MEABench

Daniel Wagenaar

Release 1.2.5, March 2011

MEABench is a set of command line and GUI utilities to process data from the Multi Channel Systems MEA60 amplifier. Adaptation for different hardware should be straightforward. MEABench was designed with extensibility in mind, and is fully modular. That means that filters can be inserted anywhere in the data processing stream. This document describes the basics of the toolset and the usage of the individual programs.

Contents

Chapter 1

Introduction

MEABench is an open-source suite of programs for acquisition and analysis of multi-electrode array (MEA) recordings. MEABench was developed by Daniel Wagenaar at Caltech, drawing on the excellent example set by MultiChannel Systems' MC_Rack suite[1].

The software runs under Linux and other Unix variants, and is freely distributable under the terms of the GNU Public License (see `http://www.gnu.org/copyleft/gpl.html`). It offers the following functionality:

- Acquisition of raw electrode data from MultiChannel Systems' MCCard;
- Complete removal of mains (60 Hz) interference using template filtering;
- Removal of stimulation artifacts using the SALPA algorithm[2];
- Online and offline detection of spikes;
- Online visualization of electrode data and spikes;
- Online sonification of spikes;
- Continuous or windowed saving of raw data and spikes;
- Saving of spike waveforms, for later spike sorting and analysis;
- Replaying of raw and spike files, at any speed;
- Instant-replay buffer for easy analysis of recent events;
- Online generation of raster plots;
- Continuous monitoring of varying noise levels;
- A variety of utilities for analysis and data format conversion, including:
  - Averaging of electrode recordings over trials;

[1] See `http://www.multichannelsystems.com`.

[2] D. A. Wagenaar, and S. M. Potter: Real-time multi-channel stimulus artifact suppression by local curve fitting. *J. Neurosci. Meth.* **120:2**, 2002, pp 113–120. This, and most other publications mentioned here, may be obtained from my website, `http://www.biology.ucsd.edu/~dwagenaa/pubs.html`.

  - Conversion of binary spike files to ASCII representation;
  - Filtering of spike files based on any mathematical expression involving shape or timing parameters;
  - Extraction of single channels from 64 channel streams;
  - Splitting of long data files into trials;
  - Splitting of long data files into channels;
  - Computing spike rates;
  - Detecting culture-wide bursts.

- Matlab functions to import MEABench data[1];
- A program to allow easy scripting of MEABench modules for offline processing.

MEABench is fully modular, and any user with some Unix programming experience can extend it to fit her or his needs. Since MEABench can stream live data to your extension modules, it is well suited, for example, to drive real-time feedback systems. In fact, the ability to communicate with other software or hardware in real-time was one of the primary motives for the conception of MEABench. It allowed a reliable, sub-100 ms feedback loop time in our Neurally Controlled Animat[2,3].

MEABench was written primarily for use with the MultiChannel Systems MEA hardware, and a driver is included for their MCCard data acquisition card, written by Thomas B. DeMarse with advice from MultiChannelSystems. If you use different data acquisition hardware, you may still find MEABench useful, because, due to its modular nature, it is possible to write plug-in modules to read data from your hardware. An experimental driver for one such board (manufactured by United Electronic Industries, but not endorsed by us at its state of development as of Nov 2002) is included as well.

MEABench has been in constant use in the Pine lab at Caltech for over four years, and at Steve Potter's group at Georgia Tech[4] since its beginning. It is also used by several other research groups in the US, Europe and Asia. MEABench remains a work in progress; we welcome suggestions for improvement (and bug reports). Please join in the development by submitting your code (patches and improvements) for inclusion in future releases.

Another introductory article about MEABench was presented at the IEEE EMBS Conference on Neural Engineering[5].

1.1 Conventions

Throughout this document, the names of MEABench programs are typeset **Like this**. When running MEABench programs, the capitalization should be omitted. Commands defined

[1] Users may also be interested in Uli Egert's comprehensive set of matlab code for MEA data analysis; freely available at `http://www.brainworks.uni-freiburg.de/projects/mea/meatools/overview.htm`.

[2] T. B. DeMarse, D. A. Wagenaar, A. W. Blau, and S. M. Potter: The neurally controlled animat: Biological brains acting with simulated bodies. *Autonomous Robots* **11**, 2001, pp 305–310.

[3] D. A. Wagenaar, and S. M. Potter: A versatile all-channel stimulator for electrode arrays, with real-time control. *J. Neural Eng.* **1**, 2004, pp 39–44.

[4] Public website: `http://www.neuro.gatech.edu/groups/potter/index.html`

[5] D. A. Wagenaar, T. B. DeMarse, and S. M. Potter: MEABench: A toolset for multi-electrode data acquisition and on-line analysis. *2nd International IEEE EMBS Conference on Neural Engineering*, Arlington, VA, March 16–19, 2005.

within MEABench programs are set **like this**. The names of MEABench streams look *like this*, and their types LIKE THIS. **Ctrl-C** means holding the Ctrl key while pressing 'C'.

In examples, text you type is set `like this`. Text the computer produces is set like this, with prompts highlighted **like this**. The unix prompt is represented as **$**.

In definitions of MEABench commands, parameter names are written *like this*, optional paramaters are enclosed in brackets [*like this*], and alternatives are separated by a vertical pipe: *this*|*that*.

1.2 Acknowledgments

I'd like to acknowledge valuable input and support from Steve Potter, Tom DeMarse and Jerry Pine. We are all grateful for financial support from the NIH-NINDS and the Burroughs-Wellcome Fund, and for cooperation, technical support, and equipment from MultiChannel Systems. Portions of MEABench were written by Tom DeMarse, Ryan Haynes, John Rolston, and Brookes Poo.

Chapter 2

Compilation and installation

If you intend to use MEABench with MultiChannel Systems hardware, installation is very straightforward. The following is a step by step guide.

- Make sure you have gcc 2.95 or later.
- Make sure you have Qt 3.0 or later. (Qt 4 or later will *not* work. Most linux distributions allow you to install Qt 3 and Qt 4 side by side.)
- Make sure your kernel is 2.4.10 or later. (Kernel 2.6.x recommended.)
- Download the latest version of MEABench, and unpack it:

  ```
  $ tar xzf meabench-1.2.5.tar.gz
  ```

- Enter the directory:

  ```
  $ cd meabench-1.2.5
  ```

- Choose the directory were you want to install MEABench, e.g., /opt/meabench, and configure:

  ```
  $ ./configure --prefix=/opt/meabench --with-hardware=mcs
  ```

  (The --with-hardware chooses the particular DAQ hardware you are using. Currently defined values are `mcs` for MultiChannel Systems' original MCCard, `mcsE` for "revision E" or later of MCCard[1], `uei` for United Electronic Industries' PD2-MF64-14H cards, and `ni` for National Instruments cards. These last two drivers are currently "experimental". I would particularly like to hear your success stories or bug reports.)

- Compile:

  ```
  $ make
  ```

  This will take a while. You may see various compiler warnings, such as:

  ```
  SD_BandFlt.C:97: warning: assignment to 'short int' from 'float'
  ```

[1] If you don't know which version you have, first try `mcs`. If the data look good in **scope**, you are set. If not, try `mcsE` instead.

These can safely be ignored. Actual compiler *errors* are another story, of course: If you see something like:

```
SD_BandFlt.C:52: 'sqr' undeclared (first use this function)
make: *** [SD_BandFlt.o] Error 1
```

you have discovered a bug, which I would like to hear about (see *Reporting bugs* in chapter 8.

- If compilation went well, you may now complete the installation by typing:

  ```
  $ make install
  ```

  Depending on the installation location you have chosen using `--prefix=...`, you may have to become super user (root) before executing `make install`.

- If the installation location you chose is in your *path* already, you can run MEABench programs simply by typing their name. If not, you can either modify your path variable, or copy the program **mea** from the MEABench 'bin' directory to a location in your path. That will allow you to run MEABench programs by typing, e.g., `mea rawsrv`.

- If you are using kernel 2.6.x, `make install` will also have installed the MCCard driver in the /lib/modules tree, so you can load the driver simply by typing `modprobe MCCard`. (This probably requires super user privileges.)

