Forschungsinstitut

umati Transformation Engine

API documentation (Release Candidate)

VDW-Forschungsinstitut e.V.

Copyright © 2021

Table of contents

Copyright © 2021

1. umati Transformation Engine

1.1 Foreword

The VDW-Forschungsinstitut e.V. is currently working with partners and its members to create a specification of a TransformationEngine.

This document describes the development process and the decisions made to develop the first API. Currently this is designed to be implemented in C to allow the application on any necessary platform and device.

Application Warning Notice

This DRAFT with date of issue 2021-10-01 is being submitted to the public for review and comment. Because the final API Specification may differ from this version, the application of this draft is subject to special agreement.

Comments are requested:

- preferably as a file by e-mail to g.goerisch@vdw.de
- or in paper form to VDW-Forschungsinstitut e.V., Lyoner Straße 18, 60528 Frankfurt

Partners

- Bridgefield (Magdeburg)
- iT Engineering Software Innovations (Pliezhausen)
- Pragmatic Minds (Kirchheim/Teck)

Machine Tool Builders

- CHIRON Group (Tuttlingen)
- DMG Mori (Pfronten)
- EMAG (Salach)
- GROB Werke (Mindelheim)
- Gebr. Heller (Nürtingen)
- Liebherr Verzahntechnik (Kempten)
- Trumpf Werkzeugmaschinen (Ditzingen)
- United Grinding Group (Bern)

Copyright © 2021

1.2 Glossary

API	Application Programming Interface
DC	Data Client
Data Source	Actual OEM data source or controller of manufacturing machine which is accessible by a south API compatible data client implementation
North API	'north bound' API to be used to provided data and functionality from transformation engine to a connected OPC UA server
OPC UA	OPC Unified Architecture: machine to machine communication protocol for industrial automation
South API	'south bound' API to be used to integrate data sources in transformation engines
TE	Transformation Engine

Copyright © 2021

1.3 Purpose and Scope

The OPC UA Server with the OPC 40501-1 UA for Machine Tools companion specification on the north side of the stack is the access for OPC-UA clients on machine or factory level which can communicate to local MES/PPS/ERP or cloud services. The Data Clients on the south side of the stack are the different sources and sinks for data and machining functions, i.e. these blocks may be for example NCs, PLC, IPCs and “intelligent” sensors or actuators in the machine. In most cases, all these sinks and sources provide non-standardized, proprietary interfaces. Hence, the objective of the proposed TransformationEngine is to transfer and convert data and commands between the OPC UA Server on the north side of the stack and the data clients on the south side. As an important requirement, the conversion must be provided in a standardized way.

This document describes the requirements on this standardization. Further details on the functionality of the different components in the stack and on single requirements are given below.

