The mocha MEX File, Making the Most of EPICS from MATLAB with CAFE

Jan Chrin
Paul Scherrer Institut

Takes full advantage of the CAFE C++ Channel Access library:

- Proper connection and re-connection management
- Memory optimization, especially as connections are broken and restored
- Separation between data acquisition and presentation
- Caching of all data related to the channel and its connection state
- Aggregation of requests to enhance performance
- CAFE ca client temporally decoupled from ca server

mocha is just the MATLAB wrapper to CAFE where all the hard work is already done!

*N.B. mca scripts have all been **mocha**-fied, hence MATLAB scripts using mca can continue to run without further modification*

mocha Format

mocha methods invoking *data operations*, take the general form of:

```
[ret1,ret2,ret3,…] = mocha (input1,input2,input3,…);
```

input1: 'message' e.g. 'open','get','set','monitor'…

input2: <pvHandle> is a handle or 'pvName' (handle is an object reference of numeric value; handle is returned by 'open' operation)

input3: (for 'set'): data input: scalar, vector, of any MATLAB data type

input3: (for 'get') is optional: data type of ret val if not 'native': 'int8', 'int16', 'int32', 'int64', 'uint8','uint16','uint32','uint64','single','double'

ret1 (for 'get'): data output: scalar, vector
ret1 (for 'set'): status (1 is normal termination)
ret2 (for 'get'): status (!1 is an error)

A Quick Start: Basic Single Channel Operations

Simple operations on a single channel by *PV Name*

```
% val (scalar or vector) returned in native data type
[val, stat] = mocha ('get', 'pvName');

% val (scalar or vector) returned as a MATLAB double
[val, stat] = mocha ('get', 'pvName', 'double');

% val (scalar or vector) returned as a MATLAB uint8 type
[val, stat] = mocha ('get', 'pvName', 'uint8');

% val (scalar or vector) in set can be in any data type
[status] = mocha ('set', 'pvName', val);
```

Remark: Data transmitted over the network is in the native data type (*), but data may be presented to user in any meaningful types

(*) except for waveforms where it is the smaller of that requested and native

A Quick Start: Basic Single Channel Operations

Simple operations on a single channel by *handle*

```
% handle returned from open method
handle = mocha ('open', 'pvName');

% val (scalar or vector) returned in native data type
[val, stat] = mocha ('get', handle);

% val (scalar or vector) returned as a MATLAB double
[val, stat] = mocha ('get', handle, 'double');

% val (scalar or vector) returned as a MATLAB uint8 type
[val, stat] = mocha ('get', handle, 'uint8');

% val (scalar or vector) can be in any data type
[status] = mocha ('set', handle, val);
```


A Quick Start: Handle to pvName mapping

Handles and 'pvname' mapping:

```
handle = mocha ('getHandleFromPV','pvname');

pvname = mocha ('getPVFromHandle',handle);
```

```
% Array of handles and matching pv names
[handles, pvs] = mocha ('getHandles');

% Array of handles, pv names and ca connection states
[handles, pvs, states] = mocha ('getHandleStates');

mocha('printHandles',[<pvHandle>]) %print info on subset

mocha('printHandles') %print info on all handles
```

A Quick Start: Basic Single Channel Operations

Simple operations on a single channel, **returning data, alarms, timestamp**

```
handle = mocha ('open','pvName'); % optional

% dStruct.val (scalar/vector) returned in native type
dStruct = mocha ('getPV',<pvHandle>);

% dStruct.val (scalar/vector) returned as MATLAB double
dStruct = mocha ('getPV',<pvHandle>,'double');
```

```
dStruct.val
dStruct.status              (int32)
dStruct.alarmStatus         (int16)
dStruct.alarmSeverity       (int16)
dStruct.ts [year,month,day,hour,min,sec,nsec] (uint32)
```


A Quick Start: Basic Single Channel Operations

Other ways to retrieve **alarms, timestamp**

```
ts=mocha('getTimestamp', <pvHandle>);

ets=mocha('getETS', <pvHandle>); [secPastEpoch, nsec]
```

```
[alarmStatSev] = mocha('getAlarm', <pvHandle>);
          [0 0] %1x2 int16 row vector

[alarmStatSev] = mocha('getAlarmAsString', <pvHandle>);
{'NO_ALARM' 'NO_ALARM'} %1x2 cell array
```

```
alarmStatSev(1) gives alarm Status
alarmStatSev(2) gives alarm Severity
```

