TAM API

Developer Manual

This document gives an overview over the different components of the application programming interface for Triamec servo drives.

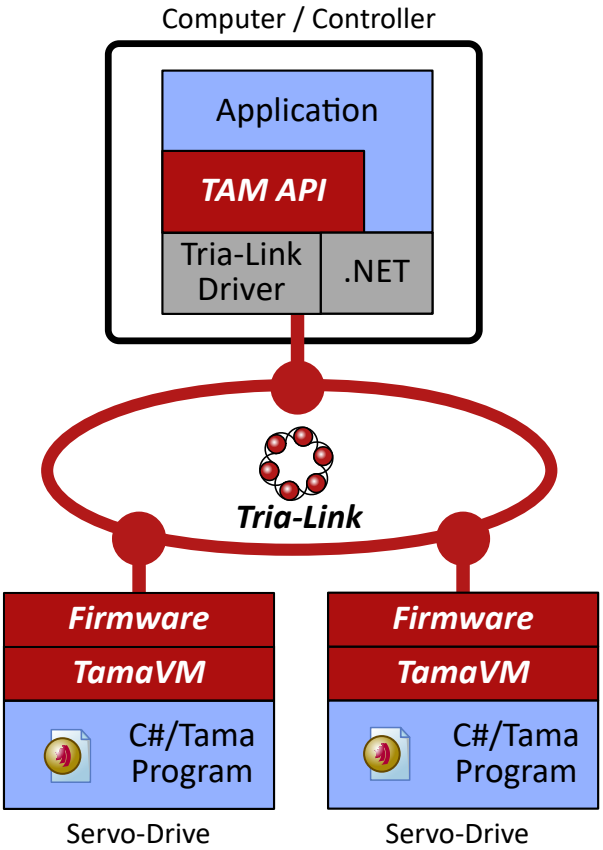

Figure 1: The TAM architecture.

The Triamec Advanced Motion solution provides fast digital servo drives, the Tria-Link real-time communication and a sophisticated software environment. Applications are built on top of the .NET framework and the TAM API. Custom code can be downloaded and run in real-time on the servo drive using the Tama virtual machine.

Document	SWNET_TamApiDeveloperManual_EP
Version	048, 2024-06-27
Source	C:\Users\Christian\Source\Repos\doc\Software\SWNET\SWNET_TamApiDeveloperManual\
Destination	T:\doc\Software\
Owner	chm

Table of Contents

List of figures

1 Introduction

The TAM software is a collection of libraries, tools and documentation helping customers to integrate Tria-Link [2] servo drives and other hardware devices in their environments. It does not require any real-time capabilities of the operating system, as real-time tasks are implemented decentralized on the individual hardware devices.

The TAM API libraries are built on top of the well recognized .NET framework [3]. Current releases of the libraries are available for .NET framework 4.6.2 up to .NET framework 4.8. A reduced set of libraries is available for .NET Compact Framework 3.9 upon request.

The .NET solution stack is also leveraged to ease development of real-time extensions to the drive firmware, referred to as *Tama* programs. Build integration and intelligent code completion is provided out of the box.

Fundamental part of the TAM software is the *TAM System Explorer* ([6], [7]), central tool for identification, setup and configuration of TAM systems. With the TAM System Explorer, the user navigates easily down to the particular registers of the servo drives. It allows editing parameters and observe signal values. The built-in oscilloscope allows tracking values with up to 100 kHz. The oscilloscope builds on top of a reusable acquisition framework.

This document focuses on the TAM API, part of the TAM Software. Typical customer motion applications build against this API.

Triamec provides various application samples on GitHub. They are organized into two topics, TAM API samples and Tama program samples.

2 Overview

After listing the most important concepts in chapter 2.1, chapter 2.2 outlines how to get hands on the TAM API in project development.

2.1 Concepts

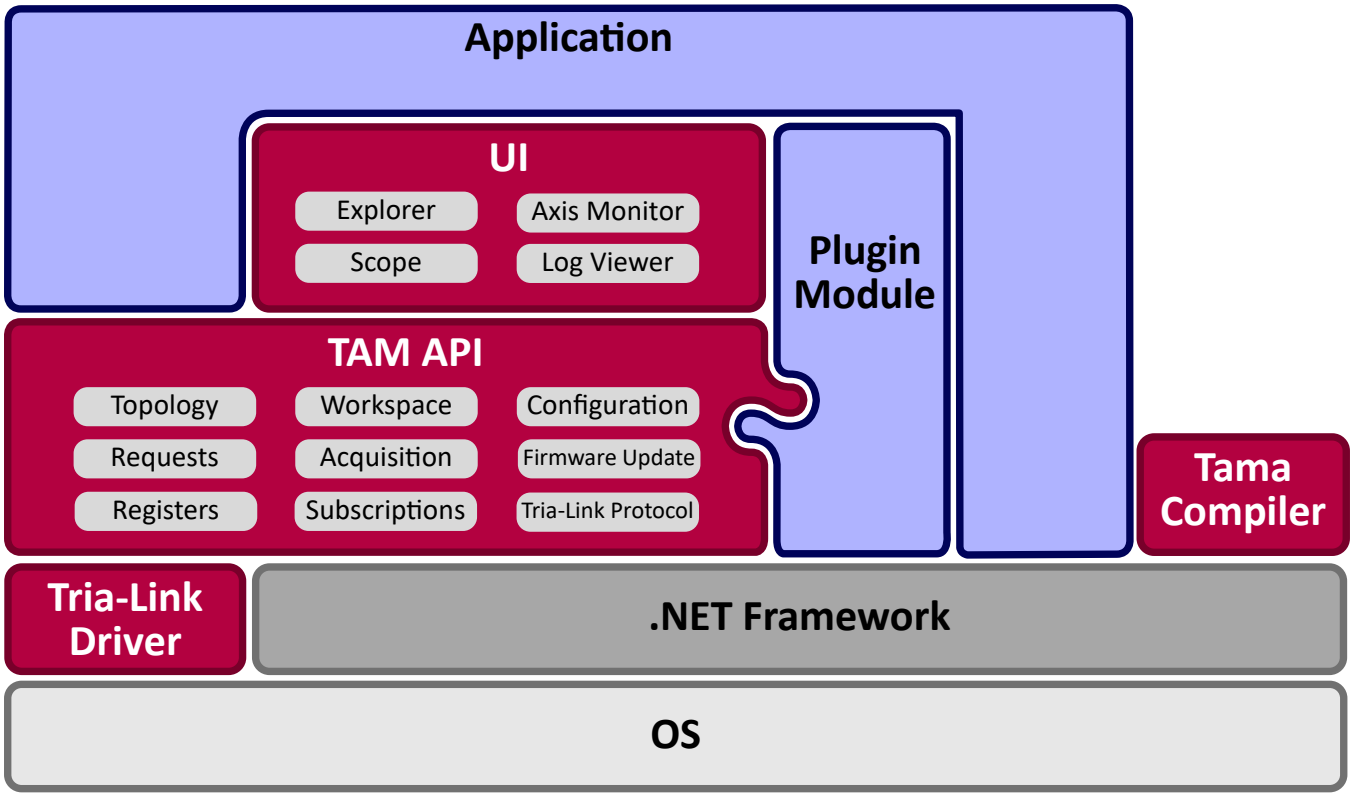

Figure 2: Software architecture

This diagram shows software components built bottom-up. Red components are provided by Triamec.

A motion application builds on top of the TAM API which presents an object oriented model of the TAM hardware and its functionalities. The Tria-Link protocol is transparent to the user and not in the scope of this document. Reusable GUI controls are offered by the TAM UI library. An extensible plug-in mechanism is provided.

The TAM API is built on top of the communication focused Tria-Link API. The Tria-Link API is not in the scope of this document. Moreover, the TAM API is open to alternatives like communication over USB.

Motion commands are similar to *PLCopen*. Isochronous data exchange and triggers allows data acquisition at high rates.

The TAM API is structured into different concepts which are explained in more detail in the following chapters.

The *TAM topology* (chapter 3) is a hierarchy of objects mapping the different parts of the system.

The *Registers* concept (chapter 4) allows configuring, controlling and acquiring data from hardware devices.

Subscriptions (chapter 5) serve as a base functionality for data acquisition, synchronization between hardware devices (for example for axis coupling) and event observation. The *Acquisition API* is an easy to use interface which builds on top of Subscriptions.

Chapter 6 about *motion* describes how to set up a drive for movement, how to move the axis and error conditions. Additionally, it describes the *scheduling* mechanism.

Tama (chapter 7) is the technology allowing extending the firmware of a hardware device with specific user defined routines for homing, electronic gears, error handling and more.

The *TAM Configuration* framework (chapter 8) allows persisting the parametrization of a hardware device, mainly its parametrization registers, in XML.

Parameters may also be persisted directly on the hardware device itself. Moreover, a device may be configured to a stand-alone mode where all information for *start-up* is persisted locally. The set-up of this mode is explained in chapter 9.

The TAM API allows *upgrading the firmware* over Tria-Link, with some restrictions (chapter 10).

To test software when hardware is not yet ready, an extensible *simulation framework* (chapter 11) can help.

The TAM Software comes with a bunch of libraries. Chapter 12 gives an overview and recommendations for *deployment*.

Encoder calibration and open- and closed-loop Bode analysis are provided out of the box including GUIs.

Modules define an extensibility point for rapid application development, integrating in the TAM system explorer and the TAM configuration framework. This feature is not currently documented, expect for the reference documentation of the `Triamec.Tam.Modules` namespace (internal issue 310).

2.2 NuGet Distribution

The TAM API libraries are provided as NuGet packages [12] on nuget.org. Find the release notes in a separate document [14]. Refer to figure 3 and have a look at the developer samples referred to above in the introduction. Triamec doesn't support linking to .dll libraries directly in your projects.

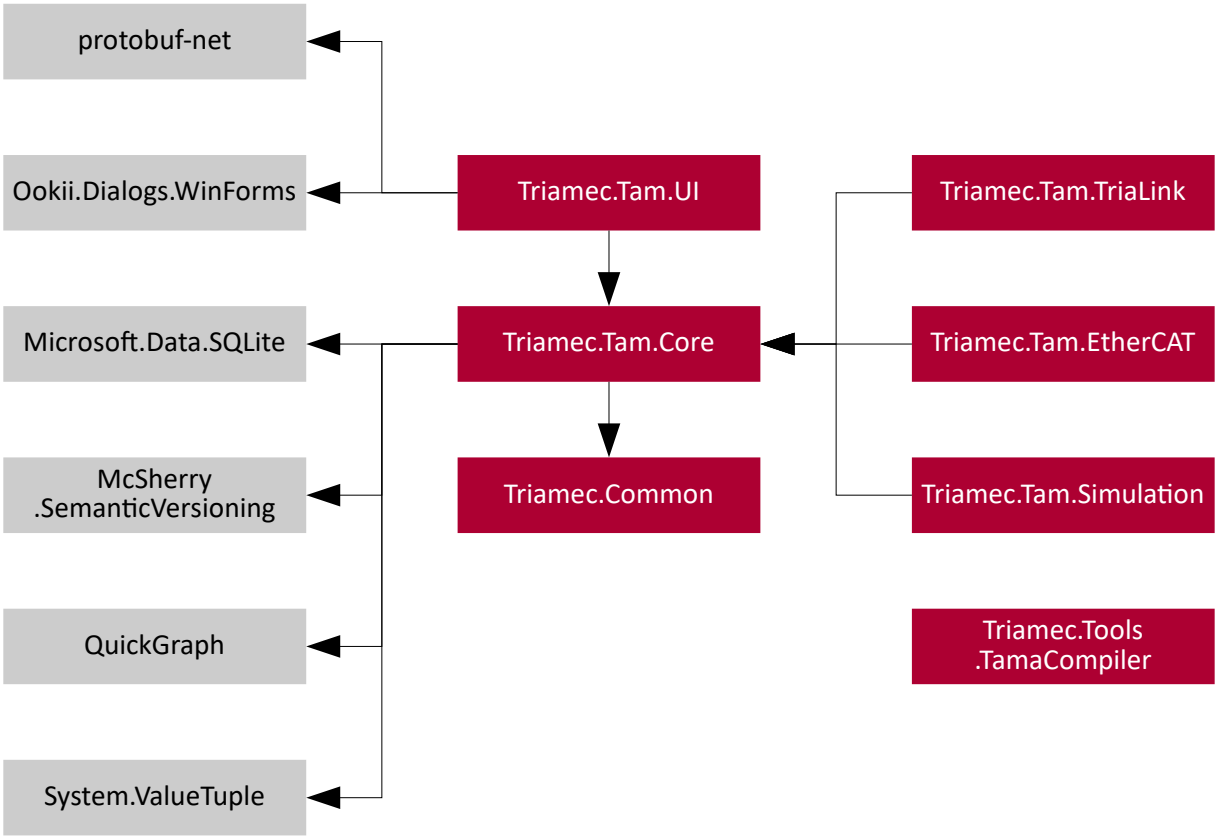

Figure 3: TAM Software NuGet packages

The typical requirements for an application project is just the Triamec.Tam.TriaLink NuGet with its dependencies. The Triamec.Tam.EtherCAT NuGet is currently a subset of the former package. Projects featuring real-time Tama programs need just to consume the Triamec.Tools.TamaCompiler NuGet in order to get it compiled. The Triamec.-Tam.UI NuGet is needed when integrating the TAM System Explorer into an application.
3D-party NuGets are drawn in gray and are not part of the TAM Software.

The packages support .NET Framework 4.6.2 and higher, but not yet .NET Core, .NET 5 and higher. Please contact us if you need support for .NET 5 and higher.

Caution The .NET Standard 2.0 version of some NuGet packages are not supported when working with Triamec drives.

Prior setting up a greater ecosystem of projects and solutions building upon NuGet functionality, please read the DevOps Recommendations in section 13.1 below.

Note There are quite some scenarios where installing the TAM Software [13] is a requirement:
- When drivers are required at runtime for access via PCI or USB
- When some GAC libraries are required while using the `Triamec.Tam.UI` NuGet package.

Updating to a new version of the TAM API is accomplished using the NuGet package manager.

3 Topology

The TAM Topology is the object-oriented tree containing corresponding instances of all hardware devices with their respective capabilities. The application queries the topology tree to find the hardware devices it is built for. It informs applications about changes within the tree through an observer pattern.

It is a logical tree of objects related to each other with a *has-a* association. All objects are instances of the `ITamNodeComposite` interface for consistent parent and child navigation. Each node has a `Nodes` collection, a `ParentNode` and two indexers to find child nodes by index or by name. This helps applications to do generic queries within the tree.

The topology is build on top of the Tria-Link protocol, which itself is supported by different *data-link* layers:

- The native Tria-Link token-ring as explained in the Tria-Link user manual.
- A hardware simulation used for early software testing.

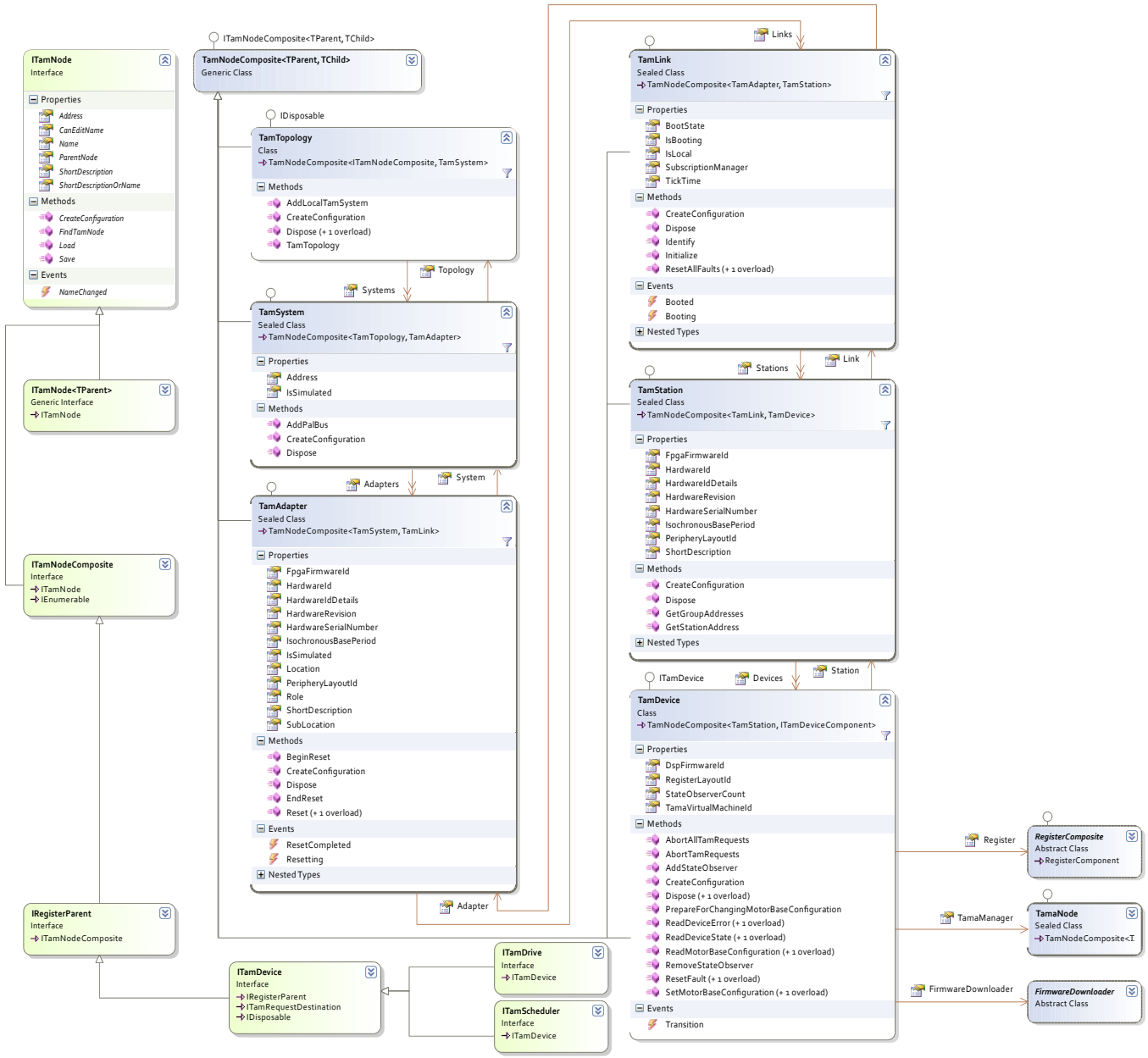

Figure 4: TAM Topology classes and interfaces overview

Each instance in the topology is TAM node composite, which can contain further TAM nodes. Please note the hierarchic levels of the different classes of TAM nodes. The bottom class, TAM Device, contains registers, a Tama manager and a firmware downloader, among others, which are themselves again TAM nodes.

- 3rd party protocols.

The TAM topology consists of the following classes, all from the `Triamec.Tam` namespace, listed top down:

`ITamNode`

Many classes in the TAM API implement the `ITamNode` interface. Such a TAM node is addressable by a

unified resource identifier (URI), the `Address`. This makes it possible to locate them given a URI, using the `FindTamNode` method. This can be used to persist references to nodes.

For most nodes, the `Name` of the node and its parent nodes defines its address. Because names may be changed, addresses are only unique among one TAM topology and references may become invalid because of renaming within the topology.

TamTopology

The root of the hierarchy. Starting point of all applications built on top of the TAM API.

```
TamTopology topology = new();
```

At the end, all hardware resources must be freed. Therefore, the topology needs to be disposed.

TamSystem

Represents a local or remote control system. This is typically the computer the application runs on.

```
TamSystem system = topology.AddLocalSystem(DataLinkLayers.TriaLink);
```

This call sets up access to the drives via a Tria-Link PCI adapter card.

The returned `TamSystem` instance is added to the topology.

Based on your hardware setup, you may want to pass a different argument:

- `DeviceUsb` for Triamec devices attached directly via USB.
- `TriaLinkUsbObserver` if you use the USB plug of a Tria-Link PCI adapter card or a TLU1 box.