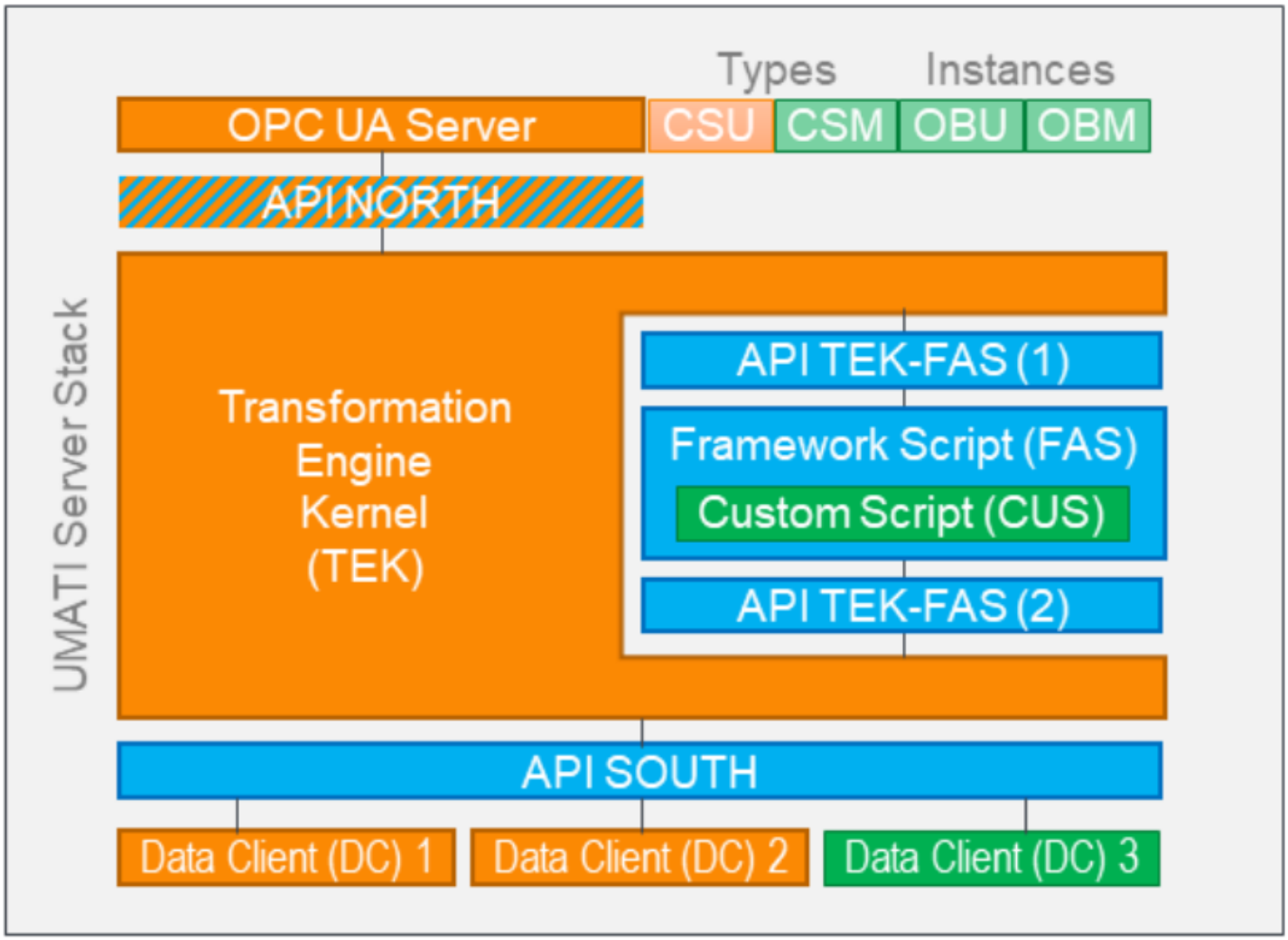
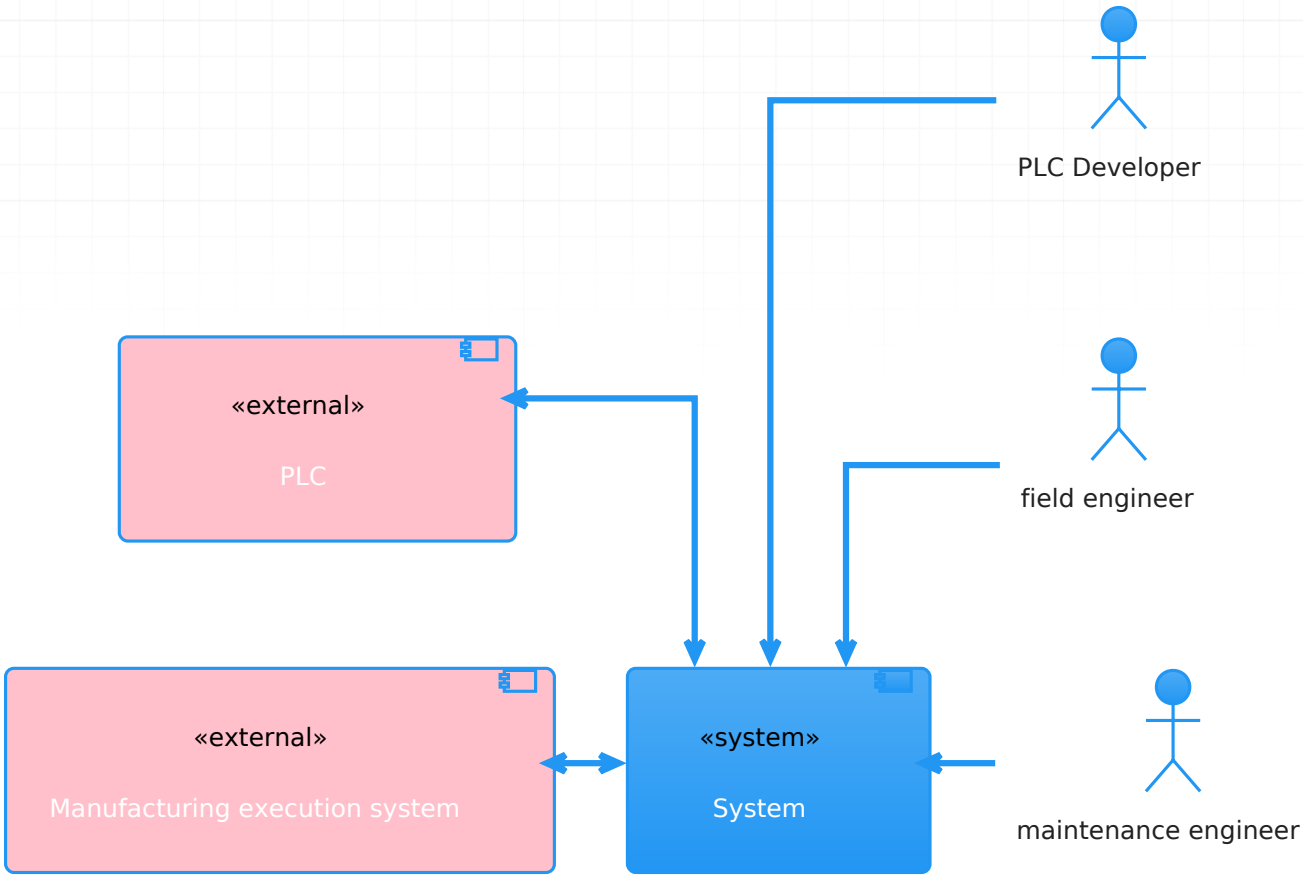


