TRIAMEC

Redefining
Motion Control

User Guide

TAMA REAL-TIME DRIVE PROGRAMMING

With *Tama Programs,* the drive firmware can be extended with customer specific implementations running at up to 10kHz real-time. These programs are written in C# and compilation is integrated in the C# project tool chain. It is possible to load and run the programs during run-time.

This guide starts with a quick start example giving a short introduction how to setup, build, load and execute a simple Tama Program.

An extensive reference follows covering programming, execution, automation and debugging.

TRIAMEC

Table of Contents

1. Introduction....2

2. Quick Start Example....3
   - 2.1 Preparations....3
   - 2.2 Open and Build in Visual Studio....3
   - 2.3 Building without Visual Studio....5
   - 2.4 Load and Run the Tama Program....6
   - 2.5 Create Your Own Programs....6

3. Authoring Tama Programs....6
   - 3.1 Principal Structure....6
   - 3.2 Isochronous and Asynchronous Tama Task 7
   - 3.3 TAM Registers....8
   - 3.4 Errors and Limitations....12
   - 3.5 Combine Multiple Programs....15
   - 3.6 Build Process....16

4. Running Tama Programs....17
   - 4.1 Loading and Enabling....17
   - 4.2 Debugging....19
   - 4.3 Tama Runtime Errors....20
   - 4.4 Task Load Monitoring....20
   - 4.5 Performance Tuning....20

5. Advanced Topics....21
   - 5.1 Tama File Compilation....21
   - 5.2 API....24
   - 5.3 Firmware Evolution....26

References....26

Revision History....27

1. Introduction

A *Tama Program* is a piece of software which can be loaded to a *Triamec Servo Drive* during run time and which is executed with up to 10kHz in real time. With a Tama Program it is possible to extend the drive firmware with user specific implementations. Typical applications with *Tama Programs* are:

- Stand alone control of a drive
- Adaptive gain controller
- Adaptive feed forward
- Compensation algorithms
- etc.

The *Tama Program* has access to all registers in the *TAM* register tree. This allows full control over the axis and the drive. The *Tama* code is typically written in C#.

The first part of this user guide gives a short introduction of how to setup, build, load and execute a simple *Tama Program*. After reading the first part, you should be able to create and run your own simple *Tama Programs*.

The second part provides additional information regarding best practice, debugging, technical details and special cases.

2. Quick Start Example

This quick start example demonstrates the best practice of how to create and run a simple *Tama Program*.

2.1 Preparations

The following software needs to be installed:

- *TAM Software* (see [1] section ‘Software Installation’)
- .NET 10 SDK or newer
- *Microsoft Visual Studio IDE* version 2026 or newer. When available, enable the .NET-development workload. Alternatively, you can use Visual Studio Code or any other C#-capable IDE or editor, see section 2.3.

For testing of the *Tama Program* a running *Triamec Servo Drive* is required which is accessible via *TAM System Explorer* from the PC (see [1] section ‘Connect Drive to PC’).

A connection to the Internet is required to allow loading of the required *Triamec* DLLs (NuGet Packages).

2.2 Open and Build in Visual Studio

This section demonstrates how a *Tama Program* can be loaded and compiled in Visual Studio:

1. Get the ‘HelloTama’ Solution from GitHub. We highly recommend using git directly to clone or fork the repository and use it as version control tool for your own development. Alternatively, you can work with the files directly as shown below.
   - Browse to https://github.com/Triamec/HelloTama
   - Expand the “Code” dropdown.
   - Choose to download the code as a ZIP archive (**Download ZIP**).

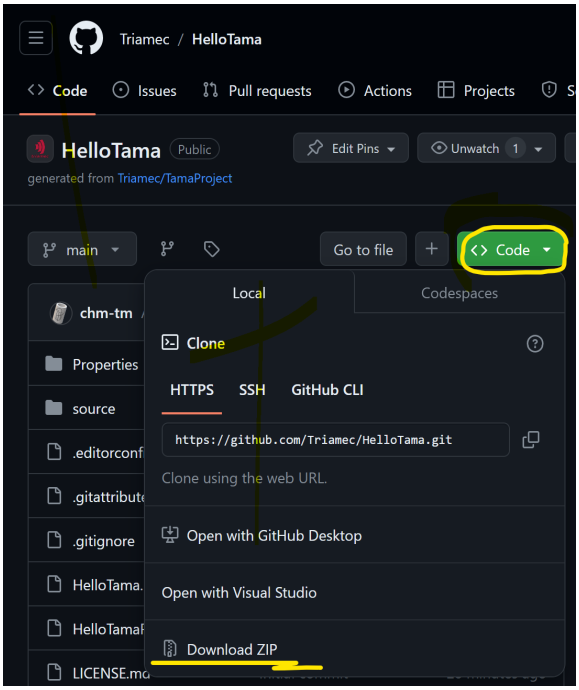

Figure 1: Download the HelloTama sample from GitHub

   - Unblock the ZIP Archive after download.

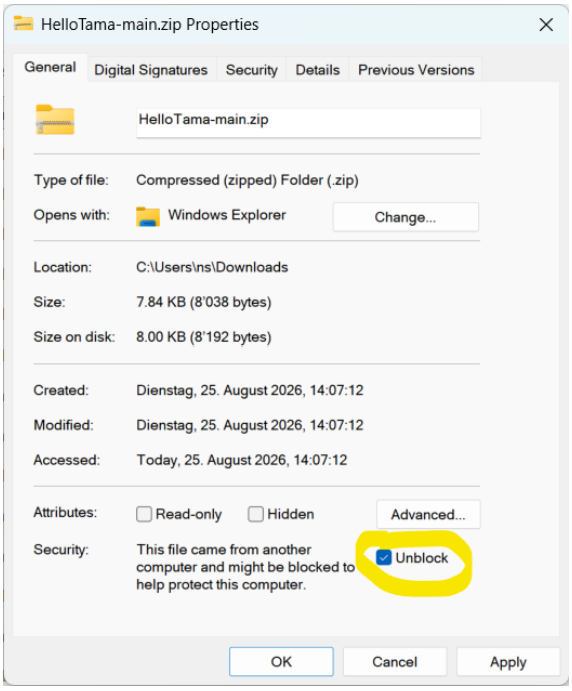

Figure 2: Unblock the ZIP Archive after download

- Extract the ZIP Archive.

2. Open your Visual Studio IDE and open the 'HelloTama' solution with
   - **File** > **Open Project** > select the file `HelloTama.sln` > **Open**
   - Now the **Solution Explorer** (menu **View** > **Solution Explorer**) should show the structure displayed in Figure 1.

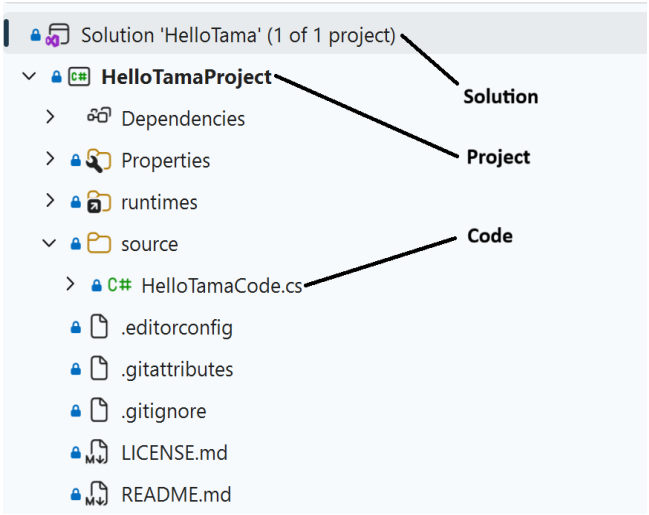

Figure 3: Tama Solution in Solution Explorer

3. In the **Solution Explorer** right click the Solution and click **Restore NuGet Packages**.
4. To view the source code double click `HelloTamaCode.cs` in the **Solution Explorer**. The sample code simply assigns Boolean[0] register to Boolean[1]. Later you might extend the sample code with your own implementation.
5. To build the *Tama Program* click menu **Build** > **Build Solution**
6. In the Output window (**View** > **Output**) check if the build succeeded and check if there are no build errors listed in the Error List (**View** > **Error List**).