A Quick Start: Understanding the status error code

```
[value, status] = mocha ('get', 'pvName');

dStruct = mocha ('getPV',<pvHandle>);  % dStruct.status
```

Status code has an enumerated value (named value)

```
%enumerator name
statusText=mocha
('getStatusText',status/dStruct.status);

%epics/cafe information concerning the status code
statusInfo=mocha ('getStatusInfo',status);
```


A Quick Start: Understanding the status error code

Error Codes <=480 are generated by underlying ca library (34 error codes)
Error Codes =>600 are generated by the CAFE interface (57 error codes)

Status Code	Status Enumeral	Status Info
	ret = mocha ('getStatusText', status)	ret = mocha ('getStatusInfo', status)
1	SUCCESS:ECA_NORMAL	Normal successful completion
600	CHANNEL STATE: ICAFE_CS_NEVER_CONN	Valid chid; server not found or unavailable
604	CHANNEL STATE: ICAFE_CS_DISCONNECTED	Channel disconnected
80	ECA_TIMEOUT	User specified timeout on ID operation expired
1021	ECAFE_TIMEOUT	Callback function not activated within specified timeout period
1017	ECAFE_INVALID_HANDLE	Handle does not exist!
1018	ECAFE_INVALID_GROUP_HANDLE	Group handle does not exist!

A Quick Start: Understanding the status error code

Another way to retrieve all status information w.r.t. previous operation on a given handle:

```
[status, textArray] = mocha ('getStatus',<pvHandle>);
```

```
Example:
status = 1
textArray(1) = 'SUCCESS: ECA_NORMAL'
textArray(2) = 'Normal successful completion'


status = 600  % status !=1 is an error
textArray(1) = 'CHANNEL_STATE: ICAFE_CS_NEVER_CONN'
textArray(2) = 'Valid chid; server not found or unavailable'
```


A Quick Start: MATLAB Exceptions for single channel operations

Rather than have to check on the status code, CAFE MATLAB exceptions can be enabled (mocha default is with exceptions dis-enabled)

```
e=mocha('haveExceptions'); %returns true/false (default)

mocha('withExceptions', true) %enable exceptions
try
 [val, status] = mocha ('get',<pvHandle>);
catch ME
       if strfind(ME.identifier,'CAFE_WARN')
              warning(ME.message);
       elseif strfind(ME.identifier,'CAFE_INFO')
              disp(ME.message);
       elseif strfind(ME.identifier,'CAFE_ERROR')
              error(ME.message);
       else
              disp(ME.message); %shouldn't happen
       end
end
```

A Quick Start: MATLAB Exceptions for single channel operations

MException with properties:

```
identifier: 'CAFE_WARN:ICAFE_CS_NEVER_CONN‘

   message: 'PV=TEST:PV  Handle=9  statusCode=600
statusText=ICAFE_CS_NEVER_CONN  statusInfo=Valid chid;
server not found or unavailable‘

     cause: {0×1 cell}
     stack: [0×1 struct]
```


Opening Channels

```
        handle = mocha ('open', 'pvname');
    [handles] = mocha ('open', {'pv1','pv2',…});
```

```
When 'opening' a channel, the default behaviour is for
each open to wait for a fixed time period, given by:

openTime = mocha ('getOpenWaitTime'); %currently 0.4s

The 'open' wait time can be configured by the user:

mocha ('setOpenWaitTime',0.2); %set to 0.2s
mocha ('setOpenWaitTime'); %reset to default (0.4s)
```

Remark: A valid handle is always returned even if the channel does not connect after the specified time. *It is not necessary to attempt a 2nd open on a disconnected channel*. It will connect when it is able to, e.g., ioc is busy and may need a few ticks more; if the ioc is currently down, the channel will connect on ioc boot.

Best Practise: Opening Multiple Channels

Best practice: Opening several channels *concurrently*

```
pvArray={'pvName1', 'pvName2', 'pvName3', 'pvName4'};
```

```
mocha ('openPrepare');
[h]   = mocha ('open', pvArray);
hpv   = mocha ('open','pvName4'); %pvName as for h(4)
...   = mocha ('open', …. );
mocha ('openNowAndWait', 0.5); %wait 0.5s, concurrent
```