Figure 1 - Conceptual design of the TE

Copyright © 2021

1.4 System context

This is a simplified description of the system context the South API specification, defined here and in accompanying header files, is intended to be used in.

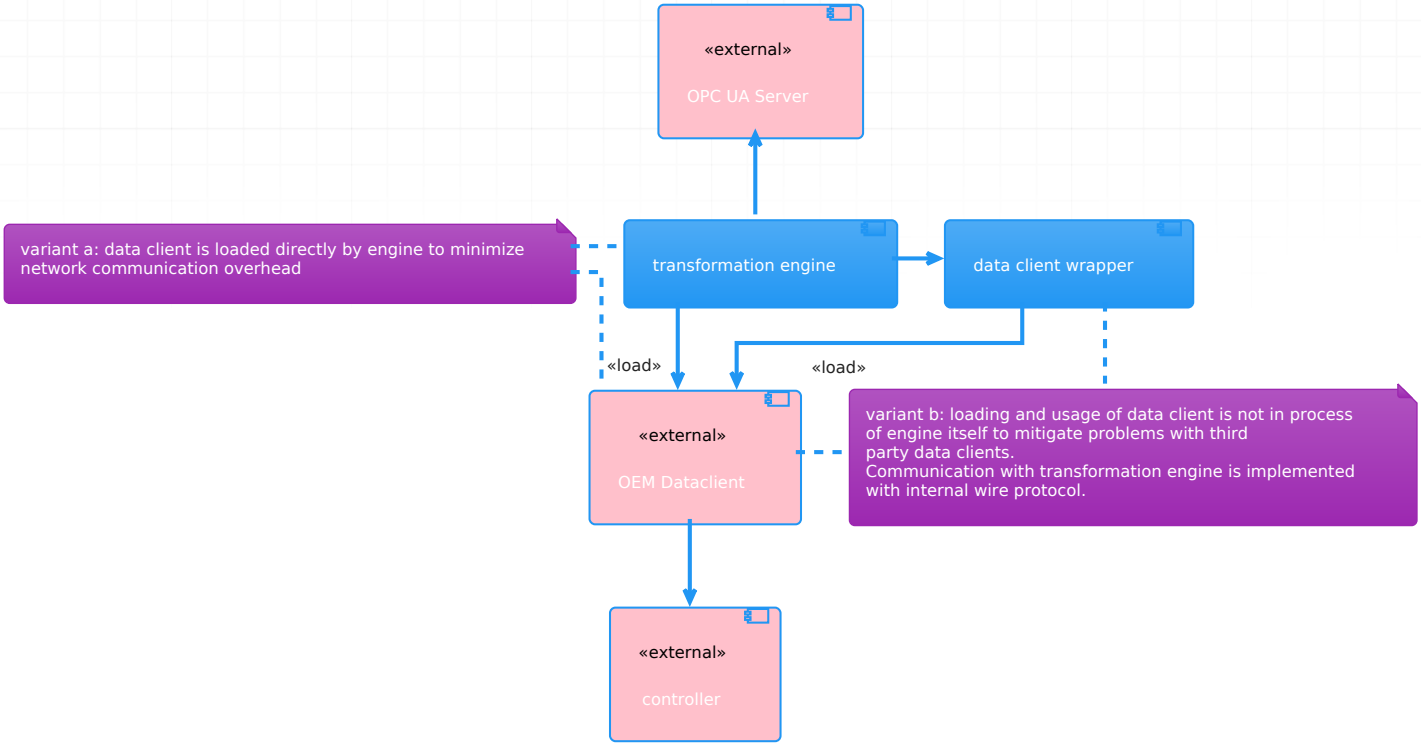


Copyright © 2021

1.5 Container diagram

This diagram describes possible Transformation Engine implementation containers assumed/anticipated by this specification.