In order to access devices attached to a network, use the following API:

```
var nicName = "Ethernet 3"; // The name of a network interface card as shown e.g. by ipconfig
TamSystem system = topology.ScanNetworkInterfaces(nicName)[0].ParentNode;
```

For more information about networked access, refer to application note AN123 [15].

Have a look at the *Acquisition* developer sample to see this code fully integrated.

Usage `TamSystem` implements `IDisposable`. However, you must not dispose the instance by yourself since ownership is taken by topology. This holds for all nodes in the hierarchy.

TamAdapter

Computer hardware connected to external hardware devices using the *Tria-Link* communication protocol.

Examples are the TL PCI card or an USB port. In large TAM systems, more than one adapter card can be connected to one PC. This may help to distribute traffic to separate Tria-Links, while maintaining the ability to operate them from the same software in one PC.

TamLink

The communication channel connecting the adapter with hardware devices using the Tria-Link protocol.

Typically, an adapter has exactly one link and plays the role of the *master* participant in the data-link

layer protocol. Within one link, all hardware devices are synchronized and may communicate with each other in real-time.

Booting
At the beginning, the TAM link does not show the connected hardware devices. Instead, client code needs to invoke one of the boot procedures:

- *Initialize* – Find all hardware devices in the Tria-Link, including newly inserted hardware.
- *Identify* – Find all hardware devices with a legal address. Typically, hardware devices don't have a legal address after power cycle.

```
TamLink link = system[0][0]; // get the 1st link on the 1st adapter.
link.Initialize();
```

Booting is a long-running process and may last for a quarter of a minute.

TamStation

Addressable party within the link.

This is typically one-to-one related with a hardware device, but there exist devices seen as multiple stations. This is the concept of virtual stations. For example, the PCI adapter card TLC100 is seen as two stations with separate addresses, one to communicate via PCI bus with the PC, and one to communicate with the microprocessor on the card.

TamDevice

The *microprocessor* in a station, for example, a digital signal processor (DSP).

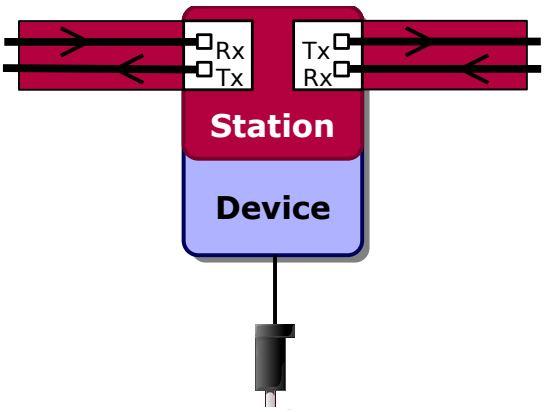

Figure 5: Station and device relationship.

The TAM station is visible in the Tria-Link and ensures communication. The TAM Device, for example a servo drive, is a CPU implementing the control of the motor.

A station may have zero or one device. For example, Triamec Motion AG servo drives are always equipped with a microprocessor. I/O-modules and PCI adapter cards may optionally be equipped with a microprocessor.

All devices have registers (see chapter 4) which they may publish (see chapter 5). Tama programs may

be loaded and executed (see chapter 7).

Each device implements the device state machine (see chapter 6.1).

ITamDrive

Servo drive with typically one axis, capable for doing motion(see chapter 6).

ITamScheduler

TAM device containing Schedules. See section 6.5 for more details.

For more information about the life-cycle of different instances of the above classes and interfaces, see section 13.7.

4 Registers

Registers is the most important concept of the TAM API to communicate with TAM devices. Instead of defining lots of different commands and command arguments in the Tria-Link protocol, most of the communication is done by reading and writing registers, and a register *commit* mechanism (section 4.2).

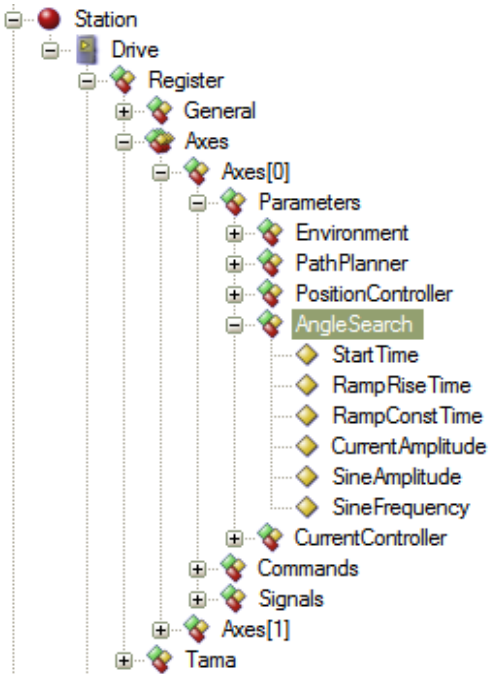

Figure 6: Part of the register tree as viewed by the TAM System Explorer.

Registers are a general purpose concept to configure a TAM Device. For a clear arrangement, registers are ordered in a tree structure.

A single register abstracts a memory location in the RAM of the device, together with its type, accessibility and name. For clarity's sake, registers are nested in a tree structure similar to the topology hierarchy; there are composites and arrays of registers.

All registers together form the *register layout* of a TAM device.

Different devices have different register layouts. Therefore, each register layout has an identifier, the *RLID*. Each device bespeaks its RLID (`TamDevice.RegisterLayoutId`), such that the correct register layout may be instantiated. Register layouts are saved in *register catalog* libraries which are loaded as needed by the TAM framework.

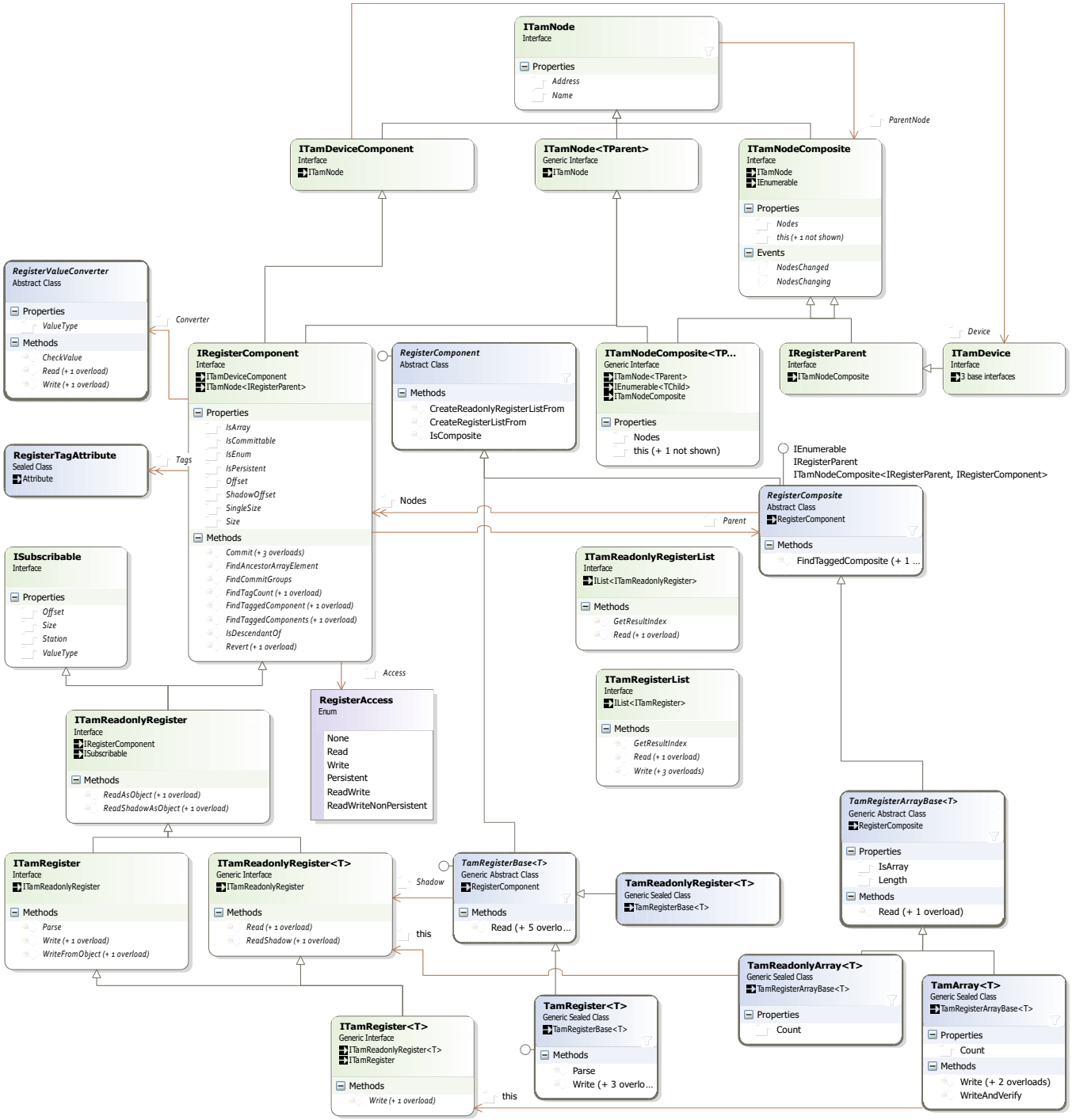

Figure 7: The register classes and interfaces

Just like the TAM Topology, the registers of a TAM Device form a hierarchy. In order to support registers with strongly typed values, composite registers like register arrays, light weight array element registers, read-only registers and more, many classes and interfaces are introduced. The most common functionalities out of these classes are finding a specific register, and reading, writing and committing that register.

In practice, three main type of registers are found:

1. *Parameters*
   Used to parametrize the firmware, for example the position controller of an axis on a drive.

2. *Commands*
   Commit switches (see section 4.2 below) and low level triggers for internal firmware state machines.
3. *Signals*
   Read-only registers publishing the state of the firmware and allowing for data acquisition and synchronization.

4.1 Retrieving Registers

Accessing registers in an application starts by consuming the `Triamec.Tam.TriaLink` NuGet package for Tria-Link drives, or the `Triamec.Tam.EtherCat` NuGet package for EtherCat drives, respectively.

Both packages contain libraries with register layout namespaces which you can import:

```
using Register = Triamec.Tam.Rlid19.Register;
```

Given a device instance, get hold to its root register:

```
ITamDevice device = …;
var register = (Register)device.Register;
```

Climb down the register by dereferencing individual members:

```
var appRegister = register.Application.Variables.Floats[0];
```

Use the TAM System Explorer to learn about the structure of the register layout and to get the addresses of the registers needed in the code.

If you need to write code supporting devices with different register layouts, read chapter 13.4: Working With Multiple Register Layouts.

Access the register:

```
float backup = appRegister.Read();
appRegister.Write(47.11f);
```

4.2 Committing

Parameters are read by the firmware in real time. Therefore, it would be dangerous when an application changed for example the current controller parameters one by one. For this sake, the *transactional* model of committing helps and enforces applications to change parameters all-at-once.

For example, the position controller parameters form the position controller *commit group*. Each commit group has a special *commit switch* command register assigned. After changing some parameters,

this boolean register needs to be set to true, which marks the end of the transaction.

Each committable register has a shadow register. Writing to a committable register is always redirected to its shadow register. When the commit switch is set, the firmware copies all shadow registers atomically to the committable registers.

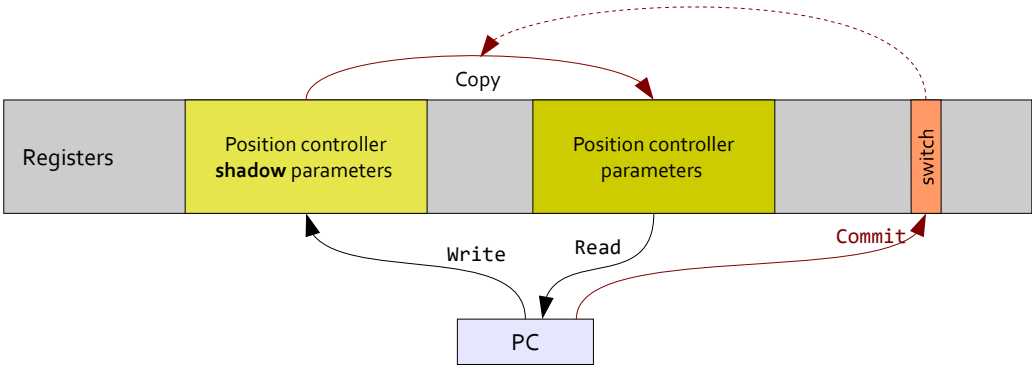

Figure 8: Committing and shadow parameter registers

Writing a value to a position controller parameter redirects the change to its shadow. When the position controller commit switch is written, all shadow parameters of the position controller commit group are copied to the actual parameters.

The uncommitted values cannot be read back. Therefore, writing values and committing should be done in sequence without any delay. Specifically, don't misuse the shadow as caching mechanism.

The commit switch is implemented as a normal command register. All commit commands are Boolean, and reverted to False automatically, when the copy procedure has finished.

Parameter Group	Commit Switch (Boolean Command Register)
General	General.Commands.CommitParameter
PathPlanner	Axes[].Commands.PathPlanner.CommitParameter
PositionController	Axes[].Commands.PositionController.CommitParameter
CurrentController	Axes[].Commands.CurrentController.CommitParameter

Warning Do not change parameters before the last commit finished.

4.3 Register Converters

Registers hold simple numeric values only. More complex values, such as strings, cannot be directly represented. Therefore, a register composite or leaf may have a register value converter specified, defined by the register layout designer. The converter provides applications with a more sophisticated view of its underlying register data, and is also able to parse such values and save them back to the registers.

For example, the device name is represented as an integer array. It's value is accessed as follows:

```
Triamec.Tam.Rlid19.Register register = …;
var name = register.General.Parameters.DeviceName.Converter.Read().ToString();
register.General.Parameters.DeviceName.Converter.Write(name + "!");
```

4.4 Speeding Up With Register Lists

Each time a register is read or written, a Tria-Link packet is sent, and a response is received. However, the protocol allows reading or write multiple registers at once. Additionally, it is possible to send multiple packets before waiting for a response.

These performance gains can be achieved using the `ITamRegisterList` and `ITamReadonlyRegisterList` interfaces, which are retrieved using `RegisterComponent.CreateRegisterListFrom` and `RegisterComponent.CreateReadonlyRegisterListFrom`.

```
ITamRegisterList list = RegisterComponent.CreateRegisterListFrom(
    modUpReg, modDownReg, velReg, accReg, jerkReg, drfReg);

TamValue32[] writeBuffer = list.CreateWriteBuffer();

TamValue32Pair pair = Float64.ConvertToTamValue32Pair(Math.PI * 2);
writeBuffer[list.GetWriteIndex(modUpReg)] = pair.Value0;
writeBuffer[list.GetWriteIndex(modUpReg) + 1] = pair.Value1;

writeBuffer[list.GetWriteIndex(modDownReg)] = 0;

writeBuffer[list.GetWriteIndex(modDownReg) + 1] = 0;

writeBuffer[list.GetWriteIndex(velReg)] = 50f;

writeBuffer[list.GetWriteIndex(accReg)] = 500f;

writeBuffer[list.GetWriteIndex(jerkReg)] = 5000f;

writeBuffer[list.GetWriteIndex(drfReg)] = 1f;

list.Write(writeBuffer);

modUpReg.Commit();
```

The listing demonstrates fast parametrization of a path planner. Note that length and used indexes of the write-buffer are an implementation detail of the TAM Software. The commit is done individually at the end, such that data is only committed if writing succeeded.

While the order in which the registers are written is maintained, they are not written atomically. The act of simultaneously copying is performed during commitment.

The similar listing for reading back the parametrization follows:

```
ITamReadonlyRegisterList rlist = RegisterComponent.CreateReadonlyRegisterListFrom(
    modUpReg, modDownReg, velReg, accReg, jerkReg, drfReg);

TamValue32[] readBuffer = rlist.Read();

int index = rlist.GetResultIndex(modUpReg);
double modUp = Float64.ToDouble(readBuffer[index], readBuffer[index + 1]);

index = rlist.GetResultIndex(modDownReg);
```

```
double modDown = Float64.ToDouble(readBuffer[index], readBuffer[index + 1]);
float vel = readBuffer[rlist.GetResultIndex(velReg)];
float acc = readBuffer[rlist.GetResultIndex(accReg)];
float jerk = readBuffer[rlist.GetResultIndex(jerkReg)];
float drf = readBuffer[rlist.GetResultIndex(drfReg)];
```

5 Subscriptions and Acquisitions

Tria-Link provides a *publish/subscribe* mechanism in order to exchange data between Tria-Link stations in real-time.

This includes receiving isochronous data on a host computer. The complexity of acquiring such data from different devices simultaneously is hidden in the *Acquisitions API* (section 5.4). However, it might be worthy to skim through the following subscription related sections in order to understand the resources consumed by the API.

5.1 Subscriptions

In a subscription, one *publisher* station sends values of up to 5 of its registers, and one or more *subscriber* stations receive these values as they are sent in isochronous data. The number of subscriptions a station can publish depends on the product. Current products by Triamec Motion AG may publish up to 16 subscriptions and subscribe to 16 other subscriptions. This results in 80 published and 80 subscribed values.

Every device defines an isochronous data rate at which isochronous subscription packets are sent (`IsochronousBasePeriod`). For servo drives, this is the frequency rate of its path planner task, which is typically 10 kHz. Subscription participants need to be on the same link for correct synchronization.

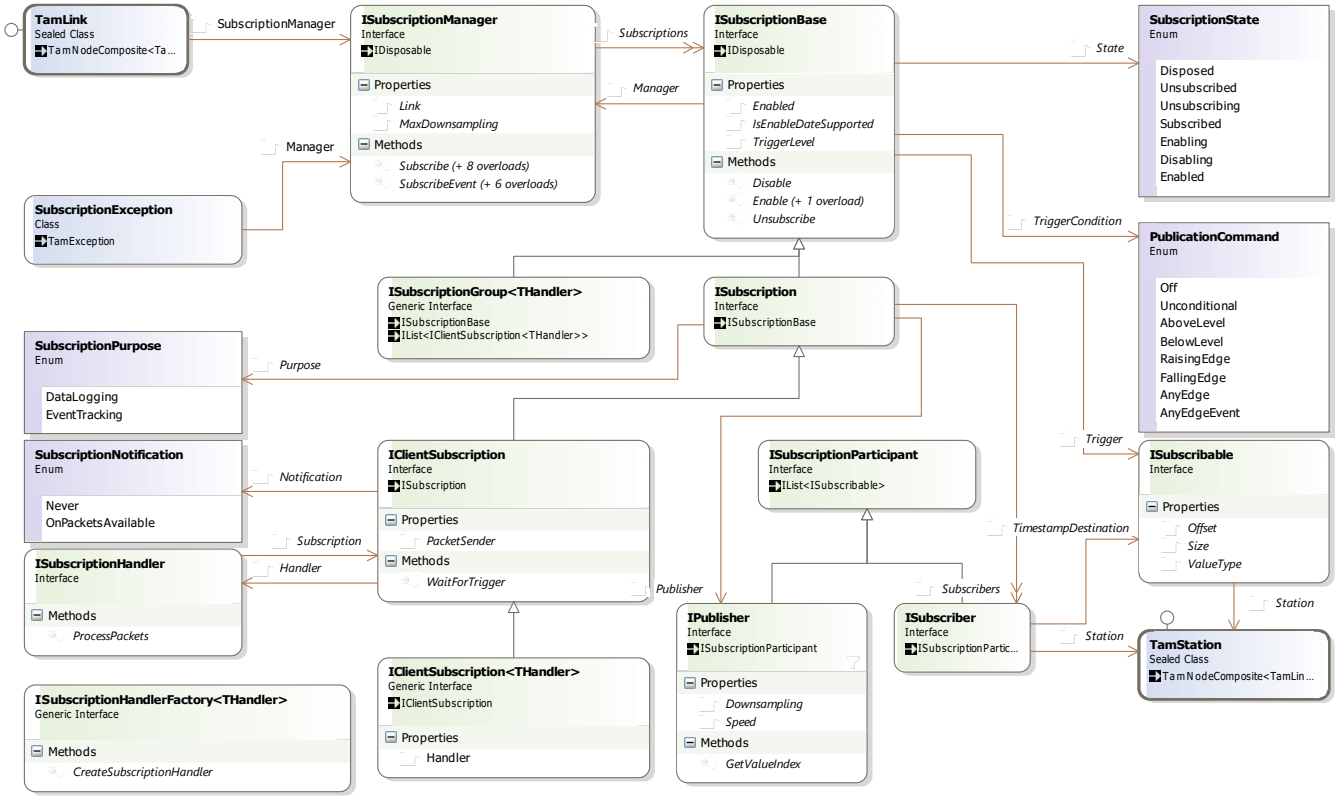

Figure 9: Subscriptions API

The TAM link exposes the subscription manager which allows to create new subscriptions among stations within the TAM link. To define which registers are to be published and where they are subscribed, publisher and subscriber participants have to be defined by the application. The publication rate, called "downsampling" and trigger conditions may be specified. A fast publication allows for transmission rates up to 100 kHz. For data acquisition, a polling or event driven model may be applied.

