



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_STRUCTURE`.
- **".HIST"** (synonym **".HST"**) returns the local history for the associated property. This meta property supports server output call should return an array of values over a time range and therefore needs to return not only values but timestamps. The doublet type capable of containing the original property's format as well as a timestamp (for example `CF_FLTINT`). Supported output formats:
 - `CF_FLTINT` value - timestamp pairs in the simplest variant (convert stored value to float and timestamp to UTC seconds)
 - `CF_DBLDBL` value - timestamp pairs in a more general (albeit larger) variant (convert value to double and timestamp to UTC seconds)
 - `CF_INTFLTINT` - traditional docs format (UTC seconds - value - status triplet).
 - `CF_HISTORY` - all inclusive variant which embeds the original format and carries the full timestamp as well as other information 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 history call (`GetHistory()`, `GetArchivedDataAsText()`, etc). This stock meta property also accepts input specifying the time range and other parameters:
 - `CF_INT32` or `CF_DOUBLE` an array of up to 8 optional values. The first value gives the start time (UTC). The second value gives the stop time (UTC). The third value is the starting index (in case the stored property is an array). The fourth value is the sampling raster. The fifth and sixth values are 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 *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 name"
 - 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 time extrapolated time based on the requested size of the call. The default index is '0', and the sampling raster is determined by the time range.
- **".HIST@"** (synonym **".HST@"**) returns the local history for the associated property. This call is a variation of the **".HIST"** call. It will deliver the record at the specified time or next stored data if there is no record at the precise specified time. The record is returned. As this call does NOT deliver an set of data over a time range, the requested output format should be that of the original property where a targeted system stamp is specified as the fifth input parameter, in which case, the initial two input parameters must be expected.

Release 5.1.0

- `<p>.HIST` input :

Datatype = Int32 or Double: order is important!

Datatype = KEYVALUE, STRING, or TEXT will work!

Order not important!
No need to specify each or any input variable

The screenshot shows the Java Instant Client interface with the following configuration:

- Context: TEST
- Subsystem: ALL
- Server: PulseServer
- Device: #0
- Property: Amplitude.HIST
- Input Data Type: KEYVALUE (circled in red)
- Timeout: 1000
- Data Size: 1000
- Data Type: DBLDBLDBL

The main display area shows a list of data points for the Amplitude.HIST property:

```
/TEST/PulseServer/#0 Amplitude.HIST @ 14:00:07.774
system stamp: 24028808, user stamp: 0
(0,0) [256.0, 1.560254364729496E9, 2.4028724E7]
(0,1) [256.0, 1.560254365733273E9, 2.4028725E7]
(0,2) [256.0, 1.56025436673811E9, 2.4028727E7]
(0,3) [256.0, 1.560254367738527E9, 2.4028729E7]
(0,4) [256.0, 1.560254368754302E9, 2.4028731E7]
(0,5) [256.0, 1.560254369748531E9, 2.4028733E7]
(0,6) [256.0, 1.560254370759413E9, 2.4028735E7]
(0,7) [256.0, 1.560254371755934E9, 2.4028737E7]
(0,8) [256.0, 1.560254372746215E9, 2.4028739E7]
(0,9) [256.0, 1.56025437374298E9, 2.4028741E7]
(0,10) [256.0, 1.560254375733963E9, 2.4028745E7]
(0,11) [256.0, 1.560254376746886E9, 2.4028747E7]
(0,12) [256.0, 1.560254378728064E9, 2.4028751E7]
(0,13) [256.0, 1.560254379720816E9, 2.4028753E7]
(0,14) [256.0, 1.560254380739204E9, 2.4028755E7]
```

Additional details from the interface:

- Write Access:
- Input Data Type: KEYVALUE (circled in red)
- starttime=10.6.2019 13:50:15
- stoptime=11.6.2019 13:50:23
- index=1
- sample=1
- startstamp=24027818
- stopstamp=24029818

