Servers for Dummies

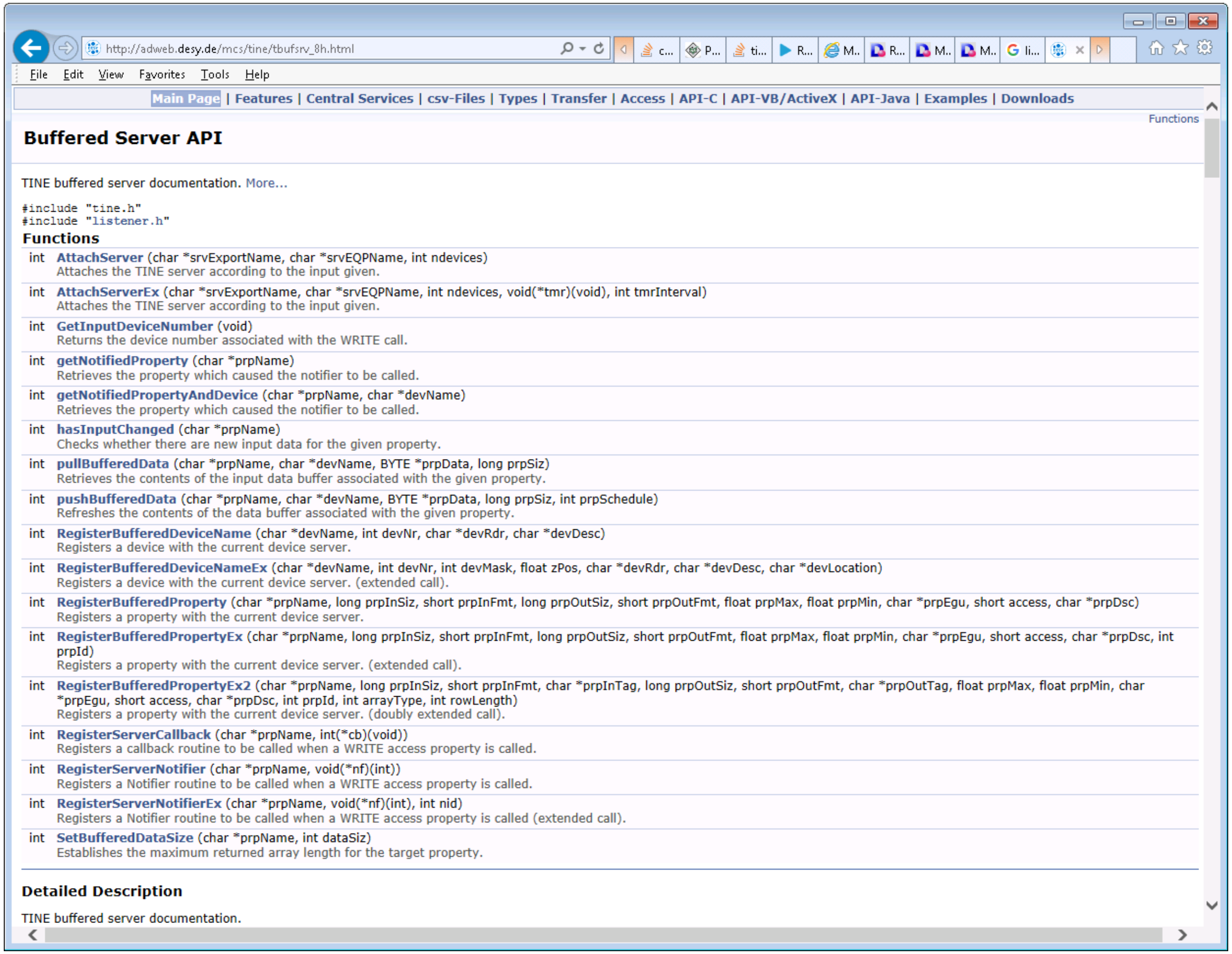
Using the ‘Buffered Server’

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++



http://adweb.desy.de/mcs/tine/tbufsrv_8h.html
File Edit View Favorites Tools Help
Main Page | Features | Central Services | csv-Files | Types | Transfer | Access | API-C | API-VB/ActiveX | API-Java | Examples | Downloads
Functions

Buffered Server API

TINE buffered server documentation. More...

```
#include "tine.h"
#include "listener.h"
```

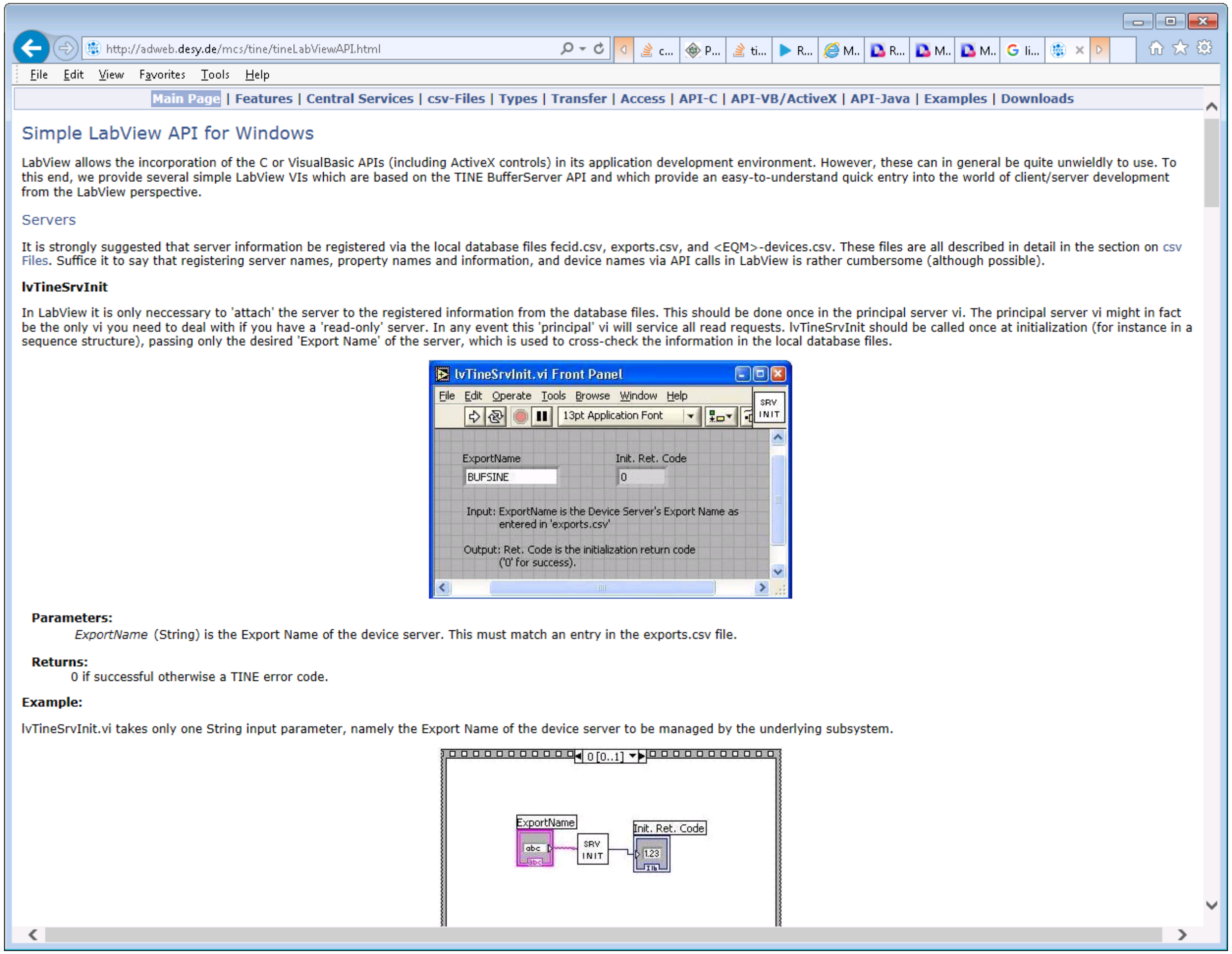
Functions

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.

Buffered Server : Labview



http://adweb.desy.de/mcs/tine/tineLabViewAPI.html
File Edit View Favorites Tools Help
Main Page | Features | Central Services | csv-Files | Types | Transfer | Access | API-C | API-VB/ActiveX | API-Java | Examples | Downloads

Simple LabView API for Windows

LabView allows the incorporation of the C or VisualBasic APIs (including ActiveX controls) in its application development environment. However, these can in general be quite unwieldly to use. To this end, we provide several simple LabView VIs which are based on the TINE BufferServer API and which provide an easy-to-understand quick entry into the world of client/server development 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>-devices.csv. These files are all described in detail in the section on csv Files. Suffice it to say that registering server names, property names and information, and device names via API calls in LabView is rather cumbersome (although possible).

lvTineSrvInit

In LabView it is only neccessary to 'attach' the server to the registered information from the database files. This should be done once in the principal server vi. The principal server vi might in fact be 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. lvTineSrvInit should be called once at initialization (for instance in a sequence structure), passing only the desired 'Export Name' of the server, which is used to cross-check the information in the local database files.

lvTineSrvInit.vi Front Panel
File Edit Operate Tools Browse Window Help
13pt Application Font
SRV INIT

ExportName: BUFSINE

Init. Ret. Code: 0

Input: ExportName is the Device Server's Export Name as entered in 'exports.csv'

Output: Ret. Code is the initialization return code ('0' for success).

