Jülich Supercomputing Centre

The Open Trace Format 2 (Version 1.0 beta 1) Format and Library Specification

August 2011

Dominic Eschweiler, Michael Wagner[1]

(1) ZIH - Technische Universität Dresden

Interner Bericht · FZJ-JSC-IB-2011-03

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

FORSCHUNGSZENTRUM JÜLICH GmbH
Jülich Supercomputing Centre
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

The Open Trace Format 2 (Version 1.0 beta 1) Format and Library Specification

Dominic Eschweiler, Michael Wagner[1]

FZJ-JSC-IB-2011-03

August 2011

(letzte Änderung: 01.12.2011)

(1) ZIH - Technische Universität Dresden

Contents

Chapter 1

Introduction

Applications, which are supposed to effectively utilize the enormous computational resources of today's HPC systems, must meet very high requirements. Developing such applications demands knowledge of the complex systems and underlying hardware, of parallel programming paradigms, and the behavior of the own source code. These tasks become more and more complex and can hardly be performed without support of appropriate tools.

Two common approaches, to analyze the performance of applications, are *profiling* and *event tracing*. Profiling is the gathering of summarized information about different performance metrics during runtime. While offering a good starting point to understand performance problems, it does not provide further insight into the application's dynamic behavior. In contrast event tracing tools record all events that are of interest for later examination, together with the time they occured and a number of event type specific properties during application runtime. Typical events are entering and leaving of functions or sending and receiving of messages.

Thus, trace-based analysis tools have access to the detailed dynamic application behavior and are therefore able to offer a much higher level of insight into occurring performance problems than profiling tools. However, the amount of collected information can be tremendous and usually results in several hundreds or thousands of megabytes of data per process. Therefore tools need highly memory efficient event trace formats and highly scalable access libraries to manage all the data.

This report describes the Open Trace Format 2 (OTF2 [1]), which consists of a detailed format specification and a according writer and reader library. OTF2 is the direct successor of EPILOG [2], which was developed at Forschungszentrum Jülich GmbH, and OTF [3], developed at Techinsche Universität Dresden. OTF2 is designed to address the demands of modern, tracing-based tools for high performance computing (HPC). The library is a colaborative development of Forschungszentrum Jülich and Technische Universität Dresden. It will serve as a common data source of the Scalasca scalable performance analyzer (Jülich [4]) and Vampir scalable trace browser (Dresden [5]). The OTF2 library is open source and therfore also suitable for other tools. The effort, to design and program a complete new tracing library, was made to enable users of our tools to analyze the same trace files with both Vampir and Scalasca. This has several advantages, but the two most severe ones are, that users do not need to measure two traces and that the efforts to maintain the related measurement software can be distributed to both partners.

Chapter 2

The Data Organization of OTF2 Files

This chapter specifies the format of the Open Trace Format 2. The actual format is always determined by the related OTF2 library, which means that the OTF2 library always stores valid formated files. Furthermore, a valid OTF2 trace must also fullfill semantic requirements, which cannot be validated by a trace library. It is therfore necessary to also fullfill the description of the records in Section 2.6 and 2.5.

2.1 Trace Archive Organization

An OTF2 trace is stored in an OTF2 archive. An archive (in terms of OTF2) are all files belonging to a trace, stored into the directory layout of OTF2. The file and directory layout can be seen in Fig. 2.1. The file and directory names are generated from a specific archive name, to allow to store two different OTF2 archives at the same folder. Some files are stored for each location. A location in the OTF2 model is an entity of execution, like a thread or process.

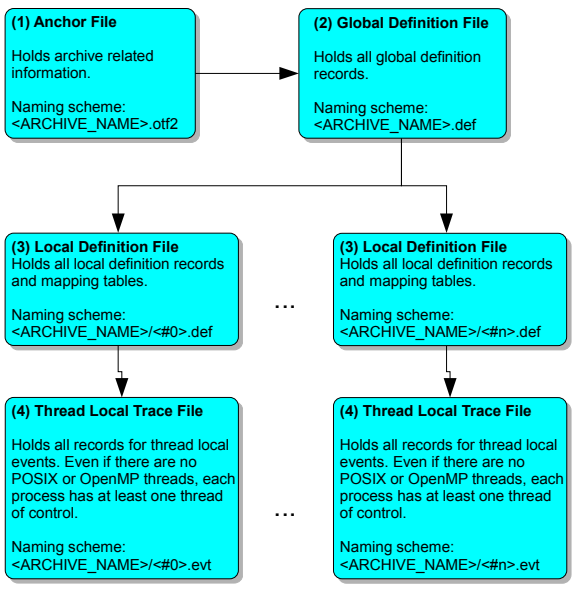

Figure 2.1: File and directory layout of an OTF2 archive.

An OTF2 archive consists of four different file types (numbers are according to Fig. 2.1):

1. **Anchor file** This is the anchor point for the user. The path to this file must be passed to the library, to open an OTF2 archive. The anchor file only contains meta data which is related to the archive organization (for example where are the other files stored).

2. **Global definition file** This file type stores all global and unified definition records.

3. **Local defintion file** This file stores mapping tables, which are used to map identifiers that are not global during measurement.

4. **Thread local trace files** This files store all local event records.

The archive name is a free form string and the location ID is a 64 unsigned integer. The different files of an OTF2 archive are named as follows:

Anchor file

```
<ARCHIVE_NAME>.otf
```

Global definition file

```
<ARCHIVE_NAME>.gdef
```

Local definition file

```
<ARCHIVE_NAME>/<LOCATION_ID>.def
```

Local trace file

```
<ARCHIVE_NAME>/<LOCATION_ID>.evt
```

Local event record files and local defintion files can only be stored for every location. To get all information in a correct way from such an OTF2 archive, the files must be read in the correct order:

- **Anchor file** To get mainly the information about which endianess is used. It is impossible to read the other binary data whithout reading this first. Furthermore the programmer can get the number global definition record from here, to implement progress bars and so on.

- **Global definition file** Traces are stored for each location, in a separate file for each location. This file must be read next to know which location files are in the trace and to be able to generate the related file paths. Furthermore each location definition stores the number of local definitions and events for each location. This is usefull for progress bars etc.

- **Local defintion file** Mainly to be able to translate all local IDs to global IDs, this file must be loaded at next.

- **Thread local trace files** This file should be read at last, to make it possible to get the definitions before the events.

An OTF2 trace must be stored in the following order:

- **Local event trace file** Because intermediate trace flushes can happen, this type of files are generated first on writing.

- **Local defintion file** The complete set of definitions is accessable not until the complete trace was recorded. Furthermore is the number of event records not known until the complete trace is measured.

- **Global definition file** Now, where all locations are complete they could be defined in the global definition file.
- **Anchor file** This file mainly stores meta data which is therfore only available on finalization.

2.2 The Anchor File Format

The following part is the specification for the data format of the anchor file. It is only intended for specifying the data format of the anchor file, not how it should be processed and nor which API the related software module has.

2.2.1 Anchor File Format Syntax

The anchor file is stored in ASCI text. Each key-value pair must be stored in a separate line. It is not allowed to include whitespaces except inside a string. The anchor file mainly stores information as so called key-value pairs in the following form (♯ is a number). All keys are always written in uppercase.

A key with an integer value:

```
KEY=#
```

A key with an floating point value:

```
KEY=#.#
```

A key with a string value (maybe file pathes for example):

```
KEY="<String>"
```

(Quoting of quotes is possible within strings by using \“ and \\)

Version information:

```
KEY=#-#-#
```

Optional data (data which is not needed to load an archive):

```
KEY~VALUE
```

An indexed key-value:

```
KEY[#]=VALUE
KEY[#]~VALUE
```

2.2.2 Storable items

Every OTF2 anchor file must begin with the following line:

```
This is an OTF2 anchor file.
```

File format version:

```
TRACE_FORMAT_VERSION=#-#-#                                                    1
```

Chunk size (kByte) of the binary data (for explaination please look in Section 2.3):

```
CHUNKSIZE=#                                                                   1
```

The used file substrate (for explaination please look in Section 3.3.4):

```
FILE_SUBSTRATE="<POSIX|SION|NON>"                                             1
```

Number of Locations:

```
LOCATION_NUMBER=#                                                             1
```

Number of global definition records:

```
GLOBAL_DEF_RECORDS_NUMBER=#                                                   1
```

Name of the machine where the trace was recorded:

```
MACHINE_NAME~"<String>"                                                       1
```

Creator of this archive:

```
CREATOR~"<String>"                                                            1
```

Trace description:

```
DESCRIPTION~"<String>"                                                        1
```

2.3 The Binary Format of Trace and Definition Files

The global definition file, local definition file, and the trace files are formated in the same binary format.

OTF2 stores the data in the endianess of the machine where the trace was recorded. For a better imagination please consider the following matrix if the endianess is flexible (the number means how often a conversion is needed) :

	Record Machine B. Endian	Record Machine L. Endian
Analysis Machine B. Endian	0	1
Analysis Machine L. Endian	1	0

... and now if the data is fix converted to little endian ...

	Record Machine B. Endian	Record Machine L. Endian
Analysis Machine B. Endian	2	1
Analysis Machine L. Endian	1	0

It is easy to see that with fix endianess version needs two times more conversions than with the flexible version.

2.3.1 Online Compression

For a simple preprocessing compression, all leading or trailing (depending on the endianess) zero bytes are removed from a variable before it is stored into the trace buffer. There are always the higher value bytes removed. The first byte always encodes the number of remaining (i.e., non-zero) bytes. This coding is not done for values which must be altered inside the trace buffer. Currently only timestamps are handled in this way.

2.3.2 Timestamps

The timestamp for all consecutive events with the same timestamp is only stored in the first event record of this chain. The timestamp is stored in a dedicated event record with the type `TIMESTAMP_RECORD`. This is discribed a little bit deeper in Section 3.2.3.

2.3.3 Chunking

A binary OTF2 file stream is parted into chunks of fixed size. Chunking makes it easier to step backwards through a trace stream, like it is described in Section 3.2.6. An event position refers to the position of a record which triggers an event. For example timestamp and optional attribute records are not referenced, because they are merged with other records. Each chunk begins with a special chunk header record. This header stores the following data:

```
uint8_t  record_type   // OTF2_INTERNAL_CHUNK_HEADER                          1
uint8_t  endianness    // OTF2_HOST_ENDIANNESS                                2
uint64_t first_event   // event position of the first event in this chunk     3
uint64_t last_event    // event position of the last event in this chunk      4
```

`first_event` and `last_event` determine which event position the first and the last record of this chunk have in the current stream. `endianess` determines the endianess in which the stream was recordet.

To get always the correct time, each chunk header is followed by a timestamp record. A chunk is filled with so called no-ops records, if it could not be filled up to the end. A no-op is a record with the type 0 (please look at Section 2.4 for a definition of record types), where no attributes are attached. Every chunk must have at least one no-op record at the end. Records can not span over different chunks.

2.4 Record Layout

The different record types in OTF2, for both definitions and events, are specified as C data structures. This has the advantage, that a programmer can easily build containers to store such records into the random access memory. The record layout, which is in the end written into a file, is therefore the record type ID followed by the serialized sequence of struct members (attributes). The record type ID has always a size of 8 bit (one byte). Additionally, this sequence is compressed with the algorithm described in Section 2.3.1. Definition and event records are basically stored in the same way, but differ slightly on which attributes are mandatory and how mappings are applied.

2.4.1 Global Definition Record Layout

An example for a global definition record can be given by the following struct:

Listing 2.1: Example for a global definition record

```
// #pragma OTF2_GEN                                                    1
typedef struct OTF2_GlobDefLocation_struct                             2
{                                                                      3
    uint64_t              location_identifier;                         4
    uint32_t              IDGS_name;                                   5
    OTF2_GlobLocationType location_type;                               6
    uint64_t              number_of_events;                            7
    uint64_t              number_of_definitions;                       8
}OTF2_GlobDefLocation;                                                 9
// #pragma OTF2_GEN                                                    10
```

Each of those structs must be surrounded by `//#pragma OTF2_GEN`, to make it possible for the generator script (see Section 4.1) to find them automatically. Data structure and type definition must always be combined like it is done in Listing 2.1. The naming scheme for the structure definition is `OTF2_GlobalDef<NameInCamelCase>_struct` and for the type definition `OTF2_GlobalDef<NameInCamelCase>`. The field `<NameInCamelCase>` must be equal in both cases. Arrays of basic data types from `stdint.h` can be used in such records, whereas complex data structures or other data types than the types in `stdint.h` are not allowed for attributes.

2.4.2 Event Record Layout

An example for an event record can be given by the following struct:

Listing 2.2: Example for an event record

```
// #pragma OTF2_GEN                                                    1
typedef struct OTF2_MpiIsend_struct                                    2
{                                                                      3
    OTF2_TimeStamp time;                                               4
    uint32_t       receiver;                                           5
    uint32_t       ID5_communicator;                                   6
    uint32_t       msgtag;                                             7
    uint64_t       msglength;                                          8
    uint64_t       requestID;                                          9
} OTF2_MpiIsend;                                                       10
// #pragma OTF2_GEN                                                    11
```

Each of those structs must be surrounded by `//#pragma OTF2_GEN`, to make it possible for the generator script (see Section 4.1) to find them automatically. Data structure and type definition must always be combined like it is done in Listing 2.2. The naming scheme for the structure definition is `OTF2_<NameInCamelCase>_struct` and for the type definition `OTF2_<NameInCamelCase>`. The field `<NameInCamelCase>` must be equal in both cases. The first attribute of event record must always be a `OTF2_TimeStamp` called `time`. The timestamp is internally written into a special record and into every event. The current timestamp is always valid until a new timestamp is written. Arrays or other data types than the types in `stdint.h` are not allowed for attributes. Identifiers which need to be mapped on reading need to be prefixed like it is done in Listing 2.2 in Line 6. The prefix always starts with an `ID` followed by the numeric identifier for the mapping type and is ended by an underscore _. The numeric mapping table identifier specifies with which mapping table the related attribute needs to be mapped.

2.5 Description of the Global Definition Records in OTF2

Global definitions defining static entities which do not directly describe an application behavior, but a part of its status and structure. A definition is intended to be referenced in an event record. For example a code region can be defined to make clear in which part of the application is currently processed.

The first field of each record in the binary layout is always a record type ID. Each record type has a unique type ID. These IDs and their values are specified as follows:

```
/* OTF2 Internals */                                1
OTF2_GLOB_DEF_END_OF_BUFFER             = 0,        2
OTF2_GLOB_DEF_END_OF_FILE               = 1,        3
                                                    4
/* global-only records */                           5
OTF2_GLOB_DEF_TIME_RANGE                = 5,        6
                                                    7
/* OTF2 Records */                                  8
OTF2_GLOB_DEF_STRING                    = 10,       9
OTF2_GLOB_DEF_ATTRIBUTE                 = 11,       10
                                                    11
OTF2_GLOB_DEF_SYSTEM_TREE_NODE          = 12,       12
OTF2_GLOB_DEF_LOCATION_GROUP            = 13,       13
OTF2_GLOB_DEF_LOCATION                  = 14,       14
                                                    15
OTF2_GLOB_DEF_REGION                    = 15,       16
OTF2_GLOB_DEF_CALLSITE                  = 16,       17
OTF2_GLOB_DEF_CALLPATH                  = 17,       18
OTF2_GLOB_DEF_GROUP                     = 18,       19
                                                    20
OTF2_GLOB_DEF_METRIC_MEMBER             = 19,       21
OTF2_GLOB_DEF_METRIC_CLASS              = 20,       22
OTF2_GLOB_DEF_METRIC_INSTANCE           = 21,       23
                                                    24
OTF2_GLOB_DEF_MPI_COMM                  = 22,       25
OTF2_GLOB_DEF_MPI_WIN                   = 23,       26
                                                    27
OTF2_GLOB_DEF_TOPOLOGY_CARTESIAN        = 24,       28
OTF2_GLOB_DEF_TOPOLOGY_CARTESIAN_COORDS = 25,       29
OTF2_GLOB_DEF_TOPOLOGY_GRAPH            = 26,       30
OTF2_GLOB_DEF_TOPOLOGY_GRAPH_EDGE       = 27,       31
                                                    32
OTF2_GLOB_DEF_PARAMETER                 = 28        33
```

The following subsections describe the full set of global definition record types in OTF2.

2.5.1 OTF2_GlobDefAttribute_struct Struct Reference

Attribute definition.

Data Fields

- uint32_t **attribute_id**
- uint32_t **IDGS_name**
- OTF2_TypeID **type**

Field Documentation

uint32_t OTF2_GlobDefAttribute_struct::attribute_id Attribute id.

uint32_t OTF2_GlobDefAttribute_struct::IDGS_name Name of the attribute.

OTF2_TypeID OTF2_GlobDefAttribute_struct::type Type of the attribute value.