- If you are using kernel 2.4.x, you should manually copy MCCard-linux-2.4/MCCard.o to an appropriate system location so that `modprobe` can load it. Alternatively, you can use `insmod` .

You may have noticed that the installation procedure closely matches the standard GNU style. Generic information about GNU style installation is provided in the file INSTALL in the top directory of the MEABench source tree.

Chapter 3

Examples

In this chapter you will learn how to do most common tasks with MEABench; subsequent chapters will provide a reference guide to individual components. I will assume that you have already compiled MEABench and installed it in a place where your unix shell can find it.

3.1 Ensuring that the driver is loaded

[This assumes you are using the MultiChannel Systems driver; for other drivers, contact the author of that specific driver.] Start by checking that all hardware is connected properly. Then open a terminal window, and type `lsmod`. If 'MCCard' is listed in the resulting output, you are good to go. Otherwise, try typing `modprobe MCCard`. (You will likely have to run that as super user; use `su` or `sudo` and be aware of the responsibility associated with super user privileges.) If that gives an error (e.g. the system cannot find the file), `cd` to the directory where you built MEABench, then type `insmod MCCard-linux-2.6/MCCard.ko` if you are using a 2.6.x kernel, or `insmod MCCard-linux-2.4/MCCard.o` if you are using a 2.4.x kernel. After all this, type `lsmod` again. This time, 'MCCard' should be listed. If not, contact the author.

3.2 Displaying electrode traces on-line

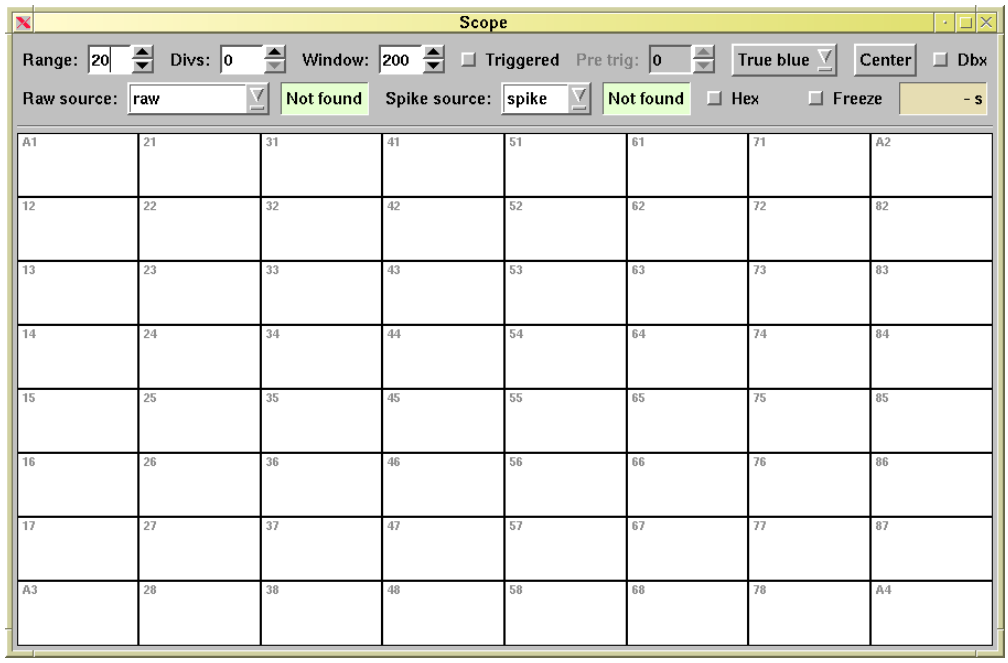
This example shows how to display incoming data fom an MEA in real time.

Start by checking that all hardware is connected properly, with an MEA locked in the pre-amplifier. Then open two terminal windows.

- Run the visualization program, **scope** from the first terminal:

  ```
  $ scope
  Error from Sockclient: Constructor failed at connect [Connection refused]
  Error from Sockclient: Constructor failed at connect [No such file or directory]
  ```

  The warnings are normal, and indicate that the scope is not yet 'connected' to any input. We will fix that in a moment. Let's first take a look at the scope window:



The top area contains a lot of controls, which you will learn to use gradually. The main part of the window displays an 8 by 8 grid of rectangular panels, blank for now, which will show electrode traces in a moment. The panels are layed out in the same shape as the physical electrodes on an MCS MEA, with the corner positions occupying the auxiliary channels A1 through A3. The bottom right corner is not connected to anything on our data acquisition board. If you are using a 'hex' MEA, you can click the 'Hex' button in the second row of controls to change the layout appropriately.

- Start the core data acquisition program, **rawsrv** in the second terminal:

```
$ rawsrv
This is rawsrv, compiled for use with the following hardware:

Pre-amp: MEA1060 by MultiChannel Systems
A/D: MCCard by MultiChannel Systems
Driver: MCCard.o by Thomas DeMarse
Plugin: MCS by Daniel Wagenaar

rawsrv>
```

The information printed by **rawsrv** may be different if you configured MEABench for different hardware.

- Put the cursor in the 'Raw source' box of **scope**, and hit return, or re-select *raw* from the pull-down menu. The label 'Not found' should be replaced by 'New/Ready', indicating that the scope and **rawsrv** are now connected. You will also notice that the graphs now are adorned with a zero-line.

- Return input focus to the terminal with **rawsrv**, and set the data acquisition running:

```
rawsrv> run
Gain setting is 2 (+/- 0.683 mV full range)
Trigger detection is disabled.
Stimulation blankout is disabled.
Stimulated channels: none
Running...
```

Notice that the time displayed at the far right of the second row of controls in the scope runs.

- Click the 'Center' button of the scope to remove DC offsets from the traces. Now may be the time to explore the other controls in the top row of the scope. Help balloons pop up when you hover the mouse cursor over any item.
- To stop data acquisition, bring input focus to **rawsrv**, and press **Ctrl-C**.

```
(interrupt)
rawsrv>
```

- The scope may be terminated using the 'Close' button provided by your window manager. The **rawsrv** should be terminated by pressing **Ctrl-D**, or by typing 'quit':

```
rawsrv> quit
$
```

3.3 Online spike detection

The next example will extend the previous one, by adding a spike detector, and a recorder to save the data to hard disk.

- Open four terminal windows, and start **scope** and **rawsrv** as before. In the following, I will let you figure out in which window to type from the prompts shown. For example, I'll write

```
rawsrv> run
```

and leave it implicit that you need to bring keyboard focus to the terminal window in which you started **rawsrv**, and then type 'run'.

- Start the spike detector, and configure it:

```
$ spikedet
spikedet> source raw
Source is raw
spikedet> type 3
Type is BandFlt-25
spikedet> thresh 5
Threshold is 5
spikedet>
```

The threshold is specified in terms of an estimate of RMS noise. The spike detector can only base this estimate on observing the noisy signal and guessing which part is noise. This is not an exact science, and **spikedet** implements more than one algorithm. See section 6.2 for details. If you can think of a better scheme, please contribute it!

- Set the **rawsrv** running:

  rawsrv> `run`
  Gain setting is 2 (+/- 0.683 mV full range)
  Trigger detection is disabled.
  Stimulation blankout is disabled.
  Stimulated channels: none
  Running...

  (Is the scope aware of the existence of **rawsrv** and **spikedet**? If not, use the 'Raw source' and 'Spike source' controls to remedy the situation.)

- The spike detector needs to be 'trained' on the amount of noise in the source:

  spikedet> `train`
  Type is BandFlt-25
  Threshold is 5
  Training...

- A few seconds will pass, and then the spike detector will return:

  Training complete
  spikedet>

- Interrupt the **rawsrv** as before.

- Start and configure the recorder:

  $ `record`
  record> `cd`
  Working directory: /home/wagenaar/tmp
  record> `source raw spike`
  Sources are:
  raw [raw]
  spike [spike]

  (The recorder tells you that it is about to record from **rawsrv**, which provides a stream called *raw* of type RAW, and from **spikesrv**, which provides a stream called *spike* of type SPIKE.)

- Prepare the recording:

  record> `record firstdata`
  Sources are:
  raw [raw]
  spike [spike]

```
Waiting for START from raw
Waiting for START from spike
```

The recorder will output files called 'firstdata.raw', and 'firstdata.spike', but for now it's waiting for the data to arrive.

