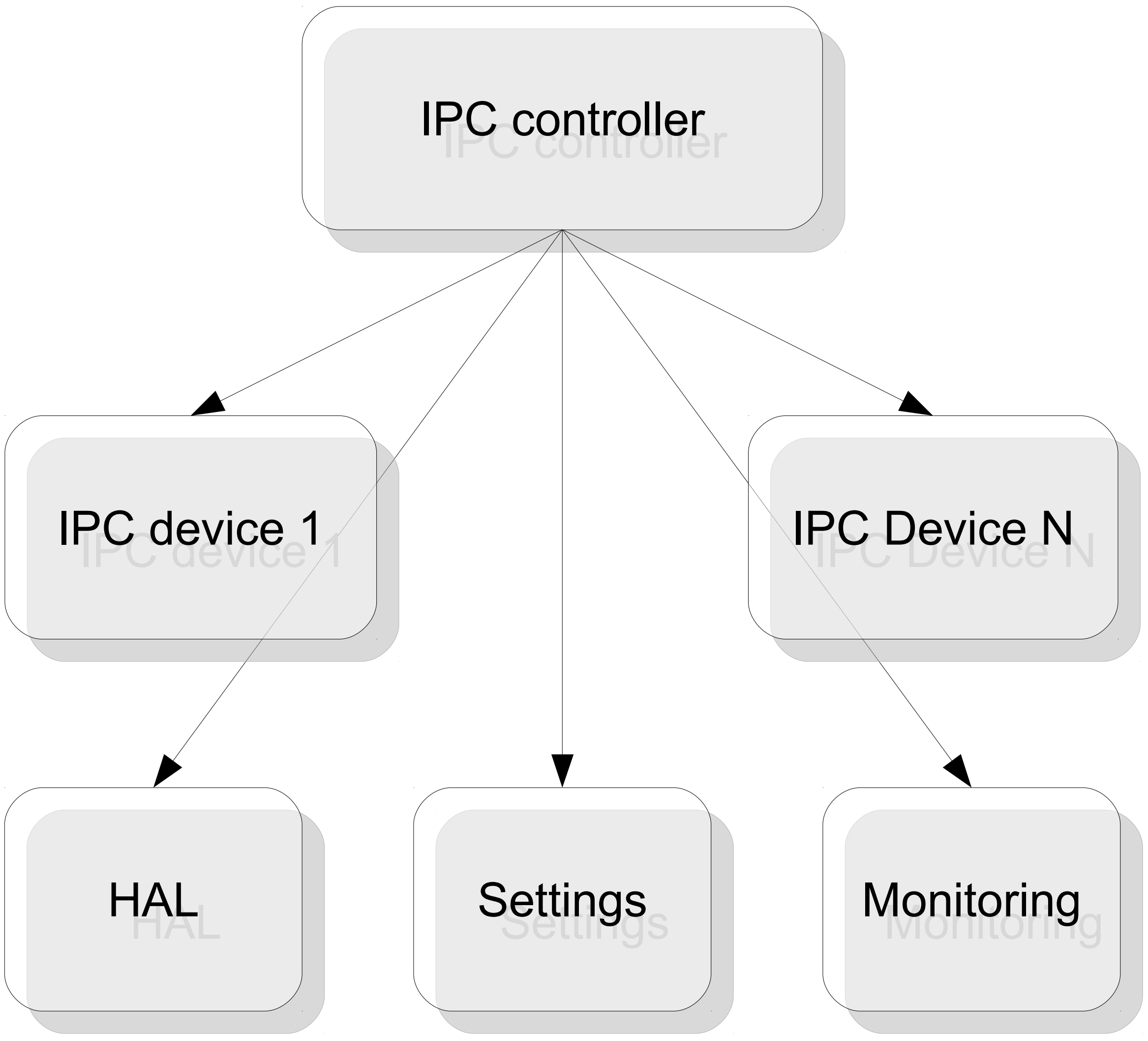
TS1 IPC soft PLC programming

Issued / changed	22.04.2013	Released / checked	22.04.2013	Signed	MC - Number	Page / Page (s)		
Document - Number						1	/	40

Content

1 Introduction

2 Ipc Controller



IPC controller is responsible for:

- creating and connecting hardware objects – IO, readers, cameras, printers, teach pendant...
- creating device tasks and controlling cell state – Start, Stop, Pause
- creating and loading settings objects from files
- monitoring general IO (service, emergency, doors), reconnecting hardware, if connection lost
- disconnecting and disposing hardware objects if application is being closed

- other custom duties that should be active also while machine is stopped.

2.1 Creating controller

The controller (Controller.cs) is inherited from IpcController base class (see Ipc.dll documentation). Controller is created as singleton object, meaning only one instance of that object can exist. This is achieved by hiding class constructor. Object is accessible only through read-only static Instance property. First one to access it, will also create instance internally:

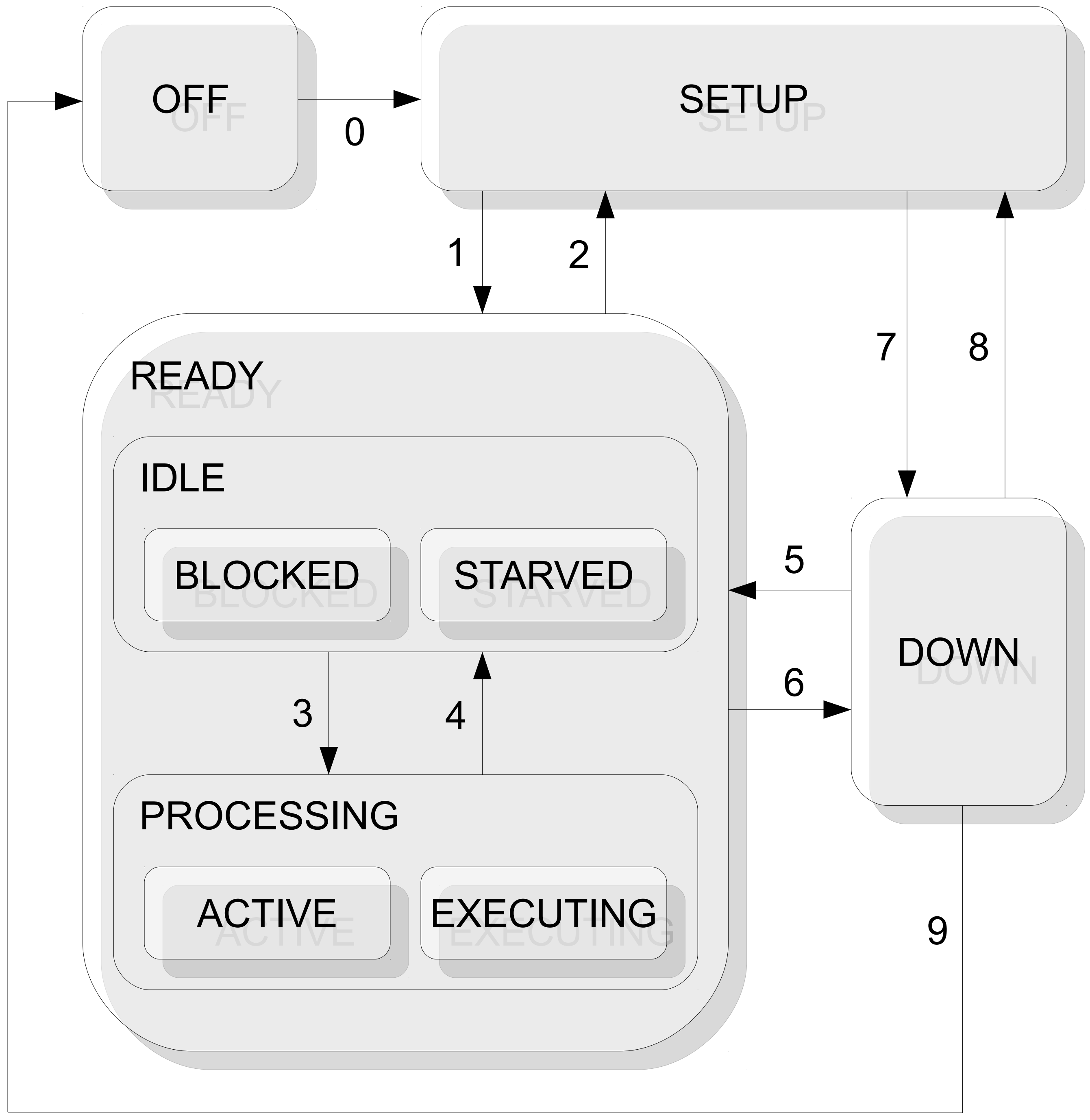
```
protected Controller(string zoneId, string laneId)
    : base(zoneId, laneId)
{...}
private static Controller _instance = null;

public static Controller Instance
{
    get
    {
        if (_instance == null)
        {
            _instance = new Controller("Cell", "Router");
        }
        return _instance;
    }
}
```

Controller will also act as proxy between business logic and GUI – all hardware and settings will be accessed trough Controller object:

```
Controller.Instance.Settings.LogLevel = Ipc.LogLevel.Verbose;
...
Controller.Instance.RoutingTaskIO.SuctionNozzleHome.Value = false;
Controller.Instance.RoutingTaskIO.SuctionNozzleWork.Value = true;
```

2.2 IPC state machine

State machine is based on IPC-2541 standard equipment state diagram:



IPC standard states are following:

- OFF *(Ipc devices are powered off. Controlling task is suspended)*
- DOWN *(Ipc device is either paused or error is active)*
- SETUP *(Ipc device is initializing)*
- READY-IDLE-BLOCKED *(Ipc device has a product and waits for next device to request it)*
- READY-IDLE-STARVED *(Ipc device is empty and waiting for product)*
- READY-PROCESSING-ACTIVE *(Ipc device is working – any activity like loading/unloading product, but not executing recipe)*
- READY-PROCESSING-EXECUTING *(Ipc equipment is executing a recipe – processing the product)*

State OFF has lowest priority and state EXECUTING has highest priority. Whole equipment state is equal to state of device task which has state with highest priority. Following table is listing possible transitions between states displayed on state diagram.