The term `container` in this context is used for separately running applications native to the target platform (see also: C4 Model).



For reference: The South API is intended to be used for either directly connecting the `transformation engine` with an `OEM data client` or indirectly through a separate `data client wrapper`.

The `OPC UA server` and `OEM data client` are marked as `external` in this diagram to symbolize different implementation responsiblities.

It is expected to have deployment scenarios where `transformation engine` and `data client wrapper` are implemented by the same implementor and combined with 3rd party `OEM data client` and `OPC UA server` implementations, which is why a South API definition and later a North API definition are needed in the first place.

Copyright © 2021

1.6 Architectural Drivers

The API design is mainly guided by the following selection of architectural drivers for now.

criteria	meaning in project context
extensibility	multiple different data clients must be supported by a single transformation engine
availability	implementing data clients must be easy and not require special runtimes or tools
portability I	implementing data clients must not require OPC UA
portability II	chosen data client technology must at least enable Windows and Linux host systems
reusability	implemented data clients must be usable by different transformation engine implementations

1.7 Decisions summary

The current South API is mainly based on decisions described in the table below.

Apart from these decisions certain parts of the OPC UA specification were also used. These are mentioned and linked explicitly from the actual South API specification.

The impact of these references was carefully limited to not add additional dependencies for data client implementors (see `portability I` criteria) and mainly used to simplify later interactions between data clients and a North API that will be more reliant on OPC UA specifications.

number	problem to solve	decision	state
001	interface type	stdcall C interface	accepted
002	api artifacts	only c header file(s)	accepted
003	read access	select fields to read	accepted
004	threading model	Async data client API	accepted
005	error codes	Same error type for all methods	accepted
006	text encoding	UTF-8	accepted
007	memory allocation	no ownership transfer	accepted
008	configuration structure	separate files for each client	accepted
009	data client model access	data client registers model at TEK	accepted
010	OPC UA data source integration	assume a South API independent direct connection from OPC UA data sources to north bound OPC UA to be defined in later iteration	accepted

Newly defined architectural drivers and or requirements from other sources might require decisions to be reevaluated.

Copyright © 2021

2. Decisions

2.1 001 Type of the interface (Socket, shared library or managed code)

- Status: accepted
- Deciders: partners

Context and Problem Statement

What is the type of communication between the data client (DC) and the transformation engine (TE)? Should we force a network protocol or is it better to use in-process communication?

Decision Drivers

- supportability: data clients can be easily implemented by a sensor/tool manufacturer
- reusability: data clients can be integrated into different TE implementations
- availability: Language for TE implementations are not unnecessarily restricted
- portability: minimum supported OS are Windows, Linux (POSIX combatible)

Considered Options

- Socket/interprocess communication
- stdcall C interface
- dotnet/java interface

Decision Outcome

stdcall C interface

Positive Consequences

- no restrictions for TE implementations
- managed runtimes as well as native code can access stdcall binaries

Negative Consequences

- The implementation language of the DC is restricted to a language which is compilable to native machine code.
- The data client implementation may influence the stability of the TE
- The platforms a data client can be used at depends on the manufacturer of the controller/sensor.

Copyright © 2021

Pros and Cons of the Options

Socket/interprocess communication

The data client is running as a separate process on the same or a different machine and is accessible via a custom network protocol.

- Good, because dataclient is completely independent runtime environment from TE
- Good, because a faulty data client cannot crash the TE
- Bad, because it requires a complex binary protocol
- Bad, because additional delays for inter process communication must be considered
- Bad, because of overall system complexity due to multiple running processes

stdcall C interface

The data client is realized as a shared library (.DLL on Windows, .so on *ix) which is loaded into the TEK process.

- Good, because all imaginable TE implementation languages can load C libraries
- Good, because it is supported on every sensible operating system and processor architecture
- Good, because it has nearly no communication overhead
- Good, because a socket interface can be added easily if ever needed
- Bad, because manufacturer must provided different implementations for all supported platforms
- Bad, because the implementation might be more complex than a Java/CLR implementation
- Bad, because the DC code runs in the process of the TEK and may influence its stability

dotnet/java interface

Use a common managed runtime like dotnet CLR, Java Virtual Machine or a Python interpreter to be able to limit the required effort to provide data client binaries for all supported platforms.