If the build failed, close the solution (**File** > **Close Solution**) and reopen it again by following the steps above.

7. If the build succeeds, the generated *Tama Program* can be found at the two following locations
   - In the **Solution Explorer** right click the Solution and click **Open Folder in File Explorer** and navigate to: '`bin\Debug\TamaProject.HelloTama.tama`'
   - In the *TAM System Explorer* click **File** > **Open Workspace Folder** and navigate to '`Tama\TamaProject.HelloTama.tama`'

2.3 Building without Visual Studio

You can also develop and build Tama programs without Visual Studio. For this approach, you still need all other resources from section 2.1 Preparations. Follow all steps listed under item 1 in section Open and Build in Visual Studio. You can then open a command line in the project folder and execute `dotnet build` if a solution or project file is in your current directory or `dotnet build <path-to-your-sln-or-csproj>` if your target is not in the same folder or ambiguous.

In this case, we recommend an IDE or editor that allows you to manage C# code, such as Visual Studio Code with suitable extensions, to facilitate an efficient coding experience. You can also define a build script in Visual Studio Code that executes `dotnet build` when using the build shortcut within the tool.

2.4 Load and Run the Tama Program

See section 4.1 for how to load and run the *Tama Program*. Only the **isochronous Tama VM** needs to be enabled for the 'HelloTama' example.

To check for the proper function of the *Tama Program* toggle the register state of

- ...Application.Variables.Booleans[0].

If the Tama Program is running correctly, the following register should change its state accordingly.

- ...Application.Variables.Booleans[1].

2.5 Create Your Own Programs

The following approaches are recommended to create your own *Tama Programs* or *Tama* Solutions.

A) It is possible to have several *Tama Programs* in one VS Solution.
To create a new Tama Program simply copy an existing program file *.cs to the same Solution, adjust the file name and the name of the *Tama* class and make sure to have no collisions in the namespace. With the command **Build** > **Build Solution** all *Tama Programs* of the current solution will be generated.
This approach is recommended if the programs share a similar context for example the programs will be assigned to different drives of the same machine or access the same library.

B) To create *Tama Programs* in a different context it is recommended to copy the whole solution, do the required renaming and delete obsolete files.

3. Authoring Tama Programs

This section shows how *Tama* code is structured and gives some basic recommendation how to design *Tama*

Programs.

3.1 Principal Structure

The TamaProject template (see section 2.2) is used to illustrate the basic structure of a *Tama Program*.

First, some using directives bring in *Tama* and *TAM* register related symbols:

```
using Triamec.Tama.Vmid5;    // used for interpretation of Tama attributes
using Triamec.Tama.Rlid19;   // used for register access
```

Now the *Tama* class can be implemented:

```
[Tama]                       // attribute to indicate the starting point to the Tama compiler
static class TamaProgram     // name of the Tama program
{
```

- The `[Tama]` Attribute indicates the starting point for the compiler. If the VS Solution contains more than one class with the `[Tama]` attribute the classes are handled and built as separate *Tama* classes and for each of them a *Tama Program* will be generated.
- The name of the class can be chosen arbitrarily. The class name will be used for the name of the files generated files e.g. `HelloTama.tama`.
- It is reasonable to set the `static` class modifier, because typically, a *Tama* class is not instantiated. The C# compiler will keep you from accidentally adding instance members.

The *Tama* class contains the following `Main()` method:

```
    // attribute to indicate the entry point for the isochronous task
    [TamaTask(Task.IsochronousMain)]
    // main function, (name is not relevant)
    static void Main()
    {
        // program start
        Register.Application.Variables.Booleans[1] = Register.Application.Variables.Booleans[0];
    }
}
```

- The `[TamaTask(…)]` attribute indicates the entry point to the *Tama Program*. Its argument sets the type of the *Tama Task* (see section 3.2). In the actual case, the task type is set to `IsochronousMain` which causes the `Main()` function to be called every 10kHz cycle.
- The method name itself is not relevant and can be chosen arbitrarily.
- The method body shows a simple read and write application of *TAM* registers.

3.2 Isochronous and Asynchronous Tama Task

As mentioned in section 3.1 with the argument of the `[TamaTask(…)]` attribute, different *Tama* tasks can be selected. The selected *Tama Task* defines how the *Tama Program* is executed. The available tasks are

- *Isochronous Main Task*
- *Asynochronous Main Task*

This section describes the properties of this tasks. As a starting point and also for most application it is recommended to use *Isochronous Main Task.*

3.2.1 Isochronous Main Task

For most applications the *Isochronous Main Task* is the fist choice as it allows real time programming of the drive at 10kHz with the most flexibility.

- With the argument `Task.IsochronousMain` the *Isochronous Main Task* is selected.
- The *Isochronous Main Task* is executed at 10kHz synchronously with the 10kHz task of the controller. As an example the `Main()` function of the 'HalloTama' program in section section 3.1 is called every 0.1ms.
- The program is started with **Enable** and stopped with the **Disable** of the *Isochronous Tama* (see 4.1).
- The task has to be completed within one 10kHz cycle (see also section 3.2.1). If the execution exceeds its time budget within one 1kHz cycle a *ComputingTime* error occurs.

3.2.2 Asynchronous Main Task

The *Asynchronous Main Task* has a cycle time of 2.5kHz. It allows to execute a program over four 10kHz cycles while it can be interrupted by other tasks. It is intended for less demanding application regarding cycle time and to reduce the CPU load of larger *Tama Programs*.

- With the argument `Task.AsynchronousMain` the *Asynchronous Main Task* is selected.
- The *Asynchronous Main Task* is executed over four 10kHz cycles.
- The program is started with **Enable** and stopped with the **Disable** of the *Asynchronous Tama* (see 4.1).
- The task has to be completed within four 10kHz cycles. If the execution exceeds its time budget a *ComputingTime* error occurs.

3.2.3 Multiple Tama Tasks in one Tama Program

It is possible to combine both *Isochronous and Asynchronous Tama Tasks* in one *Tama Program*. The *Isochronous Main Task* is called first. The remaining time budget per cycle is then used to proceed with the execution of the *Asynchronous Main Task*.

Important The *Asynochronous Main Task* has a strictly separated memory space from the *Isochronous Main Task*. The only way to share data between the *Asynochronous Main Task* and the other tasks is via *TAM* registers.

3.3 TAM Registers

TAM Registers are the interface for the interaction between the *Tama Program* and the servo drive and also for the interaction between the *Tama Program* and applications outside of the drive. All registers of the *TAM Register Tree* can be accessed by the *Tama Program*. For more information about *TAM Registers* and the *TAM Register Tree* see also [1] and [2].

Most floating point values of the TAM register tree are single precision (`float`). C# floating point literals are double precision (`double`) by default. To avoid compiler errors the `f` suffix needs to be added to the value to create a `float` literal, for instance:

```
Register.Application.Variables.Floats[0] = 0.5f;
```

3.3.1 TAM Register Layout ID (RLID)

The *Register Layout ID* (*RLID*) identifies different principal versions of the *TAM Register Tree*. The layout of the *TAM Register Tree* could vary between different devices, hardware and firmware versions.

You find the *RLID* of a certain device by selecting the device node in the TAM Register Tree (see Figure 4).

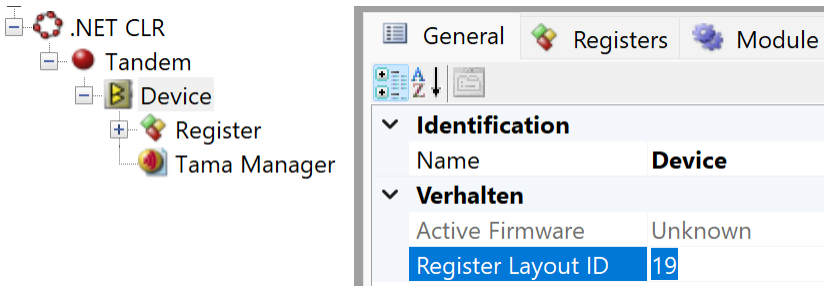

Figure 4: Display of RLID in TAM System Explorer.