Life cycle

Subscriptions need resources on the Tria-Link stations, therefore both the number of publications and the number of subscriptions is limited on every station. Clients create subscriptions and must unsubscribe them in order to free the resources they occupy.

During the life cycle of a subscription, it can be enabled and disabled at the publisher as needed. Disabling a subscription keeps its resources reserved and avoids unnecessary traffic on the Tria-Link at times when the subscription values are not needed by the subscribers. Enabling of a subscription can be done with different trigger conditions (see below).

Down-sampling

For the optimization of traffic on the Tria-Link and CPU load on the PC, a *down-sampling* can be set individually for every publisher. The down-sampling denotes an integer multiple of the isochronous data rate at which subscription packets are sent. The default value is 1, typically corresponding to a 10 kilohertz rate. A down-sampling of 10 results in subscription packets to be sent 10 times less frequent than the isochronous data rate.

Fast Subscriptions

For the analysis of signals updated at a higher rate than the isochronous data rate, it may be necessary to publish them with their full time-resolution. This is possible by means of *fast* subscriptions. Fast subscriptions also use isochronous packets with up to 5 register values. But instead of the values of up to 5 different registers, it contains up to 5 values of one register sampled at different times. The time stamp of the packet corresponds to the last register value, and the time stamps for the other register values can be calculated from the data rate of the fast subscription.

Fast subscriptions must be considered as expensive resources that cause high traffic rates on the Tria-Link and high CPU load when handled in the PC. The typical application of fast subscriptions is for the optimization of controller parameters. At the opposite, they are not used for axes coupling applications, which affect the path planner and therefore are sufficiently served with *regular* speed subscriptions.

Some devices support *superfast* subscriptions, where two packets per isochronous sampling time are transmitted.

The `Speed` property of subscriptions is configured within an `IPublisher` instance.

Subscription Purpose

Subscriptions may generate heavy traffic on the Tria-Link. This could cause problems when the computer is among the subscribers, as control data could be lost within all the data logging values like a needle in a haystack. Therefore, like in the USB protocol, the Tria-Link protocol defines different endpoints. This way, data logging values arrive in another FIFO than other data.

Subscriptions may be set up such that values are sent to the normal endpoint. This is defined by the subscription's purpose which is only relevant for *client subscriptions*. Client subscriptions are subscriptions where the only subscriber is the computer.

Subscriptions created by the `Subscribe` methods of the subscription manager always have the *data logging* purpose. Massive data is expected and therefore, the values are sent to the second endpoint.

In contrast, subscriptions created with `SubscribeEvent` always have the purpose of *event tracking*. The values published by such subscriptions has control character and is therefore sent to the normal endpoint. In the following, such a subscription is referred to as *event subscription*.

Triggers & Events

When a subscription is enabled, a trigger condition can be set on an arbitrary register of the publisher. The trigger register must meet the trigger condition before the subscription starts to send isochronous subscription data. The following table gives an overview of the possible trigger conditions:

Trigger condition	Description
Off	Disables the publication.
Unconditional	Enables the publication without condition.
Above level	Enables the publication whenever the trigger signal is above the trigger level.
Below level	Enables the publication whenever the trigger signal is below the trigger level.
Raising edge	Enables the publication when the trigger signal crosses the trigger level with a raising edge.

Trigger condition	Description
Falling edge	Enables the publication when the trigger signal crosses the trigger level with a falling edge.
Any edge	Enable the publication when the trigger signal crosses the trigger level.
Any edge event	Sends a sample whenever the signal changes.

The last trigger 'Any edge event' has event-like character: it can be used to receive state changes from a device.

Caution Because data may be lost on the Tria-Link, it cannot be guaranteed that all events are always transmitted. Even when the error certainty is very small, for reliability reasons, it may be more secure to use an any edge trigger.

5.2 Setting Up Subscriptions

Given a TAM link having two drives with the same register layout. The subscription manager is property of the link.

```
ISubscriptionManager subscriptionManager = link.SubscriptionManager;
```

The goal is to set up a subscription between the two drives which transports path values. First, references to the register roots of the drives is helpful.

```
Register publisherRegister = (Register)link[0][0].Register;
Register subscriberRegister = (Register)link[1][0].Register;
```

The cast is needed because the `Register` property of the TAM Device is of the general type `RegisterComposite`, not of the register layout specific type.

Next, the subscription participants are set up.

```
ushort downsampling = 1;
IPublisher publisher = new Publisher(downsampling,
   // timestamp implicitly included
   publisherRegister.Axes[0].Signals.PathPlanner.Position.Float32,
   publisherRegister.Axes[0].Signals.PathPlanner.Velocity,
   publisherRegister.Axes[0].Signals.PathPlanner.Acceleration);

ISubscriber subscriber = new Subscriber(
   subscriberRegister.Axes[0].Signals.PathPlanner.PathValuesTimestamp,
   subscriberRegister.Axes[0].Commands.PathPlanner.Xnew.Float32,
   subscriberRegister.Axes[0].Commands.PathPlanner.Vnew,
   subscriberRegister.Axes[0].Commands.PathPlanner.Anew);
```

The passed registers form a one-to-one relationship. The down-sampling factor and high-speed arguments may be omitted in this case, as they have the default value. The publishing and subscribing stations are explicitly defined by the passed registers. Sanity checking is being performed to ensure that all passed registers belong to the same station.

Now, the subscription can finally be created.

```
ISubscription subscription = subscriptionManager.Subscribe(publisher, subscriber);
```

This completes the set-up of the subscription. Multiple subscribers could be specified, which would automatically constitute a subscriber group.

To have the station send values, just type

```
subscription.Enable();
```

Please note that this method has an overload which allows for specifying trigger conditions.

When the publisher should stop to send values, type

```
subscription.Disable();
```

The subscription may be enabled again, as long as it is not unsubscribed.

```
subscription.Unsubscribe();
subscription.Dispose();
```

Unsubscribing the subscription frees the resources on the publisher and the subscribers and disposes possibly constituted subscriber groups. The subscription must also be disposed to free allocated computer resources. If it is disposed but not disabled and unsubscribed, the publisher remains sending values to the subscribers.

The Gear Up! sample application is a fully functional application demonstrating this type of subscription.

5.3 Establishing Data Acquisition

In this section, a client subscription is set up for a given float register `r` using a down-sampling factor of `100`. The values are appended to a file with name `f`. An above level trigger ensures that all delivered values are greater than `0`.

```
IPublisher p = new Publisher(100, r);
IClientSubscription subscription = r.Station.Link.SubscriptionManager.Subscribe(p);

subscription.Handler = new DataLogger(subscription, new StreamWriter("log.txt"));
subscription.Notification = SubscriptionNotification.OnPacketsAvailable;
subscription.Enable(PublicationCommand.AboveLevel, r, 0);
```

The `DataLogger` class implements the handler of the delivered data logging packets, like this:

```
public class DataLogger : ISubscriptionHandler {
    private int valueIndex;
    private TextWriter logger;

    public DataLogger(IClientSubscription subscription, TextWriter logger) {
        Subscription = subscription;
        this.logger = logger;

        // cache the index within the delivered packets of the subscribed register
        this.valueIndex = Subscription.Publisher.GetValueIndex(Subscription.Publisher[0]);
    }
```

```
    public IClientSubscription Subscription { get; private set; }

    public void ProcessPackets() {
        Packet[] packets = Subscription.PacketSender.Dequeue();

        foreach (Packet packet in packets) {
            this.logger.WriteLine(packet.GetPacketData()[valueIndex].AsSingle);
        }
    }
}
```

The client subscription needs to be disposed at the end. Another aspect unconsidered in the example is exception handling. Subscribing and enabling may throw exceptions for different reasons.

5.4 Acquisitions

The acquisitions API (`Triamec.Tam.Acquisitions`) allows signals from devices to be saved in a buffer with predefined size.

A *variable* specifies what to acquire together with a desired sampling time.

An *acquisition* instance is created by passing a list of variables. The `Acquire` method takes a duration as argument. The data will be acquired in a synchronized manner.

When it returns, each variable contains the acquired data. These values are retrieved by iterating over the variable using a `foreach` construct. If using LINQ is an option, the `ToArray()` extension method is also handy. Timing information is obtained by the `StartTime` and `SamplingTime` properties.
There are situations where data will contain gaps. A variable without gaps is said to be *regular*. If the `IsRegular` property returns `false`, the variable is split into *segments* which are themselves regular.

The sampling time of a variable may be different from the desired sampling time due to rounding and transfer rate limitations.

It is possible to repeatedly acquire data without data loss between subsequent acquisitions. To enable this functionality, construct an acquisition passing a *prolonging* duration, that is, a maximal duration for which space must be reserved for acquired data.

Following a MATLAB script demonstrating what can be done.

```
% import all classes from the Acquisition namespace
import Triamec.Tam.Acquisitions.*

% create acquisition parameters without trigger and fastest possible sampling rate
samplingTime = TimeSpan(double(TimeSpan.TicksPerSecond)/50000);
sampleCount = 500000;
samplingDuration = TimeSpan.FromSeconds(samplingTime.TotalSeconds * sampleCount);

% create the variable to acquire, providing double values
variable = axisReg.Signals.General.ActualCurrentA.CreateVariable(samplingTime);

% subscribe a stimulus in order to start the acquisition (requires MATLAB R2010b)
trigger = TamTrigger(axisReg.Signals.PathPlanner.Done, ...
    PublicationCommand.FallingEdge, TamValue32.FromBool(true), ...
    Stimulus(@() MoveRelative(axis, 20)), TimeSpan.FromSeconds(10));

% do acquiring
```

```
variable.Acquire(samplingDuration, trigger);

% alternative to acquire multiple variables synchronized
% twoVars = NET.createArray('Triamec.Tam.Acquisitions.ITamVariable', 2);
% twoVars(1) = variable1;
% twoVars(2) = variable2;
% acquisition = TamAcquisition.Create(twoVars);
% acquisition.Acquire(samplingDuration, trigger);
% acquisition.Dispose;

% copy result
data = GetValuesFromVariable(variable);

% get actual sample time
d = variable.SamplingTime.TotalSeconds;

% plot
t = 0:d:d*(sampleCount-1);
plot(t, data);
xlabel('t [s]');

function MoveRelative(axis, distance)
    import Triamec.TriaLink.*
    disp('Issuing stimulus')
    axis.MoveRelative(Float40.FromDouble(distance));
end

function values = GetValuesFromVariable(variable)
 if variable.IsRegular
 values = double(NET.invokeGenericMethod('System.Linq.Enumerable', ...
 'ToArray', {'System.Double'}, variable));
 else
 error('Data contains gaps.');
 end
end
```

Another common pattern with prolonging acquisitions is to retrieve as much data as available in buffers in a loop, as depicted in this C# pseudocode.

```
double[] buffer = ...; int index = 0;
using (var acquisition = TamAcquisition.Create(timeLimit, variable1)) {
    while(true) {
        acquisition.Acquire(TimeSpan.Zero);
        foreach(var sample in variable1) {
            buffer[index++] = sample;

        }
        OtherWork();
    }
}
```

The call to `Acquire` passes a zero duration, which means in case of a prolonging acquisition to return *at least* zero samples.

6 Motion

Before moving an axis with a drive (section 6.2), some setup has to be done (section 6.1), and in order not to miss the end of the move, the request technology comes in (section 6.3).

All motion related units are basically *SI units*, avoiding confusions. Positions are always relative to the encoder. With a rotary encoder, all positions are to be interpreted in radian, where linear encoders imply meters as unit.

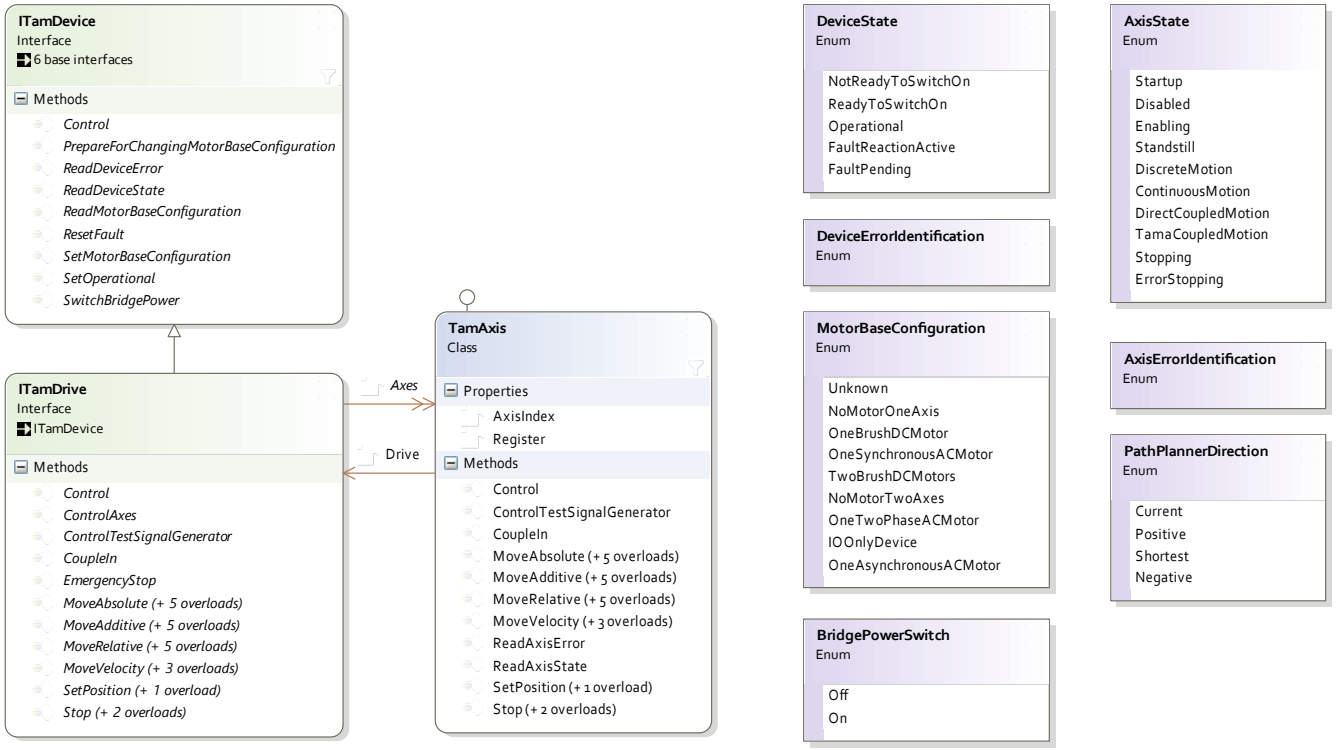

Figure 10: TAM drive elements

A TAM Device might be a drive. Drives can be set operational by turning on the bridge power. Then, its axes (usually they have exactly one of them), may be enabled by the control methods, and moves may be executed. State register read out convenience methods are provided.

6.1 Drive

Figure 10 introduces drive and axis in the topology object model, showing the motion related methods and related enumerations only. Some methods are defined on both the drive and the axis object.

A method defined on the drive affects all axes, while the equivalent method on the axis affects only the

respecting axis. `MoveAbsolute`, for example, called on a drive object, will cause all axes of the drive to move to the specified position. If a drive has only one axis, the behavior of the two sets of methods is identical.

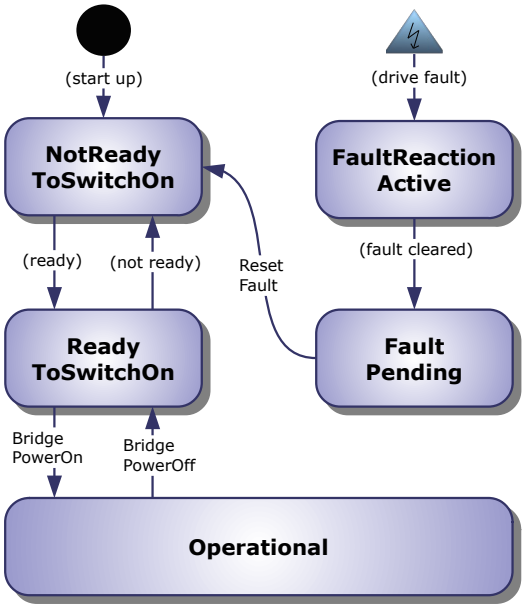

Figure 11: Device State Machine

After a power cycle, the device is not ready to switch on for a short amount of time. When it is ready to switched on, it may be set in the operational mode. When an error occurs, the device stays in the fault reaction active state until the error condition goes away. The error must then be acknowledged before it can be set operational anew.

Figure 11 shows the *state machine* implemented by all devices, although it is originally designed for drives. Some devices may never reach some states, for example the `Operational` state.

The state is read using the `ITamDevice.ReadDeviceState` method, which is just a convenience function hiding a register read operation.

At start-up or reset, the device needs to be initialized, for example, its registers. This is the state `NotReadyToSwitchOn` from the `DeviceState` enumeration, located in the `Triamec.TriaLink` namespace. The application has to wait as long as the device is in that state.

Before continuing, the *motor base configuration* needs to be correct. Principally, this is a special read-only register setting up fundamental drive firmware internal parameters. It also has an influence how many axes the device drives. It can be read out with `ITamDevice.ReadMotorBaseConfiguration` and set with the `SetMotorBaseConfiguration` method.

Note The above is only relevant for old generation drives. For new drives, the motor base configuration is an ordinary register parameter, one per axis, named `Motor/Type`. The mentioned APIs will be deprecated.

To be able to move, the bridge power must be switched on with the `ITamDrive.SwitchBridgePower` method. At the end, the same method is used to switch bridge power off.

When an error occurs, the device spontaneously transitions into state `FaultReactionActive`, and, when the fault disappears, into `FaultPending`. This forces applications to check for the error and explicitly reset the fault (using the method `ITamDevice.ResetFault`). The error is retrieved using

`ITamDevice.ReadDeviceError`. An example of such a fault is when the temperature rises over a specified limit.

Note `ReadDeviceError` can also show a warning when the drive is in state `NotReadyToSwitchOn`.

Note For older devices, the `ResetFault` transition goes from `FaultPending` directly to `ReadyToSwitchOn`.

6.2 Axis

The `ITamDrive` interface offers an `Axes` collection with element type `TamAxis`.

Caution For devices supporting the `SetMotorBaseConfiguration` command, when the motor base configuration changes, the axes instances are disposed and new ones created. Therefore, it may be inopportune to cache references to the axes instances.