2.5.2 OTF2_GlobDefCallpath_struct Struct Reference

Data Fields

- uint32_t **callpath_identifier**
- uint32_t **parent_callpath**
- uint32_t **region_identifier**
- uint8_t **call_path_order**

Field Documentation

uint32_t OTF2_GlobDefCallpath_struct::callpath_identifier

uint32_t OTF2_GlobDefCallpath_struct::parent_callpath

uint32_t OTF2_GlobDefCallpath_struct::region_identifier

uint8_t OTF2_GlobDefCallpath_struct::call_path_order

2.5.3 OTF2_GlobDefCallsite_struct Struct Reference

Data Fields

- uint32_t **callsite_identifier**
- uint32_t **IDGS_source_file**
- uint32_t **line_number**
- uint32_t **region_entered**
- uint32_t **region_left**

Field Documentation

uint32_t OTF2_GlobDefCallsite_struct::callsite_identifier

uint32_t OTF2_GlobDefCallsite_struct::IDGS_source_file

uint32_t OTF2_GlobDefCallsite_struct::line_number

uint32_t OTF2_GlobDefCallsite_struct::region_entered

uint32_t OTF2_GlobDefCallsite_struct::region_left

2.5.4 OTF2_GlobDefGroup_struct Struct Reference

Data Fields

- uint64_t **group_id**
- OTF2_GlobGroupType **type**
- uint32_t **IDGS_name**
- uint64_t **number_of_members**
- uint64_t ∗ **members**

Field Documentation

uint64_t OTF2_GlobDefGroup_struct::group_id

OTF2_GlobGroupType OTF2_GlobDefGroup_struct::type

uint32_t OTF2_GlobDefGroup_struct::IDGS_name

uint64_t OTF2_GlobDefGroup_struct::number_of_members

uint64_t∗ OTF2_GlobDefGroup_struct::members

2.5.5 OTF2_GlobDefLocationGroup_struct Struct Reference

Data Fields

- uint64_t **group_id**
- uint32_t **IDGS_name**
- OTF2_GlobLocationGroupType **type**
- uint32_t **system_tree_parent**

Field Documentation

uint64_t OTF2_GlobDefLocationGroup_struct::group_id The global unique identifier for this group

uint32_t OTF2_GlobDefLocationGroup_struct::IDGS_name Name of the group.

OTF2_GlobLocationGroupType OTF2_GlobDefLocationGroup_struct::type Type of this group.

uint32_t OTF2_GlobDefLocationGroup_struct::system_tree_parent Parent of this location group in the system tree.

2.5.6 OTF2_GlobDefLocation_struct Struct Reference

Data Fields

- uint64_t **location_identifier**
- uint32_t **IDGS_name**

- OTF2_GlobLocationType **location_type**
- uint64_t **number_of_events**
- uint64_t **number_of_definitions**
- uint64_t **timer_resolution**
- uint64_t **location_group**

Field Documentation

uint64_t OTF2_GlobDefLocation_struct::location_identifier

uint32_t OTF2_GlobDefLocation_struct::IDGS_name

OTF2_GlobLocationType OTF2_GlobDefLocation_struct::location_type

uint64_t OTF2_GlobDefLocation_struct::number_of_events

uint64_t OTF2_GlobDefLocation_struct::number_of_definitions

uint64_t OTF2_GlobDefLocation_struct::timer_resolution

uint64_t OTF2_GlobDefLocation_struct::location_group

2.5.7 OTF2_GlobDefMetricClass_struct Struct Reference

Metric class definition.

Data Fields

- uint64_t **metric_class_id**
- uint8_t **number_of_metrics**
- uint64_t ∗ **metric_members**
- OTF2_GlobMetricOccurrence **occurrence**

Detailed Description

For a metric class it is implicitly given that the event stream that records the metric is also the scope. A metric class can contain multiple different metrics.

Field Documentation

uint64_t OTF2_GlobDefMetricClass_struct::metric_class_id Metric class ID.

uint8_t OTF2_GlobDefMetricClass_struct::number_of_metrics Number of metrics with in the set.

uint64_t∗ OTF2_GlobDefMetricClass_struct::metric_members List of metric member IDs.

OTF2_GlobMetricOccurrence OTF2_GlobDefMetricClass_struct::occurrence Defines occurrence of a metric set.

2.5.8 OTF2_GlobDefMetricInstance_struct Struct Reference

Metric instance definition.

Data Fields

- uint64_t **metric_instance_id**
- uint64_t **metric_class**
- uint64_t **recorder**
- OTF2_GlobMetricScope **scope_type**
- uint64_t **scope**

Detailed Description

A metric instance is used to define metrics that are recorded at one location for multiple locations or for another location. The occurrence of a metric instance is implicitly of type OTF2_METRIC_-ASYNCHRONOUS.

Field Documentation

uint64_t OTF2_GlobDefMetricInstance_struct::metric_instance_id Metric instance id.

uint64_t OTF2_GlobDefMetricInstance_struct::metric_class Reference to metric class.

uint64_t OTF2_GlobDefMetricInstance_struct::recorder Recorder of the metric: location ID.

OTF2_GlobMetricScope OTF2_GlobDefMetricInstance_struct::scope_type Defines type of scope: location, location group, system tree node, or a generic group of locations.

uint64_t OTF2_GlobDefMetricInstance_struct::scope Scope of metric: ID of a location, location group, system tree node, or a generic group of locations.

2.5.9 OTF2_GlobDefMetricMember_struct Struct Reference

Metric member definition.

Data Fields

- uint64_t **metric_member_id**
- uint32_t **IDGS_name**
- uint32_t **IDGS_description**
- OTF2_GlobMetricType **type**
- OTF2_GlobMetricMode **mode**
- OTF2_TypeID **value_type**
- OTF2_GlobMetricBase **base**
- int64_t **exponent**
- uint32_t **IDGS_unit**

Detailed Description

A metric is defined by a metric member definition. A metric member is always a member of a metric class. Therefore, a single metric is a special case of a metric class with only one member. It is not allowed to reference a metric member id in a metric event, but only metric class IDs.

Field Documentation

uint64_t OTF2_GlobDefMetricMember_struct::metric_member_id Metric member ID.

uint32_t OTF2_GlobDefMetricMember_struct::IDGS_name Name of the metric.

uint32_t OTF2_GlobDefMetricMember_struct::IDGS_description Description of the metric.

OTF2_GlobMetricType OTF2_GlobDefMetricMember_struct::type Metric type: PAPI, etc.

OTF2_GlobMetricMode OTF2_GlobDefMetricMember_struct::mode Metric mode: accumulative, fix, relative, etc.

OTF2_TypeID OTF2_GlobDefMetricMember_struct::value_type Type of the value: int64_t, uin64_t, or double.

OTF2_GlobMetricBase OTF2_GlobDefMetricMember_struct::base Base of the value: binary or decimal.

int64_t OTF2_GlobDefMetricMember_struct::exponent Exponent of the value.

uint32_t OTF2_GlobDefMetricMember_struct::IDGS_unit Unit of the metric.

2.5.10 OTF2_GlobDefMpiComm_struct Struct Reference

Data Fields

- uint32_t **communicator_identifier**
- uint64_t **group_id**

Field Documentation

uint32_t OTF2_GlobDefMpiComm_struct::communicator_identifier

uint64_t OTF2_GlobDefMpiComm_struct::group_id

2.5.11 OTF2_GlobDefMpiWin_struct Struct Reference

Data Fields

- uint32_t **window_identifier**
- uint32_t **communicator_identifier**

Field Documentation

uint32_t OTF2_GlobDefMpiWin_struct::window_identifier

uint32_t OTF2_GlobDefMpiWin_struct::communicator_identifier

2.5.12 OTF2_GlobDefParameter_struct Struct Reference

Parameter definition.

Data Fields

- uint32_t **parameter_id**
- uint32_t **IDGS_parameter_name**
- OTF2_GlobParameterType **parameter_type**

Field Documentation

uint32_t OTF2_GlobDefParameter_struct::parameter_id Parameter ID

uint32_t OTF2_GlobDefParameter_struct::IDGS_parameter_name Name of the parameter (variable name etc.)

OTF2_GlobParameterType OTF2_GlobDefParameter_struct::parameter_type Type of the parameter, see OTF2_GlobParameterType for possible types.

2.5.13 OTF2_GlobDefRegion_struct Struct Reference

Data Fields

- uint32_t **region_identifier**
- uint32_t **IDGS_region_name**
- uint32_t **IDGS_region_description**
- OTF2_GlobRegionType **region_type**
- uint32_t **IDGS_source_file**
- uint32_t **begin_line_number**
- uint32_t **end_line_number**

Field Documentation

uint32_t OTF2_GlobDefRegion_struct::region_identifier

uint32_t OTF2_GlobDefRegion_struct::IDGS_region_name

uint32_t OTF2_GlobDefRegion_struct::IDGS_region_description

OTF2_GlobRegionType OTF2_GlobDefRegion_struct::region_type

uint32_t OTF2_GlobDefRegion_struct::IDGS_source_file

uint32_t OTF2_GlobDefRegion_struct::begin_line_number

uint32_t OTF2_GlobDefRegion_struct::end_line_number

2.5.14 OTF2_GlobDefString_struct Struct Reference

Data Fields

- uint32_t **string_identifier**
- char ∗ **string**

Field Documentation

uint32_t OTF2_GlobDefString_struct::string_identifier

char∗ OTF2_GlobDefString_struct::string

2.5.15 OTF2_GlobDefSystemTreeNode_struct Struct Reference

Data Fields

- uint32_t **node_id**
- uint32_t **IDGS_name**
- uint32_t **IDGS_class_name**
- uint32_t **node_parent**

Field Documentation

uint32_t OTF2_GlobDefSystemTreeNode_struct::node_id The unique identifier for this node.

uint32_t OTF2_GlobDefSystemTreeNode_struct::IDGS_name Free form instance name of this node.

uint32_t OTF2_GlobDefSystemTreeNode_struct::IDGS_class_name Free form class name of this node.

uint32_t OTF2_GlobDefSystemTreeNode_struct::node_parent Parent id of this node. May be OTF2_UNDEFINED_UINT32 to indicate that there is no parent.

2.5.16 OTF2_GlobDefTimeRange_struct Struct Reference

Data Fields

- uint64_t **global_offset**
- uint64_t **trace_length**

Field Documentation

uint64_t OTF2_GlobDefTimeRange_struct::global_offset The global offset to the epoch. This may be used to have relatively small timestamp

uint64_t OTF2_GlobDefTimeRange_struct::trace_length The length of the event horizont. I.e. this is greater than the global time of the very last event - the global time of the very first event.

2.5.17 OTF2_GlobDefTopologyCartesianCoords_struct Struct Reference

Data Fields

- uint32_t **cartesian_topology_identifier**
- uint64_t **location_identifier**
- uint32_t **number_of_dimensions**
- uint32_t * **coordinates_of_the_location**

Field Documentation

uint32_t OTF2_GlobDefTopologyCartesianCoords_struct::cartesian_topology_identifier

uint64_t OTF2_GlobDefTopologyCartesianCoords_struct::location_identifier

uint32_t OTF2_GlobDefTopologyCartesianCoords_struct::number_of_dimensions

uint32_t* OTF2_GlobDefTopologyCartesianCoords_struct::coordinates_of_the_location

2.5.18 OTF2_GlobDefTopologyCartesian_struct Struct Reference

Data Fields

- uint32_t **cartesian_topology_identifier**
- uint32_t **IDGS_name**
- uint32_t **number_of_locations_in_each_dimension**
- uint32_t * **locations_in_each_dimension**
- uint32_t **number_of_periodicity_of_the_grid_in_each_dimension**
- uint8_t * **periodicity_of_the_grid_in_each_dimension**

Field Documentation

uint32_t OTF2_GlobDefTopologyCartesian_struct::cartesian_topology_identifier

uint32_t OTF2_GlobDefTopologyCartesian_struct::IDGS_name

uint32_t OTF2_GlobDefTopologyCartesian_struct::number_of_locations_in_each_dimension

uint32_t* OTF2_GlobDefTopologyCartesian_struct::locations_in_each_dimension

uint32_t OTF2_GlobDefTopologyCartesian_struct::number_of_periodicity_of_the_grid_in_-each_dimension

uint8_t* OTF2_GlobDefTopologyCartesian_struct::periodicity_of_the_grid_in_each_dimension

2.5.19 OTF2_GlobDefTopologyGraphEdge_struct Struct Reference

Data Fields

- uint32_t **topology_graph_identifier**
- uint64_t **from**
- uint64_t **to**

Field Documentation

uint32_t OTF2_GlobDefTopologyGraphEdge_struct::topology_graph_identifier

uint64_t OTF2_GlobDefTopologyGraphEdge_struct::from

uint64_t OTF2_GlobDefTopologyGraphEdge_struct::to

2.5.20 OTF2_GlobDefTopologyGraph_struct Struct Reference

Data Fields

- uint32_t **topology_graph_identifier**
- uint32_t **IDGS_name**
- uint8_t **is_directed**

Field Documentation

uint32_t OTF2_GlobDefTopologyGraph_struct::topology_graph_identifier

uint32_t OTF2_GlobDefTopologyGraph_struct::IDGS_name

uint8_t OTF2_GlobDefTopologyGraph_struct::is_directed

2.6 Description of the Event Records in OTF2

Event records describing the behavior of a measured application at runtime. They can be read or written by the local event reader or writer component (see Sections 3.2.6 and 3.2.3) of the OTF2 library. It is also possible to read a merged stream from several different location with the global event reader component (see Section 3.2.7).

The first field of each record in the binary layout is always a record type ID. Each record type has a unique type ID. These IDs and their values are specified as follows:

```
/* OTF2 Internals */
OTF2_END_OF_CHUNK                   = 0,
OTF2_END_OF_BUFFER                  = 1,
OTF2_END_OF_FILE                    = 2,
OTF2_INTERNAL_CHUNK_HEADER          = 3,
OTF2_EVENT_MEASUREMENT_ON_OFF       = 4,

/* Events */
OTF2_EVENT_BUFFER_FLUSH             = 10,
OTF2_EVENT_TIMESTAMP                = 11,
OTF2_ATTRIBUTE_LIST                 = 12,

OTF2_EVENT_ENTER                    = 13,
OTF2_EVENT_LEAVE                    = 14,

OTF2_EVENT_MPI_SEND                 = 15,
OTF2_EVENT_MPI_ISEND                = 16,
OTF2_EVENT_MPI_ISEND_COMPLETE       = 17,
OTF2_EVENT_MPI_IRECV_REQUEST        = 18,
OTF2_EVENT_MPI_RECV                 = 19,
OTF2_EVENT_MPI_IRECV                = 20,
OTF2_EVENT_MPI_REQUEST_TEST         = 21,
OTF2_EVENT_MPI_REQUEST_CANCELLED    = 22,
OTF2_EVENT_MPI_COLLECTIVE_BEGIN     = 23,
OTF2_EVENT_MPI_COLLECTIVE_END       = 24,
OTF2_EVENT_MPI_RMA_SYNC             = 25,
OTF2_EVENT_MPI_RMA_PUT              = 26,
OTF2_EVENT_MPI_RMA_GET              = 27,
OTF2_EVENT_MPI_RMA                  = 28,

OTF2_EVENT_OMP_FORK                 = 29,
OTF2_EVENT_OMP_JOIN                 = 30,
OTF2_EVENT_OMP_ALOCK                = 31,
OTF2_EVENT_OMP_RLOCK                = 32,
OTF2_EVENT_OMP_COLL_EXIT            = 33,

OTF2_EVENT_FILE_OPERATION_BEGIN     = 34,
OTF2_EVENT_FILE_OPERATION_END       = 35,

OTF2_EVENT_METRIC                   = 36,

OTF2_EVENT_OMP_TASK_CREATE_BEGIN    = 37,
OTF2_EVENT_OMP_TASK_CREATE_END      = 38,
OTF2_EVENT_OMP_TASK_BEGIN_OR_RESUME = 39,
OTF2_EVENT_OMP_TASK_COMPLETED       = 40,

OTF2_EVENT_PARAMETER_STRING         = 41,
OTF2_EVENT_PARAMETER_INT            = 42,
OTF2_EVENT_PARAMETER_UNSIGNED_INT   = 43
```

Currently OTF2 supports record sets for MPI and OpenMP, as well as some basic records for generic

instrumentation (e.g. enter and leave records etc.). The following subsections describe the full set of event record types in OTF2.

2.6.1 OTF2_BufferFlush_struct Struct Reference

To signal where a buffer flushed happened, this event is returned by the reader (but generated by the buffer module).

Data Fields

- OTF2_TimeStamp **start_time**
- OTF2_TimeStamp **stop_time**

Field Documentation

OTF2_TimeStamp OTF2_BufferFlush_struct::start_time Start time of the flush event

OTF2_TimeStamp OTF2_BufferFlush_struct::stop_time Stop time of the flush

2.6.2 OTF2_Enter_struct Struct Reference

An enter record indicates that the program enters a code region.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID1_region**

Field Documentation

OTF2_TimeStamp OTF2_Enter_struct::time Timestamp

uint32_t OTF2_Enter_struct::ID1_region Needs to be defined in a definition record (needs mapping)

2.6.3 OTF2_FileOperationBegin_struct Struct Reference

Signals a begin of a file operation.

Data Fields

- OTF2_TimeStamp **time**
- uint64_t **handle_id**

Field Documentation

OTF2_TimeStamp OTF2_FileOperationBegin_struct::time Timestamp

uint64_t OTF2_FileOperationBegin_struct::handle_id Handle ID

2.6.4 OTF2_FileOperationEnd_struct Struct Reference

Signals an end of a file operation.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **file_id**
- uint64_t **handle_id**
- uint32_t **operation**
- uint64_t **size**

Field Documentation

OTF2_TimeStamp OTF2_FileOperationEnd_struct::time Timestamp

uint32_t OTF2_FileOperationEnd_struct::file_id File ID

uint64_t OTF2_FileOperationEnd_struct::handle_id Handle ID

uint32_t OTF2_FileOperationEnd_struct::operation Type of file operation

uint64_t OTF2_FileOperationEnd_struct::size Size of the handled memory

2.6.5 OTF2_Leave_struct Struct Reference

A leave record indicates that the program leaves a code region.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID1_region**

Field Documentation

OTF2_TimeStamp OTF2_Leave_struct::time Timestamp

uint32_t OTF2_Leave_struct::ID1_region Needs to be defined in a definition record (needs mapping)

2.6.6 OTF2_MeasurementOnOff_struct Struct Reference

Data Fields

- OTF2_TimeStamp **time**
- uint8_t **on_or_off**

Field Documentation

OTF2_TimeStamp OTF2_MeasurementOnOff_struct::time Timestamp, which is exactly here in every record

uint8_t OTF2_MeasurementOnOff_struct::on_or_off Is the measurement turned on or off?

2.6.7 OTF2_Metric_struct Struct Reference

Metric event.