For the register access in the *Tama Program* the used register layout must match with the register layout of the device. If the *RLID* of the device is 19 the *TAM Register* class can be accessed in *Tama Programs* with `Triamec.Tama.Rlid19.Register`. The following example shows how the register DcBusVoltage can be read to the local variable uDC:

```
float uDc = Triamec.Tama.Rlid19.Register.General.Signals.DcBusVoltage;
```

To shorten the code line it is common practice to import the register layout name space with the `using` directive.

```
using Triamec.Tama.Rlid19;    // used for register access
```

With this the example above can be written as

```
float uDc = Register.General.Signals.DcBusVoltage;
```

3.3.2 Signals

Signals can be read from the *TAM Register* as shown in the example above. The read value represents the state of the register at the beginning of the current 10kHz cycle.

3.3.3 Commands

The following example shows the usage of the Commands PathPlanner.Xnew and PathPlanner.Command

```
if (Register.Application.Variables.Booleans[0])  // start move if true
{
    Register.Axes_1.Commands.PathPlanner.Xnew = 0.5; // target position
    Register.Axes_1.Commands.PathPlanner.Command =
      PathPlannerCommand.MoveAbsolute; // move command
    Register.Application.Variables.Booleans[0] = false;
}
```

In general, commands are processed with the next 10kHz cycle. An exception is the CommitParameter command (see below).

CommitParameter:

The CommitParameter command is used to activate a bunch of parameters of the same *CommitGroup* (see also section 3.3.4). Depending on the type of parameters and the current task load the CommitParameter command could take more than one 10kHz cycle. A successful commit is acknowledged by the device firmware by setting the value of the CommitParameter command back to `False`.

A new CommitParameter command should only be issued if the state of the commit is `False`.

Inject Command:

Injection points allow to add a signal to certain controller states. For example this could be used to apply position compensation to the encoder signal. The following list shows all available injection registers with the related injection point in Figure 5.

1) PathPlanner.InjectedX
2) PathPlanner.InjectedA
3) PathPlanner.InjectedV
4) Encoder[0].InjectedPosition
5) Encoder[1].InjectedPosition
6) CurrentController.InjectedIq
7) CurrentController.InjectedId
8) CurrentController.InjectedUq
9) CurrentController.InjectedUd

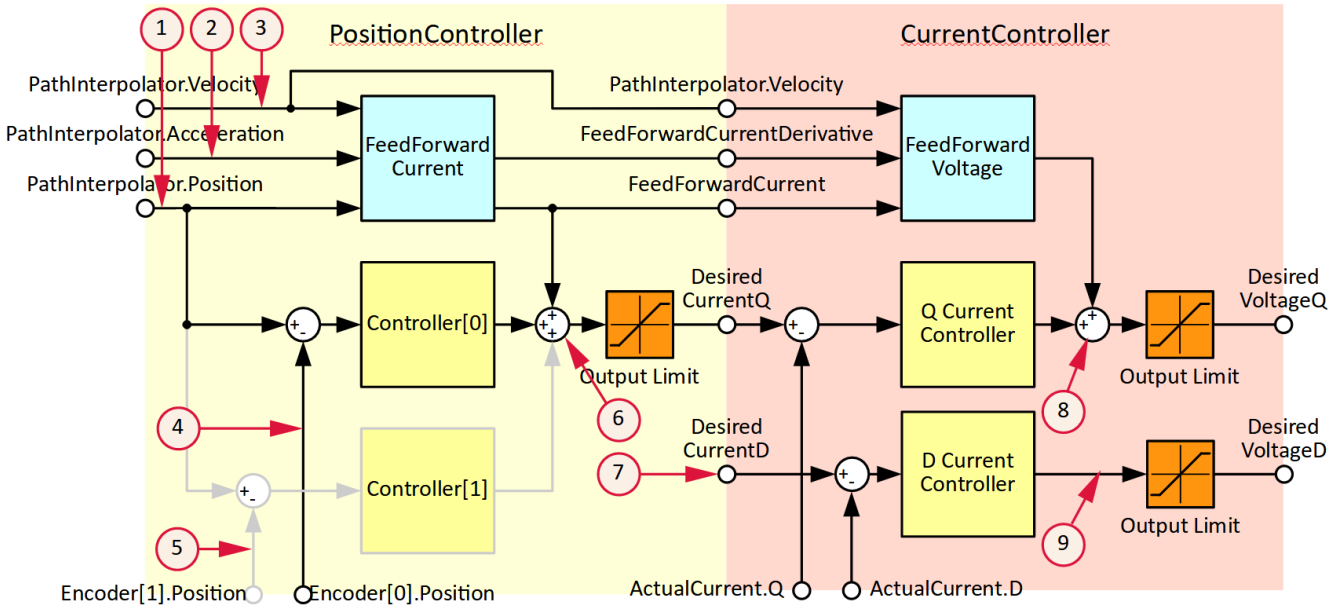

Figure 5: Controller block diagram with injection points.

If the value of an inject command is changed, the internal value is ramped linearly to the commanded value over the following 10kHz cycle.

3.3.4 Parameters

The following example shows how the value of a parameter can be changed with *Tama* code.

```
if (Register.Application.Variables.Booleans[0])  // do parameter change if true
{
   Register.Axes_0.Parameters.PositionController.Controllers_0.Kp = 0.5f; // set new gain
   if (!Register.Axes_0.Commands.PositionController.CommitParameter) // ready for commit?
   {
      Register.Axes_0.Commands.PositionController.CommitParameter = true; // set commit
      Register.Application.Variables.Booleans[0] = false;
   }
}
```

It is important to know, that parameters generally belong to a *Commit Group*. A change of a parameter only becomes active, if this *Commit Group* is committed (see also section 3.3.3). In the example above the *Commit Group* of Kp is `PositionController` and therefore the related commit command is set.

How to Find the Commit Group

The *Commit Group* can be found by selection the desired parameter in the *TAM Register Tree*. The **General** tab in the tab panel shows the *Commit Group* under **Tags > commitGroup**

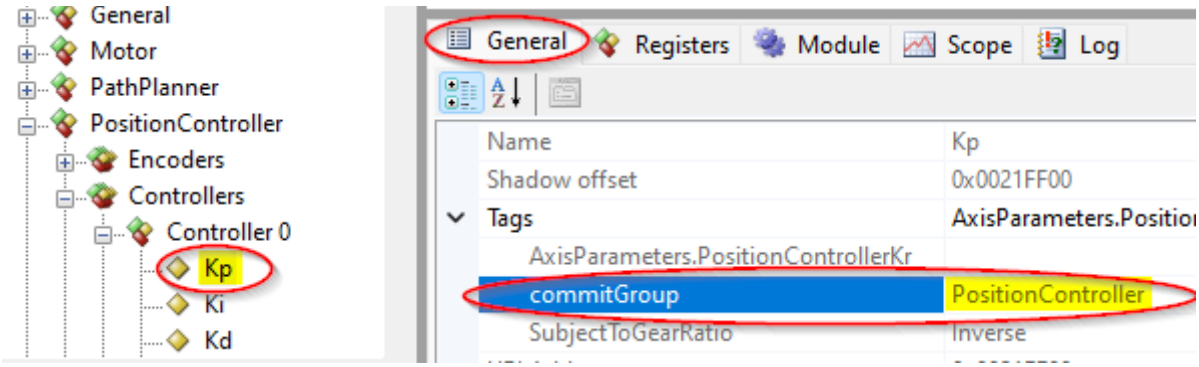

Figure 6: The Commit Group of controller gain Kp is PositionController

Remark If a parameter is part of more than one *Commit Group* a commit of all these groups is required!