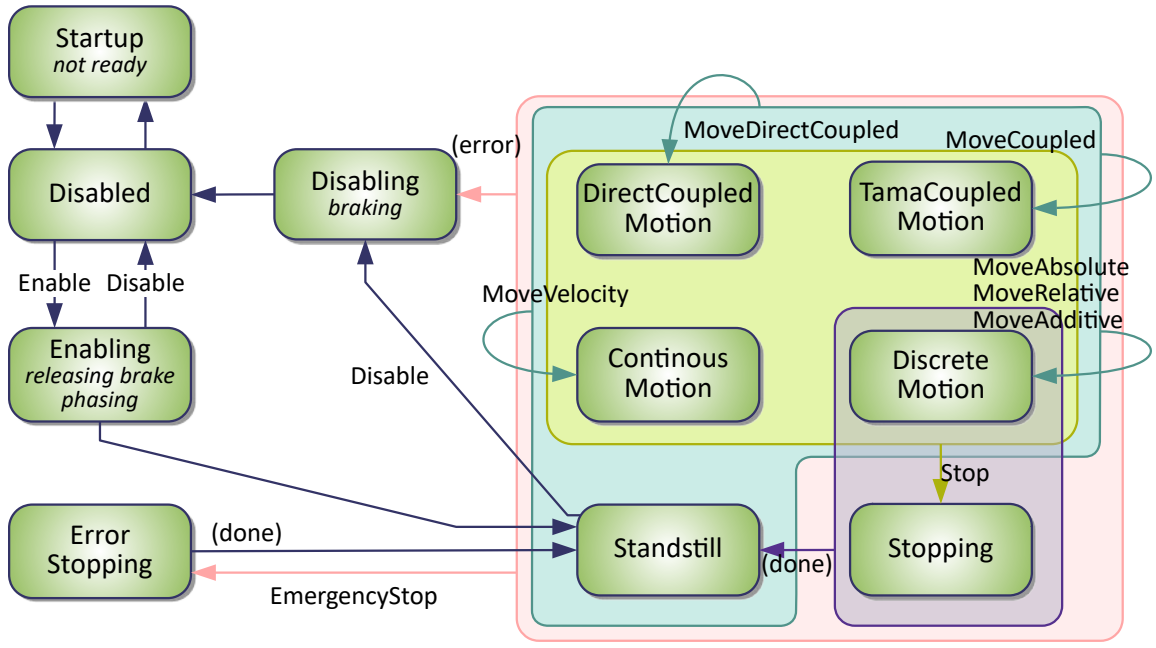

Figure 12: Axis State Machine

As soon as the axis is ready, it may be enabled. After this potentially long running process, the axis is in standstill. Using the move method, different motion modes can be established. Reprogramming is possible, when realized in real-time by a Tama program, within 100 µs. At any time when moving, the axis may be stopped, which may need some time to slowdown. In coupled motion, the path is received by a subscription which needs to be set up separately.

When the drive's bridge power is switched on, the axis is still inordinate, that is, disabled. The position and current controllers have to be enabled, a motor brake has possibly to be released, and an angle search algorithm may be executed. This procedure may last for several seconds. Enabling and disabling

the axis is done by the `Control` method, which additionally allows acknowledging axis errors.

Similar to the drive's states and errors, the axis' state and error are read out using the `ReadAxisState` and `ReadAxisError` methods. The state diagram is shown in figure 12.

A *path planner* is responsible to force the axis to move. It is controlled by a *PLCopen* based API. The following excerpt shows a part of the API.

```
TamRequest MoveAbsolute(Float40 position, PathPlannerDirection direction);
TamRequest MoveRelative(Float40 distance);
TamRequest MoveAdditive(Float40 distance);
TamRequest MoveVelocity(float velocity);
TamRequest Stop(bool emergency);
TamRequest SetPosition(Float40 position, SetPositionMode mode);
TamRequest CoupleIn(bool direct);
TamRequest TorqueControl(float torque, float torqueRamp);
```

Most of these methods have additional overloads in order to specify direction, velocity, acceleration and deceleration. For a different jerk, however, the maximum jerk register has to be changed.

The move API is doubled in the `ITamDrive` interface, allowing for commanding two axes of a single drive at once.

The calls return immediately after acknowledgment was received from the drive. See chapter 6.3 how to wait for the termination of move commands.

Movement takes place in six different states:

- *Discrete motion*.
  The axis is moved by the specified relative distance or to an absolute position.
- *Continuous motion*.
  A constant velocity, torque or force is applied forever. This mode has to be used with special care. No assumptions have to be made about when the PC could reliably stop the axis again.
- *Direct coupled motion* and
- *Tama coupled motion*.
  Move synchronously to another axis. See section 6.4 for more details.
- *Stopping* and *ErrorStopping*, respectively.
  Bring velocity to zero. These states are not reprogrammable.

The parametrization of the path planner and the position and current controllers is not in the scope of this document.

6.2.1 Control System Treatment

Most drives get integrated into a real time control system. Accessing them via TAM API is considered a secondary use case.

Using the `TamAxis.ControlSystemTreatment` property and the `Override` method, tell the axis that you're going to take control:

```
class ControlSystemTreatment {
    bool IsSupported;
    void Override(bool enabled);
    bool GetIsOverridden()
}
```

When not overriding the control system, the axis might reject your commands.

You should not do this, though, when your application is about to access the drive via the PCI interface.

All new generation drives with current firmware support this command. Use the `IsSupported` property if your application needs to support legacy devices.

6.3 Requests

The procedures described in the last sections may be *long-running*. However, the methods commanding them just wait for an acknowledgment and then return control to the caller. It is always in the caller's responsibility to detect the end of the procedure, be it enabling or a movement.

There are two principal possibilities to detect the end of a procedure, polling or events. Polling might be implemented by regularly reading out drive and axis state.

To implement an event driven model, the TAM API offers the *request* infrastructure with the `Triamec.Tam.Requests` namespace.

The starting point is the return value of the `MoveAbsolute`, `Control` and other methods, which is of type `TamRequest`. The request object stands for the issued request and the process it initiates. It can be used to wait for the termination of the request with the `WaitForTermination` method. If the method returns without timeout, check the `TamRequest.Termination` property for successful termination of the request. Figure 13 shows the involved classes.

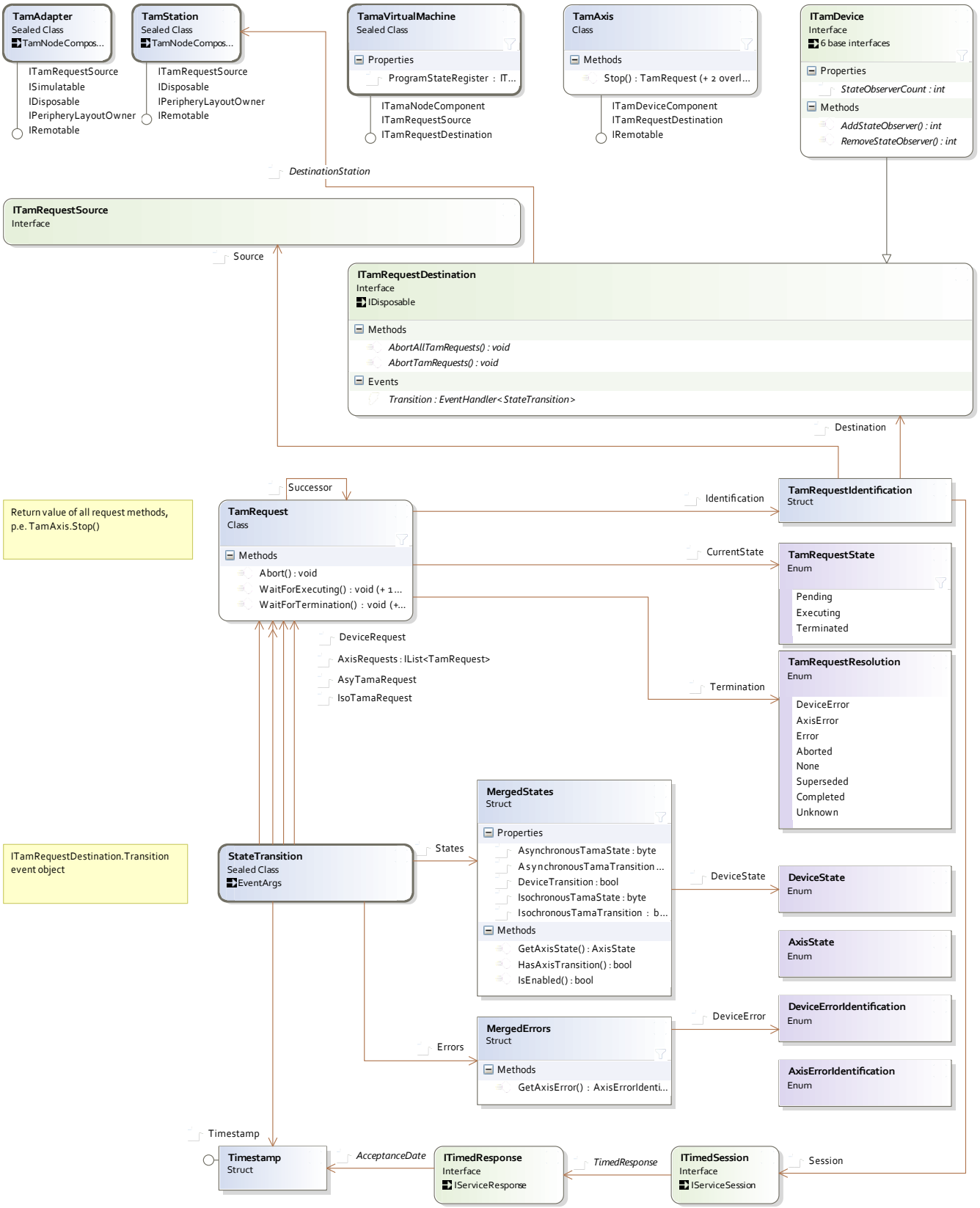

Figure 13: Request API.

Methods such as `TamAxis.MoveAbsolute` return immediately. In order to wait for the end of the commanded move, use the `WaitForTermination` method of the returned `TamRequest` instance.

You will typically just use the `WaitForSuccess` and `WaitForSuccessAsync` convenience methods which allow checking termination and success in a one-liner.

The following listing outlines a method which commands a given axis to move to a specified position. The method does not return before the movement has terminated, given the movement does not last longer than a given duration `maxTimeoutMilliseconds`. It returns `true` when no timeout occurred and the movement was successfully executed.

```
TamAxis axis;
int maxTimeoutMilliseconds;

void MoveTo(float position) => axis.MoveAbsolute(position).WaitForSuccess(maxTimeoutMilliseconds);
```

Possible reasons a request could not be successfully completed include axis errors or another request reprogramming the path planner.

Internally, an event subscription is used to track the life cycle of the request by means of the drive or axis state machine. To be careful with resources, the internally maintained event subscription is only set up if requested by the application. This is done by the `ITamDevice.AddStateObserver` and `RemoveStateObserver` methods, or when a `Transition` event is subscribed (Available in `ITamDevice`, `TamAxis` and `TamaVirtualMachine`).

Caution These methods and events implement the *reference counter* pattern. Thus, it is critical to always call the pairs of methods (add/remove) in a strictly symmetrical manner. Otherwise, internally acquired resources might not correctly be disposed of.

The *transition* event can be subscribed in order to observe the drive or axis state machines. As the name proposes, the event is also fired when a transition from a state to itself occurs.

The Device State Observer concept paper elaborately describes the internals of the request mechanism ([10]).

Reliability Considerations

The request mechanism is vulnerable to communication failures, as there is no guarantee for events to be delivered over the Tria-Link. Therefore, production quality code should always call the `WaitForTermination` method specifying a timeout. The `TamRequest.Abort` and the `AbortTamRequests` and `AbortAllTamRequests` methods defined on `ITamDevice`, `TamAxis` and `TamaVirtualMachine` instances allow for re-synchronization in case of communication loss. State observation should not entirely rely on the transition events but allow for forced re-synchronization with the current state.

6.4 Coupling

Coupling is the synchronization of an axis using to a stream of incoming isochronous data using the special path planner modes `DirectCoupledMotion` and `TamaCoupledMotion`.

The axis may receive data from

- TwinCAT – this is out of the scope of this document.
- the control system using Direct Feed. Refer to the Direct Feed developer sample.
- drive-to-drive data exchange. Refer to the respective Application Note [8].

Coupling can be *direct* or preprocessed.

In direct coupled motion, the path planner takes the subscriber values as input.

In Tama coupled motion, position, velocity and acceleration (the *triple*) are taken from three sibling registers, typically called `XNewCoupled`, `VNewCoupled` and `AnewCoupled`. A Tama program (see section 7) defines a coupling function taking the incoming triple from the master and calculating the input triple for the path planner.

The API

```
TamRequest CoupleIn(bool direct);
```

brings the path planner in coupled motion. The triple must be feasible regarding current motion, or an error will occur.

Coupling ends with the next path planner command, for example `Stop`.

6.5 Schedules

A schedule is a table of Tria-Link commands paired with relative time information to be sent over the Tria-Link one by one in real-time.

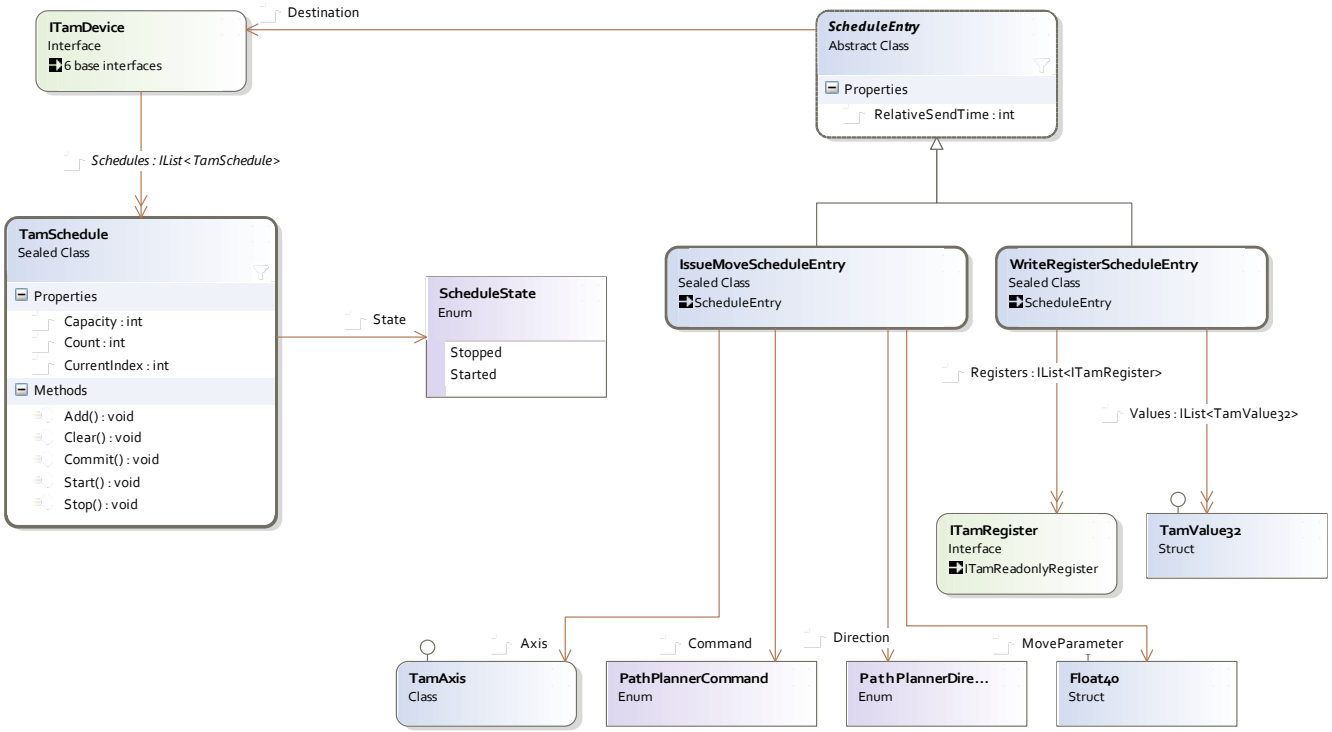

Figure 14: Schedule API

A Schedule is a list of Tria-Link commands, to be sent in real-time at specified relative times. The Schedule can contain move and register write commands. The commands can be sent to any station within the Tria-Link of the scheduling TAM Device. Schedules are maintained within registers and make use of the commit mechanism (section 4.2).

The Schedule is saved within the register layout of a TAM Device implementing the scheduling functionality. Further on, the commit mechanism (section 4.2) is used to update Schedules transactionally based.

Note Currently, only TD-Bus hosts implement Schedules. However, future implementations of other TAM Devices may contain this functionality, too.

A *Schedule entry* consists of one Tria-Link communication packet and a *relative send time* information, in milliseconds, when the communication has to take place. Currently, the software supports only move and register write commands to be saved within a Schedule entry, but other communication will not be used with Schedules, anyway. The destination station of the Schedule needs to be within the same Tria-Link as the device where the Schedule resides.

If an entry needs to be sent to multiple stations, either a group needs to be established, or the entry needs to be doubled.

It is not allowed to write the same entry instance twice in one or multiple Schedules.

Typically, a Schedule has only very few entries because exception handling is complex. The application does not have any information about whether commands have been accepted and executed, because no responses are sent by the destination stations.

The entries have to be added to the Schedule sorted by the relative send time. A Schedule can be com-

pletely cleared, but individual entries may not be removed. All those modifications have to be completed with a *commit*. This allows for preparation of a new Schedule during the execution of the same Schedule (in memory).

The Schedule can be *started* and may be *stopped* in the middle of processing. The relative send time information in the entries correspond to the point in time when the Schedule was started. The TAM scheduler device checks in each cycle whether the next entry in each of its Schedules has to be sent. Therefore, if a schedule has the same relative send time or even an earlier send time than its predecessor entry, it will be sent too late, that is, in the next cycle of the scheduler task.

The last sent entry can be read out by the *current index*. Committing a started Schedule immediately stops it.

6.5.1 Scheduling Example

This example demonstrates the usage of the Schedule API in comprised form. It assumes that the axis is already parametrized and enabled such that it is able to execute a move request. The referenced station names need to exist.

```
ITamDevice scheduler = Link["Scheduler"][0];
ITamDrive drive = Link["Drive"][0] as ITamDrive;
```

The task is to write a specific register on the scheduler device and, *after exactly half a second*, move on the specified drive.

A register of the scheduler itself is found using a concrete register layout type. The schedule is assumed to be empty.

```
TamSchedule schedule = this.scheduler.Schedules[0];
ITamRegister<float> register = ((Register)scheduler.Register).Tama.Variables.GenPurposeVar0;
schedule.Add(new WriteRegisterScheduleEntry(
    new ITamRegister[] { register }, // maximal 4 registers are allowed
    new TamValue32[] { 4.0f }, 0));
```

The move is prepared by specifying the desired axis to move with. Then, the Schedule is committed and started immediately.

```
schedule.Add(new IssueMoveScheduleEntry(drive.Axes[0], PathPlannerCommand.MoveRelative,
    PathPlannerDirection.Positive, 5.0f, 500));

schedule.Commit();

schedule.Start();
```

7 Real-Time Programming with Tama

With *Tama Programs,* the drive firmware can be extended with customer specific implementations running at up to 10kHz real-time.

Refer to the reference [9], specifically the API chapter within the advanced topics.

8 Configuration

The *TAM Configuration* mechanism allows for host-side persistence of the parameter registers, the naming within the topology, references to Tama programs, and module configuration.

Technically, the configuration framework builds on top of the .NET XML serialization framework. That is, a TAM Configuration is an *XML* file with a layout defined by the classes in the `Triamec.*.Configuration` namespaces, resembling the TAM Topology.

Each register layout defines which registers need to be persisted. A register which will be persisted needs to have read/write access and a special persistent flag set (shown by the `IsPersistent` property of the `IRegisterComponent` interface).

For the correct relationship between persisted data and hardware instances, hardware serial numbers and other identifiers are included. If at loading time, some persisted data does not match local hardware, the configuration will not be loaded.

The registers and also Tama programs may be persisted directly on the TAM Device, when set up in stand-alone mode. For more information, please read the respective chapter below.

