TINE Release 5.x.x News

(June 12, 2019: inching toward perfection …)

“What a long, strange trip it’s been ….”

Release 5.1.0

- Core team now versed in the deep details ...
  - connection tables,
  - request/response scenarios
  - contract coercion
  - etc. …
- Release 5.0.0 seen to be very stable
  - And in a *mixed environment* as well!
    - many Release 4.x.x clients and servers still operating …
  - *Some exotica:*
    - missing/delayed callback with CA_NOCALLBACKS + CM_EVENT update mode.
    - occasional unnecessary double sends when client-renewal threshold is in use.
    - unnecessary contract synchronization with extremely large payload contracts.
    - alarm watch table alarms using property-oriented multi-channel arrays.
- Declare **5.1.0**
  - Build id : 5536 (C –library)
  - Build id : 5529 (java - library)

Release 5.1.0

- New Local History Features
  - (*we should have done this years ago!*)
  - <p>.HIST calls can accept WRITE access
    - forces a ***save*** of the short-term (main memory) history data into the *saved area*.
      - the *saved area* is never removed.
      - a form of *local event/post-mortem archive* !
      - e.g. a wacky-pulse recognizer sees a strange modulator pulse and issues a save command.
  - <p>.HIST calls now accept **key-value** strings as input
    - Input sometimes more complicated than simply ‘from – to’

Release 5.1.0

- <p>.HIST input :

This same information is available via a call to the stock property "PROPERTIES" with a request for output type CF_STRU

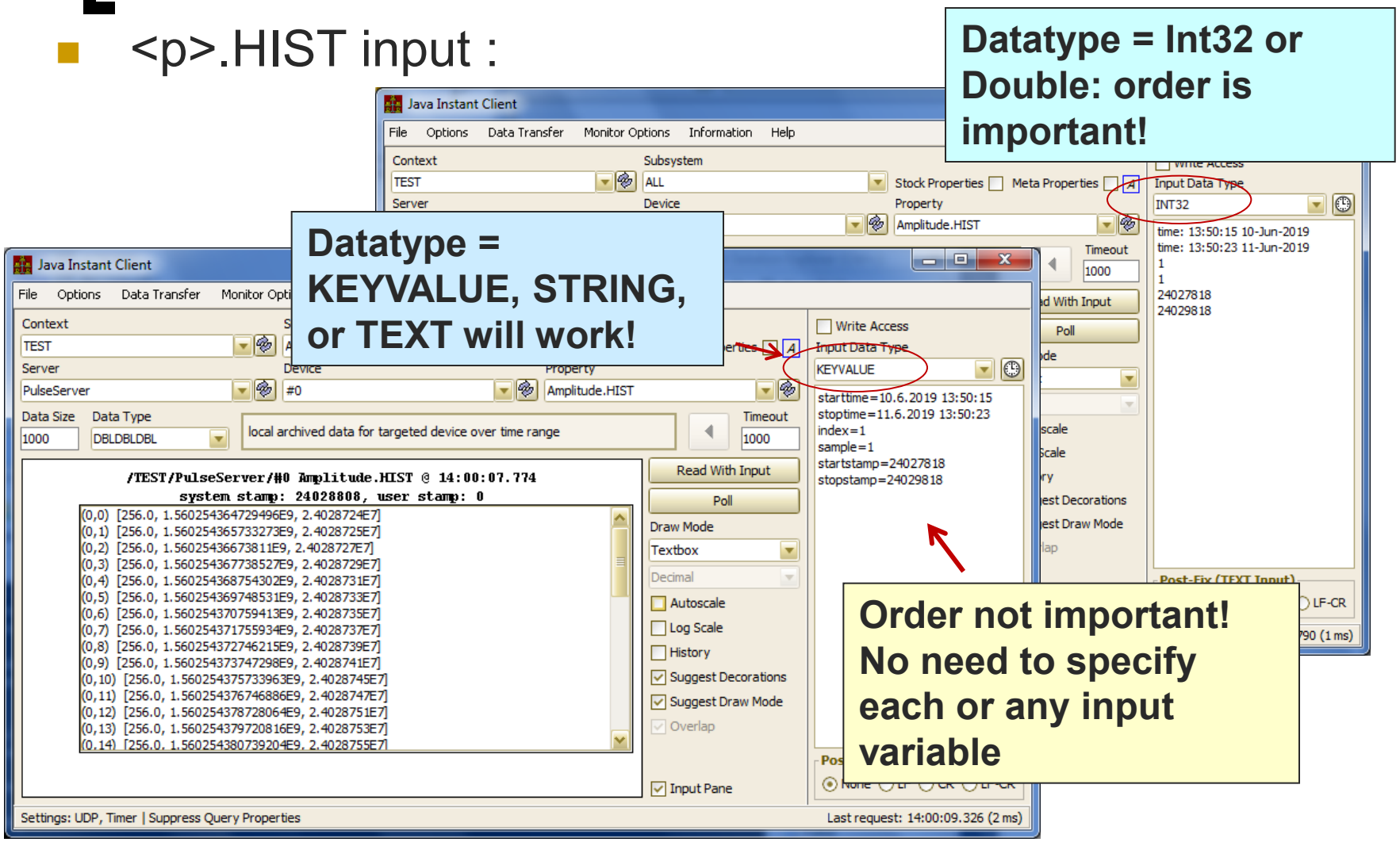
- **".HIST"** (synonym **".HST"**) returns the local history for the associated property. This meta property supports server out call should return an array of values over a time range and therefore needs to return not only values but timestamps. Th doublet type capable of containing the original property's format as will as a timestamp (for example CF_FLTINT). Suppo
  - CF_FLTINT value - timestamp pairs in the simplest variant (convert stored value to float and timestamp to UTC se
  - CF_DBLDBL value - timestamp pairs in a more general (albeit larger) variant (convert value to double and timesta
  - CF_INTFLTINT - traditional doocs format (UTC seconds - value - status triplet).
  - CF_HISTORY - all inclusive variant which embeds the original format and carries the full timestamp as well as othe use from the point of view of 'not missing anything'. However it is also a difficult format to use for the layman.

