BCI practical course : “Hello World” & ERP Viewer

Jason Farquhar <J.Farquhar@donders.ru.nl>

Radboud University Nijmegen

Learning Goals

Understand:

- What is needed to make a BCI, i.e. progress tracking, data acquisition, annotation and processing, stimulus presentation, and an overall process scheduler/sequencer
- How to used event-driven programming ideas coupled to a global shared event pool (blackboard) to provide these facilities
- How the fieldtrip buffer provides the event blackboard which is used for inter-process communication.

Know how to:

- What the struct of an 'event' is and how to use it to annotate data with experiment relevant event information
- present simple visual stimulus/feedback to the user/experimenter
- How to wait for specific events, get the necessary data, process it and post the updated results back to the event blackboard
- Startup the buffer and an experiment control Matlab, and how to connect these processes to provide a basic BCI
- Test your experiment with simulated data generated by the signal-proxy
- Debug your experiment when it fails!

Today's Plan

- Discussion : What do we need to make a BCI?
- Introduction to the Buffer-BCI framework

break

- Hands-on 1: Echo-server (event IPC)
- Hands-on 2: Hello World
- Hands-on 3: Sequenced Sentences

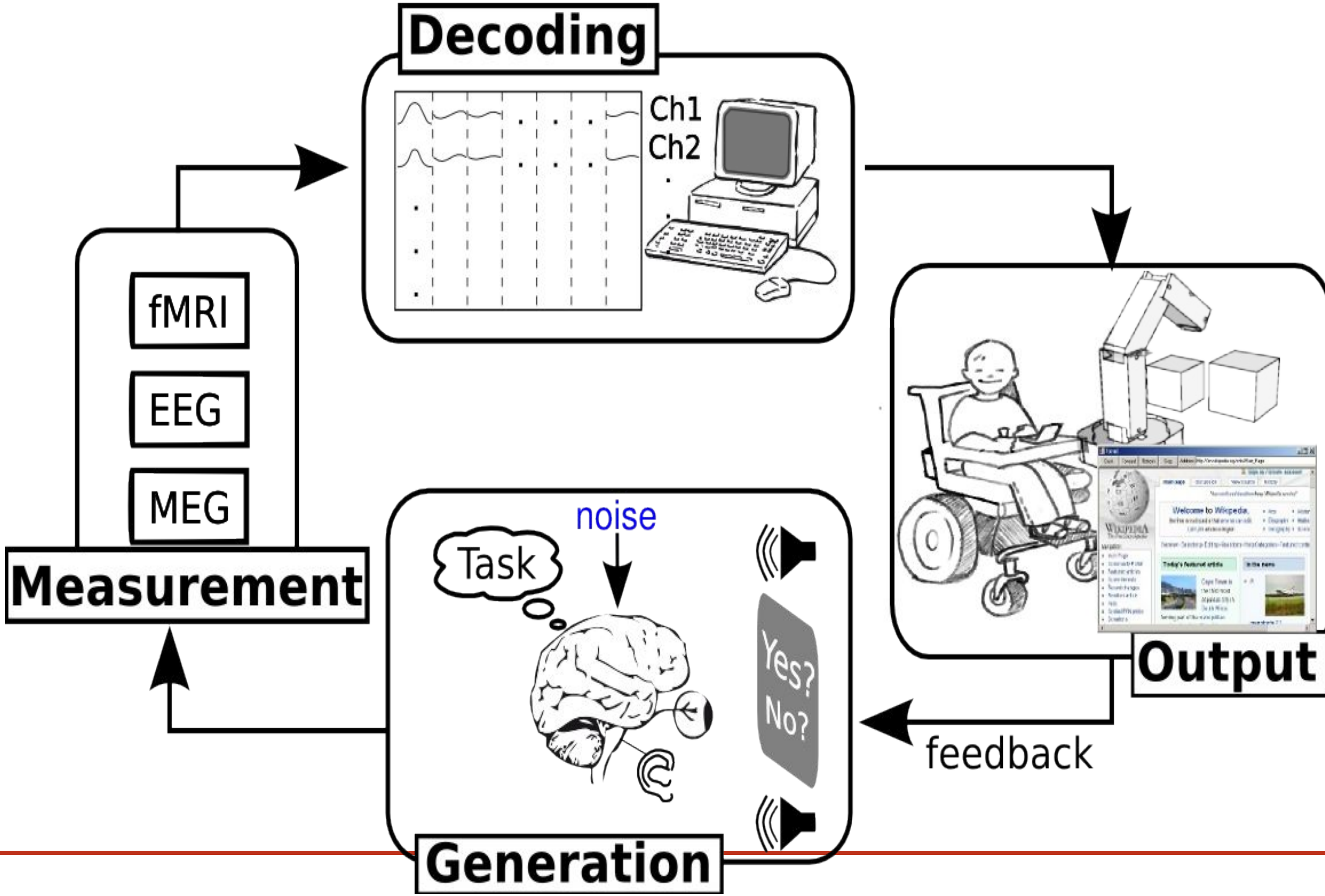
break

- Hands-on 4: Visual ERP Viewer

Discussion: What do we need to make a BCI?

- Based on your prior knowledge and experience with the hands on demo we've did last time.
- Discuss: What do we require to make a BCI system?
- Think about:
  - Hardware requirements?
  - Software requirements?
  - Information flows?