3.3.5 Application Registers

The Register.Application branch contains registers used to interact with the *Tama Program* from 'outside' e.g. *TAM System Explorer* or a higher-level control system. Most registers are generic and their purpose is only determined by the *Tama Program* or the programmer respectively.

Parameters and Variables

Register.Application.Parameters and Register.Application.Variables can be used to store `float`, `double`, `integer` and `boolean` values. In contrast to variables, the current value of a parameter is stored persistently if the persistence command is executed and saved in the TAM configuration with the command File > Save TAM Configuration… (see also [1]).

Tables

TAM Tables are a versatile feature which can be used for compensation tables, data collection, data storage etc. It is possible to access the *TAM Tables* in different ways:

- *TAM Register* access with the *TAM System Explorer*
- File access via web browser to exchange tables
- Table access via a *Tama Program*.

The following example shows reading and writing of a table with a *Tama Program*:

```
float myTableValue1 = 3.1415f;
float myTableValue2 = Register.Application.Tables.Small1.Data.Float[0]; // read table
Register.Application.Tables.Small1.Data.Float[0] = myTableValue1; // write table
```

Remark Float, Integer, Double access the same table fields, just the interpretation of the value varies.

For more information about *TAM* tables see [5].

Axes

The Register.Application.Axes registers can be used to implement a coordinate transformation with a *Tama Program*. For more information see [4].

TamaControl

AsynchronousVirtualMachineState and IsochronousVirtualMachineState: These registers reflect whether the respective tasks are executed.

AsynchronousMainCommand, IsochronousMainCommand: These are general purpose registers with the same properties as Register.Application.Variables.Integers (see section above). Only the naming indicates that these registers are intended to control a state machine programmed within a *Tama Program*.

AsynchronousMainState, IsochronousMainState: These are general purpose registers with the same properties as Register.Application.Variables.Integers (see section above). They are intended to reflect the state of a *Tama Program*.
Both states are initialized to `0` after transferring a Tama program.

Caution	The lower 6 bits of these registers and changes to those bits are tracked by the TAM software. Therefore, these registers must not be used for values changing with high frequency. Otherwise, denial of service might occur due to message flooding. Likewise, the Tama program must not change drive and axis states in a high frequent manner.

AsynchronousPc, IsochronousPc: Indicates a recently executed Tama byte code instruction of the asynchronous or isochronous *Tama* task. *PC* is an abbreviation for *program counter*. This value could be helpful for debugging (see section 4.2).

3.4 Errors and Limitations

Compilation errors from the Tama compiler are shown in the *Error List* window. Double clicking will locate the source of the error in most cases. If you don't know how to fix a compilation error, consult the Tama Compiler API and Error Reference [3] which elaborates cause and remedy for each error.

Remark	Unfortunately, for legacy C# projects (like those provided as samples on GitHub with support for legacy Visual Studio 2017), the error identifiers are not shown in the Error List window, but only in the compile output window.
Remark	The Tama compiler is not invoked for checking while typing.

The following sub-chapters describe properties and limitations which must be considered when writing *Tama* code. Please read them carefully, since Tama only supports a sub-set of the C# language.

3.4.1 Initialization

The static members of a *Tama* class are initialized directly after transferring the *Tama Program* to the device. Consequently, the static constructor of a *Tama* class is executed directly after the transfer.

Initialization Rules

Allocate new objects only in the static constructor or in a static member initialization, or in methods called therein. Otherwise, a *TamaOutOfMemory* error will likely occur at runtime. There is no *garbage collector* like on a regular platform targeted by C#.

If you separate the Tama program code in multiple classes, be aware that only the class defining the Tama tasks

may contain a `static` constructor and assignments to `static` field definitions. You can write initialization logic for the other classes in their *instance* constructor, for example.

> **Note** This last limitation exists due to the compiler and may be removed in the future.

No Array Initializers

Initialization like the following is not supported:

```
static float[] positions = {1, 2, 4, 4};
```

Instead, arrays must be initialized element wise:

```
static float[] positions = new float[4];
static TamaProgram() {
    positions[0] = 1;
    positions[1] = 2;
    for (int i = 2; i < 4; i++) {
        positions[i] = 4;
    }
}
```

3.4.2 Memory Isolation of Tasks

Be aware that the asynchronous and isochronous Tama tasks run in a separate virtual environment. Therefore,

- The static class constructor runs twice. Don't alter registers in an incremental fashion at this stage.
- Each task has its own version of static fields.
- Each task has a separate object heap. Objects allocated in the static class constructor will actually be allocated twice.

> **Caution** The only possibility to share data between the tasks is via registers.

C# experts can imagine that the code will behave as if they had specified `ThreadLocal<T>` as the field type.

3.4.3 No Register Caching

One of the most obvious peculiarities of Tama programs is their repeated dereferencing of registers throughout the code like a mantra:

```
static void ReadParameters() {
    _wait = Register.Application.Parameters.Integers[0];
    _repeats = Register.Application.Parameters.Integers[1];
    _stepSize = Register.Application.Parameters.Floats[0];
    _stepVelocity = Register.Application.Parameters.Floats[1];
    _startPositionPositive = Register.Application.Parameters.Floats[2];
    _startPositionNegative = Register.Application.Parameters.Floats[3];
}
```

Actually, you could write this method like this and get the exact same binary:

```
static void ReadParameters() {
    var parameters = Register.Application.Parameters;
    var integers = parameters.Integers;
    _wait = integers[0];
    _repeats = integers[1];
    var floats = parameters.Floats;
    _stepSize = floats[0];
```

```
    _stepVelocity = floats[1];
    _startPositionPositive = floats[2];
    _startPositionNegative = floats[3];
}
```

While this might result in less characters used, it doesn't seem to improve readability.

More important, you cannot pass such local variables to methods like this:

```
static void Read(ITamaArray<int> array) {
    _wait = array[0];
    _repeats = array[1];
}
static void ReadParameters() {
    if (true) {
        Read(Register.Application.Parameters.Integers);
    } else {
        Read(Register.Application.Variables.Integers);
    }
}
```

This ends up with an error TAMAC0002 currently.

Namely, it's not possible to parameterize the axis number of a drive in order to write code which works for either axis:

```
Register.Axes_0.Commands.PathPlanner.Xnew = step;
```

Compound Assignments

Compound assignments (+=, *=, …) to registers in one method *may* lead to the error TAMAC0002 in certain cases, since such assignments might be transformed into registers saved into temporary variables by the C# compiler.

While the following snippet compiles (with the C# compiler at the time of writing),

```
Register.Application.Variables.Integers[0] *= 5;
Register.Application.Variables.Integers[1] *= 5;
```

this slightly modified version fails with TAMAC0002:

```
Register.Application.Variables.Integers[0] *= 5;
Register.Application.Parameters.Integers[1] *= 5;
```

Rewriting a statement without compound assignment by repeating the register on the right-hand side will work around the limitation:

```
Register.Application.Variables.Integers[0] = Register.Application.Variables.Integers[0] * 5;
Register.Application.Parameters.Integers[1] = Register.Application.Parameters.Integers[1] * 5;
```

Advice Due to this limitation, we recommend avoiding compound assignments with registers involved.

3.4.4 No Support for 64-Bit Integers and Other Types

While there exist a handful of 64-Bit integer registers, Tama isn't currently able to process them.

Likewise, the attempt to work with strings or other classes from the base class libraries will end up in a pile of compiler errors.

Specifically, Tama programs aren't designed to output text.

Unordered Floating Point Values

When comparing floating point values, there exist two different ways when unordered values (NaN) are encountered. One way always yields `true` for unordered comparisons, the other way always yields `false`. For reference, compare the documentation for the cgt and cgt.un CIL opcodes.

Tama doesn't make a difference here. Therefore, a Tama program might execute differently with unordered floating point values compared to when the same algorithm was executed on the x86 architecture.