  Other output format types are possible but will not be listed here. In general one should make use of the standard histor **()**, **GetArchivedDataAsText()**, etc. This stock meta property also accepts input specifying the time range and other pa
  - CF_INT32 or CF_DOUBLE an array of up to 8 optional values. The first value gives the start time (UTC). The secor starting index (in case the stored property is an array). The fourth value is the sampling raster. The fifth and sixth of the *system* stamp value (e.g. the cycle or event number). If these two input values are indeed submitted, then data will be returned. If the seventh input variable is non-zero then the *user* stamp will be targeted instead of the the *field* index if the archived parameter is stored as a compound data type or a structure.
    - Index 0: starttime (UTC)
    - Index 1: stoptime (UTC)
    - Index 2: specific array index (if record is an array) => 0 means "just look take if from the input device nam
    - Index 3: sample raster (0 => find the 'best' raster for the given range)
    - Index 4: start system stamp (but only look for the system stamps within the time range given)
    - Index 5: stop system stamp (0 => use the current system stamp)
    - Index 6: if != 0 (and index 5 != 0) then index 4 and 5 refer to a search on the 'user stamp'.
    - Index 7: field index (in case the record is a struct or compound data type).

  If there is no input, or fewer than 4 CF_LONG or CF_DOUBLE values are passed, then the default stop time is the current extrapolated time based on the requested size of the call. The default index is '0', and the sampling raster is determined time range.
- **".HIST@"** (synonym **".HST@"**) returns the local history for the associated property. This call is a variation of the ".HIST will deliver the record at the specified time or next stored data if there is no record at the precise specified time. The retu returned. As this call does NOT deliver an set of data over a time range, the requested output format should be that of th where a targeted system stamp is specified as the fifth input parameter, in which case, the initial two input parameter m expected.

Release 5.1.0

- <p>.HIST input :