BCI information flow



Gross structure of a typical BCI experiment

Cap Fitting

↓

Offline Training (calibration)

↓

Classifier Training

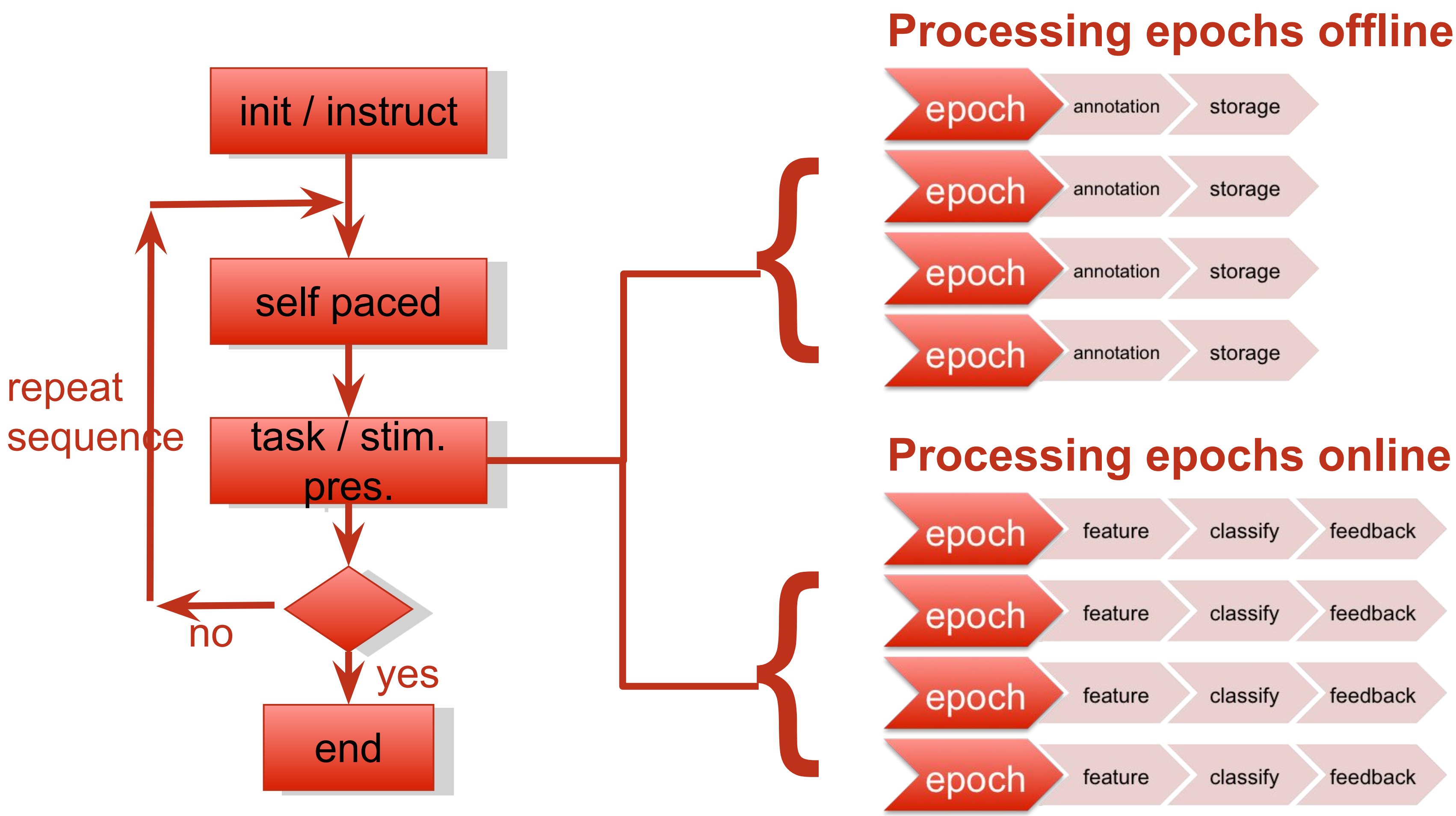
↓

Online Use (feedback)

BCI terminology (our group!)

- Epoch/Trial
  - Single BCI prediction
  - e.g. 1-imagined movement, 1 visual-speller flash
- Sequence
  - Short group of epochs (~1min)
  - v. short breaks 1-2sec between epochs (usually automatic)
  - short (usually self-paced) subject break between sequences (~10sec)
- Block/Run/Phase
  - Short group of sequences (>10min)
  - long (~1-2min) subject break between blocks
  - e.g. cap-fitting, calibration, classifier training, on-line use
- Session – during one cap-fitting
- Experiment – imagined movement, visual-speller etc.

Flow chart of an individual epoch in a simple BCI experiment



Requirements: what do you need to build a BCI?

1) Way of **tracking** where we are in execution of the experiment flowchart, i.e. block, sequence, epoch number.
2) Way of **annotating** data to what the subject was experiencing/doing at that time with what was measured from their brain/body, e.g. LH movement, reading instruction, watching queue, etc.

3) **Data acquisition**: Drivers to extract data from hardware (and combine data fro different hardware sources)
4) **Stimulus Generation**: makes stimuli that the subject will experience, for subject instruction, feedback, event-related stimuli
5) Something to **process** the signals, firstly to train the classifier, and secondly to decode the users mental state, i.e. do the BCI bit ;-)
6) **Scheduler** (sequencer?) to tie it all these bits together,
   - so the correct functions, i.e. stimulus display, signal processing, are executed
   - at the correct position in the experiment flowchart
   - based on the right bits of measured datacc

Summary

- To build a BCI we need a system to; track our progress through the experiment, acquire, annotate and process data, present the stimuli and schedule all these processes in an appropriate way.
- Next we introduce a Matlab based system which provides these facilities.

Overview – “buffer_bci” framework

- Open-source BCI development environment
- Available from Github: www.github.com/jadref/buffer_bci
- Core is “[fieldtrip-buffer]()”
  - Hardware access, data-storage, IPC
  - Platform independent: Win, Mac, Linux, (Andriod/iOS)
  - Language Neutral: C, C++, Matlab, Python, C#, java
- Matlab/Octave/Python/Java based BCI examples

