Triamec Tables

Application Note 125

A description Triamec application tables and how they are used.
Tables are available in firmware 4.11.x and newer and are accessible with a *TAM System Explorer* version 7.15.0 or newer, or with a browser. This application note originated from a division of AN124.

Table of Contents

Document	AN125_Tables_EP
Version	001, 2025-02-12
Source	Q:\doc\ApplicationNotes\
Destination	T:\doc\ApplicationNotes
Owner	ns

1 Overview

Tables can be used in a user real-time application (*Tama*) or firmware based features such as cogging or axis compensation. They can store a large amount of numerical data which can be looked up fast. Currently we support 8 small tables (~16'000 entries) and 2 large tables (~2’000'000 entries)[1] for user applications and one large table for each firmware based feature on each axis.

2 Structure

A table contains a header and data. The table is a binary file, the (numerical) data is a flat sequence of binary LittleEndian 32bit entries. 64bit entries are realized by occupying two 32bit entries. The header specifies how to look up the correct data slice based on one, two or three inputs. This data slice can then be interpreted as integer, float or double value.

2.1 Header

The header fields are shown in the following table.

Word number	Type	Register name	Description
0	Bool	Persistent	0=Table is Volatile, 1=Table is Persistent
1	Int32	-	Must be 0
2	Int32	Type	{0=User, 5=CoggingCompensationV1, 10=AxisCompensationV1}
3	Int32	ChecksumMode	{0=Ignore, 1=Check, 2=Calculate}, see chapter 2.3
4-15	Int32	Checksum	The SHA-3-384 checksum with NIST padding, set zero before calculation.
16-17	Int64	Date	The date in 64 bit POSIX format
18	Int32	-	Must be 0
19	Int32	Id	A table ID given by the user
20-35	String	Description	A description string given by the user
36	UInt32	RowSize	The number of words in a row
37-39	Int32	-	Must be 0
40 41 42 43	UInt32 Int32 Float32 Float32	Dim1.Size - Dim1.StartValue Dim1.Distance	The size of the table in the first dimension Must be zero The position of the first data point of this dimension The distance between data points in this dimension
44 45 46 47	UInt32 Int32 Float32 Float32	Dim2.Size - Dim2.StartValue Dim2.Distance	The size of the table in the first dimension Must be zero The position of the first data point of this dimension The distance between data points in this dimension
48 49	UInt32 Int32	Dim3.Size -	The size of the table in the first dimension Must be zero

1 With firmware version < 4.2 max number of entries is ~500’000 float values.

50 51	Float32 Float32	Dim3.StartValue Dim3.Distance	The position of the first data point of this dimension The distance between data points in this dimension
52-63	Integer32	-	Must be 0

2.2 Data

Because the (numerical) data is a flat sequence of binary LittleEndian 32bit entries (64bit possible by occupying two 32bit entries), the data structure of the table is quite flexible. Because of this flexibility, several parameters from the header are used for a correct interpretation of the data.

- Each Dimensions *StartValue* and *Distance* are used to translate from the scalar input to the correct lookup index in this dimension. In many table applications, e.g. the firmware-based axis compensation, interpolation is used between the indices.

Hint: The lookup strategy described above minimize the storage space needed by only saving the lookup values itself but requires these values to be equidistant in each dimension used.

- The *RowSize* specifies how many 32bit words are used for the lookup in the flat data sequence for a given set of dimension input values.

Hint: Make sure your *RowSize* is even when you want to interpret (part of) the data as double. This is necessary because Double requires two 32bit entries of the table. This means that Double[i] accesses the same 32bit words as Integer[2*i] and Integer[2*i+1] or Float[2*i] and Float[2*i+1]

For a standard one dimensional table of Float values, set:

- Header.RowSize = 1
- Header.Dim1.Size = number of Float values in the table
- Header.Dim2.Size = 1
- Header.Dim3.Size = 1

For a standard one dimensional table of Double values set:

- Header.RowSize = 2
- Header.Dim1.Size = number of Double values in the table
- Header.Dim2.Size = 1
- Header.Dim3.Size = 1

For a three dimensional table of Float values set:

- Header.RowSize = 1
- Header.Dim1.Size = number of entries in the first dimension
- Header.Dim2.Size = number of entries in the second dimension
- Header.Dim3.Size = number of entries in the third dimension

2.3 Checksum Calculation

A checksum may be attached to the header. This checksum is tested in the drive if *ChecksumMode* is set to *Check*. If a file is transmitted to the drive with *ChecksumMode* = *Calculate*, the drive will change the

ChecksumMode to *Check* and then calculate the checksum itself. This is useful if the user does not want to calculate the checksum himself. By reading back this file, he gets a file with a checksum value, and the *ChecksumMode = Check*.