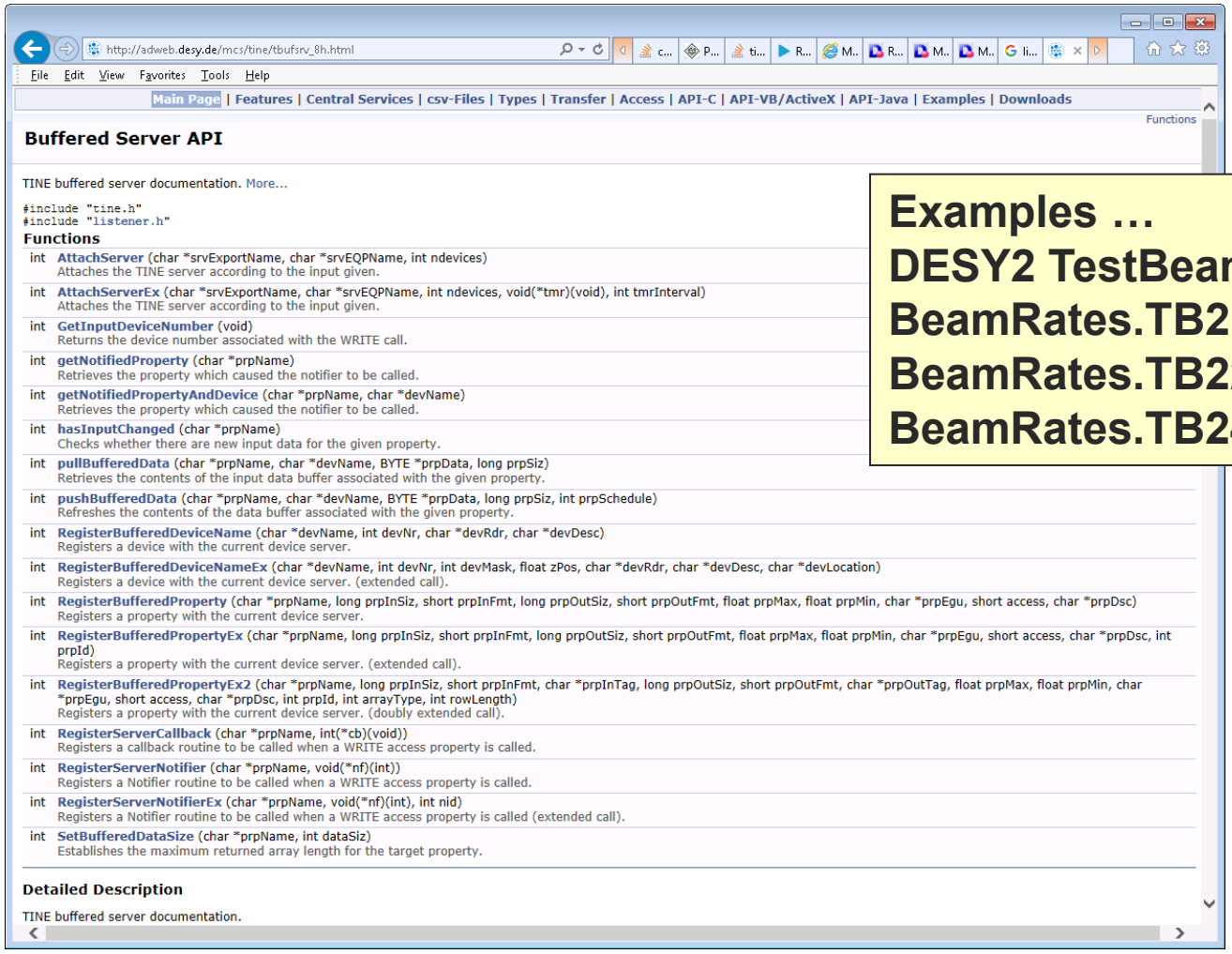


Release 5.1.0

- **Buffered Server …**
- Easiest way to write a server !
  - Directly in C/C++
  - LabView
  - MatLab
  - Python