Data Fields

- OTF2_TimeStamp **time**
- uint64_t **ID6_metric_id**
- uint8_t **number_of_metrics**
- OTF2_TypeID * **type_ids**
- **OTF2_MetricValue** * **values**

Detailed Description

A metric event is always stored at the location that recorded the metric. A metric event can be reference to a metric class or metric instance. Therefore, metric classes and instances share same ID space. Synchronous metrics are always located right before the according enter and leave. The metric event can contain multiple metrics.

Field Documentation

OTF2_TimeStamp OTF2_Metric_struct::time Timestamp.

uint64_t OTF2_Metric_struct::ID6_metric_id Metric IDs: Could be a metric class or a metric instance.

uint8_t OTF2_Metric_struct::number_of_metrics Number of metrics with in the set.

OTF2_TypeID* OTF2_Metric_struct::type_ids List of metric types.

OTF2_MetricValue* OTF2_Metric_struct::values List of metric values.

2.6.8 OTF2_MpiCollectiveBegin_struct Struct Reference

A mpi_collective_begin record marks the begin of an MPI collective operation (MPI_GATHER, MPI_SCATTER etc.). It keeps the necessary information for this event: time, type of collective operation, group, the root of this collective operation and a matching ID to match the according end record. You can optionally add further informations like sent bytes and received bytes. This event is always surrounded by appropriate enter and leave records.

Data Fields

- OTF2_TimeStamp **time**
- OTF2_Mpi_CollectiveType **type**
- uint32_t **ID5_communicator**
- uint32_t **root**
- uint64_t **matching_id**

Field Documentation

OTF2_TimeStamp OTF2_MpiCollectiveBegin_struct::time Timestamp

OTF2_Mpi_CollectiveType OTF2_MpiCollectiveBegin_struct::type Determines which collective operation it is

uint32_t OTF2_MpiCollectiveBegin_struct::ID5_communicator Communicator (needs mapping)

uint32_t OTF2_MpiCollectiveBegin_struct::root MPI rank of root in *ID5_communicator*.

uint64_t OTF2_MpiCollectiveBegin_struct::matching_id Matching ID to match with end record.

2.6.9 OTF2_MpiCollectiveEnd_struct Struct Reference

A mpi_collective_end record marks the end of an MPI collective operation (MPI_GATHER, MPI_-SCATTER etc.). It keeps the necessary information for this event: time and a matching ID to match the according begin record.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID5_communicator**
- uint64_t **matching_id**
- uint64_t **size_sent**
- uint64_t **size_received**

Field Documentation

OTF2_TimeStamp OTF2_MpiCollectiveEnd_struct::time Timestamp

uint32_t OTF2_MpiCollectiveEnd_struct::ID5_communicator Communicator (needs mapping)

uint64_t OTF2_MpiCollectiveEnd_struct::matching_id Matching ID to match with begin record.

uint64_t OTF2_MpiCollectiveEnd_struct::size_sent Size of the sended message

uint64_t OTF2_MpiCollectiveEnd_struct::size_received Size of the received message

2.6.10 OTF2_MpiIrecvRequest_struct Struct Reference

Signals the request of an receive, which can be completed later.

Data Fields

- OTF2_TimeStamp **time**
- uint64_t **requestID**

Field Documentation

OTF2_TimeStamp OTF2_MpiIrecvRequest_struct::time Timestamp

uint64_t OTF2_MpiIrecvRequest_struct::requestID ID of the requested receive

2.6.11 OTF2_MpiIrecv_struct Struct Reference

An mpi_irecv record indicates that a non-blocking MPI message was recieved (MPI_IRECV). It keeps the necessary information for this event: time, sender of the message, the communicator and the request ID. You can optionally add further information like message tag and message lenght (size of the recieve buffer).

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **sender**
- uint32_t **ID5_communicator**
- uint32_t **msgtag**
- uint64_t **msglength**
- uint64_t **requestID**

Field Documentation

OTF2_TimeStamp OTF2_MpiIrecv_struct::time Timestamp

uint32_t OTF2_MpiIrecv_struct::sender MPI rank of sender in *ID5_communicator*.

uint32_t OTF2_MpiIrecv_struct::ID5_communicator Communicator ID (Needs to be defined) (needs mapping)

uint32_t OTF2_MpiIrecv_struct::msgtag Message tag (can be NOID)

uint64_t OTF2_MpiIrecv_struct::msglength Message length (can be NOID)

uint64_t OTF2_MpiIrecv_struct::requestID ID of the related request

2.6.12 OTF2_MpiIsendComplete_struct Struct Reference

Signals the completion of non-blocking send request.

Data Fields

- OTF2_TimeStamp **time**
- uint64_t **requestID**

Field Documentation

OTF2_TimeStamp OTF2_MpiIsendComplete_struct::time Timestamp

uint64_t OTF2_MpiIsendComplete_struct::requestID ID of the related request

2.6.13 OTF2_MpiIsend_struct Struct Reference

An mpi_isend record indicates that an MPI message send process was initiated (MPI_ISend). It keeps the necessary information for this event: time, sender and receiver of the message, the communicator and the request ID. You can optionally add further information like message tag and message length (size of the send buffer).

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **receiver**
- uint32_t **ID5_communicator**
- uint32_t **msgtag**
- uint64_t **msglength**
- uint64_t **requestID**

Field Documentation

OTF2_TimeStamp OTF2_MpiIsend_struct::time Timestamp

uint32_t OTF2_MpiIsend_struct::receiver MPI rank of receiver in *ID5_communicator*.

uint32_t OTF2_MpiIsend_struct::ID5_communicator Communicator ID (Needs to be defined) (needs mapping)

uint32_t OTF2_MpiIsend_struct::msgtag Message tag (can be NOID)

uint64_t OTF2_MpiIsend_struct::msglength Message length (can be NOID)

uint64_t OTF2_MpiIsend_struct::requestID ID of the related request

2.6.14 OTF2_MpiRecv_struct Struct Reference

A mpi_recv record indicates that a MPI message was recieved (MPI_RECV). It keeps the necessary information for this event: time, sender and reciever of the message and the communicator. You can optionally add further informations like message tag and message lenght (size of the recieve buffer).

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **sender**
- uint32_t **ID5_communicator**
- uint32_t **msgtag**
- uint64_t **msglength**

Field Documentation

OTF2_TimeStamp OTF2_MpiRecv_struct::time Timestamp

uint32_t OTF2_MpiRecv_struct::sender MPI rank of sender in *ID5_communicator*.

uint32_t OTF2_MpiRecv_struct::ID5_communicator Communicator ID (Needs to be defined) (needs mapping)

uint32_t OTF2_MpiRecv_struct::msgtag Message tag (can be NOID)

uint64_t OTF2_MpiRecv_struct::msglength Message length (can be NOID)

2.6.15 OTF2_MpiRequestCancelled_struct Struct Reference

This events appears if the program cancels a request.

Data Fields

- OTF2_TimeStamp **time**
- uint64_t **requestID**

Field Documentation

OTF2_TimeStamp OTF2_MpiRequestCancelled_struct::time Timestamp

uint64_t OTF2_MpiRequestCancelled_struct::requestID ID of the related request

2.6.16 OTF2_MpiRequestTest_struct Struct Reference

This events appears if the program tests if a request has already completed.

Data Fields

- OTF2_TimeStamp **time**
- uint64_t **requestID**

Field Documentation

OTF2_TimeStamp OTF2_MpiRequestTest_struct::time Timestamp

uint64_t OTF2_MpiRequestTest_struct::requestID ID of the related request

2.6.17 OTF2_MpiRmaGet_struct Struct Reference

An RmaGet record indicates an MPI one-sided communication get operation (MPI_GET). It keeps the necessary information for this event: time, originator origin, target location, window object and the group of participating locations. You can optionally add further informations like the size of the received buffer.

Data Fields

- OTF2_TimeStamp **time**
- uint64_t **origin**
- uint64_t **target**
- uint32_t **ID3_window**

- uint64_t **size**

Field Documentation

OTF2_TimeStamp OTF2_MpiRmaGet_struct::time	Timestamp
uint64_t OTF2_MpiRmaGet_struct::origin	Defined as Location ID
uint64_t OTF2_MpiRmaGet_struct::target	Defined as Location ID
uint32_t OTF2_MpiRmaGet_struct::ID3_window	RMA Window ID (needs mapping)
uint64_t OTF2_MpiRmaGet_struct::size	Memory region size

2.6.18 OTF2_MpiRmaPut_struct Struct Reference

An RmaPut record indicates an MPI one-sided communication put operation (MPI_PUT). It keeps the necessary information for this event: time, originator origin, target location, window object and the group of participating locations. You can optionally add further informations like the size of the sent buffer.

Data Fields

- OTF2_TimeStamp **time**
- uint64_t **origin**
- uint64_t **target**
- uint32_t **ID3_window**
- uint64_t **size**
- uint8_t **match_target**

Field Documentation

OTF2_TimeStamp OTF2_MpiRmaPut_struct::time	Timestamp
uint64_t OTF2_MpiRmaPut_struct::origin	Defined as Location ID
uint64_t OTF2_MpiRmaPut_struct::target	Defined as Location ID
uint32_t OTF2_MpiRmaPut_struct::ID3_window	RMA Window ID (needs mapping)
uint64_t OTF2_MpiRmaPut_struct::size	Memory region size
uint8_t OTF2_MpiRmaPut_struct::match_target	Match end on targe

2.6.19 OTF2_MpiRmaSync_struct Struct Reference

An RmaSync record indicates MPI one-sided synchronization issued in the current region. The timestamps of the surrounding flow events (enter/leave) reference the times, where the synchronization may have started at the earliest and ended at the latest.

Data Fields

- OTF2_TimeStamp **time**
- OTF2_Mpi_RmaSyncType **type**
- uint64_t **remote**
- uint32_t **ID3_window**
- uint64_t **gats_group**
- uint8_t **sync**
- OTF2_Mpi_RmaLockType **lock_type**

Field Documentation

OTF2_TimeStamp OTF2_MpiRmaSync_struct::time Timestamp

OTF2_Mpi_RmaSyncType OTF2_MpiRmaSync_struct::type Type of MPI RMA synchronization.

uint64_t OTF2_MpiRmaSync_struct::remote Defined as Location ID.

uint32_t OTF2_MpiRmaSync_struct::ID3_window Window ID.

uint64_t OTF2_MpiRmaSync_struct::gats_group Group id of processes involved in general active target synchronization (GATS) epoch (needs mapping).

uint8_t OTF2_MpiRmaSync_struct::sync [NOTHING IN HERE]

OTF2_Mpi_RmaLockType OTF2_MpiRmaSync_struct::lock_type Lock type (exclusive or shared)

2.6.20 OTF2_MpiSend_struct Struct Reference

A mpi_send record indicates that a MPI message send process was initiated (MPI_Send). It keeps the necessary information for this event: time, sender and receiver of the message and the communicator. You can optionally add further informations like message tag and message length (size of the send buffer).

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **receiver**
- uint32_t **ID5_communicator**
- uint32_t **msgtag**
- uint64_t **msglength**

Field Documentation

OTF2_TimeStamp OTF2_MpiSend_struct::time Timestamp

uint32_t OTF2_MpiSend_struct::receiver MPI rank of receiver in *ID5_communicator*.

uint32_t OTF2_MpiSend_struct::ID5_communicator Communicator ID (Needs to be defined) (needs mapping)

uint32_t OTF2_MpiSend_struct::msgtag Message tag (can be NOID)

uint64_t OTF2_MpiSend_struct::msglength Message length (can be NOID)

2.6.21 OTF2_OmpAcquireLock_struct Struct Reference

An OmpAcquireLock record marks that a thread acquires an OpenMP lock.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **lock_id**
- uint32_t **acquire_release_count**
- uint32_t **ID1_region**

Field Documentation

OTF2_TimeStamp OTF2_OmpAcquireLock_struct::time Timestamp

uint32_t OTF2_OmpAcquireLock_struct::lock_id ID of the lock

uint32_t OTF2_OmpAcquireLock_struct::acquire_release_count cout how often the lock was used

uint32_t OTF2_OmpAcquireLock_struct::ID1_region Source region ID

2.6.22 OTF2_OmpFork_struct Struct Reference

An OmpFork record marks that an OpenMP Thread forks a thread team.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **number_of_requested_threads**
- uint32_t **ID1_region**

Field Documentation

OTF2_TimeStamp OTF2_OmpFork_struct::time Timestamp

uint32_t OTF2_OmpFork_struct::number_of_requested_threads requested size of the team

uint32_t OTF2_OmpFork_struct::ID1_region Source region ID

2.6.23 OTF2_OmpJoin_struct Struct Reference

An OmpJoin record marks that a Team of threads is joint and only the master thread continues execution.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID1_region**

Field Documentation

OTF2_TimeStamp OTF2_OmpJoin_struct::time Timestamp

uint32_t OTF2_OmpJoin_struct::ID1_region Source region ID

2.6.24 OTF2_OmpReleaseLock_struct Struct Reference

An OmpReleaselock record marks that a thread releases an OpenMP lock.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **lock_id**
- uint32_t **acquire_release_count**
- uint32_t **ID1_region**

Field Documentation

OTF2_TimeStamp OTF2_OmpReleaseLock_struct::time Timestamp

uint32_t OTF2_OmpReleaseLock_struct::lock_id ID of the lock

uint32_t OTF2_OmpReleaseLock_struct::acquire_release_count count how often the lock was used

uint32_t OTF2_OmpReleaseLock_struct::ID1_region Source region ID

2.6.25 OTF2_OmpTaskBeginOrResume_struct Struct Reference

An OMPTaskBeginOrResume record indicates that the execution of a task is started or continued.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID1_region**
- uint64_t **task_id**

Field Documentation

OTF2_TimeStamp OTF2_OmpTaskBeginOrResume_struct::time Timestamp

uint32_t OTF2_OmpTaskBeginOrResume_struct::ID1_region Source region ID.

uint64_t OTF2_OmpTaskBeginOrResume_struct::task_id A Task specific ID.

2.6.26 OTF2_OmpTaskCompleted_struct Struct Reference

On OMPTaskTerminate record indicates, that an OpenMP Tasks execution has finished.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID1_region**

Field Documentation

OTF2_TimeStamp OTF2_OmpTaskCompleted_struct::time Timestamp

uint32_t OTF2_OmpTaskCompleted_struct::ID1_region Source region ID.

2.6.27 OTF2_OmpTaskCreateBegin_struct Struct Reference

An OMPTaskCreateBegin record marks that an OpenMP Tasks creation is started.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID1_region**
- uint64_t **task_id**

Field Documentation

OTF2_TimeStamp OTF2_OmpTaskCreateBegin_struct::time Timestamp

uint32_t OTF2_OmpTaskCreateBegin_struct::ID1_region Source region ID.

uint64_t OTF2_OmpTaskCreateBegin_struct::task_id An Task specific ID.

2.6.28 OTF2_OmpTaskCreateEnd_struct Struct Reference

An OMPTaskCreateEnd record marks that an OpenMP Tasks creation is finished.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID1_region**
- uint64_t **task_id**

Field Documentation

OTF2_TimeStamp OTF2_OmpTaskCreateEnd_struct::time Timestamp

uint32_t OTF2_OmpTaskCreateEnd_struct::ID1_region Source region ID.

uint64_t OTF2_OmpTaskCreateEnd_struct::task_id An Task specific ID.

2.6.29 OTF2_ParameterInt_struct Struct Reference

A parameter_int64 record marks that in the current region, the specified 64 bit integer parameter has the specified value.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID8_parameter**
- int64_t **value**

Field Documentation

OTF2_TimeStamp OTF2_ParameterInt_struct::time Timestamp

uint32_t OTF2_ParameterInt_struct::ID8_parameter Parameter ID

int64_t OTF2_ParameterInt_struct::value Value of the recorded parameter

2.6.30 OTF2_ParameterString_struct Struct Reference

Parameter event for string parameters.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID8_parameter**
- uint32_t **ID10_string**

Detailed Description

A parameter_string record marks that in the current region, the specified string parameter has the specified value.

Field Documentation

OTF2_TimeStamp OTF2_ParameterString_struct::time Timestamp

uint32_t OTF2_ParameterString_struct::ID8_parameter Parameter ID

uint32_t OTF2_ParameterString_struct::ID10_string Value: Handle of a string definition

2.6.31 OTF2_ParameterUnsignedInt_struct Struct Reference

A parameter_uint64 record marks that in the current region, the specified unigned 64 bit integer parameter has the specified value.

Data Fields

- OTF2_TimeStamp **time**
- uint32_t **ID8_parameter**
- uint64_t **value**

Field Documentation

OTF2_TimeStamp OTF2_ParameterUnsignedInt_struct::time Timestamp

uint32_t OTF2_ParameterUnsignedInt_struct::ID8_parameter Parameter ID

uint64_t OTF2_ParameterUnsignedInt_struct::value Value of the recorded parameter

Chapter 3

The OTF2 Tracing Library

This chapter describes the implementation of the OTF2 library. OTF2 is written in pure C but in an object oriented style – every module operates on an internal struct, where the members are not visible from the outside. Every function in OTF2 is prefixed with `OTF2_` followed by the name of the related module in camelcase.

3.1 Component Overview

This a structural overview of the OTF2 library. Like you can see in Figure 3.1, the OTF2 library is divided into four different layers. Each layer can only interact with the layer directly above or below to itself. Furthermore, every layer is related to a different view to the trace data:

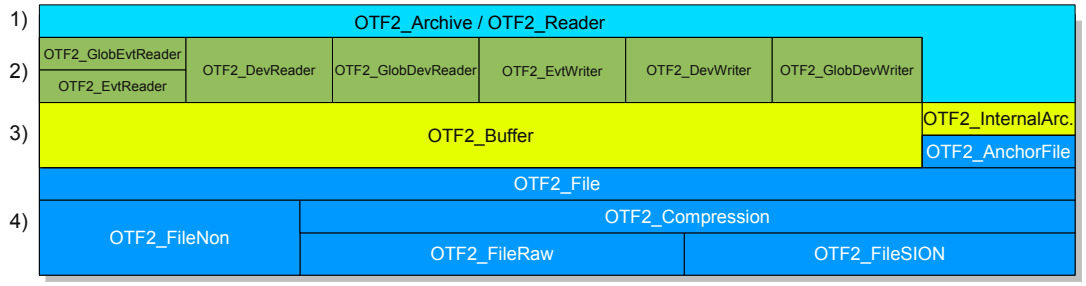

Figure 3.1: OTF2 is organized in layers.