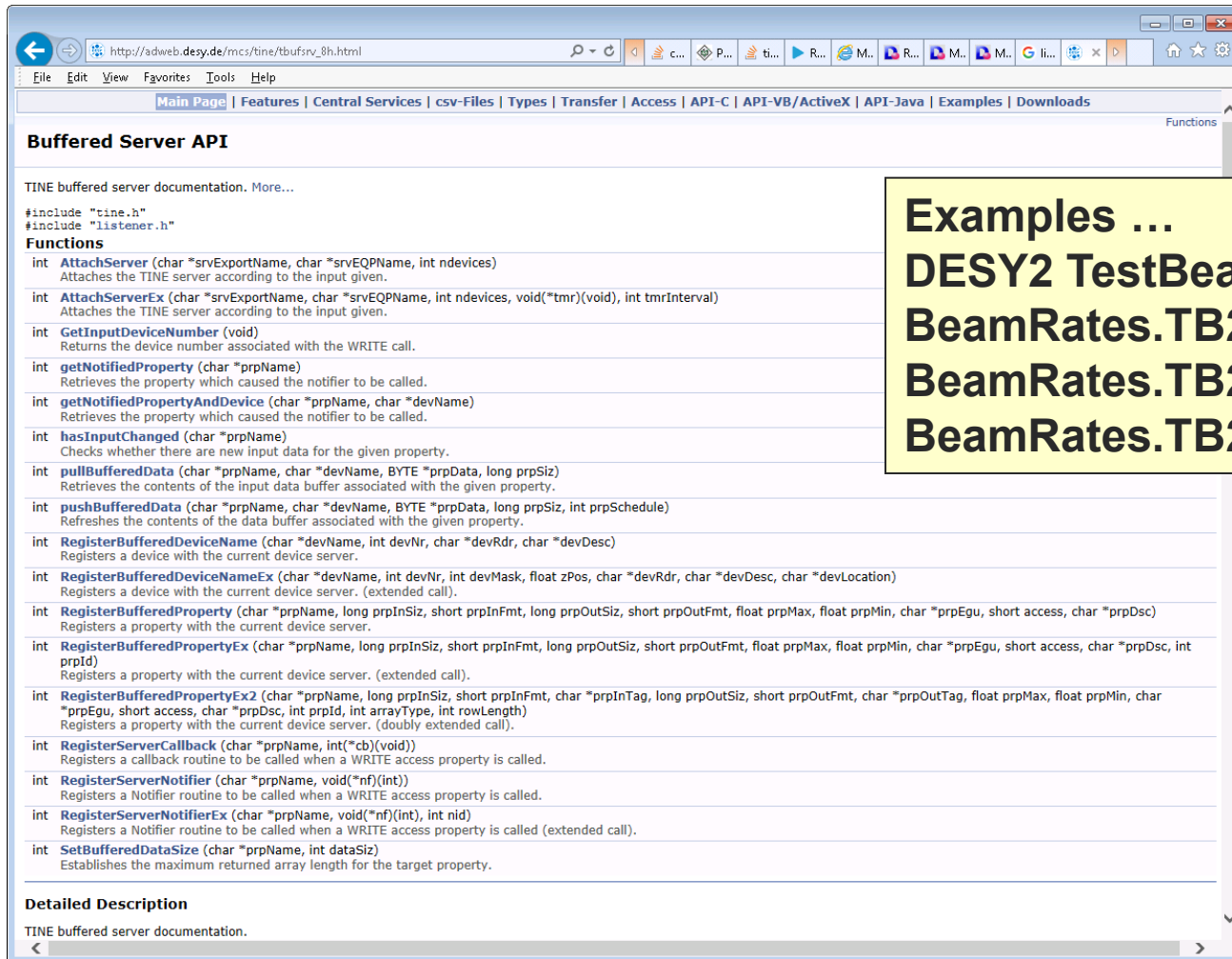
Settings: UDP, Timer | Suppress Query Properties

Last request: 14:00:09.326 (2 ms)

[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++



The screenshot shows a web browser window displaying the Buffered Server API documentation. The browser's address bar shows the URL `http://adweb.desy.de/mcs/tine/tbufsrv_8h.html`. The page title is "Buffered Server API". Below the title, there is a navigation menu with links for "Main Page", "Features", "Central Services", "csv-Files", "Types", "Transfer", "Access", "API-C", "API-VB/ActiveX", "API-Java", "Examples", and "Downloads". The main content area is titled "Buffered Server API" and contains the following text:

TINE buffered server documentation. More...

```
#include "tine.h"
#include "listener.h"
Functions
int AttachServer (char *srvExportName, char *srvEQPName, int ndevices)
Attaches the TINE server according to the input given.
int AttachServerEx (char *srvExportName, char *srvEQPName, int ndevices, void(*tmr)(void), int tmrInterval)
Attaches the TINE server according to the input given.
int GetInputDeviceNumber (void)
Returns the device number associated with the WRITE call.
int getNotifiedProperty (char *prpName)
Retrieves the property which caused the notifier to be called.
int getNotifiedPropertyAndDevice (char *prpName, char *devName)
Retrieves the property which caused the notifier to be called.
int hasInputChanged (char *prpName)
Checks whether there are new input data for the given property.
int pullBufferedData (char *prpName, char *devName, BYTE *prpData, long prpSiz)
Retrieves the contents of the input data buffer associated with the given property.
int pushBufferedData (char *prpName, char *devName, BYTE *prpData, long prpSiz, int prpSchedule)
Refreshes the contents of the data buffer associated with the given property.
int RegisterBufferedDeviceName (char *devName, int devNr, char *devRdr, char *devDesc)
Registers a device with the current device server.
int RegisterBufferedDeviceNameEx (char *devName, int devNr, int devMask, float zPos, char *devRdr, char *devDesc, char *devLocation)
Registers a device with the current device server. (extended call).
int RegisterBufferedProperty (char *prpName, long prpInSiz, short prpInFmt, long prpOutSiz, short prpOutFmt, float prpMax, float prpMin, char *prpEgu, short access, char *prpDsc)
Registers a property with the current device server.
int RegisterBufferedPropertyEx (char *prpName, long prpInSiz, short prpInFmt, long prpOutSiz, short prpOutFmt, float prpMax, float prpMin, char *prpEgu, short access, char *prpDsc, int prpId)
Registers a property with the current device server. (extended call).
int RegisterBufferedPropertyEx2 (char *prpName, long prpInSiz, short prpInFmt, char *prpInTag, long prpOutSiz, short prpOutFmt, char *prpOutTag, float prpMax, float prpMin, char *prpEgu, short access, char *prpDsc, int prpId, int arrayType, int rowLength)
Registers a property with the current device server. (doubly extended call).
int RegisterServerCallback (char *prpName, int(*cb)(void))
Registers a callback routine to be called when a WRITE access property is called.
int RegisterServerNotifier (char *prpName, void(*nf)(int))
Registers a Notifier routine to be called when a WRITE access property is called.
int RegisterServerNotifierEx (char *prpName, void(*nf)(int), int nid)
Registers a Notifier routine to be called when a WRITE access property is called (extended call).
int SetBufferedDataSize (char *prpName, int dataSiz)
Establishes the maximum returned array length for the target property.
```

Detailed Description

TINE buffered server documentation.

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

Simple LabView API for Windows

LabView allows the incorporation of the C or VisualBasic APIs (including ActiveX controls) in its application development environment. In this end, we provide several simple LabView VIs which are based on the TINE BufferServer API and which provide an easy-to-use interface from the LabView perspective.

Servers

It is strongly suggested that server information be registered via the local database files fecid.csv, exports.csv, and <EQM>-device.csv. Suffice it to say that registering server names, property names and information, and device names via API calls in LabView is the easiest way to do this.