Overview – “buffer_bci” framework

- Open-source BCI development environment
- Available from Github: www.github.com/jadref/buffer_bci
- Demos:
  - – Brain Viewer
  - – Games : Snake, Pacman, Sokoban
  - – Visual Speller
  - – Thought-buttons
- Tutorial:
  - – How a BCI works and how to build one

github SOCIAL CODING

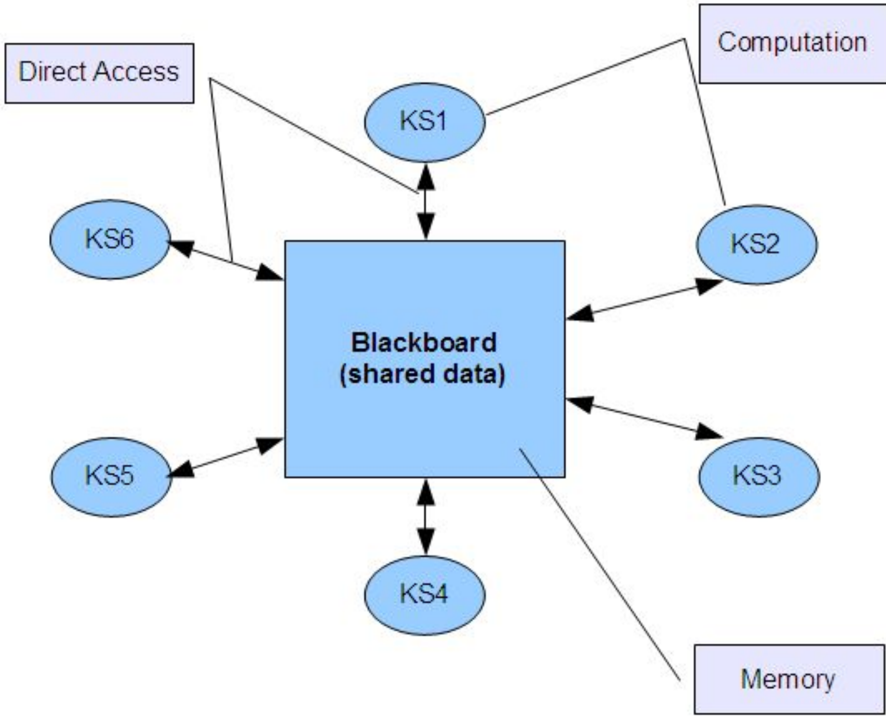
Buffer-BCI Framework

- We can break the requirements into 4 largely independent communicating processes:
  1) Data-acquisitation & annotation
     - Get data from hardware
     - Attach annotations (markers, events) to particular data sample
  2) Experiment control (scheduling)
     - Control the flow of the experiment
  3) Stimulus generation
     - Make stimuli when requested by the expt controller
     - Make feedback based on predictions generated by the sig-processor
  4) Signal Processing
     - Process the data based on the annotationss, and generate predictions

Buffer-BCI Framework

Basic Idea:

- set of independent processes
- any process can send/recieve data-annotation events
- events are visible to all other processes
- Processes communication implemented by sending recieving events
- (N.B. As all events are saved with the data, annotations are automatically archived for later off-line use).
- Similar in concept to that used in 'Blackboard architectures' for AI, see <en.wikipedia.org/wiki/Blackboard_system>