- Good, because implementation might be easier for a given manufacturer
- Good, because one data client runs on every platform
- Bad, because only platforms with existing Java/CLR runtime are supported
- Bad, because it limits the implementation languge to the languages available on the CLR/JVM

Copyright © 2021

2.2 002 API artifacts

- Status: accepted
- Deciders: partners

Context and Problem Statement

Which artifacts must be provided to data client implementors?

Decision Drivers

- extensibility: how easy is it to integrate new data clients
- upgradability: how much must be done to switch to new API versions
- usability: how easily can errors in the custom scrips and in the configuration be detected

Considered Options

- c header file(s) and shared library
- only c header file(s)

Decision Outcome

only c header file(s)

Pros and Cons of the Options

c header file(s) and shared library

Data client implementations can use c header files and also a library implementing TEK functions called from data clients.

- Good, because the execution flow is easily understandable (less usage of callback functions)
- Good, because the interface is easier to read and understand
- Bad, because it binds the data client to a specific south API binary
- Bad, because all TEK implementations are bound to a specific south API binary
- Bad, because backwards compatibility of binary **and** header files must be considered

only c header file(s)

Data client implementations use c header files containing among other definitions a function pointer based definition of TEK functions called from data clients. An instance of a TEK interface is then provided to a data client on start up.

Copyright © 2021

i.e.:

```
interface tek {
    // ...
    void notify_event(data_client,...);
    void notify_condition(data_client,...);
    // ...
};
```

- Good, because no binary dependency for data clients
- Good, because no binary dependency for TEK implementations
- Good, because backwards compatibility only for header files must be considered
- Bad, because it may be more difficult to handle function pointers instead of direct library calls

Remarks

Wrapping managed (Java, CLR) data client implementations with native wrappers

In case an already existing basic managed data client library exists where a complete rewrite with native C code would be too much of a hassle there is the possibility/necessity of wrapping managed libraries with the proposed C south API for the transformation engine.

This may be done as separately started processes or by using the integration apis:

- JVM
- CLR

Copyright © 2021

2.3 003 Granularity of read access of data clients

- Status: accepted
- Deciders: partners

Context and Problem Statement

Due to a very heterogenous system landscape with multiple different data sources we need to define a read access API which can be realized for all data sources.

Decision Drivers

- supportability: data clients can be easily implemented by a sensor/tool manufacturer
- usability: allow easy access to multiple values
- consistency: enable synchronized/consistent access to related values

Considered Options

- Only single values/fields can be read
- Complete state is read at once
- Select values/fields to read

Decision Outcome

Select values/fields to read

This option was choosen even though it might require more effort when implementing data clients to have a simple and at the same time more versatile API.

Positive Consequences

- the other two options can be easily covered by corresponding selections over single fields or the complete state
- data clients are responsible to return a consistent state of all the variables requested while the TEK is responsible to request all variables which need to be consistent with only one call

Negative Consequences

- a slightly increased complexity in the implementation of the DC (compared to the first option), especially concerning error handling

Copyright © 2021

Pros and Cons of the Options

Only single values/fields can be read

The data client provides a callback for single fields.

```
bool (*data_client_read_callback)(
          data_client dc,
          field_handle field,
          struct read_result* result);
```

- Good, because read access is simple
- Good, because data client must not handle consistent read access
- Good, because read error handling is field specific
- Bad, because overhead in TE when accessing multiple fields
- Bad, because additional APIs are required to enable consistent access to multiple values (i.e. some kind of transaction mechanism)

Complete state is read at once

The data client provides a callback for the complete current state of the data source.

```
bool (*data_client_read_callback)(
          data_client dc,
          struct read_result** results,
          int* read_results_count);
```

- Good, because no separate synchronization is required in the API
- Bad, because can return more data than needed by the TEK
- Bad, because data client must handle read access synchronization
- Bad, because error handling for read errors is more complex

Select values/fields to read

The data client provides a callback which supports selection of values/fields to read.

```
bool (*data_client_read_callback)(
          data_client dc,
          field_handle* fields,
          size_t fields_size,
          struct read_result* results);
```

- Good, because no separate synchronization is required in the API
- Good, because returns exactly the data needed by the TEK
- Good, because single field read access can be covered by the same callback
- Good, because the memory management is done completely in the TEK
- Bad, because data client must handle read access synchronization
- Bad, because error handling for read errors is more complex

Copyright © 2021

2.4 004 Data client threading model

- Status: accepted
- Deciders: partners

Context and Problem Statement

Most NC/PLC/Sensor communication involves network communication which is inherently asynchronous. There are however existing communication libraries for some of the communication targets which only support blocking synchronous calls.