3.4.5 No Recursive Calls

In a Tama program, a method must not call itself. This restriction also holds transitively, when a method *A* calls method *B*, method *B* calls method *C* and method *C* calls method *A*, for instance.

3.4.6 No Generics and Tuples

Generic types aren't supported since the runtime cannot create specialized generic types.

This also impedes the use of tuples, for example to handle multiple return values in a method.

Instead of a code like this

```
void Foo() {
    var (x, y) = Bar();
    // use x and y
}
(int x, int y) Bar() {
      // compute x and y
      return (1, 2);
}
```

use several *out* parameters instead:

```
void Foo() {
    Bar(out int x, out int y);
    // use x and y
}
void Bar(out int x, out int y) {
    // compute x and y
    x = 1; y = 1;
}
```

3.5 Combine Multiple Programs

A drive can only call one synchronous *Tama Program* or/and one asynchronous *Tama Program*. Therefore it is recommended to structure your programs in such a way, that combining them in one *Tama Program* is possible with little effort.

Best practice is to structure your programs in classes. These classes implement for example a tick method. The classes are then instantiated in the *Tama* Class and the tick method is called by the isochronous or asynchronous *Tama* task.

Special care has to be taken, to avoid overlapping register access between different classes.

3.6 Build Process

The following steps explain how to build a *Tama Program* with *Microsoft Visual Studio* (see also Figure 7):

1. Go to **Project** > **[**Project Name**] Properties** > **Build** and check if **Optimize Code** is checked.
2. To build the *Tama Program* click menu **Build** > **Build Solution**
   For each *Tama* class which is part of the solution a *Tama Program* is generated.
3. In the Output window (**View** > **Output**) check if the build succeeded and check if there are no build errors listed in the Error List (**View** > **Error List**).
4. If the build succeeds the following files are generated for each *Tama* class:
   `[ProjectName].[NameSpaceName].[TamaClassName].tama`
   `[ProjectName].[NameSpaceName].[TamaClassName].asm`
   The build files can be found at the following two locations
   - In the **Solution Explorer** right click the Project and click **Open Folder in File Explorer** and navigate to: '`...Tama\bin\Debug`' or '`...Tama\bin\Release`' depending on the used build configuration.
   - In the TAM System Explorer click **File** > **Open Workspace Folder** and navigate to '`..\Tama`'.

The `*.tama` file contains the byte-code of the compiled *Tama Program*. This is the file which is finally loaded to the drive to run the *Tama Program*.

The `*.asm` file contains the plain-text version of the compiled *Tama Program*. The `*.asm` file could be useful for code analysis and debugging (see also 4.2).

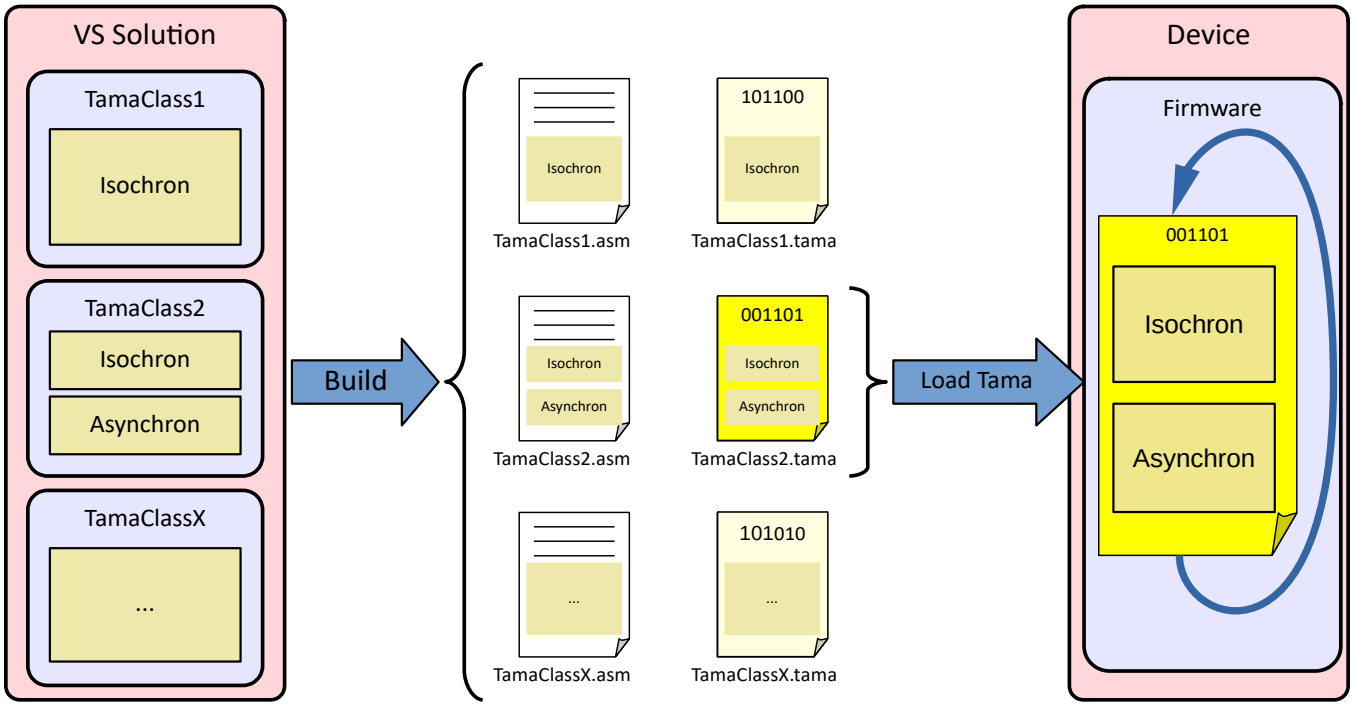

Figure 7: Build and load Tama Programs.

4. Running Tama Programs

In order to run a Tama Program, it first needs to be transferred to the device. Enabling one or more of its tasks are separate steps in this process.

4.1 Loading and Enabling

To load a *Tama Program* to the device, the **Tama Manager** in the register tree of *TAM System Explorer* is used. The **TamaManager** is located below the device node in the *TAM Register Tree*.

The following steps are needed to load and run a *Tama program* on a device:

1. Open the context menu with a right click on the *Tama Manager* (Figure **8**).
2. Click **Assign Tama program…** and select a *Tama program* which has the file extension `*.tama`. Click **Open**. The program will now be transferred to the device.
3. Click **Enable isochronous Tama VM** or **Enable asynchronous Tama VM** depending on if the isochronous or the asynchronous *Tama* task or both has to be started.

To remove the *Tama program* from the device, use the **Dismiss Tama program...** menu item.

With **Download Tama program...** an already loaded *Tama Program* is loaded again.

Warning A running *Tama Program* can strongly influence the behavior of a device. In case of unexpected behavior, first check if a *Tama Program* is enabled and consider disabling it.

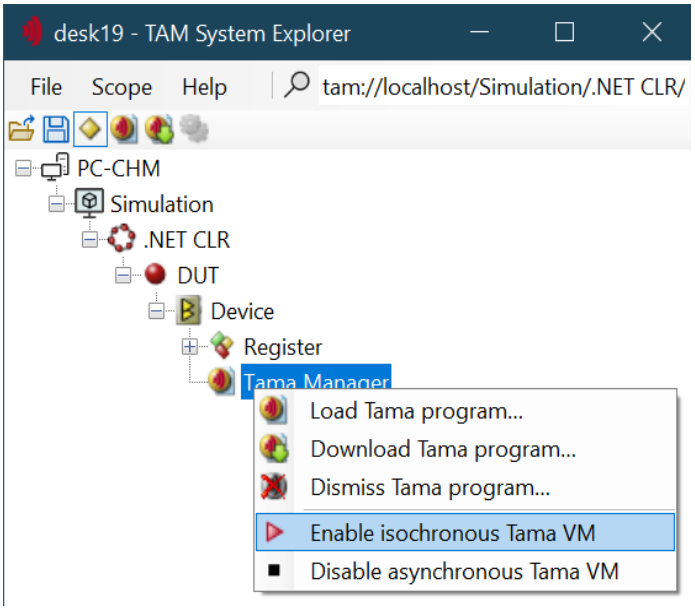

Figure 8: Context Menu of the Tama Manager. The menu entries also indicate whether the virtual machines are currently enabled or not.