- Start the spike detector:

```
spikedet> run
Source is raw
Type is BandFlt-25
Threshold is 5
Quick recording disabled
Excluded electrode channels are: none
Disabled analog channels are: None
Waiting for START from raw...
```

As you can see, the spike detector is quite chatty, and tells you lots of things you already know. It helps to prevent surprises later...

- Let's record precisely 100 seconds:

```
rawsrv> run 100
Gain setting is 2 (+/- 0.683 mV full range)
Trigger detection is disabled.
Stimulation blankout is disabled.
Stimulated channels: none
Running...
```

- The spike detector will confirm that the run started:

```
Running...
```

As will the recorder:

```
Recording from raw into firstdata.raw without triggering
Recording from spike into firstdata.spike without triggering
```

It is important to set the different programs in motion in the right order: downstream first. The programs will automatically wait for a START signal from their source, and they will wait forever if the source is already running when they open the communication channel.

- After 100 seconds, **rawsrv** will stop, and the spike detector will issue an end-of-run report:

```
STOP received - 1511 spikes detected
Buffer use percentages: 1
spikedet>  (Client 'record' lost)
spikedet>
```

- The recorder will also report some statistics:

```
Recording from raw ended
Buffer usage: 9
Recording from spike ended
Buffer usage: 1 4
record>
```

Those 'buffer usage' numbers are shown for each of the communication channels between pairs of MEABench programs. If any number is above 80 (percent), the recorder will warn of the risk of buffer overruns.

- Take a look at the 'description' files that **record** generated. They are called 'firstdata.raw.desc' and 'firstdata.spike.desc'. They contain more statistics about the run, which may be helpful for your or your computer to interpret the data later on.

3.4 Triggered recording

This final fully worked example explains how to perform event-triggered recording. Let's assume that we have a stimulator connected to the MEA, which delivers current pulses on one of the channels once every few seconds. We want to record the responses.

- The first step is to connect a TTL trigger from the stimulator to channel A1 on the data acquisition card.
- Then, set up the **rawsrv** for trigger detection:

```
$ rawsrv
rawsrv> trigchannel 1
Trigger detection is disabled
rawsrv> trigthresh 3000
Trigger detection is disabled
rawsrv> usetrig 1
Trigger detection is enabled on channel A1 - threshold is 3000
```

If you are using MCS hardware, zero volts is represented as digital value 2048, and a TTL trigger will max out the amplifier at digital value 4095, so a threshold halfway is appropriate. If you are using different hardware, use the following trick: set the **rawsrv** running, and open a **scope** on it. Control-double-click on the panel displaying A1. In the controlling terminal, you will see a lot of numbers scrolling past. Those are the digital values read from channel A1. 50 ms of data is shown, one millisecond per line (if the terminal window is wide enough). If you catch a trigger pulse using the 'Freeze' button, and display it in a 50 ms window, you can figure out both the baseline value and the peak value, and set a threshold based on those. Here's a part of the output I obtained doing this experiment:

```
36.746 2383 2384 2383 2383 2383 2383 3534 4095 4095 4095 4095 4095 4095 4095 4095 4095 4095 4095
36.747 4095 4095 4095 4095 4095 4095 4095 4095 4095 4095 4095 4095 4095 4095 2793 2446 2392
36.748 2378 2371 2370 2373 2376 2379 2380 2381 2381 2381 2382 2382 2382 2382 2382 2381 2383 2382
```

The trigger pulse is clearly visible as a sequence of maxed-out values (4095).

- Set up the recorder for triggered recording:

```
$ record
record> source raw
Sources are:
  raw [raw]
record> trecord secondfile 50 450
Sources are:
  raw [raw]
Waiting for START from raw
```

- Start the data acquisition:

```
rawsrv> run
Running...
```

- Now set the stimulator going, and watch the progress using **scope**.
- After the stimulation program has ended, stop the recording by pressing **Ctrl-C** in **rawsrv**'s terminal window. **Do NOT terminate the recorder, instead, terminate the source.** That way you are guaranteed that the final trigger window is saved to disk properly.

3.5 Using Commander

Here is an example script for commander that reads a RAW file from 'noisy.raw', filters 60 Hz noise out of it using a reference signal on channel A2, and detects spikes using BandFlt at five times RMS noise. The results are saved as 'denoised.spike'.

```
# Start programs
new replay so:noisy.raw
new filter60hz/60hz so:reraw lock:a2 nper:100
new spikedet so:60hz ty:3
new record so:spike

# Check whether they came up OK
expect replay 1 replay>
expect 60hz 1 60hz>
expect spikedet 1 spikedet>
expect replay 1 replay>

# Let's train the spike detector
tell filter60hz cont
tell replay run
sleep 5
tell spikedet train
flush spikedet
expect spikedet 20 spikedet>
dieif spikedet STOP received before training complete
flush replay
intr replay
expect replay 5 replay>
```

```
# Good, let's go.
tell spikedet run
tell record record denoised
expect record 1 Waiting
expect spikedet 1 Waiting
tell replay run
flush record
expect record 1000 record>

# All done.
quit
```

Chapter 4

Basics of the toolset

MEABench consists of a number of independent linux command line programs. Some of these programs are *servers*, that make the result of their computations available to others. Others are simply *clients*, that read data from one (or several) of the servers, but do not make their results available. Many programs are both client and server. Such programs can be thought of as generic *filters*.

This section describes some of the internals of MEABench, essential for potential developers, and hopefully helpful for users who want to understand how things work. When first reading of this document, you may wish to skip ahead to the next chapter, the list of components.

The core of MEABench is a library of C++ classes that can be used to easily construct new components. This library is stored in the meabench/base subdirectory. Presently, the only library documentation is contained in the source (header) files.

Components of MEABench communicate with each other in a standardized way. Servers publish a *shared memory stream*, from which clients can read asynchronously. Currently, two data types are supported: RAW, which contains raw digital data as read from the driver, and SPIKE, which contains information about spikes. Internally, RAW data is represented by C++ datatype *Sample*, while SPIKE data is represented as *Spikeinfo*. These datatypes are defined formally in meabench/common/Types.H.

Associated with each stream is some header information, from which clients can find out how much data is ready to be read from the stream.

Servers never check whether clients are keeping up — it is up to the individual clients to detect overruns. This philosophy was adopted because some clients may not care too much about overruns (e.g. display programs) whereas others do (filters and recorders). All current core clients detect buffer overruns and report buffer usage at the end of a data taking run.

In addition to shared memory streams, servers publish a *wakeup socket*, from which clients can receive wakeup calls whenever a given amount of data is available in the stream. The wakeup socket also notifies clients when a run starts or ends, and when triggers are detected.

Chapter 5

List of components

These are the programs that currently make up MEABench. Following sections detail each program.

- **Rawsrv** — The grandmother server. It reads data from the hardware using Tom DeMarse's driver and makes it available as a RAW stream.
- **Spikedet** — Basic spike detection. It reads from a RAW stream, and publishes a SPIKE stream.
- **60hz** — Template filter to reduce 60 Hz pickup.
- **Salpa** — Stimulus artifact filter.
- **Record** — Records RAW or SPIKE data to disk.
- **Replay** — Replays files created by **Record**.
- **Scope** — GUI program to display RAW and SPIKE data online.
- **Spikesound** — GUI program for online sonification of SPIKE data.
- **Flexraster** — GUI program to display raster plots of SPIKE data online.
- **Monitor** — A debugging aid, it shows the status of all servers.
- **Neurosock** and **Nssrv** — An alternative to **Rawsrv** that allows one to dedicate one computer to data acquisition, and another for online analysis.

The following is a set of utility programs that can be used with MEABench or on their own right.

- **Runmeab** — Opens a set of xterms from which MEABench programs can be launched.
- **Spikedump** — Converts a SPIKE file into human readable form.
- **Doublectxt** — Takes a SPIKE stream and a RAW stream, and tags additional context on to the SPIKE information from the RAW channel at which the spike occurs.
- **Noisehisto** — Takes a RAW stream and outputs a histogram of voltages observed in each channel.