Note that here h(4)==hpv since only one handle is created for any given pv

Connection States of Channels

```
%Test if channel is connected; returns true/false
logicalValue = mocha ('isConnected', <handlePV>);
logicalValue = mocha ('allConnected');

%Array of handles and matching pv names
[handles, pvs] = mocha ('getHandles');
[handles, pvs] = mocha ('getDisconnectedHandles')
[handles ,pvs] = mocha ('getConnectedHandles')


%Array of handles, pv names and connection states (1/0)
[handles, pvs, states] = mocha ('getHandleStates');

 mocha('printHandles',[<pvHandle>]) %print given handles
 mocha('printHandles') %print all handles
 mocha('printDisconnectedHandles'); %diagnostics
```

Closing Channels

Close channels when the application no longer needs them or before exit

```
mocha ('close',  handlePV );  %close given channel

mocha ('close', [handlePV]);  %close given channels


%close all channels and release all ca resources
%before exiting MATLAB
mocha ('close');
```


Accessing Channel Information

```
pvInfo = mocha ('getInfo',<pvHandle>);
```

```
pvInfo.channelID          '0x7f66cc7a7de0'
pvInfo.connectFlag        1
pvInfo.hostName           'psi-softioc-2.psi.ch:45408'
pvInfo.dataType           'DBR_FLOAT'
pvInfo.className          'waveform'
pvInfo.accessRead         1
pvInfo.acccessWrite       1
pvInfo.nelem              4080
pvInfo.connectionState    `CA_OP_CONN_UP'
```

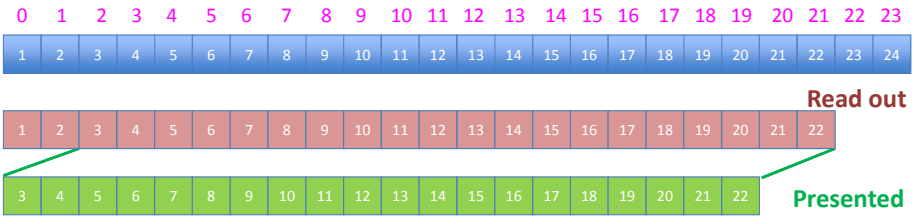
Waveforms

```
[val]=mocha('get',<handlePV>,<dataType>); %Returns array
```

The default behaviour is for all the elements of the wf to be read-out and presented.
However, this number together with an offset, can be configured by the user:

```
nelemWF = mocha ('setNelem', <handlePV>,  20);
offset  = mocha ('setOffset',<handlePV>,   2);

[val]=mocha('get',<handlePV>) %returns 20 from [2-22]
```



Handling Enumerated Data Types

```
pvInfo = mocha ('getInfo', <pvHandle>);
pvInfo.dataType % DBR_ENUM if enumerated type
```

Consider an mbbo record with enumerated value/names
0: off 1: on 2:{transient state}
with current value 1, i.e. 'on'
Default behaviour is to present the value as a string:

```
[val, status] = mocha('get',<handlePV>);  %val='on'
```

unless a numeric datatype is specified:

```
[val, status] = mocha('get',<handlePV>, 'int8'); %val=1
```

For set, a numeric input value will be sent to the ioc provided it has a valid value.
For set, a string input (name) value will be internally converted to the enum value.
If no match is found, then the string is examined for a numeric value:

```
status = mocha('set',<handlePV>, 'off' );
status = mocha('get',<handlePV>,'0'); %will set to 'off'
```

Multiple Scalar Gets

Messages are collected and sent with one call; underlying ca methods are asyn-chronous but mocha method is blocking, i.e., waits until all handles have reported

```
[val,isAllOK,s] = mocha('getScalarArray',[hpv],'string');
```

Array of handles or PVs

Data type of returned values is optional; otherwise native type of first handle/pv, hpv(0) is selected

`val` is array of returned values
`val(i)` corresponds to value returned by `hpv(i)`
If handle points to a waveform, only the first element of the waveform is returned (hence the `'getScalarArray'`)

`isAllOK` is 1 if all status replies are 1, else error code of 1st failing handle returned

`s` is an array of returned error codes; `s(i)` corresponds to status returned by `hpv(i)`

Multiple Scalar Sets

Messages are collected and sent with one call; ca methods are asynchronous but mocha method is blocking, i.e., waits until all handles have reported)

```
[isAllOK, s] = mocha('setScalarArray',[hpv], [val]);
```

Array of handles or PVs
Array of data values

isAllOK is 1 if all status replies are 1, else error code of 1st failing handle returned

s is a vector of returned status replies; s(i) corresponds to status of operation for handle h(i)

Multiple `Compound' (Cell Array) Gets

Messages are collected and sent with one call; underlying ca methods are asyn-chronous but mocha method is blocking, i.e., waits until all handles have reported.

```
[val,isAllOK,s] = mocha('getCellArray',[hpv]);
```

Returns a Cell Array; elements within can be of any size and or type.