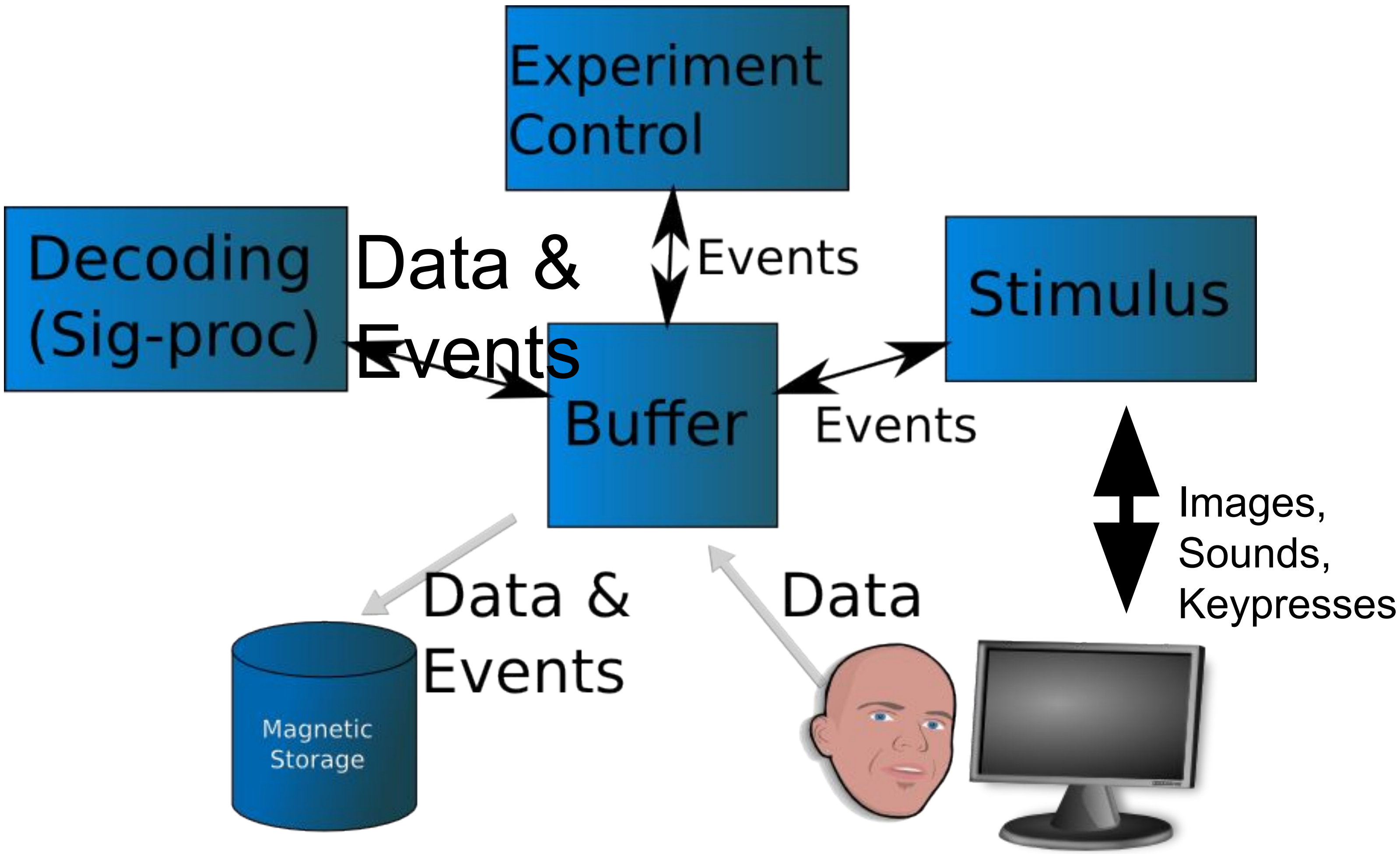
Ft-buffer based Implementation

- Buffer-BCI framework implemented using the fieldtrip-buffer system (fieldtrip.fcdonders.nl/development/realtime)
- Ft-buffer provides:
  - Drivers for data-acquisation
  1) buffer storage for data (~last 1 minute data)
  2) buffer storage for events (~last 50 events)

Idea:

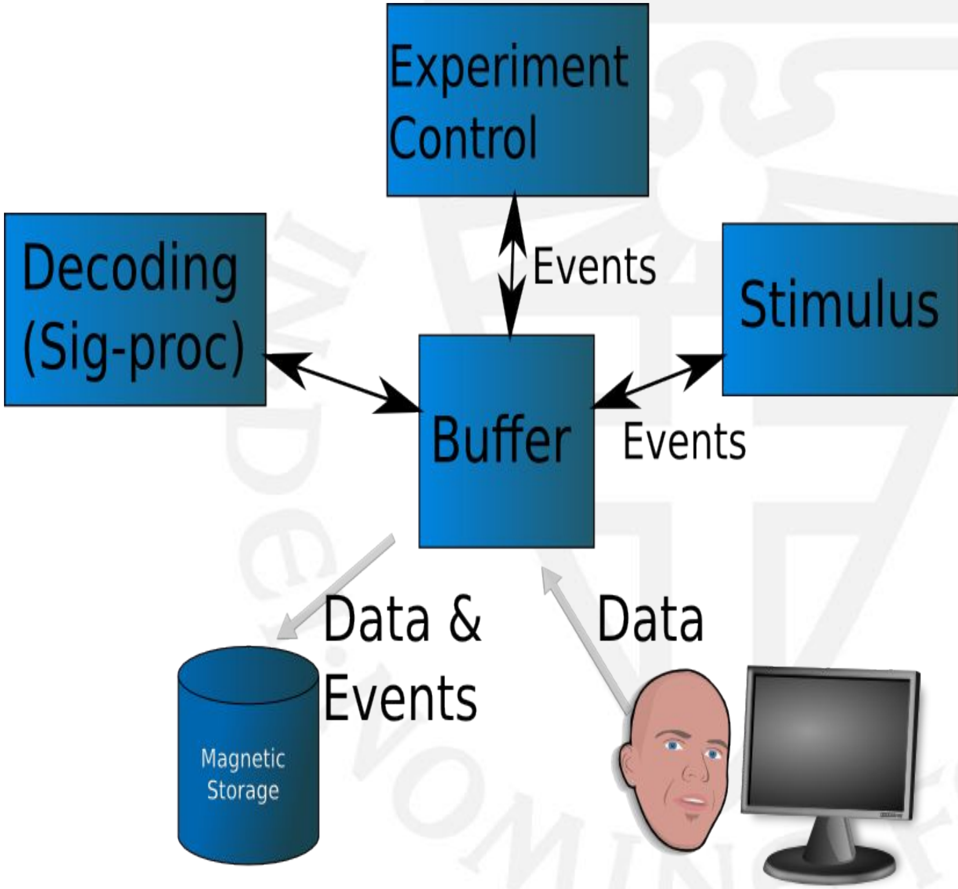
- Buffer events store represents the blackboard used for inter-process communication (IPC)
- Every event has timestamp (sample number) used for data-annotation

Typical Structure of BCI system



client/server architecture

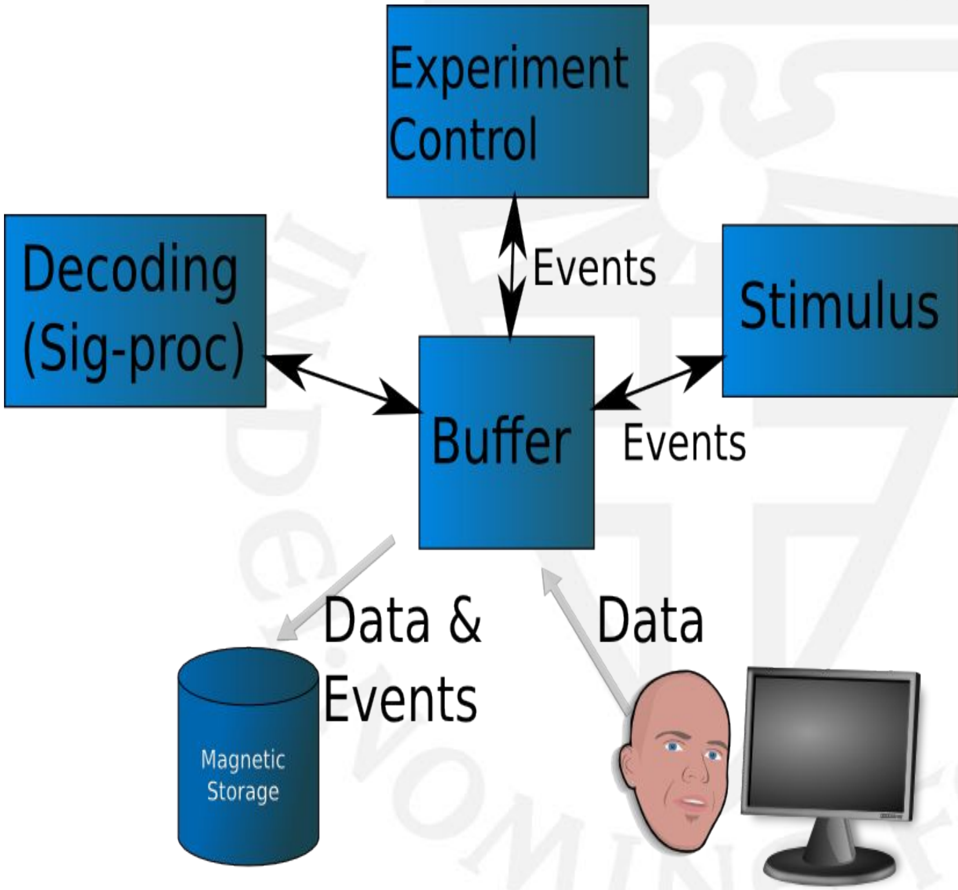
1) **Buffer server**: for data storage, annotation (IPC) and archive
2) **Acquisition driver**: to access (or simulate) the measurement hardware
3) **BCI Application**.
4) Normally, split into 2/3 pieces:
    1) Signal-processing
    2) Stimulus presentation
    3) Experiment control
    4) (commonly part of stimulus-presentation)