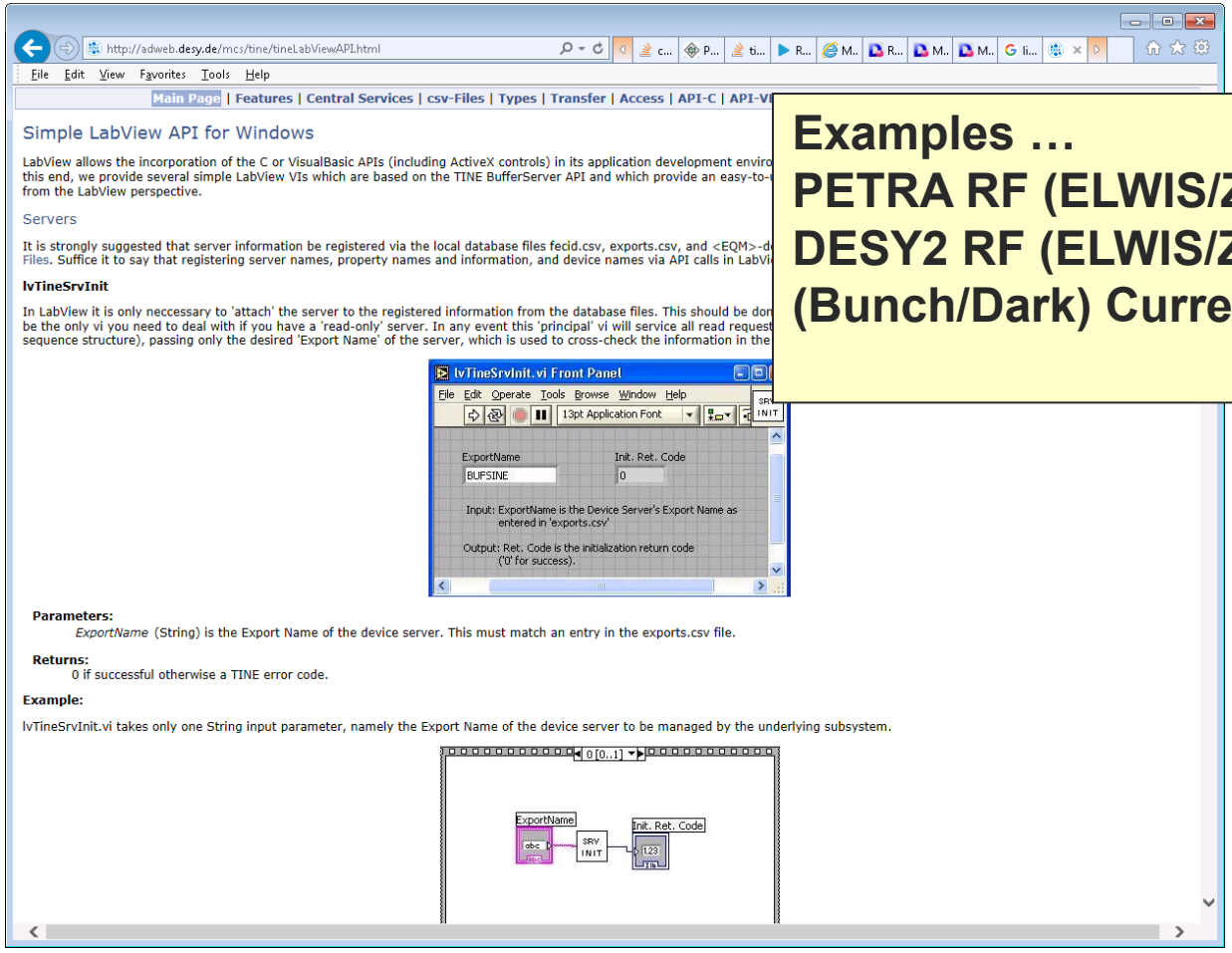
  - As yet no ‘buffered server’ in Java or .NET
    - Sorry: you’ll have to use the ‘full server API’