Arrow	Current state	Typical trigger	New state
0	OFF	Start requested	SETUP
1	SETUP	Initialization succeeded	Any READY substate
2	Any READY subsatate	ResetException thrown	SETUP
3	STARVED	Product received	ACTIVE
4	ACTIVE	Processing done	BLOCKED
5	DOWN	Error cleared or Start requested while paused	Any READY substate
6	READY	Error occured or Pause requested	DOWN
7	SETUP	Error occurred during initialization	DOWN
8	DOWN	Error cleared	SETUP
9	DOWN	Stop requested	OFF

2.3 Device tasks

Device tasks are threads where machine business logic is running. One device task represents same logical machine unit as “zone” in IPC2541 standard. Task should cover small enough part of machine where parallel processing is not any more needed – all that happens inside, must be possible to handle with single threading. Following are possible device tasks in standard inline router:

- Upper manipulator robot
- Lower routing robot
- Loading conveyor segment
- Worksegment
- Unloading conveyor segment

2.3.1 Task control interface

Device tasks are controlled by IPC controller object. Control interface has following properties:

- PendingRequest (enumeration)

  ```
      public enum Request { Stop = 0, Pause = 1, Start = 2 }
  ```

- EmergencyStop (boolean)
  Device controlling thread is interrupted by AbortException. Machine should go to OFF state (“stop”) as soon as possible.

- ServiceMode (boolean)
  Machine should go to DOWN (“pause”) state after compleating pending movement. In service mode movement speeds must be limited to harmless safe values.

- Task is reporting back following read-only properties:
  State (enumeration)

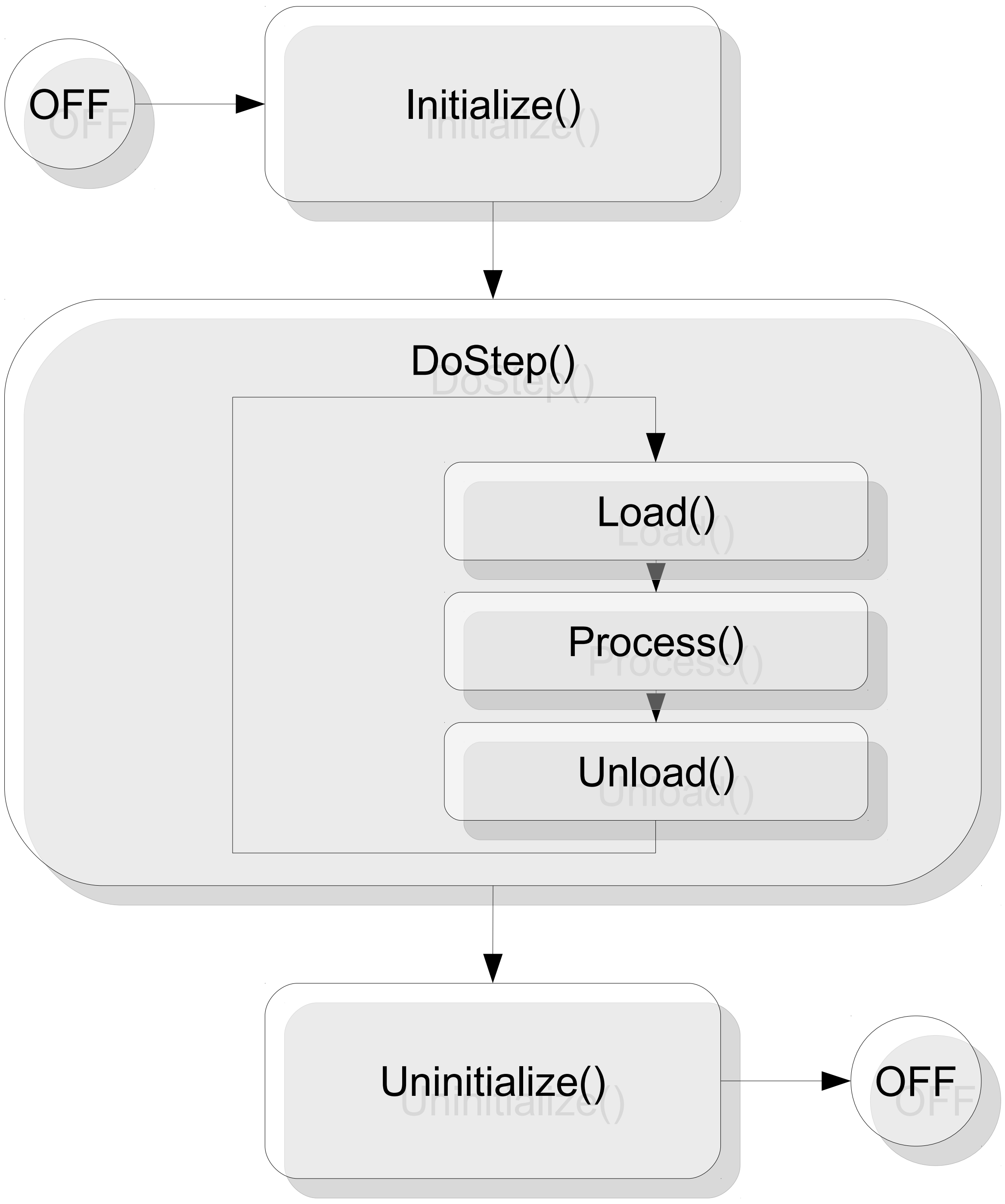
```
    public enum IpcState
    {
        Off = 0, Down = 1, Setup = 2, Blocked = 3, Starved = 4,    Active = 5, Executing = 6
    }
```

- HasAlarm (boolean)
  Cell has unrecoverable dangerous condition.

- HasError (boolean)
  Cell has unexpected situation where operator intervention is necessary for continuing.

- HasWarning (boolean)
  Cell has non-dangerous condition where production can be continued but may be interrupted soon for example due to running out of material.

2.3.2 Internal structure



IPC device task is inherited from abstract base class IpcDevice (see Ipc.dll documentation). For implementation it is mandatory to override 3 method calls:

- Initialize()
  This method is called once if task state is OFF and Start is requested. It is also called if ResetException will be thrown from any point of code in order to re-initialize device task. Initialization should load serialized product object from file, validate settings and hardware, check product presence in conveyors, grippers... If any abnormal condition detected, task start should be blocked by showing error message to operator.

- DoStep()
  Main processing code must be included here. It is good practice to keep this method as short as possible and split code into subroutines instead. This will help anybody who is reading this code, immediately understand what this task is doing. For example:

```
protected override void DoStep()
{
    DoRequests();
    Thread.Sleep(1);

    if (Product == null)
         Load();
    else if (Product.IsProcessed || Product.IsFailed)
         Unload();
    else
         ProcessPanel();
}
```

  It is not necessary to call subroutines inside DoStep() cyclically as this method is called cyclically from within base class anyways as long as machine is started.

- Uninitialize()
  This method is called once when machine is started and Stop is requested. Uninitialize() is also called if ResetException will be thrown from any point of code in order to reinitialize device task. It must dispose all resources that where initialized in Iinitialize() or DoStep() method, close ports and secure any dangerous hardware. Also serialize product data into file. Uninitialize() should complete as fast as possible to get machine response to Stop request faster. In most cases errors can be simply ignored, for example if some cylinder will be moved to initial position, then there is no danger, if it did not succeed. Also can be ignored some resources releasing as initialization might have been already failed earlier:

```
try
{
    if (IO != null)
    {
        IO.CameraLight.Value = false;
        IO.OpenGripper.Value = false;
    }
}
catch { }
```

```
        try { if (_camera != null) _camera.Disconnect(); }
        catch { }
```

Do not ignore safety related issues, like routing spindle stopping or large cylinder movements!

In those points of code where machine is waiting for something (for example SMEMA handshake from previous or next machine), method DoRequests() must be called cyclically in order to be able to stop or pause machine. If pending request will change from Start to Pause, code executing will be blocked inside DoRequests() method:

```
    while (!IO.SmemaPreviousAvailable)
    {
         DoRequests();
        Thread.Sleep(5);
    }
```

If pending request changes to Stop, then DoRequests() will throw StopException and base class will call Uninitialize() after that.

Note, that whenever anything is polled in “While” loop, Sleep() method with at least 1ms time has to be called inside it – otherwise thread would overload processor and application GUI may become unresponsive.

2.3.3 Creating task and initializing properties

IPC device tasks are created by controller in constructor. Each task type will implement own interface. This will allow create and initialize tasks according to specific settings:

```
    public ILoadingSegmentTask LoadingSegmentTaskFront { get; private set; }
    public ILoadingSegmentTask LoadingSegmentTaskRear { get; private set; }

    if (Settings.SimulationMode)
    {
        LoadingSegmentTaskFront = new SimulatedLoadingSegmentTask("Loading", "Front Lane");
        LoadingSegmentTaskRear = new SimulatedLoadingSegmentTask("Loading", "Rear Lane");
    }
    else
    {
        if (Settings.DualSegment)
        {
            LoadingSegmentTaskFront = new LoadingSegmentTask("Loading", "Front Lane");
            LoadingSegmentTaskRear = new LoadingSegmentTask("Loading", "Rear Lane");
        }
        else
        {
            LoadingSegmentTaskFront = new LoadingSegmentTask("Loading", "Front Lane");
            LoadingSegmentTaskRear = null;
        }
    }
```

In constructor Controller will also initialize task properties and connect permanent internal handshakes between tasks.