Buffer_bci Quickstart

client/server architecture

1) **Buffer server**: for data storage, annotation (IPC) and archive
2) **Acquisition driver**: to access (or simulate) the measurement hardware
3) **BCI Application.**
4) Normally, split into 2/3 pieces:
   1) Signal-processing
   2) Stimulus presentation
   3) Experiment control
   4) (commonly part of stimulus-presentation)



Running buffer-bci code:

You need to have (at least) the following processes running:

1) Buffer server
2) Data-acquisation system (real or simulated)
3) BCI Application

1) Buffer Server

- Saving server (saves all data and events)
- `dataAcq/startJavaBuffer.{sh/bat}`
- Server (saves nothing): `dataAcq/startJavaNoSaveBuffer.{sh/bat}`

C-based implementations:

- Saving: `dataAcq/startBuffer.{sh/bat}`
- NoSaving: `dataAcq/startNoSaveBuffer.{sh/bat}`

2) Data-acquisation

Simulated data: (for debugging/testing)

- Java-based: `dataAcq/startJavaSignalProxy.{sh/bat}`
- (Alternative implementions Matlab: `startMatlabSignalProxy`, c:`startSignalProxy`)

Real data: (for debugging/testing)

- FilePlayback: `dataAcq/startJavaFileProxy.{sh/bat}`
- Microphone: `dataAcq/startAudio.{sh/bat}`

EEG amplifier data: (for real usage)

- Biosemi: `dataAcq/startBiosemi.{sh/bat}`
- Mobita: `dataAcq/startMobita.{sh/bat}`
- OpenBCI: `dataAcq/startOpenBCI.{sh/bat}`
- Many others... (look in `dataAcq/startXXXXX.{bat/sh}`)

3) BCI-Application

- Generally what you will write yourself in this course ;-)
- Typically in 2 (or more) separate processes:

1) Stimulus presentation:
   - Controls what the user sees/hears, i.e. UI
     - Show App, show feedback, control output-devices, etc.

2) Signal processing:
   - Processes the EEG data and events to generate predictions for UI feedback.
     - Collect calibration data, train classifier, generate predictions

Example : Games-demo

1) Buffer Server:
   1) `dataAcq/startJavaNoSaveBuffer.{sh,bat}`
2) Data Acquisation (simulated) :
   1) `dataAcq/startJavaSignalProxy.{sh,bat}`
3) BCI Application(1) -- Stimulus Presentation:
   1) `games/runGame.{bat,sh}`
4) BCI Application(2) – Signal Processing:
   1) `games/startSigProcBuffer.{bat,sh}`

Quickstart Example : Games-demo

1) Core Quickstart: Server + Acquisation + EventViewer:
   1) Debug mode (simulated data):
   2) `debug_quickstart.{sh,bat}`
   3) EEG Mode (mobita source):
   4) `eeg_quickstart.{sh,bat}`

2) BCI Application(1) -- Stimulus Presentation:
   1) `games/runGame.{bat,sh}`
3) BCI Application(2) – Signal Processing:
   1) `games/startSigProcBuffer.{bat,sh}`

Useful (debugging) Functions: EventViewer

- Seeing the what events are sent and when is important for debugging experiment flow.
- Basic event viewer has been implemented in multiple languages
  - MATLAB*:*
    - Quickstart: `dataAcq/startMatlabEventViewer.{sh,bat}`
    - Source code: `matlab/utilities/eventViewer.m`
  - JAVA :
    - Quickstart: `dataAcq/startJavaEventViewer.{sh,bat}`
    - Source code: `java/echoClient/eventViewer.java`
  - Python:
    - Source code: `python/echoClient/eventViewer.py`
  - C :
    - Source code: `c/echoClient/eventViewer.c`

Hands-on 1: Event Echo-Server

Events for IPC

- As well as being used for data annotation, Events are used for inter-process-communication,
  - e.g. to communicate results from signal-processing to stimulus presentation
- To use events in this way, each process must
  - monitor for new events
  - filter out the events it should react to
  - send response events

(Key concept) event structure

sample
- time at which event occurred in samples from start of experiment

type
- arbitrary event type (usually a string)

value
- arbitrary event value (usually string or number)

duration (optional)
- duration of the event in samples

offset (optional)
- zero-time for the event.
- Usually, offset from sample at which the event actually started.