Buffered Server : C/C++



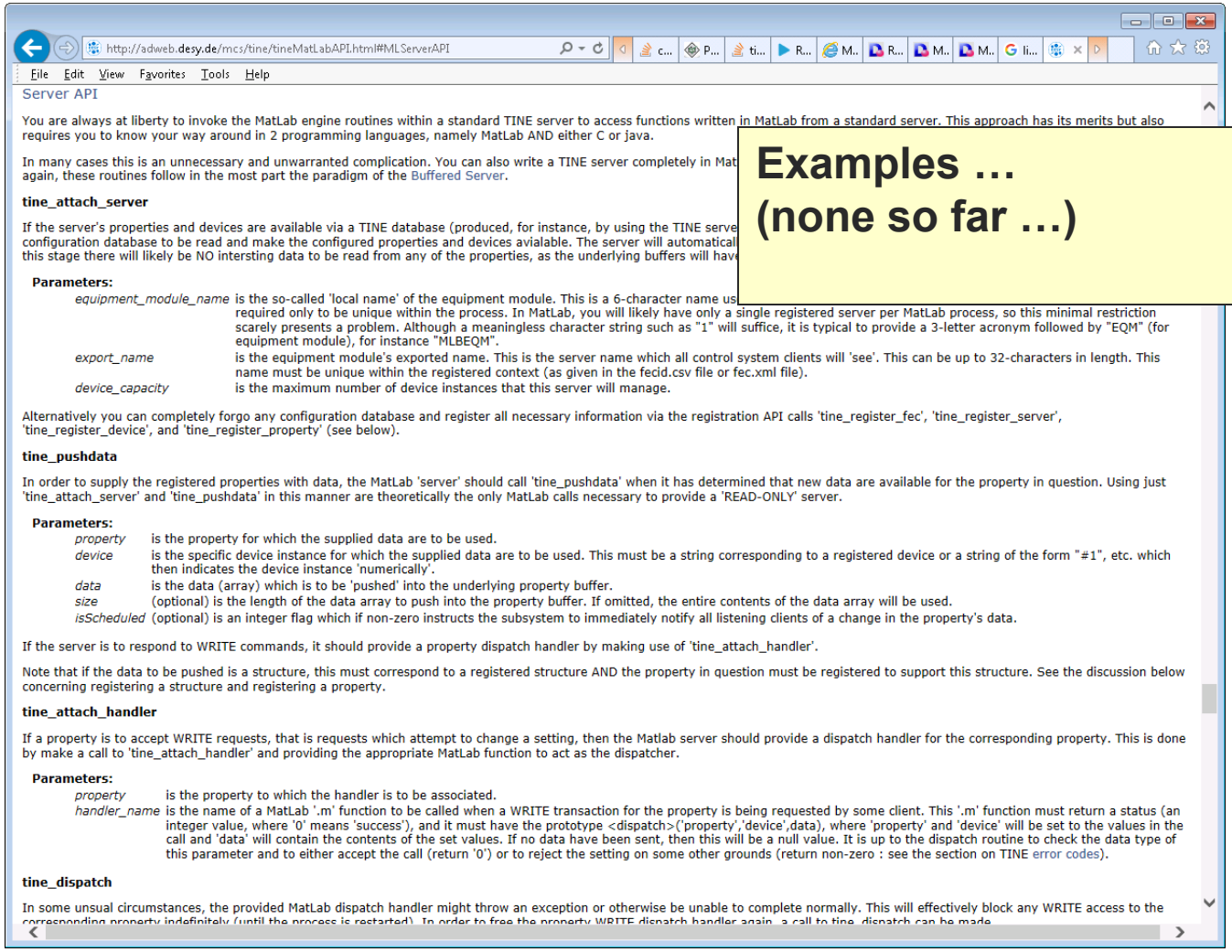
Examples …
DESY2 TestBeam:
BeamRates.TB21,
BeamRates.TB22,
BeamRates.TB24

Buffered Server : Labview



Examples …
PETRA RF (ELWIS/ZWERG)
DESY2 RF (ELWIS/ZWERG),
(Bunch/Dark) Current Monitors

Buffered Server : MatLab



http://adweb.desy.de/mcs/tine/tineMatLabAPI.html#MLServerAPI

File Edit View Favorites Tools Help

Server API

You are always at liberty to invoke the MatLab engine routines within a standard TINE server to access functions written in MatLab from a standard server. This approach has its merits but also requires you to know your way around in 2 programming languages, namely MatLab AND either C or java.

In many cases this is an unnecessary and unwarranted complication. You can also write a TINE server completely in Mat
again, these routines follow in the most part the paradigm of the Buffered Server.

tine_attach_server

If the server's properties and devices are available via a TINE database (produced, for instance, by using the TINE serve
configuration database to be read and make the configured properties and devices avialable. The server will automatical
this stage there will likely be NO intersting data to be read from any of the properties, as the underlying buffers will hav

Parameters:

equipment_module_name	is the so-called 'local name' of the equipment module. This is a 6-character name us required only to be unique within the process. In MatLab, you will likely have only a single registered server per MatLab process, so this minimal restriction scarely presents a problem. Although a meaningless character string such as "1" will suffice, it is typical to provide a 3-letter acronym followed by "EQM" (for equipment module), for instance "MLBEQM".
export_name	is the equipment module's exported name. This is the server name which all control system clients will 'see'. This can be up to 32-characters in length. This name must be unique within the registered context (as given in the fecid.csv file or fec.xml file).
device_capacity	is the maximum number of device instances that this server will manage.

Alternatively you can completely forgo any configuration database and register all necessary information via the registration API calls 'tine_register_fec', 'tine_register_server', 'tine_register_device', and 'tine_register_property' (see below).

tine_pushdata

In order to supply the registered properties with data, the MatLab 'server' should call 'tine_pushdata' when it has determined that new data are available for the property in question. Using just 'tine_attach_server' and 'tine_pushdata' in this manner are theoretically the only MatLab calls necessary to provide a 'READ-ONLY' server.