Note the re-usability of code – if for example conveyor segments on front and rear line are physically exactly same, then do not write own class for each but use same and give for each own IO object to be controlled:

```
    ((LoadingSegmentTask)LoadingSegmentTaskFront).IO = FrontLoadingSegmentIO;
    if (LoadingSegmentTaskRear != null)
        ((LoadingSegmentTask)LoadingSegmentTaskRear).IO = RearLoadingSegmentIO;

    ((LowerRobotTask)LowerRobotTask).IO = LowerRobotIO;
    ((LowerRobotTask)LowerRobotTask).Axes = LowerRobot;
    ((LowerRobotTask)LowerRobotTask).Camera = LowerCamera;

    ((UpperRobotTask)UpperRobotTask).IO = UpperRobotIO;
    ((UpperRobotTask)UpperRobotTask).Axes = UpperRobot;
```

2.3.4 IPC events

IPC device task states are changed by events – read-only State property can not be changed directly. Generating events is required for IPC CAMX messaging and statistics. Some of the events are already generated by IpcDevice base class, but some events have to be called from device task code:

- OnEquipmentStarved()
  Raised when device is waiting product. Device state changes to READY-IDLE-STARVED.

- OnEquipmentUnstarved()
  Raised when previous device offers product. Device state changes to READY-PROCESSING-ACTIVE.

```
        OnEquipmentStarved();
        while (!IO.SmemaPreviousAvailable)
        {
            DoRequests();
            Thread.Sleep(5);
        }
        OnEquipmentUnstarved();
```

- OnEquipmentBlocked()
  Raised when device is waiting Not Busy signal from next device. Device state changes to READY-IDLE-BLOCKED.

- OnEquipmentUnblocked()
  Raised when next device requests product. Device state changes to READY-PROCESSING-ACTIVE.

```
    OnEquipmentBlocked();
    while (!IO.SmemaNextNotBusy)
    {
        DoRequests();
        Thread.Sleep(5);
    }
    OnEquipmentUnblocked();
```

- OnItemIdentifierRead(itemInstanceId, laneId, zoneId, scannerId)
  Raised when product serial has been read. No state change.

- OnItemTransferIn(itemInstanceId, laneId)
  Raised when device receives product. No state change.

- OnItemWorkStart(itemInstanceId, laneId, zoneId)
  Raised when device starts processing product – executing recipe. Device state changes to READY-PROCESSING-EXECUTING.

- OnItemWorkPause(itemInstanceId, laneId, zoneId, pauseId)
  Raised when product processing will be temporarily stopped – for example machine Pause requested. Device state changes to READY-PROCESSING-ACTIVE.

- OnItemWorkResume(itemInstanceId, laneId, zoneId)
  Raised when processing of paused product will continue. Device state changes to READY-PROCESSING-EXECUTING.

- OnItemWorkComplete(itemInstanceId, laneId, zoneId)
  Raised when product is done – all instructions in recipe completed. Device state changes to READY-PROCESSING-ACTIVE.

- OnItemWorkAbort(itemInstanceId, laneId, zoneId, abortId)
  Raised when product processing is canceled before completion, for example by non-recoverable error. No state change.

- OnItemTransferOut(itemInstanceId, laneId)
  Raised when product has left device zone. No state change.

Base classes IIpc2541Events and Ipc2541Declarations contain declarations for all IPC-2541 standard events that can be used to send various information to other CAMX compliant machines.

2.3.5 Product object

Product is any item that is being handled or processed in machine. It can be different in different machine zones. Some examples are:

- In routing machine loading conveyor has product type “Panel”, but after depaneling upper robot and unloading conveyor are handling “Module”-s.
- In mounter machine Loading conveyor has product type “Panel”. Feeders and mounting robot has product type “Component”.
- In assembly pallet line “Component”-s will be put together into “Assembly”.

Product object can be any class (Module, Panel, Component...) that has been inherited from Product base class (Products.cs). Product base class is implementing minimal set of basic properties and methods required to handle product data. Most important of those are:

- Serial – product serial code, identifier (string).
- RecipePath – path to product recipe file or recipe name (string).
- IsFailed – flag to mark failed (“bad”) products (bool).
- FromFile(path) – static method to create product object from file.
- Save(path) – method to serialize product object into file.

Serialized product data files are used for storing products information while machine is stopped or powered off. Good practice is that every device task is able to continue from a point where process was interrupted before stopping. In all possible cases requirement to empty the machine before starting should be avoided.

Serialized product data files are stored in CommonApplicationData under ..\IPTE\”MachineName”\Data sub-folder. Each product data file should have unique name, for example combined from device task “ZoneId” and “LaneId” properties:

```
    _productPath = Paths.DataDirectory + "\\" + ZoneId + "@" + LaneId + ".product";
```

In most cases product is read-only property of device task. In that case product has to be loaded from file in a beginning of task Initialize() method (Important! If Initialize() will be interrupted before initializing product object, then Uninitialize() will delete product data file!). If product file does not exist, then product object is initialized as null:

```
    Product = Panel.FromFile(_productPath);
```

Product must be saved back into file in Uninitialize() method. If Product is null, existing file has to be deleted:

```
    if (Product != null)
        Product.Save(_productPath);
    else
        if (File.Exists(_productPath)) File.Delete(_productPath);
```

If product data persistence is very important in point of view of traceability, then product can be saved to file also during executing main code of device task to avoid data loss or mix-up in case of application or operating system crash.

2.3.6 Internal handshaking

Device tasks are communicating with each other using internal handshake objects HandshakeWithPreviousDevice and HandshakeWithNextDevice (Handshake.cs). Handshaking is similar to standard IPC (SMEMA) handshaking protocol by using “Available” and “Not Busy” signals. Additionally there is “Abort” signal to inform other handshake partner about transfer or process failure. This signal can have free application specific meaning.

Handshake interface includes also Data property with basic type “object”. It can be used to hand over Product data from one device to another. Both devices must use Product object with same data type.

Handshake objects are created in device task constructor.

```
public HandshakeWithPreviousDevice PreviousDeviceHs { get; private set; }
public HandshakeWithNextDevice NextDeviceHs { get; private set; }
...

public WorkSegmentTask(string zoneId, string laneId)
    : base(zoneId, laneId)
{
    PreviousDeviceHs = new HandshakeWithPreviousDevice(zoneId + " @ " + laneId);
    NextDeviceHs = new HandshakeWithNextDevice(zoneId + " @ " + laneId);
    ...
}
```

For connecting two handshake objects together is used method Connect(). Connections can be made in both directions – downstream and upstream. In most cases connections are already made by IPC Controller in constructor:

```
LoadingSegmentTask.NextDeviceHs.Connect(WorkSegmentTask.PreviousDeviceHs);
```

is equal to:

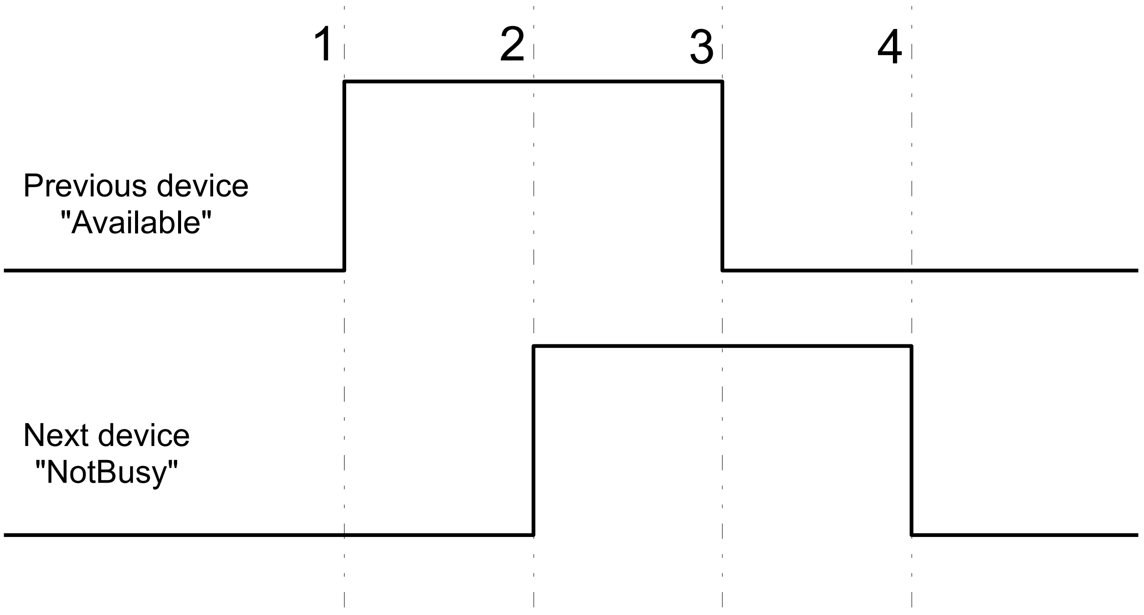
```
WorkSegmentTask.PreviousDeviceHs.Connect(LoadingSegmentTask.NextDeviceHs);
```

Handshakes can be connected also dynamically by master device if there is need to handshake with different devices:

```
this.HandshakeFromFeeder.Disconnect();
switch (selectedFeeder)
{
    case 0:
        this.HandshakeFromFeeder.Connect(Feeder0Handshake);
        break;
    case 1:
        this.HandshakeFromFeeder.Connect(Feeder1Handshake);
```

```
            break;
        case 2:
            this.HandshakeFromFeeder.Connect(Feeder2Handshake);
            break;
        default:
            // error - incorrect selection
            break;
    }
```

To keep handshaking well synchronized it is very important to follow always same signaling pattern. Following is example if previous device will set “Available” first, after that downstream device will set “Not Busy” signal. Signals should be reset in same order – sender first, then receiver.



1. Previous device has product ready to hand over and reference to Product object has been written to handshake Data property. Device is waiting “Not Busy” signal from next device.