1. Abstraction layer to support several different reader implementations for other formats than OTF2. This layer was initially intended to support backward compatibility to the two predecessor formats EPILOG and OTF.

2. Event layer, which represents the trace data already grouped into records. The programmer can handle all records on an abstract representation of the event stream, without deeper knowledge about internal encoding.

3. Memory layer, which implements the trace data representation in the random access memory. This layer can not be used from another program and is only intended to interact with the record- or the file layer.

4. File layer, which implements a POSIX-like abstraction layer for writing and reading files. The purpose of this layer is to support different I/O libraries (e.g. SIONlib or NetCDF).

The following class-diagram (see Figure 3.2) is given, to show a more detailed view to the interaction between all components of OTF2. Each rectangle symbols a sub-component, where the public API sub-components have only an external interface and the internal subcomponents have an only internal API which is not accessible for OTF2 users. All components are programmed in an object oriented fashion, which means that the are based on a data-structure with related functions. An arrow symbols a use relation, which means that a subcomponent uses an interface. No arrow is drawn if a sub-component only handles with a data structure, without accessing its interfaces.

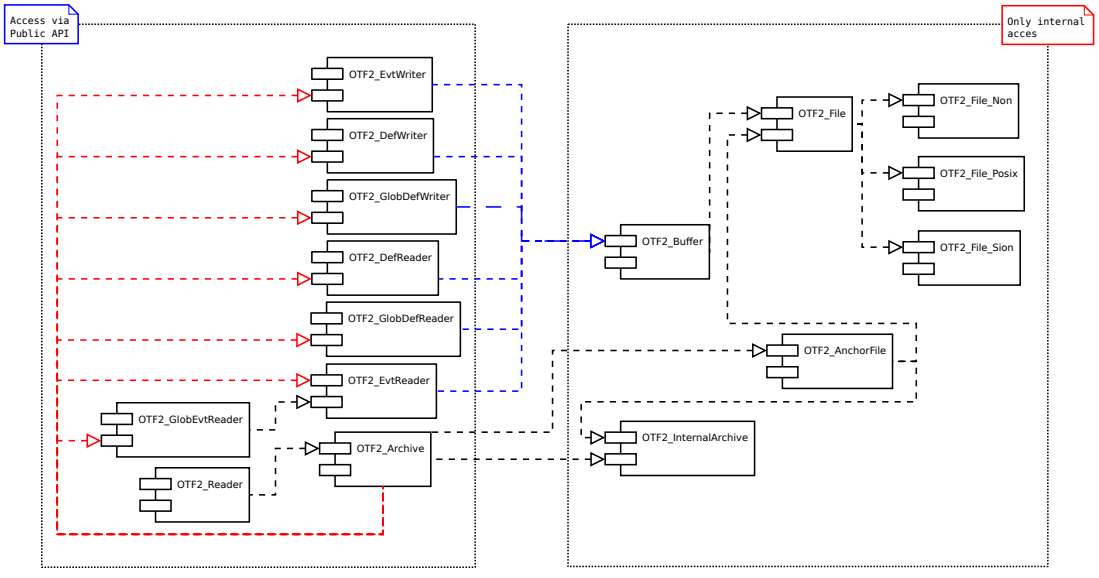

Figure 3.2: Class diagram of the main components of the OTF2 library.

The programmer always gets an `OTF2_Archive` object, if he opens a new trace archive for writing, or an `OTF2_Reader` object, if he opens an existing trace archive for reading. The archive/reader implements methods to create new reader/writer objects for all available locations. Afterwards, a hypothetic trace producer or consumer can handle a trace event or definition stream with these reader or writer objects. The reader and writer objects handle memory accesses not directly, but with help of the `OTF2_Buffer` component. Each reader and writer object has its own buffer object. The `OTF2_Buffer` object handles the on-line compression, as well as the chunk- and memory management transparently. A buffer runs full if there is no free memory left. The `OTF2_Buffer` object does automatically flush the buffer content into a file, with help of the `OTF2_File` component. `OTF2_File` is an abstraction layer for file or I/O handling and can be extended with different substrates. It also handles compression transparently.

3.2 External Components

3.2.1 Global Definition Writer

This module is designed to write globally defined definition records into a memory buffer. The global definition writer can be found in file `OTF2_GlobDefWriter.c` and `OTF2_GlobDefWriter_inc.c`.

Not Generated Routines

A new global definition writer object can be generated by using the related `OTF2_GlobDefWriter_New` function. It is not intended that the programmer uses it, usually the new function is

called by the archive management (see Section 3.2.8).

```
OTF2_GlobDefWriter*                                                    1
OTF2_GlobDefWriter_New                                                 2
(                                                                      3
    const char*            archivePath,                                4
    const char*            archiveName,                                5
    OTF2_FileSubstrate     substrate,                                  6
    OTF2_Compression       compression,                                7
    const uint64_t         chunkSize,                                  8
    OTF2_PreFlushCallback  preFlush,                                   9
    OTF2_PostFlushCallback postFlush                                   10
);                                                                     11
```

The global definition writer module has also a delete function to free resources etc. The delete function should only be called by the archive management, like the related new function. Both new and delete functions are excluded for external use by a preprocessor macro (`OTF2_INTERNAL`).

```
SCOREP_Error_Code                                                      1
OTF2_GlobDefWriter_Delete                                              2
(                                                                      3
    OTF2_GlobDefWriter* writerHandle                                   4
);                                                                     5
```

The number of written definitions or by a specific global definition writer instance can be accessed by:

```
SCOREP_Error_Code                                                      1
OTF2_GlobDefWriter_GetNumberOfDefinitions                              2
(                                                                      3
    OTF2_GlobDefWriter* writerHandle,                                  4
    uint64_t*           numberOfDefinitions                            5
);                                                                     6
```

Also the number of locations of the whole trace archive can be accessed by:

```
SCOREP_Error_Code                                                      1
OTF2_GlobDefWriter_GetNumberOfLocations                                2
(                                                                      3
    OTF2_GlobDefWriter* writerHandle,                                  4
    uint64_t*           numberOfLocations                              5
);                                                                     6
```

Generated Routines

The generated part of the global definition writer component can be found in `OTF2_GlobDefWriter_inc.c`. The functions, which actually write the definition data into a buffer, are generated. Each definition record is written by a dedicated function. This is done according to the definition record specification in `OTF2_GlobDefinitions.h`:

```
SCOREP_Error_Code                                                      1
OTF2_GlobDefWriter_<record name>( OTF2_GlobDefWriter* writerHandle,    2
                                  <all record attributes>);            3
```

Please also see Section 3.2.2 to see how this module is related to the local definition writer.

3.2.2 Local Definition Writer

This module is designed to write mapping tables into a memory buffer. Furthermore, it is also capable to write locally defined definition records. Nevertheless, it is highly recommended to write definitions always globally (see Section 3.2.1) since known tools, which are able to read OTF2 traces, are not able to handle this. The local definition writer can be found in the files `OTF2_DefWriter.c` and `OTF2_DefWriter_inc.c`.

Not Generated Routines

A new local definition writer object can be generated by using the related `OTF2_DefWriter_New` function. It is not intended that the programmer uses it directly, usually the new function is called by the archive management (see Section 3.2.8).

```
OTF2_DefWriter*
OTF2_DefWriter_New
(
    const char*            archivePath,
    const char*            archiveName,
    const uint64_t         locationID,
    OTF2_FileSubstrate     substrate,
    OTF2_Compression       compression,
    const uint64_t         chunkSize,
    OTF2_PreFlushCallback  preFlush,
    OTF2_PostFlushCallback postFlush
);
```

The local definition writer module has also a delete function to free resources etc. Like the related new function, the delete should only be called by the archive management. Both new and delete functions are excluded for external use by a preprocessor macro (`OTF2_INTERNAL`).

```
SCOREP_Error_Code
OTF2_DefWriter_Delete
(
    OTF2_DefWriter* writerHandle
);
```

The writer's location ID can be accessed by using `OTF2_DefWriter_GetLocationID`.

```
SCOREP_Error_Code
OTF2_DefWriter_GetLocationID
(
    const OTF2_DefWriter* writerHandle,
    uint64_t*             locationID
);
```

The local event writer has the main purpose to store mapping tables. Those tables are needed for identifiers (ID) in the event stream which are not globally visible during runtime. A written mapping table is applied automatically when the trace is read by the local event reader (see Section 3.2.6). The given map type signals which ID type can be mapped by the related table.

```
SCOREP_Error_Code
OTF2_DefWriter_WriteMappingTable
(
    const OTF2_DefWriter* writerHandle,
    const SCOREP_IdMap*   iDMap,
    OTF2_MappingType      mapType
);
```

OTF2 currently supports the following mapping table types:

```
enum OTF2_MappingType_enum
{
    OTF2_MAPPING_LOCATION         =  0,
    OTF2_MAPPING_REGION           =  1,
    OTF2_MAPPING_GROUP            =  2,
    OTF2_MAPPING_WINDOW           =  3,
    OTF2_MAPPING_GATS_GROUP       =  4,
    OTF2_MAPPING_MPI_COMMUNICATOR =  5,
    OTF2_MAPPING_METRIC           =  6,
    OTF2_MAPPING_ATTRIBUTE        =  7,
    OTF2_MAPPING_PARAMETER        =  8,
    OTF2_MAPPING_LOCAL_STRING     =  9,
    OTF2_MAPPING_GLOBAL_STRING    = 10
};
```

Generated Routines

The generated part of the local definition writer component can be found in `OTF2_DefWriter_inc.c`. The functions, which actually write the definition data into a buffer, are generated. Each definition record is written by a dedicated function. This is done according to the definition record specification in `OTF2_LocalDefinitions.h`:

```
SCOREP_Error_Code
OTF2_DefWriter_<record name>( OTF2_DefWriter* writerHandle,
                              <all record attributes>);
```

3.2.3 Local Event Writer

The local event writer can be found in the files `OTF2_EvtWriter.c` and `OTF2_EvtWriter_inc.c`. This module provides write routines to store event records of a single location into an OTF2 memory buffer.

Not Generated Routines

A new local event writer object can be generated by using the related `OTF2_EvtWriter_New` function. It is not intended that the programmer uses it, usually the new function is called by the archive management (see Section 3.2.8).

```
OTF2_EvtWriter*
OTF2_EvtWriter_New
(
    uint64_t               locationID,
    char*                  archivePath,
    char*                  archiveName,
    OTF2_FileSubstrate     substrate,
    OTF2_Compression       compression,
    uint64_t               chunksize,
    OTF2_PreFlushCallback  preFlush,
    OTF2_PostFlushCallback postFlush,
    OTF2_MemoryAllocate    allocate,
    OTF2_MemoryFree        free,
    void*                  allocatorData
);
```

The local event writer module has also a delete function, to free resources etc. Delete should only be called by the archive management, like the related new function. Both new and delete functions are excluded for external use by a preprocessor macro (`OTF2_INTERNAL`).

```
SCOREP_Error_Code
OTF2_EvtWriter_Delete
(
    OTF2_EvtWriter* writer
);
```

The location ID of a specific event writer instance can be accessed by:

```
SCOREP_Error_Code
OTF2_EvtWriter_GetLocationID
(
    const OTF2_EvtWriter* writer,
    uint64_t*             locationID
);
```

The number of written events by a specific event writer instance can be accessed by:

```
SCOREP_Error_Code
OTF2_EvtWriter_GetNumberOfEvents
(
    OTF2_EvtWriter* writer,
    uint64_t*       numberOfEvents
);
```

It may not be possible in some cases to set the location ID when the writer object is generated. This is for instance the case for MPI measurements, because the MPI initialization is done later than the program initialization. Therefore, the location ID can be set before the first written event:

```
SCOREP_Error_Code
OTF2_EvtWriter_SetLocationID
(
    OTF2_EvtWriter* writer,
    uint64_t        locationID
);
```

The same for the archive and trace-path:

```
SCOREP_Error_Code
otf2_evtwriter_set_archive_path
(
    OTF2_EvtWriter* writer,
    char*           archivePath
);

SCOREP_Error_Code
otf2_evtwriter_set_trace_path
(
    OTF2_EvtWriter* writer
);
```

The functionality to write dynamic attribute lists is also implemented here. The related function is used in each generated writer function of this module.

```
static inline SCOREP_Error_Code
otf2_evt_writer_write_attribute_list
(
    const OTF2_EvtWriter* writerHandle,
    OTF2_AttributeList*   attributeList,
    OTF2_TimeStamp        time
);
```

Generated Routines

The generated part of the local event writer component can be found in `OTF2_EvtWriter_inc.c`. The functions, which actually write the the event data into a buffer, are generated. Each event record is written by a dedicated function. This is done according to the event record specification in `OTF2_Events.h`:

```
SCOREP_Error_Code
OTF2_EvtWriter_<record name>( OTF2_EvtWriter*     writerHandle,
                              OTF2_AttributeList* attributeList,
                              OTF2_TimeStamp      time,
                              <all record attributes without time> );
```

3.2.4 Global Definition Reader

The global definition reader module is used to read globally defined definitions, which where written by the global definition writer modules before. The global definition reader can be found in file `OTF2_GlobDefReader.c` and `OTF2_GlobDefReader_inc.c`. The interface is very simple, the programmer can register several different callback functions by specifying their function pointers. The programmer also needs to trigger one read function, to read out the global definitions. Whenever the reader module reads a record, a callback function for the related record type is triggered and all record attributes are passed to that function. The record is ignored, if the programmer did not specify a callback function for this type.

Not Generated Routines

A new global definition reader object can be generated by using the related `OTF2_GlobDefReader_New` function. It is not intended that the programmer uses it, usually the new function is called by the archive management (see Section 3.2.7).

```
OTF2_GlobDefReader*
OTF2_GlobDefReader_New
(
    char*                       archivePath,
    OTF2_InternalArchiveData* archive
);
```

The global definition reader module implements also a delete function to free resources etc. The delete should only be called by the archive management, like the related new function. Both new and delete functions are excluded for external use by a preprocessor macro (`OTF2_INTERNAL`).

```
SCOREP_Error_Code
OTF2_GlobDefReader_Delete
(
    OTF2_GlobDefReader* reader
);
```

The programmer must register a callback function for each record type before any global definition record could be read. All callbacks are passed together in one data structure, which has entries for each type. This data structure and all callback types are generated. The programmer also has the opportunity to pass a own pointer to this call, which is passed to each callback afterwards. It can be used to provide the callbacks with storage space etc.

```
SCOREP_Error_Code
OTF2_GlobDefReader_SetCallbacks
(
```

```
    OTF2_GlobDefReader*         reader,
    OTF2_GlobDefReaderCallbacks callbacks,
    void*                       userData
);
```

The definitions could be read with the following function after callback registration. The user of this function tells the system how many definitions he is able to handle (`recordsToRead`) and the function returns how many definitions where in the stream (`recordsRead`). There where less records than requested in the stream, if both values are not the same. This can be used to read out the whole stream at once, by giving `UINT64MAX` for `recordsToRead`.

```
SCOREP_Error_Code
OTF2_GlobDefReader_ReadDefinitions
(
    OTF2_GlobDefReader* reader,
    uint64_t            recordsToRead,
    uint64_t*           recordsRead
);
```

Generated Routines

The generated part of the global definition reader component can be found in `OTF2_GlobDefReader_inc.c`. The generation of these functions is done according to the definition record specification in `OTF2_GlobDefinitions.h`:

A function pointer type is generated for each record that can be found in `OTF2_GlobDefinitions.h`.

```
typedef SCOREP_Error_Code
( *OTF2_GlobDefReaderCallback_<record name> )( void*    userdata,
                                               <all record attributes> );
```

The data structure, which is used to register all callbacks, is also generated. It has an entry for each callback type.

```
typedef struct OTF2_GlobDefReaderCallbacks_struct
{
    OTF2_GlobDefReaderCallback_<record name> <record name>;
    ...
} OTF2_GlobDefReaderCallbacks;
```

The function which reads one single record makes the decision which record type was read, and delegates further reading to a dedicated read function. Each record type has its own read function.

```
SCOREP_Error_Code
OTF2_GlobDefReader_Read( OTF2_GlobDefReader* reader )
{
    uint8_t record_type = 0;
    OTF2_Buffer_ReadUint8( reader->buffer, &record_type );

    switch ( record_type )
    {
        case OTF2_GLOB_DEF_END_OF_FILE:
            return SCOREP_ERROR_INDEX_OUT_OF_BOUNDS;

        case <record name>:
            return otf2_glob_def_reader_<record name>( reader );
...
```

The dedicated read function reads the rest of the record.

```
static inline SCOREP_Error_Code
otf2_glob_def_reader_<record name>( OTF2_GlobDefReader* reader );
```

3.2.5 Local Definition Reader

The local definition reader module is used to read locally defined mapping tables, which where written by the local definition writer modules before. Mapping tables are needed in cases when a specific ID is only known locally during runtime. Those IDs are usually unified after the measurement to make them globally unique. Mapping offers therefore a lightweight technique (compared to trace rewriting) to modify those IDs on reading. The local definition reader can also read local definition records. However, definitions should not be located in the local definition file, because of several problems regarding unification and semantics.