`val{i}` corresponds to value returned by `hpv(i)`

`isAllOK` is 1 if all status replies are 1, else error code of 1st failing handle returned

`s` is an array of returned error codes; `s(i)` corresponds to status returned by `hpv(i)`

Multiple `Compound' (Cell Array) Gets

```
hpv(1): scalar      hpv(2): waveform   hpv(3): enum
d{1} = int32(33)    d{2}=[2.6,2.8,2.7]   d{3}= 'on';
```

```
[val,isAllOK,s] = mocha('getCellArray',[hpv]);
```

Returns a Cell Array; elements within each cell can be of any size and type.

```
val = {[33],[2.6,2.7,2.8],'on'}

val{1} holds 33
val{2} holds [2.6,2.7,2.8]; %val{2}(1:3)
val{3} holds 'on'
```

`isAllOK` is 1 if all status replies are 1, else error code of 1st failing handle returned

`s` is an array of returned error codes; `s(i)` corresponds to status returned by `hpv(i)`

Multiple `Compound' Sets

```
hpv(1): scalar      hpv(2): waveform   hpv(3): enum
d{1} = int32(33)    d{2}=[2.6,2.8,2.7]   d{3}= 'on';
```

```
[isAllOK,s] = mocha('set',[hpv], d{:}); (*)

[val,isAllOK,s] = mocha('getCellArray',[hpv]);
```

```
val{1} = val{1}+ 1;
val{2} =[1,2,3];
val{3} = 'off'
```

```
[isAllOK,s] = mocha('set',[hpv], val{:});
```

`isAllOK` is 1 if all status replies are 1, else error code of 1st failing handle returned
`s` is an array of returned error codes; `s(i)` corresponds to status returned by `hpv(i)`

(*) uses `'setPVArray'` which also has the the following pseudonymns: `'setStructArray'` `'setMany'`

Multiple `Struct' Gets

Messages are collected and sent with one call; underlying ca methods are asyn-chronous but mocha method is blocking (i.e., waits until all handles have reported)

```
[dStruct, isAllOK] = mocha('getPVArray',[hpv],'string');
```

Array of handles/PVs

Data type of returned values is optional, otherwise native type of *each PV* returned

`dStruct` is an array of returned structs with members:
`dStruct.val, dStruct.status, dStruct.alarmStatus, dStruct.alarmSeverity, dStruct.ts`

`dStruct(i)` corresponds to data returned by `hpv(i)`
If handle/pv points to a waveform, the default behaviour is that *all elements of the waveform* is returned
`isAllOK` is 1 if all `dStruct.status` replies are 1, else the error code of the first failing handle is returned

Multiple Sets for Scalars or Vectors

Messages are collected and sent with one call; underlying ca methods are asyn-chronous but mocha method is blocking, i.e., waits until all handles have reported

```
[isAllOK, s] = mocha('set',[hpv], d1,d2, …); (*)
```

Array of handles/PVs
Sequence of data inputs of *any datatype*

No. of data inputs must equal no. of handles

s is a vector of returned status replies;
s(i) corresponds to status of operation for handle/pv hpv(i)

isAllOK is 1 if all status replies are 1 (OK), else the error code of the first failing handle is returned

(*) uses 'setPVArray' which also has the the following pseudonymns: 'setStructArray' 'setMany'

Example: Multiple Gets and Sets for Scalars or Vectors

Messages are collected and sent with one call; underlying ca methods are asyn-chronous but mocha method is blocking, i.e., waits until all handles have reported

```
hpv(1): scalar      hpv(2): waveform   hpv(3): enum
d{1} = double(3.3)  d{2}=[2.6,2.8,2.7]   d{3}="on";
```

```
                                        %data input
[isAllOK, s] = mocha('set', [hpv], d{1:3} …); (*)
```

No of data inputs must equal no of handles/PVs

```
[dStructN, isAllOK] = mocha('getPVArray',[hpv]);

dStructN(2).val=(1:3) %change waveform data

[isAllOK, s] = mocha('set',[hpv], dStructN.val);
```

(*) 'set' uses 'setPVArray'≡'setStructArray'≡'setMany'