To be able to support both types of communication we need to select a threading model for data clients which makes it equally easy to implement synchronous and asynchronous data clients.

The selected threading model is intended to be used by `data client wrapper` implementations for encapsulating access to data clients.

Hint: The requirements concerning file/block transfer, events and methods have to be evaluated separately.

Decision Drivers

- supportability: data clients can be easily implemented by a sensor/tool manufacturer
- reusability: data clients can be integrated into different TE implementations
- flexibility: async and sync data clients must be supported

Decision Outcome

Async data client API

Pros and Cons of the Options

Single threaded access

Data clients can only be accessed in a single thread.

- Good, because data clients can be implemented more easily
- Good, because async calls can always be mapped to blocking synchronous calls when the underlying data connection is async
- Bad, because threads will be blocked because of I/O waits
- Bad, because of performance loss

Multi threaded access

Data clients can be concurrently accessed from multiple threads.

- Good, because maxiumum performance can be achieved when accessing async data clients
- Bad, because even strict single threaded data clients need to implement access synchronization

Async data client API

Data clients have an async API where data is returned with call back methods.

Copyright © 2021

On initialization, a data client indicates whether it supports multithreaded calls. If a data client indicates single threaded operation only, the TEK must respect this and call the data client sequentially. See `struct dataclient_capabilities`.

e.g.:

```
enum ASYNC_RESULT
{
    ACCEPTED, // request was accepted and will be executed asynchronously
    RETRY_LATER, // another request is in progress, can not execute this request
    INVALID, // can not execute due to request errors
    COMPLETED // request was completed synchronously
};

enum ASYNC_RESULT read(
    DC dc,
    TEK tek,
    long request_id,
    struct FIELD* items_to_read,
    size_t number_of_items);

/// TEK callback from DC
void read_progress(
    DC dc,
    long request_id,
    size_t current_count);

void read_result(
    DC dc, long
    request_id,
    enum RESULT result,
    struct FIELD_RESULT* result_data);
```

- Good, because async data clients can use async data access
- Good, because single threaded clients are explicitly allowed to block and call callback functions while function is running and thus map async call to a sync call without affecting async data clients

Copyright © 2021

2.5 005 Error codes definition

- Status: accepted
- Deciders: partners

Context and Problem Statement

We need to define a common set of error codes for all API calls to enable better diagnostics in case of runtime errors.

Decision Drivers

- ??

Decision Outcome

Same error type for all methods

Pros and Cons of the Options

Same error type for all methods

e.g.

```
typedef int SOUTH_API_RESULT;

#define SOUTH_API_SUCCESS 0
#define SOUTH_API_UNKNOWN_VARIABLE 1
#define SOUTH_API_WRONG_ADDRESS 2
```

- Good, because this is the usual way of defining error codes in C
- Good, because this is directly supported in C
- Bad, because there is no diffentiation between different classes of api functions. Error codes to be expected have to be declared in comments.

Copyright © 2021

One error type for each class of functions

e.g.

```
enum READ_RESULT
{
   READ_RESULT_SUCCESS=0
   READ_RESULT_UNKNOWN=1
};

enum WRITE_RESULT
{
    WRITE_RESULT_SUCCESS=0
    WRITE_RESULT_WRONG_ADDRESS=1
};

enum CONNECT_RESULT
{
    CONNECT_RESULT_SUCCESS=0
    CONNECT_RESULT_WRONG_ADDRESS=1
};
```

- Good, because error types to expect are declared expicitely
- Bad, because of very long names due to missing namespace support in C
- Bad, because writing log messages becomes hard
- Bad, because error handling is case to case for each function

Copyright © 2021

One error type for each class of functions but the error codes are the same

e.g.

```
#define ERR_SUCCESS 0
#define ERR_WRONG_ADDRESS 200
#define ERR_UNKNOWN_VARIABLE 300

enum READ_RESULT
{
   READ_RESULT_SUCCESS= ERR_SUCCESS,
   READ_RESULT_UNKNOWN=ERR_UNKNOWN_VARIABLE
};

enum WRITE_RESULT
{
    WRITE_RESULT_SUCCESS = ERR_SUCCESS,
    WRITE_RESULT_WRONG_ADDRESS = ERR_WRONG_ADDRESS
};

enum CONNECT_RESULT
{
    CONNECT_RESULT_SUCCESS = ERR_SUCCESS,
    CONNECT_RESULT_WRONG_ADDRESS = ERR_WRONG_ADDRESS
};
```

- Good, because error types to expect are declared explicitly
- Good, because most values can be reused in error handling
- Bad, because of very long names due to missing namespace support in C
- Bad, because compiler does not really enforce the enum range

Copyright © 2021

2.6 006 How should text values be encoded?