8.1 Configuration API

The easiest way to load and save configurations is offered by the `ITamNode` interface through two methods, `Save` and `Load`, both taking a path string as argument. Only the information of the instance together with all children is saved, however enriched with the hierarchical information of all ancestors.

To gain more control, the `TamSerializer` and `TamDeserializer` classes are provided to refine save or load options, respectively.

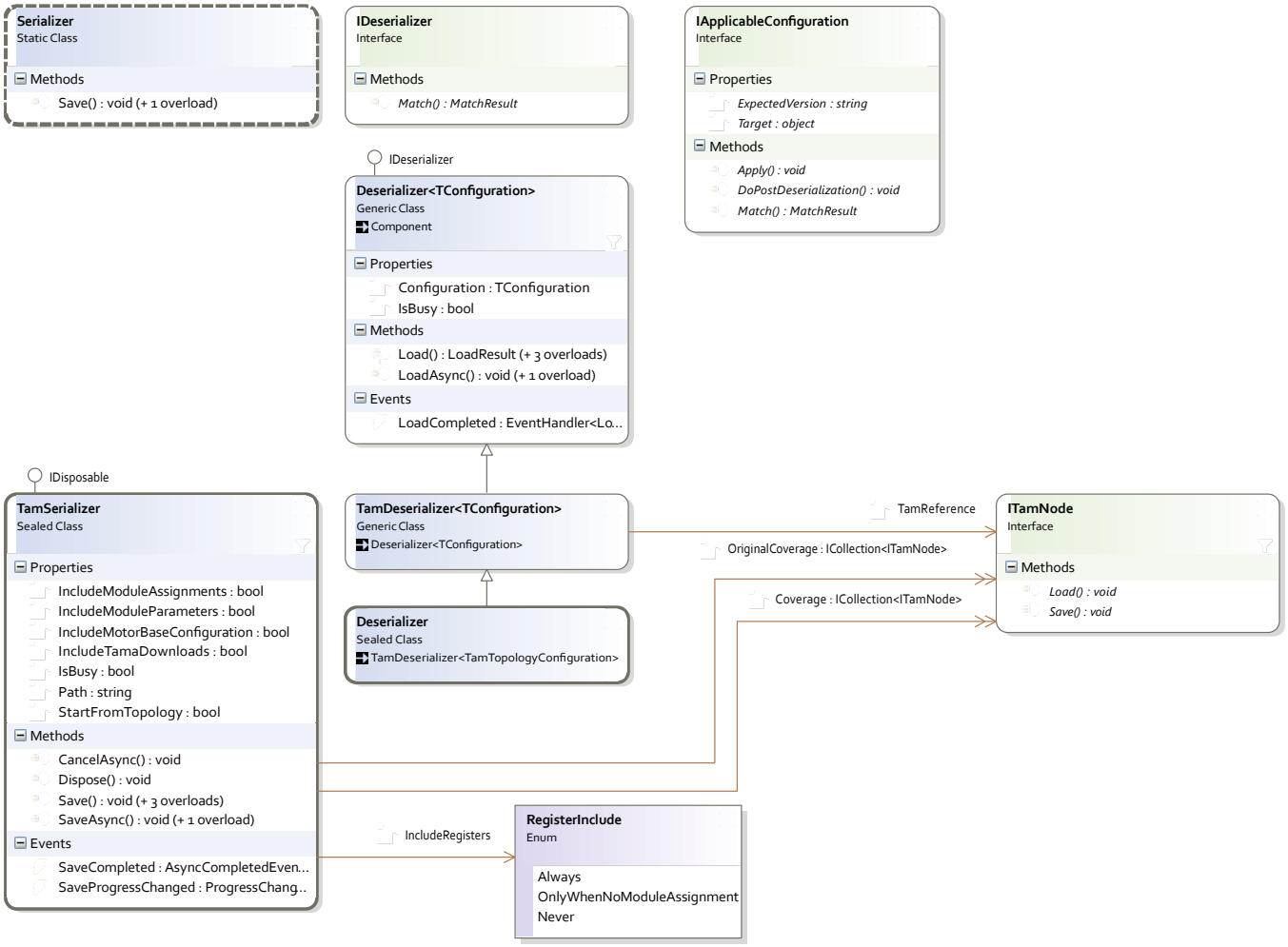

Figure 15: Basic TAM Configuration API

There is an upper layer which describe configurations independently from the TAM system, and a lower layer which is specific to the TAM system. Classes from the lower layer inherit from the upper layer.

The most convenient way to save and load configurations is by calling the respective methods of any TAM node. To gain more control, the `TamSerializer` *and* `TamDeserializer` *classes are provided. Among other features, this enables applications to embed TAM Configurations within their own XML configurations, detect hardware changes (see section 8.2) and save and load partial configurations.*

The serializer has the following features:

- A stream may be specified instead of a file name. One use case where this comes in handy are applications which embed TAM Configurations within their own XML configurations.
- A custom *coverage* over TAM nodes may be defined. That is, a configuration may be set up to contain only one TAM Device, or only some parts of the register tree of one TAM Device. Covered nodes are always saved with all of their child nodes.
- An option allows specifying whether to *start* the configuration hierarchy *from the topology* root, independently of the configured coverage. This property is set per default, such that the XML files all start with the top level hierarchy. While this setting restricts any persisted registers to be loaded to another TAM Device, at the same time, it also guards a user from applying a configuration to a wrong device.
  When the option is deactivated, the configuration's root is defined as the *nearest common ancestor*

node of all nodes within the coverage. An application which wants to save the TAM Configuration per TAM Device can just create the serializer with a device as single coverage node and this setting disabled.
The setting may be passed such that not the topology is defined as root. This can still be overwritten by the `StartFromTopology` setting.

```
bool startFromTopology = false;
using (var serializer = new TamSerializer(startFromTopology, this.drive)) {
    serializer.Save("MyDrive.xml");
}
```

- A couple of include properties, all set per default, allow omitting certain parts of a TAM Device configuration, namely Tama program location, motor base configuration, registers and module parameters.

When deserializing, the coverage is implicitly defined by the TAM Configuration. Per default, anything found in a configuration is applied. The deserialization process concentrates more on whether it succeeds to *match* all configuration with the correct device. The procedure divides into three stages:

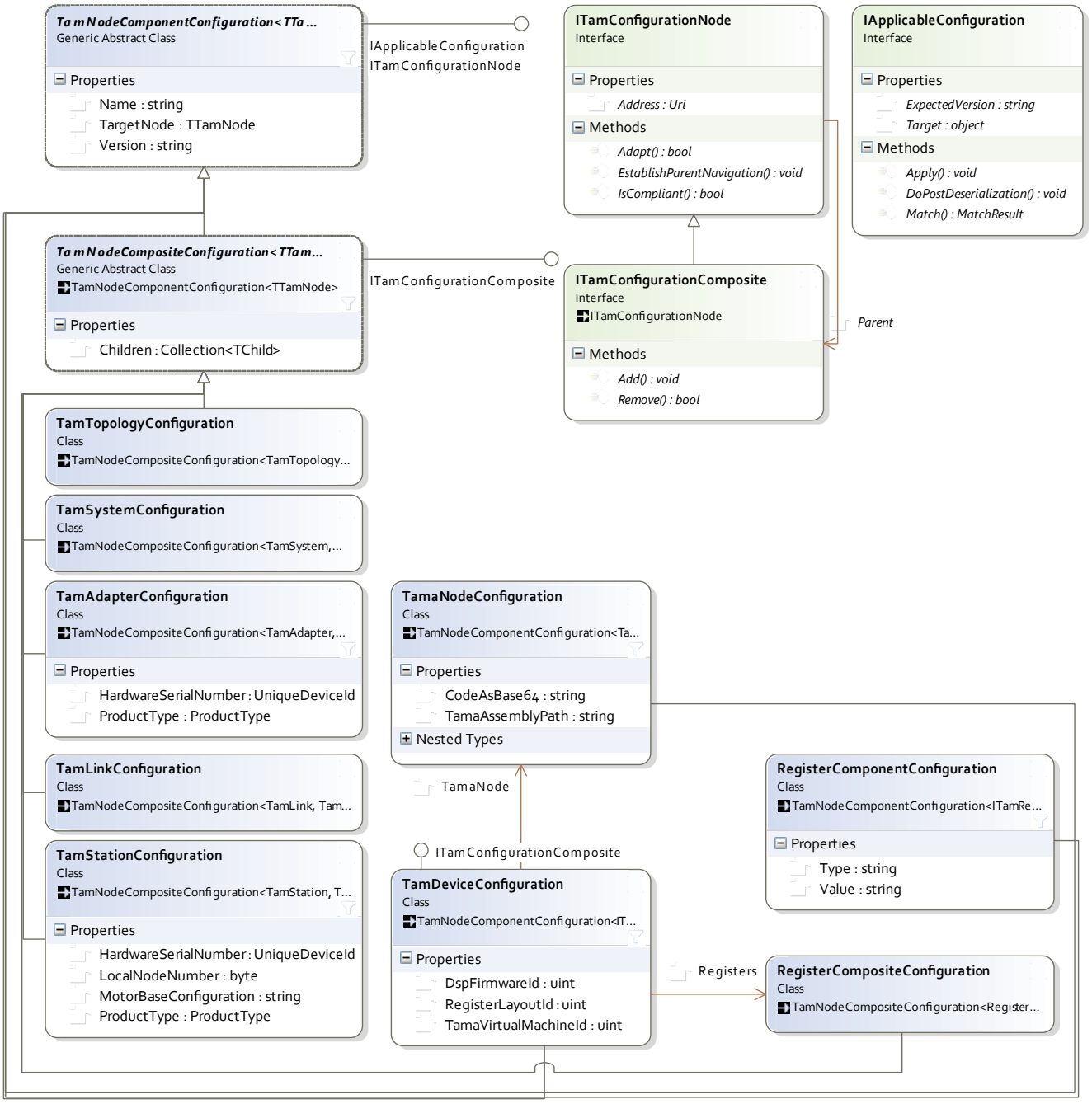

Figure 16: TAM Configuration hierarchy

The configuration hierarchy is parallel to the TAM Topology hierarchy (Figure 4). One difference is the flattened register tree. This hierarchy directly defines the XML format of the TAM Configuration as defined by the XML serializer of the .NET framework.

1. *Load* the configuration from a file or using a specified XML reader. This method is called directly on the deserializer object and returns a load result. It parses the XML file into a hierarchical data structure resembling the TAM Topology, but only containing configuration data.
2. *Match* the configuration tree against the TAM Topology. Anything found in the configuration must have a relation to a real counterpart. Additional information is saved with the configuration to re-

produce this relationship.
If a part of the configuration cannot be related, a *mismatch* is added to the match result and the configuration cannot be applied.

3. *Apply* the configuration to the TAM Topology. All registers found in the TAM Configuration is written and committed, the topology nodes are renamed according to the configuration, and so on. This step can only be invoked from a match result without any mismatches.

Figure 17 shows the classes involved in this process. For a complete deserialization example, see the section below.

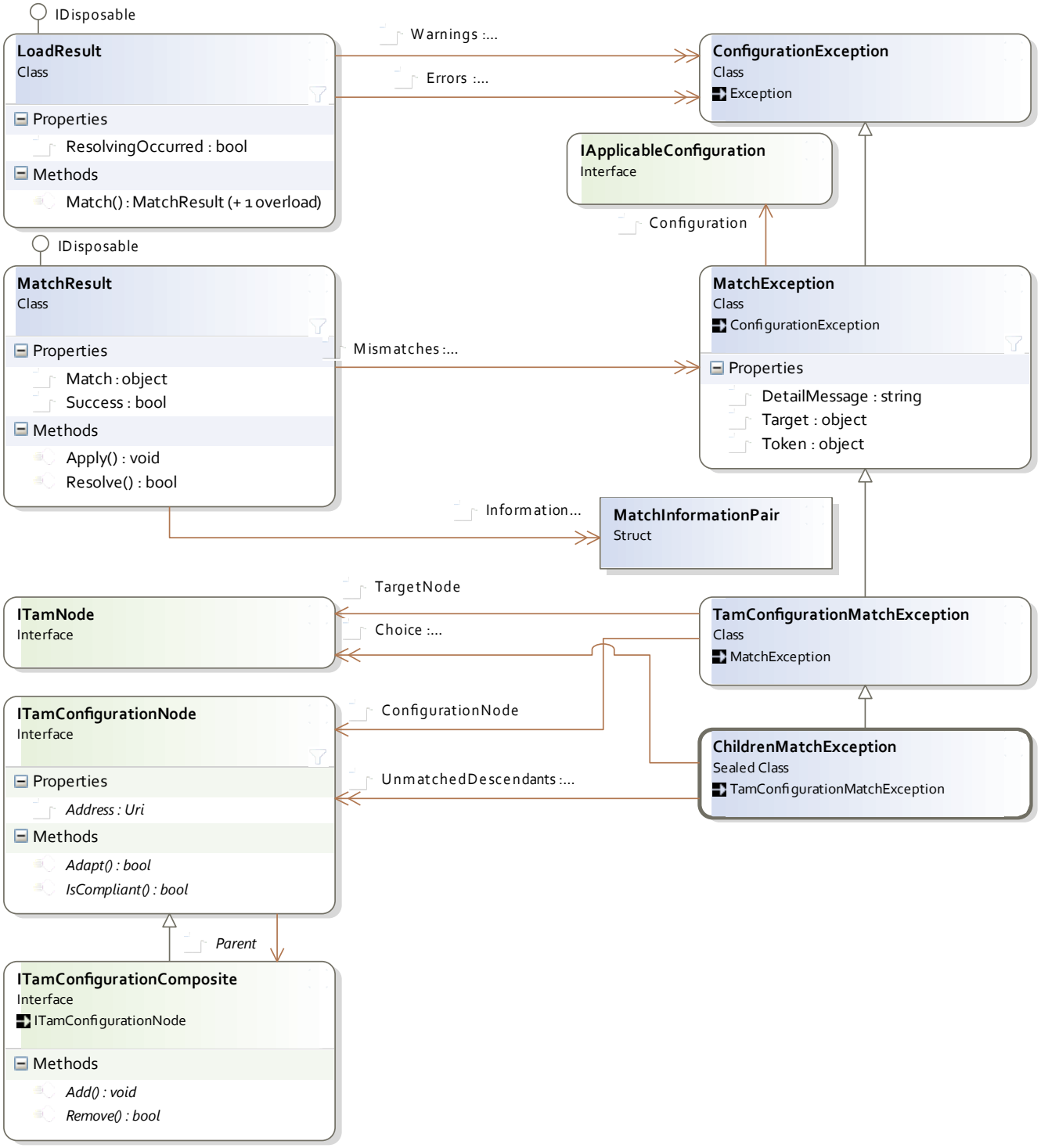

Figure 17: Configuration Loading and Errors

As in figure 15, there are two layers, the more general layer independent from the TAM system, and the other one. The `Match` and `Apply` methods have asynchronous counterparts, not being shown for convenience. The `LoadResult` class is returned using the `Deserializer` classes, while the `MatchResult` class is the result of the `LoadResult.Match` method.

The `SaveDialog` and `LoadSurveyor` classes provide GUI functionality. While the latter just shows a progress window, the `SaveDialog` lets the user choose from the different options offered by the seri-

alizer.

8.2 Mismatch Resolution

As described in the above section, a TAM Configuration can only be applied if there are no mismatches. Many use cases afford a less restrictive policy, though. Such policies need to be implemented by the application by modifying the TAM Configuration instance tree, as shown in figure 16.

By adapting the TAM Configuration tree to the actual system, mismatches can be eliminated. After this readjusting step, a new match can be tried.

In order to automate this *resolving* step, the `Triamec.Tam.Configuration.Resolving` namespace is introduced. Custom implementations of `IResolver` can be registered using the `Configuration-Resolver`. This may include update wizards or factory setup procedures.

When `MatchResult.Success` is false, call `MatchResult.Resolve()` and check the return value whether all mismatches could be resolved. `MatchResult.ResolvingOccurred` also indicates whether the configuration has changed in memory and might need to be persisted by the application.

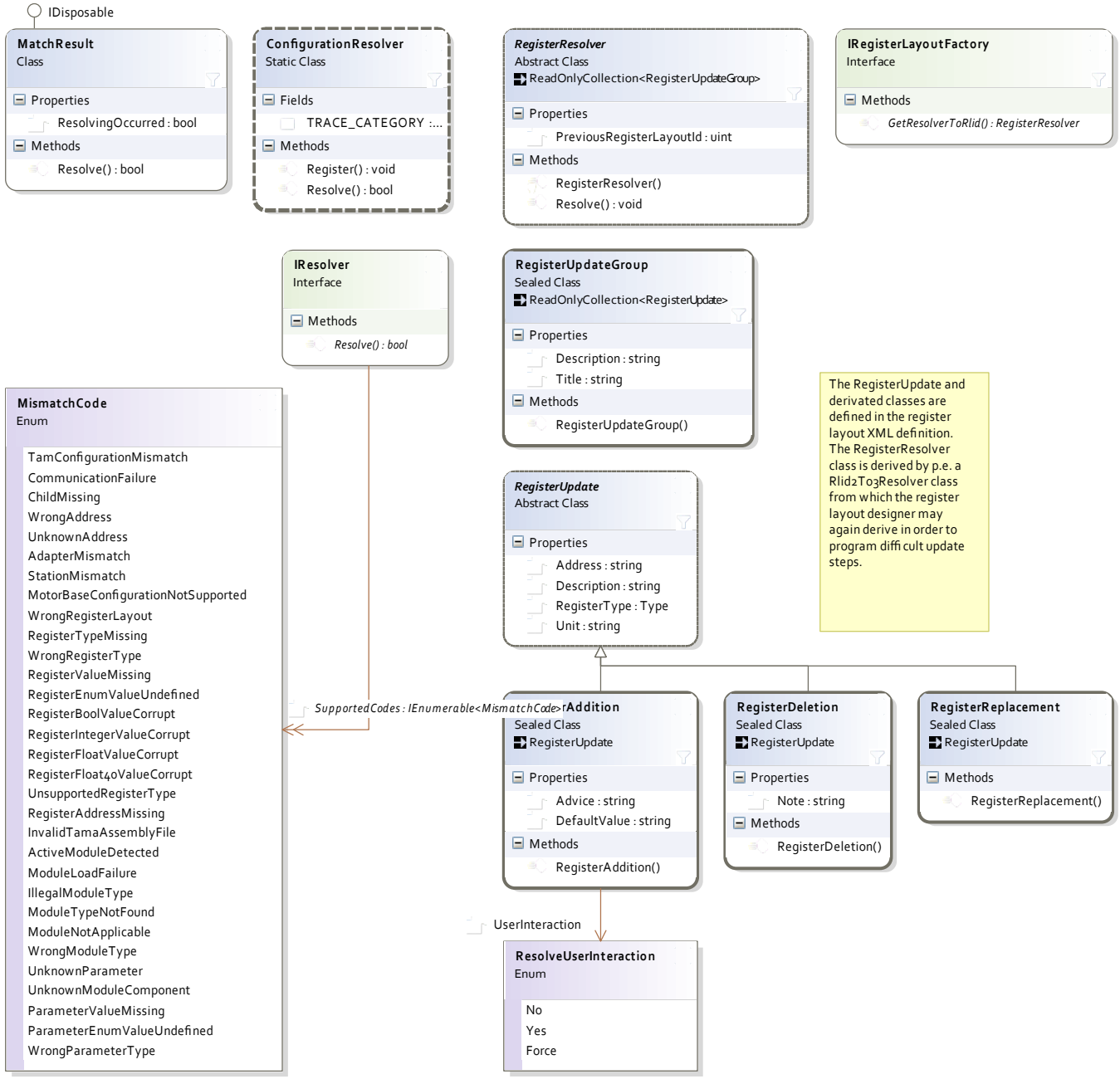

Figure 18: The Resolving Framework

TAM Configuration mismatches are updated using resolvers, which are registered to the `ConfigurationResolver`*. For new register layouts, update resolvers are authored from within the XML layout declaration. The above register related classes are then automatically generated.*

The following example assumes a booted TAM Topology `topology`.

```
using(Deserializer deserializer = new Deserializer()) {
    LoadResult loadResult = deserializer.Load("My config.xml");
    if(loadResult.Errors == null) {
        MatchResult matchResult = loadResult.Match(topology, topology);
        if(!matchResult.Success) {
          foreach(TamConfigurationMatchException mismatch in matchResult.Mismatches) {
            ChildrenMatchException childEx = mismatch as ChildrenMatchException;
            if (childEx != null) {
```

```
                for (int configIndex = 0; configIndex < childEx.UnmatchedDescendants.Count;
                    configIndex++) {

                    if (configIndex < childEx.Choice.Count) {
                        TamStationConfiguration stationConfig = childEx.
                            UnmatchedDescendants[configIndex] as TamStationConfiguration;

                        TamStation station = childEx.Choice[configIndex] as TamStation;
                        if((stationConfig != null) && (station != null)) {

                            // change unique keys
                            stationConfig.Type = station.HardwareIdDetails.Type;
                            stationConfig.HardwareSerialNumber = station.HardwareSerialNumber;
                            stationConfig.LocalNodeNumber =
                                station.HardwareIdDetails.LocalNodeNumber;
                        }
                    }
                }
              }
            }

            // try again
            matchResult = loadResult.Match(topology, topology);
        }
        if (matchResult.Success) {
           matchResult.Apply();
        }
    }
}
```

This code automatically detects when a TAM station was exchanged and relates the configuration of the old station to the new one. Please note that the TAM Configuration XML is left unchanged in this example. The example could also have been implemented using an `IResolver`. This simple task is left as an exercise to the reader.