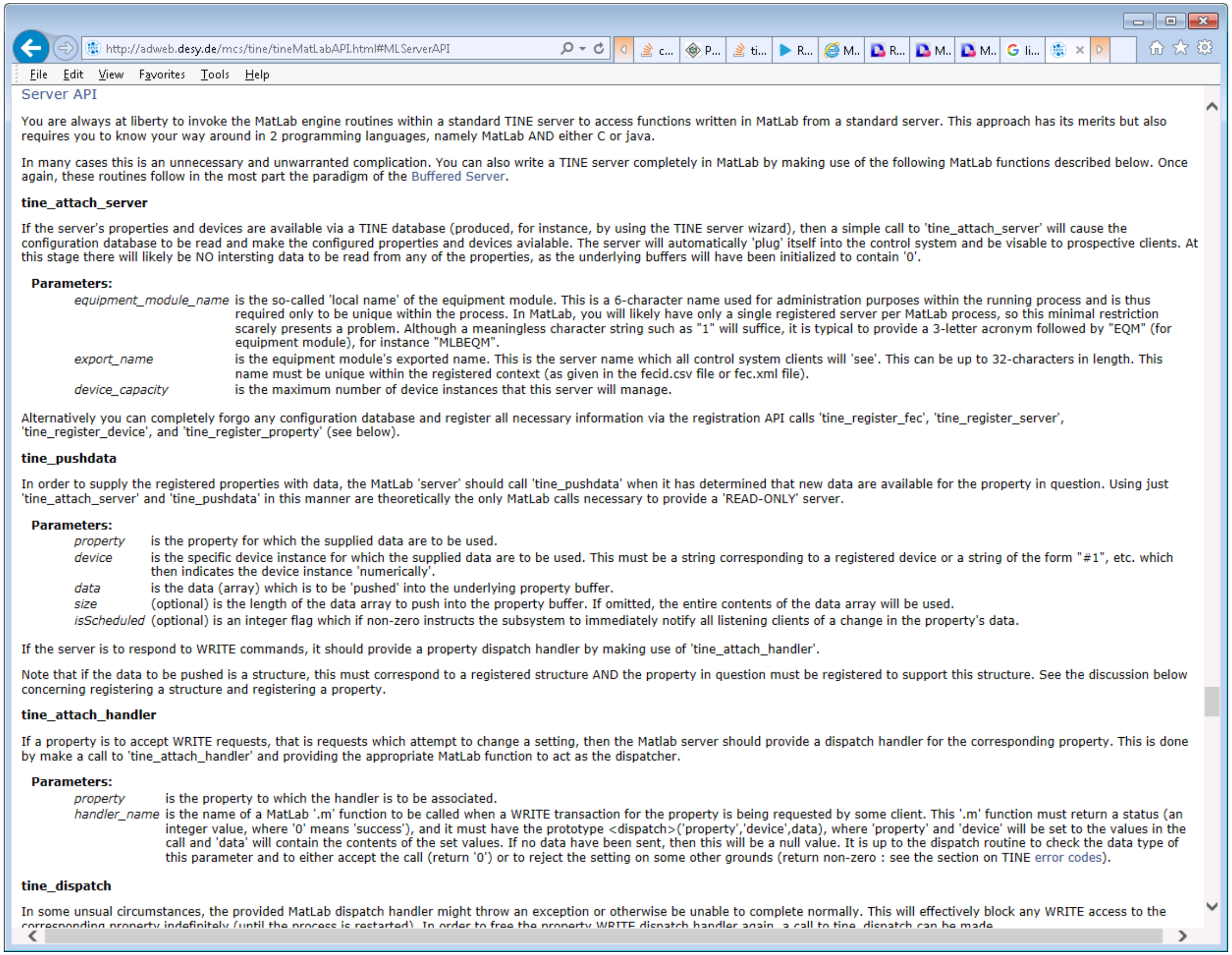
Parameters:
ExportName (String) is the Export Name of the device server. This must match an entry in the exports.csv file.

Returns:
0 if successful otherwise a TINE error code.

Example:

lvTineSrvInit.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 by making use of the following MatLab functions described below. Once 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 wizard), then a simple call to 'tine_attach_server' will cause the configuration database to be read and make the configured properties and devices avialable. The server will automatically 'plug' itself into the control system and be visable to prospective clients. At this stage there will likely be NO intersting data to be read from any of the properties, as the underlying buffers will have been initialized to contain '0'.

Parameters:

equipment_module_name is the so-called 'local name' of the equipment module. This is a 6-character name used for administration purposes within the running process and is thus 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:

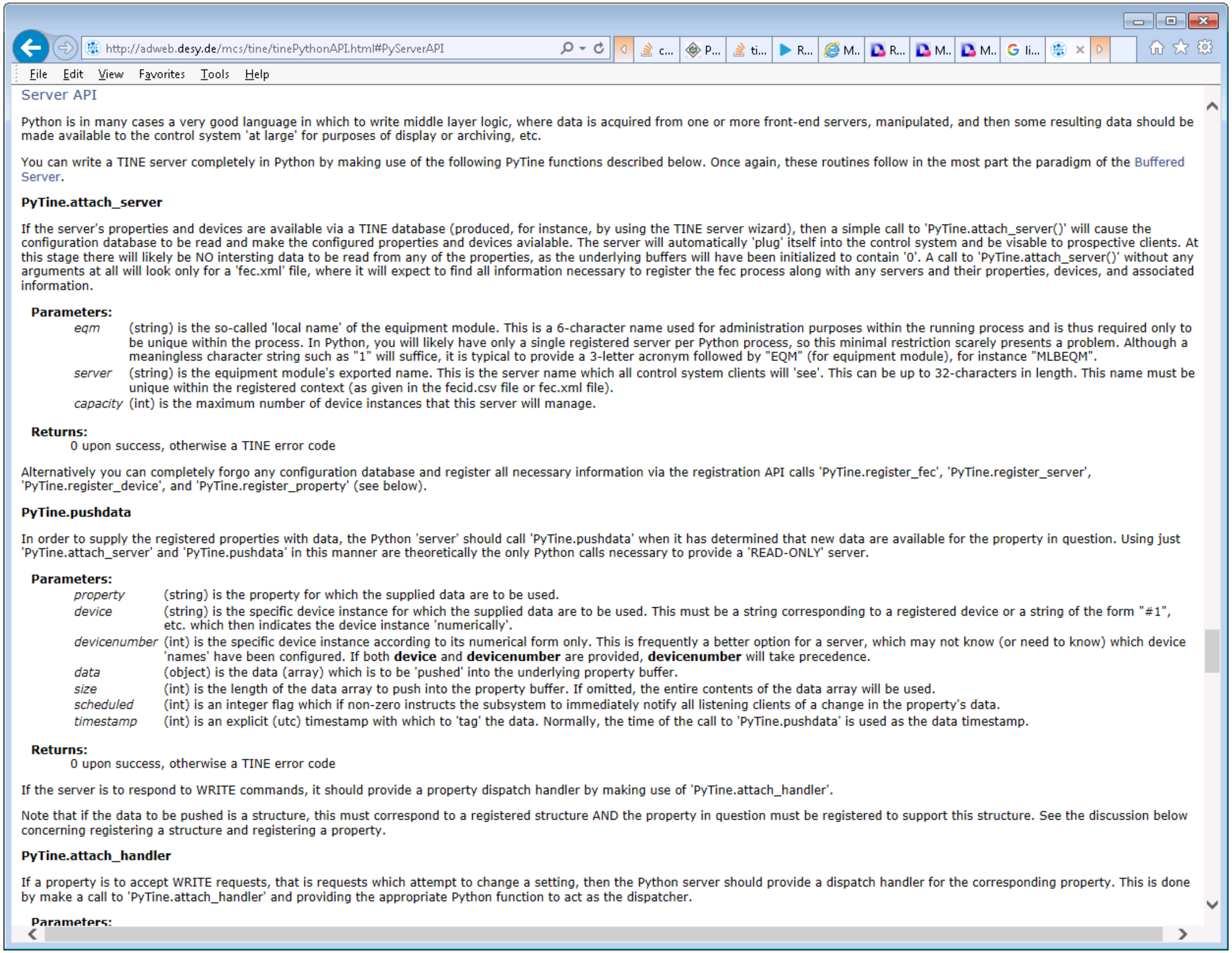
property is the property to which the handler is to be associated.

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 unsual 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.

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. Once again, these routines follow in the most part the paradigm of the Buffered Server.

PyTine.attach_server

If the server's properties and devices are available via a TINE database (produced, for instance, by using the TINE server wizard), then a simple call to 'PyTine.attach_server()' will cause the configuration database to be read and make the configured properties and devices avialable. The server will automatically 'plug' itself into the control system and be visable to prospective clients. At this stage there will likely be NO intersting data to be read from any of the properties, as the underlying buffers will have been initialized to contain '0'. A call to 'PyTine.attach_server()' without any arguments at all will look only for a 'fec.xml' file, where it will expect to find all information necessary to register the fec process along with any servers and their properties, devices, and associated information.

Parameters:

eqm (string) is the so-called 'local name' of the equipment module. This is a 6-character name used for administration purposes within the running process and is thus required only to be unique within the process. In Python, you will likely have only a single registered server per Python 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".

server (string) 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).