Parameters:

property	is the property for which the supplied data are to be used.
device	is the specific device instance for which the supplied data are to be used. This must be a string corresponding to a registered device or a string of the form "#1", etc. which then indicates the device instance 'numerically'.
data	is the data (array) which is to be 'pushed' into the underlying property buffer.
size	(optional) is the length of the data array to push into the property buffer. If omitted, the entire contents of the data array will be used.
isScheduled	(optional) is an integer flag which if non-zero instructs the subsystem to immediately notify all listening clients of a change in the property's data.

If the server is to respond to WRITE commands, it should provide a property dispatch handler by making use of 'tine_attach_handler'.

Note that if the data to be pushed is a structure, this must correspond to a registered structure AND the property in question must be registered to support this structure. See the discussion below concerning registering a structure and registering a property.

tine_attach_handler

If a property is to accept WRITE requests, that is requests which attempt to change a setting, then the Matlab server should provide a dispatch handler for the corresponding property. This is done by make a call to 'tine_attach_handler' and providing the appropriate MatLab function to act as the dispatcher.

Parameters:

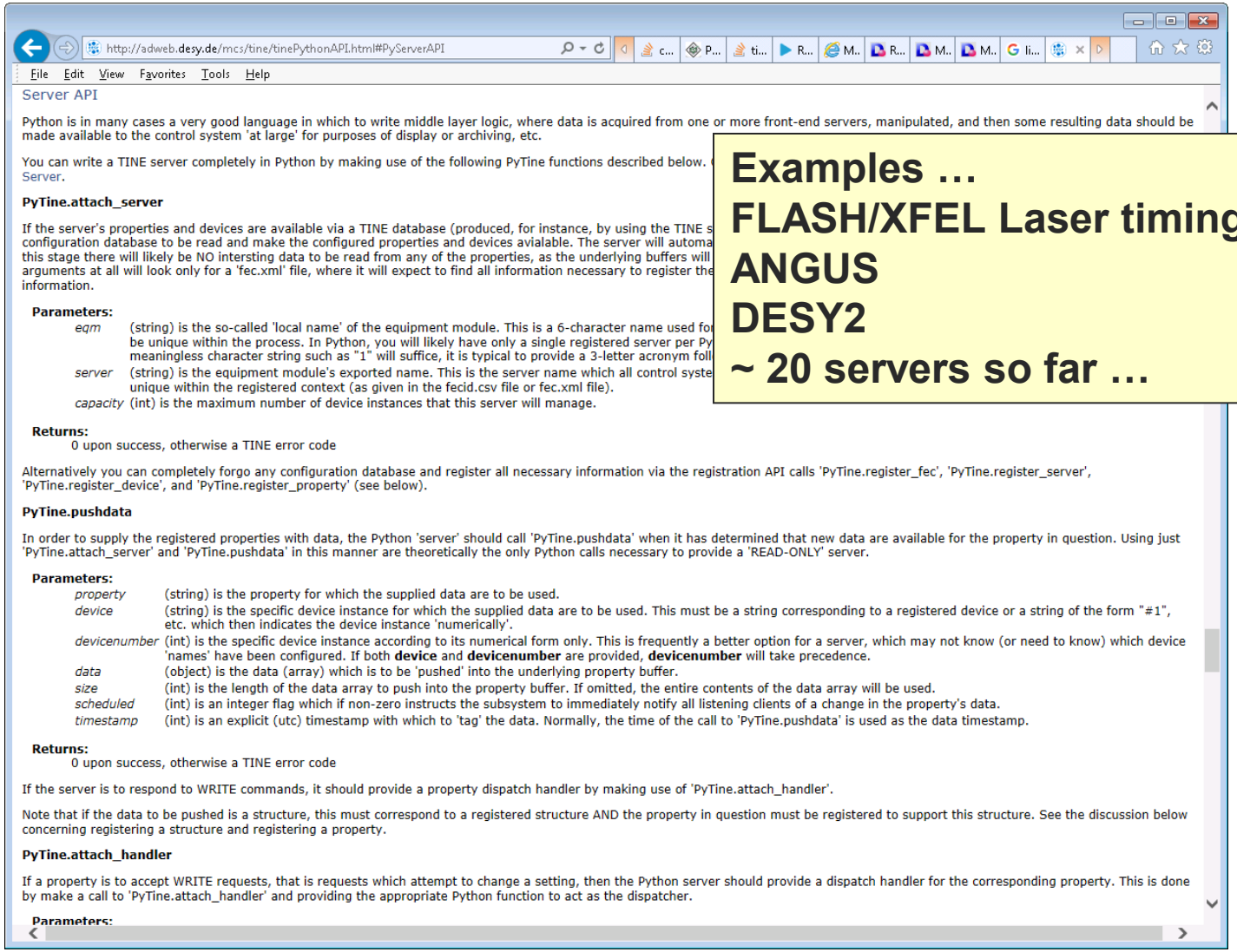
property	is the property to which the handler is to be associated.
handler_name	is the name of a MatLab '.m' function to be called when a WRITE transaction for the property is being requested by some client. This '.m' function must return a status (an integer value, where '0' means 'success'), and it must have the prototype <dispatch>('property','device',data), where 'property' and 'device' will be set to the values in the call and 'data' will contain the contents of the set values. If no data have been sent, then this will be a null value. It is up to the dispatch routine to check the data type of this parameter and to either accept the call (return '0') or to reject the setting on some other grounds (return non-zero : see the section on TINE error codes).