9 Stand-alone Mode

Some applications afford that the TAM Device is stand-alone. That is, it is not connected to a Tria-Link ring, and the device needs to start up, configure itself and do Tama program tasks.

9.1 Start-up Process

When powered on, the device checks its start-up settings. If stand-alone start-up is active, the device configures itself according to the following settings:

- *Use local clock*
  Per default, the Tria-Link delivers clock information to the device for synchronization. With this flag set in persisted memory, the device generates its own clock. This makes it possible to operate the device without attached communication cables and without computer nearby.
- *Use static addresses*
  Each station in a Tria-Link has a number of byte addresses used for communication. One of it is the unique *station address*, three *group addresses* and the predefined broadcast address.
  If this flag is set, the addresses are read from the respective start-up settings stored in persistent memory.
  This feature can be used to set up a network of stand-alone devices communicating with each other. For example, paths could be transmitted in order to couple one drive to another.
- *(no setting)*
  The parameters, subscription information and Tama code memory – all parts of the register layout – are loaded from persistent memory into volatile register memory.
  This is not done unconditionally. The register layout and virtual machine IDs are persisted together with the registers. At start-up, these IDs need to match the firmware's RLID and VMID. This might not be the case after new firmware was downloaded (see chapter 10).
  This functionality ensures the correct parametrization of a device, establishes publishing and subscribing register values and prepares a Tama program.
- *Start the isochronous Tama virtual machine*
  As described in chapter Error: Reference source not found, Tama programs are not automatically executed. With this option specified, the isochronous virtual machine is taken to be operational.
- *Start the asynchronous Tama virtual machine*
  This setting has the same functionality as the setting described above.
  With the isochronous or asynchronous task running, custom start-up actions can be executed. For example, an axis could be enabled and set into coupled movement.

Another use-case is the conditional execution of different tasks dependent on some I/O.

With all of these options together, the device receives vast autonomy which might draw a computer superfluous in a production environment, lowering costs and complexity.

9.2 Start-up API

The settings described in the above chapter are controlled and checked by *start-up* commands, defined in the Tria-Link protocol. The TAM API abstracts from these commands and offers different methods through the `ITamDevice` interface, as seen in figure 19.

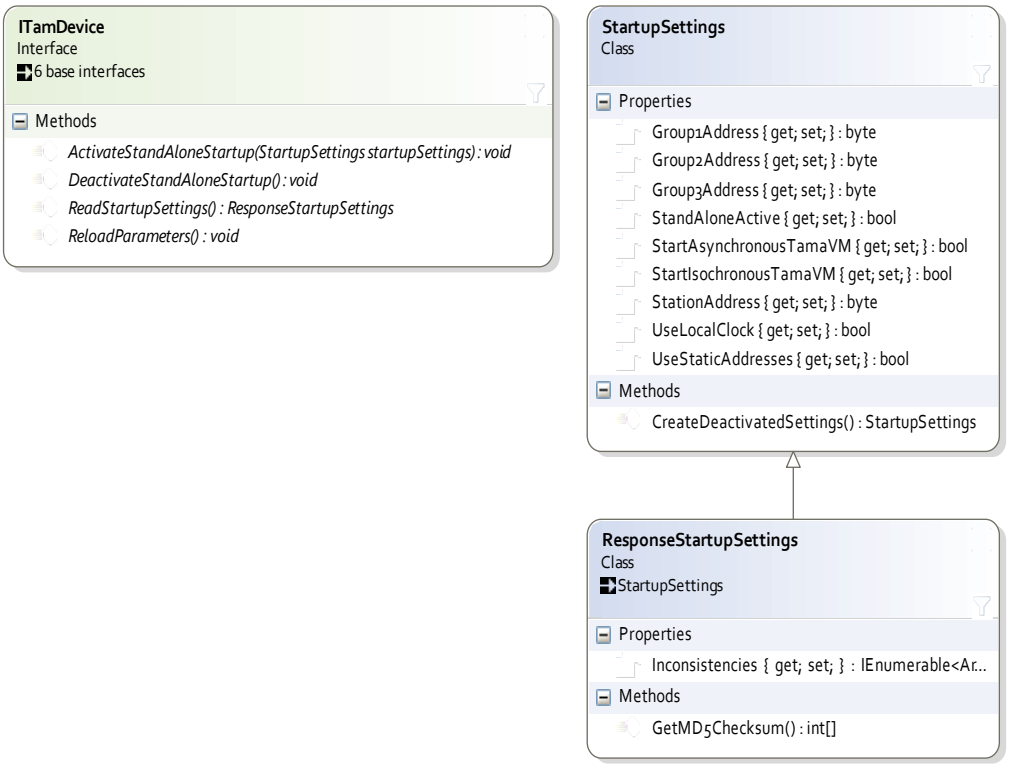

Figure 19: The Start-Up API

The device tells whether it supports the stand-alone functionality. If it does so, stand-alone mode can be established with the `ActivateStandAloneStartup` method specifying the desired start-up settings. Another method is provided to deactivate stand-alone mode again. The current settings may be queried and the persisted parameters (including Tama program) may be reloaded from persistent memory, overwriting all recent changes to the parameters.

Stand-alone mode can be established by activating the stand-alone start-up specifying the desired start-up settings. This will also write the current registers to memory.

Another method is provided to deactivate stand-alone mode again, which will invalidate all persisted settings and registers forever.

The current start-up settings may be queried and the persisted parameters, subscriptions and Tama program may be reloaded from persistent memory, which discards all recent changes to the register parameters, subscriptions and code space.

The following example shows how the start-up settings for a drive with homing Tama program would be set up:

```
device.ActivateStandAloneStartup(new StartupSettings {
    StartIsochronousTamaVM = true
});
```

10 Firmware Update

The TAM API allows updating the firmware using Tria-Link communication. This way, a TAM Device does not need to be physically opened in the field in order to gain access to the debug connector used for initial programming.

Firmware updates are based around the concept of *products* and *flash areas*. A product is member of a greater *product family*.

Firmware is deployed as *packages* for different product families. There may be multiple firmware with different functionality – *feature sets* – for one product family. Firmware packages are semantically versioned [17].

For example, Triamec Motion AG provides the packages

```
TIOB-1046+EnDat.TAMfw
TIOB-1046+PulseTrain.TAMfw
```

for its TIOB01 and TIOB02 I/O module products. So, the TIOB family comes with firmware having either feature set `EnDat` or `PulseTrain`. That is, there is no support to have an EnDat encoder connected and at the same time output a pulse train signal.

10.1 Firmware Infrastructures

To provide a firmware update executed by the running firmware itself can be dangerous. Depending on hardware resources on a TAM Device and other requirements, different firmware infrastructures have been implemented, ensuring reliable updates.

1. The *Factory/Alternative* infrastructure
   Initially, the TAM Device is programmed with a *factory* firmware. As needed, an *alternative* firmware may be loaded in the field for new features or bug fixes. Conversely, the factory firmware cannot be overwritten in the field. Both firmwares are fully functional, and the alternative firmware can be updated by itself. To activate the updated firmware, the device needs to be power cycled.
2. The *Base/Application* infrastructure
   This infrastructure is used in presence of small persistent code memory, where two full firmware versions wouldn't fit and the code is directly executed. Each firmware may only reprogram the other firmware and the other firmware can be activated without rebooting.
   Only the *application* firmware is fully functional, where the *base* firmware's sole capability is to

update the application firmware. A small independent bootloader chooses one of the firmwares at start-up, where the application firmware is preferred.

Note Triamec devices implement the Factory/Alternative infrastructure.

When referencing different firmware, the term *firmware location* has been established. When downloading new firmware, a location must be specified as *destination* of the download.

10.2 Transferring Firmware

The firmware related functionality of a product is accessed by its package downloader, shown in figure 20. Concurrent downloads are prohibited and the downloader tells about its state through the `IsBusy` property.

Progress change and completion events may be used by GUI's to show the progress and outcome of long-running download process, providing percentage and other information. Note that the method doesn’t throw an exception if the download fails. Instead, the completion event must be handled in order to get that information.

Another firmware location may be activated, and the currently active firmware location can be queried.

Whenever a firmware download cannot be completed (for example because of a checksum error), the firmware infrastructure ensures that it can never be activated.

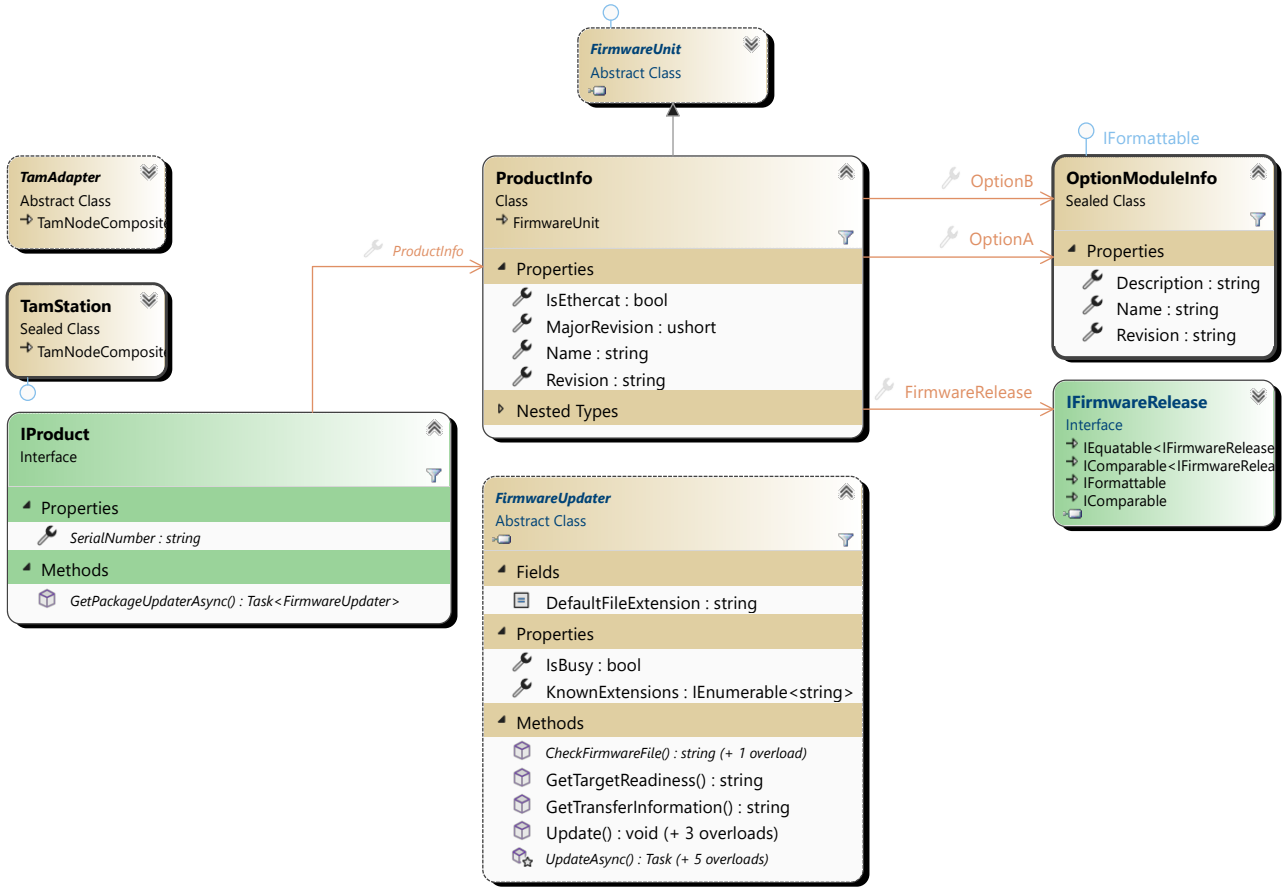

Figure 20: The Firmware API

The firmware related functionality of a product is accessed by its updater. A new firmware may be applied by starting a download. Switching between different feature sets might be prohibited.

Given an `ITamDevice` instance device and a string file with the path to a `.TAMfw` file, the code to perform a firmware update could look like this:

```
IProduct product = device.Station;
FirmwareUpdater updater = await product.GetPackageUpdaterAsync();
await updater.UpdateAsync(file);
```

Caution When transferring a new firmware with a new register layout in a Base/Application infrastructure, the link needs to be re-identified because device IDs are considered static to the TAM Software.

11 Simulation

Simulation facilitates software development when hardware is not yet ready or to set up efficient and cost-effective testing environments.

11.1 Simulated Features

The TAM simulation can be used like a hardware topology and is extensible.

It mainly implements the Tria-Link protocol from a device perspective, by executing the Tria-Link communication commands issued by the TAM API, and providing the set up needed for data acquisition.

The simulation provides a simplified path planner. It does not include the stand-alone mode or firmware updates. Physical aspects are not simulated by default. For example, movement only uses position and velocity values, and the actual measured position always equals the value forced by the path planner. Current and position controllers are not implemented.

This behavior may be refined by extending the simulation to special needs. This extensibility is out of the scope of this document. Please contact us if you are interested.

11.2 Creating Simulated Environments

The TAM API provides two fundamental ways to create simulations, both yielding one or multiple instances of the `SimulatedTriaLinkAdapter` class. These instances are passed with the `AddLocalTamSystem` method of the `TamTopology` class, resulting in a topology working without hardware.

```
TamTopology topology = ...;
SimulatedTriaLinkAdapter simulatedTriaLinkAdapter1, simulatedTriaLinkAdapter2 = ...;
TamSystem s = topology.AddLocalTamSystem(null, simulatedTriaLinkAdapter1,
simulatedTriaLinkAdapter2);
```

A simulated Tria-Link adapter can be set up from a TAM Configuration, or explicitly. The latter is out of the scope of this document.

The TAM Configuration is passed to a simulation factory method which exactly emulates the existing adapters, links and devices. As a consequence, the same TAM Configuration can be applied to a TAM Topology which is instantiated using the simulated environment.

The code below demonstrates the `SimulationFactory` class.

```
Deserializer d = new Deserializer();
LoadResult result = deserializer.Load("topology.xml");
SimulatedTriaLinkAdapter[] simulatedAdapters;
if (result.Errors != null) {
    throw new Exception("Load errors during TAM configuration loading.");
} else {
    // Create a simulated system from a TAM configuration.
    simulatedAdapters = SimulationFactory.CreateSimulatedTriaLinkAdapters(d.Configuration, null);
}
```

The second parameter of the factory method is an extension point.

Note You still need to load the TAM Configuration afterwards, as described in section 8.

12 Deployment

Applications built on top of the TAM Software mainly reference the `Tam.dll`. However, a great number of other libraries are also needed, depending on what features you need. If a library is missing in the executable directory, the system won't work as expected.

As a rule of thumb, you need to deploy anything set up by Triamec's NuGet packages. Installing the TAM Software on the target machine is a prerequisite, for the following reasons:

- Installs drivers.
- Installs support libraries in the GAC used by the `Triamec.Tam.UI` NuGet package.

Limitation

If you want to integrate the TAM System Explorer into your application, running it as a 32-bit process on a 64-bit operating system is not currently supported out of the box. You'll need to install the TAM Software on a 32-bit operating system and export the `NationalInstruments.UI.Styles3D` library out of the GAC there.

Windows XP is no longer fully supported. Your project needs to target .NET framework ≤ 4.0, and developer and target machine need a TAM SDK < 7.0 installed. Some features introduced as of TAM SDK 7.0 might not be present in the .NET framework 3.5 version of the libraries packed with the NuGet packages, most notably support for the new firmware file format introduced in 2017.

13 Advanced Topics

This chapter comprises some features which are only of interest for customers with special needs.

13.1 DevOps Recommendations

Working with NuGet packages introduces an additional level of complexity you need to tackle.

Here are some recommendations with respect to continuous integration.

13.1.1 Manage NuGet Locations

We recommend to set up a place where all NuGet packages used by your project/department/organization are hosted. A local feed should be the right tool for many uses.

Caution You need to preserve all NuGet packages used over time, since your revision control system won't typically include these binary assets.

Next, register this local feed in the `nuget.config` file.

Here is an example a `nuget.config` could look like:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <packageSources>
    <add key="Acme" value="\\Acme\Nuget" />
  </packageSources>
  <disabledPackageSources>
      <add key="nuget.org" value="true" />
  </disabledPackageSources>
  <activePackageSource>
    <add key="Acme" value="\\Acme\Nuget" />
  </activePackageSource>
</configuration>
```

Caution Avoid directly referencing libraries which are also pulled in with some NuGet package in a solution with many projects. That said, Visual Studio should warn you if a project uses more than one version of a library. But you won't see NuGet packages to consolidate when managing NuGet packages for solution.

13.1.2 Enable Package Restore in CI

When using command line builds, consider introducing a build step where NuGet packages get restored if needed. This ensures that the packages will be installed even when not building inside Visual Studio, where packages are automatically restored upon build.

In MSBuild, this could look like this:

```
<!-- RestoreExternals

     Pulls external libraries and tools not checked-in into source control into this local copy of
     the repository.