- Status: accepted
- Deciders: partners

Context and Problem Statement

When transfering text values and identifiers it is important to have a common meaning of the character values to avoid wrong presentation to the user and non matching identifiers.

Decision outcome

Arrays of `char` which represent text values are encoded as UTF-8.

No plausible alternatives were presented.

Pros and Cons of the Options

ASCII or Latin1 Codepage

- Bad, because a limitation to a specific 8-bit charater set does not meet the requirements of modern internationalized software and is not compatible with the OPC-UA specification.

UTF-8

- Good, because not transcoding is needed when sending it to th OPC-UA server.

UTF-16

- Bad, because it consumes more memory.
- Bad, because 16 bit char type is not really compatible.

UTF-32

- Bad, because of even more memory consumption.

Copyright © 2021

2.7 007 What are the rules for memory management?

- Status: accepted
- Deciders: partners

Decisions

Memory ownership is never transfered between TEK and DC.

The interface partner which allocates memory is also responsible for freeing it.

Memory ownership must not be mixed in data structures

... otherwise it could not be freed safely.

The memory referenced by pointers which are passed to functions need not be valid longer than the function call.

As consequence it is not feasible to copy pointers only for later use. If data needs to be persisted for later use, a deep copy is neccessary. If ever any exception from this rule is needed it must be documented explicitely.

Memory passed to a function call is not modified from the caller while the function call is in progress.

That means the caller is responsible for thread safety of passed memory.

Memory passed to a function call is not modified from the callee, except it is explicitely allowed.

The interface uses the const qualifier to mark unmodifiable memory.

```
void pass_unmodifiable_memory(field_data const * data)
```


Copyright © 2021

2.8 008 What are the common requirements for the data client configuration?

- Status: accepted
- Deciders: partners

Assumptions

- A data client should not access the file system directly
- The TEK reads the configuration from a TEK specific storage and forward it to the data client
- The TEK is responsible to identify the correct plugin for the each configuration file
- Different vendors may already supply configuration files in proprietary formats.
- The DC configuration of a specific client should be the same for different implementations of the TEK.
- The configuration of the TEK itself may be implementation specific.
- The complete data client configuration and data client "discovery" configuration must be compatible between alle TEK implementations.
- OS specific limitations may restrict the location of shared libraries and configuration files.

Decision

- Separated configuration files of TEK and Plugins
- Separated configuration files of plugin and data clients
- Data client configurations are not necessarily nested below plugin configurations

Ideas

The TEK uses different directories for data client shared libraries and data client configuration.
The directory structure for the `binaries` is implementation specific to the TEK implementation. The structure of the `configuration` on the other side is equal in all TEK implementations.

Possible configuration structure - does not influence the south api

1. All plugin configurations are in a subdirectory 'data_client:plugins'
2. Each Plugin has its own subdirectory. Its name is not constrained.
3. Each plugin configuration directory contains a 'plugin.json'
4. All client/device configurations are in a subdirectory 'data_clients'
5. Each client has its own subdirectory. Its name is not constrained.
6. Each client directory contains a client.json configuration file.
7. A client.json or plugin.json can contain references to additional files.

Copyright © 2021

example/proposal of the directory structure

```
.../binaries/
      |
      + data_client_plugins/
        |
        + dc-s7.dll
        + dc-ads.dll
.../configuration/
      |
      + data_clients/
      |  |
      |  + NC/
      |  |  |
      |  |  + client.json
      |  + PLC/
      |  |  |
      |  |  + client.json
      |  |  + symbols.xml // Siemens specific
      |
      + data_client_plugins/
         |
         + siemens-s7
         |  + plugin.json // TEK-Spec conformant
         |  + lic.bin // Siemens specific
         + beckhoff-ads/
            |
            + plugin.json
```

```
// plugin.json
{
    "tek.plugin-name":"Siemens Step7",
    "tek.plugin-library": "dc-s7", // loads dc-s7.so resp. dc-s7.dll
    "tek.additional-files": {
        "secret-license-file": "lic.bin"
    }
    // optional plugin specific properties
}

// client.json
{
    "tek.plugin" : "siemens-s7", // references directory inside data_client_plugins
    "tek.additional-files" : {
        "symbols" : "symbols.xml"
    }
    // ... optional plugin and client specific properties
}
```


