIC_END()
```

where the desired time in milliseconds must be put within the brackets. All the debounce period settings must be in one single line, each clause contained in double quotes. Thus,

```
USBBoxDebouncePeriods: "B_ANY_UP(40)" B_ANY_(100)" "OPTIC_START  
(200)" "OPTIC_END(400)"
```

would be well-formed.

As we recalled, in its normal use the voice key debounce periods are really the minimum signal and minimum silence durations set via the DTVK tuning procedure. However, we left the possibility to set them via *ExperimentDefinitions* because in certain conditions you may want to deactivate the voice key and

use its line as a normal AUX input line; in this case, and only in this case, you would want to modify the debounce settings. Always because of the normal use of this line for the voice key, the debounce default values are set to 0.

Port Logic

Port logic for Ports 0 and 2 can be set via a line such as the following in *ExperimentDefinitions*:

```
USBBBoxInitialPortsLog: "P0[00001111]" "P2[11110000]"
```

Zero is for negative logic, 1 for positive logic. In this example, the first four lines of Port 0 and the last four lines of Port 2 work in negative logic. By default, all lines work in positive logic.

Port Direction

Port direction for Ports 0 and 2 can be set via a line such as the following in *ExperimentDefinitions*:

```
USBBBoxInitialPortsDir: "P0[11110000]" "P2[11110000]"
```

The value 0 is for output, 1 for input. In this example, the first four lines of Port 0 and the last four lines of Port 2 are input lines. By default, all lines are output lines.

Port2 Loopback Mode

It may be useful to have direct feedback for button presses by using the LEDs. To do this, you have to activate the Loopback Mode for Port2 (the buttons). This can be done in two ways: either using a condition-action pair in an event, as [explained above](#), or else by using the following in *ExperimentDefinitions*:

```
USBBBoxInitialPortsDir:P2LoopbackSwitchOnLedOn
```

where you can change the parameter with the following values:

```
P2NoLoopback
```

```
P2LoopbackSwitchOnLedOn
```

```
P2LoopbackSwitchOffLedOn
```

Obviously, default is no Loopback. *P2LoopbackSwitchOnLedOn* will switch on the LEDs while the buttons are pressed, while *P2LoopbackSwitchOffLedOn* will switch the LEDs on at the beginning of the experiment, and will switch them off while the buttons are released.

Serial data parking

During an experiment, serial data coming in may be used only to trigger some events. In this case, as soon as the condition defined by a *SerialIn* condition clause is satisfied, the relative action will be taken. However, in some experiment, one may be interested in making other considerations on the serial data -- for example, by comparing two streams of data coming in at a different moment during the experiment. In that case, the data must be retained. It is possible to activate a temporary buffer that will park the data for further use. To do this, you have to add to the *ExperimentDefinitions* section the following line:

USBBBoxParkSerialData: TRUE

the default being FALSE. When data are parked, you can assign them to variables via expressions, as explained below.

Once you activate data parking, you will also have to take care to empty the buffer. This can be done by assigning serial in data to variables, as explained below.

Designing your experiment: Setting variables with *SERIALIN* data and *USBBOX* expressions

It is possible to retrieve several current values of the button box, which can be either used inside logico-arithmetical expressions or to assign values to variables. The variables must be defined by the user in the usual way provided by PsyScope, and must be of the right type. The reasons why we allow certain Button Box expressions to appear in a logico-arithmetical expression is that, unlike functions, which are always evaluated *before* a trial is executed, expressions are evaluated in real time. Thus, assignments of values to variable and expression computations correctly reflect the current state of the USBBbox.

If your experiment may need them, we strongly recommend using variables to save these values in the data file. For example, you may want to have a column that marks what the minimum signal duration of the Voice Key was at the moment a reaction time was used, as this may have been changed during an experiment. Definitely save more data than less!

The available commands are listed below. In the example that follow, we assume that the user has already defined the variable *MyVariable* in her script, with the appropriate type.

All these assignments, for which we present scripting examples, can be set via the graphical interface by using the *Set* option provided in the action list of the *EventActions* window, by typing the relevant variable name in *LValue* and the correct *unquoted* string in *Expression*. Alternatively, they can be scripted directly by writing the *Set* expression inside an *Action[]* clause, inserting the expression inside double quotes.

GETKEY

With the expression

```
Set[ MyVariable "USBBOX GETKEY" ]
```

inside an action, the user can retrieve the state of the two asynchronous input ports (buttons and AUX). The result is a 16 bit string.

GETMSK

With