2. Next device has copied reference to Product object into it's internal variable and is now ready to receive product. If devices are conveyors, then both will start belts at this point. Physical transfer begins.

3. Product has left previous device sensor, internal variable for product data can be reset to null. Physical transfer continues.

4. Next device has detected product arrival to sensor. Both devices can stop conveyors and continue with other tasks. Physical transfer is complete.

Example for previous device handshaking code:

```
    // phase 1
    NextDeviceHs.Data = Product;
    NextDeviceHs.BoardAvailable = true;

    OnEquipmentBlocked();
    while (!NextDeviceHs.NotBusy)
    {
        ...
        Thread.Sleep(10);
    }

    // phase 2
    OnEquipmentUnblocked();

    try
    {
        OpenStopper();
        IO.SegmentRun.Value = true;

        while (IO.PanelOnSegment)
        {
            ...
            Thread.Sleep(10);
        }
        NextDeviceHs.BoardAvailable = false;

        // phase 3
        while (NextDeviceHs.NotBusy)
        {
            ...
            Thread.Sleep(10);
        }
    }
    // phase 4
    finally
    {
        NextDeviceHs.BoardAvailable = false;
        NextDeviceHs.Data = null;
        IO.SegmentRun.Value = false;
    }

    if (!NextDeviceHs.Abort)
    {
        OnItemTransferOut(Product.Serial, LaneId);
        Product = null;
    }
```

Example for next device handshaking code:

```
    OnEquipmentStarved();
    while (!PreviousDeviceHs.BoardAvailable)
    {
        ...
        Thread.Sleep(10);
    }
```

```
// phase 1
OnEquipmentUnstarved();

Product = PreviousDeviceHs.Data as Panel;
PreviousDeviceHs.Abort = false;
PreviousDeviceHs.NotBusy = true;

// phase 2
try
{
    IO.SegmentRun.Value = true;

    while (PreviousDeviceHs.BoardAvailable)
    {
        ...
        Thread.Sleep(10);
    }

// phase 3
    while (!IO.PanelOnSegment)
    {
        ...
        Thread.Sleep(10);
    }
}
catch
{
    //abort handshake
    PreviousDeviceHs.Abort = true;
    throw;
}
// phase 4
finally
{
    IO.SegmentRun.Value = false;
    PreviousDeviceHs.NotBusy = false;
}

OnItemTransferIn(Product.Serial, LaneId);
```

If machine stop is requested, then previous device should clear the “Available” handshake and go to stop. The two devices, however, are asynchronous – there is a possibility that the previous device clears the available signal only a nanosecond after the next device has committed it to loading procedure. This case will results in a deadlock – the previous device will stop, but the next device will wait for signals from previous device and hence can not stop. The workaround for those deadlocks is to only allow stopping before the next device reads the Available signal from previous device – after the Available signal is read by next device the handshake must be fully executed before stop is allowed. For safe clearance of the signal there is a TryAbortOffering() method, it will clear the Available signal in thread safe manner or will return false if the next device has already committed to handshake.

In following example DoRequests() method is only called if next device has not yet locked handshake, otherwise will continue with unloading:

```
    NextDeviceHs.BoardAvailable = true;
    while (!NextDeviceHs.NotBusy)
    {
        // before checking stop or pause request, try to clear "available" signal to next
        if (NextDeviceHs.TryAbortOffering())
        {
            DoRequests();
            NextDeviceHs.BoardAvailable = true;
        }
        Thread.Sleep(10);
    }
```

Sometimes it is important not to lock handshake before receiving party has made some preparations, for example adjusted conveyor width according to incoming product. For that is used boolean method PeekBoardAvailable(), which will return “true” if “Available” signal is active. Difference from reading property is – it does not set internal locking flag. In following example downstream device will first adjust conveyor width and only after that will lock the handshake:

```
    OnEquipmentStarved();
    // peek if handshake is set without locking it
    while (!PreviousDeviceHs.PeekBoardAvailable())
    {
        DoRequests();
        Thread.Sleep(10);
    }
    OnEquipmentUnstarved();

    // copy incoming product data and set conveyor width accordingly
    Product = PreviousDeviceHs.Data as Panel;
    SetConveyorWidth(Product);

    // formal check if "available" is still active and lock the handshake
    if (!PreviousDeviceHs.BoardAvailable) return; // jump out from loading method

    // begin transfer
    PreviousDeviceHs.Abort = false;
    PreviousDeviceHs.NotBusy = true;
```

2.3.7 Realtime

Typical communication lag over ADS with TwinCAT PLC is 3-5 ms, witch is more than acceptable in most cases. But ADS communication is not real-time and there are random peaks in lag up to 10-20 ms. In worst cases, if actual PLC is running on different PC, even up to 200 and more.

All axes motion related processing (PTP motion profiles, multi-axes interpolation) is performed by TwinCAT NC axes real-time tasks (SAF and SVB). For abstraction and simplification can be used NcAxis and NcChannel objects from TcLib.dll.

Some specific processes require IO handling in real-time where random lag is not acceptable. Depending on requirement various techniques can be used:

- Challenge: Reading precise axis position value from hardware input signal front.
  Typical application: Measuring tool diameter by driving through laser sensor beam.
  Solution: Use server side ADS events.

  Server side events are generated by TwinCAT real-time system and events occurring exactly same time will have same time-stamp attached to them. Accuracy of events depend on real-time TwinCAT task cycle-time. Events are connected to axis position value and sensor value. If event occurs, value together with time-stamp will be stored in array. Later first sensor latching record in sensor events array will be found and using time-stamp, corresponding value from axis position events array can be retrieved.

- Challenge: Set hardware output from hardware input signal front.
  Typical application: Precise stopping of PCB on sensor without stopper.
  Solution: Use simple “trigger” in PLC that is enabled over ADS before expected event occurs.

```
PROGRAM MAIN
VAR
    StopLoadingSegmentTriggerEnable: BOOL;
END_VAR
...
IF StopLoadingSegmentTriggerEnable AND LadingSegmentInputs.PanelOnSegment THEN
    LadingSegmentOutputs.SegmentRun = FALSE;
END_IF
```

And corresponding code in PC application would be:

```
_enableHandle =_adsClient.CreateVariableHandle("MAIN.StopLoadingSegmentTriggerEnable");
...
_adsClient .WriteAny(_enableHandle, true);
IO.SegmentRun.Value = true;
while (!IO.PanelOnSegment)
{
    Thread.Sleep(10);
    ...
}
_adsClient .WriteAny(_enableHandle, false);
...
try { _adsClient .DeleteVariableHandle(_enableHandle); }
catch { }
```

2.3.8 Errors and warnings

Device task errors are reported using Error() method from IpcDevice base class. Method will take unique identifier, error caption, error description and array of possible solution actions as parameters. Solution actions will define which buttons will be enabled in GUI error panel. Possible solutions are declared as enumeration:

```
    public enum Solution { Retry = 0, Ignore = 1, Reset = 2, Close = 3 }
```

If operator will press button on error panel, method will return single solution as response. Error() is blocking method - execution will be blocked inside as long as operator will take an action. Other options to get out of method are:

- Stop request - StopException will be thrown
- Emergency stop - thread will terminate with ThreadAbortException

If operator chooses solution “Reset”, ResetException will be thrown and device task will reinitialize. ResetException can be catched if something has to be done before reinitialization.

```
    While (!IO.ProductIsPresent)
    {
        try
        {
            Solution solution = Error(Guid.NewGuid().ToString(),
                "Product lost",
                new Solution[] { Solution.Retry, Solution.Ignore, Solution.Reset },
                "Product was lost from gripper during move." + "\n" +
                string.Format("Please check gripper sensor:\n\t{0} ({1}) ",
                    IO.ProductIsPresent.Address, IO.ProductIsPresent.Caption) + "\n" +
                "Select RETRY to check sensor again." + "\n" +
                "Select IGNORE to ignore sensors state and continue." + "\n" +
                "Select RESET to delete product data and continue with empty gripper.");

           // IGNORE was selected – leave the loop
           if ( solution == Solution.Ignore) break;
        }
        catch (ResetException)
        {
            // RESET was selected – clear product data and reinitialize
            Product = null;
            throw;
        }
    }
```

Warnings are reported using Warning() method from IpcDevice base class. Method will take unique identifier, warning caption and warning description as parameters. Warning() is non-blocking method – message is set in error panel and thread will continue immediately. Warning messages will have only “Close” button available. It is also possible to close warning messages programmatically by calling ResetWarning() method with same identifier. Both methods can be called repeatedly if same identifier is used – if warning is already active in error panel, no new instance will be added, just caption and description will be updated.

```
    public FeederTask(string zoneId, string laneId)
        : base(zoneId, laneId)
    {
        _warningLowOfComponents = new EquipmentWarningEventArgs(
            "Component level low",
            Guid.NewGuid().ToString(), // generate unique identifier for warning
            laneId, zoneId);
        _warningLowOfComponents .Description =
            "Please add components into feeder.";
        ...
    }

    protected override void DoStep()
    {
        // set warning if feeder is running out of components soon
        if (IO.ComponentLevelLow)
            Warning(_warningLowOfComponents);
        else
            ResetWarning(_warningLowOfComponents);
        ...
}
```

2.4 Background monitoring thread

The monitoring thread is updating machine status properties (Service/Auto, Emergency...) according to general inputs data. It is also responsible for monitoring various hardware connectivity status and connect/ disconnect them if necessary.

2.4.1 Creation

Background monitoring thread is created in IPC Controller constructor, after all necessary hardware objects and settings have been initialized.

```
    _monitoringThread = new Thread(
    delegate()
    {
        try
        {
            MonitoringTask(); // thread function
        }
        finally
        {
            // Hal must be disconnected in same thread it was connected.
            try { DisconnectHal(); }
            catch { }
        }
    });
    monitoringThread.Name = "Background monitoring thread";
    _monitoringThread.IsBackground = true;
    _monitoringThread.Start();
```

2.4.2 Thread function

Thread function (MonitoringTask) contains endless loop that is:

- Checking ADS client connectivity status and connect/disconnect TwinCAT IO, axes, drive and interpolation channel objects accordingly.