tine_dispatch

In some unusual circumstances, the provided MatLab dispatch handler might throw an exception or otherwise be unable to complete normally. This will effectively block any WRITE access to the corresponding property indefinitely (until the process is restarted). In order to free the property WRITE dispatch handler again, a call to tine_dispatch can be made

Examples …
(none so far …)

Buffered Server : Python



http://adweb.desy.de/mcs/tine/tinePythonAPI.html#PyServerAPI

File Edit View Favorites Tools Help

Server API

Python is in many cases a very good language in which to write middle layer logic, where data is acquired from one or more front-end servers, manipulated, and then some resulting data should be made available to the control system 'at large' for purposes of display or archiving, etc.

You can write a TINE server completely in Python by making use of the following PyTine functions described below. (Server.

PyTine.attach_server

If the server's properties and devices are available via a TINE database (produced, for instance, by using the TINE s configuration database to be read and make the configured properties and devices avialable. The server will automa this stage there will likely be NO intersting data to be read from any of the properties, as the underlying buffers will arguments at all will look only for a 'fec.xml' file, where it will expect to find all information necessary to register the information.

Parameters:

eqm (string) is the so-called 'local name' of the equipment module. This is a 6-character name used fo be unique within the process. In Python, you will likely have only a single registered server per Py meaningless character string such as "1" will suffice, it is typical to provide a 3-letter acronym foll

server (string) is the equipment module's exported name. This is the server name which all control syste unique within the registered context (as given in the fecid.csv file or fec.xml file).

capacity (int) is the maximum number of device instances that this server will manage.

Returns:

0 upon success, otherwise a TINE error code

Alternatively you can completely forgo any configuration database and register all necessary information via the registration API calls 'PyTine.register_fec', 'PyTine.register_server', 'PyTine.register_device', and 'PyTine.register_property' (see below).

PyTine.pushdata

In order to supply the registered properties with data, the Python 'server' should call 'PyTine.pushdata' when it has determined that new data are available for the property in question. Using just 'PyTine.attach_server' and 'PyTine.pushdata' in this manner are theoretically the only Python calls necessary to provide a 'READ-ONLY' server.

Parameters:

property (string) is the property for which the supplied data are to be used.

device (string) is the specific device instance for which the supplied data are to be used. This must be a string corresponding to a registered device or a string of the form "#1", etc. which then indicates the device instance 'numerically'.

devicenumber (int) is the specific device instance according to its numerical form only. This is frequently a better option for a server, which may not know (or need to know) which device 'names' have been configured. If both **device** and **devicenumber** are provided, **devicenumber** will take precedence.

data (object) is the data (array) which is to be 'pushed' into the underlying property buffer.

size (int) is the length of the data array to push into the property buffer. If omitted, the entire contents of the data array will be used.

scheduled (int) is an integer flag which if non-zero instructs the subsystem to immediately notify all listening clients of a change in the property's data.

timestamp (int) is an explicit (utc) timestamp with which to 'tag' the data. Normally, the time of the call to 'PyTine.pushdata' is used as the data timestamp.