  [IN] @(solutions) full paths of solutions containing information about which external libraries
                    are needed.
  -->
<Target Name="RestoreExternals" Returns="%(solutions.Identity)">

  <!-- Force target batching because, otherwise, we'd have multiple Restore x arguments. -->
  <ItemGroup>
    <_RestoreNuGetArg Include='"$(NuGetClient)"'/>
    <_RestoreNuGetArg Include='Restore "%(solutions.Identity)"'/>
    <_RestoreNuGetArg Include='-MSBuildPath "$(MSBuildBinPath)"'/>
  </ItemGroup>

  <Exec Command="@(_RestoreNuGetArg,' ')"/>
</Target>
```

Hereby `$(NuGetClient)` represents the path to the executable of the NuGet Client command line interface.

13.2 Setup Protection

As a system supplier, you can protect your setup with a password. This can reduce support incidents due to modifications by unauthorized personnel through the TAM System Explorer.

The TAM API itself doesn't protect any resources, but provides the means to enable and consider setup protection state through the `ITamDevice.Protection` interface. Use this in your setup application for the finalization steps.

Remark The APIs accept a password argument. The `SetupProtector` UI component of the *Triamec.Tam.UI* NuGet, which is also leveraged by the TAM System Explorer, requires a user name and a password. It simply collates those two strings together and passes them to the `ITamDevice.Protection` interface.

Caution You must manage the password such that it doesn't get lost. If you forget the password, you will need to contact Triamec support.

Caution Since the password is hashed with a weak algorithm, don't reuse a sensible password.

13.3 Local-Bus Registers

Beneath the registers described in chapter 4, some devices expose a set of *peripheral* registers located on the device's local bus. The terms *periphery* and *local-bus* are used interchangeably in this document.

Largely undocumented, they are basically an internal implementation detail. However, in alpha testing, customers might need to access such registers. The namespace `Triamec.Tam.Periphery` provides the respective classes.

Caution Only use these APIs when instructed so by Triamec. Mark your code as intermediate such that a reviewer can look out for an updated method to accomplish the task.

The local-bus provides access to a set of local-bus devices. The registers on these devices can basically be writable or read-only. Due to their low level nature, reading a register might also have some side-effect, or there might be a write-only register.

Access the periphery via the `IPeripheryLayoutOwner` interface, implemented by adapters, stations and option-modules with an FPGA soldered on them.

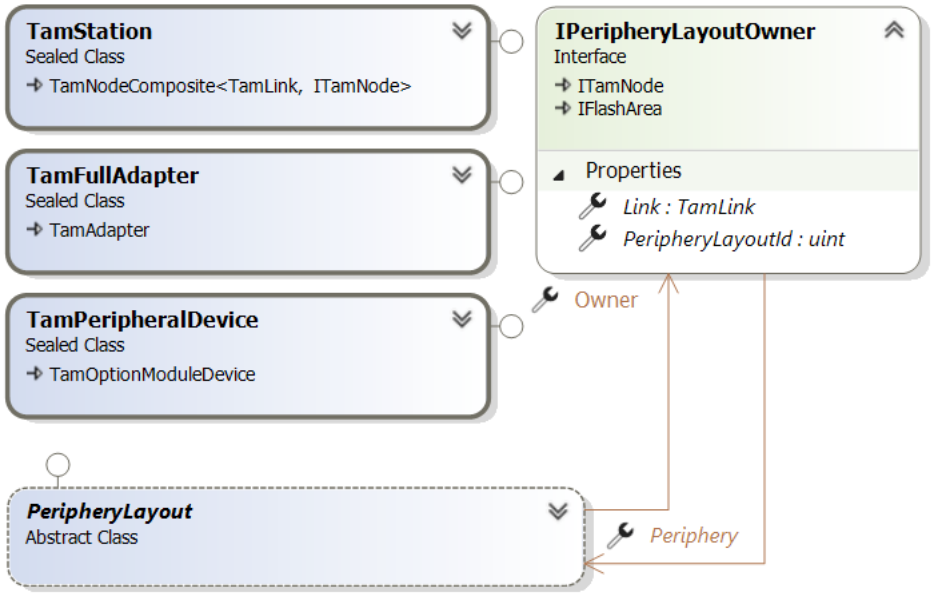

Figure 21: Accessing the local-bus

Individual devices can be enumerated or retrieved by an identification key.

It's possible that the local-bus offers more than one device with the same key. The devices have an index to distinguish them. This index is sometimes called the *local-bus axis*, but isn't stringently related to the axes of a drive.

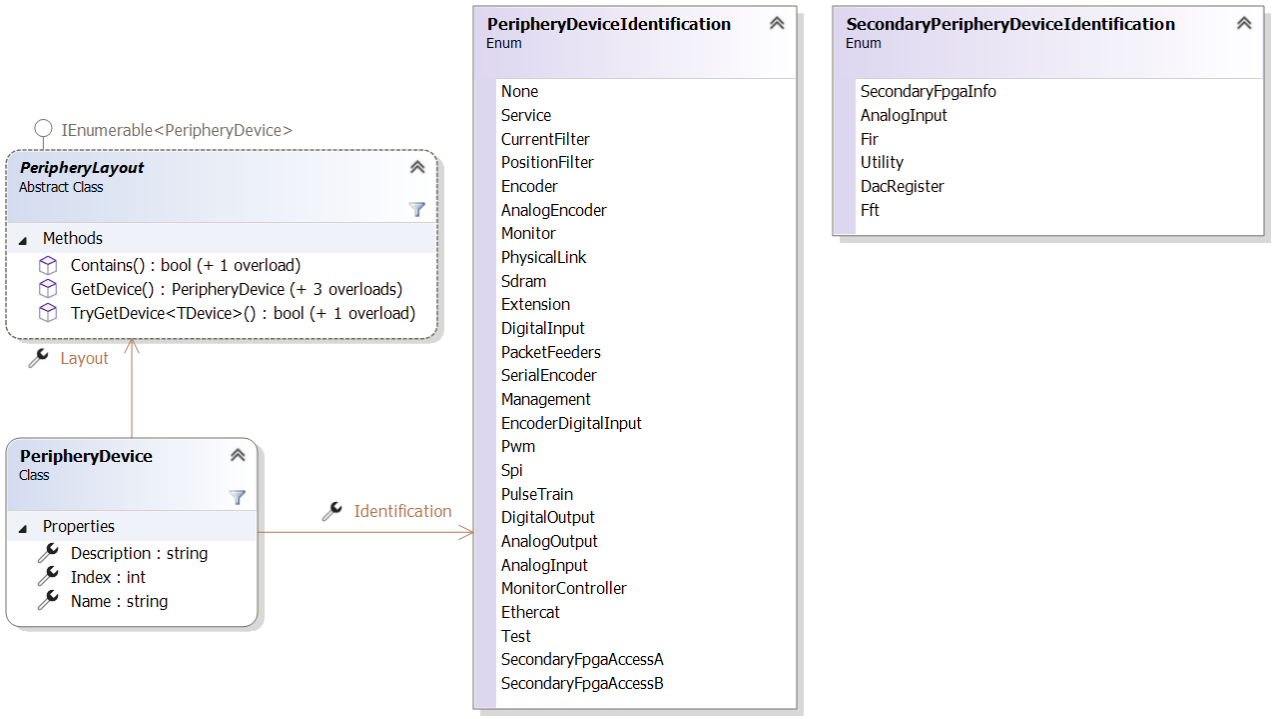

Figure 22: Get a peripheral device

With the device in hand, define peripheral registers on them by constructing an instance matching the specification. The most commonly used type is `PeripheryUInt` with a width of 32 bit.

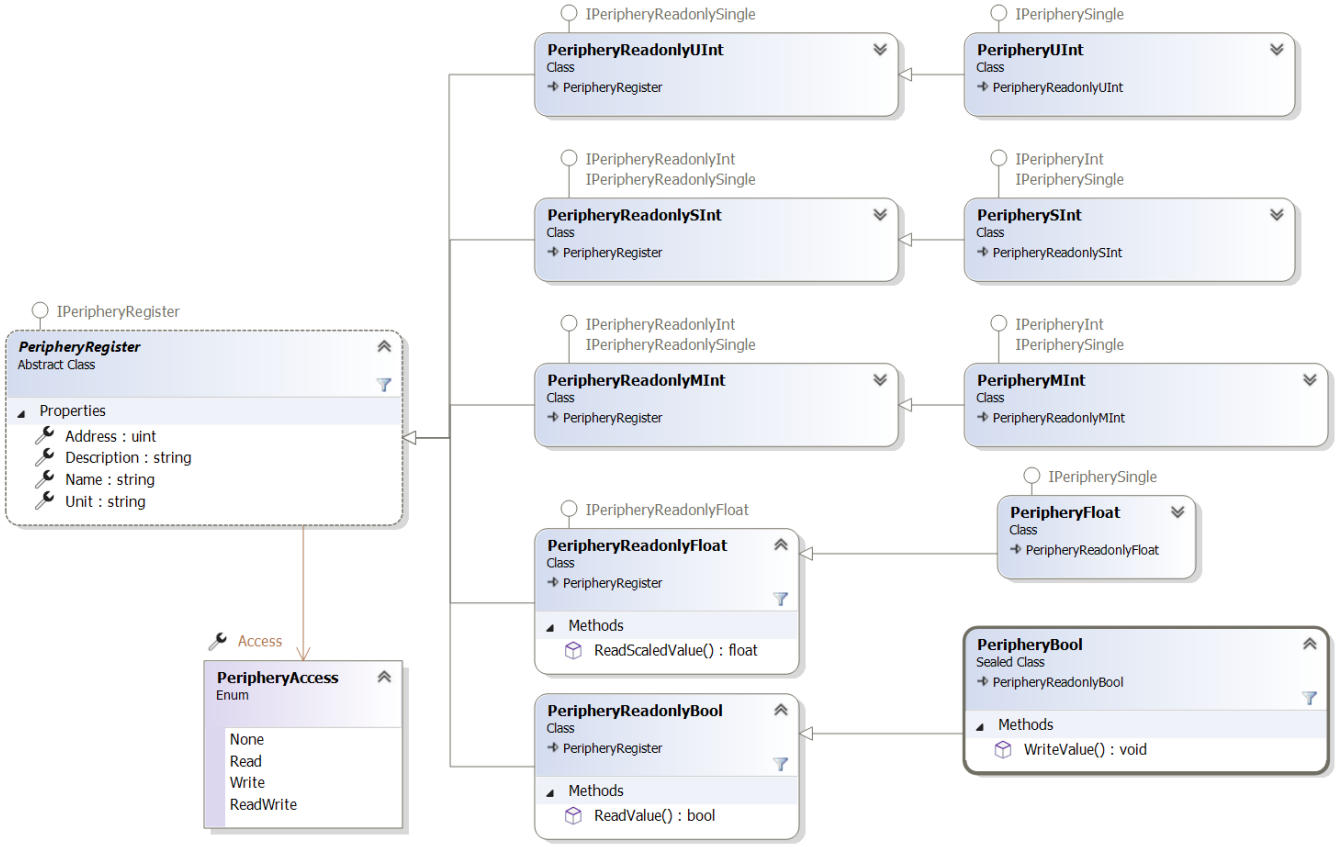

Figure 23: Local-bus register types

The following example shows how to read a local-bus register.

```
PeripheryDevice eimTimingDev = station.Periphery.GetDevice(PeripheryDeviceIdentification.Service);
var imxRdMinMaxReg =
    new PeripheryReadonlyUInt(eimTimingDev, null, null, null, PeripheryAccess.Read, 0x32, 32);
uint imxRdMinMaxVal = imxRdMinMaxReg.ReadValue();
```

Option modules are available as children of the station node. For example, to access a pulse train option module in slot B, write:

```
PeripheryLayout periphery = ((IPeripheryLayoutOwner)station["PT (B)"]).Periphery;
```

For some option module local-bus devices, you might need to use the `SecondaryPeripheryDevice-Identification` enum and cast it to `PeripheryDeviceIdentification`, if the latter doesn't contain the device's key.

13.4 Working With Multiple Register Layouts

An advanced scenario includes writing code which works with more than one register layout. Only use

the following outlined methods when strictly necessary in order to hold your code readable.

We recommend to introduce a wrapper class which abstracts away the usage of different register layouts.

```
using Register_DUT1 = Triamec.Tam.Rlid4.Register;
using Register_DUT2 = Triamec.Tam.Rlid5.Register;
using Register_DUT3 = Triamec.Tam.Rlid19.Register;

class DutRegister {

   readonly Register_DUT1 _dutReg1;
   readonly Register_DUT2 _dutReg2;
   readonly Register_DUT3 _dutReg3;

public DutRegister(ITamDevice dut) {
   _dutReg1 = dut.Register as Register_DUT1;
       _dutReg2 = dut.Register as Register_DUT2;
       _dutReg3 = dut.Register as Register_DUT3;
   }

   …
}
```

This class provides different registers based on the layouts it supports:

```
public ITamReadonlyRegister<float> Voltage =>
      _dutReg1?.Axes[0].Signals.CurrentController.ControllerOutputQ
   ?? _dutReg2?.Axes[0].Signals.CurrentController.ControllerOutputQ
   ?? _dutReg3?.Axes[0].Signals.CurrentController.DesiredVoltageQ;
```

Alternatively, it might be possible to work down the register tree with the following pattern:

```
RegisterComposite composite = device.Register;
composite = composite["General"] as RegisterComposite;
composite = composite["Signals"] as RegisterComposite;
var register = composite["TriaLinkTimestamp"] as ITamReadonlyRegister<int>;
```

Finally, different register layouts can be addressed by using the tags infrastructure as described in the next paragraph.

13.4.1 Tagging

A register may be tagged with one or multiple keys, such that a register can be found without the exact knowledge of the layout. Tags may alternatively be defined as key-value pairs.

Note This is an advanced concept only used when writing some sort of generic code which needs to work with different register layouts.

The `FindTaggedComponent`, `FindTaggedComponents` and `FindTaggedComposite` methods on a register component search for registers under that component.

The keys are always unique relative to a subtree of the layout. For example, the axis signal tags are assigned once per axis in a drive. If a drive has three axes, three registers will be found when searched on the root register. Therefore, one might probably first search the axis root (using the axes tag), and then the axis related register.

Figure 24: The most commonly used register tags

Register tags are used to find specific registers in a general register tree where the exact structure is not known. For example, the actual position signal may be found by searching for the `AxisSignalTags.ActualPosition` *tag within the root register.*

Alternatively, use the `TamAxis.FindReadonlyRegister` and `TamAxis.FindRegister` convenience methods. They can even be used to find general device registers.

Note The set of defined tags evolves with new versions of the TAM API, and some tags may only be defined in certain register layouts, while they are missing in others. For example, the `Axes` tag might not be defined in a register layout designed for an I/O module.

13.5 Customer Settings

Management of preferences is a task which needs to be solved for every application. The TAM Software uses the standard .NET way, *application settings*, for its internal configuration, with some extensions. *Preferences* allow an application to be configured in the field without rebuilding it. The *Triamec*

workspace contains most of the application settings.

13.5.1 Preferences

The preferences mechanism included in the TAM Software builds a bridge between the two .NET infrastructures *application settings* and *property descriptors*. The most common usage is the ability to populate a `PropertyGrid` with application settings.

The most important class is the `Triamec.Tam.UI.PreferencesDialog` form which you can easily integrate in your application. From scratch, it will show TAM Software specific settings, but adding your own settings is straightforward:

1. In the settings tab of the properties of your Visual Studio project, create some settings.
2. Click on the “View Code” tool-bar button. This will generate a new file.
3. Import the namespace `Triamec.Configuration`.
4. Let the class inherit from `IPreferences`.
5. Add the following code snippet to the class:

```
#region IPreferences members
IEnumerable<SettingsPreferenceDescriptor> IPreferences.Preferences {
 get {
    var preferences = new List<SettingsPreferenceDescriptor> {
        new Preference(this, "<Settings property name>", "<Pretty setting name>",
            "<Category>")
    };

    if (Preference.ShowAdvancedContent) {
        preferences.Add(new Preference(this, "<Settings property name>",
            "<Pretty setting name> (advanced)", "<Category>"));
    }

    Preference.AddDebugPreferences(preferences, this);

    return preferences;
 }
}
#endregion IPreferences members
```

   As the listing indicates, you can define normal, advanced and debug preferences. Only those settings defined here will show up in the preferences dialog. The string `"<Settings property name>"` needs to be the name of an existing settings property in the class. The categories build a flat hierarchy.
6. In the constructor, add the line

```
this.RegisterForAdditionalServices();
```

   In conjunction with using a `TamTopology`, this will automatically save changed settings as the application exits. This also works if you skip steps 4 and 5 above, as long as you declare the class as implementing `IApplicationSettings`. Note that the base class already implements this interface completely.
7. Before any namespace scope, add the line

```
[assembly: Preferences(typeof(<your namespace>.Settings))]
```

   where `<your namespace>` is the namespace in which the class resides.

All you have to do now is to make use of the `PreferencesDialog` form. The `Description` property of the setting will show up in the dialog. Application scoped settings will be read-only.

If the setting's type is an enumeration from which you have the sources, you may decorate the enumeration's fields with `Description` attributes. They will be shown in a drop-down list instead of the field names.

13.5.2 Triamec Workspace

The *workspace* is a file repository on disk with configurations, settings and measurements related to the setup of Triamec drives. Most application settings defined by the TAM Software are persisted in the workspace.

Workspace Updates

Each workspace is tied to one version of the TAM Software. If an application built against a different version of the TAM Software opens the workspace, the workspace is updated.

The default behavior is to mark the Triamec workspace with the new version. This can be changed such that the version is not touched: Subscribe to the `Workspace.Updating` event. In the handler, set the `Update` property to `WorkspaceUpdate.Ignore`.

Leveraging the Workspace

The `Triamec.Configuration.Workspace` class provides paths into the workspace which can be used to save application specific assets into the workspace.

You may wish to save your own settings in the workspace, which can be accomplished in two ways:

- In the settings designer, each setting has a `Provider` property. Set this property to the value `Triamec.Configuration.WorkspaceSettingsProvider`. Those settings where this provider is set will be persisted in the workspace.
- If you want to persist all of your settings in the workspace,
  a) Press View Code in the tool-bar of the settings designer.
  b) Decorate the settings class with the attribute `[SettingsProvider(typeof(WorkspaceSettingsProvider))]`.

Your settings will appear in a `.config` file in the `Settings` (formerly `TAM\Settings`) workspace folder. The settings group name defaults to the default project namespace with additional sub-namespaces according to the names of the sub-folders where the settings are defined within the project. The class name is also appended. This group name can be overridden by decorating the settings class with the `SettingsGroupName` attribute.

In order for your settings to be reloaded when the user changes the workspace, refer to step 6 in chapter 13.5.1.

Have a look at the `DirectFeed` sample which integrates the workspace.

Private Workspace

Sometimes it's not desired that an application interferes with the settings made to the default workspace. This section describes how to set up the workspace to be private to the application. Refer to the Acquisition sample project as an example.

Step 1: Set up a workspace

A workspace consists of a directory containing a TAM workspace file. This is a file containing application configuration similar to `app.config`. with the `.TAMws` extension. As a seed, copy the .TAMws file from the default Triamec workspace (locate it using menu **File > Open Workspace Folder** in TAM System Explorer) into your project and ensure it's copied to the output directory.
Open your workspace file and make the desired adjustments.

Step 2: Use the workspace

In order to let your application use its own workspace, complement its `app.config` as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>

    <!--(…)-->

    <sectionGroup name="userSettings"
                  type="System.Configuration.UserSettingsGroup, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089">

      <section name="Triamec.Workspace"
               type="System.Configuration.ClientSettingsSection, System, Version=4.0.0.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089"
               allowExeDefinition="MachineToLocalUser"
               requirePermission="false"/>

    </sectionGroup>
  </configSections>

  <!--(…)-->

  <userSettings>

    <Triamec.Workspace>
      <setting name="WorkspaceFile" serializeAs="String">
        <value>Workspace.TAMws</value>
      </setting>
    </Triamec.Workspace>

  </userSettings>

  <!--(…)-->

</configuration>
```

This points the workspace API to the desired workspace, relative to the `Triamec.Common.dll` library loaded by your application. This library is typically located beneath the executable.

13.6 Customer Hardware

One of those special needs is when customers create their own Tria-Link hardware. This brings in the requirement for some extension points within the TAM API library reference. It is possible to set up brand new hardware without the need for a new version of the TAM API library reference.

13.6.1 Custom Product Types

New products must be registered with TAM using the `Triamec.TriaLink.ProductType.Register` method. Some specific properties need to be passed additionally to informational arguments. These are used by TAM to create the appropriate interfaces for the new device.

Alternatively, the `UserProductType` application setting allows configuring new product types without compiling any line of code. The application configuration file shipped with the TAM System Explorer contains a template for demonstration.

If a new product is connected without being registered beforehand, it will show up as `Unknown` device.

Internally, the product type is one property used to uniquely identify a specific device. Another documentation, the TAM Device Identifications Developer Documentation, defines requirements and usage of all identifiers returned by devices.

In order to retrieve the product type you type:

```
IUpdatable adapter_station_or_device = ...;
Console.WriteLine(adapter_station_or_device.HardwareIdDetails.ProductType);
```

13.6.2 Custom Register Layouts

Please get familiar with chapter 4 about Registers before advancing in this section.