4.1.1 Saving a Tama Program Persistent on the Device

If a new *Tama program* is loaded, the steps to persist the *TAM Configuration* have to be executed again, to also persist the new *Tama Program [1]*.

A persisted *Tama Program* will automatically be loaded after a restart of the device but remains disabled. To automatically enable a *Tama program* at start-up see section 4.1.3.

> **Note** Remember that you only persist the binary form of the Tama program. Therefore, we strongly recommend using a version control system to maintain the Tama program source code.

4.1.2 Saving a Tama Program in a TAM Configuration File

When the *TAM Configuration* is saved (see [1]) while a *Tama Program* is loaded, the *Tama Program* is also saved to the configuration.

4.1.3 Auto Start

A persistent Tama Program can be enabled automatically at startup.

To enable the *Isochronous Tama* at startup set

- General.Parameters.EnableIsochronousTama = True.

To enable the *Asynchronous Tama* at startup set

- General.Parameters.EnableAsynchronousTama = True.

As with other parameters, you must persist these on the drive to be available at startup.

4.2 Debugging

Tama Programs do not permit interactive debugging. Instead, the correct function of programs is examined by monitoring the affected *TAM* registers. Scoping of such registers can be very helpful for debugging.

To observe internal variables of the program, the variables have to be made visible by assigning them to an appropriate *TAM* register e.g. Application.Variables.Floats.

```
// used for debugging
Register.Application.Variables.Floats[0] = myDebuggingVariable;
```

Debugging Tama Runtime Errors

In case of a Tama runtime error, for instance due to a null divisor or if no memory is left, the Tama program is no longer executed and the axes are disabled. In this situation, the TAM register Application.TamaControl.IsochronousPc (or AsynchronousPc respectively) points to the next bytecode instruction.

With the help of the corresponding assembly file (`*.asm`) and the program counter this failed instruction can be identified and it is possible to relate the failed instruction to the *Tama* code.

Example:

A Tama Program causes the following runtime error:

> *ERR 6976 Tama division by zero IsochronousMain PC 0xef*

Likewise, the register IsochronousPc shows the value `0x000000EF`.

Considering the corresponding `*.asm` file, one can easily identify the suspicious division in the method

`DemoFunction()`.

```
# Method Utilities.MovingAverage.DemoFunction
0x000000e8   ldc       0x3f800000          # 1
0x000000ea   ldloc     0x00000000          # load this : MovingAverage
0x000000ec   ldfld     0x00000005          # load field divisor : float
0x000000ee   div.r4                        # float division
0x000000ef   ret       0x00000001          # return from routine
```

By looking at the related *Tama* code (see below) it is apparent, that the runtime error was caused because the member variable `divisor` was never assigned to a value unequal to zero.

```
float divisor;

/*****************************************************************
* Runtime error demo
*****************************************************************/
public float DemoFunction()
{
    return 1.0f / divisor;
}
```

4.3 Tama Runtime Errors

If an instruction of a Tama Program can not be processed, for instance due to e.g. because of a null divisor, a device error is issued, the program is no longer executed and the controllers turn off.

Error name	Description
TamaOutOfMemory	The program memory became full during heap allocation. Reduce the number or size of objects allocated. Object heap, static variables and the call stack are all maintained in the same memory.
TamaDivisionByZero	An attempt was made to divide by zero.
TamaNullReference	An object property was requested, but there was no object reference set.
TamaIndexOutOfRange	An array element index was outside the range of the array.
TamaCorruptedState	Tama Program state was corrupted. This value is returned when an unknown operation code is encountered. This indicates a defect in the code supporting Tama programs and should be reported to Triamec.
ComputingTime	The limit of the computing time is exceeded. Consider chapter 4.5: Performance Tuning

Often, such an error cannot be acknowledged since it will be immediately reproduced. Dismiss the Tama program in such a circumstance by using the context menu of the *Tama Manager* (see Figure 8).

4.4 Task Load Monitoring

The task load generated by the *Tama Program* is tracked in the register General.Signals.SystemLoad.DurationTamaIso

The firmware monitors the load of the whole 10 kHz task in the register General.Signals.SystemLoad.Duration10kHz

In case of an *Isochronous Tama*, a ComputingTime error is issued if Duration10kHz exceeds 70 µs.

In case of an *Asynchronous Tama*, a ComputingTime error is issued if the task is not finished after four 10 kHz cycles and Duration10kHz exceeds 70 µs.

4.5 Performance Tuning

Consult this chapter if you have grown your Tama program up to a complexity where available computing time is exceeded.

The Tama program is programmed in C#. Compilers translate this *source code* into *bytecode instructions* which can be examined in the assembly file `*.asm` (see section 3.6).

As a rule of thumb, the drive is able to process up to 1500 *bytecode instructions* per 10 kHz cycle. The total number depends on the configuration and state of the servo drive.

Each command in the *Tama* source code generates a number *bytecode instructions*. As an example, the following table shows the number of *bytecode instructions* required for some C# statements:

Code	# bytecode instructions
`Register.Application.Variables.Floats[0] =` `  Register.Application.Variables.Floats[1];`	4
`Register.Application.Variables.Floats[0] =` `  Math.Sin(Register.Application.Variables.Floats[1]);`	5
`Register.Application.Variables.Floats[0] =` `  Register.Application.Variables.Floats[1] * Math.PI;`	6

`Register.Application.Variables.Floats[0] =`
`  Register.Application.Variables.Floats[1] *`
`  Register.Application.Variables.Floats[1];`
	7

For example, the second example above may be repeated approximately 300 times in one 10 kHz cycle without causing a computing time error.

5. Advanced Topics

This chapter provides some more insight into the tools used when writing Tama programs and alternatives ways to use them.

5.1 Tama File Compilation

Refining the build process in figure 7, we can see that multiple tools are involved:

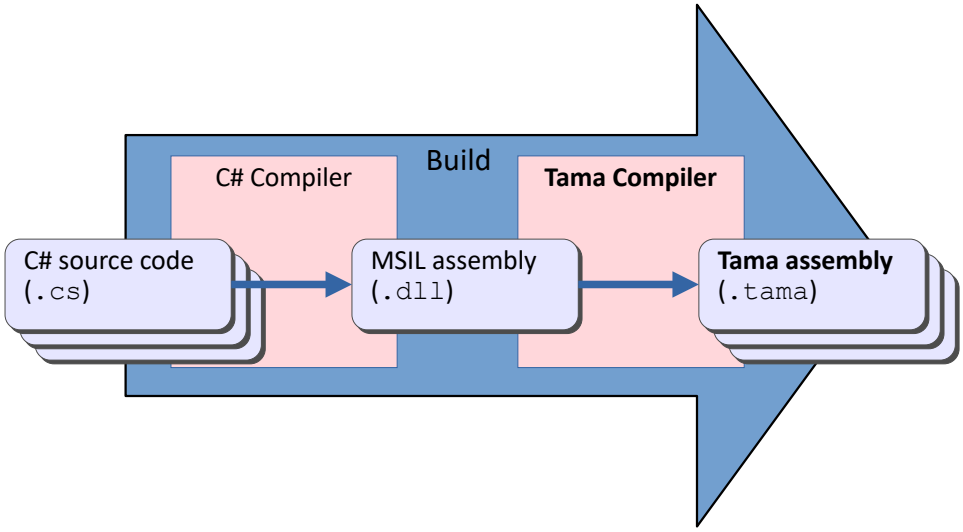

Figure 9: The Tama Compiler takes the output of the C# compiler and produces Tama programs.