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:

Getting Started

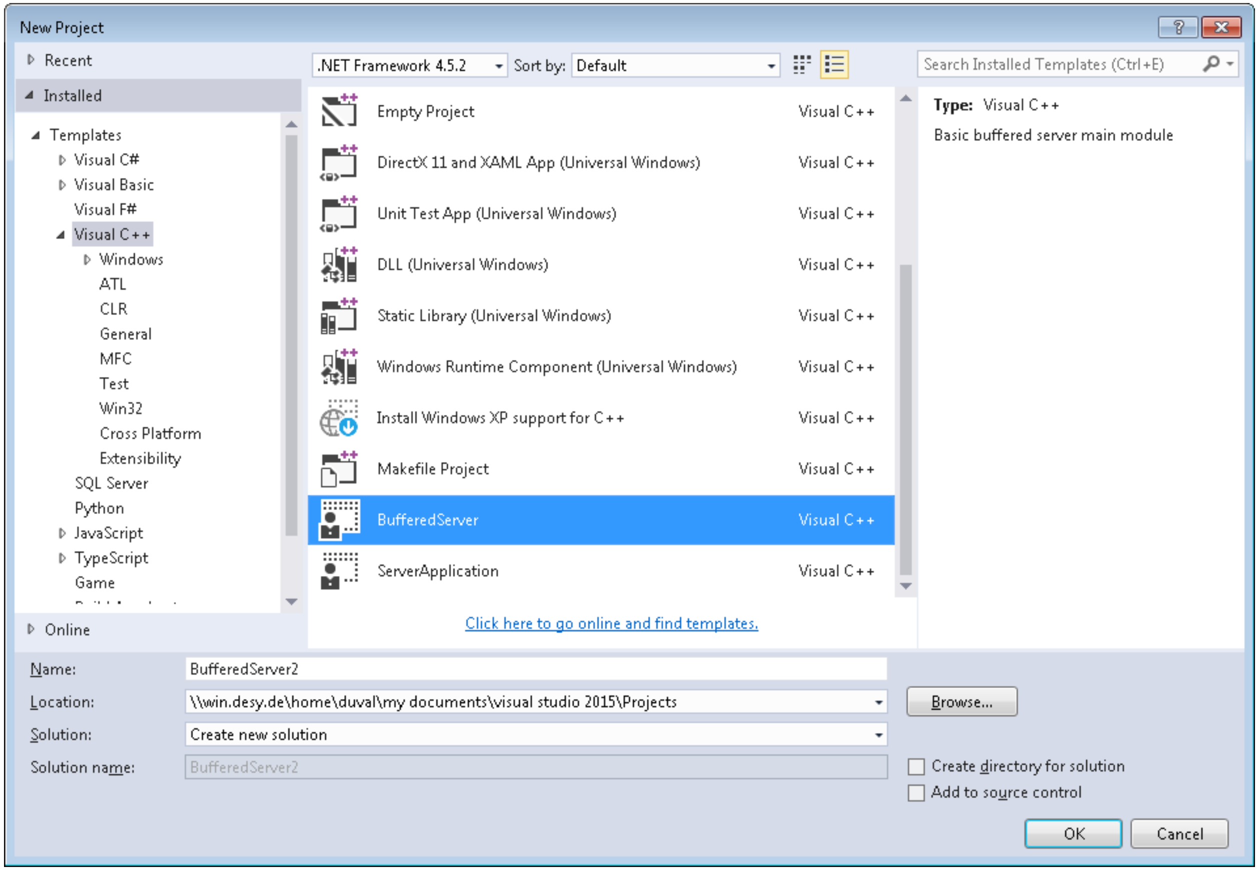
- Windows:
  - Get VS 2015 community edition for free
    - S:\services\Software\Visual Studio\Visual Studio 2015\Community-U3\
      - => vs_community.exe
  - Install the tine windows package
    - http://tine.desy.de -> downloads -> Windows Setup Installer -> Daily Build
    - http://adweb.desy.de/mcs/tine/TineArchive/setup.exe
      - Install windows
      - Install development libraries
      - Install java (so we can use the Java instant client)
      - Install Python
  - Make life comfortable with templates …
    - BufferedServer template (for development in C in Visual Studio)

In a ‘cmd’ box prompt:

```
subst L: C:\tine
subst Z: S:\services\ControlSystem\xApps\controls
```

Buffered Server in C:

Choose a new Visual C++ project and select the BufferedServer


New Project
Recent
Installed
Templates
Visual C#
Visual Basic
Visual F#
Visual C++
Windows
ATL
CLR
General
MFC
Test
Win32
Cross Platform
Extensibility
SQL Server
Python
JavaScript
TypeScript
Game
Online
.NET Framework 4.5.2
Sort by: Default
Search Installed Templates (Ctrl+E)
Empty Project Visual C++
DirectX 11 and XAML App (Universal Windows) Visual C++
Unit Test App (Universal Windows) Visual C++
DLL (Universal Windows) Visual C++
Static Library (Universal Windows) Visual C++
Windows Runtime Component (Universal Windows) Visual C++
Install Windows XP support for C++ Visual C++
Makefile Project Visual C++
BufferedServer Visual C++
ServerApplication Visual C++
Click here to go online and find templates.
Browse...
OK
Cancel

Type: Visual C++

Basic buffered server main module

Name: BufferedServer2

Location: \\win.desy.de\home\duval\my documents\visual studio 2015\Projects

Solution: Create new solution

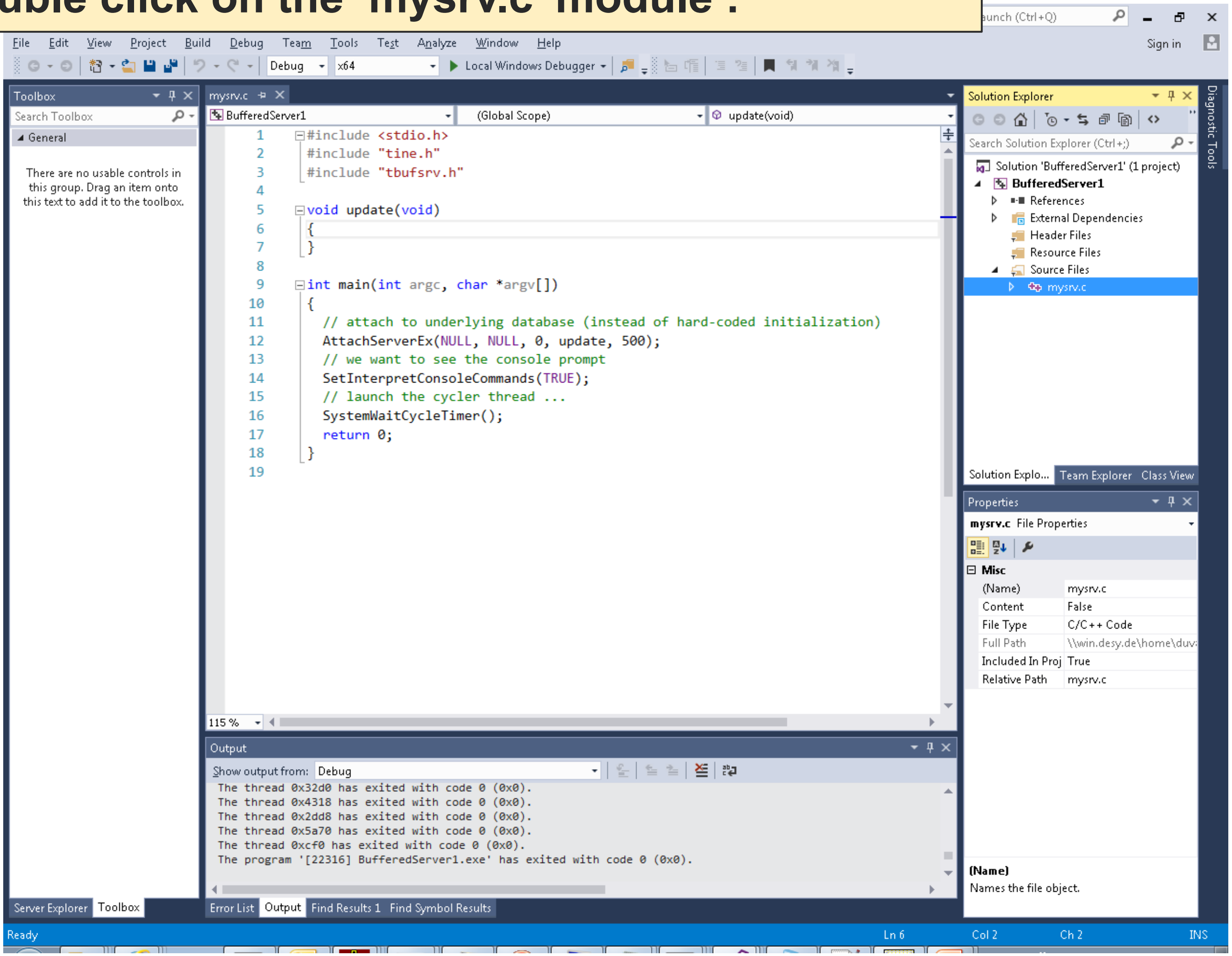
Solution name: BufferedServer2

☐ Create directory for solution

☐ Add to source control

Buffered Server in C:

Double click on the ‘mysrv.c’ module :



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

void update(void)
{
}

int main(int argc, char *argv[])
{
  // attach to underlying database (instead of hard-coded initialization)
  AttachServerEx(NULL, NULL, 0, update, 500);
  // we want to see the console prompt
  SetInterpretConsoleCommands(TRUE);
  // launch the cycler thread ...
  SystemWaitCycleTimer();
  return 0;
}
```

Output

Show output from: Debug