- **Noiseshape** — Takes a RAW stream and outputs the first few central moments of the voltage distribution for each channel.
- **Uniquespike** — Output spikes found in one but not in another file.
- **Trigvar** — Computes the variance in a triggered RAW stream as a function of time post stimulus.
- A growing set of additional utilities needs to be documented.

Note that command names are usually capitalized in this manual, but must always be spelled all lowercase on the command line.

Chapter 6

Details of each component

This section explains the core MEABench components in more detail. Most of these components have a command line interface. Thus, the following entries focus mostly on the available commands. In addition to the commands listed in the individual descriptions below, the following commands are common across components:

- **?** — Provide help in the form of a list of commands with brief descriptions.
- **quit** — Terminates the program gracefully. The same can be effected by pressing **Ctrl-D** at the prompt.

Servers that are capable of loading and saving data to disk support:

- **cd** [*directory*] — Change or report current working directory.
- **ls** [*arguments*] — Directory listing as per /bin/ls.
- **mkdir** [*directory*] — Create a new directory.
- **!** *command* [*args*] — Execute an arbitrary shell command.

Client programs support:

- **source** [*stream-name*] — Specify from which other MEABench program the data is to be taken by specifying its stream-name.

6.1 Rawsrv

Rawsrv reads RAW data from the hardware and publishes it as a shared memory stream called *raw*. It is an extremely straightforward piece of software. Other than providing a nice large buffer to prevent overruns, it is able to detect trigger signals on any of the analog channels (A1, A2 or A3) and to blank out the electrode channels for some time during and after a trigger.

These commands are supported:

- **run** [*time-in-s*] — Starts a run. Optional argument limits the duration of a run to the given time. Otherwise, press **Ctrl-C** to stop a run.

- **usetrig** [0/1] — Enables (1) or disables (0) reporting trigger events on the wakeup socket and in the auxilliary data of the RAW stream.
- **trigchannel** [1/2/3] — Selects which of the three analog channels to monitor for triggers.
- **trigthreshold** [*digivalue*] — Specifies the (digital) value of the threshold above which a trigger is detected.
- **autothresh** [*multiplier*] — Sets the threshold for trigger detection at *multiplier* standard deviations above the channel mean value.
- **gain** [*gain-step*] — Sets the gain of the MCS amplifier. Type **gain ?** to list possible values. For our Multi Channel Systems card, the values are as follows:

value	full range (uV)	step (μV)
0	3.410	1.665
1	1.205	0.588
2	0.683	0.333
3	0.341	0.167

- **blankout** [*period-in-ms* | 0] — Enables or disables blanking of electrode channels during a trigger. Blanking is performed by replacing the data by the average of four samples obtained just prior to the stimulus. Blanking only works if trigger detection is enabled. Signal blanking is largely outdated by **Salpa**.

6.2 Spikedet

Spikedet detects spikes on a RAW data stream and publishes them as a SPIKE stream called *spike*. Several different types of spike detection may be supported by **Spikedet**. Currently, these are the fully supported spikedetectors:

1. BandFlt — Straightforward threshold detector;
2. AdaFlt — Threshold detector with adaptive threshold;
3. LimAda — Better threshold detector with adaptive threshold;
4. SNEO — Detector based on instantaneous energy in signal.

BandFlt is thoroughly tested and quite stable. AdaFlt (7/12/01) seems to work well except that it's threshold varies during bursts; LimAda solves this problem. SNEO has never worked as well as I would hope. See below for details on each detector.

These commands are supported:

- **source** — See general description.
- **type** [*detector-name*] — Changes or reports the detector being used. Use **type ?** to query supported detectors.

- **run** — Starts a single run. If the detector hasn't been trained yet, the first few seconds of the input data are used for training.
- **cont** — Same as **run**, but restarts automatically after receiving a STOP command from the server.
- **train** — Trains the detector on the current source. Unusually, this command does not wait for a START message from the source to begin operation. It is recommended to let the source run for a few seconds before commencing training, to ensure that any transients have died out.
- **info** — Reports the result of training.
- **threshold** [*value*] — Sets or reports the threshold for spike detection, in units particular to each detector. Thresholds are automatically scaled for each channel.
- **disableanalog** [*channel* ... | –] — Disables spike detection on the given set of analog channels. Typical use is to exclude the 60 Hz reference signal from being recorded in spike files.
- **excludechannels** [*RC* ... | –] — Marks a set of electrode channels as 'dead'. Useful to prevent spurious spike detection on clamped down or flaky channels.
- **outputfilt** [0/1] — Enables or disables the creation of a shared memory stream called *spraw*, on which the filtered raw data is reported.
- **savenoise** *filenamebase* — Saves the current training results as '*filenamebase*.noise'. This data consists of estimated RMS noise values, and can be used by **Salpa** as well.
- **loadnoise** *filenamebase* — Loads a previous set of training results from '*filenamebase*.noise'. **Salpa** generated noise files may be loaded as well.
- **alias** — A fake command. Reminds the user of the existence of the '–alias' flag (see below).

In some circumstances it may be useful to run more than one spike detector simultaneously. To make that work, **Spikedet** can be made to publish its results on a differently named stream, by starting it as 'spikedet -alias *streamname* [*cmds*]'.

6.2.1 BandFlt

BandFlt passes the RAW data through a band pass filter. Currently a first order filter with cutoffs at 150 Hz and 2.5 kHz are used, but this may be changed in 'spikedet/Filters.H'. It detects spikes if the filtered stream exceeds a given multiple of the estimated RMS noise in each individual channel. Useful threshold values are 4 to 6. It should be noted that noise in RAW data is far from Gaussian, so future versions may be changed to employ more relevant noise measures. In the current implementation, the noise is estimated by a slightly unusual method, which is intended to minimize the effect of spikes and stimulus artifacts on the estimate. Three hundred 10 ms windows of electrode data is read. For each of these windows the RMS value is calculated. The results are sorted, and the final estimate of RMS noise is taken to be the 25th percentile of the measurements. While I recognize that this

method finds an underestimate of the RMS noise, this algorithm is much more useful than straightforward RMS measurement, for the stability reasons mentioned above.

6.2.2 AdaFlt

AdaFlt uses the same initial band pass filter and also collects 128 windows of length 10 ms from the beginning of the recording. From then on, it proceeds differently: it measures the minimum and maximum values in each of those windows, and finds the 40th percentile of both collections of extrema. The initial thresholds for upward and downward spikes are based on the result. While running, it keeps collecting minima and maxima in 10 ms windows, although it uses only one in ten windows[1]. Whenever 128 windows have been collected, the thresholds for that channel are updated.

For every detected spike, the ruling threshold at the time of detection is written into the SPIKE stream.

Useful threshold values are 1.3 to 2 (multiples of the extrema).

6.2.3 LimAda

After band pass filtering as for BandFlt, LimAda splits the data stream into 10 ms windows, and determines the 2nd and 30th percentiles of the distribution of voltages found in each such window. Call these voltages $V_{.02}$ and $V_{.30}$. (Note that both are usually negative because of the filtering, which sets $V_{.50} \sim \langle V \rangle \sim 0$.) It then performs two tests:

- Is the ratio of $V_{.02}$ over $V_{.30}$ less than 5?
- Is the absolute value of $V_{.30}$ (significantly) non-zero?

The first test makes sure that there was no actual spike in the window; the second test makes sure that the data in the window was not blanked out (e.g. by **Rawsrv** or **Salpa**). If both tests are passed, the windows is considered 'clean', and $V_{.02}$ is used to update the current noise threshold estimate. Spikes are detected whenever the absolute value of the voltage exceeds the current threshold, which is the output of passing the absolute values of $V_{.02}$ from all 'clean' windows through a low-pass filter with a time constant of 100 windows (1 second if all are clean). This algorithm adapts rapidly to changing noise situations, while not desensitizing during bursts.

The threshold settings are normalized to estimated RMS noise, so values of 4 to 6 are reasonable. As of August 26, 2004, this is my favorite spike detector.

6.2.4 SNEO

SNEO also passes the RAW data through a band pass filter, but then computes the instantaneous energy in each electrode stream:

$$E_c(t) = V_c'(t)^2 - V_c(t)V_c''(t).$$

This energy is smoothed over 5 samples and spikes are detected if it exceeds a given multiple of the RMS value of the energy. Although Kim and Kim (IEEE Biomed eng 47 (2000) 1406) report that SNEO works very well at S/N as low as unity, I am less convinced.

