FESA client implementation

What is the basic FESA-3 plc-class generation

SILECS-1 provides a service to generate a basic FESA-3 class from an SILECS design document. In many applications, the FESA-3 class reflects exactly the PLC interface, and properties/fields can be mapped to the related blocks/registers definition. Purpose of this service is to generate automatically such a FESA-3 design document and the generic code providing the data transfer between FESA-3 fields and PLC registers. In particular, it avoids tedious and error prone editing within the server and realtime actions to synchronise the Controller (PLC, PXI, RabbitCore, ..) and the FESA-3 application. The C++ FESA-3 wrapper also offers very simple methods to setup and close the Controller connection.

Principles (current revision)

- One SILECS class design generates one FESA class
- The user is responsible of configuring the FESA DeviceInstance and DeployUnit documents of his FESA class
- One SILECS register's block corresponds to one FESA property (see table below for details)
- One SILECS register corresponds to one FESA field (see table below for details)

IEPLC block	Property	Server action(1)	Realtime action(1)	Triggering	IEPLC register	Field persistency	FESA field
READ-ONLY	acquisition	default (get)	custom (recv)	timer	None (default) Master	False False	acquisition (consistent) acquisition (not consistent)(2)
WRITE-ONLY	setting	custom (send) default (get)		RDA (on-demand)	Slave (default) None	True False	setting (consistent)
READ-WRITE	setting	custom (send) custom (recv)		RDA (on-demand)	Master (default) Slave None	True(3) True False	setting(4) (not consistent)

(1): Realtime and server actions must be implemented by the user by calling the appropriate methods of the generated wrapper
(2): A master register never changes its value, thus consistency is not required
(3): It's not forbidden to overwrite a MASTER register inside a RW block (but initial value should be provided by the controller)
(4): See "Principles" last point

Basic FESA-3 design generation uses the following convention:

- server action for asynchronous settings (write-only and read-write blocks)
- real time action for periodic acquisition (read-only blocks)

The user can customise his design as needed but modification will be lost in case of regeneration. It's strongly recommended to not modify the C++ generated code! No support will be done in that case.

As much as possible, the design generation uses the default parameters with only one exception: while waiting for the next FESA release, read-write registers cannot use the data consistency mechanism (setting a setting field inside a get-server-action is not possible yet). Nevertheless this should have no impact on the proposed design and scheduling

Synchronisation issue at startup

In the generated code the specificInit method calls methods to initialize only:

- SLAVE registers belonging to WRITE-ONLY and READ-WRITE blocks
- MASTER registers belonging to READ-ONLY and READ-WRITE blocks

In the other cases (synchro mode is NONE), no initialization method is supplied.

In case multiplexing is required, users should change the SILECS registers' synchronisation mode to 'NONE' and set the multiplexing flag in the corresponding FESA fields.

Generated C++ wrapper

- General methods to setup the controller's connection (should be called in the FESA specific-init) and release SILECS resources if needed.

```
void setup (const ServiceLocator* serviceLocator);
void cleanup();
```

- From a given SILECS block object, the following methods are used to transfer all the FESA fields values to the related SILECS registers and optionally to synchronise the PLC (send/receive call). Can be done for all devices of the clusters, for all devices of a given PLC, for a device collection or for a single device.

Proposal consists to call the get methods from the rt-actions and set methods from the server actions.

```
void getAllDevices (const ServiceLocator* serviceLocator, const bool recvNow,
MultiplexingContext* pContext);
void getPLCDevices (Silecs::PLC* pPLC, const ServiceLocator* serviceLocator, const
bool recvNow, MultiplexingContext* pContext);
void getSomeDevices(std::vector deviceCol, const bool recvNow, MultiplexingContext*
pContext);
void getOneDevice  (Device* pDevice, const bool recvNow, MultiplexingContext*
pContext);
void setOneDevice  (Device* pDevice, name##Property_DataType& data, const bool
sendNow, MultiplexingContext* pContext);
```