```
The thread 0x32d0 has exited with code 0 (0x0).
The thread 0x4318 has exited with code 0 (0x0).
The thread 0x2dd8 has exited with code 0 (0x0).
The thread 0x5a70 has exited with code 0 (0x0).
The thread 0xcf0 has exited with code 0 (0x0).
The program '[22316] BufferedServer1.exe' has exited with code 0 (0x0).
```

Properties

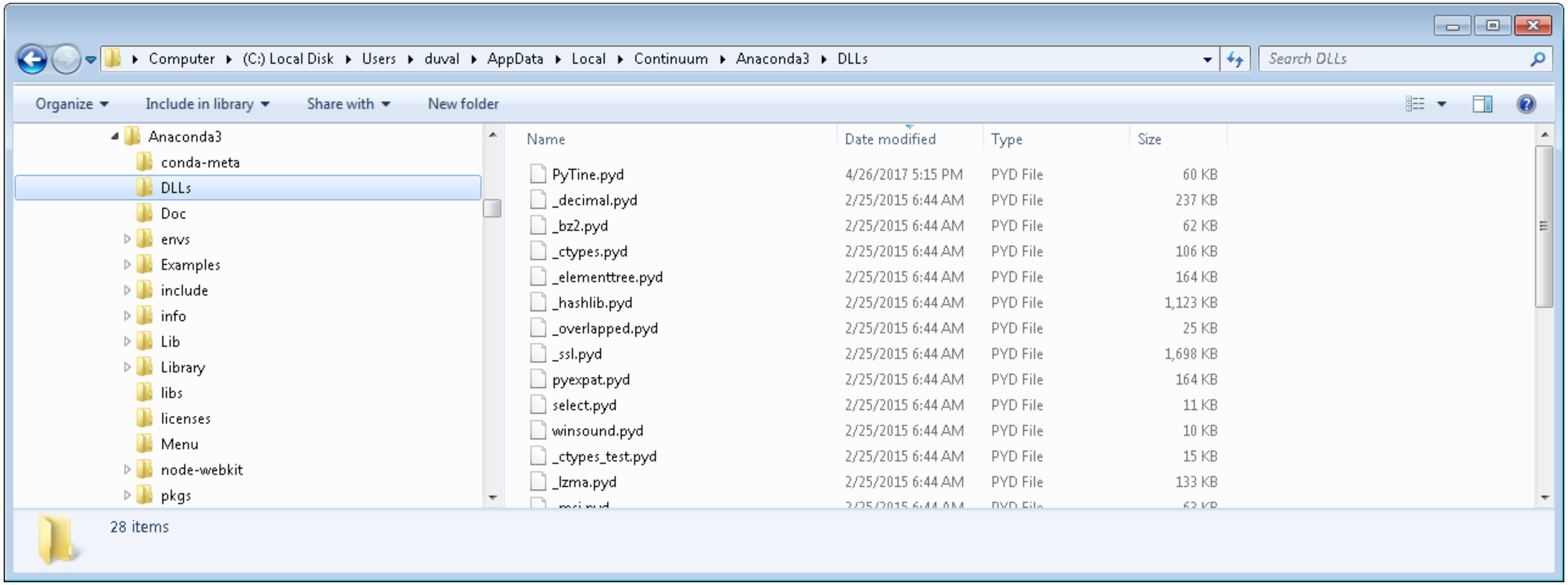
mysrv.c File Properties

Misc	
(Name)	mysrv.c
Content	False
File Type	C/C++ Code
Full Path	\\win.desy.de\home\duv
Included In Proj	True
Relative Path	mysrv.c

(Name)
Names the file object.

Buffered Server in Python

- Make sure PyTine.pyd is in the DLLs directory:



Computer ▸ (C:) Local Disk ▸ Users ▸ duval ▸ AppData ▸ Local ▸ Continuum ▸ Anaconda3 ▸ DLLs

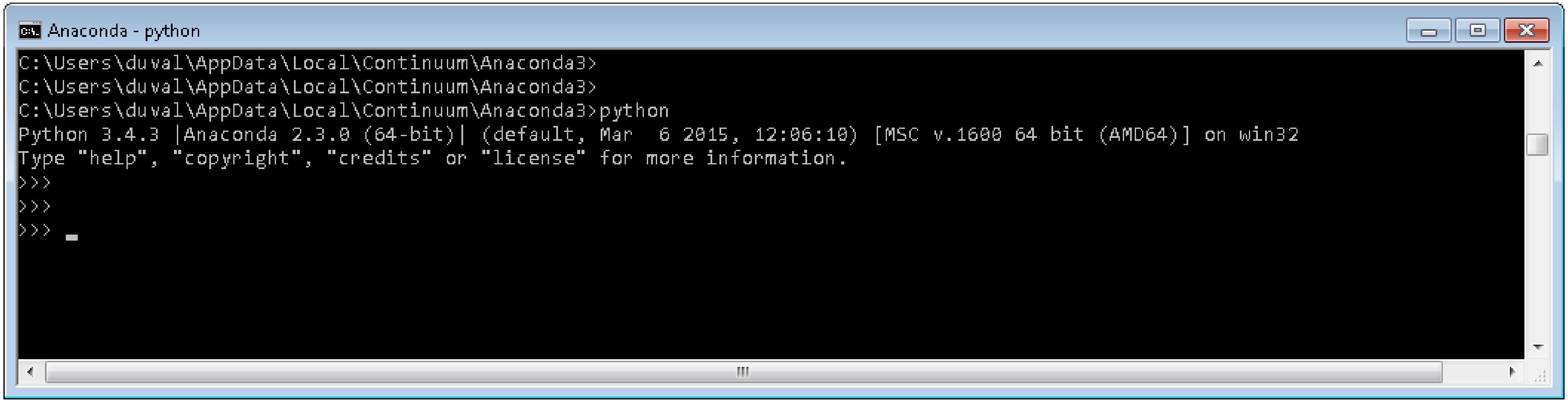
- Anaconda3
  - conda-meta
  - DLLs
  - Doc
  - envs
  - Examples
  - include
  - info
  - Lib
  - Library
  - libs
  - licenses
  - Menu
  - node-webkit
  - pkgs

Name	Date modified	Type	Size
PyTine.pyd	4/26/2017 5:15 PM	PYD File	60 KB
_decimal.pyd	2/25/2015 6:44 AM	PYD File	237 KB
_bz2.pyd	2/25/2015 6:44 AM	PYD File	62 KB
_ctypes.pyd	2/25/2015 6:44 AM	PYD File	106 KB
_elementtree.pyd	2/25/2015 6:44 AM	PYD File	164 KB
_hashlib.pyd	2/25/2015 6:44 AM	PYD File	1,123 KB
_overlapped.pyd	2/25/2015 6:44 AM	PYD File	25 KB
_ssl.pyd	2/25/2015 6:44 AM	PYD File	1,698 KB
pyexpat.pyd	2/25/2015 6:44 AM	PYD File	164 KB
select.pyd	2/25/2015 6:44 AM	PYD File	11 KB
winsound.pyd	2/25/2015 6:44 AM	PYD File	10 KB
_ctypes_test.pyd	2/25/2015 6:44 AM	PYD File	15 KB
_lzma.pyd	2/25/2015 6:44 AM	PYD File	133 KB
_msi.pyd	2/25/2015 6:44 AM	PYD File	63 KB

28 items

Buffered Server in Python

- Either open an Anaconda prompt or a command shell and type ‘python’:



```
C:\Users\duval\AppData\Local\Continuum\Anaconda3>
C:\Users\duval\AppData\Local\Continuum\Anaconda3>
C:\Users\duval\AppData\Local\Continuum\Anaconda3>python
Python 3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Mar  6 2015, 12:06:10) [MSC v.1600 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>>
>>> _
```

Linux

- Get the tar ball from http://adweb.desy.de/mcs/tine/TineArchive/tineLinux.tar.gz
  - Python: run the tine/python/setup.py after making sure that anaconda is installed
  - C : make use of the tine/server/BufferedServer/mysrv.mak make file.

Servers for Dummies

- Have a look at some other servers with the instant client (e.g.):
  - /XFEL/LLRF.CONTROLLER or any doocs server (device query precedence)
  - /XFEL/RadMonIp (property query precedence)
  - any CDI server (property query precedence)
  - ARCHIVER (property query precedence)
  - VAC.ION_PUMP (no precedence)

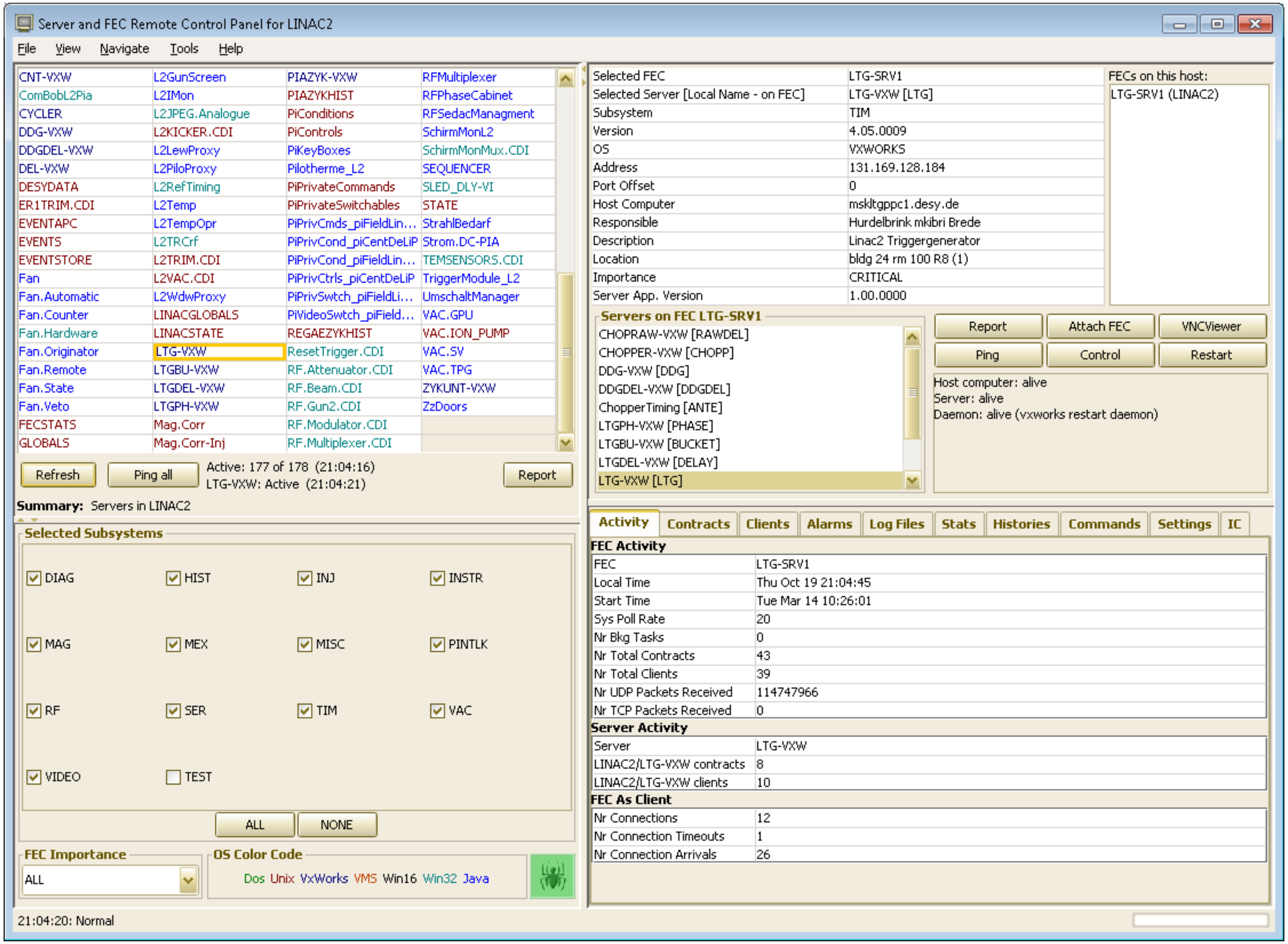
Servers for Dummies

- Multi-Channel Arrays
  - /TEST/SineServer/<device> Amplitude
- Scheduled Properties
  - /TEST/SineServer/<device>
    - Sine vs. Sine.SCHED
- Attributes
  - Read-only/Read-Write
- Commands
  - With/without input
- Read with input
  - e.g. Archive calls
  - e.g. Unit Server Echo
- Structures/Arrays

Servers for Dummies

- Our first server
  - A server belongs to a running process called a ‘Front-End Controller’ (FEC)
  - A FEC can (but usually doesn’t) contain more than 1 server
    - e.g. CAS, many VxWorks servers, several Magnet servers, etc. share a FEC with other servers.

Servers for Dummies



CNT-VXW
ComBobL2Pia
CYCLER
DDG-VXW
DDGDEL-VXW
DEL-VXW
DESYDATA
ER1TRIM.CDI
EVENTAPC
EVENTS
EVENTSTORE
Fan
Fan.Automatic
Fan.Counter
Fan.Hardware
Fan.Originator
Fan.Remote
Fan.State
Fan.Veto
FECSTATS
GLOBALS

L2GunScreen
L2IMon
L2JPEG.Analogue
L2KICKER.CDI
L2LewProxy
L2PiloProxy
L2RefTiming
L2Temp
L2TempOpr
L2TRCrf
L2TRIM.CDI
L2VAC.CDI
L2WdwProxy
LINACGLOBALS
LINACSTATE
LTG-VXW
LTGBU-VXW
LTGDEL-VXW
LTGPH-VXW
Mag.Corr
Mag.Corr-Inj

PIAZYK-VXW
PIAZYKHIST
PiConditions
PiControls
PiKeyBoxes
Pilotherme_L2
PiPrivateCommands
PiPrivateSwitchables
PiPrivCmds_piFieldLin...
PiPrivCond_piCentDeLiP
PiPrivCond_piFieldLin...
PiPrivCtrls_piCentDeLiP
PiPrivSwtch_piFieldLi...
PiVideoSwtch_piField...
REGAEZYKHIST
ResetTrigger.CDI
RF.Attenuator.CDI
RF.Beam.CDI
RF.Gun2.CDI
RF.Modulator.CDI
RF.Multiplexer.CDI

RFMultiplexer
RFPhaseCabinet
RFSedacManagment
SchirmMonL2
SchirmMonMux.CDI
SEQUENCER
SLED_DLY-VI
STATE
StrahlBedarf
Strom.DC-PIA
TEMSENSORS.CDI
TriggerModule_L2
UmschaltManager
VAC.GPU
VAC.ION_PUMP
VAC.SV
VAC.TPG
ZYKUNT-VXW
ZzDoors

Active: 177 of 178 (21:04:16)
LTG-VXW: Active (21:04:21)

Summary: Servers in LINAC2

Selected Subsystems

☑ DIAG ☑ HIST ☑ INJ ☑ INSTR
☑ MAG ☑ MEX ☑ MISC ☑ PINTLK
☑ RF ☑ SER ☑ TIM ☑ VAC
☑ VIDEO ☐ TEST

FEC Importance
ALL

OS Color Code
Dos Unix VxWorks VMS Win16 Win32 Java

21:04:20: Normal

Selected FEC	LTG-SRV1
Selected Server [Local Name - on FEC]	LTG-VXW [LTG]
Subsystem	TIM
Version	4.05.0009
OS	VXWORKS
Address	131.169.128.184
Port Offset	0
Host Computer	mskltgppc1.desy.de
Responsible	Hurdelbrink mkibri Brede
Description	Linac2 Triggergenerator
Location	bldg 24 rm 100 R8 (1)
Importance	CRITICAL
Server App. Version	1.00.0000

FECs on this host:
LTG-SRV1 (LINAC2)

Servers on FEC LTG-SRV1
CHOPRAW-VXW [RAWDEL]
CHOPPER-VXW [CHOPP]
DDG-VXW [DDG]
DDGDEL-VXW [DDGDEL]
ChopperTiming [ANTE]
LTGPH-VXW [PHASE]
LTGBU-VXW [BUCKET]
LTGDEL-VXW [DELAY]
LTG-VXW [LTG]

Host computer: alive
Server: alive
Daemon: alive (vxworks restart daemon)

FEC Activity

FEC	LTG-SRV1
Local Time	Thu Oct 19 21:04:45
Start Time	Tue Mar 14 10:26:01
Sys Poll Rate	20
Nr Bkg Tasks	0
Nr Total Contracts	43
Nr Total Clients	39
Nr UDP Packets Received	114747966
Nr TCP Packets Received	0

Server Activity

Server	LTG-VXW
LINAC2/LTG-VXW contracts	8
LINAC2/LTG-VXW clients	10

FEC As Client

Nr Connections	12
Nr Connection Timeouts	1
Nr Connection Arrivals	26

Servers for Dummies

- We’re going to use the buffered server API. 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!

Servers for Dummies

- Names
  - A FEC must have a system-wide unique name (16-characters)
    - This name is usually not visible to anyone
  - A host can have many FECs, but each must have a unique address (IP address + port)
    - The default doocs strategy: first 2 letters of server name + IPv4 address in Hex + RPC port
      - Funny names like “Bec0a8a381.52c” (good that no one sees this!)

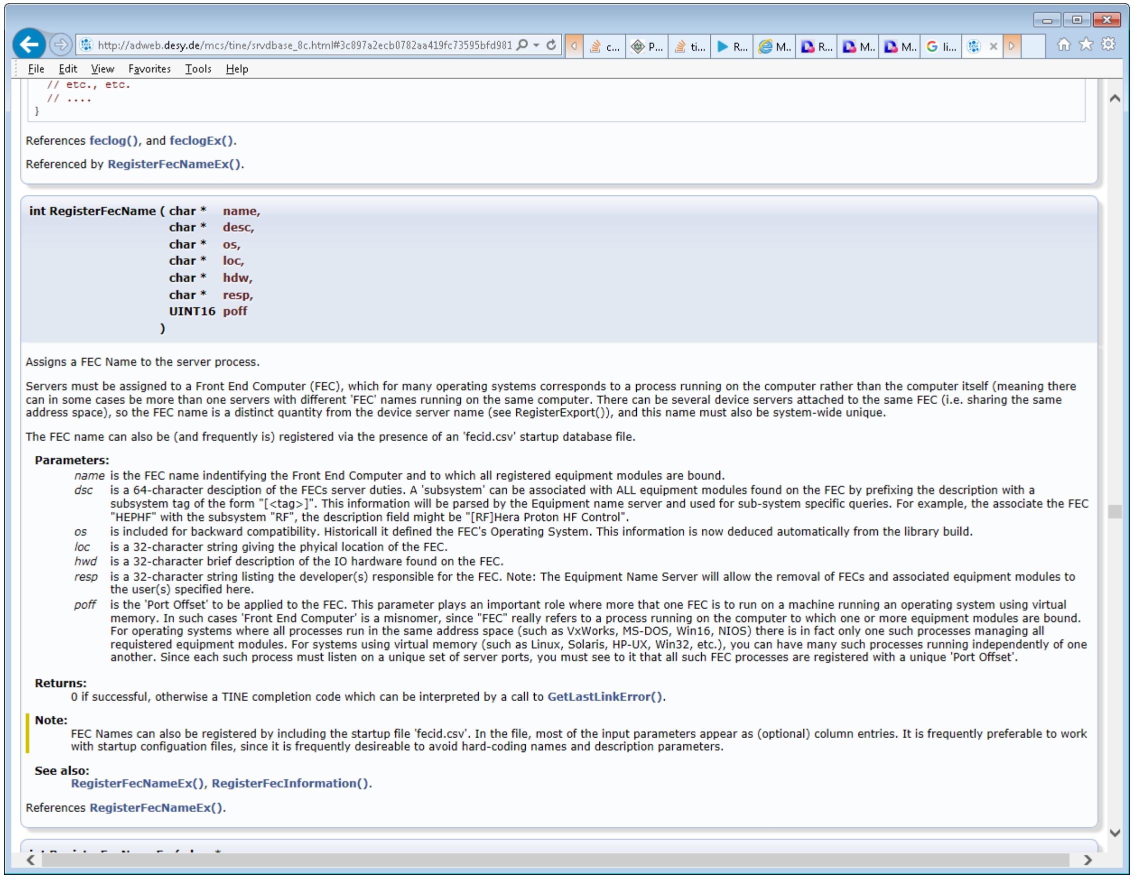
Servers for Dummies

Part of the name space !

- The combination of server name and context must be unique !
  - Can’t have two servers claiming to be /PETRA/ARCHIVER
- The exported server name and context are referenced internally at the process level via a ‘local equipment module name’ (6 characters).
  - No one sees this either.
  - Must be locally unique
- Buffered server: 1 server per FEC => automatically locally unique !

Servers for Dummies

- You can register names via API in code:


http://adweb.desy.de/mcs/tine/srvdbase_8c.html#3c897a2ecb0782aa419fc73595bfd981
c...
P...
ti...
R...
M..
R...
M..
M..
li...
File
Edit
View
Favorites
Tools
Help