[1] In fact, for every window it collects extrema for only six out of 60 electrode channels, to spread the computational load.

Useful threshold values seem to be 5 to 20.

6.3 60hz

60hz provides a template filter to reduce 60 Hz line pickup in raw data and publishes the results as *60hz*. It works best if an external lock in signal is provided on one of the analog lines. If such a signal is not available, fast adaptation should be chosen to reduce the effects of gradual desynchronization. Templates are collected for each electrode channel independently.

These commands are supported:

- **source** — See general description.
- **run** — Starts a single run. At the start of the run, a small amount of data is used to train the filter. During this period no output is generated.
- **cont** — Same as **run**, but restarts automatically after receiving a STOP command from the server.
- **nperiods** [*periods*] — Sets the adaptation time of the filter. Old contents are decayed by a factor $1/e$ after the given number of periods. (A period is 16.67 ms[1].)
- **templsize** [*size-of-template*] — Number of data points to use for each template. This value is rounded internally to a power of two. Low values reduce the efficacy of the filter, but high values require longer training times and may make the system less stable. The default value, 128, should normally be adequate.
- **wait** [0/1] — Enables (1) or disables (0) waiting for a START command from the server. Operation with waiting disabled is poorly tested and may result in desynchronized and useless recordings. Not recommended for normal use.
- **lockin** [– | A*n*] — Enables or disables the use of an external synchronization pulse on a given analog channel. Rising edge on the specified channel will be used to synchronize the filters to the physical 60 Hz signal.
- **limit** [*adaptation-period-in-seconds* or 0 for unlimited] — It may be desirable to stop adaptation altogether after a certain amount of time. For example, if very strong signals are expected occasionally on the electrode channels. Such signals might be picked up by the template and cause echos at 16.67 ms intervals. In practice, **blockonmark** provides a better solution for most cases.
- **blockonmark** [– | An [*block_ms* [*thresh_digi* [*lookahead_ms*]]]] — If your data contains (stimulation) artifacts, the adaptive filter tends to create echoes of those artifacts. This command can be used to temporarily suspend the adaptation (but not the filtering) during artifacts. When enabled, **filter60hz** will detect upward threshold crossings on the specified analog channel (*An*), and disable adaptation on all electrode channels for the given period (*block_ms*). The threshold is specified in digital units (*thresh_digi*).

[1]Note for non-US users or programmers: the period is specified in units of the sample period by the variable REALPERIOD in "60hz/DefsḢ". When changing this variable, please be aware that MCCard's sampling frequency, although very constant, is not exactly 25.000 kHz.

The final argument (*lookahead_ms*) can be used if the marker may occur (a fraction of) a millisecond after the start of an artifact[1].

The current version of **60hz** does not support **qrec**.

For off-line usage, the command **Posthoc60hz** will read from a file and output to a unix pipe. It has the additional benefit of skipping artifacts when training. (This requires some command line switches — try 'posthoc60hz –help'.)

6.4 Salpa

Salpa is the artifact suppression algorithm described in Wagenaar and Potter, Real-time multi-channel stimulus artifact suppresion by local curve fitting, *J. Neurosci. Meth.* **120:2** (2002) pp 113–120. Please refer to that paper for functional details. You may pick up a preprint from `http://www.biology.ucsd.edu/~dwagenaa/pubs.html`.

Salpa supports the following commands:

- **run** — Starts a single run. At the start of the run, a small amount of data is used to train the filter. During this period no output is generated.
- **cont** — Same as **run**, but restarts automatically after receiving a STOP command from the server.
- **digithresh** [*digital-threshold*] — Sets the threshold for acceptable asymmetry in digital units, or reports the current value. In the paper, this 'asymmetry' is referred to as the *deviation* $\mathcal{D}$.
- **noisethresh** [*threshold-in-units-of-RMS-noise*] — Sets the threshold for acceptable asymmetry in units of the estimated RMS noise, or reports the current value.
- **halfwidth** [*halfwidth-in-ms*] — Sets the SALPA filter halfwidth. Except for a factor τ_{sample}, this is the number N in the paper.
- **asymduration** [*asymmetry-window-width-in-ms*] — The window over which the asymmetry is measured. Except for a factor τ_{sample}, this is the number δ in the paper.
- **blankduration** [*blanking-duration-in-ms*] — Determines how much signal is blanked even after the asymmetry (deviation) test has been passed successfully. This is a bit of a hack, which was not used in the paper.
- **lookaheadwindow** [*look-ahead-window-in-ms*] — The last few samples before the signal pegs are probably not entirely artifact-free. This command allows you to blank a little bit of data just before the stimulus artifact onset.
- **digirails** [*digi-rail1* [*digi-rail2*]] — Specifies which digital values constitute the rails of the ADC. Defaults 0 and 4095 are for MultiChannel Systems hardware with **Rawsrv**.

[1] Even if the marker is timed to exactly coincide with the start of the artifact, *lookahead_ms* can be used to provide a safety margin of a few samples.

- **fixedperiod** [*period-ms delay-ms* [*blank-ms*]] — In some cases artifacts may occur that do not quite peg the channel. If this happens in a triggered recording, **Salpa** can still work: just specify the length of the trigger window (*period-ms*), the amount of time before the onset of the artifact in each window (*delay-ms*), and the duration of the fast part of the artifact (*blank-ms*). Typical values for *blank-ms* would be one or two milliseconds. **Salpa** will kick in after that and determine the end of the irrepairable part of the artifact using the asymmetry (deviation) test as usual.

- **pegontrigger** [– | A*n* [*blank-ms* [*tresh-digi*]]] — A more flexible solution to the problem explained above. **Salpa** can treat a positive threshold crossing on one given channel (usually A1) as a signal to consider all channels pegged. *blank-ms* has the same meaning as for **fixedperiod**, and *thresh-digi* specifies the threshold (in digital units) for the detection of stimulation markers.

- **channels** [– | + | *CR* . . .] — Normally, **salpa** operates on all electrode channels. Using this command you can restrict operation to any subset. This is useful when the artifacts on most channels reliably last less than 1 or 2 ms, so the **salpa** algorithm doesn't improve things. Say 'channels CR1 CR2 . . . ' to limit operation to channels CR1, CR2, . . . ; or 'channels –' to operate on all channels. Alternatively, say 'channels + CR . . . ' to add channels to an existing list, or 'channels – CR . . . ' to remove channels from an existing list. 'channels +' restores a list previously removed by 'channels – '. Channels excluded from operation are still subject to blanking by **fixedperiod** or **pegontrigger**. This is usually desirable. If not, you can set the blanking period to zero, e.g. by 'pegontrigger A1 0'.

- **source** — See general description.

- **train** — Estimate the RMS noise level of the input. This takes a few seconds. **train** must be executed while the source is already running, unlike **run** and **cont**, which wait for the source to start.

- **savenoise** — See **Spikedet**.

- **loadnoise** — See **Spikedet**.

- **info** — See **Spikedet**.

An off-line version of **salpa** is available as well; it's called **posthocartifilt**, and supports most options of the MEABench component through command line switches. The command is self-documenting: type `posthocartifilt --help` for details.

6.5 Record

Record is used to record RAW or SPIKE streams to disk. The program can record several streams in parallel. **Record** can optionally respect the triggers on the associated wakeup socket. In this case, only the parts of the stream immediately surrounding the triggers are saved to disk, and an auxiliary file with trigger times is constructed.

These commands are supported:

- **record** *filename* [*comments*] — Starts recording to the specified file. The filename is augmented by the type of the data. Optional comments are saved to a description file, if enabled.

- **multirecord** *base-filename* [*comments*] — Starts recording to many files one after the other. The filename is augmented by the start time of each recording, and by the type of the data.

- **trecord** *filename pretrig-ms posttrig-ms* [*comments*] — As **record**, but respects trigger information. The window of recording is specified as time before the trigger and time after the trigger, both in ms.

- **multitrecord** *base-filename pretrig-ms posttrig-ms* [*comments*] — **multitrecord** is to **trecord** as **multirecord** is to **record**.