The local definition reader can be found in file `OTF2_DefReader.c` and `OTF2_DefReader_inc.c`. The interface is very simple, the programmer can register several different callback functions by specifying their function pointers. The programmer also needs to trigger one read function, to read out the global definitions. The mapping table is automatically passed to the local event reader of the same location whenever the reader reads a mapping table. Furthermore, whenever the reader module reads a record, a callback function for the related record type is triggered and all record attributes are passed to that function. The record is ignored, if the programmer did not specify a callback function for this type.

Not Generated Routines

A new local definition reader object can be generated by using the related `OTF2_DefReader_New` function. It is not intended that the programmer uses it, usually the new function is called by the archive management (see Section 3.2.8).

```
OTF2_DefReader*
OTF2_DefReader_New
(
    uint64_t                  locationID,
    OTF2_InternalArchiveData* archive
);
```

The local definition reader module has also a delete function to free resources etc. The delete should only be called by the archive management, like the related new function. Both new and delete functions are excluded for external use by a preprocessor macro (`OTF2_INTERNAL`).

```
SCOREP_Error_Code
OTF2_DefReader_Delete
(
    OTF2_DefReader* reader
);
```

The programmer must register a callback function for each record type, before any local definition record could be read. All callbacks are passed together in one data structure, which has entries for each type. This data structure and all callbacks types are generated. The programmer also has the opportunity to pass an own pointer to this call, which is passed to each callback afterwards. The pointer can be used to provide the callbacks with storage space etc.

```
SCOREP_Error_Code
OTF2_DefReader_SetCallbacks
(
    OTF2_DefReader*         reader,
```

```
    OTF2_DefReaderCallbacks callbacks,
    void*                   userData
);
```

The definitions could be read with the following function after callback registration. The user of this function tells the system how many definitions he is able to handle (`recordsToRead`) and the function returns how many definitions were left in the stream (`recordsRead`). There were less records than requested in the stream, if both values are not the same. This can be used to read out the whole stream at once, by giving `UINT64MAX` for `recordsToRead`. The read function is also responsible to read mapping tables, since the main purpose of this module is to read and to pas them to the event reader module.

```
SCOREP_Error_Code
OTF2_DefReader_ReadDefinitions
(
    OTF2_DefReader* reader,
    uint64_t        recordsToRead,
    uint64_t*       recordsRead
);
```

The function to read a mapping table from a buffer instance is always triggered, whenever the generated part encounters a mapping table inside the record stream. This function also gets the related event reader object from the archive management and passes the mapping table to it.

```
static inline SCOREP_Error_Code
otf2_def_reader_read_mapping_table
(
    OTF2_DefReader* reader
);
```

Generated Routines

The generated part of the local definition reader component can be found in `OTF2_DefReader_inc.c`. The generation of these functions is done according to the definition record specification in `OTF2_LocalDefinitions.h`:

A function pointer type is generated for each record that can be found in `OTF2_LocalDefinitions.h`.

```
typedef SCOREP_Error_Code
( *OTF2_DefReaderCallback_DefString )( void*    userdata,
                                       uint32_t string_identifier,
                                       char*    string );
```

The data structure, which is used to register all callbacks, is also generated. It has an entry for each callback type.

```
typedef struct OTF2_DefReaderCallbacks_struct
{
    OTF2_DefReaderCallback_DefString DefString;
    ...
} OTF2_DefReaderCallbacks;
```

The function which reads one single record makes the decision which record type was read, and delegates further reading to a dedicated read function. Each record type has its own dedicated read function.

```
SCOREP_Error_Code
OTF2_DefReader_Read( OTF2_DefReader* reader )
```

```
{
    uint8_t record_type = 0;
    OTF2_Buffer_ReadUint8( reader->buffer, &record_type );

    switch ( record_type )
    {
        case OTF2_DEF_END_OF_FILE:
            return SCOREP_ERROR_INDEX_OUT_OF_BOUNDS;

        case <record name>:
            return otf2_def_reader_read_<record name>( reader );
...
```

The dedicated read function reads the rest of the record.

```
static inline SCOREP_Error_Code
otf2_def_reader_def_<record name>( OTF2_DefReader* reader );
```

3.2.6 Local Event Reader

The local event reader can be found in file `OTF2_EvtReader.c` and `OTF2_EvtReader_inc.c`. The local event reader is the most complex module in the event layer (see Section 3.1) since it implements functions for forward and backward reading as well as seeking on a compressed binary stream.

Not Generated Routines

A new local event reader object can be generated by using the related `OTF2_EvtReader_New` function. It is not intended that the programmer uses it directly, usually the new function is called by the archive management (see Section 3.2.8).

```
OTF2_EvtReader*
OTF2_EvtReader_New
(
    uint64_t                  locationID,
    OTF2_InternalArchiveData* archive
);
```

The local event reader module implements also a delete function, to free resources etc. Delete should only be called by the archive management, like the related new function. Both new and delete functions are excluded for external use by a preprocessor macro. (`OTF2_INTERNAL`).

```
SCOREP_Error_Code
OTF2_EvtReader_Delete
(
    OTF2_EvtReader* reader
);
```

The location ID of a specific event reader instance can be accessed by:

```
SCOREP_Error_Code
OTF2_EvtReader_GetLocationID
(
    const OTF2_EvtReader* reader,
    uint64_t*             locationID
);
```

The programmer must register a callback function for each record type, before any local event record could be read. All callbacks are passed together in one data structure, which has entries for each type. This data structure and all callbacks types are generated. The programmer also has the opportunity to pass an own pointer to this call, which is passed to each callback afterwards. This pointer can be used to provide the callbacks with storage space etc.

```
SCOREP_Error_Code
OTF2_EvtReader_SetCallbacks
(
    OTF2_EvtReader*      reader,
    OTF2_ReaderCallbacks callbacks,
    void*                userData
);
```

It may happen that a program needs to know the index position of the current event. This is also very important, if someone wants to seek. Please note that the following function gets the index position (n) of the nth record in the whole trace and not of a memory address or simmilar.

```
SCOREP_Error_Code
OTF2_EvtReader_GetPos
(
    OTF2_EvtReader* reader,
    uint64_t*       position
);
```

The seek function can bes used to seek to the nth record in the whole trace.

```
SCOREP_Error_Code
OTF2_EvtReader_Seek
(
    OTF2_EvtReader* reader,
    uint64_t        position
);
```

The reader component needs to index the currently used chunk internally, if seeking or backward reading is requested. This is needed because the on-line compression makes it impossible to have fixed starting addresses for records. The resulting index table for the current chunk is stored into the reader object. An index table is only generated, if there is not already an existing table for the current chunk.

```
static inline SCOREP_Error_Code
OTF2_EvtReader_Index
(
    OTF2_EvtReader* reader
);
```

The reader component also implements a function to read n records in a row. It does it by calling the function `OTF2_EvtReader_Read` n times. A programmer should use this function to read out a trace by passing `UINT64_MAX` as the value for `recordsToRead`. The pointer `recordsRead` will return the current number of events that was read.

```
SCOREP_Error_Code
OTF2_EvtReader_Readn
(
    OTF2_EvtReader* reader,
    uint64_t        recordsToRead,
    uint64_t*       recordsRead
);
```

The same like `OTF2_EvtReader_Readn` does, can be done with the following function in backward direction.

```
SCOREP_Error_Code
OTF2_EvtReader_ReadnBackward
(
    OTF2_EvtReader* reader,
    uint64_t        recordsToRead,
    uint64_t*       recordsRead
);
```

`OTF2_EvtReader_ReadBackward` does the same like `OTF2_EvtReader_Read` in backward direction. The backward reading function is not generated, because it uses the already generated `OTF2_EvtReader_Read` function. Figure 3.3 shows how this function interacts with the function for forward reading.

```
SCOREP_Error_Code
OTF2_EvtReader_ReadBackward
(
    OTF2_EvtReader*    reader,
    OTF2_GenericEvent* record,
    OTF2_EventType*    type
);
```

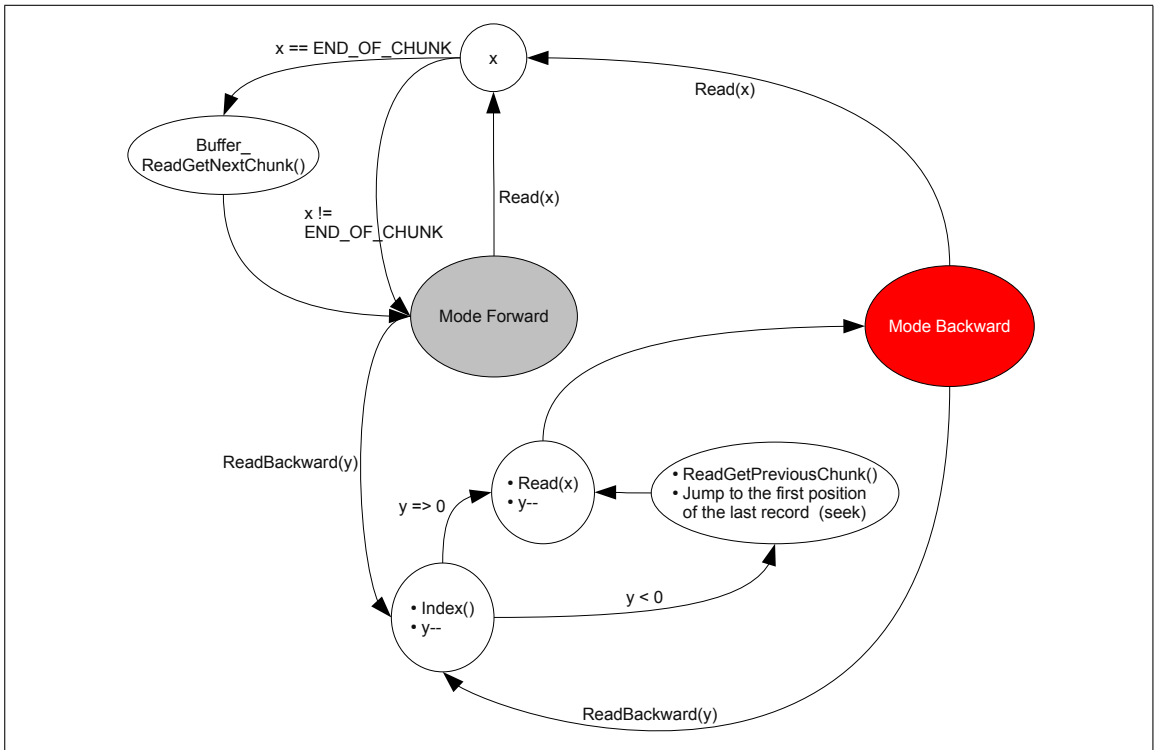

Figure 3.3: State diagram which describes the algorithmic kernel of the local event reader.

The timestamp of the the current record can also be rewritten. This functionality can be used for timestamp synchronization etc.

```
SCOREP_Error_Code
OTF2_EvtReader_TimeStampRewrite
(
    OTF2_EvtReader* reader,
    OTF2_TimeStamp  time
);
```

Mapping tables are applied automatically if they where found in the local definition file. This function is used internally by the local definition reader to pass the mapping tables to the local event reader.

```
SCOREP_Error_Code
OTF2_EvtReader_SetMappingTable
(
    OTF2_EvtReader*  reader,
    SCOREP_IdMap*    iDMap,
    OTF2_MappingType mapType
);
```

Internally the local event reader maps ID with the help of the following function.

```
static inline
uint32_t
otf2_evt_reader_map
(
    OTF2_EvtReader*  reader,
    OTF2_MappingType mapType,
    uint32_t         localID
);
```

Attribute lists are handled by the following functions.

```
SCOREP_Error_Code
OTF2_EvtReader_GetAttributeList
(
    const OTF2_EvtReader* reader,
    OTF2_AttributeList**  attributeList
);

static inline SCOREP_Error_Code
otf2_evt_reader_read_attribute_list
(
    OTF2_EvtReader* reader
);

static inline
SCOREP_Error_Code
otf2_evt_reader_skip_attribute_list
(
    OTF2_EvtReader* reader
);

SCOREP_Error_Code
OTF2_EvtReader_OperatedByGlobalReader
(
    OTF2_EvtReader* reader
);
```

Generated Routines

The generated part of the local event reader component can be found in `OTF2_EvtReader_inc.c`.

The first type of generated functions, are the functions which read out a whole record and pass the resulting data to a callback function. The local event reader implements such a function for each each record type.

```
static inline SCOREP_Error_Code
otf2_evt_reader_<record name>( OTF2_EvtReader*    reader,
```

```
                                OTF2_GenericEvent* record,                    3
                                ... );                                        4
```

The function that actually reads a record in forward direction is also generated, because it must implement a big `switch-case` statement to decide which type of record has to be read next. It also needs to implement some extra management logic to be able to switch between forward and backward reading mode. How that works can be seen in Figure 3.3. Usually the reader is operated in forward reading mode, where it just reads one record per call. Furthermore, it triggers `ReadGetNextChunk()` if the end of the current chunk is reached. The local event reader triggers the indexing function, whenever the `ReadBackward()` function (see bellow) is triggered.

```
SCOREP_Error_Code                                                             1
OTF2_EvtReader_Read( OTF2_EvtReader*     reader,                              2
                    OTF2_GenericEvent* record,                                3
                    OTF2_EventType*    type );                                4
```

Indexing of a chunk can only be done by traversing the whole chunk (see Figure 3.4), because of the variable byte length of the compressed values. A full decompression is not needed for indexing, because the reader just needs to read the byte length and jump to the next value. This is implemented by so called *skip-functions*, which skip the attributes of a complete record. *Skip-functions* make it possible to jump immediately to the beginning of the next chunk. The reader implements such a function for each related read function.

```
static inline SCOREP_Error_Code                                               1
otf2_evt_reader_measurement_<record name>_skip( OTF2_EvtReader* reader );     2
```

The *skip-functions* are triggered by the central generated`IndexSkip` function.

```
SCOREP_Error_Code                                                             1
OTF2_EvtReader_IndexSkip( OTF2_EvtReader* reader );                           2
```

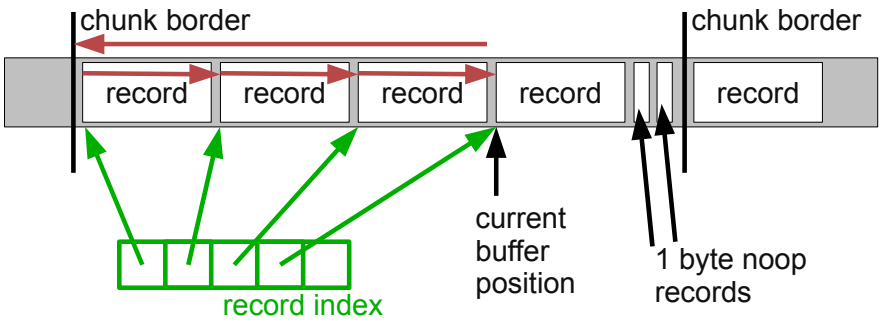

Figure 3.4: A chunk needs to be indexed on backward reading, because there are no fixed entry points.

3.2.7 Global Event Reader

The global event reader can merge the event streams from several different locations. This could be useful when a trace needs to be read on a single computer. Merging is done by sorting the event records by their timestamps and by attaching an additional location attribute to each record. The global event reader can be found in the files `OTF2_GlobEvtReader.c` and `OTF2_GlobEvtReader_inc.c`.

Not Generated Routines

A new global event reader object can be generated by using the related `OTF2_GlobEvtReader_New` function. It is not intended that the programmer uses it, usually the new function is called by the archive management (see Section 3.2.8).

```
OTF2_GlobEvtReader*
OTF2_GlobEvtReader_New
(
    SCOREP_Vector* localReaders
);
```

The global event reader module has also a delete function, to free resources etc. Delete should only be called by the archive management like the related new function. Both new and delete functions are excluded for external use by a preprocessor macro.

```
SCOREP_Error_Code
OTF2_GlobEvtReader_Delete
(
    OTF2_GlobEvtReader* readerHandle
);
```

The programmer must register a callback function for each record type, before any local event record could be read. All callbacks are passed together in one data structure, which has entries for each record type. This data structure and all callbacks types are generated. The programmer also has the opportunity to pass an own pointer to this call, which is passed to each callback afterwards. The pointer can be used to provide the callbacks with storage space etc.

```
SCOREP_Error_Code
OTF2_GlobEvtReader_SetCallbacks
(
    OTF2_GlobEvtReader*         readerHandle,
    OTF2_GlobEvtReaderCallbacks callbacks,
    void*                       userData
);
```

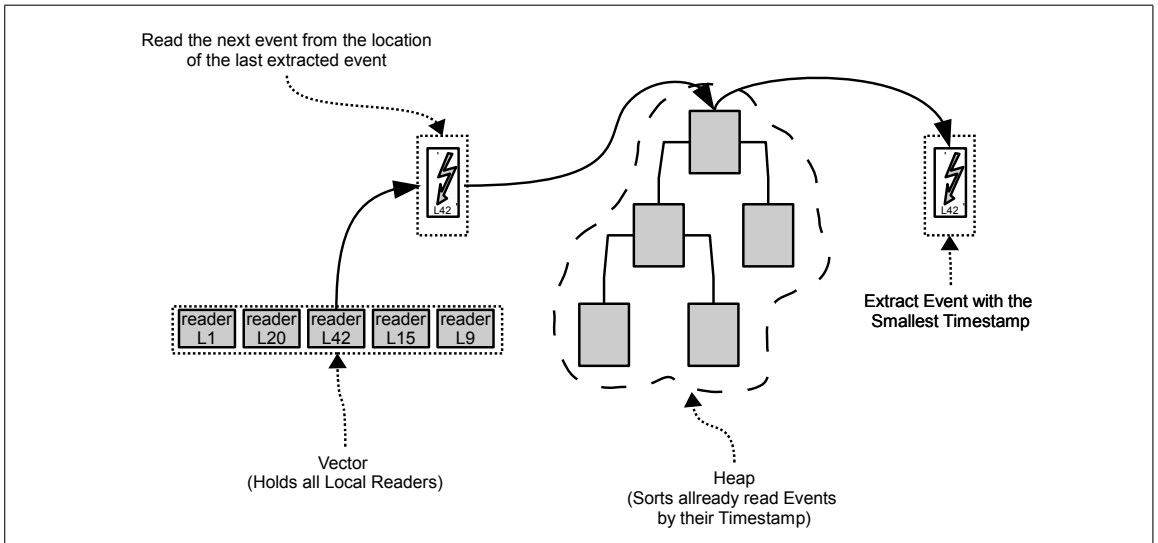

Figure 3.5: The reader extracts the first event from the heap, and puts the next event from the extracted location into the heap.