The Tama Compiler is a piece of software provided by Triamec which produces Tama programs. The compiler is integrated into the build process of your project by means of the *Triamec.Tools.TamaCompiler NuGet package*, available on nuget.org. Additionally, the *Triamec.Tam.TriaLink* or *Triamec.Tam.EtherCAT* NuGet packages are necessary to provide the register layout libraries.

As described in chapter 3.6, the Tama programs are placed by the compiler in the project's output and within the Triamec workspace.
You may suppress the latter by adding an MSBuild property `TamacCopyOutputsToWorkspace` and setting it to `false` in the project file.

Notice Since the Tama compiler takes a managed library as input, it is possible in principle to use any programming language targeting .NET to produce Tama programs, for example, Visual Basic or C++/CLI. Please contact us if you plan to use such a language, as these scenarios haven't been tested.

The following subsections describe ways to invoke the Tama Compiler outside a development environment.

5.1.1 Compiler Shortcut

With the installation of the TAM Software, a "Tama Compiler" link will be created in the Triamec workspace (like figure 10).

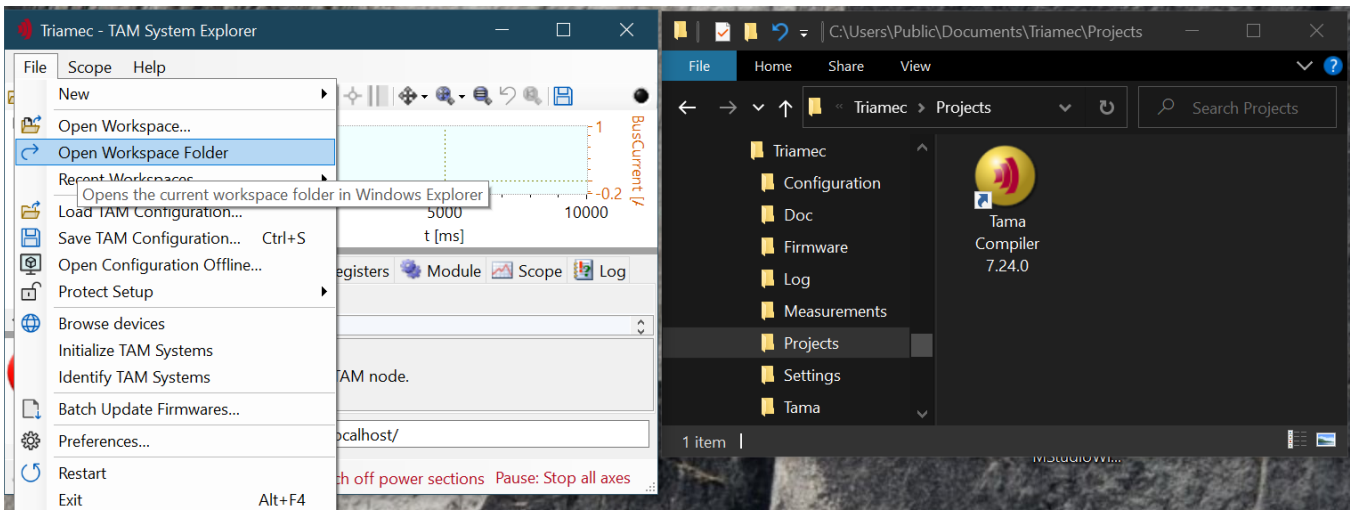

Figure 10: Tama compiler shortcut within the Triamec workspace, opened via the TAM System Explorer menu
Just drag a C# Tama program source file over the icon for compilation.

Dragging one or several C# source files over the link invokes the compilation. The output is placed beneath the first provided source file.

Feel free to copy the link to your favorite entry point, say your desktop.

5.1.2 Command Line Interface

The Tama compiler executable can be accessed via the command line interface or the Tama compiler shortcut. As shown in figure 3, the Tama compiler accepts a .NET dynamic link library as input.

Calling

```
tamac /?
```

on a command line will show a list of options, given the executable is on the path:

```
Tama Compiler 5.11.1 version 5.0.0.0
Copyright © 2022 Triamec Motion AG

/optimize[+|-]  Whether to produce an optimized Tama binary Default value:'+' (short form /o)
/verbose        Whether to produce verbose output (short form /v)
/nologo         Default value:'-' (short form /n)
@<file>         Read response file for more options
<assembly>      Dynamic link library containing Tama tasks
```

Caution The referred Tama, register and other libraries need to be beneath the compiled library with the correct version. Otherwise, unexpected errors will occur.

However, for the developer's convenience, the Tama compiler also accepts C# source file(s).

For options, type

```
tamac test.cs /?
```

on the command line. The output will be placed beneath the first provided source file.

5.1.3 Legacy Change History

This is the former change log of the Triamec.Tools.TamaCompiler NuGet package.

As of version 5.13.4, these changes are documented in the NuGet package's release notes.

Version 5.13.3

- CHG: Rename `Math.Fabs` to `Math.Abs`, lining up with `System.Math.Abs`.
- BUG: Addresses a TAMAC0013: “Value with unsupported bit width used.” false positive when the compiler encounters a 64-bit operation prior the first register layout reference.
- BUG: Fix an issue with the emitted binary when using structs or doubles in static fields, which could lead to the program rejected as illegal by the runtime for no obvious reason.

Version 5.13.2

- CHG: Limits number of generated errors to 100.
- BUG: Handles errors upon the attempt to use types from the .NET core libraries more gracefully.

Version 5.13.1

- BUG: Restores compatibility with Microsoft Visual Studio < 2022, regressed in 5.13.0.

Version 5.13.0

- NEW: Supports .NET Core projects, invocation via the `dotnet build` tool, and portable symbol files.
- CHG: Deprecates the axis coupling tasks `Axis1Init`, `Axis1Coupling`, `Axis2Init` and `Axis2Coupling`. Remove the task attributes from the original methods and add a new task like this:

```
static bool _initialized;
[TamaTask(Task.IsochronousMain)]
static void Main() {
   if (Register.Axes_0.Signals.General.AxisState == AxisState.TamaCoupledMotion) {
       if (!_initialized) {
           _initialized = true;
           Initialize();
       }
       Couple();
   } else _initialized = false;
}
```

Version 5.12.0

- NEW: Supports bit field registers.

For notes about older releases, look at a copy of the predecessor of this user guide, the Tama Compiler User Guide.

5.2 API

This chapter isn’t about authoring Tama programs, but about transferring them to the device and controlling their execution.