```
Set[ MyVariable "USBBOX GETMSK" ]
```

the user can retrieve the mask settings of the USBBbox. The result is a 16 bit string.

GETRTC

With

```
Set[ MyVariable "USBBOX GETRTC" ]
```

you can retrieve the current USBBbox timer. The result is a long integer value.

Do remember that, for the reasons explained above, the precision of the returned value is likely to be low. This expression directly queries the USBBox for the timing information, and waits for an answer. It is thus subject to the necessary delays imposed by the USB communication, plus the delay that the USBBox will add in processing the command.

GETLOG and GETDIR

The expressions

```
Set[ MyVariable "USBBOX GETLOG" ]
```

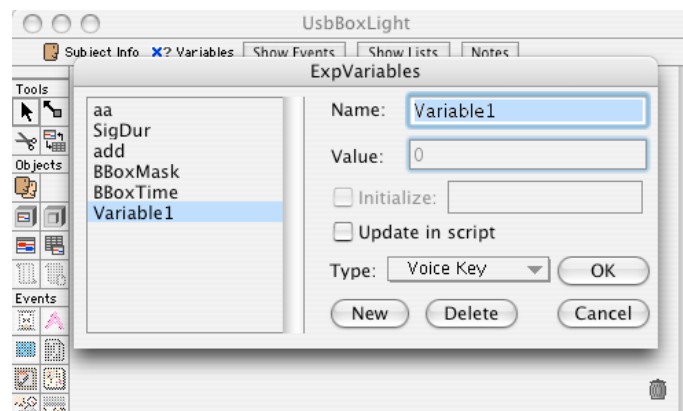
and

```
Set[ MyVariable "USBBOX GETDIR" ]
```

return, respectively, the port logic and directions, as strings of 16 bits.

VCKGET

Retrieving voice key parameters is slightly special. As there are multiple parameters, we introduced a new type of variable, *Voice Key*, which you will find under the *Variables* window:



This variable is basically an array, composed of these fields: *MinSigDur*, *MinSilDur*, *TrigThreshold*, *PriGain*, *SecGain*, *MicPass*, whose meaning should be obvious and is defined in the [VCKSET](#) section of this manual. So, in order to extract a voice key parameter, you actually need two variables: one of type *Voice Key*, and one of type integer or long integer. You can then use the double assignment

```
Set[ MyVoicekey "USBBOX VCKGET" ]  
Set[ MyNumber "MyVoicekey->MinSigDur" ]
```

assuming that *MyVoicekey* is of type *Voice Key* and *MyNumber* is of type *Integer* or *Long*. Notice that the second assignment does not use the *USBBOX* predicate, but is a normal assignment of a variable from an array field in the usual PsyScope language.

Composing USBBox predicates inside expressions

USBBox predicates can occur within expressions with arithmetic operators, provided that their types are compatible. Thus, for example, given that *GETRTC* returns a long integer, you can write

```
Set[ MyVariable "USBBOX GETRTC" ]
```

as we saw, but also

```
Set[ MyVariable "(USBBOX GETRTC) + 1" ]
```

or anything else containing arithmetic and logical operators, if you need to do so. The only thing to remember is that the *USBBOX* operator (with the exception of *Serin*) is unary, and therefore the following syntax

```
Set[ MyVariable "USBBOX GETRTC + 1" ]
```

or

```
Set[ MyVariable "USBBOX (GETRTC + 1)" ]
```

,where numbers are specified inside the *GETRTC* scope, would have a completely different meaning: the operation inside parentheses would be executed, which would result in a number, and *USBBOX* over that number would be interpreted as a code for a different command -- most likely, not the one you want. This is something you want to avoid.

In these expressions, non numerical values are converted to numbers. So

```
Set[ BBoxMask "USBBOX GETMSK" ]
```

would be evaluated as a 16 bit string (e.g., 1111111100100000) and assigned to the variable *BBoxMask* of type string. But

```
Set[ BBoxMask "(USBBOX GETMSK)+1" ]
```

would give a number (e.g., 28065, if *BBoxMask* were of type *Long_Integer*).

The *SERIALIN* assignment

The expression