The global event reader does not directly implement functions for binary decoding. Instead it used local event readers for each selected location. Initially, one local event reader object for each location is therefore placed in a vector and the first event is placed in a heap. The heap sorts all extracted

events by their timestamps. Reading is implemented in a function that just reads the next event. The heap therefore selects the next event by its timestamp and the reader delivers it to the callback interface. Afterwards, the reader inserts the next event from the location, where the extracted event came from, into the heap.

```
SCOREP_Error_Code
OTF2_GlobEvtReader_Read
(
    const OTF2_GlobEvtReader* readerHandle
);
```

The global event reader also implements a function which reads n events. Therefore it simply loops n-times over the `OTF2_GlobEvtReader_Read` function.

```
SCOREP_Error_Code
OTF2_GlobEvtReader_Readn
(
    const OTF2_GlobEvtReader* readerHandle,
    const uint64_t            recordsToRead,
    uint64_t*                 recordsRead
);
```

Generated Routines

The generated part of the local event reader component can be found in `OTF2_GlobEvtReader_inc.c`.

The global event reader implements one handler function for each record type to deliver the data to the callback interface.

```
static inline SCOREP_Error_Code
otf2_glob_evt_reader_<record name>( const OTF2_GlobEvtReader* reader,
                                    const OTF2_GenericEvent*  record,
                                    uint64_t                  locationID,
                                    OTF2_TimeStamp            time,
                                    void*                     userdata,
                                    OTF2_AttributeList*       attributeList )
```

These functions are used by a trigger function, which delegates an input record to the depending handler function.

```
static SCOREP_Error_Code
OTF2_GlobEvtReader_TriggerCallback( const OTF2_GlobEvtReader* reader,
                                    const OTF2_GenericEvent*  record,
                                    uint64_t                  locationID,
                                    OTF2_TimeStamp            time,
                                    void*                     userdata,
                                    OTF2_AttributeList*       attributeList,
                                    uint8_t                   recordType )
{
    switch ( recordType )
    {
        case <record name>:
            return OTF2_GlobEvtReader_<record name>( reader,
                                                     record,
                                                     locationID,
                                                     <attributes>
                                                   );
...
```

3.2.8 Archive

The archive class is an external module which can be used by the programmer to load, store and generate OTF2 archives. It is basically the memory representation of the meta-data of an OTF2 archive. The archive is also used to manage reader and writer handles to avoid that two or more writer objects try to write into the same file. Therefore it keeps a reference pointer to each generated reader or writer object and returns only the pointer to an already existing object. The archive implementation can be found in the file `OTF2_Archive.c`.

A new archive object can be generated by using the related `OTF2_Archive_New` function. The arguments `archivePath` and `archiveName` are used to generated file paths internally. `file Mode` can be set to `OTF2_FILEMODE_<WRITE>|<READ>|<MODIFY>`. The modify mode allows to modify timestamps in an existing trace. Chunk sizes for definition and event buffers can be set with `chunkSizeEvents` and `chunkSizeDefs` (please see also Section 2.3). The argument `fileSubstrate` is used to set the substrate for file writing (please see also Section 3.3.4). The argument `compression` is used to set the substrate to compress the data (please see also Section 3.3.4). OTF2 is also capable to handle external memory allocators, for example to realize memory pooling etc. A programmer can therefore pass callbacks, for allocating (`allocate`) and freeing (`free`) data, to the archive.

```
OTF2_Archive*                                                    1
OTF2_Archive_New                                                 2
(                                                                3
    const char*              archivePath,                        4
    const char*              archiveName,                        5
    const OTF2_FileMode      fileMode,                           6
    const uint64_t           chunkSizeEvents,                    7
    const uint64_t           chunkSizeDefs,                      8
    const OTF2_FileSubstrate fileSubstrate,                      9
    const OTF2_Compression   compression,                        10
    OTF2_MemoryAllocate      allocate,                           11
    OTF2_MemoryFree          free,                               12
    void*                    allocatorData                       13
);                                                               14
```

An archive object needs to be destroyed after trace recording. This can be done with the related delete function. A deletion of an archive also deletes all opened event reader and writer objects.

```
SCOREP_Error_Code                                                1
OTF2_Archive_Delete                                              2
(                                                                3
    OTF2_Archive* archive                                        4
);                                                               5
```

Only one process writes the global definitions and the anchor file. The information that a specific object is a master or not must be passed with a special function, since OTF2 is programming paradigm unaware.

```
SCOREP_Error_Code                                                1
OTF2_Archive_SetMasterSlaveMode                                  2
(                                                                3
    OTF2_Archive*        archive,                                4
    OTF2_MasterSlaveMode masterOrSlave                           5
);                                                               6
                                                                 7
SCOREP_Error_Code                                                8
OTF2_Archive_GetMasterSlaveMode                                  9
(                                                                10
```

```
    OTF2_Archive*         archive,
    OTF2_MasterSlaveMode* masterOrSlave
);
```

Meta-Data

The functions described in this part are used to set the trace meta-data that is stored into the anchor file (see Section 2.2) on trace finalization.

The following function sets `LOCATION_NUMBER` in the anchor file.

```
SCOREP_Error_Code
OTF2_Archive_SetNumberOfLocations
(
    OTF2_Archive* archive,
    uint64_t      locations
);

SCOREP_Error_Code
OTF2_Archive_GetNumberOfLocations
(
    OTF2_Archive* archive,
    uint64_t*     locations
)
```

The number of global definitions is counted by the related writer object. However, the following function can be used to get the current number of global definition records that written up to now.

```
SCOREP_Error_Code
OTF2_Archive_GetNumberOfGlobalDefinitions
(
    OTF2_Archive* archive,
    uint64_t*     definitions
)
```

The following function can be used to write the name of the machine (for example Jugene), where the trace was recorded, to the anchor file. The related attribute there is `MACHINE_NAME`.

```
SCOREP_Error_Code
OTF2_Archive_SetMachineName
(
    OTF2_Archive* archive,
    const char*   machineName
);

SCOREP_Error_Code
OTF2_Archive_GetMachineName
(
    OTF2_Archive* archive,
    char**        machineName
);
```

An trace archive sometimes needs to be described, to make them distinguishable. The related function sets the attribute `DESCRIPTION` at the anchor file.

```
SCOREP_Error_Code
OTF2_Archive_SetDescription
(
    OTF2_Archive* archive,
    const char*   description
);
```

```
                                                                    7
SCOREP_Error_Code                                                   8
OTF2_Archive_GetDescription                                         9
(                                                                   10
    OTF2_Archive* archive,                                          11
    char**        description                                       12
);                                                                  13
```

The creator tag can be used to identify the author of a trace. `CREATOR` is the related attribute at the anchor file.

```
SCOREP_Error_Code                                                   1
OTF2_Archive_SetCreator                                             2
(                                                                   3
    OTF2_Archive* archive,                                          4
    const char*   creator                                           5
);                                                                  6
                                                                    7
SCOREP_Error_Code                                                   8
OTF2_Archive_GetCreator                                             9
(                                                                   10
    OTF2_Archive* archive,                                          11
    char**        creator                                           12
);                                                                  13
```

Reader and Writer Components

Reader and Writer objects have to be generated from the archive, to avoid that a program generates two different writer objects for the same trace file. The programmer must therefore pass a valid location ID and two different callbacks to the related functions. The callbacks are triggered before and after a buffer flush. Please read also Sections 3.2.3, 3.2.2, and 3.2.1 for further information.

```
OTF2_EvtWriter*                                                     1
OTF2_Archive_Get<Evt|Def|GlobDef>Writer                             2
(                                                                   3
    OTF2_Archive*          archive,                                 4
    const uint64_t         locationID,                              5
    OTF2_PreFlushCallback  preFlush,                                6
    OTF2_PostFlushCallback postFlush                                7
);                                                                  8
```

Also reader objects need to be generated by the central archive management. This offers the opportunity to handle process local data (for example just one definition file per process). Please read also Sections 3.2.7, 3.2.4, 3.2.3, and 3.2.2 for further information.

```
OTF2_GlobEvtReader*                                                 1
OTF2_Archive_Get<GlobEvt|GlobDef>Reader                             2
(                                                                   3
    OTF2_Archive* archive                                           4
);                                                                  5
                                                                    6
OTF2_DefReader*                                                     7
OTF2_Archive_Get<Evt|Def>Reader                                     8
(                                                                   9
    OTF2_Archive*  archive,                                         10
    const uint64_t locationID                                       11
);                                                                  12
```

Local reader and writer objects can be deleted with the following functions. The global definition writer or reader is deleted on trace archive finalization.

```
SCOREP_Error_Code
OTF2_Archive_Close<EvtWriter|DefWriter|EvtReader|DefReader>
(
    OTF2_Archive*   archive,
    OTF2_<EvtWriter|DefWriter|EvtReader|DefReader>* writer
);
```

3.2.9 Reader

The reader component is designed to make it possible to extend OTF2 for reading capabilities of other formats. It was initially designed to make the implementation of backward compatibility readers for EPILOG and OTF easier. However, the current version of OTF2 does only support OTF2 traces, but further reader components are planned. A programmer should therefore use the reader API instead of the archive API (described in Section 3.2.8) to read a trace. The reader implementation can be found in the file `OTF2_Reader.c`.

A new archive object can be generated by using the related `OTF2_Reader_New` function. This function does only need the path of the anchor file, since all other meta-data (like chunk sizes etc.) is already stored in the anchor file.

```
OTF2_Reader*
OTF2_Reader_New
(
    const char* anchorFilePath
);
```

A reader object needs to be destroyed after trace recording. This can be done with the related delete function. A deletion of a reader also deletes all opened event and definition reader objects.

```
SCOREP_Error_Code
OTF2_Reader_Delete
(
    OTF2_Reader* reader
);
```

Reader Creation and Deletion

Reader objects need to be generated before any event can be read. This is similar to how it is done with the archive component (see Section 3.2.8).

```
OTF2_EvtReader*
OTF2_Reader_GetEvtReader
(
    OTF2_Reader* reader,
    uint64_t     locationID
);

OTF2_GlobEvtReader*
OTF2_Reader_GetGlobEvtReader
(
    OTF2_Reader* reader
);

OTF2_DefReader*
OTF2_Reader_GetDefReader
(
    OTF2_Reader* reader,
    uint64_t     locationID
```

```
);

OTF2_GlobDefReader*
OTF2_Reader_GetGlobDefReader
(
    OTF2_Reader* reader
);
```

Reader Objects also need to be deleted after the trace was read. Local readers are automatically deleted when the main reader object is deleted.

```
SCOREP_Error_Code
OTF2_Reader_CloseEvtReader
(
    OTF2_Reader*    reader,
    OTF2_EvtReader* evtReader
);

SCOREP_Error_Code
OTF2_Reader_CloseDefReader
(
    OTF2_Reader*    reader,
    OTF2_DefReader* defReader
);
```

Callback Registration

The programmer needs to register a callback for each record type. This is done in the same way like for the dedicated reader components (please see Sections 3.2.4, 3.2.5, and 3.2.6). Even the function types for the callbacks are the same like for the dedicated reader components.

```
SCOREP_Error_Code
OTF2_Reader_Register<Evt|GlobEvt|Def|GlobDef>Callbacks
(
    OTF2_Reader*                                reader,
    OTF2_<Evt|GlobEvt|Def|GlobDef>Reader*       evtReader,
    const OTF2_<Evt|GlobEvt|Def|GlobDef>ReaderCallbacks* callbacks,
    size_t                                      sizeOfCallbacks,
    void*                                       userData
);
```

Definition Reading

Definition reading is done similarly like in the local and global definition readers (see Sections 3.2.5 and 3.2.4).

A single event can be read by using the single read function. The record data is passed to a related callback and to the arguments `event` and `type` when the function `OTF2_Reader_Read<Global|Local>Event` is used.

```
SCOREP_Error_Code
OTF2_Reader_Read<Global|Local>Definition
(
    OTF2_Reader*    reader,
    OTF2_<Glob>DefReader* defReader,
    uint64_t        definitionsToRead,
    uint64_t*       definitionsRead
);
```

It is also possible to read n definitions, by passing n as the value for `definitionsToRead` to

the related function. The return value of `definitionsRead` might be smaller than the requested amount of records. The reader has reached the end of the trace in this case.

```
SCOREP_Error_Code
OTF2_Reader_ReadAll<Global|Local>Definitions
(
    OTF2_Reader*          reader,
    OTF2_<Glob>DefReader* defReader,
    uint64_t*             definitionsRead
);
```

Event Trace Reading

Reading a trace is very similar to how it is done with the local event reader (please see Section 3.2.6). A single event can be read by using the single read function. The event data is passed to a related callback and to the arguments `event` and `type` when the function `OTF2_Reader_Read<Local|Global>Event` is used. The resulting `OTF2_GenericEvent` pointer can be casted to the right event type given by `type`.

```
SCOREP_Error_Code
OTF2_Reader_Read<Local|Global>Event
(
    OTF2_Reader*       reader,
    OTF2_EvtReader*    evtReader,
    OTF2_GenericEvent* event,
    OTF2_EventType*    type
);
```

It is also possible to read n events, by passing n as the value for `eventsToRead` to the related function. The return value of `eventsRead` might be smaller than the requested amount of records. The reader has reached the end of the trace in this case.

```
SCOREP_Error_Code
OTF2_Reader_Read<Local|Global>Events
(
    OTF2_Reader*    reader,
    OTF2_EvtReader* evtReader,
    uint64_t        eventsToRead,
    uint64_t*       eventsRead
);
```

The most convenient function for reading a trace might be the function that just reads all events in one single stream.

```
SCOREP_Error_Code
OTF2_Reader_ReadAll<Local|Global>Events
(
    OTF2_Reader*    reader,
    OTF2_EvtReader* evtReader,
    uint64_t*       eventsRead
);
```

Event Trace Backward Reading

OTF2 also makes backward reading for local event streams possible. The following functions read events backwards and their arguments are the same like the related functions for forward reading.

```
SCOREP_Error_Code
OTF2_Reader_ReadLocalEventBackward
(
```

```
    OTF2_Reader*       reader,
    OTF2_EvtReader*    evtReader,
    OTF2_GenericEvent* event,
    OTF2_EventType*    type
);

SCOREP_Error_Code
OTF2_Reader_ReadLocalEventsBackward
(
    OTF2_Reader*    reader,
    OTF2_EvtReader* evtReader,
    uint64_t        eventsToRead,
    uint64_t*       eventsRead
);
```

3.3 Internal Components

3.3.1 Buffer

The buffer class is a completely internal module which is not exposed to the end user. It is a part of layer three of the OTF2 library (see Section 3.1). Its purpose is to abstract the handling of a memory buffer and to provide basic routines encapsulating the encoding and decoding of elementary data types. This layer will even be used for directly reading or writing to/from a file, since a certain amount of buffering is required to achieve a reasonable performance by reducing the number of I/O function calls as well as increasing the size of the data block to be written/read. The buffer implementation can be found in the file `OTF2_Buffer.c`.