- Will apply axes SpeedOverride parameter according to settings. SpeedOverride is applied even while machine is started.
- Monitoring cameras, HF spindle controller and other hardware objects and connecting/disconnecting them.
- Monitoring service key and emergency voltage inputs and write controller ServiceMode and EmergencyStop properties accordingly. Controller will propagate those properties to device tasks.
- Monitoring Beckhoff servodrives error and warning statuses, displays warning message in GUI and performs reset to drive errors.
- Will display warning message in GUI if air pressure drops or if door is not locked.
- Update and blink beacon lights (red, blue, yellow, green) according to machine current status.

2.4.3 Termination

Monitoring thread is terminated by Controller object Dispose() method:

```
protected override void Dispose(bool disposing)
{
    //dispose monitoring thread
    //will also disconnect HAL; look how the thread is initialized.
    if (disposing && _monitoringThread != null)
    {
        _monitoringThread.Abort();
        _monitoringThread.Join();
        _monitoringThread = null;
    }
    ...
}
```

Thread Abort() will force execution to jump out from thread function endless loop and HAL will be disconnected (see try-finally example code in 2.4.1 Creation)

3 Settings

There are two kinds of configuration objects in a machine: machine specific configuration (settings) and product specific configuration (recipes). The settings object is responsible for machine specific configuration.

There are multiple ways to store configuration: a database, Visual Studio Application Settings framework and custom configuration objects. We chose to use the path of custom configuration objects because it enables following features:

- Provides variable based access. Usage of hard coded string constants for variable access is a source of coding errors.
- Enables type safety. Configuration field type checking at compile time will reduce mistakes from type casting. In addition the type safety will enable intellisense (auto-complete), which will significantly reduce amount of needed work when using configuration fields in code.
- Has tool support, integration with Visual Studio. Again, the intellisense simplifies configuration access. Also the “consume-first” mode introduced in Visual Studio 2010 allows to use properties in code before they are declared and automatic generation of backing fields. This drastically simplifies moving of code constants into configuration.
- Easy to serialize. XML serialization, provided by .NET framework, makes loading and saving the settings objects a breeze. The setting files themselves are human readable and can be modified using virtually all third party text editors.
- Enables usage of Subversion for version handling. By integrating subversion into the program we can ensure that all changes are backed up in server and are easy to track. The XML files can be differentiated against previous versions allowing to revert all or selected changes.
- Easy to make snapshots of settings objects. To prevent accidental configuration changes while machine is running we make a snapshot of the settings before starting the machine. The machine will then operate on the snapshot ignoring all configuration changes until the machine is stopped and started again.
- We can store the configuration file in custom location. We chose to use Environment.SpecialFolder.CommonApplicationData folder, because all users in system have full read/write access to this folder. In this folder subfolder “\IPTE\{machine name}” will be created as root for settings:

```
_rootDirectory =
    Environment.GetFolderPath(Environment.SpecialFolder.CommonApplicationData) +
    "\\IPTE\\" + CellName;
```

If Windows has been installed with default folders and machine name is “DemoMachine”, it will create root for settings in:

- Windows 7 - “C:\ProgramData\IPTE\DemoMachine”
- WinXP - “C:\Documents and Settings\All Users\Application Data\IPTE\DemoMachine”

Settings root folder contains following subfolders:

- Settings.
  Folder to save general Settings object (Settings.cfg). This folder can contain additionally machine specific settings objects like ToolManager.cfg in routers.
- Products.
  Default folder to save product recipes ({product name}.zprd).
- Data.
  Folder to save temporary product objects. Those objects are used to store information about products in machine to be able to continue after stop, emergency stop or power-down.
- Logs.
  Folder to save daily log files in format “log {date}.log”. Also trace files for program debugging are stored here.
- Databases.
  Folder for statistics database.

3.1 General settings object

General settings are saved in subfolder “Settings” with filename “Settings.cfg”. Machine settings object (MachineSettings.cs) collects all common machine settings and all device specific setting objects.

In settings object constructor all reference types have to be created except those that are left intentionally null. It would be reasonable to initialize in constructor all settings so that in basic machine there is minimal need to change anything (except positions teaching).

```
    public class MachineSettings : ICloneable
    {
        public MachineSettings()
        {
            Mode = Mode.Offline;
            LogLevel = LogLevel.Verbose;
            AdsAmsNetId = string.Empty;
            SpeedOverride = 50; // [%]
            UpperRobotTaskSettings = new UpperRobotTaskSettings();
            LowerRobotTaskSettings = new LowerRobotTaskSettings();
        }

        public string AdsAmsNetId { get; set; }
        public Mode Mode { get; set; }
        public LogLevel LogLevel { get; set; }
        public double SpeedOverride { get; set; }
        public UpperRobotTaskSettings UpperRobotTaskSettings { get; set; }
        public LowerRobotTaskSettings LowerRobotTaskSettings { get; set; }
        ...
    }
```

Before starting machine, controller will make clone copy object from all device settings and write it to device task settings property. Using cloned copy object will prevent accidental unwanted behaviors if master settings are changed while machine is started. Those changes will be applied to device tasks only after stop/start cycle:

```
    public override void Start()
    {
        MachineSettings settingsSnapshot = (MachineSettings)Settings.Clone();
        UpperRobotTask.Settings = settingsSnapshot.UpperRobotTaskSettings;
        LowerRobotTask.Settings = settingsSnapshot.LowerRobotTaskSettings;
        ...
    }
```

.NET standard XmlSerializer class is used to write settings object into XML file. Loading from file:

```
    public static MachineSettings Load(string path)
    {
        FileStream stream = File.Open(path, FileMode.Open);
        try
        {
            XmlSerializer serializer = new XmlSerializer(typeof(MachineSettings));
            return serializer.Deserialize(stream) as MachineSettings;
        }
        finally
        {
            stream.Close();
        }
    }
```

And serializing into file:

```
    public void Save(string path)
    {
        lock (fileLock)
        {
            FileStream file = File.Open(path, FileMode.Create);
            try
            {
                XmlSerializer serializer = new XmlSerializer(typeof(MachineSettings));
                serializer.Serialize(file, this);
            }
            finally { file.Close(); }
        }
    }
```

Not all kind of objects can be serialized. Following types can not be used as settings:

- Dictionary
- TimeSpan

All settings you need to serialize must be public properties with public read and write access. Private fields will be ignored by XML serializer. If there are properties you would like intentionally leave out from serializing, mark them with XmlIgnore attribute:

```
    private bool _someFlag;

    [XmlIgnore]
    public bool SomeFlag { get { return _someFlag; } set { _someFlag = value } }
```

3.2 Recipe

Recipes are files that contain information about processes that has to be done in machine with product. All information in recipe should be product related, not machine specific. For example all positions must be given in product coordinate system not in global machine coordinates. Recipe file is zipped folder that may contain following files:

- Serialized recipe settings object.
- Serialized product drawing object.
- Recipe description and modification date as zip package metadata.
- Any machine specific files that are needed to define production process.

By default product recipes are stored in subfolder Products with extension .zprd (zipped product). This location can be customized depending on application needs.

As each recipe is single zip file, management of recipes is simple file handling – one can copy, duplicate, rename, move recipes as simple files. Recipe names have same rules and restrictions as file names in operating system have.

Recipes are packed and accessed using .NET ZipPackage class. Both recipe parts, settings and drawing, will be serialized to XML format. For drawing is used customized serializer that can be accessed through Save() method:

```
    Package package = ZipPackage.Open(fileName, FileMode.Create, FileAccess.ReadWrite);
    try
    {
        PackagePart part;
        XmlSerializer serializer;

        //Save the settings
        part = package.CreatePart(_settingsUri, MediaTypeNames.Text.Xml);
        serializer = new XmlSerializer(typeof(RecipeSettings));
        serializer.Serialize(part.GetStream(), Settings);
        package.Flush();

        //Save the drawing
        part = package.CreatePart(_drawingUri, MediaTypeNames.Text.Xml, CompressionOption.Normal);
        Drawing.Save(part.GetStream());
        package.Flush();
    }
    finally
    {
        package.Close();
    }
```