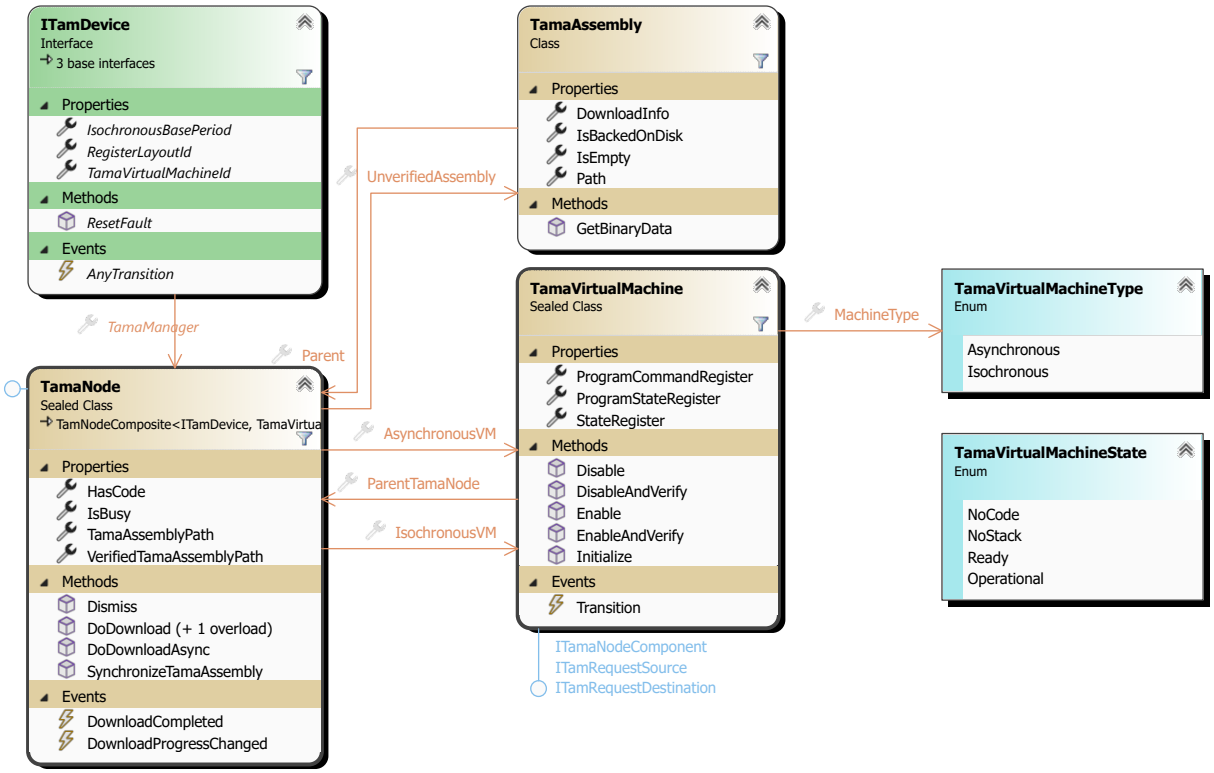

Figure 11: The Triamec.Tam.Tama API

In the API, there is the `TamaManager` root instance within an `ITamDevice` providing all the functionality related to Tama programs.

Tama Programs are represented by the `TamaAssembly` class.

The two instances running the isochronous and asynchronous tasks are the *Tama Virtual Machines*, represented by a respective `TamaVirtualMachine` class.

5.2.1 Transfer to the Device

Tama programs are located within the device's memory reserved for registers, called *code memory*.

To prepare a transfer, set the `TamaNode.TamaAssemblyPath` to the path of a Tama program. A transfer will not be permitted if the RLID or VMID of the Tama program don't match with the device.

Start the transfer with one of the `DoDownload` or `DoDownloadAsync` methods.

If you've set the path to the empty string, the virtual machines are stopped and the current Tama program is dismissed.

Otherwise, the new Tama program is

1. transferred
2. verified
3. both `AsynchronousMainState` and `IsochronousMainState` registers reset to `0`, and
4. the static constructor of the Tama program is run.

5.2.2 Virtual Machines

Tama programs are interpreted by the firmware using Tama virtual machines. At start-up, no code is loaded. When a Tama program is transferred to the device, it gets verified. Both virtual machines get initialized. In order

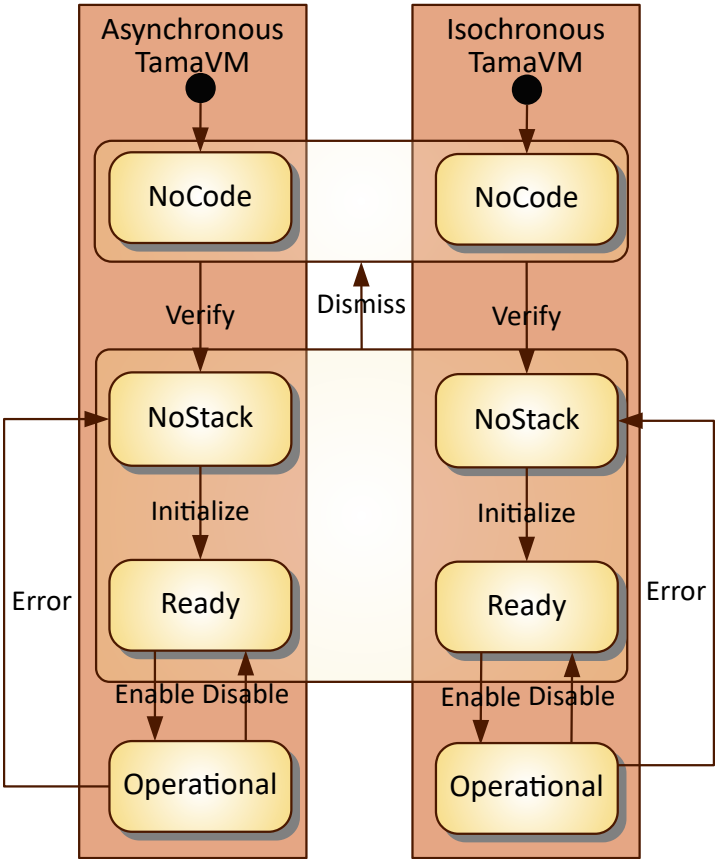

Figure 12: Tama Virtual Machine State Diagram

to execute the task, the virtual machine needs to be enabled. It may also be disabled, and the Tama program as a whole may be dismissed.

5.2.3 Controlling and Monitoring

A task of a Tama program typically waits for a command before it starts executing. Often, a state is maintained for control flow, monitoring and debugging.

Both state and command registers are integers, but consider to use constants or enumerations for state and command values.

For additional input/output, general purpose floating point and integer registers are reserved for Tama programs.

Refer to chapter 3.3 for more information about about registers.

The `TamaVirtualMachine.Transition` event allows to track the state changes of a Tama program, given that the predefined state register is actually used as provided.

Caution	Don’t change the state too often, as explained in the chapter about TamaControl registers.

Refer to the TAM API Developer Manual [2], chapter 6.3: Requests for general considerations about transition events.

5.3 Firmware Evolution

As new features are added to the device firmware, its interface, the register layout, changes. This does not change the Register Layout ID. Therefore, the TAM API and the firmware releases need to roughly correspond.

That said, the changes to the register layout are applied in a backward compatible manner, so you don't need to take care about this.

Using a newly introduced register in a Tama program causes error 6973 when transferring it to a device running an aged firmware release [6]. This happens immediately at the end of the transfer, the device doesn't execute any of such code in this case.

Likewise, a register introduced in a new firmware release isn't available in an old TAM Software version as well.

In both cases, consult the firmware release notes to identify a more feasible TAM Software release. Please update the `Triamec.Tam.TriaLink` or `Triamec.Tam.EtherCat` NuGet package, respectively, to the corresponding version. The TAM Software release is referred to in the release notes of the NuGet package. An overview of all versions is given in the TAM Software release table [7].

References

[1] "Servo Drive Setup Guide", ServoDrive-SetupGuide_EP021.pdf, Triamec Motion AG, 2023
[2] "TAM API Developer Manual", SWNET_TamApiDeveloperManual_EP048.pdf, Triamec Motion AG, 2023
[3] "TAMA Compiler API and Error Documentation", SWTAMA_ApiAndErrorReference-5.11.1_EP001.chm, Triamec Motion AG, 2022
Accessed using TAM System Explorer, menu **Help > Documentation > Software**.
[4] "Transformation PathPlanning", AN113_TransformationPathPlanning_EP001.pdf, Triamec Motion AG, 2017
[5] "Triamec Drive File System", AN124_Filesystem_EP005.pdf, Triamec Motion AG 2023
[6] "Drive messages", AN102_DriveMessages_EP, Triamec Motion AG, 2024
[7] "TAM Software Release Table", SWNET_ReleaseTable, Triamec Motion AG, 2024

Revision History

Version	Date	Editor	Comment
001	2024-05-17	DG, chm	Consolidate Tama Compiler user guide, TAM System Explorer handling of Tama programs and API documentation in one single document
004	2025-01-24	chm	Renamed Isochronous/AsynchronousStackOffset registers to IsochronousPc etc.
005	2026-08-25	ns	Update from VS 2017 to 2026, .NET 4.8 to 10. Add section 'Building without VS'

Copyright © 2026
Triamec Motion AG
All rights reserved.

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.

Source Q:\doc\Software\SWTAMA\
Destination T:\doc\Org\Templates
Owner chm