Examples:

Visual speller “flash”;
```
ev=struct('sample',123,...
          'type','stimulus.flash',...
          'value',[0 0 1 0 0],...
          'offset',0,'duration',0)
```

Classifier prediction:
```
ev=struct('sample',123,...
          'type','prediction',...
          'value',[-1 -1 -1 1 -1],...
          'offset',0,'duration',0)
```

Imagined Movement event:
```
ev=struct('sample',123,...
          'type','stimulus.move',...
          'value','left-hand',...
          'offset',0,'duration',300)
```

Compact notation:
```
s:123,t:'stimulus.flash',v:[0 0 1 0 0],o:0,d:0
```

(key functions) Event manipulation

evt=mkEvent(*type,value,[sample,offset,duration]*)

- make a buffer event, with sensible defaults

sendEvent

- *evt*=sendEvent(*type,value,[sample,offset,duration,host,port]*)
- *evt*=sendEvent(*evt*,[host,port])
- Send event to the buffer on machine host at port.

mi=matchEvents(*evts,mtype,mval*)

- Find events with type *mtype* and value *mval* in *evts* a vector of event structures.
- *mtype* – can be cell-array of types to match, e.g. {'type1' 'type2'}
- mval – can be cell-array of values to match, e.g. {'val1' 10 'val3'}
- match if any *mtype* matches and any *mval* matches,
  - i.e. above matches (t:'type1',v:10), (t:'type2',v:10),(t:'type1',v:'val1')

- mi is logical vector of which evts matched
- N.B. Empty ([]) or '*' mtype/mvalue matches everything

(key functions) Event manipulation

`import Fieldtrip`

evt=Fieldtrip.Event(*type,value,[sample,offset,duration]*)

- make a buffer event, with sensible defaults

sendEvent

- *evt*=bufhelp.sendEvent(*type,value,[sample,offset,duration,host,port]*)

evtfilter=bufhelp.createeventfilter((*mtype,mvalue)*)

- Create a function to filter a list of events and return only events with with type *mtype* and value *mval* in *evts* a vector of event structures.
- *Can also use a list of (mtype,mvalue) pairs to match multiple types, e.g. [('type1','val1'),('type2','val2')]*
- *Or a list of strings to only match on the type: e.g. ['type1','type2']*
- *Apply the resulting filter function:*

mevts=evtfilter(evts)

Hands-on 1: Event Echo-Server

Experiment Task

- Write a simple echo-server which:
  - Connects to the Buffer server
  - Waits for **any** incoming event, and
  - Responds by sending a 'echo' event with the same value but type='echo'
  - Quits if it receives an event with type='exit'
- N.B. Don't 'echo' your own echo events!

Assignment:

- start from : `echoServer_skel.{m/py}`

N.B. send 'keyboard' events by pressing keys in the eventViewer window.

Useful Functions:

- Setup MATLAB paths and connect to buffer:
  - See the header code block in echoServer_skel.m
  - This will initialize the paths:
    - `run ../../utilities/initPaths.m`
  - Then try to connect to the buffer server until valid header is returned.
    - `while ( isempty(hdr) ||.....`
  - Then initialize some utility functions for high-precision timing
    - `initsleepSec();initgetwTime();`

Useful Functions:

[devents,state]=...
buffer_newevents(*host*,*port*,*state*,*mtype*,*mval*,*timeout_ms*)

- wait for any new events matching (*mtype*,*mval*)
  - Matching done by *matchEvents*
    - *mtype* – can be cell-array of types to match, e.g. {'type1' 'type2'}
    - *mval* – can be cell-array of values to match, e.g. {'val1' 10 'val3'}
    - match if **any** mtype matches **and any mval** matches
- return the matched events in the vector of structure(s) *devents*
- *state* is the match state, used to track which events have been processed between function calls
- Return after timeout_ms milliseconds even if no matching events found

Useful Functions:

- Setup PYTHON paths and connect to buffer:
  - See the header code block in echoServer_skel.py
  - This will initialize the paths:
    - `bufhelpPath = "../../python/signalProc"`
    - `sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)),bufhelpPath))`
  - Then try to connect to the buffer server until valid header is returned.
    - `ftc,hdr=bufhelp.connect();`

Useful Functions:

devents=bufhelp.buffer_newevents(*mtype,timeout_ms*)

- wait for any new events matching (*mtype*)
  - Matching done by *matchEvents*
    - *mtype* – can be list of type strings to match, e.g. ['type1', 'type2']
    - match if **any** mtype matches **and any mval** matches
- return the matched events in the vector of structure(s) *devents*
- *state* is the match state, used to track which events have been processed between function calls
- Return after *timeout_ms* milliseconds even if no matching events found

Debug/Test your echoSever

- Start an event viewer: `dataAcq/startJavaEventViewer.{sh,bat}`
- Or directly buffer+simulated EEG+eventViewer: `debug_quickstart.{sh,bat}`
- Generate events to be echo'ed:
  - Directly in the javaEventViewer.
    - Press return
    - Enter the event type, e.g. 'hello'
    - Press return
    - Enter the event value, e.g. 'world'
    - Press return
  - You should now see a copy of your created event in the event log.
  - If the your echoServer is working you should also see your 'echoed' version.

Echo-Server in different languages

Basic echo-server example has been implemented in multiple languages

- MATLAB: `echoClient/matlabclient.m`
- JAVA : `java/echoClient/javaclient.java`
- C# : `csharp/echoClient/csharpclient.cs`
- Python: `python/echoClient/pythonclient.py`
- C : `c/echoClient/cclient.c`

Hands-on 2: “Hello World”

Experiment Task

- Display the string “Hello World” (or any other pre-specified string) on the screen, and wait for a key to be pressed to exit
- Send events to annotate what has happened, e.g. startup, string display, key-pressed, shutdown etc.