Asynchronous, Non-blocking, Get Operations

Messages are collected but user determines when exactly to send message.
Note that there is no variable for the return data!

```
           s = mocha('getAsyn', handlePV);
[isAllOK,s] = mocha('getAsyn', [hpv]);
```

```
 mocha('sendNow');  %optional

%get cache methods are used to retrieve data; methods
will wait until any outstanding callback is complete
or a timeout is reached
[d, s]     = mocha('getCache',   hpv(1), 'double');
[dStruct]  = mocha('getPVCache', hpv(2), 'native');

%get scalarArray / get PVArray
[d, isAllOK, s] = mocha('getScalarArrayCache', hpv);
[d, isAllOK, s] = mocha('getCellArrayCache', hpv);
[dStructN, isAllOK] = mocha('getPVArrayCache', h);
```


Aggregating Set Operations

Messages are collected but user determines when exactly to send message

```
mocha('setPutPrepare',[hpv]); %single or array of hPVs
```

'set' method configured per handle with 'setPutPrepare';
set operation is buffered and only invoked on user
request (or when buffer is full)

```
         s = mocha('set', <handlePV1>, val1);
         s = mocha('set', <handlePV2>, val2);
             mocha('sendNow');
```

May also be useful for GUIs, e.g. slider

Reset to previous Put Policy

```
mocha('setPutWait',<handlePV>); % Synchronous Put
mocha('setPutNoWait',<handlePV>); % Asyn. Put is default
```

Synchronous Groups

A synchronous group, used to guarantee a set of ca requests have completed .
Define a group as follows:

```
simpleGroup ={'magnet1:set','magnet2:set','magnet3:set'};
complexGroup={'magnet:set','bpm:intensity','ps:mode'};
                % ao          waveform     mbbi (enum)
```

```
s = mocha('defineGroup','gMagnet',simpleGroup);

s = mocha('defineGroup','gSnapshot',complexGroup);

grpHandle1 = mocha('openGroup','gMagnet');
grpHandle2 = mocha('openGroup','gSnapshot');
```


Synchronous Groups Get Operations

Messages sent with one call; ca method is a synchronous group call, thus mocha method is blocking

```
grpHandle = mocha('openGroup','gMagnet');

[d, isAllOK] = mocha('getGroup',<grpHandleName>,'string');
```

Group handle or group name

Data type of returned values is optional, otherwise data pertaining to each PV is returned in its native type

`d` is an array of returned structs with members:
`d.val, d.status, d.alarmStatus, d.alarmSeverity, d.ts`

`d(i)` corresponds to data returned by i*th* group member.
To pick out a specific group member:

```
d(mocha('idxgrpmem','gMagnet','FIND1-MSOL10:I-SET')).val;
```

Synchronous Groups Set Operations

Messages sent with one call; ca method is a synchronous group call, thus mocha method is blocking

```
gh = mocha('openGroup','gSnapshot');
```

```
h1: scalar;          h2: waveform;       h3: enum
d{1}=double(3.3); d{2}=[2.6,2.8,2.7]; d{3}='on';
```

```
[s, isAllOK] = mocha('setGroup', <ghName>, d{:});
```

No of data inputs must equal no. of handles

```
[d, isAllOK] = mocha('getGroup',<ghName>);
```

```
d(1).val = d(1).val +0.1;
```

```
[s, isAllOK] = mocha('setGroup',<ghName>, d.val);
```


Configuring Groups through XML

```
status = mocha('loadXMLGroups','test.xml');
[groups] = mocha('listGroups');
[members] = mocha('listGroupMembers', 'gName');
```

A minimal xml configuration file

```
<cafe:group id="gRF-GET-BEAM-PHASE">
  <cafe:member>
    <cafe:name> SINEG01-RSYS:GET-BEAM-PHASE </cafe:name>
  </cafe:member>
  <cafe:member>
    <cafe:name> SINSB01-RSYS:GET-BEAM-PHASE </cafe:name>
  </cafe:member>
…
</cafe:group>
```

Monitors

```
monitorID = mocha ('monitor', <handlePV>);
```

```
[d, s] = mocha ('getCache',<handlePV>);  % data, d, (scalar or vector) returned in native data type

[d, s] = mocha ('getCache',<handlePV>,'double'); data, d, (scalar or vector) returned as a MATLAB double

[d, s] = mocha ('getCache',<handlePV>,'uint8'); data, d, (scalar or vector) returned as a MATLAB UINT8 type
```

```
s = mocha ('monitorStop', <handlePV>, monitorID); %stop monitor for handle with given monitorID
s = mocha ('monitorStop', <handlePV>); %stop all monitors for this handle
s = mocha ('monitorStop'); %stop all monitors for all handles
```