Returns:

0 upon success, otherwise a TINE error code

If the server is to respond to WRITE commands, it should provide a property dispatch handler by making use of 'PyTine.attach_handler'.

Note that if the data to be pushed is a structure, this must correspond to a registered structure AND the property in question must be registered to support this structure. See the discussion below concerning registering a structure and registering a property.

PyTine.attach_handler

If a property is to accept WRITE requests, that is requests which attempt to change a setting, then the Python server should provide a dispatch handler for the corresponding property. This is done by make a call to 'PyTine.attach_handler' and providing the appropriate Python function to act as the dispatcher.

Parameters:

Examples …
FLASH/XFEL Laser timing
ANGUS
DESY2
~ 20 servers so far …

Release 5.1.0 (buffered server)

- *Getting started …*

Either :

- Attach to a database (.csv or fec.xml)

Or :

- Register server, properties, devices via API

Then :

- Push associated data when it changes

Release 5.1.0 (buffered server)

- Really simple sine server (c):

```c
#include <stdio.h>
#include "tine.h"
#include "tbufsrv.h"

#define NPOINTS 1024
float sinbuf[NPOINTS];

void update(void)
{
  int i;
  for (i=0; i<NPOINTS; i++)
  {
    sinbuf[i] = (float)(rand()%10.0) + 100.0 * (float)(sin(i*6.2832/(NPOINTS/8)));
  }
  pushBufferedData("Sine","SineDev0",(BYTE *)sinbuf,NPOINTS,FALSE);
}

int main(int argc,char *argv[])
{
  char c;
  AttachServerEx(NULL,NULL,0,update,500);

  SystemWaitCycleTimer();
  return 0;
}
```

Release 5.1.0 (buffered server)

- Really simple sine server (python) :

```
import PyTine as pt
import numpy as np
import random

N = 1024
ix = np.arange(N)
vals = np.zeros(shape=(N))

def updateSineCurve() :
  vals =  10.0 * random.random() +  100.0 * np.sin(2 * np.pi * ix / N)
  rc = pt.pushdata(property='Sine',device='SineDev0',data=vals.tolist())
  return;

rc = pt.attach_server()

for i in range(0,10):
  rc = pt.pushdata(property='Amplitude',device='SineDev0',data=ampl)

updateSineCurve()
```

Release 5.1.0 (buffered server)

- Are there any disadvantages?
  - Can only have 1 server per FEC.
  - Cannot overload properties.
  - Cannot have ‘READ with input’
    - Input is coupled to WRITE access !
  - Some aspects of property handling are not available (but nothing serious).
    - The registered property information is taken literally!

Release 5.1.0

- Python news:
  - PyTine now supported in

    python **2.7** -> **3.7**
  - PyTine.history() :
    - bug-fix: Depth string was getting clobbered …
    - more input specification available …
  - PyTine.set() and PyTine.call()
    - improved ‘best guess’ as to input size and format

Release 5.1.0 (python)

- Consider something like:

```
>>> PyTine.set(address='/TEST/SineServer/SineGen4',property='Amplitude',input=278)
```

  - How to send ‘278’ ?
    - Is it a floating point or integer value?
    - Does the server expect an array of some length?
  - PyTine (1st call) asks the server how property ‘Amplitude’ was registered and what it expects for WRITE commands …
    - bug-fix: if server did NOT register the property for WRITE calls, then the call above returned an error!

or explicitly pass : format=‘float’, size=1

```
>>> pt.set(
        'address','property'[,input,'format',size,timeout,'mode']
```

Release 5.1.0

- Best Guess …
  - determine what the input looks like (float, int, or string) and how many

```
>>> PyTine.set(address='/TEST/SineServer/SineGen4',property='Amplitude',input=278)
```

  - find/match to a registered WRITE property
    - no WRITE property ?
    - then use the read property attributes
    - If data type discovered then use it, else go with guess
- what if ?
  - (e.g. PETRA Kicker) property registered to accept 1 FLTINT and deliver 1 INTFLTINT) ?
    - Find the server programmer and ask him to change it ?
    - Must use PyTine.call() with mode=WRITE

```
pt.call(
        'address','property'[,input,'mode','format',size,'inputformat',inputsize,timeout]
```

Release 5.1.0

- PyTine.history()

```
pt.history(
           address,property[,'stop','depth','flags',timeout]
```

  - Two new arguments to offer more flexibility:
    - sample
    - numberPoints