```
  // etc., etc.
  // ....
}
```

References feclog(), and feclogEx().

Referenced by RegisterFecNameEx().

```
int RegisterFecName ( char *   name,
                      char *   desc,
                      char *   os,
                      char *   loc,
                      char *   hdw,
                      char *   resp,
                      UINT16   poff
                    )
```

Assigns a FEC Name to the server process.

Servers must be assigned to a Front End Computer (FEC), which for many operating systems corresponds to a process running on the computer rather than the computer itself (meaning there can in some cases be more than one servers with different 'FEC' names running on the same computer. There can be several device servers attached to the same FEC (i.e. sharing the same address space), so the FEC name is a distinct quantity from the device server name (see RegisterExport()), and this name must also be system-wide unique.

The FEC name can also be (and frequently is) registered via the presence of an 'fecid.csv' startup database file.

Parameters:

name is the FEC name indentifying the Front End Computer and to which all registered equipment modules are bound.

dsc is a 64-character desciption of the FECs server duties. A 'subsystem' can be associated with ALL equipment modules found on the FEC by prefixing the description with a subsystem tag of the form "[<tag>]". This information will be parsed by the Equipment name server and used for sub-system specific queries. For example, the associate the FEC "HEPHF" with the subsystem "RF", the description field might be "[RF]Hera Proton HF Control".

os is included for backward compatibility. Historicall it defined the FEC's Operating System. This information is now deduced automatically from the library build.

loc is a 32-character string giving the phyical location of the FEC.

hwd is a 32-character brief description of the IO hardware found on the FEC.

resp is a 32-character string listing the developer(s) responsible for the FEC. Note: The Equipment Name Server will allow the removal of FECs and associated equipment modules to the user(s) specified here.

poff is the 'Port Offset' to be applied to the FEC. This parameter plays an important role where more that one FEC is to run on a machine running an operating system using virtual memory. In such cases 'Front End Computer' is a misnomer, since "FEC" really refers to a process running on the computer to which one or more equipment modules are bound. For operating systems where all processes run in the same address space (such as VxWorks, MS-DOS, Win16, NIOS) there is in fact only one such processes managing all requistered equipment modules. For systems using virtual memory (such as Linux, Solaris, HP-UX, Win32, etc.), you can have many such processes running independently of one another. Since each such process must listen on a unique set of server ports, you must see to it that all such FEC processes are registered with a unique 'Port Offset'.

Returns:

0 if successful, otherwise a TINE completion code which can be interpreted by a call to GetLastLinkError().

Note:

FEC Names can also be registered by including the startup file 'fecid.csv'. In the file, most of the input parameters appear as (optional) column entries. It is frequently preferable to work with startup configuation files, since it is frequently desireable to avoid hard-coding names and description parameters.

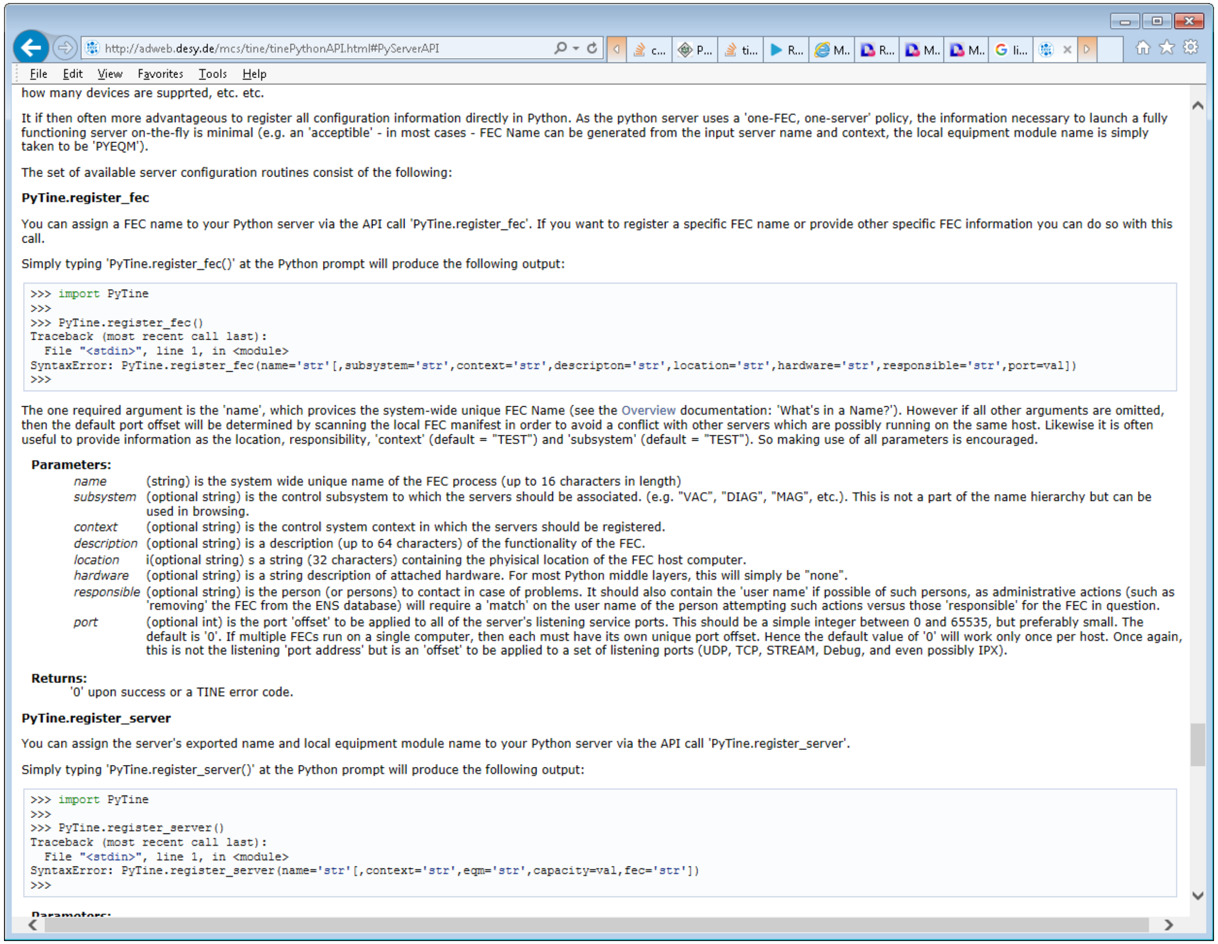
See also:

RegisterFecNameEx(), RegisterFecInformation().

References RegisterFecNameEx().

Servers for Dummies

- Python as well …


http://adweb.desy.de/mcs/tine/tinePythonAPI.html#PyServerAPI
File Edit View Favorites Tools Help

how many devices are supprted, etc. etc.

It if then often more advantageous to register all configuration information directly in Python. As the python server uses a 'one-FEC, one-server' policy, the information necessary to launch a fully functioning server on-the-fly is minimal (e.g. an 'acceptible' - in most cases - FEC Name can be generated from the input server name and context, the local equipment module name is simply taken to be 'PYEQM').

The set of available server configuration routines consist of the following:

PyTine.register_fec

You can assign a FEC name to your Python server via the API call 'PyTine.register_fec'. If you want to register a specific FEC name or provide other specific FEC information you can do so with this call.

Simply typing 'PyTine.register_fec()' at the Python prompt will produce the following output:

```
>>> import PyTine
>>>
>>> PyTine.register_fec()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: PyTine.register_fec(name='str'[,subsystem='str',context='str',descripton='str',location='str',hardware='str',responsible='str',port=val])
>>>
```

The one required argument is the 'name', which provices the system-wide unique FEC Name (see the Overview documentation: 'What's in a Name?'). However if all other arguments are omitted, then the default port offset will be determined by scanning the local FEC manifest in order to avoid a conflict with other servers which are possibly running on the same host. Likewise it is often useful to provide information as the location, responsibility, 'context' (default = "TEST") and 'subsystem' (default = "TEST"). So making use of all parameters is encouraged.

Parameters:

name (string) is the system wide unique name of the FEC process (up to 16 characters in length)

subsystem (optional string) is the control subsystem to which the servers should be associated. (e.g. "VAC", "DIAG", "MAG", etc.). This is not a part of the name hierarchy but can be used in browsing.

context (optional string) is the control system context in which the servers should be registered.

description (optional string) is a description (up to 64 characters) of the functionality of the FEC.

location i(optional string) s a string (32 characters) containing the phyisical location of the FEC host computer.

hardware (optional string) is a string description of attached hardware. For most Python middle layers, this will simply be "none".

responsible (optional string) is the person (or persons) to contact in case of problems. It should also contain the 'user name' if possible of such persons, as administrative actions (such as 'removing' the FEC from the ENS database) will require a 'match' on the user name of the person attempting such actions versus those 'responsible' for the FEC in question.

port (optional int) is the port 'offset' to be applied to all of the server's listening service ports. This should be a simple integer between 0 and 65535, but preferably small. The default is '0'. If multiple FECs run on a single computer, then each must have its own unique port offset. Hence the default value of '0' will work only once per host. Once again, this is not the listening 'port address' but is an 'offset' to be applied to a set of listening ports (UDP, TCP, STREAM, Debug, and even possibly IPX).

Returns:

'0' upon success or a TINE error code.

PyTine.register_server

You can assign the server's exported name and local equipment module name to your Python server via the API call 'PyTine.register_server'.

Simply typing 'PyTine.register_server()' at the Python prompt will produce the following output:

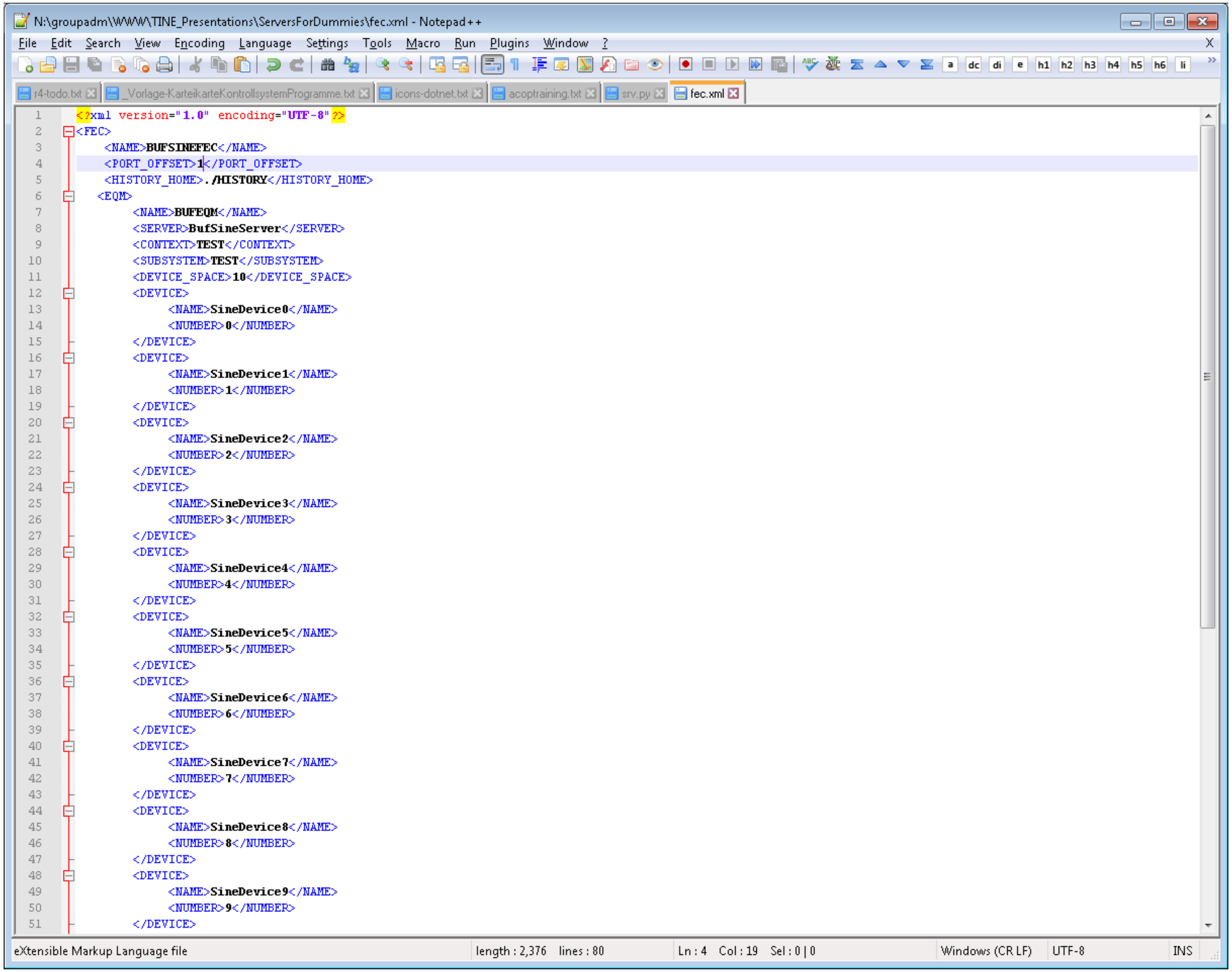
```
>>> import PyTine
>>>
>>> PyTine.register_server()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: PyTine.register_server(name='str'[,context='str',eqm='str',capacity=val,fec='str'])
>>>
```

Parameters:

Servers for Dummies

- But let’s make life easy with configuration files !
- Two ways to go …
  - fec.xml contains all configuration information for a FEC in a single file
  - fecid.csv + associated .csv Files contain the configuration information
- Suggestion: go with the .csv Files …

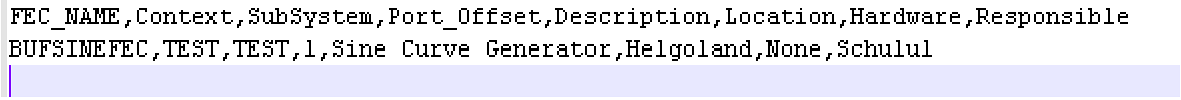
fec.xml


N:\groupadm\WWW\TINE_Presentations\ServersForDummies\fec.xml - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
a dc di e h1 h2 h3 h4 h5 h6 li
r4-todo.txt
_Vorlage-KarteikarteKontrollsystemProgramme.txt
icons-dotnet.txt
acoptraining.txt
srv.py
fec.xml
eXtensible Markup Language file
length : 2,376 lines : 80
Ln : 4 Col : 19 Sel : 0 | 0
Windows (CR LF)
UTF-8
INS