```
Set[ VAR "SERIALIN n" ]
```

(no *USBBOX* predicate here), where n is a number from 1 to 256, return the n latest characters parked in the serial port buffer, and removes them from the buffer at the same time. Any n bigger than 256 will be accepted in the expression, but it will be "squeezed" onto 256. The resulting value is a string. This command can only be used if [serial data parking](#) has been set to TRUE.

Analyzing your experiment: The Data File

The USBBbox provides a rich set of data that can be relevant for any given experiment. We had to balance the clutter that exposing then all in the data file would have generated with the potential interest of having them in the data file. We decided to follow a middle line, by inserting into the data file a selection that seemed reasonable to us. This means that somebody won't be happy, either because there is too much or because there is too little. We hope to have managed to dissatisfy everybody, so that somebody will feel compelled to improve what we did.

When the USBBbox is activated as an input device, several new columns are recorded in the data file:

PsyScope X B46 started: 10/17/06 9:46:38													
Script file: simple bbox between													
Run on: DualMini													
Ran 5292971													
Input devices active: Key Mouse USBBox													
Timing Device: USBBox													
Trial	Condition	Time	key	mouse_down	UBRelativeT	UBAbsoluteT	UBSystemTS	UBDrift	UBButtons	UBVoice	UBOptic	UBIOPorts	UBQueueLength
1		1034	[N/A]	0	1042	1109	1044	-1	01000000	X	X	0000000000000000	0
2		245	[N/A]	0	244	1359	245	-1	00000000	X	X	0000000001010101	0
3		71	[N/A]	0	80	1445	81	-1	00100000	X	X	0000000001010101	0
4		210	[N/A]	0	209	1662	210	-1	00000000	X	X	0000000001010101	0
5		374	[N/A]	0	382	2051	384	-1	00010000	X	X	0000000001010101	0
6		126	[N/A]	0	126	2183	126	-1	00000000	X	X	0000000001010101	0
7		198	[N/A]	0	207	2394	208	-1	00001000	X	X	0000000001010101	0
8		136	[N/A]	0	135	2534	136	-1	00000000	X	X	0000000001010101	0
9		174	[N/A]	0	183	2723	184	-1	00000100	X	X	0000000001010101	0
10		134	[N/A]	0	134	2863	134	-1	00000000	X	X	0000000001010101	0
11		168	[N/A]	0	178	3045	178	-1	00000010	X	X	0000000001010101	0
12		145	[N/A]	0	145	3194	145	-1	00000000	X	X	0000000001010101	0
13		934	[N/A]	0	943	4143	944	0	10000000	X	X	0000000001010101	0
14		166	[N/A]	0	165	4314	166	0	00000000	X	X	0000000001010101	0
15		867	[N/A]	0	876	5196	877	0	00000001	X	X	0000000001010101	0
16		151	[N/A]	0	150	5353	151	0	00000000	X	X	0000000001010101	0
17		409	[N/A]	0	418	5778	419	0	01000000	X	X	0000000001010101	0
18		142	[N/A]	0	141	5924	142	0	00000000	X	X	0000000001010101	0
19		86	[N/A]	0	93	6022	96	0	00100000	X	X	0000000001010101	0
20		182	[N/A]	0	182	6211	182	0	00000000	X	X	0000000001010101	0
21		49	[N/A]	0	58	6274	59	0	00010000	X	X	0000000001010101	0

In all columns, 0 is OFF, 1 is ON and X is masked.

UBButtons

This column is probably the one that you are most interested in. It contains an array specifying the state of the buttons, from left to right. Nothing strange here.

UBPorts

This column presents the state of the two synchronous ports Port 1 and Port 2.

UBDrift

This column reports the estimated drift, adjourned during the experiment (always by an estimation procedure), between the computer and the Mac clocks. In a separate way, by acting on the [Drift Estimation](#) section of the USBBox window, you can estimate the drift and decide to integrate it into the time values, or else leave things as they are and take care of adding the values later if they are important for your experiment. The column allows you to monitor the differences and take your decisions.

UBVoice and UBOptic

These fields contain the values for the Voice Key and the Optic Key (which does not exist in the first release of the ioLab USBBbox). We decided to treat them as separate fields, as in most cases you will want to analyze them separately. However, do remember that the Voice and Optic keys are two auxiliary standard lines. Thus, in principle, they can be rewired towards other devices, although they will maintain these names in the data file. As usual, 0 is *OFF*, 1 is *ON* and X is *masked*.

UBQueueLength

Internally, the USBBbox has a queue in which all the events that happen are recorded. This buffer is emptied when the data are moved to the computer, which can only happen in intervals determined by the USB communication protocol. In particular occasions, this queue may be full or close to full. When the data overflow the buffer, the program will stop. However, sometimes the buffer may not be completely full and yet have lots of data in it. As the data are stacked in a FILO buffer, the fuller the buffer, the longer the data will take to be extracted. Therefore, it is good practice to check abnormal data against a measure how full the buffer is. *UBQueueLength* offers this measures. In normal conditions, the buffer should be almost empty. The maximum value is 64. Because the current USBBbox firmware can only communicate one data package per exchange, and because every “pass” takes at least 1 ms, a certain value in *UBQueueLength* indicates that it took *at least* that many milliseconds for it to be communicated to the computer; possibly, more, if the USB communication does not go through at its best.