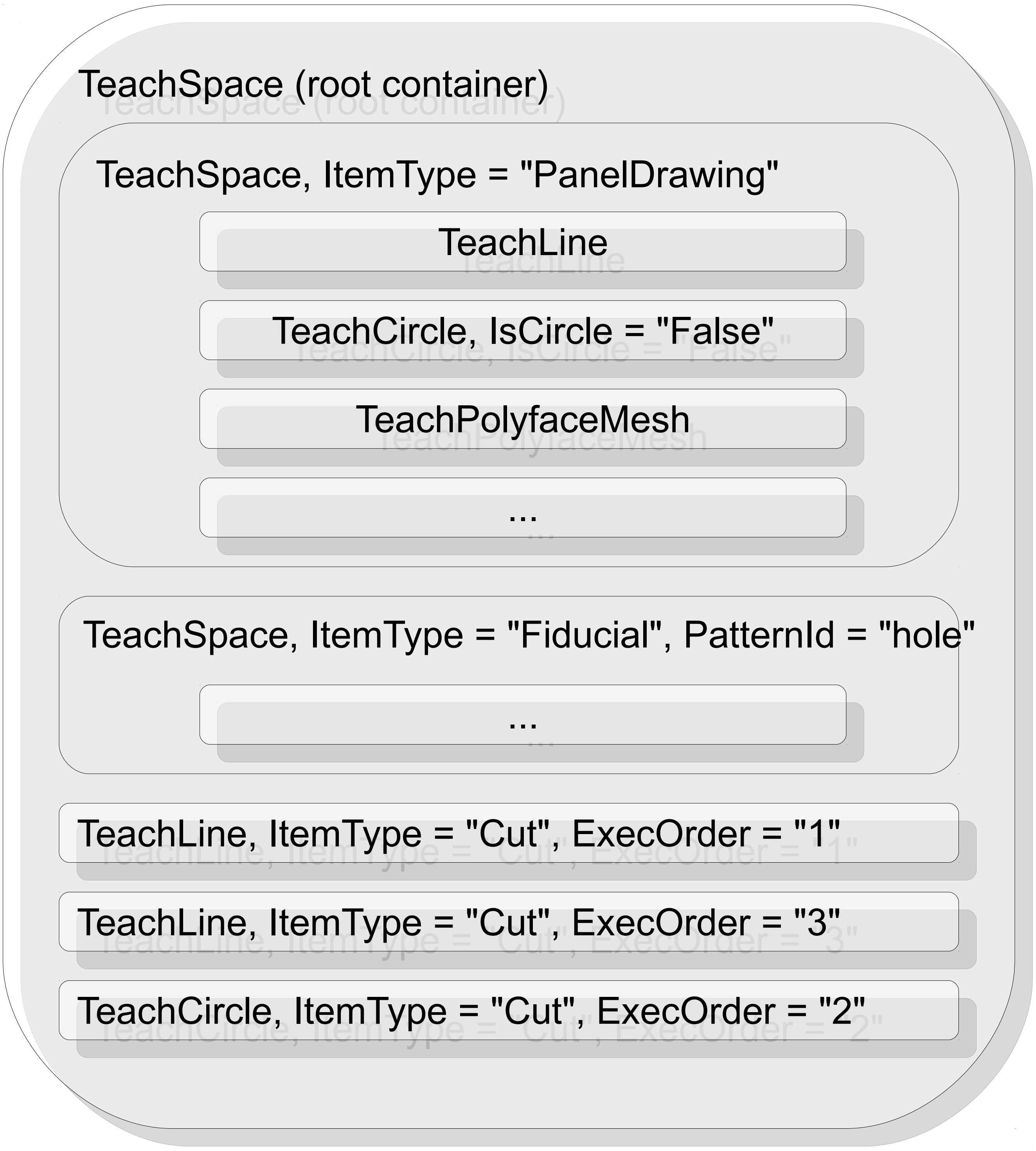
For recipe version handling is used SVN. Recipe Save() method will also commit changes if new recipe is created and will add it to repository. SNV repository has to be configured manually.

3.2.1 Recipe settings object

Recipe settings object is very similar to machine settings object – it is serialized into file using XmlSerializer class. Instead of having it as separate file, it will be included into recipe zip package.

In recipe settings are included all common product parameters that are not directly related to any recipe drawing element like for example: PCB measurements, default cutting speed and tool type for routers, fixture ID, etc.

3.2.2 Recipe drawing object


TeachSpace (root container)
TeachSpace, ItemType = "PanelDrawing"
TeachLine
TeachCircle, IsCircle = "False"
TeachPolyfaceMesh
...
TeachSpace, ItemType = "Fiducial", PatternId = "hole"
...
TeachLine, ItemType = "Cut", ExecOrder = "1"
TeachLine, ItemType = "Cut", ExecOrder = "3"
TeachCircle, ItemType = "Cut", ExecOrder = "2"

For creating/modifying recipe drawing object supporting classes from Teach.dll are used.

If machine supports CAD teaching, then also product drawing can be loaded into recipe drawing object. Currently two file formats are supported: AutoCAD DXF and Gerber. Simple drawings can be exported back to DXF format.

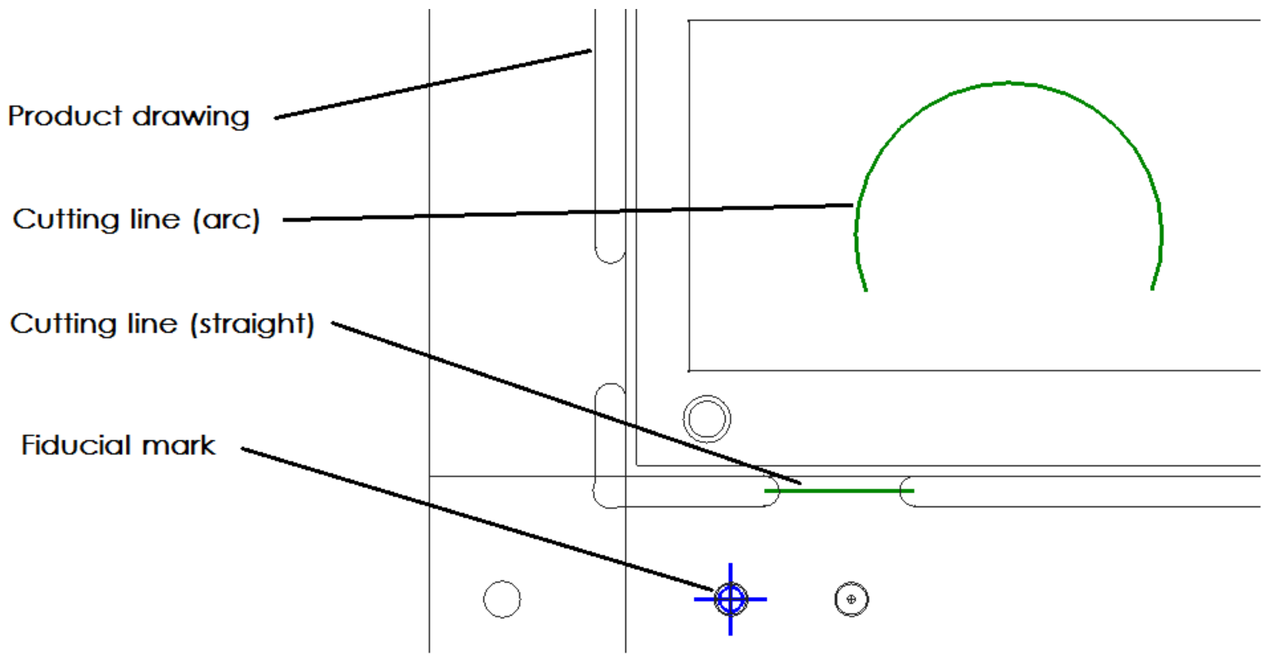
Recipe drawing is built as tree and is composed of two types of objects:

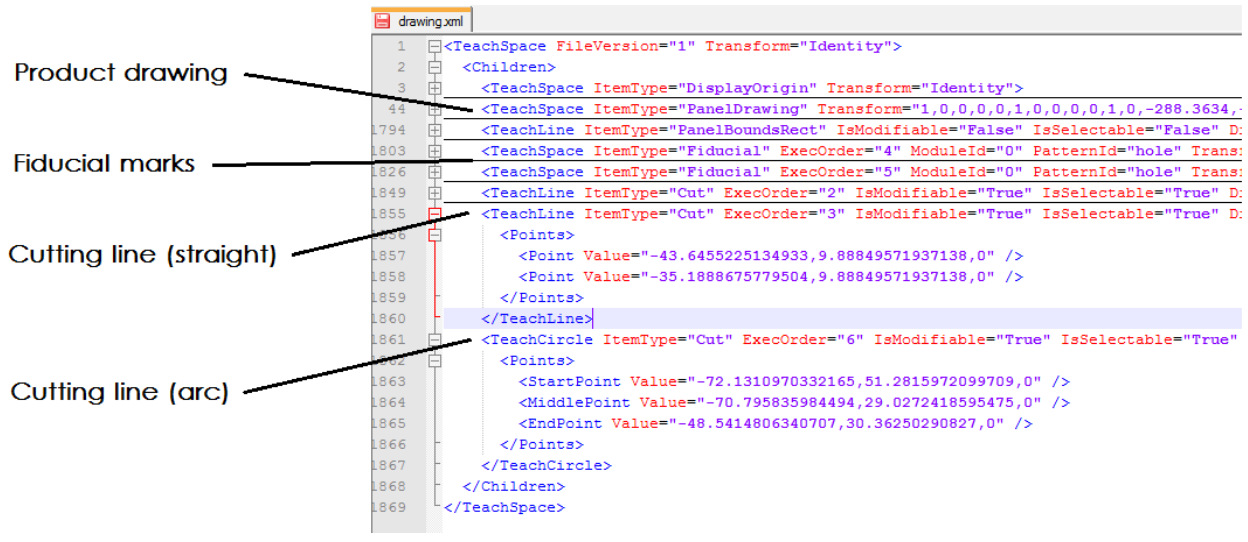
- Primitive objects – lines and shapes (TeachLine, TeachCircle, TeachPolyfaceMech). With line and shape objects all product drawing and teaching lines are interpreted. Primitive objects must be included in container type object.
- Container objects (TeachSpace) are used for grouping other objects. Containers can hold primitives as well as other containers.

All objects in recipe drawing can have attributes (Dictionary<string, string>). With attributes specific meaning or properties are applied to objects:

- “ItemType” attribute will define for what container or primitive is used.
    - ItemType = “PanelDrawing”, container holds product drawing objects;
    - ItemType = “Cut”, primitive is cutting line;
    - ItemType = “Fiducial”, container is panel or module fiducial;
- “ExecOrder” attribute will define in which order specific objects must be executed (for example cutting lines in routers).
- “Transform” atribut will apply 3D matrix transformation to container objects. In that case transformation will be applied also to all containing objects.