A new buffer object can be generated by using the related `OTF2_Buffer_New` function. The argument `chunkSize` determines the size of the chunks which are used internally (please see also Section 3.2.6). `bufferMode` can be set to `OTF2_BUFFER_WRITE`, `OTF2_BUFFER_MODIFY` and `OTF2_BUFFER_READ`, which determines that the buffer is used for reading, writing, or for modifying timstamps of an already loaded trace. A buffer can be used in a chunked or a not chunked way, which is is determined by the argument `chunkMode`. The file substrate (please also see Section 3.3.4), which is responsible for reading and writing the data to disk, can be set with the argument `substrate`. Whether the resulting file is compressed or not can be set with `compression`. Since the buffer module handles file reads and writes completely automatically and transparent to the layers above, it needs some information about the file which has to be used. This file path must be given with the argument `filePath`. The buffer module automatically stores its data into a file if there is no more memory available or the buffer is deleted. The callbacks `preFlush` and `postFlush` are triggered before and after such a flush event. The local event reader, given by the argument `evtWriter`, is also passed to the `preFlush` callback, for in-memory analysis. The buffer module can also make use of external functions for memory allocation, if the programmer passes callbacks for `allocate`, `free`, and a the `allocatorData` object. This is similar to how it is done in the archive module (see Section 3.2.8).

```
OTF2_Buffer*
OTF2_Buffer_New
(
    uint64_t               chunkSize,
    OTF2_BufferMode        bufferMode,
    OTF2_ChunkMode         chunkMode,
    OTF2_FileSubstrate     substrate,
    OTF2_Compression       compression,
```

```
    const char*            filePath,
    OTF2_PreFlushCallback  preFlush,
    OTF2_PostFlushCallback postFlush,
    void*                  evtWriter,
    OTF2_MemoryAllocate    allocate,
    OTF2_MemoryFree        free,
    void*                  allocatorData
);
```

A buffer object needs to be destroyed after trace recording. This can be done with the related delete function, which also triggers that the buffer writes the data to the given file.

```
SCOREP_Error_Code
OTF2_Buffer_Delete
(
    OTF2_Buffer* bufferHandle
);
```

The buffer needs to be switched from reading to the modify mode, to modify the timestamps of a loaded trace.

```
SCOREP_Error_Code
OTF2_Buffer_SwitchMode
(
    OTF2_Buffer*    bufferHandle,
    OTF2_BufferMode bufferMode
);
```

It may happen that the file path can only be set after the creation of the buffer module (see Section 3.2.3). The buffer module therefore implements a related set function.

```
SCOREP_Error_Code
OTF2_Buffer_SetFilePath
(
    OTF2_Buffer* bufferHandle,
    const char*  filePath
);
```

Write Functions

Pre-flush callbacks can also be registered after buffer creation. The buffer is automatically flushed to disk if it is full. This is always done by the method which was passed to the `OTF2_Buffer_New` function of the buffer object. The related memory buffer is inserted into an `OTF2_ EvtReader` and passed to the pre-flush-callback function before flushing. The return value of this callback function can prevent disk flushing if the return value is `OTF2_NO_FLUSH`. It is always possible to prevent a buffer flush but it is not possible to use different flush methods within one experiment. For the pre-flush-callback a default implementation exists.

```
SCOREP_Error_Code
OTF2_Buffer_RegisterPreFlushCallback
(
    OTF2_Buffer*          bufferHandle,
    OTF2_PreFlushCallback callback
);

SCOREP_Error_Code
OTF2_Buffer_RegisterPostFlushCallback
(
    OTF2_Buffer*           bufferHandle,
    OTF2_PostFlushCallback callback
);
```

It must be ensured that there is enough space left in the buffer before a record is written. If there is not enough memory left the buffer flushed to disk. Therefore, this call should always succeed even if a buffer is full. If the call returns an error, buffer handling failed and the recording should be aborted.

```
SCOREP_Error_Code
OTF2_Buffer_WriteMemoryRequest
(
    OTF2_Buffer* bufferHandle,
    size_t       recordLength
);
```

Writing a timestamp is a special case, because a special timestamp record is written into a buffer only once for all consecutive records with the same timestamp. Timestamps are also not compressed, to make their modification easier afterwards. The buffer module therefore implements a dedicated function for writing timestamps.

```
SCOREP_Error_Code
OTF2_Buffer_WriteTimeStamp
(
    OTF2_Buffer*         bufferHandle,
    const OTF2_TimeStamp time,
    size_t               recordLength
);
```

The main purpose of the buffer module is the abstraction of memory accesses. Therefore it implements a set of functions to encode basic data types and write them compressed to the buffer. Functions ending with `Full` are writing uncompressed values. The uncompressed functions call should be used for values that may be altered later (e.g. timestamps). The other functions write in a compressed manner (see Section 2.3) if possible.

```
void
OTF2_Buffer_WriteInt8
(
    OTF2_Buffer* bufferHandle,
    const int8_t value
);

void
OTF2_Buffer_WriteUint8
(
    OTF2_Buffer*  bufferHandle,
    const uint8_t value
);

void
OTF2_Buffer_WriteInt16
(
    OTF2_Buffer*  bufferHandle,
    const int16_t value
);

void
OTF2_Buffer_WriteUint16
(
    OTF2_Buffer*   bufferHandle,
    const uint16_t value
);

void
OTF2_Buffer_WriteInt32
```

```
(
    OTF2_Buffer*  bufferHandle,
    const int32_t value
);

void
OTF2_Buffer_WriteInt32Full
(
    OTF2_Buffer*  bufferHandle,
    const int32_t value
);

void
OTF2_Buffer_WriteUint32
(
    OTF2_Buffer*   bufferHandle,
    const uint32_t value
);

void
OTF2_Buffer_WriteUint32Full
(
    OTF2_Buffer*   bufferHandle,
    const uint32_t value
);

void
OTF2_Buffer_WriteInt64
(
    OTF2_Buffer*  bufferHandle,
    const int64_t value
);

void
OTF2_Buffer_WriteInt64Full
(
    OTF2_Buffer*  bufferHandle,
    const int64_t value
);

void
OTF2_Buffer_WriteUint64
(
    OTF2_Buffer*   bufferHandle,
    const uint64_t value
);

void
OTF2_Buffer_WriteUint64Full
(
    OTF2_Buffer*   bufferHandle,
    const uint64_t value
);

void
OTF2_Buffer_WriteFloat
(
    OTF2_Buffer* bufferHandle,
    const float  value
);

void
OTF2_Buffer_WriteDouble
```

```
(
    OTF2_Buffer* bufferHandle,
    const double value
);

void
OTF2_Buffer_WriteString
(
    OTF2_Buffer* bufferHandle,
    const char*  value
);
```

Arrays can also be written into a buffer that is not operated in chunked mode.

```
void
OTF2_Buffer_WriteArray
(
    OTF2_Buffer* bufferHandle,
    void*        values,
    uint32_t     elements,
    OTF2_TypeID  type,
    uint8_t      bytes
);
```

The OTF2 format allows only to store complete records into a chunk (see Section 2.3). That means that it is not allowed to scatter a record over two different chunks and the buffer needs to know how much space is needed for a new record.

```
static inline SCOREP_Error_Code
otf2_buffer_record_request
(
    OTF2_Buffer*   bufferHandle,
    OTF2_TimeStamp time,
    uint64_t       recordLength,
    uint8_t        requestMode
);
```

The chunk-header, which is described in Section 2.3, is written by the following function.

```
static inline void
otf2_buffer_write_header
(
    OTF2_Buffer* bufferHandle
);
```

The chosen buffer overflow method is automatically triggered if the buffer is full. Sometimes it is desired to trigger this method at a special point within the program flow. This is also done with this call.

```
SCOREP_Error_Code
OTF2_Buffer_FlushBuffer
(
    OTF2_Buffer* bufferHandle
);
```

Read Functions

The buffer returns always the last timestamp that was read, since timestamps are handled in a special way (see Section 2.3). This function also checks that the end of the current chunk was reached, to make automatic loading of the next chunk possible.

```
SCOREP_Error_Code
OTF2_Buffer_ReadTimeStamp
(
    OTF2_Buffer*    bufferHandle,
    OTF2_TimeStamp* time
);
```

The buffer also implements reading routines for the same basic data types like for writing. They read and decode basic data types from the buffer. Functions ending with `Full` read uncompressed values. The uncompressed functions call should be used for values that are per default not compressed e.g. timestamps.

```
void
OTF2_Buffer_ReadUint8
(
    OTF2_Buffer* bufferHandle,
    uint8_t*     returnValue
);

void
OTF2_Buffer_ReadInt8
(
    OTF2_Buffer* bufferHandle,
    int8_t*      returnValue
);

void
OTF2_Buffer_ReadUint16
(
    OTF2_Buffer* bufferHandle,
    uint16_t*    returnValue
);

void
OTF2_Buffer_ReadInt16
(
    OTF2_Buffer* bufferHandle,
    int16_t*     returnValue
);

void
OTF2_Buffer_ReadUint32
(
    OTF2_Buffer* bufferHandle,
    uint32_t*    returnValue
);

void
OTF2_Buffer_ReadInt32
(
    OTF2_Buffer* bufferHandle,
    int32_t*     returnValue
);

void
OTF2_Buffer_ReadUint32Full
(
    OTF2_Buffer* bufferHandle,
    uint32_t*    returnValue
);

void
```

```
OTF2_Buffer_ReadInt32Full
(
    OTF2_Buffer* bufferHandle,
    int32_t*     returnValue
);

void
OTF2_Buffer_ReadUint64
(
    OTF2_Buffer* bufferHandle,
    uint64_t*    returnValue
);

void
OTF2_Buffer_ReadInt64
(
    OTF2_Buffer* bufferHandle,
    int64_t*     returnValue
);

void
OTF2_Buffer_ReadUint64Full
(
    OTF2_Buffer* bufferHandle,
    uint64_t*    returnValue
);

void
OTF2_Buffer_ReadInt64Full
(
    OTF2_Buffer* bufferHandle,
    int64_t*     returnValue
);

void
OTF2_Buffer_ReadFloat
(
    OTF2_Buffer* bufferHandle,
    float*       returnValue
);

void
OTF2_Buffer_ReadFloat
(
    OTF2_Buffer* bufferHandle,
    float*       returnValue
);

void
OTF2_Buffer_ReadString
(
    OTF2_Buffer* bufferHandle,
    char**       returnValue
);
```

The buffer module also supports reading arrays from not chunked buffers.

```
void
OTF2_Buffer_Read<Int8|Uint8|Int16|Uint16|Int32|Uint32|Int64|Uint64>Array
(
    OTF2_Buffer* bufferHandle,
    <int8|uint8|int16|uint16|int32|uint32|int64|uint64>_t**     values
);
```

```
                                                                    7
OTF2_Buffer_Read<Float|Double>Array                                 8
(                                                                   9
    OTF2_Buffer* bufferHandle,                                      10
    <float|double>**     values                                     11
);                                                                  12
```

The OTF2 library does not need to decode every record member to index a chunk. It just needs to read the length of each member and skip the decompression. Therefore the buffer module implements a couple of skip functions.

The `SkipFull` Function is used to skip an uncompressed variable.

```
void                                                                1
OTF2_Buffer_SkipFull                                                2
(                                                                   3
    OTF2_Buffer* bufferHandle,                                      4
    uint8_t      size                                               5
);                                                                  6
```

The `SkipCompressed` Function is used to skip an uncompressed variable.

```
void                                                                1
OTF2_Buffer_SkipCompressed                                          2
(                                                                   3
    OTF2_Buffer* bufferHandle                                       4
);                                                                  5
```

A whole array can be skipped with the function `SkipCompressed`.

```
void                                                                1
OTF2_Buffer_SkipArray                                               2
(                                                                   3
    OTF2_Buffer* bufferHandle                                       4
);                                                                  5
```

Chunk Management

The next function is needed to make it obsolete for the buffer in reading mode to check every time if the requested date is in the current chunk. If a local reader reads a NOOP event in forward reading mode, it has to trigger the `ReadGetNextChunk` function.

```
SCOREP_Error_Code                                                   1
OTF2_Buffer_ReadGetNextChunk                                        2
(                                                                   3
    OTF2_Buffer* bufferHandle                                       4
);                                                                  5
```

The `ReadGetPreviousChunk` function is needed to make it obsolete for the buffer in reading mode to check every time if the the requested date is in the current chunk. A local reader has to trigger the `ReadGetPrevoiusChunk` function if it gets a further request for reading backward after delivering the first event record of the current chunk.

```
SCOREP_Error_Code                                                   1
OTF2_Buffer_ReadGetPreviousChunk                                    2
(                                                                   3
    OTF2_Buffer* bufferHandle                                       4
);                                                                  5
```

The buffer module also needs a function that reads a new chunk from a file into the memory.

```
SCOREP_Error_Code
otf2_buffer_read_chunk
(
    OTF2_Buffer*  bufferHandle,
    OTF2_FilePart filePart
);
```

The buffer needs to extract the chunk header, after reading a chunk.

```
static inline SCOREP_Error_Code
otf2_buffer_read_header
(
    OTF2_Buffer* bufferHandle
);
```

The position inside a chunk can also be set. This is used to implement backward reading and seeking.

```
SCOREP_Error_Code
OTF2_Buffer_GetPosition
(
    OTF2_Buffer* bufferHandle,
    uint8_t**    position
);

SCOREP_Error_Code
OTF2_Buffer_SetPosition
(
    OTF2_Buffer* bufferHandle,
    uint8_t*     position
);
```

The function `GetBeginOfChunk` is needed for generating an index of the whole chunk.

```
SCOREP_Error_Code
OTF2_Buffer_GetBeginOfChunk
(
    OTF2_Buffer* bufferHandle,
    uint8_t**    position
);
```

Meta-Data

The buffer module internally counts the number of recorded events. It can do this very easily since the `WriteMemoryRequest` function is triggered for each record.

```
SCOREP_Error_Code
OTF2_Buffer_GetNumberEvents
(
    OTF2_Buffer* bufferHandle,
    uint64_t*    firstEvent,
    uint64_t*    lastEvent
);
```

The timestamp of the current event can also be rewritten. A special function is needed, since there is only one timestamp record written for all records which where recorded at the same time.

```
SCOREP_Error_Code
OTF2_Buffer_RewriteTimeStamp
(
    OTF2_Buffer*   bufferHandle,
```

```
    OTF2_TimeStamp time
);
```

3.3.2 Anchor File

The anchor file module loads and stores the meta-data that is stored into the anchor file (please see also Section 2.2). It can therefore be used to parse and generate valid OTF2 anchor files. The archive implementation can be found in the file `OTF2_AnchorFile.c`.

The anchor file module does not implement a `new` or a delete `function`, which is different from other modules. Instead, a related `load` function parses the content of an OTF2 anchor file and passes the information to the given archive handle.

```
SCOREP_Error_Code
OTF2_AnchorFile_Load
(
    OTF2_Archive* archive
);
```

The meta data, which is stored in an archive handle, can also be stored to a file.

```
SCOREP_Error_Code
OTF2_AnchorFile_Save
(
    OTF2_Archive* archive
);
```

The `get_value` function extracts the value string from a given line. Therefore, it uses the given key to identify how the value_string has to look like. Then it adds the gained value to the anchor file data handle. At the end it adds the key to the bit mask.

```
static inline SCOREP_Error_Code
otf2_anchorfile_get_value
(
    char*                 line,
    otf2_anchorfile_key   key,
    otf2_anchorfile_data* data
);
```

The anchor file module uses a special `get_key` function internally, to extract the value of a given line.

```
static inline SCOREP_Error_Code
otf2_anchorfile_get_key
(
    char*                line,
    otf2_anchorfile_key* key
);
```

3.3.3 Internal Archive

The internal archive component is the internal memory representation of an OTF2 trace archive. It is very closely related to the externally visible archive component (please see Section 3.2.8). The external archive component uses the internal archive. The archive implementation can be

found in the file OTF2_InternalArchive.c. A new archive object can be generated by using the related OTF2_Archive_New function. To generate file paths internally, the arguments archivePath and archiveName are used. Substrate is used to set the substrate for file writing (please see also Section 3.3.4). compression is used to set the substrate to compress the data (please see also Section 3.3.4). Chunk sizes for definition and event buffers can be set with chunkSizeEvents and chunkSizeDefs (please see also Section 2.3). FileMode can be set to OTF2_FILEMODE_<WRITE>|<READ>|<MODIFY>. The modify-mode allows to modify timestamps in an existing trace. OTF2 is also capable to handle external memory allocators, for example to realize memory pooling etc. A programmer can therefore pass callbacks, for allocating (allocate) and freeing (free) data, to the archive.

```
OTF2_InternalArchiveData*                                                1
OTF2_IntArchive_New                                                      2
(                                                                        3
    const char*          archiveName,                                    4
    const char*          archivePath,                                    5
    OTF2_FileSubstrate   Substrate,                                      6
    OTF2_Compression     compression,                                    7
    uint64_t             chunkSizeEvents,                                8
    uint64_t             chunkSizeDefs,                                  9
    OTF2_FileMode        FileMode,                                       10
    OTF2_MemoryAllocate  allocate,                                       11
    OTF2_MemoryFree      free,                                           12
    void*                allocatorData                                   13
);                                                                       14
```

An internal archive object needs to be destroyed after trace recording. This can be done with the related delete function. A deletion of an internal archive also deletes all opened event reader and writer objects.

```
SCOREP_Error_Code                                                        1
OTF2_IntArchive_Delete                                                   2
(                                                                        3
    OTF2_InternalArchiveData* archive                                    4
);                                                                       5
```

The SetFilePath can be used to set the file path to the folder where the archive is stored.

```
SCOREP_Error_Code                                                        1
OTF2_IntArchive_SetFilePath                                              2
(                                                                        3
    OTF2_InternalArchiveData* archive,                                   4
    const char*               filePath                                   5
);                                                                       6
```

SetArchiveName is used to set the archive name. This name is used to generate file names of an OTF2 archive (please see Section 2.1).

```
SCOREP_Error_Code                                                        1
OTF2_IntArchive_SetArchiveName                                           2
(                                                                        3
    OTF2_InternalArchiveData* archive,                                   4
    const char*               archiveName                                5
);                                                                       6
```

OTF2 also needs to be aware of the trace format version which is used. The trace format version is stored to the anchor file, to make format incompatibilities detectable.

```
SCOREP_Error_Code                                                        1
OTF2_IntArchive_SetTraceFormatVersion                                    2
```

```
(
    OTF2_InternalArchiveData* archive,
    uint8_t                   current,
    uint8_t                   revision,
    uint8_t                   age
);

SCOREP_Error_Code
OTF2_IntArchive_GetTraceFormatVersion
(
    OTF2_InternalArchiveData* archive,
    uint8_t*                  current,
    uint8_t*                  revision,
    uint8_t*                  age
);
```

Chunk sizes for definition and event buffers can be set with `SetChunkSize` (please see also Section 2.3).

```
SCOREP_Error_Code
OTF2_IntArchive_SetChunkSize
(
    OTF2_InternalArchiveData* archive,
    uint64_t                  chunkSizeEvents,
    uint64_t                  chunkSizeDefs
);

SCOREP_Error_Code
OTF2_IntArchive_GetChunkSize
(
    OTF2_InternalArchiveData* archive,
    uint64_t*                 chunkSizeEvents,
    uint64_t*                 chunkSizeDefs
);
```

The machine name, archive description, and the creators name is optionally stored into the anchor file. Please read also Section 2.1 for more information about meta-data that is stored into the anchor file. All this dates can be set with the related function.

```
SCOREP_Error_Code
OTF2_IntArchive_SetMachineName
(
    OTF2_InternalArchiveData* archive,
    const char*               machineName
);

SCOREP_Error_Code
OTF2_IntArchive_GetMachineName
(
    OTF2_InternalArchiveData* archive,
    char**                    machineName
);

SCOREP_Error_Code
OTF2_IntArchive_SetDescription
(
    OTF2_InternalArchiveData* archive,
    const char*               description
);

SCOREP_Error_Code
OTF2_IntArchive_GetDescription
```

```
(
    OTF2_InternalArchiveData* archive,
    char**                    description
);

SCOREP_Error_Code
OTF2_IntArchive_SetCreator
(
    OTF2_InternalArchiveData* archive,
    const char*               creator
);

SCOREP_Error_Code
OTF2_IntArchive_GetCreator
(
    OTF2_InternalArchiveData* archive,
    char**                    creator
);
```