The checksum is calculated with the SHA-3-384 method. Before calculation, zero the checksum in the header, add NIST-type of padding, then calculate the SHA3-384 hash of the file. Finally write it back into the header.

When using the TAM API implementation, the checksum handling is done automatically also on the PC, see 3.4.

3 Accessibility and Interfaces.

3.1 Browser

The most straightforward way to handle tables on a drive is to access the drive's web interface with a browser. The interface allows you to get tables from the drive to your PC and vice versa. For more details, refer to AN124.

3.2 TAM System Explorer

Reading table values is done using Application.Tables.General.Data

- Source select the table to be accessed
- Index select the index of the table array
- Integer shows the 32bit integer value of the table at the index chosen
- Float shows the 32bit float value of the table at the index chosen
- Double shows the double value of the table at the index chosen

To access the header or commands for table Small1 for example, use register Application.Tables.Small1. This contains the following elements. Contrary to the data of a table, its header can be manipulated and committed via the System Explorer.

Hint: Due to internal limitations, only the first 1000 entries can be read out in the TAM System Explorer.

Command

The command register allows to run one of the following table commands.

Command	Description
Commit	This command is described in detail below.
Reload	Reload a persistent table from the persistent memory.
Erase	Erase the persistent memory of this table and set values to default.

Committing a table calculates the size of the table from the header parameters (see chapter Structure below). After this the table and its header can be read from the file system. If the file is persistent (see

chapter 2.1) the header and data are saved to the persistent memory.

Warning: Committing a persistent table can wear out the persistent memory. If a certain limit has been reached, committing a table is denied with an error message and the user must wait some time before trying again.

Changing the table header without committing does not change the size of the table as visible from the file system nor the header seen from the file system nor change persistent memory. Repeat commit after changing the header and the new header will be visible from the file system and in persistent memory.

Changing the table data after committing the table changes the temporary memory (RAM) and will be immediately visible over the file system but does not update persistent memory (FLASH).

Header

The table header is described in detail in chapter 2.1.

State

This shows the current state of the table. 3 means the table is ready.

3.3 Tama Code

From Tama a table is accessed the same as using the *TAM System Explorer* with two exceptions:

First: The data of the table can be accessed directly. It is important to note that the data is not saved as float, integer or double but can be interpreted in all these types. To set/get a specific float value at index 10000 of table Small1, for example, simply use the code:

```
Register.Application.Tables.Small1.Data.Float[10000] = 1.234f;
float myFloat = Register.Application.Tables.Small1.Data.Float[10000];
```

Note: Data.Float[100] and Data.Integer[100] and Data.Double[50] point to the same table item, as described in 2.2

Second: Unlike in the *TAM System Explorer*, all data of a table can be read in Tama.

Write Table

Generate the values of the table by something like the following code example, which sets the table to a value, that depends on the function customerFunction(pos1, pos2, pos3) at the three dimensional position (pos1, pos2, pos3) for two float and one double value per three dimensional position

```
int size1 = Register.Application.Tables.Small1.Header.Dim1.Size;
int size2 = Register.Application.Tables.Small1.Header.Dim2.Size;
int size3 = Register.Application.Tables.Small1.Header.Dim3.Size;
int rowSize = Register.Application.Tables.Small1.Header.RowSize; // 4 in this example
for (int k = 0; k < size3; k++) {          // loop over the third dimension
   for (int j = 0; j < size2; j++) {       // loop over the second dimension
      for (int i = 0; i < size1; i++) {    // loop over the first dimension
         int indexFor32bitValues = (i + size1 * (j + size2 * k)) * rowSize;
```

```
            float pos1 =  Register.Application.Tables.Small1.Header.Dim1.StartValue +
                          Register.Application.Tables.Small1.Header.Dim1.Distance * (float)i;
            float pos2 =  Register.Application.Tables.Small1.Header.Dim2.StartValue +
                          Register.Application.Tables.Small1.Header.Dim2.Distance * (float)j;
            float pos3 =  Register.Application.Tables.Small1.Header.Dim3.StartValue +
                          Register.Application.Tables.Small1.Header.Dim3.Distance * (float)k;
            Register.Application.Tables.Small1.Data.Float[indexFor32bitValues] =
               customerFunctionFirstValue(pos1, pos2, pos3);
            Register.Application.Tables.Small1.Data.Float[indexFor32bitValues + 1] =
               customerFunctionSecondValue(pos1, pos2, pos3);
            Register.Application.Tables.Small1.Data.Double[(indexFor32bitValues + 2) / 2] =
                  customerFunctionSecondValue(pos1, pos2, pos3); // This will occupy word 3 and 4
        }
    }
}
```