Method:

- Start from the 'helloworld_skel.{m/py}' function skeleton
  - contains initialisation code to connect to the ft_buffer
  - Some examples of functions you may find useful

Note : event timestamps

- Accurate event time-stamps are critical for evoked potential analysis
  - >10ms event jitter causes significant reduction in signal quality
- However,
  - data-acquisation may only send data every >20ms
  - And this data may be subject to additional network delays of >20ms
- Stop this jitter reducing time-stamp accuracy by;
  - aligning (and tracking) computers real-time-clock and data-sample clock to prevent this jitter reducing

Hands-on 3: Sequenced Sentences

Experiment Task

- display set of sentences on the screen where every second 1 more character gets added to the sentence
- pause for 5 seconds between sentences (and/or wait for key press)
- send events for everything that happens

Assignment:

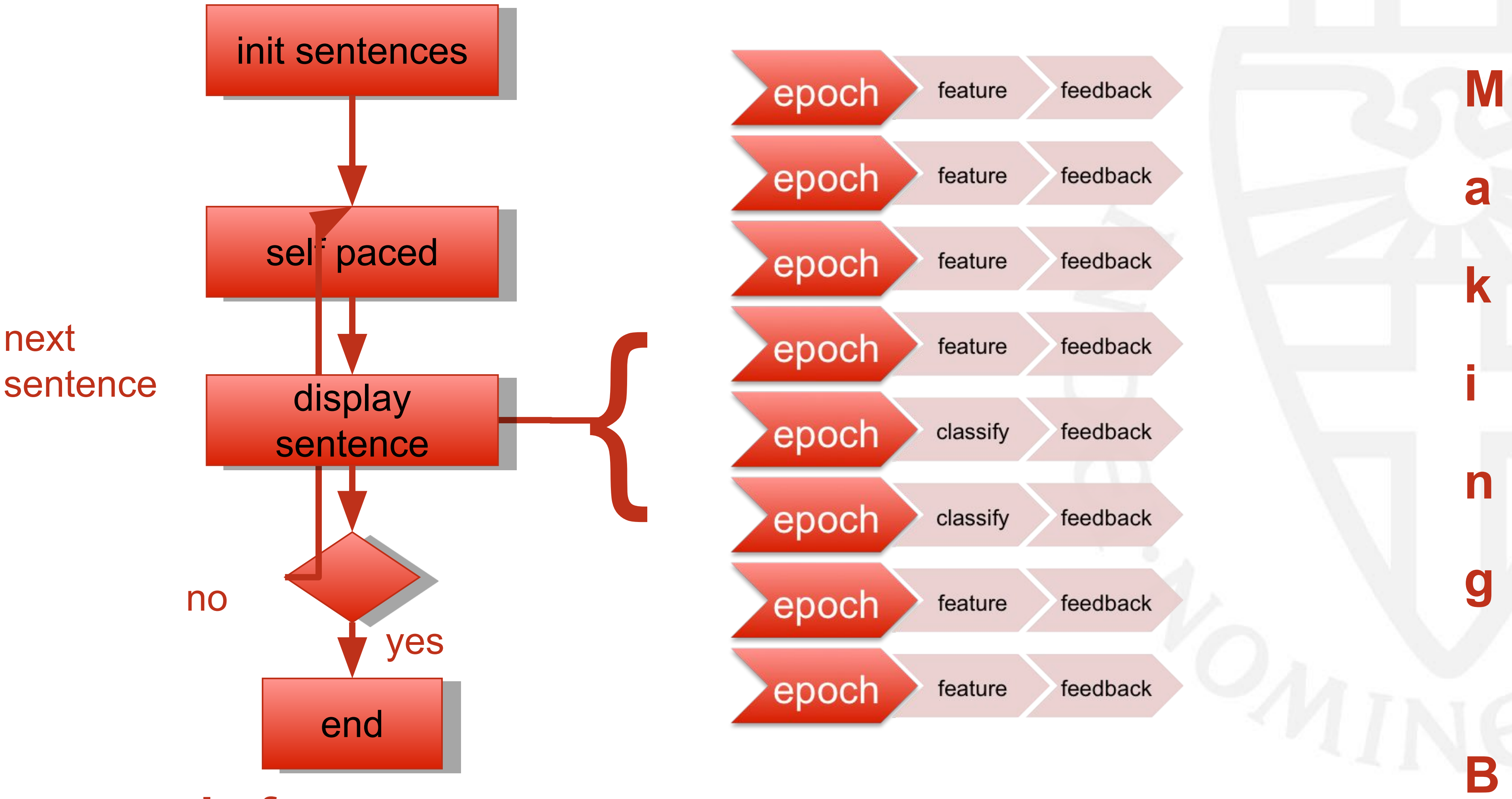
- Make flowchart
- Write code -> test -> debug -> until it works :-)
- Start from runSentences_skel.{m/py}

Useful Functions:

- MATLAB: sleepSec(time) sleep for the indicated duration in second
- PYTHON: sleep(time) sleep for the indicated duratin in seconds