- **source** [*name*[/*type*] ...] — Specifies the sources for recording. **Record** knows the type of most core MEABench streams. If it doesn't know for the stream you name, the type can be specified by appending it to the stream name after a slash. (For example, if *coolsort* is a new stream of type SPIKE, you would say 'source coolsort/spike'.)

- **describe** [0/1] — Enables (1) or disables (0) the generation of a description file (filename constructed by augmenting the data filename by '.desc'). Description files contain lots of useful information pertaining to a run and are in human readable form.

Record can record from several sources simultaneously: just specify them all together as arguments to **source**. For example, 'source raw 60hz spike' would prepare a recording from three sources. If more than one stream of the same type is recorded, the streamnames are incorporated in the filenames. Recording from several sources does have a few limitations:

- Recording ends immediately when the first stream terminates. A small amount of data from the end of the other streams may be lost.

- For triggered recording, only the first stream will yield a '.trig' file.

To avoid these limitations, it is possible to run several instances of **Record**, and have each of them record from a single stream.

It is possible to terminate a recording session before the source ends by pressing **Ctrl-C** in the **Record** terminal window. This is useful for recording a short segment of RAW data parallel to the beginning of a longer SPIKE data recording.

6.6 Replay

Replay replays files recorded by **Record**. Currently, RAW and SPIKE data are supported, and are published as *reraw* and *respike* respectively. Replaying a SPIKE file results in both a RAW and a SPIKE stream, the RAW stream reporting the contexts stored in the SPIKE stream.

These commands are supported:

- **play** [*filename* [*type*]] — Plays the given file. Normally, **Replay** automatically detects the type of the data. If it doesn't, help it by specifying it explicitly. Without a filename, plays the last played file again.

- **slow** [*slowdown-factor*] — Slows down playback by a given (real valued) factor. Useful for output to **Scope**. Arguments smaller than one cause speed up.
- **run** — Alias for **play** without arguments.
- **source** [*filename* [*type*]] — Specifies a filename (and optional type) for later playback.
- **blankout** [*period-in-ms* or 0] — Enables or disables blanking of electrode channels during a trigger. Blanking is performed by replacing the data by the average of four samples obtained just prior to the stimulus. Blankout works only for triggered files. **Replay** does not detect stimuli itself.
- **cd** [*directory name*] — Changes the current working directory.
- **!** — Shell escape. Useful, e.g. for 'ls'.
- **selftrig** [0/1] — Enables or disables spike detection from an analog channel in the stream. When disabled, triggers stored in the .desc file of a file produced by **Record**'s **trecord** command are still reported.
- **trigchannel** [1/2/3] — Sets the analog channel on which triggers are detected (if **selftrig** is enabled).
- **trigthreshold** [*digivalue*] — Sets the threshold for such trigger detection.

6.7 Scope

Scope shows raw data and spikes in a similar format as the graphical parts of my older **Qmeagraph** program, or Multi Channel Systems' **MCRack**. It mostly explains itself. Here are some useful hints:

- Double clicking on any of the small electrode traces opens a separate window showing that channel.
- Selecting window width, pre-trigger length, or voltage ranges in **Scope** does not affect recording in any way.
- The scrollback buffer (enabled by clicking the 'Freeze' button) is very useful to home in on some interesting event. Raw data in the scrollback buffer can be saved to disk using the 'Save' button which appears whenever 'Freeze' is enabled. Currently, the scrollback buffer is about 5 seconds long. This is a direct function of the length of the shared memory segments used to pass RAW data around.
- These known bugs exist in scope:
  - Spike circles at the edges of the electrode traces leave semi-permanent smudges.
  - When scrolling back, older spikes may lose their red marks if there are many detected spikes. This is the result of the SPIKE data shared memory segments being too short.
  - When replaying a spike stream, red spike marks sometimes appear out of nowhere. These ghosts are easily recognized, because no context data is plotted around them. This bug has been a mystery so far.

6.8 Spikesound

Spikesound makes spikes audible through a PC sound card. It can read of any SPIKE stream (e.g. straight from the spike detector, or from **replay**). The GUI controls are minimal:

- Source — Selects the MEABench stream to get spikes from. That usually is *spike* to read from **spikedet**, or *respike* to read from **replay**.
- Play — Switch sound on or off
- Volume — Master volume control.
- –ve only — Limit sonification to downward spikes. The obvious counterpart can be implemented on request.
- Rethreshold — It is often useful to set the spike detection threshold fairly low in **spikedet** and use off-line spike sorting to clean up the data. However, hearing all the near-noise-level spikes is not very appealing. This control allows you to hear only strong spikes. For example, I like to set the **spikedet** threshold to 4.5σ, and set the **spikesound** rethreshold factor to 120%.
- A1, A2, A3 — Enable or disable beeps when a spike is detected on one of the auxillary channels. For example, I like to use A1 and A3 for triggers, and A2 as a 60 Hz lock-in signal. So I may want to hear a sound when a trigger happens, but I don't want to hear the 60 Hz signal.

Customization note: The current version uses output buffers of about 25 ms to improve timing accuracy and reduce lag. If you prefer hearing longer noises, or your sound card / CPU do not allow you to use such short buffers, you may change the value of `AUDIO_LOG_FRAG` in `spikesound/Audio.H`. The length of the buffer is:

$$\tau = \frac{2^{\texttt{AUDIO\_LOG\_FRAG}}}{176.4} \text{ ms.}$$

6.9 Flexraster

Flexraster displays raster plots of spike activity in triggered recordings. It currently relies on 'trigger spikes' on channel A1. There are six different ways to create rasters:

- Spont — Spikes from all channels are combined as blue dots. Each line of the raster is *Pre* plus *Post* ms wide. The raster plot scrolls vertically, showing the most recent interval on top. A new raster line is generated whenever there is a trigger on A1, or every second if there has not been any trigger.
- 8x8Rec — Spikes from each channel are displayed in separate panels. Within each panel, the display is as for Spont.
- 8x8Stim — If the shape of the trigger pulse is used to encode a CR-position, each panel displays all spikes that occurred in response to stimulation on a certain channel. See figure for details.

- V Stim and H Stim — Similar to 8x8Stim, but panels are generated only for those CRs that actually receive stimuli.
- Cont — Arguably the most useful function, creates a scrollable and scalable raster plot with 60 electrodes stacked vertically and time running continuously left to right. Trigger pulses are indicated by red marks.

Flexraster is under development. Suggestions for improvement are especially welcome.

6.10 Commander

Commander allows you to start and control MEABench programs from within a central shell-like language. This is mainly useful for off-line analysis. You may find an example of its use in chapter 3. Commander logs all interaction with the programs it controls to the screen, as well as to an optional logfile. Creating logfiles is highly recommended, because it allows you to keep track of how exactly you processed the data, and to check whether MEABench behaved as you expected it to behave.

Commander supports the following commands:

- **new** *program*[/*id*] [*args*] — Start a new program. An optional *id* may be assigned to the program to disambiguate references to two instances of the same program, or for ease of reference. *Args* are passed to the program unchanged.
- **tell** *program—id command* [*args*] — Send a command to the named program. *Args* are passed unchanged.
- **expect** *program|id timeout regexp* — Wait until the named program produces output that matches *regexp*. If this does not happen within *timeout* seconds, report an error.
- **dieif** *program|id regexp* — Look back at the output captured by the last expect, and abort if any line matches *regexp*. (This may be a line that was output long before the line that made **expect** happy.)
- **dieunless** *program|id regexp* — Look back at the output captured by the last expect, and abort unless some line matches *regexp*. (This may be a line that was output long before the line that made **expect** happy.)
- **flush** *program|id* — Flush all output from the named program, so that future **expect** commands will not match any output prior to the **flush**.
- **intr** *program|id* — Send an Interrupt signal to the named program, i.e. simulate pressing **Ctrl-C**.
- **kill** *program|id* — Send a Term signal to the named program, terminating it. Normally, you'd use **close** to achieve the same result more gracefully.
- **close** *program|id* — Close a running program normally, as if **Ctrl-D** was pressed. If closing normally fails, the program is terminated as per the **kill** command.
- **wait** *program|id* — Wait for the named program to terminate. Use this if you just sent it a **quit** command. Normally, **close** is an easier way to terminate subprocesses.