The number of available locations is a mandatory information since a potential reader must know how much locations are available in a trace archive. `SetNumberOfLocations` can therefore used to set the number of locations.

```
SCOREP_Error_Code
OTF2_IntArchive_SetNumberOfLocations
(
    OTF2_InternalArchiveData* archive,
    uint64_t                  locationNumber
);

SCOREP_Error_Code
OTF2_IntArchive_GetNumberOfLocations
(
    OTF2_InternalArchiveData* archive,
    uint64_t*                 locationNumber
);
```

The number of global definitions is actually counted by the related writer object. However, the following function can be used to get the current number of global definition records that written up to now.

```
SCOREP_Error_Code
OTF2_IntArchive_SetNumberOfGlobDefs
(
    OTF2_InternalArchiveData* archive,
    uint64_t                  globalDefRecordsNumber
);

SCOREP_Error_Code
OTF2_IntArchive_GetNumberOfGlobDefs
(
    OTF2_InternalArchiveData* archive,
    uint64_t*                 globalDefRecordsNumber
);
```

Only one process writes the global definitions and the anchor file. The information that a specific object is a master or not must be passed with a special function, since OTF2 is programming paradigm unaware.

```
SCOREP_Error_Code
OTF2_IntArchive_SetMasterSlaveMode
```

```
(
    OTF2_InternalArchiveData* archive,
    OTF2_MasterSlaveMode      masterOrSlave
);

SCOREP_Error_Code
OTF2_IntArchive_GetMasterSlaveMode
(
    OTF2_InternalArchiveData* archive,
    OTF2_MasterSlaveMode*     masterOrSlave
);
```

`SetFileMode` can be used to set the internal mode of all opened files to `OTF2_FILEMODE_<WRITE>|<READ>|<MODIFY>`.

```
SCOREP_Error_Code
OTF2_IntArchive_SetFileMode
(
    OTF2_InternalArchiveData* archive,
    OTF2_FileMode             fileMode
);

SCOREP_Error_Code
OTF2_IntArchive_GetFileMode
(
    OTF2_InternalArchiveData* archive,
    OTF2_FileMode*            fileMode
);
```

The function `SetFileSubstrate` is used to set the substrate for file writing (please see also Section 3.3.4).

```
SCOREP_Error_Code
OTF2_IntArchive_SetFileSubstrate
(
    OTF2_InternalArchiveData* archive,
    OTF2_FileSubstrate        Substrate
);

SCOREP_Error_Code
OTF2_IntArchive_GetFileSubstrate
(
    OTF2_InternalArchiveData* archive,
    OTF2_FileSubstrate*       Substrate
);
```

The function `SetCompression` is used to set the substrate to compress the data (please see also Section 3.3.4).

```
SCOREP_Error_Code
OTF2_IntArchive_SetCompression
(
    OTF2_InternalArchiveData* archive,
    OTF2_Compression          compression
);

SCOREP_Error_Code
OTF2_IntArchive_GetCompression
(
    OTF2_InternalArchiveData* archive,
    OTF2_Compression*         compression
);
```

File Paths

The internal archive component is also capable to generate all direct file paths of the anchor file, global and local definition file, and the trace files (please also read Section 2.1).

```
SCOREP_Error_Code
OTF2_IntArchive_GetPathAnchorFile
(
    OTF2_InternalArchiveData* archive,
    char**                    anchorFilePath
);

SCOREP_Error_Code
OTF2_IntArchive_GetPathGlobDefFile
(
    OTF2_InternalArchiveData* archive,
    char**                    globalDefinitionFilePath
);

SCOREP_Error_Code
OTF2_IntArchive_GetPathLocalDefFile
(
    OTF2_InternalArchiveData* archive,
    char**                    localDefinitionFilePath,
    uint64_t                  locationId
);

SCOREP_Error_Code
OTF2_IntArchive_GetPathLocal
(
    OTF2_InternalArchiveData* archive,
    char**                    localFilePath
);

SCOREP_Error_Code
OTF2_IntArchive_GetPathLocalTraceFile
(
    OTF2_InternalArchiveData* archive,
    char**                    localTraceFilePath,
    uint64_t                  locationId
);

SCOREP_Error_Code
OTF2_IntArchive_GetPathArchive
(
    OTF2_InternalArchiveData* archive,
    char**                    ArchivePath
);
```

Reader and Writer Components

Reader and Writer objects have to be generated from the internal archive, to avoid that a program generates two different writer objects for the same trace file. The external archive component uses the internal archive component to do this (please see Section 3.2.8). The programmer must therefore pass a valid location ID and two different callbacks to the related functions. The callbacks are triggered before and after a buffer flush. Please read also Sections 3.2.3, 3.2.2, and 3.2.1.

```
SCOREP_Error_Code
OTF2_IntArchive_<GetEvt|Def|GlobDef>Writer
(
    OTF2_InternalArchiveData*       archive,
    OTF2_<Evt|Def|GlobDef>Writer**  writer,
```

```
    uint64_t                        locationID,                                 6
    OTF2_PreFlushCallback           preFlush,                                   7
    OTF2_PostFlushCallback          postFlush                                   8
);                                                                              9
```

Also reader objects need to be generated by the central archive management. This offers the opportunity to handle process local data (for example just one definition file per process). Please read also Sections 3.2.7, 3.2.4, 3.2.3, and 3.2.2.

```
SCOREP_Error_Code                                                               1
OTF2_IntArchive_Get<Evt|Def|GlobDef|GlobEvt>Reader                              2
(                                                                               3
    OTF2_InternalArchiveData*               archive,                            4
    OTF2_<Evt|Def|GlobDef|GlobEvt>Reader**  reader,                             5
    uint64_t                                locationID                          6
);                                                                              7
```

3.3.4 File Abstraction Layer

The file abstraction layer provides a low-level API for accessing files. Such an abstraction layer can be used, to provide more file handles to the user than the operating system may provide and to abstract the file handling. This layer is easily extendible by implementing so called substrates. A substrate must implement a couple of functions for reading and writing and a sing registration function. All functions of this module are more or less similar to their POSIX equivalents. The file layer implementation can be found in the file `OTF2_File.c`.

Each substrate can use the original OTF2 file descriptor, or at least a structure which can be casted into one. Such a data structure needs to implement the same members like the original struct first and can be extended afterwards (after line 27 in the following listing). A substrate needs to implement functions that are compatible with the function pointers of the `OTF2_File_struct` (after line 12 in the following listing).

```
struct OTF2_File_struct                                                         1
{                                                                               2
    const char*         file_name;                                              3
    OTF2_FileMode       mode;                                                   4
    OTF2_FileSubstrate  substrate;                                              5
    OTF2_Compression    compression;                                            6
    uint32_t            last_block_size;                                        7
    void*               buffer;                                                 8
    uint32_t            buffer_used;                                            9
                                                                                10
                                                                                11
    SCOREP_Error_Code ( * Close )( OTF2_File* file );                           12
    SCOREP_Error_Code ( * Reset )( OTF2_File* file );                           13
    SCOREP_Error_Code ( * Write )( OTF2_File*  file,                            14
                                   const void* buffer,                          15
                                   uint64_t    size );                          16
    SCOREP_Error_Code ( * Read )( OTF2_File* file,                              17
                                  void*      buffer,                            18
                                  uint64_t   size );                            19
    SCOREP_Error_Code ( * Seek )( OTF2_File*    file,                           20
                                  int64_t       offset,                         21
                                  OTF2_FileSeek origin );                       22
    SCOREP_Error_Code ( * GetFileSize )( OTF2_File* file,                       23
                                         uint64_t*  size );                     24
                                                                                25
                                                                                26
```

```
    /* Additional file substrate specific information can be added here. */
};
```

The main `OTF2_File` component implements a special open function, that is used to open a file and to setup the related substrate. The main function to open a file does not set the function-pointers for `Close`, `Reset`, `Write`, `Read`, `Seek`, an `GetFileSize`. These pointers have to be set by the open functions which are substrate specific.

```
OTF2_File*
OTF2_File_Open
(
    const char*         fileName,
    OTF2_FileMode       mode,
    OTF2_FileSubstrate  substrate,
    OTF2_Compression    compression
);
```

The `Close` call closes an opened file.

```
SCOREP_Error_Code
OTF2_File_Close
(
    OTF2_File* file
);
```

It is also possible to reset an opened file, i.e. the file pointer is set to the beginning of the file and the file size is set to zero.

```
SCOREP_Error_Code
OTF2_File_Reset
(
    OTF2_File* file
);
```

The `Write` function can be used to write the content of a buffer into a file.

```
SCOREP_Error_Code
OTF2_File_Write
(
    OTF2_File*  file,
    const void* buffer,
    uint64_t    size
);
```

The `Read` function can be used to read the content of a file into a buffer.

```
SCOREP_Error_Code
OTF2_File_Read
(
    OTF2_File* file,
    void*      buffer,
    uint64_t   size
);
```

Even a directory can be created with the related function. Please note that this function is the only one besides `Open`, which has an argument to determine the file substrate to be used, since directories are not handle by a file pointer.

```
SCOREP_Error_Code
OTF2_File_CreateDirectory
(
```

```
    OTF2_FileSubstrate substrate,
    const char*        mainPath
);
```

File Substrate Non

The Non substrate for file writing is simply a substrate that does nothing. It is needed to make the support for in-memory-analysis available. The programmer (who uses OTF2) needs to get the trace data directly from the flush callbacks, instead opening a trace archive, to use this substrate correctly. Please read also Section 3.2.8 for a description of the flush callback mechanism. The implementation of the Non substrate can be found in the file `OTF2_File_Non.c`.

File Substrate Posix

The Posix substrate simply maps the POSIX functions into the file layer. The implementation of the Posix substrate can be found in the file `OTF2_File_Posix.c`.

File Substrate Sion

The Sion substrate uses the Sion library for high scaling I/O. This library can be used for cases where every rank of a MPI program needs to open a file handle. Sion achieves very good I/O performance in those cases, by mapping all file handles into one single file. It makes therefore sense to use Sion for local definition and event files, since these files are generated for each location (see Section 2.1 for a complete description of an OTF2 archive).

The open function of the Sion substrate makes therefore a decision based on the file name. All filenames ending with an `.evt` or `.def` are handled by special Sion routines, all other will be handled by the callbacks of the Posix substrate. The Sion substrates simply sets the related function pointers to the POSIX functions in the latter case. Event and definition data is mapped into the files `<trace name>/sion.evt` and `<trace name>/sion.def` . The implementation of the Sion substrate can be found in the file `OTF2_File_Sion.c`.

Chapter 4

Appendix

4.1 Generator Scripts

The generator, which generates several parts of the reader and writer components, is also delivered with the tar-ball. It can be found under `otf2/src`. A user needs first to change his current pwd to this directory, to be able to execute the script afterwards. The output of the generator script should look like:

```
./gen.sh

0a) Tokenize OTF2_Events.h
0b) Tokenize OTF2_GlobDefinitions.h
0c) Tokenize OTF2_LocalDefinitions.h
1)  Generating OTF2_Events_inc.h
2a) Generating OTF2_EvtWriter_inc.{c,h}
2b) Generating OTF2_EvtReader_inc.{c,h}
3)  Generating OTF2_GlobEvtReader_inc.{c,h}
4a) Generating OTF2_DefWriter_inc.{c,h}
4b) Generating OTF2_DefReader_inc.{c,h}
5a) Generating OTF2_GlobDefWriter_inc.{c,h}
5b) Generating OTF2_GlobDefReader_inc.{c,h}
```

The script performs several steps on the input data from the file `include/otf2/OTF2_Events.h`, `include/otf2/OTF2_GlobDefinitions.h`, and `include/otf2/OTF2_LocalDefinitions.h` Usually, the input looks like the following snipped:

```
#define OTF2_MEASUREMENT_ON  1
#define OTF2_MEASUREMENT_OFF 2

// #pragma OTF2_GEN
typedef struct OTF2_MeasurementOnOff_struct
{
    OTF2_TimeStamp time;      /**< Timestamp, which is exactly here in every
                                   record */
    uint8_t        on_or_off; /**< Is the measurement turned on or off? */
} OTF2_MeasurementOnOff;
// #pragma OTF2_GEN
```

The first step is to remove everything which is not surrounded by the `#pragma OTF2_GEN` clause. That will result in the following output:

```
typedef struct OTF2_MeasurementOnOff_struct
```

```
{                                                                              2
    OTF2_TimeStamp time;      /**< Timestamp, which is exactly here in every    3
                                   record */                                   4
    uint8_t        on_or_off; /**< Is the measurement turned on or off? */     5
} OTF2_MeasurementOnOff;                                                       6
```

The script will remove all comments afterwards, which will look like the following example:

```
typedef struct OTF2_MeasurementOnOff_struct                                    1
{                                                                              2
    OTF2_TimeStamp time;                                                       3
    uint8_t        on_or_off;                                                  4
} OTF2_MeasurementOnOff;                                                       5
```

The last step of parsing the input is to tokenize all keyword that are separated by white spaces. The result looks like:

```
typedef                                                                        1
struct                                                                         2
OTF2_MeasurementOnOff_struct                                                   3
{                                                                              4
OTF2_TimeStamp                                                                 5
time                                                                           6
;                                                                              7
uint8_t                                                                        8
on_or_off                                                                      9
;                                                                              10
}                                                                              11
OTF2_MeasurementOnOff                                                          12
;                                                                              13
```

This resulting list of items is then used to generate parts of all reader and writer components. Please be aware that this parser can not handle syntax errors. It is therefore necessary that the input is syntactically correct. Furthermore each struct definition needs to be typedefed. The struct name needs to have the form `OTF2_<name>_struct` and the type definition `OTF2_<name>`. Other constructs than the structure definition showed in the example are not allowed between `#pragma OTF2_GEN` statements.

4.2 otf2-config

The `otf2-config` tool should be used to pass compiler and linker flags to the build environment. This makes the OTF2 library easier usable in a modules environment and in makefiles, since the building mechanism does not need to figure out where the library is located etc. It is also possible to have multiple installations of OTF2 in one single system without trouble.

A simple call to this tool should return the following help text:

```
otf2-config                                                                    1
                                                                               2
Usage:                                                                         3
otf2-config (--cflags|--libs|--cc|--cxx) [--backend] [--config=<config_file>]  4
```

The options can be explained as follows:

`--cflags` This option will return the needed compiler flags. These are mainly include directories etc.

`--libs` The libs option will return all necessary linker flags.

`--cc` Gives the compatible C compiler.

`--cxx` Gives the compatible C++ compiler.

`--backend` The config utility returns linker and compiler flags for the back-end of the current architecture.

`--config=<config_file>` The config utility gets its information from a single configuration file. The built-in file path can be substituted with the config flag. Please do not use this option unless you really know what you are doing.

A related makefile could look like this:

```
all:
    mpicc `otf2-config --backend --cflags` -C main.c
    mpicc -o main main.o `otf2_config --backend --libs`
```

4.3 otf2_print

OTF2 also provides a simple tool to examine a trace archive. A call to this tool with the option `--help` reveals all its functionalities:

```
otf2_print --help
Usage: otf2_print [OPTION]... ANCHORFILE
Print the content of all files of an OTF2 archive with the ANCHORFILE.

  -A, --show-all           Print all output including definitions and anchor
                           file.
  -G, --show-glob-defs     Print all global definitions.
  -M, --show-mappings      Print mappings to global definitions.
  -O, --show-clock-offsets Print clock offsets to global timer..
  -L, --location LID       Limit output to location LID.
  -s, --step N             Step through output by steps of N events.
      --silent             Only validate trace and do not print any events.
      --time MIN MAX       Limit output to events within time interval.
  -d, --debug              Turn on debug mode.
  -V, --version            Print version information.
      --system-tree        Output system tree to dot-file.
  -h, --help               Print this help information.
```

Chapter 5

Acknowledgements

The OTF2 library was a development of the SILC project [6]. It was funded by the German Federal Ministry of Education and Research under funding number 01IH08006. Further project partner, besides Forschungszentrum Jülich GmbH, where the Technische Universität Dresden, Aachen University (RWTH), Gesellschaft für numerische Simulation (GNS), and Technische Universität München. Further associated partners in SILC where the German Research School for Simulation Science (GRS) and the University of Oregon (PRIMA project, Tau toolkit [7]). OTF2 was mainly developed by Forschungszentrum Jülich GmbH and Technische Universität Dresden.

Bibliography

[1] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open trace format 2 - the next generation of scalable trace formats and support libraries. In *Proc. of the International Conference on Parallel Computing (ParCo), Ghent, Belgium*, 2011. (to appear).

[2] F. Wolf and B. Mohr. EPILOG Binary Trace-Data Format. Technical Report FZJ-ZAM-IB-2004-06, Forschungszentrum Jülich, May 2004.

[3] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In *Computational Science ICCS 2006: 6th International Conference*, LNCS 3992, Reading, UK, May 2006. Springer.

[4] Markus Geimer, Felix Wolf, Brian J. N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In *Proc. of the 13th European PVM/MPI Users' Group Meeting (EuroPVM/MPI), Bonn, Germany*, volume 4192 of *Lecture Notes in Computer Science*, pages 303–312. Springer, September 2006.

[5] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool Set. In *Tools for High Performance Computing*, pages 139–155. Springer, July 2008.

[6] Dieter an Mey, Scott Biersdorff, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P–A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany*, pages 1–12. Gauß-Allianz, Springer, June 2010. (to appear).

[7] S. Shende and A. D. Malony. The TAU Parallel Performance System, SAGE Publications. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.