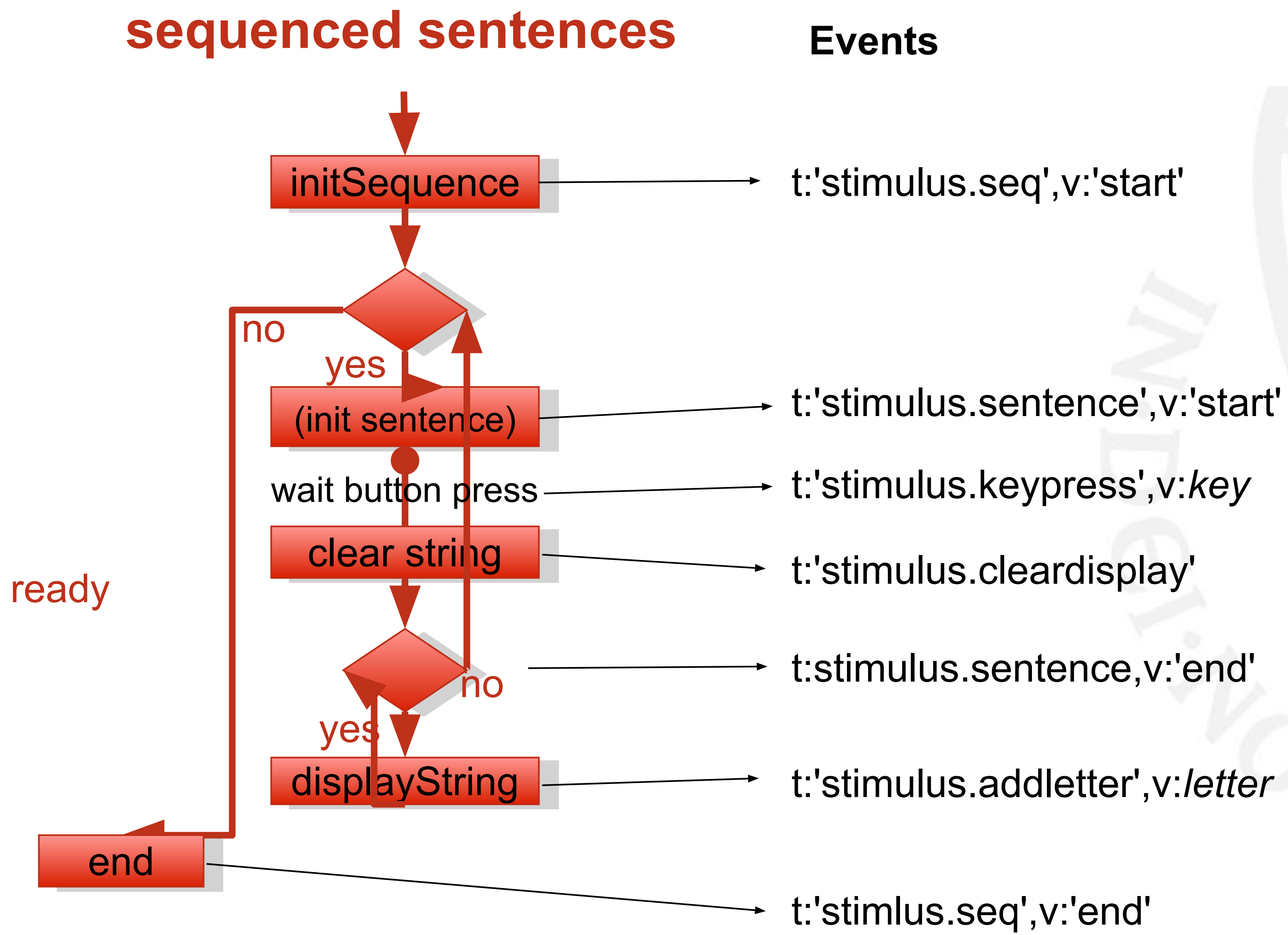
Flowchart : sequenced sentences

sequenced sentence **Sentence** **Stimulus presentation**



end of sequence

.Events and processing functions



Hands-on 4: ERP Viewer

Experiment Task

- In 5 sequences of 10 seconds:
  - Every 1 seconds: either randomly display or don't display a cross (+) on the screen for 200ms
- Display a 'Press key to continue string' between sequences, and wait for key press to move to the next sequence
- For every 'stimulus event', i.e. point when the '+' could have been displayed, record 600ms of data annotated with whether it was a '+' or not
- Every time you get some data, compute an average of the EEG data for that type of stimulus, i.e. + or no-+, and display the resulting averages as a multi-plot on the screen
-
- N.B. You will need a separate signal processing process to: get the data, compute the ERP and display the results!
- Assignment:
- Make flowchart for each of the processes, i.e. stimulus, and signalProcessing
- For the expt-control & simulus presentation start from : runStimulus_skel.{m/py}
- For the signalProcessing & results generation use : runSigProc_skel.{m/py}

Useful Functions:

[data,devents,state]=buffer_waitData(*host*,*port*,*state*,...
'startSet',*startEvts*,'trlen_samp',*samp*,'exitSet',*exitEvts*)

- for all events matching *startEvts* record *samp* samples of data
- until an event matching *exitEvts* is generated
- *startEvts* and *exitEvts* specify the events to match in the format:
  - type – event type has this value
  - {'type' val} – event has type==type and value==val
  - {{'type1' ''type2'}} – event has type == 'type1' or 'type2'
  - {{'type1' 'type2'} {val1 val2}} – event has type == 'type1' or 'type2' **and** value== val1 or val12
- return the matched event structure(s) in *devents* and corrospending data in *data*
  - *Data* is a vector of structures. data.buf = [nChannels x nSamples] raw EEG data
- *state* is the match state, used to identify which events have been processed between function calls
-
- N.B. *ExitEvts* has the special event type 'data' which returns as soon as the data is available for the first matched *startEvt*

Useful Functions:

[data,devents,stopevents]=bufhelp.gatherdata(trigger,trlen,exitTrigger)

- for all events matching *trigger* record *trlen* samples of data
- until an event matching *exitTrigger* is recieved
- *trigger* and *exitTrigger* specify the events to match in the format used for `bufhelp.createeventfilter`. e.g.
  - [type1, type2,...] - list of types to match
  - [(type1,val1),(type2,val2)...] - list of type+val pairs to match
- return the matched event structure(s) in *devents* and corrospending data in *data*
- *Data* is a vector of structures. data.buf = [nChannels x nSamples] raw EEG data

DEBUGGING: ERP Injection

- Debugging the correct signal-analysis script is difficult without a **true-signal** to validate that the trigger event timing/value is correct.
- To make this easier the **simulated-eeg** data supports **ERP-injection**.
  - This allows the presentation system to ‘tell’ the EEG simulator to add an erp to the **TRG** channel at this point.
  - This can then be used to check the correctness of the signal processing, e.g. do all the ‘flash’ events have a trigger ERP at time=0? do all the ‘non-flash’ events have no ERP?

DEBUGGING: ERP Injection

- ERP injection works by sending a number for the ERP strength on UDP port 8300.

- Setup the socket connection:

```
trigsocket=javaObject('java.net.DatagramSocket');
trigsocket.connect(javaObject('java.net.InetSocketAddress','localhost',8300));
```

- Use the socket to inject an ERP

```
trigsocket.send(javaObject('java.net.DatagramPacket',int8([10]),1));
```

DEBUGGING: ERP Injection

- ERP injection works by sending a number for the ERP strength on UDP port 8300.

- Send a number to inject an ERP:

```
import socket

socket.socket(socket.AF_INET,socket.SOCK_DGRAM,0).sendto(bytes(1),('l
ocalhost',8300))
```

Summary

- BCI can be broken into 4 processes: data-acquisation, experimental control, signal processing, and stimulus presentation
- buffer_bci framework : uses buffer events as a blackboard for inter-process communication