- **log** [*logfile*] — Write all future output to the named file, or stop logging if no argument is given.
- **comment** *comments* — Write the specified comments to the log file.
- **sleep** *time_s* — Sleep for the given number of seconds. The subprocesses are not affected.
- **quit** — Exit **Commander** after closing all subprocesses.

Unlike other MEABench programs, commander does not present the user with a prompt. You are not really expected to type away at commander from a terminal (although you can). Instead, you'd normally prepare a script, and then run commander on it:

```
$ mea commander < myscript.cmdr
```

Any error that happens during script execution (e.g. failure to start a subprocess or failure to read an **expect**-ed string) causes **Commander** to terminate immediately with an error report written to the screen and the log file.

An auxillary program, **Cmdlog2html** exists to convert log files produced by **Commander** to html format. Log files are human readable, but the html format looks nicer.

6.11 Monitor

Monitor is mostly a debugging tool. It displays the status of each of the MEABench servers.

6.12 Neurosock and NSsrv

Run **Neurosock** on the machine that contains the physical hardware, and MEABench on any other machine that can connect to the first machine through the internet. It can be run without any arguments for 64-channel MCCards, or as `neurosock -set MC128` for 128-channel MCCards.

NSsrv is exactly like **Rawsrv**, except that it doesn't record directly from the MEA hardware. Instead, it connects to **Neurosock** on the same or another computer. For 64-channel MCCards, run **NSsrv** without arguments; to use the 2nd half of 128-channel cards, run it like `nssrv -s raw2`.

NSsrv has the same commands as **Rawsrv**, with one addition:

- **ip** aa.bb.cc.dd — Specify the IP address of the computer running **Neurosock**.

Note: Only one instance of **NSsrv** can read from a 64-channel **Neurosock** or half a 128-channel **Neurosock** at a time. It is not possible to specify gain separately for the two halves, and the **gain** command will fail if you try to set gain on one half while the other half is running.

6.13 Spikedump

Spikedump converts SPIKE files to human readable form, dropping the context data. It can read from a specified file, or from stdin. It does not presently run of MEABench streams. Output are time (in seconds), channel (hardware order, counting from zero), height (digital value) and width (in samples).

6.14 Doublectxt

Doublectxt combines a RAW stream with a SPIKE stream to construct a file with two context fields per spike. The resulting file can be read with the *loaddblctxt.m* matlab function.

6.15 Other utilities

This documentation is presently incomplete regarding the utilities in the meabench/utils and meabench/perl subdirectories. These utilities will provide a brief Usage message if invoked with a '−−usage' argument.

6.16 Matlab functions

A number of matlab functions included with MEABench can be used to load MEABench data files into matlab, to perform channel numbering convention conversions and to perform some common visualization tasks. These functions are installed in /opt/meabench/matlab (if you follow the suggested installation procedure in chapter 2). In order to use them, you need to tell matlab about that directory:

```
>> addpath('/opt/meabench/matlab');
```

The functions are the following:

- **burstdet_timeclust**
  [tt,chcnt,dtt] = BURSTDET_TIMECLUST(spks, bin_s, thr_mean)
  Given a set of spikes SPKS loaded by LOADSPIKE or LOADSPIKE_NOC, finds synchronized bursts (on 5 or more channels).
  This works by first finding bursts on individual channels using TIMECLUST, and then finding synchronized bursts by clustering the single-channel bursts, again using TIMECLUST.
  Both BIN_S and THR_MEAN are optional; default values are 0.1 s and 5x resp.

- **cleanctxt**
  [ctxts, idx] = CLEANCTXT(contexts) returns cleaned up contexts:
  - The first and last 15 values are averaged and used to compute DC offset.
  - These two estimates are weighted according to their inverse variance.
  - The DC offset is subtracted.
  - If any sample in -1:-0.5 or 0.5:1.5 ms is more than half the peak at 0 ms, the spike is rejected.
  - Use CLEANCTXT(contexts, testidx, relthresh) to modify this test:

TESTIDX are indices (1:74) of samples to test,
RELTHRESH is a number between 0 and 1.
- Additionally, the area immediately surrounding the peak is tested at the 0.9 level: The spike is also rejected if any sample in -1:-0.16 or 0.16:1.5 ms has an absolute value more than 0.9 x the absolute peak value. This test is modified on its outer edges by the edges of testidx, but cannot be modified independently.
Returns: ctxts: the accepted contexts, with DC subtracted
idx: the index of accepted spikes.
Requirements: contexts must be as read from loadspike, i.e. 74xN (or 75xN).
Acknowledgment: The algorithm implemented by this function is due to Partha P Mitra.

- **cr2hw**
hw = CR2HW(c,r) converts row and column to hardware channel number. c, r count from 1 to 8; hw counts from 0.
hw = CR2HW(cr) converts combined row and column to hardware channel number. cr can be in the range 11..88.
Illegal c, r values return -1. c, r or cr may be matrices, in which case the output is also a matrix.
The dimensions of c, r must agree.

- **heightscat88**
HEIGHTSCAT88(spks) plots scatter plots of the (spontaneous activity) spikes in SPKS. It stacks thin horizontal raster plots for each channel. Within each plot, spikes are positioned based on their height (amplitude).
SPKS must have been loaded using LOADSPIKE or LOADSPIKE_NOC.
Column-row numbers are computed from channel numbers using hw2cr.
p = HEIGHTSCAT88(spks) returns the plot handle.

- **hist2dar**
mat=hist2dar(X,Y,nx,ny,flag) returns a matrix suitable for pcolor containing the crosstab counts of X and Y in automatically selected bins: you pick the number of bins, the code determines the edges based on the min and max values in X and Y.
If optional argument FLAG is present, the result is also plotted.

- **hist2d**
mat=hist2d(X,Y,x0,dx,x1,y0,dy,y1,flag) returns a matrix suitable for pcolor containing the crosstab counts of X and Y in the bins edges defined by x0:dx:x1 resp y0:dy:y1.
If optional argument FLAG is present, the result is also plotted. [mat,xx,yy]=hist2d(...) returns x and y arrays as well, so you can call pcolor(xx,yy,mat).

- **hw2cr**
[c,r] = HW2CR(hw) converts the hardware channel number hw (0..63) to column and row numbers (1..8).
cr = HW2CR(hw) converts the hardware channel number hw (0..63) to a 1+row+10*col format (11..88). hw may be a matrix, in which case the result is also a matrix.
Illegal hardware numbers result in c,r == 0.

- **loaddesc**
  d = LOADDESC(fn) reads the description file FN or FN.desc and returns a structure with values for each line read.
  The fields are named from the label in the description file.
  All values are converted to double. Original strings are stored in the field LABEL_str.
  Repeated keys are stored in a cell array.

- **loadraw**
  y=LOADRAW(fn) reads the raw MEA datafile fn and stores the result in y.
  y=LOADRAW(fn,range) reads the raw MEA datafile fn and converts the digital values to voltages by multiplying by range/2048.
  Range values 0,1,2,3 are interpreted specially:

range value	electrode range (uV)	auxillary range (mV)
0	3410	4092
1	1205	1446
2	683	819.6
3	341	409.2

  "electrode range" is applied to channels 0..59, auxillary range is applied to channels 60..63. Note that channel HW is stored in the (HW+1)-th row of the output.

- **loadspike**
  y=LOADSPIKE(fn) loads spikes from given filename into structure y with members time (1xN) (in samples) channel (1xN) height (1xN) width (1xN) context (75xN) thresh (1xN)
  y=LOADSPIKE(fn,range,freq_khz) converts times to seconds and width to milliseconds using the specified frequency, and the height and context data to microvolts by multiplying by RANGE/2048.
  As a special case, range=0..3 is interpreted as a MultiChannel Systems gain setting:

range value	electrode range (uV)	auxillary range (mV)
0	3410	4092
1	1205	1446
2	683	819.6
3	341	409.2

  "electrode range" is applied to channels 0..59, auxillary range is applied to channels 60..63.
  In this case, the frequency is set to 25 kHz unless specified.