```xml
<?xml version="1.0" encoding="UTF-8"?>
<FEC>
    <NAME>BUFSINEFEC</NAME>
    <PORT_OFFSET>1</PORT_OFFSET>
    <HISTORY_HOME>./HISTORY</HISTORY_HOME>
   <EQM>
        <NAME>BUFEQM</NAME>
        <SERVER>BufSineServer</SERVER>
        <CONTEXT>TEST</CONTEXT>
        <SUBSYSTEM>TEST</SUBSYSTEM>
        <DEVICE_SPACE>10</DEVICE_SPACE>
        <DEVICE>
             <NAME>SineDevice0</NAME>
             <NUMBER>0</NUMBER>
        </DEVICE>
        <DEVICE>
             <NAME>SineDevice1</NAME>
             <NUMBER>1</NUMBER>
        </DEVICE>
        <DEVICE>
             <NAME>SineDevice2</NAME>
             <NUMBER>2</NUMBER>
        </DEVICE>
        <DEVICE>
             <NAME>SineDevice3</NAME>
             <NUMBER>3</NUMBER>
        </DEVICE>
        <DEVICE>
             <NAME>SineDevice4</NAME>
             <NUMBER>4</NUMBER>
        </DEVICE>
        <DEVICE>
             <NAME>SineDevice5</NAME>
             <NUMBER>5</NUMBER>
        </DEVICE>
        <DEVICE>
             <NAME>SineDevice6</NAME>
             <NUMBER>6</NUMBER>
        </DEVICE>
        <DEVICE>
             <NAME>SineDevice7</NAME>
             <NUMBER>7</NUMBER>
        </DEVICE>
        <DEVICE>
             <NAME>SineDevice8</NAME>
             <NUMBER>8</NUMBER>
        </DEVICE>
        <DEVICE>
             <NAME>SineDevice9</NAME>
             <NUMBER>9</NUMBER>
        </DEVICE>
```

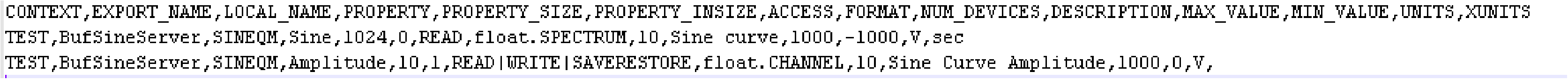
.csv Files

fecid.csv



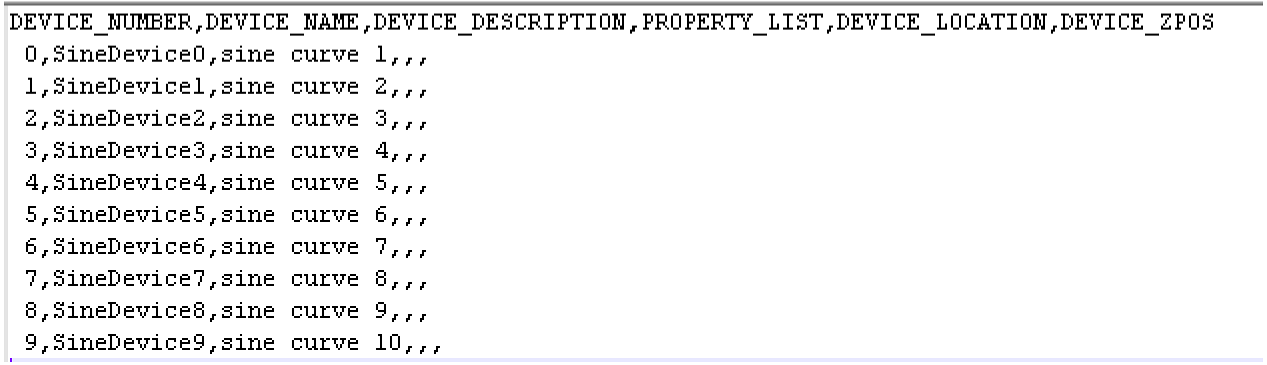
```
FEC_NAME,Context,SubSystem,Port_Offset,Description,Location,Hardware,Responsible
BUFSINEFEC,TEST,TEST,1,Sine Curve Generator,Helgoland,None,Schulul
```

exports.csv



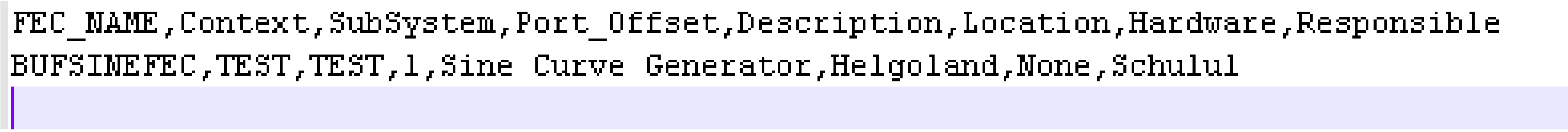
```
CONTEXT,EXPORT_NAME,LOCAL_NAME,PROPERTY,PROPERTY_SIZE,PROPERTY_INSIZE,ACCESS,FORMAT,NUM_DEVICES,DESCRIPTION,MAX_VALUE,MIN_VALUE,UNITS,XUNITS
TEST,BufSineServer,SINEQM,Sine,1024,0,READ,float.SPECTRUM,10,Sine curve,1000,-1000,V,sec
TEST,BufSineServer,SINEQM,Amplitude,10,1,READ|WRITE|SAVERESTORE,float.CHANNEL,10,Sine Curve Amplitude,1000,0,V,
```

devices.csv



```
DEVICE_NUMBER,DEVICE_NAME,DEVICE_DESCRIPTION,PROPERTY_LIST,DEVICE_LOCATION,DEVICE_ZPOS
 0,SineDevice0,sine curve 1,,,
 1,SineDevice1,sine curve 2,,,
 2,SineDevice2,sine curve 3,,,
 3,SineDevice3,sine curve 4,,,
 4,SineDevice4,sine curve 5,,,
 5,SineDevice5,sine curve 6,,,
 6,SineDevice6,sine curve 7,,,
 7,SineDevice7,sine curve 8,,,
 8,SineDevice8,sine curve 9,,,
 9,SineDevice9,sine curve 10,,,
```

fecid.csv



```
FEC_NAME,Context,SubSystem,Port_Offset,Description,Location,Hardware,Responsible
BUFSINEFEC,TEST,TEST,1,Sine Curve Generator,Helgoland,None,Schulul
```

Unique Name ! So add your station number to the FEC_NAME :

BUFSINEFEC1, BUFSINEFEC2, etc.

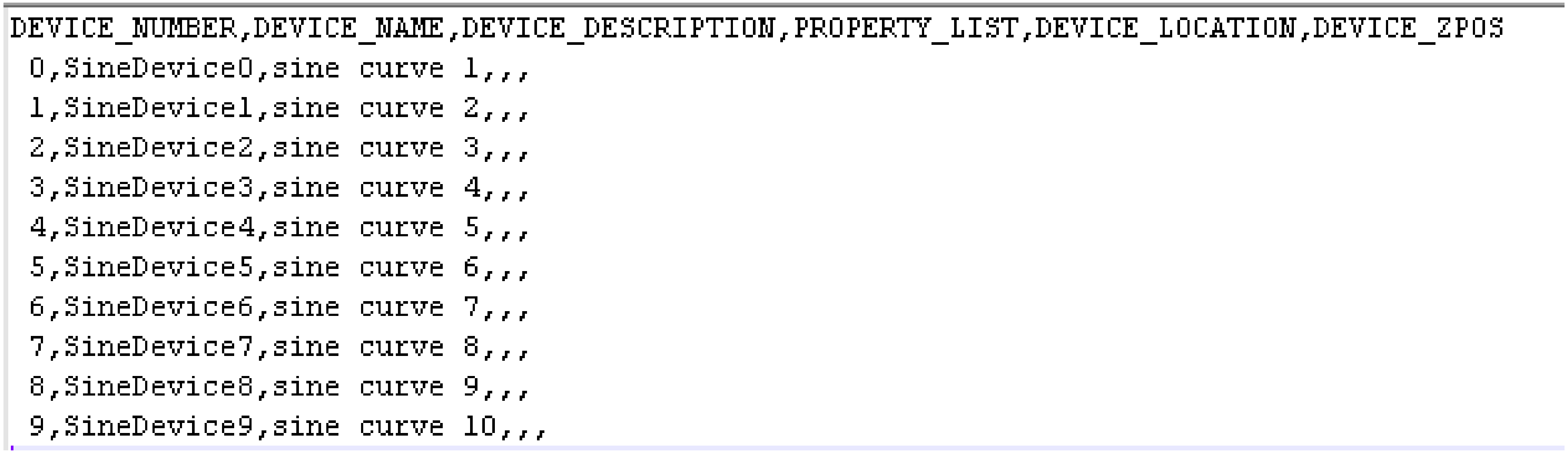
exports.csv

```
CONTEXT, EXPORT_NAME, LOCAL_NAME, PROPERTY, PROPERTY_SIZE,
PROPERTY_INSIZE, ACCESS, FORMAT, NUM_DEVICES, DESCRIPTION,
MAX_VALUE, MIN_VALUE, UNITS, XUNITS

TEST, BufSineServer, SINEQM, Sine,1024, 0, READ, float.SPECTRUM, 10,
Sine curve, 1000, -1000, V, sec

TEST, BufSineServer, SINEQM, Amplitude, 10, 1, READ|WRITE,
float.CHANNEL, 10, Sine Curve Amplitude,1000, 0, V,
```

Unique Server Name ! So add your station number to the EXPORT_NAME:

BUFSineServer1, BUFSineServer2, etc.

devices.csv



```
DEVICE_NUMBER,DEVICE_NAME,DEVICE_DESCRIPTION,PROPERTY_LIST,DEVICE_LOCATION,DEVICE_ZPOS
 0,SineDevice0,sine curve 1,,,
 1,SineDevice1,sine curve 2,,,
 2,SineDevice2,sine curve 3,,,
 3,SineDevice3,sine curve 4,,,
 4,SineDevice4,sine curve 5,,,
 5,SineDevice5,sine curve 6,,,
 6,SineDevice6,sine curve 7,,,
 7,SineDevice7,sine curve 8,,,
 8,SineDevice8,sine curve 9,,,
 9,SineDevice9,sine curve 10,,,
```

Plug-and-Play

- Automatic registration in tine ENS
  - Subsystems
    - Not part of name-space
    - Useful for browsing
    - Decorated contexts will strip off the subsystem
      - e.g. context = PETRA.VAC -> context = PETRA + subsystem = VAC
      - Allowed decorations: .TEST, .SIM, .EXT

Stock and Meta Properties

- All server support a set of ‘Stock’ properties
  - e.g. “PROPERTIES”, “DEVICES”, etc.
- All registered properties support a set of ‘meta’ properties
  - e.g. P.HIST, P.EGU, P.NAM, P.MAX

Exercises

- Local histories
  - ‘HIST’ flag
  - Or make use of history.csv
- Save/Restore
  - ‘SAVERESTORE’ flag
- Scheduling
  - Pass non-zero value in ‘scheduled’ argument in push_data
- Coercion
  - Forcing multicast : ‘NETWORK’
  - Forcing data-length/data format : ‘FORCEOUTPUT’
  - Forcing polling intervals
    - API: SetMinimumAllowedPollingInterval(value)
    - Or environment variable: FEC_POLLRATE
  - Flagging as static : ‘STATIC’