Monitors with Actions

```
monitorID = mocha ('monitor', <handlePV>, 'action');

mocha('monitorFlushEvent', <handlePV>) %flush <handlePV>
mocha('monitorFlushEvent') %flush action for all handles
```

'action' is typically a MATLAB script, e.g.,

'monAction(<handlePV>)' that retrieves data from cache:

d= mocha('getCache',<handlePV>) and posts it to a widget

Used in conjunction with MATLAB timer (see mcamontimer.m)

```
obj.t=timer( 'TimerFcn', 'mocha(''monitorFlushEvent'')',
'Period', 2.0, 'ExecutionMode', 'fixedSpacing')

obj.t=timer( 'TimerFcn', 'mocha(402)', 'Period', 2.0,
'ExecutionMode', 'fixedSpacing')
```

Retrieving Display Data

```
[cStruct, s] = mocha ('getCtrl', h1);  % get Ctrl display data
from IOC
[cStruct, s] = mocha ('getCtrlCache', h1); % get Ctrl display
                                            data from cache
```

```
              val: {'on'} % mbbi record
           status: 1
      alarmStatus: 0
    alarmSeverity: 0
        precision: 0
            units: ' '
    noEnumStrings: 3
      enumStrings: {'off'  'on'  'monitor'}
upperDisplayLimit: 0
lowerDisplayLimit: 0
  upperAlarmLimit: 0
  lowerAlarmLimit: 0
upperWarningLimit: 0
lowerWarningLimit: 0
upperControlLimit: 0
lowerControlLimit: 0
```


Timeouts for Set/Get Operations

```
[t_put,t_get]=mocha('getTimeout',<handlePV>);%per handlePV

[t_put_min, t_get_min]  = mocha ('getTimeout');  % min put
 and get timeout from among all handles


mocha('setTimeout', <handlePV>, val); %handle, put/get
mocha('setTimeout', val);  %all handles, put/get
mocha('setTimeoutPut', <handlePV>, val); %handle, put
mocha('setTimeoutPut', val);  %all handles, put
mocha('setTimeoutGet', <handlePV>, val); %handle, get
mocha('setTimeoutGet', val);  %all handles, get
```

Timeouts in CAFE are also self-regulating (within reason);
Configured internally by CAFE by nTries and δTimeout

Timeouts for Synchronous Group Operations

```
[t_put,t_get]=mocha('getSGTimeout',<grphName>);

[t_put_min, t_get_min]  = mocha ('getSGTimeout');  % min
 put and get timeout from among all group handles


mocha('setSGTimeout', <grphName>, val); %ghandle, put/get
mocha('setSGTimeout', val);  %all grp handles, put/get
mocha('setSGTimeoutPut', <grphName>, val); %ghandle, put
mocha('setSgTimeoutPut', val);  %all grp handles, put
mocha('setSGTimeoutGet', <grphName>, val); %ghandle, get
mocha('setSGTimeoutGet', val);  %all grp handles, get
```

Timeouts in CAFE are also self-regulating (within reason);
Configured internally by CAFE separately by nTries and δTimeout

Specialized Methods

```
mocha ('show');  % will list all available methods
```

```
% set a PV and immediately retrieve its value
[getValue, status]
        = mocha ('setAndGet', <handlePV>, setValue);
```

A lot more in the CAFE C++ library can be exposed to MATLAB on request, e.g.,

`setAndMatch for n-dimensional (e.g. ID gap) scans`

Method sets n PVs followed by readback of n corresponding PVs; method verifies whether or not the set/readback values agree within the given tolerance and timeout

Summary

CAFE/mocha highlights:

- Synchronous, asynchronous interaction for individual, collection/groups of channels (optional XML configuration)
- Proper (re-)connection management and memory optimization
- Error messages caught and reported with integrity in every eventuality
- Compilations for 32/64 bit architectures, Linux & Windows
- Stable and robust!
- Simple, yet powerful, user-friendly interface

Finally…

- CAFE Home Page: `http://ados.web.psi.ch/cafe`
- load cafe-matlab module from AIT (Achim Gsell):
  ```
  module avail
  load cafe-matlab2016b/1.0.0
  ```
- or if another matlab module already loaded
  ```
  switch cafe-matlab2016a/1.0.0 cafe-matlab2016b/1.0.0
  ```
- On Windows, download from switch cloud:
  `https://drive.switch.ch/index.php/s/WaAVHCqYpGTc9N4`
- Need help with mocha? Just ask (really!)