ivTineSrvInit

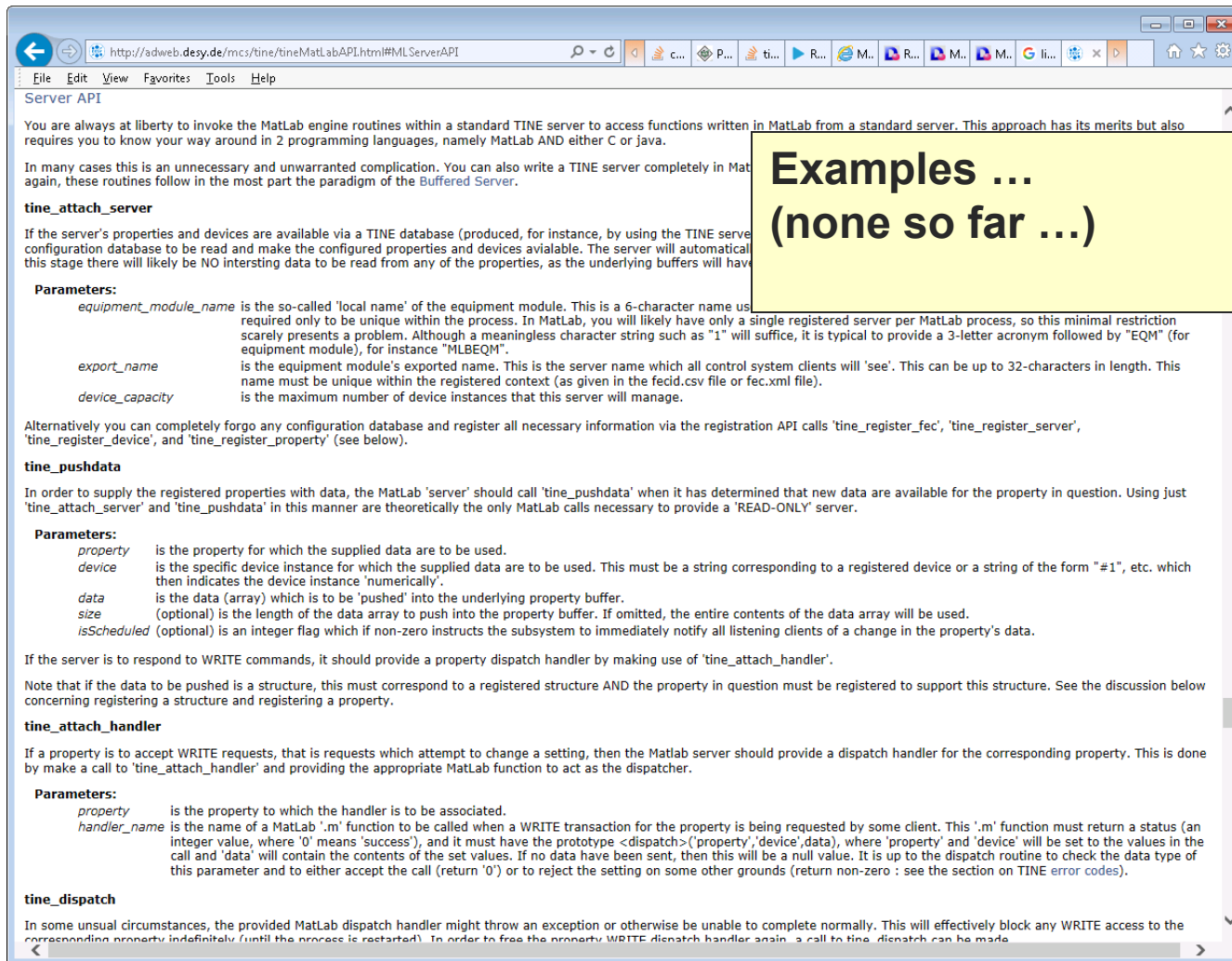
In LabView it is only necessary to 'attach' the server to the registered information from the database files. This should be done by the 'ivTineSrvInit' VI. It is the only VI you need to deal with if you have a 'read-only' server. In any event this 'principal' VI will service all read requests (regardless of sequence structure), passing only the desired 'Export Name' of the server, which is used to cross-check the information in the database files.

Parameters:
ExportName (String) is the Export Name of the device server. This must match an entry in the exports.csv file.

Returns:
0 if successful otherwise a TINE error code.

Example:
ivTineSrvInit.vi takes only one String input parameter, namely the Export Name of the device server to be managed by the underlying subsystem.

Buffered Server : MatLab



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 MatLab, 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 server configuration database to be read and make the configured properties and devices available. The server will automatically at this stage there will likely be NO interesting data to be read from any of the properties, as the underlying buffers will have

Parameters:

- equipment_module_name* is the so-called 'local name' of the equipment module. This is a 6-character name used 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 scarcely 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

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.

PyTine.attach_server

If the server's properties and devices are available via a TINE database (produced, for instance, by using the TINE configuration database to be read and make the configured properties and devices available. The server will automate this stage there will likely be NO interesting 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 for the server. It must be unique within the process. In Python, you will likely have only a single registered server per process, so a meaningless character string such as "1" will suffice, it is typical to provide a 3-letter acronym followed by a number.
- server* (string) is the equipment module's exported name. This is the server name which all control systems will use to identify the server. It must be 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 making 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):

```
#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