Copyright © 2021

```
struct additional_file
{
    char* name; // name of additional file
    char* content; // content of additional file
};


struct configuration
{
    char * config; // json content
    additional_file* additional_files;
    size_t additional_files_count;
};
```

Configuration options

TEK and plugins

one common configuration file for tek and data client

- Bad, because ist may be large and hard to edit.
- Bad, because the configuration of the data clients can not be copied to another TEK without modifications

different configuration files for tek and data client

- Good

Plugin and data client

different configuration for plugin and device

- Good

combined configuration of plugin and device

- Bad, because it is not easy to disable a device

Plugin and data client hierarchy

device configuration below plugin configuration

- Bad, because a single device can not be copied/moved between different TEKs.

device configuration in own directory

- Good, because naming conflicts are impossible

Copyright © 2021

2.9 009 data client model access

- Status: accepted
- Deciders: partners

Context and Problem Statement

TEK implementations need to know the model and capabilities of a data client. i.e. which fields can be read, are there any events which can be emitted by the data client or which alarms and conditions are provided.

This data model of a data client is needed to know what to expect from the data client at runtime and to validate script interactions with a given data client.

Decision Drivers

- simplicity: the execution flow ist easily understandable
- understandability: how ist this problem solved in other similar contexts (e.g. data binding in OPC-UA SDK's or data access in known PLC access libraries)
- extensibility, upgradability: how easy and backwards compatible can new features be integrated
- usability: how easily can errors in the custom scrips and in the configuration be detected

Considered Options

- TEK reads data client model/capabilities directly from data client
- data client registers model at TEK

Decision Outcome

data client registers model at TEK

Pros and Cons of the Options

TEK reads data client model/capabilities directly from data client

The data client implements methods for all relevant model parts which are used by the TEK.

```
interface south_api {
    List<field> get_declared_field();
    List<field> get_declared_events();
};
```

- Good, because the execution flow is easily understandable (less usage of callback functions)
- Good, because the interface is easier to read and understand
- Bad, because data clients must implement functions for model parts which are not supported by a specific data client (i.e. events might not be of interest)
- Bad, because it may be unclear which functionality in the DC is mandatory and which is optional

data client registers model at TEK

The TEK interface contains method definitions for registering fields, events and other model parts.

Copyright © 2021

With this option a data client can decide which model it actively must support and is not forced to implement empty functions.

```
interface tek {
    // ...
    RESULT register_field(data_client, field, type);
    RESULT register_event(data_client, event);
    // ...
};
```

A data client implements a single method `register` containing code responsible for registering all model related parts at the TEK.

```
interface data_client {
    RESULT register(tek);
}
```

- Good, because the data client must not implement code for not used model parts
- Good, because the interface can be extended in a way that existing DC can be reused without modification
- Good, because it is assumed that the required amount of code in the DC is smaller
- Bad, because the data client implementor must deal with function pointers

Copyright © 2021

2.10 010 OPC UA data source integration

- Status: accepted
- Deciders: partners

Context and Problem Statement

There are potential existing data sources already providing an OPC UA interface.

Here we want to outline how integration of OPC UA data sources can be accomplished and if these are treated differently to other data clients due to the higher complexity of OPC UA.

Decision Drivers

- portability: consider consequences of feature parity between OPC UA and South API for involved parties

Considered Options

- mapping to South API and back to North API/OPC UA with already outlined functionality
- assume a South API independent direct connection from OPC UA data sources to north bound OPC UA to be defined in later iteration

Decision Outcome

As long as the OPC UA data client does not require more functionality than the south API provides, there is no reason to use another API than the South API.

If there are additional requirements that do not match the intention and structure of the South API or would lead to an increased complexity for other data clients, than fallback to option:
assume a South API independent direct connection from OPC UA data sources to north bound OPC UA to be defined in later iteration

Pros and Cons of the Options

Since no immediate action is required for the chosen option it is still possible to revert this decision without having spent effort and chose to integrate OPC UA data sources with South API compatible data clients.

mapping to South API and back to North API/OPC UA with already outlined functionality

- Good, because only a single data source integration via South API must be implemented by transformation engines
- Bad, because feature parity between South API and OPC UA or between a OPC UA feature subset must be considered and potential limitations must be defined and might limit usability of existing data sources
- Bad, because even if direct OPC UA knowledge is not required by data client implementors it might still "leak through" via South API specifications

assume a South API independent direct connection from OPC UA data sources to north bound OPC UA to be defined in later iteration

- Good, South API must not enable full feature set of OPC UA
- Good, because transformation engine implementors should already be familiar with OPC UA features
- Bad, because transformation engine implementations must handle an additional data source type

Copyright © 2021