New devices usually have a new register layout. Firmware upgrades may come with a changed layout, too. It is possible for the device manufacturing site to deploy a corresponding register catalog together with the hardware. The TAM Software does not need to be redeployed, because it will find the new register catalog and instantiate the new device's register using that catalog.

User applications need to be recompiled if they rely on specific register layouts. If they solely rely on register tags, they may not need to be rebuilt.

One constraint is that the register catalog must have been built against that exact version of the TAM Software.

In order to support plug & play insertion of new device types, the `LayoutManager` in the `Triamec.Tam` namespace provides the `UnknownLayout` event. This enables a customer framework to ship device specific application software – including the register catalog – together with the new device and to establish an automatic installation mechanism.

```
LayoutManager.Instance.UnknownLayout +=
    new UnknownLayoutEventHandler(OnLayoutManagerUnknownLayout);
```

The event handler needs to copy the register catalog such that the following procedure will succeed:

```
Assembly assembly = Assembly.Load("RegisterLayout." + tamDevice.RegisterLayoutId);
```

In other words, register catalogs need to follow a strictly defined naming scheme, and one register catalog must always contain only one register layout.

```
string OnLayoutManagerUnknownLayout(object sender, UnknownLayoutEventArgs args) {
    // pseudo definition code
    uint myNewRlid = 9u;
    string sourceDirectory;
    string myNewRegisterLayoutAssembly = LayoutManager.REGISTER_LAYOUT_NAME + '.' +
        myNewRlid+".dll";

    // check whether we can provide a new layout for the request
    if ((args.LayoutType == LayoutType.Register) && (args.Id == myNewRlid)) {
```

```
        // copy my new register layout to the executing directory
        File.Copy(
            Path.Combine(sourceDirectory, myNewRegisterLayoutAssembly),
            Path.Combine(Path.GetDirectoryName(
                Assembly.GetExecutingAssembly().GetName().CodeBase),
                myNewRegisterLayoutAssembly), false);

        // return simple assembly name without version and public key information
        return Path.GetFileNameWithoutExtension(myNewRegisterLayoutAssembly);

    } else {
        return null;
    }
}
```

This example event handler knows where to get register layout `9`. If the event arguments indicate that this register layout is requested, it copies the new register catalog to the application directory.

13.6.3 Extending Enumeration Registers

A framework, such as the TAM Software, may impose a *general* enumeration type upon a certain register. New *specific* values may need to be added to the enumeration of that register without the need of recompiling the framework.

While it is not possible to directly adding values to the general enumeration type without recompiling the framework, enumerations allow having arbitrary integer values or values from other enumeration types casted to itself.

Before going into the details, lets consider an example, the register tagged with `GeneralSignalTags.DeviceError`. This register is supposed to have values of the general type `DeviceErrorIdentification`. However, a firmware writer may redefine the register to have values of a specific enumeration. This allows her to define additional device-specific errors in the same manner as in general firmware areas, where general `DeviceErrorIdentification` values are used.

Step-Through Guide

Go through this instructions in order to extend a register. The device error example is continued.

Refining the register
First, define a new enumeration within the register layout XML file:

```
<Enum flags="false" prefix="RDEt" name="KnackerErrorIdentification">
  <Descriptions>
    <Description cultureInfo="en">Error identification for the Knacker.</Description>
  </Descriptions>

  <EnumValue prefix="RDEeDRVERR_" name="KnackerJam" value="128">
    <Descriptions>
      <Description cultureInfo="en">The knacker has a jam.</Description>
    </Descriptions>
  </EnumValue>

  <EnumValue prefix="RDEeDRVERR_" name="KnackerBroken" value="129">
    <Descriptions>
      <Description cultureInfo="en">The knacker is broken.</Description>
    </Descriptions>
  </EnumValue>
</Enum>
```

Don't repeat the general values, just define the new ones. Start with values far above the general values in order to allow the general enumeration to grow.

Change the register from

```
<Member name="deviceError" type="DeviceErrorIdentification">
```

to

```
<Member name="deviceError" type="KnackerErrorIdentification" headerType="int"
   converter="ExtendedEnumRegisterValueConverter&lt;DeviceErrorIdentification&gt;">
```

The `headerType` attribute forces the register's type to `int` in the generated header file. Because every enumeration is implicitly casted to `int` in C, this doesn't brake existing firmware code, while allowing to directly assign values from the `KnackerErrorIdentification` enumeration.

The `converter` attribute specifies the `DeviceErrorIdentification` type as general enumeration. In the TAM System Explorer, the register will be shown with the correct value, either from the general or the specific enumeration.

Note You cannot currently use this converter for writable registers. If a writable register's enumeration is refined in the described way, only values from the specific enumeration can be selected in the TAM System Explorer, unless you provide your own custom converter. One way to circumvent this limitation would be to copy the values from the general enumeration into the specific enumeration. However, doing so is not recommended as it breaks the single source principle and forward compatibility of the device-specific code.

Using the register in the firmware

As written above, the register will have type `int`. Assigning a value from the specific enumeration to the register is trivial. Getting a value can be done using a cast.

Existing references to the register don't need to be fixed.

Using the register in the application

Assume the `KnackerErrorIdentification` is defined in register layout `11000`.

Evaluate the register directly:

```
using Triamec.Tam.Rlid11000;
...
var error = register.General.Signals.DeviceError.Read();
if (error == KnackerErrorIdentification.KnackerJam) ...
if ((DeviceErrorIdentification)error == DeviceErrorIdentification.ExternalError) ...
```

The same through a framework API:

```
using Triamec.Tam.Rlid11000;
...
DeviceErrorIdentification error = device.ReadDeviceError();
if ((KnackerErrorIdentification)error == KnackerErrorIdentification.KnackerJam) ...
if (error == DeviceErrorIdentification.ExternalError)
```

The same usage holds when using the register in a Tama program. Note that casts from one enumeration to another won't have any performance impact to existing code.

Occurrences in the TAM Software

The only tagged registers currently supported by the TAM Software to be extended in the described way are the `DeviceError` and `AxisError` registers. Values read from these registers are always exposed using the general types `DeviceErrorIdentification` and `AxisErrorIdentification`, respectively. Device-specific code may safely cast these values to the specific enumeration type as needed.

Limitation

The specific error enumeration should not have values greater than `255` for `DeviceError` and `511` for `AxisError` because of its occurrence in the `ITamDevice.Transition` event. At the same time, values should not be smaller than `64` in order to allow the general `DeviceErrorIdentification` and `AxisErrorIdentification` to grow over time.

You may use this technique with other registers as long as they are not referred by the tags defined by the TAM Software, i.e., if they don't have some general meaning to the TAM Software.

13.6.4 Log Tria-Link Traffic

Firmware or hardware issues often manifest themselves in the form of communication timeouts or other malfunction in an application. In order to help to track down such failures, it is possible to log all traffic over the Tria-Link channel.

This feature can be enabled in two ways.

Configuring Trace Listeners

One of the Listeners of the `Triamec.Diagnostics.Log.Source` must have its `Filter` set to an `EventTypeFilter` whose `EventType` subsumes `SourceLevels.Verbose`.
This can be accomplished by adding the following snippet to the application configuration:

```
<system.diagnostics>
  <sources>
    <source name="Log">
      <listeners>
        <clear/>

        <add name="WorkspaceHigh"
             traceOutputOptions="DateTime"
             type="Triamec.Diagnostics.WorkspaceTraceListener,Triamec.Common"
             initializeData="high"
             maxFileSize="10000000">
          <filter type="System.Diagnostics.EventTypeFilter" initializeData="Warning" />
        </add>

        <add name="WorkspaceLow"
             traceOutputOptions="DateTime"
             type="Triamec.Diagnostics.WorkspaceTraceListener,Triamec.Common"
             initializeData="all"
             maxFileSize="10000000">
          <filter type="System.Diagnostics.EventTypeFilter" initializeData="Verbose" />
        </add>

      </listeners>
    </source>
  </sources>
</system.diagnostics>
```

This snippet retains the existing logging behavior apart from including verbose messages into the *all* workspace log. For a simpler example how to configure the listeners, see the TextWriterTraceListener remarks. Note, however, that addressing the *Log* source is mandatory.

Setting a Preference

Alternatively, set the *Log Tria-Link traffic* preference in the Communication (advanced) section of the Preferences dialog. This preference is persisted in the Triamec workspace. The preference is ignored when verbosity is configured using the trace listener.

The trace listener configuration approach conceptually enables more logging than just Tria-Link traffic. This is not the case with the preference approach.

For performance reasons, you need to restart the application for any configuration or settings change to become effective.

13.7 Life-Cycle Considerations

Using the TAM System Explorer, you might have realized that some instances in the topology have limited lifetime. For example, a device attached via USB might be disconnected.

Depending on the use-case of your application, you might need to track new instances or disposal of such instances. Working with disposed instances will cause `ObjectDisposedException`s.

The following list depicts and discusses changes down the hierarchy.

- Workspace Changes
  The TAM Topology saves connection settings and other details in the current workspace (see section 13.5.2). Opening another workspace should therefore lead to the disposal of the old TAM Topology and the set-up of a new TAM Topology.
  The straightforward way to implement this is by restarting the application. The Eclipse IDE is a prominent example for this approach.
- Connections
  TamSystems may be connected or disconnected from a TamTopology.
- Hot-plugging and Surprise-removal
  TamAdapters as well as TamLinks may appear or disappear spontaneously.
- Adapter Reset
  When resetting a TamAdapter, all of its TamLinks might be removed and repopulated.
- Link Booting
  When identifying or initializing a TamLink, all of its TamStations will be removed and repopulated.
- Firmware Upgrades
  In a Base/Application firmware infrastructure (see section 10.1), the subscriptions on the device are reset upon firmware upgrade.
- Axis Changes

When setting a new Motor Base Configuration on an ITamDrive, its `TamAxis` instances will be removed and the Axes property repopulated (only old generation drives).

Objects which use instances from the TAM Topology often have observing character. There are different options for listening to the above changes:

- Use the `IComponent.Disposed` event. Some, but not all instances in the TAM Topology implement the `IComponent` interface. If an instance doesn't implement `IComponent`, navigate up the hierarchy using `ITamNode.ParentNode` until you find an `IComponent` instance.
- Use the `ITamNodeComposite.NodesChanging` event.
  Since this event is fired for new as well as for removed nodes, the instances returned by the `NodesChangingEventArgs.ChangingNodes` need to be tested, see the remarks in the reference documentation.
  This approach has the drawback that the event must be subscribed on all hierarchical levels.

For subscription reset, there is not currently an event in the TAM API, nor will the `ISubscriptionManager` instance reflect that reset.

13.8 Versioning

The different NuGet packages comprising the TAM Software are versioned independently of each other. That is, a new version of the TAM Software doesn’t contain a new release of each of the NuGet packages.

The NuGet package version numbers conform to Semantic Versioning ([17]). The public API comprises the following elements:

- APIs from the `Tam.dll` assembly from the following core namespaces:
  - `Triamec.Tam`
  - `Triamec.Tam.Acquisitions`
  - `Triamec.Tam.Configuration`
  - `Triamec.Tam.Registers`
  - `Triamec.Tam.Requests`
  - `Triamec.Tam.Subscriptions`
- The non-advanced user settings provided in the Preferences dialog of the TAM System Explorer.

This list will grow as APIs get commonly used and stable.

API breaking changes are typically done such that old APIs are still present, but marked with an `ObsoleteAttribute`. This also holds for most APIs not listed above.
When using another programming environment like Python or Matlab, this attribution isn’t accessible. However, many of the obsolete API will throw an appropriate error message at runtime.

You might get a prerelease-version of the TAM Software. Such releases denote testing or experimental versions and may introduce incompatible and unstable API changes.

Glossary

Firmware Code, executed by the microprocessor on a TAM Device. Also referred to as *image*. See chapter 10 how to update the firmware of a device.

GAC Global assembly cache where .dlls are installed for machine-wide access [18]

Module A TAM module is a plug-in instance associated with a TAM Device which may contain 3rd party functionality.

Parameter In a context of registers (chapter 4), a parameter is referred to as a register contributing to the parametrization of a TAM Device. As such, it is committable and saved in the TAM Configuration (chapter 8) and/or persistent on the device (chapter 9).

In the context of modules, a parameter is a special property of a module component class persisted by the TAM Configuration and enumerated by the module infrastructure. In the TAM System Explorer, they are shown in a special *Parameters* tab page. Most often, such parameters save information not present in register parameters, for example the results of an encoder calibration.

PLCopen PLCopen is an association engaged in standards for automation and motion ([16]).

Register Registers is the most important concept of the TAM API to communicate with TAM Devices. An introduction is given in chapter 4.

Schedule Table of deposited Tria-Link messages to be sent over the Tria-Link one by one in real-time at specified relative times. See section 6.5 for an introduction.

Station Uniquely addressable party within the Tria-Link.

This is typically one-to-one related with a TAM Device, but there exist hardware devices seen as multiple stations.

This term is used when talking about communication within the Tria-Link, because the Tria-Link protocol does not know about TAM Devices.

Subscription A publish/subscribe mechanism in order to exchange Register data between Tria-Link stations in real-time and to establish data acquisition from the computer. See chapter 5.

TAM Triamec Advanced Motion.

Term used for software produced by Triamec Motion AG to support their and proprietary drives talking the Tria-Link protocol.

TAM Configuration Configuration framework for computer-side persistency of the parameter registers, the naming within the topology, and module configuration. See chapter 8 for an introduction.

TAM Device Hardware device with a microprocessor implementing the Tria-Link protocol, such that it can be represented by the `TamDevice` class of the TAM API.

TAM Topology The object oriented tree containing corresponding instances of all hardware devices with their respective capabilities.

Tama Solution stack allowing users to easily write code running in real-time on TAM Devices. The code is organized as a Tama program, transferred and started using the `TamaManager` of a TAM Device. The Tama programming language is a subset of C#.

TD-Bus Low-volume collision bus transporting Tria-Link packets to TD-Bus devices such as the TD servo motor.

Tria-Link

1. A protocol developed by Triamec Motion AG for asynchronous and isochronous communication with drives.
2. The Ethernet-based, real-time communication layer using a ring topology, developed by Triamec Motion AG.

Triamec Motion AG Triamec Motion AG is an independent, incorporated company located in Zug, Switzerland. Since its establishment in 2001, it is owned by the management. The company provides to its customers its superior knowledge in the field of mechatronics, particular in dimensioning, design and control of highly dynamic systems, as well as in software engineering as a substantial part.

References

[1] The TAM API library reference
Shipped with the TAM Software and TAM System Explorer releases.
Documents the most important dynamic link libraries delivered with the TAM Software releases.

[2] Tria-Link 10kHz Echtzeitbus
http://triamec.com/de/tria-link.html

[3] .NET - Powerful Open Source Cross Platform Development
https://www.microsoft.com/net

[4] Triamec Motion AG, 2008
Short introduction to the Tria-Link protocol and data link layer.

[5] Triamec Motion Servo Drives product home page
http://www.triamec.com/en/products/TS/index.html

[6] TAM System Explorer home page
http://www.triamec.com/en/products/TAM/SystemExplorer.html

[7] Drive Setup Guide
`SW_TSD-TSP360-TSP710-Setup-Guide_EP001.pdf`, Triamec Motion AG, 2019
How to set up and work with the TAM System Explorer.

[8] Drive-to-Drive Data Exchange with Tria-Link,
`AN142_TriaLink-DriveToDriveDataExchange_EP001.pdf`, Triamec Motion AG, 2022

[9] Tama Real-Time Drive Programming User Guide
`SWTAMA_UserGuide_EP001.pdf`, Triamec Motion AG, 2024
Tama program developer manual deployed with the Tama compiler.

[10] Device State Observer
Version 0.10; Triamec Motion AG, 2007
Concept paper about how to observe the device state machine from a PC application in order to track the execution and completion of commands

[11] TAM Device Identifications Developer Documentation
Version 1.12; Triamec Motion AG, 2006
Document describing the different identifiers defined in the Tria-Link protocol.

[12] NuGet, NuGet Documentation
https://docs.microsoft.com/en-us/nuget/

[13] TAM Software Release Table
`SWNET_ReleaseTable_EP001.pdf`, Triamec Motion AG, 2018

[14] TAM API Release Notes, `SWNET_TamApiReleaseNotes_EP001.pdf`, Triamec Motion AG, 2021

[15] Ethernet Interface
Application Note 123, Triamec Motion AG, 2023

[16] PLCopen TC2 home page
http://www.plcopen.org/pages/tc2_motion_control/

[17] Semantic Versioning home page
http://semver.org/

[18] Global Assembly Cache, https://docs.microsoft.com/en-us/dotnet/framework/app-

domains/gac, Microsoft, 2017

Revision History

Version	Date	Editor	Comment
023	2015-04-01	chm	11: Simulation: Adapt breaking changes; Added section 13.7: Life-Cycle Considerations.
024	2015-11-25 2015-11-26 2015-12-10	chm chm chm	4.2: Committing: Dropped possibility to read shadow. Error: Reference source not found: Error: Reference source not found: Added note about the need to load it twice. 5: Subscriptions and Acquisitions: Revisions according to most recent API changes.
025	2016-08-08	chm	5.4: Acquisitions: Added usage pattern.
026	2016-12-23	chm	11: Simulation and 12: Deployment: Adapt breaking changes.
027	2017-03-15 2017-05-08	chm chm	10: Firmware Update: Change to package format. 12: Deployment: Adapt breaking changes. The motor base configuration is no longer used in new generation drives.
028	2017-08-29 2017-08-08	chm chm	Simplify application development and deployment with NuGet packages. The workspace can now be leveraged by third-party applications out of the box.
029	2017-08-30 2017-09-13	chm chm	2.2: NuGet Distribution: Fix link to NuGet downloads. 2.1: Concepts: Revise introduction of modules.
030	2018-03-13	chm	Added section 13.1: DevOps Recommendations. Updated section 13.8: Versioning.
031	2018-07-10	chm	Added new reference to the TAM SDK Release Table in 2.2: NuGet Distribution
032	2018-10-15	chm	4.4: Speeding Up With Register Lists revised according to refined implementation
033	2018-11-27 2018-11-28	chm chm	2.2: NuGet Distribution: Adjust 3d party dependencies 13.7: Life-Cycle Considerations: Adjustments due to new network layer
034	2019-09-23	chm	Reflect current products in figures.
035	2019-11-14	chm	Added section 13.6.4: Log Tria-Link Traffic
036	2020-11-10	chm	Update Figure 12: Axis State Machine. Introduce TamRequest.WaitForSuccess API
038	2021-05-03	chm	Minor addition in Error: Reference source not found: Error: Reference source not found
039	2021-09-15	chm	State .NET library support of the NuGet packages
040	2021-09-24	chm	The NuGet packages are no longer bundled with the setup
041	2022-02-25	chm	Added chapter Private Workspace
042	2022-05-30	chm	Better describe axis state machine and asynchronous Tama virtual machine

043	2022-09-28	chm	Added chapter 13.3: Local-Bus Registers
044	2022-12-12	chm	Added chapters 6.2.1: Control System Treatment and 13.2: Setup Protection
045	2023-02-15	chm	Revised chapter 6.4: Coupling
046	2023-05-01	chm, sm	Revised chapter 4: Registers
047	2023-06-19	chm	Improved documentation for TamSystem. Reflect samples moved to GitHub.
048	2024-04-24	chm	Improve figure 2. Move content of chapter 7: Real-Time Programming with Tama, to the new Tama Real-Time Drive Programming User Guide [9]

Triamec Motion AG
Lindenstrasse 16
6340 Baar / Switzerland

Phone +41 41 747 4040
Email info@triamec.com
Web www.triamec.com

Disclaimer

This document is delivered subject to the following conditions and restrictions:

- This document contains proprietary information belonging to Triamec Motion AG. Such information is supplied solely for the purpose of assisting users of Triamec products.
- The text and graphics included in this manual are for the purpose of illustration and reference only. The specifications on which they are based are subject to change without notice.
- Information in this document is subject to change without notice.