Application specific attributes can be freely defined. To make use of attributes easier, static class TeachExtensions will declare project specific extension methods to recipe drawing objects. For example {object}.GetItemType() can return item type as enumeration.





```
<TeachSpace FileVersion="1" Transform="Identity">
  <Children>
    <TeachSpace ItemType="DisplayOrigin" Transform="Identity">
    <TeachSpace ItemType="PanelDrawing" Transform="1,0,0,0,0,1,0,0,0,0,1,0,-288.3634,
    <TeachLine ItemType="PanelBoundsRect" IsModifiable="False" IsSelectable="False" D
    <TeachSpace ItemType="Fiducial" ExecOrder="4" ModuleId="0" PatternId="hole" Trans
    <TeachSpace ItemType="Fiducial" ExecOrder="5" ModuleId="0" PatternId="hole" Trans
    <TeachLine ItemType="Cut" ExecOrder="2" IsModifiable="True" IsSelectable="True" D
    <TeachLine ItemType="Cut" ExecOrder="3" IsModifiable="True" IsSelectable="True" D
      <Points>
        <Point Value="-43.6455225134933,9.88849571937138,0" />
        <Point Value="-35.1888675779504,9.88849571937138,0" />
      </Points>
    </TeachLine>
    <TeachCircle ItemType="Cut" ExecOrder="6" IsModifiable="True" IsSelectable="True"
      <Points>
        <StartPoint Value="-72.1310970332165,51.2815972099709,0" />
        <MiddlePoint Value="-70.795835984494,29.0272418595475,0" />
        <EndPoint Value="-48.5414806340707,30.36250290827,0" />
      </Points>
    </TeachCircle>
  </Children>
</TeachSpace>
```

3.2.3 Recipe buffer

To avoid slow file reading to affect cycle-time, recipe buffer is used. Recipe buffer is a class containing static dictionary of recipes loaded from files. Buffer allows to reuse already loaded recipe data for all products of same type without need to reload it every time from file when product arrives to machine. Recipe buffer is mainly meant to be used by device tasks.

Recipe will be loaded from file when product of given type first time arrives to machine. Recipe buffer will be cleared by Controller when machine is started. This will ensure that changes made to recipes, while machine was stopped, will be activated.

4 HAL

Hardware Abstraction Layer is collection of classes (hardware components) that provide abstract interfaces between hardware and controlling device tasks. In perfect cases device task should not be aware f cpecific hardware option or if there is linked hardware present at all.

4.1 IO objects

IO in machine is split into groups for easier modularization and re-usability of classes. Usually there is one IO object per one device task. Most machines have general IO object, that is not directly related with any of tasks. This object contains IO for control buttons, beacon, door locks, safety, etc.

Additionally there might be need to create separate objects for some specific hardware components in machine like teach pendant, servodrive inputs, etc.

TwinCAT IO objects are inherited from base classes ImageBase or StructBase from TcLib.dll. IO points are collected into lists named Inputs and Outputs. It is possible to access them directly from those lists, but for better usability it is advised to use named properties instead – one for each IO point:

```
public class GeneralIO : StructBase
{
    public Input<bool> StartButton { get; private set; }
    public Input<bool> StopButton { get; private set; }
     ...
    public Output<bool> StartButtonLight { get; private set; }
    public Output<bool> StopButtonLight { get; private set; }
     ...
}
```

IO point types are not limited only with boolean type. Supported types are:

C# type	TwinCAT type	TwinCAT variable size in bytes
bool	BOOL	1
byte	BYTE or USINT	1
char	STRING(1) or BYTE	1
decimal	not supported in PLC	16
double	LREAL	8
short	INT	2
int	DINT	4
long	LINT not supported in PLC	8

C# type	TwinCAT type	TwinCAT variable size in bytes
sbyte	SINT	1
float	REAL	4
string	STRING	depends on declaration
ushort	WORD or UINT	2
uint	DWORD or UDINT	4
ulong	ULINT not supported in PLC	8
DateTime	DATE	4
TimeSpan	TIME	4

"not supported in PLC" in table above does not mean that this type can not be used, just there is no data type currently available in TwinCAT PLC. Instead of that byte array or structure with specified size can be used.

In device task hardware is accessed through named properties of IO object:

```
    public RoutingTaskIO IO { get; set; }
    private void NozzleHomeAsync()
    {
        IO.SuctionNozzleWork.Value = false;
        IO.SuctionNozzleHome.Value = true;
    }
```

For reading inputs "Value" property can be omitted as implicit operator exists for `Input<T>` class:

```
    if (IO.SuctionNozzleIsHome) {...}
```

is same as:

```
    if (IO.SuctionNozzleIsHome.Value) {...}
```

4.1.1 TwinCAT structure based objects

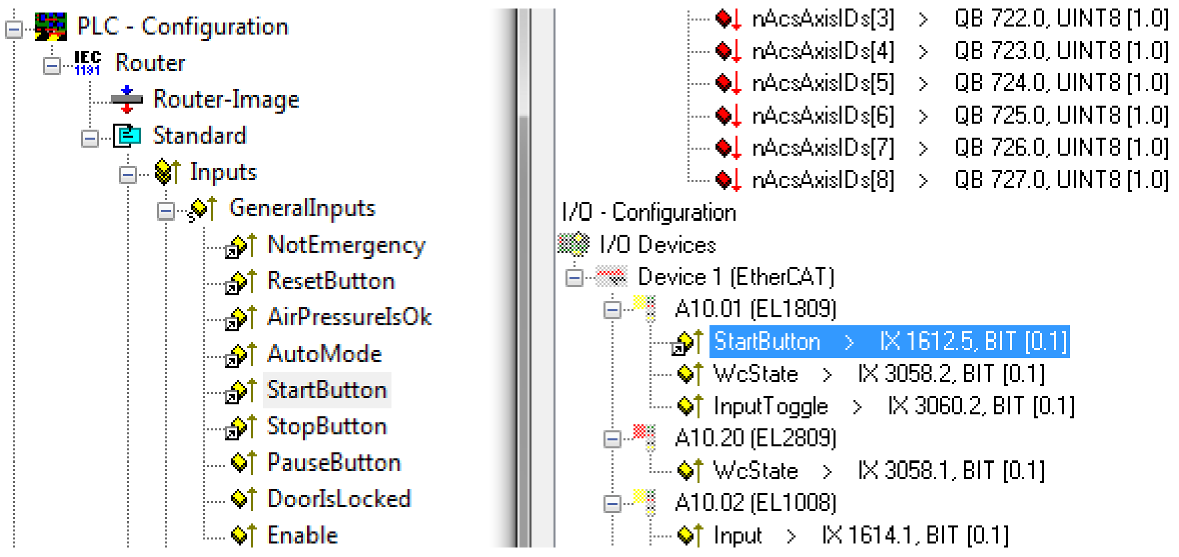
TwinCAT structure based IO objects are inherited from StructBase class. IO points are reading and writing PLC variable values therefore it requires analogues structures to be defined in TwinCAT PLC – one for inputs, one for outputs per each IO object:

```
    TYPE GeneralInputs :
    STRUCT
        StartButton: BOOL;
        StopButton: BOOL;
        ...
    END_STRUCT
    END_TYPE

    TYPE GeneralOutputs :
    STRUCT
        StartButtonLight: BOOL;
        StopButtonLight: BOOL;
        ...
    END_STRUCT
    END_TYPE
```

And to be able to link them with hardware, instances have to be declared in PLC image - inputs structure in inputs area, outputs structure in outputs area of image:

```
    VAR_GLOBAL
        GeneralInputs AT %I*: GeneralInputs;
        GeneralOutputs AT %Q*: GeneralOutputs;
        ...
    END_VAR
```

From PLC image IO is linked to hardware images using System Manager:


```
PLC - Configuration
  Router
    Router-Image
    Standard
      Inputs
        GeneralInputs
          NotEmergency
          ResetButton
          AirPressureIsOk
          AutoMode
          StartButton
          StopButton
          PauseButton
          DoorIsLocked
          Enable
```

```
      nAcsAxisIDs[3]  >  QB 722.0, UINT8 [1.0]
      nAcsAxisIDs[4]  >  QB 723.0, UINT8 [1.0]
      nAcsAxisIDs[5]  >  QB 724.0, UINT8 [1.0]
      nAcsAxisIDs[6]  >  QB 725.0, UINT8 [1.0]
      nAcsAxisIDs[7]  >  QB 726.0, UINT8 [1.0]
      nAcsAxisIDs[8]  >  QB 727.0, UINT8 [1.0]
I/O - Configuration
I/O Devices
  Device 1 (EtherCAT)
    A10.01 (EL1809)
      StartButton  >  IX 1612.5, BIT [0.1]
      WcState  >  IX 3058.2, BIT [0.1]
      InputToggle  >  IX 3060.2, BIT [0.1]
    A10.20 (EL2809)
      WcState  >  IX 3058.1, BIT [0.1]
    A10.02 (EL1008)
      Input  >  IX 1614.1, BIT [0.1]
```