- **loadspike_noc**
  y=LOADSPIKE_NOC(fn) loads spikes from given filename into structure y with members time (1xN) (in samples) channel (1xN) height (1xN) width (1xN) thresh (1xN)
  Context is not loaded.
  y=LOADSPIKE_NOC(fn,range,freq_khz) converts times to seconds and width to milliseconds using the specified frequency, and the height and context data to microvolts by multiplying by RANGE/2048.
  As a special case, range=0..3 is interpreted as a MultiChannel Systems gain setting:

range value	electrode range (uV)	auxillary range (mV)
0	3410	4092

1	1205	1446
2	683	819.6
3	341	409.2

"electrode range" is applied to channels 0..59, auxillary range is applied to channels 60..63.
In this case, the frequency is set to 25 kHz unless specified.

- **randscat88**
  RANDSCAT88(spks) plots scatter plots of the (spontaneous activity) spikes in SPKS. It stacks thin horizontal raster plots for each channel. Within each plot, spikes are randomly positioned vertically for clarity.
  SPKS must have been loaded using LOADSPIKE or LOADSPIKE_NOC.
  Column-row numbers are computed from channel numbers using hw2cr.

- **timeclust**
  [t0,cnt,dt] = TIMECLUST(tms_s,bin_s,thr_mean,thr_abs)
  Given a set of times TMS_S and a bin size BIN_S (both nominally in seconds), find the locations, volumes, and widths of peaks in the time distribution.
  A peak is (primitively) defined as a contiguous area of bins exceeding
  THR_ABS **and** exceeding THR_MEAN times the mean bin count. Either THR_MEAN or THR_ABS may be left unspecified, in which case THR_ABS defaults to >=2.

Chapter 7

File formats

The current version of MEABench uses two binary file formats: RAW, and SPIKE. All other files are plain text.

7.1 Raw files

Raw files contain electrode voltage values in digitized form. The scaling factor between digital values and microvolts is not stored in the .raw file, but in an accompanying .raw.desc file. The number of channels in a raw file is currently fixed to 64, and not noted in the .raw.desc file. This will be changed in a future release. File format is: one 16-bit integer for every channel, 64 channels per scan, repeated for the length of the file. Thus, in (almost) C notation, a raw file would be defined as:

```
typedef short int scan[64];
typedef scan rawfile[];
```

7.2 Spike files

Spike files contain information about detected spikes. The time of detection (in samples) as well as the channel number, height (in digital units) and width (in samples) are stored in a structure that also contains a limited amount of 'context', or sample values immediately surrounding the spike. The BandFlt and AdaFlt spike detectors also record the threshold used for detecting that particular spike in the structure. In (almost) C notation, a spike file would be defined as:

```
typedef struct {
  long long time; // 64 bits of time (in sample periods from start o   f file)
  short channel; // a channel number, 0–63
  short height; // height from baseline, possibly negative (in digital units)
  short width; // the width of the spike (in samples)
  short context[74]; // 24 samples before and 49 samples after peak
  short threshold; // detection threshold used for this spike
} spike;
typedef spike spikefile[];
```

As for raw files, conversion factors may be found in a description file (.spike.desc).

Chapter 8

Hints and tips

This section contains solutions to some common problems and provides some hopefully helpful hints.

8.1 Help! None of the programs will run.

You may find that none of the programs will run, and complain like this:

> /opt/meabench/bin/replay: error in loading shared libraries: libmea.so: cannot open shared object file: No such file or directory

This means that they cannot find the common libraries *libmea.so* or *libmeagui.so*. This problem may be fixed by typing

```
export LD_LIBRARY_PATH=/opt/meabench/lib:$LD_LIBRARY_PATH
```

before running the command. Similarly, if the perl programs complain about missing libraries, try:

```
export PERL5LIB=/opt/meabench/libexec:$PERL5LIB
```

(If you installed MEABench somewhere else, you'll know how to modify the above lines.)

If you use **mea** this problem is less likely to show.

8.2 Shared memory problems

You may find that a server program refuses to run, complaining like this:

> Error from ShmSrv: Segment exists, please delete using 'ipcrm shm 3417399'

This normally means that the server crashed on a previous run, leaving behind the shared memory segment used for its output stream. Executing the suggested command often solves the problem, but you may be rewarded by this when you rerun the program:

> Error from ShmSrv: Segment exists, and cannot be accessed. Any lingering clients? Before deleting the segment using ipcrm (see manpage) these may have to be stopped.

This means that some client is still connected to the (defunct) crashed server. Quitting and restarting such clients is a pretty sure way of solving the problem. If quitting the client is undesirable, most clients can be convinced to renegotiate the connection to their server using the command **source**. Once the clients have been cleared, the segment normally goes away spontaneously. If the problem persists, you may have to resort to **ipcs** and **ipcrm**. See the linux man pages for details.

Error from ShmSrv: creat

This exceedingly unhelpful error message may mean that you don't have a .meabench subdirectory in your home directory. This is where MEABench tries to link all its shared memory segments and wakeup sockets. To make this problem go away for ever, create the directory using `mkdir ~/.meabench`.

8.3 Help! Client X keeps saying 'waiting for START from Y' and won't run.

Most clients synchronize the start of their run with the start of a run of the server they are connected to. To make this work smoothly, start the client off (typically by typing **run**) *before* starting the server. Some clients allow you to disable this synchronization behavior, e.g. through a **wait** command.

8.4 Abbreviating commands

All programs that provide the user with a command line accept non-ambiguous abbreviations for commands. For example, in most programs, **run** can be abbreviated to **r**.

8.5 Passing commands at run time

All programs that provide the user with a command line can also process commands passed at startup time. For example, instead of entering into the following dialog:

$ `spikedet`
spikedet> `type 3`
Type is BandFlt-25
spikedet> `threshold 4`
Threshold is 4
spikedet> `source 60hz`
Source is 60hz
spikedet>

you may also say:

$ `spikedet ty:3 thr:4 so:60hz`
Type is BandFlt-25
Threshold is 4
Source is 60hz
spikedet>

This is especially useful if you find yourself quitting and restarting a program frequently, since with the second scheme, the shell history facilities can be used.

At start up time, commands are separated from their arguments by a colon rather than a space, and arguments are separated from each other by a comma rather than a space.

8.6 Interrupting long commands

All command line programs respond to **Ctrl-C** by returning to their prompt. If they don't respond immediately, press **Ctrl-C** again after a second or two. Pressing **Ctrl-C** twice in quick succession kills the program forcefully, quite possibly leaving a shared memory segment behind as explained above. Pressing **Ctrl-C** while the program is waiting for user input no longer exits the program. All programs support a **quit** command to exit cleanly, but **Ctrl-D** (end of input) is also an acceptable way to exit them.

8.7 Debugging information

Most programs support two additional commands to aid debugging. These are:

- **dbx** [0/1] — Enables (1) or disables (0) debugging output. Debugging output varies wildly from component to component and from release to release. Mainly of use for developers, who can sprinkle their code with sdbx(...) calls to track bugs down.
- **clients** — Prints a list of all clients currently connected to this server program. Mainly intended for debugging, the output format is not very user friendly.

8.8 Contacting the author

For further information, or to report bugs, please contact:

Daniel Wagenaar
UC San Diego, Neurobiology Section 9500 Gilman Drive m/c 0357
La Jolla, CA 92093
dwagenaar@ucsd.edu

Suggestion for improvement are always welcome, but I cannot guarantee I'll have time to implement them. You too can contribute by sending me your bug fixing and feature adding patches by e-mail. Please use 'diff –C2' against the latest public release version.

The latest versions of MEABench and this documentation can be found on

`http://www.its.caltech.edu/~pinelab/wagenaar/meabench.html`

8.9 Reporting bugs

When reporting bugs, please include the following information:

- The version of MEABench you are using.

- The version of g++ you are using (type `g++ --version` to get this information).
- The version of Qt you are using (type `moc -v` to get this information).
- The output of the 'configure' script.
- If compilation failed: the output of the 'make' command (not just the lines containing the error).
- For runtime errors: Which programs you are running, which commands you have executed within those programs, and a description of what you are trying to do as well as any relevant screen output. A succinct set of conditions that reproducibly produces the error is very much appreciated.