UBRelativeTS, UBAbsoluteTS, and UBSystemTS

To understand these columns you have to go back to the discussion of the timing issues at the beginning of this booklet.

When an experiment begins, the time of the USBBbox is set to zero, just as the internal timer of the computer.

As the two timers take different times to execute the command *Set me to zero*, there is no guarantee that the zero values will be exactly the same. In particular, the USBBbox timer can start counting from zero a bit later than the internal timer, because the *Set me to zero* command must travel through the USB interface.

When a button is being pressed/released, or any other event occurs in the button box, the USBBox immediately timestamps it. We provide two columns that give the USBBox timestamps: the *UBRelativeTS*, which reports the timestamp relative to the beginning of the event that the reaction time is relative to, and the *UBAbsoluteTS*, which reports the timestamp relative to the beginning of the experiment, or more exactly, to the moment in which the USBBox clock has been actually set to zero.

Then the USBBox sends the packet containing the timestamp (and the other information such as what button/line triggered the event) to the computer, via the USB port. On the other side, the Operative system receives the packages. As a mean of control of the accurateness of this passage, we provide another "internal" timestamp: the time at which the system (indeed, the USBBox driver) receives the package. This is the *UBSystemTS*.

Ideally, the difference between the *UBRelativeTS* and the *UBSystemTS* should be on the order of one-two milliseconds. You have to control this in order to see how accurate your whole setup is.

UBRelativeTS and *UBSystemTS* can and should be checked against the *Time* column. This is the time information routinely provided by PsyScope, marked via the internal clock. Differences between *UBSystemTS* and the *Time* column may signal that the operative system is running some process that sneaks in between the arrival of the package from the USBBox and the possibility on the part of PsyScope of seeing that a button/line of the USBBox changed of state. This may be unavoidable in certain conditions, but you have to monitor that this reduces to the minimum.

In general, you have to understand how the computer processes can be reduced to the least possible ones before running your experiment. Remember that the accuracy of your result may crucially depend on your understanding of how your computer and its OS work.

What time to analyze, then? Well, if your experiment only consists of inputs coming from the USBBox, most likely you will want to use the *UBRelativeTS* for your analysis. However, if your experiment has several devices triggering RTs, then the *UBRelativeTS* cannot be sufficient: Obviously, timestamps can only be taken by the USBBox, and so if a mouse click triggers a reaction time in your experiment, then the *UBRelativeTS* field will be empty for that event. In other words,

the only column that will always contain a piece of time information

will be the *Time* column.

In this case, probably the best strategy is to compare the difference between the values contained in the *UBRelativeTS* and the *Time* column, for the RT that are originated from the USBBox, and see what difference exists between the two. If you need to choose between them, do remember that

UBRelativeTS contains time stamps, that is, values that are stamped independently of any delay the computer may introduce. However, *Time* will contain the moment at which PsyScope realizes that something happened, and hence, only after the values recorded at *Time* will any planned action be initiated.

Other Values

According to the needs of your experiment, you can in principle always define a variable, assign it a value relative to one or the others of the settings of the USBBox, and then mark that variable to be saved in the data file in the usual way allowed by PsyScope. If your experiment modifies any of those parameters, or if in any case you want to have a column marking them in the data file for your convenience (e.g., knowing what the minimum signal duration was at the moment in which the RT was taken), you are strongly suggested to do so. If you do not, all the values that the USBBox has at the beginning of the experiment will be saved in the Log file, and you can always retrieve them from there.

A final point: Reference timer and time values

No matter what column you are considering, do remember that you can always align all the values to the internal timer, or else to the USBBox timer, by selecting the *Reference Timer* from the *Experiment* menu entries. As explained above, this is only a reference point, *not* a change of the timer that is being consulted by PsyScope for its operations (which is always the internal clock). However, it will affect all the data recorded into the data file.

Miscellaneous issues

TO BE COMPLETED