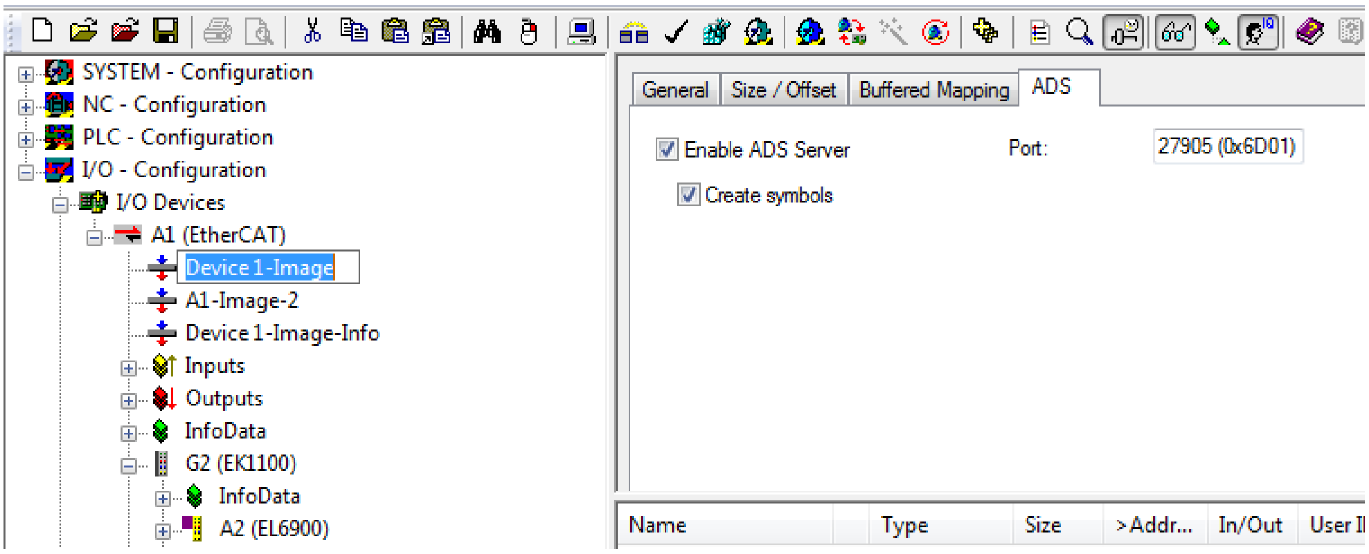
IO points are created in object constructor by methods AddInput<T>(...) and AddOutput<T>(...). Those methods will create the IO point, add it to list, and return reference to it. Returned reference is assigned to named public property:

```
    public GeneralIO(string inputStructName, string outputStructName)
        : base(inputStructName, outputStructName)
    {
        StartButton = AddInput<bool>("A1.01.06", "Start button", ".StartButton");
        StopButton = AddInput<bool>("A1.01.07", "Stop button", ".StopButton");
        ...
        StartButtonLight = AddOutput<bool>("A1.20.05", "Start button light", ".StartButtonLight");
        StopButtonLight = AddOutput<bool>("A1.20.06", "Stop button light", ".StopButtonLight");
        ...
    }
```

Third parameter in AddInput and AddOutput method must be exact field name inside PLC structure declaration and prefixed with dot (.). String parameters “inputStructName” and “outputStructName” in StructBase class constructor must be exact structure instance names in PLC. If structures are declared in VAR_GLOBAL section, then name must be prefixed with dot (“.GeneralInputs”). If structures are declared under program variables then names must be prefixed with program name and dot (“MAIN.GeneralInputs”).

4.1.2 TwinCAT image based objects

Image base IO objects are accessing directly hardware images in TwinCAT system, so there is actually no direct need for PLC project file. To enable direct reading, ADS server and symbol generation for this IO device has to be enabled:



☑ Enable ADS Server   Port: 27905 (0x6D01)

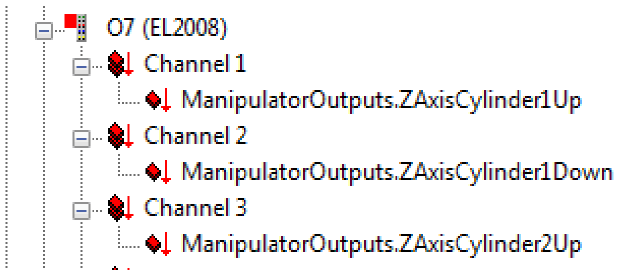
☑ Create symbols

NB! If hardware is linked to PLC variable, then it can be read with image based IO object, but not written. Use structure based object in that case instead. If hardware linked with PLC variables – two images are created, so make sure you are using correct image ADS port and ID:

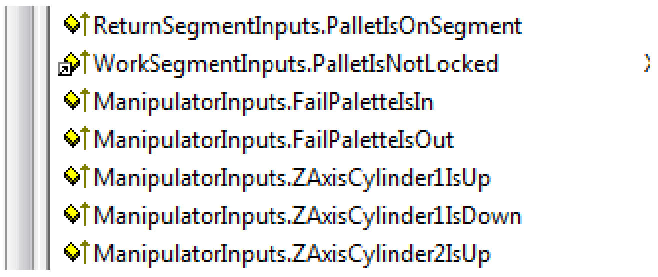
```
  public void ConnectHal()
  {
     string hostName = Settings.AdsAmsNetId;
     int port = 27905;
```

```
        int id = 1;

        if (!SafetyIO.IsConnected) SafetyIO.Connect(hostName, port, id);
        if (!ManipulatorIO.IsConnected) ManipulatorIO.Connect(hostName, port, id);
        ...
    }
```

Image based IO requires giving unique names to hardware points in System Manager:



```
O7 (EL2008)
  Channel 1
    ManipulatorOutputs.ZAxisCylinder1Up
  Channel 2
    ManipulatorOutputs.ZAxisCylinder1Down
  Channel 3
    ManipulatorOutputs.ZAxisCylinder2Up
```



```
ReturnSegmentInputs.PalletIsOnSegment
WorkSegmentInputs.PalletIsNotLocked
ManipulatorInputs.FailPaletteIsIn
ManipulatorInputs.FailPaletteIsOut
ManipulatorInputs.ZAxisCylinder1IsUp
ManipulatorInputs.ZAxisCylinder1IsDown
ManipulatorInputs.ZAxisCylinder2IsUp
```

This unique name in image is used as third parameter of AddInput and AddOutput methods. Wild-card character “*” can be used to match anything that is preceding this name:

```
    public ManipulatorIO()
    {
        FailPaletteIsIn =
            AddInput<bool>("G2.I09.1", "Palette is in", "*ManipulatorInputs.FailPaletteIsIn");
        FailPaletteIsOut =
            AddInput<bool>("G2.I09.2", "Palette is out", "*ManipulatorInputs.FailPaletteIsOut");
        ZAxisCylinder1IsUp =
            AddInput<bool>("G2.I09.3", "Z1 is up",  "*ManipulatorInputs.ZAxisCylinder1IsUp");
        ...

        ZAxisCylinder1Up =
            AddOutput<bool>("G2.Q07.1", "Z1 up", "*ManipulatorOutputs.ZAxisCylinder1Up");
        ZAxisCylinder1Down =
            AddOutput<bool>("G2.Q07.2", "Z1 down",  "*ManipulatorOutputs.ZAxisCylinder1Down");
        ...
    }
```

4.1.3 Other PLC-s

There is no limitation what kind of PLC-s can be accessed by IO objects as long as communication interface between PC and PLC is fast enough. OmronFins protocol has been used to create IO objects to communicate with OMRON PLC.

4.2 Hardware objects

Hardware are wrappers to communicate with different hardware components in machine: readers, printers, robots...

To make application better configurable, use standardized interfaces to declare hardware in controller:

```
    public ICamera Camera { get; private set; }
    public ICodeReader CodeReader { get; private set; }
    public ILaserMarker LaserMarker { get; private set; }

    protected Controller(string zoneId, string laneId)
        : base(zoneId, laneId)
    {
        ...
        switch (Settings.CameraType)
        {
            case CameraType.uEye:
                Camera = new uEyeCamera(); break;
            case CameraType.AVT:
                Camera = new AvtCamera(); break;
            default:
                Camera = null; break;
        }

        switch (Settings.CodeReaderType)
        {
            case CodeReaderType.MicroscanSerial:
                CodeReader = new MicroscanSerialReader(Settings.ReaderSettings); break;
            case CodeReaderType.CognexDatamanSerial:
                CodeReader = new CognexDataManSerialReader(Settings.ReaderSettings); break;
            default:
                CodeReader = null; break;
        }
        ...
    }
```

5 Logging

Logging is needed for:

- Traceability of production.
- Resolving configuration and teaching faults
- Analysing error conditions
- Debugging program code

5.1 Device task log

Logging from device tasks and controller are done using Log() method from IpcDevice base class. Log method is calling internally IPC2541 event EquipmentInformation, so all logging can be reported by using CAMX. EquipmentInformation event is also used in GUI log page to display on screen and save messages to file.

Log messages can be differentiated by there level of importance.

```
    public enum LogLevel { Silent = 0, Brief, Verbose }
```

- Silent - no logging should be done, except mandatory messages
- Brief - moderate amount of messages (e.g. mashine activity)
- Verbose - detailed log (e.g. debug messages)

There's no need to add time-stamp into log message as it's added by base class. Programmer has to define importance level of specific logging message in code. Which level messages are currently displayed will be defined by device task property LogLevel.

```
Log("This is very important information.", LogLevel.Silent );
Log("This is not so important message.", LogLevel. Brief );
Log("This is just blaa-blaa.", LogLevel. Verbose );
```

There is no need to log errors and warnings as both - Error() and Warning() - methods in IpcDevice base class will call EquipmentInformation event internally.

5.2 Trace log

Trace logging is used to collect debugging messages that have too specific programmers oriented meaning to be published to users of machine. Trace log files are saved in same folder where device task log files, just filenames have “Trace” prefix. For collecting messages is used TraceError() method from System.Diagnostics,Trace class.

```
    void AdsClient_AdsNotificationError(object sender, AdsNotificationErrorEventArgs e)
    {
        System.Diagnostics.Trace.TraceError(e.Exception.ToString());
    }
```

Tracing is activated in MainWindow constructor:

```
    public MainWindow()
    {
        //Output the trace errors to a log file
        string traceLogFile = string.Concat(
            Paths.LogDirectory,
            @"\trace log ",
            DateTime.Today.ToString("yyy MM dd"),
            ".log");

        System.Diagnostics.Trace.Listeners.Add(
            new System.Diagnostics.TextWriterTraceListener(traceLogFile));
        ...
}
```

Trace messages will be written to file when MainWindow is closed.

```
    protected override void OnClosing(System.ComponentModel.CancelEventArgs e)
    {
        ...
        try { Trace.Flush(); }
        catch { }

        base.OnClosing(e);
    }
```