To save the table persistently it is important to assign the correct values to the dimension and row sizes to allow the firmware to calculate the correct memory space for the table. The size is calculated as:

```
size = 4 * (64 + Header.RowSize * Header.Dim1.Size * Header.Dim2.Size * Header.Dim3.Size)
```

Call the following code **once**. Don't call it frequently to prevent flash wear!

```
Register.Application.Tables.Small1.Header.Dim1.Size = size1;
Register.Application.Tables.Small1.Header.Dim2.Size = size2;
Register.Application.Tables.Small1.Header.Dim3.Size = size3;
Register.Application.Tables.Small1.Header.RowSize = rowSize;
Register.Application.Tables.Small1.Header.Persistent = true;
Register.Application.Tables.Small1.Command = TableCommand.Commit;
```

Read Table

Below is an example on how to read a 1D table saved on the drive under Small1. The input signal used for the lookup is Axes[0]/Signals/PathPlanner/PositionFloat. The lookup value is linearly interpolated. This template could serve as a prototype for a 1D axis compensation. We would never the less recommend to use the built-in axis compensation features and tools shown in AN140 unless you need specific features that are currently not implemented (e.g. higher order interpolation).

```
float posAbs = Register.Axes_0.Signals.PathPlanner.PositionFloat;
float startValue = Register.Application.Tables.Small1.Header.Dim1.StartValue;
float distance = Register.Application.Tables.Small1.Header.Dim1.Distance;
float posRel = posAbs – startValue;
float idxFloat = posRel / distance;
index = (int)Math.Floor(idxFloat);
offset = idxFloat – index;
float c0 = Register.Application.Tables.Small1.Data.Float[idx];
float c1 = Register.Application.Tables.Small1.Data.Float[idx + 1];
float lookupValue = c0 + offset * (c1 - c0);
```

3.4 TAM API

Tables can be send to and get from the drive by software either via the TAM API or by directly using HTTP GET and PUT to the respective URI. Please note that after a successful transfer of a table to the drive, additional time is necessary to validate and potentially persist the table as described in 3.2. You can check the state of this processing via the *State* register of the table.

Note: The TAM API interface explained in this AN is for general tables that are not related to a firmware feature. For tables associated with a firmware feature (e.g. axis compensation), use the specific API for this feature.

The API components for the creation and handling of tables are associated with the `Triamec.Tam` namespace.

TamTbl

The `TamTbl` is the object representation of a table. A `TamTbl` cannot be instantiated directly via the constructor but with `TamTblFactory.Create(…)`. Other ways of creating a `TamTbl` object are transferring an existing table from a drive to the PC by using `TamTbl.GetFromDrive(…)` or loading an existing table from a `.TAMtbl` file with `TamTbl.Deserialize(…)`. These methods internally also use `TamTblFactory.Create(…)` for checksum handling, see below.
The `TamTbl` can then be used in your code, saved to a file with `TamTbl.Serialize(…)` or transferred to a drive with `TamTbl.SendToDrive(...)`

TamTblFactory

The `TamTblFactory` is responsible for the creation of `TamTbl` objects. The checksum is handled the same way as in the firmware. On one hand, it is checked when a table is deserialized, which occurs when reading a table from a file or a transferring a table from a drive to the PC, in order to be sure that the table is not corrupted. On the other hand, the checksum is calculated and added to a table when serializing, which is necessary to save the table as a file or when transferring the table from a PC to a drive. The checksum is then checked when (re)loading the file with the API or by the firmware of the drive respectively.

If a table has been generated with another tool, it is possible to encounter tables with *ChecksumMode = Ignore*. This is generally not recommended to do, as corrupted tables cannot be recognized as such easily. The API will then also ignore the checksum.

3.5 Low Level Software Interface

External software can access the filesystem using the IP-address shown in chapter 3.1. This is especially useful if the PC is connected with the drive over its auxiliary Ethernet port. For further information on how to read and write files with this interface, consider AN124.

Revision History

Version	Date	Editor	Comment
001	2025-02-12	ns	Initial release

Triamec Motion AG
Lindenstrasse 16
6340 Baar / Switzerland

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

Disclaimer